Darstellung von Sternentwicklung mit der Programmiersprache Java

Christian Carazo Ziegler

28. September 2006

Inhaltsverzeichnis

1 Einleitung **1**

2 Sternentwicklung und Sternmodelle **3**
2.1 Morphologie der Sterne . . . 3
2.1.1 Braune Zwerge . . . 3
2.1.2 Hauptreihe . . . 3
2.1.3 Hertzsprung Gap . . . 5
2.1.4 Rote Riesen . . . 5
2.1.5 Horizontalaststerne . . . 5
2.1.6 Blueloop-Sterne . . . 5
2.1.7 AGB-Sterne . . . 6
2.1.8 Planetarische Nebel . . . 6
2.1.9 Weisse Zwerge . . . 6
2.1.10 Neutronensterne . . . 6
2.1.11 Schwarze Löcher . . . 7
2.1.12 Superriesen . . . 7
2.1.13 Supernovae . . . 7
2.1.14 Wolf-Rayet-Sterne . . . 7
2.2 Entwicklungsphasen und -prozesse . . . 8
2.2.1 Zentrales Wasserstoffbrennen . . . 8
2.2.2 H-Schalenbrennen . . . 8
2.2.3 He-Flash . . . 8
2.2.4 Zentrales Heliumbrennen . . . 8
2.2.5 He-Schalenbrennen . . . 8
2.2.6 C/O-brennen, N-brennen, Nukleosynthese . . . 8
2.2.7 Massenverluste . . . 9
2.2.8 Finale Abkühlphasen . . . 9
2.3 Sternmodelle . . . 10
2.3.1 Stellare Entwicklungs Codes . . . 10
2.3.2 Welche Sternmodelle gibt es? . . . 11
2.3.3 Ergänzende Modelle und analytische Formeln . . . 12
2.3.4 Die Weiterentwicklung bestehender Modelle . . . 12

3 Kurzüberblick UML / Java **13**
3.1 Das UML Modell und seine Elemente . . . 13
3.2 Die Programmiersprache Java . . . 16
3.3 Java-Applets im WWW . . . 17
3.4 Simulation und Darstellung von Sternentwicklung . . . 19

4 javaHRD **21**
4.1 Die Datengrundlage des Programms . . . 21
4.1.1 Die Daten-Pakete von javaHRD . . . 21
4.1.2 Die Struktur der Datenpakete . . . 24
4.1.3 Festlegung der Daten-Referenzen mit den Klassen `LogL`, `LogT` und `Years` . . . 24
4.2 Darstellung der Entwicklungswege und Animation . . . 26
4.3 Steuerung und Funktion . . . 28
4.4 Java-Klassen . . . 29
4.4.1 Die Klasse `JavaHRD` . . . 29
4.4.2 Die Klasse `HRDPanel` . . . 29
4.4.3 Das javaHRD-Layout . . . 30
4.4.4 Das Interface `sVar` . . . 31
4.4.5 Die Klasse `StarInHRD` . . . 31
4.4.6 Die Klasse `AnimationVariable` . . . 33
4.4.7 Die Klasse `Star` . . . 34
4.4.8 Die Klasse `Track` . . . 35
4.4.9 Die Klassen `Diagram` & `DiagramBackground` . . . 37
4.4.10 Die Klassen `ZAMS`, `TAMS` und `Hayashi` . . . 37
4.4.11 Die Klassen `HburnCoreArea`, `HburnShellArea`, `HeBurnCoreArea` und `HorizontalBranch` . . . 38
4.4.12 Die Klasse `Cross` . . . 38
4.4.13 Die Klasse `TimeProgress` . . . 38
4.4.14 Die Klasse `StageLabel` . . . 39
4.4.15 Die Klasse `HighlightStage` . . . 40
4.4.16 Die Klasse `Values` . . . 40
4.4.17 Die Klasse `FreezeTrack` . . . 40
4.4.18 Die Klasse `StarColor` - Zeichnen der Sternfarbe in Abhängigkeit der Temperatur . . . 40
4.4.19 Simulation des Sternradius . . . 41
4.4.20 Die Klasse `InfoBox` . . . 41
4.5 Funktionsweise im Detail . . . 42
4.5.1 Die innere Klasse `TimerAction` . . . 42
4.5.2 Die javaHRD-Animation . . . 43
4.5.3 Die inneren Klassen von `HRDPanel` . . . 43
4.5.4 Die Methoden von `StarInHRD` . . . 47

5 Didaktische Aspekte **51**
5.1 Interaktivität . . . 51
5.2 Darstellung der zeitlichen Entwicklung . . . 52
5.3 Didaktischer Mehrwert . . . 52

6 Zusammenfassung und Ausblick **55**

A javaHRD Quelldateien (Auszug) **57**
A.1 JavaHRD.java . . . 57
A.2 sVar.java . . . 58
A.3 Track.java . . . 59
A.4 Star.java . . . 60

Kapitel 1

Einleitung

Sterne weisen eine große Vielfalt an Strukturen auf. Alle Sterne, außer den Braunen Zwergen, beginnen ihre Entwicklung mit der Fusion von Wasserstoff zu Helium, um in den folgenden Entwicklungsphasen die unterschiedlichsten Entwicklungswege zu durchlaufen. All das wird durch die Anfangsmasse der Sterne gesteuert. Diese Vielfalt der Entwicklung einleuchtend, umfassend und einschließlich des zeitlichen Ablaufs in seinen unterschiedlichen Zeitskalen darzustellen, ist eine Herausforderung für die Astrophysik.

Java Applets erfreuen sich in den letzten Jahren zunehmender Beliebtheit in verschiedenen naturwissenschaftlichen Bereichen. Ob anschauliche Darstellung komplexer Zusammenhänge oder virtuelle Durchführung von Versuchen, ob nützliche Lernhilfen oder komplizierte Werkzeuge der Wissenschaft, Java Applets spielen in der heutigen Onlinewelt zunehmend eine wichtige, auch didaktische Rolle.

Der wesentliche Vorteil eines Java-Applets zur Darstellung von Inhalten, ganz gleich welcher Art, liegt in der Plattformunabhängigkeit der objektorientierten Programmiersprache Java. Verwendete Hardware, Betriebssysteme und Internet-Browser sind der Darstellung von Information untergeordnet und austauschbar. Solange eine virtuelle Java Maschine installiert ist und die Übersetzung des Programmcodes vom jeweiligen System vollzogen werden kann, erfolgt die Darstellung auf allen Systemen im gleichen Layout und mit gleichen Funktionen.

Die Zielsetzung dieser Diplomarbeit ist die zeitliche Darstellung von Sternentwicklungswegen im Hertzsprung-Russell-Diagramm (HRD), dem Referenzdiagramm der Astrophysik (siehe theoretisches HRD in Abb. 1.1), in dem zwei für die Sternoberfläche charakteristische Parameter gegeneinander aufgetragen werden. Mit der Programmiersprache Java wird ein plattformunabhängiges Java Applet auf einem Webseitenangebot öffentlich präsentiert, um schnell und übersichtlich einen Überblick über die Entwicklungswege verschiedener Sterne und deren Entwicklungsphasen zu erhalten.

Die Umsetzung des Diagramms erfordert eine Auseinandersetzung mit gängigen Entwicklungsmodellen von Sternen und eine Betrachtung ihrer Morphologie und Entwicklungswege, sowie deren Darstellung im Hertzsprung-Russell-Diagramm (die theoretische Grundlage dazu liefern [1], [2] und [3]). Weiterhin ist eine Bewertung und Diskussion der dem programmierten Java Applet zugrunde liegenden Daten nötig. Verschiedene Aspekte der Java Programmiersprache, wie UML Modelle und Java Applets werden angesprochen. Die Idee von *javaHRD* wird erläutert und die Funktionsweise des programmierten Applets beschrieben und diskutiert. Abschließend wird der didaktische Wert der Arbeit erörtert und ein Ausblick gegeben.

Das Ergebnis dieser Diplomarbeit wird auf der Homepage der Sternwarte Bonn zur Verfügung gestellt.[1] Das Angebot richtet sich sowohl an Laien, als auch an Studenten der Astronomie und natürlich alle, die sich für die Entwicklung der Sterne interessieren. Die Webseite leistet einen grundlegenden Beitrag zum anschaulichen und inhaltlichen Verständnis des Hertzsprung-Russell-Diagramms.

[1] http://www.javaHRD.de

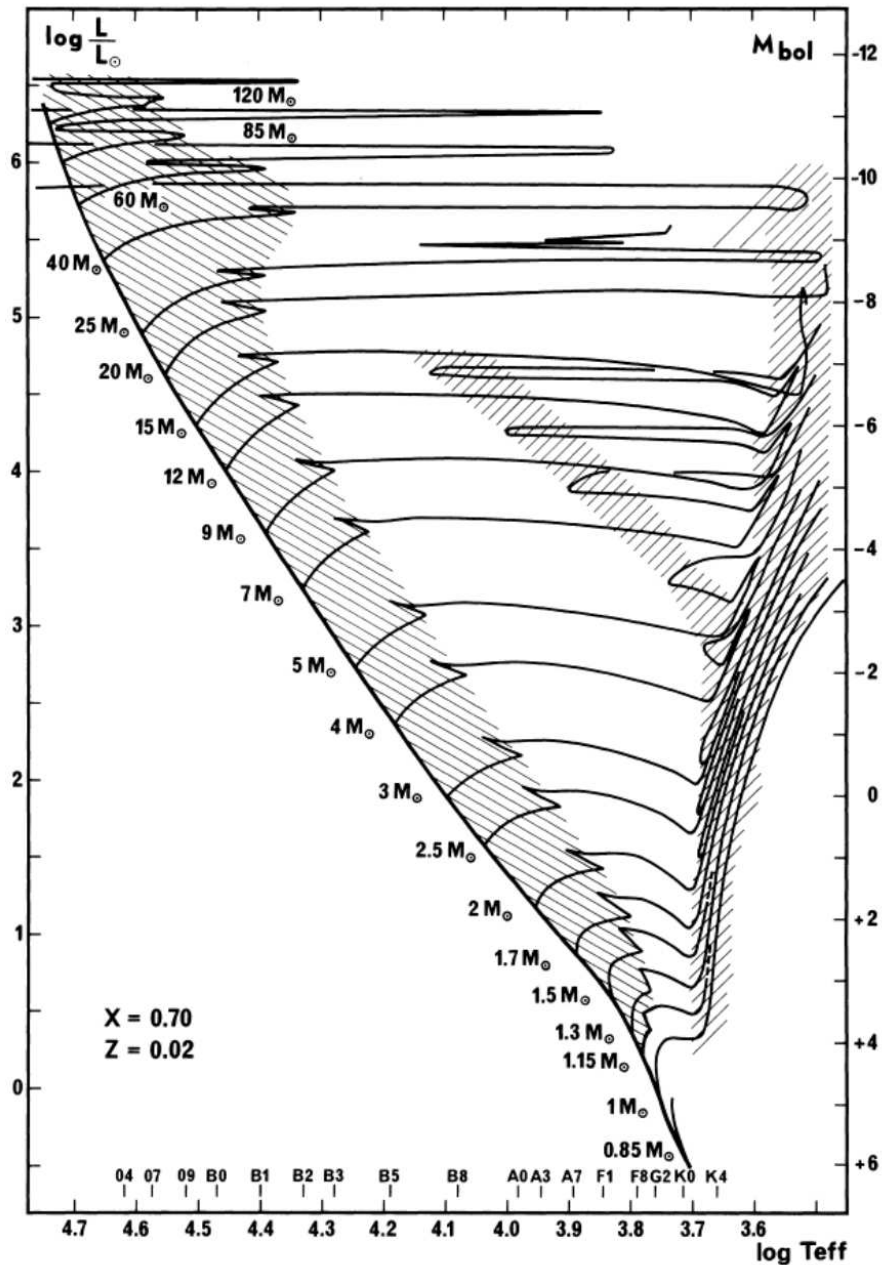

Abbildung 1.1: Beispiel für ein HRD nach Maeder und Meynet (siehe [4]). Sterne entwickeln sich vom Hauptreihenstern zum Roten Riesen und danach in vielen verschiedenen weiteren Stadien. Das Diagramm zeigt für eine Reihe von Anfangsmassen, wie die Entwicklung abläuft und wie die beobachtbaren Merkmale sich mit der Zeit ändern. Die schraffierten Bereiche deuten lang andauernde Entwicklungsstufen an.

Kapitel 2

Sternentwicklung und Sternmodelle

2.1 Morphologie der Sterne

Das Universum umfasst eine Vielzahl von Objekten deren morphologische Bezeichnungen oft vom Erscheinungsbild der ersten optischen Beobachtung abgeleitet sind. Auch bei Sternen haben sich solche Bezeichnungen der ersten Stunde, meistens korrekt, manchmal irreführend, durchgesetzt.

Besonders helle, am Himmel im optischen Bereich rötlich leuchtende Sterne mit einer relativ grossen absoluten Helligkeit und einem grossen Radius wurden als Rote Riesen bezeichnet, kleine, kompakte Sterne mit hoher Leuchtkraft als Weisse Zwerge. Nicht immer stimmte die erste Bezeichnung mit dem tatsächlichem Wesen und den Eigenschaften der beobachteten Objekte überein. So war der durch F. Wilhelm Herschel geprägte Begriff der Planetarischen Nebel, zunächst ein Fehlgriff, da man neblige Objekte beobachtete, die Ähnlichkeiten zu planetaren Objekten (Uranus, etc.) vermuten liessen (siehe [5]). Spätere detailliertere Beobachtungen zeigten, dank fortschreitender Technik und zunehmender Qualität der optischen Teleskop-Auflösung, dass Planetarische Nebel ganz andere Eigenschaften als zunächst angenommen besitzen.

In den folgenden Abschnitten werden die einzelnen morphologischen Begriffe der Sternentwicklung und deren wichtigsten Eigenschaften zusammengestellt. Dabei wird eine Einteilung vorgenommen, die auch im programmierten javaHRD wiederzufinden ist.

Zuvor soll an eine kurze und einfache Sterndefinition erinnert werden: *Ein Stern ist eine kugelförmige Ansammlung von Gas, das durch thermonukleare Fusionsprozesse Energie freisetzt.*

2.1.1 Braune Zwerge

Stellare Objekte, deren Masse gerade noch ausreicht um der o.g. Definition zu genügen, sind Braune Zwerge. Die Anfangsmasse von Braunen Zwergen reicht nicht aus, um Temperaturen zu erreichen, die hoch genug sind um die Fusion von Wasserstoff zu zünden. Braune Zwerge gewinnen eine Weile Energie, meistens durch Fusion von Deuterium und manchmal durch Fusion von Lithium, und werden anschliessend zu passiven Objekten. Der Massenbereich dieser Objekte liegt zwischen 0,012 und 0,08 $M_\odot$. Man findet Sie im HRD im Bereich niedriger Leuchtkräfte und Temperaturen, also im unteren Bereich des Diagramms.

2.1.2 Hauptreihe

Das Leben aller Sterne mit Massen grösser als 0,08 $M_\odot$ beginnt, sobald die Temperatur im Sterninnern hoch genug ist, um den ersten thermonuklearen Fusionsprozess des zentralen Wasserstoffbrennens auszulösen. Vergleicht man die gesamte Lebensdauer eines Sterns mit der Dauer

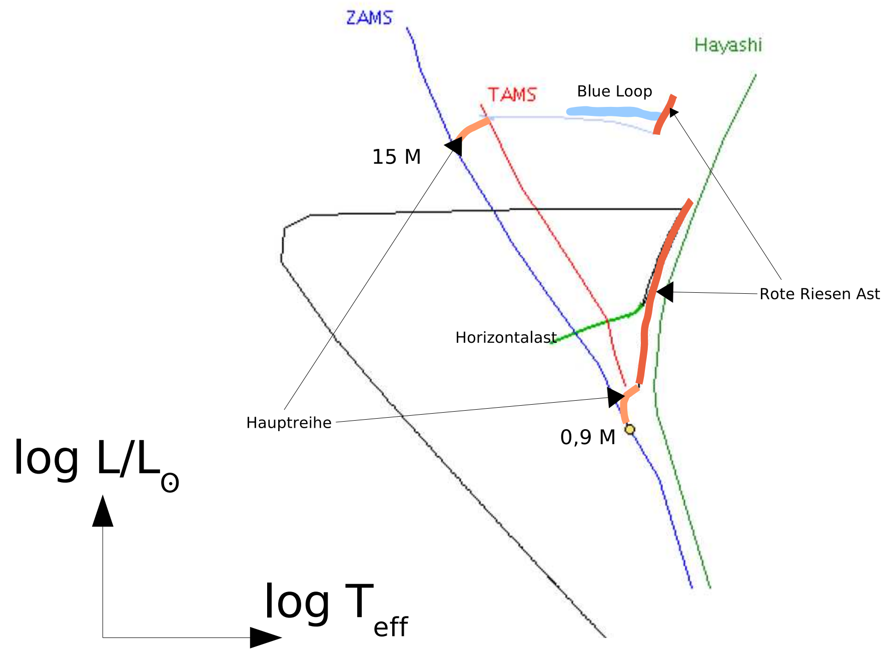

Abbildung 2.1: Skizze der Morphologie I: Auszug aus dem javaHRD mit zwei Entwicklungswegen (0,9 $M_\odot$ und 15 $M_\odot$). Hier wurden Hauptreihe, Rote Riesen Ast und der Blue Loop (bei 15 $M_\odot$) markiert. Ausserdem sind Horizontalast, ZAMS, TAMS und Hayashi Linie zu sehen

des Hauptreihenabschnitts, so ist dieser der stabilste und am längsten andauernde. Während der gesamten Brennphase findet im Fusionsbereich eine kontinuierliche Änderung des Molekulargewichts statt. Weiterhin spielt bei Sternen mit Massen grösser als 1,2 $M_\odot$ innere Konvektion für den folgenden Verlauf eine wichtige Rolle.

Bedingt durch innere Fusionsprozesse und darauffolgende Anpassung der Oberfläche nimmt die Leuchtkraft im Laufe dieser Phase leicht zu und die Oberflächentemperatur nimmt ab. Zum Schluss dieser Phase gibt es abhängig von der Anfangsmasse des Sterns zwei mögliche Entwicklungen, die das Ende des Wasserstoffbrennens im Kern des Sterns einleiten. Bei Sternen ohne innere Konvektion sammelt sich Helium im Kern und eine wasserstoffbrennende Schale wandert langsam nach aussen. Das H-Brennen verschwindet hier also mit der Zeit aus dem Kernbereich. Bei Sternen mit innerer Konvektion erlischt das zentrale Wasserstoffbrennen ab einem bestimmten Verhältnis von Helium zu Wassertoff im Kern. Dieser kontrahiert dann, wobei durch die folgende, anpassende Kontraktion die Temperatur der darauf ruhenden Schichten ansteigt.

Was passiert demnach im HRD? Hauptreihensterne befinden sich im HRD innerhalb des Hauptreihen-Bandes, das sich von hohen Leuchtkräften und Temperaturen quer nach unten rechts zu niedrigen Leuchtkräften und Temperaturen zieht. Die beiden Grenzen dieses Bandes bilden die sogenannte Zero Age Main Sequence (ZAMS) und die Terminal Age Main Sequence (TAMS) (siehe Abb. 1.1). Ein junger, gerade geborener Stern erscheint auf der ZAMS und wandert während der Hauptreihen-Phase nach rechts oben Richtung TAMS, dem Punkt bei dem das zentrale H-Brennen stoppt, oder aus dem Kernbereich nach Aussen in eine Schale wandert. Bei Sternen mit innerer Konvektion ist dieser Punkt klar durch einen Temperatur-Umkehrpunkt ausgezeichnet, während der Übergang bei Sternen ohne innere Konvektion kontinuierlich ist. Die Hayashi Linie zieht eine Grenze zum rechten, instabilen Bereich im HRD, in dem keine Sterne vorkommen.

2.1.3 Hertzsprung Gap

Ursprünglich bezeichnet der Hertzsprung Gap den Bereich zwischen der oberen Hauptreihe und dem Rote Riesen Ast, in dem die Wahrscheinlichkeit ein Stern zu finden, aufgrund der rasanten, stellaren Entwicklung während dieser Phase, klein ist. Es ist aber in vielen Publikationen verbreitet, diese kurz andauernde Entwicklungsphase zwischen der Hauptreihe und der Rote Riesen Phase mit diesem Begriff zu bezeichnen.

Zu Beginn dieser Phase wird das H-Brennen in der Schale um den He-Zentralbereich gezündet. Dadurch steigt die Temperatur dieser nun aktiven Schale an und deren umhüllende Schichten expandieren. Gleichzeitig kontrahiert der He-Kern weiter. Die Expansion der Basisschichten der gesamten Sternhülle erreicht im weiteren Verlauf ein Limit, ab dem die äusseren Oberflächenschichten sich auf der Suche nach neuem Gleichgewicht der inneren Entwicklung des Sterns beginnen anzupassen. Zu diesem Zeitpunkt ist bereits das Ende der Hertzsprung-Gap-Phase erreicht. Während diesem Abschnitt nehmen Oberflächentemperatur und Leuchtkraft ab.

Im HRD wandert der Stern nach rechts, gleichzeitig etwas nach unten und landet schliesslich auf dem Rote Riesen Ast.

2.1.4 Rote Riesen

Sterne, die sich in der Phase roter Riesen befinden, sind zunächst einmal Schalenbrenner. Das bedeutet, das zuvor gezündete H-Schalenbrennen bestimmt die weitere Entwicklung. Der isothermische He-Kern wird durch das H-Schalenbrennen weiter mit Wasserstoff versorgt, so dass die Kerntemperatur langsam zunimmt. Die hohen Temperaturen führen zur Degeneration des Gases, wodurch der Kern weiter kontrahieren kann. Der höhere Temperaturgradient, bedingt durch die Nähe der brennenden Schale zur Oberfläche, führt zu zunehmender Konvektion. Die Gesamtstruktur des Sterns verändert sich, so dass sich die Oberflächenschichten der inneren Entwicklung anpassen und expandieren. Abhängig von der Gesamtmasse gibt es nun zwei mögliche Entwicklungswege, die mit dem Zünden des He-Kernbrennens einsetzen. Bis zu einer Anfangsmasse zwischen 0,6 $M_\odot$ und 2,5 $M_\odot$ kommt es zum Helium-Flash. Diese Sterne landen auf dem Horizontalast. Sterne mit höheren Massen bis zu 8 $M_\odot$ brennen im Kern stabil Helium zu höheren Elementen und durchlaufen einen Blueloop.

Im HRD wandern Sterne in dieser Phase langsam den Rote Riesen Ast hoch. Bei geringer Abnahme der Effektivtemperatur nimmt die Leuchtkraft dramatisch zu. Die Position des Riesenastes im HRD hängt dabei im wesentlichen von der Opazität der äusseren Sternschichten ab.

2.1.5 Horizontalaststerne

Horizontalast Sterne sind solche Sterne, die nach einem Helium-Flash den Riesenast blitzartig verlassen und auf dem Horizontalast auftauchen. Dabei bestimmt die Dicke der umhüllenden Schichten die Farbe und damit die Position im HRD Diagramm. Nackte He-brennende Sterne, also solche ohne Hülle, würden auf einer Sequenz parallel zur Hauptreihe und links davon zu sehen sein, eine Sequenz, die als Helium-Hauptreihe bezeichnet wird. Je dichter die umgebende Hülle nun ist, desto röter, also zu niedrigen Temperaturen hin verschoben, erscheint der Stern im HRD. Abhängig von der verbleibenden Kernmasse, zeichnen sich die unterschiedlichsten Entwicklungen dieser Sterne ausgehend vom Horizontalast ab.

2.1.6 Blueloop-Sterne

Wie in Kapitel 2.1.4 bereits erwähnt gibt es neben der Entwicklung als Horizontalast-Stern einen zweiten Weg, den Sterne mit einer Anfangsmasse zwischen 2,5 und 8 $M_\odot$ (unter Umständen sogar bis zu 15 $M_\odot$) gehen. In dieser Phase fusioniert im Kern Helium zu Kohlenstoff. Abgesehen davon existiert meist eine H-brennende Schale, die den Kern mit Helium versorgt. Nun gibt es wiederum zwei Möglichkeiten der Entwicklung, die abhängig von der Anfangsmasse durchlaufen werden. Bei einer Anfangsmasse unter 4 $M_\odot$ wird der Kern kontinuierlich mit Kohlenstoff

versorgt und es entsteht eine He-brennende Schale, welche die H-brennende Schale nach aussen in kühlere Temperaturbereiche drückt und dort die H-Fusion stoppt. Der Stern entwickelt sich zum Heliumschalenbrenner und versorgt den Kern weiterhin mit Kohlenstoff. Sobald die Helium-Schale weit genug nach aussen vorgedrungen ist, zündet erneut das H-Schalenbrennen. Hat der Stern eine Anfangsmasse von mehr als 4 $M_\odot$, steigen die Temperaturen im Kern auf Werte, die hoch genug sind um das Kohlenstoffbrennen zu zünden.

Im HRD wandert ein Blueloop Stern in einer Art Schleife nach links, verweilt dort, um danach auf den AGB-Ast zuzusteuern.

2.1.7 AGB-Sterne

Ist Helium im Kern einmal aufgebraucht, kommt der Stern abermals in eine schalenbrennende Phase. Die weitere Entwicklung ähnelt der Phase der Roten Riesen. Durch das Schalenbrennen wird dem Kern weiter Kohlenstoff und Sauerstoff hinzugefügt und die Kerntemperatur steigt. Aber nur in Sternen mit Anfangsmassen grösser als 4 $M_\odot$ wird eine Temperatur erreicht, die hoch genug ist, um die Fusion von Kohlenstoff zu höheren Elementen zu zünden. Da der Kern auf Grund der hohen Temperaturen degeneriert ist, kommt es zum C/O-Flash. Bei AGB-Sternen kommt es zu erheblichen Massenverlusten. Bei kleineren Anfangsmassen sind diese Verluste an der Oberfläche so stark, dass die Zündung des C/O Brennens nicht erreicht werden kann. In beiden Fällen kontrahiert der Kern und die äusseren Schichten expandieren stark. Es wird die Post-AGB Entwicklung eingeleitet, welche die Entwicklung der abgeblasenen Sternhülle zum planetarischen Nebel und schliesslich die Entwicklung des Zentralsterns zum Weissen Zwerg beschreibt.

Ein AGB Stern wandert im HRD nach rechts oben, wobei die Leuchtkraft stark ansteigt und die Oberflächentemperatur abnimmt.

2.1.8 Planetarische Nebel

Während dieser Phase kontrahiert der Stern bei fast gleichbleibender Leuchtkraft, wodurch seine Oberflächentemperatur stark ansteigt. Durch die hohen Temperaturen an der Sternoberfläche und die dadurch erzeugte Strahlung wird das zuvor weggeblasene Sternhüllengas ionisiert. Die leuchtenden HII Regionen werden beobachtet. Solche Objekte werden als planetarische Nebel bezeichnet. Schliesslich erlöschen alle Fusionsprozesse und der Zentralstern strahlt nur noch durch die verbleibende gravothermische Energie.

Im HRD wandert der Stern in einer horizontalen konstant nach links.

2.1.9 Weisse Zwerge

Sterne mit Anfangsmassen zwischen 0,6 und 8 $M_\odot$ entwickeln sich zu Weissen Zwergen, deren Masse zwischen 0,5 und 1,4 $M_\odot$ liegt. Sobald die Fusion im Kern das Sterns erloschen ist, kontrahiert der Zentralstern in kurzer Zeit sehr stark und die Leuchtkraft sinkt dabei ab. Die Entwicklung bremst nun, bedingt durch gravothermisch freigesetze Energie und Neutrinoverluste, mehr und mehr ab. Hat der Stern ein sehr kompaktes Stadium erreicht, setzt sein Kühlungsprozess ein, was die Entwicklungsgeschwindigkeit weiter bremst. Leuchtkraft und Temperatur nehmen von nun an kontinuierlich ab. Im HRD wandern Weisse Zwerge nach rechts unten.

2.1.10 Neutronensterne

Neutronensterne entstehen im Rahmen einer Supernova vom Typ II, wie sie beispielsweise beim Kollaps des stellaren Zentralbereiches mit einer Masse zwischen 1,44 $M_\odot$ (Chandrasekhar-Grenze) und etwa 3 $M_\odot$ stattfindet. Diese sehr heissen Objekte kühlen kontinuierlich aus.

Der Entwicklungsweg im HRD verläuft ähnlich dem von Weissen Zwergen nur bei sehr viel höheren Temperaturen.

2.1.11 Schwarze Löcher

Stellare Schwarze Löcher sind astronomische Objekte die am Ende der Entwicklung sehr massereicher Sterne steht (Anfangsmasse $> 20\ M_\odot$). In dieser Phase krümmt dessen starkes Gravitationsfeld die Raumzeit so stark, dass weder Materie noch Licht oder andere Information entweichen können. Die Grenze dieses Bereiches heißt Ereignishorizont.

Schwarze Löcher tauchen nicht im HRD auf, da diese Objekte naturgemäss keine Strahlung emitieren und somit keine Leuchtkraft besitzen.

2.1.12 Superriesen

Superriesen sind entwickelte Sterne mit extrem hohen Anfangsmassen. Blaue Superriesen sind solche Sterne die bereits zu Beginn der Hauptreihe bei sehr hohen Leuchtkräften eine grosse Ausdehnung haben. Rote Superriesen sind Sterne mit Anfangsmassen grösser 15 $M_\odot$, die sich in der Phase des H-Schalenbrennens befinden.

2.1.13 Supernovae

Supernovae sind Sternexplosionen und treten bei Sternen mit einer Anfangsmasse grösser als 8 $M_\odot$ auf. Es gibt verschiedene Supernova Typen, die in unterschiedlichen Intervallen der Anfangsmasse am Ende der AGB-Phase zu stande kommen. Die Überreste von Supernovae sind in der Regel Neutronensterne.

2.1.14 Wolf-Rayet-Sterne

Wolf-Rayet-Sterne sind heisse und sehr massereiche Sterne die sich am Ende der Entwicklung durch hohen Massenverlust auszeichnen. Eine weitere Eigenschaft ist das Auftreten von starken He-Emmissionslinien.

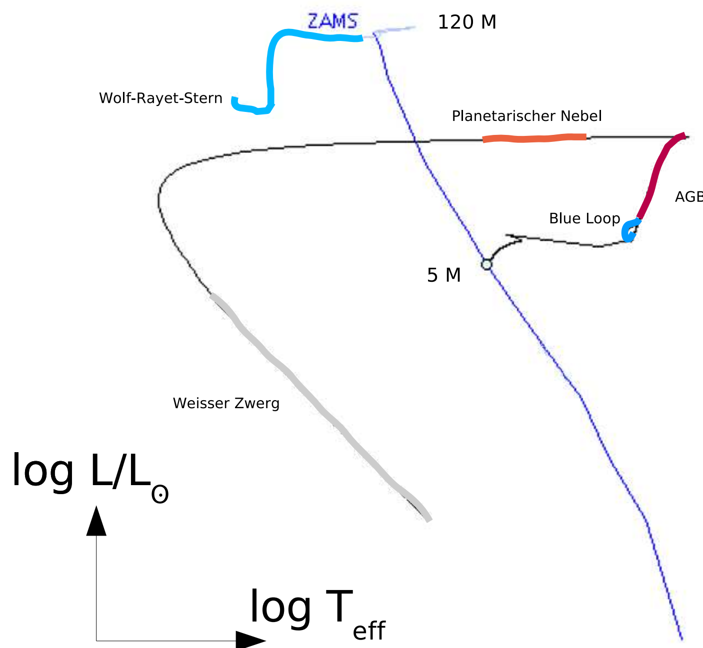

Abbildung 2.2: Skizze der Morphologie II: Auszug aus dem javaHRD mit zwei Entwicklungswegen (5 $M_\odot$ und 120 $M_\odot$). Hier sind PN- und WZ- und AGB-Phase markiert (bei 5 $M_\odot$). Ausserdem ist der Bereich des WR-Sterns bei 120 $M_\odot$ zu sehen.

2.2 Entwicklungsphasen und -prozesse

Solange ein Stern existiert, entwickelt er sich ständig fort. Dabei spielen verschiedene Prozesse eine Rolle, welche die Entwicklung im einzelnen steuern. Die folgende Übersicht der Sternentwicklungsphasen, die abhängig von der Masse durchlaufen werden, soll darlegen, welche physikalischen Prozesse in der jeweiligen Phase von Bedeutung sind und wie diese die Entwicklung beinflussen.

2.2.1 Zentrales Wasserstoffbrennen

Das zentrale Wasserstoffbrennen leutet die erste Entwicklungsphase eines jeden Sterns ein. Dieser thermonukleare Fusionsprozess setzt ein, sobald die Temperatur in Kerninnern so hoch ist, dass Wasserstoff zu Helium verschmelzen kann. Diese Umwandlung erfolgt, abhängig von der Kerntemperatur in zwei unterschiedlichen Prozessen, dem Proton-Proton Prozess und dem CNO-Zyklus.

2.2.2 H-Schalenbrennen

Sobald im Kern eines Sterns keine Fusionsaktivität mehr auftritt und die Fusion sich auf eine Schale um den Kern verlagert, spricht man von einem Schalenbrenner. Die Temperatur im Kern ist in diesem Fall nicht mehr hoch genug, um das Produkt der Fusion zu zünden. Thermonukleare Brennprozesse spielen sich dann in einer Schale um den Kern herum ab. Dabei wird der von der Hülle umgebene Zentralbereich mit den Produkten des Brennprozesses, in diesem Fall Helium, versorgt. Sobald die Temperatur im Kerninnern so hoch ist, dass die Fusion von Helium gezündet werden kann, schreitet die Entwicklung des Sterns zur nächsten Phase voran. Es kommt wie bereits beschrieben, in Abhängigkeit von der Anfangsmasse, entweder zum Helium-Flash bei Sternen ohne Konvektion, oder zum unspektakulären Zünden des zentralen Heliumbrennens.

2.2.3 He-Flash

Ist das Heliumbrennen einmal gezündet, kommt es in Sternen mit einer Anfangsmasse kleiner als 2,5 $M_\odot$ zu einer rasanten Entwicklung. Da die Kernmaterie bei solchen Sternen entartet ist, führt die Energieerzeugung des Heliumbrennprozess zu immer mehr Fusion. Dieser Prozess läuft in einer sehr kurzen Zeit ab und wird daher Helium Flash genannt. Der Stern wird dadurch aber so heiss, dass die Entartung der Materie wieder aufgehoben wird. Im folgenden brennt der Stern stabil Helium.

2.2.4 Zentrales Heliumbrennen

Das Zentrale Heliumbrennen wird nur bei entsprechend hohen Temperaturen im Kerninneren des Sterns erreicht. Die Temperaturen müssen so hoch sein, dass der Triple-Alpha Prozess starten kann, dessen Hauptprodukt Kohlenstoff ist.

2.2.5 He-Schalenbrennen

Wenn der Kohlenstoffanteil im Kerninneren des Sterns so hoch geworden ist, dass im zentralen Bereich kein Heliumbrennen mehr stattfindet, bildet sich eine Schale aus einer heliumbrennenden Schicht, die den Kern weiter mit Kohlenstoff versorgt. In diesem Stadium gibt es meist noch eine weiter aussenliegende H-brennende Schale. Man spricht in diesem Fall vom Zwei-Schalen-Brennen.

2.2.6 C/O-brennen, N-brennen, Nukleosynthese

Bei sehr massereichen Sternen werden bei immer höheren Temperaturen weitere Fusionsprozesse im Kerninnernen ausgelöst. Es kommt durch die extremen Bedingungen zu einer Vielzahl von Prozessen: C/alpha Prozess, O-Brennen, N-Brennen und Nukleosynthese durch r- und p-Prozesse, bei massseärmeren Sternen zum langsameren s-Prozess.

2.2.7 Massenverluste

Die Masse eines Sterns nimmt im Laufe eines Sternlebens ab. Je nach Entwicklungsphase spielen Massenverluste eine mehr oder weniger wichtige Rolle, welche die weitere Entwicklung beeinflussen. Dies wird besonders deutlich bei dramatischen Ereignissen, wie zum Beispiel dem He-Flash, wo die übrig gebliebene Sternmasse die folgende Entwicklung stark beinflusst, oder bei der späten AGB-Phase in der Reimerswinde, thermische Pulse und andere Mechanismen die für den Zentralstern verbleibende Masse bestimmen.

2.2.8 Finale Abkühlphasen

Nachdem alle Sterne mit verschiedenen Anfangsmassen die unterschiedlichen, charakteristischen Phasen mit den zugehörigen Prozessen durchlaufen haben, gelangen sie in eine finale Kühlphase. Diese beginnt, sobald Fusionsprozesse erloschen sind. Es stehen dann keine thermonukleraren Prozesse mehr für die Energieerzeugung zur Verfügung. Der Stern kühlt also nach und nach ab, was im HRD auf einer linearen Sequenz zu sehen ist. Sterne bis 8 $M_\odot$ kühlen als Weisse Zwerge. Sterne die in einer Supernova explodieren, kühlen als Neutronensterne aus.

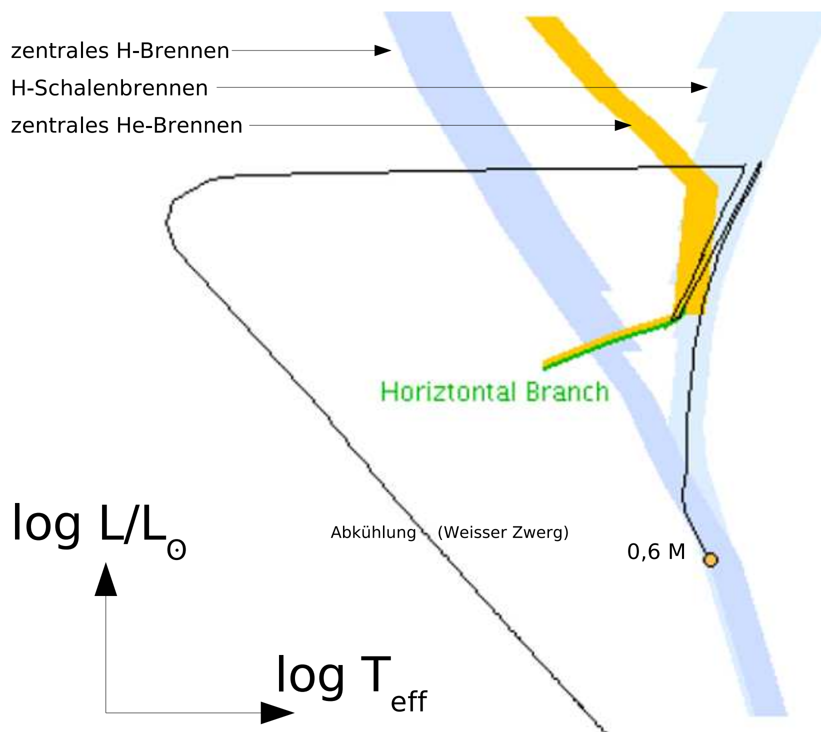

Abbildung 2.3: Ein Auszug aus dem javaHRD, der die Zonen mit H-Brennen im Kern, H-Schalenbrennen und He-Brennen im Kern skizziert. Ausserdem ist die Abkühlphase des Sterns als Weisser Zwerg zu erkennen.

2.3 Sternmodelle

Wie werden Stellare Modelle entwickelt und kalkuliert? Es ist von entscheidender Bedeutung sich darüber klar zu werden, dass die Beschreibung der Sternentwicklung immer nur so gut sein kann, wie die Modelle die dazu herangezogen werden. Mathematische Gleichungen zur Beschreibung der Sternstruktur sind seit langem bekannt und klar zu beschreiben. Auch exakte Lösungen der Differentialgleichungen stellen in der heutigen Zeit, in der leistungsstarke und schnelle Computer komplizierte Rechnungen übernehmen, kein Hinderniss mehr dar. Vielmehr sind die Randbedingungen wichtig, die zur Lösung verwendet werden. Sind diese im Sterninneren trivial zu lösen, gilt dies in keinem Fall für die äusseren Randbedingungen.

Bei deren Auswahl geht es nun darum, Temperatur und Druck in den untersten Schichten der Sternatmosphäre zu bestimmen. Dies geschieht zum einen durch die Annahme von empirischen Beziehungen zur thermischen Schichtung, zum anderen werden theoretische Näherungen (Eddington approximation) verwendet. Ein weitaus strengerer Weg wäre es, Berechnungsergebnisse von Modelatmosphären zu verwenden, und diese für die Randbedingungen zu verwenden. Im Allgemeinen werden Modellatmosphären in einer planparallelen Geometrie gerechnet. Es wird die Gleichung zum hydrostatischen Gleichgewicht zusammen mit der frequenzabhängigen Strahlungstransportgleichung (wenn nötig inklusive Konvektion) und die zugehörige Zustandsgleichung gelöst. Die Entwicklung und Kalkulation von Sternmodellen erfordert das grundlegende Verständnis von drei wesentlichen Themengebieten.

Erstens die Physik der Sterne, also das physikalische Verhalten der Sternmaterie mit den thermischen Charakteristika im Sterninneren und der Sternatmosphäre. Dies wird durch Betrachtung von Opazität, Zustandsgleichung, Energieerzeugungsrate und Energieverlust von Neutrinos abgedeckt. Zweitens mikroskopische Mechanismen, wobei atomare Diffusion, Strahlungsüberlagerung und Strahlungstransport zu berücksichtigen sind. Zuletzt die makroskopischen Mechanismen, die die Durchmischung der Sternmaterie beschreiben. Dabei spielen Temperaturgradient und reale Ausdehnung der Konvektionszone eine wesentliche Rolle.

Eine ausführliche Beschreibung zu diesem Thema findet man in einer Veröffentlichung zu stellaren Entwicklungsmodellen von S. Cassisi (siehe [6]).

2.3.1 Stellare Entwicklungs Codes

Bevor Sternmodelle entworfen werden, müssen Entwicklungscodes zusammengestellt und berechnet werden. Dabei umfasst die Berechnung: Numerik, Zustandsgleichung, Opazität, Energieerzeugung, Konvektion/Strahlungstransport und Atmosphäre. Eine Überblick über die einzelnen Entwicklungscodes geben die Ergebnisse der ESTA/CoRoT Mission (siehe [7] und [8]):

- ASTEC -Aarhus Stellar Evolution Code
  Entwickelt von J. Christensen-Dalsgaard (siehe auch [9])
- CESAM - Code d'Evolution Stellaire Adaptatif et Modulaire
  Entwickelt von P. Morel (siehe auch [10])
- CLES - Code Liegeois d'Evolution Stellaire
  Entwickelt von R. Scuflaire und der *BAG*[1] (siehe auch [11])
- TGEC - Toulouse-Geneva Evolution Code
  Entwickelt von M. Castro (siehe auch [12])
- FRANEC - Pisa Evolution Code
  Entwickelt von S. Degl'Innocenti et al. (siehe auch [13])
- STAROX - Roxburgh's Stellar Evolution Code
  Entwickelt von I. Roxburgh (siehe auch [14])
- GARSTEC - Garching Stellar Evolution Code
  Entwickelt von A. Weiss (siehe auch [15])

[1] Belgian Asteroseismology Group

2.3.2 Welche Sternmodelle gibt es?

Verschiedene Gruppen von Astronomen befassen sich mit Sternentwicklungsmodellen. Die bedeutesten werden im folgenden kurz vorgestellt. Deren Arbeit, Forschungsergebnisse und entworfenen Modelle sind in den entsprechenden Publikationen nachzulesen.

Im wesentlich müssen die entwickelten Modelle die Parameter berücksichtigen, die als Berechnungen der verwendeten stellaren Entwicklungscodes ermittelt werden. Für die einzelnen Parameter wird auf solche Berechnungen oder entsprechende Tabellen zurückgegriffen. Es können aber auch eigene Berechnungen implementiert werden. Die entscheidenen Parameter berücksichtigen auch hier Opazität, Zustandsgleichung, Energieerzeugung und -verlust. Mikro- und makroskopische Effekte wie zum Beispiel Elementdiffusion, Convective Overshoots oder Thermische Pulse werden je nach Modell mehr oder weniger berücksichtigt. Es sei nochmals daran erinnert, dass je mehr Parameter korrekt berücksichtigt werden, und je stärker die zugrundeliegende Physik aktuell gehalten wird, desto hochwertiger und genauer sollten die Ergebnisse der stellaren Entwicklungsmodelle sein. Dies kommt schliesslich auch jeder Darstellung von Sternentwicklung zugute.

Die Genfer Sternmodelle (Geneva Models)

Die Genfer Modelle finden Ihren Ursprung in einer Gruppe von Astronomen des Genfer Observatoriums, die verschiedene Datenbanken mit Ergebnissen zu stellaren Entwicklungsmodellen zur Verfügung stellen (siehe [16]).

Der Grundstein der umfangreichen Datenbank wurden von G.Schaller, D.Schaerer, G.Meynet und A.Maeder in einer Veröffentlichung aus dem Jahre 1989 (vgl. Abb. 1.1) und deren Revision im Jahre 1992 gelegt. Diese präsentieren Daten für die Massenbereiche von 0.8 - 120 $M_\odot$ mit Metallizitäten von Z=0.020 und Z=0.001 (siehe [17]).

Die Opazität wird hier nach Tabellen von Roger & Iglesias (siehe [18]) für Z-Werte von 0.0001 bis 0.03 in die Modelle einbezogen. Ab Z-Werten von 0.03 werden Tabellen von Huebner et al. (1977), bei niedrigen Temperaturen Werte von Kurucz (1991) verwendet. Sowohl nukleare Streuung, Abschirmung und Neutrinoverluste, als auch Ionisation, Konvektionsparameter und Massenverluste massereicher Sterne werden in diesen Modellen berücksichtigt.

Später wurden weitere Genfer Modelle von Maeder und Meynet entwickelt, die zusätzlich die Sternrotation und Ihre Effekte berücksichtigen. Beide Autoren haben seit 1997 eine ganze Reihe von Publikationen unter dem Sammeltitel *Stellar evolution with rotation*[2] veröffentlicht, die unterschiedliche Aspekte der Rotation betrachten.

Die Padova-Sternmodelle (Padova Models)

Die Padova Modelle wurden nach einer Gruppe der gleichnamigen Universitätsstadt benannt (siehe Bressan et al. 1993). Die Ergebnisse dieser Modellrechnungen werden in einer vollständigen Datenbank zur Verfügung gestellt (siehe u.a. [24]), die Massenbereiche von 0.6 - 120 $M_\odot$ abdeckt, und Metallizitäten zwischen Z=0.0004 und Z=0.05 berücksichtigt. Die vorausgesetzte Physik ist homogen für alle Sternentwicklungswege. Die Hauptmerkmale dieser Modelle sind die Annahme der OPAL Opazitäten, ein konstantes Helium/Metall Anreicherungsverhältnis $\Delta Y/\Delta Z$, moderates konvektives Überschwingen im konvektiven Kern und der Massenverlust bei Sternen hoher Masse.

Die Datenbank der Padova Gruppe wurde von Girardi für sehr kleine Z-Werte (z.B. 0.0001) erweitert. (siehe Girardi et al. 1996a und [24])

In späteren Jahren wurde diese Datenbanken ergänzt, aktualisiert und unter Berücksichtigung neuer oder geänderter physikalischer Voraussetzungen neu berechnet. So stellt die Padova Gruppe heute verschiedene Sammlungen von Daten zur Verfügung (siehe [24]).

---
[2] siehe z.B. [19], [20] , [21], [22] oder [23]

Y^2 Stellar Models (Yonsei-Yale) 2001

Diese Modelle wurden von der *Yale Group* entwickelt und stellen eine Datenbank von Isochronen zur Verfügung, die sogenannten Y^2-Isochronen. Die Modellrechnungen der Gruppe berücksichtigen eine Entwicklung, die bereits mit der Sternentstehungsphase der Vor-Hauptreihenphase[3] und nicht erst auf der ZAMS beginnt (siehe [25], [26] und [27]). Darüber hinaus wird hier der innere Prozess des Convective Overshooting in das Modell mit einbezogen.

2.3.3 Ergänzende Modelle und analytische Formeln

Die bisher erwähnten Sternmodelle erfassen all die Entwicklungsphasen, die zwischen ZAMS und AGB-Entwicklung liegen. Die Entwicklungswege Planetarischer Nebel oder Weisser Zwerge dagegen, werden von diesen Modellen nicht berücksichtigt. Daher muss für diese Phasen auf zusätzliche, ergänzende Modelle zurückgegriffen werden um vollständige Entwicklungswege zu erhalten. Entwicklungsmodelle für Weisse Zwerge werden beispielsweise von L.G. Althaus et al. (siehe [28]) zur Verfügung gestellt.

Für die Berechnung kompletter Sternentwicklungswege haben J. R. Hurley et al. analytische Formeln entwickelt, welche die vollständigen Entwicklungswege von ZAMS bis zum Ende der finalen Abkühlphasen berechnen (siehe [29]). Allerdings zeigen diese Entwicklungswege in diesen späten Phasen erhebliche Ungenauigkeiten, die bei der Verwendung berücksichtigt werden müssen (vgl. Kap. 4.1.1). Die individuelle Berechnung von Entwicklungswegen ist im Digital Demo Room der University of Illinois möglich (siehe [30]). Die Daten stehen nach Durchführung der Berechnung als ASCII-Textdatei zum Download zur Verfügung.

2.3.4 Die Weiterentwicklung bestehender Modelle

Die Beispiele des vorhergehenden Kapitels deuten an, dass es immer wieder eine ganze Reihe von Weiterentwicklungen oder Verbesserungen von stellaren Entwicklungsmodellen gegeben hat und geben wird. Dabei werden je nach Stand der Forschung, Entwicklungsparameter verändert oder zusätzliche physikalische Prozesse, Sichtweisen oder Erkenntnisse berücksichtigt. Es steht ausser Frage, dass die Modellierung von Sternen noch lange nicht am Ziel ist und mit fortschreitenden technischen Möglichkeiten und zunehmenden physikalischem Verständnis sich zukünftig weiterentwickeln wird. Grundlage für die gegenwärtig anerkannten Modelle und deren Weiterentwicklung bilden zum jetzigen Zeitpunkt aber wohl die Ergebnisse der im vorigen Kapitel erwähnten Genfer Modelle und die Ergebnisse der Padova Gruppe.

[3]Sternentstehungsphase bevor der Stern auf der MS auftaucht. (pre-main-sequence)

Kapitel 3

Kurzüberblick UML / Java

3.1 Das UML Modell und seine Elemente

Die Unified Modelling Language (UML) beschreibt als Sprache die Spezifikation, Visualisierung, Konstruktion und Dokumentation von Softwaresystem- und Geschäftsmodellen. Als grafische Beschreibungsform bietet diese die Möglichkeit auf Grund von einheitlichen Darstellungsprinzipien Softwaremodelle übersichtlich zu beschreiben und zu diskutieren. Die Elemente der UML bilden verschiedene Diagramme, die je nach Einsatzgebiet verwendet werden. Sie erleichtern den Umgang mit Programmieraufgaben, so dass auch Personen am Entwicklungsprozess beteiligt werden können, die selbst das Programmieren nicht beherrschen.

In diesem Kapitel werden die Elemente des UML Modells (die verschiedenen Diagrammtypen) zusammengefasst, wobei die aufgeführten Elemente nach UML Version 1.3 beschrieben werden und dessen Beschreibungen sich an Martin Fowler (vgl. [31]) anlehnen. Es sei an dieser Stelle weiterhin auf andere übersichtliche Darstellungen in der Literatur hingewiesen (z.B. [32]). Man sollte hier aber auch nicht die gut strukturierte Präsentation der UML Sprache in der aktuellen Version 2.0 auf wikipedia.de (siehe [33]) vergessen.

Klassendiagramme

Klassendiagramme dienen der statischen Beschreibung eines Programmentwurfes. Ein solches Diagramm modelliert die statischen Beziehungen zwischen Systemkomponenten, zum Beispiel zwischen verschiedenen Klassen. Es werden deren Attribute und möglichen Methoden bzw. Operationen ersichtlich. Darüber hinaus lassen sich die wesentlichen Vererbungsprinzipien (Generalisierung), Bedingungen und Schnittstellen anzeigen.

In Abbildung 3.1 ist ein Klassendiagramm mit den drei Klassen `Stern`, `Sonne` und `RoterRiese` zu sehen. Die Klasse `Stern` hat die Attribute *Temperatur* und *Sonnenmasse* und vererbt diese durch Generalisierung auf die Klassen `Sonne` und `RoterRiese`.

Objektdiagramme

Im Objektdiagramm zeigt sich eine Momentaufnahme des Systems und dessen Struktur zu einem bestimmten Zeitpunkt. Ein Objektdiagramm ist dem Klassendiagramm ähnlich. Es werden aber statt allgemeinen Klassen, konkrete Objekte beschrieben. Objekte sind konkrete Exemplare[1] von Klassen, die während des Programmablaufs initialisiert werden. Die Methoden dieser Exemplare sind Operationen, die auf einem Objekt aufgerufen werden. Nebenbei sei bemerkt, dass Objektdiagramme sind Exemplare von Klassendiagrammen sind. Man sollte sich dies nochmals anhand des Beispiels in Abbildung 3.2 klar machen. Hier werden zwei Objekte gezeigt. Einmal

[1] In dieser Arbeit soll der Begriff *Exemplar* für die unter Java Programmierern üblichen Begriffe *Instanz* oder *Ausprägung* verwendet werden.

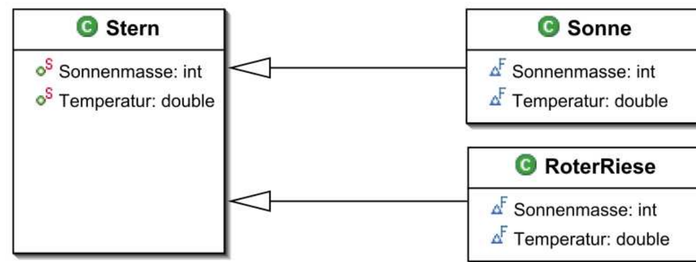

Abbildung 3.1: Beispiel eines UML Klassendiagramms mit den in *Eclipse* (siehe Kap. 3.2) üblichen Symbolen für Variablen und Klassen

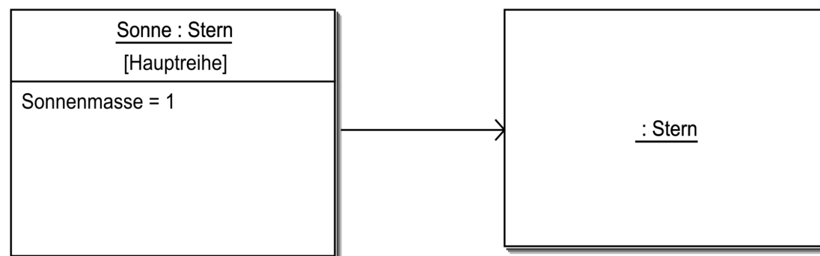

Abbildung 3.2: Beispiel eines UML Objektdiagramms

das konkrete Objekt `Sonne`, das ein Exemplar der Klasse `Stern` darstellt. Dieses Objekt befindet sich im Zustand Hauptreihe und hat natürlicherweise einen Sonnenmassewert von eins. Das zweite Objekt im Diagramm zeigt ein Exemplar der Klasse `Stern`, welches einen Roten Riesen repräsentiert. Durch die Generalisierung (Pfeil) wird gekennzeichnet, dass die Objekte `Sonne` und `RoterRiese` ihre Eigenschaften vom Objekt Stern erben.

Komponentendiagramme

Das Komponentendiagramm ist ein Strukturdiagramm. Es zeigt eine bestimmte Sicht auf die Struktur des modellierten Systems. In einem Komponentendiagramm werden typischerweise Komponenten, deren Schnittstellen und die Zusammenhänge der einzelnen Komponenten der späteren Softwarelösung dargestellt.

In Abbildung 3.3 werden als Beispiel die Hauptkomponenten der in dieser Arbeit vorgestellten javaHRD-Anwendung dargestellt. Ein Benutzer der Anwendung ruft eine HTML Seite auf, welche das javaHRD-Applet einbindet. Der Java-Quellcode verwendet die Daten aus den Tabellen der Sternmodelle.

Anwendungsfalldiagramme (usecase)

Ein Anwendungsfalldiagramm besteht aus einer Menge von Anwendungsfällen und stellt die Anforderungen eines Systems dar. Beziehungen zwischen Akteuren und Anwendungsfällen werden aufgezeigt. Weiterhin wird das äußerlich erkennbare Systemverhalten aus der Sicht eines Anwenders deutlich.

Im Beispieldiagramm 3.4 wird der einfache Anwendungsfall gezeigt, in der die Abfrage einer Internetseite durch einen Student an einen Server gestellt wird.

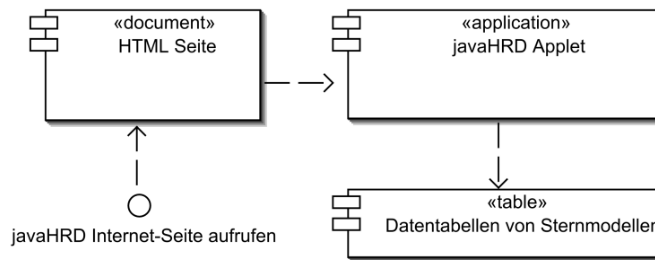

Abbildung 3.3: Beispiel eines UML Komponentendiagramms

Abbildung 3.4: Beispiel eines UML Anwendungsfalldiagramms (usecase)

Sequenz- und Kollaborationsdiagramme

Das Sequenzdiagramm beschreibt die zeitliche Abfolge von Interaktionen zwischen einer Menge von Objekten innerhalb eines zeitlich begrenzten Kontextes. Im Kollaborationsdiagramm agieren die verschiedenen Modellelemente eines Programms innerhalb des Programmablaufes miteinander. Dieses Diagramm wird verwendet, um diese Interaktionen für einen bestimmten begrenzten Kontext, unter besonderer Beachtung der Beziehungen einzelner Objekte und ihrer Topographie, darzustellen.

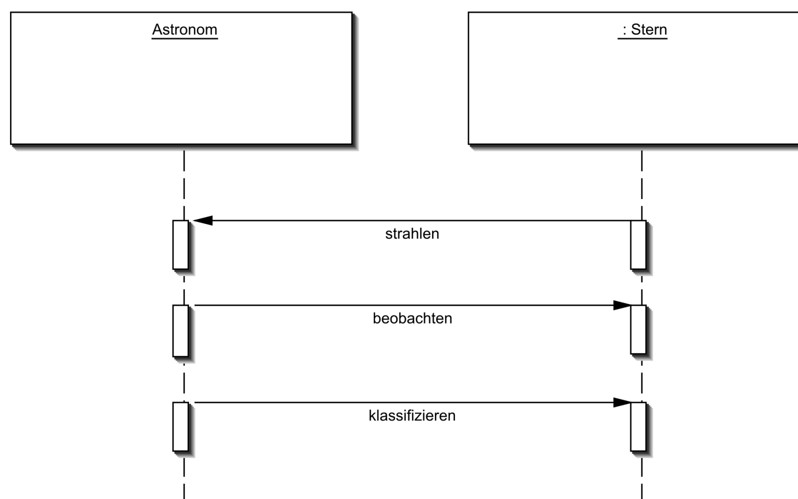

Abbildung 3.5: Beispiel eines UML Sequenzdiagramms

Zustandsdiagramme

Ein Objekt kann in seinem Leben verschiedenartige Zustände annehmen. Mit Hilfe des Zustandsdiagrammes visualisiert man diese, sowie Funktionen oder Ereignisse, die zu Zustandsänderungen des Objektes führen.

Das Beispieldiagramm zeigt - mit einem weiteren Bezug zum Inhalt dieser Diplomarbeit - die grobe Entwicklung eines Sterns mit einer Anfangsmasse zwischen 0.6 und 2.5 $M_\odot$. Man sieht wie der Stern verschiedene Zustände während seines Lebens durchläuft.

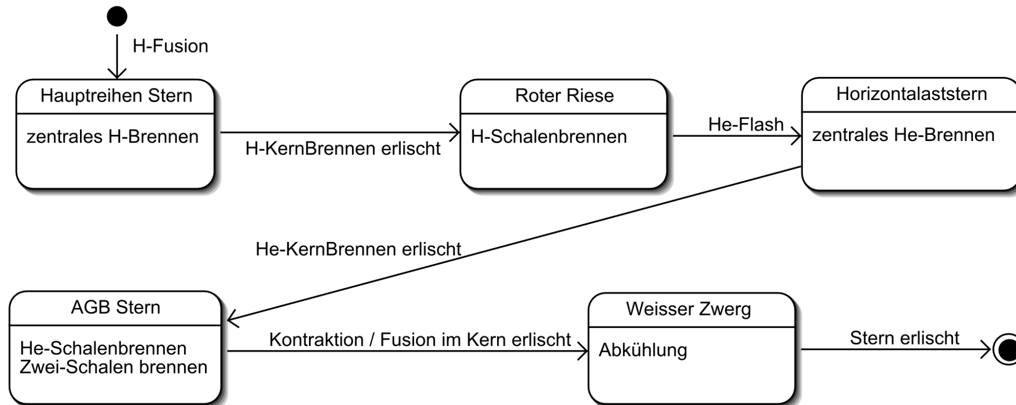

Abbildung 3.6: Beispiel eines UML Zustandsdiagramms

Aktivitätsdiagramme

In einem Aktivitätsdiagramm werden die Objekte eines Programms mittels der Aktivitäten beschrieben, die sie während des Programmablaufes vollführen. Eine Aktivität ist ein einzelner Schritt innerhalb eines Programmablaufes, d.h. ein spezieller Zustand eines Modellelementes, der eine interne Aktion sowie eine oder mehrere von ihm ausgehende Transitionen enthält. Gehen mehrere Transitionen von der Aktivität aus, so müssen diese mittels Bedingungen voneinander zu entscheiden sein. Somit gilt ein Aktivitätsdiagramm als Sonderform eines Zustandsdiagrammes, dessen Zustände der Modellelemente in der Mehrzahl als Aktivitäten definiert sind.

In Abbildung 3.7 wird als Beispiel ein Diagramm gezeigt, das die unterschiedliche Entwicklung von Sternen in Abhängigkeit der Existenz innerer Konvektion bei Anfangsmassen grösser oder kleiner als 1,15 $M_\odot$ zeigt. Die unterschiedliche Entwicklung eines Sterns mit H-Fusion im Kern zu einem Stern mit Schalenbrennen wird hier verdeutlicht.

3.2 Die Programmiersprache Java

Java ist eine objektorientierte Programmiersprache mit der Besonderheit, dass der Programmcode, der sogenannte Bytecode, vom Java-Compiler zunächst generiert wird, um schliesslich auf der virtuellen Java Maschine (JVM) ausgeführt zu werden. Dadurch ist diese Programmiersprache plattfomunabhängig, also nicht abhängig von dem Betriebssystem auf dem die virtuelle Maschine läuft. Der programmierte Code läuft auf allen Betriebssystemen gleich, sofern keine systemspezifischen Klassen verwendet werden. Java eignet sich also hervorragend zur Darstellung von dynamischen, interaktiven Online-Inhalten, deren Layout dem Nutzer unabhängig vom Betriebsystem des verwendeten Rechners, ja sogar des verwendeten Browsers, erscheinen soll.

Wie bei allen objektorientierten Programmiersprachen steht auch hier die möglichst intuitive Entwicklung von realen, der menschlichen Erfahrung angepassten Programmkomponenten im Mittelpunkt. Die verschiedenen Bestandteile des Programmiercodes, wie zum Beispiel Klassen,

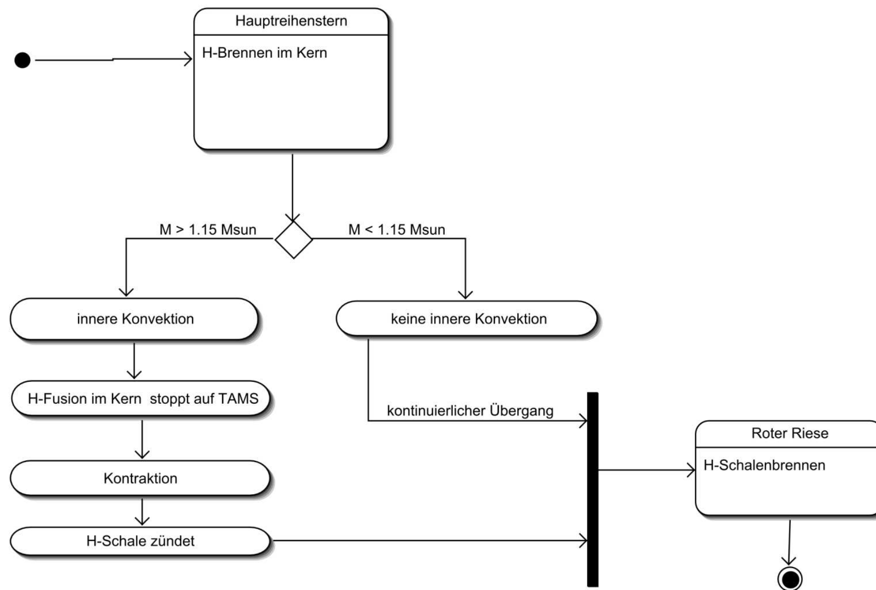

Abbildung 3.7: Beispiel eines UML Aktivitätsdiagramms

Methoden oder Schnittstellen, sollten in der Form gewählt werden, die der Erfahrung und Wahrnehmung der Menschen so weit wie möglich entspricht. Dies fördert das intuitive Verständnis des Programmcodes, vereinfacht die Verwaltung grosser Programmierprojekte und soll Programmierfehler einschränken.

Der modulare Aufbau von Java unterstreicht einen weiteren Vorteil, das Konzept des *Write Once, Run everyware*. Ist der Programmcode geschrieben und wurde einmal erfolgreich kompiliert, so kann das Programm auf jedem beliebigen Rechner mit installierter JVM ausgeführt werden. Objektklassen, die der Allgemeinheit zur Verfügung gestellt werden, können weiterverwendet und wiederverwertet werden. Natürlich ergeben sich dadurch vielfältige und umfangreiche Programmbibliotheken. Die Hauptbibliothek der Java-Standardklassen ist die *API specification for the Java 2 Platform Standard Edition* (siehe [34]). Der Zugang zu dieser Bibliothek während der Programmierung wird durch die Verwendung einer modernen, integrierten Entwicklungsumgebung (IDE) um ein vielfaches vereinfacht. Im Rahmen dieser Diplomarbeit wurde dazu die von IBM entwickelte *Eclipse*[2] Umgebung (vgl. [36]) verwendet, die selbst eine komplett in Java programmierte Softwareanwendung ist.

Aus der vielfältigen zu dieser Programmiersprache publizierten Literatur, ist an dieser Stelle das Buch von C. Ullenboom (siehe [37]) hervorzuheben, welches einen guten Überblick über sämtliche Bereiche der Java-Programmierung liefert.

3.3 Java-Applets im WWW

Auf Internetseiten, die meist in einer Auszeichnugssprache wie zum Beispiel HTML geschrieben sind, werden Java-Programme als Java-Applets eingebettet. In Applets wird das fertig programmierte Programm im Code der Internetseite durch eine entsprechende Markierung, einem sogenannten *tag*, aufgerufen. Vorraussetzung für die Ausführung eines Java-Applets ist natürlich eine installierte virtuelle Maschine. Durch die rasante Expansion des Internets in den letzten Jahren und dem zunehmenden Einsatz von Applets zur Darstellung der unterschiedlichsten Themen wurde natürlich auch die Programmiersprache Java populär.

[2]Laut Erich Gamma, einem der Entwickler, soll der Name darauf hinweisen, dass Eclipse proprietäre Entwicklungsumgebungen in den Schatten stelle. (siehe [35])

Aus dem unerschöpflichen Angebot des *World Wide Web* möchte ich auf ein paar Seiten verweisen, die einen Eindruck des didaktischen Potenzials vermitteln, der heutzutage durch die Verwendung von Applets zur Darstellung von Inhalten möglich ist.

Unter den akademischen Seiten sei das virtuelle Labor der *Universtiy of Oregon* (siehe [38]) hervorgehoben. Hier findet man ein umfangreiches Angebot an Applets zu Themen der Astronomie, der Thermodynamik, der Energie und der Mechanik.

Als Vertreter der Sparte eBooks sei auf die elektronische Version des Buches *Universe* ([39]) verwiesen. Nahezu alle astrophysikalischen Themen werden hier diskutiert und durch Flash-Animationen und Java-Applets unterstützt dargestellt (vgl. *Active Integrated Media Modules* unter [40]). Ein Beispiel aus der Schulphysik ist die Sammlung *Java-Applets zur Astronomie* des Lehrers W. Fendt (siehe [41]).

Schliesslich kommen neben populären Anwedungen der Amateurastronomie, wie zum Beispiel dem *Sky View Cafe* ([42]), vermehrt nützliche und professionelle Anwendungen, wie der *Aladin Sky Atlas* ([43]) des *Centre de Données astronomiques de Strasbourg*, zum Einsatz.

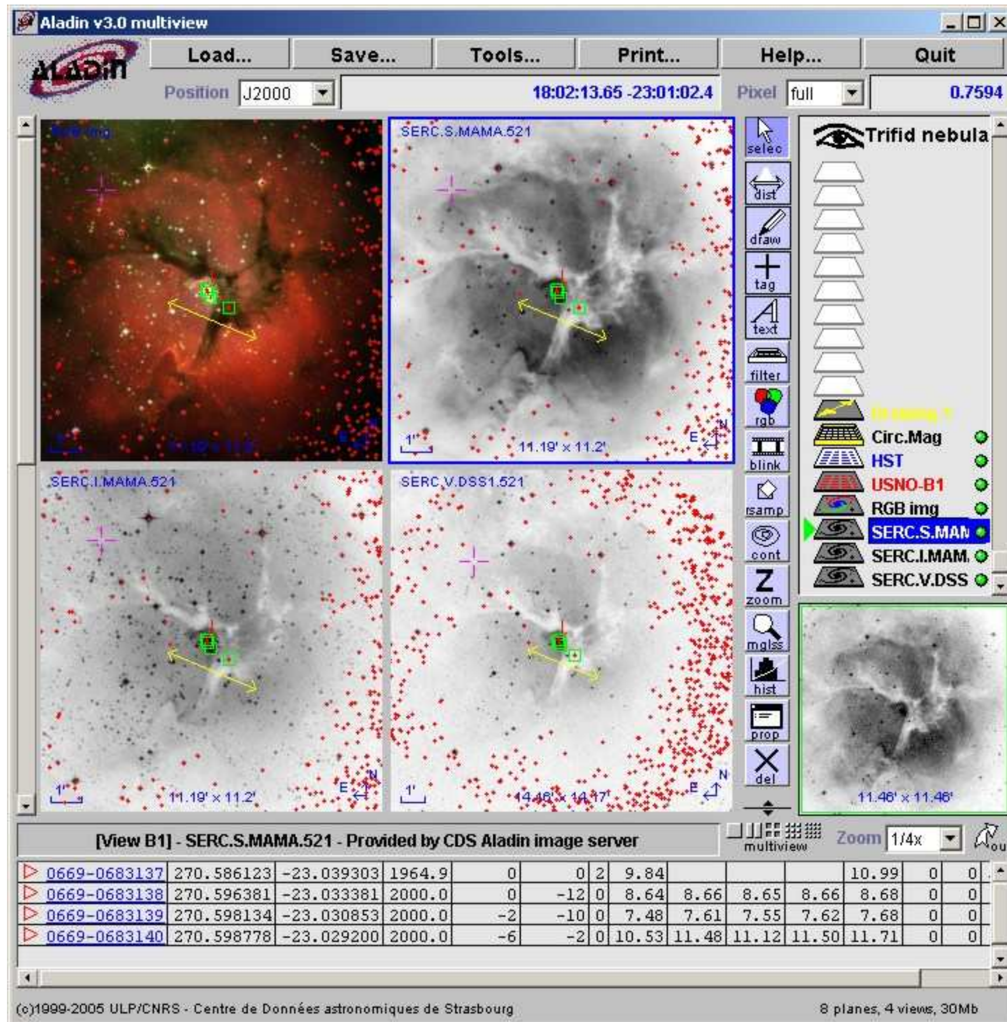

Abbildung 3.8: Beispiel einer komplexen Java Anwedung der Astronomie. Der Trifid Nebel in einer Multiview-Ansicht des Aladin Sky Atlas (©1999-2005 ULP/CNRS - Centre de Données astronomiques de Strasbourg)

3.4 Simulation und Darstellung von Sternentwicklung

Der Begriff *Simulation* kommt aus dem Lateinischen und bedeutet soviel wie Vorspiegelung, Nachahmung. Bei der Simulation eines Objekts oder Sachverhalts geht es also darum, unter bestimmten Voraussetzungen und Regeln dessen Darstellung mit ausgewählten Mitteln zu erreichen. Merkmale, Verhaltensweisen und Entwicklung von realen Objekten möglichst naturgetreu nachzubilden, ist das primäre Ziel jeder Simulation. Geht es um die Darstellung und Simulation von Objekten, liegt es daher nahe, auf objektorientierte Programmiersprachen zurückzugreifen. Soll das Ergebnis im Internet präsentiert werden, bietet Java ausgezeichnete Mittel, das Simulationsziel umzusetzen.

Wie nah die Darstellung an die Realität reicht, hängt natürlich immer vom Umfang der zur Simulation verwendeten Basisinformation und vom technischen Aufwand ab. Letztendlich also von dem Wissen, dass in die Simulation fliesst. Im Idealfall täuscht eine Simulation dem Menschen das reale Objekt vor. Dies würde in diesem Fall bedeuten, man würde neben der zeitlichen Entwicklung im Hertzsprung-Russel-Diagramm, eine 3D-Animation des sich entwickelnden Sterns, beispielsweise in einer Art 3D-Videosequenz, betrachten können. Dies wäre im Prinzip zu realisieren, würde aber wohl den Zeitrahmen dieser Arbeit sprengen.

Das Ziel dieser Diplomarbeit ist primär die Animation des HRD, welches in seiner statischen Printversion zwar einen zeitlichen Verlauf impliziert, durch die Vielzahl der Information aber schnell unübersichtlich werden kann. Die zeitliche Entwicklung der gegeneinander aufgetragenen Größen hängt im wesentlichen von den inneren Prozessen und Eigenschaften der Sterne ab. Versucht man nun diese zeitliche Entwicklung in der Animation zu simulieren, werden jedem Zeitpunkt die entsprechenden Eigenschaften, Prozesse und Ereignisse zugeordnet. Während der laufenden Simulation werden diese schliesslich wiedergeben, und ergeben in der Summe die Darstellung eben jener Eigenschaften, Prozesse und Ereignisse, welche die Sternentwicklung bestimmen.

Die javaHRD-Simulation ermöglicht schliesslich eine individuelle Betrachtung der stellaren Entwicklung in Abhängigkeit der Masse als Hauptentwicklungsparameter. In die Darstellung werden zusätzliche Eigenschaften und Zusammenhänge integriert. Dazu gehören die Simulation des Radius oder Anpassung der Sternfarbe an den aktuellen Wert der Oberflächentemperatur, um nur zwei Beispiele zu nennen.

Kapitel 4

javaHRD

Die Grundlage für das javaHRD-Programm bilden die zur Verfügung stehenden Daten von Sternmodellen, die entweder als Tabelle (z.B. in [24]) bereit stehen oder online im Digital Demo Room (vgl. Kapitel 2.3.3 und [30]) berechnet werden. In beiden Fällen stehen die für die Darstellung des Hertzsprung-Russell-Diagramms relevanten Grössen Leuchtkraft $logL$, Temperatur $logT$ und Alter a zur Verfügung.

Mit diesen Daten werden Java-Datenklassen gebildet und in Datenpaketen zusammengefasst, auf die das darstellende Java-Applet zugreifen kann.

Ziel der Darstellung ist eine interaktive und dynamische Version des HRD, in dem die Leuchtkraft $logL$ gegen die Temperatur $logT$ aufgetragen, und ein animierter Sternentwicklungsweg angezeigt wird. Um eine Animation zu realisieren, wird ein Timer eingesetzt, der beim i-ten Animationsschritt die komplette Zeichenebene der Anwendung neu zeichnet. Dadurch ergibt sich ein zeitliche Abfolge von Bildern, welche die zeitliche Darstellung der Entwicklung ermöglichen. Bei jedem dieser Schritte wird zunächst der Sternentwicklungsweg komplett gezeichnet. Anschliessend wird an der aktuellen Position der Entwicklung, also während dem entsprechenden Animationsschritt, ein ausgedehnter Sternpunkt innerhalb des HRDs dargestellt.

Damit die Darstellung der Wertepaare von Leuchtkraft und Temperatur auf dem Bildschirm möglich wird, ist die Transformation des Wertebereichs dieser Grössen auf sinnvolle Pixel-Werte einer Ausgabekomponente, z.B. eines Computerbildschirms, notwendig. Dies kann im Allgemeinen formal mit folgendem, simplen Zusammenhang erreicht werden:

$$x = q + p \cdot (logT)_i$$

$$y = w + v \cdot (logL)_i$$

Die Funktionen x und y sind die Bildschirmkoordinaten und p,q,v und w lineare Transformationsparameter, die empirisch ermittelt wurden (vgl. Kap. 4.4.4 und Anhang A.2).

Die für ein HRD relevanten Komponenten werden durch entsprechende Java-Klassen erzeugt und schliesslich im Rahmen der Animation gezeichnet. Die Organisation dieses Prozesses übernimmt das programmierte Java-Applet.

Die Umsetzung dieser Idee durch javaHRD wird in den folgenden Kapiteln beschrieben. Zunächst wird die Zusammenstellung der Daten diskutiert. Anschliessend wird die Darstellung eines Entwicklungsweges und die Animation des Sternpunktes besprochen. Schliesslich werden Aufbau und Funktion beschrieben.

4.1 Die Datengrundlage des Programms

4.1.1 Die Daten-Pakete von javaHRD

Aus den in Kapitel 2.3 beschrieben Sternmodellen, werden Java-Klassen gebildet, die in Paketen zusammengefasst zur Verfügung stehen. Die Bezeichnung der Pakete wird in Anlehnung an die

zugrunde liegenden Modelle gewählt. So beinhaltet das Paket *DDR* die im Digital Demo Room berechneten Daten (vgl. [30]), das Paket *padova* die Daten der Padova-Modelle (vgl. [24]) und das Paket *girardi2000* Daten von Girardi (vgl. [44]). Oft fehlen bei den genannten Datenreferenzen späte Entwicklungsphasen, so dass diese durch zusätzliche Daten ergänzt werden. Das Paket *padova* wurde so zum Beispiel durch Daten der Entwicklung von Weissen Zwergen aus [45] ergänzt.

Zu Beginn der Entwicklung von javaHRD werden die Daten des DDR-Paketes mit Hilfe der Webanwendung des Digital Demo Room online berechnet. Die Berechnung erfolgt nach den analytischen Entwicklungsformeln von Hurley et al. (siehe [29]) und wird zunächst für eine Metallizität von Z=0.02 durchgeführt. Diese Datengrundlage wird gewählt, da sie schnell zur Verfügung steht und komplette Entwicklungswege von der Hauptreihe bis zum Weissen Zwerg liefert. Auch wenn die Daten im Bereich der Abkühlphasen ungenau sind und aus didaktischen Gründen korrigiert werden müssen, kann mit dessen Hilfe leicht eine erste Zusammenstellung als Grundlage der Programmierarbeit geschaffen werden (siehe Abb. 4.1 und 4.2).

Die Korrektur geschieht durch lineare Regression und anschliessendes Verschieben der Ausgleichsgeraden, so dass ein stetiger Entwicklungsweg erhalten bleibt. Dieser Weg wird zum einen gewählt, um die korrekte Entwicklung während den finalen Abkühlungsphasen darzustellen (vgl. Kap. 5.2), zum anderen um die Darstellung der unterschiedlichen Zeitskalen zu erhalten. Letzteres ist ein wesentlicher Vorteil der zeitlichen Darstellung unter didaktischen Gesichtspunkten. Diese Darstellung wird nur dann erreicht, wenn zeitlich äquidistante Datenwerte verwendet werden. Dies ist bei den DDR-Entwicklungswegen, mit Ausnahme der Phasen zwischen dem letzten Datenpunkt des AGB und dem ersten Datenpunkt der Abkühlphase der Fall. Um auch diese Entwicklungswege vollständig darzustellen, wird der Bereich zwischen diesen Punkten mit einer höheren zeitlichen Auflösung im DDR berechnet.

Bei allen anderen Paketen wird auf die Datenbanken zurückgegriffen die online auf den Seiten der entsprechenden Sternentwicklungsmodelle zur Verfügung stehen (vgl. Kap. 2.3.2). Nachdem die Datentabellen heruntergeladen sind, werden diese in die für das javaHRD-Programm notwendige Struktur gebracht (siehe 4.1.2). Um einen Vergleich zum DDR-Paket möglich zu machen, werden Daten mit möglichst gleicher Metallizität ausgewählt. Für Das *padova*-Datenpaket werden Daten mit Z=0,02 gewählt, für das *girardi2000*-Paket Daten mit Z=0,19.

Der wesentliche Vorteil dieser Datenpakete liegt in der Qualität der präsentierten Entwicklungswege, da diese direkt von den Forschungsgruppen stammen, welche die Sternmodelle erstellt haben. Allerdings gibt es auch zwei Nachteile: Zunächst werden keine vollständigen Entwicklungswege zur Verfügung gestellt sondern nur solche von der ZAMS bis hin zum AGB. Diese Entwicklungswege müssen also durch zusätzliche Daten ergänzt werden (vgl. Kap. 2.3.3). Weiterhin liegen die Datenwerte nicht in zeitlich äquidistantem Abstand vor, so dass Schwierigkeiten bei der Darstellung der unterschiedlichen Zeitskalen auftauchen. Dies könnte mit einer Zeitfunktion in der Animation berücksichtigt werden, konnte aber im Rahmen dieser Arbeit aus Zeitgründen nicht umgesetzt werden.

Zur Ergänzung der *padova*-Daten werden die Entwicklungswege für Weisse Zwerge von Althaus und Benvenuto verwendent (siehe [28]). Allerdings stehen hier keine Entwicklungswege mit Metallizitäten Z$\sim$0.02 zur Verfügung, so dass Daten für Z=0.001 ausgewählt werden. Durch manuell gesetzte Pixel werden diese Entwicklungswege mit den vorangegangen *padova*-Entwicklungswegen verbunden (siehe Abb. 2.3).

Es können jederzeit neue Datenpakete erstellt und eingebaut werden. Die neuen Daten müssen allerdings der Struktur der Datenpakete angepasst werden. Dies ist mit entsprechendem Zeitaufwand verbunden, da die Datenwerte auf die verschiedenen Datenklassen manuell verteilt werden müssen. Ein maschinelles Auslesen der Datentabellen und Erzeugen der Datenklassen wäre von Vorteil und wurde zum Teil auch schon vorbereitet. Aus Zeitgründen kann diese Automation im Rahmen dieser Arbeit aber nicht vollständig umgesetzt werden. Dies hängt auch mit der Inhomogenität und Komplexität der von den Sternentwicklungsmodellen bereit gestellten Datentabellen zusammen. Für jede Datenquelle müsste daher eine eigenes Verfahren entwickelt werden, um die Daten automatisch auslesen zu können.

Zum Zeitpunkt der Abgabe der Diplomarbeit sind die im javaHRD-Programm (Version v0.66)

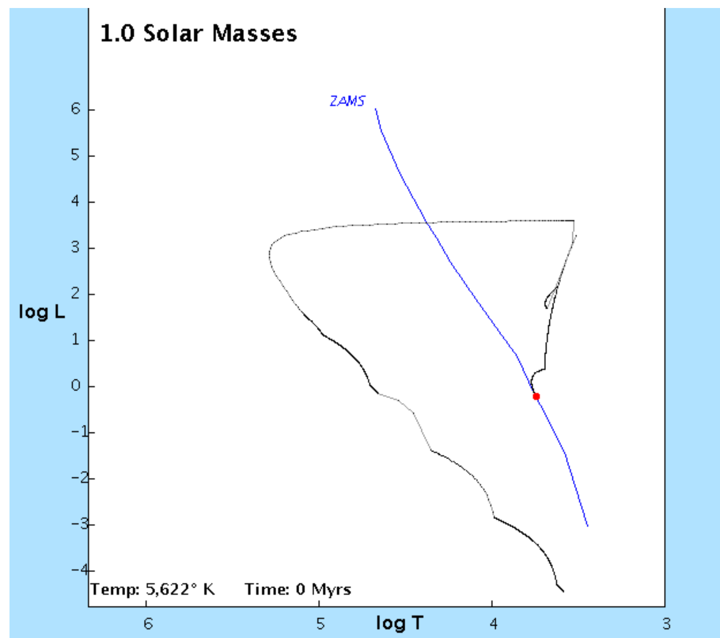

Abbildung 4.1: 1 $M_{\odot}$-Entwicklungsweg aus dem DDR-Paket (vgl. javaHRD v0.36)

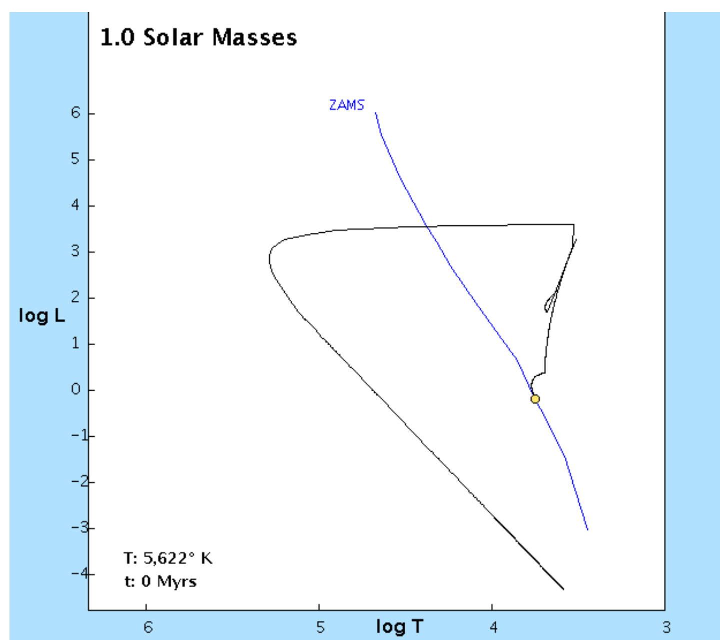

Abbildung 4.2: In der Abkühlphase korrigierter 1 $M_{\odot}$-Entwicklungsweg aus dem DDR-Paket (vgl. javaHRD v0.50)

präsentierten Entwicklungswege nicht homogen. Die Entwicklungswege für 0,1 $M_\odot$, 1,5 $M_\odot$, 2 $M_\odot$, 3 $M_\odot$, 5 $M_\odot$, 7 $M_\odot$, 9 $M_\odot$, 20 $M_\odot$, 50 $M_\odot$ und 90 $M_\odot$ greifen auf das DDR-Paket zu. Den Entwicklungswegen von 0,6 $M_\odot$, 0,9 $M_\odot$, 15 $M_\odot$, und 120 $M_\odot$ liegen die Daten des padova-Pakets zugrunde, wobei für 0,6 $M_\odot$ und 0,9 $M_\odot$ zusätzliche Entwicklungswege für Weisse Zwerge wie beschrieben hinzugefügt wurden.

4.1.2 Die Struktur der Datenpakete

Die einzelnen Datenpakete haben eine ähnliche Struktur und beinhalten Klassen, die Daten für Leuchtkraft, Oberflächentemperatur und Alter speichern. Bei der Benennung der Klassen wird in dieser Arbeit folgendes festgelegt: Alle Dateinamen enden mit einer vierstelligen Zahl μ, welche die Masse impliziert. Die ersten drei Ziffern entsprechen der Anfangsmasse in $M_\odot$ des zugrunde liegenden Sternentwicklungsweges. Die vierte Ziffer steht für die erste Dezimalstelle. Freie Stellen werden mit 0 vervollständigt. So gehören zum Beispiel die Daten-Klassen `LogL0015`, `LogT0015`, und `Years0015` zum Entwicklungsweg eines Sterns mit der Anfangsmasse 1,5 $M_\odot$. Der Grund für die Wahl von drei Datenklassen anstelle einer, liegt in der Geschichte der Programmierung von javaHRD. Zu Beginn wurden primär die im DDR (siehe [30]) berechneten Daten verwendet. Um alle stellaren Entwicklungsphasen im Detail aufzulösen, wurden schliesslich die einzelnen Phasen getrennt berechnet. Dadurch ergab sich eine Menge an Daten, welche die begrenzte Grösse einer Java-Klasse überschritten. Dies wurde dadurch gelöst dass die vorhandenen Werte-Paare auf drei Datenklassen verteilt wurden.

Die Datenklassen der Leuchtkraft (`LogL`μ`.class`) und der Temperatur (`LogT`μ`.class`) speichern die Entwicklungs-Daten jeweils in einem eindimensionalen Array mit dem Namen `value`. Die Datenklassen des Alters (`Years`μ`.class`) speichern zusätzlich zum `value`-Array Information zum Beginn und Ende der einzelnen Sternentwicklungsphasen. Dies erfolgt in den Arrays `stage` und `stageName`.

Beispiel für eine Datenklasse: LogL0010

```
public class LogL0010 {
    public static double value[] = {
            -0.1561,
            -0.1561,
            .
            -4.426,
            -4.4271,
    };
}
```

4.1.3 Festlegung der Daten-Referenzen mit den Klassen `LogL`, `LogT` und `Years`

Wählt der Benutzer eine neue oder andere Masse aus, so werden alle bisherigen Referenzen von `LogL`, `LogT` und `Years` neu zugewiesen (siehe Abb. 4.3).

Der Zugriff über Referenzen auf die Datenklassen der jeweiligen Pakete (vgl. Kap. 4.1.2) erfolgt durch das im folgenden beschriebene Verfahren.

Zunächst wird ein nicht initialisiertes Datenarray vom Datentyp *Double* deklariert. Eine Methode prüft dann über die Prüfvariable μ den durch die Klasse `Star` übergebenen Massewert. Damit Massenwerte bis auf 0,1 $M_\odot$ unterschieden werden können, wird der Massewert M mit 10 multipliziert um eine ganzzahlige Prüfvariable zu erhalten. Dies ist notwendig, da switch-Anweisungen Prüfvariablen des Typs *Integer* voraussetzen. Je nach eingestelltem Massewert, setzt die switch-Anweisung nun eine neue Referenz auf das gleichnamige Array der entsprechende Datenklasse. Das neu referenzierte Datenarray ist gleichzeitig der Rückgabewert der zugehörigen

Methode, auf den externe Klassen zugreifen können. So kann man zum Beispiel auf den n-ten Leuchtkraftwert eines Sternentwicklungsweges von M $M_\odot$ über die Referenz `LogL.value(M)[n]` zugreifen. Als Standard-Wert für alle Switch-Anweisungen in allen Methoden wird für μ der Wert 10, also M = 1,0 $M_\odot$ gesetzt. Dadurch wird der $M_\odot$-Startwert von javaHRD festgelegt.

Die Klassen `LogL`, `LogT` und `Years` haben also eine ähnliche Struktur, wobei in `Years` zusätzliche Variablen und Methoden zur Verfügung gestellt werden.

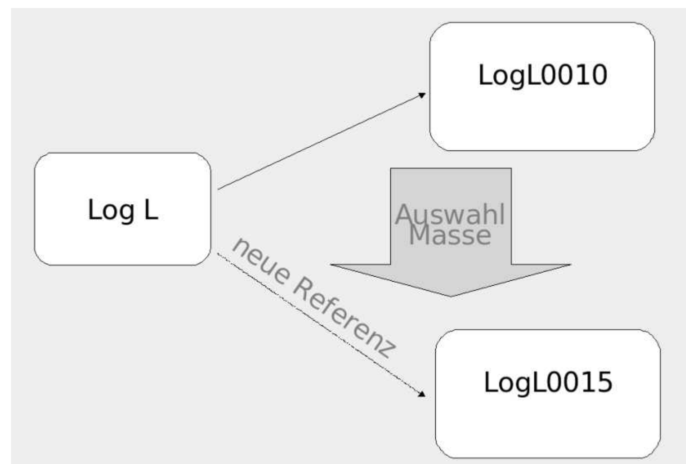

Abbildung 4.3: Umreferenzierung der Daten

Gemeinsamkeiten der Klassen `LogL`, `LogT` und `Years`

In jeder dieser Klassen wird zunächst das Array `value[]` deklariert. Anschliessend wird die Methode `value(M)` mit dem Parameter M definiert. Der Parameter wird über die Prüfvariable an die switch-Anweisung übergeben. Je nach Wert der Prüfvariable wird nun die Referenz des Arrays `value` neu gesetzt. Der Rückgabewert der Methode ist dann genau dieses neue Array. Als Beispiel sei hier die Klasse `LogL` aufgezeigt:

```
public class LogL {
   public static double value[];
   public static double[] value(double M) {
      double mu = 10 * M;
         switch ( (int)mu ) {
         case 1:
            value = DDR.LogL0001.value;
            break;
         case 6:
            value = padova.LogL0006.value;
            break;
              .
              .
         case 1200:
            value = padova.LogL1200.value;
            break;

         default:
            value = DDR.LogL0010.value;
         }
   return value;
   }
}
```

Die speziellen Methoden der Klasse `Years`

Neben der `value(M)`-Methode werden in dieser Klasse die Methoden `stage(M)` und `stageName(M)` nach dem gleichen Verfahren eingesetzt um zusätzliche Information für das javaHRD-Programm zu erhalten. Danach werden die Arrays `value`, `stage` und `stageName` deklariert. Sowohl Variablen als auch Methoden werden durch entsprechende Referenzierung initialisiert.

Zum Schluss wird noch die Methode `length()` verwendet, welche die entsprechende Länge der verwendeten Arrays ermittelt. Durch diese Methode können unterschiedlich grosse Datenmengen verwendet werden.

4.2 Darstellung der Entwicklungswege und Animation

Die Darstellung und Animation eines Entwicklungsweges von javaHRD erfordert drei wesentliche Schritte:

Erstens, die Umsetzung aller Komponenten, die zur Darstellung des Diagrammes ohne Entwicklungsweg nötig sind. Dies ist ein semistatischer Schritt, da sowohl statische als auch dynamische Elemente angezeigt werden. Zu den statischen Elementen gehörend Diagrammtitel, -achsen oder Achsenabschnitte. Zu den dynamischen die ausgewählte Masse, die aktuellen Datenwerte oder die Anzeige der momentan durchlaufenen Entwicklungsphase. Zur Umsetzung all dieser Anzeigen benötigt man ein Objekt, das von der Klasse `StarInHRD` repräsentiert wird.

Zweitens, die Darstellung eines Sternenwicklungsweges, der bei jedem Animationsschritt gezeichnet wird. Dies ist sozusagen die konventionelle Darstellung eines Entwicklungsweges im HRD, die hier bei jedem Animationsschritt zum Einsatz kommt. Die zugehörige Java-Klasse heisst `Track`.

Schliesslich die zeitliche Animation eines Datenpunkts, der den Stern repräsentiert und im Laufe der Animations-Zeit seinen Entwicklungsweg nachfährt. Ein solches Objekt wird durch die Java-Klasse `Star` repräsentiert.

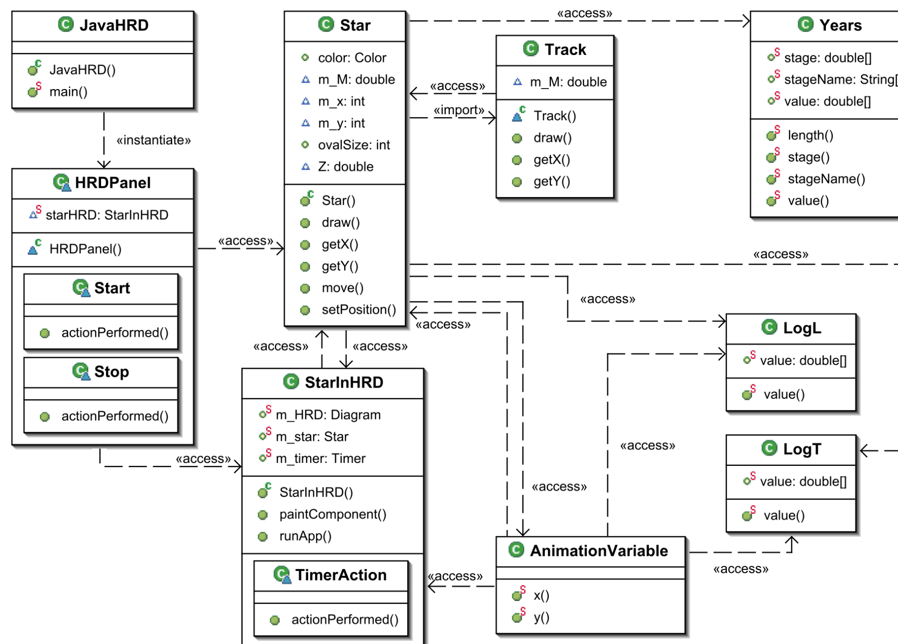

Abbildung 4.4: Die wichtigsten javaHRD-Klassen und deren Zusammenhang im UML-Überblick. Dieses UML-Diagramm verwendet die in *Eclipse* üblichen Symbole zur Markierung der Eigenschaften von Variablen, Methoden und Klassen.

Die zu Beginn dieses Kapitels erläuterte Darstellungstransformation wird durch die Klasse `AnimationVariable` umgesetzt. Die Grundlage für die zeitliche Animation wird durch die innere Klasse[1] `TimerAction` gebildet (vgl. Kap. 4.5.1 und 4.5.2)

All diese Objekte müssen dargestellt, verwaltet, gesteuert und aufgerufen werden. Dies wird im wesentlichen von der Klasse `HRDPanel` geleistet. Den Aufruf übernimmt die Klasse `JavaHRD`.

Eine Übersicht dieser wichtigsten javaHRD-Klassen gibt das UML Klassendiagramm in Abbildung 4.4. Hier werden für jede Klasse nur die wichtigsten Variablen, Methoden und inneren Klassen aufgelistet. Die Klassen `Star`, die einen Sternpunkt zeichnet, und die Klasse `Animationsvariable` mit den Transformationsmethoden x und y, greifen auf die Datenklassen zu. Die Klasse `Track`, die für das Zeichnen eines kompletten Entwicklungweges zuständig ist, wird innerhalb von `Star` aufgerufen. Das HRD wird schliesslich von der Klasse `StarInHRD` gezeichnet, die auch die Timer-Methode enthält. Schliesslich verwaltet die Klasse `HRDPanel` das gesamte Applet das von der Klasse `JavaHRD` aufgerufen wird. Eine genaue Beschreibung der Klassen und deren Funktion erfolgt in Kapitel 4.4. Bevor die Animation gestartet werden kann, müssen Exemplare erzeugt werden, welche die Grundlage der Darstellung bilden. Einen Eindruck dazu gibt das UML Objektdiagram in Abbildung 4.5. Hier sind die wichtigsten javaHRD-Objekte und deren Exemplare die beim ersten Aufruf von javaHRD zu sehen sind. Eine genaue Beschreibung liefert Kapitel 4.4.

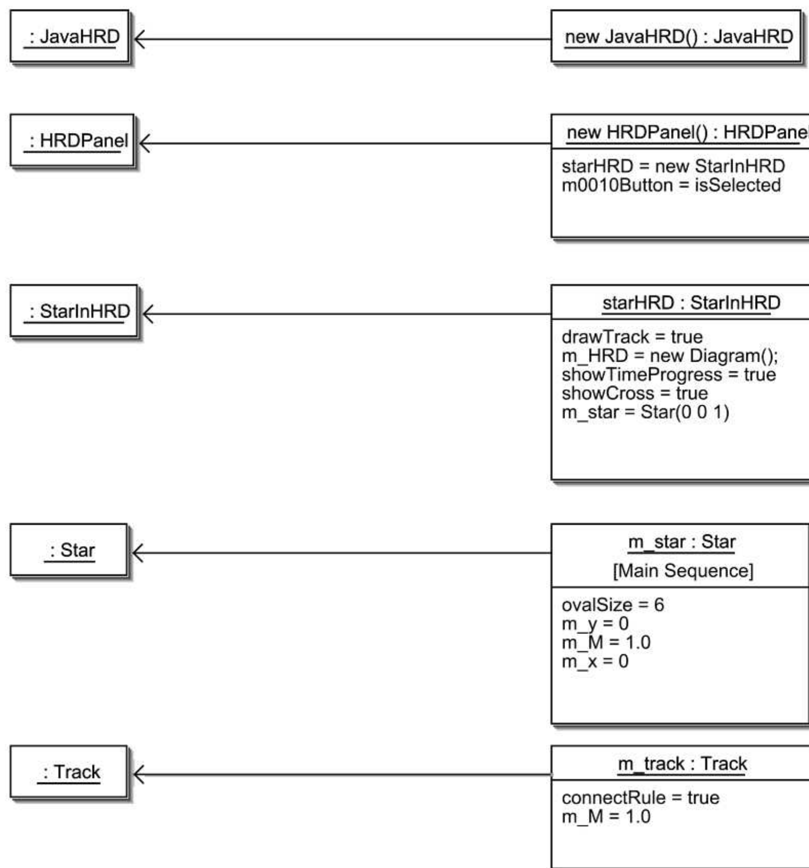

Abbildung 4.5: Objektdiagramm der wichtigsten javaHRD Objekte. Zu sehen sind Objekte mit entsprechenden Exemplaren der Klassen `JavaHRD`, `HRDPanel`, `StarInHRD`, `Star` und `Track`. Dies sind die Exemplare die beim ersten Aufruf des Applets mit dem Parameter 1.0 $M_\odot$ entstehen.

[1] Innere Klassen sind solche, die innerhalb von Klassen programmiert werden.

Klasse	Funktion
`JavaHRD`	Aufruf von `HRDPanel`
`HRDPanel`	Organisation der Steuerungsoberfläche
`StarInHRD`	Verwaltung von Zeichnung, Animation und Exemplaren
`sVar`	Interface für globale Variablen
`AnimationVariable`	Transformation der Datenwerte auf Pixelwerte
`Star`	Zeichnen des Sternpunktes / Import von `Track`
`Track`	Zeichnen des Sternentwicklungsweges
`Diagram`	Zeichnen des Diagrammgerüstes
`DiagrammBackground`	Zeichnen des Diagrammhintergrundes
`ZAMS / TAMS`	Zeichnen der ZAMS- bzw. TAMS-Sequenz
`Hayashi`	Zeichnen der Hayashi-Sequenz
`HburnCoreArea`	Zeichnen der Bereiche mit stabilem H-Brennen im Kern
`HburnShellArea`	Zeichnen der Bereiche mit stabilem H-Schalen-Brennen
`HeBurnCoreARea`	Zeichnen der Bereiche mit stabilem He-Brennen im Kern
`HorizontalBranch`	Zeichnen des Horizontalastes
`Cross`	Zeichnen des Datenwert-Fadenkreuzes und der Werte
`TimeProgress`	Darstellung des Zeitentwicklungsbalkens
`StageLabel`	Anzeige der morphologischen Bezeichnung neben dem Sternpunkt
`HighLightStage`	Hervorhebung der ausgewählten Entwicklungsphase
`Values`	Anzeige von genauen Daten-Werten
`FreezeTrack`	Einfrieren des aktuellen Entwicklungsweges
`StarColor`	Definition der Farbe des Sternpunktes
`Radius`	Radius Simulation
`InfoBox`	Anzeige einer allg. Zusammenfassung der gewählten Phase

Tabelle 4.1: Übersicht aller javaHRD-Klassen

4.3 Steuerung und Funktion

javaHRD besteht aus einer Vielzahl von Java-Klassen. Zum Zeichnen eines Sternentwicklungsweges sind im wesentlichen die Klassen `StarInHRD`, `Star` und `Track` notwendig. Der Aufruf von javaHRD erfolgt durch `HRDPanel`. Die Steuerung von javaHRD übernimmt die Klasse `HRDPanel`. Alle Benutzereingaben und -anfragen laufen über diese Klasse, die man als Steuerklasse von javaHRD bezeichnen kann. Darüber hinaus gibt es eine ganze Reihe von Klassen, die zusätzliche Funktionen übernehmen. So ist beispielsweise die Klasse `ZAMS` für das Zeichnen der ZAMS-Sequenz zuständig.

In der Benutzung von javaHRD ergibt sich aus der Anzahl der Funktionen und Features vielfältige Anwendungsfälle. Die wichtigsten sind *Masse auswählen*, *Entwicklungsphase auswählen*, *Auswahl bestimmter Features* und *Animation steuern*. Diese sind in UML Anwendungsfalldiagramm in Abbildung 4.6 zusammengefasst. Eine Übersicht aller Klassen und deren Funktion findet man in Tabelle 4.1.

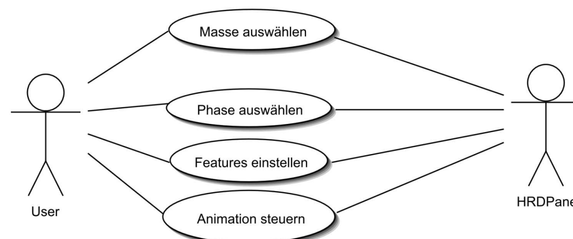

Abbildung 4.6: Die Zentralen Anwendungsfälle des javaHRD. Auswahl der Anfangsmasse des Sterns und der stellaren Entwicklungsphase, Einstellung bestimmter Eingeschaften der Simulation und Steuerung der Animation des Applets.

4.4 Java-Klassen

4.4.1 Die Klasse `JavaHRD`

Die `JavaHRD`-Hauptklasse organisiert lediglich den Aufruf von javaHRD. Je nachdem ob der Aufruf online über das Internet oder lokal auf einem Rechner geschieht, wird die zugehörige jar-Datei als Applet direkt aus einer HTML-Seite oder als Java-Applikation aufgerufen. Der Aufruf des Applets erfolgt über den Konstruktor `JavaHRD()`, der ein Objekt der Klasse `HRDPanel` erzeugt und sogleich der Applet-Ebene[2] hinzufügt.

Darüber hinaus wird in einer Standard `main()`-Methode ein `JFrame`, das Java-Anwendungshauptfenster erzeugt, in dessen Content-Ebene[3] ebenfalls ein Objekt der Klasse `HRDPanel` gesetzt wird. Das `JFrame` wird in üblicher Weise zentral auf dem Bildschirm ausgegeben (siehe Anhang A.1).

Durch diese Vorgehensweise bleibt der Einsatz von javaHRD nicht auf die Veröffentlichung im Rahmen einer Internetpräsenz beschränkt, sondern kann auch lokal als eigenständige Applikation auf Rechnern angezeigt werden, die nicht an das Internet angebunden sind. In diesem Fall ist keine installierte Browsersoftware nötig. Das Programm kann direkt als lokale Anwendung gestartet werden.

4.4.2 Die Klasse `HRDPanel`

Diese Klasse organisiert sämtliche Inhalte von javaHRD, also alle Steuerelemente, die komplette Diagrammdarstellung und sämtliche Bedieneroberflächen.

Die Darstellungsebene von javaHRDs wird in zwei Hauptbereiche eingeteilt. Der Bereich aller Eingaben und Steuerungen wird im folgenden als *Steuerungsebene*, der Bereich aller Ausgaben, Anzeigen oder gezeichneten Elemente, als *Diagrammebene* bezeichnet. (siehe Abb. 4.7).

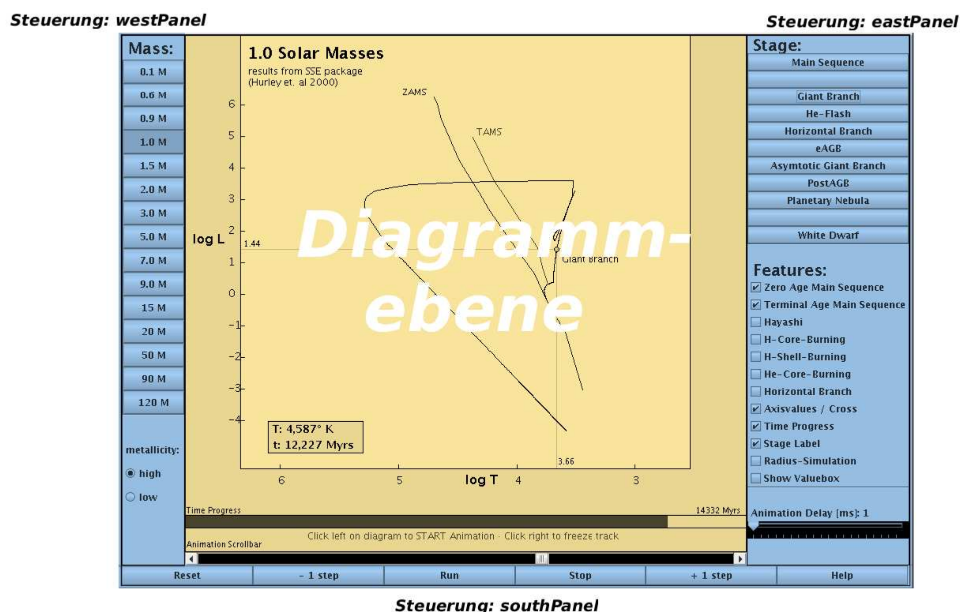

Abbildung 4.7: Steuerungs- und Diagrammebene von javaHRD. Im zentralen Bereich des Diagramms ist die Diagrammebene (gelb) zu sehen, die im Westen, Osten und Süden von der Steuerungsebene umgeben werden (blau).

---

[2]Als Applet-Ebene sei die Hauptebene des Applets, also die Gesamtfläche die das Applet auf einem Bildschirm ausfüllt, bezeichnet.

[3]Standard-Content-Pane eines JFrame Objekts

Die Steuerungsebene lässt sich in drei Gruppen untergliedern. Die linke Steuerspalte mit den Massenauswahl-Schaltflächen, die rechte Steuerspalte, in der die Phasenauswahl-Schaltflächen und die Funktionsauswahlboxen untergebracht sind. Weiterhin die untere Steuerzeile, die verschiedene Steuer-Schaltflächen zur Verfügung stellt. Eine Ausnahme bildet eine Bildlaufleiste zur manuellen Steuerung der Animation, die aus layouttechnischen Gründen innerhalb der Diagrammebene untergebracht wurde.

Die Diagrammebene zeigt das eigentliche, animierte HRD. Hier werden alle Merkmale des Diagramms wie Diagrammachsen, Achsenabschnitte, Textinformation und vor allem die Entwicklungswege der Sterne gezeichnet. Letzteres geschieht durch Aufruf einer bestimmten inneren Massenklasse (siehe Kap. 4.5.3). Diese ruft dann wiederum die entsprechende Massenmethode der Klasse `Star` innerhalb der aktuellen Instanz der Klasse `StarInHRD` auf. Als Standardeinstellung wurde 1 $M_\odot$ gewählt, so dass der Benutzer beim erstmaligen Aufruf des Programms den Entwicklungsweg unserer Sonne angezeigt bekommt.

Im folgenden soll nun die Klasse `HRDPanel` besprochen werden, wobei eine detaillierte Beschreibung der inneren Klassen erst später erfolgt (siehe Kap. 4.5.3).

Der Konstruktor dieser Klasse lautet `HRDPanel()` und ist, wie Eingangs erläutert, für die Darstellung von javaHRD zuständig. Hier werden die sichtbaren Elemente der Bedienoberfläche und deren Methoden gesetzt, initialisiert und bei Bedarf gesteuert. Zunächst wird aber eine Variable der Klasse `StarInHRD` deklariert, so dass später ein Objekt dieser zentralen Klasse erzeugt werden kann. Bevor der Klassenkonstruktor aufgerufen wird, werden die Phasenauswahl-Schaltflächen und die Massenauswahl-Schaltflächen erzeugt. Weiterhin wird die Animations-Bildlaufleiste (`animationScroll`) und der Animations-Schieberegler der Zeit `delaySlide` samt Bezeichner `slideT` gesetzt. Diese Elemente werden global gesetzt, da sie dynamische Eigenschaften haben. Sie müssen von externen Klassen angesprochen werden und auch direkte oder indirekte Steuerbefehle von diesen empfangen können.

Sobald der Konstruktor aufgerufen wird, passiert folgendes: Ein Objekt der Klasse `StarInHRD` wird initialisiert und erhält den Namen `starHRD`. Diesem Objekt werden ein `MouseWheelListener` und ein `MouseListener` zugewiesen. Die zugewiesenen Listener[4] rufen die inneren Klassen `WheelX` und `Start` auf. Danach werden Eigenschaften der Bildlaufleiste und des Animations-Schiebereglers gesetzt, die später im Detail beschrieben werden (siehe Kap. 4.5.3).

Schliesslich werden alle weiteren Elemente deklariert und erzeugt. Das Prinzip der meisten Elemente ist schlicht. Ein Objekt der benötigten Klasse wird generiert, mit Eigenschaften versehen und schliesslich wird ihm ein Listener zugewiesen, der eine bestimmte innere Klasse aufruft. Das sei am Beispiel einer Massenauswahl-Schaltfläche demonstriert, indem ein `ActionListener` eine innere Klasse aufruft:

```
JButton startButton = new JButton("Start");
startButton.addActionListener(new Start());
```

Im Bespiel wird ein `JButton`-Objekt *startbutton* erzeugt, das den Stringwert 'Start' erhält. Diesem Objekt wird abschliessend über die Methode `addActionListener()` der Aufruf der inneren Klasse `Start` zugewiesen (vgl. Kap 4.5.3).

Nachdem nun alle benötigten Elemente erzeugt wurden, müssen diese in Objekten der Klasse `JPanel` organisiert werden. Für das spätere Layout werden vier solcher Objekte benötigt: `westPanel`, `eastPanel`, `controlPanel` und `diagramPanel` (siehe Abb. 4.7). Die gewählte Bezeichnung wird im folgenden Kapitel deutlich.

4.4.3 Das javaHRD-Layout

Die Klasse `HRDPanel` steuert das gesamte Layout von javaHRD (siehe Abb. 4.7 und 4.8). Im Verlauf der Programmierarbeit wurde ein einfaches Standard-Java-Layout, das BorderLayout

[4] Die Standard-Java-Klasse der *Listener* stellt eine Reihe von Klassen zu Verfügung, die auf Ereignisse hören. So hört zum Beispiel ein Mouselistener auf die elementaren Ereignisse, die eine PC-Maus auslösen kann.

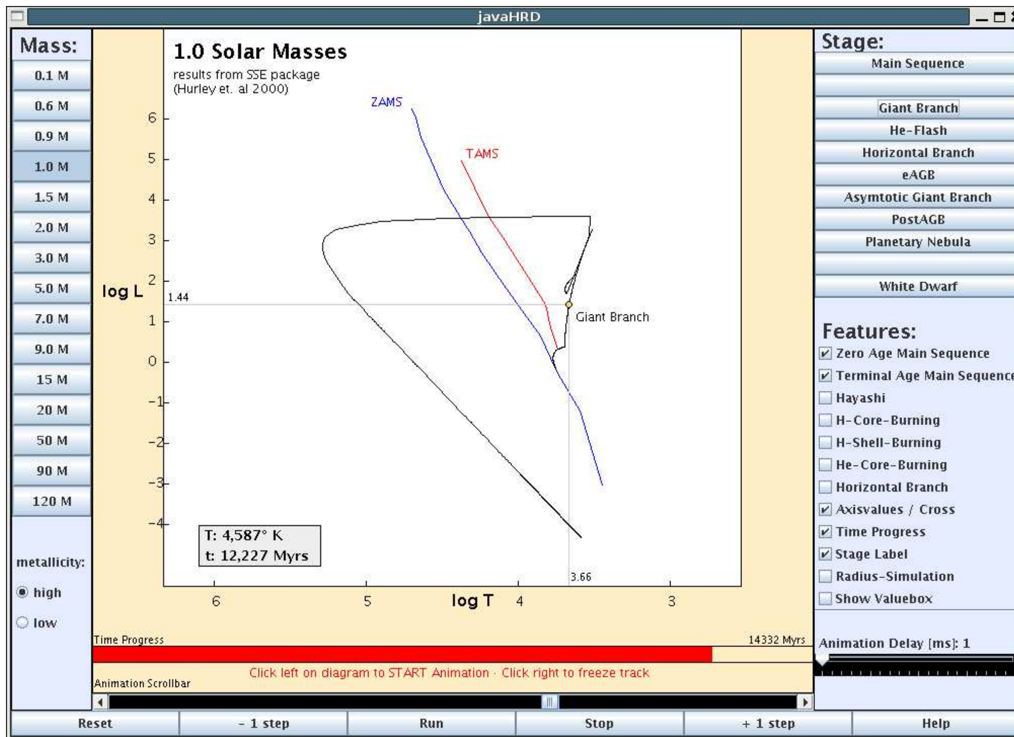

Abbildung 4.8: Layout von javaHRD

gewählt. Im zentralen Bereich dieses Layouts wird die Diagrammebene gezeichnet. Der Nord-Bereich wurde übersichtshalber frei gelassen, im West-Bereich liegt das `westPanel` mit den Massenauswahl-Schaltflächen, im Ost-Bereich das `eastPanel` mit den Phasen-Schaltflächen, den Funktions-Schaltflächen und dem Animations-Schieberegler und schliesslich im Süd-Bereich das `controlPanel` mit den Steuerschaltflächen. Alle äußeren Bereiche stellen somit die verschiedenen Gruppen der *Steuerungsebene* dar und sind ihrerseits wieder im sogenannten GridLayout gehalten. Durch diese Wahl des Layouts ergibt sich eine übersichtliche und gleichzeitig einfach zu verwaltende Darstellung von javaHRD.

4.4.4 Das Interface `sVar`

Diese Klasse definiert als Interface die globalen Abmessungen von javaHRD. Dabei werden die Werte für die Transformationsvariablen $q = xPos$, $w = yPos$, $q = xSt$ und $p = ySt$ gesetzt, welche empirisch ermittelt wurden (siehe Anhang A.2). Die Position des eigentlichen HRDs innerhalb der Diagrammebene wird über die Variablen $xAchsenPosY$ und $yAchsenPosX$, die Y-Position der x-Achse und die X-Position der y-Achse gesetzt (siehe Abb. 4.9). Ausserdem werden die verschieden Dimensionen und die Hintergrundfarben des Applets bestimmt, so dass diese immer global zur Verfügung stehen und der Zugriff aus anderen Java-Klassen in einfacher Weise möglich ist.

4.4.5 Die Klasse `StarInHRD`

Die Klasse `StarInHRD` zeichnet alle Komponenten der Diagrammebene, die nicht zum eigentlichen Sternentwicklungsweg gehören. Die wichtigsten Bestandteile dieser Klasse sind der `StarInHRD()` Konstruktor, die innere Klasse `TimerAction` und die Java-Standard-Methode `paintComponent`. Darüber hinaus werden ein ganze Reihe von öffentliche Methoden definiert, welche im Kapitel 4.5.4 im Detail erklärt werden.

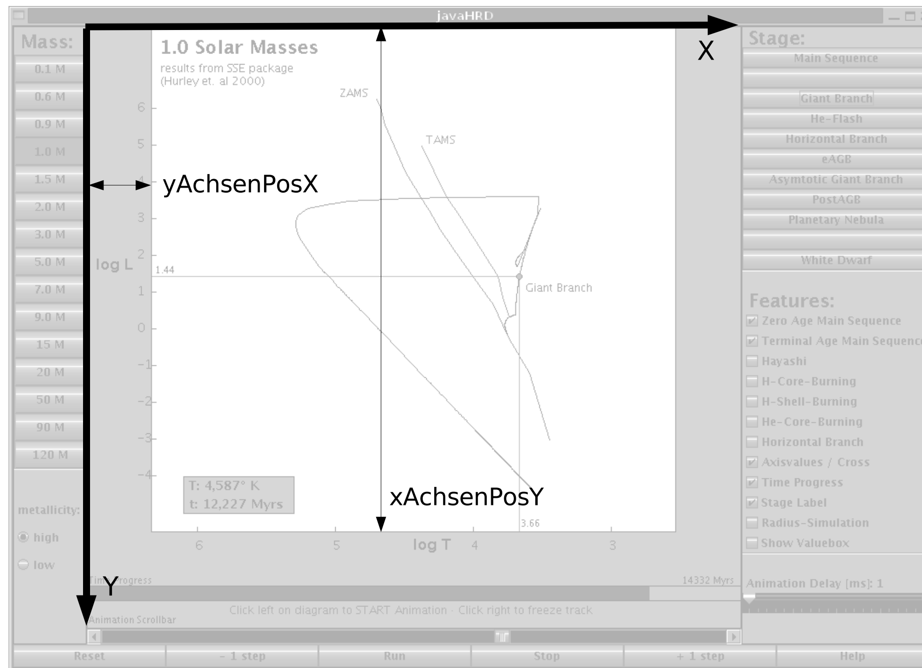

Abbildung 4.9: Skizze zur Klasse `sVar`

Zu Beginn werden alle in dieser Klasse benötigten Variablen deklariert und bei Bedarf sofort erzeugt (siehe Tab. 4.2). Bestimmte Objekte sollten entweder vom Benutzer an- bzw. abgeschaltet werden können oder unter bestimmten Bedingungen automatisch angezeigt werden. Dazu werden boolsche Variablen definiert, denen je nach dem ob das entsprechende Objekt gezeichnet werden soll oder nicht, der Wert *wahr* oder $falsch$ übergeben wird. Auch hier wurden möglichst intuitive Variablennamen gewählt.

Der Konstruktor `StarInHRD()` der Klasse wird anschliessend aufgerufen. Dadurch werden Layout und Eigenschaften der Diagrammebene definiert. Die bevorzugte Größe, die Eigenschaften des Rands und des Hintergrundes werden aufgerufen und eine Instanz der Standard-Java-Klasse `Timer` wird über die Variable `m_timer` initialisiert, wobei ein Zeitintervall festgelegt, und die innere Klasse `TimerAction` aufgerufen wird. Die Klasse `TimerAction` steuert die zeitliche Animation des gesamten javaHRDs. Jedesmal wenn diese Klasse aufgerufen wird, werden die Animationslaufvariablen `m_x` und `m_y` der Klasse `Star` um 1 erhöht. Befindet sich der entsprechende `m_x` Laufwert in einem bestimmten Wertebereich (`m_star.m_x` ≤ `Years.Length(m_star.m_M)`) wird anschließend die Diagrammebene über den Aufruf der Standard-Methode `repaint()` komplett neu gezeichnet. Es wird also indirekt nach jeder Schritterhöhung die `paintComponent` Methode neu ausgeführt. Das zeitliche Intervall wird über den *delay*-Wert Δt der Standard-Timer-Klasse in Einheiten von ms gesteuert. Beim ersten Aufruf wird der Wert auf 1 ms gesetzt und kann über den Zeitschiebe-Regler mit Hilfe der Standard-Methode `setDelay` aktiv vom Benutzer verändert werden.

Die Methode `paintComponent` zeichnet schliesslich alle Komponenten der Diagrammebene unter Berücksichtigung der zuvor eingestellten Parameter. Dazu wird die Standard-Methode `draw(g)` verwendet. Zunächst werden die Komponenten der Klasse `Diagram` gezeichnet. Anschließend werden die restlichen Komponenten nach Abfrage der zugehörigen boolschen Variable oder bei Eintreten einer bestimmten Bedingung nacheinander gezeichnet. Die aufsteigende Reihenfolge α der Zeichenebenen kann in Tabelle 4.2 abgelesen werden, wobei die Ebene mit $\alpha = 1$ zuerst gezeichnet wird.

Instanzvariable	Klasse	Boolsche Variable	α
m_timer	Timer	-	-
m_HRDarea	DiagramBackground	-	1
m_highlight	HighlightStage	highlightStage	2
m_HburnShellArea	HburnShellArea	showHburnShellArea	3
m_HburnCoreArea	HburnCoreArea	showHburnCoreArea	4
m_HeBurnCoreArea	HeBurnCoreArea	showHeBurnCoreArea	5
m_HB	HorizontalBranch	showHorizontalBranch	6
m_freezeTrack	FreezeTrack	showFreezedTrack	7
m_zams	ZAMS	showZAMS	8
m_tams	TAMS	show TAMS	9
m_hayashi	Hayashi	showHayashi	10
m_cross	Cross	showCross	11
m_star	Star	-	12
m_value	Values	showValues	13
m_titel	Titel	-	14
m_progress	TimeProgress	showTimeProgress	15
m_label	StageLabel	showStageLabel	16
m_stageBH	BlackHole	-	17
m_HRD	Diagram	-	18
m_infoBox	InfoBox	showInfoBox	19

Tabelle 4.2: Übersicht aller in der Klasse `StarInHRD` verwendeten Variablen mit zugehöriger boolschen Variable und Reihenfolge der Zeichenebenen α

4.4.6 Die Klasse `AnimationVariable`

Diese Klasse definiert die für alle Klassen zugängliche Laufzeitvariablen, welche die Grundlage für die Animation von javaHRD bildet. Diese werden über die Methoden $x(m_x)$ und $y(m_y)$ in Abhängigkeit der Animationslaufvariablen m_x und m_y gesetzt und als x bzw. y zurückgegeben.

```
public static int x(int m_x){
  int x = (int)
  (
  sVar.xPos
  - sVar.xSt * LogT.value(StarInHRD.m_star.m_M)[StarInHRD.m_star.m_x]
  )
  - (StarInHRD.m_star.ovalSize/2);

  return x;
}
```

Diese Transformation ist notwendig um die Ausgangswerte der Sternentwicklungswerte auf Pixelwerte abzubilden, die innerhalb des sichtbaren Pixelbereiches der Diagrammebene liegen. Bei dieser Gelegenheit werden gleich die $logL/logT$-Wertepaare, die als Variablentyp *Double* vorliegen, in Werte des Variablentyps *Integer* umgewandelt, der zur Deklaration von ganzzahligen Pixelwerten vorausgesetzt wird.

Die hier auftauchende Variable $ovalSize$ ist der in der Klasse `Star` definierte Durchmesser des gezeichneten Sternpunkts. Sein halber Wert, der Radius in Pixeln, wird an dieser Stelle abgezogen, damit schliesslich der gezeichnete Sternpunkt zentral auf dem gezeichneten Sternweg liegt (vgl. Kap. 4.4.8). Dies hängt mit den Eigenschaften der verwendeten Standardzeichenmethode `drawOval(a,b,width,height)` zusammen. Bei dieser Methode zum Zeichnen von Ellipsen bezeichnen die Eingabeparameter a_{oval} und b_{oval} die obere linke Ecke eines Rechtecks, innerhalb dessen die Ellipse auf der Ausgabeebene gezeichnet wird.

4.4.7 Die Klasse `Star`

Die Klasse `Star` ist die Klasse, die im wesentlichen alle Eigenschaften definiert oder aufruft, um einen Sternpunkt samt Entwicklungsweg in der Diagrammebene zu zeichnen und zeitlich darzustellen. Dazu werden vier Methoden verwendet, um die Animation zu steuern und eine Hauptmethode um einen Sternpunkt zu zeichnen.

Zunächst werden die Instanzlaufvariablen `m_x`, `m_y` und `m_M` definiert. Weiterhin wird ein `Color`-Objekt *color* für die farbliche Darstellung des Sternpunktes deklariert und der Durchmesser für den Sternpunkt über die Variable *ovalSize* auf eine Größe von 6 Pixeln gesetzt.

Anschließend wird ein Konstruktor für die Klasse `Star` definiert, der im Allgemeinen an `m_x`, `m_y` und `m_M` die Parameter x, y und M übergibt. x und y stellen dabei die vertikale bzw. horizontale Animations-Laufvariablen des Diagramms dar (siehe 4.4.6). M definiert den Wert für die gewählte Sonnenmasse des Entwicklungswegs. Die Methode `move()` setzt nun `m_x` und `m_y` schrittweise hoch und steuert so die Animation:

```
public void move() {
  if (m_x < Years.value.length) {
    m_x = m_x+1;
    m_y = m_y+1;
  }
}
```

Die Methoden `getX()` und `getY()` ermitteln in gleicher Weise die aktuelle Diagrammebenen-Postion des gezeichneten Entwicklungswertes:

```
public int getX() {
  return m_x;
}
public int getY() {
  return m_y;
}
```

Die Methode `setPosition(x,y)` übergibt die aktuellen Werte der Animations-Laufvariablen zurück an die Instanzlaufvariablen `m_x` und `m_y`:

```
public void setPosition(int x, int y) {
  m_x = x;
  m_y = y;
}
```

Die Hauptmethode dieser Klasse ist die Standard-Zeichenmethode `draw(Graphics)`. Hier wird all das in die Diagrammebene gezeichnet, was zur animierten Darstellung eines Sternpunktes, seines Entwicklungsweges und zusätzlich dargestellten Eigenschaften notwendig ist.

In dieser Methode werden zunächst die Transformationsvariablen aus `AnimationVariable` abgerufen. Zusätzlich wird der aktuelle Leuchtkraft- und Temperaturwert in den Variablen *vLeucht* bzw. *vTemp* gespeichert, die unter anderem für die Radiussimulation benötigt werden (vgl. Kap. 4.4.19). Weiterhin wird der Wert des Alters in der Variable *vTime* gespeichert. Temperatur in Kelvin und Alter in Millionen Jahren werden im animierten Diagramm in einem Kästchen unten links angezeigt. Dazu werden beide Werte zunächst mit der Java-Standard-Klasse `DezimalFormat` formatiert und dann als String gezeichnet (siehe Anhang A.4).

Die zentrale Aufgabe dieser Klasse, das Zeichnen eines Entwicklungspunktes erfolgt durch die Methode `drawOval()` und wird wie in Kapitel 4.4.5 besprochen bei jedem Schritt der Animation ausgeführt.

4.4.8 Die Klasse `Track`

Die Klasse `Track` zeichnet einen kompletten Sternentwicklungsweg. Als Instanzvariablen werden M und *connectRule* deklariert. Der Konstruktor der Klasse `Track` erhält den Massenparameter `M`, und der aktuelle Massenwert im Diagramm $StarInHRD.m_star.m_M$ wird auf M referenziert.

Die Darstellung des gesamten Sternentwicklungsweges erfolgt in zwei Schritten mit Hilfe der Methoden `getX()` und `getY()`, die zunächst die Datenpunkte in Pixelwerte umrechnen:

```
public static int getX (int i) {
  int x_i = (int)(sVar.xPos - sVar.xSt * LogT.value(m_M)[i]);
  return x_i;
}

public static int getY (int i) {
  int y_i = (int)(sVar.yPos - sVar.ySt * LogL.value(m_M)[i]);
  return y_i;
}
```

Als erstes werden alle Datenpunkte, die durch die Ausgangswerte der $value(M)$-Daten-Arrays von LogL und LogT zur Verfügung stehen gezeichnet. Dazu wird eine for-Schleife über i von 0 bis $Years.length(m_M)$ durchlaufen. Bei jedem Durchlauf werden Leuchtkraft- und Temperaturwerte in den Variablen x_i und y_i als transformierte Pixelwerte gespeichert und durch `getX(`i`)` bzw. `getY(`i`)` übergeben. Anschliessend wird ein Punkt an der Position (x_i, y_i) gezeichnet. Damit der Sternentwicklungsweg 1 Pixel breit ist, muss die Standardmethode `fillOval()` verwendet werden:

```
for (int i = 0; i < Years.Length(d_M); i++){
  int x_i = getX(i);
  int x_i = getY(i);
  g.fillOval(x_i, y_i, 1, 1);
}
```

Danach werden alle gezeichneten Sternpunkte durch Linien miteinander verbunden, so dass ein kontinuierlicher Sternentwicklungsweg dargestellt wird. Auch hier wird eine for-Schleife über alle benachbarten Datenpunkte durchlaufen. Zum Zeichnen der Verbindungslinie werden vier Variablen schleifenintern benötigt: x_i, y_i, x_{i+1} und y_{i+1}. Es wird nun bei jedem Durchgang eine Linie mit Hilfe der Standard-Methode `drawLine()` gezeichnet. Dazu werden zunächst für Temperatur- und Leuchtkraftwerte die Differenzvariablen dL und dT gebildet. Die in `connectRule()` abgefragte Bedingung lautet:

```
boolean rule = Math.abs(dT) < 0.5 || Math.abs(dL) < 0.8 && d_M < 50;
```

Die Verbindungslinien sollen allerdings nur dann gezeichnet werden, wenn der Betrag der Temperaturdifferenz dT kleiner als 0,5 ist, oder wenn der Betrag der Leuchtkraftdifferenz kleiner als 0,8 ist und die Anfangsmasse des Sterns kleiner als $50M_\odot$ ist.

Der Grund hierfür ist folgender: Wenn die verwendeten Daten in einer zeitlich äquidistanten Auflösung zur Verfügung stehen, so kann es vorkommen, dass bestimmte Entwicklunsphasen des Sterns nicht in den Datenpunkten erfasst werden, weil sie in einem sehr kurzen Zeitraum ablaufen. Ein Beispiel dafür wäre die postAGB-Entwicklung, über einen Planetarischen Nebel, hin zu einem Weissen Zwerg bei einem Stern mit $5M_\odot$. In diesem Fall würde dem letzten Datenpunkt des AGB gleich ein Datenpunkt eines Weissen Zwergs folgen. Diese beiden Sternpunkte dürfen natürlich nicht linear miteinander verbunden werden. Dies wäre eine falsche Darstellung der tatsächlichen Entwicklung. (siehe Abb. 4.10 und 4.11) Aus diesem Grunde wird dem Zeichnen der Linie die Methode `connectRule()` vorgeschaltet (siehe Anhang A.3).

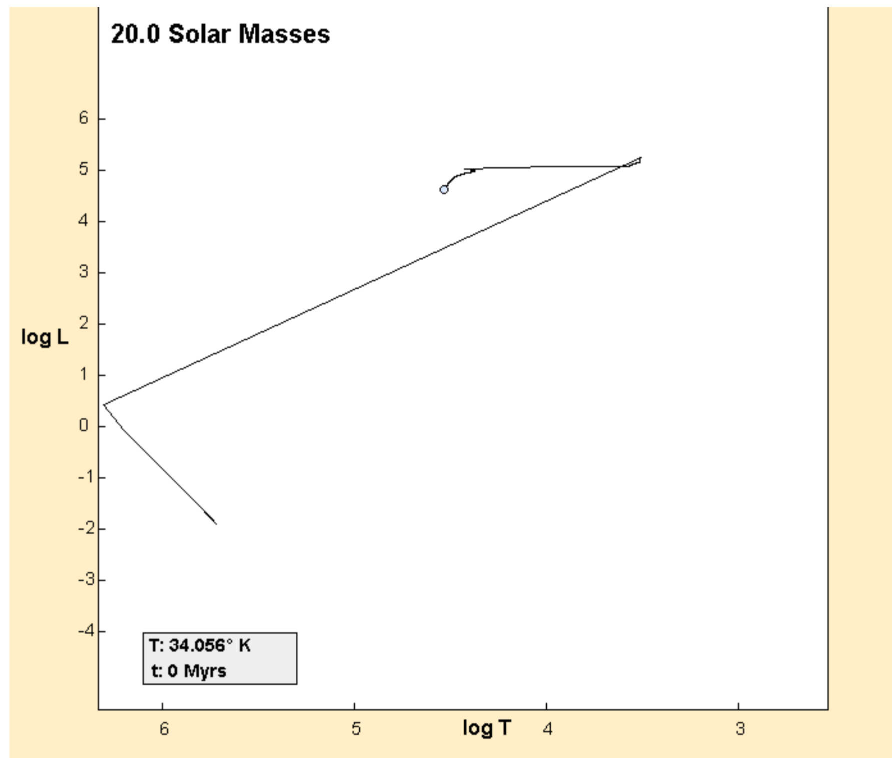

Abbildung 4.10: Zeichnen der Sternpunktverbindungslinien ohne `Track.connectRule()`

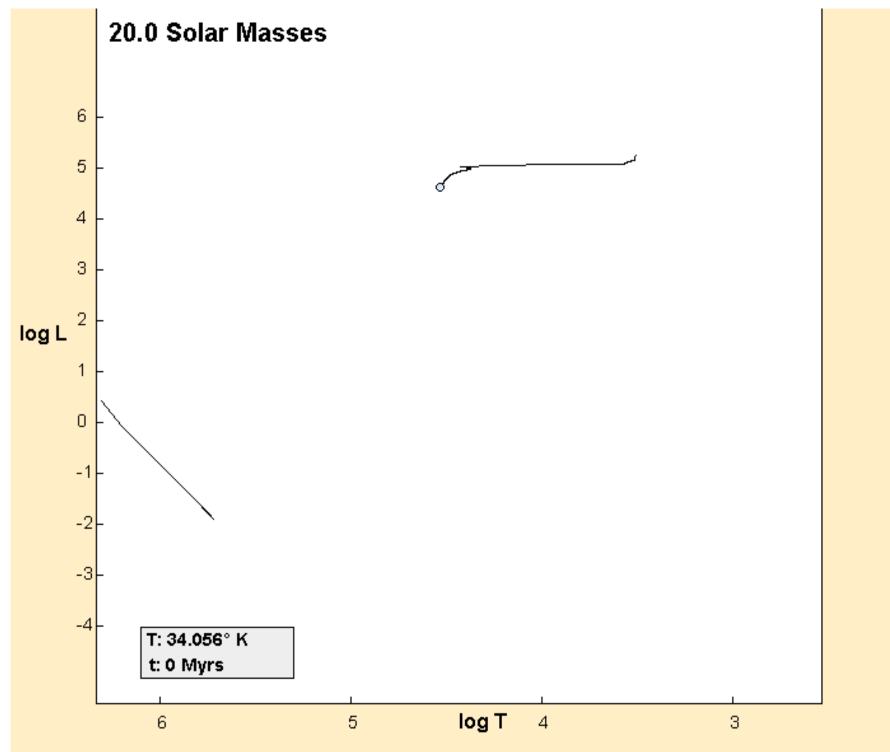

Abbildung 4.11: Zeichnen der Sternpunktverbindungslinien mit `Track.connectRule()`

4.4.9 Die Klassen `Diagram` & `DiagramBackground`

Diese Klasse zeichnet die Diagrammachsen und -achsenabschnitte in Abhängigkeit des Interfaces `sVar`.

Zunächst wird in der Klasse `DiagramBackground` mit der Methode `setColor()` die Hintergrundfarbe der Diagrammfläche gesetzt und ein Rechteck mit den Eckpunkten $yAchsenPosX$, 0, $axisWidth$ und $xAchsenPosY$ gezeichnet und gefüllt. Dies geschieht mit der Methode `fillRect()`. Die Trennung von Hintergrund und Diagrammelementen erfolgt aus Gründen des Layouts, da der Hintergrund als erstes ($\alpha = 1$) und der Rest des Diagramms fast als letztes ($\alpha = 18$) gezeichnet wird. (siehe Tab. 4.2).

In der Klasse `Diagram` werden alle wesentlichen Diagramelemente erzeugt. Die Zeichenfarbe wird mit `setColor()` auf Schwarz gesetzt und der Rand des Rechtecks mit der Methode `drawRect()` gezeichnet. Vor dem Zeichnen der Achsenabschnitte werden noch die Achsenabschnittsbezeichnungen gesetzt.

Jetzt können die Achsenabschnitte gezeichnet werden. Die Abschnitte werden entlang der Diagrammflächen-Ränder als Linien der Pixellänge 5 gezeichnet. Die Position der Achsenabschnitte wird nach einer empirischen Formel ermittelt.

Zeichnen der X-Achsenabschnitte

Zum Zeichnen der X-Achsenabschnitte wird eine for-Schleife durchlaufen, dessen Zählvariable in Anlehnung an den beim HRD interessanten $logT$-Wertebereich von 3 bis 7 läuft. In Abhängigkeit von dieser Zählvariable i und den Variablen $xPos$ und xSt der Klasse `sVar`, wird die Hilfsvariable $interceptValue$ ermittelt. Schliesslich wird an der Position $interceptValue$ mit Hilfe der Methode `drawLine` der Achsenabschnitt gezeichnet:

```
for (int i = 3; i < 7; i++){
  int interceptValue = (-sVar.xSt*(i-4))+sVar.xPos-600;
  g.drawString(""+(i),interceptValue-2,sVar.xAchsenPosY+20);
  g.drawLine (interceptValue,sVar.xAchsenPosY-5,interceptValue,sVar.xAchsenPosY);
}
```

Zeichnen der Y-Achsenabschnitte

Aus gleichem Grund wie zuvor, wird um Zeichnen der Y-Achsenabschnitte, in Anlehnung an den $logL$-Wertebereich im HRD eine Zählvariable verwendet, die diesmal von -4 bis 7 verläuft. Von dieser Zählvariable i und den Variablen $yPos$ und ySt der Klasse `sVar` hängt wiederum die lokale Hilfsvariable $interceptValue$ ab:

```
for (int i = -4; i < 7; i++){
  int interceptValue = (-sVar.ySt*i)+sVar.yPos-1;
    g.drawString(""+(i),sVar.yAchsenPosX-15,interceptValue+4);
    g.drawLine(sVar.yAchsenPosX,interceptValue,sVar.yAchsenPosX+5,interceptValue);
}
```

4.4.10 Die Klassen `ZAMS`, `TAMS` und `Hayashi`

Die Klasse `ZAMS` zeichnet die Zero Age Main Sequence (ZAMS) in das Diagramm. Die Wertepaare sind in einem zweidimensionalem Array `zams` gespeichert, dass seine Werte aus den Startwerten der einzelnen Sternentwicklungswege bezieht. Die lokale Variable $length$ ermittelt die Anzahl der Wertepaare minus eins. Zunächst wird die Linienfarbe bestimmt und der Linientitel gesetzt. Schliesslich werden mit Hilfe der `drawline`-Methode über eine for-Schleife Verbindungslinien zwischen dem i-ten und $(i+1)$-ten Wertepaars des `ZAMS.zams`-Arrays gezeichnet. Dabei werden die einzelnen Punkte wie folgt ermittelt:

```
int x1 = (int)(sVar.xPos - sVar.xSt * zams[0][i]);
int x2 = (int)(sVar.xPos - sVar.xSt * zams[0][i+1]);
int y1 = (int)(sVar.yPos - sVar.ySt * zams[1][i]);
int y2 = (int)(sVar.yPos - sVar.ySt * zams[1][i+1]);
```

Die Klasse TAMS zeichnet unter der Verwendung anderer Ausgangspunkte in gleicherweise die Terminal Age Main Sequence (TAMS). Die Hayashi Linie wird gleichermassen von der Klasse Hayashi in das Diagramm gezeichnet. Allerdings werden die zugrundeliegenden Werte manuell gewählt und angepasst, da für diese Linie keine theoretischen Werte zu Verfügung stehen. Die Datenpunkte werden gesetzt und in einer Kurve möglichst nah an den Rote Riesen Ast des Entwicklungweges eines 0,6$M_{\odot}$ -Sterns genähert (vgl. Abb. 2.2). Dieses Verfahren wird aus didaktischen Gründen angewendet, damit diese wichtige Grenze zum Bereich im HRD, in dem keine Sterne vorkommen, angezeigt werden kann.

4.4.11 Die Klassen HburnCoreArea, HburnShellArea, HeBurnCoreArea und HorizontalBranch

Durch diese Klassen werden Bereiche farblich hervorgehoben, in denen stabile thermonukleare Prozesse stattfinden. Die einzelnen Klassen übernehmen dass Zeichnen der zugehörigen Flächen. Die Klasse HburnCoreArea zeichnet den Bereich stabilen Wasserstoffbrennens im Kern, die Klasse HburnShellArea den Bereich stabilen Wasserstoffbrennens in einer Schale und die Klasse HeBurnCoreArea den Bereich stabilen Heliumbrennens im Kern.

Bei der Klasse HburnCoreArea wird dazu ein einfacher Trick verwendet. Die ZAMS Linie wird jeweils um ein Pixel nach rechts verschoben gezeichnet. So entsteht schliesslich das Hauptreihen-Band, dessen Pixelbreite von der Anzahl der Schleifendurchläufe bestimmt wird und so gewählt wurde, dass der Bereich zwischen ZAMS und TAMS eingefärbt wird.

Ähnlich wird durch die Klasse HburnShellArea das Riesenastband gezeichnet, wobei als Grundlage aber die Hayashi Linie genommen wird. Statt pixelweise nach rechts zu zeichnen, erfolgt die Zeichnung zur linken Seite. Schliesslich werden durch die HeBurnCoreArea in gleicher Weise wie bei der Klasse HburnCoreArea gezeichnet, wobei nun aber die Daten aus einem Array der Klassen BlueLoop und HorizontalBranch zusammengestellt werden. Bei diesen beiden Bereichen wird die Pixelbreite nach didaktischen Ansprüchen manuell festgelegt (vgl. Abb. 2.3).

Die Klasse HorizontalBranch zeichnet den Horizontalast in das HRD. So wird der Bereich hervorgehoben, auf dem Sterne landen, die einen Helium-Flash am Ende des Riesenastes erleben.

4.4.12 Die Klasse Cross

Die Klasse zeichnet ein halbes Fadenkreuz um den aktuellen Entwicklungspunkt mit Schnittpunkten an der x- bzw. y-Achse. Dort werden die aktuellen log T_{eff}- beziehungsweise log L/$L_{\odot}$ -Werte angezeigt. Dazu wird zunächst der jeweils aktuelle Datenpunkt der Übersicht halber auf die zweite Dezimalstelle mit Hilfe der Standard-Methode Math.rint gerundet. Danach wird der Wert in Abhängigkeit der Achsenposition und des aktuellen Animationsvariablen-Wertes gezeichnet. Zusätzlich wird jeweils eine orthogonale Linie von Datenpunkt zu den Diagrammachsen gezeichnet, deren Position und Längen von den aktuellen Animationsvariablen abhängen.

4.4.13 Die Klasse TimeProgress

Diese Klasse zeichnet einen Fortschrittsbalken, der die verstrichene Zeit, also das Alter der Sterne grafisch darstellt. Die Gesamtbreite des Balkens entspricht dabei der Lebensdauer des animierten stellaren Entwicklungswegs. Der Wert der Lebensdauer des Sterns wird unten rechts neben dem Zeitbalken angegeben. Man kann hier also sofort sehen wie alt der Stern werden wird und anhand des Fortschritts des Balkens abschätzen, wie alt Stern bereits geworden ist.

Programmtechnisch gesehen funktioniert der Fortschrittsbalken wie folgt: Zunächst wird die Höhe des Balkens durch die Variable *height* gesetzt. Die Position auf der Diagrammebene wird durch die Variablen *posX* und *posY* bestimmt. Die Breite der Diagramebene wird in $diagrammEbeneWidth$ angegeben. Durch die Vorgabe der Daten über die Years-Klasse und die gewählte Anfangsmasse wird zunächst die Länge des Daten-Arrays Years.Length in der Variable *length* gespeichert. Der maximale Zeitwert des Fortschrittbalkens, also die gesamte Lebensdau-

er des Sterns, wird in der Variablen $balkenMax$ und der aktuelle Zeitwert in der Variablen $balkenRel$ gespeichert:

```
double balkenMax = 1000*Years.value(StarInHRD.m_star.m_M)[length];
double balkenRel = 1000*Years.value(StarInHRD.m_star.m_M)[StarInHRD.m_star.m_x];
```

Anschliessend wird $balkenRel$ zur Darstellung in $balkenWidth$ skaliert:

$$balkenWidth(m_x) = diagrammEbeneWidth \cdot (balkenRel(m_x)/balkenMax);$$

Da die Datenwerte in Millionen Jahren vorliegen, wurde der Faktor 1000 gesetzt, um auch die Nachkommastellen der vorliegenden Daten bei der Umwandlung in den zu zeichnenden Integer Wert zu berücksichtigen. So wird eine höhere Genauigkeit zu erreicht.

Das Zeichnen des Fortschrittbalkens innerhalb eines Animationsschrittes erfolgt in drei Schritten. Als erstes wird der Titel der Balkenleiste und die Angabe der Gesamtlebensdauer als String gezeichnet. Danach wird ein Rahmen für den eigentlichen Balken gesetzt. Der Fortschrittsbalken wird nun mit der `drawRect` Methode in Abhängigkeit von $balkenWidth$ gezeichnet.

Abbildung 4.12: zeitlicher Fortschrittsbalken, Animationsbildlaufleiste und -schieberegler

4.4.14 Die Klasse `StageLabel`

Die Klasse ist für die Anzeige der aktuellen morphologischen Bezeichnung der stellaren Entwicklungsphase zuständig. Anhand der zeitlichen Grenzen der Phase aus dem Array `stage` der aktuell referenzierten Klasse `Years` wird ermittelt, in welcher Phase der Stern sich gerade befindet. Gesteuert wird dies über mehrere verschachtelte if-Anweisungen. Zunächst werden die Positionslaufvariablen dPX und dPY aus der `AnimationVariable`-Klasse importiert (siehe Kap. 4.4.6). Ausserdem wird eine zeitliche Laufzeitvariable t in Abhängigkeit des aktuellen m_x-Wertes und des aktuellen Massenwertes m_M gesetzt:

```
double t = Years.value(StarInHRD.m_star.m_M)[StarInHRD.m_star.m_x];
```

Nach dem Schriftgrösse und Farbe gesetzt werden, kann über die bereits angesprochene if-Anweisungen ermittelt werden, ob t den letzten Wert der Phase überschritten hat. Die abgefragte Bedingung lautet dann je nach Phasenwert s:

```
t < Years.stage(StarInHRD.m_star.m_M)[s],
```

Sobald also t den letzten Wert überschreitet, wird die Bezeichnung der Entwicklungsphase angezeigt, die als String im Array `Years.stage` aktuell referenziert ist.

Nach Abfrage der Bedingung wird schliesslich die morphologische Bezeichnung mit der Standardmethode `drawString` gezeichnet:

```
g.drawString(Years.stageName(StarInHRD.m_star.m_M)[a],dPX+10,dPY+20)
HRDPanel.stageButton(a).setSelected(true);
```

Zusätzlich wird der Zustand der zur Phase gehörenden Phasenauswahl-Schaltfläche mit der Methode `setSelected()` auf aktiv geschaltet. Damit die Bezeichnung nicht den Sternpunkt überschreibt, wird diese aus Gründen der Übersichtlichkeit um 20 Pixel nach oben und um 10 Pixel nacht rechts verschoben gezeichnet. Es können insgesamt bis 10 Phasen unterschieden werden.

4.4.15 Die Klasse `HighlightStage`

Die Klasse `HighlightStage` hebt eine Sternentwicklungsphase hervor. Beim Drücken einer bestimmten Phasen-Schaltfläche, wird die entsprechende Phase farblich hervorgehoben. Zunächst wird die Farbe gewählt mit der die Phase hervorgehoben werden soll. Dann wird, wie bereits in Kapitel 4.4.14 beschrieben eine zeitliche Laufvariable t definiert, die wiederum mit Hilfe einer verschachtelten if-Anweisung die ausgewählte Phase abfragt und diese dann zeichnet.

Das Zeichnen der Phase übernimmt die Methode `drawStage()` mit Hilfe der Integervariable $stage$. Zunächst wird der aktuelle Massenwert in bekannter Weise auf eine Massen-Variable h_M referenziert und ein Integer-Wert w für Dicke des hervorgehobenen Bereichs festgelegt. In einer for-Schleife über i, die bis $Years.Length(h_M)$ läuft, werden zunächst zwei Bedingungen in einer if-Anweisung abgefragt.

Durch diese werden die anschliessenden Aktionen nur dann ausgeführt, wenn der i-te Datenpunkt kleiner als der höchste Datenpukt, oder grösser als der kleinste Datenpukt der Phase $stage$ ist. So wird also der benötigte Teilbereich des Entwicklungsweges bestimmt und gezeichnet.

Die Zeichnung umfasst pro Sternpunkt ein `fillOval`-Objekt, ein `drawoval`-Objekt und eine Linie der Dicke w, die den i-ten Sternpunkt mit dem vorhergehenden Sternpunkt verbindet. Zum Zeichnen der Linie benötigt man erneut eine zweite Zählvariablen $j = i - 1$. Weiterhin die aus der Klasse `Track` bekannten Variablen, x_i, y_i, x_j und y_j, die hier in ähnlicher Weise über die Methoden `getX()` und `getY()` innerhalb der Schleife gesetzt werden. Über eine zweite for-Schleife werden mehrere Linien nebeneinander zeichnet. All dies geschieht nur dann, wenn die aus der Klasse `Track` bekannte Zeichen-Methode `connectRule` erfüllt ist.

Als Sonderfall wird die 0-te Phase behandelt. Die Abfrage erfolgt über eine if-Anweisung. Tritt dieser Fall ein, so wird nur die Bedingung $Years.value(h_M)[i] \leq Years.stage(h_M)[0]$ abgefragt. Eine Abfrage der `connectRule` ist auf Grund des bekannten Verlaufs aller Hauptreihen-Phasen nicht notwendig.

4.4.16 Die Klasse `Values`

In dieser Klasse werden die aktuellen Zustands-Werte der zugrundeliegenden Daten angezeigt ($logL$, $logT$ und die Zeitangabe). Ausserdem wird der aktuelle $delay$-Wert und der Animationslaufzeitwert m_x angezeigt. Diese Anzeige kann bei Bedarf an- oder abgeschaltet werden.

4.4.17 Die Klasse `FreezeTrack`

Die Klasse `FreezeTrack` erbt alle Eigenschaften von der Klasse `Track` und zeichnet in gleicher Weise den Entwicklungsweg nach. Ein Unterschied ist aber die Zeichenfarbe. Um zu verdeutlichen, dass dieser Entwicklungsweg eingefroren, bei einem Wechsel der Masse also weiterhin gezeichnet wird, wurde ein helles, eisiges Blau gewählt. Zusätzlich wird der Massewert des fixierten Entwicklungsweges in der Mitte das Diagramms angezeigt.

4.4.18 Die Klasse `StarColor` - Zeichnen der Sternfarbe in Abhängigkeit der Temperatur

Die Temperaturwerte werden mit einer einfachen Formel auf die HSV-Farbskala[5] abgebildet, so dass der Stern in Abhängigkeit von der Temperatur in den entsprechenden Farben gezeichnet wird. Dazu wird zunächst die Farbvariable $starColor$ gesetzt. Ausserdem wird eine Float-Variable c deklariert. Dem Konstruktor der Klasse wird ein Double Parameter $inputValue$ zugewiesen. Die Variable c hängt in folgender Weise von $inputValue$ ab:

```
c = (float)(2*(inputValue/10-0.24)-0.13);die
```

[5]Beim HSV-Farbmodell werden Farben mit Hilfe des Farbtons (*hue*), der Sättigung (*saturation*) und dem Grauwert (*value*) definiert.

Um grüne Farbwerte zu überspringen, wird zunächst mit Hilfe einer if-Anweisung geprüft, ob c grösser als 0.17F ist. Dann wird $starColor$ in Abhängigkeit von c ermittelt:

```
if (c > 0.17F) {
  starColor = Color.getHSBColor( c +0.30F, 0.15F,1);
} else {
  starColor = Color.getHSBColor( c , 1-( c *3),1);
}
```

Die hier verwendeten Formeln wurden empirisch ermittelt. Die Werte wurden so eingestellt, dass die Farbe des Sternes ungefähr der erwarteten Farbe entspricht.

4.4.19 Simulation des Sternradius

Zur Simulation des Radius wird die Beziehung zwischen Leuchtkraft, Radius und Temperatur ausgenutzt. Für die Leuchtkraft, als Gesamtenergie, die von einer nach Planck strahlenden Kugel freigesetzt wird, gilt:

$$L = 4\pi\sigma R^2 T^4$$

Für den Radius gilt dann:

$$R \sim \sqrt{L}/T^2$$

Diese Proportionalität macht sich der Programmcode zur Radius-Simulation zu nutze. Dazu wird zunächst die Variablen $vLeucht$ und $vTemp$ aus den zur Verfügung stehenden Daten ermittelt:

```
double vLeucht = Math.pow(10,LogL.value(StarInHRD.m_star.m_M)[StarInHRD.m_star.m_x]);
double vTemp = Math.pow(10,LogT.value(StarInHRD.m_star.m_M)[StarInHRD.m_star.m_x]);
```

Anschliessend wird der Variable $starRadius$ der Wert $\sqrt{vLeucht}/vT^2 * \beta$ zugewiesen, wobei $\beta = 5 * 10^7$ ein empirischer Faktor ist, der das Ergebnis der Division auf im Diagramm sichtbare Integer-Pixelwerte abbildet:

```
int starRadius = (int)( Math.sqrt(vLeucht)/(vTemp*vTemp) * 5E7 );
```

So wird also eine Simulation erzeugt, die den gewünschten Effekt in den erforderlichen Phasen darstellt. Dabei wird keine Rücksicht auf reale Proportionen genommen. Es besteht während der Simulation kein direkter Zusammenhang zwischen dem Pixelwert $starRadius$ und dem für den Sternpunkt gewählten Anfangswert $ovalSize$ (siehe Abb. 4.13).

4.4.20 Die Klasse `InfoBox`

Diese Klasse ist für die Anzeige einer Informationsbox zuständig, die eine allgemeine Erklärung der einzelnen Entwicklungsphasen liefert. Die Box erscheint sobald die Phasenschaltflächen mit der rechten Maustaste gedrückt und gehalten werden.

Als Instanzvariablen, werden eine Integer $boxID$ und eine boolsche Variable $showInfobox$ gesetzt. Nach Aufruf des Klassenkonstuktors werden zunächst die Dimensionen der Box über die Variablen $boxWidth$ und $boxHeight$ festgelegt. In den Variablen boY und boX wird die Position der oberen linken Boxecke festgelegt. Damit die Box keinen Sternentwicklungsweg überdeckt, wird für boY eine if-Anweisung abgefragt, welche die Box bei $M \leq 9\ M_\odot$ auf eine andere Position setzt.

Nach dem Zeichnen der Box und Setzen der Schriftart wird eine Überschrift gezeichnet. Bestandteil der Überschrift ist ein Stringwert, der wie folgt aus der aktuellen Schaltflächenbezeichnung ermittelt wird:

```
Years.stageName(StarInHRD.m_star.m_M)[boxID]
```

Anschliessend wird in mehreren if-Anweisungen geprüft, ob der von der inneren Klasse der Phasenschaltflächen durch die $boxID$ übergebene String-Wert $Years.stageName$ einem bestimmten String-Wert (z.B Beispiel *Main Sequence*) gleicht. Bei Übereinstimmung wird der zu der ausgewählten Entwicklungsphase gehörende Informationstext angezeigt (siehe Abb. 4.13).

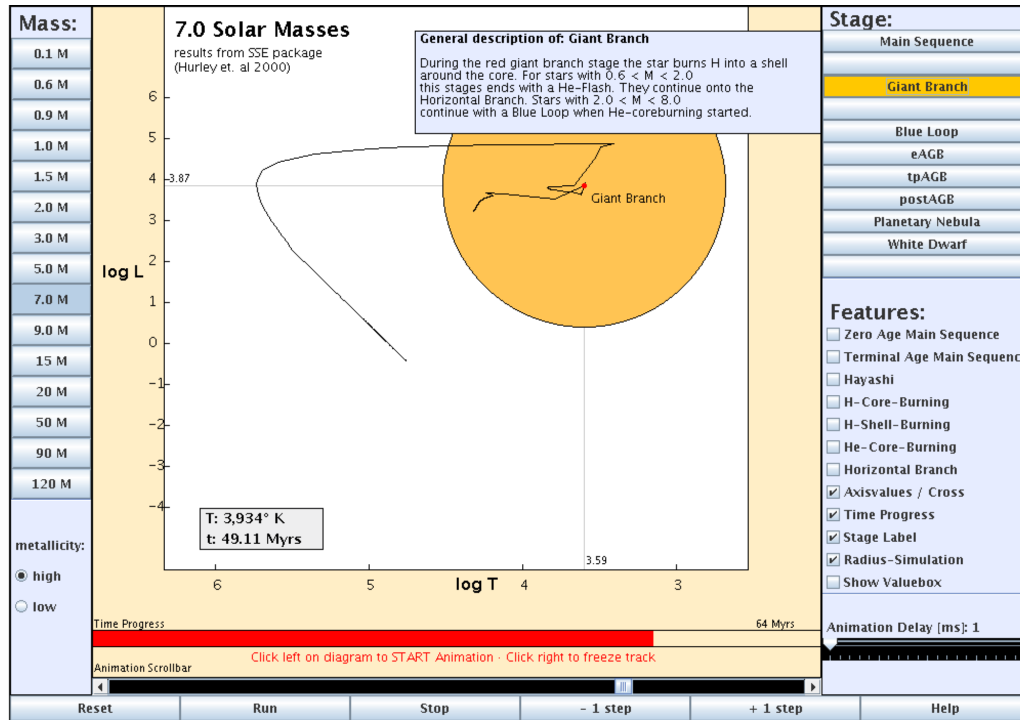

Abbildung 4.13: Radiussimulation und Infobox

4.5 Funktionsweise im Detail

In diesem Kapitel werden einzelne Funktionsdetails des javaHRD-Applets beschrieben. Dazu gehören die inneren Klassen der Klasse `HRDPanel`, die spezielle Funktionen übernehmen, aber auch Methoden der Klasse `StarInHRD` aufrufen. Zunächst wird die Funktionsweise der inneren Klasse `TimerAction` beschrieben, die für die Animation von zentraler Bedeutung ist.

4.5.1 Die innere Klasse `TimerAction`

Diese innere Klasse ist die zentrale Klasse der gesamten Animation von javaHRD und implementiert einen `ActionListener`, der auf Ereignisse e_{timer} des Timers hört. Tritt ein Timer-Ereignis ein, wird die in Kapitel 4.4.7 beschriebene Methode `move()` aufgerufen. Über zwei if-Anweisungen wurde gesteuert, dass der Timer Listener sofort stoppt, wenn der m_x-Wert ausserhalb des sinnvollen Bereiches zwischen 0 und $Years.length(m_star.m_M) - 1$ liegt:

```
class TimerAction implements ActionListener {
  public void actionPerformed(ActionEvent e_time) {
    m_star.move();
    if (m_star.m_x == Years.length(m_star.m_M)-1) {
      m_timer.stop();
    }
    if (m_star.m_x >= 0 && m_star.m_x <= Years.length(m_star.m_M)) {
      repaint();
    } else {
        m_timer.stop();
      }
  }
}
```

4.5.2 Die javaHRD-Animation

Die Animation als zentraler Bestandteil der dynamischen Darstellung von javaHRD sei hier nochmal zusammengefasst beschrieben. Die Grundlage der Animation bildet die Erzeugung eines Objektes der Standard-Klasse `Timer` die vom Standard-Paket `javax.swing` zur Verfügung gestellt wird:

```
m_timer = new Timer(1, new TimerAction());
```

Der Aufruf geschieht im Konstruktor der Klasse `StarInHRD`. Der Timer ist also wesentlicher Bestandteil eines jeden `StarInHRD`-Objektes (siehe 4.4.5):

```
public StarInHRD() {
    .
    m_timer = new Timer(1, new TimerAction());
}
```

und setzt sozusagen die Zeit des gezeichneten Sternpunktes. Bei jedem Ereignis e_{timer} auf den die innere Klasse `TimerAction` hört, wird dort die Methode `Star.move()` aufgerufen (siehe 4.5.1):

```
public void actionPerformed(ActionEvent e_time) {
    m_star.move()
    .
}
```

Die aufgerufene Methode wiederum setzt den Wert der Animationsvariablen hoch (vgl. 4.4.7):

```
public void move() {
    .
    m_x = m_x+1;
    m_y = m_y+1;
    .
}
```

Jedes `Timer`-Ereignis setzt also einen Animationsschritt, nach dem die Diagrammebene komplett neu gezeichnet wird. Dadurch erhält man eine Abfolge von Diagrammebenen und das menschliche Auge nimmt diese Sequenz als bewegtes Bild wahr. Das Ergebnis ist hier schliesslich die Animation des Hertzsprung-Russell-Diagramms.

4.5.3 Die inneren Klassen von `HRDPanel`

Die inneren Klassen lassen sich in vier Gruppen mit zum Teil gruppenspezifisch ähnlichen Funktionen einteilen: Steuerungsklassen, Massenauswahlklassen, Phasenauswahlklassen und Funktionsklassen.

Die Steuerungsklasse Start, Stop und Reset

Die Klasse `Start` implementiert sowohl einen `ActionListener`, der auf ein `ActionEvent` a hört und einen `MouseListener` der auf ein `MouseEvent` e hört. Je nach Event werden bestimmte Methoden der Klasse `StarInHRD` aufgerufen.

Dem `ActionListener` wurde die Schaltfläche *startButton* zugeordnet. Bei Betätigung der Schaltfläche wird die Methode `startApp()` aufgerufen und die Animation des javaHRD-Applets gestartet. Der `MouseListener` wurde dem gesamten Diagrammobjekt `StarInHRD` zugewiesen. Beim Drücken des linken Mausknopfes wird die Methode `runApp()` aufgerufen. Durch einen beliebigen Klick auf die Diagrammebene wird also ebenfalls die Animation gestartet. Wird die rechte Maustaste betätigt erfolgt das Einfrieren des Entwicklungsweges über `setFreezeTrack()`.

Die Klasse `Stop` implementiert einen `ActionListener`, hört auf einen `ActionEvent` *e* und ruft beim Betätigen der Schaltfläche *stopButton* mit der linken Maustaste die Methode `stopApp()` auf, wodurch die Animation gestoppt wird.

Die Klasse `Reset` implementiert in gleicher Weise einen `ActionListener`. Sie ruft beim Betätigen der Schaltfläche *resetButton* mit der linken Maustaste die Methode `resetApp()` auf, welche das Applet zurückgesetzt.

Die Steuerungsklassen StepFWD & StepRWD

Die Klassen `StepFWD` und `StepRWD` implementieren einen `ActionListener`, der auf ein Ereignis *e* hört. Wird die Schaltfläche *fwdButton* bzw. *rwdButton* mit der linken Maustaste gedrückt, so wird die jeweils entsprechende Methode `stepFWD()` bzw. `stepRWD()` aufgerufen. Anschliessend wird der Wert der Bildlaufleiste *animationScroll* über die Standardmethode `StarInHRD.m_star.getX()` aktualisiert.

Die Steuerungsklassen DelaySlider & SetDelayByWheel

Diese beiden Klassen dienen der Aktualisierung und Steuerung des Animationszeit-Schiebereglers.

Die Klasse `DelaySlider` implementiert einen `ChangeListener`, der auf ein ChangeEvent *evt* hört und dient der Steuerung des Delay-Reglers *delaySlide*. Dessen Position wird über die Variable *slideVal* ermittelt. Danach wird der Delay-Wert von *m_timer* auf *slideVal* gesetzt. Ausserdem wird der aktuell eingestellte Delay-Wert *delay* über die Standard-Timer-Methode `getDelay()` ermittelt. Der aktuelle Wert wird über *sliderLabel* mit Hilfe der Methode `setText` gesetzt und angezeigt. Schliesslich wird die Diagrammebene neu gezeichnet.

Die Klasse `SetDelayByWheel` implementiert einen MouseWheelListener, der auf ein MouseWheelEvent *e* hört. Dies erlaubt die Steuerung des Delay-Reglers über das Rad der Maus. Dazu werden zunächst die Integer Variablen *count*, *wheelRange* und *delay* gesetzt:

```
int count = e.getWheelRotation();
int wheelRange = 50;
int delay = StarInHRD.m_timer.getDelay();
```

count ermittelt über die Methode `getWheelRotation()` die Anzahl der MouseWheelEvents. *wheelRange* erhält eine ganzzahligen Wert, der das Intervall für ein Event festlegt. *delay* wird wiederum über die bereits erwähnte Standard-Timer-Methode ermittelt. Nun wir geprüft, wie oft und in welche Richtung sich das Mausrad dreht. Ausserdem werden Bedingungen für die Reaktionsgrenzen des Sliders *delaySlide* gesetzt. Nach Abfrage aller Bedingungen wird der Delay-Wert neu gesetzt.

Die Steuerungsklassen SetX, ScrollPosX & WheelX

Diese drei Klassen dienen der Steuerung der über die Standard-Klasse `JScrollBar` erzeugten Animations-Bildlaufleiste *animationScroll*.

Die Klasse `SetX` ist für die manuelle Steuerung der Animation zuständig und implementiert einen `AdjustmentListener`, der auf ein `AdjustmentEvent` *evt* hört. Tritt ein *evt*-Ereignis ein, wird zunächst der Wertebereich über die Standard-Methode `setMaximum()` auf die Anzahl der Datenpunkte gesetzt. Über die Standard-Methode `getValue()` übergibt der Regler seinen aktuellen Wert an die Animationsvariablen *m_x* und *m_y*. Stellt man also mit der Maus einen bestimmten Wert der Laufleiste ein, wird dieser sofort übernommen und der Sternpunkt springt an die eingestellte Animations-Position *m_x*:

```
class SetX implements AdjustmentListener {
  public void adjustmentValueChanged( AdjustmentEvent evt ) {
    animationScroll.setMaximum((int)Years.length(StarInHRD.m_star.m_M)-1);
    StarInHRD.m_star.m_x = animationScroll.getValue();
    StarInHRD.m_star.m_y = animationScroll.getValue();
```

```
      repaint();
    }
  }
```

Die Klasse `ScrollPosX` ist dafür zuständig, dass die Position der Bildlaufleiste aktualisiert wird, sobald die Animation automatisch läuft. Hier wird über ein `ActionListener` mit Hilfe der Methode `Star.getX()` auf den aktuellen m_x-Wert gehört und durch die Methode `setValue()` gesetzt:

```
class ScrollPosX implements ActionListener  {
  public void actionPerformed( ActionEvent e ) {
    animationScroll.setValue( StarInHRD.m_star.getX() );
  }
}
```

Mit Hilfe der Klasse `WheelX` steht dem Benutzer eine zweite Möglichkeit zur Steuerung über das Mausrad zur Verfügung. Mit dieser Funktion kann man die Bildlaufleiste, und damit die Animations-Position des Sternpunktes, über die Mausradbewegung steuern. Dies wird durch einen `MouseWheelListener` umgesetzt, der auf ein `MouseWheelEvent` *e* hört. Zunächst werden die drei Hilfsvariablen *anzahlDaten*, *range* und *count* definiert. Wie der Name schon sagt, definiert *anzahlDaten* die Anzahl der aktuell verwendeten Datenpunkte. Für *range* wird das Intervall für die Anzahl der Animationsschritte gesetzt, die bei einem Ereignis des Mausrads übersprungen werden. *count* fragt über die Standard-Methode `getWheelRotation` die Anzahl der Klicks des Mausrades ab. Wird das Mausrad vom Benutzer weg gedreht, erhält *count* ein negatives Vorzeichen. Bei der Drehung auf den Benutzer zu, bleibt der Wert positiv. Mit einer einfachen if-Anweisung, die den Betrag von *count* abfragt, wird nun m_x um den Wert von *range* erhöht oder verringert:

```
class WheelX implements MouseWheelListener {
  public void mouseWheelMoved(MouseWheelEvent e) {
    int anzahlDaten = (int)Years.length(StarInHRD.m_star.m_M)-1;
    int range = anzahlDaten/100;
    int count = e.getWheelRotation();
    if (count* (-1) > 0) {
      animationScroll.setValue(StarInHRD.m_star.m_x+range);
    } else {
      animationScroll.setValue(StarInHRD.m_star.m_x-range);
    }
    repaint();
  }
}
```

Phasenauswahlklassen

Alle Phasenauswahlklassen haben einen Namen der Form `Stage`P, wobei P die Nummer der ausgewählten Phase angefangen bei 0 angibt. Der Aufruf geschieht über die entsprechenden Phasenauswahl-Schaltfläche *stageButtonP* und lautet für $P = 2$:

```
public static JToggleButton stageButton2  = new JToggleButton();
.
stageButton2.setText(Years.stageName(StarInHRD.m_star.m_M)[2]);
stageButton2.setToolTipText("click right for InfoBox");
.
stageButton2.addMouseListener(new Stage2());
```

Am Beispiel von `Stage2` sei die Funktion der Phasenauswahlklassen erläutert: Jede Klasse implementiert einen `MouseListener`, der auf ein `MouseEvent` Ereignis *arg*0 hört. Es gibt zwei Fälle,

bei denen der Listener reagiert und die entsprechenden Standard-Methoden eines Mouselisteners aufruft: Drücken der Maustaste und Lösen der Maustaste.

Im ersten Fall werden über die `mousePressed()`-Methode die folgenden Funktionen aufgerufen. Die Methode `StarInHRD.stage2()` wird aufgerufen, der Wert von *animationScroll* wird auf den aktuellen m_x Wert gesetzt[6], über zwei if-Anweisungen wird schliesslich geprüft, ob die rechte oder linke Maustaste gedrückt wurde. Bei der linken Maustaste wird der Wert von *highLightStage* auf *true* gesetzt und danach die Methode `StarInHRD.highlightStage()` aufgerufen. Bei der rechten Maustaste wird die Variable *InfoBox.boxID* auf den Wert der ausgewählten Phase gesetzt und danach die Methode `StarInHRD.showInfoBox()` aufgerufen. Damit beim Lösen der Maustaste, die aufgerufene Aktion zurückgesetzt wird, wird beim Lösen einer Maustaste im Allgemeinen *highLightStage* auf den Wert *false* gesetzt, die Methode `StarInHRD.highlightStage()` aufgerufen und schliesslich *showInfoBox* auf den Wert *true* gesetzt:

```
class Stage2 implements MouseListener {

    public void mousePressed(MouseEvent arg0) {

        starHRD.stage2();
        animationScroll.setValue( StarInHRD.m_star.getX() );

        if (SwingUtilities.isLeftMouseButton(arg0)) {
            starHRD.highLightStage = true;
            starHRD.highlightStage();
        }

        if (SwingUtilities.isRightMouseButton(arg0)) {
            InfoBox.boxID = 2;
            starHRD.showInfoBox();
        }
    }

    public void mouseReleased(MouseEvent arg0) {
        starHRD.highLightStage = false;
        starHRD.highlightStage();
        StarInHRD.showInfoBox = false;
    }

}
```

Massenauswahlklassen

Mit Hilfe dieser Klassen werden die Methoden aufgerufen, die notwendig sind um die Datenwerte neu zu referenzieren. Der Aufruf geschieht über die entsprechenden Massenauswahl-Schaltfläche:

```
JToggleButton m0010Button = new JToggleButton("1.0 M");
.
m0010Button.addMouseListener(new SetM0010());
```

Alle Massenauswahlklassen funktionieren nach dem gleichen Prinzip. Sie hören in gleicher Weise wie die Phasenauswahlklassen auf einen `MouseListener`. Das einzige Ereignis, dem eine Funktion zugeordnet wurde, ist das Drücken der linken Maustaste. Wird diese über der entsprechenden Schaltfläche betätigt, wird zunächst die Methode `StarInHRD.setValue()` aufgerufen. Mit Hilfe

[6]Da dieser Schritt nach dem Aufrufen der Methode `stage2()` geschieht, wird immer der Anfangspunkt der Phase angesteuert.

der `setText`-Methode der Phasenauswahl-Schaltflächen, werden deren Bezeichnungen aktualisiert:

```
public void mousePressed(MouseEvent arg0) {
  if (SwingUtilities.isLeftMouseButton(arg0)) {
    starHRD.m0010();
    animationScroll.setValue(0);
    stageButton0.setText(Years.stageName(StarInHRD.m_star.m_M)[0]);
    stageButton1.setText(Years.stageName(StarInHRD.m_star.m_M)[1]);
    stageButton2.setText(Years.stageName(StarInHRD.m_star.m_M)[2]);
    stageButton3.setText(Years.stageName(StarInHRD.m_star.m_M)[3]);
    stageButton4.setText(Years.stageName(StarInHRD.m_star.m_M)[4]);
    stageButton5.setText(Years.stageName(StarInHRD.m_star.m_M)[5]);
    stageButton6.setText(Years.stageName(StarInHRD.m_star.m_M)[6]);
    stageButton7.setText(Years.stageName(StarInHRD.m_star.m_M)[7]);
    stageButton8.setText(Years.stageName(StarInHRD.m_star.m_M)[8]);
    stageButton9.setText(Years.stageName(StarInHRD.m_star.m_M)[9]);
    stageButton10.setText(Years.stageName(StarInHRD.m_star.m_M)[10]);
  }
}
```

Funktionsklassen

Für alle Funktionsauswahlklassen gilt, dass diese lediglich eine entsprechende Methode der Klasse `StarInHRD` über einen `ActionListener` aufrufen. So ruft zum Beispiel die innere Klasse `SetHayashi` die entsprechende Methode `showHayashi()` auf:

```
class SetHayashi implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        starHRD.showHayashi();
    }
}
```

4.5.4 Die Methoden von `StarInHRD`

Die Methoden `startApp()`, `stopApp()` & `resetApp()`

Diese Methoden starten beziehungsweise stoppen den Timer m_{timer} mit Hilfe der Standard-Methoden der `Timer`-Klasse `start()` bzw. `stop()`.

Die Methode zum Zurücksetzen `resetApp()` setzt drüber hinaus, die Werte m_x und m_y auf Null und setzt die Freeze-Funktion über die zugehörige boolsche Variable zurück.

Die Methode `runApp()`

Diese Methode startet unter bestimmten Vorraussetzungen den Timer erneut. Die erste Bedingung lautet:

```
m_star.m_x > 0 && m_star.m_x < Years.length(m_star.m_M)
```

Ist diese nicht erfüllt wird der Timer wie bekannt gestartet. m_x und m_y werden auf 0 gesetzt.

Andernfalls wird zunächst geprüft, ob der Timer läuft. Wenn ja, wird er gestoppt. Wenn nein, wird er gestartet.

Die Methoden `stepFWD()` & `setpRWD()`

Mit diesen Methoden kann der Benutzer um einen Datenpunkt vor bzw. zurückspringen. Beim Ausführen der Methoden werden m_x und m_y um eins erhöht bzw. erniedrigt. Dies geschieht nur nach Abfrage von Bedingungen, die vermeiden, dass man ausserhalb des Wertebereiches gelangt:

stepFWD:

```
if (m_star.m_x >= 0 || m_star.m_x <= Years.length(m_star.m_M)) {
    m_star.m_x = m_star.m_x + 1;
    m_star.m_y = m_star.m_y + 1;
}
```

stepRWD:

```
if (m_star.m_x > 0) {
    m_star.m_x = m_star.m_x - 1;
    m_star.m_y = m_star.m_y - 1;
}
```

Die Methoden zur Auswahl der stellaren Entwicklungsphase

Mit der Methode `repaint()` wird hier zunächst die Komplette Diagramebene neu gezeichnet und anschliessend über die Phase q die Methode `setStage(`q`)` der Klasse `SelectStage` aufgerufen.

```
public void stage2() {
 repaint();
 SelectStage.setStage(2);
}
```

Die Klasse `SelectStage` stellt die Methode `setStage(`s`)` zur Verfügung, in der zunächst die Hilfsvariable $u = s - 1$ definiert wird. Mit Hilfe einer for-Schleife über k werden alle Datenpunkte durchlaufen. In einer if-Anweisung wird danach abgefragt, ob der k-te Datenpunkt grösser oder gleich dem p-ten Datenpunkt, dem letzten der Phase ist. Ist dies der Fall, werden m_x und m_y auf den Wert k gesetzt. Danach wird die Animation mit `StarInHRD.m_timer.stop()` gestoppt und die for-schleife abgebrochen:

```
public static int setStage(int s) {
  int u = s-1;
  for (int k=0; k < Years.length(StarInHRD.m_star.m_M); k++){
    if (
    Years.value(StarInHRD.m_star.m_M)[k]
    >= Years.stage(StarInHRD.m_star.m_M)[u]){

    StarInHRD.m_star.m_x = k;
    StarInHRD.m_star.m_y = k;
    StarInHRD.m_timer.stop();
    break;
    }
  }
  return p;
}
```

Die Methoden zur Auswahl der Anfangsmasse

Die Methoden zur Auswahl der Anfangsmasse haben die Syntax `m`μ`()`, wobei μ die aus 4.1.1 bekannte 4-stellige Zahl ist. Zunächst wird der Timer gestoppt. Danach wird auf m_star neues

`Star(0,0,M)` Objekt mit dem Massenwert M erzeugt. Schliesslich wird noch der aktuelle *delay*-Wert des Timers mit `getDelay()` abgefragt, damit der aktuelle Wert übernommen wird. Danach wird die Diagrammebene wie üblich komplett neu gezeichnet:

```
public void m0010() {
  m_timer.stop();
  m_star = new Star(0, 0, 1.0);
  m_timer.getDelay();
  repaint();
}
```

Die Methoden zur Auswahl der javaHRD-Features

Zur Auswahl der Features des Applets werden Methoden verwendet, bei denen es in gleicher Form von einer boolschen Variable abhängt, ob ein Feature angezeigt oder aktiv ist. Der Anfangswert der zum Feature gehörenden boolschen Variable wird in `HRDPanel` gesetzt und bei einem `ActionEvent` der zugehörigen Auswahlbox neu gesetzt. War er vorher $true$, wird er beim Eintreten des Ereignisses auf $false$ gesetzt und umgekehrt. Als Beispiel sei die Methode `showHayashi()` angegeben, die die Hayashi Linie zeichnet:

```
public void showHayashi() {
  if (showHayashi) {
    showHayashi = false;
  } else {
    showHayashi = true;
  }
    repaint();
}
```

Kapitel 5

Didaktische Aspekte

Befasst man sich mit der Untersuchung von Sternspektren, erhält man einen Einblick in das physikalische und chemische Wesen stellarer und kosmischer Objekte. Erst aber die Idee, spektrale Eigenschaften und Helligkeiten der Sterne kombiniert zu betrachten, führte die Astronomie über das Farbenhelligkeitsdiagramm zum Hertzsprung-Russell-Diagramm. Als zentrales Diagramm zur Darstellung von Sternentwicklung ist das Verständnis aller Erkenntnisse und Ergebnisse, welche dieses Diagramm vermittelt, von zentraler Bedeutung für das Verständnis der Entwicklung stellarer Objekte (siehe [46]).

5.1 Interaktivität

Ein wesentlicher didaktischer Aspekt der vorliegenden Diplomarbeit und damit des hier programmierten Java-Applets liegt in der dynamischen Präsentation der Inhalte des HRD. Eine kompakte, übersichtliche und anschauliche Darstellung des Diagramms fördert im Verbund mit einer interaktiven Bedienoberfläche das schnelle Verständnis der präsentierten Inhalte.

Das javaHRD-Programm ist als Multimedia-Komponente Bestandteil des Lernsystems innerhalb dessen es präsentiert wird. Ein wichtiger Aspekt bei der didaktischen Bewertung von multimedialer Komponenten ist das Interaktivitätsniveau. Je höher der Grad der Interaktivität ist, desto didaktisch wertvoller lässt sich eine Multimedia-Komponente einstufen. Im Sinne der von R. Schulmeister beschriebenen Taxonomie der Interaktivität multimedialer Komponenten, nimmt javaHRD, als Java-Applet, eine Stufe ein, in der Inhalte nicht komplett vorgefertigt, sondern auf Anforderung durch den Benutzer generiert werden (vgl. [47]). Sowohl Parameter als auch Datengrundlage sind variabel und können ausgewählt werden.

Je nach Betrachtungsweise beschränkt sich dabei der inhaltliche Einfluss auf die Variation von voreingestellten Parametern, wie der Anfangsmasse, oder frei wählbaren Parametern, wie der Animationsgeschwindigkeit. Betrachtet man einerseits nun alle im Java-Code implementierten Entwicklungswege als Datengrundlage, so ist diese nicht variabel, sondern wurde im Vorfeld festgelegt. Andererseits können die Daten eines jeden Entwicklungswegs für sich, als Datengrundlage bezeichnet werden. In diesem Sinne kann von Datenvariation gesprochen werden, und die Merkmale der nach der genannten Taxonomie vierten Interaktivitäts-Stufe sind somit erfüllt.

Das javaHRD-Programm unterstützt also explorative Lernaktivitäten (vgl. [48]), bei denen der Inhaltsbereich von Lehrenden oder Experten zusammengestellt und präsentiert wird, die schliesslich von Lernenden untersucht werden können. Dies ist immer dann von Bedeutung, wenn eine hohes Mass an Abstraktion und Komplexität des Lerninhalts besteht, der nicht ohne weiteres vom Benutzer vorrausgesetzt werden kann. Naive Sichtweisen und Fehlinterpretationen gilt es zu vermeiden. Die Qualität der Darstellung bildet dabei letzten Endes das Fundament für die korrekte Interpretation der zu vermittelnden Lerninhalte.

Unter diesem Gesichtspunkt lässt sich ein statisches HRD, wie jedes statische Diagramm, in der niedrigsten Interaktivitätsstufe einordnen. Bei gleichbleibenden Inhalt übernimmt es lediglich

eine illustrative Darstellung der Zusammenhänge. Dies wird insbesondere dann problematisch, wenn sich illustrierte Inhalte überlappen oder die Illustrationsfläche überladen wird. So lassen sich beispielsweise in Abbildung 1.1 kaum auf den ersten Blick AGB- oder Rote-Riesen-Phasen von Sternen in den unteren Massebereichen zuordnen. Ein solche Darstellung kommt nicht ohne erläuternden Text oder kompetenten Kommentar aus.

Der wesentliche Vorteil von javaHRD liegt also in der dynamischen, animierten Darstellung, die eine viel höhere Interaktivitäts-Stufe erreicht, als ein statisches, abgedrucktes Diagramm. Das Diagramm bietet nicht nur die individuellen Betrachtungsmöglichkeit der einzelnen Entwicklungswege und deren Vergleich, sondern auch die Möglichkeit der differenzierten Betrachtungsweise der einzelnen Entwicklungsphasen. Für die Interpration nützliche Hilfen, wie die Darstellung von entwicklungsrelevanten Sequenzen oder ausgezeichneten Bereichen, unterstreichen den didaktischen Nutzen genauso, wie die Integration bestimmter Simulationen und deren Zugriff darauf.

5.2 Darstellung der zeitlichen Entwicklung

Ein wichtiger Aspekt der Sternentwicklung und damit der Umsetzung zugehöriger Darstellungen, ist der zeitliche. In einem statischen Diagramm lässt sich Zeit weniger gut vermitteln. Die Veränderungen der Sterne vollziehen sich meistens in langen Zeiträumen, sind gelegentlich aber auch schnell. Solche Unterschiede graphisch zu vermitteln ist in einer statischen Umsetzung der Darstellung schwer.

javaHRD erlaubt die Darstellung der zeitlichen Entwicklung durch die Eigenschaften von interaktiver Animation. Die Geschwindigkeit der Entwicklung wird durch den Timer und die Abfolge der Animationsschritte bestimmt. Durch den internen Parameter der Zeitanimation, den delay-Wert, wird die zeitliche Abfolge der Animationsschritte gesetzt. Dadurch erhält der Betrachter ein Zeitgefühl für das Zeitintervall zwischen den Animationsschritten.

Ein weiterer Aspekt der zeitlichen Darstellung ist die Anzahl der zugrundeliegenden Wertepaare. So wird die Animation (bei gleichem delay-Wert) bei einem Entwicklungsweg mit 1000 Datenpunkten über einen längeren Zeitraum andauern, als bei einem Entwicklungsweg der nur 50 Datenpunkte umfasst. Die Gesamtanzahl der Datenpunkte hängt dann natürlich mit dem zeitlichen Eindruck der gesamten Sternlebensdauer zusammen.

Darüber hinaus ist die Betrachtung der zeitlichen Abstände der zugrundeliegenden Datenpunkte wesentlich für die vermittelte Darstellung. Liegen die Daten in zeitlich äquidistanten Werten vor, so sind die Zeitintervalle zwischen den Animationsschritten immer gleich und es wird intuitiv vermittelt, dass der Stern sich langsam entwickelt, wenn viele Datenpunkte pro Intervall vorliegen. In der Hauptreihe erhält der Betrachter in diesem Fall zu Recht den Eindruck, dass die Entwicklungsphase des Sterns länger andauert, als beispielsweise während der AGB-Phase, was eine gute Simulation der Realität ist. Dies ist insbesondere bei den Entwicklungswegen des DDR-Pakets der Fall.

Bei allen anderen Paketen stehen nicht äquidistante Datenpunkte zur Verfügung, die aber den Vorteil haben, alle relevanten Entwicklungsphasen in der Darstellung abzudecken. Um einen ähnlichen Effekt des Zeiteindrucks zu vermitteln, können hier zwei Lösungen des Zeitproblems gewählt werden. Einerseits die Interpolation der Datenpunkte und damit die Erhöhung der Datendichte in den gewünschten Phasen. Andererseits die aktive Steuerung des Animationsparameters in den entsprechenden Phasen. Beide Darstellungswege haben den gleichen positiven Effekt der Simulation (siehe Kap. 4.1.1).

5.3 Didaktischer Mehrwert

Worin besteht also der didaktische Mehrwert des javaHRD-Programms? Die Antwort auf diese Frage liegt auf der Hand, sobald man unter Berücksichtigung der genannten Hauptaspekte, einen Vergleich mit statischen, gedruckten Diagrammen vornimmt.

Gemessen an der Interaktivität ist javaHRD höher Einzustufen als ein statisches Diagramm. Zieht man die zeitliche Darstellung in Betracht, so gibt es die zuvor angesprochenen Möglichkeiten, die Zeit zu Berücksichtigen und in die Simulation einfliessen zu lassen. Allein diese beiden Aspekte geben javaHRD einen erheblichen, didaktischen Mehrwert gegenüber der statischen Darstellung.

Weitere Vorteile sind die flexiblen Einsatzmöglichkeiten der Applikation und die intuitive Steuerung. Darüberhinaus sei nochmals auf die bereits erwähnte, kompakte und übersichtliche Darstellung der Lehrinhalte verwiesen. Die differenzierte Darstellung erhöht die Lernmotivation des Betrachters. Selbst der Laie lernt in spielerischer Weise die Entwicklungseigenschaften von Sternen kennen, und der Student der Astronomie kann erlangtes Wissen durch Vergleich überprüfen.

Zuletzt sei der schnelle Zugang zum präsentierten Lehrangebot zu erwähnen. Der Benutzer muss beispielsweise nicht erst die Berechnung von Entwicklungswerten abwarten oder eine Programminstallation vornehmen, um das Ergebnis betrachten zu können. Das komplette javaHRD-Programm hat darüberhinaus eine Grösse von $\sim$1,3 MB und ist dadurch schnell über das Internet verfügbar. Liegt die Applikation lokal vor, wird sie bei Aufruf sofort angezeigt.

Alles in allem liefert javaHRD also einen didaktisch wertvollen Beitrag zum Verständnis von Sternentwicklung.

Kapitel 6

Zusammenfassung und Ausblick

Das in dieser Diplomarbeit beschriebene Java-Applet (javaHRD) zur Darstellung von Sternentwicklung durch die Animation des Hertzsprung-Russell-Diagramms wurde im Rahmen dieser Arbeit entworfen und neu entwickelt. Die Programmierung der Anwendung mit der Programmiersprache Java erfolgte mit Hilfe der plattformunabhängigen Entwicklungsumgebung *Eclipse*. Entwurf und Konzeption wurden zu Beginn der Programmierarbeit skizziert und modeliert, wobei im Mittelpunkt die didaktischen Ziele der Anwendung lagen. Design und Layout haben sich im Laufe der Programmierarbeit ergeben und wurden den Bedürfnissen des Anwenders angepasst.

Als erste Datengrundlage wurden Entwicklungswege im Digital Demo Room der University of Illinois berechnet und in Java-Daten-Pakete zusammengefasst. Nachdem die grundlegenden Funktionen programmiert waren, wurden zusätzliche Daten von Sternmodellen herangezogen und eingebaut. Zusätzliche Funktionen, wie zum Beispiel die Radius-Simulation wurden später implementiert. Während der gesamten Entwicklungszeit war eine aktuelle Version des Programms bereits online zugänglich. Dabei wurde der Zugriff auf ältere Versionen erhalten, so dass stets alle Entwicklungsstufen von javaHRD zugänglich sind. Dadurch wurde die Entwicklungsgeschichte der Programmierung protokolliert. Die entwicklungsbegleitende Präsentation des Applets hat Benutzern ermöglicht, die Anwendung bereits zu testen. So konnte die Qualität der Bedienung und eine möglichst intuitive Steuerung gesichert werden.

Ein zentraler Punkt bei der Weiterentwicklung von javaHRD, der zwar vorbereitet wurde, aber aus Zeitgründen nicht eingebaut werden konnte, ist die Auswahlmöglichkeit von Sternentwicklungswegen unterschiedlicher Metallizität. Dadurch würde der didaktische Mehrwert nochmals gesteigert. Weiterhin sollte die Verbesserung einzelner Funktionen, wie zum Beispiel der Sternfarben-Funktion oder der zeitlichen Darstellung umgesetzt werden. Zusätzliche Aspekte und Details der Sternentwicklung, wie zum Bespiel Massenverluste oder die chemische Veränderung könnten darüber hinaus in die Darstellung aufgenommen werden.

Programmiertechnisch wäre ein automatisches Auslesen der Datengrundlage wünschenswert. Dies wurde zum Teil vorbereitet, konnte aber aus Zeitgründen nicht vollständig umgesetzt werden. Durch automatisiertes Auslesen der Daten würde neben dem erheblichen Zeitgewinn bei der Zusammenstellung neuer Datenpakete, sowohl die Aktualisierung der vorhadenen Datenpakete erleichtert, als auch die Eingabe eigener Daten für den Benutzer ermöglicht. Um die Übersichtlichkeit mancher Bereiche des programmierten HRD zu erhöhen, wäre beispielsweise eine dynamische Zoom-Funktion interessant, die schliesslich auch dynamische Anpassung von javaHRD an unterschiedliche Ausgabemedien ermöglicht. Dies würde neben der Plattformunabhängigkeit zusätzlich eine Unabhängigkeit vom verwendeten Ausgabegerät und dessen Displaygrösse bedeuten. Schliesslich wäre die komplette Steuerung der Anwendung über die Tastatur wünschenswert.

javaHRD soll durch die dynamische und interaktive Darstellung des HRD als Referenzdiagramm der Astrophysik, zum einen die akademische Lehre begleiten, zum anderen den Einstieg in das Verständnis von Sternentwicklung für jedermann erleichtern.

Anhang A

javaHRD Quelldateien (Auszug)

In diesem Anhang werden einige Auszüge aus dem Quellcode angegeben. Der gesamte Quellcode steht unter *http://www.astro.uni-bonn.de/∼javahrd/source* zur Ansicht und zum Download bereit.

A.1 JavaHRD.java

```
import java.awt.Dimension;
import java.awt.Toolkit;

import javax.swing.*;

public class JavaHRD extends JApplet {

    //applet constructor
    public JavaHRD() {
       this.add(new HRDPanel());
    }

    //application part (main)
    public static void main(String[] args) {
    Dimension d = Toolkit.getDefaultToolkit().getScreenSize();

       JFrame win = new JFrame("javaHRD");
       win.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       win.setContentPane(new HRDPanel());
       win.pack();
       win.setVisible(true);

       int w = win.getSize().width;
       int h = win.getSize().height;

       win.setLocation(d.width/2-w/2,d.height/2-h/2);
       win.addWindowListener(new WindowWatch());

       System.out.println(win.getSize());

    }

}
```

A.2 sVar.java

```
public interface sVar {
    //Transformationsvariablen
    final int xPos = 1020;
    final int yPos = 330;
    final int xSt = 150;
    final int ySt = 40;

    //Diagrammposition
    final int xAchsenPosY = 550;
    final int yAchsenPosX = 70;

    //Diagramm Dimension
    final int HRD_WIDTH = 675;
    final int HRD_HEIGHT = 600;

    //JPanel Dimension
    final int PANEL_WIDTH = 1000;
    final int PANEL_HEIGHT = 700;

    final int WESTPANEL_WIDTH = 80;

    //Colors
    final String CONTROL = "#e6ecff"; // light blue
    final String PANEL = "#ffeec6"; // yellow
}
```

A.3 Track.java

```
import java.awt.Color;
import java.awt.Graphics;
import java.io.IOException;

public class Track {

  double m_M;
  public static boolean connectRule;

  Track(double M) {
    m_M = StarInHRD.m_star.m_M;
  }

  public void draw(Graphics g) throws IOException {
    g.setColor(Color.decode("#000000"));
    //draw points
    for (int i = 0; i < Years.Length(m_M); i++){
        int x_i = getX(i,m_M);
        int y_i = getY(i,m_M);
        g.fillOval(x_i, y_i, 1, 1);
    }
    //draw connect-lines
    for (int i = 0; i < Years.Length(m_M)-1; i++){
        int j = i+1;
        int x_i = getX(i,m_M);
        int y_i = getY(i,m_M);
        int x_j = getX(j,m_M);
        int y_j = getY(j,m_M);
        double dT = LogT.value(m_M)[i] - LogT.value(m_M)[j];
        double dL = LogL.value(m_M)[i] - LogL.value(m_M)[j];
          if (connectRule(dT,dL)) {
              g.drawLine(x_i,y_i,x_j,y_j);
          }

    }
  }

  public static boolean connectRule (double a, double b) {
    boolean rule = Math.abs(a) < 0.5 || Math.abs(b) < 0.8 && StarInHRD.m_star.m_M < 50;
    if(rule) {
      connectRule = true;
    } else {
         connectRule = false;
    }
    return connectRule;
  }

  public int getX (int i, double M) {
    int x_i = (int)(sVar.xPos - sVar.xSt * LogT.value(m_M)[i]);
    return x_i;
  }

  public int getY (int i, double M) {
    int y_i = (int)(sVar.yPos - sVar.ySt * LogL.value(m_M)[i]);
    return y_i;
  }
}
```

A.4 Star.java

```
import java.awt.*;
import java.text.DecimalFormat;


public class Star {

  int m_x;
  int m_y;
  double m_M;
  double Z; /** TODO: metallizität einbauen */

  public Color color;
  public int ovalSize = 6;

  public Star(int x, int y, double M) {
    m_x = x;
    m_y = y;
    m_M = M;
  }

  public void move() {
    if (m_x < Years.value.length) {
      m_x = m_x+1;
      m_y = m_y+1;
    }
  }

  public int getX() {
    return m_x;
  }

  public int getY() {
    return m_y;
  }

  public void setPosition(int x, int y) {
    m_x = x;
    m_y = y;
  }

  public void draw(Graphics g) {
    StarColor sc = new StarColor(LogT.value(m_M)[m_x]);
    color = sc.starColor;

    try {

      int dPX = AnimationVariable.x(m_x);
      int dPY = AnimationVariable.y(m_y);

      double vLeucht = Math.pow(10,LogL.value(StarInHRD.m_star.m_M)[StarInHRD.m_star.m_x]);
      double vTemp = Math.pow(10,LogT.value(StarInHRD.m_star.m_M)[StarInHRD.m_star.m_x]);

      DecimalFormat vT = new DecimalFormat("###,##0");
      String Temp = vT.format(vTemp);
      double vTime = Years.value(StarInHRD.m_star.m_M)[StarInHRD.m_star.m_x];
      String Time;
```

```
      if (StarInHRD.m_star.m_M < 4.0) {
         DecimalFormat vt = new DecimalFormat("##,##0");
         Time = vt.format(vTime);
      } else {
         DecimalFormat vt = new DecimalFormat("##,##0.##");
         Time = vt.format(vTime);
        }

      int starRadius = (int)( Math.sqrt(vLeucht)/(vTemp*vTemp) * 5E7 );

      if (starRadius > ovalSize && Radius.show) {
        g.setColor(color);
        g.fillOval(dPX-starRadius/2+ovalSize/2,dPY-starRadius/2+ovalSize/2,starRadius,starRadius);
        g.setColor(Color.decode("#000000"));
        g.drawOval(dPX-starRadius/2+ovalSize/2,dPY-starRadius/2+ovalSize/2,starRadius,starRadius);
        g.setColor(Color.decode("#eeeeef"));
        g.fillRect(sVar.yAchsenPosX+40-5,sVar.xAchsenPosY-60,120,40);
        g.setColor(Color.black);
        g.drawRect(sVar.yAchsenPosX+40-5,sVar.xAchsenPosY-60,120,40);
        g.setFont(new Font("Arial", Font.BOLD, 14));
        g.drawString("T: " + Temp + "° K", sVar.yAchsenPosX+40, sVar.xAchsenPosY-45);
        if (false) {
          g.drawString("t: " + Time + " Gyrs",  sVar.yAchsenPosX+42,sVar.xAchsenPosY-25);
        } else {
            g.drawString("t: " + Time + " Myrs",  sVar.yAchsenPosX+42,sVar.xAchsenPosY-25);
          }

        Track m_track = new Track(m_M);
        m_track.draw(g);
        if (StarInHRD.showFreezedTrack) {
          StarInHRD.m_freezeTrack.draw(g);
        }

        g.setFont(new Font("Arial", Font.PLAIN, 12));
        g.setColor(color);

        if (starRadius >ovalSize && Radius.show) {
          g.setColor(Color.RED);
          g.fillOval(dPX+1,dPY+1,4,4);
          g.drawOval(dPX+1,dPY+1,4,4);
        } else {
            g.fillOval(dPX,dPY,ovalSize,ovalSize);
          }
        g.setColor(Color.black);
        if (starRadius >ovalSize && Radius.show) {
        } else {
            g.drawOval(dPX,dPY,ovalSize,ovalSize);
          }

        } catch (Exception e) {
            StarInHRD.m_timer.stop();
            e.printStackTrace();
          }

      }

}
```

Abbildungsverzeichnis

1.1 Hertzsprung-Russell-Diagramm nach Maeder und Meynet, 1989 2

2.1 Skizze der Morphologie I 4
2.2 Skizze der Morphologie II 7
2.3 Skizze der Bereiche im HRD mit stabiler Fusion 9

3.1 Beispiel eines UML Klassendiagramms 14
3.2 Beispiel eines UML Objektdiagramms 14
3.3 Beispiel eines UML Komponentendiagramms 15
3.4 Beispiel eines UML Anwendungsfalldiagramms 15
3.5 Beispiel eines UML Sequenzdiagramms 15
3.6 Beispiel eines UML Zustandsdiagramms 16
3.7 Beispiel eines UML Aktivitätsdiagramms 17
3.8 Multiview-Ansicht des Trifid Nebels im Aladin Sky Atlas 18

4.1 Entwicklungsweg aus dem DDR-Paket 23
4.2 korrigierter Entwicklungsweg aus dem DDR-Paket 23
4.3 Skizze zur Darstellung der Umreferenzierung 25
4.4 UML Klassendiagramm der wichtigsten javaHRD-Klassen 26
4.5 Objektdiagramm der wichtigsten javaHRD-Objekte. 27
4.6 Zentrale javaHRD-Anwendungsfälle 28
4.7 Steuerungs- und Diagrammebene von javaHRD 29
4.8 Layout von javaHRD 31
4.9 Skizze zur Klasse `sVar` 32
4.10 Zeichnen der Sternpunktverbindungslinien ohne `Track.connectRule()` 36
4.11 Zeichnen der Sternpunktverbindungslinien mit `Track.connectRule()` 36
4.12 zeitlicher Fortschrittsbalken, Animationsbildlaufleiste und -schieberegler 39
4.13 Radiussimulation und Infobox 42

Literaturverzeichnis

[1] A. Unsöld and B. Baschek. *The New Cosmos*. Springer-Verlag, Berlin, etc., 4th edition, 1991.

[2] R. Kippenhahn and A. Weigert. *Stellar Structure and Evolution*. Springer-Verlag, 1990.

[3] K.S. de Boer and W. Seggewiss. Stars and stellar evolution, 2005.

[4] A. Maeder and G. Meynet. Grids of evolutionary models from 0.85-m. to 120-m. - observational tests and the mass limits. *Astron. Astrophys.*, 210:155–173, 1989.

[5] H. Gärtner. *Er durchbrach die Schranken des Himmels: das Leben des William Herschel*. Ed. Leipzig, 1996.

[6] S. Cassisi. Stellar evolutionary models: uncertainties and systematics, 2005.

[7] M.J.P.F.G. Monteiro et.al. Esta/corot task 1 - models comparison - preliminary results, 2006.

[8] M.J.P.F.G. Monteiro, Y. Lebreton, J. Montalban, J. Christensen-Dalsgaard, M. Castro, S. Degl'Innocenti, A. Moya, I. W. Roxburgh, R. Scuflaire, A. Baglin, M. S. Cunha, P. Eggenberger, J. Fernandes, M. J. Goupil, A. Hui-Bon-Hoa, M. Marconi, J. P. Marques, E. Michel, A. Miglio, P. Morel, B. Pichon, P. G. Prada Moroni, J. Provost, A. Ruoppo, J. C. Suarez, M. Suran, and T. C. Teixeira. Report on the corot evolution and seismic tools activity, 2006.

[9] J. Christensen-Dalsgaard. Astec - aarhus stellar evolution code, 2005.

[10] P. Morel et. al. Cesam - code d'Évolution stellaire adaptatif et modulaire, 2005.

[11] R. Scuflaire and BAG (Belgian Asteroseismology Group). Cles - code liegeois d'evolution stellaire, 2005.

[12] M. Castro. Tgec - toulouse geneva evolution code, 2005.

[13] S. Degl'Innocenti. Franec - pisa evolution code, 2006.

[14] I. Roxburgh. Starox - roxburgh's stellar evolution code, 2006.

[15] A. Weiss. Garstec - the garching stellar evolution code, 2006.

[16] N. Mowlavi. Geneva grids of stellar evolution models, 2004.

[17] G. Schaller, D. Schaerer, G. Meynet, and A. Maeder. New grids of stellar models from 0.8 to 120 msolar at z=0.020 and z=0.001. *Astron. Astrophys.*, 96:269–331, 1992.

[18] C. A. Iglesias and F. J. Rogers. Updated opal opacities. *Astrophysical Journal*, 464:943, 1996.

[19] Georges Meynet and Andre Maeder. Stellar evolution with rotation. i. the computational method and the inhibiting effect of the μ-gradient. *Astron. Astrophys.*, 321:465–476, 1997.

[20] Georges Meynet and Andre Maeder. Stellar evolution with rotation v: Changes in all the outputs of massive star models, 2000.

[21] G. Meynet and A. Maeder. Stellar evolution with rotation x: Wolf-rayet star populations at solar metallicity. *ASTRON.ASTROPHYS.*, 404:975, 2003.

[22] R. Hirschi, G. Meynet, and A. Maeder. Stellar evolution with rotation xii: Pre-supernova models. *ASTRON.ASTROPHYS.*, 425:649, 2004.

[23] P. Eggenberger, A. Maeder, and G. Meynet. Stellar evolution with rotation and magnetic fields iv: The solar rotation profile, 2005.

[24] A. Bressan, F. Fagotto, and L. Girardi. Padova database of stellar evolutionary tracks and isochrones, 2006.

[25] Sukyoung K. Yi, Yong Cheol Kim, and Pierre Demarque. The y^2 stellar evolutionary tracks, 2002.

[26] S. Yi, P. Demarque, Y. C. Kim, Y. W. Lee, C. Ree, Th Lejeune, and S. Barnes. Towards better age estimates for stellar populations: The y^2 isochrones for solar mixture. *The Astrophysical Journal*, 136:417, 2001.

[27] Yong Cheol Kim, Pierre Demarque, Sukyoung K. Yi, and David R. Alexander. The yy isochrones for alpha-element enhanced mixtures. *The Astrophysical Journal*, 143:499, 2002.

[28] L. G. Althaus et al. Stellar evolution and pulsations research group - evolutionary tracks, 2006.

[29] J. R. Hurley, O.R. Pols, and C.A. Tout. Comprehensive analytic formulae for stellar evolution as a function of mass and metallicity. *Mon. not. R. Astron. Soc.*, 315, 2000.

[30] C.E. Hansen and J.B. Simon. Visualization of multiple star evolution. 2002.

[31] Martin Fowler. *UML distilled*. Addison-Wesley, 2nd edition, 2000.

[32] Dan Pilone. *UML kurz & gut*. O'Reilly, 1st edition, 2004.

[33] wikipedia.de: Unified modeling language, 2006.

[34] Sun Microsystems Inc. Api specification for the java 2 platform standard edition 5.0, 2006.

[35] wikipedia.de: Eclipse (ide), 2006.

[36] Steve Holzner. *Eclipse*. O'Reilly, 1st edition, 2004.

[37] Christian Ullenboom. *Java ist auch eine Insel*. Galileo Computing, 4th edition, 2005.

[38] G. Bothun, D. Caley, S. Russel, and A. McGrew. Virtual laboratory of the universtiy of oregon, department of physics, 2006.

[39] R. Freedman and W.J. Kaufmann. *Universe*. W. H. Freeman, 2004.

[40] R. Freedman and W.J. Kaufmann. Universe online, 2006.

[41] W. Fendt. Java-applets zur physik, 2006.

[42] K. Shetline. Sky view cafe, 2003.

[43] F. Bonnarel and P. Fernique. The aladin sky atlas, 2006.

[44] L. Girardi, A. Bressan, G. Bertelli, and C. Chiosi. Evolutionary tracks and isochrones for low- and intermediate-mass stars: from 0.15 to 7 $m_\odot$, and from z=0.0004 to 0.03. *A AND AS*, 141:371, 2000.

[45] O. G. Benvenuto and L. G. Althaus. Grids of white dwarf evolutionary models with masses from m= 0.1 to 1.2 ms. 1998.

[46] D. B. Herrmann. *The history of astronomy from Herschel to Hertzsprung.* Cambridge: University Press, 1984, Geschichte der Astronomie von Herschel bis Hertzsprung (orig.t.), 1984.

[47] R. Schulmeister. Interaktivität in multimedia-anwendungen, 2005.

[48] Harvey Mellar, Joan Bliss, Richard Boohan, Jon Ogborn, and Chris Tompsett, editors. *Learning with artificial worlds: computer based modelling in the curriculum.* The Falmer Press, Bristol, PA, USA, 1994.