# Evaluierung der aspektorientierten Sprache ObjectTeams/Java zur Strukturverbesserung des Frameworks JHotDraw

## Diplomarbeit

**Christine Hering**
Matr.-Nr. 176785

**Technische Universität Berlin**
Fakultät IV – Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Softwaretechnik

Gutachter: Prof. Dr.-Ing. Stefan Jähnichen
2.Gutachter: Dr.-Ing. Stephan Herrmann

August 2004

## Eidesstattliche Erklärung

Die selbständige und eigenhändige Anfertigung versichere ich an Eides statt.

Berlin, den

Christine Hering

# Inhaltsverzeichnis

1 Einleitung..... 6
2 ObjectTeams..... 8
3 Frameworks..... 17
3.1 Definition..... 17
3.2 Klassifikation..... 18
3.3 Dokumentation..... 19
3.3.1 Beispiele..... 20
3.3.2 Recipes / Cookbooks..... 21
3.3.3 Design Pattern..... 21
3.4 JHotDraw..... 22
4 ObjectTeams-basierte Restrukturierung anhand des Beispiels der Programmfunktion „Laden und Speichern einer Zeichnung“..... 25
4.1 Benutzersicht..... 25
4.2 Objektorientierte Originalimplementierung..... 26
4.2.1 Grundstruktur..... 26
4.2.2 Das Standard-Speicherformat..... 28
4.3 Ansatzpunkte und Ideen für Strukturverbesserungen..... 30
4.3.1 Grundstruktur..... 31
4.3.2 Das Standard-Speicherformat..... 31
4.4 ObjectTeams-orientierte Implementierung..... 32
4.4.1 Grundstruktur..... 32
Erster Design-Ansatz..... 33
Zweiter Design-Ansatz..... 34
Dritter Design-Ansatz..... 36
Speicherformate..... 37
4.4.2 Das Standard-Speicherformat..... 39
Das Design..... 39
Aufgetretene Probleme und eingesetzte Workarounds..... 43
4.5 Verwendung der Framework-Implementierung bei der Erstellung einer Applikation..... 44
4.5.1 Festlegung des Funktionsumfangs..... 44
Objektorientierte Implementierung..... 45
ObjectTeams-orientierte Implementierung..... 45
4.5.2 Festlegung der zu unterstützenden Speicherformate..... 45
Objektorientierte Implementierung..... 45

ObjectTeams-orientierte Implementierung..... 46
4.5.3 Speicherbarmachen selbst definierter Klassen im Standardspeicherformat..... 46
Objektorientierte Implementierung..... 47
ObjectTeams-orientierte Implementierung..... 47
4.6 Fazit..... 52
4.6.1 Anwendbarkeit der ObjectTeams-Konzepte..... 53
4.6.2 Funktionsfähigkeit von Compiler und Laufzeitumgebung und sprachliche Schwächen..... 53
4.6.3 Restrukturierungsprozess..... 54
4.6.4 Spracherweiterungsvorschläge..... 55
4.6.5 Ist ObjectTeams bei der Implementierung komplexer Systeme hilfreich..... 55
Lokalität..... 55
Verständlichkeit und Instanziierbarkeit..... 55
4.6.6 Weitere Eigenschaften der ObjectTeams-orientierten Implementierung..... 56
4.7 Ausblick..... 57
4.7.1 Interne Strukturierung von Teams mit einer großen Anzahl von Rollenklassen..... 58
4.7.2 Speicherung von Rollenklassen..... 58
5 Aspekte der Team-Erzeugung und -Aktivierung..... 61
5.1 Starten JHotDraw-basierter Programme (Öffnen von Editorfenstern)..... 61
5.2 Bestimmung von Ort und Art der Team-Erzeugung und -Aktivierung..... 64
5.3 Konsequenzen der Team-Handhabungs-Verfahren..... 73
5.3.1 Funktionale Einschränkung durch erstes Team-Handhabungs-Verfahren..... 73
5.3.2 Erweiterung der Team-internen Strukturierungsmöglichkeiten durch zweites Team-Handhabungs-Verfahren..... 75
6 ObjectTeams-basierte Restrukturierung anhand des Beispiels der Programmfunktion „Rückgängig machen von Benutzeraktionen“ (Undo)..... 78
6.1 Benutzersicht..... 78
6.2 Begriffdefinition..... 78
6.3 Objektorientierte Originalimplementierung..... 79
6.4 Diskussion der objektorientierten Implementierung..... 83
6.5 Ansatzpunkte für die ObjectTeams-orientierte Umstrukturierung..... 84

6.6 ObjectTeams-orientierte Implementierung..... 84
6.7 Verwendung der Framework-Implementierung bei der Erstellung einer Applikation..... 106
6.7.1 Festlegen des Vorhandenseins der Undo-Programmfunktion.....106
Objektorientierte Implementierung.....106
ObjectTeams-orientierte Implementierung..... 107
6.7.2 Festlegen des Undo-Funktionsumfangs..... 107
Objektorientierten Implementierung.....107
ObjectTeams-orientierte Implementierung..... 108
6.7.3 „Rückgängigmachbar-Machen“ selbst definierter Benutzeraktionsklassen..... 109
Objektorientierte Implementierung.....109
ObjectTeams-orientierte Implementierung..... 109
6.8 Fazit..... 110
6.8.1 Anwendbarkeit der ObjectTeams-Konzepte..... 111
6.8.2 Funktionsfähigkeit von Compiler und Laufzeitumgebung und sprachliche Schwächen..... 111
6.8.3 Restrukturierungsprozess..... 111
6.8.4 Spracherweiterungsvorschäge..... 111
6.8.5 Ist Object Teams bei der Implementierung komplexer Systeme hilfreich..... 112
Lokalität..... 112
Verständlichkeit und Instanziierbarkeit..... 112
6.8.6 Weitere Eigenschaften der ObjectTeams-orientierten Implementierung..... 113
6.9 Ausblick..... 113
7 Fazit und Ausblick..... 115
Abbildungsverzeichnis..... 117
Literaturverzeichnis..... 119

# 1 Einleitung

Im Zentrum der vorliegenden Arbeit steht das neue aspektorientierte Modellierungs- und Programmierkonzept ObjectTeams. Es ist das Ergebnis mehrjähriger Forschungsaktivität am Institut für Softwaretechnik und Theoretische Informatik der Fakultät für Elektrotechnik und Informatik der Technischen Universität Berlin.

Bei der Verfolgung des Ziels, eine flexible Software zu entwickeln, die einfach gewartet, erweitert und an sich verändernde Anforderungen angepasst werden kann, stößt man ab einer gewissen Komplexität an die Grenzen der objektorientierten Modularisierungskonzepte. Da komplexe Software mittlerweile fast alle Lebensbereiche durchzieht, besteht ein gesteigertes Interesse an einer Lösung dieses Problems. Das führte dazu, das seit einigen Jahren rege Forschung im Bereich der Aspektorientierung betrieben wird. Das Produkt eines Zweiges dieser Forschungsaktivitäten ist ObjectTeams. Im Rahmen seiner Entwicklung wurde versucht, die Vorteile verschiedener aspektorientierter Ansätze einzubeziehen. ObjectTeams erweitert den objektorientierten Ansatz um die Möglichkeit, komplexes Verhalten in Form von Objekt-Kollaborationen in so genannten Teams zu kapseln. Ein Objekt übernimmt im Kontext eines bestimmten Teams eine bestimmte Rolle. Teams, und damit ein komplexes Verhalten, können einer Anwendung dynamisch hinzugefügt und mittels erweiterter Vererbungskonzepte spezialisiert werden.

Da ObjectTeams eine Erweiterung des objektorientierten Modellierungsansatzes darstellt, war es nicht notwendig, eine vollkommen neue Sprache zu entwerfen. Stattdessen können objektorientierte Programmiersprachen um die Konzepte von ObjectTeams erweitert werden. Zum jetzigen Zeitpunkt ist die Entwicklung der auf Java basierenden Sprache ObjectTeams/Java, des zugehörigen Compilers und der Laufzeitumgebung weit fortgeschritten. In Zusammenarbeit mit dem Fraunhofer Institut für Rechnerarchitektur und Softwaretechnik FIRST in Berlin wird an der Erweiterung der integrierten Entwicklungsumgebung Eclipse zum Einsatz mit ObjectTeams/Java gearbeitet. Über die während der Entwicklung von ObjectTeams entstandenen kleinen und mittelgroßen „Sandkasten“-Beispiele hinaus besteht noch kaum Erfahrung bei der Anwendung von ObjectTeams für komplexere Systeme.

Daher ist die der vorliegenden Arbeit zugrunde liegende Aufgabenstellung die Durchführung einer Fallstudie, in deren Rahmen ObjectTeams/Java bei der Modellierung und Implementierung eines komplexen Systems eingesetzt, seine Leistungsfähigkeit geprüft und mit der objektorientierter Konzepte, insbesondere der von Java, verglichen werden soll. Da ObjectTeams das Ziel verfolgt, die Modularität und Flexibilität von Software zu erhöhen, dient als Vergleichsobjekt ein bewährtes Framework, da bei diesem davon ausgegangen werden kann, dass es im Rahmen des in der Objektorientierung Möglichen sehr gut modularisiert und flexibel ist. Bei diesem Framework handelt es sich um JHotDraw, ein Framework zur Erstellung von Editoren für strukturierte Graphiken. Es wurde ursprünglich unter dem Namen HotDraw entwickelt, war Gegenstand verschiedener Design Studien, wurde mehrere Male (in verschiedenen Sprachen) reimplementiert und liegt mittlerweile in seiner Java-Variante JHotDraw in der Version 6.01b vor.

Die konkrete Aufgabe der vorliegenden Arbeit bestand darin, die vorhandene objektorienterte Framework-Implementierung zu untersuchen, dabei Programmfunktionalitäten zu identifizieren, deren Implementierung eine hohe Streuung über das Gesamtsystem und/oder eine starke Kopplung an die Implementierungsanteile anderer Programmfunktionalitäten aufweisen, diese Systemteile sofern möglich mit den Mitteln von ObjectTeams zu reimplementieren bzw. zu restrukturieren und die resultierende Implementierung mit der originalen bezüglich verschiedener Kriterien zu vergleichen. In diesem Rahmen sollten verschiedene Fragen beantwortet werden.

Die Hauptfragestellungen beziehen sich hierbei auf die Konzepte von ObjectTeams.

Ganz allgemein stellen sich die Fragen:
- (In welchem Umfang) sind die Konzepte von ObjectTeams bei der Implementierung komplexer Systeme anwendbar? (Grenzen)
- Kann die Modularität und Lokalität der Implementierung einzelner Programm-Funktionalitäten erhöht werden?
- Können ObjectTeams-orientierte Programme leichter erweitert und an sich verändernde Anforderungen angepasst werden?

Im speziellen bezogen auf Frameworks:
- Wird die Verständlichkeit des Frameworks für den Benutzer erhöht?
- Können Features gezielt adaptiert werden, ohne dass große Teile des Systems verstanden werden müssen?
- Ist die Instanziierung einer Anwendung auf der Grundlage der ObjectTeams-orientierten Framework-Version einfacher?

Zusammenfassend stellt sich also die Frage, ob ObjectTeams ein besseres Design ermöglicht und die Komplexität von Software besser handhabbar macht.

Neben der Überprüfung der Anwendbarkeit und der Zielerreichung der Konzepte dient die Fallstudie dazu, Erfahrungen mit der Modellierung in ObjectTeams zu sammeln, verschiedene Herangehensweisen zu erproben und einen diesbezüglichen Leitfaden zu erstellen. Darüber hinaus wird die Anwendbarkeit der Sprache ObjectTeams/Java und die Funktionsfähigkeit von Compiler und Laufzeitumgebung einem Test unter realen Bedingungen unterzogen, mit dem Ziel, Schwachstellen aufzudecken und Verbesserungen anzuregen.

# 2 ObjectTeams

Dieses Kapitel beschäftigt sich mit dem aspektorientierten Programmiermodell ObjectTeams ([OT04]). Da dieses im Rahmen der vorliegenden Fallstudie hinsichtlich der Erreichung seiner Ziele überprüft werden soll, wird einleitend motiviert, mit welchen Zielsetzungen es entwickelt wurde. Das heißt, es werden die Probleme erläutert, die mit Hilfe von ObjectTeams gelöst werden sollen. In diesem Zusammenhang werden Konzepte und Begriffe, die für das Verständnis der nachfolgenden Abschnitte und Kapitel erforderlich sind, eingeführt. Im Anschluss an diesen einleitenden Teil werden dann die in ObjectTeams enthaltenen Konzepte erklärt. Dabei wird einerseits der Zusammenhang zwischen dem jeweiligen Konzept und dem mittels diesem zu lösenden Problem hergestellt. Andererseits wird die spezielle Ausprägung der Konzepte in der Sprache ObjectTeams/Java erläutert. Der Fokus der Erläuterungen liegt hierbei auf der Anwendung der Sprache bei der Software-Entwicklung.

**Motivation und Einführung allgemeiner Konzepte und Begriffe**

Das Prinzip der Trennung der Belange

Die Welt ist sehr komplex. Ein Objekt (oder Subjekt) der realen Welt hat vielfältige Eigenschaften und Fähigkeiten, verschiedene Beziehungen zu anderen Objekten und ist dadurch Teil von vielfältigen Interaktionen (*Kollaborationen*). Da das menschliche Auffassungsvermögen begrenzt ist, werden diese Eigenschaften und Beziehungen schon bei einem relativ kleinen zu betrachtenden Bereich für uns unüberschaubar. Trotzdem können wir komplexe Sachverhalte verstehen und nicht-triviale Probleme lösen. Das ist möglich, weil wir abstrahieren, und zwar indem wir einen zu betrachtenden Bereich nach Kontexten (*Concerns*) organisieren, das heißt, die Elemente des Bereichs (Objekte, ihre Eigenschaften und Beziehungen untereinander) bestimmten Kontexten zuordnen, und diese Kontexte dann getrennt voneinander einzeln betrachten. Dieses Organisations-Prinzip wird auch als Trennung der Belange (*Separation of Concerns*) bezeichnet. Auch wenn jeder von uns intuitiv diesem Prinzip folgt, soll es an dieser Stelle noch einmal kurz erläutert und dabei elementare Konzepte und Begriffe geklärt werden.

Ein Kontext (*Concern*) ist ein bestimmter Zusammenhang, ein Gebiet von Interesse, man könnte auch sagen, ein Abstraktionsniveau. Innerhalb eines bestimmten Kontexts übernimmt ein Objekt eine bestimmte Funktion. Man sagt auch, es spielt in diesem Kontext eine bestimmte *Rolle*. Innerhalb des Kontexts ist nur eine Teilmenge der Objekte und von diesen nur die Eigenschaften und die Beziehungen und Interaktionen (*Kollaborationen*) untereinander sichtbar, die sie in ihrer jeweiligen kontextbezogenen Funktion (*Rolle*) besitzen bzw. ausführen. Wenn ein Objekt nicht nur in einem sondern in mehreren Kontexten eine Rolle spielt, vereint es in sich verschiedene *Aspekte* (Teilfunktionen).

Die Anwendung des Prinzips der Trennung der Belange in der Softwaretechnik kann vor allem auf die Arbeiten von Parnas [Par72] und Dijkstra [Dij76] zurückgeführt werden, die anregen, die Elemente eines Softwaresystems intentional und lokal (Lokalitätsprinzip) zu beschreiben, mit dem Ziel, die Software verständlicher, wiederverwendbarer und besser wartbar zu machen. In der Praxis erfolgt die Identifikation und Trennung der Belange mittels Dekompositionstechniken, wie modularer und objektorientierter Dekomposition (siehe [Loh02]).

Die Objektorientierung unterstützt die lokale Definition einzelner Belange besser als ihre Vorgänger, stößt aber ab einer gewissen Komplexität an ihre Grenzen. Ihre Dekompositionstechniken sind unzureichend, um alle auftretenden Belange getrennt voneinander zu behandeln. In komplexen Softwaresystemen sind die Implementierungen verschiedener Belange daher oft vermischt und miteinander verwoben. Sie weisen eine hohe Abhängigkeit voneinander auf, was die Verständlichkeit, Wiederverwendbarkeit, Wartbarkeit und Erweiterbarkeit der Software verringert.

Dabei wird die Kopplung, also der Umstand, dass eine Klasse Implementierungselemente verschiedener Belange enthält, als „*Code Tangling*“ bezeichnet. Liegen die Implementierungselemente eines Belangs verstreut über viele Klassen der Software, wird dies als „*Code Scattering*“ bezeichnet. Treten Kopplung und Streuung gemeinsam auf, schneidet die Implementierung eines bestimmten Belangs also durch die objektorientierte Struktur der Software, handelt es sich bei dem betreffenden Belang um einen „*Crosscutting Concern*“.

Aspektorientierte Konzepte wie ObjectTeams versuchen, diese Grenzen der Objektorientierung zu überwinden und die Möglichkeiten der Trennung der Belange in Softwaresystemen zu verbessern.

**ObjectTeams und ObjectTeams/Java**

Modulkonzept Team

Das Kernkonzept von ObjectTeams ist das *ObjectTeam* oder auch kurz *Team* genannt. Es handelt sich dabei um ein Modulkonzept, welches es ermöglichen soll, einen Kontext und die in ihm stattfindenden Objekt-Kollaborationen als eigenständigen, unabhängigen Software-Baustein zu definieren und einer existierenden Software noninvasiv, das heißt ohne Veränderung ihrer Strukturen, hinzuzufügen. Es kapselt einen *crosscutting concern*, weil es die Möglichkeit bietet, alle einen bestimmten Kontext betreffenden Implementierungselemente, welche in einer objektorientierten Software verstreut und an die Strukturen anderer Kontexte gekoppelt wären, lokal und in sich geschlossen zu definieren. Dadurch können die Probleme des *code scattering* und *code tangling* vermieden werden.

Teamlevel-Attirbute und -Methoden und Rollen

Ein Team verbindet Eigenschaften der Konzepte Paket und Klasse miteinander. Es enthält einerseits Objekte, die miteinander in Beziehung stehen und kollaborieren, und kann andererseits auch eigene Eigenschaften und Funktionen (*Teamlevel-Attribute* und *-Methoden*) haben. Da die in einem Team enthaltenen Objekte nur solche Eigenschaften und Funktionalitäten besitzen, die in dem logischen Kontext, den das Team repräsentiert, von Bedeutung sind, werden sie, angelehnt an den natürlichsprachlichen Gebrauch dieses Begriffs, *Rollen* genannt. Die Definition von Teams und enthaltenen Rollen erfolgt, dem Abstraktionskonzept der Objektorientierung folgend, in der Form instanziierbarer Klassen. Abbildung 1 zeigt, wie eine Team-Klasse und die enthaltenen Rollenklassen in der Notation der Sprache UFA (UML for Aspects, siehe [Her02a]), einer Erweiterung der Sprache UML um die Konzepte von ObjectTeams, dargestellt werden.

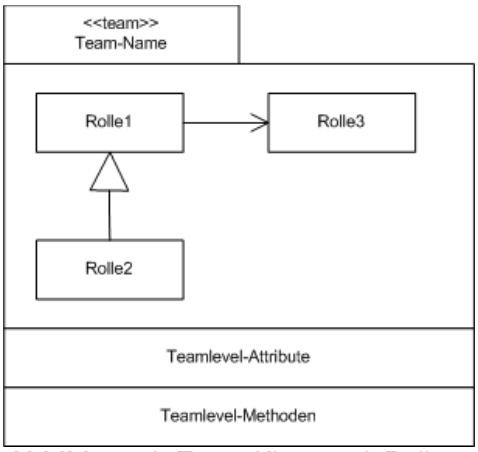


***Abbildung 1:*** *Team-Klasse mit Rollen-Klassen, Darstellung in UFA*

Definition von Team- und Rollen-Klassen in ObjectTeams/Java

In der Sprache ObjectTeams/Java wird eine Team-Klasse wie eine gewöhnliche Klasse definiert und erhält zur Kennzeichnung das zusätzliche Schlüsselwort `team` (`public team class SampleTeamClass`). Zur Kennzeichnung von Rollenklassen existiert kein spezielles Schlüsselwort. Sie werden entweder als innere Klassen ihrer Team-Klasse in deren Datei oder wie eine eigenständige Klasse in einer eigenen Datei definiert. Im letzteren Fall wird zur Kennzeichnung als Rollenklasse und Zuordnung zu einer bestimmten Team-Klasse anstelle des Paketes der voll qualifizierte Name der Team-Klasse angegeben. Die verschiedenen Möglichkeiten der Definition von Rollenklassen unterscheiden sich dabei nur technisch, nicht aber inhaltlich. Sie dienen ausschließlich der Vereinfachung der praktischen Anwendung des Rollenkonzepts.

Spezialisierung von Team-Klassen

Die durch ein Team gekapselte Kontext-Definition kann mit Hilfe von Vererbung spezialisiert werden. Dabei gibt es die Einschränkung, dass Team-Klassen nur untereinander in eine Vererbungsbeziehung treten dürfen. Das heißt, eine Team-Klasse kann nicht von einer Klasse erben, die keine Team-Klasse ist und umgekehrt. In ObjectTeams/Java erbt eine Team-Klasse, die keine Vererbungsbeziehung deklariert, automatisch von der Klasse `org.objectteams.Team`.

Implizite Vererbung von Rollenklassen

Im Rahmen der Spezialisierung einer Team-Klasse können alle in ihr enthaltenen Rollenklassen spezialisiert werden. Dafür ist kein spezielles Schlüsselwort erforderlich. Stattdessen wird die Vererbungsbeziehung über Namensgleichheit hergestellt. Das heißt, eine Rollenklasse SubRole erbt automatisch von einer Rollenklasse SuperRole, wenn erstens beide Rollenklassen den selben Namen tragen und zweitens SubRole in einem Sub-Team der Team-Klasse von SuperRole definiert ist. Diese Art der Vererbung zwischen gleichnamigen Rollenklassen wird als *implizite Vererbung* bezeichnet.

Explizite Vererbung von Rollenklassen

Daneben kann eine Rollenklasse auch wie gewohnt auf deklarative Art und Weise mit Hilfe des Schlüsselwortes extends in eine Vererbungsbeziehung treten. Diese Art der Vererbung wird zum Zwecke der Unterscheidung als *explizite Vererbung* bezeichnet. Dabei können Rollenklassen sowohl von gewöhnlichen Klassen außerhalb ihres Teams als auch von anderen Rollenklassen erben, wobei zu beachten ist, dass eine Rollenklasse nur von einer Rollenklasse erben kann, die im selben Team oder in einem Super-Team dieses Teams definiert ist.

Unterscheidung und Schlüsselwort tsuper

Eine Rollenklasse kann gleichzeitig Teil einer impliziten und einer expliziten Vererbungshierarchie sein. Auf die selbe Art und Weise, wie beim Überschreiben einer explizit geerbten Methode mit Hilfe des Schlüsselwortes `super` die Ausführung der Originalimplementierung veranlasst werden kann, funktioniert dies bei einer implizit geerbten Methode mit Hilfe des neuen Schlüsselwortes `tsuper`.

Objektorientierte Integration von Teams

Eine Team-Instanz kann in einer Software wie ein ganz normales Objekt benutzt werden. In einem solchen Fall erweitert ObjectTeams die in der Objektorientierung und Java enthaltenen Konzepte von äußeren und inneren Klassen lediglich um die beschriebenen erweiterten Möglichkeiten der Spezialisierung.

Aspektorientierte Integration von Teams

Darüberhinaus kann ein in einem Team definierter Kontext auch in aspektorientierter Form noninvasiv in eine existierende Software-Domäne integriert werden. Man sagt auch, das Team adaptiert die Domäne.

Rolle-Basis-Beziehung

Dies geschieht durch die Deklaration von Bindungen der im Team enthaltenen Rollenklassen an Klassen der Software-Domäne. Dabei wird die an eine bestimmte Rollenklasse gebundene Domänenklasse als ihre *Basisklasse* bezeichnet. Die Deklaration dieser Beziehung erfolgt mit Hilfe des Schlüsselwortes `playedBy` (`Roleclass playedBy Baseclass`) innerhalb der Rollenklasse. Die Basisklasse enthält keinerlei Hinweise auf diese Bindung, ist sich ihrer also sozusagen nicht bewußt.

Abbildung 2 zeigt die Bindung eines Teams an eine Software-Domäne in der Darstellung der Sprache UFA. Dabei ersetzt das Gleichheitszeichen das Schlüsselwort `playedBy`. Neben dieser Darstellungsart wird im Rahmen der vorliegenden Arbeit auch häufig die in Abbildung 4 auf Seite 13 gezeigte verwendet, um das intuitive Verständnis zu fördern.

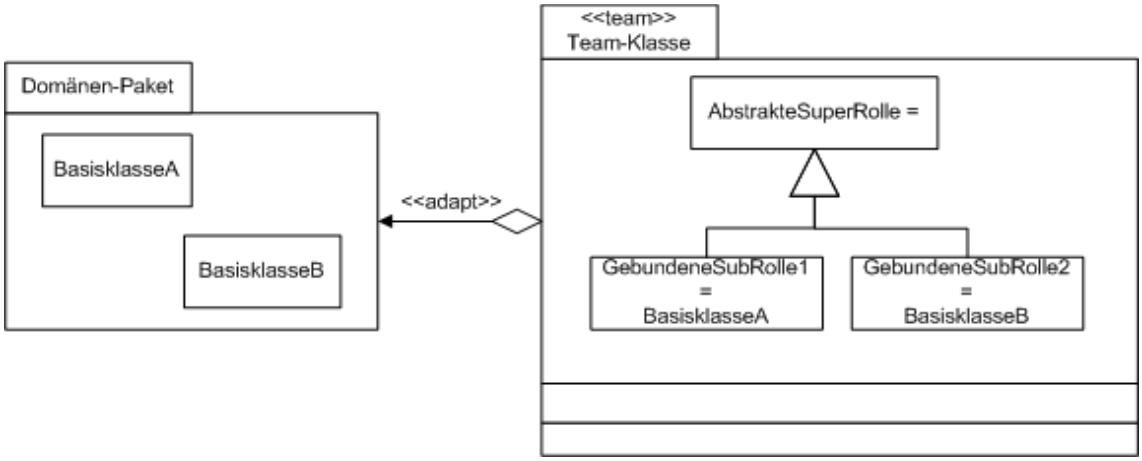


***Abbildung 2:*** *Anbindung eines Teams an ein Paket mittels Rolle-Basis-Bindung, Darstellung in UFA*

Sowohl die Bezeichnung Rolle als auch das Schlüsselwort `playedBy` legen nahe, dass es sich bei dem programmiersprachlichen Konzept der Bindung einer Rolle an eine Basis um eine Abbildung des Denkmodells handelt, in dem wir uns ganz intuitiv bewegen. In der realen Umwelt gibt es zwar Kontexte. Die in ihnen enthaltenen Rollen existieren aber nicht losgelöst von realen Objekten, die auch außerhalb eines bestimmten Kontexts existieren und dort auch

Eigenschaften besitzen, die nichts mit diesem Kontext zu tun haben. Eine Rolle in der realen Welt muss von einem konkreten Objekt gespielt werden. Auf die selbe Art und Weise kann in ObjectTeams eine Rolle von einer konkreten Domänenklasse, ihrer Basisklasse, gespielt werden.

Methodenbindung

Zu diesem Zweck ist es möglich entlang der Rolle-Basis-Beziehung zwei verschiedene Arten der *Methoden-Bindung* zu deklarieren. Dabei definiert die eine eine Delegation von einer Methode der Rolle zu einer Methode der Basis und wird dieser Aufrufrichtung (aus dem Team heraus) entsprechend als *callout*-Bindung bezeichnet. Die andere definiert eine Delegation in die entgegengesetzte Richtung (ins Team hinein) und trägt daher die Bezeichnung *callin*-Bindung.

Callout-Bindung

Eine *callout*-Bindung dient der Definition von Funktionalität, welche eine bestimmte Rolle zwar benötigt, dessen konkrete Ausprägung aber von der adaptierten Software-Domäne abhängig ist. Solche Funktionalitäten werden in der Rollenklasse in Form abstrakter Methoden deklariert. Sie können dann in der selben oder einer Sub-Rollenklasse an eine Methode der zugehörigen Basisklasse gebunden und dadurch implementiert (zu einer konkreten Methode gemacht) werden. Dies geschieht durch die Deklaration einer callout-Bindung mit Hilfe des Operators `->` (`rolemethod -> basemethod`), welche dazu führt, dass zur Laufzeit jeder Aufruf der Rollenmethode an die entsprechende Basismethode delegiert wird.

Callin-Bindung

Eine *callin*-Bindung dient der Erweiterung oder Ersetzung der Funktionalität der Basisklasse um die durch die Rolle definierte Funktionalität. Die Deklaration einer solchen Bindung erfolgt innerhalb der Rollenklasse mit Hilfe des Operators `<-` (`rolemethod <- basemethod`). Dabei können mehrere Methoden der Basisklasse an die selbe Methode der Rollenklasse gebunden werden. Diese Form der Bindung bewirkt, dass zur Laufzeit jeder Aufruf einer der gebundenen Basismethoden an die entsprechende Rollenmethode delegiert wird. Dieser Vorgang wird auf der konzeptuellen Ebene als *Aspektweben* bezeichnet. Abbildung 3 zeigt die Darstellung von Methodenbindungen in UFA.

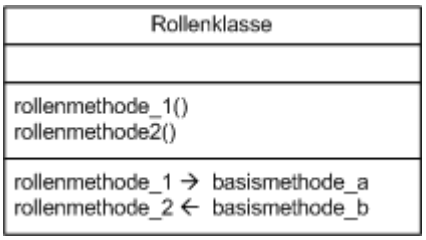


***Abbildung 3:****Methodenbindungen, Darstellung in UFA*

Arten der callin-Bindung: before, replace und after

Es existieren drei verschiedene Arten der callin-Bindung, die entsprechend dem Zeitpunkt des Aufrufs der Rollenmethode im Verhältnis zum Aufruf der Basismethode mit Hilfe der Schlüsselworte `before`, `replace` und `after` (`rolemethod <- replace basemethod`) unterschieden werden. Bei einem before-callin wird bei einem Aufruf der Basismethode zuerst die an diese gebundene Rollenmethode ausgeführt und danach sie selbst. Bei einem after-callin ist es umgekehrt. Und bei einem replace-callin wird die Rollenmethode anstelle der Basismethode ausgeführt.

Replace-callin und base-call

In diesem letzten Fall (replace-callin) kann die Rollenmethode die Basismethode entweder vollständig überschreiben oder deren Ausführung mit Hilfe des Schlüsselwortes `base` veranlassen und eigenes Verhalten sowohl vor als auch nach diesem Aufruf der Basismethode hinzufügen. Dieser Aufruf der Basismethode aus der per callin aufgerufenen Rollenmethode heraus wird als *base-call* bezeichnet. Eine Rollenmethode, die einen base-call enthält, muss mit dem zusätzlichen Schlüsselwort `callin` (`public callin void rolemethod`) gekennzeichnet werden und kann nicht aus der Rolle heraus sondern nur über callin aufgerufen werden.

Parameter-Mapping

Für die Definition der Bindung einer Rollenmethode an eine Basismethode ist weder deren Namens- noch Signaturgleichheit erforderlich. Unterscheiden sich die zu bindenden Methoden in Anzahl oder Typ ihrer Ein- oder Ausgabeparameter, muss in Form eines *Parameter-Mappings* die gewünschte Anpassung definiert werden. Dabei können Parameter versteckt gehalten, in ihrer Reihenfolge vertauscht, verknüpft oder beliebige andere Operationen auf sie angewendet werden.
Unterscheiden sich die zu bindenden Methoden in ihrer Signatur derart, dass der Parametertyp der Rollenmethode ein Rollentyp ist und der Parametertyp der Basismethode der an genau jenen Rollentyp gebundene Basistyp oder ein Subtyp desselben ist, ist die Deklaration eines Parameter-Mappings nicht erforderlich, weil der Parametertyp während der Delegation des Methodenaufrufs vom Laufzeitsystem automatisch übersetzt (siehe Abschnitt „Lifting und Lowering“ auf Seite 14) wird. Insbesondere gilt dies auch für Arrays, also wenn der Basisparameter ein Array eines bestimmten Typs und der Rollenparameter ein Array des an diesen gebundenen Rollentyps ist.

Vererbung und Spezialisierung der Rolle-Basis-Beziehung

Die Rolle-Basis-Bindung vererbt sich entlang der Rollen-Hierarchie und kann spezialisiert werden, das heißt eine Rollenklasse kann an eine Subklasse der Basisklasse ihrer Elternrollenklasse gebunden werden (siehe Abbildung 4).

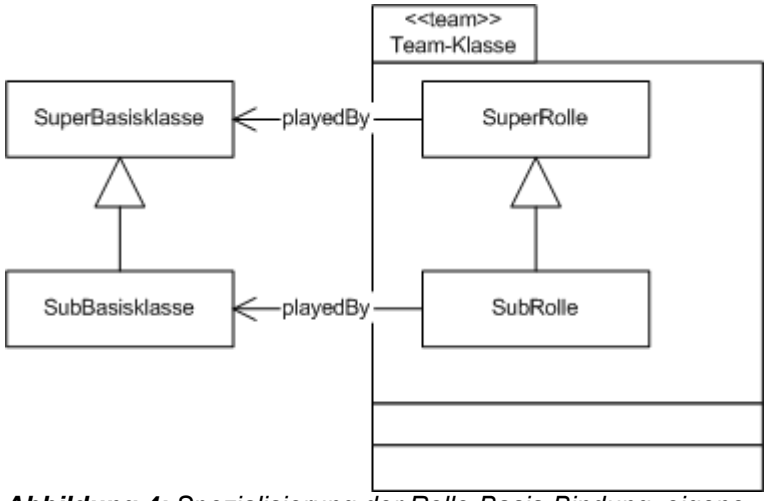


***Abbildung 4:*** *Spezialisierung der Rolle-Basis-Bindung, eigene Notation*

Merhfachbindung einer Basisklasse

Es ist möglich, sowohl Rollenklassen aus verschiedenen Team-Klassen als auch Rollenklassen aus verschiedenen Rollentyphierarchien der selben Team-Klasse an ein und die selbe Basisklasse zu binden. Diese Möglichkeiten spiegeln den Umstand wieder, dass auch in der realen Umwelt ein konkretes

Objekt in verschiedenen Kontexten eine oder mehrere Rollen spielen kann.
Wie bereits erwähnt, sind alle Bindungsdeklarationen in den Rollenklasse enthalten. Dem Quelltext der Basisklasse ist nicht anzusehen, ob und durch welche Rollenklassen zur Laufzeit ihre Funktionalität erweitert oder verändert wird.

Abstraktheit versus Instanziierbarkeit

Team-Klassen und Rollenklassen können wie gewöhnliche Klassen auch abstrakt oder konkret, das heißt instanziierbar sein. Eine Rollenklasse ist dann abstrakt und muss auch dementsprechend gekennzeichnet werden, wenn sie abstrakte Methoden enthält. Diese können sowohl in der Rollenklasse selbst deklariert als auch durch explizite oder implizite Vererbung geerbt worden sein. Eine Methode gilt dann nicht mehr als abstrakt, wenn sie entweder in gewohnter Weise selbst implementiert oder durch eine callout-Bindung an eine Basismethode gebunden wird. Dabei kann die Konkretisierung durch Deklaration einer callout-Bindung auch direkt in der Rollenklasse erfolgen, die die abstrakte Methode deklariert.
Die Unterscheidung zwischen Abstraktheit und Instanziierbarkeit einer Team-Klasse ist etwas schwieriger zu klären und soll an dieser Stelle vereinfacht dargestellt werden. Detailliertere Informationen können der ObjectTeams/Java Sprachdefinition ([Her03]) entnommen werden.
Allgemein kann gesagt werden, dass ein Team dann nicht instanziierbar ist und als abstrakt gekennzeichnet werden muss, wenn es eine relevante Rollenklasse enthält, die abstrakt ist. Dabei wird als relevante Rollenklasse eine solche Rollenklasse bezeichnet, die im Rahmen der für das Team deklarierten oder implizit erfolgenden Lifting-Operationen oder durch explizit im Team oder einem Super-Team enthaltene Erzeugungs-Anweisungen instanziiert werden müsste. Wegen der durch das Konzept der impliziten Rollenvererbung gegebenen Möglichkeit des Überschreibens ganzer Rollenklassen ist es nämlich möglich, dass ein Super-Team instanziierbar ist, weil alle enthaltenen relevanten Rollen instanziierbar sind, ein Sub-Team diese Rollenklassen aber so überschreibt, dass diese in ihm abstrakt sind und nicht instanziiert werden können. In einem solchen Fall, kann das Sub-Team nicht instanziiert werden und muss daher als abstrakt gekennzeichnet werden.

Lifting und Lowering

Wie in den bisherigen Erläuterungen bereits angeklungen ist, verwenden sich Rollen- und Basisobjekte gegenseitig nicht in gewohnter Weise, das heißt, sie besitzen keine Referenzen auf entsprechende Instanzen, um auf diesen Methoden aufzurufen. Stattdessen erfolgt zur Laufzeit bei Bedarf eine Art Übersetzung. Die Laufzeitumgebung veranlasst, dass ein Basisobjekt in seine Rolle "schlüpft" wenn es den Kontext des Teams betritt und die Rollenfunktionalität wieder "abstreift" wenn es den Team-Kontext wieder verlässt. Der Vorgang des in die Rolle "Hineinschlüpfens" wird als *Lifting* bezeichnet, die entgegengesetzte Richtung als *Lowering*.
Ein *Lifting* erfolgt zum Beispiel dann, wenn auf einem Basisobjekt eine Methode aufgerufen wird, für die im Team eine callin-gebundene Rollenmethode (siehe Abschnitt „Callin-Bindung“ auf Seite 12) existiert. In diesem Fall wird das Basisobjekt geliftet, das heißt der Kontrollfluss wird an das mit ihm assoziierte Rollenobjekt übergeben, damit die Rollenmethode ausgeführt werden kann.
Ein *Lowering* erfolgt immer dann, wenn ein Rollenobjekt die Funktionalität des ihm zugeordneten Basisobjekts direkt benötigt. Das ist zum Beispiel der Fall beim Aufruf callout-gebundener Rollenmethoden (siehe Abschnitt „Callout-Bindung“ auf Seite 12) oder bei einem base-call (siehe Abschnitt „Replace-callin und base-call“ auf Seite 13).
Im Rahmen einer Teamlevel-Methode kann ein Parameter-Lifting explizit

deklariert werden. Diese Deklaration erfolgt mit Hilfe des Schlüsselwortes `as` (`public void teamlevelmethod(basetype as roletype)`). Beim Aufruf der Methode des Teams muss dann ein Objekt des Basistyps übergeben werden, welches von der Laufzeitumgebung geliftet wird, so dass innerhalb der Methode ausschließlich ein Objekt des entsprechenden Rollentyps sichtbar ist. Der Rückgabewert einer Teamlevel-Methode wird automatisch gelowert, wenn die return-Anweisung ein Rollenobjekt liefert, der in der Signatur deklarierte Rückgabetyp aber der zugehörige Basistyp ist.

Rollenerzeugung und Kardinalität

Ein Objekt einer gebundenen Rollenklasse kann nicht ohne zugehöriges Basisobjekt existieren. Daher erfolgt die Erzeugung eines Rollenobjekts für gewöhnlich nicht durch einen im Quelltext der Rollenklasse integrierten Aufruf eines Rollenkonstruktors, sondern auf Initiative der Laufzeitumgebung bei Bedarf, das heißt wenn ein Basisobjekt geliftet werden soll und noch kein mit diesem assoziiertes Rollenobjekt existiert.
Pro Basisobjekt und an den Typ dieses Basisobjekts gebundene Rollenklasse existiert innerhalb einer Team-Instanz maximal ein Rollenobjekt, wobei die Zuordnung zwischen Basisobjekt und Rollenobjekt immer eindeutig ist, also für jedes Basisobjekt ein eigenes Rollenobjekt erzeugt wird.

Team-Aktivierung

Der in Form eines Teams unabhängig definierte und auf Implementierungsebene mit Hilfe von callin-Bindungen deklarativ integrierte Kontext, zum Beispiel eine bestimmte Programmfunktion, kann zur Laufzeit explizit ein- und auch wieder abgeschaltet (aktiviert und deaktiviert) werden. Zu diesem Zweck existieren die automatisch für jedes Team vorhandenen Methoden `activate` und `deactivate`.
Der Aufruf der Methode `activate` aktiviert alle in einem Team vorhandenen callin-Bindungen. Dabei kann durch Übergabe eines Parameters bei der Aktivierung der Grad (das Level) der Aktivierung bestimmt werden. Wird ein Team ohne besonderen Parameter aktiviert, ist seine Aktivierung uneingeschränkt. Es werden alle im System vorhandenen potentiellen Basisobjekte beobachtet und bei Bedarf Lifting und Aufrufe callin-gebundener Rollenmethoden ausgelöst. Wird eine Team-Aktivierung mit Hilfe des Parameters `FROZEN` (`org.objectteams.Team.FROZEN`) eingeschränkt, reagiert das Team nur auf solche Basisobjekte, die es kennt, das heißt, die schon einmal geliftet wurden. Zu diesem Zweck können die zu beobachtenden Basisobjekte beim Team mit Hilfe der Methode `add` oder einer selbst definierten Teamlevel-Methode mit deklariertem Parameter-Lifting registriert werden. Der Aufruf der Methode `deactivate` auf einem Team deaktiviert jegliche callin-Aktivität seinerseits.

Aufrufreihenfolge callin-gebundener Methoden

Existieren im System mehrere aktive Instanzen von Team-Klassen, die alle Rollenklassen enthalten, die an den selben Basistyp gebunden sind, unabhängig davon ob es sich dabei um Instanzen der selben Team-Klasse oder verschiedener Team-Klassen handelt, richtet sich die Reihenfolge der callin-Aufrufe nach der Reihenfolge, in der die Team-Instanzen aktiviert wurden, wobei das zuletzt aktivierte Team bei der Aufrufreihenfolge der callin-gebundenen Rollenmethoden an erster Stelle steht. Das heißt, dass wenn auf einem Basisobjekt eine Methode aufgerufen wird, für die in mehreren aktiven Team-Instanzen callin-gebundene Rollenmethoden existieren, zuerst die Rollenmethode aus dem Team aufgerufen wird, welches zuletzt aktiviert wurde.

Externalized Roles

Rollenobjekte sind innere Objekte der sie umgebenden Team-Instanz und in ihrer Erzeugung und Existenz von dieser abhängig. Normalerweise sind sie in

ihrer Team-Instanz eingeschlossen und werden von dieser nicht (als Rückgabewert einer Teamlevel-Methode) nach außen gegeben. Sollen sie trotzdem außerhalb der Team-Instanz direkt verwendet werden, sind besondere Regeln einzuhalten. Erstens muss die Variable, die das Rollenobjekt referenziert, den Rollentyp Team-Instanz-gebunden deklarieren und zweitens darf das Rollenobjekt nur mit Rollenobjekten der selben Team-Instanz oder der Team-Instanz selbst interagieren. Auf diese Art und Weise außerhalb des Teams verwendete Rollenobjekte werden als *externalized roles* bezeichnet. Abbildung 5 zeigt eine beispielhafte Anwendung.

```
final Teamtype teaminstance = new Teamtype();
teaminstance.Roletype roleobj = teaminstance.getSpecialRoleobject();
...
teaminstance.delete(roleobj);
```

***Abbildung 5:*** *Externalized Role, Instanz-gebundene Typ-Deklaration*

Team-Schachtelung

Im konzeptuellen Rahmen von ObjectTeams ist die Möglichkeit einer verschachtelten Team-Struktur vorgesehen. Das heißt, eine Team-Klasse soll in einer anderen Team-Klasse enthalten, also deren Rolle sein können. In einem solchen Fall besitzt die entsprechende Klasse sowohl die Eigenschaften eines Teams, als auch die einer Rolle. Dieser konzeptuelle Ansatz ist zum jetzigen Zeitpunkt in der Sprache ObjectTeams/Java noch nicht realisiert. Im Rahmen der vorliegenden Fallstudie wird aber deutlich werden, dass es für ihn durchaus reale Anwendungsmöglichkeiten gibt (siehe Abbildung 21 auf Seite 50).

Die im Rahmen dieses Kapitels erfolgte, anwendungsfokussierte Einführung in die Konzepte von ObjectTeams und deren Ausprägung in der Sprache ObjectTeams/Java erhebt keinen Anspruch auf Vollständigkeit. Dort wo es im Rahmen der folgenden Kapitel zum Verständnis notwendig ist, werden bestimmte Teilaspekte detaillierter beschrieben.
Tiefergehende Einsichten in die Konzepte von ObjectTeams, ihre Einbettung in den Rahmen der Aspektorientierung, Fragen der Typsicherheit und die Sprache ObjectTeams/Java ermöglichen die Arbeiten [Her02a], [Vei02], [VH03] und [Her03]. Dieses Material ist neben der aktuellen Version des ObjectTeams/Java-Sprachpakets auf der ObjectTeams Home Page unter http://www.objectteams.org zugänglich

# 3 Frameworks

## 3.1 Definition

Ralph Johnson und Brian Foot definieren den Begriff Framework in [JF88] folgendermaßen:

> *"A framework is a reusable, semi-complete application that can be specialized to produce custom applications"*

Einige Jahre später definiert wieder Ralph Johnson diesmal zusammen mit Kent Beck in [BJ94]**:**

> *"A framework is the reusable design of a system or a part of a system expressed as a set of abstract classes and the way instances of (subclasses of) those classes collaborate"*

Diese beiden Definitionen widersprechen sich nicht, sondern betonen lediglich zwei verschiedene Sichtweisen auf die selbe Sache, beschreiben einerseits den durch sie verfolgten Zweck und andererseits die ihr zugrunde liegende Struktur und vermitteln in Kombination miteinander einen guten Eindruck davon, worum es sich bei einem Framework handelt.

Beginnend mit dem verfolgten Zweck ist zu sagen, dass es sich bei Frameworks um eine Wiederverwendungstechnik handelt, bei der nicht nur, wie bei früheren Ansätzen, einzelne Systembausteine, sondern eine ganze Systemarchitektur, bestehend aus den einzelnen Systemkomponenten in Form von Klassen und zusätzlich, als integraler Bestandteil, deren Kollaborationen wiederverwendet werden können. Ein Framework ist, wie das erste Zitat nahe legt, ein halb-fertiges Programm, auf dessen Grundlage eine Vielzahl von konkreten Anwendungen entwickelt werden kann. Durch die Art der auf seiner Grundlage erstellbaren Anwendungen deckt es einen bestimmten Problembereich, eine Anwendungsdomäne ab. Dabei werden die für alle in der Domäne befindlichen Anwendungen immer gleichen (invarianten) sowohl strukturellen als auch Verhaltens-Anteile durch die Framework-Implementierung vorgegeben und bei der Erstellung einer bestimmten Applikation um die für diese eine Applikation spezifischen Anteile erweitert. Bedingt durch diesen Zweck besitzt ein Framework besondere Eigenschaften.

Auch wenn dies nicht unbedingt erforderlich ist, ist ein Framework typischerweise in einer objektorientierten Sprache programmiert. Alle zur Beschreibung der abgedeckten Anwendungsdomäne erforderlichen Komponenten sind in Form abstrakter Klassen definiert. Dadurch können die beteiligten Komponenten einerseits anhand ihrer statischen Eigenschaften, also ihrer Schnittstellen, definiert werden, andererseits können invariante Algorithmen, sowie Kollaborationen zwischen den Komponenten bereits vom Framework implementiert werden. Auf diese Weise wird sozusagen ein Skelett zur Verfügung gestellt, welches durch Ableiten von konkreten Klassen und Implementieren der abstrakten Methoden um spezifische Funktionalität erweitert werden kann. In Programmiersprachen wie Java, die eine Unterscheidung zwischen abstrakten Klassen und Interfaces kennen, können die Komponenten der Domäne in Form von Interfaces definiert werden, wobei dadurch nicht der kollaborative Anteil abgedeckt werden kann, weshalb im

allgemeinen zusätzlich abstrakte Klassen vorhanden sind, die die definierten Interfaces implementieren und die genannte Lücke füllen.
Neben der durch Interfaces und abstrakte Klassen beschriebenen System-Architektur, die nach [BMMB99] als *"core framework design*" bezeichnet wird, stellt ein Framework für gewöhnlich zusätzlich konkrete Klassen zur Verfügung, die von den abstrakten Klassen abgeleitet sind bzw. die Interfaces implementieren und in der vom Framework abgedeckten Anwendungsdomäne häufig benötigte Funktionalität zur Verfügung stellen. Dieser Teil eines Frameworks wird nach [BMMB99] als *"framework internal increment"* bezeichnet.
Soll nun aufbauend auf dem Framework eine konkrete Applikation erstellt werden, können eigene Subklassen von den vom Framework zur Verfügung gestellten sowohl abstrakten als auch konkreten Klassen abgeleitet werden, um in diesen dann abstrakte Methoden zu implementieren bzw. konkrete Methoden zu redefinieren, um auf diesem Weg die speziellen Eigenschaften der Applikation zu definieren. Daneben können auch Klassen erstellt werden, die die Interfaces des Frameworks implementieren. Die Menge dieser applikations-spezifischen Klassen wird nach [BMMB99] als *"application specific increment"* bezeichnet. Neben der Möglichkeit eigene Klassen zu definieren, kann eine Anwendung auch die vom Framework zur Verfügung gestellten Klassen des "framework internal increment" benutzen.
Abbildung 6 Auf Seite 23 zeigt einen Ausschnitt der Struktur und Instanziierung des Frameworks JHotDraw. Dabei sind die Klassen des „core framework design“ rot, die des „framework internal increment“ blau und die des „application internal increment“ grün unterlegt.

Frameworks grenzen sich von anderen Wiederverwendungstechniken, insbesondere von Klassenbibliotheken, vor allem durch die Eigenschaft der *Inversion des Kontrollflusses* ab. Dabei ruft nicht die auf dem Framework basierende Applikation die Methoden der vom Framework definierten Klassen auf, sondern umgekehrt. Der Kontrollfluss liegt beim Framework. Dieses ruft die Methoden der Applikations-Klassen auf. Dieses Verhalten ist auch unter der Bezeichnung "Hollywood-Prinzip" ("don't call us, we'll call you!") bekannt.

## 3.2 Klassifikation

Frameworks können nach unterschiedlichen Kriterien klassifiziert werden. Die in der Literatur am häufigsten verwendeten Kriterien sind dabei einerseits der vom Framework abgedeckte Problembereich und andererseits die Art und Weise der Benutzung, wobei diese Kriterien und die auf ihrer Grundlage hergeleiteten Kategorien orthogonal zueinander verlaufen.

Bei der Betrachtung des von einem Framework abgedeckten Problembereichs unterscheidet Mattson in [Mat96] drei Kategorien, Application-Frameworks, Domain-Frameworks und Support-Frameworks.

Zur Kategorie der Application-Frameworks zählen zum Beispiel Frameworks für graphische Benutzerschnittstellen (GUI-Frameworks), zu denen auch das in der vorliegenden Arbeit verwendete Framework JHotDraw gehört. Diese Frameworks implementieren spezifische Funktionalitäten, können aber zur Erstellung von Applikationen ganz verschiedener Anwendungsdomänen benutzt werden.

Im Gegensatz dazu beschränken sich Domain-Frameworks auf eine ganz

bestimmte Anwendungsdomäne, z.B. Messsysteme, und eignen sich ausschließlich als Grundlage für die Entwicklung einer Applikation dieses speziellen Bereichs.

Zur Letzten Kategorie, den Support-Frameworks, zählen solche, die Low-Level Services, z.B. für Betriebssysteme, zur Verfügung stellen.

Wird bei der Klassifikation von Frameworks die Art und Weise der Benutzung berücksichtigt, welche davon abhängt, wie viel vom Design und der Implementierung des jeweiligen Frameworks für den Anwendungsentwickler sichtbar ist, werden die Kategorien Whitebox-Framework, Blackbox-Framework und Graybox-Framework unterschieden.

Bei Whitebox-Frameworks sind Design und Implementierung vollständig offen gelegt und für den Anwendungsentwickler sichtbar. Frameworks dieser Art werden vor allem durch Ableiten von Subklassen von den vom Framework zur Verfügung gestellten Klassen und Überschreibung vordefinierten Hook-Methoden verwendet. Ihre Verwendung erfordert vom Anwendungsentwickler relativ gute Kenntnis der internen Design-Struktur und damit einen recht hohen Lernaufwand. Im Gegenzug schränken Whitebox-Frameworks die Anpassungsmöglichkeiten nur wenig ein und bieten dadurch eine hohe Flexibilität. Das in der vorliegenden Arbeit verwendete Framework JHotDraw fällt in diese Kategorie.

Blackbox-Frameworks bieten dem Anwendungsentwickler im Gegensatz zu Whitebox-Frameworks keinen Einblick in die internen Details ihrer Klassen. Ihre Verwendung beruht nicht auf Vererbung, sondern auf Objektkomposition und Delegation. Zu diesem Zweck erfolgt entweder eine Konfiguration existierender Framework-Klassen und/oder es werden neue Klassen erstellt, welche die vom Framework zur Verfügung gestellten Interfaces implementieren, damit diese dann zum Beispiel mit Hilfe des Strategy Design Patterns (siehe [GHJV95], S.315ff.) integriert werden können. Blackbox-Frameworks sind im allgemeinen einfacher zu verwenden als Whitebox-Frameworks, bieten dafür aber weniger Flexibilität als diese.

Bei Graybox-Frameworks handelt es sich um eine Mischform aus Whitebox- und Blackbox-Frameworks. Diese unternimmt den Versuch, die Vorteile beider Framework-Arten miteinander zu verbinden und deren Nachteile zu reduzieren. Ein Graybox-Framework versucht eine Balance zu finden zwischen dem Verstecken von Details der Implementierung und dem Bieten von Flexibilität bei der Erstellung einer Applikation auf seiner Grundlage.

## 3.3 Dokumentation

Ein für die Güte eines Frameworks Ausschlag gebendes Kriterium ist, dass die Entwicklungszeit einer auf ihm basierenden Applikation wesentlich kürzer ist, als die einer äquivalenten Applikation, die ohne Zuhilfenahme des Frameworks entwickelt wurde. Um diese Anforderung erfüllen zu können, ist aufgrund der Tatsache, dass es sich bei einem Framework um eine komplexe Software handelt, eine ausreichende Dokumentation unentbehrlich und muss als fester Bestandteil eines Frameworks angesehen werden. Dabei sollte diese Dokumentation nach [Mat96] verschiedene Abstraktionsebenen abdecken, die die verschiedenen Bedürfnisse der potentiellen Framework-Benutzer widerspiegeln.

Auf der obersten dieser Ebenen sollte dokumentiert werden, welche Anwendungsdomäne das Framework abdeckt, das heißt, welche Arten von Applikationen auf seiner Grundlage erstellt werden können. Diese Information ist notwendig, um entscheiden zu können, ob ein Framework für die Entwicklung einer bestimmten Applikation überhaupt geeignet ist.
Die mittlere Ebene erläutert die Benutzung des Frameworks bei der Erstellung einer Applikation in der von den Framework-Entwicklern beabsichtigten Art und Weise. Sie spricht damit jene Entwickler an, die sich dafür entschieden haben, das Framework als Basis der von ihnen zu erstellenden Applikation zu verwenden. Die Abgrenzung dieser Ebene von der darunter liegenden, in die Details des Designs gehenden Ebene ist im Hinblick auf die oben definierte Anforderung der schnellen Anwendbarkeit von großer Bedeutung. Die benutzungsorientierte Dokumentation sollte nicht mehr Details über die interne Struktur des Frameworks Preis geben, als erforderlich, da die meisten Framework-Benutzer kein derartiges Detailwissen benötigen. Auf der Abstraktionsebene der Benutzung bewegen sich im Rahmen der vorliegenden Arbeit die Kapitel 4.5 und 6.7, wobei in Kapitel 4.5 die Benutzung der Implementierung der Speicherfunktionalität und in Kapitel 6.7 die der Undo-Funktionalität des Frameworks JHotDraw beschrieben wird.
Die unterste Ebene beschreibt detailiert Framework-Design und -Implementierung. Sie richtet sich an Entwickler, die das Framework als solches weiterentwickeln oder es abseits der eigentlichen Absichten der Framework-Entwickler zur Erstellung einer Applikation benutzen wollen. Auf dieser Ebene bewegen sich im Rahmen der vorliegenden Arbeit die Kapitel 4.2, 4.4, 6.3 und 6.6, wobei in Kapitel 4.2 die objektorientierte und in 4.4 die ObjectTeams-orientierte Implementierung der Speicher-Funktionalität des Frameworks JHotDraw und analog dazu in 6.3 und 6.6 die Implementierungen der Undo-Funktionalität erläutert werden.

Im folgenden werden einige der verschiedenen Möglichkeiten zur Dokumentation von Frameworks vorgestellt.

### 3.3.1 Beispiele

Eine Form der Dokumentation eines Frameworks besteht aus Beispiel-Applikationen, die auf der Grundlage des Frameworks entwickelt wurden, und deren Quelltext. Dabei handelt es sich nach Butler und Dénommée ([BD99]) oftmals um die einzige mitgelieferte Dokumentation, wobei es sich um vollständige Applikationen handelt, die regen Gebrauch von den vom Framework zur Verfügung gestellten Funktionalitäten machen. Sparks, Benner und Faris bemängeln in [SBF96], dass derartige Beispielprogramme für sich allein stehend besonders für Erstbenutzer eines Frameworks zu schwierig zu verstehen seien. Aus diesem Grund fordert er eine schritt- bzw. stufenweise Einführung der *"Hot Spots"* des Frameworks, das heißt der Stellen, die zur Erweiterung durch eine konkrete Anwendung gedacht sind, und zwar beginnend mit einfachen hin zu fortgeschritteneren Benutzungen des Frameworks. Diese kann zum Beispiel in Form eines Satzes von in ihrem Schwierigkeitsgrad steigenden Übungsbeispielen erfolgen, wobei jedes Beispiel der Illustration eines ganz bestimmten Hot Spots dient.
Neben der Nutzung von Beispielprogrammen als für sich allein stehende Dokumentation, werden Quelltext-Beispiele auch in Kombination mit bzw. als Element in anderen Dokumentationsformen, darunter Cookbooks und Design Patterns, verwendet.

### 3.3.2 Recipes / Cookbooks

Ein Recipe (Rezept) ist eine informelle, natürlichsprachliche Beschreibung einer bestimmten typischen Benutzung des Frameworks bei der Entwicklung einer Applikation. Es ist für gewöhnlich angereichert um einfache Beispiele in Form von Quelltext und eventuell auch Graphiken und folgt oft auch einer bestimmten Struktur, z.B. in der Form: Zweck, auszuführende Arbeitsschritte, Querverweise auf andere Rezepte und Beispiel-Quelltext.

Ein Cookbook (Kochbuch) ist eine Sammlung solcher Rezepte, wobei es entweder in Form eines Inhaltsverzeichnisses oder mittels des ersten in ihm präsentierten Rezepts einen Überblick über alle enthaltenen Rezepte bietet.

Johnson führt in [Joh92] den Begriff Pattern ein und stellt damit ein Rezept-Format und eine Cookbook-Organisationsstruktur zur Verfügung. Dabei beschreibt ein Pattern zuerst ein häufig auftretendes Problem bei der Entwicklung einer Applikation in der Anwendungsdomäne des Frameworks. Danach beschreibt es anhand von Beispielen (verschiedene Möglichkeiten) wie dieses Problem mit Hilfe des Frameworks gelöst werden kann. Zuletzt folgen neben einer Zusammenfassung Verweise auf andere Patterns, die benötigt werden, um das beschriebene zu realisieren oder anzureichern. Patterns bilden über diese Querverweise einen gerichteten Graphen. Dabei gibt das erste Pattern immer einen Überblick über das Framework-Konzept und andere Patterns, die zu dessen Verständnis erforderlich sind. Patterns, die häufig auftretende Probleme bzw. Benutzungssituationen erläutern, werden weiter vorn präsentiert, während speziellere weiter hinten anzufinden sind.

### 3.3.3 Design Pattern

Ein Design Pattern beschreibt nach Gamma ([GHJV95]) ein häufig auftretendes Design-Problem und dessen erfolgreiche Lösung. Dabei wird nicht nur die Struktur der Lösung sondern auch die Gründe für die getroffenen Design-Entscheidungen verdeutlicht. Die Beschreibung folgt immer dem gleichen Schema. Am Anfang steht der Name, unter dem das Design Pattern bekannt ist. Daran schließt sich eine Beschreibung des zu lösenden Problems an. Im Anschluss daran wird das Design vorgestellt, mit dessen Hilfe das Problem gelöst werden kann. Dabei werden neben der Struktur in Form der teilnehmenden Klassen, vor allem deren Kollaborationen und die im Kontext dieser Kollaborationen stattfindende Rollenverteilung der Klassen untereinander beschrieben. Dies geschieht durch ein Zusammenspiel aus natürlichsprachlichen Erläuterungen, Klassendiagrammen zur Veranschaulichung der Struktur und Sequenzdiagrammen zur Verdeutlichung der Kollaborationen. Abschließend werden die Konsequenzen dargestellt, die die Anwendung des Design Patterns hat. Dies umfasst sowohl die Vor- als auch die möglichen Nachteile seiner Anwendung. Daneben kann die Beschreibung eines Design Patterns weitere Bestandteile, wie konkrete Anwendungsbeispiele anhand von Quelltext umfassen. Die Dokumentation von Frameworks mittels Design Pattern eignet sich zur Veranschaulichung der internen Design Struktur und bewegt sich auf unterster und mittlerer Abstraktionsebene.

Neben den genannten existieren zahlreiche weitere Ansätze zur Framework-Dokumentation, von denen sich manche deutlich von den beschriebenen abgrenzen, wie die formale Spezifikation mittels Contracts ([HHG90]), wohingegen z.B. die Mikroarchitektur-Beschreibung nach [LK95] alle genannten

in sich vereint.

## 3.4 JHotDraw

Bei JHotDraw handelt es sich um ein Framework zur Erstellung von Editoren für strukturierte Graphiken. Es lässt sich anhand der vorgestellten Klassifikationen einerseits in die Gruppe der Application Frameworks und andererseits in die Gruppe der Whitebox Frameworks einordnen. Es wurde ursprünglich unter dem Namen HotDraw von Kent Beck und Ward Cunningham entwickelt, mehrere Male (in verschiedenen Sprachen) reimplementiert und liegt mittlerweile in seiner Java-Variante JHotDraw in der Version 6.01b vor.
(J)HotDraw ist ein beliebtes Studienobjekt. Johnson erläutert in [Joh92] anhand von HotDraw die Dokumentation der Anwendungsdomäne und der Benutzung eines Frameworks mit Hilfe von Pattern. Beck und Johnson veranschaulichen in [BJ94] an seinem Beispiel die Motivation und Herleitung einer Architektur mittels Design Patterns.
Auch in der vorliegenden Arbeit dient JHotDraw als Studienobjekt. Dabei wurde sein Design und seine Implementierung auf Vorkommen von Streuung (Code Tangling) und Kopplung (Code Scattering) hin untersucht und die identifizierten Bereiche anschließend einer Restrukturierung mittels der durch ObjectTeams eingeführten Konzepte unterzogen, um einerseits die Anwendbarkeit der Konzepte, andererseits ihre Auswirkungen auf die Eigenschaften der Software überprüfen zu können. Das zu diesem Zweck erforderliche detaillierte Wissen über bestimmte Bereiche des Designs und der Implementierung von JHotDraw wird weiter hinten, jeweils in den Kapiteln vermittelt, in denen es benötigt wird. Die Beschreibung der Implementierung und Benutzung der Programmfunktion "Speichern einer Zeichnung" erfolgt in Kapitel 4.2 Die Beschreibung der Programmfunktion "Öffnen eines Editorfensters" in Kapitel 5.1 und "Rückgängigmachen einer Benutzeraktion" in Kapitel 6.3. An dieser Stelle soll lediglich überblicksartig der zum allgemeinen Verständnis wichtigste Teil des "core framework design", eine zentrale Klasse des "framework internal increment" und die elementarste Benutzungsbeziehung vorgestellt werden.

Das Interface `DrawingEditor` (siehe Abbildung 6) definiert die Schnittstelle eines Editorprogramms bzw. eines Editorfensters. Ein solches Editorfenster kann eine beliebige Anzahl von `DrawingViews` enthalten, die wiederum jedes für sich für die Darstellung einer Zeichnung (`Drawing`) verantwortlich sind. Eine Zeichnung besteht ihrerseits aus einer beliebigen Anzahl von Figuren (`Figure`), welche Griffe (`Handles`) besitzen können, über die sie vom Benutzer manipuliert werden können. Neben diesen `Handles`, die von den Figuren selbst gehalten werden, existieren zum Zwecke ihrer Manipulation zusätzlich Werkzeuge (`Tools`). Diese werden vom `DrawingEditor` gehalten, wobei der Benutzer aus der Menge der vorhandenen `Tool`-Objekte jeweils eines auswählen und zum aktiven `Tool` bestimmen kann.

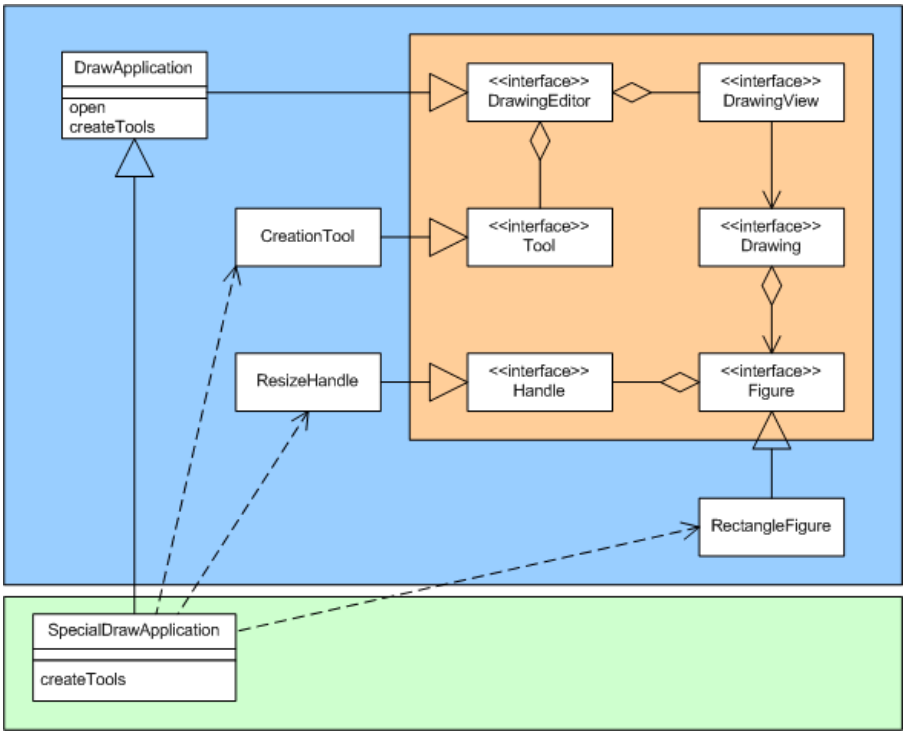


***Abbildung 6:*** *JHotDraw - Struktur und Instanziierung*

Eine zentrale Klasse des "framework internal increment" (blauer Bereich in Abbildung 6) ist die Klasse `DrawApplication`. Es handelt sich dabei um eine konkrete Klasse, die das Interface `DrawingEditor` implementiert, also ein Editor-Programm bzw. -Fenster repräsentiert. `DrawApplication` stellt bereits viel von der Funktionalität zur Verfügung, welche ein Graphik-Editorprogramm für gewöhnlich bietet. Eine Instanz dieser Klasse entspricht einem Editorfenster, welches bereits verschiedene Menüs, darunter Datei- und Edit-Menü mit deren gebräuchlichsten Auswahloptionen und ein Werkzeug zur Auswahl von Figuren zur Verfügung stellt. Neben diesem Werkzeug stellt `DrawApplication` keine weiteren Werkzeuge zur Verfügung. Insbesondere implementiert sie noch keine Möglichkeit zur Erzeugung von Figuren. Um diesbezüglich von Applikationsklassen erweitert werden zu können, definiert sie die Hook-Methode `createTools`.

Die einfachste Art der Instanziierung einer Applikation auf der Grundlage des Frameworks JHotDraw besteht darin, eine Subklasse von der Klasse `DrawApplication` abzuleiten (siehe Abbildung 6, grün unterlegter Bereich), wobei die Hook-Methode `createTools` um die Instanziierung konkreter `Tool`-Klassen, insbesondere solcher zur Figurerzeugung, erweitert wird. Abgesehen davon muss ein Hauptprogramm (`Main`-Methode) erstellt werden, in der ein Objekt dieser `DrawApplication`-Subklasse erzeugt und auf dem erzeugten Objekt die geerbte Methode `open` aufgerufen wird. Dadurch öffnet sich auf dem Bildschirm das von diesem Objekt repräsentierte Editorfenster.

Neben der Klasse `DrawApplication` stellt JHotDraw eine Vielzahl von konkreten Klassen zur Verfügung, die häufig benötigte Funktionalität zur Verfügung stellen und von einer Applikation einfach benutzt werden können.

Dazu gehören unter anderem die Klassen `CreationTool` zur Erzeugung und `ResizeHandle` zur Manipulation der Größe von Figuren, sowie die Klasse `RectangleFigure`, die die Eigenschaften rechteckiger Figuren implementiert.

# 4 ObjectTeams-basierte Restrukturierung anhand des Beispiels der Programmfunktion „Laden und Speichern einer Zeichnung“

Das folgende Kapitel beschäftigt sich mit der Fragestellung, ob die Implementierung einer Programmfunktion, die in einer bestehenden objektorientierten Software starke Streuung (code scattering) und Kopplung an die Implementierungsstrukturen anderer Programmfunktionen (code tangling) aufweist, mit Hilfe von ObjectTeams aus der bestehenden Implementierung herausgelöst werden kann und wie dabei vorzugehen ist. Desweiteren wird untersucht, ob und wenn, dann in welchen Punkten die resultierende ObjectTeams-orientierte Struktur eine Verbesserung oder eventuell auch eine Verschlechterung gegenüber der originalen objektorientierten Struktur darstellt. Dabei erfolgt die Bewertung auf zwei verschiedenen Ebenen, wobei einmal die Eigenschaften der Software in ihrer Funktion als Applikation und einmal in ihrer Funktion als Framework Berücksichtigung finden. Als Untersuchungsbeispiel dient im Rahmen dieses Kapitels die Programmfunktion „Laden und Speichern einer Zeichnung“ des Frameworks JHotDraw und deren ObjectTeams-basierte Restrukturierung.

Das Kapitel gliedert sich in folgende Unterkapitel auf:

## Inhaltsverzeichnis

4.1 Benutzersicht ..... 26
4.2 Objektorientierte Originalimplementierung ..... 26
4.3 Ansatzpunkte und Ideen für Strukturverbesserungen ..... 30
4.4 ObjectTeams-orientierte Implementierung ..... 32
4.5 Verwendung der Framework-Implementierung bei der Erstellung einer Applikation ..... 44
4.6 Fazit ..... 52
4.7 Ausblick ..... 57

## 4.1 Benutzersicht

Das Editorfenster einer auf JHotDraw aufbauenden Applikation enthält in seinem Dateimenü die Einträge "Save As..." und "Open...", mit deren Hilfe eine Zeichnung gespeichert bzw. geladen werden kann.
Die Auswahl eines der beiden Menüunterpunkte führt zur Anzeige eines Dateiauswahldialogs. Dieser enthält ein Auswahlmenü, mit dem bestimmt werden kann, in welchem Dateiformat die Zeichnung abgespeichert bzw. die Dateien welchen Typs zur Auswahl angezeigt werden sollen. Dabei ist die Voreinstellung das JHotDraw-eigene Standardformat mit der Dateiendung „draw“. Durch Eingabe eines Dateinamens bzw. Auswahl einer dort angezeigten Datei und Bestätigung durch Betätigung des entsprechenden Buttons wird der Speicher- bzw. Ladevorgang gestartet.

## 4.2 Objektorientierte Originalimplementierung

Das folgende Kapitel beschreibt detailiert die originale objektorientierte Implementierung der Programmfunktion „Laden und Speichern einer Zeichnung". Es ist in zwei große Abschnitte unterteilt. Der Fokus des ersten Abschnittes liegt auf der Implementierung der allgemeinen Grundstruktur der Programmfunktion, die immer gleich ist, unabhängig davon in welchem Dateiformat eine Zeichnung gespeichert oder aus welchem sie geladen wird. Daran schließt sich im zweiten Abschnitt eine detailierte Betrachtung der Implementierung des Lade-Speicherverfahrens an, welches Anwendung findet, wenn eine Zeichnung im JHotDraw-eigenen Standardspeicherformat gespeichert oder aus ihm geladen werden soll. Obwohl JHotDraw beim Laden und Speichern verschiedene Dateiformate unterstützt, bleibt die Darstellung im begrenzten Rahmen der vorliegenden Arbeit auf die Implementierung der Speicherung in diesem einen Format beschränkt, weil diese als einzige Ansatzmöglichkeiten für eine ObjectTeams-basierte Restrukturierung bietet.

### 4.2.1 Grundstruktur

Die Klasse DrawApplication

Die Klasse `DrawApplication` enthält die für den Benutzer sichtbaren Anteile der Programmfunktion „Laden und Speichern von Zeichnungen“. Dazu gehören die Erzeugung der Menüunterpunkte im Datei-Menü (`createFileMenu`), die Erzeugung der Dateiauswahldialoge (`createSave/OpenFileChooser`) und die Implementierung der Aktionen, die durch Auswahl der Menüunterpunkte angestoßen werden (`promptSaveAs` / `promptOpen`).
Zusätzlich zu diesen sichtbaren Anteilen enthält `DrawApplication` in ihren Methoden `loadDrawing` und `saveDrawing` die Grundgerüste der Lade- und Speicherfunktionalität.

Der im konkreten Anwendungsfall verwendete Speicheralgorithmus ist davon abhängig, in welchem Dateiformat die Zeichnung gespeichert werden soll. Er legt fest, auf welche Art und Weise die in der Zeichnung enthaltenen Objekte ausgelesen und in welcher Form die so gewonnenen Daten in die Datei geschrieben werden. Die für verschiedene Dateiformate verwendeten Speicheralgorithmen können sich recht stark voneinander unterscheiden.

Um die flexible Verwendung und Erweiterung der von JHotDraw unterstützten Speicherformate zu ermöglichen, wurde das Strategy Design Pattern (siehe [GHJV95], Seite 315ff.) eingesetzt. Das bedeutet, dass die Klasse `DrawApplication` die konkreten Speicheralgorithmen (= Speicherstrategien) nicht selbst implementiert und auch nicht wissen muss, welches Speicherformat der Benutzer im konkreten Anwendungsfall ausgewählt hat.
Für jedes von JHotDraw unterstützte Dateiformat existiert eine eigene Klasse, die den speziell auf dieses Format zugeschnittenen Speicheralgorithmus enthält. Jede derartige Klasse implementiert das Interface `StorageFormat`, welches die Speicherstrategie – Schnittstelle definiert (siehe Abbildung 7).

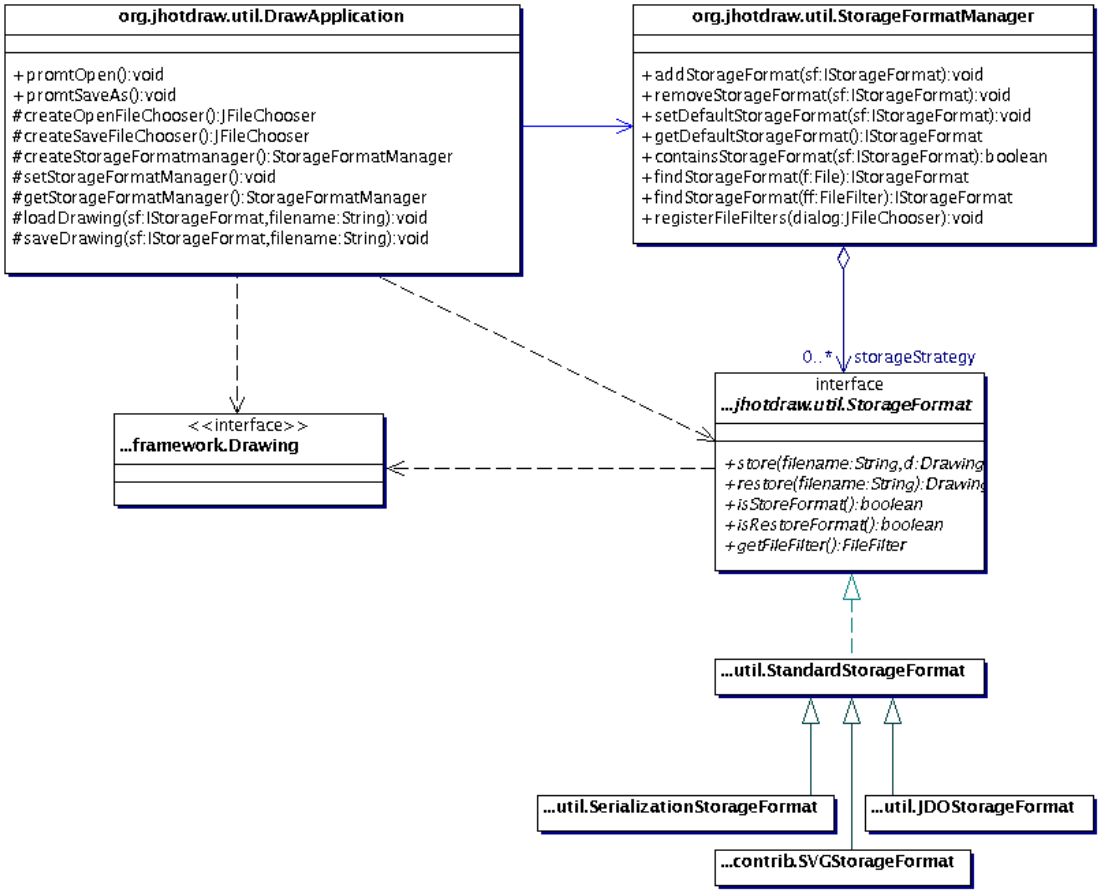


***Abbildung 7:*** *Laden / Speichern - objektorientierte Grundstruktur*

Die Verwaltung der Speicherformate übernimmt ein Objekt der Klasse `StorageFormatManager`. `DrawApplication` erzeugt es bei Programmstart (`createStorageFormatManager`) und übergibt ihm alle Speicherformate, die dem Benutzer zugänglich sein sollen (`addStorageFormat`). Danach ist es für die Registrierung der unterstützten Dateiformate im Dateiauswahldialog (`registerFileFilters`) und für das korrekte Auslesen des vom Benutzer gewählten Formats (`findStorageFormat`) verantwortlich.
Nachdem `DrawApplication` vom `StorageFormatManager` das zu verwendende `StorageFormat`-Objekt übergeben bekommen hat, ruft es auf diesem die Methode `store` auf und übergibt die zu speichernde Zeichnung als Objekt des Typs `Drawing` (siehe Abbildung 8). Alle weiteren Aktionen übernimmt das `StorageFormat`-Objekt.

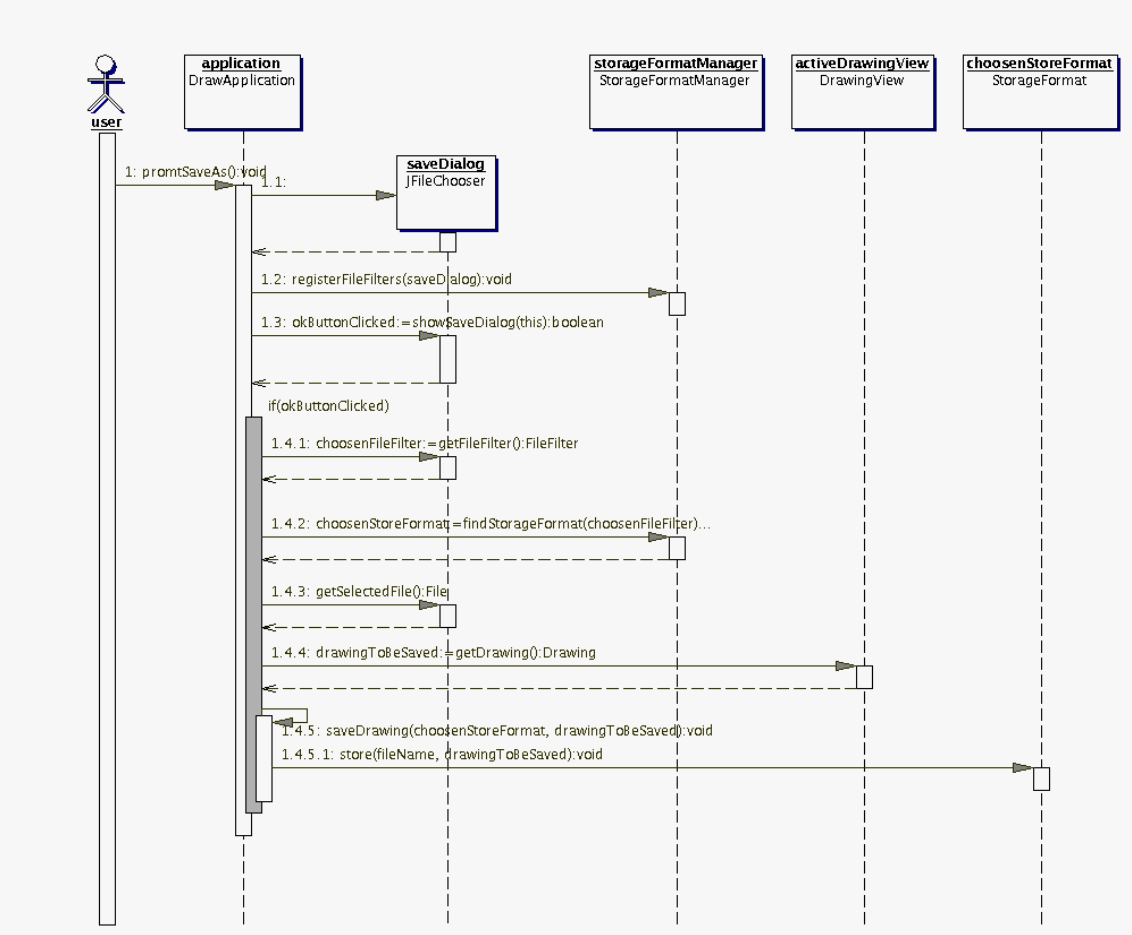


***Abbildung 8:*** *Laden / Speichern - objektorientiertes Verhalten*

Die derzeitige Version des Frameworks JHotDraw unterstützt vier verschiedene Speicherformate, die durch die StorageFormat-Implementierungsklassen `StandardStorageFormat`, `SerializationStorageFormat`, `SVGStorageFormat` und `JDOStorageFormat` repräsentiert werden und deren Vererbungshierarchie in Abbildung 7 dargestellt ist.
Im Rahmen der vorliegenden Arbeit beschränken sich die Betrachtungen der speziellen Speicherverfahren auf das durch die Klasse `StandardStorageFormat` repräsentierte Verfahren zur Speicherung von Zeichnungen im JHotDraw-eigenen Standardspeicherformat.

### 4.2.2 Das Standard-Speicherformat

Die Klasse `StandardStorageFormat` spielt gewissermaßen die Management-Rolle in der Implementierung des JHotDraw-spezifischen Speicherformats mit der zugehörigen Dateiendung "draw". `StandardStorageFormat` delegiert das Speichern von Daten in eine Datei an ein Objekt der Klasse `StorableOutput`. Das Lesen von Daten aus einer Datei und die Instanziierung von Objekten übernimmt ein Objekt der Klasse `StorableInput`. `StorableOutput` und `StorableInput` stellen Methoden zur Verfügung, um primitive Typen, `Strings`, `java.awt.Color`-Objekte und Objekte des JHotDraw-spezifischen Datentyps `Storable` in eine Datei zu schreiben bzw. aus einer Datei zu lesen (siehe Abbildung 9).
Jede Klasse, deren Objekte im Standardformat speicherbar sind, implementiert den Typ `Storable`. Insbesondere gehört dazu der Zeichnungstyp `Drawing`.

Abbildung 9 zeigt nur einen kleinen Ausschnitt der `Storable`-Vererbungshierarchie.

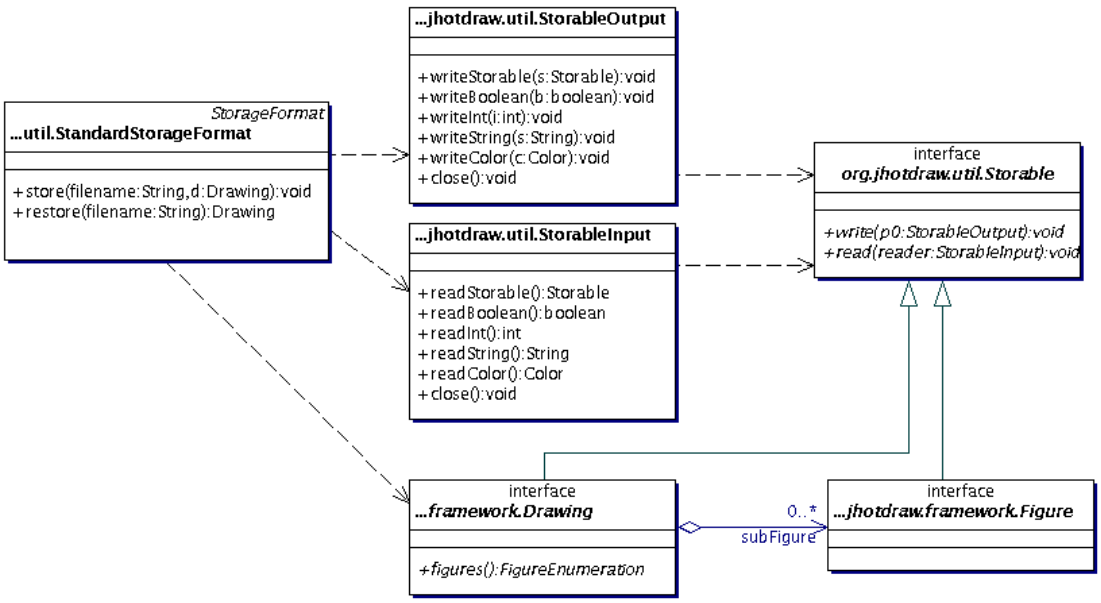


***Abbildung 9:*** *Laden / Speichern - Standardformat, objektorientierte Struktur*

Das Interface `Storable` definiert die Methoden `write(StorableOutput)` und `read(StorableInput)`. Mit der Methode `write` speichert ein `Storable`-Objekt seinen Zustand, indem es seine Attributwerte unter Zuhilfenahme des übergebenen `StorableOutput`-Objekts in eine Datei schreibt. Mit der Methode `read` stellt es seinen gespeicherten Zustand wieder her, indem es unter Zuhilfenahme des übergebenen `StorableInput`-Objekts Werte einliest und seinen Attributen zuweist. Welcher der eingelesenen Werte zu welchem Attribut gehört, ergibt sich durch die beim Schreiben verwendete Reihenfolge. Besitzt ein Attribut einen nicht primitiven Typen außer `String` oder `Color`, kann es, wie bereits erwähnt, nur dann gespeichert werden, wenn dieser seinerseits auch das Interface `Storable` implementiert.
Wird auf einem `Storable`-Objekt die Methode `write` aufgerufen, ruft es für alle seine Attribute vom Typ `Storable` die Methode `writeStorable` des `StorableOutput`-Objekts auf. Die Methode `writeStorable` tut nichts anderes, als auf dem übergebenen `Storable`-Objekt wiederum die Methode `write` aufzurufen. Das führt zu einer Kaskade abwechselnder Aufrufe von `StorableOutput.writeStorable(Storable)` und `Storable.write(StorableOutput)`, so lange bis die zu speichernden Objekte nur noch Attribute der anderen oben erwähnten Typen besitzen. Abbildung 10 versucht, diesen Mechanismus darzustellen. Bedingt durch diese Technik genügt der von `StandardStorageFormat.store` initiierte Aufruf von `StorableOutput.writeStorable` mit Übergabe des `Drawing`-Objekts, um die gesamte Zeichnung zu speichern (siehe Abbildung 10). Das Laden funktioniert analog.

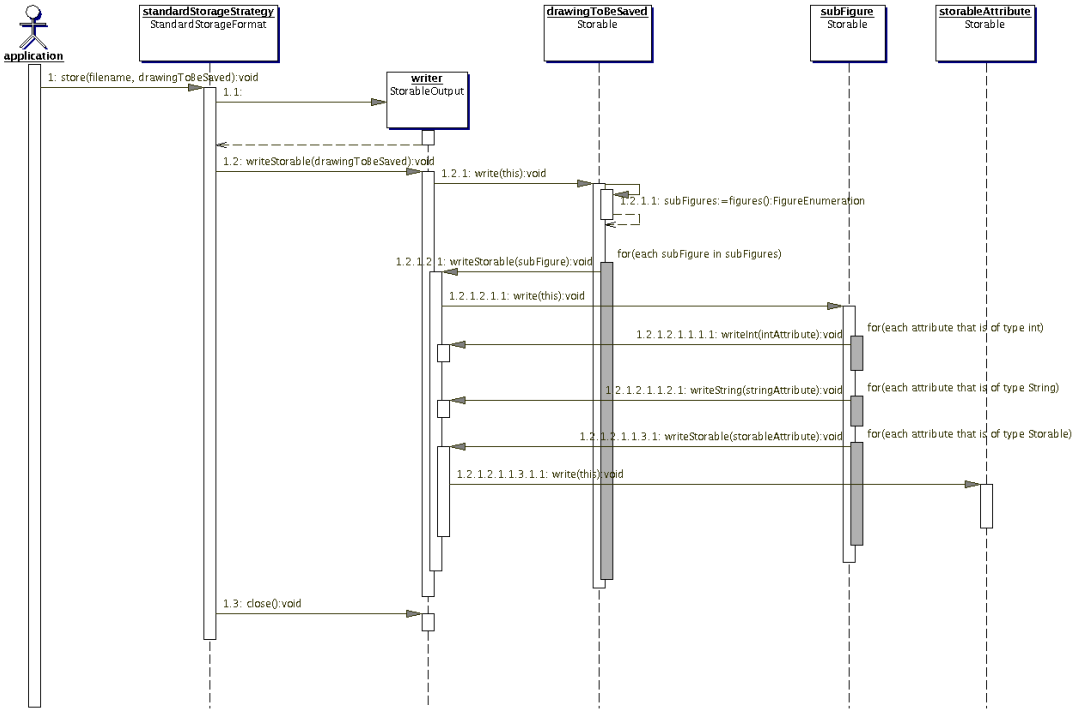


***Abbildung 10:*** *Laden / Speichern - Standardformat, objektorientiertes Verhalten*

Bevor innerhalb der Methode `StorableOutput.writeStorable` auf dem übergebenen `Storable`-Objekt write aufgerufen wird, damit es seine Attributwerte in die Datei schreiben kann, wird der tatsächliche (dynamische) Typ des `Storable`-Objekts ermittelt und der Typname als `String` in die Datei geschrieben. Beim Laden wird diese Information von `StorableInput.readStorable` gelesen und mit Hilfe von Java-Reflection-Mechanismen ein Objekt des entsprechenden Typs erzeugt. Zu diesem Zweck benötigt jede Klasse, deren Objekte im Standardformat speicherbar sein sollen, zusätzlich zu den Methoden `write` und `read` einen parameterlosen Konstruktor.

## 4.3 Ansatzpunkte und Ideen für Strukturverbesserungen

Nachdem im vorangegangenen Kapitel ausführlich die originale objektorientierte Implementierung der Programmfunktion „Laden und Speichern einer Zeichnung" beschrieben wurde, soll im folgenden Kapitel herausgearbeitet werden, welche strukturellen Schwachstellen diese aufweist. Dabei liegt der Fokus auf dem Vorkommen von Streuung (code scattering) und/oder Kopplung (code tangling). Diese Schwachstellen dienen als Ansatzpunkte für den Versuch, die Struktur der Implementierung mit Hilfe von ObjectTeams besser zu modularisieren, in der Hoffnung, dadurch die Programmeigenschaften zu verbessern. Im Rahmen der Erläuterung der Schwachstellen werden erste ObjectTeams-basierte Restrukturierungsideen vorgestellt. In Anlehnung an die Untergliederung des vorhergehenden Kapitels betrachtet auch dieses Kapitel die Implementierungen der Grundstruktur des Speichervorgangs und des speziellen Verfahrens zur Speicherung im Standardspeicherformat getrennt voneinander.

### 4.3.1 Grundstruktur

Der unabhängig vom konkreten Dateiformat immer gleiche grundsätzliche Speichervorgang wird wie in Kapitel 4.2.1 auf Seite 26 beschrieben durch ein Zusammenspiel der Klassen `DrawApplication`, `StorageFormatManager` und des Interfaces `StorageFormat` realisiert. Dabei stellt die Klasse `DrawApplication` eine strukturelle Schwachstelle dar. Sie enthält nicht nur die im Abschnitt „Die Klasse DrawApplication“ auf Seite 26 beschriebenen und in Abbildung 7 dargestellten Implementierungselemente der Lade-Speicher-Programmfunktion, sondern auch solche vieler anderer Programmfunktionen. Sie enthält "code tangling". Dabei gehören die Aufgaben, die sie im Rahmen des Ladens und Speicherns übernimmt, nicht zu ihrem Kernaufgabenbereich, sondern sind lediglich Teilfunktionalitäten, die sie im Rahmen ihrer Rolle bei der Realisierung dieser einen Programmfunktion wahrnimmt. Deshalb dient die Klasse `DrawApplication` als Ansatzpunkt für den Versuch einer ObjectTeams-basierten Strukturverbesserung.
Die zugrundeliegende Idee ist folgende. Mit Hilfe von ObjectTeams kann die eben beschriebene Teilfunktionalität, die `DrawApplication` aufgrund ihrer Rolle bei der Realisierung der Lade-Speicher-Programmfunktion besitzt, strukturell in Form einer Rollenklasse gekapselt werden. Zu diesem Zweck werden alle Implementierungselemente (Attribute, Methoden, Teile von Methoden), die dieser Teilfunktionalität zuzuordnen sind, aus der Klasse `DrawApplication` entfernt und in eine Rollenklasse verschoben, die an `DrawApplication` gebunden wird. In diesem Zusammenhang wird eine Team-Klasse als Kontekt für die Rollenklasse benötigt. Diese Team-Klasse repräsentiert den Kontext der Programmfunktion "Laden und Speichern einer Zeichnung".
Die anderen beiden Akteure, die an der Realisierung des Ladens und Speicherns beteiligt sind, die Klasse `StorageFormatManager` und das Interface `StorageFormat` referenzieren sich gegenseitig. Darüber hinaus werden sie nur von der Klasse `DrawApplication` und zwar auch nur von den Anteilen referenziert, die im Rahmen einer ObjectTeams-basierten Restrukturierung in eine Rolle verschoben werden können. Das heißt, nach einer Restrukturierung würden sie, abgesehen von der gegenseitigen Referenzierung, ausschließlich aus dem Team heraus zugegriffen werden. Da sie keine Implementierungselemente anderer Programmfunktionen, also kein "code tangling" enthalten, sie inhaltlich also vollständig der dem Kontext der Programmfunktion "Laden und Speichern einer Zeichnung" zugeordnet werden können, ist es daher sinnvoll, sie vollständig in das Team zu verschieben, das diesen Kontext repräsentiert.
Sollte dieser Ansatz realisierbar sein, das heißt nicht an Schwierigkeiten, die an dieser Stelle noch nicht absehbar sind, scheitern, würde das bedeuten, dass die gesamte Objektkollaboration, die die Lade-Speicher-Programmfunktion realisiert, strukturell in Form eines Teams gekapselt werden kann.

### 4.3.2 Das Standard-Speicherformat

Wie in Kapitel 4.2.2 auf Seite 28 beschrieben, muss jede Klasse, deren Objekte im Standardformat speicherbar sein sollen, das Interface `Storable` implementieren. Das führt dazu, dass 10 Interfaces und 67 Klassen des Frameworks Bestandteil der Hierarchie des Typs `Storable` sind. 7 Interfaces beerben `Storable` direkt, 3 indirekt. 36 Klassen besitzen eigene Implementierungen der Methoden `read` und `write`, 31 Klassen implementieren das Interface indirekt über ihre Superklassen. Es liegt also

einerseits "code tangling" vor, da die Klassen zusätzlich zu ihren Kernaufgaben ihre eigene Speicherung implementieren. Andererseits handelt es sich um massives "scattering", weil die Implementierung des Speicherverfahrens für das Standardformat über viele Klassen und Pakete des Frameworks verteilt ist.

Der Ansatz zur Strukturverbesserung besteht darin, die Interface-Hierarchie des Typs Storable in ein Team zu extrahieren. Das bedeutet, dass eine Klasse das Interface Storable und damit die Methoden read und write nicht mehr selbst implementieren muss, wenn ihre Objekte im Standardformat speicherbar sein sollen. Stattdessen erhält jede Klasse die in der objektorientierten Framework-Version das Interface Storable implementiert, in der ObjectTeams-Version eine an sie gebundene Rolle, die diese Aufgabe übernimmt.

## 4.4 ObjectTeams-orientierte Implementierung

### 4.4.1 Grundstruktur

Wie in Kapitel 4.2.1 auf Seite 26 beschrieben, zeichnen die Klassen `DrawApplication`, `StorageFormatManager` und das Interface `StorageFormat` verantwortlich für die Umsetzung der Lade- und Speicherfunktionalität. Während die beiden letztgenannten vollständig der Realisierung dieser Programmfunktionalität dienen, enthält `DrawApplication` auch Implementierungsanteile anderer Programmfunktionen, beinhaltet also "code tangling". Der Versuch, die Programmstruktur mit Hilfe von ObjectTeams zu verbessern, setzt an dieser Stelle an.

DrawApplication und ihre Rolle StorageManager

Die Klasse `DrawApplication` kann aufgeteilt werden, indem die Implementierungsanteile der Lade/Speicher-Programmfunktion in eine an `DrawApplication` gebundene Rolle extrahiert werden. Da `DrawApplication` in der ursprünglichen Framework-Implementierung die zentrale Steuerungsrolle bezüglich Laden und Speichern spielt, heißt die neue Rollenklasse dieser Funktion entsprechend `StorageManager`.
Mit Ausnahme der Methoden `createFileMenu` und `open` können alle im Abschnitt „Die Klasse DrawApplication“ auf Seite 26 beschriebenen und in Abbildung 7 gezeigten Methoden der Klasse `DrawApplication` vollständig der zu extrahierenden Programmfunktion zugeordnet und daher in die Rolle `StorageManager` verschoben werden.
Die Methoden der Klasse `DrawApplication`, die von den in die Rolle verschobenen Methoden aber auch außerhalb des Kontexts Laden und Speichern benötigt werden, dürfen nicht in die Rolle verschoben werden. Stattdessen erhalten sie in der Rollenklasse eine Art Stellvertreter-Methode, die abstrakt definiert und per callout an die Originalmethode gebunden wird. Weichen die Namen der Rollenmethoden von denen der an sie gebundenen Originalmethoden ab, müssen die Methodenaufrufe innerhalb der Rolle angepasst werden. Dies ist bei den in Abbildung 11 auf Seite 34 gezeigten Methoden `getDrawingView` und `openNewWindow` der Fall.

Menüerweiterung

Die Methode `createFileMenu` erzeugt alle Menüunterpunkte des Datei-Menüs. Dazu gehören nicht nur die Einträge für Laden und Speichern, sondern auch solche anderer Programmfunktionen. Daher wurde die Methode `createFileMenu` in einen Basis- und einen Rollenteil aufgeteilt. Die Erzeugung der Menüeinträge für Laden und Speichern wurde in die neue Rollenmethode `extendMenu` verschoben. Diese erweitert, wie ihr Name verrät, das von der Basisklasse erzeugte Menü. Sie wird per replace callin an die

Ursprungsmethode `createFileMenu` gebunden. Abbildung 11 zeigt die Rolle StorageManager, ihre Methoden und deren Bindungen im Überblick.

StorageFormatManager und StorageFormat

Im Rahmen der ObjectTeams-orientierten Restrukturierung gibt es verschiedene Möglichkeiten, mit der Klasse `StorageFormatManager` und dem Interface `StorageFormat`, sowie mit den Methoden `open` und `createStorageFormatManager` der Klasse `DrawApplication` umzugehen. Diese werden im folgenden diskutiert.

### *Erster Design-Ansatz*

Verschiebung von StorageFormatManager und StorageFormat

Die Klasse `StorageFormatManager` und das Interface `StorageFormat` enthalten keine Implementierungselemente anderer Programmfunktionen außer Laden und Speichern. Desweiteren werden sie ausschließlich von der Rolle `StorageManager` verwendet. Im Hinblick auf diese Merkmale ist der nahe liegendste Ansatz, beide Klassen in das Team zu verschieben, in dem sich `StorageManager` befindet.

Implementierungsklassen folgen dem Interface ins Team

Wenn das Interface `StorageFormat` ins Team verschoben wird, ist es danach ein Rollentyp. Das hat zur Folge, dass es nur noch von Klassen beerbt oder implementiert werden kann, die auch Rollen sind und sich im selben Team oder in einem Subteam befinden. Alle Klassen, die `StorageFormat` implementieren, müssen daher mit ins Team verschoben werden. Die resultierende Struktur ist in Abbildung 11 dargestellt.

Attributinitialisierung

In der objektorientierten Implementierung hält die Klasse `DrawApplication` ein Attribut des Typs `StorageFormatManager`. Dieses wird in der Initialisierungsmethode `open` erzeugt (`createStorageFormatManager`) und gesetzt (`setStorageFormatManager`).
Da das Attribut im Rahmen der ObjectTeams-orientierten Restrukturierung in die Rollenklasse `StorageManager` verschoben wurde, müssen auch Erzeugung und Zuweisung des entsprechenden Objekts innerhalb der Rollenklasse vorgenommen werden. Die Verschiebung der Methode `open` in die Rollenklasse ist dabei keine Option, da diese vielfältige Aufgaben wahrnimmt, die nicht zum Kontext des Ladens und Speicherns gehören.
Es gibt nun zwei verschiedene Möglichkeiten, diese Funktionalität aus der Methode `DrawApplication.open` in die Rollenklasse `StorageManager` zu extrahieren.
Die erste Möglichkeit besteht darin, sie in eine neue Rollenmethode zu verschieben, die per before oder after callin an die Basismethode `open` gebunden wird.
Daneben gibt es eine zweite Möglichkeit, die als konzeptuell besser zu bewerten ist. Da das `StorageFormatManager`-Objekt in der ObjectTeams-orientierten Implementierung ein Attribut der Rollenklasse ist, der geeignetste Ort für die Initialisierung eines Attributs eine Initialisierungsmethode ist und jede Rollenklasse implizit eine Initialisierungsmethode (`initializeRole`) besitzt, die von der ObjectTeams/Java-Laufzeitumgebung automatisch aufgerufen wird, scheint es am sinnvollsten, Erzeugung und Setzen des StorageFormatManager-Attributs in diese Methode zu verschieben.

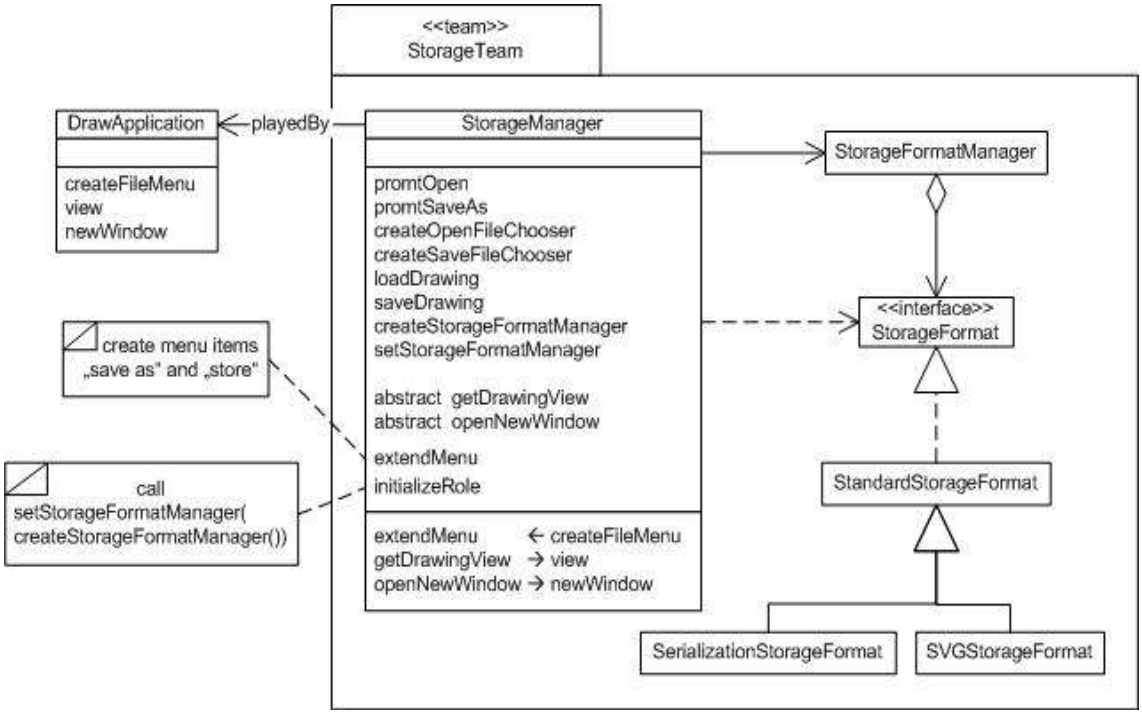


***Abbildung 11:*** *Laden/Speichern - ObjectTeams-orientiertes Design, Ansatz 1*

Vorteil

Der große Vorteil dieses ersten Design-Ansatzes ist seine Einfachheit. Fast alle betroffenen Klassen und Methoden können in sich so bleiben, wie sie sind, und müssen lediglich verschoben werden.

Nachteile und Beschränkungen

Als Nachteil könnte gewertet werden, dass dieses Design die Implementierungselemente der eigentlichen Grundfunktionalität des Ladens und Speicherns in einem gewissen Maß an die Implementierung der verschiedenen speziellen Speicheralgorithmen bindet. Man könnte argumentieren, dass es damit der durch die Anwendung des Strategy Design Patterns erreichten Entkopplung entgegenwirkt.

Im Zusammenhang mit der im Rahmen dieser Diplomarbeit verwendeten ObjectTeams/Java-Version hat die Existenz des Interface `StorageFormat` innerhalb des Teams aber noch einen schwerwiegenderen Nachteil. Die ObjectTeams/Java-Implementierung war insofern beschränkt, als dass eine Rollenklasse nicht selbst wieder eine Teamklasse sein durfte (siehe Abschnitt „Team-Schachtelung“ auf Seite 16). Wie in Kapitel 4.4.2 noch erläutert werden wird, wurde im Rahmen der ObjectTeams-orientierten Restrukturierung die `StorageFormat`-Implementierungsklasse `StandardStorageFormat`, in eine Teamklasse umgewandelt. Das wäre aufbauend auf der eben geschilderten Struktur nicht möglich gewesen.

### *Zweiter Design-Ansatz*

Das Interface StorageFormat und die Rollenklasse RStorageFormat

Um diese Nachteile zu vermeiden, wurde ein alternativer Ansatz gewählt, der es erlaubt, die Lade/Speicher-Grundfunktionalität als solche zu kapseln. Dabei wurde das Interface `StorageFormat` nicht ins Team verschoben, sondern an eine zusätzlich erstellte Rollenklasse des Teams gebunden. Diese Rollenklasse besitzt genau die Methoden, die in ihrer Basis, dem Interface `StorageFormat`, deklariert sind, wobei jede Rollenmethode per callout an die gleichnamige Methode der Basis gebunden ist. Die Rolle dient einzig dem Zweck, Methodenaufrufe an ihre Basis weiterzuleiten. Dadurch wird dem Typ `StorageFormat` ermöglicht, sowohl außerhalb des Teams als auch virtuell im

Team zu existieren.
Daher ist der sinnvollste Name für die an das Interface `StorageFormat` gebundene Rollenklasse auch `StorageFormat`. Im Rahmen einer Implementierung in ObjectTeams/Java ist diese Benennung möglich. Allerdings ist dabei zu beachten, dass Team-intern das Basis-Interface mit voll qualifiziertem Namen angesprochen werden muss, damit der Compiler Rollen- und Basistyp voneinander unterscheiden kann.
Um bessere Verständlichkeit des Textes zu erreichen, erhielt die Rolle im Rahmen der vorliegenden Darstellung aber den Namen `RStorageFormat`, wobei das R für Rolle steht. Die Klassen `StorageFormatManager` und `StorageManager` arbeiten bei dieser Konstellation nicht auf dem Typ `StorageFormat`, sondern auf dem Rollentyp `RStorageFormat`. Abbildung 12 zeigt die beschriebene Struktur.

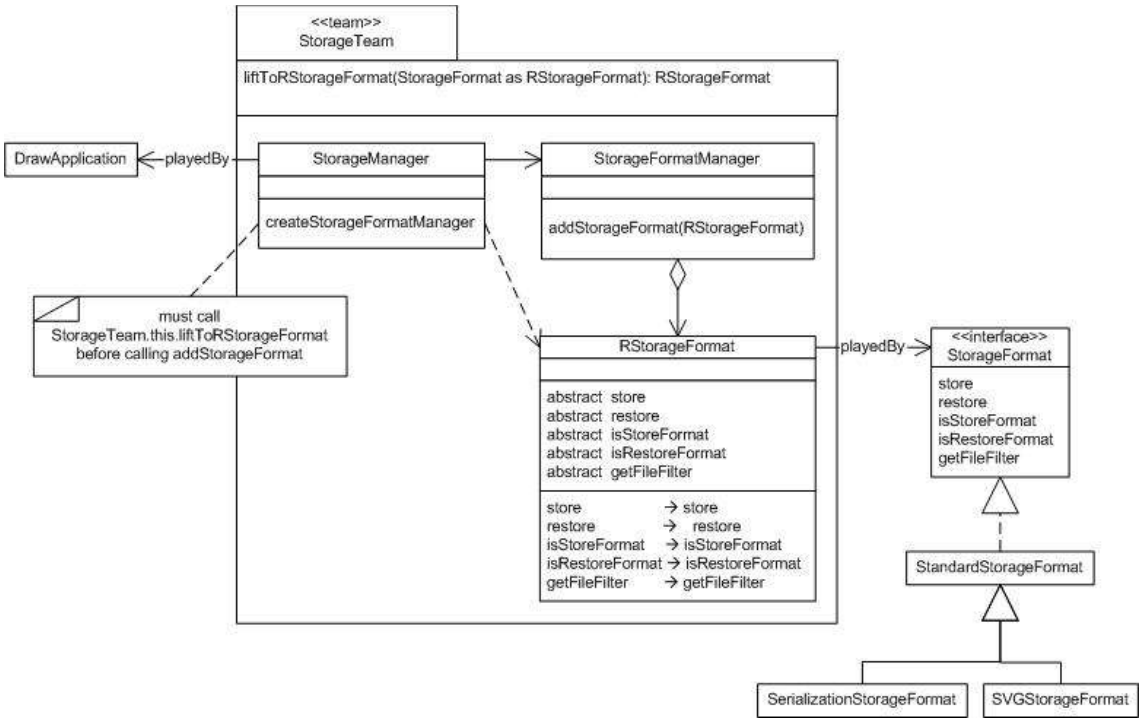


***Abbildung 12:*** *Laden/Speichern - ObjectTeams-orientiertes Design, Ansatz 2*

Vorteil

Dieser Ansatz macht es einerseits möglich, dass die Klassen, die das Interface `StorageFormat` implementieren, selbst Teamklassen sein können. Andererseits läßt er aber ein neues Problem entstehen.

Neuer Nachteil: explizites Lifting erforderlich

Zum Aufgabenbereich der Rolle `StorageManager` gehört es, im Rahmen der Erzeugung ihres `StorageFormatManager`-Objekts in der Methode `createStorageFormatManager` auch die von diesem zu verwaltenden `StorageFormat`-Objekte zu erzeugen und ihm mit der Methode `addStorageFormat` zu übergeben. Diese Objekte sind Instanzen der Implementierungsklassen des Interfaces `StorageFormat` und befinden sich wie dieses auch außerhalb des Team-Kontexts. Da sich der Typ `StorageFormat` außerhalb des Teams befindet, ist er dem `StorageFormatManager` aber nur in seiner gelifteten Form als Rollentyp `RStorageFormat` bekannt. Daher erwartet dieser kein `StorageFormat`- sondern ein `RStorageFormat`-Objekt.
Daraus resultiert, dass der `StorageManager` jedes erzeugte

`StorageFormat`-Objekt explizit liften muss, bevor es dem `StorageFormatManager` übergeben werden kann. Zu diesem Zweck muss das Team eine Methode zur Verfügung stellen, die ein `StorageFormat`-Basisobjekt zu einem `RStorageFormat`-Rollenobjekt liftet (`liftToRStorageFormat`). Abbildung 13 zeigt das beschriebene Vorgehen.

```
StorageManager.createStorageFormatManager
{
    StorageFormatManager sfm = new StorageFormatManager();

    sfm.addStorageFormat(
        StorageTeam.this.liftToRStorageFormat(
            new StandardStorageFormat()))
    ...
}
```

***Abbildung 13:*** *Erzeugung und Lifting eines StorageFormat-Objekts beim zweiten Design-Ansatz*

Dieses Vorgehen hat zum einen, aus einem ganz praktischen Blickwinkel betrachtet, den Nachteil, dass es umständlich und der Quelltext schwerer lesbar ist. Zum anderen ist es aber auch auf konzeptueller Ebene problematisch. Die Tatsache, dass eine Rollenklasse Typen, die außerhalb des Teamkontexts existieren und innerhalb eine Rolle besitzen, selbst instanziiert und explizit liftet, durchbricht die Funktionskapselung, die durch ein Team erreicht werden soll.

### *Dritter Design-Ansatz*

Der in diesem Abschnitt unter der Bezeichnung "Drittes Design" vorgestellte Ansatz ist keine eigenständige umfassende Modellierung der Klasse `StorageTeam`, sondern kann sowohl in Kombination mit dem im Abschnitt "Erster Design-Ansatz" als auch mit dem im Abschnitt "Zweiter Design-Ansatz" vorgestellten Ansatz zur Handhabung des Typs `StorageFormat` verwendet werden. Er wird in Kapitel 5.3.2„Erweiterung der Team-internen Strukturierungsmöglichkeiten durch zweites Team-Handhabungs-Verfahren" auf Seite 75 ausführlich erläutert und ist in seiner Anwendbarkeit insofern beschränkt, als dass er nur verwendet werden darf, wenn das `StorageTeam` auf die im Abschnitt 5.2„Zweites Team-Handhabungs-Verfahren“ auf Seite 68 vorgestellte Art und Weise instanziiert und aktiviert wird, und dadurch für jedes `DrawApplication`-Basisobjekt eine eigene `StorageTeam`-Instanz vorhanden ist. Bei diesem Ansatz enthält `StorageTeam` keine ungebundene Rollenklasse `StorageFormatManager` sondern übernimmt deren Funktionalitäten selbst in Form von Teamlevel-Methoden (siehe Abbildung 14).

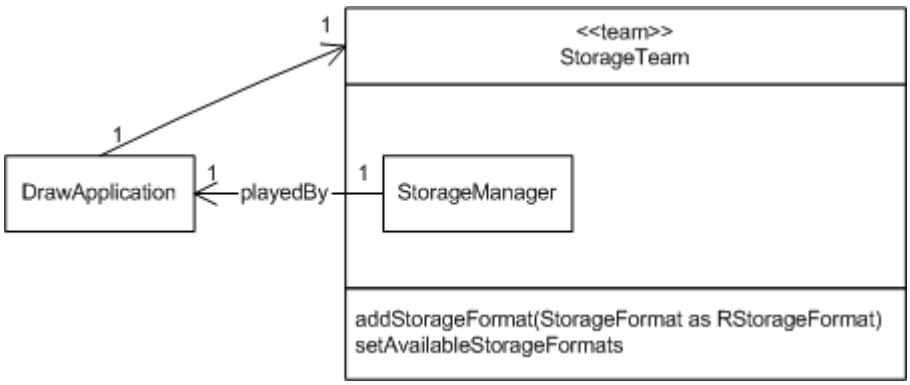


***Abbildung 14:*** *Laden/Speichern - ObjectTeams-orientiertes Design Ansatz 3*

Deklariertes versus explizites Lifting

Aufbauend auf dem in Abschnitt "Zweiter Design-Ansatz" vorgestellten Ansatz können dessen Schwächen behoben werden. Da die Methode `addStorageFormat` beim dritten Design-Ansatz eine Teamlevel-Methode ist, hat sie die Möglichkeit, in ihrer Signatur Parameter-Liftings zu deklarieren ("declared lifting") und kann erweitert werden zu `addStorageFormat (StorageFormat as RStorageFormat)`. Dadurch kann das Problem beseitigt werden, dass die Rolle `StorageManager` die erzeugten `StorageFormat`-Objekte selbst explizit liften muss.
Um zusätzlich das Problem zu beseitigen, dass die Rolle `StorageManager` das außerhalb des Teamkontexts existierende Interface `StorageFormat` und dessen Implementierungsklassen überhaupt kennen muss, kann die Verantwortung für die Instanziierung der `StorageFormat`-Objekte der Rolle `StorageManager` entzogen und der Teamklasse übertragen werden. Zu diesem Zweck kann das Team um die Methode `setAvailableStorage-Formats` erweitert werden. Das resultierende Verhalten zeigt Abbildung 15.

```
public team class StorageTeam
{
    void setAvailableStorageFormats()
    {   ...
        addStorageFormat(new StandardStorageFormat());
        ...
    }

    void addStorageFormat(StorageFormat as RStorageFormat){...}
}
```

***Abbildung 15:*** *Erzeugung und Lifting eines StorageFormat-Objekts beim dritten Design-Ansatz*

### *Speicherformate*

Die derzeitige Version des Frameworks JHotDraw unterstützt mehrere Speicherformate, die durch die in Abbildung 12 dargestellten `StorageFormat`-Implementierungsklassen repräsentiert werden. In der objektorientierten Originalimplementierung des Frameworks implementiert die Klasse `StandardStorageFormat` das allen `StorageFormat`-Implementierungs-klassen gemeinsame Verhalten und dient daher als Superklasse für die anderen.
Da, wie im folgenden Kapitel ausführlich beschrieben werden wird, im Rahmen

der ObjectTeams-orientierten Restrukturierung die Klasse `StandardStorageFormat` in eine Team-Klasse umgewandelt (und zu `StandardStorageFormatTeam` umbenannt wurde) und Team-Klassen ausschließlich untereinander in einer Vererbungsbeziehung stehen dürfen, musste eine entsprechende Anpassung der `StorageFormat`-Typhierarchie vorgenommen werden. Dafür gibt es zwei Möglichkeiten.

Die eine Möglichkeit besteht darin, die Struktur der Hierarchie so zu belassen, wie sie ist (siehe Abbildung 12 auf Seite 35), und um dies zu ermöglichen, alle `StorageFormat`-Implementierungsklassen zu Team-Klassen zu machen (Schlüsselwort `team` hinzufügen). Dadurch würden alle `StorageFormat`-Implementierungsklassen implizit die in `StandardStorageFormatTeam` enthaltenen Rollenklassen erben. Da die erbenden Klassen für diese (mehr als 80!) Rollenklassen keinerlei Verwendung haben, ist dieses Vorgehen konzeptuell fragwürdig.

Daher wurde ein alternativer Ansatz gewählt. Das allen `StorageFormat`-Implementierungsklassen gemeinsame Verhalten wurde aus der Klasse `StandardStorageFormatTeam` in die neue Klasse `AbstractStorageFormat` verschoben. Diese nimmt den Platz in der `StorageFormat`-Hierarchie ein, den `StandardStorageFormat` zuvor innehatte. Damit `StandardStorageFormatTeam` die Funktionalität dieser neuen Klasse nutzen kann, obwohl es ihr als Team-Klasse nicht möglich ist, diese zu beerben, hält es eine Referenz auf ein Objekt dieser Klasse. Dafür ist es notwendig, dass `AbstractStorageFormat` entgegen ihres Namens instanziierbar ist. Abbildung 16 zeigt die resultierende Hierarchie-Struktur.

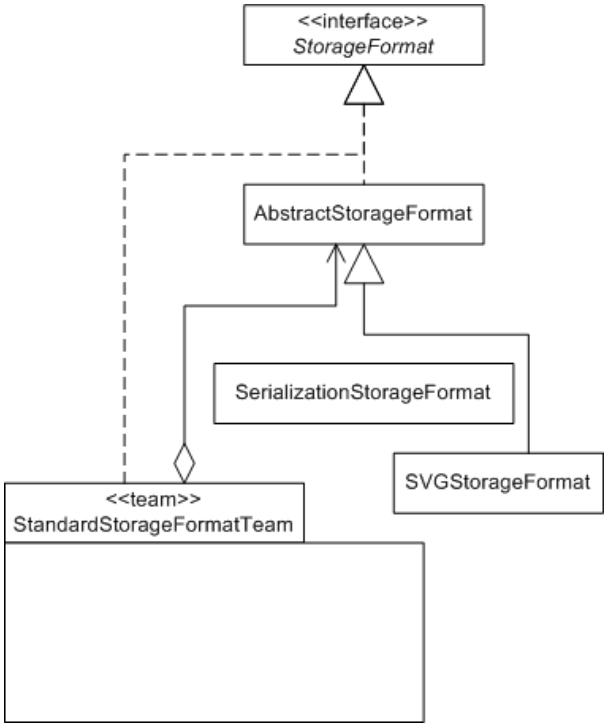


***Abbildung 16:*** *StorageFormat-Hierarchie, ObjectTeams-orientiert*

### 4.4.2 Das Standard-Speicherformat

Ziel der Restrukturierung

Wie bereits im Abschnitt 4.3.2 geschildert, besteht das Problem der objektorientierten Implementierung der Programmfunktion "Laden/Speichern im Standardspeicherformat" darin, dass sie über das System verstreut liegt (code scattering) und sich in Klassen befindet, die andere Kernaufgaben haben (code tangling). Ziel der Restrukturierung ist daher, die Implementierungselemente der Programmfunktion zu kapseln.

#### *Das Design*

Der Rollentyp Storable

Um dieses Ziel zu erreichen, wurde die Interface-Hierarchie des Typs `Storable` in ein Team extrahiert. Jede Klasse, die in der objektorientierten Framework-Version das Interface `Storable` und damit die Methoden `read` und `write` implementiert, besitzt in der ObjectTeams-orientierten Version eine an sie gebundene Rolle, die diese Aufgabe für sie übernimmt. Da die Funktion der Rolle darin besteht, das Interface `Storable` zu implementieren, wurden die Methoden `read` und `write` aus der Basisklasse entfernt und in die Rollenklasse verschoben. Ein Rollenobjekt liest in der Methode `write` die Attributwerte seines Basisobjekts aus, um sie in eine Datei zu schreiben. In der Methode `read` initialisiert es sein Basisobjekt und setzt dessen Attribute auf Werte, die es zuvor aus einer Datei gelesen hat. Der für diese Operationen notwendige Zugriff auf die Attribute der Basisklasse, erfolgt dabei über callout auf `get`- und `set`-Methoden. Wo dies erforderlich war, wurde die Basisklasse um entsprechende Zugriffsmethoden erweitert.

Besonderheiten der Hierarchie des Rollentyps Storable

Bezüglich der Extraktion der Interface-Hierarchie des Typs `Storable` sind folgende Punkte zu beachten:
Die Methode `write` dient, wie eben erwähnt, dazu, alle Attribute einer Klasse auszulesen, um sie in eine Datei zu schreiben. Dazu gehören auch die von eventuell vorhandenen Elternklassen geerbten Attribute. Deswegen enthalten die `write`-Methoden fast aller Klassen, die das Interface `Storable` implementieren, einen Aufruf der `write`-Methode ihrer Elternklasse ("super call"). Der Einlese-Mechanismus der Methode `read` funktioniert analog.
Damit diese Supermethoden-Aufrufe nach der Verschiebung der Methoden `read` und `write` in die Rollen weiterhin funktionsfähig sind, das heißt, eine Elternrolle vorhanden ist und die Attribute der richtigen Basisklasse gelesen werden, muss die Struktur der Vererbungshierarchie des Typs `Storable` erhalten bleiben. Das bedeutet, dass die Rollen untereinander die selbe Vererbungshierarchie-Struktur haben müssen, wie die an sie gebundenen Basisklassen.
Um gewährleisten zu können, dass alle in der Originalimplementierung des Standardspeicheralgorithmus vorgefundenen Castings auch in der ObjectTeams-orientierten Implementierung entsprechend für die Rollen funktionieren, müssen auch alle Interfaces, die in der objektorientierten Version das Interface Storable beerben, und alle Klassen, die das Interface nur indirekt implementieren, also die Methoden read und write nicht selbst enthalten, in der ObjectTeams-orientierten Version eine an sie gebundene Rollenklasse in der Storable-Hierarchie besitzen.

Die ungebundenen Rollenklassen StorableInput und StorableOutput

Die Klassen `StorableInput` und `StorableOutput` sind die einzigen, die auf dem Datentyp `Storable` arbeiten. Sie benötigen Zugriff auf den Typ `Storable`, der sich in der ObjectTeams-orientierten Implementierung in einem Team befindet. Da `StorableInput` und `StorableOutput` ausschließlich

dem Zweck dienen, Daten zu lesen und zu schreiben, also kein "code tangling" enthalten, wäre die Aufteilung in einen Basisteil und einen Rollenteil nicht sinnvoll gewesen. Stattdessen wurden beide Klassen in das Team verschoben, in dem sich der Typ `Storable` befindet.

Die Teamklasse StandardStorage-Format

Die Klasse `StandardStorageFormat` bekommt in ihrer Methode `store` ein Objekt des Typs `Drawing` übergeben. In der objektorientierten Implementierung wird dieses einfach an die Methode `StorableOutput.writeStorable` weitergereicht (siehe Abbildung 10 auf Seite 30). Obwohl die Methode `writeStorable` ein Objekt des Typs `Storable` erwartet, funktioniert dieses Vorgehen, weil der Typ `Drawing` ein Untertyp des Typs `Storable` ist.
In der ObjectTeams-orientierten Implementierung funktioniert diese Strategie nicht, weil `Drawing` kein Subtyp von `Storable` ist. Stattdessen existiert eine an `Drawing` gebundene Rollenklasse, die Subtyp von `Storable` ist (`StorableDrawing`), und das `Drawing`-Objekt muss vor der Weiterleitung an `StorableOutput.writeStorable` geliftet werden. Analog dazu muss beim Laden einer Zeichnung das von `StorableInput.readStorable` gelieferte Objekt in der Methode `StandardStorageFormat.restore` gelowert werden, bevor diese es als `Drawing`-Objekt zurückgeben kann.
`StandardStorageFormat` ist fast (siehe Abschnitt „Externalized Roles“ auf Seite 41) die einzige Klasse, die Zugriff auf `StorableInput` und `StorableOutput` benötigt. Sie muss die Übersetzung (Lifting und Lowering) zwischen Basisobjekten und Objekten des Rollentyps `Storable` vornehmen. Damit kann sie als logischer Kontext für die Verwendung der Typen `Storable`, `StorableInput` und `StorableOutput` betrachtet werden. Es war daher naheliegend, die Klasse `StandardStorageFormat` auch zum Implementierungskontext dieser Rollentypen zu machen. Deshalb wurde sie in ein Team umgewandelt. Die Signatur der Methode store wurde um das in diesem Abschnitt besprochene Parameterlifting erweitert (`StandardStorageFormatTeam.store(Drawing as Storable)`).

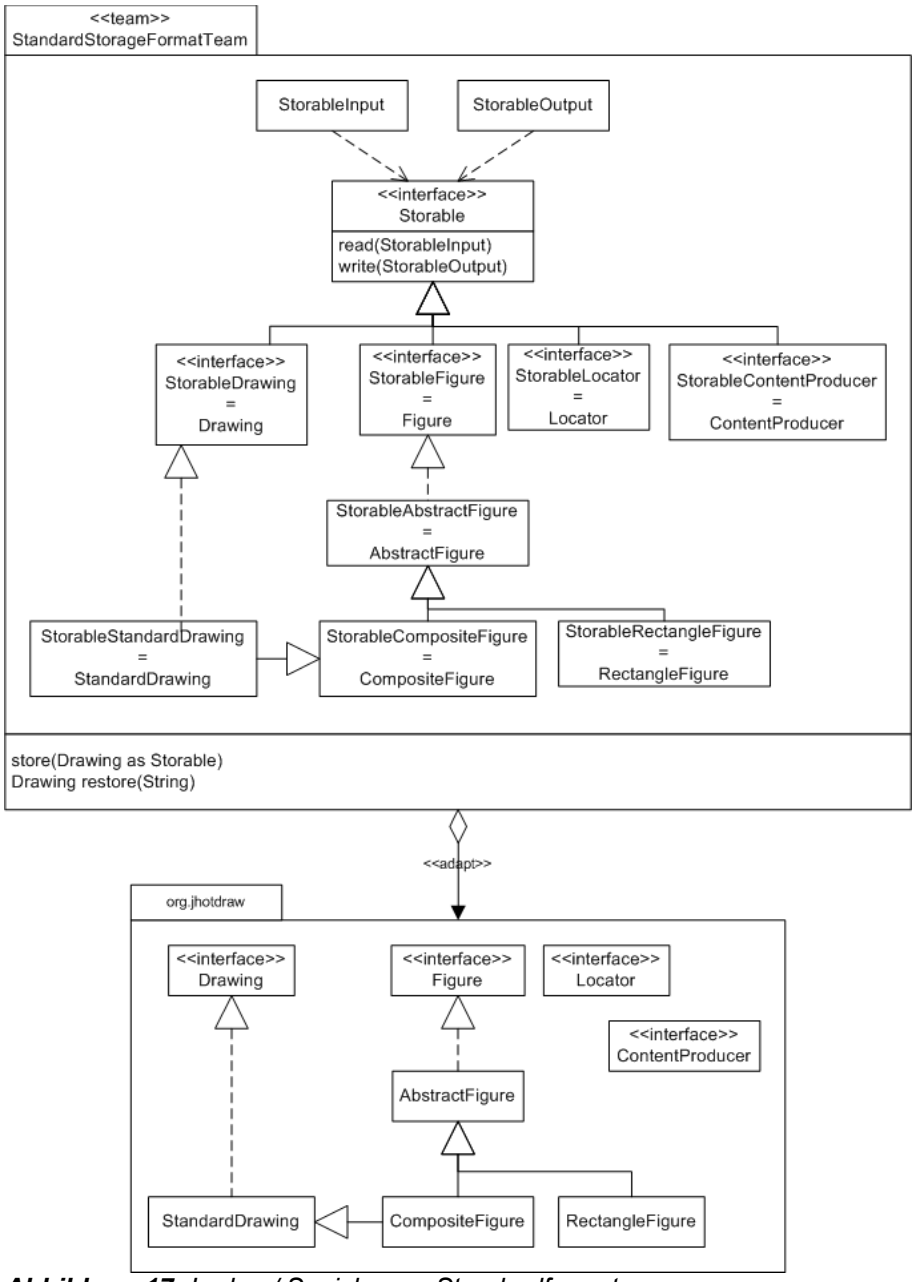


***Abbildung 17:*** *Laden / Speichern – Standardformat*
*ObjectTeams-orientierte Struktur*

Externalized Roles

In der bisherigen Darstellung wird vereinfachend davon ausgegangen, dass der Zweck der Klasse `StorableOutput` darin besteht, Daten in eine Datei zu schreiben. Das ist nicht ganz richtig. Ein `StorableOutput`-Objekt kann Daten in einen beliebigen `OutputStream` schreiben, den es bei seiner Erzeugung übergeben bekommt. Im Zusammenhang mit der Programmfunktion "Speichern einer Zeichnung" werden die Daten in eine Datei geschrieben, weil das `StorableOutput`-Objekt von der Methode `StandardStorageFormat.store` einen `FileOutputStream` übergeben bekommt. `StandardStorageFormat` ist eine von zwei Klassen des Frameworks, die `StorableOutput` verwendet. Die andere, `StandardFigureSelection`, benutzt `StorableOutput`, um Daten in einem ByteArray(`byte[]`)-Attribut zwischenzuspeichern und zu kopieren. Dafür wird dem `StorableOutput`-Objekt ein `ByteArrayOutputStream` übergeben.

In der objektorientierten Framework-Implementierung übergibt `StandardFigureSelection` der Methode `StorableOutput.writeStorable` ein Objekt des Typs `Figure`. Dieses ist als Parameter für die Methode `writeStorable` geeignet, weil Figure Subtyp des Typs `Storable` ist. In der ObjectTeams-orientierten Implementierung ist diese Form der Parameterübergabe aus den im vorherigen Abschnitt („Die Klasse StandardStorageFormat“ auf Seite 40) geschilderten Gründen nicht möglich. Das `Figure`-Objekt muss zuerst zu einem `Storable`-Objekt geliftet werden. Darüber hinaus sind sowohl `StorableOutput` als auch `Storable` Rollentypen. Und da die Klasse `StandardFigureSelection` außerhalb des Teams liegt, kann sie die Rollenobjekte nur im Kontext der sie umgebenden Teaminstanz erzeugen und verwenden.

Ein möglicher Ansatz zur Lösung dieses Problems ist die Verwendung von „externalized roles“ (siehe Kapitel 2). Zu diesem Zweck muss das Team folgende Eigenschaften besitzen:

- die Typen `StorableOutput` und `Storable` müssen nach außen sichtbar (`public`) sein.
- es muss eine Teamlevel-Methode geben, die einen `OutputStream` übergeben bekommt, ein `StorableOutput`-Objekt erzeugt und nach außen reicht (`getStorableOutput(OutputStream stream)`)
- es muss eine Teamlevel-Methode geben, die `Figure`-Objekte zu `Storable`-Objekten liftet (`liftFigureToStorable(Figure as Storable fig)`).

Das folgende Quelltext-Fragment der Klasse `StandardFigureSelection` zeigt die Anwendung.

```
final StandardStorageFormatTeam storageFormat = new StandardStorageFormatTeam();

ByteArrayOutputStream        output = new ByteArrayOutputStream(200);
storageFormat.StorableOutput writer = storageFormat.getStorableOutput(output);

writer.writeInt(figureCount);
while (figureEnum.hasNextFigure())
{
  writer.writeStorable(storageFormat.liftFigureToStorable(figureEnum.nextFigure()));
}
```

***Abbildung 18:*** *Storage - Externalized Roles : Quelltextfragment der Klasse StandardFigureSelection*

Dieses Verfahren ist relativ aufwendig. Die externe Klasse `StandardFigureSelection` steckt, umgangssprachlich ausgedrückt, ein `Figure`-Objekt ins Team hinein, bekommt dessen Rollenobjekt zurück (lift-Methode), um dieses dann wieder ins Team hineinzustecken (`StorableOutput.writeStorable`). Damit wird die Funktions-Kapselung, die durch das Team erreicht werden soll, aufgebrochen. Dies könnte vermieden werden, indem die Klasse `StandardStorageFormatTeam` die von `StandardFigureSelection` benötigte Funktionalität in Form von Teamlevel-Methoden zur Verfügung stellt. Ob sich das Design durch diesen Ansatz verbessern lässt oder ob sich aus ihm andere Nachteile ergeben, bleibt an dieser Stelle eine für weitere Untersuchungen offene Frage.

### *Aufgetretene Probleme und eingesetzte Workarounds*

Rollen und statische Methoden

Bei der Extraktion der Interface-Hierarchie des Typs `Storable` in ein Team wurden nicht nur, wie im Abschnitt "Der Rollentyp Storable" auf Seite 39 beschrieben, die Methoden `read` und `write` aus ihren Ursprungsklassen in deren neue Rollenklassen verschoben. Zusätzlich wurden die ausschließlich von den Methoden `read` und `write` verwendeten Hilfsmethoden aus den Basisklassen in die Rollenklassen verschoben, weil auch diese zur Programmfunktion „Speichern und Laden im Standardspeicherformat“ gehören, die im Team gekapselt werden soll. In diesem Zusammenhang trat das folgende Problem auf:
In der objektorientierten Implementierung besitzt eine der Klassen, die das Interface `Storable` implementieren, eine statische Methode. Diese wird ausschließlich von Klassen verwendet, die das Interface `Storable` implementieren, und kann eindeutig der zu extrahierenden Programmfunktion zugeordnet werden. Das bedeutet, dass diese Methode nach der ObjectTeams-orientierten Restrukturierung ausschließlich innerhalb des Teams von Objekten des Rollentyps `Storable` benötigt wird. Daher wäre es sinnvoll, auch diese Methode in die Rollenklasse ihrer ursprünglichen Klasse zu verschieben. Das ist aber nicht möglich, solange die Methode statisch ist. Eine Rollenklasse ist eine innere Klasse ihres Teams. Innere Klassen können in Java keine statischen Elemente enthalten. Daher dürfen Rollenklassen in der derzeitigen ObjectTeams/Java-Implementierung auch keine statischen Elemente besitzen.
Die Lösungsansätze für dieses Problem richten sich nach verschiedenen Kriterien. Benötigt die Methode lediglich Konstanten und Werte, die als Parameter übergeben werden, muss sie nicht statisch sein. In einer rein objektorientierten Implementierung könnte sie durch eine "normale" Methode ersetzt werden. Der Klient müsste dann zuerst ein Objekt instanziieren, um auf diesem die Methode aufzurufen. In der ObjectTeams-orientierten Implementierung unterliegt dieses Vorgehen einer Einschränkung. Gebundene Rollen, die ihre Basisobjekte nicht selbst erzeugen, wie im beschriebenen konkreten Anwendungsfall, können nicht beliebig instanziiert werden. Der Klient müsste Zugriff auf ein Objekt des Basistyps haben und dieses liften, um die Methode verwenden zu können.
Eine Alternative besteht darin, eine neue ungebundene Rollenklasse zu erstellen, und die (nicht statische) Methode dorthin zu verschieben. Eine andere Alternative ist, die Methode zu einer Teamlevel-Methode zu machen. In diesem Fall könnte sie statisch bleiben.

Spracherweiterungs-vorschlag

Auch wenn es alternative Strukturierungsmöglichkeiten gibt, die die Funktionsfähigkeit der Software gewährleisten können, so gehört die Methode doch inhaltlich logisch immernoch zu einer bestimmten Rollenklasse, strukturell aber nicht. Daher ist die Tatsache, dass eine Rollenklasse, anders als eine gewöhnliche Klasse, keine statischen Methoden besitzen kann, eine negativ zu bewertende Einschränkung der Möglichkeiten eines guten Designs. Daher soll an dieser Stelle eine diesbezügliche Erweiterung der Konzepte von ObjectTeams angeregt werden.

Lifting, Lowering und Container

Handelte es sich bei einem zu speichernden Attribut der Basisklasse um ein Objekt eines Container-Typs (z.B. eine Liste), dessen Elemente in Java auf den allgemeinsten Typ Object getypt sind, wurde folgendermaßen vorgegangen:

Die Elemente des Containers müssen einzeln durchiteriert, zu Storable geliftet

und mittels der Methode StorableOutput.writeStorable gespeichert werden. Um die Möglichkeiten des impliziten Liftings entlang einer Methodenbindung nutzen und so explizite Lift-Operationen durch die Rollenklasse vermeiden zu können, wurde in die Basisklasse eine get-Methode eingefügt, die den Inhalt des Container-Attributs in Form eines Arrays liefert, welches auf den wirklichen Typ der Container-Elemente getypt ist. In der an die Basisklasse gebundenen Rollenklasse wurde dann eine Rollenmethode, die als Parameter ein Array mit Objekten des Rollentyps Storable erwartet, (über eine callout-Bindung) an diese get-Methode der Basisklasse gebunden. Durch dieses Vorgehen werden bei einem Aufruf entlang der deklarierten Methodenbindung alle Elemente des als Parameter übergebenen Arrays automatisch geliftet.

Die Vorgehensweise für das Laden von Attributen von Container-Typen funktioniert analog. Die Basisklasse wird in diesem Fall um eine entsprechende set-Methode erweitert, Die übergebenen Parameter werden automatisch gelowert.

## 4.5 Verwendung der Framework-Implementierung bei der Erstellung einer Applikation

Nachdem in Kapitel 4.2 die originale objektorientierte Implementierung und in Kapitel 4.4 verschiedene Formen der restrukturierten ObjectTeams-orientierten Implementierung der Programmfunktion „Laden und Speichern einer Zeichnung" vorgestellt wurden, zeigt das vorliegende Kapitel, wie diese im Rahmen der Entwicklung einer auf dem Framework JHotDraw aufbauenden Applikation verwendet werden. Dabei orientiert sich die Darstellung an den folgenden drei Fragestellungen, die bezüglich dieser Verwendung zu beantworten sind:

- Wie kann festgelegt werden, ob die Lade-Speicher-Funktionalität im Funktionsumfang der Applikation enthalten sein soll?
- Wie kann festgelegt werden, welche der Dateiformate, die mit Hilfe des Frameworks unterstützt werden können, die Applikation unterstützen soll?
- Was muss getan werden, damit Objekte selbst definierter Klassen (z.B. spezielle Figuren) im Standardspeicherformat gespeichert werden können?

Um einen besseren Vergleich zu ermöglichen, wird jede Fragestellung jeweils einzeln einerseits für die Verwendung der objektorientierten Originalversion und andererseits für die Verwendung der drei in Kapitel 4.4 vorgestellten ObjectTeams-orientierten Versionen des Frameworks beantwortet.

### 4.5.1 Festlegung des Funktionsumfangs

Die Lade-Speicher-Programmfunktion ist in JHotDraw-basierten Applikationen standardmäßig über die Unterpunkte "SaveAs..." und "Open..." des Dateimenüs erreichbar. Soll im Funktionsumfang einer Applikation das Laden und Speichern nicht inbegriffen sein, darf diese solche Menüoptionen nicht anbieten. Die Möglichkeit, aufbauend auf JHotDraw solche Applikationen zu erstellen, die Laden und Speichern anbieten, und auch solche, die dies nicht tun, ist sowohl bei der Verwendung der originalen objektorientierten Framework-Implementierung als auch bei der Verwendung einer der ObjectTeams-orientierten Implementierungen gegeben.

***Objektorientierte Implementierung***

Das Vorhandensein der Lade-Speicher-Funktionalität ist bei der Verwendung der objektorientierten Framework-Implementierung der Standardfall. Das heißt, damit eine auf JHotDraw aufbauende Applikation diese Funktionalität enthält, sind keine zusätzlichen Arbeitsschritte erforderlich.
Soll die Applikation die Funktionalität nicht enthalten, muss ihre `DrawApplication`-Subklasse (welche sie in jedem Fall besitzt), die Methode `createFileMenu` überschreiben. Dabei darf die überschreibende Methode die Supermethode nicht aufrufen, sondern muss bis auf die Erzeugung der Unterpunkte für Laden und Speichern die selben Operationen ausführen wie diese (deren Inhalt kopieren). Dies ist möglich, weil sowohl die Supermethode `DrawApplication.createFileMenu` als auch alle von dieser aufgerufenen Methoden frei zugänglich (`public` oder `protected` sind).

***ObjectTeams-orientierte Implementierung***

In einer ObjectTeams-orientierten Framework-Implementierung ist die Implementierung der Lade-Speicher-Programmfunktion in der Klasse StorageTeam gekapselt. Damit einer auf JHotDraw basierenden Implementierung diese Programmfunktion zur Verfügung steht, muss ein Objekt dieser Team-Klasse erzeugt und vor allem aktiviert werden, bevor das Dateimenü erzeugt wird. Ob eine auf JHotDraw basierende Applikation diese Aktivierung selbst erledigen muss, wenn sie die Lade-Speicher-Funktionalität verwenden will, oder ob sie die Aktivierung unterdrücken muss, wenn die Funktion nicht vorhanden sein soll, ist davon abhängig, auf welches der in Kapitel 5 „Aspekte der Team-Erzeugung und -Aktivierung„ ab Seite 61 erläuterten Team-Handhabungsverfahren die Framework-Implementierung strukturell ausgerichtet ist.

### 4.5.2 Festlegung der zu unterstützenden Speicherformate

Die Dateiformate, die mit Hilfe des Frameworks unterstützt werden können, werden repräsentiert durch die Klassen, die das Interface `StorageFormat` implementieren, z.B. die Klasse `StandardStorageFormat`. Die Festlegung, welche Speicherformate eine auf JHotDraw basierende Applikation standard-mäßig unterstützt, erfolgt in der Methode `createStorageFormatManager`. Dort werden die `StorageFormat`-Implementierungsklassen instanziiert und die erzeugten Objekte an den `StorageFormatManager` der Applikation übergeben.
Soll bei einer auf JHotDraw basierenden Applikation die Liste der unterstützten Speicherformate von dieser Standardfestlegung abweichen, muss die Methode `createStorageFormatManager` vollständig überschrieben werden. Welche Klasse beerbt werden muss, um die Methode `createStorageFormat-Manager` überschreiben zu können und welche zusätzlichen Aufgaben erfüllt werden müssen, ist davon abhängig, welche Framework-Implementierung als Grundlage der Applikation verwendet wird.

***Objektorientierte Implementierung***

In der objektorientierten Originalversion des Frameworks befindet sich die Methode `createStorageFormatManager` in der Klasse `DrawApplication`. Jede auf JHotDraw basierende Applikation besitzt als zentrale Klasse eine Subklasse von `DrawApplication`. Soll die Liste der unterstützten Speicherformate von der Standardimplementierung abweichen, muss die

Methode `createStorageFormatManger` von dieser `DrawApplication`-Subklasse überschrieben werden.

***ObjectTeams-orientierte Implementierung***

**Erster Design-Ansatz**

Bei der im Abschnitt „Erster Design-Ansatz“ auf Seite 33 beschriebenen und in Abbildung 11 dargestellten ObjectTeams-orientierten Framework-Implementierung befindet sich die Methode `createStorageFormatManager` in der Rollenklasse `StorageTeam.StorageManager`. Soll die Liste der unterstützten Speicherformate von der Standardimplementierung abweichen, muss eine Teamklasse von `StorageTeam` abgeleitet werden und die Methode `createStorageFormatManager` von der sich in diesem Team befindenden implizit geerbten Rollenklasse `StorageManager` überschrieben werden. Zusätzlich muss die Aktivierung des originalen `StorageTeams` verhindert und durch die Instanziierung und Aktivierung der spezielleren Teamklasse ersetzt werden. Wie das gemacht wird, wird im Abschnitt „Spezialisierbarkeit der verwendeten Team-Klassen“ ab Seite 70 erläutert.

**Zweiter Design-Ansatz**

Bei Verwendung der im Abschnitt „Zweiter Design-Ansatz“ auf Seite 34 beschriebenen und in Abbildung 12 dargestellten ObjectTeams-orientierten Framework-Implementierung, erfolgt die Anpassung der Liste der zu unterstützenden Speicherformate mit einer kleinen Ausnahme auf die selbe Art und Weise, die im vorigen Abschnitt beschrieben wurde. Die Ausnahme besteht darin, dass in der Methode `createStorageFormatManager` die erzeugten `StorageFormat`-Objekte explizit geliftet werden müssen, bevor sie dem `StorageFormatManager` übergeben werden können.

**Dritter Design-Ansatz**

Bei der in Abschnitt „Dritter Design-Ansatz“ auf Seite 36 beschriebenen ObjectTeams-orientierten Framework-Implementierung existiert keine Methode `createStorageFormatManager`. Die standardmäßig unterstützten Speicherformate werden in der Teamlevel-Methode `setAvailableStorage-Formats` der Klasse `StorageTeam` festgelegt. Soll bei Verwendung dieser Framework-Version die Liste der unterstützten Speicherformate von der Standardimplementierung abweichen, muss eine Teamklasse von `StorageTeam` abgeleitet und in dieser die Methode `setAvailableStorage-Formats` überschrieben werden. Auch in diesem Fall ist die Ersetzung des zu aktivierenden Teams durch ein Objekt der spezielleren Teamklasse erforderlich.

### 4.5.3 Speicherbarmachen selbst definierter Klassen im Standardspeicherformat

Nachdem erläutert wurde, wie eine auf JHotDraw basierende Applikation festlegen kann, welche Speicherformate sie unterstützt, soll nun beschrieben werden, welche Anforderungen erfüllt werden müssen, damit die von einer Applikation selbst definierten Klassen (z.B. spezielle Figurtypen) im Standardformat gespeichert werden können. Diese Anforderungen unterscheiden sich voneinander in Abhängigkeit davon, welche Framework-Implementierung als Grundlage der Applikation verwendet wird.

Für alle Framework-Versionen gleich ist, dass eine von der Applikation definierte Klasse, deren Objekte im Standardspeicherformat speicherbar sein sollen, einen parameterlosen Konstruktor besitzen muss.

### *Objektorientierte Implementierung*

Basiert die Applikation auf der originalen objektorientierten Framework-Version, muss eine Klasse, deren Objekte im Standardspeicherformat speicherbar sein sollen, außerdem direkt oder indirekt das Interface `Storable` implementieren ( siehe Abbildung 19).

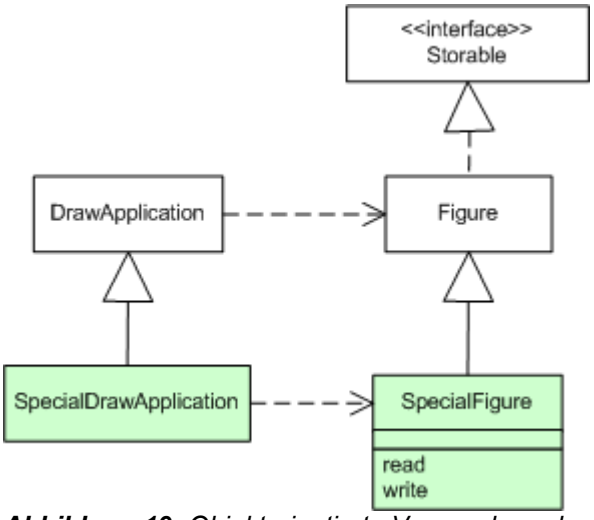


***Abbildung 19:*** *Objektorientierte Verwendung der Lade-Speicher-Funktionalität*

Zu diesem Zweck muss sie die Methoden `write(StorableOutput)` und `read(StorableInput)` definieren. In der Methode `write` muss sie die Werte der Attribute, die gespeichert werden sollen, mit Hilfe der Methoden des übergebenen `StorableOutput`-Objekts schreiben. Und in der Methode `read` muss sie mit Hilfe der Methoden des übergebenen `StorableInput`-Objekts Werte einlesen und ihre Attribute setzen. Dabei muss die Reihenfolge eingehalten werden, in der in der Methode `write` geschrieben wurde. Implementiert die Klasse das Interface `Storable` indirekt, indem sie von einer Klasse erbt, die Teil der Hierarchie des Typs `Storable` ist, muss sie sowohl in der Methode `write` als auch in der Methode `read` die jeweilige Supermethode aufrufen.

### *ObjectTeams-orientierte Implementierung*

Basiert die Applikation auf einer der ObjectTeams-orientierten Framework-Versionen, muss eine Klasse, deren Objekte im Standardspeicherformat speicherbar sein sollen, zusätzlich zu einem parameterlosen Konstruktor eine an sie gebundene Rollenklasse besitzen, die das in der Klasse `StandarStorageFormatTeam` enthaltene Rollen-Interface `Storable` implementiert. Als Kontext für diese Rollenklasse muss eine von der Klasse `StandardStorageFormatTeam` abgeleitete Team-Klasse erstellt werden. Um diese in die Applikation zu integrieren, muss weiterhin eine von der Klasse `StorageTeam` abgeleitete Team-Klasse erstellt werden.

Die Rollenklasse muss die Methoden `write(StorableOutput)` und `read (StorableInput)` definieren, wobei sich deren Implementierung von der im vorhergehenden Abschnitt beschriebenen, objektorientierten nur dadurch unterscheidet, dass die Rolle nicht ihre eigenen Attribute liest und schreibt, sondern die der an sie gebundenen Basisklasse. Zu diesem Zweck benötigt die Rollenklasse entsprechende callout-gebundene `get`- und `set`-Methoden.

Weiterhin ist zu beachten, dass die Rollenklasse an der „richtigen" Stelle in die Hierarchie des Rollen-Interface `Storable` eingehängt werden muss. Die „richtige" Stelle ist dabei folgendermaßen zu ermitteln (siehe Abbildung 20): Es ist die Vererbungshierarchie in der sich die an die Rolle (`StorableSpecialFigure`) gebundene Basisklasse (`SpecialFigure`) befindet, von der Basisklasse ausgehend in Richtung nach "oben" zu betrachten. Kann dabei eine Superklasse (`Figure`) identifiziert werden, die eine an sie gebundene Rollenklasse (`StorableFigure`) besitzt, die das Rollen-Interface `Storable` implementiert, so muss die neue Rollenklasse (`StorableSpecialFigure`) die Rollenklasse dieser Superklasse seiner Basisklasse direkt beerben. Abbildung 20 versucht, dies zu veranschaulichen, wobei die für den „richtigen“ Platz in der Rollenhierarchie Ausschlag gebende Beziehung orange und die zu erstellenden Elemente grün dargestellt sind. Besitzt die zu speichernde Klasse keine Superklasse, die eine derartige Rolle besitzt, muss ihre Rollenklasse das Rollen-Interface `Storable` direkt implementieren.

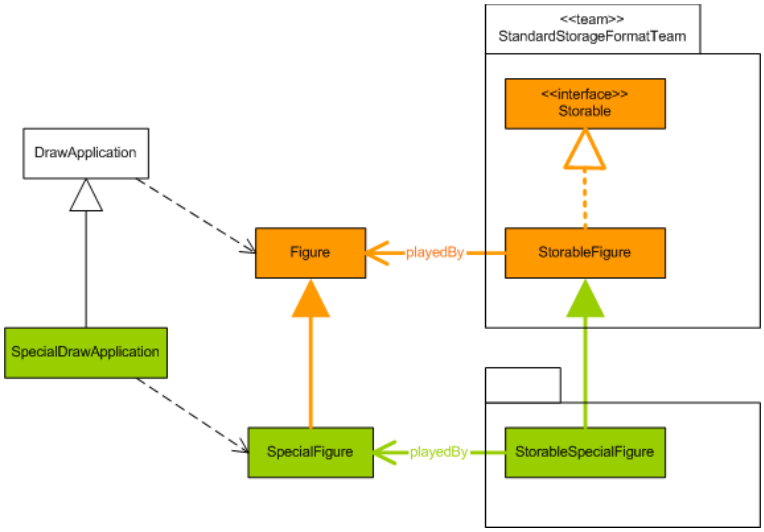


***Abbildung 20:*** *ObjectTeams-orientierte Verwendung der Lade-Speicher-Funktionalität, Hierarchie-Beziehungen*

Die eben erläuterte allgemeine Notwendigkeit einer an die zu speichernde Klasse gebundenen `Storable`-Rollenklasse, einer als Kontext für diese Rolle dienenden `StandardStorageFormatTeam`-Subklasse und einer zu deren Integration notwendigen `StorageTeam`-Subklasse manifestiert sich in verschiedenen strukturellen Ausprägungen, je nachdem, welche der drei verschiedenen ObjectTeams-orientierten Framework-Versionen verwendet wird. Diese werden in den folgenden Abschnitten beschrieben.

Für alle gleich ist jedoch, das die Aktivierung des vom Framework verwendeten `StorageTeams` verhindert und durch die Instanziierung und Aktivierung des

spezielleren `StorageTeam`-Typs ersetzt werden muss. Wie das gemacht wird, wird im Abschnitt „Spezialisierbarkeit der verwendeten Team-Klassen“ ab Seite 70 erläutert.

### Erster Design-Ansatz

Wird die Applikation auf der Grundlage der in Abbildung 11 auf Seite 34 dargestellten Framework-Implementierung erstellt, enthält die von `StorageTeam` abgeleitete Team-Klasse eine Team-Klasse `StandardStorageFormatTeam`, die implizit von der Klasse `StorageTeam.StandardStorageFormatTeam` erbt (siehe Abbildung 21). Diese muss um eine Rollenklasse erweitert werden, welche das wiederum implizit geerbte Rollen-Interface `Storable` implementiert. Weitere Arbeitsschritte sind nicht erforderlich. Abbildung 21 zeigt die resultierende Struktur, wobei die zu erstellenden Elemente farbig unterlegt sind.

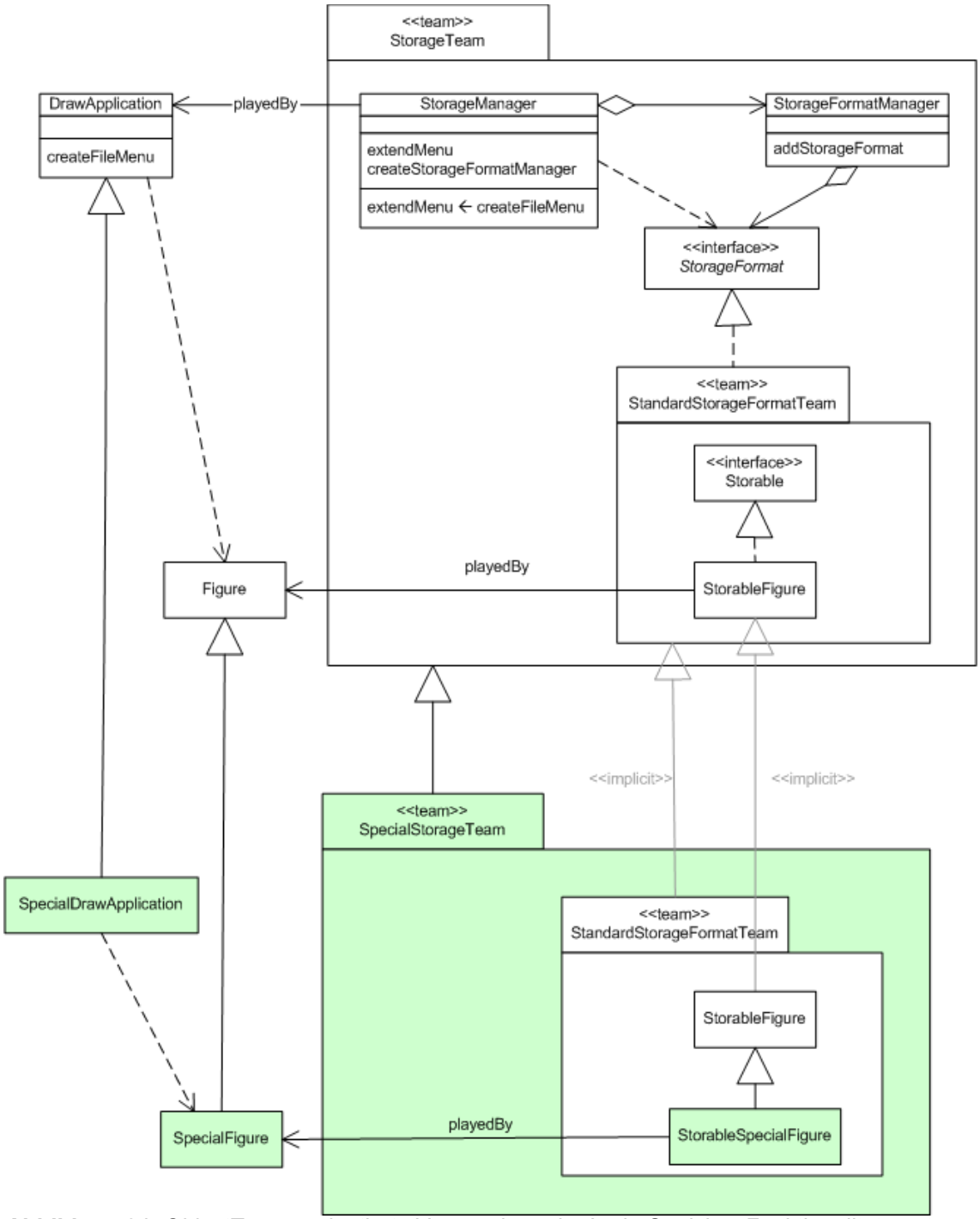


***Abbildung 21:*** *ObjectTeams-orientierte Verwendung der Lade-Speicher-Funktionalität, erster Design-Ansatz*

### Zweiter Design-Ansatz

Wird die Applikation auf der Grundlage der im Abschnitt „Zweiter Design-Ansatz“ auf Seite 34 beschriebenen Framework-Implementierung entwickelt, muss die Rollenklasse innerhalb einer selbst zu erstellenden, von der Klasse `StandardStorageFormatTeam` abgeleiteten Team-Klasse definiert werden (siehe Abbildung 22). Da in diesem Fall, im Gegensatz zu der im vorhergehenden Abschnitt vorgestellten Variante, die Vorteile der impliziten Vererbung nicht genutzt werden können, muss sich diese Team-Klasse erstens

namentlich von ihrer Superklasse unterscheiden. Zweitens muss dass von der Framework-Implementierung standardmäßig verwendete Objekt der Klasse `StandardStorageFormatTeam` explizit durch eines der neuen, spezielleren Team-Klasse ersetzt werden. Dies geschieht, genauso wie im Kapitel 4.5.2 „Festlegung der zu unterstützenden Speicherformate“ beschrieben, durch das Überschreiben der Methode `createStorageFormatManager` der im `StorageTeam` enthaltenen Rollenklasse `StorageManager`. Dabei ist einerseits zu beachten, dass der `StorageFormatManager` nicht mehr als ein Objekt des Typs `StandardStorageFormatTeam` halten darf, und es daher nicht ausreicht, die Supermethode aufzurufen und danach ein Objekt der neuen `StandardStorageFormatTeam`-Subklasse hinzuzufügen. Andererseits ist zu beachten, dass die Rollenklasse, die die Methode `createStorageFormatManager` überschreibt, an die `DrawApplication`-Subklasse der Applikation gebunden werden muss. Abbildung 22 zeigt die resultierende Struktur, wobei die neu zu erstellenden Elemente wieder farbig unterlegt sind.

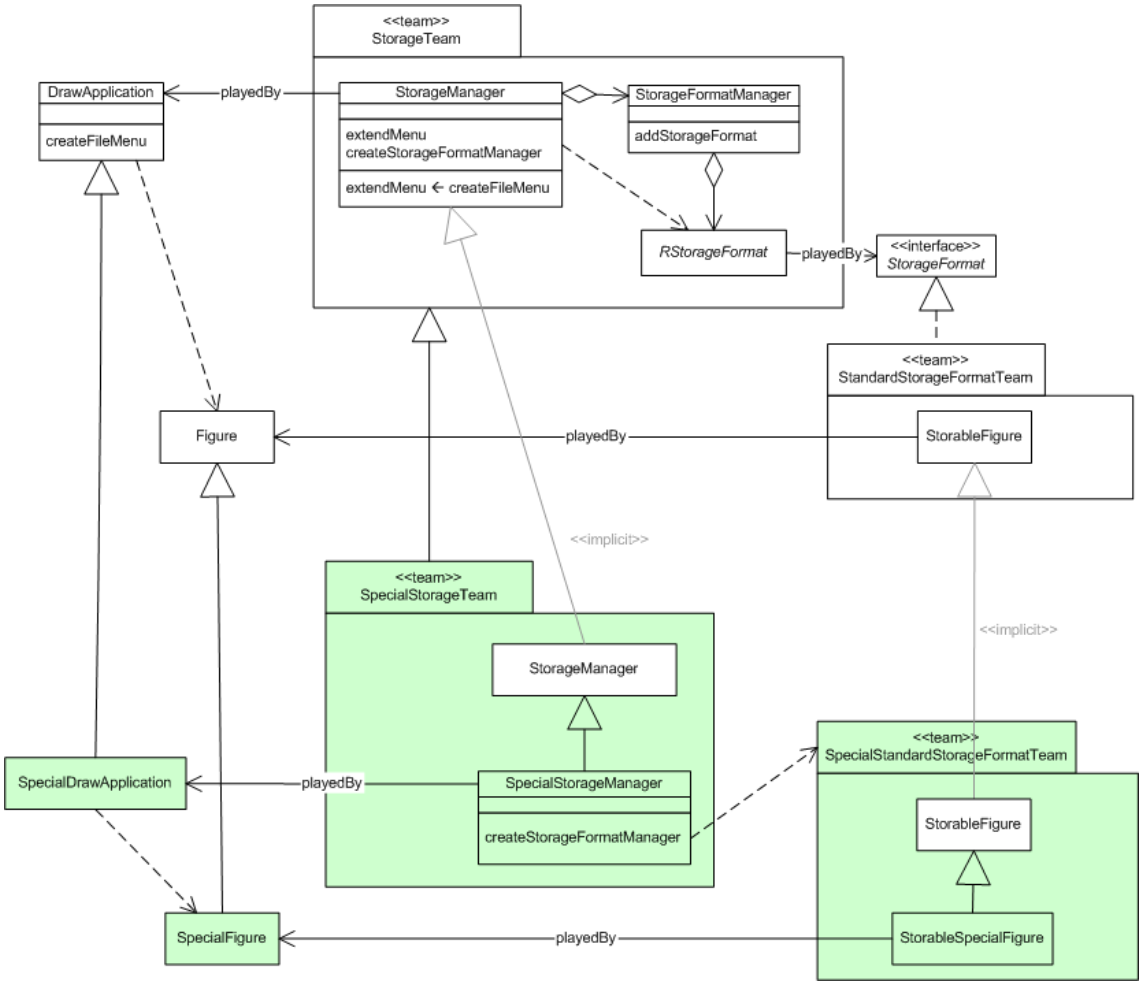


***Abbildung 22:*** *ObjectTeams-orientierte Verwendung der Lade-Speicher-Funktionalität, zweiter Design-Ansatz*

### Dritter Design-Ansatz

Das Verfahren bei der Verwendung der im Abschnitt „Dritter Design-Ansatz“ auf Seite 36 beschriebenen Framework-Version unterscheidet sich von dem im

vorangegangenen Abschnitt beschriebenen dadurch, dass nicht die Rollenmethode `StorageTeam.StorageManager.createStorageFormat-Manager` sondern die Teamlevel-Methode `StorageTeam.setAvailable-StorageFormats` überschrieben werden muss, um das Objekt der Klasse `StandardStorageFormatTeam` durch ein Objekt der spezielleren Klasse zu ersetzen. Dadurch muss weder eine neue Rollenklasse von `StorageTeam.StorageManager` abgeleitet, noch die `DrawApplication`-Subklasse der Applikation gebunden werden.

## 4.6 Fazit

Nachdem in den vorangegangenen Kapiteln ausführlich die originale objektorientierte Implementierung der Programmfunktion "Laden und Speichern einer Zeichnung" des Frameworks JHotDraw, verschiedene Möglichkeiten ihrer ObjectTeams-orientierten Restrukturierung und die Verwendung der verschiedenen Implementierungen bei der Erstellung einer auf JHotDraw aufbauenden Applikation erläutert wurden, sollen im Rahmen dieses Kapitels Schlussfolgerungen aus diesem Anwendungsbeispiel gezogen werden. Dabei orientiert sich die Darstellung an den Fragestellungen, die im Rahmen der vorliegenden Arbeit in Bezug auf ObjectTeams untersucht und beantwortet werden sollen. Es geht darum zu überprüfen, inwieweit die von ObjectTeams verfolgten Ziele, im Rahmen dieses einen Beispielfalls erreicht werden. Die im Rahmen dieses Kapitels zu beantwortenden Fragen sind die folgenden:

- (In welchem Umfang) sind die Konzepte von ObjectTeams bei der Implementierung einer komplexen Software anwendbar? Insbesondere, läßt sich eine Programmfunktion aus einer bestehenden objektorientierten Implementierung extrahieren und in Form eines Teams kapseln?
- Überprüfung der Funktionsfähigkeit von Compiler und Laufzeitumgebung - Inwiefern wurden die vorgestellten Lösungen implementiert/ waren kompilier- und ausführbar? Welche sprachlichen Schwächen konnten ausfindig gemacht werden?
- Welche Aussagen können über den Restrukturierungsprozess als solchen getroffen werden?
- Welche Verbesserungsvorschläge gibt es in diesem Rahmen bezüglich der Konzepte und des Sprachumfangs?
- Ist ObjectTeams bei der Implementierung komplexer Systeme hilfreich? Insbesondere
  - Kann die Modularität und Lokalität der Programmfunktion erhöht werden?
  - Wird die Verständlichkeit des Frameworks für den Benutzer erhöht?
  - Kann die Programmfunktion gezielt adaptiert werden, ohne dass große Teile des Systems verstanden werden müssen?
  - Wird die Instanziierung einer Anwendung auf der Grundlage des Frameworks dadurch einfacher?
  - Zusammenfassend gefragt: Ist ObjectTeams eine Bereicherung für Frameworks? Ermöglicht es ein besseres Design? Macht es die Komplexität des Frameworks besser handhabbar?

All diese Fragen sollen, soweit dies im Rahmen des Beispiels der Lade-Speicher-Programmfunktion möglich ist, in diesem Kapitel beantwortet werden.
Im Anschluss an diesen Teil werden noch kurz die Besonderheiten der ObjectTeams-orientierten Lade-Speicher-Implementierung dargestellt, bevor dann abschließend ein Ausblick gegeben wird, welche Fragen im Rahmen der vorliegenden Arbeit unbeantwortet geblieben sind und welche weiterführenden

Aufgaben interessante und wertvolle Ergebnisse versprechen.

### 4.6.1 Anwendbarkeit der ObjectTeams-Konzepte

Sind die Konzepte von ObjectTeams bei der Implementierung einer komplexen Software anwendbar? Insbesondere, läßt sich eine Programmfunktion aus einer bestehenden objektorientierten Implementierung extrahieren und in Form eines Teams kapseln?

Für den hier behandelten Beispielfall der Lade-Speicher-Programmfunktion des Frameworks JHotDraw sind diese Fragen eindeutig positiv zu beantworten. Kapitel 4.4 zeigt, dass es möglich ist, die Programmfunktion mit Hilfe von ObjectTeams-Konzepten zu kapseln.

### 4.6.2 Funktionsfähigkeit von Compiler und Laufzeitumgebung und sprachliche Schwächen

In welchem Umfang wurden die in Kapitel 4.4 vorgestellten Lösungen implementiert und stehen in kompilier- und ausführbarem Zustand zur Verfügung? Welche sprachlichen Schwächen konnten ausfindig gemacht werden?

Der in Kapitel 4.4.1 auf Seite 33 dargestellte „erste“ Design-Ansatz wurde nicht implementiert, da er im Zusammenhang mit der in Kapitel 4.4.2 auf Seite 39 dargestellten Implementierung des Standardspeicherformates auf eine Schachtelung von Team-Klassen (Team im Team) hinausgelaufen wäre und eine solche im Funktionsumfang der derzeitigen ObjectTeams/Java-Version nicht enthalten ist.

Die in Kapitel 4.4.2 auf Seite 39 beschriebene Extraktion der Standard-speicherformat-Implementierung wurde im Zusammenhang mit dem in Kapitel 4.4.1 auf Seite 34 dargestellten „zweiten“ Design-Ansatz vollständig durchgeführt. Die resultierende Implementierung liegt in kompilierbarem und in begrenztem Umfang auch ausführbaren Zustand vor, wobei sie aufgrund bestehender Schwächen des ObjectTeams/Java-Compilers und der Laufzeitumgebung zur Sicherstellung ihrer Kompilier- und Ausführbarkeit zahlreiche "workarounds" und über die Extraktion der Programmfunktion hinausgehende invasive Eingriffe in die Basisklassen enthält. Diese werden im folgenden aufgelistet, wobei kein Anspruch auf Vollständigkeit erhoben wird.

Es musste eine Erweiterung der Sichtbarkeit callout-gebundener Basismethoden vorgenommen werden (`protected -> public`).

Da es nicht möglich ist, auf Attribute der Basisklasse direkt per callout zuzugreifen, mussten die Basisklassen, wo dies erforderlich war, um `get`- und `set`-Methoden erweitert werden.

Es traten Lifting- und Casting-Probleme im Zusammenhang damit auf, dass das Wurzel-Interface der aus der Basissoftware in das `StandardStorage-FormatTeam` extrahierten `Storable`-Interface-Hierarchie ungebunden ist. Daher war es notwendig, die durch die Verschiebung des Interface `Storable` entstandene Lücke in der Hierarchie der Basisklassen mit einem leeren "Stellvertreter"-Interface zu füllen und das Rollen-Interface `Storable` an dieses zu binden.

Während der Implementierung der in Kapitel 4.4 vorgestellten Ansätze gab es zahlreiche Schwierigkeiten mit Compiler und Laufzeitumgebung, insbesondere im Zusammenhang mit der in Kapitel 4.4.2 beschriebenen Extraktion der `Storable`-Typhierarchie. Diese wurden durch sukzessive Erweiterung und Reparatur des Funktionsumfangs von Compiler und Laufzeitumgebung während der Arbeiten an der vorliegenden Fallstudie soweit behoben, dass es möglich war, die beschriebenen Ansätze zumindest teilweise zur Lauffähigkeit zu bringen.

### 4.6.3 Restrukturierungsprozess

Der Restrukturierungsprozess als solcher war konzeptuell relativ einfach, praktisch aber sehr aufwendig und teilweise auch schwierig.

Konzeptuell einfach bedeutet in diesem Fall, dass nur in einem Fall eine callin-Anbindung erfolgte und größtenteils vollständige Methoden aus der Basissoftware in die Teamklassen verschoben werden konnten, ohne dass es erforderlich geworden wäre, in den Basisklassen selbst weitergehende Restrukturierungen vorzunehmen.

Die ObjectTeams-orientierte Restrukturierung war praktisch sehr aufwendig und Konzentration erfordernd, weil für die in Kapitel 4.4.2 beschriebene Extraktion und Verschiebung der Hierarchie des Interface `Storable` für mehr als 80 Klassen jeweils eine eigene an sie gebundene Rollenklasse erstellt, ihre Methoden `read` und `write` in diese Rollenklasse verschoben und für jede Methode und jedes Attribut, welche von `read` und `write` aufgerufen werden, jeweils eine callout-gebundene Rollenmethode erstellt werden musste, wobei darauf zu achten war, dass die Vererbungsstruktur der Rollenklassen eine Schattenhierarchie der Vererbungsstruktur der an sie gebundenen Basisklassen bildet. Insbesondere die Sicherstellung der korrekten Vererbungsstruktur der Rollenklassen als Spiegelbild der Vererbungsstruktur der Basisklassen erforderte aufgrund der sich wegen der großen Klassenanzahl einstellenden Unübersichtlichkeit und der fehlenden Werkzeugunterstützung erhebliche Konzentration und erwies sich als große Fehlerquelle.
Da die genannten Aufgaben automatisierbar sind und dies einerseits den zu ihrer Ausführung erforderlichen Zeitaufwand und andererseits die Fehlerquelle erheblich verringern würde, wäre die Entwicklung geeigneter Werkzeuge zur Extraktion einer Interface-Hierarchie und zur Generierung von callout-gebundenen Rollenmethoden sehr sinnvoll.

Neben den genannten Gründen für den hohen Aufwand wurde der Restrukturierungsprozesses zusätzlich durch Unzulänglichkeiten in der aktuellen Compiler- und Laufzeitumgebungsimplementierung erschwert. Bei mehr als 80 in separaten Dateien definierten Rollenklassen des selben Teams war es aufgrund mangelhafter oder sogar falscher Fehlermeldungen teilweise sehr schwierig, den Fehler zu lokalisieren. In vielen Fällen war unklar, ob es sich um einen Fehler in der ObjectTeams-orientierten JHotDraw-Implementierung oder um einen Fehler in der Implementierung des Compilers oder der Laufzeitumgebung handelte. Aus diesem Grund wäre die Realisierung der in Kapitel 4.4 vorgestellten Modellierungen ohne den ständigen Kontakt zum Entwickler-Team der ObjectTeams/Java-Sprachumgebung und die Reparatur und Weiterentwicklung von Compiler und Laufzeitumgebung nicht möglich gewesen.

### 4.6.4 Spracherweiterungsvorschläge

Es empfiehlt sich, neben dem Binden von Methoden per callout auch das Binden von Attributen zu ermöglichen. Durch diese Maßnahme kann vermieden werden, dass für einen Zugriff der Rollenklasse auf ein Attribut seiner Basisklasse, die Basisklasse um `get`-Methoden erweitert werden muss (siehe Abschnitt „Der Rollentyp Storable“ auf Seite 39).

Weiterhin wäre es wünschenwert, dass Rollenklassen genauso wie gewöhnliche Klassen statische Methoden besitzen dürfen (siehe Abschnitt „Rollen und statische Methoden“ auf Seite 43).

Ein weiterer Verbesserungsvorschlag, der sich weniger auf die sprachlichen Konzepte von ObjectTeams als vielmehr auf deren konkrete praktische Anwendung bezieht, betrifft die Deklaration von Importen. In der momentanen ObjectTeams/Java-Implementierung müssen alle Importe, die für die Kompilierung von Rollenklassen erforderlich sind, in der Datei deklariert werden, in der sich die zugehörige Team-Klasse befindet. Werden die Rollenklassen jedoch in separaten Dateien definiert, was schon bei einigen wenigen Rollenklassen pro Team-Klasse aus Gründen der Übersichtlichkeit als sinnvoll angesehen werden muss, erweist sich diese Vorschrift als unpraktisch. Es wäre besser, wenn der Entwickler die Importe dort deklarieren könnte, wo diese (zumindest aus seiner Sicht) auch benötigt werden. So kann er besser überschauen, ob alle erforderlichen Importe deklariert wurden.

### 4.6.5 Ist ObjectTeams bei der Implementierung komplexer Systeme hilfreich

#### *Lokalität*

Kann die Modularität und Lokalität der Implementierung einer Programmfunktion durch ObjectTeams erhöht werden? Können "crosscutting" und "scattering" reduziert oder sogar vermieden werden?

Für den hier behandelten Beispielfall der Lade-Speicher-Programmfunktion des Frameworks JHotDraw können diese Fragen eindeutig positiv beantwortet werden. Bis auf den Umstand, dass jede Klasse, deren Objekte im Standardspeicherformat speicherbar sein soll, also eine an sie gebundene `Storable`-Rolle besitzt, einen parameterlosen Konstruktor benötigt, können alle Implementierungselemente der Lade-Speicher-Programmfunktion in Form eines Teams gekapselt werden.

Der Preis, der dafür gezahlt wurde, ist allerdings eine starke Erhöhung der Anzahl der im Framework existierenden Klassen. Allein das `StandardStorageFormatTeam` erhöht die Klassenanzahl des Frameworks um mehr als 80 Rollenklassen.

#### *Verständlichkeit und Instanziierbarkeit*

Wird die Framework-Implementierung durch den Einsatz von ObjectTeams für den Benutzer verständlicher? Ist die Instanziierung einer Applikation auf der Grundlage der ObjectTeams-orientierten Framework-Implementierung (dadurch) einfacher?

Diese Fragen können im Rahmen der vorliegenden Arbeit nicht erschöpfend

beantwortet werden. Aufgrund der Tatsache, dass ich in meiner Rolle als Framework-Entwicklerin sowohl mit den Details des Designs und der Implementierung der objektorientierten als auch mit denen der ObjectTeams-orientierten Version des Frameworks JHotDraw vertraut bin, kann ich die Verständlichkeit für einen Erstbenutzer des Framworks nur schlecht objektiv beurteilen und bin darauf angewiesen, Annahmen zu treffen. Die Überprüfung dieser, im folgenden präsentierten Annahmen durch die Erstellung äquivalenter Applikationen auf der Grundlage beider Framework-Implementierungen durch "neutrale" Personen, das heißt durch solche Personen, die mit den sprachlichen Konzepten von Java und ObjectTeams/Java, nicht aber mit JHotDraw vertraut sind, sollte Thema weiterführender Forschungsaktivität sein.

Ich gewann den Eindruck, dass die Implementierung der Lade-Speicher-Programmfunktion, insbesondere die des Standardspeicherformates, durch die Kapselung in Teams übersichtlicher und dadurch verständlicher ist als ihr objektorientiertes Gegenstück.

Dieser Umstand führt aber nicht dazu, dass die Instanziierung einer auf JHotDraw aufbauenden Applikation einfacher wird. Kapitel 4.5 zeigt, dass der Aufwand für eine Instanziierung sogar steigt.
Wo bei der objektorientierten Framework-Instanziierung lediglich Methoden überschrieben werden müssen, und zwar in Klassen, die die Applikation ohnehin zur Bereitstellung ihrer Grundfunktionalität selbst erstellen muss (Subklasse von `DrawApplication`, spezifische Figurtypen), müssen bei der ObjectTeams-orientierten Instanziierung existierende Team- und Rollenklassen überschrieben und/oder neue von ihnen abgeleitet, das vom Framework verwendete Team-Objekt durch eine Instanz der spezielleren Teamklasse der Applikation ersetzt und aktiviert werden.
Abgesehen vom praktischen Mehraufwand, müssen bei der Verwendung der ObjectTeams-orientierten Framework-Version aber vor allem mehr Abhängigkeiten explizit beachtet werden. Dazu gehört zum Beispiel, dass die Hierarchie der `Storable`-Rollenklassen im `StandardStorageFormatTeam` und deren Fortführung in Sub-Teams ein Spiegelbild der Vererbungshierarchie der an die Rollen gebundenen Basisklassen sein muss. Das heißt also, bestimmte Abhängigkeiten, die in der objektorientierten Framework-Version explizit in Form von "Code Tangling" strukturell verankert sind, müssen bei der Instanziierung nicht mehr beachtet werden, wohingegen die selben Abhängigkeiten in der ObjectTeams-orientierten Framework-Version nur implizit strukturell verankert sind, dafür aber bei der Instanziierung explizit beachtet und daher nicht nur zusätzlich in die Framework-Dokumentation aufgenommen werden müssen, sondern auch eine potentielle Fehlerquelle bilden. Für den in diesem Kapitel besprochenen Beispielfall der Implementierung der Lade-Speicher-Programmfunktion erhöht sich also die Komplexität der Instanziierung des Frameworks durch die Verwendung von ObjectTeams.

### 4.6.6 Weitere Eigenschaften der ObjectTeams-orientierten Implementierung

Die im Rahmen der Restrukturierung der Lade-Speicher-Funktion entstandene und in Kapitel 4.4.2 beschriebene Klasse `StandardStorageFormatTeam` ist ein Beispiel für ein Team, welches wie eine normale Klasse benutzt wird. Es wird über den Aufruf seiner Teamlevel-Methoden verwendet, enthält keine callin-Bindungen und muss daher nicht aktiviert werden.

Die in Kapitel 4.4.1 beschriebene Klasse `StorageTeam` ist (in allen drei vorgestellten möglichen Varianten) ein Beispiel für ein Team, welches keine öffentliche Teamlevel-Methode sondern (nur eine einzige) callin-Bindung enthält. Daher muss es aktiviert werden, damit die von ihm zur Verfügung gestellte Programmfunktion in einer Applikation enthalten ist.

Beim Vergleich der drei in Kapitel 4.4.1 vorgestellten Design-Ansätze untereinander schneidet das erste, auf Seite 33 beschriebene, konzeptuell am besten ab, weil es die Lade-Speicher-Funktionalität am besten kapselt (die größte Lokalität aufweist), keine expliziten Liftings erfordert, am verständlichsten ist, und auf seiner Grundlage einfacher und sauberer als bei den anderen Design-Ansätzen eine Applikation erstellt werden kann (siehe Abbildung 21 auf Seite 50 im Vergleich zu Abbildung 22). Leider ist es aufgrund der verschachtelten Team-Struktur mit der derzeitigen ObjectTeam/Java-Version nicht realisierbar.

## 4.7 Ausblick

Über die im Rahmen der vorliegenden Fallstudie durchgeführten Arbeiten und gesammelten Erkenntnisse hinaus bleiben noch Fragen und Aufgaben offen, die im folgenden erläutert werden sollen.

Zuallererst müssen die im Rahmen dieses Kapitels vorgestellten ObjectTeams-orientierten Modellierungen der Lade-Speicher-Programmfunktion vollständig implementiert und in diesem Rahmen der ObjectTeams/Java-Compiler und die zugehörige Laufzeitumgebung soweit repariert und erweitert werden, dass die Implementierungen uneingeschränkt kompilierbar und lauffähig werden.

Weiterhin sollten aufbauend auf den verschiedenen Framework-Implementierungen, und zwar sowohl auf der originalen objektorientierten als auch auf den ObjectTeams-orientierten Versionen, zum Zwecke des Vergleichs und der Berwertung der Framework-Verständlichkeit und -Instanziierbarkeit Applikationen erstellt werden. Insbesondere wäre es vorteilhaft, wenn diese Aufgabe von "neutralen" Personen ausgeführt würde. Das heißt, von solchen Personen, die sowohl mit Java als auch mit ObjectTeams/Java und den zugehörigen Konzepten, nicht aber mit JHotDraw eingehend vertraut sind.

Anhand der in einem solchen Rahmen entstandenen Beispiel-Applikationen können dann die verschiedenen ObjectTeams-orientierten Framework-Versionen getestet werden, um zu untersuchen, ob durch die Restrukturierung des Frameworks Seiteneffekte und/oder Wechselwirkungen mit anderen Programmfunktionen auftreten, die im Rahmen der bisher angestellten Untersuchungen und Überlegungen nicht berücksichtigt wurden.

Im Rahmen der beschriebenen Restrukturierung der in der objektorientierten Originalversion des Frameworks JHotDraw vorgefundenen Implementierung der Lade-Speicher-Programmfunktion lag der Fokus auf der Frage der Machbarkeit. Die zur Programmfunktion gehörenden Implementierungs-elemente wurden aus der Basissoftware extrahiert, in ein Team verschoben und direkt, das heißt in der selben Team-Klasse, wieder an die Klassen gebunden, aus denen sie zuvor entfernt wurden.
Im Zusammenhang mit der Frage der Wiederverwendbarkeit einer in einem Team gekapselten Programmfunktion ist es notwendig zu untersuchen, inwieweit im Falle der Klassen `StorageTeam` und `StandardStorage-`

`FormatTeam` eine Abstraktion vorgenommen werden kann, die es ermöglichen würde, die ObjectTeams-orientierte Implementierung der Lade-Speicher-Programmfunktion auch in anderen Kontexten als dem Framework JHotDraw einzusetzen.
Neben den eben genannten gibt es noch zwei weitere offene Forschungsbereiche, die schon etwas detaillierter angedacht wurden und in den folgenden beiden Unterkapiteln dargestellt werden sollen. Dabei handelt es sich um die interne Strukturierung von Teams mit sehr vielen Rollenklassen und das Speichern von Rollenobjekten.

### 4.7.1 Interne Strukturierung von Teams mit einer großen Anzahl von Rollenklassen

Das im Rahmen der in diesem Kapitel beschriebenen Restrukturierung der Lade-Speicher-Programmfunktion entstandene `StandardStorageFormat-Team` enthält sehr viele, nämlich ca. 80 Rollenklassen. Darunter sind einerseits die Klassen `StorableInput` und `StorableOutput` und andererseits die Hierarchie des Interface `Storable`. Im Zusammenhang mit zu speichernden Rollen (siehe Kapitel 4.7.2) könnte noch einmal eine erhebliche Anzahl von `Storable`-Subtypen hinzukommen. Spätestens dann würde der Inhalt des Teams sehr unübersichtlich (auch wenn die Rollenklassen natürlich in separaten Dateien definiert sind). Es stellt sich die Frage, ob und wie ein Team, welches sehr viele Rollenklassen enthält, weiterführend selbst besser strukturiert werden kann? Es muss gefragt werden, welche Möglichkeiten es gibt, die Rollenklassen zu gruppieren, und welche Konsequenzen mit einer derartigen Gruppierung verbunden sind.

Die folgenden drei Punkte stellen Ideen für potentielle Gruppierungs-möglichkeiten dar:
- Pakete innerhalb des Teams
- nested Teams innerhalb des Teams
- Subteams

Für die erste Variante, Pakete im Team, die in den derzeitigen Konzepten von ObjectTeams nicht vorgesehen ist, ist zu untersuchen, ob sie realisierbar wäre. Insbesondere ist zu erörtern, welche Vor- und Nachteile oder eventuelle Anforderungen sich aus der resultierenden Trennung der Namensräume innerhalb des Teams ergeben.

Für die zweite und dritte Variante ist für das Beispiel des `StandardStorageFormatTeams` zu untersuchen, von wem, wo und in welcher Reihenfolge, welche der einzelnen Teams erzeugt und aktiviert werden müssten. Es muss erforscht werden, ob diese Varianten der Team-Strukturierung in Bezug auf den Kontrollfluss beim Speichern und Laden überhaupt möglich sind.

Sollte sich bei den eben erläuterten Nachforschungen herausstellen, dass die genannten Strukturierungsvarianten anwendbar sind, muss untersucht werden, welche Konsequenzen sie in Bezug auf die Instanziierung einer auf JHotDraw basierenden Applikation haben.

### 4.7.2 Speicherung von Rollenklassen

Ein Objekt wird in einer Datei gespeichert, indem der Name seiner Klasse und

sein Zustand gespeichert wird. Dieser Objektzustand definiert sich in der Objektorientierung über die Werte seiner Attribute und kann einfach ermittelt werden, weil es sich bei Attributen um Referenzen handelt, die das Objekt selbst hält.

Durch die Einführung der ObjectTeams-Konzepte definiert sich der Zustand eines Objekts nicht mehr ausschließlich über seine eigenen Attributwerte. Existieren im System Rollenobjekte, die mit ihm assoziiert sind, und besitzen diese Rollenobjekte selbst Attribute und damit eigene Zustände, müssen diese Zustände der Rollenobjekte unter Umständen bei der Betrachtung des Zustands des Basisobjekts mit berücksichtigt werden.

Dabei ist die Ermittlung dieses "Gesamtzustandes" nicht ganz so einfach, weil zu einem Basisobjekt einerseits beliebig viele Rollenobjekte aus ein und demselben sowie aus verschiedenen Team-Kontexten existieren können, ein Basisobjekt von der Existenz dieser Rollenobjekte nichts "weiß", das heißt keine Referenzen auf diese hält, ObjectTeams/Java in seiner derzeitigen Version keine Reflection-Funktionalitäten anbietet und Rollenobjekte außerhalb der sie umschließenden Team-Instanz nur in Form Teaminstanz-gebunden getypter Attribute gehalten werden können.

Daher ist es erforderlich, sich mit der Frage zu beschäftigen, ob und wie der derzeitige Mechanismus zur Speicherung von Objekten im Standardspeicherformat erweitert werden kann, damit eine vom Entwickler festzulegende Teilmenge der an ein Objekt gebundenen Rollenobjekte im Zuge der Speicherung des Objekts mitgespeichert werden kann.

Sollen die mit einem bestimmten Basisobjekt assoziierten Rollenobjekte mitgespeichert werden, ist es zuallererst erforderlich, diese zu kennen. Das heißt, es wird eine ObjectTeams/Java-Reflection-Methode benötigt, die zu einem übergebenen Objekt beliebigen Typs eine Liste aller im System vorhandenen zugehörigen Rollenobjekte zurückliefert. Dabei ist es in diesem Fall unerheblich, ob die Rollen aus aktiven Teams stammen oder nicht, weil auch eine Rolle, die z.B. nur über callouts gebunden ist, einen eigenen Zustand haben kann, der gespeichert werden soll. In diesem Zusammenhang ist zu klären, welchen Rückgabetyp eine solche Methode hat bzw. ob und wenn in welcher Form solch eine Methode überhaupt möglich ist, da Rollenobjekte außerhalb ihres Team-Kontexts nur in Teaminstanz-gebunden-getypter Form zugegriffen werden dürfen. Die erforderliche Methode müsste in etwa so aussehen: `Role[] OTSystem.getAllRoles(Object baseObj)`

Neben der eben erläuterten Notwendigkeit, auf alle zu einem Basisobjekt gehörenden Rollenobjekte zugreifen zu können, muss es möglich sein, festzulegen bzw. zu ermitteln, welche dieser Rollenobjekte gespeichert werden sollen. Diese Festlegung könnte mit Hilfe der existierenden, in Kapitel 4.4.2 beschriebenen Struktur erfolgen. Genauso, wie eine gewöhnliche Klasse, deren Objekte speicherbar sein sollen, müsste jede Rollenklasse, deren Objekte gespeichert werden sollen, eine an sie gebundene Rollenklasse des Typs `Storable` im `StandardStorageFormatTeam` besitzen. Dies ist in der derzeitigen ObjectTeams/Java-Version nicht möglich.
Da ein Rollenobjekt nur im Zusammenhang mit einer umschließenden Team-Instanz existieren kann, müssten auch die Team-Klassen, die speicherbare Rollenklassen beinhalten, jeweils eine an sie gebundene Rolle des Typs `Storable` im `StandardStorageFormatTeam` enthalten.

Unter der Voraussetzung, dass einerseits eine Reflection-Methode existiert, die zu einem Objekt beliebigen Typs alle existierenden Rollenobjekte liefert und zweitens Rollenklassen an Rollenklassen gebunden werden können, ist es denkbar, dass der vorhandene rekursive Speichermechanismus so erweitert werden kann, dass bei der Speicherung eines Objekts alle zu ihm gehörenden speicherbaren Rollenobjekte mitgespeichert werden.
Dabei ist zu beachten, dass ein Rollenobjekt nur in Abhängigkeit von seiner Team-Instanz und seinem Basisobjekt existieren kann und durch Lifting erzeugt wird. Daher funktioniert das Laden aus einer Datei bei Rollenobjekten anders als bei Objekten gewöhnlicher Klassen und es ist notwendig, die Daten eines Rollenobjekts beim Speichern in die Datei als solche zu kennzeichnen und Referenzen auf Team-Instanz und Basisobjekt mitzuspeichern.

Die weitere Untersuchung und beispielhafte Umsetzung der beschriebenen Sachverhalte sollte das Thema zukünftiger Forschungsaktivität sein.

# 5 Aspekte der Team-Erzeugung und -Aktivierung

Nachdem im vorangegangenen Kapitel ausführlich untersucht wurde, wie mit Hilfe der ObjectTeams-Konzepte eine bestimmte Programmfunktion strukturell aus einer Software herausgelöst und ihre Strukturen und Kollaborationen in einem Team gekapselt werden können, beschäftigt sich dieses Kapitel nun mit der Integration der resultierenden Teile zu einem lauffähigen, korrekt funktionierenden Programm. Diese Integration erfolgt in ObjectTeams mittels des Konzepts der Team-Aktivierung. Im vorliegenden Kapitel werden die bei der Arbeit mit dem Framework JHotDraw sichtbar gewordenen Aspekte der Team-Erzeugung und -Aktivierung anhand des Beispiels der Integration der Programmfunktion "Laden und Speichern" untersucht.
Der Hauptteil des Kapitels beschäftigt sich damit, wie die zur Aktivierung eines bestimmten Teams geeigneten Kontrollflussstellen und die vorzunehmende Art der Aktivierung ermittelt werden können. Insbesondere wird dabei darauf eingegangen, welche der Anforderungen, die sich ganz allgemein an das Programm und im Besonderen an ein Framework stellen, sich darauf auswirken, das heißt durch welche Faktoren Team-Erzeugung und -Aktivierung beeinflusst werden. Die Annäherung an diese Thematik erfolgt dabei schrittweise. Sie zeichnet den während der Entwicklung durchlaufenen gedanklichen Prozess nach. Im Laufe dieses Prozesses kristallisieren sich zwei verschiedene mögliche Verfahren zur Handhabung des Beispiel-Teams bezüglich seiner Erzeugung und Aktivierung heraus. Diese unterscheiden sich sowohl in der Anzahl der erzeugten Team-Instanzen, als auch in Bezug auf den Ort und die Art der Aktivierung.
An die Beschreibung des erwähnten Prozesses und der in seinem Rahmen gefundenen zwei möglichen Team-Handhabungs-Verfahren, schließt sich dann die Darstellung der Konsequenzen an, die die Entscheidung für eines der beiden Verfahren mit sich bringt.
Da dies zum Verständnis der nachfolgenden Teile erforderlich ist, werden einleitend wichtige Implementierungsdetails im Zusammenhang mit dem Start JHotDraw-basierter Programme und der Programmfunktion "Öffnen eines Editorfensters" beleuchtet.

## 5.1 Starten JHotDraw-basierter Programme (Öffnen von Editorfenstern)

Die zentrale Bedeutung der Klasse DrawApplication

Die Klasse `DrawApplication` spielt die zentrale Rolle bei der Erstellung und dem Programmstart von JHotDraw basierten Applikationen (siehe Abbildung 23). Sie beerbt die Java-Swing-Klasse `JFrame` und stellt die graphische Benutzeroberfläche zur Verfügung. Desweiteren ist sie in der objektorientierten Implementierung für die Verwaltung zentraler Controller-Objekte wichtiger Programmfunktionen verantwortlich. Dazu gehört unter anderem `StorageFormatManager` für die Funktionen Laden und Speichern. Die zentrale Klasse einer auf JHotDraw aufbauenden Applikation ist immer eine Subklasse von `DrawApplication`. Ein Objekt des Typs `DrawApplication` entspricht einem Editorfenster. Daher werden diese Begriffe in den folgenden Ausführungen synonym verwendet.

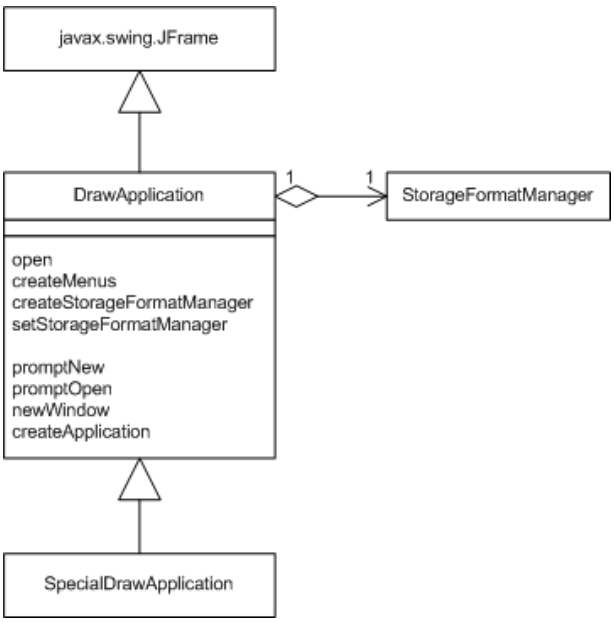


***Abbildung 23:*** *Programmfunktion „Editorfenster öffnen“, objektorientierte Struktur*

Die Initialisierungsmethode open

Zur Initialisierung des Editorfensters existiert die Methode `open` (siehe Abbildung 23). Sie veranlasst unter anderem die Erzeugung der Menüs (`createMenus`) und des oben erwähnten Controller-Objekts (`set/createStorageFormatManager`), layoutet die sichtbaren Elemente und macht das Editorfenster auf dem Bildschirm sichtbar. Um ein Editorfenster zu öffnen, muss ein Objekt des Typs `DrawApplication` instanziiert und auf diesem die Methode `open` aufgerufen werden. Ist noch kein Editorfenster offen, geschieht dies entweder ganz klassisch durch eine statische `Main`-Methode oder zum Beispiel auch aus einem beliebigen Programm heraus.

Programmfunktion „Öffnen eines neuen Editorfensters“

Ist bereits ein Editorfenster offen, kann der Benutzer durch Auswahl der Menüoptionen "New" (entspricht Methode `DrawApplication.promptNew` ) oder "Open" (`promptOpen`) des Dateimenüs weitere Editorfenster öffnen. Die Editorfenster sind voneinander unabhängig. Wird eins von ihnen geschlossen, hat dies keine Auswirkungen auf die noch offenen. Zur Realisierung dieser Programmfunktion verfügt die Klasse `DrawApplication` über die Methode `newWindow`. Diese veranlasst die Erzeugung und Öffnung neuer Editorfenster, indem sie zuerst `createApplication` und anschließend auf dem von dieser Methode gelieferten Objekt des Typs `DrawApplication` die Methode `open` aufruft (siehe Abbildung 24).

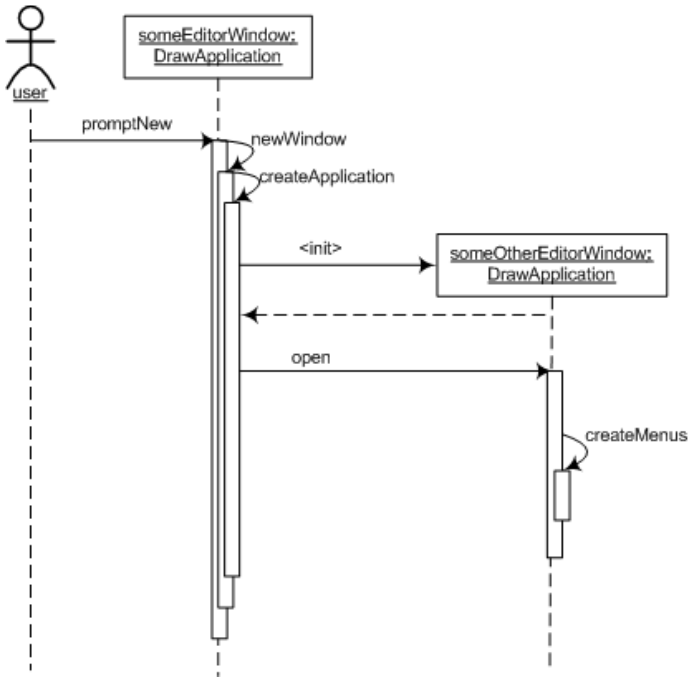


***Abbildung 24:*** *Programmfunktion „Editorfenster öffnen“, objektorientiertes Verhalten*

Factory-Methode createApplication

In der Framework-Implementierung erzeugt die Methode `createApplication` der Klasse `DrawApplication` ein neues Objekt der Klasse `DrawApplication`. Die Methode `createApplication` ist eine Factory-Methode (siehe [GHJV95], Seite 107ff.). Sie ist dazu gedacht, von Subklassen der Klasse `DrawApplication` bei Bedarf überschrieben zu werden.

„Erwartungskonforme“ Implementierung

Von einer Editor-Applikation, also zum Beispiel einem UML-Editor, erwartet der Benutzer normalerweise, dass sich nach Auswahl von "New" oder "Open" im Dateimenü ein neues Fenster des schon laufenden Editor-Programms, also im Beispielfall wieder ein UML-Editor, öffnet. Damit eine auf JHotDraw basierende Applikation diese Anforderung erfüllen kann, muss die zugehörige `DrawApplication`-Subklasse die Methode `createApplication` entsprechend überschreiben. Im angesprochenen Beispiel des UML-Editors müsste die Methode `createApplication` der `DrawApplication`-Subklasse `UMLEditor` also ein Objekt der Klasse `UMLEditor` erzeugen (siehe Abbildung 25,weiß unterlegte Implementierungsnotiz).

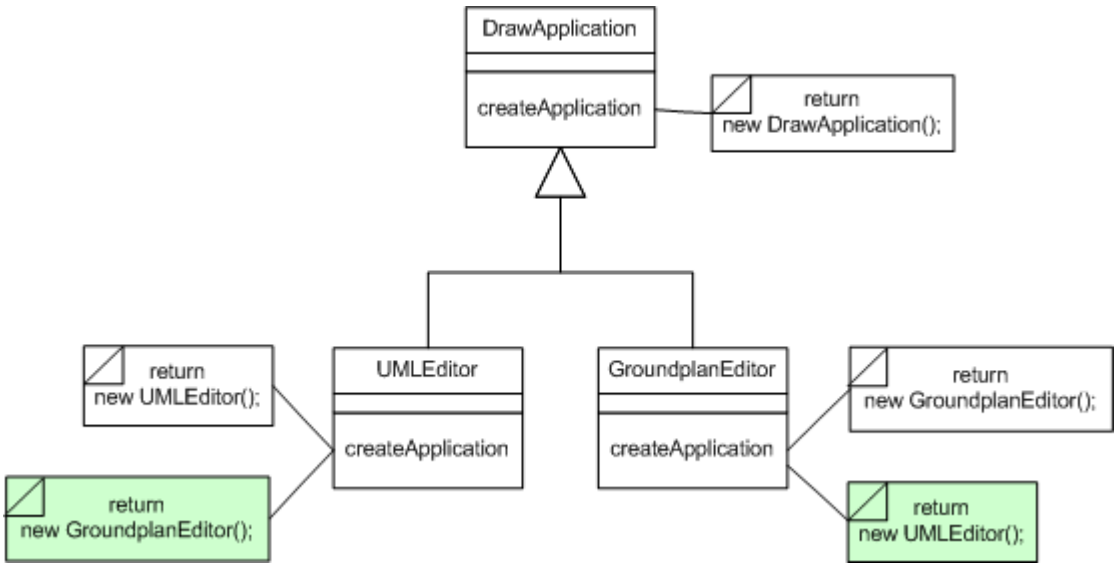


***Abbildung 25:*** *Programmfunktion „Editorfenster öffnen“, objektorientierte Struktur und Verhalten, Ausschnitt*

„Untypische“ Implementierung

Obwohl die Factory-Methode `createApplication` sehr wahrscheinlich entworfen wurde, um das eben beschriebene intuitiv erwartete Programmverhalten zu realisieren, ermöglicht diese Programmstruktur auch andere Anwendungen. Unter der Annahme, dass auch die Klasse `GroundPlanEditor` eine Subklassen von `DrawApplication` darstellt, könnte zum Beispiel `UMLEditor` in ihrer Methode `createApplication` `GroundPlanEditor` instanziieren und `GroundPlanEditor` in ihrer Methode `createApplication` wiederum `UMLEditor` (siehe Abbildung 25, farbig unterlegte Implementierungsnotiz). Aus Benutzersicht betrachtet hieße das, dass sich nach Auswahl der Menüoption „New“ des Dateimenüs eines Editors für UML-Diagramme ein Editor für Grundrisszeichnungen öffnen würde und umgekehrt. Ein solches Programmverhalten ist untypisch und für den Benutzer daher unerwartet, weshalb dieses Beispiel spielerisch und eher theoretisch anmutet. Im Hinblick auf die im folgenden diskutierten Aspekte der Team-Erzeugung und -Aktivierung ist es aber wichtig, im Hinterkopf zu behalten, dass das Framework diese Möglichkeiten bietet. Insbesondere betrifft dies das Kapitel 5.3.1„Funktionale Einschränkung durch erstes Team-Handhabungs-Verfahren“ auf Seite 73.

## 5.2 Bestimmung von Ort und Art der Team-Erzeugung und -Aktivierung

Soll eine bestimmte Funktionalität in Form eines Teams in eine Software integriert werden, ergeben sich hinsichtlich des geeigneten Ortes (Kontrollflussstelle) und der möglichen Art der Integration (Level der Team-Aktivierung) bestimmte Einschränkungen. Diese Einschränkungen können zurückgeführt werden auf bestimmte Anforderungen an die Funktionsweise der Software. Dieses Kapitel zeigt, wie die zur Aktivierung eines bestimmten Teams geeigneten Kontrollflussstellen und die vorzunehmende Art der Aktivierung ermittelt werden können. Zu diesem Zweck wird beispielhaft der bei der Integration der Programmfunktion „Laden und Speichern" durchlaufene Lösungsfindungsprozess systematisch nachgezeichnet, indem die aufgetauchten Anforderungen Schritt für Schritt aufgezeigt und die in diesem Rahmen ermittelten zwei möglichen Lösungen vorgestellt werden.

Erste Anforderung – Aktivierung vor dem Aufruf callin-gebundener Basis-methoden

Wenn eine durch ein Team gekapselte Programmfunktion in ein Programm integriert und dafür eine für die Team-Aktivierung geeignete Kontrollflussstelle gefunden werden soll, müssen zuallererst die funktionalen Verbindungsstellen zwischen Team und Programm betrachtet werden. Es muss der Frage nachgegangen werden, welche Funktionalität des Teams zur Laufzeit ins Programm eingewebt werden muss und vor allem, wann diese Funktionalität zum Einsatz kommen soll. Da das Einweben von Funktionalität technisch durch callin-Bindung von Rollenmethoden an die Ausführung von Basismethoden realisiert wird, stehen in diesem ersten Schritt also die im Team enthaltenen callin-Bindungen im Blickfeld.
Eine Rollenmethode, die per callin an eine Basismethode gebunden ist, wird genau dann ausgeführt, wenn die Basismethode aufgerufen wird, allerdings nur dann, wenn das Team zu diesem Zeitpunkt aktiv, also die Programmfunktion „eingeschaltet“ ist. Soll die Rollenmethode bei jedem Aufruf der Basismethode ausgeführt werden, muss das Team also aktiviert werden, bevor die entsprechende Basismethode zum ersten Mal aufgerufen wird. Damit alle in einem Team enthaltenen callin-gebundenen Rollenmethoden auf diese Weise zum Einsatz kommen können, muss dafür Sorge getragen werden, dass das Team aktiviert ist, bevor zum ersten Mal ein Aufruf auf einer der callin-gebundenen Basissmethoden stattfindet. Soweit der allgemeine Ansatz. Seine Anwendung auf das Beispiel der Integration der Programmfunktion "Laden und Speichern" in das Framework JHotDraw bzw. in ein auf diesem basierendes Programm zeigt der folgende Abschnitt.

Die Programmfunktion „Laden und Speichern" wird durch die Team-Klasse `StorageTeam` gekapselt. Diese enthält insgesamt nur eine einzige callin-gebundene Rollenmethode. Diese Rollenmethode dient der Erweiterung eines Menüs um die für die Programmfunktion „Laden und Speichern“ relevanten Optionen „Open" und „Save" und trägt daher den Namen `extendMenu`. Weil diese Menüoptionen im Dateimenü des Editorfensters enthalten sein sollen, wurde die Rollenmethode `extendMenu` an die Basismethode `createFileMenu` gebunden, die für die Erzeugung des Dateimenüs verantwortlich ist (siehe Abbildung 26).

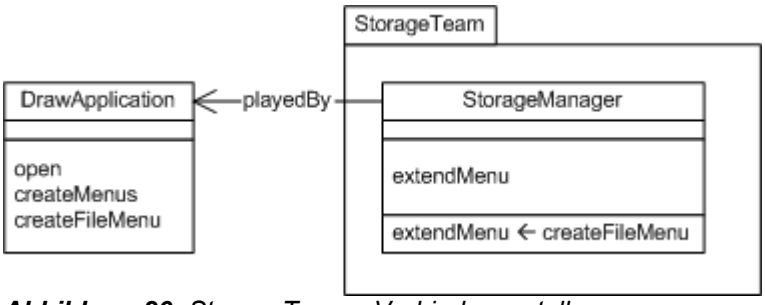


***Abbildung 26:*** *StorageTeam - Verbindungsstelle zum Programm*

Damit die Erweiterung des Dateimenüs um die für das Laden und Speichern notwendigen Optionen erfolgen kann, muss das `StorageTeam` aktiviert werden, bevor die Erzeugung des Dateimenüs durch den Aufruf der Basismethode `createFileMenu` angestoßen wird. Dieser Aufruf erfolgt im Rahmen der im Abschnitt „Die Initialisierungsmethode open“ auf Seite 62 erläuterten Initialisierung des Editorfensters. Dabei ruft die

Initialisierungsmethode `open` die Menüerzeugungsmethode `createMenus` und diese wiederum `createFileMenu` auf. Unter Berücksichtigung der Einschränkung, dass das Team vor dem Aufruf von createFileMenu aktiviert werden muss, bieten sich alle Stellen innerhalb der Methode `createMenus`, die sich vor dem Aufruf der Methode `createFileMenu` befinden, und alle Stellen innerhalb der Methode `open`, die sich vor dem Aufruf der Methode `createMenus` befinden, als potentiell geeignete Kontrollflussstellen zur Erzeugung und Aktivierung des `StorageTeams` an.
Auch wenn sich die Verbindungsstelle des `StorageTeams` zum Programm im Bereich der Menüerzeugung und -Erweiterung befindet, geht dessen Funktionalität jedoch weit darüber hinaus. Daher wurde entschieden, dass die Methode `open` aufgrund ihrer Eigenschaft als allgemeine Initialisierungsmethode des Programms (siehe Abschnitt „Die Initialisierungsmethode open“ auf Seite 62) einen passenderen Rahmen für die Team-Erzeugung und -Aktivierung bietet als createMenus.

Erste Lösung

Das Ergebnis der ersten Etappe auf der Suche nach einer geeigneten Kontrollflussstelle zur Erzeugung und Aktivierung des `StorageTeams` ist also die Methode `open` der Klasse `DrawApplication`. Da davon ausgegangen wurde, dass es ausreichend ist, die Einschränkungen zu berücksichtigen, die sich hinsichtlich der vorhandenen callin-Bindungen ergeben, wurde angenommen, es handele sich dabei um eine angemessene, funktionierende Lösung und die Suche wäre damit beendet.

Problematik der ersten Lösung

Diese Annahme erwies sich im Laufe der weiteren Erforschung des Frameworks JHotDraw als zu kurzsichtig. Es wurde nicht berücksichtigt, dass der Umstand, dass die Klasse `DrawApplication` gleichzeitig Erzeuger des `StorageTeams` und Basis einer im Team enthaltenen Rolle ist, zu Problemen und Wechselwirkungen mit anderen Programmfunktionen führen könnte.
Für ein Editorfenster, das heißt für eine Instanz des Typs `DrawApplication`, wird die Methode `open`, welche das `StorageTeam` erzeugt und aktiviert, zwar nur genau einmal aufgerufen. Unter Umständen gibt es aber mehrere Editorfenster und damit auch mehrere `StorageTeam`-Instanzen gleichzeitig im System. Dabei kommt es zu fehlerhaftem Programmverhalten, weil jede aktive `StorageTeam`-Instanz auf jedes vorhandene `DrawApplication`-Objekt mit callin-Aktivität reagiert.
Zu Beginn existiert nur ein Editorfenster, das heißt eine `DrawApplication`-Instanz. Wenn auf dieser die Methode `open` ausgeführt wird, wird eine `StorageTeam`-Instanz erzeugt. Durch den anschließend erfolgenden Aufruf der Methode `createFileMenu`, wird in der `StorageTeam`-Instanz ein `StorageManager`-Rollenobjekt erzeugt und auf diesem die callin-Methode `extendMenu` aufgerufen. Diese erweitert das Dateimenü des Editorfensters um die Optionen "Open" und "Save".
Nachdem das Editorfenster auf dem Bildschirm sichtbar geworden ist, kann der Programmbenutzer mit der Dateimenüoption "New" die Erzeugung eines weiteren Editorfensters veranlassen (siehe Abschnitt „Programmfunktion „Öffnen eines neuen Editorfensters““ auf Seite 62). Programmintern passiert dabei folgendes: Es wird eine zweite `DrawApplication`-Instanz erzeugt und auch auf dieser `open`, dadurch indirekt `createMenus` und `createFileMenu` aufgerufen. Dieses zweite `DrawApplication`-Objekt erzeugt und aktiviert nun innerhalb von `open` eine neue, zweite `StorageTeam`-Instanz. Da sich nun insgesamt zwei `StorageTeam`-Instanzen im System befinden, wird beim Aufruf

von createFileMenu auf dem neuen DrawApplication-Objekt in beiden StorageTeam-Instanzen jeweils ein neues StorageManager-Rollenobjekt erzeugt und die callin-Methode extendMenu ausgeführt. Das hat zur Folge, dass das Dateimenü des zweiten Editorfensters zweimal um die Menüoptionen „Open" und „Save" erweitert wird.
Jedes weitere vom Benutzer geöffnete Editorfenster enthält genau so viele Paare der Menüoptionen „Open" und „Save" in seinem Dateimenü, wie Editorfenster und damit auch StorageTeam-Instanzen im System existieren. Dieses Programmverhalten ist eindeutig unerwünscht. Es handelt sich um einen Programmfehler, der auf den Ort bzw. die Art der Team-Aktivierung zurückgeführt werden kann.

Zweite Anforderung – callin-Methode darf pro Basisobjekt nur einmal ausgeführt werden

Das eben beschriebene Problem macht eine zuvor übersehene, zweite wichtige Anforderung deutlich. Die Menüerweiterung darf für jedes Editorfenster nur einmal erfolgen. Übertragen auf die Implementierungsebene bedeutet das, dass die Rollenmethode StorageManager.extendMenu für jedes DrawApplication-Basisobjekt nur einmal ausgeführt werden darf.

Um in Erfahrung zu bringen, wie diese Anforderung erfüllt werden kann, muss die Implementierung noch einmal genauer betrachtet werden. Die Rollenmethode StorageManager.extendMenu ist per callin an die Basismethode DrawApplication.createFileMenu gebunden und wird daher nur dann ausgeführt, wenn diese aufgerufen wird. Die Basismethode createFileMenu wird ausschließlich im Rahmen der Initialisierung eines DrawApplication-Objekts, das heißt nur genau einmal pro Objekt, ausgeführt. Das bedeutet, dass auch die Rollenmethode extendMenu nicht öfter als einmal pro StorageManager-Rollenobjekt ausgeführt wird. Dieser Sachverhalt führt zu der Schlussfolgerung, dass die Anforderung, die Rollenmethode StorageManager.extendMenu bezogen auf den gesamten Programmlauf nur einmal pro DrawApplication-Basisobjekt auszuführen, erfüllt wird, wenn zu jedem Basisobjekt nicht mehr als ein StorageManager-Rollenobjekt existiert.

Dieses Ziel, systemweit nur ein StorageManager-Rollenobjekt pro DrawApplication-Basisobjekt zu halten, kann auf zwei Arten erreicht werden, die zwei verschiedene funktionierende Team-Handhabungs-Verfahren darstellen.

Erstes Team-Handhabungs-Verfahren

Beim ersten Team-Handhabungs-Verfahren gibt es systemweit insgesamt nur eine StorageTeam-Instanz. Diese enthält für jedes existierende DrawApplication-Basisobjekt ein assoziiertes StorageManager-Rollenobjekt (siehe Abbildung 27).

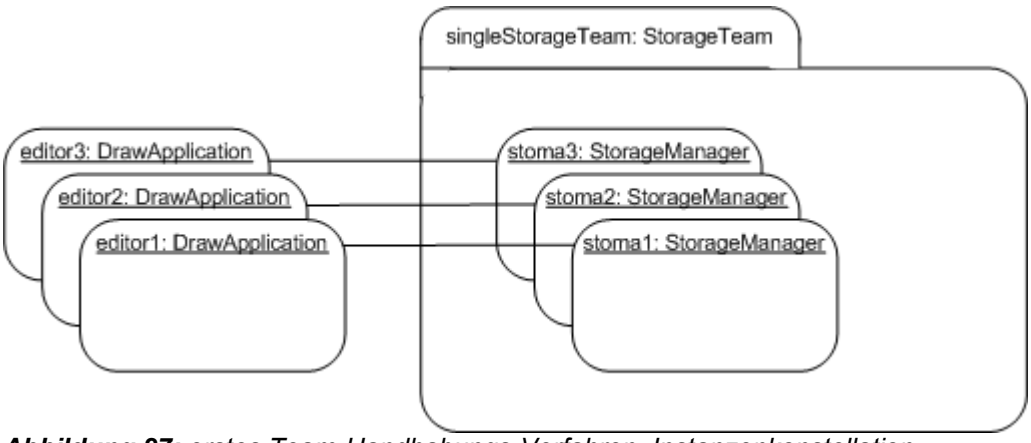


***Abbildung 27:*** *erstes Team-Handhabungs-Verfahren, Instanzenkonstellation*

Um diese Konstellation zu erreichen, muss die Kontrollflussstelle, an der die Erzeugung und Aktivierung des Teams erfolgt, zwei Ansprüchen gerecht werden. Einerseits darf sie selbst nur einmal durchlaufen werden, damit nicht mehrere Team-Instanzen erzeugt werden. Andererseits muss die Team-Aktivierung, wie im Abschnitt „Erste Anforderung“ auf Seite 65 erläutert, vor dem ersten Aufruf der callin-gebundenen Basismethode `DrawApplication.createFileMenu` erfolgen.
Da dieser erste Aufruf der Methode `createFileMenu` im Zuge des ersten Aufrufs der Initialisierungsmethode `DrawApplication.open` erfolgt, welcher seinerseits von der `Main`-Methode des Programms angestoßen wird, ist jeder beliebige Punkt innerhalb der `Main`-Methode, der vor dem Aufruf der Methode `open` und nicht innerhalb einer Schleife liegt, eine zur Team-Erzeugung und -Aktivierung geeignete Kontrollflussstelle (siehe Abbildung 28).

```
public class SomeSpecialDrawingEditor extends DrawApplication
{
    public static void main(String[] args)
    {
         ...
        (new StorageTeam()).activate();
         ...
        (new SomeSpecialDrawingEditor()).open();
    }
}
```

***Abbildung 28:*** *erstes Team-Handhabungs-Verfahren, Implementierungs-Elemente*

Zweites Team-Handhabungs-Verfahren

Anders als beim ersten Team-Handhabungs-Verfahren, bei dem insgesamt nur eine `StorageTeam`-Instanz existiert, existieren beim zweiten Verfahren viele. Genauso wie bei dem in den Abschnitten „Erste Lösung“ und „Problematik der ersten Lösung“ auf Seite 66 beschriebenen gescheiterten Ansatz, erzeugt jedes `DrawApplication`-Objekt in seiner Methode `open` eine `StorageTeam`-Instanz. Der Unterschied besteht darin, dass jede Team-Instanz nur ein einziges `StorageManager`-Rollenobjekt enthält. Dieses ist an das `DrawApplication`-Basisobjekt gebunden, welches die Team-Instanz erzeugt hat (siehe Abbildung 29). Dadurch ist für jedes `DrawApplication`-Objekt wie gefordert systemweit nicht mehr als ein `StorageManager`-Rollenobjekt vorhanden , weswegen die Menüerweiterung für jedes Editorfenster nur einmal erfolgt.

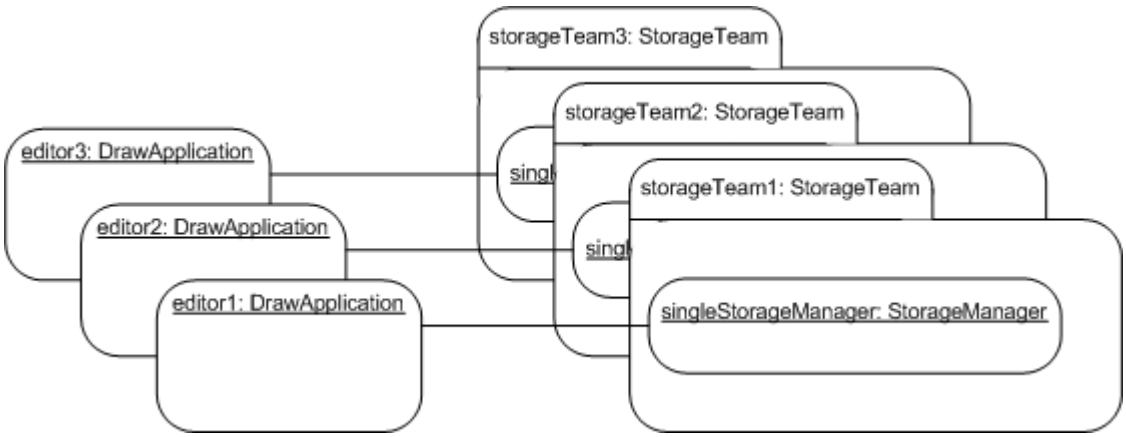


***Abbildung 29:****zweites Team-Handhabungsverfahren, Instanzenkonstellation*

Diese Konstellation wird erreicht, indem die Team-Aktivierung nicht vollständig und uneingeschränkt, sondern beschränkt auf das Aktivierungs-Level FROZEN erfolgt und sich das Team-erzeugende `DrawApplication`-Objekt bei seiner Team-Instanz als zu beobachtendes Basisobjekt registriert (siehe Abbildung 30 auf Seite 70).

Ein uneingeschränkt aktiviertes Team, wie beim ersten Team-Handhabungs-Verfahren, reagiert auf die Aktivitäten aller Objekte, zu deren Typen es gebundene Rollenklassen enthält, das heißt die für das Team Basisobjekte darstellen. Wird auf so einem Basisobjekt zum ersten Mal eine Methode ausgeführt, die per callin an eine Rollenmethode des Teams gebunden ist, erzeugt das Team zuerst ein neues mit dem Basisobjekt assoziiertes Rollenobjekt und führt dann auf diesem die callin-Methode aus.

Eine auf das Aktivierungs-Level FROZEN beschränkt aktivierte Team-Instanz reagiert dagegen anders. Wenn auf einem potentiellen Basisobjekt zum ersten Mal eine Methode ausgeführt wird, die per callin an eine Rollenmethode des Teams gebunden ist, führt dieser Vorgang nicht zur Erzeugung eines neuen Rollenobjekts. Die Rollenmethode wird nur dann ausgeführt, wenn schon vorher ein Rollenobjekt existiert hat. Existiert zu einem potentiellen Basisobjekt kein assoziiertes Rollenobjekt im Team, werden seine Aktivitäten vom Team ignoriert. Um ein Rollenobjekt zu erzeugen, muss bei einer FROZEN-aktivierten Team-Instanz das Basisobjekt explizit geliftet werden. Zu diesem Zweck muss die Teamklasse eine von außen zugreifbare (`public`) Registrierungsmethode enthalten, die ein Parameterlifting von Basistyp zu Rollentyp deklariert.

Für den hier behandelten konkreten Anwendungsfall bedeutet das, dass ein Objekt des Typs `DrawApplication` vor dem Aufruf seiner Methode `createFileMenu` nicht nur eine Instanz der Klasse `StorageTeam` erzeugen und mit dem Parameter FROZEN aktivieren, sondern zusätzlich sich selbst bei dieser Team-Instanz registrieren, das heißt liften lassen, muss (siehe Abbildung 30). Erst nach Durchführung dieser Maßnahme reagiert die Team-Instanz auf den Aufruf der Basismethode `createFileMenu` mit Ausführung der callin-Methode `extendMenu`.

```
public team class StorageManager
{
    public void registerAsStorageManager(DrawApplication as StorageManager){}
    ...
}

public class DrawApplication
{
    public void open()
    {    ...
         StorageTeam storageTeam = new StorageTeam();
         storageTeam.activate(org.objectteams.Team.FROZEN);
         storageTeam.registerAsStorageManager(this);
         ...
         createMenus();
         ...
    }

    public void createMenus(JMenuBar mb)
    {    ...
         addMenu(mb, createFileMenu());
         ...
    }
}

public class SomeSpecialDrawingEditor extends DrawApplication
{
    public static void main(String[] args)
    {
        (new SomeSpecialDrawingEditor()).open();
    }
}
```

***Abbildung 30:*** *zweites Team-Handhabungs-Verfahren, Implementierungs-Elemente*

Dritte Anforderung - Spezialisierbarkeit der verwendeten Team-Klassen

In der bisherigen Diskussion der Fragen der Team-Erzeugung und -Aktivierung wurde JHotDraw vereinfachend als Editor-Programm betrachtet. Im folgenden sollen die Anforderungen beleuchtet werden, die sich zusätzlich aus seiner Eigenschaft als Framework ergeben.

JHotDraw ist selbst kein lauffähiges Programm, sondern ein Framework. Als solches dient es als Grundlage für Programme. Seine Klassen sind dazu gedacht, von anderen Klassen beerbt zu werden. Seine Methoden sind so entworfen, dass sie in abgeleiteten Klassen den Bedürfnissen des jeweiligen Programms entsprechend einfach modifiziert werden können. Um diesem Ziel gerecht zu werden, sind die Methoden so modular wie möglich implementiert. Das bedeutet, dass Anweisungsfolgen, die funktionale Einheiten bilden, systematisch in Untermethoden zusammengefasst sind. Dieses Vorgehen erleichtert es, in Subklassen einzelne Aspekte des Programmverhaltens zu überschreiben. In diesem Zusammenhang wurde viel Gebrauch von den Design Pattern „Template Method" sowie „Factory Method“ (siehe [GHJV95], Seite 325ff. bzw. 107ff.) gemacht.
Die Notwendigkeit derartiger Flexibilität macht auch vor den im Rahmen der ObjectTeam-orientierten Umstrukturierung des Frameworks entstandenen Teams nicht halt. Um das korrekte Funktionieren eines auf JHotDraw basierenden Programms gewährleisten zu können, ist es unter Umständen erforderlich, diesem die Möglichkeit zu geben, anstelle der vom Framework zur Verfügung gestellten Team-Klasse `StorageTeam` eine eigene von diesem abgeleitete Team-Klasse zu verwenden.

Implizite Erfüllung der dritten Anforderung beim ersten Team-Handhabungs-Verfahren

Bei dem im Abschnitt „Erstes Team-Handhabungs-Verfahren“ auf Seite 67 vorgestellten Ansatz erfolgt die Erzeugung und Aktivierung der benötigten Team-Instanz innerhalb der `main`-Methode des Programms (siehe Abbildung 28). Da `main`-Methoden statisch sind und es für statische Methoden nicht die

Möglichkeit des Überschreibens gibt, muss jedes auf JHotDraw aufbauende Programm eine eigene `Main`-Methode besitzen und in dieser festlegen, welche Teamklasse verwendet werden soll. Dadurch ist bei diesem Verfahren die Möglichkeit, eine speziellere Team-Klasse zu verwenden und damit die gewünschte Flexibilität im Umgang mit Teams bereits implizit vorhanden.

Erweiterung des zweiten Team-Handhabungs-Verfahrens um Template- und Factory-Methoden

Bei dem im Abschnitt „Zweites Team-Handhabungs-Verfahren“ auf Seite 68 vorgestellten Ansatz erfolgt die Erzeugung und Aktivierung der benötigten Team-Instanzen innerhalb der Methode `DrawApplication.open` (siehe Abbildung 30).
Wollte ein auf JHotDraw basierendes Programm eigene von den originalen Team-Klassen abgeleitete Teams verwenden, müsste es, obwohl dies nur einen kleinen Teil der Funktionalität der Methode ausmacht, die gesamte Methode `open` überschreiben. Dieses Vorgehen wäre nicht sinnvoll, weil dadurch bereits definiertes Verhalten verloren ginge bzw. in der Methode der Subklasse noch einmal implementiert werden müsste.
Um den Erfordernissen, die sich aus der Framework-Eigenschaft von JHotDraw ergeben, gerecht zu werden, muss das zweite Team-Handhabungs-Verfahren erweitert werden. Die Anweisungen, die das `StorageTeam`, insbesondere dessen Instanziierung, betreffen, müssen in eigene Untermethoden (`create/handleStorageTeam`) ausgelagert werden. Diese Maßnahme ermöglicht es Subklassen von `DrawApplication`, ganz gezielt die Erzeugung eines spezielleren Teams zu veranlassen (siehe Abbildung 31), ohne anderes vordefiniertes Verhalten zu zerstören oder selbst noch einmal implementieren zu müssen.

Erweiterung des ersten Team-Handhabungs-Verfahrens

```
public class DrawApplication
{
    public void open()
    {    ...
         handleStorageTeam();
         ...
         createMenus();
         ...
    }

    protected void handleStorageTeam()
    {
        StorageTeam storageTeam = createStorageTeam();
        if (storageTeam != null)
        {
             storageTeam.activate(org.objectteams.Team.FROZEN);
             registerAsStorageManager(this);
        }
    }

    protected StorageTeam createStorageTeam()
    {
        return new StorageTeam();
    }
}


public team class SomeSpecialStorageTeam extends StorageTeam {...}


public class SomeSpecialDrawingEditor extends DrawApplication
{
    protected StorageTeam createStorageTeam()
    {
         return new SomeSpecialStorageTeam();
    }

    public static void main(String[] args)
    {
        (new SomeSpecialDrawingEditor()).open();
    }
}
```

***Abbildung 31:*** *erweitertes zweites Team-Handhabungs-Verfahren , Implementierungs-Elemente*

Auch wenn die Einführung von Factory-Methoden zur Team-Erzeugung und Template-Methoden zur Aktivierung beim ersten Team-Handhabungs-Verfahren nicht zwingend notwendig ist, ist sie dennoch empfehlenswert, weil sie die Verständlichkeit des Frameworks erhöht und zu seiner Dokumentation beiträgt.

Vierte Anforderung - Ausschaltbarkeit der Programm-funktion

Der Einsatz von ObjectTeams soll die Möglichkeit bieten, Software einfach und noninvasiv um Eigenschaften (Programmfunktionen) zu erweitern oder sie ihnen zu nehmen. Dieses Ziel gibt es auf zwei Ebenen, einer statischen und einer dynamischen. Auf der statischen Ebene geht es darum, mit möglichst wenig Aufwand verschiedene Versionen einer Software zu erstellen. Während es auf der dynamischen Ebene darum geht, Programmfunktionen zur Laufzeit ein- oder abschalten zu können.

Im hier diskutierten Fall der Programmfunktion "Laden und Speichern" kann das statische Ziel bedient werden. Ein auf dem Framework JHotDraw basierendes Programm kann als "Version" der Software JHotDraw angesehen werden. Diese beinhaltet die Programmfunktion "Laden und Speichern", wenn die Klasse `StorageTeam` wie bisher beschrieben instanziiert und aktiviert wird. Soll ein auf JHotDraw basierendes Programm diese Programmfunktion nicht enthalten, darf `StorageTeam` nicht aktiviert werden.

Bei Verwendung des im Abschnitt „Erstes Team-Handhabungs-Verfahren“ auf Seite 67 beschriebenen Ansatzes bedeutet das, dass die `Main`-Methode des

Programms die nicht benötigten Teams einfach nicht erzeugt.

Bei Verwendung des im Abschnitt „Zweites Team-Handhabungs-Verfahren“ auf Seite 68 vorgestellten Ansatzes mit der auf Seite 71 dargestellten Erweiterung ist das Vorgehen etwas anders. Die `DrawApplication`-Subklasse des Programms muss die Team-Erzeugungsmethode (`createStorageTeam`) so überschreiben, dass sie keine Team-Instanz sondern `null` zurückliefert. Damit dieser Rückgabewert nicht zum Programmabsturz führt, muss die aufrufende Methode (`handleStorageTeam`) diesen Fall berücksichtigen und darf dann keine Team-Aktivierung versuchen (siehe Abbildung 31).

Mit dem dynamischen Ein- und Ausschalten der Programmfunktion zur Laufzeit verhält es sich etwas schwieriger. Enthält das Dateimenü erst einmal die Auswahloptionen, die den Zugriff auf die Funktionalität des Teams ermöglicht, kann der Benutzer die Teamaktivität auch anstoßen. Es müsste also möglich sein, entweder diese Menüoptionen während des Programmlaufs wieder zu entfernen oder sie zumindest auf „nicht auswählbar“ (disabled) zu schalten. Ob und wie dies erreicht werden kann, wurde im Rahmen der vorliegenden Arbeit nicht untersucht und bleibt eine offen Frage für weitere Forschungsaktivitäten.

## 5.3 Konsequenzen der Team-Handhabungs-Verfahren

Der Entwickler eines auf JHotDraw basierenden Programms kann nicht selbst entscheiden, welches Team-Handhabungs-Verfahren verwendet werden soll, weil dieses strukturell im Framework verankert und daher durch dieses vorgegeben ist. Diese Design-Entscheidung muss also im Rahmen der Umstrukturierung des Frameworks getroffen werden. Um so wichtiger ist es, dass dabei mögliche Konsequenzen berücksichtigt werden. Daher beschäftigen sich die folgenden beiden Abschnitte mit den positiven und negativen Auswirkungen beider Team-Handhabungs-Verfahren.

### 5.3.1 Funktionale Einschränkung durch erstes Team-Handhabungs-Verfahren

Verwendung spezieller StorageTeams

Ein auf JHotDraw basierendes Programm verwendet, wie im Abschnitt „Dritte Anforderung“ auf Seite 70 bereits erläutert, unter Umständen nicht die vom Framework zur Verfügung gestellte Team-Klasse `StorageTeam`, sondern eine eigene, speziellere Team-Klasse. Zum Beispiel könnte bei einem UML-Editor-Programm die `DrawApplication`-Subklasse `UMLEditor` in ihrer Team-Erzeugungsmethode `createStorageTeam` (bzw. in ihrer `main`-Methode) die `StorageTeam`-Subklasse `UMLStorageTeam` erzeugen. Wohingegen bei einem Editor für Grundrisszeichnungen die `DrawApplication`-Subklasse `GroundPlanEditor` die `StorageTeam`-Subklasse `GroundPlanStorage-Team` erzeugen würde (siehe Abbildung 32).

Mögliche Wechselwirkungen mit Programmfunktion „Öffnen von Editorfenstern“

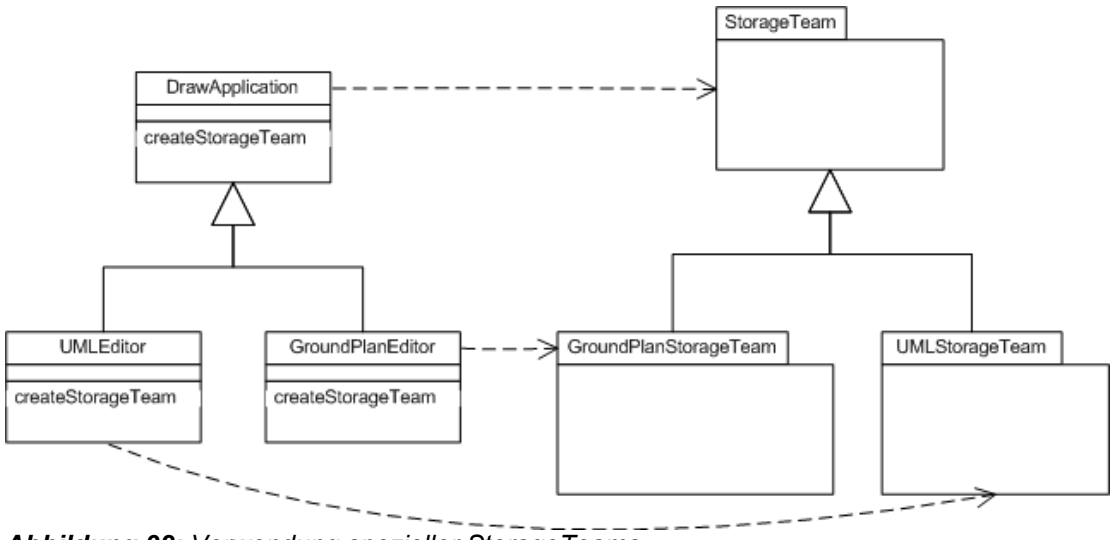


***Abbildung 32:*** *Verwendung spezieller StorageTeams*

Ist bei einem auf JHotDraw basierenden Programm eine derartige Anpassung der zu verwendenden Teamklasse erforderlich, kann es bei Verwendung des im Abschnitt „Erstes Team-Handhabungs-Verfahren“ auf Seite 67 beschriebenen Verfahrens in Wechselwirkung mit der Implementierung der Programmfunktion „Öffnen eines neuen Editorfensters“ (siehe Seite 62 zu fehlerhaftem Programmverhalten kommen.

„Erwartungskonforme“ Implementierung der Programmfunktion „Öffnen von Editorfenstern“

Die Programmfunktion „Öffnen eines neuen Editorfensters“ ermöglicht es dem Benutzer, durch Auswahl der Optionen „New“ oder „Open“ des Dateimenüs, neue Editorfenster zu öffnen. Wie bereits im Abschnitt „„Erwartungskonforme“ Implementierung“ auf Seite 63 ausführlich erläutert, werden JHotDraw-basierte Editorprogramme diese Programmfunktion typischerweise so implementieren, dass ausschließlich weitere Editorfenster des schon offenen Typs erzeugt werden. Im Beispielfall eines UML-Editors hieße das, dass alle neu geöffneten Editorfenster wieder UML-Editoren sind. Aus Implementierungssicht betrachtet, bedeutet das, dass alle im System existierenden Editorfenster-Objekte Instanzen des selben `DrawApplication`-Subtyps sind. Im besagten Beispielfall existieren also ausschließlich Instanzen des `DrawApplication`-Subtyps `UMLEditor` nebeneinander im System.

Keine Anwendbarkeitsbeschränkung für Team-Handhabungs-Verfahren

Implementiert ein JHotDraw-basiertes Programm die Programmfunktion „Öffnen eines neuen Editorfensters“ in der eben beschriebenen Art und Weise, sind beide auf Seite 67 dargestellten Verfahren zur Team-Handhabung anwendbar. Insbesondere führt das im Abschnitt „Erstes Team-Handhabungs-Verfahren“ beschriebene Verfahren in diesem Fall nicht zu fehlerhaftem Programmverhalten. Dabei spielt es keine Rolle, ob das Programm die vom Framework zur Verfügung gestellte oder eine eigene speziellere `StorageTeam`-Klasse verwendet. Die bei diesem Verfahren zu Programmbeginn erzeugte einzige Instanz des Typs StorageTeam ist für alle später im System vorhandenen Editorfenster passend, weil diese alle vom selben dynamischen Typ sind.
Im Beispiel des UML-Editor-Programms würde eine Instanz der `StorageTeam`-Subklasse `UMLStorageTeam` erzeugt werden. Dass diese ganz speziell auf den `DrawApplication`-Subtyp `UMLEditor` abgestimmt ist, ist unproblematisch, weil keine Objekte anderer `DrawApplication`-Subtypen im System vorhanden sind. Es werden ausschließlich Objekte der Klasse `UMLEditor` erzeugt und für diese ist die `UMLStorageTeam`-Instanz passend.

„Untypische“ Implementierung der Programmfunktion „Öffnen eines Editorfensters“

Das Framework JHotDraw bietet in Bezug auf die Programmfunktion „Öffnen eines neuen Editorfensters" Möglichkeiten, die über den eben betrachteten "Normalfall" hinausgehen (siehe Abschnitt „„Untypische“ Implementierung“ auf Seite 64). Ein auf JHotDraw aufbauendes Programm kann diese Funktion so modifizieren, dass verschiedene Editorfenster, das heißt Instanzen verschiedener `DrawApplication`-Subklassen erzeugt werden und sich dementsprechend gleichzeitig im System befinden.

Fehlerquelle

Falls diese `DrawApplication`-Subklassen verschiedene `StorageTeam`-Subklassen benötigen, um korrekt zu funktionieren, oder falls in der einen Editorart die Team-basierte Programmfunktion enthalten sein soll und in einer anderen nicht, erweist sich das erste Team-Handhabungs-Verfahren als ungeeignet. Die bei diesem Verfahren zu Beginn des Programmlaufs erzeugte, einzige Team-Instanz ist dann nicht für alle im System befindlichen Editorarten passend und kann deren Bedürfnisse unter Umständen nicht korrekt erfüllen. Daher kann die Verwendung dieses Team-Handhabungs-Verfahrens in solchen Fällen zu Programmfehlern führen.

Konsequenz

Fällt bei der Umstrukturierung des Frameworks die Entscheidung auf dieses Team-Handhabungs-Verfahren, wird dadurch eine potentielle Fehlerquelle eingebaut bzw. die Verwendungsmöglichkeiten des Frameworks gegenüber der objektorientierten Originalimplementierung eingeschränkt. Da dieser Umstand für den Framework-Benutzer nicht offensichtlich erkennbar ist, muss er in der Framework-Dokumentation Erwähnung finden.

Alternative

Eine Alternative, die die Verwendungsmöglichkeiten des Frameworks nicht einschränkt, bietet das im Abschnitt „Zweites Team-Handhabungs-Verfahren“ auf Seite 68 dargestellte Verfahren. Dieses kann auch dann die auftretenden Anforderungen erfüllen, wenn Instanzen verschiedener `DrawApplication`-Subklassen gleichzeitig nebeneinander existieren, weil bei diesem Verfahren jedes Objekt des Typs `DrawApplication` seine eigene, zu ihm passende Team-Instanz selbst erzeugt.

### 5.3.2 Erweiterung der Team-internen Strukturierungsmöglichkeiten durch zweites Team-Handhabungs-Verfahren

Wie bereits im Abschnitt „Die zentrale Bedeutung der Klasse DrawApplication“ auf Seite 61 erläutert wurde, gehört es in der objektorientierten Originalimplementierung des Frameworks JHotDraw zu den Aufgabenbereichen der Klasse `DrawApplication`, ein Objekt der Klasse `StorageFormatManager` zu erzeugen und in Form eines Attributs zu halten. Dabei handelt es sich um eine 1-1-Beziehung ( siehe Abbildung 33). Ferner ist `DrawApplication` die einzige Klasse, die `StorageFormatManager` erzeugt und in Form eines Attributs referenziert.

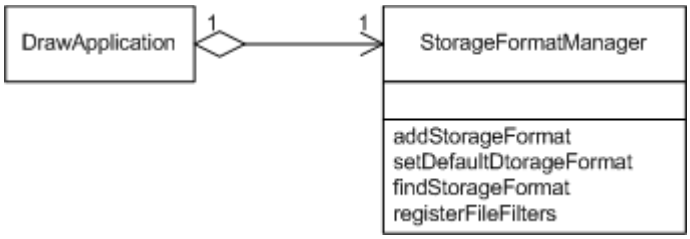


***Abbildung 33:*** *Beziehung zwischen den Klassen DrawApplication und StorageFormatManager, objektorientiert*

Im Rahmen der ObjectTeams-orientierten Umstrukturierung des Frameworks wurde die Struktur so verändert, dass die eben beschriebenen Aufgaben und die resultierende Beziehung in die Team-Klasse `StorageTeam` ausgelagert wurde (siehe Abbildung 34). Der aus diesem Umstand resultierende indirekte Charakter der Beziehung zwischen `DrawApplication` und `StorageFormatManager` verändert ihre Kardinalität nicht, da die Kardinalität der Beziehung zwischen einer Rollenklasse und ihrer Basisklasse immer 1 zu 1 ist. Das heißt, auch in der ObjectTeams-orientierten Struktur besteht zwischen `DrawApplication` und `StorageFormatManager` eine (indirekte) 1-1-Beziehung.

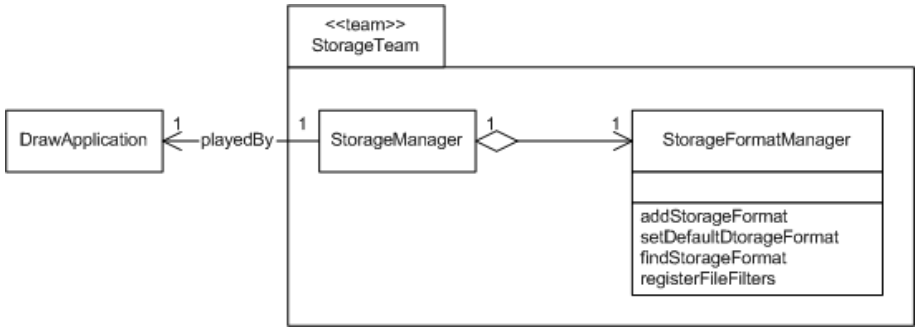


***Abbildung 34:*** *Beziehung zwischen den Klassen DrawApplication und StorageFormatManager, ObjectTeams-orientiert*

Wenn nun die in der Klasse `StorageTeam` gekapselte Funktionalität mit Hilfe des im Abschnitt „Zweites Team-Handhabungs-Verfahren“ auf Seite 68 vorgestellten Verfahrens integriert wird, besteht auch zwischen der Klasse `DrawApplication` und der Team-Klasse `StorageTeam` eine 1-1-Beziehung. Die Tatsache, dass sich in diesem Fall in jeder erzeugten `StorageTeam`-Instanz nur ein einziges `StorageManager`-Rollenobjekt und damit auch nur ein einziges `StorageFormatManager`-Objekt befindet, eröffnet eine alternative Möglichkeit zur internen Strukturierung des `StorageTeams`.
Anstelle die Funktionalität der Klasse `StorageFormatManager` in Form einer eigenständigen ungebundenen Rollenklasse zu kapseln, kann die Team-Klasse die entsprechenden Aufgaben selbst übernehmen (siehe Abbildung 35). Der Zugriff der Rolle `StorageManager` auf diese Funktionalität ist gesichert, weil erstens zu einer Rolleninstanz (in diesem Fall der Rollenklasse `StorageManager`) immer eine diese umschließende Team-Instanz (hier der Klasse `StorageTeam`) existiert und zweitens die Rolleninstanz auf diese

zugreifen kann (über `StorageTeam.this`).

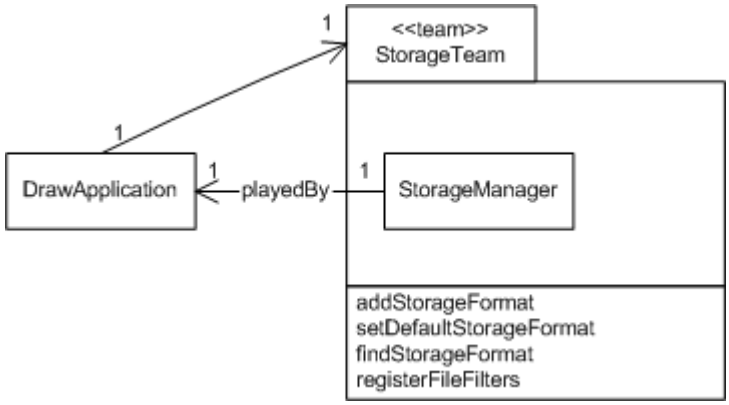


***Abbildung 35:*** *zweites Team-Handhabungsverfahren: Beziehung zwischen DrawApplication und StorageTeam und alternative StorageTeam-Struktur*

Diese interne Struktur der Klasse `StorageTeam` hilft durch die Möglichkeit der Nutzung von "declared lifting" bei Teamlevel-Methoden, explizite Lift-Operationen innerhalb von Rollenklassen zu vermeiden (siehe Abschnitt „Deklariertes versus explizites Lifting“ auf Seite 37).

Die bei der in Kapitel 6 noch zu beschreibenden Restrukturierung der Undo-Programmfunktion entstandene Klasse `UndoTeam` gleicht der Klasse `StorageTeam` in den in diesem Abschnitt beschriebenen Eigenschaften. Auch in ihrem Fall wäre die hier vorgestellte alternative Strukturierung in Bezug auf die Vermeidung expliziter Lift-Operationen und zusätzlich zur Vereinfachung des Zugriffs von Rollenobjekten auf häufig benötigte Operationen von Vorteil gewesen. Diese interne Struktur war für das `UndoTeam` aber nicht möglich. Die Gründe hierfür beschreibt Abschnitt „Schwierigkeiten bei der Registrierung von Handle-Objekten“ ab Seite 94.

# 6 ObjectTeams-basierte Restrukturierung anhand des Beispiels der Programmfunktion „Rückgängig machen von Benutzeraktionen“ (Undo)

Das folgende Kapitel beschreibt die Implementierung der Programmfunktions „Rückgängigmachen einer Benutzeraktion“ des Frameworks JHotDraw und deren Restrukturierung mittels ObjectTeams, in der Art und Weise wie dies in Kapitel  auch schon für die Lade-Speicher-Programmfunktion erfolgt ist. Es gliedert sich in folgende Teile:

## Inhaltsverzeichnis

6.1 Benutzersicht..... 78
6.2 Begriffdefinition..... 78
6.3 Objektorientierte Originalimplementierung..... 79
6.4 Diskussion der objektorientierten Implementierung..... 83
6.5 Ansatzpunkte für die ObjectTeams-orientierte Umstrukturierung..... 84
6.6 ObjectTeams-orientierte Implementierung..... 84
6.7 Verwendung der Framework-Implementierung bei der Erstellung einer Applikation..... 106
6.8 Fazit..... 110
6.9 Ausblick..... 113

## 6.1 Benutzersicht

Das Edit-Menü einer auf JHotDraw basierenden Applikation enthält die Einträge "Undo Command" und "Redo Command". Mit dem Undo-Kommando kann eine zuvor vom Benutzer ausgeführte Aktion, wie zum Beispiel die Erzeugung oder Verschiebung einer Figur, rückgängig gemacht werden. Es ist möglich, mehrere nacheinander ausgeführte Aktionen (in der zu ihrer Ausführung inversen Reihenfolge) rückgängig zu machen. Das Redo-Kommando dient dazu, nach der Ausführung des Undo-Kommandos den Originalzustand wieder herzustellen.

## 6.2 Begriffdefinition

Im folgenden werden oft die Begriffe *Benutzeraktion*, *Benutzeraktionsklasse* und *Benutzeraktionsobjekt* verwendet.
Mit *Benutzeraktion* sind dabei alle Aktionen gemeint, die ein Benutzer eines auf JHotDraw basierenden Graphikprogramms (zum Beispiel durch Auswahl einer Menüoption) ausführen kann, wie zum Beispiel das Erzeugen einer Figur (siehe 6.1). Mit *Benutzeraktionsklasse* ist die Klasse gemeint, die die jeweilige Benutzeraktion implementiert. Das ist für das Erzeugen einer Figur zum Beispiel die Klasse `CreationTool`. *Benutzeraktionsobjekt* bezeichnet eine Instanz einer solchen Benutzeraktionsklasse.

## 6.3 Objektorientierte Originalimplementierung

Der UndoManager

Ein Objekt der Klasse `DrawApplication` hält ein Objekt der Klasse `UndoManager`, welches in der Methode `open` erzeugt wird und über die Methode `getUndoManager` von außen zugegriffen werden kann. Der `UndoManager` dient der Verwaltung der ausgeführten Benutzeraktionen. Er besitzt zwei Stacks, den Undo- und den Redo-Stack. Mit den Methoden `pushUndo/Redo` und `popUndo/Redo` kann ein Element auf dem jeweiligen Stack abgelegt bzw. das oberste Element entnommen werden. `ClearUndos/Redos` löscht alle Elemente vom entsprechenden Stack (siehe Abbildung 36).

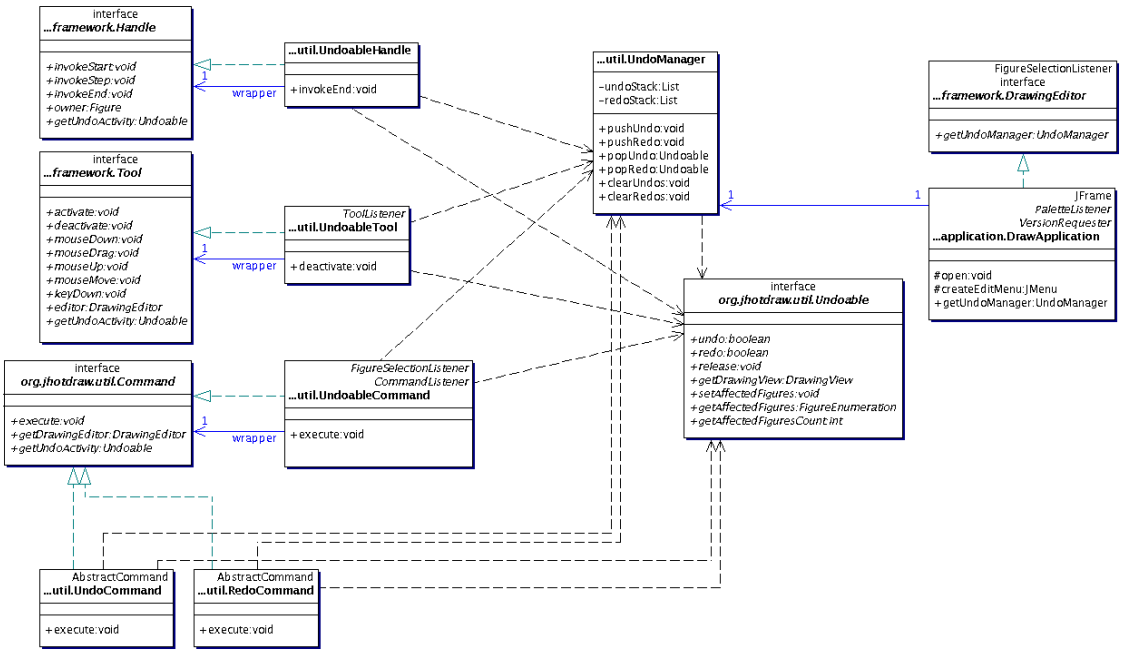


***Abbildung 36:*** *Undo - objektorientierte Struktur*

Der Typ Undoable

Bei den vom `UndoManager` verwalteten Elementen handelt es sich um Objekte des Typs `Undoable`. `Undoable` ist ein Interface. In der Methode `undo` implementieren die speziellen `Undoable`-Typen die Operation, die notwendig ist, um eine bestimmte Benutzeraktion rückgängig zu machen. Die Methode `redo` enthält die Implementierung der dazu inversen Operation.
Eine Benutzeraktion im Sinne der Framework-Implementierung der Undo-Programmfunktion bezieht sich auf einen bestimmten `DrawingView` und eine bestimmte Menge von `Figure`-Objekten. Zumeist handelt es sich dabei um die Figuren, die zum Zeitpunkt der Ausführung der Benutzeraktion im aktiven `DrawingView` selektiert sind. Damit ein `Undoable`-Objekt eine ganz konkrete Benutzeraktion rückgängig machen kann, muss es daher sowohl den `DrawingView` als auch die betroffenen Figuren speichern. Zu diesem Zweck wird ihm der `DrawingView` bei seiner Erzeugung und die Figuren danach mit der Methode `setAffectedFigures` übergeben.

Undoable-Erzeugung und nested innerclasses

Bei jeder Ausführung einer Benutzeraktion wird ein Objekt des zu dieser Aktion passenden `Undoable`-Subtyps erzeugt und, wie eben bereits erwähnt, der betroffene `DrawingView` und die betroffenen Figuren übergeben (siehe Abbildung 37). Die Verantwortung dafür obliegt jeweils der Klasse, die die entsprechende Benutzeraktion implementiert.

Zur Bereitstellung eines geeigneten `Undoable`-Subtyps, besitzt fast jede Klasse, die eine Benutzeraktion implementiert, eine innere Klasse, die das Interface `Undoable` implementiert. Diese inneren Klassen sind statisch (nested innerclass), das heißt, sie benötigen keine Referenz auf ein Objekt der sie umgebenden Klasse und können nicht nur von dieser instanziiert werden. Diese Eigenschaft trägt der Tatsache Rechnung, dass sich zum Rückgängigmachen verschiedener Benutzeraktionen in manchen Fällen der selbe `Undoable`-Subtyp eignet. Ein Beispiel hierfür ist die `Undoable`-Implementierungsklasse `PasteCommand.UndoActivity`. Sie wird von verschiedenen Benutzeraktionsklassen verwendet, darunter `PasteCommand` (Einfügen kopierter oder ausgeschnittener Figuren), `DuplicateCommand` (Verdopplung von Figuren) und `CreationTool` (Erzeugung von Figuren).

Undoable-Weiterleitung und Decorator Design Pattern

Soll eine ganz konkrete Benutzeraktion rückgängig machbar sein, muss nicht nur ein passendes `Undoable`-Objekt erzeugt, sondern dieses auch dem `UndoManager` übergeben werden (`pushUndo`), damit es später für das Undo-Kommando erreichbar ist. Desweiteren muss der Menüeintrag "Undo Command" für den Benutzer auswählbar (enabled) sein. Zu diesem Zweck ist es gegebenenfalls erforderlich, die Aktualisierung des Status der Menüeinträge anzustoßen (`DrawingEditor.figureSelectionChanged`, siehe Abbildung 37).
Um dem Entwickler auf JHotDraw basierender Applikationen die Möglichkeit zu geben, frei darüber zu entscheiden, ob eine Benutzeraktion rückgängig machbar sein soll, wurde das Decorator Design Pattern (siehe [GHJV95], Seite 175ff.) angewandt. Das bedeutet, dass das Objekt, dass die Benutzeraktion implementiert und `Undoable`-Objekte erzeugt, diese nicht selbst an den `UndoManager` weiterleitet. Stattdessen wird diese Aufgabe von einem sogenannten Wrapper-Objekt übernommen.
Das Wrapper-Objekt umhüllt das Benutzeraktionsobjekt und bildet eine Zwischenschicht zwischen diesem und dessen Klienten. Es implementiert das gleiche Interface wie das umhüllte Objekt und besitzt daher die gleichen Methoden. Der Klient hält keine Referenz auf das Objekt der Aktionsklasse, sondern auf das Wrapper-Objekt. Methodenaufrufe, die eigentlich dem Aktionsobjekt gelten, erfolgen auf dem Wrapper-Objekt. Dieses verhält sich aus Sicht des Aufrufers wie das Aktionsobjekt selbst, ist also quasi unsichtbar, da es alle Methodenaufrufe an das in ihm eingeschlossene Aktionsobjekt weiterleitet. Es fügt lediglich Funktionalität hinzu. Seine Aufgabe besteht darin, nach Ausführung einer Benutzeraktion das neu erzeugte `Undoable`-Objekt vom eingehüllten Aktionsobjekt zu holen (`getUndoActivity`), dem `UndoManager` zu übergeben (`pushUndo`) und gegebenenfalls eine Aktualisierung der Menüeinträge anzustoßen (`DrawingEditor.figure-SelectionChanged`) (siehe Abbildung 37).
Soll eine bestimmte Benutzeraktionsart rückgängig machbar sein, wird das Objekt, dass die Benutzeraktion implementiert, mit einem Wrapper umhüllt. Soll sie nicht rückgängig machbar sein, erfolgt keine Ummantelung. Damit beispielsweise die Erzeugung rechteckiger Figuren rückgängig gemacht werden kann, erfolgt die Instanziierung der Benutzeraktionsklasse `CreationTool` folgendermaßen:

```
new UndoableTool(new CreationTool(new RectangleFigure())).
```

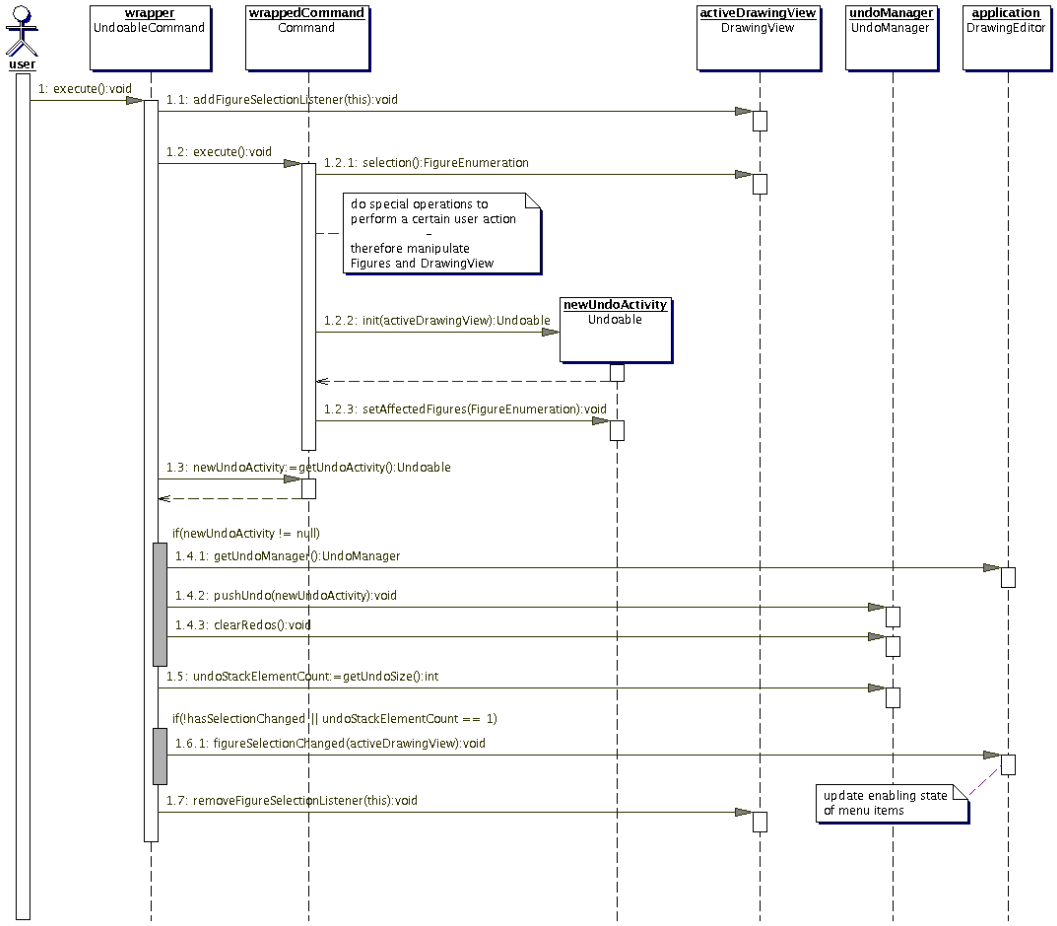


***Abbildung 37:*** *Undo - objektorientiertes Verhalten, Teil 1 - „Vorbereitung“*
*Ausführung einer Benutzeraktion mit Erzeugung eines Undoable-Objekts und Übergabe an UndoManager*

Die Typen Command, Tool, Handle und ihre Wrapper

Allen Klassen, die Benutzeraktionen im hier besprochenen Sinn implementieren, ist gemeinsam, dass sie eins der Interfaces `Command`-, `Tool`- oder `Handle` implementieren. Um den für einen Wrapper erforderlichen Zugriff auf die von ihnen erzeugten `Undoable`-Objekte zu gewährleisten, stellen alle drei Typen die Methode `getUndoActivity` zur Verfügung.
Wie bereits im Abschnitt „Undoable-Weiterleitung und Decorator Design Pattern“ auf Seite 80 erläutert, muss ein Wrapper die gleiche Schnittstelle bieten, wie das umhüllte Objekt, und daher auch das gleiche Interface implementieren. Da die Interfaces `Command`, `Tool` und `Handle` keinen gemeinsamen Supertyp besitzen, existieren drei verschiedene Wrapper-Klassen, `UndoableCommand`, `UndoableTool` und `UndoableHandle` (siehe Abbildung 36 auf Seite 79).
Bis auf die in Abbildung 36 in den Wrapper-Klassen eingetragenen Methoden (`UndoableHandle.invokeEnd`, `UndoableTool.deactivate` und `UndoableCommand.execute`) leiten alle anderen Wrapper-Methoden ihre Aufrufe lediglich an das umhüllte Objekt weiter. Die eingetragenen Methoden hingegen, holen nach dem Aufruf der entsprechenden Methode des umhüllten Objekts das erzeugte Undoable-Objekt, leiten es an den UndoManager weiter und stoßen gegebenenfalls die Menüaktualisierung an.

UndoCommand und RedoCommand

Die Klassen `UndoCommand` und `RedoCommand` sind die Klassen, die die Benutzeraktionen des Rückgängigmachens und der dazu inversen Operation,

also das Rückgängigmachen des Rückgängigmachens, implementieren. `DrawApplication` erzeugt in ihrer Menüerzeugungsmethode `createEditMenu` jeweils ein Objekt jeder dieser beiden Klassen und fügt diese dem erzeugten Menü hinzu.
Die Auswahl des Menüunterpunktes “Undo Command“ durch den Benutzer bewirkt einen Aufruf der Methode `UndoCommand.execute` (siehe Abbildung 38). `execute` entnimmt das oberste Element vom Undo-Stack des `UndoManagers` (`popUndo`) und ruft auf diesem Objekt des Typs Undoable die Methode `undo` auf. Die Methode `undo` macht, wie in Abschnitt „Der Typ Undoable“ auf Seite 79 beschrieben, die durch das Undoable-Objekt repräsentierte Benutzeraktion sozusagen rückgängig. Danach wird das `Undoable`-Objekt auf dem Redo-Stack des `UndoManagers` abgelegt (`pushRedo`). Dadurch wird es dem Benutzer ermöglicht, durch Auswahl des Menüunterpunktes „Redo Command“ auch diese Aktion des Rückgängigmachens ihrerseits selbst wieder rückgängig zu machen. Abschließend wird ein Neuzeichnen des `DrawingViews` (`checkDamage`) und die Aktualisierung der Menüs (`figureSelectionChanged`) eingeleitet. Abbildung 38 zeigt den gesamten dargestellten Vorgang. Der Aufruf des Menüeintrags „Redo Command“ funktioniert analog, mit dem Unterschied, dass das `Undoable`-Objekt vom Redo-Stack entnommen (`popRedo`), auf ihm die Methode `redo` aufgerufen und es danach auf dem Undo-Stack abgelegt wird (`pushUndo`).

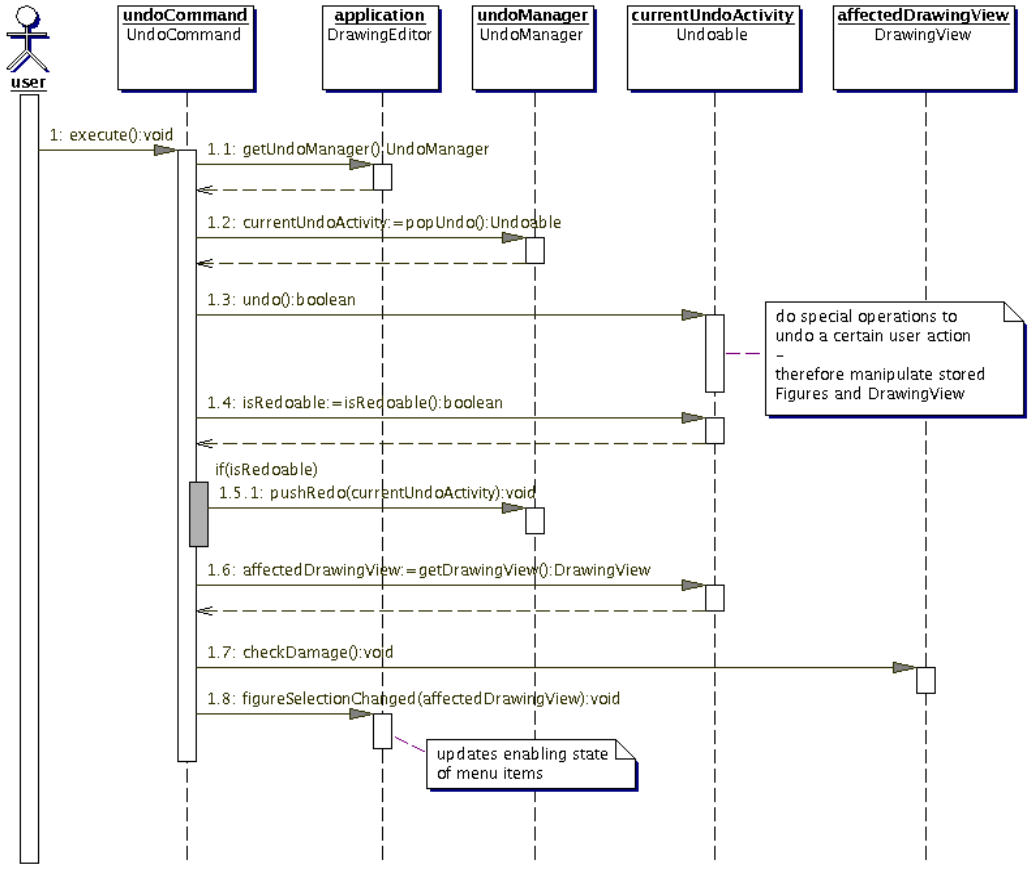


***Abbildung 38:*** *Undo - objektorientiertes Verhalten, Teil 2 - „Rückgängigmachen“*

## 6.4 Diskussion der objektorientierten Implementierung

Bereitstellung und Instanziierung der Undoable-Subtypen - code tangling und scattering

Fast jede Klasse, die eine potentiell rückgängig machbare Benutzeraktion implementiert, besitzt eine eigene innere Klasse, um einen zu ihr passenden `Undoable`-Subtyp bereitzustellen. Bei jeder Ausführung der Benutzeraktion, das heißt bei jedem Aufruf der entsprechenden Methode der äußeren Klasse, erzeugt diese ein neues Objekt des eben erwähnten eigenen inneren `Undoable`-Subtyps. Sowohl die Bereitstellung des `Undoable`-Subtyps als auch die Erzeugung von `Undoable`-Objekten sind inhaltlich der Programmfunktion "Rückgängigmachen von Benutzeraktionen (Undo)" zuzuordnen. Damit stellen sie für die Klasse, die die Benutzeraktion implementiert, eine zusätzliche Aufgabe abseits ihrer eigentlichen Bestimmung dar. Es handelt sich somit um ein Auftreten von "code tangling". Da viele Klassen aus unterschiedlichen Paketen derartige Anteile der Undo-Programmfunktion beinhalten, liegen die Implementierungsfragmente dieser Programmfunktion zudem über die Software verstreut („code scattering“). Diese beiden Eigenschaften können als Nachteile des objektorientierten Designs angesehen werden.

Innerclass-Eigenschaft der Undoable-Subtypen

Die `Undoable`-Subtypen sind als statische innere Klassen (nested class) realisiert worden. Das bedeutet, dass ihre Instanzen keine Referenz auf ein Objekt der äußeren Klasse benötigen und unabhängig von einem solchen existieren können. Es wäre also auch in der objektorientierten Framework-Version möglich gewesen, die `Undoable`-Subtypen als eigenständige Klassen zu definieren. Dadurch hätten sie in einem eigenen Paket zusammengefaßt werden können, was zur Reduzierung des "scatterings" beigetragen hätte. Dies ist die Bewertung aus einem eher technischem Blickwinkel.
Da, bis auf wenige Ausnahmen, jede `Undoable`-Implementierungsklasse mit nur einer einzigen ganz bestimmten Benutzeraktionsklasse logisch sehr stark verknüpt ist und auch nur von Objekten dieser Klasse instanziiert wird, der jeweilige `Undoable`-Subtyp also seine Daseinsberechtigung erst aus der Existenz der entsprechenden Benutzeraktionsklasse zieht, erscheint die Entscheidung, die `Undoable`-Subtypen als innere Klassen zu realisieren von einer logisch inhaltlichen Perspektive aus betrachtet dagegen sinnvoll. Diese Vorgehensweise birgt zudem den Vorteil getrennter Namensräume. Fast alle Klassen der objektorientierten Originalversion des Frameworks, die das Interface `Undoable` implementieren, haben den selben Namen: `UndoActivity`. Anhand ihres qualifizierten Namens, der den Namen der äußeren (Benutzeraktions-)Klasse enthält, können sie unterschieden werden. Beispiele hierfür sind `CreationTool.UndoActivity` (Erzeugung von Figuren), `PasteCommand.UndoActivity` (Einfügen nach Ausschneiden oder Kopieren) oder `ResizeHandle.UndoActivity` (Manipulation der Figurgröße).

Wrapper und überflüssige Instanziierung der Undoable-Subtypen

Die Benutzeraktionsklassen als Erzeuger der `Undoable`-Objekte kümmern sich weder darum, diese dem `UndoManager` zu übergeben (`pushUndo`), noch darum, die Applikation darüber zu informieren, wenn der Zustand der Menü-Einträge "Undo Command" und "Redo Command" bezüglich ihrer Ausführbarkeit (enabled/disabled) aktualisiert werden muss. Stattdessen wurden diese Aufgaben auf Wrapper-Klassen übertragen. Dieses Vorgehen dient der Erhöhung der Flexibilität des Frameworks. Der Framework-Benutzer kann selbst entscheiden, ob er im konkreten Anwendungsfall dem Endprogrammbenutzer die Möglichkeit geben will, eine bestimmte Aktion

rückgängig zu machen. Soll die Benutzeraktion rückgängig machbar sein, wird das Objekt, dass die Benutzeraktion implementiert, mit einem Wrapper umhüllt. Soll sie nicht rückgängig machbar sein, erfolgt keine Ummantelung.
In beiden Fällen wird bei jeder Ausführung der Benuzteraktion ein `Undoable`-Objekt erzeugt. Dieses kommt aber nur im ersten Fall beim `UndoManager` an und wird dadurch für `UndoCommand` und `RedoCommand` erreichbar. Die Tatsache, dass `Undoable`-Instanzen auf den Verdacht hin erzeugt werden, sie würden später eventuell benötigt, kann als weiterer Nachteil der originalen objektorientierten Implementierung angesehen werden. Ressourcen werden unter Umständen unnötig verbraucht, was die Performanz der Applikation beeinträchtigen könnte. Im Hinblick auf die große Speicherkapazität und Geschwindigkeit der aktuellen Rechnergeneration ist dieser Punkt allerdings vernachlässigbar.

## 6.5 Ansatzpunkte für die ObjectTeams-orientierte Umstrukturierung

Es gibt mehrere Ansatzpunkte für den Versuch, die Software-Struktur mit Hilfe von ObjectTeams zu verbessern.
Ein erster Ansatzpunkt besteht darin, dass das Decorator Design Pattern mit Hilfe von ObjectTeams-Konzepten nachgebildet werden kann. Die Funktionen, die in einer rein objektorientierten Implementierung mit Hilfe von Wrappern realisiert werden, können in einer ObjectTeams-orientierten Implementierung mit Hilfe des Konzepts der gebundenen Rolle mit callin-gebundenen Methoden realisiert werden. Daher können die Wrapper-Klassen der Original-Implementierung des Frameworks JHotDraw, die die Übergabe von `Undoable`-Objekten an den `UndoManager` realisieren, zu Rollenklassen umgeformt werden.
Da sowohl die Klassen `UndoManager`, `UndoCommand`, `RedoCommand`, als auch das Interface `Undoable` und deren Implementierungsklassen ausschließlich der Implementierung der Undo-Programmfunktion dienen, und auch die `Undoable`-Subtypen, wie oben erläutert, technisch gesehen unabhängig sind, können all diese Klassen vollständig in ein Team verschoben werden.
Um die Programmfunktion des "Rückgängigmachens von Benutzeraktionen" vollständig zu kapseln, muss neben der Bereitstellung von `Undoable`-Subtypen und der Handhabung der `Undoable`-Objekte von Übergabe an `UndoManager` bis Benutzung durch `UndoCommand` und `RedoCommand` auch die Erzeugung der `Undoable`-Objekte ins Team verschoben werden. Im Gegensatz zu allen bisher genannten Klassen enthalten die Benutzeraktionsklassen, welche die `Undoable`-Objekte erzeugen, auch Anteile anderer als der Undo-Programmfunktion. Daher macht es keinen Sinn, sie vollständig ins Team zu verschieben. Stattdessen müssen sie in einen Rollenteil, der die Implementierungsfragmente der Undo-Funktion enthält und innerhalb des Teams definiert wird, und einen Basisteil aufgeteilt werden.

## 6.6 ObjectTeams-orientierte Implementierung

Die Programmfunktion "Rückgängigmachen von Benutzeraktionen (Undo)" soll möglichst vollständig in einem Team gekapselt werden. Hierfür ist es erforderlich, alle Implementierungselemente, die ausschließlich dieser Programmfunktion zugeordnet werden können, in dieses Team zu verschieben. Das vorliegende Kapitel beschreibt die erforderlichen Umstrukturierungen des Frameworks JHotDraw schrittweise, begründet auf diesem Weg getroffene

Design-Entscheidungen, erläutert Probleme und Alternativen und macht Vorschläge für die Erweiterung des Umfangs des ObjectTeams-Sprachkonzepts.

DrawApplication und ihre Rolle DrawAppRole

In der objektorientierten Originalimplementierung übernimmt die Klasse `DrawApplication` die folgenden Aufgaben, die der Undo-Programmfunktion zugeordnet werden können. Sie erzeugt die Menüeinträge „Undo Command“ und „Redo Command“ innerhalb des Edit-Menüs, erzeugt ein Objekt des Typs `UndoManager`, welches sie als Attribut hält und ermöglicht über die öffentliche Methode `getUndoManager` Zugriff auf dieses Attribut. Da die Klasse `DrawApplication` auch Implementierungselemente anderer Programmfunktionen beinhaltet, kann sie nicht als ganzes ins `UndoTeam` verschoben werden. Stattdessen erhält sie im Team eine an sie gebundene Rollenklasse `DrawAppRole` (siehe Abbildung 39 auf Seite 87). Die Implementierungsfragmente der Undo-Programmfunktion werden aus ihr herausgelöst und in diese Rolle verschoben. Abhängig von Ort und Art der originalen Implementierungsfragmente und bestimmten logischen Gesichtspunkten entstehen dabei innerhalb der Rollenklasse verschiedene Strukturen.

Menüerweiterung und „implicit externalized roles“

In der objektorientierten Framework-Version erzeugt die Methode `DrawApplication.createEditMenu` das gesamte Edit-Menü. In der ObjectTeams-orientierten Implementierung sollen die Menüeinträge des Edit-Menüs, die zur Undo-Programmfunktion gehören, im Team, also von der an `DrawApplication` gebundenen Rolle `DrawAppRole` erzeugt werden. Zu diesem Zweck wird die Erzeugung dieser Menüeinträge in die neue Rollenmethode `extendMenu` verschoben. Damit die Rollenmethode das Menü erweitern kann, muss sie per replace-callin an die Basismethode gebunden sein. Die Rollenmethode führt dann durch einen `base`-call die Basismethode aus. Danach erweitert sie das von der Basismethode gelieferte Menü um die gewünschten Unterpunkte (Instanzen von `UndoCommand` und `RedoCommand`) und gibt es schließlich als eigenes Ergebnis zurück. Diese Form der Menüerweiterung tritt auch bei der in Kapitel 4 beschriebenen Kapselung der Speicher-Programmfunktion auf.
Besonders interessant im Fall des `UndoTeams` ist, das Objekte ungebundener Rollenklassen (`UndoCommand` und `RedoCommand`), die ein Interface der Basissoftware (`Command`) implementieren, als Teile des Menüs aus dem Team herausgereicht werden. Es handelt sich also sozusagen um "implicit externalized roles".

Der UndoManager als Rollenattribut

In der Originalversion des Frameworks ist `fUndoManager` ein Attribut von `DrawApplication` und kann über eine öffentliche `get`-Methode von außen zugegriffen werden. In der ObjectTeams-orientierten Implementierung soll es ein Attribut der Rolle sein. Daher werden Attribut und `get`-Methode in die Rolle verschoben. Die Erzeugung des `UndoManager`-Objekts und seine Zuweisung zum Attribut erfolgt im Original neben vielen anderen Anweisungen innerhalb der Methode `DrawApplication.open`. Die folgenden beiden Abschnitte zeigen zwei verschiedene Möglichkeiten, diese Operationen in die Rollenklasse zu verschieben.

Callin-gebundene Initialisierung des Rollenattributs

Ein naheliegender Ansatz besteht darin, die Operationen in eine Rollenmethode zu verschieben, die per callin an die Basismethode `open` gebunden wird. Diese Vorgehensweise hat einen schwerwiegenden Nachteil. Damit eine callin-Anbindung funktionieren kann, muss grundsätzlich sichergestellt werden, dass das Team beim Aufruf der Basismethode bereits

aktiviert ist. Nun handelt es sich bei der Basismethode `DrawApplication.open` aber um eine ganz besondere, nämlich die Initialisierungsmethode der Applikation. Dieser Ansatz schränkt den Rahmen der für die Team-Aktivierung möglichen Orte also insofern ein, als dass er festlegt, dass die Aktivierung des Teams vor der Ausführung der eigentlichen Initialisierungsmethode des Programms erfolgen muss. Wie in Kapitel 5 ausführlich erläutert wurde, ist der Ort der Team-Aktivierung aber von entscheidender Bedeutung und es ist aus mehreren Gründen vorteilhaft, die Möglichkeit offen zu lassen, die Team-Aktivierung innerhalb der Methode `open` vorzunehmen zu können. Dies ist auch möglich, da ein alternativer Ansatz für die Erzeugung des `UndoManager`-Objekts und das Setzen des Attributs `fUndoManager` existiert, der die beschriebenen Nachteile nicht hat und neben der technischen auch aus einer inhaltlich logischen Perspektive heraus betrachtet geeigneter erscheint.

Ungebundene Initialisierung des Rollenattributs

Der erste Ansatz bindet die Erzeugung des `UndoManager`-Objekts und seine Zuweisung an das Attribut per callin an die Initialisierungsmethode der Basisklasse. Da es sich in der ObjectTeams-orientierten Framework-Version aber um ein Attribut der Rollenklasse handelt und jede Rollenklasse in ObjectTeams/Java eine eigene Initialisierungsmethode, `initializeRole`, enthält, die von der Laufzeitumgebung automatisch als erste Methode auf einem neuen Rollenobjekt aufgerufen wird, ist ein sinnvollerer Ansatz, die Initialisierung des Attributs von dieser Rollen-eigenen Initialisierungsmethode ausführen zu lassen. Auch dieses Vorgehen garantiert, dass das Attribut gesetzt ist, bevor eine Methode benutzend darauf zugreift. Da die Rolleninitialisierungsmethode ungebunden und daher unabhängig von der Basisklasse ist, entstehen aus diesem Ansatz außerdem keinerlei Einschränkungen, was die zur Team-Aktivierung geeigneten Kontrollflussstellen betrifft.

Das Interface DrawingEditor und das zugehörige Rollen-Interface DrawingEditorRole

`DrawApplication` implementiert das Interface `DrawingEditor`. Die Aufgabe, ein `UndoManager`-Objekt zur Verfügung zu stellen, geht auf diese Implementierungsbeziehung zurück, da das Interface in der objektorientierten Implementierung die bereits erwähnte Methode `getUndoManager` deklariert. Nachdem die Klasse `DrawApplication` die Methode `getUndoManager` nicht mehr implementiert, weil der zugehörige Aufgabenbereich in die Rolle `DrawAppRole` ausgelagert wurde, muss die Methode auch aus dem Interface `DrawingEditor` entfernt und in eine an dieses gebundene Rolle `DrawingEditorRole` verschoben werden.

Aufgrund des Framework-Charakters der Software werden Referenzen auf `DrawApplication`-Objekte immer in Form von Referenzen des Typs `DrawingEditor` gehalten und übergeben. Um den Zugriff auf das zu einer bestimmten `DrawApplication`-Instanz gehörende `UndoManager`-Objekt auch in der ObjectTeams-orientierten Implementierung zu ermöglichen, muss die Implementierungshierarchie auch innerhalb des Team-Kontextes erhalten bleiben. Die Rolle `DrawAppRole` muss also das Rollen-Interface `DrawingEditorRole` implementieren (siehe Abbildung 39).

Die ungebundenen Rollenklassen UndoCommand, RedoCommand und UndoManager

Die Klassen `UndoCommand` und `RedoCommand` implementieren die Funktionalität der gleichnamigen Einträge des Edit-Menüs. Sie sind von der Klasse `UndoManager` abhängig. Da alle drei Klassen ausschließlich der Undo-Programmfunktion dienen, können sie ins Team verschoben werden. Dort sind sie ungebundene Rollenklassen.

`UndoCommand`- und `RedoCommand`-Objekte erhalten in der objektorientierten Implementierung bei ihrer Erzeugung eine Referenz auf ein `DrawingEditor`-Objekt als Parameter übergeben, über dessen Methode `getUndoManager` sie den erforderlichen Zugriff auf ein `UndoManager`-Objekt erhalten. Da in der ObjectTeams-orientierten Implementierung die Methode `getUndoManager` nicht von `DrawingEditor` sondern von seiner Rolle `DrawingEditorRole` zur Verfügung gestellt wird, muss der erwartete Typ des Parameters entsprechend angepasst werden.
Der Sourcecode der Instanziierung kann dagegen unverändert bleiben. In der objektorientierten Implementierung werden beide `Command`-Typen ausschließlich von `DrawApplication` instanziiert. `DrawApplication` kann dabei eine `this`-Referenz übergeben, da es `DrawingEditor` implementiert. Nach der Umstrukturierung werden `Undo`- und `RedoCommand` von der an `DrawApplication` gebundenen Rolle `DrawAppRole` instanziiert. Da diese ihrerseits das Rollen-Interface `DrawingEditorRole` implementiert, welches in der umstrukturierten Implementierung der erwartete Parametertyp ist, kann auch in diesem Fall eine `this`-Referenz übergeben werden.

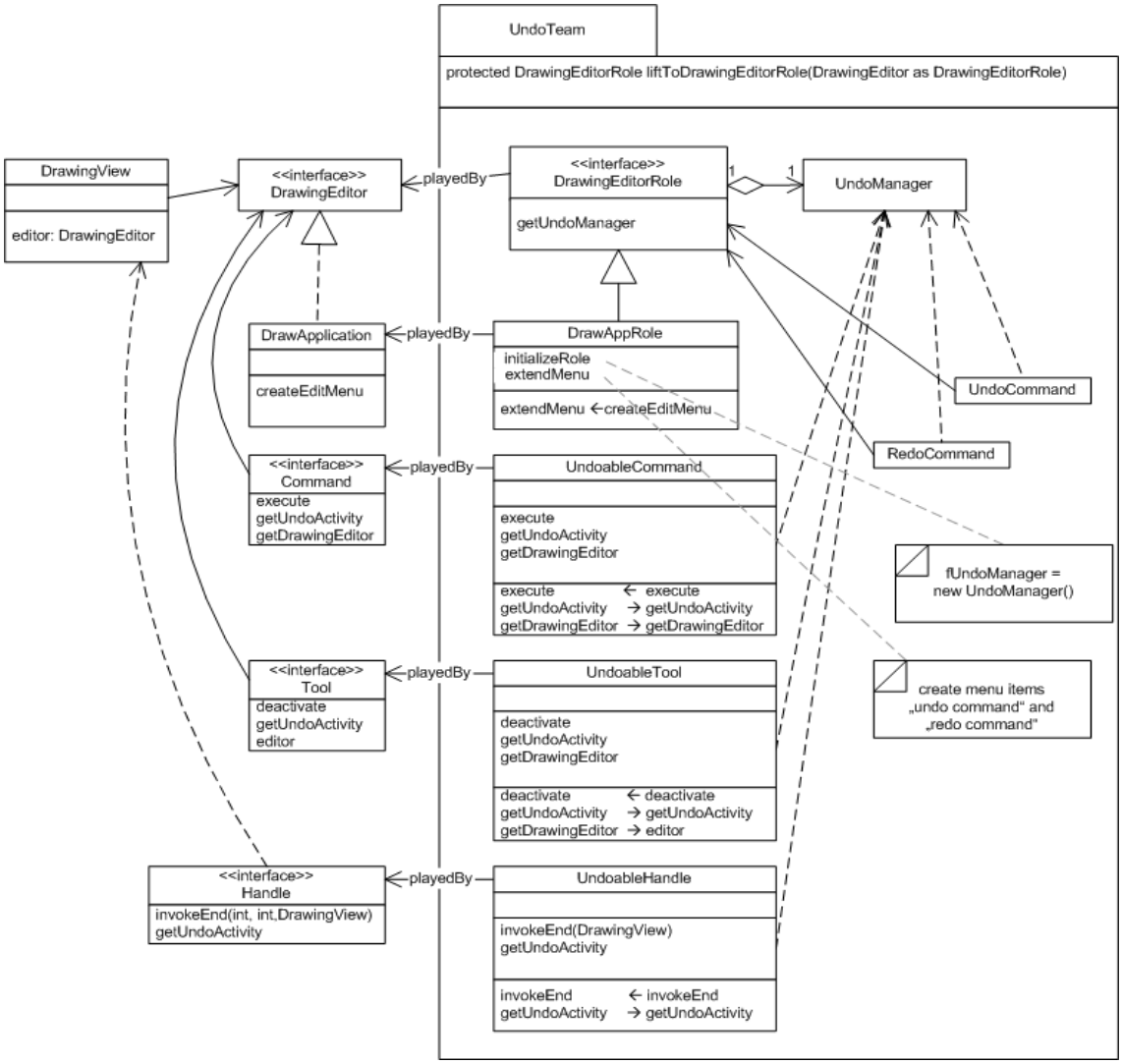


***Abbildung 39:*** *Undo - ObjectTeams-orientierte Umstrukturierung, erste Etappe*

Neben der Erzeugung der Menüeinträge und der Bereitstellung des

UndoManagers muss der gesamte den Typ Undoable betreffende Funktionskomplex aus der Basissoftware herausgelöst und ins Undo-Team extrahiert werden. Dazu gehören das Interface Undoable und alle dieses Interface implementierenden Klassen, die Erzeugung von Undoable-Objekten und die Übergabe derselben an den UndoManager.

Transfer der Undoable-Objekte zum UndoManager - Umwandlung der Wrapper-Klassen in Rollen-Klassen

Die in der objektorientierten Framework-Implementierung eingesetzten Wrapper-Objekte (Instanzen der Typen UndoableCommand, UndoableTool und UndoableHandle) erweitern das Verhalten der von ihnen umhüllten Benutzeraktionsobjekte (Instanzen der Typen Command, Tool und Handle) um den Transfer der von diesen erzeugten Undoable-Objekte zum UndoManager und die Aktualisierung des Status der Menüeinträge (siehe Abschnitt "Undoable-Weiterleitung und Decorator Design Pattern" auf Seite 80).
Obwohl sie nur die Funktionalität einzelner Methoden der umhüllten Objekte erweitern, müssen sie trotzdem alle öffentlichen Methoden besitzen, die das umhüllte Objekt hat, das heißt, das selbe Interface implementieren. Jeder Methodenaufruf auf einem Wrapper-Objekt wird von diesem an die gleichnamige Methode des umhüllten Objekts weitergeleitet. Der Klient des Benutzeraktionsobjekts hält zwar eine Referenz auf den Wrapper anstatt auf das Aktionsobjekt. Durch das eben beschriebene Vorgehen ist die durch den Wrapper gebildete Zwischenschicht aber quasi unsichtbar. Nur so kann in der objektorientierten Implementierung gewährleistet werden, dass die Erweiterung der Funktionalität des Benutzeraktionsobjekts um den Transfer der erzeugten Undoable-Objekte zum UndoManager weder für dieses noch für dessen Klienten mit zusätzlichem Aufwand verbunden ist.
Die gleiche Funktionalität, die in der objektorientierten Framework-Implementierung durch das eben beschriebene, Decorator Design Pattern ([GHJV95],Seite 175ff.) genannte Prinzip realisiert wird, kann mit ObjectTeams-Konzepten folgendermaßen erreicht werden. Der Wrapper wird durch eine gebundene Rolle ersetzt. Das umhüllte Objekt entspricht dem Basisobjekt dieser Rolle. Zu jeder Basismethode, deren Verhalten erweitert werden soll, enthält die Rolle eine callin-gebundene Rollenmethode. Je nachdem, ob das zusätzliche Verhalten vor, nach oder vor und nach der Ausführung der Basismethode erfolgen soll, handelt es sich dabei um ein before, after oder replace callin. Da ein Rollenobjekt für sein Basisobjekt und auch für dessen Klienten wirklich unsichtbar ist, das heißt von beiden nicht referenziert wird, und es das callin-Konzept von ObjectTeams ermöglicht, gezielt ganz bestimmte Methoden der Basisklasse zu erweitern, muss die Rollenklasse, anders als die Wrapper-Klasse in der objektorientierten Implementierung, nicht das selbe Interface implementieren wie seine Basisklasse.
Die Wrapper-Klassen UndoableCommand, UndoableTool und UndoableHandle können also zu gebundenen Rollenklassen umgewandelt werden. Zu diesem Zweck werden sie ins UndoTeam verschoben. Die jeweilige Wrapperklasse wird an das Interface gebunden, welches sie in der objektorientierten Version implementiert hat. Das heißt, die "implements"-Phrase wird durch "playedBy" ersetzt. Die Rolle UndoableCommand wird dadurch an das Interface Command, UndoableTool an Tool und UndoableHandle an Handle gebunden (siehe Abbildung 39 auf Seite 87 und zum Vergleich Abbildung 36 auf Seite 79). Die Methoden, die keine eigene Funktionalität implementieren, sondern nur ihren Aufruf weiterleiten, werden in der ObjectTeams-orientierten Implementierung nicht benötigt und daher entfernt. Die anderen Methoden werden per callin an die jeweils gleichnamige Methode der Basisklasse gebunden. Dabei wird bei solchen Methoden, die per

before- oder after-callin gebunden werden, die aus der objektorientierten Implementierung stammende Weiterleitung zur Methode des umhüllten Objekts entfernt. Bei replace-callin-gebundenen Methoden wird diese Weiterleitung durch einen base-call ersetzt.

Ermittlung des „richtigen“ UndoManagers

Es wurde bereits mehrfach erwähnt, dass die Hauptaufgabe der Klassen `UndoableCommand`, `UndoableTool` und `UndoableHandle` sowohl in der objektorientierten als auch in der ObjectTeams-orientierten Implementierung, im Transfer von `Undoable`-Objekten zum `UndoManager` besteht. Dabei wurde in den bisherigen Ausführungen oftmals vereinfachend die Formulierung "der" `UndoManager` verwendet, obwohl durchaus mehrere Instanzen der Klasse `UndoManager` gleichzeitig existieren können. In diesem Abschnitt soll nun näher darauf eingegangen werden, wie die Klassen, die für den Transfer verantwortlich sind, ermitteln, welches `UndoManager`-Objekt das jeweils richtige Zielobjekt ist und welche Änderungen es im Bezug auf diese Ermittlung durch die ObjectTeams-orientierte Umstrukturierung gibt.
`UndoManager`-Objekte werden in der objektorientierten Implementierung von Instanzen des Typs `DrawingEditor` als Attribut gehalten und über die öffentliche Methode `getUndoManager` zur Verfügung gestellt. Eine Instanz des Typs `DrawingEditor` entspricht einem Editorfenster. Es können während eines Programmlaufs mehrere Editorfenster geöffnet werden. Das heißt, es können mehrere `DrawingEditor`-Instanzen nebeneinander existieren. Jede Benutzeraktion, wie zum Beispiel die Erzeugung oder Verschiebung einer Figur, geschieht in einem bestimmten Editorfenster, das heißt, sie kann einer bestimmten `DrawingEditor`-Instanz zugeordnet werden. Um eine zuvor ausgeführte Benutzeraktion rückgängig zu machen, existiert die Option "Undo Command" im Edit-Menü. Dabei ist es für gewöhnlich selbstverständlich, dass nach der Auswahl dieses Eintrages im Menü eines bestimmten Editorfensters eine der Aktionen rückgängig gemacht wird, die auch in diesem Editorfenster ausgeführt wurden. Diese Anforderung wird dadurch erfüllt, dass bei jeder Ausführung einer Benutzeraktion das anlässlich dieser Aktion erzeugte `Undoable`-Objekt an den `UndoManager` des Editorfensters, das heißt der `DrawingEditor`-Instanz, übergeben wird, in dem die Aktion ausgeführt wurde.

Die zu einer bestimmten Aktion gehörende `DrawingEditor`-Instanz läßt sich folgendermaßen ermitteln. Als Benutzeraktion wird im hier diskutierten Zusammenhang die Ausführung bestimmter Methoden auf Objekten der Typen `Command`, `Tool` oder `Handle` bezeichnet, weswegen diese Objekte auch Benutzeraktionsobjekte genannt werden. Benutzeraktionsobjekte der Typen `Command` und `Tool` sind einer bestimmten `DrawingEditor`-Instanz zugeordnet, halten eine Referenz auf diese und stellen sie über die Methode `getDrawingEditor` bzw. `editor` zur Verfügung (siehe Abbildung 39). Bei einer Benutzeraktion, die sich auf ein `Command`- oder `Tool`-Objekt zurückführen lässt, kann der zugehörige `DrawingEditor` (und damit der Ziel-`UndoManager`) also über diese Methoden ermittelt werden. Im Gegensatz dazu gehören Benutzeraktionsobjekte vom Typ `Handle` nicht zu einer bestimmten `DrawingEditor`-Instanz. Stattdessen wird bei ihnen bei jedem Methodenaufruf ein Parameter des Typs `DrawingView` übergeben. Ein `DrawingView`-Objekt seinerseits kann einer bestimmten `DrawingEditor`-Instanz zugeordnet werden und stellt eine Referenz auf diese über die Methode `editor` zur Verfügung. Bei Benutzeraktionen, die sich auf `Handle`-Objekte zurückführen lassen, kann der zugehörige `DrawingEditor` also mit Hilfe des bei der Aktion übergebenen `DrawingView`-Parameters ermittelt werden.

Abbildung 39 auf Seite 87 zeigt die beschriebenen Beziehungen.

Zugriff mittels explizitem Lifting

In der objektorientierten Implementierung ist es für den Zugriff auf das `UndoManager`-Objekt ausreichend, die zu einer Benutzeraktion gehörende `DrawingEditor`-Instanz zu ermitteln, weil diese das zu ihr gehörende `UndoManager`-Objekt selbst referenziert und über die Methode `getUndoManager` den Zugriff darauf ermöglicht. In der ObjectTeams-orientierten Implementierung ist dies nicht mehr ausreichend, weil `DrawingEditor`-Instanzen ihr jeweiliges `UndoManager`-Objekt nicht selbst referenzieren. Stattdessen gibt es eine an `DrawingEditor` gebundene Rolle `DrawingEditorRole`, die diese Aufgabe übernimmt (siehe Abbildung 39). Deshalb kommt zur Ermittlung der zu einer Benutzeraktion gehörenden `DrawingEditor`-Instanz ein weiterer Schritt hinzu. Die `DrawingEditor`-Instanz muss explizit geliftet werden. Erst in Form ihrer Rolle gewährt sie Zugriff auf ihr `UndoManager`-Objekt. Aus diesem Grund benötigt die Klasse `UndoTeam` eine Methode, die `DrawingEditor`-Objekte zu `DrawingEditorRole`-Objekten liftet. Diese Methode muss nicht öffentlich sichtbar sein, weil die Objekte, die sie für den Zugriff auf `UndoManager`-Objekte brauchen (Instanzen von `Undoable-Command`, `-Tool` und `-Handle`), selbst Rollen im `UndoTeam` sind.

Die ObjectTeams-Variante des Decorator Design Patterns

Wie im Abschnitt „Undoable-Weiterleitung und Decorator Design Pattern“ auf Seite 80 erläutert, dient der Einsatz von Wrappern in der objektorientierten Implementierung der Erhöhung der Flexibilität des Frameworks. Der Wrapper implementiert einen Teil der Funktionalität der Benutzeraktionsklasse, der nicht obligatorisch ist. Ein Benutzeraktionsobjekt kann bei Bedarf dynamisch um die Wrapper-Funktionalität erweitert werden, indem es mit einem Wrapper-Objekt umhüllt wird.
Die in der objektorientierten Implementierung durch Wrapper realisierte Verhaltenserweiterung wird in der ObjectTeams-orientierten Implementierung durch den Einsatz von Rollen erzielt. Dabei ist zu beachten, dass dadurch die eben beschriebene flexible Komponente nicht verloren gehen darf. Auch in der ObjectTeams-orientierten Implementierung muss es möglich sein, gezielt festzulegen, welche Benutzeraktionsobjekte wann durch die Rollenfunktionalität erweitert werden sollen.

Auswirkungen verschiedener Team-Aktivierungs-arten

Welche Benutzeraktions-Basisobjekte um die Rollen-Funktionalität erweitert werden, ist in der ObjectTeams-orientierten Implementierung davon abhängig, auf welche Art, das heißt auf welchem Level, die `UndoTeam`-Instanz bzw. die Instanzen aktiviert werden.
Wird eine `UndoTeam`-Instanz unbeschränkt aktiviert, reagiert sie auf die Aktivitäten jeder existierenden Instanz der gebundenen Basisklassen gegebenenfalls mit callin-Aufrufen. Das heißt, jedes im System vorhandene Benutzeraktionsobjekt wird um die Rollen-Funktionalität erweitert. Ein derartiges Programmverhalten entspricht nicht dem der objektorientierten Original-Implementierung. Es kann in bestimmten Anwendungsfällen erwünscht sein, bietet aber nicht die Flexibilität der Originallösung.
Damit die zu erweiternden Basisobjekte gezielt festgelegt werden können, dürfen `UndoTeam`-Instanzen nicht uneingeschränkt aktiviert werden. Stattdessen muss die Aktivierung auf dem Level FROZEN erfolgen. Dadurch reagieren die Team-Instanzen ausschließlich auf die Aktivitäten solcher Basisobjekte mit callin-Aufrufen, die beim Team registriert wurden. Die Festlegung der Benutzeraktions-Basisobjekte, die durch Rollenfunktionalität

erweitert werden sollen, erfolgt also durch die Registrierung derselben bei einer FROZEN-aktivierten `UndoTeam`-Instanz. Diese Registrierung muss an der Stelle im Programm-Code erfolgen, an der das Objekt in der objektorientierten Implementierung mit dem Wrapper umhüllt worden wäre. Im vorliegenden Fall ist das direkt nach seiner Erzeugung.

Nebeneffekt der eingeschränkten Team-Aktivierung

Die eben beschriebene eingeschränkte Art der Team-Aktivierung wirkt sich immer auf alle Rollenklassen eines Teams aus, die callin-gebundene Methoden besitzen. Das heißt, es wird nicht unterschieden zwischen Rollenklassen, bei denen auf alle potentiellen Basisobjekte reagiert wird, und Rollenklassen, bei denen nur auf bestimmte, registrierte Basisobjekte reagiert wird. Daher hat die eingeschränkte Aktivierung der `UndoTeam`-Instanzen einen Nebeneffekt. Die im Abschnitt „DrawApplication und ihre Rolle DrawAppRole“ auf Seite 85 beschriebene Rollenklasse `DrawAppRole` würde normalerweise, das heißt in einem unbeschränkt aktivierten Team, bei jedem Aufruf ihrer callin-Methode `extendMenu` instanziiert werden. Da dieser Mechanismus durch die eingeschränkte Aktivierung nicht greift, müssen alle Instanzen ihrer Basisklasse `DrawApplication` explizit beim Team registriert werden. Diese Registrierung muss bei jedem `DrawApplication`-Objekt vor dem Aufruf seiner Methode `createEditMenu` erfolgen, weil die callin-Methode `extendMenu` an diese gebunden ist, und andernfalls kein callin-Aufruf stattfindet.

Die Anzahl existierender Team-Instanzen und die Registrierung von Basisobjekten

Nachdem im vorangegangenen Abschnitt erläutert wurde, welche besonderen Anforderungen sich aus der Nachbildung des Decorator Design Patterns in Bezug auf die zu verwendende Team-Aktivierungsart ergeben, soll nun diskutiert werden, ob mehrere `UndoTeam`-Instanzen gleichzeitig im System existieren dürfen und welche Auswirkungen die Anzahl der vorhandenen Team-Instanzen auf die Registrierung der Basisobjekte hat.
Die Klasse `UndoTeam` verfügt über gewisse Gemeinsamkeiten mit der im Rahmen der in Kapitel 4.4.1 ab Seite 32 beschriebenen Umstrukturierung der Speicher-Programmfunktion entstandenen Klasse `StorageTeam`, die im hier diskutierten Kontext von Belang sind. `UndoTeam` ähnelt `StorageTeam` insofern, als dass beide eine an die Basisklasse `DrawApplication` gebundene Rolle besitzen, die eine callin-gebundene Methode `extendMenu` zur Erweiterung eines von der Basis erzeugten Menüs enthält. Diese Menüerweiterung ist dabei für `StorageTeam` die einzige callin-getriebene Aktivität. Da sie für `UndoTeam` zwar nicht die einzige, aber die jeweils erste callin-getriebene Aktivität einer Instanz ist, kann angenommen werden, dass die für `StorageTeam` im Zusammenhang mit dieser callin-Aktivität hergeleiteten Instanziierungs-Möglichkeiten, die in Kapitel „Aspekte der Team-Erzeugung und -Aktivierung“ ab Seite 61 ausführlich erläutert wurden, auch auf `UndoTeam` angewendet werden können. Deshalb dienen sie hier als Diskussionsgrundlage.
Basierend auf den in Kapitel 5.2 für die Klasse `StorageTeam` ermittelten Handhabungsmöglichkeiten wird davon ausgegangen, dass es grundsätzlich zwei verschiedene Möglichkeiten gibt, die Anzahl der Instanzen der Klasse `UndoTeam` zu handhaben. Die eine besteht darin, dass während eines Programmlaufs systemweit nur eine einzige Instanz des Teams existiert (siehe Abschnitt „Erstes Team-Handhabungs-Verfahren„ auf Seite 67), während die andere vorsieht, genau so viele Instanzen des Teams zu halten wie Editorfenster, das heißt Instanzen von `DrawApplication` bzw. `DrawingEditor`, existieren, wobei jede Team-Instanz fest einer `DrawingEditor`-Instanz zugeordnet ist und jedes Basisobjekt, welches durch

eine Rolle in einer bestimmten Team-Instanz erweitert werden soll, bei dieser Team-Instanz registriert werden muss (siehe Abschnitt Zweites Team-Handhabungs-Verfahren auf Seite 68). Diese beiden Handhabungsverfahren werden im folgenden auf Anwendbarkeit für die Klasse `UndoTeam` überprüft. Dabei wird insbesondere berücksichtigt, ob und wie der für die Registrierung von Basisobjekten erforderliche Zugriff auf die Team-Instanzen gewährleistet werden kann.

Eine zentrale Team-Instanz

Bei Anwendung des im Abschnitt „Erstes Team-Handhabungs-Verfahren„ auf Seite 67 beschriebenen Verfahrens auf die Klasse `UndoTeam` gäbe es systemweit nur eine einzige Instanz dieser Team-Klasse. Diese müsste von einer auf JHotDraw basierenden Applikation selbst in ihrer `Main`-Methode erzeugt und aktiviert werden. Bei dieser Team-Instanz-Handhabungsvariante ist es möglich, das Team unbeschränkt zu aktivieren und dadurch die Rückgängigmachbarkeit für alle Benutzeraktionsobjekte einzuschalten, ohne dass diese explizit registriert werden müssen.

Daneben ist es auch möglich, die Team-Aktivierung zu beschränken und dem Entwickler dadurch, wie im Abschnitt "Auswirkungen verschiedener Team-Aktivierungsarten" auf Seite 90 erläutert, die Möglichkeit zu geben, selbst zu bestimmen, welche Benutzeraktionsobjekte durch Rollenfunktionalität erweitert werden sollen. In diesem Fall müssen die entsprechenden Benutzeraktionsobjekte und alle `DrawingEditor`-Instanzen direkt nach ihrer Erzeugung bei der zentralen `UndoTeam`-Instanz registriert werden.
Bezüglich dieser Registrierung stellen sich nun drei Anforderungen. Einerseits muss allen potentiellen Erzeugern von Basisobjekten direkter Zugriff auf die Team-Instanz oder eine andere Möglichkeit zur Durchführung der Registrierung gewährt werden. Andererseits ist es, wie in Kapitel 6.7 noch erläutert werden wird, im Rahmen der Framework-Eigenschaft der Implementierung erforderlich, dass eine auf JHotDraw basierende Applikation die Möglichkeit hat, anstelle einer Instanz der Klasse `UndoTeam` eine Instanz einer spezielleren Team-Klasse zu verwenden oder sogar gar keine Team-Instanz zu erzeugen. Eine einfache Möglichkeit, alle genannten Anforderungen zu erfüllen, ist, eine zusätzliche Klasse bereitzustellen, die mittels statischer Methoden das Setzen der konkreten Instanz des Typs `UndoTeam` ermöglicht und eine indirekte Möglichkeit zur Registrierung von Basisobjekten bei dieser Team-Instanz bietet (siehe Abbildung 40).

```
public class TeamDispatcher
{
    private UndoTeam _undoPerformer;

    public static void setUndoPerformer(UndoTeam undoPerformer)
    {
        _undoPerformer = undoPerformer;
    }

    public static void registerForUndo(Object baseObject)
    {
        if (_undoPerformer != null)
        {
            if ((baseObject instanceOf DrawingEditor)
               || (baseObject instanceOf Command)
               || (baseObject instanceOf Tool)
               || (baseObject instanceOf Handle))
            {
                _undoPerformer.add(baseObject);
            }
        }
    }
}
```

***Abbildung 40:*** *Undo - zentrale Team-Instanz, Registrierung von Basisobjekten*

Eine Team-Instanz pro Editorfenster

Bei der Anwendung des im Abschnitt Zweites Team-Handhabungs-Verfahren auf Seite 68 beschriebenen Verfahrens, erzeugt und aktiviert jedes `DrawingEditor`-Objekt eine eigene `UndoTeam`-Instanz, bei der es sich auch selbst registriert. Daher kann es seine Team-Instanz als Attribut halten und über eine öffentliche `get`-Methode Zugriff darauf gewähren.
Da in diesem Szenario mehrere `UndoTeam`-Instanzen im System vorhanden sein können, stellt sich die Frage, bei welcher Team-Instanz ein bestimmtes Benutzeraktionsobjekt registriert werden muss. Zur Beantwortung dieser Frage muss der aus dem Abschnitt „Ermittlung des „richtigen“ UndoManagers“ auf Seite 89 bekannte Zusammenhang zwischen Benutzeraktion und dem zu dieser gehörenden `UndoManager`-Objekt noch einmal beleuchtet werden. Wie dort ausführlich erläutert wurde, muss das anlässlich der Ausführung einer bestimmten Benutzeraktion erzeugte `Undoable`-Objekt zu dem `UndoManager`-Objekt transferiert werden, der zu dem Editorfenster, das heißt zu der `DrawingEditor`-Instanz gehört, in dem die Aktion erfolgt ist. Da dieser Transfer durch die an die Benutzeraktionsobjekte gebundenen Rollen erfolgt, in der hier diskutierten Team-Instanz-Handhabungsvariante jedes `DrawingEditor`-Objekt seine eigene `UndoTeam`-Instanz besitzt und sich das zu ihr gehörende `UndoManager`-Objekt in dieser Team-Instanz befindet, muss ein bestimmtes Benutzeraktionsobjekt bei der Team-Instanz registriert werden, die zu der `DrawingEditor`-Instanz gehört, in der die auf dem Benutzeraktionsobjekt aufgerufenen Methoden ihre Aktionen ausführen.
Für Benutzeraktionsobjekte, die Instanzen der Typen `Command` oder `Tool` sind, ist die Ermittlung der richtigen Team-Instanz und die Registrierung sehr einfach. Sie sind während ihres gesamten Lebenszyklus an ein und dieselbe `DrawingEditor`-Instanz gebunden. Daher ist die zu ihren Aktionen gehörende `DrawingEditor`-Instanz für jede Aktion die selbe und sie müssen bei deren `UndoTeam`-Instanz registriert werden. Da einem `Command`- oder `Tool`-Objekt bei seiner Erzeugung eine Referenz auf seine `DrawingEditor`-Instanz übergeben werden muss, muss der jeweilige Erzeuger eine solche besitzen und kann über diese die zugehörige `UndoTeam`-Instanz zugreifen und das neu erzeugte Benutzeraktionsobjekt bei dieser registrieren.

Bei Benutzeraktionsobjekten vom Typ `Handle` ist die Ermittlung der richtigen `UndoTeam`-Instanz nicht so einfach. Eine `Handle`-Instanz gehört nicht zu einer

Schwierigkeiten bei der Registrierung von Handle-Objekten

bestimmten `DrawingEditor`-Instanz. Stattdessen wird ihr bei jedem Methodenaufruf ein Parameter des Typs `DrawingView` übergeben. Ein `DrawingView`-Objekt seinerseits gehört zu einer bestimmten `DrawingEditor`-Instanz. Das heißt, bei einer Benutzeraktion, die sich auf ein `Handle`-Objekt zurückführen lässt, kann erst zum Zeitpunkt ihrer Ausführung entschieden werden, zu welcher `DrawingEditor`-Instanz sie gehört. Oder anders ausgedrückt, für ein `Handle`-Objekt lässt sich nicht eindeutig festlegen, bei wessen `UndoTeam`-Instanzen es registriert werden muss. Daher müsste jedes zu registrierende `Handle`-Objekt bei jeder existierenden `UndoTeam`-Instanz registriert werden.

Spracherweiterungsvorschlag

Würde es darum gehen, jedes existierende `Handle`-Objekt mit Rollenfunktionalität zu erweitern, obwohl die Team-Instanzen beschränkt aktiviert sind, müsste das ObjectTeams-Sprachkonzept um folgende Möglichkeit erweitert werden. Es müsste möglich sein, zu deklarieren, dass bestimmte Rollenklassen auch bei beschränkter Team-Aktivierung so funktionieren, wie bei unbeschränkter, also auf die Aktivität jedes potentiellen Objekts der Basisklasse gegebenenfalls mit callin-Aktivität reagieren.

Nun geht es im hier diskutierten Fall aber darum, dass jede Instanz des `UndoTeams`, unabhängig davon, wann sie erzeugt wird, auf die Aktivitäten bestimmter, vom Programmierer festgelegter Objekte des Basistyps `Handle` mit callin-Aktivität reagiert. Um dem Programmierer eine entsprechende Registrierungsmöglichkeit zu eröffnen, müsste bei Verwendung der hier besprochenen Team-Handhabungs-Variante die Klasse `UndoTeam` oder eine entsprechende Hilfsklasse statische Methoden zur Verfügung stellen, mit deren Hilfe einerseits alle erzeugten UndoTeam-Instanzen und andererseits die bei diesen zu registrierenden `Handle`-Basisobjekte registriert werden können (siehe Abbildung 41).

```
public class TeamDispatcher
{
  private static List _undoPerformers;
  private static List _handleBaseObjects;

  public static void addUndoPerformer(UndoTeam undoPerformer)
  {
      if (undoPerformer == null) return;

      _undoPerformers.add(undoPerformer);

      //register all existing baseObjects of type Handle with new UndoTeam instance
      for (Iterator iter = _handleBaseObjects.iterator(); iter.hasNext();)
      {
          undoPerformer.add((Handle) iter.next());
      }
  }

  public static void registerForUndo(Handle newBaseObject)
  {
      if (newBaseObject == null) return;

      _handleBaseObjects.add(newBaseObject);

      //register new baseObject of Type Handle with all existing UndoTeam instances
      for (Iterator iter = _undoPerformers.iterator(); iter.hasNext();)
      {
          ((UndoTeam) iter.next()).add(newBaseObject);
      }
  }
}
```

***Abbildung 41:*** *Undo - mehrere Team-Instanzen, Registrierung von Handle-Basisobjekten*

Nachdem es also möglich ist, bestimmte `Handle`-Basisobjekte bei allen `UndoTeam`-Instanzen zu registrieren, egal wann diese erzeugt werden, stellt

sich nun ein neues Problem. Um es in Erinnerung zu rufen, ein `Handle`-Basisobjekt muss bei jeder existierenden `UndoTeam`-Instanz registriert werden, weil erst beim Aufruf der callin-gebundenen Basismethode ermittelt werden kann, zu welcher `DrawingEditor`-Instanz die dadurch erfolgte Benutzeraktion gehört und damit, in welcher Team-Instanz sich die zu diesem `DrawingEditor`-Objekt gehörende Rolle und deren `UndoManager`-Objekt befindet. Eigentlich müsste also bei jedem Aufruf der callin-gebundenen Basismethode nur in einer ganz bestimmten `UndoTeam`-Instanz von der zum Basisobjekt gehörenden Rolle ein `Undoable`-Objekt erzeugt und zum dem im Team befindlichen `UndoManager` transferiert werden. Wenn aber in jeder existierenden `UndoTeam`-Instanz ein zum Basisobjekt gehörendes Rollenobjekt existiert, werden bei jedem Aufruf der callin-gebundenen Basismethode auch in jeder Team-Instanz `Undoable`-Objekte erzeugt. Dann ermittelt das jeweilige Rollenobjekt, wie im Abschnitt „Ermittlung des „richtigen“ UndoManagers“ auf Seite 89 beschrieben, die zum Basismethodenaufruf gehörende `DrawingEditor`-Instanz und liftet diese in seiner Team-Instanz, um dann dem zugehörigen `UndoManager`-Objekt das erzeugte `Undoable`-Objekt zu übergeben.
Dies erzeugt zwar kein Fehlverhalten des Programms, weil nur das `UndoManager`-Objekt der `DrawingEditor`-Instanz, deren Rolle sich von Anfang an in einer bestimmten `UndoTeam`-Instanz befindet, über die Menüeinträge der `DrawingEditor`-Instanz, also des entsprechenden Editorfensters, erreichbar ist. Es erzeugt aber je nach Anzahl der im System befindlichen `DrawingEditor`-Instanzen und damit auch `UndoTeam`-Instanzen einen erheblichen Mehraufwand. Darüber hinaus, kann die in Kapitel 5.3.2 „Erweiterung der Team-internen Strukturierungsmöglichkeiten durch zweites Team-Handhabungs-Verfahren“ auf Seite 75 für das `StorageTeam` hergeleitete Vereinfachung der internen Team-Struktur auf das `UndoTeam` nicht angewendet werden.

Spracherweiterungs-vorschlag

Um diese Nachteile bei der Anwendung des im Abschnitt Zweites Team-Handhabungs-Verfahren auf Seite 68 beschriebenen Verfahrens auf die Klasse `UndoTeam` zu vermeiden, müsste das ObjectTeams-Sprachkonzept so erweitert werden, dass es den zu den `Handle`-Basisobjekten gehörigen Rollenobjekten möglich wäre zu ermitteln, ob für ein bestimmtes `DrawingEditor`-Objekt in ihrer Team-Instanz bereits ein zugehöriges Rollenobjekt existiert oder nicht, ohne dabei ein neues Rollenobjekt zu erzeugen.

Da die Anwendung des im Abschnitt „Zweites Team-Handhabungs-Verfahren“ beschriebenen Verfahrens bei der Klasse `UndoTeam` zu der soeben erläuterten Erhöhung der Komplexität der Software führen würde und das im Abschnitt „Erstes Team-Handhabungs-Verfahren“ auf Seite 67 beschriebene Verfahren darüber hinaus die Möglichkeit bietet, bei Bedarf durch unbeschränkte Team-Aktivierung, alle Benutzeraktionsobjekte ohne Registrierung um die Rollenfunktionalitäten zu erweitern, ist das „erste“ dem „zweiten“ Verfahren für den Fall des `UndoTeams` vorzuziehen.

Erzeugung von Undoable-Objekten - Extraktion der Implementierungs-elemente ins UndoTeam

Nachdem in den vorangegangenen Abschnitten ausführlich darauf eingegangen wurde, wie der Aufgabenbereich des Transfers der `Undoable`-Objekte zum `UndoManager` aus der ursprünglichen objektorientierten Implementierung herausgelöst und ins `UndoTeam` extrahiert werden kann, soll nun der Bereich der Erzeugung der `Undoable`-Objekte näher beleuchtet

werden.

Erweiterung des bisher beschriebenen Designs

In der bisherigen Beschreibung der ObjectTeams-orientierten Umstrukturierung wurde davon ausgegangen, dass lediglich der Transfer von `Undoable`-Objekten durch Rollen übernommen wird. Die Erzeugung derselben wurde weiterhin von den Benutzeraktions-Basisklassen selbst erledigt. In diesem Szenario ist die transferierende Rollenmethode per callin an eine Basismethode gebunden, nach deren Ausführung garantiert werden kann, dass ein neues `Undoable`-Objekt existiert. Das heißt, die Rollenmethode ist an eine Basismethode gebunden, die für jede erfolgte Benutzeraktion genau einmal aufgerufen wird, und dabei entweder selbst das `Undoable`-Objekt erzeugt (wie bei `Command`-Objekten) oder die Benutzeraktion abschließt (wie bei `Handle`-Objekten). Das Rollenobjekt holt sich in seiner callin-Methode das zu transferierende `Undoable`-Objekt von seiner Basis durch Aufruf der callout-Methode `getUndoActivity`.
Wenn nun die Erzeugung der `Undoable`-Objekte aus der Basis entfernt und in eine Rolle verschoben wird, stellt sich die Frage, ob dadurch das bisher beschriebene Rollendesign (siehe Abbildung 39 auf Seite 87) hinfällig wird. Dem ist nicht so. Die `Undoable`-Objekte werden dann zwar nicht mehr von den Benutzeraktions-Basisklassen erzeugt und können daher nicht mehr einfach von diesen geholt werden. Es bleibt aber die Notwendigkeit, sie zum `UndoManager` zu transferieren. Und es bleibt auch die Anforderung, dass dabei der Zeitpunkt des Transfers gegenüber der Original-Implementierung nicht verändert werden darf, weil nur so garantiert werden kann, dass wirklich alle `Undoable`-Objekte und dabei jedes genau einmal zum `UndoManager` transferiert werden, und die Umstrukturierung des Programms keine unerwünschte Verhaltensänderung nach sich zieht. Diese beiden Anforderungen werden von den Rollen in ihrer bisher beschriebenen Form erfüllt. Auch wenn die Basis die zu transferierenden `Undoable`-Objekte nicht mehr erzeugt, muss die bisherige callin-Bindung zwischen Basismethode und Rollenmethode trotzdem erhalten bleiben, weil sie den eben erläuterten richtigen Zeitpunkt für den Transfer festlegt. Der einzige Bereich der Rollen-Implementierung, der angepasst werden muss, ist der Zugriff auf das erzeugte `Undoable`-Objekt. Dieser kann nicht mehr mittels callout auf die Basis erfolgen. Eine nahe liegende Lösung für dieses Problem ist, dass die Rollen selbst die `Undoable`-Objekte zur Verfügung stellen, indem sie deren Erzeugung übernehmen. Das bisherige Rollendesign wird also nicht hinfällig, sondern um den Aufgabenbereich der Erzeugung der Undoable-Objekte erweitert. Diese Erweiterung ist nicht trivial. Das heißt, die Erzeugung der `Undoable`-Objekte kann nicht einfach aus den Basisklassen in die existierenden Rollenklassen "verschoben" werden.

Erzeugung von Undoable-Objekten ist nicht generalisierbar

In der objektorientierten Original-Implementierung erzeugen ausschließlich Objekte der Typen `Command`, `Tool` und `Handle`, welche im hiesigen Zusammenhang auch als Benutzeraktionsobjekte bezeichnet werden, `Undoable`-Objekte. Für jedes Objekt des Typs `Handle` steht fest, dass nach jedem Aufruf seiner Methode `invokeEnd` eine Benutzeraktion abgeschlossen ist und ein neues `Undoable`-Objekt erzeugt worden sein muss. Deswegen ist es möglich, verallgemeinernd für alle Objekte des Typs `Handle` zu wissen, wann ein neues `Undoable`-Objekt zum Transfer bereitliegt. Aus diesem Grund ist es für den Transfer von `Undoable`-Objekten, die von `Handle`-Objekten erzeugt werden, ausreichend, eine Rollenklasse an das Interface `Handle` zu binden. Für die Interfaces `Command` und `Tool` und die an sie gebundenen

Rollenklassen gilt entsprechendes. Geht es nur um den Transfer der `Undoable`-Objekte, sind also drei Rollenklassen ausreichend.
Für die Erzeugung der `Undoable`-Objekte und das Setzen ihrer Attributwerte ist eine derartige Verallgemeinerung leider nicht möglich. Das liegt daran, dass für fast jede Benutzeraktionsklasse eine eigene, genau auf diese abgestimmte Implementierungsklasse des Interfaces `Undoable` existiert, und es außerdem auch von der jeweiligen Benutzeraktionsklasse abhängig ist, wann, bezogen auf die Ausführung der Benutzeraktion, die `Undoable`-Objekte erzeugt werden, mit welchen Werten sie initialisiert werden müssen und wann diese Werte zur Verfügung stehen.
Ein `Undoable`-Objekt repräsentiert, wie im Abschnitt „Der Typ Undoable“ auf Seite 79 beschrieben, eine ganz bestimmte ausgeführte Benutzeraktion. Eine solche Aktion wird in einem bestimmten `DrawingView` ausgeführt und betrifft eine bestimmte Menge von Figuren. Um die Aktion später rückgängig machen zu können, muss das `Undoable`-Objekt Referenzen auf diese speichern. Wann im Verlauf der Ausführung der Aktion diese Figuren zur Verfügung stehen, ist dabei von der Art der Benutzeraktion abhängig. Sollen zum Beispiel die in einem bestimmten `DrawingView` selektierten Figuren gelöscht werden, existieren die Figuren nur vor der Ausführung durch die Aktion. Besteht die Benutzeraktion dagegen darin, eine neue Figur zu erzeugen, steht die betroffene, für das `Undoable`-Objekt relevante Figur erst nach der Ausführung der Aktion zur Verfügung.
Um eine Benutzeraktion rückgängig zu machen, reicht es nicht aus, den betroffenen `DrawingView` und die Figuren zu kennen. Das `Undoable`-Objekt muss in der Lage sein, die Figuren in den Zustand zurückzuversetzen, den diese vor der Manipulation durch die Benutzeraktion hatten. Zu diesem Zweck muss es die ursprünglichen Werte der durch die Aktion manipulierten Figur-Attribute speichern. Um welche Attribute es sich dabei handelt, ist wiederum von der Art der Benutzeraktion abhängig. Besteht die Aktion zum Beispiel darin, die betroffenen Figuren zu verschieben, also ihre Koordinaten zu manipulieren, muss das `Undoable`-Objekt die ursprünglichen Figur-Koordinaten speichern. Besteht die Aktion darin, die Farbe der Figuren zu verändern, muss sich das `Undoable`-Objekt dagegen die ursprünglichen Farbwerte der Figuren merken.

Fast jede Benutzeraktions-klasse benötigt eine eigene Rollenklasse

Da neben der passenden `Undoable`-Implementierungsklasse der Zeitpunkt und die Art und Weise der Erzeugung und Initialisierung von `Undoable`-Objekten derartig von der Art der zugehörigen Benutzeraktion abhängig ist, kann bei der Auslagerung ihrer Implementierung ins `UndoTeam` keine Verallgemeinerung getroffen werden. Das heißt, jede Benutzeraktionsklasse, die in der objektorientierten Implementierung die Erzeugung ihrer `Undoable`-Objekte selbst implementiert, benötigt eine eigene, an sie gebundene Rollenklasse zur Erzeugung und Initialisierung der `Undoable`-Objekte.

Vererbungsbeziehung der Rollenklassen

Damit diese neuen Rollenklassen die Funktionalität der bereits existierenden, für den Transfer der `Undoable`-Objekte verantwortlichen Rollen nutzen können, müssen sie diese beerben. Dabei muss eine Rollenklasse, die direkt an eine Benutzeraktionsklasse gebunden ist, die Rollenklasse beerben, die an das Interface gebunden ist, welches die Benutzeraktions-Basisklasse implementiert. Zum Beispiel muss die Rollenklasse, die an die Benutzeraktionsklasse `CreationTool` (Erzeugung von Figuren) gebunden ist, die Rollenklasse `UndoableTool` beerben, welche an das Interface `Tool` gebunden ist, weil die Basisklasse `CreationTool` dieses Interface implementiert.

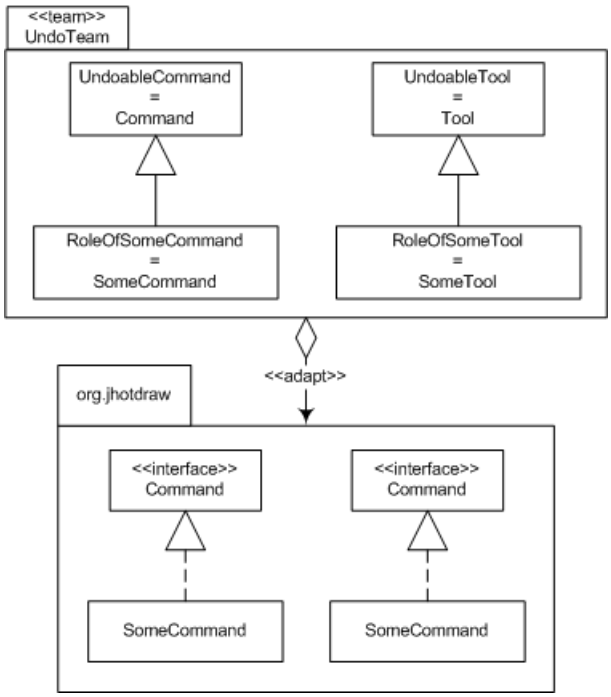


***Abbildung 42:*** *Undo - Vererbungshierarchien der gebundenen Rollenklassen*

Extraktion der Implementierungs-fragmente

Die Extraktion der Implementierungselemente, die die Erzeugung und Initialisierung von `Undoable`-Objekten betreffen, aus den Benutzeraktions-Basisklassen in die an diese gebundenen Rollenklassen ist nicht trivial. Sie ist mit hohem Aufwand verbunden und in vielen Fällen nicht vollständig möglich. Das liegt daran, dass, wie bereits erläutert, das "Rückgängigmachen einer Aktion" inhaltlich sehr starke Abhängigkeiten von der Ausführung derselben und auf der Ebene der Implementierung sehr starke Verwebungen mit den entsprechenden Implementierungselementen aufweist.

Initialisierung von Undoable-Objekten ist nicht vollständig in Team-Form kapselbar

Die die Initialisierung des `Undoable`-Objekts betreffenden Implementierungs-elemente lassen sich nur bei solchen Benutzeraktionsklassen vollständig in die Rollenklasse verschieben, bei denen die für das Rückgängigmachen relevanten Figuren bereits jeweils vor der Ausführung der Aktion feststehen, wie z.B. bei der Benutzeraktion des Verschiebens von Figuren. Werden die relevanten Figuren erst im Rahmen der Ausführung der Benutzeraktion erzeugt, wie z.B. beim Duplizieren von Figuren, muss die Benutzeraktionsklasse diese und eventuell zusätzlich benötigte Werte in Form von Attributen und `get`-Methoden zur Verfügung stellen, damit die Rolle das erzeugte `Undoable`-Objekt initialisieren kann. In einem solchen Fall muss also die Benutzeraktions-Basisklasse auch in der ObjectTeams-orientierten Version Implementierungs-elemente enthalten, die unter Umständen ausschließlich der Undo-Programm-funktion dienen. Das heißt, die Undo-Programmfunktion kann nicht vollständig in Form eines Teams gekapselt werden.
Auch in den Fällen, in denen die für das Rückgängigmachen relevanten Figuren schon vor der Ausführung der Benutzeraktion feststehen, empfiehlt es sich, diese in Form einer `get`-Methode von der Benutzeraktions-Basisklasse selbst zur Verfügung stellen zu lassen und diese Methode über eine callout-Bindung von der Rollenklasse aus bei der Initialisierung des `Undoable`-Objekts aufzurufen. So kann sichergestellt werden, dass für eine von der

Benutzeraktionsklasse erbende Klasse, die sich von dieser nur in Bezug auf die Figuren unterscheidet, nur diese `get`-Methode überschrieben werden und keine neue Rollenklasse angelegt werden muss. Insbesondere im Hinblick darauf, dass es sich bei der betrachteten Software um eine Framework-Implementierung handelt, sollte diese Tatsache berücksichtigt werden.

Komplexität der auszulagernden Implementierungs-elemente

Neben den Benutzeraktionsklassen, bei denen die Erzeugung des `Undoable`-Objekts nur in einer Methode und zwar bedingungslos bei jedem Aufruf stattfindet und das erzeugte `Undoable`-Objekt immer vom selben dynamischen Typs ist, gibt es zahlreiche Benutzeraktionsklassen, bei denen die Erzeugung und Initialisierung von `Undoable`-Objekten komplexere Züge annimmt. Es gibt Fälle in denen, die Erzeugung als solche oder die zu instanziierende `Undoable`-Klasse von einer (unter Umständen recht komplexen) Bedingung abhängt und/oder in verschiedenen Methoden der Benutzeraktionsklasse verschiedene `Undoable`-Objekte erzeugt werden und/oder das `Undoable`-Objekt in einer Methode erzeugt und initialisiert und in einer anderen wieder gelöscht wird, zum Beispiel weil die betroffenen Figuren im Rahmen der Ausführung der Benutzeraktion nicht verändert wurden. Außerdem kommt es vor, dass eine Benutzeraktionsklasse ihr `Undoable`-Objekt in einer anderen Methode erzeugt, als ihre Superklasse.

Im Rahmen der vorliegenden Fallstudie konnte aufgrund zeitlicher Beschränkung die Verschiebung der Erzeugung und Initialisierung der `Undoable`-Objekte in Rollenklassen nicht für alle vorhandenen Benutzer-aktionsklassen sondern nur exemplarisch für einige vorgenommen werden. Dabei war diese Verschiebung auch in den genannten schwierigeren Fällen möglich, wobei der Aufwand für die die ObjectTeams-orientierte Umstrukturierung vorbereitende, Struktur verbessernde, interne Restrukturie-rung (Refactoring, siehe [FOW99]) der Benutzeraktionsklassen und die Komplexität der Rollenklassen entsprechend höher war.

Refactoring zur Bereitstellung geeigneter Join Points

Insbesondere in solchen Fällen, in denen in der objektorientierten Version die Erzeugung des `Undoable`-Objekts zwar in einer Methode der Benutzeraktionsklasse stattfindet, aber in Abhängigkeit von komplexen, erst im Verlauf der Ausführung der Methode ermittelbaren Bedingungen, unterschiedliche konkrete `Undoable`-Subtypen instanziiert werden, war es nötig, vor der Verschiebung der betreffenden Implementierungselemente in die Rollenklasse die Benutzeraktionsklasse selbst in Form eines Refactorings umzustrukturieren. Dabei wurden die in den betreffenden Benutzeraktions-methoden enthaltenen Anweisungen, wo dies möglich war, in logisch zusammengehörige Gruppen aufgeteilt und in neue Untermethoden ausgelagert. Diese Maßnahme war notwendig, um geeignete Methoden ("join points") für die callin-Anbindung der Rollenmethoden zur Verfügung zu stellen.

„callin without super call“ - Problematik

Weiterhin ist anzumerken, dass im Zusammenhang mit der Auslagerung der Erzeugung der `Undoable`-Objekte in Rollenklassen ein konzeptuelles Problem von ObjectTeams auffiel. Wenn in einer objektorientierten Implementierung eine Subklasse eine Methode ihrer Superklasse überschreibt, ohne die Methode der Superklasse aufzurufen (ohne "super-call"), dann wird bei einem Aufruf der Methode auf einem Objekt der Subklasse das in der Supermethode definierte Verhalten auch nicht ausgeführt. Besitzt im Gegensatz dazu in einer ObjectTeams-orientierten Implementierung die Superklasse eine an sie gebundene Rollenklasse und besitzt die Methode der Superklasse in der

Rollenklasse eine per callin-Bindung an sie gebundene Rollenmethode, so wird diese Rollenmethode nicht nur bei Aufrufen der entsprechenden Basismethode der Superklasse, sondern auch bei Aufrufen der Basismethode auf Objekten der Subklasse ausgeführt, und zwar unabhängig davon, ob die Methode der Subklasse die Methode der Superklasse aufruft oder nicht (siehe Abbildung 43). Der derzeitige Umfang der Konzepte von ObjectTeams beinhaltet nicht die Möglichkeit, dieses Verhalten zu unterbinden.

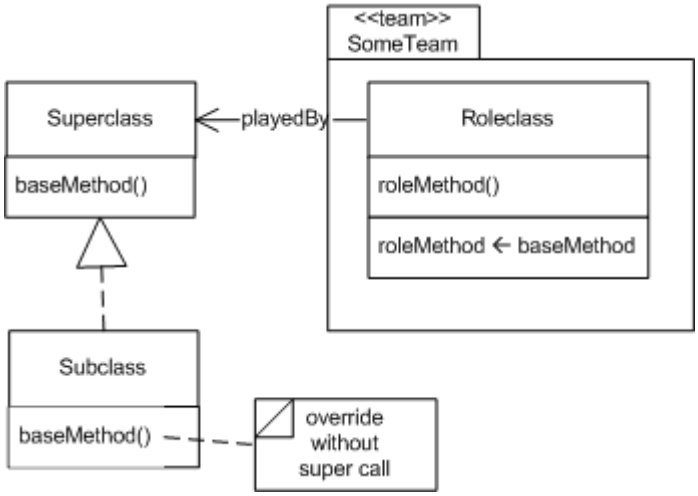


***Abbildung 43:*** *"callin without super call"-Problematik*

Im Zusammenhang mit dem hier betrachteten Fall der Verschiebung der Erzeugung und Initialisierung von `Undoable`-Objekten aus Benutzeraktionsklassen in an diese gebundene Rollenklassen ist diese Tatsache insofern relevant, als das manche Benutzeraktionsklassen in der objektorientierten Originalimplementierung ihre `Undoable`-Objekte nicht in der selben Methode erzeugen wie ihre Superklasse und aus diesem Grund die entsprechende Methode der Superklasse ohne super-Aufruf überschreiben. Werden die Methodenanteile, die die Erzeugung der `Undoable`-Objekte betreffen, nun im Rahmen der ObjectTeams-orientierten Restrukturierung in Rollenmethoden verschoben, die über callin an die Basismethoden gebunden werden, die in der objektorientierten Version die entsprechenden Aktionen ausführen, dann muss unterbunden werden, dass beim Aufruf der Basismethode, die bei der Super-Benutzeraktions-Basisklasse zur Erzeugung eines Undoable-Objekts durch callin-Aktivität führen soll, auf einem Objekt der Sub-Benutzeraktionsklasse ein `Undoable`-Objekt erzeugt wird.

Lösung für „callin without super call"-Problematik

Da es in der derzeitigen ObjectTeams/Java-Sprachversion nicht möglich ist, eine callin-Bindung der Superrolle aufzuheben bzw. umzudefinieren, ist die einzige Möglichkeit, in solchen Fällen korrektes Programmverhalten sicherzustellen, dass die entsprechenden Rollenklassen im Gegensatz zu den gebundenen Basisklassen nicht in einer Vererbungsbeziehung stehen dürfen. Abbildung 44 zeigt die resultierende Struktur.

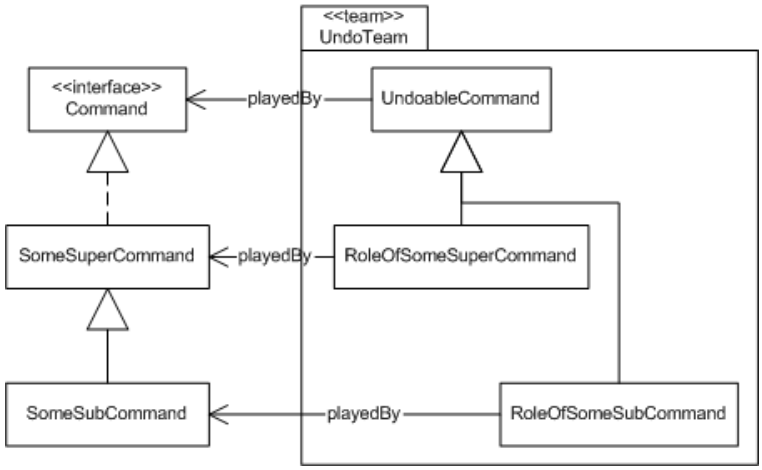


***Abbildung 44:*** *Lösung für "callin without super call"-Problem*

Auch wenn im Zusammenhang mit der Umstrukturierung der Undo-Programmfunktion diese Schwäche von ObjectTeams ins Auge fiel, kann festgestellt werden, dass es oftmals durch Refactoring der sich in einer Vererbungsbeziehung befindenden Benutzeraktions-Basisklassen möglich ist, diese Strukturschwäche zu vermeiden, weil die Methode der Rolle der Superklasse an eine Basismethode gebunden werden kann, die von der Sub-Basisklasse gar nicht aufgerufen wird.

Kopplung und Wiederverwendung

Neben den Verwebungen der Implementierungselemente der Ausführung der Benutzeraktion und der Erzeugung und Initialisierung des `Undoable`-Objekts, die sich aus logisch inhaltlichen Abhängigkeiten ergeben und sich in einer objektorientierten Implementierung nicht vermeiden lassen, enthält die JHotDraw-Originalimplementierung auch solche, die ausschließlich der Wiederverwendung und Vereinfachung der Implementierung dienen. Dazu gehört, dass die Benutzeraktionsklassen den Listenkopiermechanismus der `Undoable`-Klassen ausnutzen, ihr `Undoable`-Objekt teilweise zur Zwischenspeicherung von Werten verwenden und in einigen Fällen sogar die Methoden ihrer `Undoable`-Klasse zur Ausführung der eigentlichen Aktion nutzen. Diese Verwebungen erhöhen den Aufwand der Restrukturierung noch einmal zusätzlich.

Verschiebung der Hierarchie des Interface Undoable ins UndoTeam

Da es das Ziel ist, möglichst die gesamte Implementierung der Undo-Programmfunktion aus der originalen objektorientierten Framework-Struktur herauszulösen und sie in Form eines Teams zu kapseln, müssen neben der Erzeugung, Initialisierung und Übermittlung der `Undoable`-Objekte auch das Interface `Undoable` selbst und alle dieses implementierenden Klassen ins Team verlagert werden.
Das Interface `Undoable` und die einzige direkt implementierende Klasse `UndoableAdapter` können hierbei so wie sie sind ins `UndoTeam` verschoben werden. Alle weiteren Klassen, die `Undoable` implementieren, sind in der objektorientierten Implementierung nicht eigenständig und freistehend, sondern jeweils als innere Klasse an eine Benutzeraktionsklasse gebunden. Wie im Abschnitt „Undoable-Erzeugung und nested innerclasses“ auf Seite 79 ausführlich erläutert, trägt diese Struktur der Tatsache Rechnung, dass die meisten `Undoable`-Implementierungsklassen ausschließlich von der sie jeweils umgebenden Benutzeraktionsklasse instanziiert werden. Da im Rahmen der ObjectTeams-orientierten Umstrukturierung diese Instanziierung aus den

Benutzeraktionsklassen in an diese gebundene Rollen verschoben wurde und in diesem Zusammenhang für jede Benutzeraktions-Basisklasse eine eigene Rollenklasse existiert, das heißt die Zuordnung eindeutig ist, ist es naheliegend, jede `Undoable`-Implementierungsklasse aus der sie umgebenden Benutzeraktionsklasse in deren Rollenklasse zu verschieben.

Rollenklassen und statische Elemente

Als Ergebnis dieser Verschiebung wären die `Undoable`-Implementierungsklassen innere Klassen von Rollenklassen. Im Zusammenhang mit dem Umstand, dass die `Undoable`-Implementierungsklassen in der objektorientierten Implementierung keine normalen sondern statische innere Klassen sind, gibt es dabei folgendes Problem. Eine Rollenklasse ist bezogen auf das sie umgebende Team eine innere Klasse und darf als solche zwar selbst wieder innere Klassen, aber keine statischen Elemente besitzen.
Nun muss eine Möglichkeit gefunden werden, dieses Problem zu umgehen. Das heißt, es muss eine Struktur gefunden werden, bei der die `Undoable`-Implementierungsklassen keine statischen inneren Klassen von Rollen sind. Dabei darf sich diese Struktur nicht nachteilig auf die Qualität der Software auswirken. Das soll heißen, die Software darf durch die neue Struktur keine wichtigen Eigenschaften einbüßen. Um eine solche Struktur zu finden, ist es notwendig zu ermitteln, warum die `Undoable`-Implementierungsklassen in der objektorientierten Implementierung keine normalen sondern statische innere Klassen sind. Es muss herausgefunden werden, welche Anforderungen durch diese Struktur erfüllt werden und welche Vorteile sie mit sich bringt.
Eine statische innere Klasse unterscheidet sich von einer normalen inneren Klasse im Bereich der Instanziierung. Eine normale innere Klasse kann nur in Abhängigkeit von der Existenz einer Instanz der sie umgebenden Klasse instanziiert werden. Im Gegensatz dazu kann eine statische innere Klasse genauso wie eine eigenständige, freistehende Klasse instanziiert werden. Übertragen auf den Fall der `Undoable`-Implementierungsklassen heißt das also, dass diese in der objektorientierten Implementierung statisch sind, damit sie nicht nur von Objekten der sie umgebenden Benutzeraktionsklasse instanziiert werden können. Nun werden aber die meisten `Undoable`-Implementierungsklassen im Framework JHotDraw ausschließlich von der sie jeweils umgebenden Benutzeraktionsklasse instanziiert. In diesen Fällen ist die statische Eigenschaft, zumindest oberflächlich betrachtet, also überflüssig. Das heißt, solche `Undoable`-Implementierungsklassen könnten auch in der ObjectTeams-orientierten Implementierung innere Klassen bleiben. Die statische Eigenschaft müsste dafür entfernt werden, was weder zu verändertem noch zu fehlerhaftem Programmverhalten führen würde. Für alle anderen `Undoable`-Implementierungsklassen, also für solche, die von den Objekten unterschiedlicher Benutzeraktionsklassen bzw. Rollenklassen instanziiert werden, ist die Entfernung der statischen Eigenschaft keine Lösung. Diese müssen zu selbständigen ungebundenen Rollenklassen umgewandelt werden (siehe Abbildung 45).

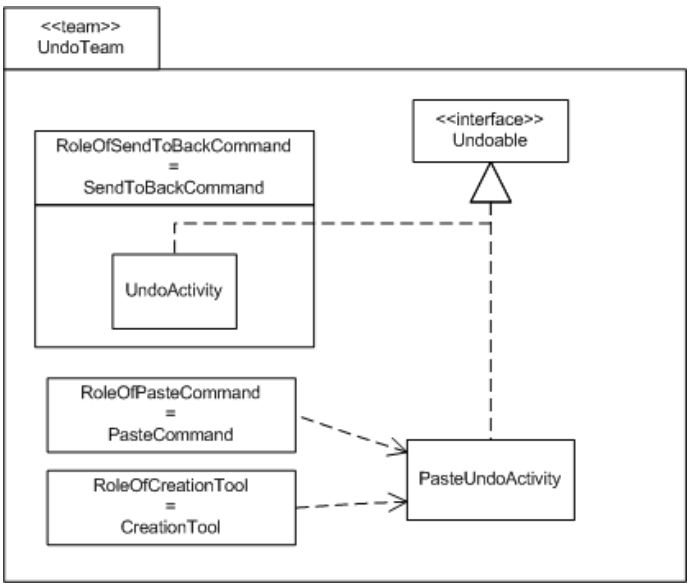


***Abbildung 45:*** *Undo - Struktur der Undoable-Hierarchie, inhomogen*

Die eben beschriebene Lösung, manche `Undoable`-Implementierungsklassen zu inneren Klassen der sie instanziierenden Rollenklassen und andere zu ungebundenen Rollenklassen umzuwandeln, schränkt die Qualität der Implementierung nicht nur durch ihren inhomogenen Charakter ein. Da es sich um eine Framework-Implementierung handelt und sie als solche dafür erstellt wurde, um als Basis für Erweiterungen zu dienen, ist die statische Eigenschaft der `Undoable`-Implementierungsklassen, die nur von der sie jeweils umgebenden Klasse instanziiert werden, nur oberflächlich betrachtet überflüssig. Sie erhöht bzw. ermöglicht Wiederverwendbarkeit. Dadurch, dass sie keine normale sondern eine statische innere Klasse ist, kann mit der objektorientierten Originalimplementierung bei Bedarf jede im Framework existierende `Undoable`-Implementierungsklasse von den Klassen einer auf JHotDraw basierenden Applikation verwendet werden. Diese Eigenschaft darf durch die ObjectTeams-orientierte Umstrukturierung des Frameworks nicht verloren gehen. Deshalb müssen alle `Undoable`-Implementierungsklassen zu freistehenden ungebundenen Rollenklassen umgewandelt werden (siehe Abbildung 46).

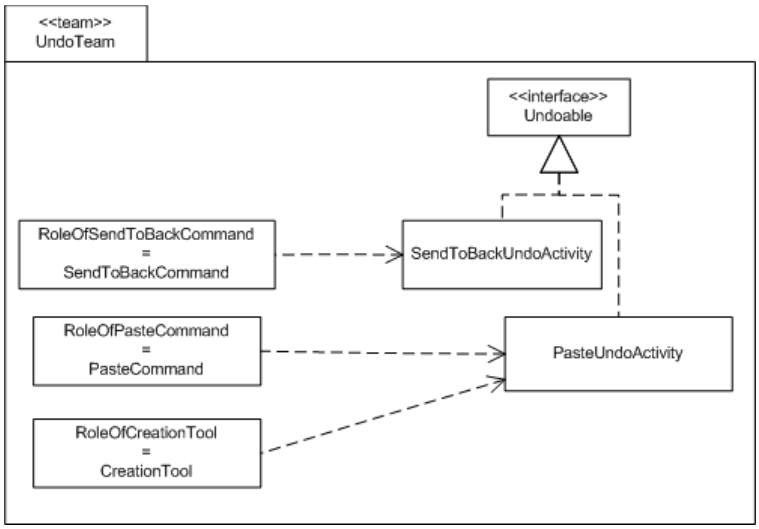


***Abbildung 46:*** *Undo - Struktur der Undoable-Hierarchie, homogen*

Dabei ist zu beachten, dass sie zu diesem Zweck umbenannt werden müssen. In der objektorientierten Implementierung besitzen nämlich alle `Undoable`-Implementierungsklassen, die innere Klasse sind, den selben Namen, `UndoActivity`. Dies ist möglich, weil sich eine innere Klasse im geschlossenen Namensraum der sie umgebenden Klasse befindet. Von außen wird sie über ihren qualifizierten Namen angesprochen, der den Namen der umgebenden Klasse enthält. Beispiele hierfür sind die Klassen `SendToBack-Command.UndoActivity` und `CreationTool.UndoActivity`. Werden die `Undoable`-Implementierungsklassen nun ins `UndoTeam` verschoben und verlieren dabei ihre "Innerclass-Eigenschaft", befinden sie sich im selben Namensraum und müssen umbenannt werden, um in diesem unterschieden werden zu können.

Alternativer Ansatz zur internen Strukturierung des UndoTeams

Nachdem nun ausführlich erläutert wurde, wie der gesamte den Typen `Undoable` betreffende Implementierungskomplex, bestehend aus der Bereitstellung der `Undoable`-Implementierungsklassen, ihrer Instanziierung und dem anschließenden Transfer der erzeugten `Undoable`-Objekte zum `UndoManager`, in ein Team ausgelagert werden kann, soll nun ein alternativer Ansatz zur internen Strukturierung des Teams, die mit diesem verbundenen Probleme und die daraus resultierenden Erweiterungsvorschläge für das ObjectTeams-Sprachkonzept vorgestellt werden.
Im `UndoTeam` in seiner bisher vorgestellten Form gibt es einerseits ungebundene Rollenklassen, die das Interface `Undoable` implementieren und andererseits gebundene Rollenklassen, die, callin-getrieben, die eben genannten `Undoable`-Implementierungsklassen instanziieren und die erzeugten Objekte zum `UndoManager` transferieren. Der im folgenden erläuterte alternative Ansatz würde es erlauben, beide Seiten miteinander zu verschmelzen und dadurch die Bereitstellung einer `Undoable`-Implementierungsklasse, ihre Instanziierung und den Transfer der Objekte in jeweils einer Rollenklasse zusammenzufassen.

Idee

Die zugrundeliegende Idee ist die folgende: ObjectTeams sieht vor, dass wenn auf einem potentiellen Basisobjekt, das heißt auf einem Objekt, zu dessen Klasse es im Team eine gebundene Rollenklasse gibt, eine Methode

aufgerufen wird, für die erstens in der Rollenklasse eine callin-gebundene Rollenmethode existiert, es zweitens im Team noch kein mit diesem Basisobjekt assoziiertes Rollenobjekt gibt und drittens die Team-Instanz unbeschränkt aktiviert ist, von der Laufzeitumgebung ein neues Rollenobjekt erzeugt wird, um die callin-gebundene Rollenmethode auf diesem ausführen zu können. Dieses Prinzip kann man sich nun zunutze machen. Die `Undoable`-Implementierungsklasse kann selbst mit einer callin-Methode an die Basisklasse gebunden werden. Der Aufruf der callin-gebundenen Methode führt dann automatisch zu ihrer Instanziierung. Das erzeugte Objekt kann sich bzw. eine Referenz auf sich danach selbst dem `UndoManager` übergeben.

Probleme

Dieser Ansatz kommt derzeitig über das Stadium einer Idee nicht hinaus, weil er an zwei verschiedenen Stellen an den Grenzen des ObjectTeams-Sprachkonzepts scheitert. Das derzeitige ObjectTeams-Konzept sieht vor, das es für jedes Basisobjekt höchstens ein mit diesem assoziiertes Objekt einer bestimmten Rollenklasse pro Team-Instanz gibt. Oder wie oben bereits erwähnt, nur wenn zu einem bestimmten Basisobjekt noch kein assoziiertes Rollenobjekt existiert, führt der Aufruf der callin-gebundenen Methode zur Erzeugung eines neuen Rollenobjekts. Das ist im hier besprochenen Anwendungsfall der Undo-Programmfunktion aber unzureichend. Bei jeder Ausführung einer Benutzeraktion muss ein neues `Undoable`-Objekt erzeugt werden. Übertragen auf die Implementierungsebene bedeutet das, dass bei jedem Aufruf einer bestimmten callin-gebundenen Basismethode ein neues Rollenobjekt erzeugt werden müsste. Dadurch gäbe es zu ein und demselben Basisobjekt unter Umständen mehrere mit diesem assoziierte Rollenobjekte der selben Rollenklasse in der selben Team-Instanz. Da dieser Fall im derzeitigen ObjectTeams-Sprachkonzept nicht vorgesehen ist, leitet sich daraus der erste Vorschlag für dessen Erweiterung ab.

Spracherweiterungs-vorschlag

Ein Rollenobjekt sollte sich bei seiner Team-Instanz für die Annahme weiterer callin-Aufrufe abmelden können. In einem solchen Fall sollte der nächste Aufruf einer callin-gebundenen Methode auf dem Basisobjekt, welches mit der abgemeldeten Rolle assoziiert ist, zur Erzeugung eines neuen Rollenobjekts führen.

Nun zum zweiten Problem. Wie im Abschnitt „Die ObjectTeams-Variante des Decorator Design Patterns“ auf Seite 90 bereits erläutert wurde, sollen nur die Aktionen bestimmter, durch den Programmierer festgelegter Benutzeraktionsobjekte rückgängig machbar sein. Das heißt, eine `UndoTeam`-Instanz soll nur das Verhalten dieser Benutzeraktions-Basisobjekte durch die callin-Aktivität zugehöriger Rollenobjekte erweitern. Handelt es sich dabei nicht um alle im System vorhandenen Benutzeraktionsobjekte, darf das `UndoTeam` nur beschränkt auf das Level FROZEN aktiviert und die zu erweiternden Benutzeraktions-Basisobjekte müssen explizit beim Team registriert werden.

Das derzeitige ObjectTeams-Sprachkonzept sieht vor, dass direkt bei der Registrierung eines Basisobjekts durch Lifting ein mit diesem assoziiertes Rollenobjekt erzeugt wird. Dies ist notwendig, weil FROZEN-aktivierte Team-Instanzen nur dann auf den Aufruf einer Basismethode mit der Ausführung der an sie gebundenen Rollenmethode reagieren, wenn schon vor diesem Aufruf ein entsprechendes Rollenobjekt existiert hat. Das heißt, anders als bei einer unbeschränkten Team-Instanz, führt bei FROZEN-Aktivierung der Aufruf einer callin-gebundenen Basismethode in keinem Fall zur Erzeugung eines Rollenobjekts. Das Problem ist nun, dass genau dieser Mechanismus der Rollenerzeugung durch callin-Aufruf die Grundlage des hier diskutierten alternativen Ansatzes zur internen Strukturierung des `UndoTeams` ist. Daher

muss der oben beschriebene Vorschlag zur Erweiterung des ObjectTeams-Sprachkonzepts für die Anwendung in beschränkt aktivierten Team-Instanzen erweitert werden.

Erweiterung des Spracherweiterungs-vorschlags

Es sollte möglich sein, dass sich ein Objekt bei einer beschränkt aktivierten Team-Instanz sozusagen als "zu beobachtendes Basisobjekt" registriert, und auch dann weiter bezüglich erforderlicher callin-Aktivität beobachtet wird, wenn sich das erste zugehörige Rollenobjekt für die Annahme weiterer callin-Aufrufe abgemeldet hat. Anders ausgedrückt, es sollte auch bei beschränkt aktivierten Team-Instanzen möglich sein, dass der Aufruf einer callin-gebundenen Basismethode zur Erzeugung eines neuen Rollenobjekts führt.

## 6.7 Verwendung der Framework-Implementierung bei der Erstellung einer Applikation

Nachdem in Kapitel 6.3 die originale objektorientierte Implementierung und in Kapitel 6.6 die ObjectTeams-orientierten Implementierung der Programmfunktion „Rückgängigmachen einer Benutzeraktion" vorgestellt wurden, zeigt das vorliegende Kapitel, wie diese im Rahmen der Entwicklung einer auf dem Framework JHotDraw aufbauenden Applikation verwendet werden. Dabei orientiert sich die Darstellung an den folgenden drei Fragestellungen, die bezüglich dieser Verwendung zu beantworten sind:

- Wie kann festgelegt werden, ob die Applikation die Undo-Programmfunktion überhaupt zur Verfügung stellt?
- Wie kann festgelegt werden, welchen Umfang diese haben soll, also welche Benutzeraktionen rückgängig gemacht werden können?
- Was muss getan werden, um selbst definierte Benutzeraktionsklassen mit der Fähigkeit auszustatten, rückgängig gemacht werden zu können?

Um einen besseren Vergleich zu ermöglichen, wird jede Fragestellung jeweils einzeln einerseits für die Verwendung der objektorientierten Originalversion und andererseits für die Verwendung der ObjectTeams-orientierten Versionen des Frameworks beantwortet.

### 6.7.1 Festlegen des Vorhandenseins der Undo-Programmfunktion

Die Undo-Programmfunktion ist in JHotDraw-basierten Applikationen standardmäßig über die Optionen "Undo" und "Redo" des Edit-Menüs erreichbar. Soll in einer Applikation das Rückgängigmachen von Benutzeraktionen nicht möglich sein, darf sie die entsprechenden Menüoptionen nicht anbieten. Die Möglichkeit, aufbauend auf JHotDraw sowohl solche Applikationen zu erstellen, die das Rückgängigmachen ermöglichen, als auch solche, die dies nicht tun, ist sowohl bei der Verwendung der originalen objektorientierten Framework-Implementierung als auch bei der Verwendung der ObjectTeams-orientierten Implementierung gegeben.

#### *Objektorientierte Implementierung*

Bei der Verwendung der objektorientierten Framework-Implementierung ist die Undo-Programmfunktion standardmäßig vorhanden. Soll die Applikation ihren Benutzern die Funktionalität nicht anbieten, muss ihre zentrale Klasse die von der Klasse `DrawApplication` geerbte Methode `createEditMenu` überschreiben. Dabei darf die überschreibende Methode die Supermethode nicht aufrufen, sondern muss deren Inhalt bis auf die Erzeugung der "Undo"-

und "Redo"-Menüoptionen kopieren.

***ObjectTeams-orientierte Implementierung***

Wird eine Applikation aufbauend auf der ObjectTeams-orientierten Framework-Implementierung erstellt, ist das Vorhandensein der Undo-Programmfunktion nicht der Standardfall. Damit die Applikation die entsprechenden Menüoptionen bereitstellt, muss innerhalb der `Main`-Methode des Programms, vor dem Öffnen des ersten Editorfensters eine Instanz des Typs `UndoTeam` erzeugt und aktiviert werden. Falls diese Team-Instanz nur beschränkt aktiviert wird (siehe nächstes Unterkapitel), muss zusätzlich das Objekt, welches das Editorfenster repräsentiert, bei der Team-Instanz registriert und die Team-Instanz der Klasse `TeamDispatcher` mittels dessen Methode `setUndoPerformer` übergeben werden (nähere Informationen hierzu siehe Abschnitt „Eine zentrale Team-Instanz“ ab Seite 92 und Abbildung 40).

### 6.7.2 Festlegen des Undo-Funktionsumfangs

JHotDraw bietet die Möglichkeit, den Umfang der Undo-Funktionalität einer Applikation selbst zu bestimmen. Das heißt, der Entwickler kann festlegen, welche Benutzeraktionsarten in einem konkreten Programm rückgängig machbar sein sollen.

***Objektorientierten Implementierung***

Bei der Verwendung der objektorientierten Framework-Implementierung geschieht die Festlegung, dass eine bestimmte Benutzeraktionsart rückgängig machbar ist, dadurch, dass nach der Instanziierung der entsprechenden Benutzeraktionsklasse das erzeugte Objekt mit einer passenden Wrapper-Instanz ummantelt wird und der Klient anstelle einer Referenz auf das Benutzeraktionsobjekt eine Referenz auf das umhüllende Wrapper-Objekt erhält (siehe Abschnitt „Undoable-Weiterleitung und Decorator Design Pattern“ auf Seite 80).

Handelt es sich bei der instanziierten Benutzeraktionsklasse um eine Implementierungsklasse des Interface `Command`, muss das erzeugte Objekt mit einer Instanz der Klasse `UndoableCommand` ummantelt werden, handelt es sich um eine `Tool`-Implementierungsklasse mit einer `UndoableTool`-Instanz und in dem Fall, dass es sich um eine `Handle`-Implementierungsklasse handelt ist die passende Wrapper-Klasse `UndoableHandle` (siehe Abschnitt „Die Typen Command, Tool, Handle und ihre Wrapper“ auf Seite 81).

Die meisten der von der Framework-Implementierung bereitgestellten Benutzeraktionen, zum Beispiel die, welche über das Edit-Menü erreichbar sind, sind standardmäßig rückgängig machbar, weil die entsprechenden Benutzeraktionsobjekte mit einer Wrapper-Instanz umhüllt sind. Sollen bestimmte Benutzeraktionsarten, die in der Standardimplementierung rückgängig machbar sind, in der aufsetzenden Applikation diese Eigenschaft nicht besitzen, muss die Applikation die verantwortlichen Methoden in einer eigenen Subklasse überschreiben und die entsprechenden Benutzeraktionsklassen neu instanziieren ohne die erzeugten Objekte danach mit einer Wrapper-Instanz zu umhüllen. Die für den Inhalt des Edit-Menüs verantwortliche Methode ist zum Beispiel `DrawApplication.createEdit-Menu`.

***ObjectTeams-orientierte Implementierung***

Bei Verwendung der ObjectTeams-orientierten Framework-Implementierung geschieht die Festlegung des Umfangs der Undo-Programmfunktion über ein Zusammenspiel aus dem konkreten Typ des instanziierten `UndoTeams`, der Art der Team-Aktivierung und der Registrierung von Benutzeraktionsobjekten.

Verwendet die Applikation ausschließlich rückgängig machbare Benutzeraktionsklassen, die vom Framework zur Verfügung gestellt werden, ist es ausreichend, die ebenfalls vom Framework zur Verfügung gestellte Klasse `UndoTeam` zu instanziieren und zu aktivieren.

Verwendet die Applikation aber auch selbst definierte Benutzeraktionsklassen, für die in einer Subklasse des `UndoTeams` eigenes Undo-relevantes Verhalten definiert wurde (siehe Kapitel „ObjectTeams-orientierte Implementierung“ auf Seite 109), so muss anstelle der Framework-Klasse `UndoTeam` die entsprechende Subklasse instanziiert und das erzeugte Objekt aktiviert werden.

Bei der Verwendung der ObjectTeams-orientierten Implementierung hat der Entwickler die Möglichkeit, indem er die Team-Instanz unbeschränkt aktiviert, alle Benutzeraktionen in den Funktionsumfang der Undo-Programmfunktion mit einzuschließen, ohne dass diese explizit registriert werden müssen.
Dabei gibt es die Einschränkung, dass nur solche Benutzeraktionen rückgängig gemacht werden können, zu deren Benutzeraktionsklasse oder Superklasse derselben eine entsprechende, an sie gebundene Rollenklasse im Team existiert, die `Undoable`-Objekte erzeugt und transferiert.

Um zu erreichen, dass nicht alle sondern nur ganz bestimmte, selbst festzulegende Benutzeraktionsarten rückgängig gemacht werden können, muss die Team-Instanz beschränkt (mittels Übergabe des Parameters `org.objectteams.Team.FROZEN`) aktiviert werden. Danach muss sie der Klasse `TeamDispatcher` übergeben werden.
Die Festlegung, dass eine bestimmte Benutzeraktionsart rückgängig machbar ist, erfolgt dadurch, dass nach der Instanziierung der entsprechenden Benutzeraktionsklasse das erzeugte Objekt bei der Team-Instanz registriert wird. Hierzu existiert die statische Methode `TeamDispatcher.registerForUndo` (siehe Abschnitt „Eine zentrale Team-Instanz“ auf Seite 92).

Die meisten der von der Framework-Implementierung bereitgestellten Benutzeraktionen, zum Beispiel die, welche über das Edit-Menü erreichbar sind, sind nach der Aktivierung einer `UndoTeam`-Instanz standardmäßig rückgängig machbar, weil die entsprechenden Benutzeraktionsobjekte bereits durch die Framework-Implementierung registriert werden. Sollen bestimmte Benutzeraktionsarten, die bei gegebener `UndoTeam`-Aktivierung in der Standardimplementierung rückgängig machbar sind, in der aufsetzenden Applikation diese Eigenschaft nicht besitzen, muss die Applikation die verantwortlichen Methoden, wie auch bei Verwendung der objektorientierten Implementierung, in einer eigenen Subklasse überschreiben und die entsprechenden Benutzeraktionsklassen neu instanziieren ohne die erzeugten Objekte danach bei einer `UndoTeam`-Instanz zu registrieren.

### 6.7.3 „Rückgängigmachbar-Machen“ selbst definierter Benutzeraktionsklassen

Definiert eine auf JHotDraw basierende Applikation neue Benutzeraktionsklassen und sollen die durch diese repräsentierten Benutzeraktionen rückgängig gemacht werden können, sind bei der Erstellung der Applikation die im folgenden erläuterten Dinge zu beachten und Arbeitsschritte auszuführen.

#### *Objektorientierte Implementierung*

Falls die Framework-Implementierung keine zum Rückgängigmachen der entsprechenden Benutzeraktionen geeignete `Undoable`-Implementierungsklasse zur Verfügung stellt, muss eine solche erstellt werden. Diese kann, wie die `Undoable`-Implementierungsklassen des Frameworks als (statische) innere Klasse der Benutzeraktionsklasse oder als selbstständige Klasse definiert werden.

Die für das Rückgängigmachen geeignete `Undoable`-Implementierungsklasse muss von der Benutzeraktionsklasse an geeigneter Stelle innerhalb der zur Ausführung der eigentlichen Benutzeraktion zuständigen Methoden instanziiert und mit den für das Rückgängigmachen relevanten Figuren und eventuellen weiteren Werten initialisiert werden.

Besitzt die neue Benutzeraktionsklasse eine Superklasse, muss berücksichtigt werden, in welchen Methoden diese `Undoable`-Objekte erzeugt. Soll bei der Ausführung einer bestimmten Methode, bei der in der Superklasse ein `Undoable`-Objekt erzeugt wird, in der neuen Subklasse kein `Undoable`-Objekt erzeugt werden, so muss die Subklasse die entsprechende Methode der Superklasse überschreiben, ohne dabei die originale Version der Methode aus der Superklasse aufzurufen (ohne "super-call").

Handelt es sich bei der Benutzeraktionsklasse um eine Implementierungsklasse der Interfaces `Command`, `Tool` oder `Handle`, muss sie das erzeugte `Undoable`-Objekt über die Methode `getUndoActivity` von außen zugreifbar machen.

Handelt es sich bei der Benutzeraktionsklasse nicht um eine Implementierungsklasse der genannten Interfaces, muss sie den Transfer der erzeugten `Undoable`-Objekte zum `UndoManager` und eventuell erforderliche Menü-Aktualisierungen selbst implementieren. Diese Implementierung bietet nicht die Flexibilität, für jede Instanz der Benutzeraktionsklasse festzulegen, ob die im Rahmen der Ausführung ihrer Methoden erfolgten Benutzeraktionen rückgängig machbar sein sollen oder nicht.

Um diese Flexibilität zur Verfügung zu stellen, muss eine Wrapper-Klasse erstellt werden, die die Aufgaben des `Undoable`-Transfers und der Menü-Aktualisierung übernimmt. Diese Wrapper-Klasse muss das selbe Interface implementieren wie die Benutzeraktionsklasse. Falls dies erforderlich ist, muss zu diesem Zweck ein Interface erstellt werden.

#### *ObjectTeams-orientierte Implementierung*

Definiert eine auf JHotDraw basierende Applikation eine neue Benutzeraktionsklasse, die die Eigenschaft der "Rückgängigmachbarkeit"

besitzen soll, muss eine neue Team-Klasse von der vom Framework zur Verfügung gestellten Klasse `UndoTeam` abgeleitet werden. Innerhalb dieser Team-Klasse muss eine neue Rollenklasse definiert werden, die an die Benutzeraktionsklasse gebunden wird.

Falls die Framework-Implementierung in der Klasse `UndoTeam` keine zum Rückgängigmachen der entsprechenden Benutzeraktionen geeignete `Undoable`-Implementierungsklasse zur Verfügung stellt, muss in dem neuen Sub-Team der Klasse `UndoTeam` eine solche erstellt werden.

Die für das Rückgängigmachen geeignete `Undoable`-Implementierungsklasse muss von der an die Benutzeraktionsklasse gebundenen Rollenklasse instanziiert und mit den für das Rückgängigmachen relevanten Figuren und eventuellen weiteren Werten initialisiert werden. Der Aufruf der erzeugenden Rollenmethode erfolgt dabei über callin. Sollte es nicht möglich sein, die erforderlichen Figuren und Werte als Parameter beim Aufruf der callin-gebundenen Methode zu übergeben, erfolgt der Zugriff über callout auf entsprechende Methoden der Basisklasse, die diese zur Verfügung stellen.
Bei der Erstellung der Benutzeraktionsklasse muss darauf geachtet werden, dass diese eine für die callin-Bindung der Rollenmethode geeignete Methode (join point) zur Verfügung stellt. Sollte es nicht möglich sein, die für das Rückgängigmachen relevanten Figuren und eventuell zusätzlich benötigte Werte als Parameter dieser Methode zu übergeben, muss die Benutzeraktionsklasse zusätzlich Methoden zur Verfügung stellen, die den Zugriff auf diese Figuren und Werte ermöglichen.

Besitzt die neue Benutzeraktionsklasse eine Superklasse, muss berücksichtigt werden, ob diese auch eine an sie gebundene Rollenklasse im `UndoTeam` oder dessen Sub-Team besitzt und an welche Methode der Benutzeraktions-Superklasse die `Undoable`-Objekte erzeugende Rollenklasse per callin gebunden ist. Soll bei der Ausführung einer bestimmten Methode, bei der bei der Benutzeraktions-Superklasse durch callin-Aktivität ein `Undoable`-Objekt erzeugt wird, bei der neuen Benutzeraktions-Subklasse kein `Undoable`-Objekt erzeugt werden, so darf die Rollenklasse der Subklasse die Rollenklasse der Superklasse nicht beerben (siehe hierzu Abschnitt „Lösung für „callin without super call“-Problematik“ auf Seite 100).

Handelt es sich bei der Benutzeraktionsklasse um eine Implementierungsklasse der Interfaces `Command`, `Tool` oder `Handle`, muss die gebundene Rollenklasse die an das entsprechende Interface gebundene Rollenklasse, also `UndoableCommand`, `UndoableTool` oder `UndoableHandle` beerben (siehe Abschnitt „Vererbungsbeziehung der Rollenklassen“ auf Seite 97).

Handelt es sich bei der Benutzeraktionsklasse nicht um eine Implementierungsklasse der genannten Interfaces, muss die an sie gebundene Rollenklasse zusätzlich den Transfer der erzeugten `Undoable`-Objekte zum `UndoManager` und eventuell erforderliche Menü-Aktualisierungen selbst implementieren.

## 6.8 Fazit

Nachdem in Kapitel 6.3 die objektorientierte Originalimplementierung der Programmfunktion "Rückgängigmachen einer Benutzeraktion" des Frameworks JHotDraw, in Kapitel 6.6 die ObjectTeams-orientierte Implementierung und in Kapitel 6.7 die Instanziierung beider Framework-Versionen ausführlich

beschrieben wurden, sollen im Rahmen dieses Kapitels Schlussfolgerungen aus diesem Anwendungsbeispiel gezogen werden. Dabei orientiert sich die Darstellung, wie auch schon bei der Lade-Speicher-Programmfunktion in Kapitel 4.6 auf Seite 52, an den Fragen, die im Rahmen der vorliegenden Arbeit beantwortet werden sollten.

### 6.8.1 Anwendbarkeit der ObjectTeams-Konzepte

Sind die Konzepte von ObjectTeams bei der Implementierung einer komplexen Software anwendbar? Insbesondere, lässt sich die Implementierung einer Programmfunktion aus einer existierenden objektorientierten Software extrahieren und in Form eines Teams kapseln.

Kapitel 6.6 zeigt, dass sich die Implementierung der Programmfunktion "Rückgängigmachen einer Benutzeraktion" des Frameworks JHotDraw zu großen Teilen aus der objektorientierten Originalsoftware herauslösen und in Form eines Teams kapseln lässt. Eine vollständige Kapselung ist in diesem Fall allerdings nicht möglich, da die Basisklassen die für das Rückgängigmachen relevanten Figur-Objekte und, falls im Einzelfall erforderlich, weitere Werte explizit zur Verfügung stellen müssen (siehe Abschnitt „Initialisierung von Undoable-Objekten ist nicht vollständig in Team-Form kapselbar“ auf Seite 98).

### 6.8.2 Funktionsfähigkeit von Compiler und Laufzeitumgebung und sprachliche Schwächen

In welchem Umfang wurden die in Kapitel 6.6 vorgestellten Ansätze implementiert und stehen in kompilier- und ausführbarem Zustand zur Verfügung? Welche sprachlichen Schwächen konnten ausfindig gemacht werden?

Aufgrund des großen Aufwandes zur Umsetzung der vorgestellten Ansätze wurde im Rahmen der vorliegenden Arbeit lediglich eine Teilimplementierung durchgeführt, die es ermöglichte, Aussagen über die Durchführbarkeit der vorgestellten Ansätze zu machen ("proof of concept").

Dabei hatte die ObjectTeams/Java-Laufzeitumgebung Schwierigkeiten im Zusammenhang mit super-Aufrufen in callin-gebundenen Basismethoden, die vom ObjectTeams/Java-Entwickler-Team zwischenzeitlich behoben wurden.

In diesem Zusammenhang fiel auch die im Abschnitt „„callin without super call“ - Problematik“ auf Seite 99 ausführlich erläuterte konzeptuelle Schwäche von ObjectTeams auf.

### 6.8.3 Restrukturierungsprozess

Der Restrukturierungsprozess als solcher war praktisch sehr aufwendig. Die Gründe dafür liegen in der starken inhaltlichen und physischen Kopplung der Programmfunktion des Rückgängigmachens an die Programmfunktion des Ausführens der Benutzeraktionen und den daher erforderlichen vorbereitenden internen Umstrukturierungen der Basisklassen (siehe hierzu die Abschnitte „Refactoring zur Bereitstellung geeigneter Join Points“ auf Seite 99 und „Kopplung und Wiederverwendung“ auf Seite 101).

### 6.8.4 Spracherweiterungsvorschäge

Eine Erweiterung bzw. Verfeinerung der Spezifizierungsmöglichkeiten für die Deklaration von callin-Bindungen wäre sehr sinnvoll (siehe Abschnitt „„callin

without super call“ - Problematik“ auf Seite 99). Es sollte möglich sein, eine von einer Superrolle geerbte callin-Bindung zu entfernen oder zu überschreiben bzw. bei der Deklaration einer callin-Bindung zu spezifizieren, dass die callin-gebundene Rollenmethode für Subklassen der gebundenen Basisklasse nicht aufgerufen werden soll, wenn diese die zugehörige Basismethode ohne super-Aufruf überschreiben.

Weiterhin wäre es sinnvoll, bei der eingeschränkten Aktivierung eines Teams spezifizieren zu können, dass das Team für bestimmte Rollenklassen trotz eingeschränkter Aktivierung genauso funktionieren soll wie bei uneingeschränkter Aktivierung, das heißt, die Objekte der an diese Rollenklassen gebundenen Basisklassen nicht explizit registriert werden müssen (siehe Abschnitt „Nebeneffekt der eingeschränkten Team-Aktivierung“ auf Seite 91 und „Spracherweiterungsvorschlag“ auf Seite 94).

Wie auch schon bei der Restrukturierung der Lade-Speicher-Programmfunktion, so fiel auch bei der Restrukturierung der Undo-Programmfunktion auf, dass es sinnvoll wäre, wenn Rollenklassen genauso wie gewöhnliche Klassen statische Elemente, in diesem Fall statische innere Klassen, besitzen könnten (siehe Abschnitt „Rollenklassen und statische Elemente“ ab Seite 102).

Ein weiterer Vorschlag für eine Erweiterung der ObjectTeams-Konzepte, der voteilhaft sein könnte, ist der, mehrere Rollenobjekte der selben Rollenklasse in der selben Team-Instanz zum selben Basisobjekt zuzulassen, wobei nur eines davon callin-Aufrufe entgegen nimmt (siehe „Spracherweiterungsvorschlag“ auf Seite 105). Und im Zusammenhang damit die Rolleninstanziierung per callin-Aufruf auch bei eingeschränkt aktivierten Teams (siehe „Erweiterung des Spracherweiterungsvorschlags“ auf Seite 106)

### 6.8.5 Ist Object Teams bei der Implementierung komplexer Systeme hilfreich

#### *Lokalität*

Durch den Einsatz der Konzepte von ObjectTeams konnte die Modularität und Lokalität der Implementierung der Undo-Programmfunktion verbessert werden. Es war möglich, Code Tangling und Scattering zu reduzieren, sie konnten aber aufgrund zu starker logischer Abhängigkeit der Aktion des Rückgängigmachens von der Aktion des Ausführens einer Benutzeraktion nicht vollständig vermieden werden (siehe Abschnitt „Initialisierung von Undoable-Objekten ist nicht vollständig in Team-Form kapselbar“ auf Seite 98).

Wie auch schon bei der Kapselung der Lade-Speicher-Programmfunktion war der zu zahlende Preis für die verbesserte Lokalität eine starke Erhöhung der Anzahl der Klassen.

#### *Verständlichkeit und Instanziierbarkeit*

Wird die Framework-Implementierung durch den Einsatz von ObjectTeams für den Benutzer verständlicher? Ist die Instanziierung einer Applikation auf der Grundlage der ObjectTeams-orientierten Framework-Version (dadurch) einfacher?

Besonders die Frage der Verständlichkeit kann für die Undo-Programmfunktion im Rahmen dieser Arbeit, genauso wie bereits für die Lade-Speicher-

Programmfunktion, nicht erschöpfend beantwortet werden, weil es nicht möglich ist, objektiv zu beurteilen, wie verständlich das Design und die Benutzung des Frameworks für einen Erstbenutzer sind, wenn man mit den internen Details der beiden Framework-Implementierungen (objektorientiert und ObjectTeams-orientiert) vertraut ist. Darüber hinaus kommt erschwerend hinzu, dass noch keine vollständige Implementierung der in Kapitel 6.6 präsentierten Modellierungen existiert. Daher ist es notwendig, dieser Frage im Rahmen weiterführender Forschungsaktivitäten nachzugehen, indem "neutrale" Personen, die zwar mit den Konzepten von Java und ObjectTeams/Java, nicht aber mit den beiden Framework-Implementierungen vertraut sind, mit der Aufgabe betraut werden, aufbauend auf beiden Framework-Versionen jeweils die gleiche Applikation zu erstellen.

Ich gewann den Eindruck, dass die Komplexität der Implementierung der Undo-Programmfunktion und insbesondere die der Benutzung bei der Instanziierung einer Applikation auf der Grundlage des Frameworks wesentlich höher ist als die der objektorientierten Original-Implementierung. Es müssen mehr Klassen erstellt und mehr Abhängigkeiten explizit beachtet und daher auch dokumentiert werden. Dazu gehört neben der Tatsache, dass eine neue Team-Klasse von UndoTeam abgeleitet und in dieser Team-Klasse neue Rollenklassen erstellt werden müssen, dass die Vererbungsbeziehung der Rollenklassen untereinander wegen dem im Abschnitt „„callin without super call“ - Problematik“ auf Seite 99 beschriebenen Problem unter Umständen von der Hierarchie der an die Rollenklassen gebundenen Basisklassen abweichen muss. Zu diesen Abhängigkeiten gehört auch die Tatsache, dass eine Instanz des neuen Teams vom Benutzer explizit aktiviert werden muss und die diesbezüglichen Besonderheiten der verschiedenen Aktivierungsarten.

### 6.8.6 Weitere Eigenschaften der ObjectTeams-orientierten Implementierung

Das `UndoTeam` beinhaltet mehrere aus relativ vielen Rollenklassen bestehende Hierarchien (Wurzelklassen: `UndoableCommand`, `UndoableTool`, `UndoableHandle`, `Undoable`) und viele callin-Bindungen. Es ist ein Beispiel für die Erfordernis eingeschränkter Team-Aktivierung und expliziter Registrierung von Basisobjekten.

Im Verlauf der Arbeit an diesem Beispiel konnte deutlich herausgearbeitet werden, dass bei der Nachbildung eines Design Patterns mittels der Konzepte von ObjectTeams die Rolle der Team-Aktivierungsart und die sich aus einer eingeschränkten Aktivierung ergebenden Konsequenzen für die Basissoftware (explizite Registrierung und damit Erfordernis der Bekanntheit des Teams in der Basissoftware) für den Vergleich objektorientierter und ObjectTeams-orientierter Konzepte mit berücksichtigt werden müssen (siehe Abschnitt „Die ObjectTeams-Variante des Decorator Design Patterns“ auf Seite 90).

## 6.9 Ausblick

Über die im Rahmen der vorliegenden Fallstudie durchgeführten Arbeiten und gesammelten Erkenntnisse, bleiben, wie bereits angeklungen ist, Aufgaben und Fragen offen, die im Rahmen weiterführender Forschungsaktivität durchgeführt bzw. beantwortet werden müssen.

Dazu gehört in erster Linie, dass die in Kapitel 6.6 vorgestellten Ansätze

vollständig implementiert werden müssen. In diesem Zusammenhang muss die Funktionsfähigkeit des ObjectTeam/Java-Compilers und der zugehörigen Laufzeitumgebung weiter erprobt und, falls dies erforderlich sein sollte, verbessert und ausgebaut werden.

Aufbauend auf dieser ObjectTeams-orientierten und der originalen objektorientierten Framework-Implementierung sollten äquivalente Applikationen erstellt werden, um in diesem Zusammenhang die Verständlichkeit und Instanziierbarkeit der beiden Framework-Versionen objektiv vergleichen zu können.

Daneben muss die Funktionsfähigkeit der ObjectTeams-orientierten Implementierung als solcher getestet und dabei untersucht werden, ob diese gegenüber der Originalimplementierung zu Seiteneffekten oder Einschränkungen führt, die im Rahmen der vorliegenden Arbeit nicht berücksichtigt wurden.

Ein weiterer offener Punkt ist die Frage, ob die vorgestellte Modellierung einer Abstraktion unterzogen werden kann, so dass sie auch außerhalb des Frameworks JHotDraw einsetzbar wird.

# 7 Fazit und Ausblick

Im Rahmen der vorliegenden Arbeit wurden die Implementierungen zweier Programmfunktionen, der Lade-Speicher- und der Undo-Programmfunktion, des Frameworks JHotDraw mit Hilfe von ObjectTeams umstrukturiert.

Dabei konnte gezeigt werden, dass es möglich ist, mit den Mitteln der Konzepte von Object Teams eine komplexe objektorientierte Software zu restrukturieren und dadurch die Modularität und Lokalität einzelner Programmfunktionen, deren objektorientierte Implementierungen eine hohe Streuung (Code Scattering) und Kopplung (Code Tangling) aufweisen, zu verbessern.

Es wurde dargelegt, wie Ansatzpunkte für eine solche Restrukturierung erkannt werden können, auf welche Art und Weise sie durchgeführt werden kann und was dabei zu beachten ist.

Dabei bilden die im Rahmen dieser Arbeit vorgestellten Beispiele einen guten Querschnitt durch die Möglichkeiten der Anbindung von Funktionalität mittels Teams. Das in Kapitel 4.4.1 vorgestellte `StorageTeam` ist mittels einer einzigen callin-Bindung mit der Basissoftware verbunden und kann im Zusammenhang mit dem in Kapitel 4.4.2 vorgestellten `StandardStorage-FormatTeam` eine geschachtelte Team-Struktur bilden. Das eben erwähnte `StandardStorageFormatTeam` enthält keine einzige callin-Bindung, wird wie eine gewöhnliche Klasse mittels ihrer Teamlevel-Methoden benutzt und kapselt einen rekursiven Algorithmus. Das in Kapitel 6.6 vorgestellte `UndoTeam` wird mittels vieler callin-Bindungen in die Basissoftware integriert und ist ein gutes Studienobjekt für die Anforderungen und Schwierigkeiten, die sich aus der Erfordernis einer eingeschränkten Team-Aktivierung ergeben.

Auch wenn gezeigt werden konnte, dass die Modularität einer Software durch den Einsatz von ObjectTeams verbessert werden kann, konnte die Frage nach der Verbesserung der Verständlichkeit der Framework-Implementierung nicht erschöpfend beantwortet werden und bedarf weiterer Untersuchungen. Es wurde aber deutlich, dass der Aufwand der Framework-Instanziierung durch den Einsatz von ObjectTeams, zumindest für die hier betrachteten Beispielfälle, steigt.

Im Rahmen des Entwurfs und der Realisation der vorgestellten Design-Ansätze wurde sowohl die Funktionsfähigkeit der sprachlichen Konzepte als auch die der derzeit vorhandenen Implementierungen des ObjectTeams/Java-Compilers und der zugehörigen Laufzeitumgebung überprüft. Dabei wurden eine Reihe von Schwachstellen aufgedeckt, die teilweise bereits begleitend zur Durchführung der vorliegenden Fallstudie in Zusammenarbeit mit dem ObjectTeams-Entwickler-Team behoben werden konnten. Zusätzlich wurden Ideen für den Ausbau der Konzepte von ObjectTeams gesammelt. Übersichten über die aufgedeckten Schwachstellen und Spracherweiterungsvorschläge befinden sich in Kapitel 4.6 ab Seite 52 und in Kapitel 6.8 ab Seite 110.

Trotz der Verschiedenheit der entwickelten Team-Klassen besitzen `StorageTeam` und `UndoTeam` eine Gemeinsamkeit bezüglich ihrer Anbindung an die Basissoftware, welche den Eindruck macht, dass sie eine über den Kontext des Frameworks JHotDraw hinaus einsetzbare Struktur darstellt und es

sich bei ihr eventuell um ein neues ObjectTeams-orientiertes Design Pattern für den Einsatz in Anwendungen mit graphischen Benutzeroberflächen (GUI) handelt. Dies zu überprüfen bleibt an dieser Stelle eine offene Aufgabe für weitere Forschungstätigkeit.
Sowohl das `StorageTeam` als auch das `UndoTeam` stellen dem Applikations-Benutzer ihre Funktionalität (zumindest teilweise) über Menüoptionen zur Verfügung. Dabei erzeugt die Basissoftware Benutzeroberfläche und Menüs. Ein Team integriert die zur Benutzung der in ihm gekapselten Funktionalität erforderlichen Menüoptionen dann selbst in die von der Basissoftware erzeugten Menüs. Dabei wird die Basismethode, welche ein bestimmtes Menü erzeugt, mittels eines replace-callins an die Rollenmethode `extendMenu` (siehe Abschnitte „Menüerweiterung“ auf Seite 32 und 85) gebunden. Diese führt zuerst mittels eines base-calls die Basismethode aus, lässt sich das Ergebnis, also das erzeugte Menü geben, erweitert dieses um die erforderlichen Menüoptionen, um es dann als eigenes Ergebnis zurückzugeben.

Neben den im Rahmen der vorliegenden Arbeit beschriebenen Anwendungsgebieten konnte ein weiterer Bereich identifiziert werden, in dem die Anwendung ObjectTeams-orientierter Konzepte Erfolg versprechend erscheint. Dabei handelt es sich um die Anbindung von Domänenlogik mittels Teams. Damit ist gemeint, dass bei der Erstellung eines (JHotDraw-basierten) graphischen Editors für eine ganz bestimmte Anwendungsdomäne, zum Beispiel eines Editors für UML-Klassendiagramme, die dieser Domäne innewohnende Logik, für dieses Beispiel also die spezifischen Eigenschaften von Paketen, Klassen, Vererbungsbeziehungen usw., mit Hilfe von Rollen an die Figuren gebunden werden kann, die diese Konzepte graphisch repräsentieren. Die Untersuchung dieses Ansatzes bleibt eine offene Aufgabe für weitere Forschungstätigkeit.

In diesem Zusammenhang ist die Untersuchung des Zusammenspiels der verschiedenen auf der Basis von Teams realisierten Funktionalitäten besonders interessant und wichtig. Es ist zu untersuchen, ob bzw. inwiefern die Implementierungen der Lade-Speicher-Programmfunktion (`StorageTeam` und `StandardStorageFormatTeam`) und der Undo-Programmfunktion (`UndoTeam`) verändert bzw. erweitert werden müssen, wenn die Anbindung von Domänenlogik mittels Teams realisiert wird.

# Abbildungsverzeichnis

Abbildung 1: Team-Klasse mit Rollen-Klassen, Darstellung in UFA ..... 10
Abbildung 2: Anbindung eines Teams an ein Paket mittels Rolle-Basis-Bindung, Darstellung in UFA ..... 11
Abbildung 3: Methodenbindungen, Darstellung in UFA ..... 12
Abbildung 4: Spezialisierung der Rolle-Basis-Bindung, eigene Notation ..... 13
Abbildung 5: Externalized Role, Instanz-gebundene Typ-Deklaration ..... 16
Abbildung 6: JHotDraw - Struktur und Instanziierung ..... 23
Abbildung 7: Laden / Speichern - objektorientierte Grundstruktur ..... 27
Abbildung 8: Laden / Speichern - objektorientiertes Verhalten ..... 28
Abbildung 9: Laden / Speichern - Standardformat, objektorientierte Struktur ..... 29
Abbildung 10: Laden / Speichern - Standardformat, objektorientiertes Verhalten ..... 30
Abbildung 11: Laden/Speichern - ObjectTeams-orientiertes Design, Ansatz 1 ..... 34
Abbildung 12: Laden/Speichern - ObjectTeams-orientiertes Design, Ansatz 2 ..... 35
Abbildung 13: Erzeugung und Lifting eines StorageFormat-Objekts beim zweiten Design-Ansatz ..... 36
Abbildung 14: Laden/Speichern - ObjectTeams-orientiertes Design, Ansatz 3 ..... 37
Abbildung 15: Erzeugung und Lifting eines StorageFormat-Objekts beim dritten Design-Ansatz ..... 37
Abbildung 16: StorageFormat-Hierarchie, ObjectTeams-orientiert ..... 38
Abbildung 17: Laden / Speichern – Standardformat, ObjectTeams-orientierte Struktur ..... 41
Abbildung 18: Storage - Externalized Roles : Quelltextfragment der Klasse StandardFigureSelection ..... 42
Abbildung 19: Objektorientierte Verwendung der Lade-Speicher-Funktionalität ..... 47
Abbildung 20: ObjectTeams-orientierte Verwendung der Lade-Speicher-Funktionalität, Hierarchie-Beziehungen ..... 48
Abbildung 21: ObjectTeams-orientierte Verwendung der Lade-Speicher-Funktionalität, erster Design-Ansatz ..... 50
Abbildung 22: ObjectTeams-orientierte Verwendung der Lade-Speicher-Funktionalität, zweiter Design-Ansatz ..... 51
Abbildung 23: Programmfunktion „Editorfenster öffnen“, objektorientierte Struktur ..... 62
Abbildung 24: Programmfunktion „Editorfenster öffnen“, objektorientiertes Verhalten ..... 63
Abbildung 25: Programmfunktion „Editorfenster öffnen“, objektorientierte Struktur und Verhalten, Ausschnitt ..... 64
Abbildung 26: StorageTeam - Verbindungsstelle zum Programm ..... 65
Abbildung 27: erstes Team-Handhabungs-Verfahren, Instanzenkonstellation ..... 68
Abbildung 28: erstes Team-Handhabungs-Verfahren, Implementierungs-Elemente ..... 68
Abbildung 29: zweites Team-Handhabungsverfahren, Instanzenkonstellation ..... 69
Abbildung 30: zweites Team-Handhabungs-Verfahren, Implementierungs-Elemente ..... 70
Abbildung 31: erweitertes zweites Team-Handhabungs-Verfahren, Implementierungs-Elemente ..... 72
Abbildung 32: Verwendung spezieller StorageTeams ..... 74

Abbildung 33: Beziehung zwischen den Klassen DrawApplication und StorageFormatManager, objektorientiert....76
Abbildung 34: Beziehung zwischen den Klassen DrawApplication und StorageFormatManager, ObjectTeams-orientiert....76
Abbildung 35: zweites Team-Handhabungsverfahren: Beziehung zwischen DrawApplication und StorageTeam und alternative StorageTeam-Struktur....77
Abbildung 36: Undo - objektorientierte Struktur....79
Abbildung 37: Undo - objektorientiertes Verhalten, Teil 1 - „Vorbereitung“ Ausführung einer Benutzeraktion mit Erzeugung eines Undoable-Objekts und Übergabe an UndoManager....81
Abbildung 38: Undo - objektorientiertes Verhalten, Teil 2 - „Rückgängigmachen“....82
Abbildung 39: Undo - ObjectTeams-orientierte Umstrukturierung, erste Etappe....87
Abbildung 40: Undo - zentrale Team-Instanz, Registrierung von Basisobjekten....93
Abbildung 41: Undo - mehrere Team-Instanzen, Registrierung von Handle-Basisobjekten....94
Abbildung 42: Undo - Vererbungshierarchien der gebundenen Rollenklassen....98
Abbildung 43: "callin without super call"-Problematik....100
Abbildung 44: Lösung für "callin without super call"-Problem....101
Abbildung 45: Undo - Struktur der Undoable-Hierarchie, inhomogen....103
Abbildung 46: Undo - Struktur der Undoable-Hierarchie, homogen....104

# Literaturverzeichnis

[BD99] Greg Butler, Pierre Denommee. Documenting Frameworks. In Fayad, Schmidt, Johnson (Hg.). Building Application Frameworks. Wiley & Sons, 1999, Seite 495-503

[BJ94] Kent Beck, Ralph Johnson. Patterns Generate Architectures. In Proceedings of the 8th European Conference on Object-Oriented Programming, Bologna, 1994

[BMMB99] Jan Bosch, Peter Molin, Michael Mattson, Per-Olof Bengtsson. Object-Oriented Frameworks - Problems & Experiences. In Fayad, Schmidt, Johnson (Hg.). Building Application Frameworks. Wiley & Sons, 1999, Seite 58

[Dij76] Edsger W. Dijkstra. A Discipline of Programming. Prentice-Hall, 1976

[Fow99] Martin Fowler. Refactoring – Improving the Design of Existing Code. Addison Wesley Longman, 1999

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995

[Her02a] Stephan Herrmann. Object Teams: Improving modularity for crosscutting collaborations. In Proc. Net Object Days 2002, Erfurt, 2002

[Her02b] Stephan Herrmann. Composable Design With UFA, Reviewed position paper at the Workshop on Aspect-Oriented Modeling with UML -- (1st International Conference on Aspect-Oriented Software Development), 2002

[Her03] Stephan Herrmann, ObjectTeams/Java Language Definition, http://www.objectteams.org (unter Docs → Language Definition), 2003

[HHG90] Richard Helm, Ian M.Holland, Dipayan Gangopadhyay. Contracts: Specifying behavioural compositions in object-oriented systems. In Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications, Ottawa, 1990, S.169-180

[JF88] Ralph Johnson, Brian Foote. Designing reusable classes. In Journal of Object-Oriented Programming, Vol 1, No.2, Juni 1988

[Joh92] Ralph Johnson, Documenting frameworks using patterns. Proceedings of OOPSLA 1992. New York: ACM/SIGPLAN, 1992, Seite 63-76

[LK95] R.Lajoie, R.K.Keller: Design and reuse in object-oriented frameworks: Patterns, contracts and motifs in concert. In V.S.Alagar und R.Missaoui (Hg.).Object-Oriented Technology for Database and Software Systems. World Scientific Publishing, Singapore, 1995, S.295-312

[Loh02] Daniel Lohmann. Multidimensionales Trennen der Belange im Softwareentwurf, Diplomarbeit, Universität Koblenz, 2002, Seite 1

[Mat96] Michael Mattson. Object-oriented Frameworks – A Survey of Methodological Issues. Licentiate Thesis, Department of Computer Science and Business Administration, University College of Karlskrona/Ronneby, 1996

[OT04] Object Teams home page. http://www.objectteams.org, 2004

[Par72] David L.Parnas. On the Criteria to be used in Decomposing Systems into Modules. In Communications of the ACM 15(12), 1972 , Seite 1053-1058

[SBF96] S.Sparks, K.Benner, C.Faris. Managing object-oriented framework reuse. IEEE Computer, Theme Issue on Managing Object-Oriented Software Development, M.E.Fayad, M. Cline (Hg.), 29(9), September 1996, Seite 52-61

[Vei02] Matthias Veit. Evaluierung modularer Softwareentwicklung mit "Object Teams" am Beispiel eines Projektmanagementsystems. Diplomarbeit, TU-Berlin, 2002

[VH03] Matthias Veit, Stephan Herrmann. Model-View-Controller and ObjectTeams: A Perfect Match of Paradigms. In Proc. 2nd International Conference on Aspect-Oriented Software Development, Boston, 2003