Ein Design-Tool zur Erstellung objektorientierter portabler Programmierschnittstellen

von
Elias Volanakis

Diplomarbeit

in Informatik

vorgelegt der
Fakultät für Informations- und Kognitionswissenschaften der
Eberhard-Karls-Universität Tübingen
im Juni 2004

angefertigt am
Arbeitsbereich für Symbolisches Rechnen
Professor Dr. Wolfgang Küchlin

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Tübingen, den 15. Juni 2004

Danksagung

Dr. Wolfgang Blochinger vom Arbeitsbereich für Symbolisches Rechnen möchte ich für die engagierte und hilfreiche Betreuung danken.

Diese Arbeit schließt mein Studium ab. Für ihre moralische, finanzielle und gelegentlich auch konkrete Unterstützung meines Studiums danke ich meinen Eltern herzlich.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

1.1 Motivation

Die Entwicklung von Software ist mit beträchtlichem Aufwand verbunden. Daraus ergibt sich der Wunsch portable Software zu erstellen, um diese für auf mehreren Systemen verwenden zu können.

Allerdings stellt die Erstellung portabler Software zusätzliche Anforderungen, die während des Softwareentwurfs berücksichtigt werden müssen:

- Systemabhängige Teile sollten klar vom Rest der Software getrennt sein, um den Aufwand bei einer Anpassung der Software an ein neues Zielsystem in Grenzen zu halten.
- Systemabhängige Teile sollten hinter einer einheitlichen Schnittstelle verborgen werden, damit sie ohne große Umstände ausgetauscht werden können.
- Systemnahe Ressourcen, werden häufig mit Hilfe prozeduraler Programmierschnittstellen verwaltet. Bei der Konzeption einer objektorientierten Software müssen diese Schnittstellen sinnvoll in das Gesamtsystem integriert werden.

Für diese Anforderungen stellt das *Wrapper Façade Entwurfsmuster* eine Lösung dar. Die vorliegende Arbeit befasst sich mit der softwaregestützten Entwicklung von Wrapper Façades und beschreibt die Konzeption und Implementierung einer graphischen Modellierungssoftware für diesen Zweck.

1.2 Aufbau der Arbeit

An dieser Stelle soll ein kurzer Überblick über den Aufbau und Inhalt der vorliegenden Arbeit gegeben werden.

Kapitel 2 führt das Wrapper Façade Entwurfsmuster ein und erläutert seine Vor- und Nachteile. Anschließend wird die Erstellung einer Wrapper Façade anhand einer schrittweisen Entwurfsanleitung beschrieben und die dabei zu treffenden Entscheidungen aufgezeigt. Abschließend werden vier Implementierungsverfahren für das Entwurfsmuster vorgestellt und verglichen.

Kapitel 3 bildet den Kern der Arbeit. Es führt die Anforderungen auf, die sich bei einer softwaregestützten Wrapper Façade Entwicklung stellen und beschreibt die Modellierungssoftware, die während der Diplomarbeit entwickelt wurde. Die Software enthält, neben der Modellierungskomponente, einen Codegenerator, der die erstellte Wrapper Façade in C++ Quellcode umwandelt, und ein Import-Tool, dass Funktions- und Variablendeklarationen aus existierendem Quellcode extrahiert und in das Modell einfügt. Die Verwendung der Software wird anhand eines Beispiels dargestellt. Anschließend wird diskutiert, wie die Software zur Unterstützung des Entwurfsprozesses beiträgt. Eine Beschreibung weiterer Forschungsmöglichkeiten, die sich aus dieser Arbeit ergeben, und eine Auflistung der Möglichkeiten zur Weiterentwicklung der Software schließt dieses Kapitel ab.

Kapitel 4 stellt ein umfangreiches Anwendungsbeispiel vor. Hierbei wurde mit Hilfe des Modellierungstools eine Wrapper Façade entwickelt, welche die Socket-API in einer objektorientierten Klassenbibliothek kapselt. Die Klassenbibliothek ist für drei Betriebsysteme (Windows XP, Solaris, Linux) implementiert worden. Sie kann zur Erstellung plattformunabhängiger Client- und Serverprogramme verwenden werden. Die damit erstellten Programme sind, nach einer Neukompilierung, ohne Änderungen auf allen drei Betriebsystemen lauffähig.

Kapitel 5 befasst sich mit den Design- und Implementierungsaspekten der entwickelten Software. Vorgestellt werden das Design der Modellierungskomponente nach dem *Model-View-Controller*-Prinzip und das Design des Codegenerators mit Hilfe des *Visitor*-Entwurfsmusters. Zuletzt wird das XML-basierte Datenformat beschrieben, welches zur Speicherung der Modelldaten verwendet wird.

Kapitel 6 schließt die Arbeit mit einer kurzen Zusammenfassung der Ergebnisse ab und enthält Referenzen auf verwandte Publikationen.

1.3 Begriffseinführung: Entwurfsmuster

Ein Entwurfsmuster (*Design Pattern*) ist eine allgemein verwendbare Problemlösung für ein wiederkehrendes Entwurfsproblem.

In der Informatik bezeichnet der Begriff Entwurfsmuster eine „Beschreibung kommunizierender Objekte und Klassen, die angepasst werden, um ein allgemeines Ent-

wurfsproblem in einem speziellen Kontext zu lösen“ [GHJV95]. Die Beschreibung eines Entwurfsmusters besteht aus folgenden Grundelementen (nach [GHJV95]):

- **Einem Namen**: Durch Benennung des Entwurfsmusters kann ein „Design-Vokabular“ von Entwurfsmustern geschaffen werden. Der Name kann eine kurze Beschreibung des Problems, der Lösung oder der Auswirkungen des Entwurfsmusters sein.
- **Einer Problembeschreibung:** Sie stellt ein Designproblem vor, erläutert den Problemkontext und erklärt wie das beschriebene Entwurfsmuster zur Lösung dieses Problems beitragen kann. Sie führt die Nebenbedingungen auf, die für den Einsatz des Entwurfsmusters erfüllt sein müssen, und gibt Beispiele für schlechte Lösungen des Entwurfsproblems, um Indikationen für den Einsatz des Entwurfsmusters aufzuzeigen.
- **Einer abstrakten Problemlösung:** Sie beschreibt die Bestandteile der Problemlösung, also die partizipierenden Klassen und Objekte, und deren Beziehungen untereinander. Sie erläutert die Verantwortlichkeiten der einzelnen Bestandteile und erklärt die Zusammenarbeit zwischen ihnen. Zur Illustration der Problemlösung wird häufig UML-Notation verwendet.
- **Einer Analyse der Konsequenzen:** Sie beschreibt welche Auswirkungen die Anwendung des Entwurfsmusters auf das Gesamtsystem hat und zeigt auf, welche Abwägungen gemacht werden können. Dadurch soll die Bewertung mehrerer Entwurfsalternativen erleichtert werden. Betrachtet werden z.B. die Auswirkungen des Entwurfsmusters auf die Flexibilität, Erweiterbarkeit oder Portabilität der Software, implementierungs- und sprachspezifische Besonderheiten oder Zeit vs. Platz trade-offs.

Vorraussetzungen nützlicher Entwurfsmuster

Bevor aus einer Problemlösung ein nützliches Entwurfsmuster entwickelt werden kann, sollten folgende Vorraussetzungen erfüllt sein:

1. Die Problemlösung muss bereits in mehreren Fällen angewandt worden sein (*known uses*).
2. Die Auswirkungen (*consequences*) der Problemlösung auf die verschiedenen Aspekte eines Softwaresystems, wie z.B. Flexibilität oder Performanz, sollten bekannt sein und die zu treffenden Abwägungen (*trade-offs*) dokumentiert sein.

Durch den ersten Punkt soll sichergestellt werden, dass die dargestellte Lösung eine gewissen Allgemeingültigkeit und Flexibilität aufweist. Durch den zweiten Punkt soll eine bewusste Entscheidung für oder gegen den Einsatz einer Problemlösung ermöglicht werden, unter Berücksichtigung der damit verbundenen Auswirkungen auf das Gesamtsystem.

Vorteile

Durch die Katalogisierung, Publikation und Verwendung von Entwurfsmustern ergeben sich folgende Vorteile ([GHJV95]):

- Die Software-Qualität wird erhöht, weil bewährte Problemlösungen wieder verwendet werden. Ein willkommener Nebeneffekt ist, dass die Wiederentwicklung schon bekannter Problemlösungen vermieden wird.
- Der Entwurfsvorgang wird vereinfacht: Durch die Dokumentation und Publikation von Entwurfsmustern, z.B. in [GHJV95] oder den Serien „*Pattern Languages of Program Design Vol. 1–4*“ und „*Pattern-Oriented Software Architecture Vol. 1–3*“, wird eine gemeinsame Wissensbasis geschaffen. Der Entwurf von Software wird somit erleichtert, weil der Softwarearchitekt auf einen „Werkzeugkasten“ von publizierten Entwurfsmustern zurückgreifen kann.
- Die Kommunikation und Dokumentation einer Problemlösung wird erleichtert: Die Benennung der Entwurfsmuster schafft ein einheitliches „Design-Vokabular“. Durch Benutzung dieses Vokabulars kann die Beschreibung der verwendeten Problemlösung vereinfacht und die Kommunikation beschleunigt werden.

Kapitel 2

Das Wrapper Façade Entwurfsmuster

In diesem Kapitel wird zunächst das Wrapper Façade Entwurfsmuster eingeführt und seine Vor- und Nachteile besprochen. Anschließend wird eine Entwurfsanleitung zur Erstellung von Wrapper Façades vorgestellt. Zuletzt werden vier verschiedene Implementierungsstrategien beschrieben und miteinander verglichen.

2.1 Überblick

Die Benutzung systemnaher Ressourcen, wie z.B. Dateien, Sockets oder Threads, erfolgt häufig über prozedurale, systemnahe APIs[1]. Die Systemnähe ermöglicht eine effiziente Nutzung dieser Ressourcen, bedingt aber auch einen niedrigen Abstraktionsgrad. Dies kann wiederum zu Portabilitätsproblemen und gesteigerter Komplexität der Software führen.

Motiviert wird die Benutzung des Wrapper Façade Entwurfsmusters durch den Wunsch eine portable, robuste und performante Zugriffsmöglichkeit auf systemnahe, prozedurale APIs anzubieten.

Aufgabe des Entwurfsmusters ist es, systemnahe APIs (a) zu kapseln und (b) statt dessen eine objektorientierte Schnittstelle anzubieten, die einen höheren Abstraktionsgrad aufweist [Sch99].

Das Entwurfsmuster besteht aus folgenden Teilen [SSRB00]:

- *Funktionen*: Eine von der Zielplattform abhängige Menge von Funktionen der zu kapselnden API und die dazugehörigen Datenstrukturen.
- *Wrapper Façade*: Eine oder mehrere Klassen, die für die Kapselung der Funktionen und Datenstrukturen verantwortlich sind. Klassen-Methoden werden auf

[1] API: *Application Programming Interface*, Programmierschnittstelle

Funktionen abgebildet. Die zu den Funktionen gehörenden Daten werden als Klassen-Attribute verwaltet.

Abbildung 2.1 verdeutlicht die Beziehung zwischen den Funktionen und der Wrapper Façade.

2.2 Vorteile

Durch die Anwendung des Entwurfsmusters werden folgende Vorteile angestrebt [SSRB00]:

1. Kompakterer Code und verbesserte Robustheit.

   Von der Wrapper Façade wird eine objektorientierte Schnittstelle mit höherem Abstraktionsgrad angeboten, die anstatt der ursprünglichen API benutzt werden kann. Der hohe Abstraktionsgrad ermöglicht es, mehrere Funktionsaufrufe der ursprünglichen, flachen API durch wenige Methode-Aufrufe der objektorientierten Schnittstelle zu ersetzen.

   Die Ausnutzung objektorientierter Mechanismen wie Konstruktoren, Destruktoren, *Exceptions* oder automatischer Heap-Verwaltung (*garbage collection*) erlaubt es, eine objektorientierte Schnittstelle zu erstellen, die robuster ist als die ursprüngliche, ungekapselte API. Auch das Zusammenfassen mehrerer API-Aufrufe unter einer Klassen-Methode trägt zur Steigerung der Robustheit bei.

   Dadurch soll der Quelltext insgesamt verständlicher, fehlerresistenter und wartungsfreundlicher werden, als es bei einer direkten Nutzung der flachen API möglich wäre.

2. Verbesserte Modularität.

   Die Modularität wird auf zwei Arten verbessert: Einerseits durch die Gruppierung der lose zusammenhängenden Funktionen der flachen API in eng zusammenhängende Komponenten („cohesive components“ [SSRB00]) in Form von Klassen, andererseits durch die geeignete Organisation dieser Klassen durch Ausnutzung der objektorientierten Sprachmöglichkeiten, wie z.B. Vererbung und abstrakte Klassen.

   Durch die Erstellung eng zusammenhängender Komponenten soll die Wrapper Façade verständlicher und leichter benutzbar werden, als die ursprüngliche API. Die Modularisierung hat außerdem positive Auswirkungen auf die Erweiterbarkeit und Wiederverwendbarkeit des Quellcodes.

3. Verbesserte Portabilität.

   Die Konzentration der systemnahen API-Aufrufe an wenigen Stellen (bestimmte Klassen der Wrapper Façade) erhöht die Portabilität. Verschiedene Mechanismen von objektorientierten Sprachen, wie z.B. virtuelle Funktionen, Tem-

plates oder auch bedingte Kompilierung[2], ermöglichen das Austauschen einer Klassenimplementierung bei gleich bleibender Klassenschnittstelle. Bei richtiger Anwendung dieser Sprachmechanismen kann systemspezifischer Code zur Übersetzungszeit ausgetauscht werden, ohne das Anpassungen in höheren Abstraktionsschichten notwendig werden.

Dadurch werden die notwendigen Modifikationen bei einer Portierung bzgl. des Umfangs und der Lokalität eingeschränkt und eine Übertragung auf andere Zielplattformen erleichtert.

2.3 Nachteile

Die Verwendung des Entwurfsmusters bringt folgende Nachteile mit sich [SSRB00]:

1. Einschränkungen im Funktionsumfang der gekapselten API.

   Die Kapselung der API kann zu reduziertem Funktionsumfang führen, denn evtl. stellt die Wrapper Façade nicht alle Möglichkeiten der ungekapselten API zur Verfügung. In besonderen Fällen kann es daher sinnvoll sein, den direkten Zugriff auf manche Implementierungsdetails zu ermöglichen (siehe 2.5, Punkt 3).

2. Effizienzeinbußen.

   Mit der Kapselung der flachen API durch die Wrapper Façade wird eine zusätzliche Abstraktionsschicht eingeführt. Dies kann, je nach verwendetem Implementierungsverfahren, zu Effizienzeinbußen führen. Die Unterschiede diesbezüglich werden in Abschnitt 2.6 behandelt. Die Verwendung bestimmter Sprachmechanismen, wie z.B. die Technik des *Inlining* [Str97, 7.1.1], kann zur Reduzierung der Effizienzeinbußen beitragen.

3. Schwierigkeiten durch Sprach- oder Compilerbeschränkungen.

   Darunter zu verstehen sind Schwierigkeiten, die bei der Integration der systemnahen API-Aufrufe in die objektorientierte Implementierungssprache des Wrapper Façade auftauchen. Dieser Punkt sollte im Vorfeld berücksichtigt werden, denn der Integrationsaufwand einer prozeduralen in eine objektorientierte Sprache kann je nach verwendeter Sprachkombination sehr unterschiedlich ausfallen. Relativ einfach ist zum Beispiel die Einbindung von C APIs in C++ oder in Java.

2.4 Abstraktions- und Anpassungsschicht

Die Modularität und Portabilität des Entwurfsmusters wird erreicht, indem die von der Wrapper Façade exportierte Schnittstelle gleich bleibt, die jeweilige Abbildung auf

[2] Wobei letztere nicht von der Sprache, sondern von der Übersetzersoftware abhängt.

systemnahe Funktionen jedoch anpassbar ist.

Ausgehend von dieser Beobachtung kann eine schichtenbasierte Darstellung des Wrapper Façade Entwurfsmusters eingeführt werden [Blo02, S.89]. Diese besteht aus einer *Abstraktionsschicht* und einer *Anpassungsschicht*. Programmiertechnisch können beide Schichten auch als Einheit realisiert werden [Blo02]. Die Beziehung zwischen den beiden Schichten wird in Abbildung 2.2 dargestellt.

Abstraktionsschicht

Die Abstraktionsschicht enthält eine oder mehrere Klassen. Diese übernehmen zwei Rollen:

1. Die Kapselung von systemspezifischen Eigenheiten.

   Dazu ist eine Kapselung systemspezifischer Datentypen, Funktionsaufrufe und Konstanten notwendig: Es werden abstrakte Datentypen (ADTs) statt systemspezifischer Datentypen eingeführt. Variationen in den Funktionsaufrufen werden hinter Klassen-Methoden gekapselt. Für Konstanten werden statische Klassen-Variablen definiert. Ermöglicht werden soll dadurch das Anbieten einer einheitlichen Schnittstelle auf allen Zielsystemen.

2. Die Erstellung einer geeigneten objektorientierten Schnittstelle.

   Diese soll einheitlich für alle Zielsysteme ausfallen und einen höheren Abstraktionsgrad im Vergleich zur prozeduralen API aufweisen, um dadurch leichter verwendbar zu sein.

Anpassungsschicht

Die Anpassungsschicht enthält systemspezifische Ausprägungen derjenigen Klassen, die eine Kapselung von systemspezifischen Eigenheiten übernommen haben. Jede Ausprägung dieser Klassen ersetzt die abstrakten Datentypen durch konkrete Datentypen, enthält geeignete Implementierungen der Klassen-Methoden und bildet Konstanten auf statische Klassen-Variablen ab.

Diskussion

Die Abstraktionsschicht zielt auf die Vereinheitlichung der systembedingten Unterschiede ab. Dies ist jedoch noch nicht hinreichend um Portabilität herzustellen, denn es bedarf eines Mechanismus, der den transparenten Austausch der systemabhängigen Programmteile ermöglicht.

Um auf jeder Zielplattform eine andere Ausprägung der Anpassungsschicht verwenden zu können, kann man sich unterschiedlicher Verfahren bedienen, die in Abschnitt 2.6 vorgestellt werden.

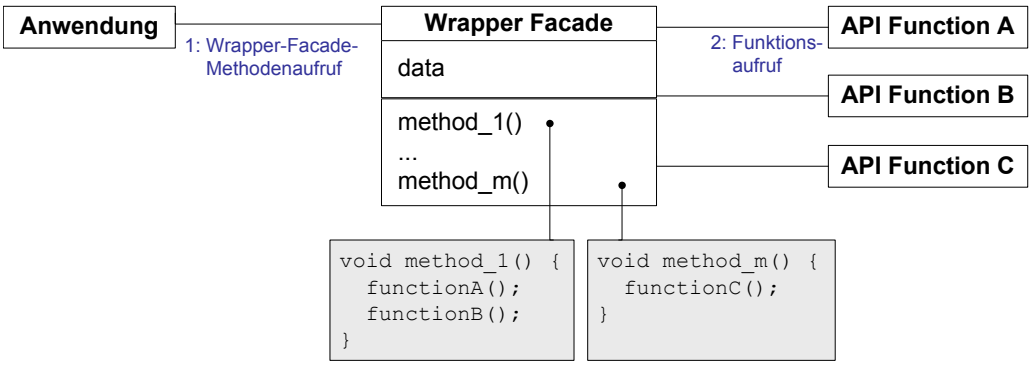

Abbildung 2.1: Das Wrapper Façade Entwurfsmuster. Abbildung nach [SSRB00].

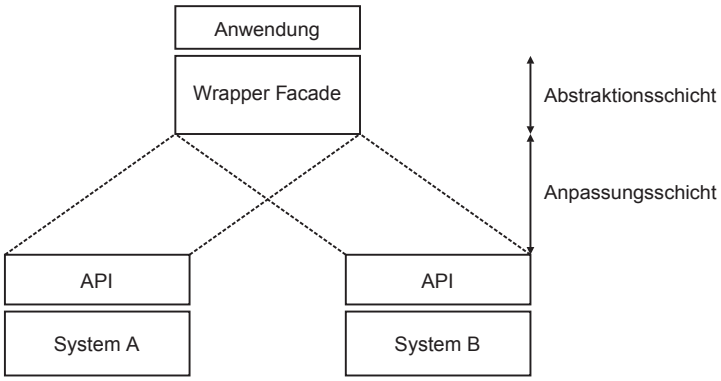

Abbildung 2.2: Abstraktions- und Anpassungsschicht beim Wrapper Façade Entwurfsmuster [Blo02].

2.5 Entwurfsanleitung

Bis jetzt unbeantwortet blieb die Frage, wie vorzugehen ist, wenn aus einer flachen API eine Wrapper Façade erstellt werden soll.

In [SSRB00] und [Sch99] ist eine aus fünf Schritten bestehende Entwurfsanleitung enthalten. Diese beschreibt eine allgemeine Vorgehensweise, die bei der Erstellung von Wrapper Façades angewandt werden kann: Angefangen mit der Betrachtung der flachen API, werden zusammenhängende Funktionen gruppiert und aus den Gruppen entsprechende Wrapper Façade Klassen erstellt.

Es folgt eine genauere Beschreibung der einzelnen Schritte:

1. Identifikation zusammenhängender API-Funktionen.

   In der Regel enthalten systemnahe APIs bestimmte konzeptuelle Abstraktionen, wie z.B. Dateien, Sockets oder Threads, in Form von Datenstrukturen und eine Reihe von Funktionen, um diese zu manipulieren. In diesem Schritt werden die zu kapselnden Datenstrukturen und dazugehörige Funktionsgruppen identifiziert und ausgewählt.

2. Gruppierung von zusammenhängenden Funktionen zu Wrapper Façade Klassen und Methoden.

   Dieser Schritt beschäftigt sich mit der Erstellung der Wrapper Façade. Sie kann aus einer oder mehreren Klassen bestehen. Diese Klassen sollen eine zusätzliche Abstraktionsebene einführen, die den Entwickler von systemspezifischen Datenstrukturen und Funktionssignaturen sowie anderen Implementierungsdetails abschirmt.

   Die Erstellung der Wrapper Façade kann wiederum in fünf Unterschritte eingeteilt werden:

   2.1. Erstellung zusammenhängender Wrapper Façade Klassen.
   Für die zuvor identifizierten Datenabstraktionen und Funktionsgruppen werden nun eine oder mehrere Wrapper Façade Klassen erstellt. Dabei werden folgende Leitlinien angewandt:

   - Es wird versucht zusammenhängende Funktionen und Daten in einzelnen Klassen zu gruppieren, welche möglichst unabhängig von einander sind.
   - Es wird versucht gleich bleibende und variable Aspekte in den gekapselten Funktionen und Daten zu erkennen und variable Teile hinter einer einheitlichen Schnittstelle zu verstecken. Gleich bleibend sind meistens konzeptuelle Abstraktionen, wie z.B. Dateien oder Threads, und variabel in der Regel deren Implementierung.

2.2. Zusammenfassung einzelner API-Funktionen in einer Methode.

Dies stellt sicher, dass zusammenhängende API-Funktionen in der richtigen Reihenfolge aufgerufen werden und kann die Benutzbarkeit der Wrapper Façade erhöhen.

2.3. Automatisierung von Erstellungs- und Freigabeaufrufen (falls möglich).

Der Zugriff auf systemnahe Ressourcen, z.B. Dateien oder Speicher, erfordert in der Regel die Nutzung von entsprechenden Erstellungs- bzw. Freigabeaufrufen einer systemnahen API. Durch die Nutzung von Konstruktoren und Destruktoren[3] zur Automatisierung solcher Aufrufe kann sichergestellt werden, dass solche Operationen immer ausgeführt werden. Damit kann die Robustheit der Wrapper Façade erhöht werden.

2.4. Festlegung des Indirektionsausmaßes.

Falls einige Wrapper Façade Methoden nur einen direkten API-Aufruf enthalten, kann unter C++ die Technik des *Inlining* benutzt werden, um Effizienzeinbuße zu vermeiden [Str97, 7.1.1].

2.5. Auswahl eines Implementierungsverfahrens für die Anpassungsschicht.

Plattformspezifische Variationen sollen durch die Wrapper Façade minimiert werden. Zu diesem Zweck wird durch die Wrapper Façade eine einheitliche Schnittstelle angeboten und systemspezifische Variationen werden hinter unterschiedlichen Implementierungen dieser Schnittstelle verborgen (in der Anpassungsschicht, vgl. 2.4).

Für die Erstellung der Anpassungsschicht wird in diesem Schritt ein geeignetes Implementierungsverfahren ausgewählt. In der Literatur wurden vier Verfahren vorgestellt, die sich bezüglich ihrer Eigenschaften unterscheiden und in Abschnitt 2.6 behandelt werden.

3. Ermöglichung eines kontrollierten Zugriff auf Implementierungsinterna (falls sinnvoll).

Die Wrapper Façade stellt u.U. nicht den gesamten Funktionsumfang der ungekapselten API zur Verfügung. Dies kann Absichtlich erfolgen, um das Aufrufen fehleranfälliger oder inkompatibler API-Funktionen zu verhindern, oder um nicht benötigte API-Funktionen aus der Wrapper Façade herauszuhalten.

Um dem Fall vorzubeugen, dass gerade die nicht verfügbaren API-Funktionen aus einem unvorhersehbaren Grund benötigt werden, kann man abwägen ob man den Zugriff auf Implementierungsinterna ermöglichen will, die es erlauben würden diese Einschränkungen zu umgehen. Dies sollte gut überlegt erfolgen, weil dadurch Portabilitätsprobleme entstehen können oder Klasseninvarianten verletzt werden können.

[3] Zumindest unter C++. Java hat keinen mit Destruktoren vergleichbaren Mechanismus [Blo01, Item 6].

4. Auswahl eines Fehlerbehandlungsmechanismus.

   Nicht-objektorientierte APIs zeigen Fehler in der Regel durch bestimmte Rückgabewerte bzw. das Setzen von globalen Variablen an[4]. Objektorientierte Sprachen bieten mit dem sog. Ausnahmemechanismus (*Exceptions*) eine modernere Alternative.

   Für die Benutzung von Ausnahmen spricht die Möglichkeit zur Hierarchiebildung, die klare Entkopplung von Anwendungs- und Fehlerbehandlungscode und die Überprüfbarkeit zur Übersetzungszeit[5].

   Einige Nachteile sind jedoch damit verbunden: Die Freigabe von systemnahen Ressourcen kann verkompliziert werden, weil durch das Auftreten von Ausnahmen nun mehrere Austrittspunkte aus einer Methode existieren. Zu klären ist auch, ob der Ausnahmemechanismus von der benutzten Sprache auf allen Zielplattformen auf ähnliche Art und Weise unterstützt wird[6].

5. Erstellung weiterer Klassen, welche auf die Wrapper Façade aufbauen (optional).

   Durch die objektorientierte Kapselung der systemnahen API können sich neue Nutzungsmöglichkeiten ergeben, die vorher nicht möglich oder ersichtlich waren. Die Wrapper Façade kann z.B. von anderen Entwurfsmustern benutzt werden. Bei entsprechendem Bedarf werden in diesem Schritt weitere Klassen erstellt, die auf die Wrapper Façade aufbauen.

Aus den begleitenden Codebeispielen in [SSRB00] ist nicht ersichtlich, wie konkrete Implementierungsverfahren für die Anpassungsschicht angewandt werden können (Abschnitt 2.6).

2.6 Implementierungsverfahren für die Anpassungsschicht

Durch den Einsatz des Entwurfsmusters wird Quellcode-Portabilität[7] angestrebt. Im folgenden werden vier Verfahren betrachtet, die eine Veränderung der Anpassungsschicht zur Übersetzungszeit ermöglichen. Die ersten beiden Verfahren wurden in [SSRB00] vorgestellt, die anderen beiden in [Blo02], [BK03].

[4] Zum Beispiel `errno` in C.

[5] Wird eine Klassen-Methode aufgerufen, bei der eine Ausnahme auftreten kann, muss der Programmierer die Ausnahme auffangen oder an die nächste Stufe in der Aufrufhierarchie weitergeben. Falls dies versäumt wird, tritt zur Übersetzungszeit eine entsprechende Fehlermeldung auf.

[6] Dies war z.B. bei C++ in der Vergangenheit nicht immer der Fall.

[7] Quellcode-Portabilität: Durch Neukompilierung des Quellcodes kann eine ausführbare Datei für das Zielsystem erstellt werden (s.a. [Moo90]).

2.6.1 Verzeichnisbasiertes Verfahren

Dabei besteht die Anpassungsschicht aus unterschiedlichen Implementierungen der Abstraktionsschicht. Diese werden in getrennten Verzeichnishierarchien gehalten. Bei der Übersetzung wird durch Auswahl des passenden Verzeichnisses die gewünschte Implementierung eingebunden.

2.6.2 Präprozessorverfahren

Hierbei werden Präprozessordirektiven (`#ifdef`) benutzt. Die Präprozessordirektiven prüfen, ob bestimmte Präprozessorkostanten gesetzt sind und fügen, vor der Übersetzung, die geeigneten plattformspezifischen Codestücke ein. Das Setzen der Präprozessorkonstanten kann durch Nutzung von Konfigurationswerkzeugen, wie z.B. GNU `autoconf`, automatisiert werden, um die Erstellung einer systemspezifischen Anpassungsschicht zu erleichtern.

2.6.3 Abstract Factory Verfahren

Dieses Implementierungsverfahren basiert auf dem Abstract Factory Entwurfsmuster [GHJV95]. Die Abstraktionsschicht wird hierbei durch abstrakte Interfaces[8] realisiert. Die Anpassungsschicht besteht aus Klassen, die eine plattformspezifische Implementierung dieser Interfaces bereitstellen. Um eine Klasse der Anpassungsschicht zu erzeugen, wird für jede Plattform eine passende Factory-Klasse benötigt. Die Factory-Klassen sind ebenfalls unter einem abstrakten Interface vereint. Dieser Zusammenhang wird noch einmal in Abbildung 2.3 verdeutlicht.

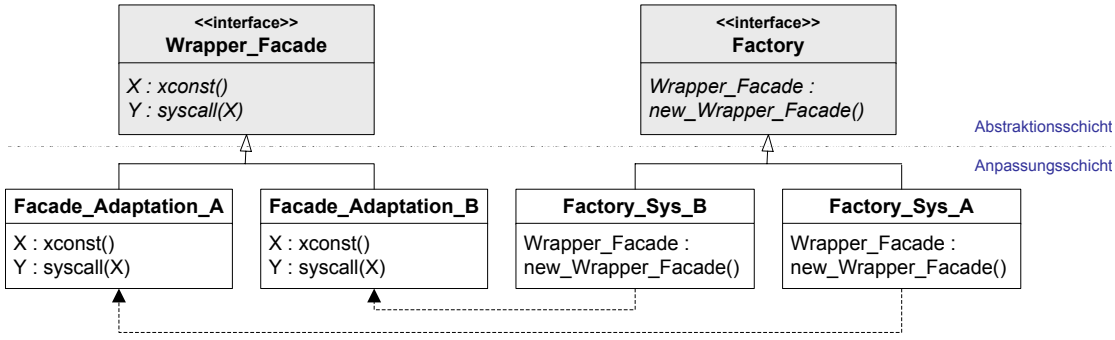

Abbildung 2.3: Verwendung des Abstract Factory Verfahrens zur Implementierung von Abstraktions- und Anpassungsschicht.

[8] Also Klassen, die nur aus virtuellen Funktionen bestehen [Str97, 12.2.6].

2.6.4 Template-Verfahren

Dieses Verfahren nutzt den Template-Mechanismus von C++ aus. Templates erlauben es Typen als Parameter für Klassen anzugeben [Str97, Ch. 13]. Dies wird ausgenutzt, um die Abstraktionsschicht mit der Anpassungsschicht zu parametrisieren und hat eine gewisse Ähnlichkeit zur *Traits* Technik [Mye95].

Die Abstraktionsschicht wird hierbei durch Template-Klassen realisiert. Die in einer Template-Klasse benutzten Datentypen, Konstanten und Methoden sind in systemspezifischen Klassen der Anpassungsschicht definiert. Diese werden als Typparameter an die Template-Klassen übergeben. Durch Änderung der Typparameter kann die Abstraktionsschicht ohne weitere Veränderungen auf das jeweilige Zielsystem angepasst werden.

Zur Auswahl des Typparameters kann eine Präprozessordirektive verwendet werden. Im Unterschied zum Präprozessorverfahren findet diese Unterscheidung jedoch nur an einer Stelle im Code statt.

Beispielimplementierungen von Wrapper Façades unter Benutzung des Template-Verfahrens sind in Abschnitt 3.4 und Kapitel 4 zu finden.

2.6.5 Vergleich der Implementierungsverfahren

Die Präprozessor-, Abstract Factory- und Template-Verfahren werden in [Blo02], [BK03] bezüglich ihrer Effizienz, Portabilität und Überprüfbarkeit verglichen.

Effizienz

Die im Entwurfsmuster vorhandene Abstraktions- und Anpassungsschicht soll nach Möglichkeit zu keinen Leistungseinbußen führen.

Ursache für Leistungseinbußen sind die zusätzlichen Indirektionsstufen, die durch die Abstraktions- und Anpassungsschichten eingeführt wurden. Für den Zugriff auf die gekapselte API sind nun mehrere Funktionsaufrufe durch die Abstraktions- und Anpassungsschicht notwendig. Dadurch können sich Effizienzeinbußen ergeben. In C++ kann dem durch Verwendung der *Inlining* Technik entgegengewirkt werden. Hierbei wird bei einem Aufruf einer entsprechend gekennzeichneten Funktion kein Unterprogrammaufruf vorgenommen. Statt dessen ersetzt der Compiler den Funktionsaufruf durch den Funktionsrumpf.

Bei den Präprozessor- und Template-Verfahren kann *Inlining* ohne Probleme angewandt werden. Die bei dem Abstract Factory Verfahren benötigten virtuellen Funktionen verhindern den Einsatz der *Inlining* Technik, denn hierbei wird der auszuführende

Funktionsrumpf erst zur Laufzeit bestimmt (*late binding*). Ferner wird durch die virtuellen Funktionen eine zusätzliche Indirektionsstufe eingeführt, denn beim Aufruf einer solchen Funktion wird über den entsprechenden Eintrag der *Virtual Function Table (vtbl)* der dazugehörige Funktionsrumpf ermittelt [Str97, 2.5.5].

Portabilität

Die Portierung auf eine neue Zielplattform soll möglichst einfach sein. Vorraussetzungen dafür sind: Erstens, eine hohe Lokalität systemspezifischer Eigenheiten, die durch Isolierung von plattformspezifischem Code in getrennten Übersetzungseinheiten erreicht werden kann. Zweitens, die Möglichkeit zur Redefinition von systemspezifischen Typen, Konstanten und Methoden, um diese auf einheitliche Gegenstücke in der Abstraktionsschicht abbilden zu können (vgl. 2.4). Damit soll der Umfang der notwendigen Anpassungen reduziert werden.

Das Präprozessorverfahren weist keine Lokalität systemspezifischer Eigenheiten auf. Bei der Portierung auf eine neue Zielplattform sind Anpassungen an mehreren Stellen notwendig. Ferner sind Anpassungen für unterschiedliche Systeme nicht durch getrennte Klassen isoliert. Eine übermäßige Verwendung der bedingten Übersetzung führt schnell zu Problemen bzgl. der Verständlichkeit und Wartbarkeit [SC92].

Bei dem Abstract Factory Verfahren muss während der Portierung eine plattformspezifische Implementierung der Abstraktionsschicht in entsprechenden Unterklassen erstellt werden (vgl. Abbildung 2.3). Die Lokalität ist hoch, da der Code für unterschiedliche Systeme in getrennten Klassen isoliert ist. Dafür gibt es Einschränkungen bei der Redefinition von Konstanten und Datentypen: Virtuelle Funktionen und deren Implementierung müssen dieselben Argument- und Rückgabetypen aufweisen [Str97, 12.2.6, §3]. Dadurch geht die Möglichkeit verloren, in zwei Implementierungen unterschiedliche Redefinitionen dieser Typen vorzunehmen. Davon betroffen ist auch die Kapselung von Konstanten, weil diese beim Abstract Factory Verfahren nur mit Hilfe virtueller Funktionen möglich ist. Zur Vereinheitlichung von Kostanten und Datentypen muss also ein anderer Mechanismus verwendet werden (z.B. bedingte Übersetzung).

Das Template-Verfahren bietet eine hohe Lokalität und erlaubt eine Redefinition von Datentypen, Konstanten und Methoden. Zur Portierung wird eine neue Implementierung der Anpassungsschicht als Template-Parameter an die Abstraktionsschicht übergeben.

Überprüfung auf Vollständigkeit der Implementierung

Es sollte zur Übersetzungszeit der Anpassungsschicht überprüfbar sein, ob diese eine adäquate Implementierung der Abstraktionsschicht auf dem aktuell verwendeten Ziel-

system darstellt.

Da bei dem Präprozessorverfahren der plattformspezifische Code nicht auf bestimmte Klassen beschränkt ist, sondern über die ganze Anpassungsschicht verteilt ist, kann während der Übersetzung nicht festgestellt werden, ob die Anpassungsschicht für das aktuelle Zielsystem vollständig implementiert ist.

Bei dem Abstract Factory Verfahren kann für zumindest für Methoden ein Vollständigkeitstest durchgeführt werden: Eine Instanz der Abstraktionsschicht lässt sich erst dann erzeugen, wenn für alle virtuellen Methoden eine Implementierung vorhanden ist. Dies wird während der Übersetzung überprüft.

Über die sog. explizite Instanziierung[9] ist ein Vollständigkeitstest für die Template-Klassen möglich. Fehlende Methoden-Implementierungen, Datentypen oder Konstanten werden somit im Template-Verfahren erkannt.

Fazit

Das Präprozessorverfahren ist zwar effizient, hat aber Nachteile bzgl. der Portabilität und Überprüfbarkeit, die aus der geringen Lokalität des systemspezifischen Codes resultieren. Das Abstract Factory Verfahren ist durch die virtuellen Funktionen und das fehlende *Inlining* weniger Effizient als die beiden anderen Techniken. In den Kategorien Portabilität und Überprüfbarkeit hat es Vorteile gegenüber dem Präprozessorverfahren, schneidet aber im Vergleich zum Template-Verfahren schlechter ab, weil es keine Redefinition von Typen und Konstanten ermöglicht und nur eine Vollständigkeitsprüfung für Methoden anbietet. Insgesamt gibt nur das Template-Verfahren in allen drei Vergleichskriterien ein gutes Bild ab.

[9] In C++ werden bei Template-Klassen nur die Klassen-Methoden erzeugt, die auch wirklich benutzt werden; dies kann mit der expliziten Instanziierung Umgangen werden [Str97, C13.9.1 ff].

Kapitel 3

Softwaregestützte Wrapper Façade Entwicklung

Dieses Kapitel beschäftigt sich mit der softwaregestützten Wrapper Façade Entwicklung. Es werden die Motivation für diesen Ansatz vorgestellt und die sich daraus ergebenden Anforderungen diskutiert. Anschließend wird ein Design-Tool zur Erstellung von C++ basierten Wrapper Façades vorgestellt und seine Verwendung an einem Anwendungsbeispiel demonstriert. Abschließend wird der Beitrag der Software besprochen und Weiterentwicklungsmöglichkeiten der Software aufgeführt.

3.1 Motivation und Anforderungen

Die treibende Idee für die vorliegende Arbeit war, eine geeignete Softwareunterstützung zur Erstellung von C++ basierten Wrapper Façades zu entwickeln.

Es stellte sich heraus, dass die Aufgaben, die bei der Erstellung einer Wrapper Façade zu bewältigen sind, nämlich (a) der Entwurf einer geeigneten objektorientierten Schnittstelle und (b) die Erstellung einer passenden Anpassungsschicht zur Kapselung von systemspezifischen Variationen, durch ein interaktives Design-Tool unterstützt werden können.

Aus diesem Ansatz heraus wurde eine Reihe von Anforderungen entwickelt, die unter den folgenden Punkten zusammengefasst werden können:

1. **Unterstützung der Wrapper Façade Entwurfsanleitung:**

   In Abschnitt 2.5 wurde eine Entwurfsanleitung vorgestellt, die aus fünf Schritten besteht. Die Schritte beschreiben eine Vorgehensweise, die bei dem Entwurf einer Wrapper Façade angewandt werden kann und sollten durch das Design-Tool unterstützt werden.

2. **Unterstützung bei der Erstellung der Abstraktionsschicht:**

   Die Abstraktionsschicht der Wrapper Façade besteht aus einer objektorientierten Schnittstelle, hinter der systembedingte Unterschiede verborgen werden.

   Der Entwurf einer Schnittstelle ist, neben der konkreten Implementierung, auch ein schöpferischer Prozess, bei dem es darum geht, die am besten geeignete Variante aus allen möglichen Entwürfen auszuwählen. Hierbei ist eine graphische Darstellung des Entwurfs vorteilhaft. Sie erleichtert die Kommunikation zwischen den Beteiligten, macht Unterschiede bildlich sichtbar und beschleunigt die Entscheidungsfindung. Deshalb ist es sinnvoll, die erstellte Wrapper Façade im Design-Tool graphisch darzustellen.

   Zusätzlich soll die dargestellte Schnittstelle möglichst einfach zu verändern sein, um dadurch die Erstellung verschiedener Entwurfsvarianten zu erleichtern.

   Selbstverständlich ist, dass das Design-Tool die Erstellung aller üblichen Elemente der Abstraktionsschicht unterstützen sollte. Dazu gehören: Klassen, abstrakte und konkrete Datentypen, Variablen, Konstanten und Methoden.

3. **Unterstützung bei der Erstellung der Anpassungsschicht:**

   Die Anpassungsschicht enthält systemspezifische Gegenstücke für die Elemente der Abstraktionsschicht. Bei der Erstellung der Anpassungsschicht sollte sichergestellt werden, dass diese die erforderlichen Gegenstücke enthält.

   Die Anpassungsschicht ist für folgende Aufgaben zuständig: die Ersetzung von abstrakten Datentypen durch konkrete Datentypen, das Abbilden von API-Konstanten auf statische Klassen-Variablen und das Kapseln von API-Funktionsaufrufen in Klassen-Funktionen. Diese Aufgaben sollen durch das Design-Tool unterstützt und erleichtert werden.

4. **Unterstützung bei der Erstellung einer Implementierung:**

   Das Design-Tool sollte in der Lage sein, aus den eingegeben Daten eine C++ basierte Implementierung der Abstraktions- und Anpassungsschichten zu erstellen. Diese sollte, soweit es die vorhandenen Informationen aus dem Design-Tool erlauben, vollständig sein. Falls manuelle Anpassungen im erstellten Code notwendig sind, sollten die zu ändernden Stellen klar gekennzeichnet sein.

   In Abschnitt 2.6 wurden vier Implementierungsverfahren vorgestellt und verglichen – weitere Varianten sind denkbar. Gemeinsam bei allen Verfahren ist, dass bei einer Änderung des Implementierungsverfahrens die von der Wrapper Façade definierte Schnittstelle gleich bleibt. Das Design-Tool sollte deshalb die Verwendung unterschiedlicher Implementierungsverfahren bei der Erstellung einer Implementierung erlauben bzw. die Integration neuer Verfahren erleichtern.

5. **Erkennung und Vermeidung von Falscheingaben:**

Durch fehlerhafte Benutzereingaben im Design-Tool können Fehler im anschließend generierten Quellcode entstehen, deren Behebung zeitaufwendig ist und deren Ursache vermeidbar wäre. Daher sollte das Design-Tool, soweit wie möglich, inkorrekte oder inkonsistente Eingaben erkennen und anzeigen.

Zur Erstellung einer korrekten Implementierung müssen zwei Arten von Bedingungen im Design-Tool überprüft werden:

- Bedingungen, die mit der verwendeten Implementierungssprache zusammenhängen. Deren Überprüfung ist eigentlich Aufgabe des Compilers, aber es ist benutzerfreundlicher, häufige und leicht zu erkennende Falscheingaben schon im Design-Tool abzufangen. Dazu gehören die Einhaltung sprachspezifischer Einschränkungen bei Benennung von Typbezeichnern oder die Vermeidung ungültiger Eingabekombinationen (z.B. können in C++ Klassen-Methoden nicht gleichzeitig *static* und *const* sein).
- Bedingungen, die mit der Aufgabe der Software zusammenhängen (*domain-specific constraints*). Hierbei muss die Einhaltung aufgabespezifischer Bedingungen in den Eingabedaten sichergestellt werden. Dazu gehört z.B., dass ein Element der Anpassungsschicht auch eine zugehörige Präprozessorkonstante benötigt, damit es bei Verwendung des Template- oder Präprozessorverfahrens (vgl. 2.6) auch ausgewählt werden kann.

6. **Integration in eine Entwicklungsumgebung:**

   Dadurch sollen Schnittstellendesign und Programmieraufgaben in derselben Anwendung stattfinden können: Die Betrachtung der prozeduralen API, die Erstellung der objektorientierten Schnittstelle und Vervollständigung der generierten Wrapper Façade sollten unter einer einheitlichen Oberfläche stattfinden können.

3.2 Das Wrapper Façade Eclipse-Plugin

Motiviert durch die oben genannten Anforderungen wurde eine Software zur Erstellung von Wrapper Façades entwickelt. Sie besteht aus drei Komponenten: Erstens, einem graphischen Editor zur Modellierung und Bearbeitung von Wrapper Façades. Zweitens, einem Import-Tool zur Extraktion von Methoden- und Variablendeklarationen aus existierenden C++ Dateien. Drittens, einem Code-Generator zur Erzeugung von Quellcode aus der modellierten Wrapper Façade. Die Komponenten sind in einem Plugin für die Eclipse-Entwicklungsumgebung gebündelt.

Eclipse [Fou04] ist eine frei verfügbare, open-source Entwicklungsumgebung für Java und andere Programmiersprachen. Gleichzeitig bietet Eclipse einen Rahmen (*Framework*) zur Erstellung und Integration komponentenbasierter Software-Tools an, die als Plugins in die Entwicklungsumgebung eingebunden werden. Ein *Plugin* stellt zusätzliche Funktionalität in Form von *Extensions* zur Verfügung. Solche *Extensions* können

z.B. neue Editoren, Sichten (*Views*) oder Assistenten (*Wizards*) sein, die an einem passenden *Extension Point* in die Entwicklungsumgebung integriert werden (vgl. [SDF+03]).

Es folgt eine genauere Vorstellung der einzelnen Komponenten.

3.2.1 Der graphische Editor

Der graphische Editor ist die zentrale Komponente des Plugins. Er ist für die Darstellung der zu bearbeitenden Wrapper Façade zuständig, ermöglicht eine interaktive Bearbeitung, übernimmt das Laden und Speichern der Daten und integriert weitere Komponenten des Plugins.

Der Editor wurde mit Hilfe des *Graphical Editing Frameworks* (GEF) implementiert, welches die Erstellung graphischer Editoren nach dem *Model-View-Controller*-Entwurfsmuster ermöglicht. Mehr dazu in Kapitel 5.

Abbildung 3.1 zeigt eine typische Instanz des Wrapper Façade Editors. Konzeptionell kann der Editor in vier Bereiche gegliedert werden: die Komponentenleiste, die Arbeitsfläche, kontextabhängige Befehlsmenüs und die Befehlsleiste. Diese Bereiche sind in der Abbildung durch entsprechende Ziffern markiert und werden nun vorgestellt:

1. Die Komponentenleiste (Bereich #1):

   Die Komponentenleiste enthält alle Elemente, die dem Diagramm hinzugefügt werden können, um eine Wrapper Façade zu modellieren. Dies sind: konkrete Klassen, Abstraktions-Klassen, Klassen-Variablen und -Funktionen, Datentypen und Adaptation-Elemente.

   Das Erstellen und Hinzufügen eines Elements geschieht durch Anklicken der gewünschten Zielklasse bzw. Zielstelle im Diagramm. Daraufhin erscheint ein Dialog, der die zur Erstellung benötigten Daten abfragt. Während der Eingabe erfolgt eine Überprüfung der Daten, um häufige Fehlerursachen zu vermeiden. Erkannte Fehler werden oben links im Dialog dargestellt und die Erstellung des Elementes bis zur Fehlerbehebung verhindert. Abbildung 3.2 zeigt ein derartiges Beispiel.

   Zusätzlich enthält die Komponentenleiste noch zwei Auswahlwerkzeuge, die zur Selektion von einem oder mehreren Diagrammelementen benutzt werden können (*Select*- bzw. *Marquee*-Werkzeug).

2. Die Arbeitsfläche (Bereich #2):

   In der Arbeitsfläche erfolgt die Darstellung und Bearbeitung der Diagrammelemente.

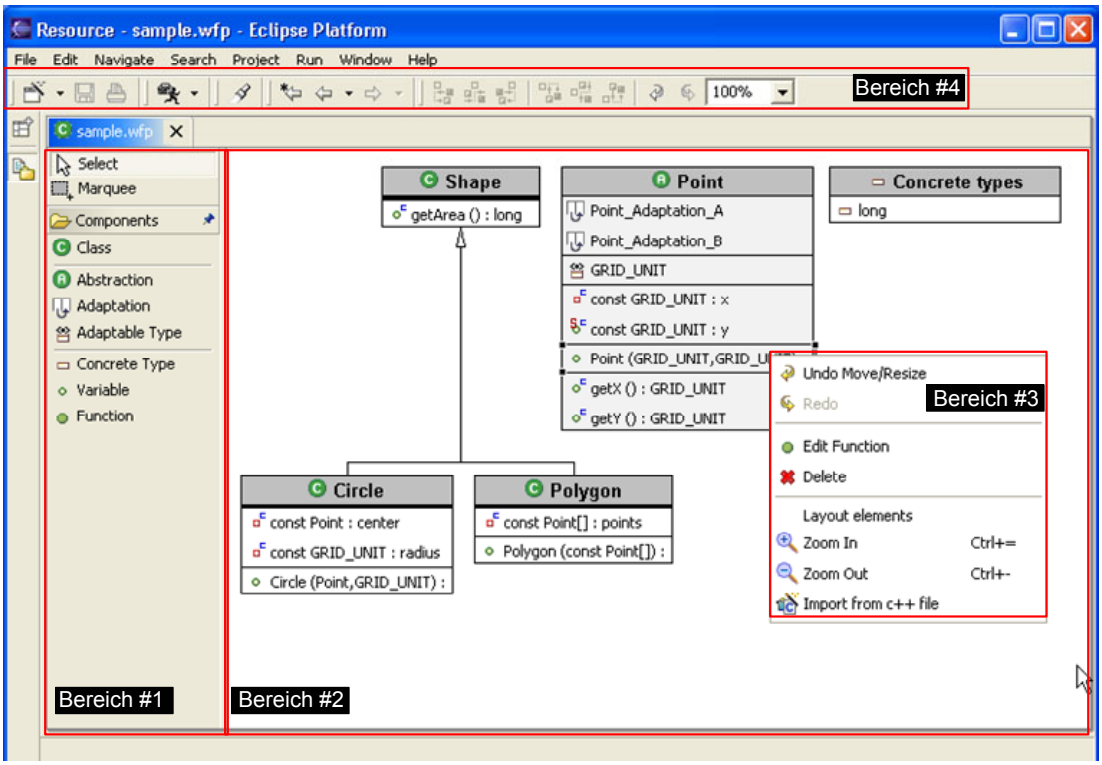

Abbildung 3.1: Der Wrapper Façade Editor.

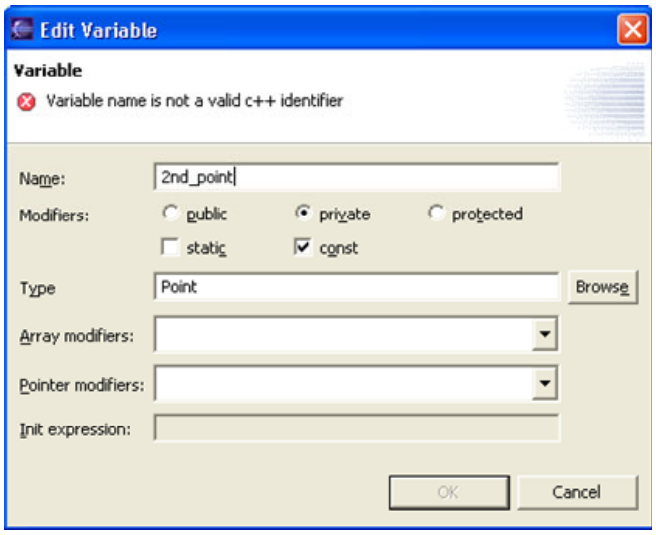

Abbildung 3.2: Erstellungsdialog für eine Klassenvariable.

Die dargestellten Elemente können ausgewählt und frei verschoben werden oder mit Hilfe des Layout-Befehls neu angeordnet werden. Der verwendete Layout-Algorithmus minimiert die Anzahl von Kantenkreuzungen und platziert abgeleitete Klassen unter der Vater-Klasse (Abb. 3.1).

Methoden und Variablen können über den *drag & drop* Mechanismus von einer Klasse in eine andere verschoben werden.

Die aktuell ausgewählten Diagrammelemente können mit Hilfe kontextabhängiger Befehle bearbeitet werden.

3. Kontextabhängige Befehlsmenüs (Bereich #3):

   Je nach Art und Anzahl der in der Arbeitsfläche ausgewählten Diagrammelemente, werden im kontextabhängigen Befehlsmenü unterschiedliche Aktionen ermöglicht:

   - Bearbeiten oder Entfernen der aktuell ausgewählten Elemente.
   - Rücknahme oder Wiederholung kürzlich ausgeführter Befehle (*Undo* bzw. *Redo*).
   - Ausrichten mehrerer Klassen zueinander (zentriert oder an einer ihrer vier Kanten).
   - Vergrößerte und verkleinerte Darstellung des Diagramms.
   - Aufrufen des Code-Generators für die dargestellte Wrapper Façade (3.2.3).
   - Aufrufen des Import-Tools (3.2.2).

4. Die Befehlsleiste (Bereich #4):

   Am oberen Rand des Editors befindet sich die Befehlsleiste (Abb. 3.1). Sie ist permanent sichtbar und enthält häufig benötigte Befehle, z.B. zum Speichern, Undo / Redo oder Verändern der Darstellung.

3.2.2 Das Import-Tool

Über das Import-Tool können Definitionen von Methoden und Variablen aus existierendem C++ Code extrahiert und in den graphischen Editor eingefügt werden. Dies soll die Erstellung einer objektorientierten Schnittstelle aus existierenden prozeduralen APIs erleichtern. Die Verwendung des Tools wird in Abbildungen 3.3 und 3.4 dargestellt.

Das Import-Tool besteht aus zwei Komponenten:

- Einer Import-Ansicht (*View*), zur hierarchischen Darstellung der beim Parsen des Quellcodes erkannten Codeelemente (Abb. 3.3). Aus dieser Darstellung können Funktionen, Methoden und Variablen ausgewählt und in den graphischen Editor importiert werden.

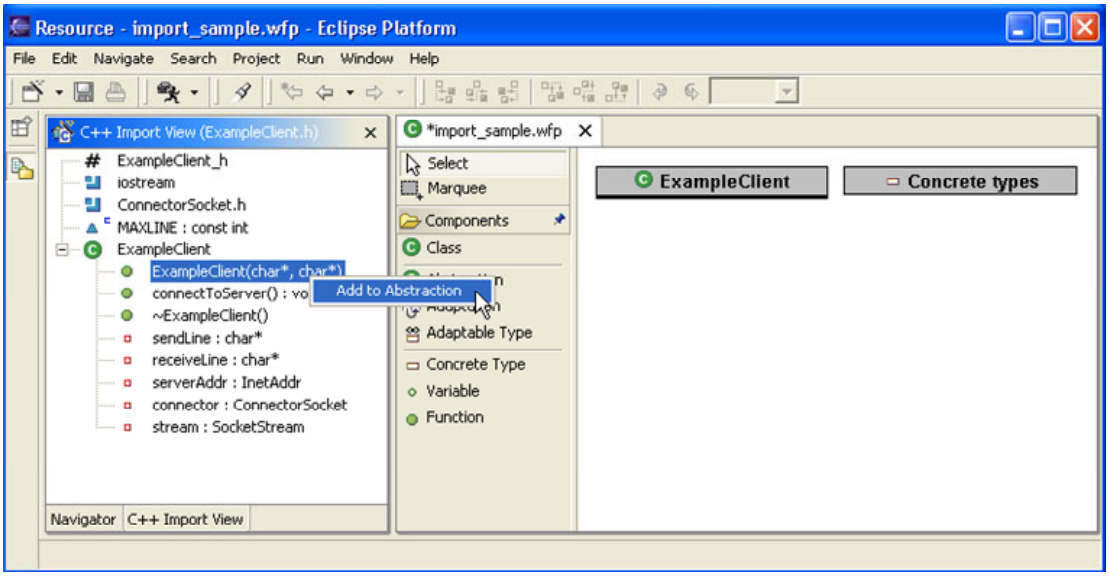

Abbildung 3.3: Import-Ansicht – Hierarchische Darstellung der geparsten C++ Datei.

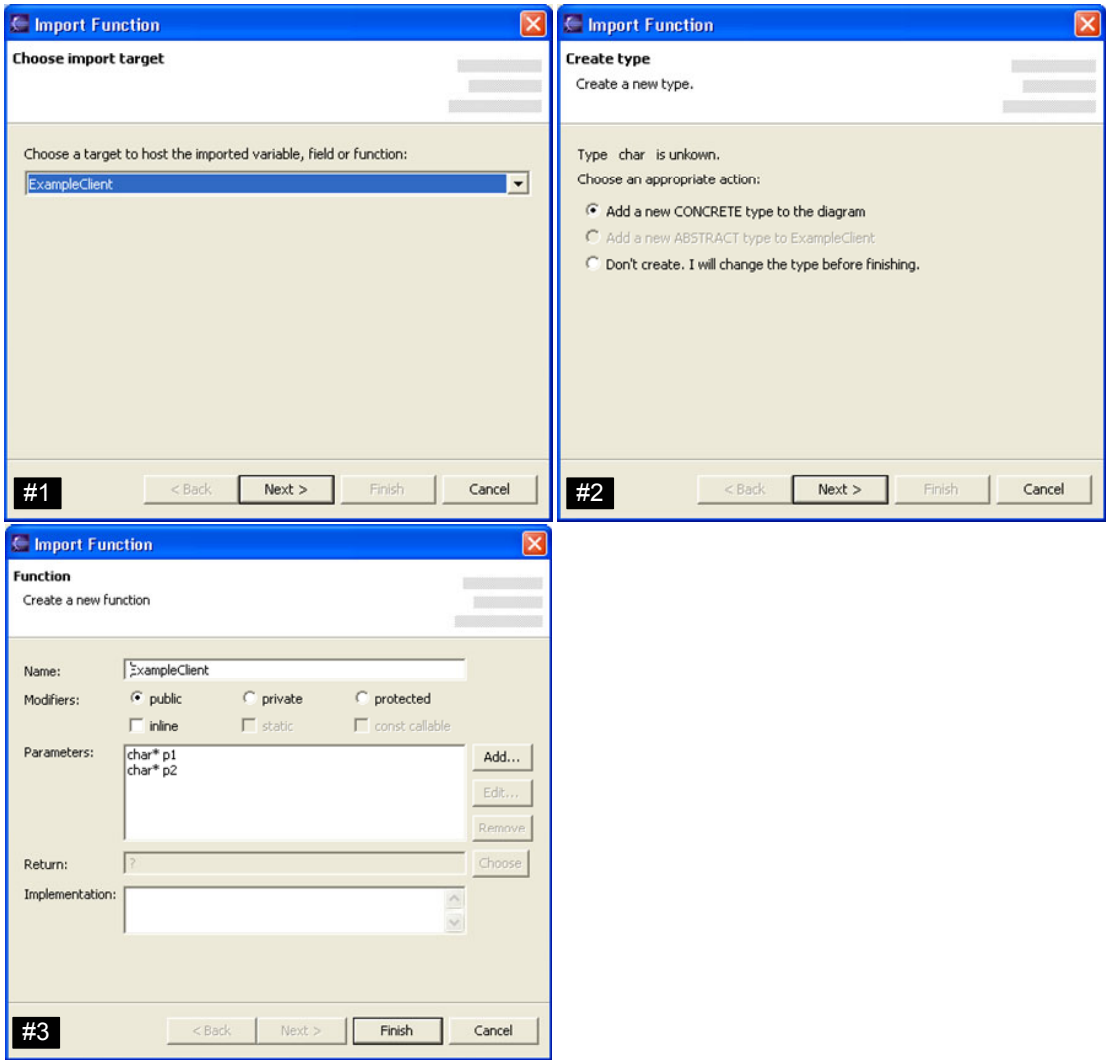

Abbildung 3.4: Import-Assistent – Einfügen einer Methodendefinition.

- Einem Import-Assistenten (*Wizard*), der die für den Einfügevorgang notwendigen Schritte durchführt: die Auswahl der Zielklasse, die Erzeugung der notwendigen Datentypen im Editor und das Einfügen der ausgewählten Variablen- bzw. Methodendefinition (Abb. 3.4).

Das Import-Tool nutzt den Parser des CDT-Plugins[1] zur Analyse des importierten Quellcodes und benutzt die Darstellungsmöglichkeiten des Plugins zur Erstellung der Import-Ansicht.

3.2.3 Der Codegenerator

Aus einem Wrapper Façade Modell des graphischen Editors kann der Codegenerator eine entsprechende Initialimplementierung des Entwurfsmusters erzeugen. Der momentan verwendete Codegenerator erzeugt eine Implementierung nach dem Template-Verfahren (2.6.4). Abbildungen des erzeugten Codes und der dazugehörigen Wrapper Façade Modelle finden sich bei den zwei Anwendungsbeispielen (Abschnitt 3.4 und Kapitel 4).

Die Initialimplementierung der Abstraktionsschicht besteht aus einer Header- und Cpp-Datei, für jede der im Design-Tool erstellten Klassen:

- Die Header-Datei enthält eine Deklaration der gleichnamigen Klasse und der darin vorhanden Klassenelemente. Zusätzlich erlaubt das Design-Tool die Eingabe einer Implementierung für Methoden bzw. einer Wertzuweisung für Variablen. Der Codegenerator integriert diese Daten in die Header-Datei. Benötigen bestimmte Datentypen aus der Header-Datei *#include*-Direktiven, werden diese vom Codegenerator eingefügt.

  Für *Abstraktions-Klassen* wird in der Header-Datei eine Template-Klasse deklariert. Außerdem werden Präprozessordirektiven hinzugefügt, die zur Übersetzungszeit eine systemspezifische Konfiguration der Template-Klasse erstellen und als Datentyp zur Verfügung stellen. Dies wird in Zeilen 26–35 des Codefragmentes 3.1 auf Seite 41 verdeutlicht.

- Die Cpp-Datei dient als Platzhalter für das spätere Hinzufügen von fehlenden Implementierungen der in der Header-Datei deklarierten Elemente. Sie enthält deshalb nur eine *#include*-Direktive für die gleichnamige Header-Datei.

Die Initialimplementierung der Anpassungsschicht besteht aus Header-Dateien die zur Parametrisierung der Abstraktions-Klassen verwendet werden. Dazu müssen die abstrakten Datentypen der Abstraktions-Klasse auf konkrete Datentypen abgebildet werden, systemspezifische Methodenimplementierungen erstellt werden (die aus der Abstraktionsschicht aufgerufen werden) und Wertzuweisungen für systemspezifische

[1] http://www.eclipse.org/cdt/

Konstanten vorgenommen werden. Die erstellte Header-Datei enthält entsprechende Deklarationen, die um systemspezifische Informationen ergänzt werden müssen. Die zu modifizierenden Stellen sind durch Kommentare markiert. Eine durch den Codegenerator erzeugte Header-Datei der Anpassungsschicht ist in Codefragment 3.3 auf Seite 43 abgebildet.

Um bei der Codegenerierung die Verwendung unterschiedlicher Implementierungsverfahren (2.6) zu ermöglichen, wurde besonders auf die Austauschbarkeit und leichte Nachimplementierbarkeit des Codegenerators geachtet. Das Design des Codegenerators wird in Abschnitt 5.2 genauer besprochen.

3.3 Vergleich mit einem UML-Editor

An dieser Stelle soll noch auf die Unterschiede des Wrapper Façade Plugins im Vergleich mit einem graphischen UML-Editor hingewiesen werden:

- UML ist prinzipiell eine sprachunabhängige Modellierungssprache. Der graphische Editor ist jedoch auf die Modellierung von C++ Programmen ausgerichtet. Er kann die eingegeben Daten auf Einhaltung von sprachspezifischen Einschränkungen überprüfen. Bei der Dateneingabe können sprachspezifische Eigenschaften angegeben werden (z.B. *inline* oder *const*).

- Ein UML-Editor mit C++ Unterstützung hat kein konzeptuelles Kenntnis des Wrapper Façade Entwurfsmusters. Daher ist die Modellierung der Abstraktions- und Anpassungsschicht problematisch.

- Ein UML-Editor mit C++ Unterstützung hat kein Kenntnis der verschiedenen Verfahren, die bei der Implementierung des Entwurfsmusters angewandt werden können. Die hier vorgestellte Codegenerierungsfunktion, erstellt eine Initialimplementierung der Abstraktions- und Anpassungsschicht nach dem Template-Verfahren (2.6.4).

3.4 Anwendungsbeispiel: Simple Wrapper Façade

3.4.1 Zweck und Umfang

Die Simple Wrapper Façade wurde als Demonstrationsbeispiel entworfen. Sie enthält alle wesentlichen Elemente einer Wrapper Façade in möglichst einfacher Form: zwei abstrakte Datentypen (*X* und *Y*), eine Konstante *pi* vom Typ *X* und eine Funktion *circle_area(X radius)* mit Rückgabetyp *Y*. Die Anpassungsschicht besteht aus konkreten Implementierungen für zwei fiktive Systeme *SYS_A* und *SYS_B*.

Der von der Wrapper Façade angebotene Funktionsumfang wurde möglichst einfach gehalten, um dem Beispielcharakter gerecht zu werden: Die Funktion *circle_area()* soll die Kreisfläche des übergebenen Radius berechnen. Hierbei wird *X* als Parametertyp und *Y* als Rückgabetyp verwendet. Die Konstante *pi* enthält eine Repräsentation der Kreiszahl, mit einer vom Zielsystem abhängigen Genauigkeit (*double* für *SYS_A* und *int* für *SYS_B*).

Wegen des geringen Umfangs eignet sich die Simple Wrapper Façade besonders, um an ihr die Funktionsweise des Modellierungstools zu demonstrieren und den damit erzeugten Code zu betrachten.

3.4.2 Vorgehen

Zunächst wird die Erstellung des Beispiels unter Verwendung des Modellierungstools in Abschnitt 3.4.3 beschrieben. Die fertige Wrapper Façade ist in Abbildung 3.11 dargestellt.

Nachdem die Erstellung der Simple Wrapper Façade abgeschlossen wurde, ist die Codegenerierungsfunktion des Design-Tools benutzt worden um eine Initialimplementierung zu erzeugen. Als Implementierungstechnik wurde das Template-Verfahren verwendet (2.6.4).

Der hierbei erzeugte Code besteht aus einer vollständigen Implementierung der Abstraktionsschicht und einer partiellen Implementierung der Anpassungsschicht. Letztere muss an den entsprechend gekennzeichneten Stellen um die systemspezifischen Eigenheiten des jeweiligen Zielsystems ergänzt werden.

Die erzeugte Initialimplementierung und die zur Vervollständigung notwendigen Modifikationen werden in den Abschnitten 3.4.4 und 3.4.5 vorgestellt. Die Verwendung der fertigen Wrapper Façade in Anwendungscode wird in Abschnitt 3.4.6 beschrieben.

3.4.3 Erstellungsanleitung

Dieser Abschnitt beschreibt die zur Erstellung der Simple Wrapper Façade notwendigen Aktionen, unter Verwendung des Wrapper Façade Eclipse-Plugins. Die folgende Anleitung setzt ein Vorhandensein der Eclipse-Entwicklungsumgebung samt installiertem Wrapper Façade Plugin vorraus.

Die einzelnen Schritte der Anleitung beschreiben das Anlegen eines neuen Wrapper Façade Diagramms, die Erzeugung der Abstraktions- und Anpassungsschicht, das Hinzufügen von Datentypen, Variablen und Funktionen und das automatische Erzeugen einer Initialimplementierung.

1. Anlegen eine neuen Wrapper Façade:

   Das Anlegen einer neuen Wrapper Façade geschieht mit Hilfe eines entsprechenden Assistenten. Dazu wird zunächst der Dateierstellungsdialog über *File, New, Other* aufgerufen. In der Kategorie *Examples, GEF* ist nun der Eintrag *Wrapper Façade Diagramm* auszuwählen. Über den Befehl *Next* kann nun der zweite Dialog-Schritt aufgerufen werden. Dort ist der Dateiname der zu erzeugenden Wrapper Façade anzugeben (*simple_facade.wfp*).

   Durch Ausführen des *Finish*-Befehls wird eine neue Wrapper Façade Datei angelegt und sogleich im Design-Tool geöffnet (Abb. 3.5).

2. Definition der Abstraktionsschicht:

   Die soeben erzeugte Wrapper Façade enthält zunächst nur die unter C++ verfügbaren Standardtypen (char, int, double, usw.). In diesem Schritt wird nun die Abstraktionsschicht erstellt.

   Die Abstraktionsschicht besteht in diesem Fall aus der Klasse *Wrapper_Facade*. Um diese anzulegen, wird die Komponente *Abstraction* aus der Komponentenleiste des Design-Tools ausgewählt (links in Abb. 3.5 zu sehen) und anschließend ein Linksklick auf eine beliebige leere Stelle im Arbeitsbereich des Design-Tools ausgeführt.

   Daraufhin erscheint ein Erstellungsdialog, in dem der Name der Abstraktions-Klasse (*Simple_Wrapper_Facade*) anzugeben ist. Zusätzlich ist im Feld „*Typedef to*“ eine Typbezeichnung anzugeben (*INTERFACE*). Diese wird später zur Vereinheitlichung der verschiedenen Konfigurationen der Abstraktions-Klasse verwendet. Nachdem mit *OK* die Erstellung bestätigt wird, erscheint die neue Klasse als Kästchen auf der Arbeitsfläche des Design-Tools (Abb. 3.6).

3. Definition der Anpassungsschicht:

   Die Anpassungsschicht soll Konfigurationen für zwei unterschiedliche Zielsysteme *SYS_A* und *SYS_B* enthalten. Um die Simple Wrapper Façade entsprechend zu parametrisieren, werden ihr zwei *Adaptation*-Elemente hinzugefügt.

   Dazu wird die Komponente *Adaptation* ausgewählt und der Zeiger über die Simple Wrapper Façade platziert. Ein Plus-Symbol an der unteren rechten Ecke des Zeigers zeigt an, dass die ausgewählte Komponente durch einen Linksklick zur Simple Wrapper Façade hinzugefügt werden kann.

   Im nun erscheinenden Dialog werden die notwendigen Informationen zur Erstellung des *Adaptation*-Elementes abgefragt. Anzugeben sind der Name des Elementes (*Facade_Adaptation_A*) und eine Präprozessorkonstante (*SYS_A*). Die Präprozessorkonstante wird verwendet, um beim Kompilieren die gewünschte Abstraktionsschicht auszuwählen (s. Seite 41).

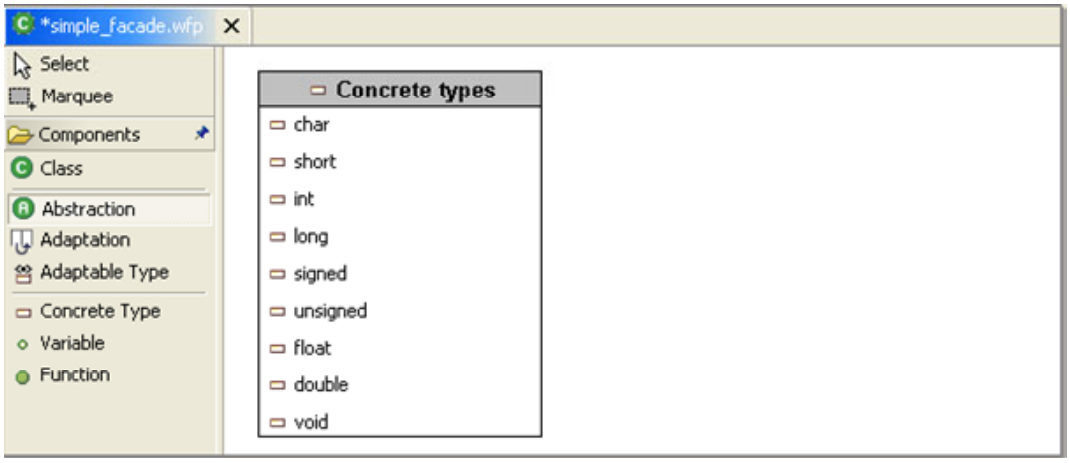

Abbildung 3.5: Ansicht der neu erzeugten Simple Wrapper Façade.

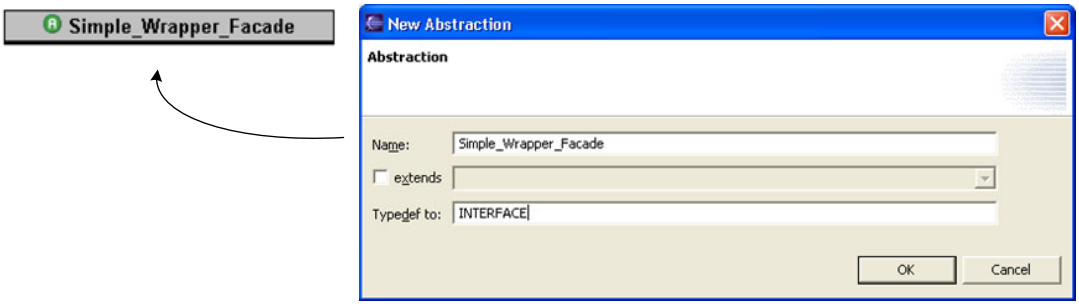

Abbildung 3.6: Abstraktions-Klasse – Erstellungsdialog und Darstellung im Editor.

Das zweite *Adaptation*-Element wird auf gleiche Art und Weise erstellt. Der anzugebende Name ist *Facade_Adaptation_B* und die Präprozessorkonstante lautet *SYS_B*.

Das Ergebnis ist in Abbildung 3.7 dargestellt.

4. Anlegen der abstrakten Datentypen:

   In diesem Schritt sollen zwei abstrakte Datentypen *X* und *Y* zur Wrapper Façade hinzugefügt werden. Dies geschieht durch den schon bekannten Mechanismus: Nach Auswahl des *Adaptable Type* Elementes werden die beiden Datentypen durch Linksklicken zur Wrapper Façade hinzugefügt. Der Erstellungsdialog fragt nach dem Namen der zu erzeugenden Datentypen. Der erste abstrakte Datentyp wird *X* genannt, der zweite *Y* (Abb. 3.8). Ersterer wird später als Variablen- und Parametertyp verwendet, letzterer als Rückgabedatentyp.

5. Deklaration von Konstanten:

   Um die Konstante *pi* deklarieren, wird der Wrapper Façade ein *Variablen*-Element hinzugefügt. Im Erstellungsdialog werden Name (*pi*) und Typ (*X*) der Variable angegeben und die gewünschten Eigenschaften ausgewählt (*public, static, const*).

   Zusätzlich kann eine Wertzuweisung für die Konstante direkt im Erstellungsdialog angegeben werden. Die Wertzuweisung wird dann bei der Codegenerierung berücksichtigt und zur Initialisierung der Konstante benutzt. Durch Angabe von *'= api_t::pi;'* wird die Konstante mit dem gleichnamigen Wert aus der Anpassungsschicht initialisiert.

   Der vollständig ausgefüllte Dialog ist in Abbildung 3.9 dargestellt.

6. Deklaration von Funktionen:

   Zuletzt muss noch die Funktion *circle_area(radius)* erstellt werden. Dazu wird ein *Function*-Element der Wrapper Façade hinzugefügt. Im Erstellungsdialog müssen zunächst der Name (*circle_area*) und Eigenschaften der Funktion (*public, inline, static*) angegeben werden.

   Anschließend wird der Funktion ein Parameter durch betätigen des *Add*-Kommandos hinzugefügt und es öffnet sich ein Dialog zur Angabe der Parameterdaten (Abb. 3.10). Die anzugebenden Parameterdaten sind Name (*radius*) und Typ (*X*).

   Zuletzt muss noch der Rückgabetyp der Funktion deklariert werden. Das *Choose* Kommando aktiviert den entsprechenden Dialog. Der anzugebende Rückgabetyp ist *Y*.

   Neben der Funktionsdeklaration kann zusätzlich noch eine Implementierung angegeben werden. Während der Codegenerierung werden angegebene Implementierungen berücksichtigt und an die entsprechende Stelle im Quellcode hinzu-

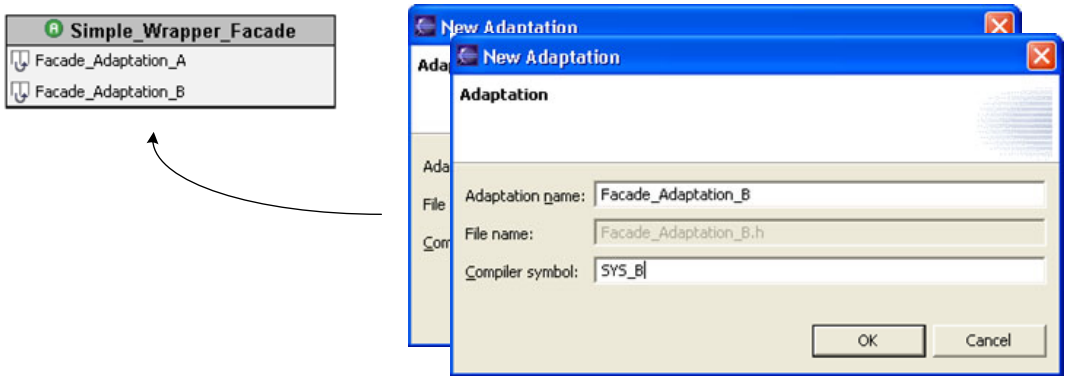

Abbildung 3.7: *Adaptation*-Elemente – Erstellungsdialog und Darstellung im Editor.

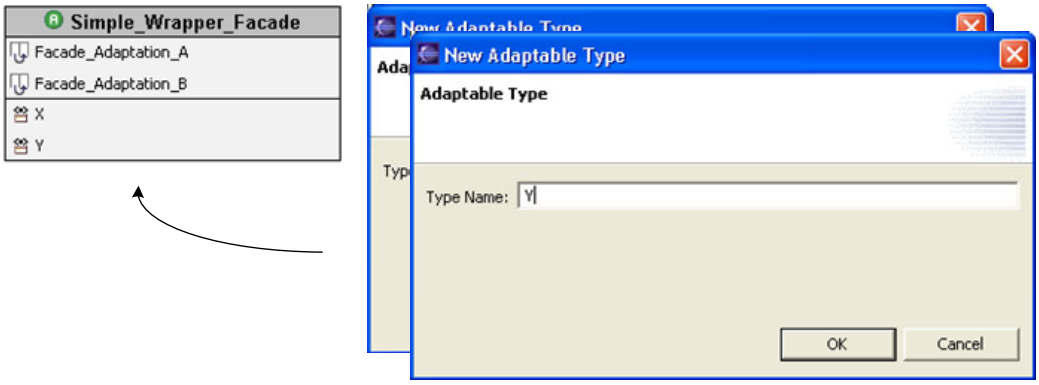

Abbildung 3.8: Abstrakte Datentypen – Erstellungsdialoge und Ergebnis.

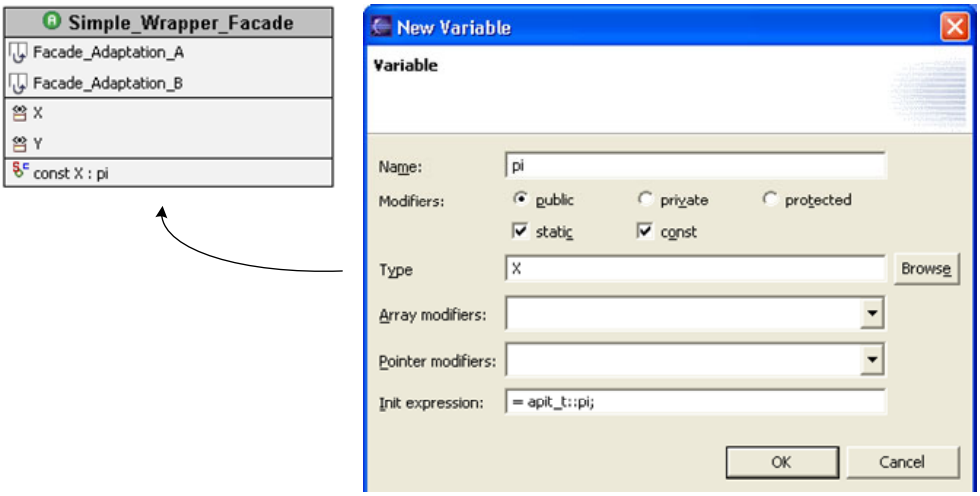

Abbildung 3.9: Ausgefüllter Erstellungsdialog für die Konstante *pi*.

gefügt. Funktionen aus der Anpassungsschicht werden in der Regel so implementiert, dass sie ein systemspezifisches Gegenstück in der Abstraktionsschicht aufrufen. Dazu wird für die Funktion *circle_area* folgende Implementierung angegeben '{ *return api_t::circle_area(radius);* }'

Die entsprechenden Dialogschritte sind in Abbildung 3.10 dargestellt.

7. Speichern des Beispiels:

   Nach den vorangegangenen Schritten ist die Erstellung der Simple Wrapper Façade abgeschlossen (vgl. Abb. 3.11) und kann nun über den Befehl *File, Save* gespeichert werden.

8. Codegenerierung:

   Zuletzt wird die Codegenerierungsfunktion des Design-Tools betätigt, um eine Initialimplementierung der eben erstellten Wrapper Façade zu erzeugen. Dazu wird in der Arbeitsfläche des Editors ein Rechtsklick ausgeführt und der Befehl *Generate Code* ausgewählt. Der dadurch erzeugte Quelltext wird in einem Verzeichnis mit Namen *codegen*, unterhalb des aktuellen Eclipse-Projektes gespeichert und wird in den nächsten Abschnitten genauer beschrieben.

3.4.4 Code der Abstraktionsschicht

Der generierte Code für die Abstraktionsschicht, besteht aus je einer Header- und Cpp-Datei für die Simple Wrapper Façade. Beide werden in diesem Abschnitt besprochen.

1. Header-Datei (Codefragment 3.1)

   Die generierte Header-Datei enthält die Klassendefinition der Simple Wrapper Façade.

   Die Klassendefinition verwendet das Template-Verfahren zur Parametrisierung der Anpassungsschicht (2.6.4): In den Zeilen 5–6 wird die Simple Wrapper Façade als Template-Klasse definiert. Die Template-Klasse verwendet den Typ-Parameter *api_t*, um auf die Anpassungsschicht zu zugreifen.

   Der Klassen-Rumpf enthält Deklarationen für die im Design-Tool erstellten Datentypen (Zeilen 8–9), Variablen (Zeile 10) und Funktionen (Zeile 12). Mit Hilfe des Typ-Parameters *api_t* werden die abstrakten Datentypen auf konkrete Datentypen in der Anpassungsschicht abgebildet.

   Die während der Erstellung des Beispiels (Abschnitt 3.4.3) angegebene Wertzuweisung der Konstante *pi* wurde berücksichtigt und entsprechender Code in Zeilen 22–24 erstellt. Die ebenfalls zuvor angegebene Implementierung der Funktion *circle_radius* wurde in Zeile 13 eingefügt. Falls diese Informationen nicht schon im Design-Tool angegeben werden, kann die Implementierung auch manuell in die zugehörige Cpp-Datei eingetragen werden.

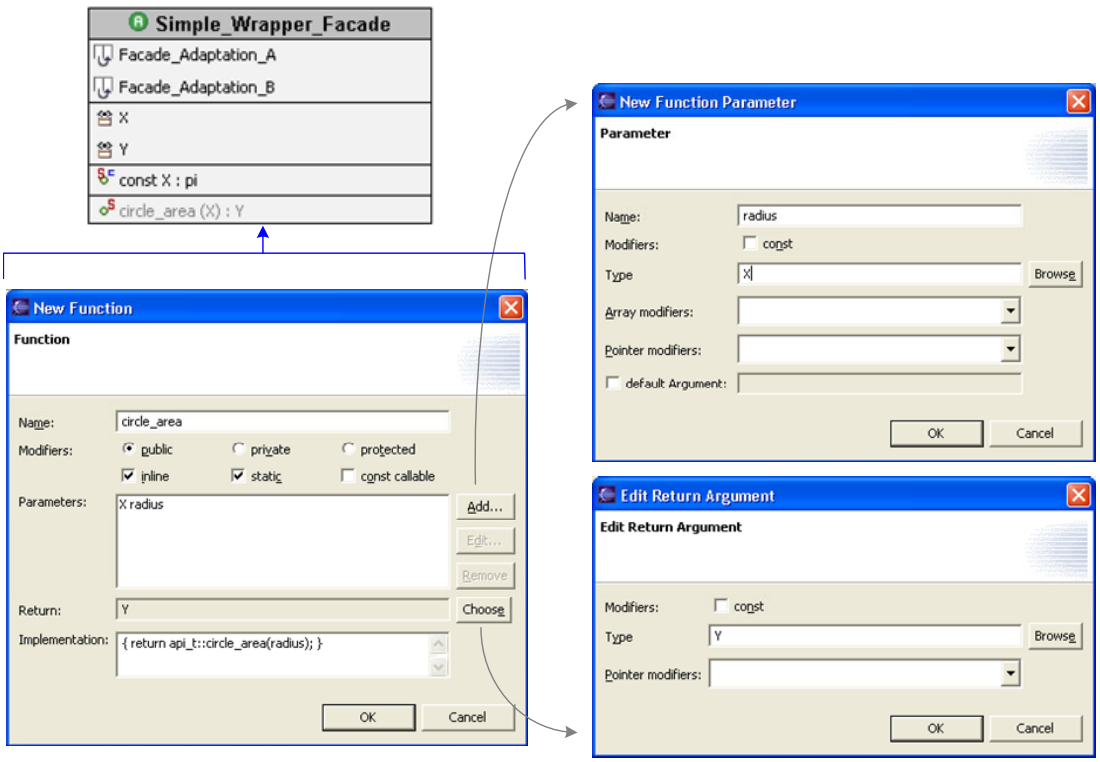

Abbildung 3.10: Erstellungsschritte der *circle_area* Funktion.

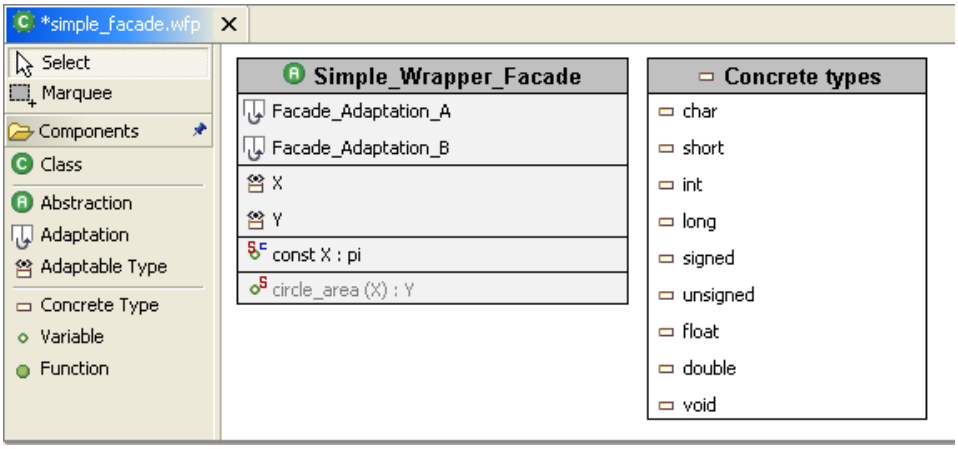

Abbildung 3.11: Darstellung der fertigen Simple Wrapper Façade im Design-Tool.

In den Zeilen 27–35 findet die Parametrisierung der Abstraktionsschicht statt: Mit Hilfe der Präprozessor-Konstanten *SYS_A* und *SYS_B* wird an dieser Stelle zwischen den beiden Zielsystemen unterschieden und eine systemspezifische Konfiguration der Template-Klasse unter dem Typ *INTERFACE* erstellt. Dieser Typ dient zur Vereinheitlichung der unterschiedlichen Template-Konfigurationen. In Anwendungscode kann über den *INTERFACE*-Typ die Funktionalität der Wrapper Façade benutzt werden, ohne Kenntnis der systemspezifischen Konfiguration haben zu müssen.

2. Cpp-Datei (Codefragment 3.2)

   Die Cpp-Datei dient zur Ergänzung der zuvor erzeugten Header-Datei. Ausstehende Implementierungen von Klassen-Funktionen und Initialisierungen von Klassen-Variablen können hier manuell hinzugefügt werden, wenn diese Informationen nicht zuvor schon im Design-Tool angegeben wurden.

   Weil im hier vorgestellten Fall alle notwendigen Informationen im Design-Tool angegeben wurden (vgl. Erstellungsanleitung, Abschnitt 3.4.3) kann die erzeugte Cpp-Datei unverändert bleiben.

3.4.5 Code der Anpassungsschicht

Die vom Design-Tool erzeugte Initialimplementierung der Anpassungsschicht wird in diesem Abschnitt vorgestellt.

Für die Simple Wrapper Façade besteht die Anpassungsschicht aus je einer Header-Datei für die Zielsysteme A und B:

1. Anpassungsschicht für Zielsystem A (Codefragment 3.3)

   Aufgabe der Anpassungsschicht ist die Ersetzung von abstrakten Datentypen durch konkrete Datentypen, die Implementierung von systemspezifischen Klassen-Methoden und die Angabe systemspezifischer Werte für Konstanten.

   Die Anpassungsschicht enthält dazu entsprechende Gegenstücke für die Typen, Konstanten und Funktionen der Abstraktionsschicht. Diese müssen noch um die fehlenden systemspezifischen Informationen ergänzt werden. Um diese Aufgabe zu erleichtern, werden die zu ändernden Stellen während der Codegenerierung mit entsprechenden Kommentaren versehen (vgl. Codefragment 3.3, Zeilen 7–9, 12).

   Zur Vervollständigung der Anpassungsschicht müssen die abstrakten Datentypen *X* und *Y* auf konkrete Datentypen abgebildet werden, eine systemspezifische Implementierung der Methode *circle_area* muss angegeben und die Konstante *pi* initialisiert werden.

Codefragment 3.1 Abstaktionsschicht - Datei *Simple_Wrapper_Facade.h*

```
1  // This is an autogenerated file. Manual changes might be lost.
2  #ifndef SIMPLE_WRAPPER_FACADE_H
3  #define SIMPLE_WRAPPER_FACADE_H
4
5  template<typename api_t>
6  class Simple_Wrapper_Facade {
7  public:
8   typedef typename api_t::X X;
9   typedef typename api_t::Y Y;
10  static const X pi;
11
12  inline static Y circle_area(X radius)
13  { return api_t::circle_area(radius); }
14
15 protected:
16
17 private:
18
19 };
20
21 // constant values
22 template<typename api_t>
23 const typename Simple_Wrapper_Facade<api_t>::X
24       Simple_Wrapper_Facade<api_t>::pi= api_t::pi;
25
26 // here is the only distinction between Operating Systems
27 #ifdef SYS_A
28 #include "Facade_Adaptation_A.h"
29 typedef Simple_Wrapper_Facade<Facade_Adaptation_A> INTERFACE;
30 #endif
31
32 #ifdef SYS_B
33 #include "Facade_Adaptation_B.h"
34 typedef Simple_Wrapper_Facade<Facade_Adaptation_B> INTERFACE;
35 #endif
36
37 #endif // SIMPLE_WRAPPER_FACADE_H
```

Codefragment 3.2 Abstraktionsschicht - Datei *Simple_Wrapper_Facade.cpp*

```
1  // This is an autogenerated file. Manual changes might be lost.
2
3  #include "Simple_Wrapper_Facade.h"
```

Die notwendigen Änderungen, um die Anpassungsschicht des Zielsystems A zu vervollständigen, sind in Codefragment 3.4 dargestellt. Im dargestellten Szenario verwendet das Zielsystem A Gleitkommaarithmetik. Die abstrakten Datentypen werden deshalb auf den Typ *double* abgebildet (Zeilen 7–8) und die Konstante *pi* entsprechend initialisiert (Zeile 9).

Zuletzt wird die Methode *circle_area* vervollständigt (Zeilen 11–13). Falls eine systemnahe API gekapselt werden sollte, würden an dieser Stelle die entsprechenden API-Funktionen aufgerufen werden. Stattdessen wird im dargestellten Beispiel die gewünschte Funktionalität (Flächenberechnung eines Kreises) direkt implementiert.

2. Anpassungsschicht für Zielsystem B (Codefragment 3.5)

   Die automatisch erzeugte Header-Datei für das Zielsystem B unterscheidet sich nicht wesentlich von Codefragment 3.3. Daher wird auf eine erneute Darstellung verzichtet. Der einzige Unterschied ist, dass *Facade_Adaptation_A* durch *Facade_Adaptation_B* ersetzt wird (Zeilen 2-3, 5, 20).

   Für das Zielsystem B, stellt Codefragment 3.5 die notwendigen Ergänzungen der Anpassungsschicht dar. Im dargestellten Szenario benutzt das Zielsystem B Ganzzahlarithmetik. Daher werden die abstrakten Datentypen auf den Typ *int* abgebildet und die Konstante *pi* mit einem ganzzahligen Wert initialisiert (Zeilen 7–9). Die Implementierung der Methode *circle_area* (Zeilen 11–13) ist in diesem Fall identisch mit der des ersten Zielsystems.

3.4.6 Code einer Beispielanwendung

Die Verwendung der fertigen Wrapper Façade in Anwendungscode wird anhand des Codefragments 3.6 demonstriert.

Zunächst muss die zu benutzende Wrapper Façade über eine entsprechende *Include*-Direktive eingebunden werden (Zeile 1). Der Datentyp *INTERFACE* ermöglicht einen systemunabhängigen Zugriff auf die Datentypen, Konstanten und Methoden der Wrapper Façade: In Zeile 5 wird ein abstrakter Datentyp der Façade bei einer Variablendefinition verwendet. In Zeilen 6 bzw. 8 wird eine Konstante bzw. eine Methode der Façade aufgerufen.

Hinter dem Datentyp *INTERFACE* versteckt sich eine systemabhängige Konfiguration der Simple Wrapper Façade. Die Konfiguration kann durch Angabe der Präprozessorkonstanten *SYS_A* bzw. *SYS_B* zur Übersetzungszeit ausgetauscht werden, so dass jeweils eine andere Implementierung der Anpassungsschicht verwendet wird. Dies wird in Codefragment 3.7 dargestellt.

Codefragment 3.3 Anpassungsschicht – Generierte Header-Datei für Zielsystem A.

```
1  // This is an autogenerated file. Manual changes might be lost.
2  #ifndef FACADE_ADAPTATION_A_H
3  #define FACADE_ADAPTATION_A_H
4
5  class Facade_Adaptation_A {
6  public:
7   typedef /*TODO insert type*/ X;
8   typedef /*TODO insert type*/ Y;
9   static const X pi = /*TODO insert literal constant*/;
10
11  inline static Y circle_area(X radius) {
12   /*TODO implement circle_area*/
13  }
14
15 protected:
16
17 private:
18
19 };
20 #endif // FACADE_ADAPTATION_A_H
```

Codefragment 3.4 Anpassungsschicht – Angepasste Header-Datei für Zielsystem A.

```
5  class Facade_Adaptation_A {
6  public:
7   typedef double X;
8   typedef double Y;
9   static const X pi = 3.14159;
10
11  inline static Y circle_area(X radius) {
12    return pi * radius * radius;
13  }
```

Codefragment 3.5 Anpassungsschicht – Angepasste Header-Datei für Zielsystem B.

```
5  class Facade_Adaptation_B {
6  public:
7   typedef int X;
8   typedef int Y;
9   static const X pi = 3;
10
11  inline static Y circle_area(X radius) {
12    return pi * radius * radius;
13  }
```

Codefragment 3.6 Verwendung der Wrapper Façade in einer Beispielanwendung.

```
 1  #include "Simple_Wrapper_Facade.h"
 2  #include <iostream>
 3
 4  int main() {
 5    INTERFACE::X radius = 10;
 6    std::cout << "pi= " << INTERFACE::pi << std::endl;
 7    std::cout << "A circle with radius " << radius
 8              << " has area " << INTERFACE::circle_area(radius)
 9              << std::endl;
10    return 0;
11  }
```

Codefragment 3.7 Übersetzung und Aufruf der Beispielanwendung unter Verwendung der unterschiedlichen Implementierungen.

```
1  # g++ -D SYS_A example.cpp
2  # a.out
3  pi= 3.14159
4  A circle with radius 10 has area 314.159
5
6  # g++ -D SYS_B example.cpp
7  # a.out
8  pi= 3
9  A circle with radius 10 has area 300
```

3.5 Anforderungserfüllung

In diesem Abschnitt wird erläutert, wie die zuvor gestellten Anforderungen (3.1) durch das Wrapper Façade Plugin erfüllt werden.

1. Anforderung: Unterstützung der Wrapper Façade Entwurfsanleitung.

Die Entwurfsanleitung zur Erstellung von Wrapper Façades (2.5) besteht aus fünf Schritten, die hier noch einmal aufgeführt werden. Zusätzlich wird aufgeführt, inwiefern jeder Schritt durch das Design-Tool unterstützt wird.

1. Identifikation zusammenhängender API-Funktionen.

   Der Identifikationsvorgang an sich, wird vom Design-Tool nicht unterstützt, weil die Auswahl der API-Funktionen, die in der Wrapper Façade gekapselt werden sollen, durch den Benutzer erfolgen muss.

2. Gruppierung von zusammenhängenden Funktionen zu Wrapper Façade Klassen und Methoden:

   2.1. Erstellung zusammenhängender Wrapper Façade Klassen.

   Dieser Schritt befasst sich mit der Erstellung einer objektorientierten Schnittstelle zur Kapselung systemspezifischer Funktionen. Für die im 1. Schritt ausgewählten Funktionen sollen entsprechende Methoden erstellt und auf Klassen verteilt werden.

   Die Erstellung der Methoden wird mit Hilfe des Import-Tools erleichtert. Dieses kann Funktions- und Methodendeklarationen aus existierendem C++ Code in den graphischen Editor importieren. Diese können dann verändert werden, indem z.B. die Datentypen von Parametern durch abstrakte Datentypen ersetzt werden, um daraus eine systemunabhängigen objektorientierten Schnittstelle zu erstellen. Um die Verteilung von Methoden auf Klassen ändern zu können, können Methoden zwischen zwei Klassen verschoben werden.

   2.2. Zusammenfassung einzelner API-Funktionen in einer Methode.

   Um mehrere API-Funktionen in einer Methode zusammenzufassen, kann im Design-Tool der entsprechende Programmcode im Bearbeitungsdialog des Methodenelementes eingetragen werden.

   2.3. Automatisierung von Erstellungs- und Freigabeaufrufen (falls möglich).

   Das Design-Tool unterstützt die Deklaration von Konstruktoren und Destruktoren, die zur Automatisierung von Erstellungs- und Freigabeaufrufen systemnaher Ressourcen benutzt werden können.

2.4. Festlegung des Indirektionsausmaßes.

Wird bei der Erstellung eines Methoden-Elementes die *inline*-Eigenschaft gesetzt, wird dies bei der Codegenerierung berücksichtigt, um durch *Inlining* Effizienzeinbuße zu vermeiden.

2.5. Auswahl eines Implementierungsverfahrens für die Anpassungsschicht.

Prinzipiell ist die Integration mehrerer Codegeneratoren in das Design-Tool möglich, die bei der Erzeugung von Programmcode aus dem Wrapper Façade Modell unterschiedliche Implementierungsverfahren anwenden können. Der genaue Mechanismus wird in Abschnitt 5.2 beschrieben. In der aktuellen Implementierung wird jedoch nur ein Codegenerator verwendet.

3. Ermöglichung eines kontrollierten Zugriff auf Implementierungsinterna.

Dies wird vom Design-Tool unterstützt: Der Benutzer kann eigene *get* bzw. *set*-Methoden erstellen oder gleich Klassen-Variablen auf *protected* oder *public* setzen, um so den Zugriff auf Implementierungsinterna einer Klasse zu erlauben.

4. Auswahl eines Fehlerbehandlungsmechanismus.

Die Verwendung von C++ Ausnahmen (*Exceptions*) im Moment nicht unterstützt.

5. Erstellung weiterer Klassen, welche auf die Wrapper Façade aufbauen.

Das Design-Tool kann auch als normales Modellierungstool zur Erstellung von C++ Klassen verwendet werden: Es können Klassen modelliert werden, die nicht auf die Anpassungsschicht abgebildet werden. Diesen Klassen können Methoden und Variablen hinzugefügt werden. Bei der Codegenerierung wird eine Header-Datei mit den entsprechenden Deklarationen erstellt. Im Design-Tool angegebene Implementierungen von Methoden oder Wertzuweisungen von Variablen, werden ebenfalls in den erstellten Programmcode eingefügt.

2. Anforderung: Unterstützung bei der Erstellung der Abstraktionsschicht.

Die Entwurf der Abstraktionsschicht wird durch eine graphische Darstellung ihrer Komponenten erleichtert. Das Design-Tool ermöglicht das Erstellen eines Wrapper Façade Modells durch Kombinieren seiner Grundelemente, also Klassen, Datentypen, Variablen und Methoden. Jedes Element der Darstellung kann über sein Kontextmenü editiert werden. Deklarationen von Methoden und Variablen können per *drag & drop* zwischen zwei Klassen verschoben werden. Bestimmte Bearbeitungsschritte können zurückgenommen oder wiederholt werden (*Undo / Redo*).

3. Anforderung: Unterstützung bei der Erstellung der Anpassungsschicht.

Die Erstellung der Anpassungsschicht wird über die „*Adaptation*-Elemente“ des Design-Tools gesteuert.

Die *Adaptation*-Elemente können in „Abstraktions-Klassen“ (s.u.) verwendet werden. Ein *Adaptation*-Element stellt eine systemspezifische Konfiguration seiner Abstraktions-Klasse dar. Pro Zielsystem muss die Abstraktions-Klasse also je ein *Adaptation*-Element enthalten. Während der Codegenerierung nach dem Template-Verfahren, wird für jedes *Adaptation*-Element eine eigene Klasse erzeugt, die zur Anpassung an ein Zielsystem verwendet werden kann und die dafür notwendigen Gegenstücke enthält.

Abstraktions-Klassen sind anpassbare Klassen, die mehrere *Adaptation*-Elemente aufnehmen können. Zur Unterscheidung von herkömmlichen (nicht konfigurierbaren) Klassen, werden Abstraktions-Klassen im Editor durch den Buchstaben A gekennzeichnet (vgl. Abbildung 3.1). Bei der Codegenerierung unter Anwendung des Template-Verfahrens wird für eine Abstraktions-Klasse ein Template erzeugt, das über den Template-Parameter die Auswahl der passenden Adaptation erlaubt.

4. Anforderung: Unterstützung bei der Erstellung einer Implementierung.

Zur Erstellung einer Implementierung kann die Codegeneratorkomponente des Wrapper Façade Plugins verwendet werden. Diese erzeugt für jede Klasse des Wrapper Façade Modells eine Header- und Cpp-Datei. Inhalt und Aufbau der erzeugten Dateien wurden in Abschnitt 3.2.3 vorgestellt. Manuell anzupassende Stellen im generierten Code sind durch Kommentare gekennzeichnet.

5. Anforderung: Erkennung und Vermeidung von Falscheingaben.

Zur Erkennung und Vermeidung von Falscheingaben wird eine dynamische Überprüfung der Modelldaten vorgenommen. Diese erfolgt während der Dateneingabe im jeweiligen Bearbeitungsdialog. Erkannte Fehler werden durch entsprechende Meldungen im Dialog angezeigt und die Bearbeitung des Modellelementes kann nicht abgeschlossen werden, bis die Fehler durch den Benutzer behoben worden sind.

Während der Dateneingabe können unter anderem folgende Fehler erkannt werden:

- Zyklische Vererbung (Klasse A erbt von Klasse B erbt von Klasse A usw.).
- Typbezeichner die nicht den C++ Richtlinien entsprechen.
- Falsche Reihenfolge von Methodenparametern mit *Default*-Werten (solche Parameter müssen in der Methodendeklaration immer an letzter Stelle kommen, weil sie weggelassen werden können).

- Bestimmte unzulässige Wertkombinationen bei Variablen- und Methodendeklarationen (z.B. Variablen vom Typ *void* oder statische Konstruktoren).
- Nicht wohlgeformte Argumente von *#include*-Direktiven.
- Nicht zulässiges Überladen von Operatoren (das Design-Tool verfügt über eine Liste aller C++ Operatoren die überladen werden können).
- Die Deklaration abstrakter Datentypen in Abstraktions-Elementen, die keine Anpassungsschicht besitzen.
- Das Verschieben von Konstruktoren und Destruktoren zwischen zwei Klassen wird verhindert.

6. Anforderung: Integration in eine Entwicklungsumgebung.

Diese Anforderung wird durch die Integration der erstellten Software in die Eclipse-Entwicklungsumgebung erfüllt.

Die Bearbeitung eines Wrapper Façade Modells findet innerhalb eines graphischen Editors statt, der in die Entwicklungsumgebung integriert ist und neben anderen Editorinstanzen laufen kann.

Die Datei Ein- und Ausgabe des Editors ist in die Ressourcenverwaltung der Eclipse-Entwicklungsumgebung integriert:

- Ein Eintrag in der Plugin-Definition (Datei *plugin.xml*) stellt sicher, dass die Entwicklungsumgebung den Wrapper Façade Editor aufruft, wenn eine Datei mit der Endung `.wfp` geöffnet oder erstellt wird.
- Aufrufen des Speichern-Befehls in der Entwicklungsumgebung hat zur Folge, dass bestimmte Methoden im Editor ausgeführt werden (*callback* Mechanismus). Dieser benutzt wiederum das *Ressource-Framework* [SDF+03] der Eclipse-API, um die zu speichernde Datei im richtigen Projektverzeichnis abzulegen.

Der generierte C++ Code ist mit der Projektverwaltung der Entwicklungsumgebung verbunden, wird in einem Unterverzeichnis des aktuellen Projektes erstellt und kann innerhalb der Entwicklungsumgebung weiterbearbeitet werden[2].

3.6 Forschungs- und Entwicklungsmöglichkeiten

Im Verlauf der Arbeit sind auch Ideen entstanden, die auf Grund des engen zeitlichen Rahmens nicht weiterverfolgt wurden. Dieser Abschnitt soll Ideen für weitere Forschung und Möglichkeiten zur Weiterentwicklung der Software vorstellen.

[2] Mit Hilfe des *C/C++ Development Tools* (CDT), dass Eclipse um C/C++ Unterstützung erweitert.

Parametrisierung mit mehreren Freiheitsgeraden

Die in der Arbeit vorgestellten Implementierungsverfahren (2.6) ermöglichen eine Parametrisierung der Abstraktionsschicht mit nur einem Freiheitsgrad. Damit ist gemeint, dass eine Klasse der Abstraktionsschicht auf eine einzelne Klasse der Anpassungsschicht abgebildet wird, in der alle variablen Aspekte der Abstraktionsschicht festgelegt werden.

Nicht untersucht wurde, ob mit Hilfe fortgeschrittener Template-Konzepte [VJ02], eine Parametrisierung einer Klasse der Abstraktionsschicht mit *mehreren* Freiheitsgeraden möglich ist. Eine Möglichkeit dafür wäre, unter Benutzung mehrerer Template-Parameter – einen für jeden variablen Aspekt der zu parametrisierenden Klasse – eine Abbildung auf mehrere Klassen der Anpassungsschicht vorzunehmen.

Hierbei wären folgende Fragen zu untersuchen: Wie könnte eine Parametrisierung mit mehreren Freiheitsgeraden aussehen? Welche Implementierungstechniken könnten angewandt werden? In wiefern wäre diese Weiterentwicklung des Entwurfsmusters mit den Zielen des Aspekt-Orientierten-Programmierens [EFB01] vereinbar? Wie könnte diese Weiterentwicklung von der hier vorgestellten Software unterstützt werden?

Erstellung weiterer Anwendungsbeispiele

Um die Verwendbarkeit des Design-Tools und des Code-Generators zu untersuchen, wurden zwei Anwendungsbeispiele erstellt: die bereits vorgestellte Simple Wrapper Façade (Abschnitt 3.4) und die noch zu besprechende Socket Wrapper Façade (Kapitel 4).

Die Erstellung zwei weiterer Anwendungsbeispiele, unter Verwendung des Template-Implementierungsverfahrens, wurde in Betracht gezogen, aber nicht weiter verfolgt: Erstens, die Erstellung einer Wrapper Façade zur Kapselung verschiedener GUI-Bibliotheken. Zweitens, die ansatzweise Reimplementierung des ACE-Frameworks ([Sch04], [Sch98]), welches einen einheitlichen Zugriff auf Betriebsystemressourcen ermöglicht, unter Verwendung des Template- anstatt des Präprozessorverfahrens.

Verwendung mehrerer Implementierungsverfahren

Bei der Implementierung des Codegenerators wurde das *Visitor*-Entwurfsmuster verwendet (5.2). Dies ermöglicht den einfachen Austausch des verwendeten Codegenerators und somit die Verwendung verschiedener Implementierungsverfahren (2.6).

Diese Möglichkeit wurde bis jetzt noch nicht ausgenutzt, da erst ein Codegenerator implementiert wurde, der das Template-Verfahren anwendet. Die Erstellung weiterer Codegeneratoren, z.B. zur Anwendung des Präprozessorverfahrens oder des Abstract-Factory-Verfahrens, würde eine Verwendung mehrerer Implementierungsverfahren auf

ein Wrapper Façade Modell ermöglichen und dadurch eine einfache Vergleichsmöglichkeit der verschiedenen Verfahren schaffen.

Weiterentwicklung der Software

Neben den oben vorgestellten konzeptuellen Weiterentwicklungsmöglichkeiten, können auch noch einige konkrete Verbesserungen vorgenommen werden:

- Die Anwenderfreundlichkeit der Software könnte durch Benutzung von *Tooltips* und Ausnutzung des von Eclipse bereitgestellten Hilfesystems verbessert werden.
- Der für das Anordnen der Diagrammelemente verwendete Layoutalgorithmus berücksichtigt nicht, dass es sich hierbei um Klassendiagramme handelt. Die Darstellung der Diagramme könnte durch Verwendung spezialisierter Layoutalgorithmen (z.B. [See97] oder [EKS03]) verbessert werden.
- Um im automatisch erzeugten Quellcode die Notwendigkeit für manuelle Anpassungen zu minimieren, erlaubt das Design-Tool die direkte Eingabe einer Implementierung für Methoden bzw. einer Wertzuweisung für Variablen. Der eingegebene Code wird hierbei nur rudimentär überprüft. Die Überprüfung könnte durch eine stärkere Integration des CDT-Eclipse-Plugins während der Codeeingabe verbessert werden.
- Manuelle Änderungen in den automatisch erzeugten Quellcode-Dateien werden nicht erkannt und bei erneuter Codeerzeugung überschrieben (das sog. *roundtrip* Problem). Dies könnte durch eine Weiterentwicklung des Codegenerators vermieden werden.
- Die Speicherung des Layouts wird noch nicht unterstützt, da nur die eingegebenen Modelldaten gespeichert werden. Beim nächsten Ladevorgang wird das Layout automatisch neu berechnet. Ein persistentes Layout könnte durch eine entsprechende Ergänzung der gespeicherten Informationen ermöglicht werden.
- Die von der Software angebotene Unterstützung bei der Modellierung der Anpassungsschicht könnte ausgebaut werden. Denkbar wäre z.B. eine Erweiterung, die es ermöglicht, die Abbildung der abstrakten Datentypen auf konkrete Datentypen im Design-Tool vorzunehmen.

Kapitel 4

Ein komplexes Anwendungsbeispiel

In diesem Kapitel wird ein komplexes Anwendungsbeispiel für das Wrapper Façade Entwurfsmuster vorgestellt.

4.1 Socket Wrapper Façade

Hierbei handelt es sich um eine Klassenbibliothek, die eine Kapselung der Socket-API vornimmt und stattdessen eine objektorientierte Schnittstelle mit höherem Abstraktionsgrad anbietet. Die Anpassungsschicht der Socket Wrapper Façade enthält Implementierungen für Windows (mit *WSASockets* [Sin97]), Linux und Solaris (beide mit *BSD Sockets* [Ste98]). Vervollständigt wird das Beispiel durch zwei Anwendungen in Form eines einfachen Client- bzw. Serverprogramms, welche die Socket Wrapper Façade benutzen um miteinander zu kommunizieren. Der Anwendungscode ist ohne Änderungen auf allen drei Betriebssystemen lauffähig.

Die Implementierung der Socket Wrapper Façade wurde mit der Codegenerierungsfunktion des Design-Tools (3.2.3) erstellt. Als Implementierungsverfahren wurde das Template-Verfahren (2.6) gewählt.

Im Verlauf dieses Abschnitts werden zunächst die Vorgehensweise bei der Erstellung und die Einschränkungen der Klassenbibliothek besprochen. Anschließend werden die Bestandteile der Abstraktions- und Anpassungsschichten vorgestellt. Zuletzt wird die Nutzung der Klassenbibliothek anhand eines Client- bzw. Serverprogramms vorgeführt.

4.1.1 Vorgehen

Bei der Erstellung der Socket Wrapper Façade wurde auf eine bereits vorhandene Implementierung zurückgegriffen [Krä99]. Diese wurde im Design-Tool nachgebildet (vgl. Abbildung 4.1), um eine Implementierung der Anpassungsschicht für Windows

XP ergänzt und an die in der Arbeit verwendeten Compiler angepasst[1].

Die Header-Dateien der Klassenbibliothek wurden über die Codegenerierungsfunktion erzeugt. Später wurde die Implementierung der einzelnen Methoden und die verwendeten Variablenzuweisungen mit in das Design-Tool eingefügt, so dass im Endstadium die komplette Implementierung der Klassenbibliothek, bis auf das *Makefile* und die beiden Beispielprogramme, über den Codegenerator erstellt werden kann.

4.1.2 Einschränkungen

Um den Implementierungsaufwand und Umfang des Anwendungsbeispiels in Grenzen zu halten wurden folgenden Einschränkungen in Kauf genommen:

1. Verbindungslose Kommunikation (über UDP [Ste98]) wird nicht unterstützt.

2. Nichtblockierender Verbindungsaufbau und Nachrichtenversand ist nicht möglich – der Kontrollfluss des aufrufenden Programms wird angehalten, bis die entsprechenden Aufrufe abgeschlossen sind.

3. Zur Adressierung der Kommunikationsendpunkte können nur 32bit-lange IPv4 Adressen ([Ste98, Ch.3]) verwendet werden.

4. Neben der Socket-API, wurden keine anderen C-Bibliotheksaufrufe gekapselt. Dazu gehören z.B. Funktionen zur Stringverarbeitung (*strchr, strcpy*) oder zur Konvertierung zwischen Host- und Netzwerk-Byteanordung (*htonl, htons, ntohl*).

4.1.3 Abstraktionsschicht

Das Design der Abstraktionsschicht ist an [SH01] und [Sch92] angelehnt und besteht aus sieben Klassen. Diese sind in Abbildung 4.1 dargestellt und werden jetzt vorgestellt:

1. CommonAPI:

   Dies ist die zentrale Klasse der Socket Wrapper Façade.

   Sie übernimmt die systemunabhängige Kapselung der Socket-API und bietet somit eine Grundlage, auf die der Rest der Klassenbibliothek aufbauen kann. Dadurch wird die Erstellung der Klassenbibliothek vereinfacht, weil alle systemspezifischen Eigenheiten in der Anpassungsschicht der `CommonAPI` versteckt werden. Über den Template-Parameter wird zur Übersetzungszeit die für das Zielsystem passende Konfiguration ausgewählt.

[1] Visual C++ 7.0 .NET unter Windows XP und GCC 3.2 unter Linux und Solaris.

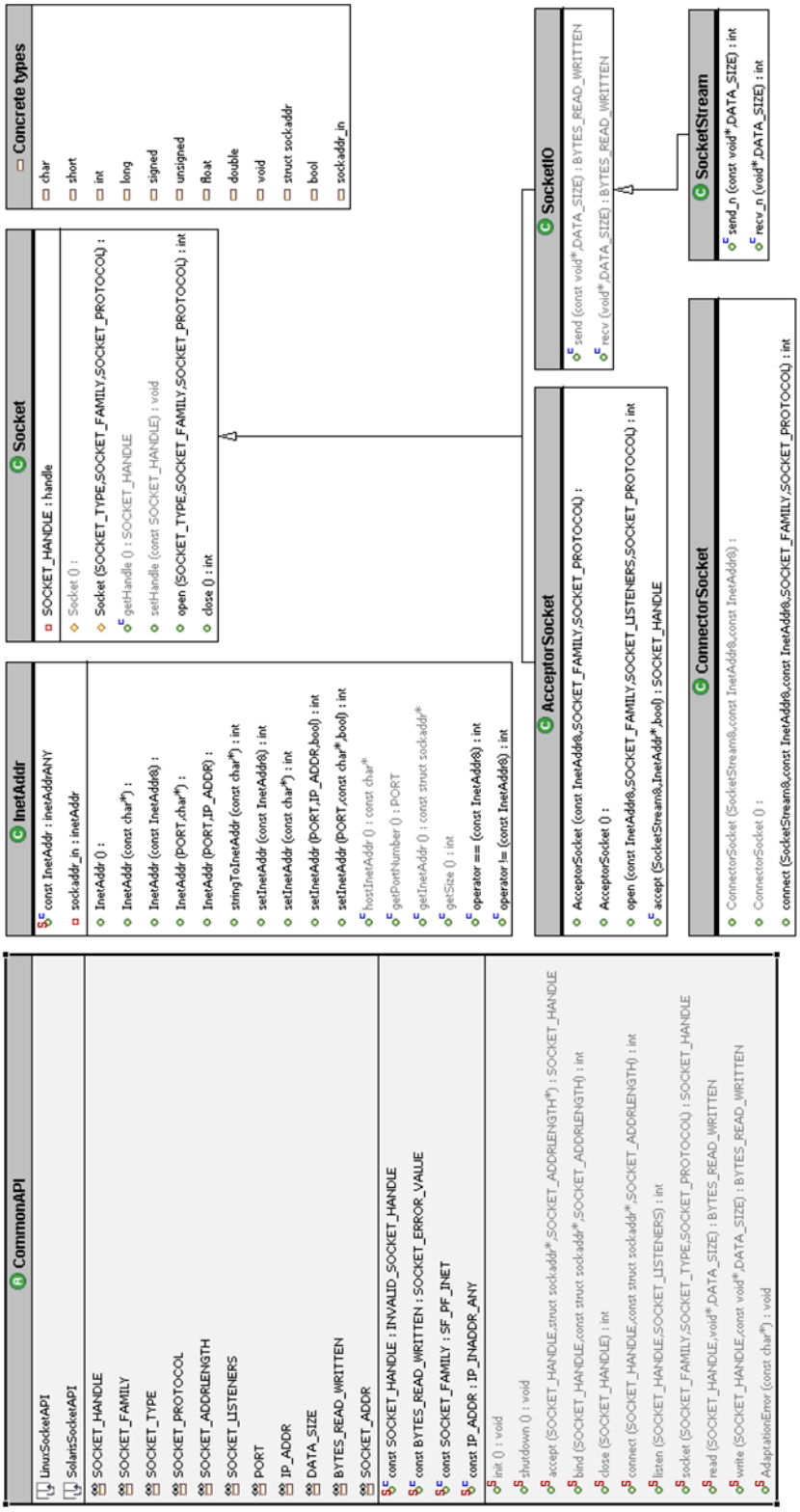

Abbildung 4.1: Darstellung der Socket Wrapper Façade im Design-Tool.

Die Kapselung der Socket-API geschieht durch drei Elemente: Abstrakte Datentypen, wie z.B. *SOCKET_HANDLE*, erzeugen die Grundlage für die Kapselung der eigentlichen API-Elemente. Durch die Verwendung statischer Klassen-Variablen (z.B. *INVALID_SOCKET_HANDLE*), werden API-Konstanten gekapselt. Durch Klassen-Methoden mit angepassten Signaturen werden die API-Funktionen gekapselt. Hierbei werden, soweit wie nötig, abstrakte Datentypen bei der Angabe von Parameter- und Rückgabetypen verwendet.

2. InetAddr:

   Die Klasse *InetAddr* wird zur Adressierung von Kommunikationsendpunkten verwendet. Die Adresse eines Endpunktes setzt sich aus einer Netzwerk-Adresse und einer Port-Nummer zusammen.

   InetAddr soll den Programmierer von den Feinheiten der Netzwerkadressierung abschirmen. Dazu bietet die Klasse diverse Konstruktoren, mit Hilfe derer aus zeichenbasierten oder numerischen Datenformaten eine gültige Netzwerk-Adresse und Portangabe erstellt werden kann.

3. Socket:

   Diese Klasse verwaltet einen Kommunikationsendpunkt (*Socket*) und enthält dazu eine Klassen-Variable des abstrakten Datentyps *SOCKET_HANDLE*. Zusätzlich stellt sie Methoden zum Öffnen und Schließen des gekapselten Sockets zur Verfügung.

4. AcceptorSocket:

   Die *AcceptorSocket*-Klasse erzeugt einen Kommunikationsendpunkt, der zur Annahme von eingehenden Verbindungen benutzt werden kann (passiver Socket).

   Bei der Instanziierung der Klasse wird ein Kommunikationsendpunkt an die übergebene lokale Netzwerk-Adresse und Port-Nummer gebunden und für die Verbindungsannahme vorbereitet. Daraufhin kann über die *accept(SocketStream, bool)*-Methode eine Verbindung angenommen werden: Ein Aufruf dieser Methode blockiert den Kontrollfluss. Das Programm wartet daraufhin auf das Eintreffen einer Verbindungsanfrage an diesem Kommunikationsendpunkt. Die Verbindungsanfrage führt zum Erzeugen eines neuen Kommunikationsendpunktes und Wiederaufnahme des Programmablaufs. Nach Rückkehr aus dem *accept()*-Aufruf kann das, als Parameter übergebene, *SocketStream* Objekt (s.u.) verwendet werden, um den weiteren Nachrichtenaustausch abzuwickeln.

5. ConnectorSocket:

   Die Klasse *ConnectorSocket* wird zum Verbindungsaufbau mit einem wartenden Kommunikationsendpunkt (*AcceptorSocket*) benutzt.

   Bei der Instanziierung bzw. dem Aufruf der *connect()*-Methode, wird eine Zieladresse und ein *SocketStream* Objekt angegeben. Ist der Verbindungsaufbau zur

angegebenen Zieladresse erfolgreich, kann über das mit übergebene *SocketStream* Objekt kommuniziert werden.

6. SocketIO:

   Die *SocketIO*-Klasse ermöglicht das Senden und Empfangen von Nachrichten über ein *Socket*. Der Übertragungsvorgang ist hierbei „unzuverlässig“, d.h. es ist nicht sichergestellt, dass ein Aufruf der Sende- bzw. Empfangsmethode die vollständige Nachricht übermittelt.

7. SocketStream:

   Diese Klasse ist eine Erweiterung von *SocketIO* und ermöglicht den „zuverlässigen“ Versand von Nachrichten. Hierbei wird zunächst die Größe der zu sendenden oder zu empfangenden Nachricht ermittelt. Anschließend werden ggf. mehrere Sende- oder Empfangsaufrufe durchgeführt, bis die Nachricht vollständig übermittelt ist.

4.1.4 Anpassungsschicht

Die Anpassungsschicht besteht aus je einer Klasse für die Zielsysteme Windows XP[2], Linux und Solaris. Mit Hilfe dieser Klassen wird eine systemspezifische Implementierung der *CommonAPI* gebildet. Sie übernehmen die Abbildung der abstrakten Datentypen auf systemspezifische Datentypen, initialisieren Konstanten mit systemspezifischen Werten und kapseln die Aufrufe der Socket-API hinter Klassen-Methoden.

Dieser Vorgang wird durch die Codefragmente 4.1 und 4.2 verdeutlicht, in denen Auszüge aus den Implementierungen für Windows XP und Linux gegenübergestellt werden. Zu sehen sind jeweils die Redefinition eines abstrakten Datentyps (Zeile 1), die Initialisierung einer Konstante (Zeile 2) und die Kapselung von Aufrufen der Socket-API hinter den Klassen-Methoden *socket()* und *read()*.

4.1.5 Client- und Server-Beispielprogramme

Anhand zweier Beispielprogramme soll die Benutzung der Klassenbibliothek in Anwendungscode demonstriert werden.

Das Serverprogramm nimmt Verbindungen auf einer festgelegten Netzwerk-Adresse und Port-Nummer entgegen, die zuvor über die Kommandozeile übergeben werden. Dazu wird ein *AcceptorSocket* mit der angegeben Adresse eingerichtet (Zeilen 12-13). Anschließend wartet der Server auf eine eingehende Verbindungsanfrage (Zeile 18).

[2]Die verwendeten API-Funktionen sind kompatibel zu Windows NT, d.h. die Klassenbibliothek sollte auch auf Windows NT und Windows 2000 laufen. Als Testplattform wurde jedoch nur Windows XP verwendet.

Codefragment 4.1 Anpassungsschicht – Auszug aus der Implementierung für Windows XP.

```
 1  typedef SOCKET SOCKET_HANDLE;
 2  static const SOCKET_HANDLE INVALID_SOCKET_HANDLE = INVALID_SOCKET;
 3
 4  inline static SOCKET_HANDLE socket(
 5   SOCKET_FAMILY family, SOCKET_TYPE type, SOCKET_PROTOCOL protocol) {
 6    return ::WSASocket(family, type, protocol, NULL, NULL, NULL);
 7  }
 8
 9  inline static BYTES_READ_WRITTEN read(
10   SOCKET_HANDLE sockfd, void* buff, DATA_SIZE nbytes) {
11    WSABUF dataBuff;
12    dataBuff.buf = (char FAR *) buff;
13    dataBuff.len = (u_long) nbytes;
14    DWORD bytesRec = 0;
15    DWORD flags = 0;
16    if (WSARecv(sockfd, &dataBuff, 1, &bytesRec, &flags, NULL, NULL)
17       == SOCKET_ERROR) {
18      AdaptationError("WSARecv");
19      return 0;
20    }
21    return bytesRec;
22  }
```

Codefragment 4.2 Anpassungsschicht – Auszug aus der Implementierung für Linux.

```
 1  typedef int SOCKET_HANDLE;
 2  static const SOCKET_HANDLE INVALID_SOCKET_HANDLE = -1;
 3
 4  inline static SOCKET_HANDLE socket(
 5   SOCKET_FAMILY family, SOCKET_TYPE type, SOCKET_PROTOCOL protocol) {
 6    return ::socket(family, type, protocol);
 7  }
 8
 9  inline static BYTES_READ_WRITTEN read(
10   SOCKET_HANDLE sockfd, void* buff, DATA_SIZE nbytes) {
11    return ::read(sockfd, buff, nbytes);
12  }
```

Nach Zustandekommen einer Verbindung wird ein *SocketStream*-Objekt benutzt, um eine Nachricht zu verschicken (Zeile 19). Dieser Vorgang wird wiederholt, bis das Serverprogramm abgebrochen wird.

Das Clientprogramm verbindet sich mit einer bestimmten Netzwerk-Adresse und Port, die ebenfalls über die Kommandozeile angegeben werden. Zum Verbindungsaufbau wird ein *ConnectorSocket*-Objekt erstellt (Zeilen 14-16). Dazu werden eine Zieladresse und ein *SocketStream*-Objekt benötigt. Kommt die Verbindung zustande, wird über das *SocketStream*-Objekt eine Nachricht empfangen und anschließend ausgegeben (Zeilen 18-20).

Zur Vereinfachung der Beispielprogramme wurde auf Fehlerbehandlungscode verzichtet.

Codefragment 4.3 Beispielanwendung – Minimales Server-Programm

```
1  // miniserver.cpp
2  // compiles with: g++ -D SYS_LINUX -o miniserver miniserver.cpp
3  // or: g++ -D SYS_SOLARIS -o miniserver miniserver.cpp -lsocket -lnsl
4  // runs with: miniserver server_ip:server_port
5
6  #include "AcceptorSocket.h"
7  #include <iostream>
8  #include <string.h>
9
10 int main(int arc, char* argv[]) {
11   COMMON::init();
12   InetAddr serverAddr(argv[1]);
13   AcceptorSocket acceptor(serverAddr);
14
15   char* message = "This is the server!\0";
16   SocketStream stream;
17   while(1) {
18     acceptor.accept(stream);
19     stream.send_n(message, (COMMON::DATA_SIZE) strlen(message));
20   }
21 }
```

Codefragment 4.4 Beispielanwendung – Minimales Client-Programm

```
1  // miniclient.cpp
2  // compiles with: g++ -D SYS_LINUX -o miniclient miniclient.cpp
3  // or: g++ -D SYS_SOLARIS -o miniserver miniserver.cpp -lsocket -lnsl
4  // runs with: miniclient server_ip:server_port
5
6  #include "ConnectorSocket.h"
7  #include <iostream>
8  #include <string.h>
9
10 int main(int arc, char* argv[]) {
11   char* recvBuf = new char[4096];
12
13   COMMON::init();
14   InetAddr serverAddr(argv[1]);
15   SocketStream stream;
16   ConnectorSocket connector(stream, serverAddr);
17
18   int n = stream.recv(recvBuf, 4096);
19   recvBuf[n] = 0;
20   std::cout << "received: " << recvBuf << std::endl;
21   COMMON::shutdown();
22 }
```

Kapitel 5

Design und Implementierung

Dieses Kapitel beschreibt die wichtigsten Implementierungsaspekte des Wrapper Façade Plugins. Zuerst wird auf das Design des graphischen Editors nach dem *Model-View-Controller*-Prinzip und seiner Implementierung mit Hilfe des *Graphical Editing Frameworks* (GEF) eingegangen. Anschließend wird das Design des Codegenerators nach dem *Visitor*-Entwurfsmuster erläutert. Zuletzt wird der Entwurf des XML-basierten Datenformats des Editors beschrieben.

5.1 Design des graphischen Editors

Hauptbestandteil des Wrapper Façade Plugins ist ein graphischer Editor zur Modellierung von Wrapper Façades. Der Editor wurde nach dem *Model-View-Controller*-Prinzip entworfen und wurde mit Hilfe des *Graphical Editing Frameworks* implementiert.

Model-View-Controller

Das Model-View-Controller Entwurfsmuster [BMR97], abgekürzt MVC, erlaubt die Trennung von Datenmodell, Datenrepräsentation und Applikationslogik. Ein Datenmodell (z.B. Verkaufszahlen) kann mehrere unterschiedliche Darstellungen, sog. *Views*, haben (z.B. numerisch vs. Tortengraphik vs. Balkengraphik). Die *Controller* enthalten die Applikationslogik (z.B. welche Bearbeitungsschritte sind gerade erlaubt) und initiieren die zur Manipulation des Modells notwendigen Aktionen. Die Trennung der Zuständigkeiten führt zu verständlicherem und flexiblerem Applikationscode und erhöht die Wiederverwendbarkeit der einzelnen Komponenten.

Graphical Editing Framework

Das *Graphical Editing Framework*[1] ist eine Erweiterung von Eclipse, welche die Erstellung von Editoren nach dem MVC-Prinzip erleichtert. Dazu werden zwei Programmierschnittstellen zur Verfügung gestellt:

1. **Die Draw2D API** erleichtert die Erstellung und Verwendung zweidimensionaler graphischer Elemente auf folgende Art und Weise:

   - Bereitstellung fertiger graphischer Elemente. Dazu gehören häufig verwendete geometrische Objekte (Linien, Rechtecke, Polygone, Kreise, Ellipsen, usw.) und einige GUI-Elemente (Bedienelemente, Checkboxen, Textdarstellungselemente). [MDG+03].
   - Bereitstellung von Containern (*Panes* und *Layers*) zur Aufnahme und Darstellung mehrerer Elemente, inklusive Skalierung und *Scrolling*. Durch die Kombination mehrerer *Layers* in einer *Pane* können graphische Darstellungen mit mehreren Ebenen erstellt werden. [MDG+03]
   - Neue graphische Elemente können mit geringem Aufwand durch Kombination existierender Elemente erstellt werden. Die Implementierung eigener Elemente ist ebenfalls möglich.
   - Dargestellte Elemente können mit unterschiedlichen Rahmenarten (*Borders*) dekoriert werden [MDG+03].
   - Die Anordnung der Elemente wird durch *Layoutmanager* festgelegt. Für oft benötigte Anordnungen stellt Draw2D bereits geeignete Layoutmanager zur Verfügung.
   - Erstellung und Verwaltung von Verbindungen zwischen den graphischen Elementen: Ein graphisches Element kann mehrere Verbindungspunkte (*Anchors*) definieren. Diese dienen als Start- oder Endpunkt einer Verbindung zwischen zwei Elementen. Eine Verbindung kann z.B. ein einfache Linie oder ein Polygonsegment sein. Die Endpunkte der Verbindung können mit geometrischen Elementen dekoriert werden (z.B. Pfeil, Kreis). Der Pfad einer Verbindung kann frei bestimmt oder durch einen *Routing*-Algorithmus berechnet werden. Einige bekannte Routing-Algorithmen sind in Draw2D bereits implementiert (z.B. Manhattan- und Fan-Routing). [MDG+03]

2. **Die GEF API** definiert einen Rahmen zur Erstellung von MVC-Editoren, stellt Controller-Klassen zur Verfügung (vgl. 5.1.3) und verwendet Draw2D-Elemente zur Modellvisualisierung.

Die nächsten drei Abschnitte stellen die Model-, View- und Controller-Komponenten des Wrapper Façade Editors vor. Anschließend wird beschrieben wie diese drei Aspekte durch das *Graphical Editing Framework* integriert werden.

[1] http://www.eclipse.org/gef/

5.1.1 Model

Für die verschiedenen Bestandteile einer Wrapper Façade, also herkömmliche und anpassbare Klassen, abstrakte und konkrete Datentypen, Variablen, Konstanten und Funktionen, werden entsprechende Modellelemente bereitgestellt. Die Abbildung von Wrapper Façade Bestandteilen auf Modellelemente ist in Tabelle 5.1 dargestellt.

Die erste Aufgabe des Modells ist es, in seinen Modellelementen alle Daten zu speichern, die zur Rekonstruktion einer Wrapper Façade notwendig sind. Aus einem einzelnen Modellelement kann das entsprechende Wrapper Façade Bestandteil rekonstruiert werden. Bestimmte Modellelemente können verschachtelt werden und bilden so eine hierarchische Datenstruktur. Die gesamte hierarchische Datenstruktur beschreibt eine einzelne Wrapper Façade. Die Wurzel der Hierarchie ist immer ein *IModel*-Objekt. Die Struktur der Hierarchie ist in Abbildung 5.1 illustriert.

Die zweite Aufgabe des Modells ist, eine API bereit zu stellen, um die gespeicherten Daten und die Modellhierarchie auf kontrollierte Art und Weise bearbeiten zu können. Eine detaillierte Beschreibung der API ist in der Dokumentation der Quelltexte (javadoc) enthalten. An dieser Stelle sollen die Entwurfsmuster vorgestellt werden, die bei der Konzeption der Modell-API verwendet wurden:

1. Um eine Trennung der Schnittstellendefinition von der eigentlichen Implementierung vorzunehmen wurde das *Factory*-Entwurfsmuster [GHJV95] verwendet. Die Schnittstellendefinition besteht aus den Klassen des Paketes *wfp.model*. Die Schnittstellenimplementierung setzt sich aus den Klassen des Paketes *wfp.model.internal* zusammen. Klassen der Modellhierarchie werden über die *ModelFactory* erstellt. Sie bildet das Bindeglied zwischen Schnittstellendefinition und Schnittstellenimplementierung. Sollte die Verwendung einer alternativen API-Implementierung notwendig werden, kann diese durch Austausch der *ModelFactory* in das Programm integriert werden.

2. Die Integration des Modells in die restliche Anwendung erfordert manchmal das Implementieren modellfremder Interfaces von der Modell-API, die konzeptuell mit dem eigentlichen Modell nichts gemeinsam haben.

   Während der Arbeit war dies zweimal der Fall: Das Design des GEF erfordert, dass Modellelemente eine Liste der *EditParts* (vgl. 5.1.3) enthalten, in denen sie verwendet werden, um diese bei Datenänderungen benachrichtigen zu können. Im GEF werden die graphischen Darstellungen innerhalb des Editors durch die Erstellung eines neuen Modellelementes initiiert, deshalb muss das Modellelement auch die Zielkoordinaten für die spätere graphische Darstellung enthalten.

   Das *Extension-Interface*-Entwurfsmuster [SSRB00] erlaubt die Integration beliebiger Interfaces in die Modellelemente, ohne zusätzliche Veränderungen der Klassenhierarchie.

Wrapper Façade Bestandteil	Modellgegenstück
gesamte Wrapper Façade	`IModel`
konkrete Datentypen	`IConcreteType`
abstrakte Datentypen	`IAdaptableType`
herkömmliche & anpassbare Klassen	`IAbstraction`
Adaptation-Elemente	`IAdaption`
Variablen & Konstanten	`IVariable`
Funktionen	`IFunction`
Funktionsparameter	`IParameter`
Funktionsrückgabewert	`IReturnArg`

Tabelle 5.1: Abbildung der Bestandteile einer Wrapper Façade auf Modellelemente.

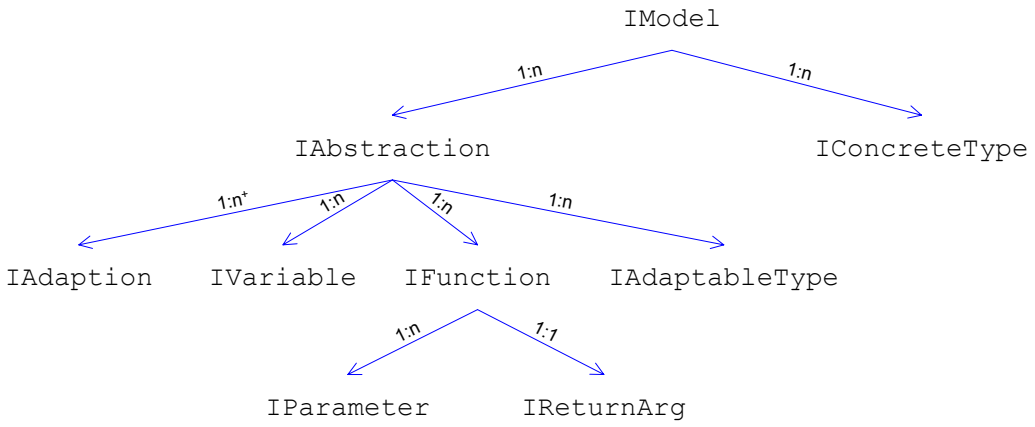

Abbildung 5.1: Hierarchischer Aufbau eines Wrapper Façade Modells.

Um in Eclipse dieses Entwurfsmuster anzuwenden, müssen die Modellelemente das Interface *IAdaptable* implementieren [BG03]. Dieses Interface enthält eine einzige Methode mit folgender Signatur: *Object getAdapter(Class adapter)*. Der Parameter *adapter* wird zur Anforderung eines bestimmten Interfaces benutzt. Die Methode antwortet mit einer Instanz des angeforderten Interfaces oder mit dem Wert *null*. Soll ein (modellfremdes) Interface hinzugefügt werden, wird die Implementierung der *getAdapter()*-Methode entsprechend angepasst. Ein Beispiel zur Anwendung dieser Technik wird in Codefragment 5.1 dargestellt.

3. Bestimmte Datenverarbeitungsschritte erfordern einen Durchlauf der hierarchischen Datenstruktur zur Verarbeitung ausgewählter Modellelemente. Notwendig ist dies z.B. während der Codegenerierung oder während des Lade- und Speichervorgangs. Zur Verarbeitung von Modellelementen während des Durchlaufens des Datenstruktur wird das *Visitor*-Entwurfsmuster [GHJV95] verwendet. Es wird in Abschnitt 5.2 beschrieben.

Codefragment 5.1 Anwendung des *Extension-Interface*-Entwurfsmusters.

```
 1 // Bereitstellung eines Interfaces ueber die getAdapter()-Methode
 2 public Object getAdapter(Class adapter) {
 3   if (adapter == IInitialLocationDelegate.class) {
 4     return locationDelegate;
 5   }
 6   return null;
 7 }
 8
 9 // Zugriff aus das Interface ueber Aufruf der getAdapter()-Methode
10 Object delegate
11   = newElement.getAdapter(IInitialLocationDelegate.class);
12 if (delegate instanceof IInitialLocationDelegate) {
13   ((IInitialLocationDelegate) delegate).setLocation(...);
14 }
```

Abschließend noch eine kurze Beschreibung der Modell-API:

- Die *IModelElement*-Schnittstelle bildet die Wurzel der Modell-API. Alle Modellelemente implementieren diese Schnittstelle.
- Zur Datenmanipulation wird die *setData()*-Methode des jeweiligen Modellelementes aufgerufen. Sie erlaubt die „kontrollierte“ Veränderung eines oder mehrerer Attribute des Modellelementes. Kontrolliert heißt, dass vor einer Veränderung überprüft wird, ob diese mit den Modellinvarianten vereinbar ist. Wird eine Modellinvariante verletzt, so wird eine Fehlermeldung zurückgegeben und es findet keine Veränderung statt.
- Zur Datenabfrage sind entsprechende *get()*-Methoden vorhanden.

- Zur Überprüfung der Modellinvarianten für ein beliebiges Modellelement kann seine *isValid()*-Methode aufgerufen werden.

- Um Elemente in die hierarchische Datenstruktur einzufügen oder sie daraus zu entfernen, können die Methoden der *IContainer*-Schnittstelle benutzt werden. Diese Schnittstelle wird von allen Modellelemente implementiert, die eine Knoten-Rolle in der Daten-Hierarchie einnehmen.

5.1.2 View: Draw2D API

Die Klassen der *View* sind für die graphische Darstellung der Modellelemente verantwortlich. Sie sind mit Hilfe der Draw2D API implementiert.

Alle Elemente der *View* befinden sich im Paket *wfp.draw2d* und sind von der Klasse *Figure* abgeleitet, welche die Grundlage aller graphischen Objekte der Draw2D API darstellt. Sie ermöglicht u.a. folgende Aktionen:

- Das Verändern der Verschachtelung von Figuren, z.B. durch Hinzufügen oder Herausnehmen von Elementen aus einer Figur.

- Das Erstellen von *Tooltips* für eine Figur.

- Das Verändern der Sichtbarkeit, Position und Größe der Figur.

- Den Zugriff auf den *Layoutmanager* einer Figur. Layoutmanager beeinflussen die Anordnung der Subelemente einer Figur.

Die erstellten Elemente der *View* können konzeptuell in zwei Kategorien aufgeteilt werden:

1. Einfache Elemente: Diese sind von der *Label*-Klasse abgeleitet. Ein *Label* ist eine Figur, die zur Darstellung von Text verwendet wird. Optional kann neben dem Text noch eine Graphik angezeigt werden. Labels werden z.B. zur Visualisierung von Variablen und Funktionen verwendet.

2. Komplexe Elemente: Grundlage für diese Elemente ist die *MultiCompartmentFigure*-Klasse, die eine Figur mit einer beliebigen Anzahl von Fächern beschreibt. Die Fächer sind untereinander angeordnet und können andere Elemente aufnehmen. Jedes Fach wird durch einen Rahmen von den anderen getrennt. Zusätzlich hat die Figur einen Titel-Bereich zur Anzeige eines Textes mit Graphik. Der Titel-Bereich ist am oberen Rand der Figur platziert und wird durch eine dunklere Hintergrundfarbe von Rest der Figur abgehoben. Die *MultiCompartmentFigure* ist die Grundlage zur Visualisierung von Klassenelementen.

Abbildung 5.2 illustriert anhand einer Klassendarstellung, wie die verschiedenen Elemente miteinander kombiniert werden.

Die *View* definiert ebenfalls eine eigene API, die zur Veränderung bestimmter Aspekte der graphischen Darstellung benutzt werden kann. Zur Trennung von API-Definition und API-Implementierung wird, wie auch beim Modell, das *Factory*-Entwurfsmuster verwendet.

5.1.3 Controller: EditParts

Zur Erstellung von Controllern stellt das GEF sog. *EditParts* zur Verfügung. Die *EditPart*-Klasse ist für folgende Aufgaben zuständig:

1. Ein *EditPart* ist mit einem Modellelement und einer graphischen Darstellung assoziiert.

   Bei der Erzeugung eines neuen Modellelementes wird über eine Fabrik (*EditPartFactory*) ein entsprechendes *EditPart* erzeugt und diesem das Modellelement zugewiesen. Die Fabrik wird vom Wrapper Façade Editor bereitgestellt (5.1.4). Das neue *EditPart* kann bei Bedarf über seine *getFigure()*-Methode eine graphische Darstellung des Modellelementes erzeugen.

2. Ein *EditPart* übernimmt die Aktualisierung der graphischen Darstellung.

   Um bei Änderungen des Modells informiert zu werden, wird vom *EditPart* das *Observer*-Entwurfsmuster [GHJV95] angewandt. Hierbei meldet sich das *EditPart* bei seinem Modellelement an. Wird das Modellelement verändert, wird daraufhin eine entsprechende Nachricht an das *EditPart* verschickt. Dieses aktualisiert dann ggf. die graphische Darstellung.

3. *EditParts* steuern indirekt die Verschachtelung der graphischen Darstellung über ihre *getModelChildren()*-Methode. Diese Methode gibt eine Menge von Modell(unter)elementen zurück, für die ebenfalls *EditParts* und graphische Darstellungen erzeugt werden. Die Figuren der Modellunterelemente werden in die graphische Darstellung des übergeordneten Modellelementes eingefügt.

4. *EditParts* stellen Befehle zur Veränderung des Modells und der graphischen Darstellung bereit.

   Dies geschieht über folgenden Mechanismus: Ein *EditPart* verfügt über eine oder mehrere *EditPolicy*-Klassen. Eingehende Anfragen (*Requests*) werden von dem *EditPart* empfangen und an seine *EditPolicy*-Klassen weitergeleitet. Diese beantworten die Anfrage entweder mit einem bestimmten Befehl (ein Objekt des Typs *Command*) oder dem Wert *null*. Der zurückgegebene Befehl kann dann vom Absender der Anfrage (in der Regel eine Instanz des graphischen Editors) ausgeführt werden. Dieser Vorgang ist auch in Abbildung 5.3 dargestellt.

Abweichend zu Abbildung 5.3 kann eine eingehende *Request* auch direkt von einem *EditPart* beantwortet werden. Die Verwendung von *EditPolicies* erlaubt jedoch eine Reduzierung von Redundanzen und dient der Wiederverwendung von Quellcode.

5.1.4 Zusammenführung der MVC-Komponenten

Die Model-, View- und Controller-Komponenten werden mit Hilfe des graphischen Wrapper Façade Editors integriert. Der Editor ist von der GEF-Klasse *GraphicalEditorWithPallete* abgeleitet und übernimmt u.a. folgende Aufgaben:

- Er enthält die Wurzel der Modellhierarchie (*IModel*) und die Wurzel der Kontrollerhierarchie (*RootEditPart*).
- Er enthält eine *EditPart*-Fabrik, um bei Veränderung der Modellhierarchie passende *EditParts* instanziieren zu können. Die EditParts erzeugen ihrerseits eine graphische Darstellung (*View*) des entsprechenden Modellelementes und aktualisieren diese bei Modellveränderungen.
- Er definiert eine Menge von Anfragen (*Requests*), die verschiedene Befehle (*Commands*) auslösen können. Die Befehle ermöglichen eine Veränderung des Modells und der graphischen Darstellung. Der genaue Mechanismus wurde bereits beschrieben und in Abbildung 5.3 dargestellt.
- Er verwendet das *Command*-Entwurfsmuster [GHJV95], um ausgeführte Befehle wiederholen oder zurücknehmen zu können. Dies geschieht auf folgende Art und Weise: Ausgeführte Befehle werden auf einem Stapel gespeichert. Die Befehle verfügen über *canUndo()* und *canRedo()* Methoden, über die ermittelt werden kann, ob eine Wiederholung oder Rücknahme möglich ist, und außerdem *undo()* und *redo()* Methoden zur Ausführung der entsprechenden Aktionen.
- Er stellt eine Komponentenleiste, eine Arbeitsfläche und eine Befehlsleiste zur Verfügung (vgl. Abb. 3.1 auf Seite 27).

5.2 Der Codegenerator

Der Codegenerator wandelt das hierarchische Wrapper Façade Modell in Programmcode um. Das Design der Codegenerator-Komponente wurde durch zwei Ziele beeinflusst:

- Der verwendete Codegenerator sollte leicht austauschbar sein. Dies soll die Integration unterschiedlicher Codegeneratoren in das Design-Tool erleichtern und so die Gelegenheit geben, verschiedene Implementierungsverfahren für das Wrapper Façade Entwurfsmuster auszuprobieren.

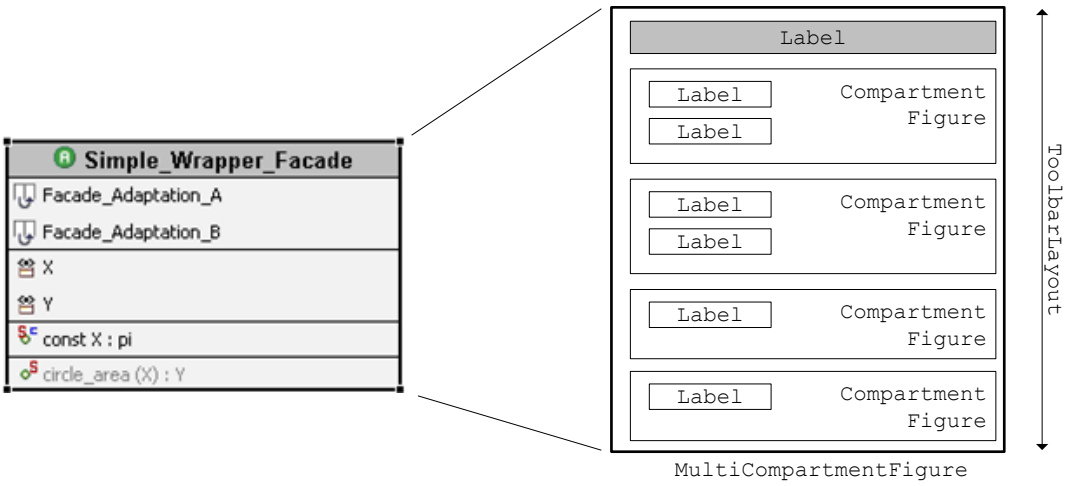

Abbildung 5.2: Aufbau eines komplexen Draw2D-Elementes.

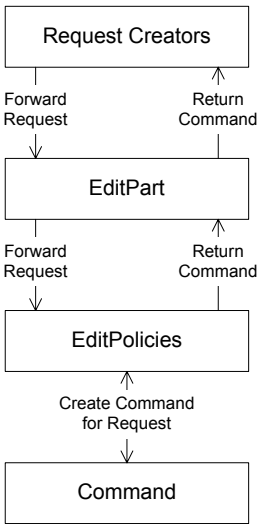

Abbildung 5.3: Kommunikationsablauf bei der Beantwortung von *Requests* durch *EditParts* und *EditPolicies* [MDG+03].

- Der Aufwand bei der Erstellung neuer Codegeneratoren sollte möglichst gering sein, um die Programmierung zusätzlicher Codegeneratoren nicht zu behindern.

5.2.1 Design

Folgendes Design soll den oben erläuterten Zielen gerecht werden:

- Die Anbindung des Codegenerators in das Design-Tool erfolgt über die Schnittstelle *ICodeGenerator*. Sie enthält eine Methode *generateCode(IModel)*, die eine Modelldatenstruktur *IModel* als Parameter annimmt und eine Reihe von Textdateien zurückgibt, welche den generierten Code enthalten.

  Der Aufruf des Codegenerators wird durch einen Kontextmenüeintrag (*Action*) im Design-Tool ausgelöst. Dies läuft wie folgt ab: Beim Aufruf des Kontextmenüeintrags wird eine *Request* abgesetzt, die mit einem *Command*-Objekt beantwortet wird (vgl. Abb. 5.3). Im *Command*-Objekt wird dann eine bestimme Codegeneratorinstanz erzeugt, an die die Modelldatenstruktur übergeben wird.

  Zur Integration eines neuen Codegenerators in das Design-Tool sind also folgende Schritte notwendig: Der Codegenerator muss die *ICodegenerator*-Schnittstelle implementieren. Danach wird eine neue *Action*- und eine neue *Command*-Klasse erstellt, um den Codegenerator in das Kontextmenü des Design-Tools einbinden zu können.

- Bei der Erzeugung von Programmtext aus einem Wrapper Façade Modell, muss die hierarchische Modelldatenstruktur ein oder mehrere Male durchlaufen werden, um die benötigten Informationen zu extrahieren. Der Durchlauf des Modells ist eine wiederkehrende Aufgabe, die bei der Erstellung jedes neuen Codegenerators zu erledigen ist.

  Um die Erstellung neuer Codegeneratoren zu erleichtern wurde das *Visitor*-Entwurfsmuster [GHJV95] benutzt: Durch Anwendung dieses Entwurfsmusters kann die Programmlogik zum Durchlaufen des Modells von der Codegenerierung getrennt werden und muss deshalb nicht von jedem Codegenerator neu implementiert werden. Das Entwurfsmuster erlaubt außerdem die Ausführung von beliebigen Operationen auf eines, bestimmte oder alle Elemente der Modelldatenstruktur, während diese durchlaufen wird. Dies ist bei der Implementierung eines Codegenerators sehr nützlich, da je nach Modellelement (z.B. Variable oder Funktion) unterschiedliche Aktionen auszuführen sind bzw. unterschiedlicher Code erstellt werden soll.

  Durch die Anwendung des *Visitor*-Entwurfsmusters wurde der Code zum Modelldurchlauf in die Implementierung des Modells integriert. Um bei der Erstellung eines Codegenerators das Entwurfsmuster zu benutzen, muss ein Codegenerator eine *Visitor*-Klasse implementieren und eine Instanz dieser an das

Modell übergeben. Beim Durchlaufen des Modells werden Methoden der *Visitor*-Instanz aufgerufen. Der genaue Mechanismus und der Aufbau des *Visitor*-Entwurfsmusters werden im nächsten Abschnitt beschrieben.

5.2.2 Das Visitor-Entwurfsmuster

Das *Visitor*-Entwurfsmuster ermöglicht die Anwendung beliebiger Operationen auf Elemente einer hierarchischen Datenstruktur während diese durchlaufen wird.

Das Entwurfsmuster kann in mehrere Bestandteile aufgeteilt werden, die an dieser Stelle beschrieben werden sollen. Der in Klammern gesetzte Name, neben den einzelnen Bestandteilen, bezeichnet die dazugehörige Implementierungsklasse, die im Paket *wfp.model* zu finden ist. Die Beziehung der Bestandteile zu einander ist in Abbildung 5.4 dargestellt.

Die Bestandteile des *Visitor*-Entwurfsmusters sind [GHJV95]:

- Visitor-Schnittstelle (*IModelElementVisitor)*:

  Die *Visitor*-Schnittstelle deklariert eine *accept*-Methode für jeden Elementtyp der Modellhierarchie.

  Der Parameter der *accept*-Methode ermöglicht die Übergabe eines gleichnamigen Elementtypen, d.h. an die *acceptModel*-Methode kann ein Element des Typs *IModel* übergeben werden, an die *acceptVariable*-Methode ein Element des Typs *IVariable* und entsprechend für alle weiteren Elementtypen.

  Während des Durchlaufs der hierarchischen Datenstruktur werden die *accept-Methoden* aufgerufen. Auf diese Weise, wird dem *Visitor* das aktuell besuchte Element, als Parameter einer *accept*-Methode übergeben und so eine weitere Bearbeitung des Elementes im *Visitor* ermöglicht.

- Visitor-Implementierung (z.B. *DefaultVisitor*):

  Er stellt eine Implementierung für alle von der *Visitor*-Schnittstelle deklarierten Methoden zur Verfügung. Die Implementierung einer *accept*-Methode enthält eine Anzahl von Operationen, die auf den entsprechenden Elementtyp angewendet werden sollen. Während des Durchlaufs der hierarchischen Datenstruktur kann die *Visitor*-Implementierung auch Kontext- und Zustandsinformationen speichern.

  Oft soll eine *Visitor*-Implementierung erstellt werden, die nur bestimmte Elementtypen verarbeitet. Deshalb wird die Implementierung der *Visitor*-Schnittstelle durch folgenden Mechanismus erleichtert: Es wird eine Standardimplementierung erstellt (*DefaultVisitor*-Klasse), in der die einzelnen *accept*-Methoden einen leeren Methodenkörper haben. Durch Erweiterung der *DefaultVisitor*-Klasse können neue Implementierungen erstellt werden, die nur diejenigen

accept-Methoden enthalten müssen, die im jeweiligen Anwendungsfall relevant sind.

- Element-Schnittstelle (*IModelElementVisitorCallback)*:

  Eine Schnittstelle, welche eine *acceptElement*-Methode deklariert, die zur Annahme einer *Visitor*-Instanz verwendet wird. Diese Schnittstelle muss von allen Elementtypen der Modellhierarchie implementiert werden, die dem *Visitor* zugänglich sein sollen.

- Implementierung der Element-Schnittstelle (*IModel*, *IVariable*):

  Die Typen der Modellhierarchie müssen die *Element*-Schnittstelle implementieren, um eine *Visitor*-Instanz annehmen zu können.

  Eine Implementierung der *acceptElement*-Methode übernimmt in der Regel zwei Aufgaben: Sie ruft die *accept*-Methode der übergebenen *Visitor*-Instanz auf (vgl. Abb. 5.4) und übermittelt danach den *Visitor* an ihre Unterelemente (falls notwendig).

- objektorientierte Datenstruktur (*IModel*-Instanz):

  Dies ist die Wurzel-Instanz der hierarchischen Datenstruktur. Sie kann einen *Visitor* annehmen und auf ihre Unterelemente anwenden.

Die vorangegangene Ausführung soll durch Codefragment 5.2 verdeutlicht werden: Im dargestellten Beispiel wird zunächst eine konkrete Implementierung der *Visitor*-Schnittstelle *IModelElementVisitor* erstellt. Sie hat die Aufgabe, die Namen aller Variablenelemente aus einer Modelldatenstruktur auszudrucken (Zeilen 3–7). Um nicht sämtliche *accept*-Methoden implementieren zu müssen, wird die Implementierung durch Ableiten der *DefaultVisitor*-Klasse erzeugt. In Zeile 9 wird der erstellte *Visitor* an die Modelldatenstruktur übergeben und so der Durchlauf der Modellhierarchie gestartet. Während des Durchlaufs wird für jedes Modellelement von Typ *IVariable* die in Zeilen 4–6 dargestellte Methode des Visitors aufgerufen.

5.3 Speicherung der Modelldaten

Vorraussetzung für eine sinnvolle Nutzung des Modellierungstools ist die Möglichkeit das erstellte Modell auf persistenten Medien zu speichern und zu einem späteren Zeitpunkt wiederherstellen zu können.

5.3.1 Datenformat

Bei der Wahl des Datenformats wurde XML (Extended Markup Language) bevorzugt und zwar aus folgenden Gründen:

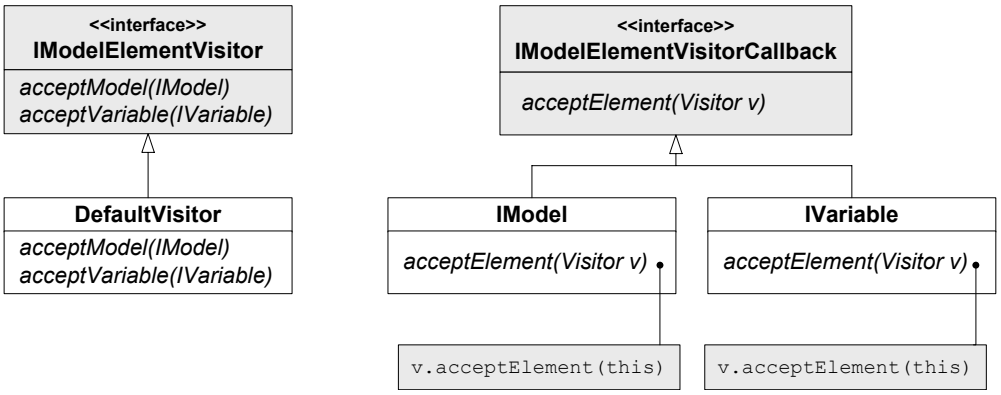

Abbildung 5.4: Struktur des *Visitor*-Entwurfsmusters in UML-Notation.

Codefragment 5.2 Implementierung und Verwendung einer Visitor-Klasse zum Ausdrucken aller Klassen-Variablen aus der Modelldatenstruktur.

```
 1 public static void printVariables(IModel aModel) {
 2   // Implementierung und Instanziierung einer neuen Visitor-Klasse
 3   IModelElementVisitor visitor = new DefaultVisitor() {
 4     public void acceptVariable(IVariable element) {
 5       System.out.println(element.getName());
 6     }
 7   };
 8   // Durchlaufen des Modells mit parallelem Aufrufen des Visitors
 9   aModel.acceptElement(visitor);
10 }
```

1. Der XML-Standard [BPSM+04] stellt mit der *Dokument Type Definition* (DTD) ein Spezifikationsformat zur Beschreibung hierarchischer XML-Dokumente zur Verfügung. Mit Hilfe der DTD wurde das im Editor verwendete Datenmodell auf eine hierarchische Dokumentendefinition abgebildet. Sie ist in Codefragment 5.3 dargestellt. Während des Speichervorgangs wird aus dem objektorientierten, hierarchischen Datenmodell des Editors eine DTD-konforme XML-Datei erstellt. Sie ist in Codefragment 5.4 abgebildet.

2. Eine XML-Datei kann gegen eine vorhandene DTD verifiziert werden. Während der Analyse einer XML-Datei kann ein validierender Parser die DTD zur Integritätsüberprüfung verwenden. Von dieser Möglichkeit wurde bei der Implementierung des Ladevorgangs Gebrauch gemacht, um beschädigte oder unvollständige Dateien leicht erkennen zu können.

3. Die Verarbeitung von XML-Dokumenten wird von Java sehr gut unterstützt. Dadurch wurde insbesondere die Implementierung des Ladevorgangs beschleunigt: Während des Ladevorgangs wird die XML-Beschreibung einer Wrapper Façade in das Datenmodell des Editors zurückgewandelt. Durch Verwendung der *Dokument Object Model* (DOM) Funktionalität – hierbei stellt Java eine objektorientierte Repräsentation des eingelesenen XML-Dokumentes bereit – konnte die Rekonstruktion des Datenmodells aus den XML-Daten ohne großen Programmieraufwand erfolgen.

4. XML-Dateien sind in einem zeichenbasierten Format (z.B. Unicode) abgespeichert, können also in einem herkömmlichen Texteditor betrachtet und einfach weiterverarbeitet werden.

5.3.2 Implementierung

Es soll nun kurz die Implementierung des Speicher- und Ladevorgangs beschrieben werden.

Während des Speichervorgangs wird das bereits bekannte *Visitor*-Entwurfsmuster benutzt. Hierbei genügt ein Aufruf der *acceptElement()*-Methode des Wurzelelementes der Modelldatenstruktur, um einen hierarchischen Durchlauf der Datenstruktur zu initiieren. An die *acceptElement()*-Methode wird eine *Visitor*-Instanz übermittelt, die dafür sorgt, dass für jedes Modellelement ein passendes XML-Element erzeugt wird. Nachdem die Datenstruktur vollständig durchlaufen ist, werden die erzeugten XML-Daten in eine Datei geschrieben.

Der Ladevorgang kann konzeptionell in zwei Schritte unterteilt werden: der Analyse der XML-Datei und der Rekonstruktion des Datenmodells. Die XML-Datei wird mit Hilfe einer *DocumentBuilder*-Instanz geparst. Der *DokumentBuilder* liest die XML-Datei ein, benutzt die DTD um die eingelesenen Daten zu validieren und erstellt dar-

aus ein *Document Object Model* (DOM). Das DOM ist ein objektorientiertes Abbild des XML-Dokumentes. Es bietet eine API an, um auf die in ihm gespeicherten XML-Daten zuzugreifen. Über diese API werden die XML-Daten genutzt, um daraus das Datenmodell für den graphischen Editor zu rekonstruieren.

Während des Ladevorgangs müssen die Abhängigkeiten zwischen den Modellelementen beachtet werden, um Fehler bei der Rekonstruktion zu vermeiden. Dies geschieht durch eine geschickte Wahl der Rekonstruktionsreihenfolge und Anwendung einer 2-Pass-Strategie. Im ersten Durchlauf werden die Elemente rekonstruiert, die unabhängig voneinander sind (Datentypen und Klassen). Die Rekonstruktion von Elementen, die auf andere aufbauen (ein Variablenelement braucht einen bestimmten Datentyp), und die Wiederherstellung von Abhängigkeiten (Klasse X erbt von Klasse Y), findet im zweiten Durchlauf statt. Zuletzt wird das erstellte Datenmodell an den Editor zurückgegeben.

Codefragment 5.3 Definition des Datenformats (ohne Element-Attribute).

```
<!DOCTYPE model [
<!ELEMENT model (ctype*, abstraction*)>
  <!ELEMENT ctype (includearg*)>
    <!ELEMENT includearg EMPTY>
  <!ELEMENT atype EMPTY>
  <!ELEMENT abstraction (adaption*, atype*, variable*, function*)>
    <!ELEMENT adaption EMPTY>
    <!ELEMENT variable EMPTY>
    <!ELEMENT function (parameter*, returnarg?, body?)>
      <!ELEMENT parameter EMPTY>
      <!ELEMENT returnarg EMPTY>
      <!ELEMENT body (#PCDATA)>
]>
```

Codefragment 5.4 XML-Repräsentation der Simple Wrapper Façade aus Abschnitt 3.4.

```
<?xml version='1.0' encoding='iso-8859-1'?>
<!-- Hier steht die Definition des Datenformats (s.o.) -->
<model dtdversion="0.1">
  <abstraction name="Simple_Wrapper_Facade"
           typedefto="INTERFACE" isClass="false">
    <adaption name="Facade_Adaptation_A" symbol="SYS_A"/>
    <adaption name="Facade_Adaptation_B" symbol="SYS_B"/>
    <atype  name="X"/>
    <atype  name="Y"/>
    <variable name="pi"
           visibility="public" const="true" static="true"
           arraymods="" pointermods="" typeref="X"
           defaultvalue="= api_t::pi;"/>
    <function name="circle_area"
           visibility="public" static="true" inline="true"
           constcallable="false">
      <parameter name="radius"
             const="false" arraymods="" pointermods=""
             typeref="X"/>
      <returnarg const="false" arraymods="" pointermods=""
             typeref="Y"/>
      <body>{ return api_t::circle_area(radius); }</body>
    </function>
  </abstraction>
</model>
```

Kapitel 6

Zusammenfassung

6.1 Ergebnisse

Die Leistung der vorliegenden Arbeit kann in folgenden Punkten zusammengefasst werden:

1. Zunächst wurde ein objektorientiertes Datenmodell konzipiert, das zur Beschreibung von C++ basierten Wrapper Façades verwendet werden kann. Neben den üblichen Wrapper Façade Elementen, also Datentypen, Klassen, Methoden und Variablen, kann damit auch die Abstraktions- und Anpassungsschicht beschrieben werden.

2. Das Datenmodell wurde mit Hilfe einer graphischen Darstellung visualisiert. Dazu wurde für jedes Modellelement ein entsprechendes graphisches Element konzipiert.

3. Mit Hilfe des *Graphical Editing Frameworks* wurden Modell und Darstellung in ein Design-Tool integriert, das die Erstellung von Wrapper Façades ermöglicht.

4. Ergänzend dazu wurde ein Codegenerator erstellt, der das Wrapper Façade Modell in C++ Quellcode umwandelt und hierbei das Template-Verfahren (2.6.4) anwendet. Um die Verwendung unterschiedlicher Implementierungsverfahren zu ermöglichen, wurde der Codegenerator so konzipiert, dass eine neue Implementierung leicht zu erstellen und mit geringem Aufwand in das Design-Tool zu integrieren ist.

5. Ein Import-Tool erlaubt die Extraktion von Methoden-, Funktions- und Variablendeklarationen aus existierendem C++ Quellcode und deren Importierung in das Design-Tool. Dadurch soll die Erstellung von Wrapper Façades zu beschleunigt werden.

6. Zur persistenten Speicherung des Wrapper Façade Modells wurde ein XML-basiertes Datenformat entworfen und verwendet.

7. Um die Funktionsfähigkeit der Software zu demonstrieren wurden zwei Anwendungsbeispiele entwickelt: Mit der Simple Wrapper Façade wurde ein kleines aber vollständiges Beispiel geschaffen, das alle wesentlichen Elemente einer Wrapper Façade enthält. Die Socket Wrapper Façade stellt ein komplexeres Anwendungsbeispiel dar. Hierbei handelt es sich um plattformunabhängige Klassenbibliothek zur Socket-Kommunikation, die mit Hilfe des Wrapper Façade Entwurfsmusters implementiert wurde.

6.2 Verwandte Arbeiten

Allgemeine Überlegungen über Sprach- und Toolunterstützung für Entwurfsmuster werden anhand einer Gegenüberstellung drei unterschiedlicher Ansätze in [CHV00] beschrieben.

Für die Entwurfsmuster aus *Gamma et. al.* [GHJV95] wird in [BFVY96] ein Codegenerator vorgestellt. In [FL00] wird ein Modellierungstool beschrieben, mit dem diese Entwurfsmuster zu Programmen zusammengefügt werden können.
Eine Software, die Entwurfsmuster in Smalltalk-Programmen identifiziert und deren Reorganisation unterstützt wurde von [FMvW97] vorgestellt. Mit der Analyse von C++ Programmen und der automatischen Erkennung von Entwurfsmustern beschäftigen sich [PK98] und [FGMP].

Die Probleme, die bei der Speicherung der Ergebnisse softwaregestützter Codeanalysen von C++ Programmen in XML-basierten Datenformaten zu bewältigen sind, werden in [MK00], [FGS+01] und [GK02] besprochen. In [BG00], [PM02] und [FBTG] werden drei Datenformate vorgestellt, die in Analysetools für C++ Programme verwendet worden sind.

Literaturverzeichnis

[BFVY96] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, 1996.

[BG00] Marat Boshernitsan and Susan L. Graham. Designing an XML-Based Exchange Format for Harmonia. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, page 287. IEEE Computer Society, 2000.

[BG03] Kent Beck and Erich Gamma. *Contributing to eclipse: Principles, Patterns, and Plug-Ins*. The eclipse series. Addison-Wesley, October 2003.

[BK03] Wolfgang Blochinger and Wolfgang Küchlin. Cross-Platform Development of High Performance Applications Using Generic Programming. In T. Gonzales, editor, *Proc. of the IASTED Intl. Conference Parallel and Distributed Computing and Systems (PDCS 2003)*, volume 2, pages 654–659, Marina del Rey, CA, USA, November 2003. ACTA Press.

[Blo01] Joshua Bloch. *Effective Java Programming Language Guide*. Addison-Wesley, June 2001.

[Blo02] Wolfgang Blochinger. *Eine Systemumgebung zur Erstellung paralleler C++ Programme und deren Ausführung in heterogenen verteilten Systemen*. PhD thesis, Wilhelm-Schickard-Institut f. Informatik, Universität Tübingen, 2002. Chapt. 4.

[BMR97] Frank Buschmann, Regine Meunier, and Hans Rohnert. *A System of Patterns*, volume 1 of *Pattern-Oriented Software Architecture*. John Wiley and Sons, March 1997.

[BPSM+04] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau (editors). Extensible Markup Language (XML) 1.0 – W3C Recommendation. 3rd edition. http://www.w3.org/TR/2004/REC-xml-20040204, February 2004.

[CHV00] Craig Chambers, Bill Harrison, and John Vlissides. A debate on language and tool support for design patterns. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 277–289. ACM Press, 2000.

[EFB01] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001.

[EKS03] Markus Eiglsperger, Michael Kaufmann, and Martin Siebenhaller. A Topology-Shape-Metrics Approach for the Automatic Layout of UML Class Diagrams. *Proceedings of the ACM 2003 Symposium on Software Visualization (SOFTVIS'2003)*, pages 189–198, 2003.

[FBTG] Rudolf Ferenc, Arpad Beszedes, Mikko Tarkiainen, and Tibor Gyimothy. Columbus – Reverse Engineering Tool and Schema for C++. http://citeseer.ist.psu.edu/542146.html.

[FGMP] Rudolf Ferenc, Juha Gustafsson, László Müller, and Jukka Paakki. Recognizing Design Patterns in C++ programs with the integration of Columbus and Maisa. http://citeseer.ist.psu.edu/460179.html.

[FGS+01] Rudolf Ferenc, Tibor Gyimóthy, Susan Elliott Sim, Richard C. Holt, and Rainer Koschke. Towards a Standard Schema for C/C++. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 49. IEEE Computer Society, 2001.

[FL00] Peter Forbrig and Ralf Lämmel. Programming with Patterns. In *Proceedings TOOLS-USA 2000*. IEEE, 2000.

[FMvW97] Gert Florijn, Marco Meijers, and Pieter van Winsen. Tool support for object-oriented patterns. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP'97, Jyvaskyla, Finland, June 1997*, volume 1241 of *LNCS*, pages 472–495, 1997.

[Fou04] The Eclipse Foundation. Eclipse webpage. http://www.eclipse.org, May 2004.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Objected-Oriented Software*. Addisson-Wesley, 1995.

[GK02] Katsuhiko Gondow and Hayato Kawashima. Towards ANSI C Program Slicing using XML. In Mark van den Brand and Ralf Lämmel, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier, 2002.

[Krä99] Thomas Krätschmer. Entwurf und Implementierung einer plattformunabhängigen C++ Klassenbibliothek. Lehrstuhl für Symbolisches Rechnen. Wilhelm-Schickard-Institut f. Informatik, Universität Tübingen, July 1999.

[MDG+03] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, and Phillipe Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. Number SG24-6302-00 in Redbooks. IBM International Technical Support Organization, preliminary edition, October 2003.

[MK00] E. Mamas and K. Kontogiannis. Towards Portable Source Code Representations Using XML. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, page 172. IEEE Computer Society, 2000.

[Moo90] James D. Mooney. Strategies for supporting application portability. *IEEE Computer*, 23(11):59–70, November 1990.

[Mye95] Nathan C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.

[PK98] L. Prechelt and C. Krämer. Functionality versus Practicality: Employing Existing Tools for Recovering Structural Design Patterns. *Journal of Universal Computer Science*, 4(12):866–882, Dec. 1998.

[PM02] J. F. Power and B. A. Malloy. Program Annotation in XML: A Parse-Tree Based Approach. In *Proceedings of the Ninth Working Conference on Reverse Engineering WCRE'02)*, page 190. IEEE Computer Society, 2002.

[SC92] Henry Spencer and Geoff Collyer. #ifdef considered harmful, or portability experience with C news. In *USENIX, Summer 1992 Technical Conference*, pages 185–197, San Antonio (Texas), June 1992.

[Sch92] Douglas C. Schmidt. IPC SAP: A Family of Object-Oriented Interfaces for Local and Remote Interprocess Communication. *C++ Report*, November/December 1992.

[Sch98] Douglas C. Schmidt. An Architectural Overview of the ACE Framework: A Case-study of Successful Cross-platform Systems Software Reuse. *USENIX* `login` *magazine*, November 1998. Tools special issue.

[Sch99] Douglas C. Schmidt. Wrapper Facade – A Structural Pattern for Encapsulating Functions within Classes. *C++ Report*, 11(2), February 1999.

[Sch04] Douglas C. Schmidt. Overview of ACE. http://www.cs.wustl.edu/ schmidt/ACE-overview.html, May 2004.

[SDF+03] Sherry Shavor, Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. *The Java Developer's Guide to Eclipse*. Addison-Wesley, September 2003.

[See97] Jochen Seemann. Extending the Sugiyama algorithm for drawing UML class diagrams: Towards automatic layout of object-oriented software diagrams. In G. DiBattista, editor, *Proc. Graph Drawing, 5th International Symposium, GD '97, Rome, Italy, September, 1997*, volume 1353 of *LNCS*. Springer, 1997.

[SH01] Douglas C. Schmidt and Stephen D. Huston. *C++ Network Programming Vol. 1– Mastering Complexity with ACE and Patterns*. C++ In-Depth Series. Addisson-Wesley, 2001.

[Sin97] Alok. K. Sinha. *Netzwerkprogrammierung unter Windows NT 4.0: Systemarchitektur und Methoden der Interprozeß-Kommunikation*. Addison-Wesley, 1997.

[SSRB00] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Patterns for Concurrent and Distributed Objects*, volume 2 of *Pattern-Oriented Software Architecture*, chapter Wrapper Facade, pages 47–74. John Wiley and Sons, 2000.

[Ste98] W. R. Stevens. *Networking APIs: Sockets and XTI*, volume 1 of *UNIX Network Programming*. Prentice Hall, Upper Saddle River, NJ, 2nd edition, 1998.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

[VJ02] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 1st edition, 2002.