Technische Universität Berlin — Fakultät IV - Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik

Diplomarbeit

bei Prof. Dr.-Ing. Stefan Jähnichen

Anpassbarkeit komplexer Software-Module durch ObjectTeams/C++ am Beispiel von graphischen Benutzungsschnittstellen

Angefertigt unter der Leitung und Betreuung von
Dr.-Ing. Stephan Herrmann

vorgelegt von

Carsten Pfeiffer
<carp@cs.tu-berlin.de>
<carsten.pfeiffer@first.fraunhofer.de>

10. Juni 2003

Erklärung:

Die selbständige und eigenhändige Anfertigung versichere ich an Eides statt.

Berlin, den 10. Juni 2003

____

Unterschrift

Inhaltsverzeichnis

Einleitung

Eine funktionierende IT-Infrastruktur ist von immenser Bedeutung; sowohl im öffentlichen Bereich als auch in der Industrie werden Software-Systeme eingesetzt, die — würden sie ausfallen — den gesamten Betrieb lahmlegen würden. Software sollte daher höchsten Qualitätsansprüchen genügen, um solche Ausfälle zu verhindern.

Da Software jedoch immer unfangreicher wird und damit auch ihre Komplexität steigt, ist es schwierig all den Anforderungen genüge zu tun, die da wären:

- Sie muss zuerst einmal die geforderte *Funktionalität* bieten.
- Die Funktionalität sollte möglichst *performant* geboten werden. Eine Umsetzung der funktionalen Anforderungen ist nicht genug — wenn ein Softwaresystem diese nicht mit adäquatem Ressourcenverbrauch zur Verfügung stellen kann, ist es ungeeignet.[1]
- Da häufig Standardsoftware eingesetzt wird, muss diese in hohem Maße *konfigurierbar* sein, um allen Wünschen gerecht zu werden. Ist dies nicht oder nur unzureichend möglich, wird eine individuelle *Anpassung* der Software an die Bedürfnisse nötig sein.
- Software unterliegt einem Evolutionsprozess. Sie muss gelegentlich aktualisiert, d.h. *gewartet* werden, um von Fehlern bereiningt oder an neue Umgebungen angepasst zu werden sowie um eventuelle neue Standards zu unterstützen.
- Insbesondere das spätere Hinzufügen von Features, die im originalen Entwurf nicht vorgesehen waren, verursacht häufig Probleme.

Um der hohen Komplexität Herr zu werden und dabei eine höchstmögliche Flexibilität der Software zu erreichen wird die Software in kleine, überschau- und beherrschbare Einheiten aufgeteilt. Nicht jeder einzelne Entwickler muss das Gesamtsystem verstehen; er kann sich auf die Entwicklung einzelner Module beschränken.

Eine hohe Anpassbarkeit kann u.a. dadurch erreicht werden, dass diese einzelnen Module nicht stark, sondern eher lose gekoppelt werden. Ein Ereignisbenachrichtungssystem als Kommunikationsmedium zwischen den Modulen ist eine Möglichkeit, um eine solche lose Kopplung zu erzielen. Damit wird ermöglicht, auch nachträglich neue Module

[1] Der Report über das elektronische Fahnundungs- und Informationssystem INPOL-neu des Bundeskriminalamtes demonstrierte dies eindrucksvoll [1].

in den Informationsfluss einzuhängen und somit Softwareerweiterungen einzuschleusen, ohne invasiv Änderungen am Gesamtsystem durchzuführen.

Das Programmiermodell Object Teams [2, 3] definiert u.a. einen Weg, existierende Software a-posteriori zu adaptieren, ohne deren Code direkt zu verändern. Im Rahmen dieser Diplomarbeit wurde Object Teams für die Sprache C++ umgesetzt sowie erste Erfahrungen mit dessen Anwendung hinsichtlich der Modularisierung von komplexen, graphischen Benutzungsschnittstellen gesammelt. Die Wahl der Sprache C++ ist begründet durch die Verwendung des C++-Toolkits Qt als Grundlage für graphische Anwendungen.

Struktur der Arbeit Kapitel 1 erörtert die Grundlagen entkoppelter, ereignisbasierter Programmierung, mit Fokus auf den vom Qt-Toolkit bereitgestellten Kommunikationsmechanismen.

Kapitel 2 thematisiert die Konzepte von aspektorientierter Softwareentwicklung als Mechanismus für die erweiterte Modularisierung von Software. Weiterhin wird die Theorie des neuen Programmiermodells Object Teams vorgestellt.

Technologien für Aspektorientierung mit C++ werden in Kapitel 3 diskutiert.

In Kapitel 4 wird die Realisierung von Object Teams mit Hilfe des im vorigen Kapitel vorgestellten Toolkits OpenC++ beschrieben.

Kapitel 5 stellt eine komplexe, ereignisbasierte Anwendung mit graphischer Benutzungsoberfläche vor. Die unterschiedlichen Möglichkeiten von Object Teams werden anhand dieser Anwendung evaluliert, wobei die Vor- und Nachteile beider Ansätze herausgearbeitet werden.

In Kapitel 6 wird ein Resumée gezogen über die unterschiedlichen Ereignisbenachrichtigungssysteme sowie die Eignung dieser für Anpassbarkeit von komplexer Software.

1. Event-basierte GUI-Programmierung

In unserer komplexen Welt geschieht ständig etwas, laufend treten Ereignisse ein. Jeder einzelne Mensch interessiert sich jedoch nur für eine geringe Teilmenge all dieser Ereignisse. Die Masse der Ereignisse muss somit gefiltert werden. Es bestehen die Möglichkeiten, regelmäßig zu überprüfen, ob ein Ereignis eingetreten ist oder sich von jemandem benachrichtigen zu lassen, sobald ein Ereignis eintritt. Während ersteres einen hohen Zeit- und Arbeitsaufwand für jeden einzelnen bedeutet, erfordert zweiteres, dass andere wissen, wen sie über was informieren sollen.

Auch wenn es oft mehrere Quellen eines Ereignisses gibt, so ist die Herkunft nicht notwendigerweise von Interesse — allein die Tatsache, dass das Ereignis eingetreten ist ist von Bedeutung. Idealerweise würde man nur die Arten von Ereignissen „abonnieren", für die man sich interessiert, so dass man benachrichtigt wird, wenn sie eintreten.

Wie in der realen Welt ist Ereignisbenachrichtigung auch in Computersystemen ein nützliches Kommunikationsparadigma. Dieses Kapitel beschreibt die grundlegenden Konzepte und Probleme der Ereignisbenachrichtigung und stellt verschiedene Strategien mit all ihren Vor- und Nachteilen dafür vor.

1.1. Model-View-Controller

Programme mit grafischen Benutzeroberflächen verwenden häufig eine Model-View-Controller-Architektur (MVC), da sie Daten, Funktionalität und grafische Repräsentation als getrennte Entitäten betrachtet und sich diese Modularisierung als sehr geeignet herausgestellt hat. Die Daten werden hier als *Model* repräsentiert, Sichten darauf als *Views*. Benutzereingaben per Tastatur, Maus ö.ä. werden vom *Controller* entgegengenommen und verarbeitet. Die Separierung von View und Controller ist mit aktuellen Toolkits nicht mehr ganz so eindeutig gegeben; grafische Elemente (Widgets) werden bereitgestellt, die Benutzereingaben selbst verarbeiten und die Anwendung ggf. über Zustandsänderungen informieren. Allgemeiner behandelt der Controller daher Ereignisse der Views [4].

Bei Änderungen des Modells müssen selbstverständlich die Sichten[1] informiert werden, damit sie die geänderten Daten reflektieren. Dies könnte vom Controller übernommen werden. Bei jeder Änderung, die er am Modell vornimmt muss er die Sichten informieren. Bei diesem Verfahren wird das Modell als „Passives Modell" bezeichnet. [5]

[1] Es kann mehrere Sichten auf ein Modell geben

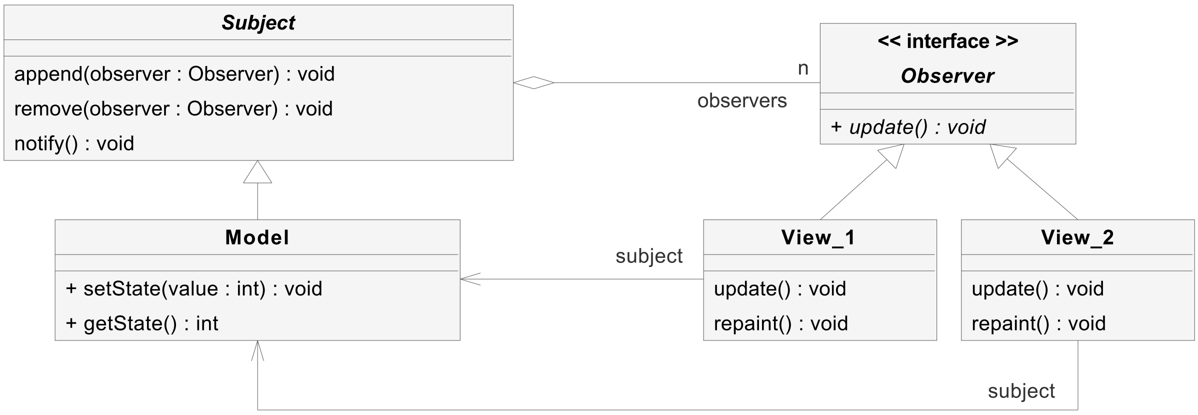

Abbildung 1.1.: Observer in einem MVC Modell

1.2. Observer

Offensichtlich ist das im vorangehenden Abschnitt beschriebene passive Modell jedoch nicht besonders skalierbar — sobald das Modell von anderer Stelle als vom Controller modifiziert würde, müsste die Sicht auch von dieser Stelle benachrichtigt werden. Da nur das Modell „weiß", wann es verändert wird, ist es naheliegend, das Modell mit den Sichten zu koppeln, so dass das Modell die Sichten zum Aktualisieren (*Updaten*) bewegt.

Dafür hat sich das publish/subscribe-Muster (*Design Pattern*), auch *Observer* genannt, bewährt [6]. Hierbei registrieren sich die Sichten als Interessenten beim Modell. Das Modell verwaltet eine Liste der Sichten (`observers` in Abbildung 1.1) und ruft bei Modelländerungen eine `update()`-Methode auf jeder einzelnen Sicht auf, die sie zum Neuzeichnen mit den aktualisierten Daten des Modells veranlasst. Dadurch, dass das Modell nicht mehr nur für die Datenhaltung zuständig ist, sondern Interessenten *aktiv* über Änderungen an den Daten informiert, bezeichnet man es als „Aktives Modell".

Abbildung 1.1 skizziert den Aufbau des Observer-Musters innerhalb einer MVC Architektur (der Controller ist der Vereinfachung halber weggelassen). Wichtigste Merkmale sind die abstrakte Oberklasse `Subject` und das Interface `Observer` mit der abstrakten `update()`-Methode. Erstere implementiert die generische Behandlung von `Observer`-Instanzen[2], d.h. deren Verwaltung (Registrieren, Deregistrieren) sowie das Aufrufen von `update()` aus der `notify()`-Methode.

Instanzen von `View_1` oder `View_2`, die das `Observer` Interface implementieren, besitzen eine Referenz auf das `Subject`, d.h. auf eine Instanz des davon erbenden `Models`. In ihrer `update()`-Implementierung erfragen sie den aktuellen Zustand des Models und stellen diesen dann dar.

Abbildung 1.2 auf der nächsten Seite zeigt den Nachrichtenfluss der beim Observer-Muster beteiligten Objekte.

Nachdem sich die beiden Sichten `view1` und `view2` beim Modell als `Observer` registriert haben, löst ein `Controller` eine Zustandsänderung des Modells durch den Aufruf

[2]Ich folge in dieser Arbeit den nicht irren könnenden „Millionen von Fliegen" und benutze das geläufigere Wort *Instanz* anstatt der vielleicht korrekteren Übersetzung von instance *Exemplar*.

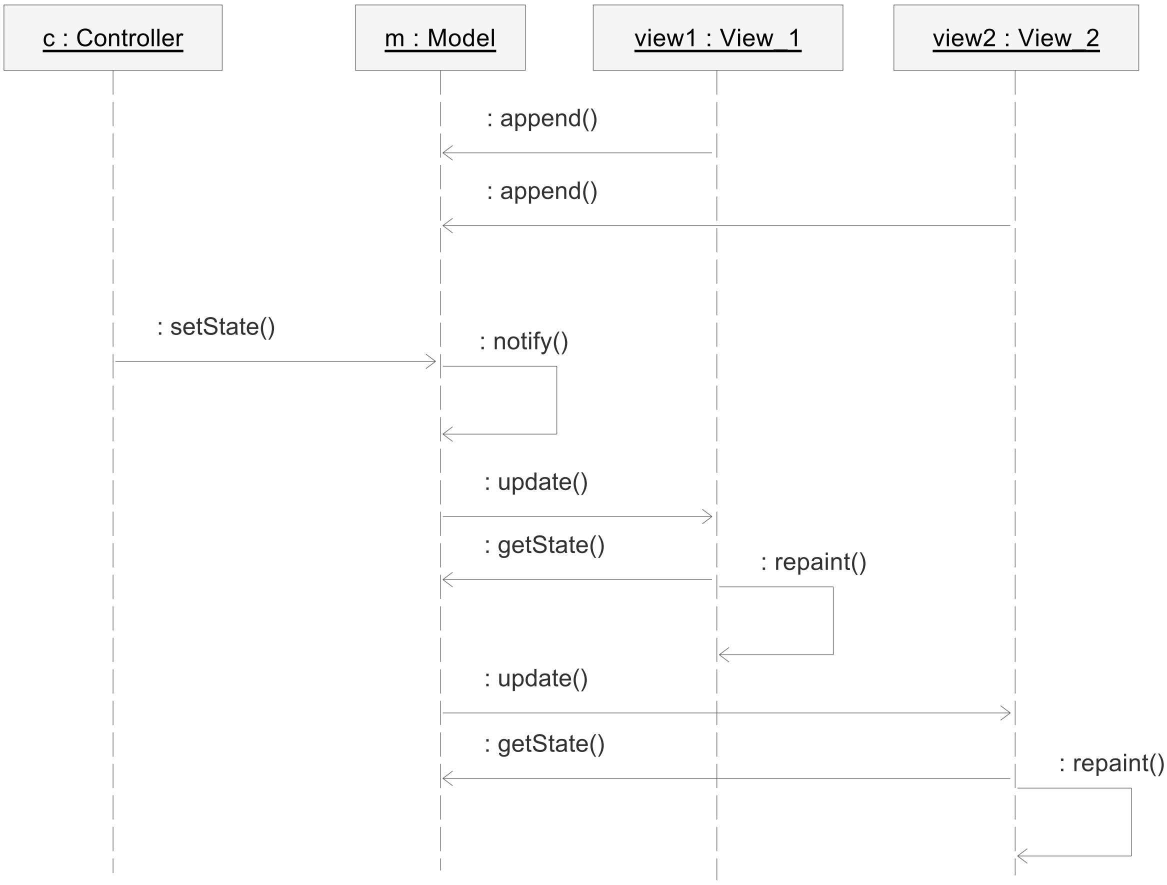

Abbildung 1.2.: Sequenzdiagramm eines Observers

der Methode `setState(int value)` aus[3]. Die observierte Modellinstanz quittiert die Zustandsänderung mit dem Aufruf ihrer eigenen `notify()`-Methode. Diese iteriert über alle registrierten Observer (`view1` und `view2`) und ruft deren `update()`-Methode auf. Die Sichten aktualisieren ihre Informationen über das Modell durch das Aufrufen dessen Methode `getState()`. Mit dem aktuellen Zustand versorgt, zeichnen sie sich durch Aufruf ihrer Methode `repaint()` neu.

Hierbei fällt auf, dass die Sichten nicht über die konkrete Zustandsänderung des Modells informiert werden, sondern nur generell, dass sich dessen Zustand geändert hat. Die Sichten müssen sich nach dem neuen Zustand des Modells selbst erkundigen. Dies wird als *pull*-Verfahren bezeichnet (nicht zu verwechseln mit *polling*, dem regelmäßigen Überprüfen des Zustandes eines passiven Objekts, das Zustandsänderungen nicht von sich aus mitteilt).

[3] Die Methoden `void setState( int value )` und `int getState()` haben keine weitere Bedeutung, sie dienen nur als Beispiel einer Zustandsänderung und Zustandsabfrage.

Abbildung 1.3.: Ein Texteingabefeld, das abhängig vom Zustand der Checkbox anwählbar ist oder nicht

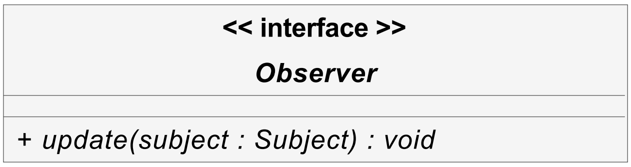

Abbildung 1.4.: Um `subject` als Parameter der `update()`-Methode erweitertee `Observer`-Schnittstelle

1.2.1. Einschränkungen des Observer-Musters

In grafischen Anwendungen müssen Elemente oftmals über Zustandsänderungen von anderen grafischen Elementen informiert werden.

Ein Eingabefeld wie in Abbildung 1.3 könnte beispielsweise abhängig vom Zustand einer davorgeschalteten Checkbox sein. Nur wenn die Checkbox ein Häkchen hat, soll das Eingabefeld anwählbar sein. Jedesmal, wenn die Checkbox betätigt wird, soll also das Eingabefeld aktiviert bzw. deaktiviert werden.

1.2.2. Observation mehrerer Objekte

Dieses Problem ist mit dem oben beschriebenen Observer-Muster lösbar. Das Eingabefeld könnte sich als Observer bei der Checkbox registrieren und in der `update()`-Methode je nach Zustand der Checkbox sich aktivieren oder deaktivieren. Dies würde jedoch bedeuten, dass eine vorgefertigte Klasse für ein Texteingabefeld nicht direkt benutzt werden kann. Es müsste eine Subklasse davon erstellt werden, die die `update()`-Methode entsprechend implementiert. Wenn man bedenkt, dass ein Dialogfenster typischerweise eine Vielzahl solcher Elemente enthält, erscheint es nicht sinnvoll, ständig Subklassen zu bilden, nur um diese `update()`-Methode zu implementieren.

Eine separate Klasse, die sich bei allen in Frage kommenden Elementen als Observer registriert und eine zentrale `update()`-Methode implementiert, wäre eine Alternative. Von Nachteil ist dabei allerdings, dass in dieser Methode alle Elemente behandelt werden müssten. Auch wenn sich nur ein einziges Element geändert hat, müssten alle `Subjects` abgefragt werden. Ein Ausweg aus diesem Dilemma wäre, das Subject wie in Abbildung 1.4 in die Signatur der `update()`-Methode des Observers einzufügen.

Damit wäre es möglich in der zentralen Implementierung nur diejenigen Elemente zu aktualisieren, die von `subject` abhängig sind, wie im folgenden Beispiel Listing 1.1 demonstriert.

Listing 1.1: Die update()-Methode eines erweiterten Observers der mehrere Objekte gleichzeitig observiert.

```
void SomeController::update( Subject * subject )
{
    if ( subject == m_checkBox )// wenn sich der Zustand der Checkbox geändert hat
    {
        m_lineEdit->setEnabled( m_checkBox->isChecked() );
    }
    else if ( subject == m_radioButton )// ein anderes Widget wurde geändert
    {
        // tue etwas anderes...
    }
    // [...]
}
```

Eine elegantere und den Prinzipien der „Separation of concerns"[4] näher kommende Lösung wäre, wiederum eine separate Klasse für die Behandlung aller zusammengehöriger Elemente zu verwenden, jedoch unterschiedliche Methoden für jedes `Subject`. Jedes observierte Element (`Subject`) würde dann eine frei festzulegende Methode aufrufen, anstatt die in der `Observer`-Schnittstelle vorgeschriebene `update()`-Methode. Beim Registrieren eines Observers übergäbe man somit nicht nur die Referenz auf den Observer sondern zusätzlich auch noch die Methode, die von `Subject::notify()` aufgerufen werden soll.

1.2.3. Feingranulare Zustandsobservation

Ein weiteres Problem des Observer Patterns ist die Granularität, mit der Zustandsänderungen publiziert werden. Es besteht keine Möglichkeit, sich nur für bestimmte Zustandsänderungen einzuschreiben. Jedwede Zustandsänderung wird gnadenlos an alle Observer gebroadcastet. Im obigen Beispiel ist beispielsweise eine Änderung der Checkbox-Beschriftung irrelevant für die Aktivierung des dahintergeschalteteten Eingabefeldes. Trotzdem würden auch bei einer Beschriftungsänderung die `update()`-Methode aufgerufen, der `isChecked()`-Status abgefragt und die `setEnabled()`-Methode des Eingabefeldes aufgerufen werden.

Sinnvoll wäre es, den „Gesamtzustand" eines solchen Objektes aufzuteilen und für jeden separierten „Teilzustand" eigene Änderungsbenachrichtigungen bereitzustellen. Um beim Checkbox Beispiel zu bleiben — dort könnte es eine Änderungsbenachrichtung für die Beschriftung „textChanged" geben sowie eine „checkingChanged" für den `isChecked()`-Status.

[4] Mit „separation of concerns" ist gemeint, dass Sprachen oder Werkzeuge nicht die Struktur diktieren sollen. Die Modularisierung von Design oder Code sollte stattdessen durch Konzept, Ziel und die eigene Betrachtungsweise bestimmt sein.[7]

Heruntergebrochen auf diese Ebene könnte mit den Benachrichtigungen zusätzlich auch noch der neue Zustand übermittelt werden, d.h. die neue Beschriftung bzw. der neue `isChecked()`-Status. Damit wäre es möglich, `Subject` und `Observer` vollständig zu entkoppeln, da der `Observer,` sobald er beim `Subject` registriert ist, keine Referenz auf das `Subject` benötigt, um dessen Zustand zu bekommen.

Das Versenden der Daten zusammen mit der Benachrichtigung wird als *push*-Verfahren bezeichnet.

1.3. Interobjektkommunikation mit Qt

In meiner Freizeit beschäftige ich mich viel mit Anwendungsentwicklung für den KDE Desktop [8], dessen Basistechnologie das Qt Toolkit ist. Nicht nur die Struktur und Funktionalität von Qt, sondern auch die bereitgestellten Konzepte sind Grund für den hohen Grad an Wiederverwendung und die erfolgreiche evolutionäre Entwicklungsweise von KDE (KDE ist seit 1996 in ständiger Entwicklung).

Die Wahl, die Kommunikationskonzepte Qts zu untersuchen ist durch deren erschöpfende Anwendung in KDE sowie meine langjährige Erfahrung damit begründet.

1.3.1. Qt – ein modernes Toolkit für C++

Im Vergleich zu Java, zu dem standardmäßig eine umfangreiche Klassenbibliothek gehört, ist C++ eher rudimentär ausgestattet. Neben der Sprachdefinition gibt es eine Standardbibliothek, die u.a. eine Reihe von Datentypen definiert und Klassen für I/O-Behandlung, Speicherverwaltung sowie generische Containerklassen und Algorithmen für diese beinhaltet [9]. Klassen für Oberflächenprogrammierung jeglicher Art sind jedoch nicht im C++ Standard enthalten.

Diesem Problem hat sich die norwegische Firma Trolltech angenommen und mit dem Qt Toolkit eine Bibliothek entwickelt, die plattformübergreifend[5] Elemente für grafische Anwendungen bereitstellt. Neben grafischen Elementen bietet es u.a. Unterstützung für Drucken, Datenbankzugriff, XML, 2D- und 3D Grafik (OpenGL), Threads, Netzwerkzugriff und Internationalisierung [10].

Des Weiteren bietet Qt ein Objektmodell basierend auf der Klasse `QObject`, die u.a. für die folgenden Features verantwortlich ist:

- Inter-Objektkommunikation (siehe 1.3.2)
- Ein Ereignisbenachrichtigungs- und Filtersystem (siehe 1.4.2)
- Eingeschränkte Unterstützung für Metaprogrammierung (Introspektion/Reflection):

[5] Die aktuelle Version Qt 3.1.2 ist für X11 auf allen möglichen Unices verfügbar sowie für Microsoft Windows und den Apple Macintosh. Qt-Anwendungen brauchen nur für die jeweilige Plattform kompiliert zu werden, plattformspezifische Details und Unterschiede sind in Qt gekapselt.

- Ein *Property*-Konzept, ähnlich dem von *Java Beans*: auf Attribute einer Klasse kann generisch lesend und schreibend zugegriffen werden, mit den Methoden

  QVariant QObject::property(**const char** ∗name)**const**; und
  bool QObject::setProperty(**const char** ∗name, **const** QVariant& value);

  `QVariant` ist eine Klasse, die eine ähnliche Funktionalität, wie die C++ `union` bietet. Sie kann genau einen Wert beinhalten, der jedoch unterschiedlichen Typs sein kann. Alle integralen Datentypen wie int, long, bool etc. sowie die wichtisten Qt-Datentypen werden unterstützt und typsicher Zugriff darauf geboten. `QVariant` umgeht damit ein Problem der `union`, die nur Klassen mit default-Konstruktoren verwalten kann (eine Reihe von Qt-Datentypen besitzen gerade keine default-Konstruktoren).

- Informationen über die vorhandenen Properties sowie Klassenname und Vererbungsbeziehungen können zur Laufzeit mit Hilfe der Klasse `QMetaObject` gewonnen werden, von der jedes `QObject` eine Instanz aggregiert.
  `QMetaObject` bietet darüber hinaus Zugriff auf einige Methoden des mit ihm assoziierten `QObjects` (genau gesagt auf die Signale und Slots, siehe 1.3.2).

• Die Organisierung aller `QObject`-Instanzen erfolgt in Bäumen. Dies bewirkt eine Eigentümerschaft zwischen Eltern- und Kindobjekten und wirkt sich insbesondere auf den Lebenszyklus letzterer aus: Wenn ein Objekt zerstört wird, zerstört es seine Kindobjekte. Gerade bei grafischen Objekten macht dies Sinn. Ein komplexes Dialogfenster besteht aus einem Dialogobjekt mit vielen Kindobjekten (Gruppierungsfelder Eingabefelder, Checkboxen, etc.), welche wiederum Kinder haben können. Sobald das Dialogfenster nicht mehr benötigt und zerstört wird, sorgt die Baumstruktur dafür, dass sämtliche beteiligten Objekte in einer korrekten Reihenfolge zerstört und ihre Ressourcen freigegeben werden. Da C++ standardmäßig keine *Garbage Collection* (das automatische Freigeben von nicht mehr benötigten Ressourcen) unterstützt, ist dieser Mechanismus eine zeitsparende und wirkungsvolle Methode, Ressourcenlecks[6] zu verhindern.

Qt für X11 und für das Linux Framebuffer Device ist Freie Software (unter der GNU General Public License verfügbar) und dient u.a. als Basis des KDE Desktops. Um die Dimensionen zu veranschaulichen, hier die ungefähre Anzahl der Codezeilen:

Qt	600.000 Zeilen Code
KDE	5 Mio Zeilen Code

[6]Ein Ressourcenleck entsteht, wenn die letzte Referenz auf eine Ressource gelöscht wird, ohne dass die Ressource freigegeben wird. In der Objektorientierung bedeutet das, dass der Destruktor eines Objekts aufgerufen werden muss, damit z.B. dessen verwendeter Speicher freigegeben werden kann oder eventuelle Dateien geschrieben werden, bevor die letzte Referenz darauf gelöscht wird.

1.3.2. Signale und Slots

Ein besonderes Merkmal von Qt ist die Unterstützung ereignisbasierter Kommunikation zwischen Objekten. Ein Objekt kann beispielsweise ein Ereignis oder einen bestimmten Zustand nach außen signalisieren. Andere Objekte, die an diesem Ereignis interessiert sind, werden darüber benachrichtigt. Die Eigenschaften dieses Kommunikationsmodells sehen folgendermaßen aus:

- Ein Signal ist eine Methode eines Senders.
- Ein Slot ist eine Methode eines Empfängers.
- Ein Signal eines Senders kann mit beliebig vielen Slots von Empfängern „verbunden" werden.
- Das Emittieren eines Signals bewirkt das Aufrufen aller damit verbundenen Slots.
- Ein signalisierendes Objekt kennt die Empfänger des Signals nicht.
- Ein Empfänger muss den Sender nicht kennen.
- Die Verbindung kann von überall aus hergestellt werden:
  - Ein Empfänger kann sich selbst mit einem Signal eines anderen Objekts verbinden.
  - Ein Sender kann sein eigenes Signal mit einem Empfänger verbinden.
  - Ein externes Objekt kann das Signal eines Senderobjekts mit einem Empfängerobjekt verbinden.

Dadurch, dass ein Sender die Empfänger nicht kennt, ja nicht einmal weiß, ob es überhaupt Empfänger gibt, sondern einfach nur stupide Signale emittiert, ist eine sehr lose Kopplung gegeben, welche generische Programmierung und einen hohen Grad von Wiederverwendung ermöglicht. Gleiches gilt für die Slots der Empfänger, die nicht wissen müssen, ob sie überhaupt mit einem Signal verbunden sind. [11]

Signale. Ein Signal hat einen Namen und kann optional Daten (Objekte, Referenzen) an den oder die Empfänger übermitteln. Es ist genau genommen nichts weiter als eine Methode mit einer beliebigen Signatur. Ein Signal wird emittiert, indem einfach sein Methodenname aufgerufen wird. „Syntaktischer Zucker" ist das `emit`-Schlüsselwort, das vor den Signalaufruf gestellt wird, um den Unterschied zu einem normalen Methodenaufruf zu verdeutlichen. Durch das `emit` ist sofort ersichtlich, dass dieser Aufruf Auswirkungen auf beliebige andere Objekte haben kann oder auch gar nichts bewirkt (wenn niemand an dem Signal interessiert ist). Ansonsten hat das `emit` keine Bedeutung, es ist lediglich ein leeres Makro[7].

[7] #define emit

Listing 1.2: Beispielemission eines Signales textChanged(const QString&)

```
void Label::setText( const QString& text )
{
    m_text = text;
    repaint();
    emit textChanged( text );
}
```

Listing 1.3: Die Deklaration eines Signales textChanged(const QString&)

```
class Label : public QWidget // QWidget erbt von QObject
{
Q_OBJECT

public:
    virtual void setText( const QString& text );
signals:
    void textChanged( const QString& text );
private:
    QString m_text;
};
```

Listing 1.2 zeigt das Versenden eines Signals `textChanged( const QString& )` aus der `setText( const QString& )`-Methode einer fiktiven `Label`-Klasse. Es bewirkt, dass alle Objekte, die ein Interesse an diesem Signal bekundet haben, benachrichtigt werden, indem ihre spezifizierte Methode aufgerufen wird und den neuen Text `text` als Parameter übergeben bekommt.

Signale werden mit Hilfe des Schlüsselworts `signals` in der Klassendeklaration, d.h. in der Regel in einer Headerdatei, deklariert. Alle Methoden, die einer `signals:`-Direktive folgen, sind nichts als geschützte Methoden der Klasse, da `signals` wiederum nur ein Makro ist, welches zu `protected` expandiert. Das `protected` macht deutlich, dass Signale vererbt werden — Subklassen können auch Signale ihrer Oberklassen emittieren.

Obwohl Signale geschützte Methoden sind, sind ihre Auswirkungen öffentlich, d.h. sie können mit Slots beliebiger anderer Klassen verbunden werden. Nur die Emission ist der Klasse selbst und ihren Subklassen vorbehalten.

Es gibt keine Beschränkung für die Anzahl der Signale einer Klasse oder die Anzahl ihrer Parameter. Der Rückgabewert eines Signals muss `void` sein; es gibt keine Rückmeldung von verbundenen Slots.

Slots. Wie Signale sind auch Slots nicht viel mehr als gewöhnliche Methoden. Ihr einziger Unterschied zu „normalen" Methoden ist, dass sie eben mit Signalen verbunden werden können. Sobald ein Signal emittiert wird, werden alle mit ihm verbundenen Slots

Listing 1.4: Die Deklaration einiger Slots

```
class Beeper : public QObject
{
Q_OBJECT

public slots:
    void setLevel( int level );
protected slots:
    void slotTextChanged( const QString& text )
    {
        std::cerr « "MEEEEP! " « m_level « ": " « text.latin1() « std::endl;
    }
private:
    int m_level;
};
```

aufgerufen. Die Aufrufe geschehen synchron, d.h. der Kontrollfluss des Senders nach der Emission des Signals wird erst fortgesetzt, wenn alle Empfängerslots abgearbeitet sind. Die Reihenfolge der Aufrufe ist undefiniert, sie gleicht nicht notwendigerweise der Reihenfolge, mit der die Verbindungen hergestellt werden.

Es gibt öffentliche, geschützte und private Slots. Sie werden wie normale Methoden einer Klasse deklariert, stehen jedoch wie die Signale in eigenen Sektionen `public slots:`, `protected slots:` und `private slots:`. Die Sichtbarkeit ist derzeit jedoch lediglich für Dokumentationszwecke zu gebrauchen. Die Verbindung von Signalen mit Slots wird erst zur Laufzeit auf Korrektheit überprüft wobei die Sichtbarkeit der Slots nicht ausgewertet wird. Es können folglich Signale auch beliebig mit privaten Slots anderer Klassen verbunden werden. Auf meine Anfrage erklärte Trolltech, dass eine Sichtbarkeitsüberprüfung eventuell in späteren Qt-Versionen realisiert werde[8]. Immerhin berücksichtigen API-Dokumentationsgenerierungstools wie kdoc[9] oder Doxygen[10] die Sichtbarkeit und verstecken beispielsweise private Slots in der Ausgabe.

Der Rückgabewert von Slots sollte ebenso wie bei Signalen `void` sein; er wird beim Aufruf über Signale nicht berücksichtigt. Falls ein Slot jedoch hauptsächlich als normale Methode verwendet wird und nur der Bequemlichkeit halber ein Slot ist, mag ein Rückgabewert Sinn machen.

Obwohl nicht vorgeschrieben, verwenden viele Qt-Programmierer die Konvention, Slots

[8] Leider hat der Trolltech-Support nicht wirklich eingesehen, dass das Ausgeben einer Fehlermeldung und ggf. das Nicht-Zulassen der Verbindung eine angemessene Reaktion auf einen derartigen `connect()`-Versuch wäre. Die API-Dokumentation sei das Mittel der Wahl, um Verbindungen zu den richtigen Slots herzustellen. Ein weiteres Problem sei Geschwindungkeit — die Überprüfung der Sichtbarkeit mit den Typen der beteiligten Objekten zur Laufzeit würde einige Performanzeinbußen mit sich bringen.

[9] http://www.ph.unimelb.edu.au/~ssk/kde/kdoc/

[10] http://www.doxygen.org

mit dem Präfix `slot` zu benennen, wie im Listing 1.4 der Slot `slotTextChanged()`. Dabei wird in der Namensgebung unterschieden zwischen dem, was eine Methode tut (z.B. `setLevel()`) und wodurch sie aufgerufen wird (`slotTextChanged()`). Letztere Namensgebung wird meist nur bei geschützten und privaten Slots verwendet, die als Ereignisbehandlungsroutinen verdeutlicht werden sollen.

Die Verbindung von Signalen mit Slots. Damit ein Slot durch eine Emission eines Signals aufgerufen wird, muss zuvor eine Verbindung mit ihm hergestellt werden. Solche Verbindungen werden dynamisch zur Laufzeit durch Aufruf der Klassenmethode `static bool QObject::connect()` hergestellt. Folgende Parameter müssen dabei übergeben werden:

1. das Senderobjekt
2. die Signatur des zu verbindenden Signals
3. das Empfängerobjekt
4. die Signatur des zu verbindenden Slots

Sender und Empfänger müssen vom Typ `QObject`, bzw. ein Subtyp davon sein. Signal und Slot werden als Zeichenketten (`const char *`) übergeben, die mit den Makros `SIGNAL` und `SLOT` erzeugt werden. Der Rückgabewert gibt Aufschluss darüber, ob die Verbindung erfolgreich hergestellt werden konnte, oder nicht.

Eine sehr lose Kopplung der Verbindungspartner wird dadurch erreicht, dass

- Sender und Empfänger nur als Basistyp `QObject` angegeben werden müssen,
- Signale und Slots als Zeichenketten angegeben werden und somit keine Kompilierabhängigkeit besteht,
- die Verbindung erst zur Laufzeit hergestellt wird.

Analog zur Verbindungsherstellung kann eine Verbindung auch wieder abgebrochen werden. Die dafür zuständige Methode nennt sich originellerweise `QObject::disconnect()`. Die Argumente von `disconnect()` haben neben der `connect()`-analogen Semantik noch einige weitere Möglichkeiten. Ein Wert 0 als Parameter wirkt als Wildcard, die je nach Parameter „jedes Signal", „jeder Empfänger" oder „jeder Slot" bedeutet. Einzig der Sender muss konkret angegeben werden. Damit ist es möglich,

- sämtliche Verbindungen zu einem Sender zu kappen.
- sämtliche Verbindungen zu einem bestimmten Signal eines Senders zu kappen.
- sämtliche Verbindungen eines bestimmten Empfängerobjektes zu einem Sender zu kappen.

Listing 1.5: Eine Signal-Slot-Verbindung

```
1 QLabel *label = new QLabel();
2 QSlider * slider  = new QSlider();
3 QObject::connect( slider , SIGNAL( valueChanged(int) ),
4                   label ,  SLOT( setNum(int) ));
```

Listing 1.6: Parameternamen dürfen innerhalb von SIGNAL und SLOT nicht genannt werden.

```
// falsch:
connect(sender, SIGNAL( mouseClicked( int button, const QPoint& pos )),
        receiver , SLOT( mouseHandler( int button, const QPoint& pos )));
// richtig:
connect(sender, SIGNAL( mouseClicked( int, const QPoint& )),
        receiver , SLOT( mouseHandler( int, const QPoint& )));
```

In Listing 1.5 werden erst zwei Widgets erzeugt, ein `QLabel`, das Text darstellen und ein `QSlider`, mit dem ein numerischer Wert eingestellt werden kann. In den Zeilen 3 und 4 wird eine Verbindung zwischen `slider` und `label` hergestellt.

Das Signal `valueChanged( int value )` wird mit dem Slot `setNum( int number )` verbunden. `QSlider` emittiert das Signal wann immer sich sein aktueller Wert ändert, beispielsweise durch Bewegen des Sliders. Der `int`-Parameter des Signals beinhaltet den neuen Wert.

Der Slot `QLabel::setNum( int number )` veranlasst das Label, die Zahl `number` anzuzeigen.

Die Verbindung von `valueChanged()` des Sliders und `setNum()` des Labels bewirkt daher, dass bei jeder Slideränderung das Label den neuen Wert des Sliders anzeigt.

Damit ein Signal mit einem Slot verbunden werden kann, müssen die Signaturen von Signal und Slot „passen". Passend bedeutet, dass sie entweder identische Parameterlisten haben oder die Parameterliste des Slots eine Teilliste der Parameterliste des Signals ist. Beliebig viele Parameter vom Ende der Parameterliste des Signals dürfen von in der Parameterliste des Slots weggelassen werden. Die Verbindungen eines Signals `mouseClicked( int button, const QPoint& pos )` mit einem Slot `mouseHandler( int button )` oder `mouseHandler()` wären damit „passend".

Signale und Slots sind somit typsicher. Wenn die Verbindung auch nicht statisch zur Kompilierzeit überprüft wird, dann gibt es immerhin zur Laufzeit eine Überprüfung, die fehlerhafte `connect()`s verhindert.

Default-Werte für Parameter in Slots werden bis dato nicht unterstützt.

Zu beachten ist, dass im `connect()`-Aufruf einzig die Signatur, d.h. ohne Parameternamen innerhalb der `SIGNAL` und `SLOTS` Makros aufgeführt werden darf, da sonst der Verbindungsaufbau fehlschlagen würde (vgl. Listing 1.6).

Weitere Details.

- Innerhalb eines Slots kann mit Hilfe der Methode `const QObject * QObject::sender()` eine Referenz auf das emittierende Objekt ermittelt werden. Dies sollte nur in Spezialfällen getan werden, da es die Entkopplung von Sender und Empfänger verletzt. Außerdem sollte bedacht werden, dass ein Slot auch wie eine „normale" Methode aufgerufen werden kann und in dem Fall kein „Sender" vorhanden ist.

- Ein Sender kann sich informieren lassen, wenn eine Verbindung zu einem seiner Signale hergestellt wird. Er reiplementiert dafür die virtuelle Methode `void QObject::connectNotify( const char *signal )`, welche immer aufgerufen wird wenn sich ein Objekt mit dem Signal `signal` verbindet. Ebenso kann das Lösen von Verbindungen erfasst werden, durch Reimplementieren der Methode `void QObject::disconnectNotify( const char *signal )`. Das Überwachen der Verbindungen könnte z.B. nützlich sein, wenn ein zu emittierendes Signal Daten als Parameter übermittelt, deren Erzeugung hohen Aufwand erfordert. Unter Verwendung der beiden Methoden könnte ein Verbindungszähler eingerichtet werden, anhand dessen das Signal nur dann emittiert (und dessen Paramter nur dann erzeugt werden), wenn Verbindungen vorhanden sind.

- Das Versenden von Signalen (genauer gesagt das Aufrufen mit ihnen verbundener Slots) eines Objektes kann temporär unterbrochen werden, indem dessen Methode `void QObject::blockSignals( bool block )` mit dem Wert `true` aufgerufen wird. Die Unterbrechung wird durch erneutes Aufrufen mit dem Wert `false` wieder aufgehoben. Dies kann von Nutzen sein, wenn beispielsweise ein Slot für die Behandlung einer Zustandsänderung eines Objektes selbst den Zustand des emittierenden Objektes ändert und somit eine Endlosrekursion auslösen würde. Der Status der Blockierung von Signalen kann mittels der Methode `bool QObject::signalsBlocked()` erfragt werden, wie auch im Listing 1.7 demonstriert.

- Ein Signal kann nicht nur mit einem Slot verbunden werden, sondern auch mit einem anderen Signal. Die Emission des ersten Signales bewirkt dann einfach die Emission des zweiten Signales. Das ist hilfreich, wenn eine Klasse ein signalemittierendes Attribut besitzt und dieses Signal nach außen „weiterleiten" möchte. Die Syntax hierfür lautet:

  ```
  connect(m_sender, SIGNAL( valueChanged(int) ),
          this,     SIGNAL( myValueChanged(int) ));
  ```

  Dabei würde das Signal `valueChanged( int )` des Objektes `m_sender` weitergeleitet werden an das Signal `myValueChanged( int )` des aktuellen Objektes (`this`).

Der Meta-Object-Compiler `moc`. Die Unterstützung für Introspektion, Properties sowie Signale und Slots bedingt, dass einige Informationen über die Klasse, z.B. deren Klassenname, Superklassen, Signale und Slots und Properties zur Kompilierzeit erfasst

Listing 1.7: Ein Beispiel für den Einsatz von blockSignals()

```
void Beispiel::setup()
{
    m_dateEdit = new QDateEdit();
    connect( m_dateEdit, SIGNAL( valueChanged( const QDate& )),
             this,       SLOT( slotDateChanged( const QDate& )));
    // [...]
}

void Beispiel::slotDateChanged( const QDate& date )
{
    if ( date.day() == 1 && date.month() == 4 )
    {
        bool wasBlocked = m_dateEdit->signalsBlocked();
        m_dateEdit->blockSignals( true );
        QDate fun( date.year() , 12, 24 ) ;
        m_dateEdit->setDate( fun ); // jo is denn jetz scho Weihnachten?
        m_dateEdit->blockSignals( wasBlocked );
    }
}
```

und in die Klasse aufgenommen werden. Dies erledigt Qts Präprozessor mit dem Namen *moc* (**M**eta **O**bject **C**ompiler). Er wertet alle neuen Schlüsselwörter (z.B. `signals`, `slots`) in der Klassendeklaration aus und generiert einige Methoden hinzu. Um den Quellcode der Dateien nicht zu ändern, wird der Code einfach in eine separate Datei generiert, die dann inkludiert bzw. extra kompiliert und hinzugelinkt werden kann. Damit überhaupt weitere Methoden zu einer Klasse hinzugeneriert werden können, müssen jedoch deren Deklarationen bereits in der Klassendeklaration vorhanden sein.

Das `Q_OBJECT` Makro fügt die notwendigen Methodendeklarationen in die Klasse ein (vgl. Listing 1.8). Deren Implementierung wird vom *moc* generiert, nachdem er die Klassendeklaration parsiert und die benötigten Informationen gelesen hat. Ist das `Q_OBJECT` Makro dort nicht präsent, bricht der *moc* die Bearbeitung ab und generiert keinen Code.

Für jedes deklarierte Signal und jeden deklarierten Slot generiert der *moc* Daten, die zur Laufzeit angefordert werden können. Sie ermöglichen das Aufrufen von Slots anhand ihrer Namen sowie Introspektion der Signale und Slots, d.h. sie bieten Signaturinformationen für alle Signale und Slots.

Die Implementierung eines Signales ist in Listing 1.9 ersichtlich. Die Methode `receivers()` liefert eine Liste aller Verbindungen mit diesem Signal, wobei das Signal durch ein bestimmtes Offset in einer Tabelle des `QMetaObjects` angegeben wird. Die Parameter des Signales werden in ein `QUObject` Array verpackt, welches zusammen mit der Verbindungsliste in der `activate_signal()`-Methode für den Aufruf aller mit dem Signal ver-

Listing 1.8: Die Expansion des Q_OBJECT Makros. QT_PROP_FUNCTIONS und QT_TR_FUNCTIONS beinhalten Funktionen für die Behandlung von Properties und kontextsensitiver Übersetzung

```
public:
    virtual QMetaObject *metaObject() const {
        return staticMetaObject();
    }
    virtual const char *className() const;
    virtual void* qt_cast( const char* );
    virtual bool qt_invoke( int, QUObject* );
    virtual bool qt_emit( int, QUObject* );
    QT_PROP_FUNCTIONS
    static QMetaObject* staticMetaObject();
    QObject* qObject() { return (QObject*)this; }
    QT_TR_FUNCTIONS
private:
    static QMetaObject *metaObj;
```

bundenen Slots verwendet wird. `QUObject` beinhaltet eine `union,` die unterschiedlichste Datentypen verwalten kann. Primitive Datentypen werden dabei direkt abgespeichert, andere werden als generischer Zeiger (`void *`) behandelt.

Binärkompatibilität. Verwendet eine Klasse eine andere, so muss sie in der Regel deren Schnittstelle zur Kompilationszeit kennen. Wird die verwendete Klasse im Nachhinein modifiziert, neu kompiliert und installiert, so müssen ihre Nutzer genau dann nicht neu kompiliert werden, wenn ihre neue binäre Schnittstelle kompatibel zur vorigen ist.

Änderungen an Klassen sind *binär inkompatibel*, wenn bestimmte Änderungen an öffentlichen oder geschützten Entitäten durchgeführt werden, nämlich wenn:

- die Signatur existierender Methoden geändert wird (selbst wenn mittels default-Parametern die Sourcekompatibilität gewahrt oder nur ein abschwächendes `const` hinzugefügt wird)
- die *virtual table* verändert wird, durch
  - Hinzufügen von neuen virtuellen Methoden
  - Umordnen von existierenden virtuellen Methoden in der Deklaration
- nicht-statische Attribute beliebiger Sichtbarkeit hinzugefügt oder entfernt werden
- die Klassenhierarchie geändert wird.

Das Hinzufügen von Signalen oder Slots zu einer Klasse ist binärkompatibel, da keines dieser Kriterien dabei verletzt wird.

Listing 1.9: Die vom moc generierte Implementierung eines Signales void fooChanged(const Foo&).

```
void FooWidget::fooChanged( const Foo& t0, int t1 )
{
    if ( signalsBlocked() )
        return;
    QConnectionList *clist = receivers( staticMetaObject()->signalOffset() + 0 );
    if ( ! clist )
        return;
    QUObject o[3];
    static_QUType_ptr.set(o+1,&t0);
    static_QUType_int.set(o+2,t1);
    activate_signal( clist , o );
}
```

1.3.3. Einschränkungen von Signalen und Slots

- Die einzige Voraussetzung für die Verwendung von Signalen oder Slots ist, dass sowohl Sender als auch Empfänger von der Klasse `QObject` erben müssen, die die notwendige Funktionalität bereitstellt.
- Signale und Slots dürfen keinen Rückgabewert haben.
- Bei der Emission eines Signals ist die Reihenfolge, in der alle damit registrierten Slots aufgerufen werden, nicht definiert.
- Geschachtelte Klassen sowie Templates werden vom *moc* derzeit noch nicht unterstützt. Signale und Slots können daher nicht ohne Weiteres in geschachtelten Klassen oder Templates verwendet werden, da der nötige Code vom *moc* nicht generiert wird. Es ist jedoch möglich, den benötigten Code von Hand zu schreiben.
- Verbindungsfehler auf Grund von Signaturinkompatibilitäten werden erst zur Laufzeit erkannt und geahndet.
- Signale sind keine Entitäten ersten Grades, sie bewirken lediglich Methodenaufrufe, bei denen Parameter übergeben werden können. Sie bieten keine Kapselung von Daten und Funktionen.

1.3.4. Vergleich zu Callbacks

Callbacks waren bisher das gebräuchlichste Mittel, um Ereignisse mitzuteilen. Ein Callback ist ein Zeiger auf eine Funktion, der an ein anderes Objekt (den Ereignissender) übermittelt wird. Dieses dereferenziert den Zeiger und ruft die Methode auf, um ein Ereignis zu signalisieren. Diese Technik bringt folgende Probleme mit sich:

1. Die Signalisierung ist nicht typsicher. Es wird nirgends sichergestellt, dass die richtigen Parameter an den Callback übergeben werden.

2. Die Kopplung von Callback und Sender ist relativ hoch, da der Sender die aufzurufende Methode des Empfängers kennen muss.

3. Der Sender wird nicht automatisch benachrichtigt, wenn ein Empfänger zerstört wird. Wird ein Empfänger zerstört und besitzt der Sender immer noch seinen Callback, dann ist das ein sog. „dangling pointer", d.h. ein Zeiger der ein nicht mehr existentes Objekt referenziert. Das Aufrufen eines solchen Callbacks führt unweigerlich zu einem Absturz des Programmes.

4. Überladene Callbacks mit dem gleichen Methodennamen aber unterschiedlichen Parametern sind nicht möglich.

Die Nachteile 1) und 2) sind bei der Verwendung von Signalen und Slots nicht vorhanden. Auch Punkt 3) ist bei Signalen und Slots inexistent. Der `QObject`-Destruktor unterbricht automatisch alle noch existierenden Verbindungen mit anderen Objekten. So kann es niemals vorkommen, dass ein Objekt eine Verbindung mit einem bereits zerstörten anderen Objekt besitzt. Ebenso ist Punkt 4), das Überladen sowohl von Signalen als auch von Slots möglich.

Einziger Nachteil von Signalen und Slots gegenüber Callbacks ist die Ausführgeschwindigkeit. Ein direkter Methodenaufruf über einen Callback ist rund 10 Mal schneller als ein Slot-Aufruf durch die Emission eines Signals. In der Praxis ist dies jedoch kaum von Bedeutung, da die Behandlung von Ereignissen typischerweise viel aufwendiger ist, als die Ereignisemission. Lediglich bei *sehr* oft auftretetenden Ereignissen sollte ein anderes Kommunikationsmodell in Betracht gezogen werden. [12, 13]

Weiterer immenser Vorteil von Signalen und Slots gegenüber Callbacks ist die Unterstützung von Introspektion — ein wichtiges Feature, das generische Programmierung erlaubt und leider mit regulären C++ Methoden nicht möglich ist.

1.4. Ereignisse (Events)

Signale und Slots und auch das Observer-Muster haben sich als geeignete Abstraktionen für Kommunikation zwischen lose gekoppelten Objekten erwiesen. Dennoch bieten auch sie für viele Einsatzgebiete nicht die nötige Flexibilität, so dass andere Techniken benötigt werden.

Die bisher vorgestellten Lösungen beschränken sich auf synchrone Kommunikation, was bedeutet, dass der Kontrollfluss des Senders so lange unterbrochen ist, bis alle Empfänger das Ereignis verarbeitet haben. Sie bieten weder die Möglichkeit, Ereignisse zu filtern, noch sie zu priorisieren. Auch ist es nicht möglich, Ereignisse an zentraler Stelle abzufangen (*interception*) und separat zu bearbeiten. Weiterhin sind sie nur für Interobjektkommunikation innerhalb eines Prozesses verwendbar.

Diese Probleme werden von Ereignisbehandlungssystemen adressiert, deren Façetten in diesem Kapitel diskutiert werden. Als konkretes Beispiel wird das Ereignisbehandlungssystem von Qt vorgestellt.

1.4.1. Synchrone und asynchrone Ereignisbehandlung

Grundsätzlich gibt es zwei unterschiedliche Methoden, Ereignisse zu behandeln. Die Kontrollflüsse von Sender und Empfänger können entweder gekoppelt oder entkoppelt sein. Im ersten Fall spricht man von synchroner, im zweiten Fall von asynchroner Ereignisbehandlung.

Synchron bedeutet, dass der Kontrollfluss des Senders bei einer Ereignisemission erst fortfährt, wenn das Ereignis an alle Empfänger übertragen wurde und diese dessen Bearbeitung abgeschlossen haben.

Bei asynchroner Ereignisbehandlung ist der Sender entkoppelt von den Empfängern und fährt im Kontrollfluss nach der Ereignisemission fort, ohne auf die Behandlung des Ereignisses durch die Empfänger zu warten. Dabei ist nicht definiert, wann die Empfänger mit der Ereignisbehandlung beginnen. Dies kann entweder nebenläufig zum Kontrollfluss des Senders geschehen oder, wenn Nebenläufigkeit nicht möglich ist, sobald der Sender seine aktuelle Aufgabe beendet hat und der Kontrollfluss nacheinander an die Empfänger übergeben wird.

Nebenläufige Ereignisbehandlung in *multithreaded* Software kann realisiert werden, indem Sender und Empfänger in unterschiedlichen Kontexten (*Threads*) laufen, die nur für die Ereignisbenachrichtigung kommunizieren müssen.

EventLoops. In ereignisbasierter, nicht-nebenläufiger Software ist eine übergeordnete Kontrollinstanz vonnöten, die die Ereignisemission und -behandlung koordiniert. Hierfür kommt typischerweise eine sog. *EventLoop* zum Einsatz. Diese hat die ehrenvolle Aufgabe, den Kontrollfluss zu verwalten und bei Bedarf an bestimmte Objekte zu delegieren. Bei interaktiven Anwendungen beispielsweise „wartet” die Anwendung oft auf Eingaben des Benutzers. In dem Moment hat die EventLoop die Kontrolle und wartet z.B. auf Tastatur- oder Mausereignisse des zugrundeliegenden Systems. Sobald diese eintreffen, werden Ereignisbehandlungsroutinen der die Ereignisse betreffenden Objekte aufgerufen.

Nicht nur Systemereignisse, sondern auch Anwendungsereignisse können von der EventLoop erfasst und an den bzw. die Empfänger weitergeleitet werden. Ein Sender könnte somit ein Ereignis über die EventLoop versenden, welche das Ereignis aber nicht sofort weiterleitet, sondern nur speichert und den Kontrollfluss an den Sender zurückgibt. Erst wenn der Sender den Kontrollfluss wieder an die EventLoop abgibt, leitet diese das Ereignis weiter.

Ähnlich einem Mediator [14] ist mit dieser Methode eine Entkopplung von Sender und Empfänger gegeben, ohne nebenläufige Techniken wie *Multithreading* zu verwenden.

Das Speichern von Events in einer EventLoop und deren spätere Versendung eröffnet noch weitere Möglichkeiten. Häufig vorkommende Ereignisse, wie z.B. solche, die das Neuzeichnen eines grafischen Elements verursachen („paintEvent”), können „komprimiert” werden. Dadurch würde das grafische Element nur einmal neu gezeichnet werden,

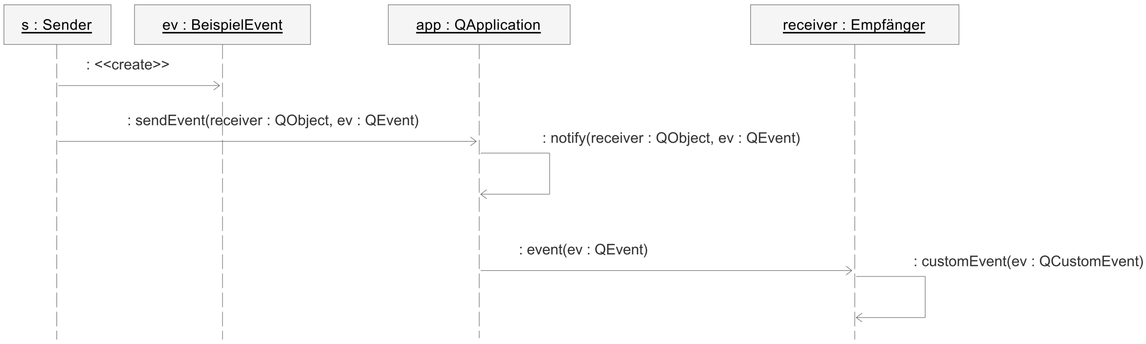

Abbildung 1.5.: Die Sequenz einer synchronen Ereignisbehandlung

da alle überflüssigen paintEvents vor ihrer Auslieferung eliminiert würden. Somit können die Prozessorbelastung und benötigte Bandbreite erheblich reduziert werden.

Eine weitere Möglichkeit, die eine EventLoop eröffnet, ist Ereignispriorisierung, die die Auslieferungsreihenfolge beeinflusst. Bestimmte Events könnten eine höhere Priorität als andere haben. Ereignisse höherer Priorität würden hierdurch eher ausgeliefert werden, als Ereignisse niedriger Priorität, auch wenn letztere zeitlich vorher emittiert wurden.

1.4.2. Qts Ereignissystem

Qt offeriert noch ein weiteres Mittel zur Ereignisverarbeitung, das für gewisse Anwendungen besser geeignet ist, als Signale und Slots. Das Qt Ereignissystem behebt die in 1.4 beschriebenen Mängel. Die damit gewonnene Flexibilität wird mit der Aufgabe der Signaturunabhängigkeit der Signale und Slots und einer engeren Kopplung von Sender und Empfänger erkauft. Weiterhin gibt es im Unterschied zu Signalen und Slots nicht beliebig viele, sondern nur einen einzigen Empfänger.

Die Klasse `QEvent` modelliert ein generisches Ereignis und dient gleichzeitig als Oberklasse für viele vordefinierte konkrete Ereignisse. Derartige Ereignisse sind somit Entitäten ersten Grades, sie kapseln Daten und Funktionen in einem Objekt.

Versenden von Events Instanzen von `QEvent` können sowohl synchron als auch asynchron versendet werden. Sie sind an einen einzigen Empfänger gerichtet, der bei der Versendung zusammen mit dem `QEvent`-Objekt angegeben werden muss. Dieses System folgt daher nicht dem publish-subscribe-Muster. Das Versenden geschieht mittels der Klassenmethoden

static bool QApplication::sendEvent(Object$*$ receiver, QEvent$*$ event) und
static bool QApplication::postEvent(QObject$*$ receiver, QEvent$*$ event).

Erstere leitet das Ereignis `event` synchron an den Empfänger `receiver` weiter. `postEvent()` bewirkt einen asynchronen Aufruf beim `receiver` mit Hilfe der Qt EventLoop. Abbildung 1.5 zeigt die Sequenz einer synchronen Ereignisbenachrichtigung.

Empfang von Events. Eintreffen tut das `event`-Objekt beim Empfänger in der Methode `QObject::event( QEvent *event )`. Die Standardimplementierung in `QObject` wirkt als *Dispatcher*, indem sie den Typ des Ereignisses auswertet und anhand dessen ggf. spezialisierte Ereignisbehandlungsroutinen aufruft (`timerEvent(), childEvent(), customEvent()`). `QWidget`, die Oberklasse aller grafischen Elemente in Qt verfeinert die Behandlung weiter und führt eine Reihe zusätzlicher Ereignisbehandlungsroutinen ein (`keyPressEvent()`, `focusInEvent()`, `mousePressEvent()` und viele mehr).

Ereignisse können somit sowohl durch die Reimplementierung der speziellen Methoden als auch durch Reimplementierung der `event()`-Methode behandelt werden [15].

Abfangen und Filtern von Events (Interception). Erst die Möglichkeit, Ereignisse, die für ein bestimmtes Objekt gedacht sind, von einem anderen Objekt abzufangen und dort zu bearbeiten öffnet das Tor für dynamisch adaptierbare Software. Qt erlaubt das Registrieren von sog. *Eventfiltern* an einem `QObject`, die fortan jegliche Ereignisse, die an das Objekt gerichtet sind, empfangen und verarbeiten können. Beliebig viele Eventfilter können sich an einem Objekt registrieren. Die Aufrufe erfolgen in der umgekehrten Reihenfolge der Registrierung, d.h. der zuletzt registrierte Filter wird zuerst mit der Ereignisbehandlung betraut. Entscheidet dieser, das Ereignis nicht zu behandeln, wird der nächste aufgerufen. Dies geschieht solange, bis ein Filter die Ereignisbehandlung stoppt oder bis das Ereignis beim ursprünglichen Empfänger ankommt und dort behandelt wird.

`QObject` enthält die Methoden

void installEventFilter (**const** QObject∗ filterObj) und
bool eventFilter(QObject∗ receiver, QEvent∗ event),

welche für das Abfangen und Filtern von Events verantwortlich sind. Per `obj->installEventFilter( filterObj )` wird `filterObj` als Filter für `obj` registriert und empfängt fortan alle Ereignisse, die an das Objekt `obj` gerichtet sind. Diese Ereignisse werden an die Methode `eventFilter()` des Filters geliefert.

Da ein Filterobjekt die Ereignisse beliebig vieler Objekte abfangen kann und diese alle in seiner `eventFilter()` Methode ankommen, hat diese nicht nur das `QEvent`-Objekt als Parameter, sondern auch das `QObject`, an das das Event eigentlich gerichtet war.

Der Rückgabewert von eventFilter() gibt Aufschluss darauf, ob das Ereignis als behandelt gelten oder ob die Behandlung durch eventuelle weitere Eventfilter oder das Empfängerobjekt fortgeführt werden soll. Ein Rückgabewert von `true` bedeutet, dass die Bearbeitung gestoppt werden soll, `false` weist Qt an, das nächste Filterobjekt mit der Bearbeitung des Ereignisses zu beauftragen.

Das Entfernen eines installierten Eventfilters ist mit der Methode `QObject::removeEventFilter( const QObject *filterObj )` möglich.

Neben der Möglichkeit Ereignisse für spezifische Objekte abzufangen, können auch *alle Ereignisse für alle Objekte* zentral abgefangen werden. Da sämtliche Ereignisse für alle Objekte die Methode `QApplication::notify( QObject *receiver, QEvent *event )` passieren, kann dies durch Reimplementierung dieser in einer Subklasse von `QApplication` geschehen oder durch Installieren eines Eventfilters auf dem `QApplication` Objekt (jede Qt Anwendung hat genau ein `QApplication` Objekt) [10].

Ein gefiltertetes Objekt hat keine Informationen über etwaige installierte Filter — es „merkt” z.B. nur, dass ihm gewisse Ereignisse vorenthalten werden.

Ebenso wie bei Signalen und Slots werden installierte Eventfilter aufgehoben, sobald eins der beteiligten Objekte zerstört wird.

1.4.3. Einschränkungen des Qt Ereignissystems

- Der Sender eines Ereignisses unterliegt keinerlei Einschränkungen, der Empfänger muss jedoch zwingendermaßen vom Typ `QObject`, bzw. ein Subtyp dessen sein.
- Die Kommunikation findet in einer gerichteten 1:1 Beziehung statt, d.h.
  - es gibt nur einen einzigen Empfänger
  - dieser muss beim Versenden angegeben werden, was eine relativ starke Kopplung von Sender und Empfänger bedeutet (obwohl eine generische `QObject` Referenz des Empfängers ausreicht).

1.5. Events über Prozessgrenzen hinweg

Die bisher vorgestellten Technologien zur Ereignisbehandlung sind lediglich als Kommunikationsmittel innerhalb eines Prozesses anwendbar (*in-process*). Um verteilte Anwendungen oder aber auch einfach Kommunikation zwischen unterschiedlichen Anwendungen (*interprocess communication*) zu ermöglichen muss ein anderes Transportmittel für die Ereignisse bzw. Methodenaufrufe zum Einsatz kommen.

Dafür wird eine Technologie benötigt, die die Fähigkeit hat, Nachrichten zu serialisieren, zu anderen Prozessen (ggf. auch mittels eines Netzwerkes über Rechner- und Plattformgrenzen hinweg) zu versenden, um sie dort wieder zu deserialisieren und in einen Methodenaufruf im Zielprozess umzuwandeln.

1.5.1. Distributed Object Computing Middleware

Gebräuchliche Technologien, die dies umsetzen sind *CORBA* (**C**ommon **O**bject **R**equest **B**roker **A**rchitecture [16]), *Java RMI* (**R**emote **M**ethod **I**nvocation [17]) oder Microsofts *DCOM* (**D**istributed **COM** [18]). Sie können unter dem klangvollen Namen *Middleware für Distributed Object Computing* zusammengefasst werden. Gemein ist ihnen die Fähigkeit, Methoden auf Objekten aufzurufen, unabhängig davon, ob das Objekt im selben Adressraum des Aufrufers liegt, oder nicht. Damit dies transparent funktionieren kann, werden sog. *Proxy*-Objekte eingesetzt. Der Aufrufer (Client) ruft die gewünschte Methode eines Objektes über dessen Proxy auf, welcher den Methodenaufruf als Nachricht an das eigentliche Zielobjekt (Servant) weiterleitet. Dort angekommen wird die Nachricht interpretiert und die gewünschte Methode aufgerufen. Bei Beendigung des Aufrufes wird der Rückgabewert wieder zurück zum Proxy transportiert und von diesem an den Client übergeben.

1.5.2. DCOP

Innerhalb des KDE Projektes wurde *DCOP* (**D**esktop **Co**mmunication **P**rotocol) als Mittel zur Interprozesskommunikation und verteilten Ereignisbenachrichtigung entwickelt. [19] DCOP unterstützt eine Vielzahl der bislang beschriebenen Features. Folgende Arten der Kommunikation werden damit ermöglicht:

- Synchrone und asynchrone Kommunikation
- Kardinalität von Empfänger zu Sender ist 1 zu n, d. h. Nachrichten können an beliebig viele Empfänger gleichzeitig versendet werden (*multicasting* und *broadcasting*).
- Gekoppelte Kommunikation (direkter Methodenaufruf vom Sender beim Empfänger)
- Ungekoppelte Kommunikation (publish/subscribe-Muster, ähnlich Signalen und Slots)
- Dynamische Adaption der Schnittstellen zur Laufzeit: ein Servant-Objekt kann das Interface, das es anbietet dynamisch ändern.

Funktionsweise von DCOP. Im Zentrum steht der *dcopserver*, ein Prozess der allen DCOP-basierten Anwendungen eines Benutzers ständig zur Verfügung steht. Er dient als Vermittler für Methodenaufrufe und Ereignisbenachrichtigung zwischen unterschiedlichen Prozessen. Sowohl der Client als auch der Servant kommunizieren dabei nur mit dem dcopserver, es gibt keine „Direktverbindung" zwischen Client und Servant.

Generell wirkt der dcopserver als *Mediator* (siehe Abbildung 1.6, Seite 29) zwischen Objekten unterschiedlicher Prozesse. In der Rolle als Ereignisvermittler im publish-subscribe Verfahren wirkt er jedoch als *Event Notifier* [20], dessen Struktur in Abbildung 1.7 auf Seite 31 ersichtlich ist. Die beiden unterschiedlichen Funktionsweisen werden im Folgenden dargestellt.

Vermittlerfunktion. Um sich als DCOP-Servant zur Verfügung zu stellen, muss eine Anwendung ein oder mehrere Instanzen der Klasse `DCOPObject`, genauer gesagt, Subklassen davon erzeugen. Jedes `DCOPObject` hat einen Namen, mit dem es referenziert werden kann. Die Methoden, die per DCOP aufrufbar sein sollen, werden ähnlich deklariert wie *Slots*. In der Subklasse von `DCOPObject` muss das Makro `K_DCOP` verwendet werden, ähnlich dem `Q_OBJECT` Makro, um einige interne Methoden zu deklarieren, deren Implementierung vom *dcopidl2cpp*-Compiler generiert wird. So wie `public slots:` alle folgenden Methodendeklarationen innerhalb dieser Sektion als öffentliche Slots markiert, so markiert die `k_dcop:` Sektion alle folgenden Methodendeklarationen als DCOP-aufrufbar.

Die so präparierte Klassendeklaration wird vom *dcopidl2cpp*-Compiler parsiert, der eine Funktionstabelle aller DCOP-aufrufbaren Methoden sowie einen Aufrufdispatcher generiert. Die generierte Dispatchmethode

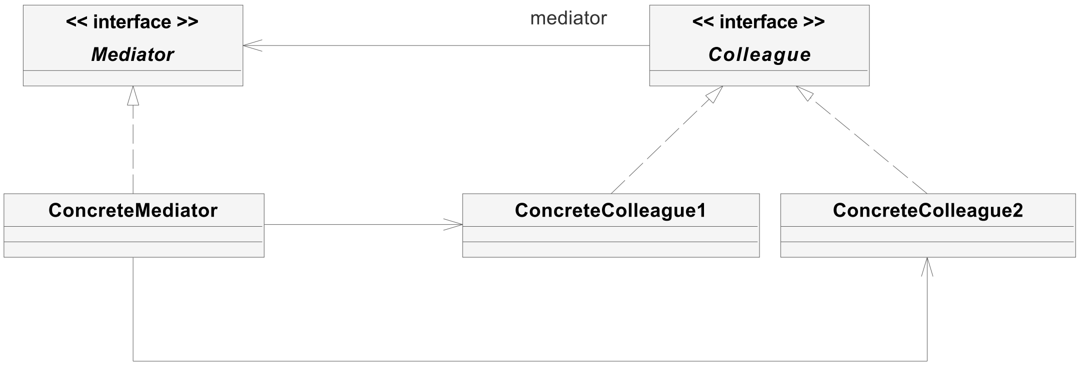

Abbildung 1.6.: Klassendiagramm des Mediator-Musters

bool process(**const** QCString& fun, **const** QByteArray& data, QCString& replyType, QByteArray& replyData)

analysiert die als Nachricht serialisierten Methodenaufrufe, deserialisiert und delegiert sie an die mit dem Parameter `fun` spezifizierte Methode. Der Rückgabewert dieser wird wiederum serialisiert und an den Aufrufer übermittelt. Übergabeparameter werden immer als Wert behandelt, nicht als Referenzen (*call by value* vs. *call by reference*). Ein Servant kann daher keine Objekte eines Clients referenzieren[11].

Kann die spezifizierte Methode im `process()`-Aufruf nicht in der Funktionstabelle gefunden werden, so wird die virtuelle Methode `processDynamic()` aufgerufen, in der der Servant dynamische Methodenbehandlung durchführen kann.

Mit dem Makro `ASYNC` können rückgabelose Methoden als asynchron deklariert werden, was bewirkt, dass ein Aufrufer nicht auf die Beendigung der Methode wartet, sondern direkt mit seinem Kontrollfluss fortfährt.

Um DCOP Aufrufe zu tätigen, wird in der Regel ein Proxy-Objekt (bzw. *Stub*) erzeugt, auf dem man die gewünschte Methode aufrufen kann, ohne sich um das Serialisieren von Parametern kümmern zu müssen. Solch ein Proxy bietet genau die Schnittstelle an, die ein Servant spezifiziert und implementiert hat, so dass alle Aufrufe typsicher sind. *dcopidl2cpp* generiert neben des Dispatchcodes für den Servant auch ebendiese Proxy-Implemententation für Clients.

Der Proxy ist dabei nur ein *Frontend* — das eigentliche Versenden geschieht mit Hilfe der Klasse `DCOPClient`. Diese beinhaltet die u.a. die beiden Methoden

bool send(**const** QCString &remApp, **const** QCString &remObj,
const QCString &remFun,
const QByteArray &data);

und

bool call(**const** QCString &remApp, **const** QCString &remObj,
const QCString &remFun,

[11] Natürlich könnte ein Client eine DCOP-Objektreferenz (`DCOPRef`) seiner selbst an den Server übermitteln, wodurch dieser wiederum auf dem Client Methoden aufrufen könnte.

const QByteArray &data,
QCString& replyType, QByteArray &replyData,
bool useEventLoop, **int** timeout);

`send()` bewirkt einen asynchronen, `call()` einen synchronen Aufruf, was auch anhand der Signatur ersichtlich ist — nur `call()` beinhaltet Parameter für den Rückgabewert (`replyType` und `replyData`). Mit den Parametern `remApp` und `remObj` wird der Emppänger spezifiziert, wobei `remApp` der Name ist, mit dem der Servant beim dcopserver registriert ist und `remObj` dessen gewünschtes Zielobjekt festlegt. Beide können mit Wildcards belegt werden, was Broad- und Multicasting ermöglicht (siehe Listing 1.10). Die `QByteArray` Objekte beinhalten die serialisierten Parameter, deren Funktionsweise in 1.5.3 auf Seite 33 zu lesen ist.

Falls kein Proxy für einen Servant vorhanden ist oder falls eine Nachricht an mehrere Empfänger gleichzeitig gesendet werden soll (*Multicast*), dann kann entweder die lowlevel API `DCOPClient::call()` bzw. `send()` verwendet werden oder aber die komfortablere API von `DCOPRef` [21], auf die hier aus Platzgründen nur verwiesen wird.

Listing 1.10: Ein DCOP Multicast an alle Konqueror Prozesse.

```
QByteArray unused;
DCOPClient::mainClient()->send("konqueror-*", "MainApplication-Interface",
                               "quit()", unused);
```

Benachrichtigungsfunktion. Neben direkten Methodenaufrufen via DCOP gibt es DCOP-Signale als Äquivalent zu Signalen und Slots. Ein Sender deklariert DCOP-Signale in einer `k_dcop_signals:` Sektion seiner von `DCOPObject` erbenden Klasse. So wie der *moc* eine Implementierung der Qt-Signale erzeugt, generiert *dcopidl2cpp* die Implementierung der deklarierten DCOP-Signale. Emittiert werden sie genauso wie Qt-Signale, durch einfachen Aufruf, ggf. mit vorangestelltem `emit` Schlüsselwort.

Ein Interessent verbindet sich mit einem DCOP-Signal mit Hilfe der Methode

bool DCOPClient::connectDCOPSignal(**const** QCString& sender, **const** QCString
& senderObj, **const** QCString& signal, **const** QCString& receiverObj,
const QCString& slot, **bool** Volatile).

Die Parameter haben dabei folgende Bedeutung:

- `sender` ist der Name des Servants, d.h. der Name mit dem der Servant beim dcopserver registriert ist. Wird eine leere Zeichenkette angegeben, so bleibt der Sender unberücksichtigt, d.h. es ist egal, wer das Signal emittiert.

- `senderObj` bezeichnet das Objekt, welches das Signal emittiert.

- `signal` ist die Signatur des Signals, mit dem die Verbindung hergestellt werden soll.

- `receiverObj` legt das Empfänger-Objekt fest, das den aufzurufenden Slot enthält.

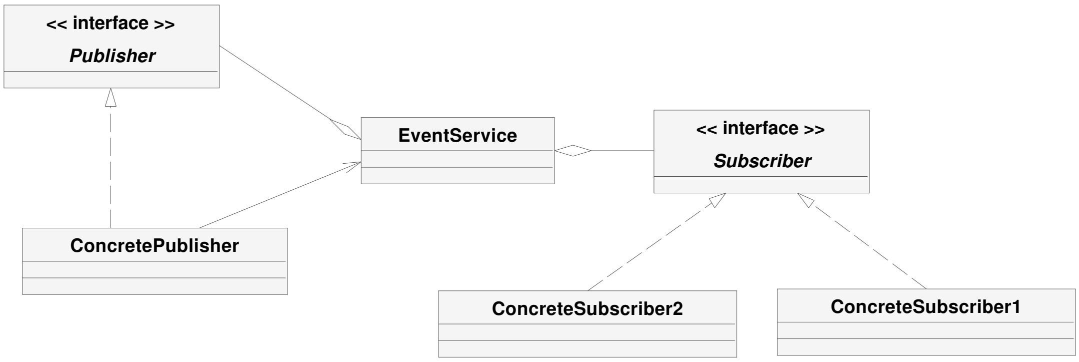

Abbildung 1.7.: Klassendiagramm des Event-Notifier-Musters

- `slot` beinhaltet die Signatur des aufzurufenden Slots.

- `Volatile`: Auf `false` gesetzt, ermöglicht es, Verbindungen mit Signalen herzustellen unabhängig davon, ob der Sender beim dcopserver registriert ist.

Der Rückgabewert von `connectDCOPSignal()` gibt an, ob die Verbindung erfolgreich aufgebaut werden konnte. Verbindungen können fehlschlagen, wenn entweder die Signatur des Signals nicht zu der des Slots passt oder `volatile` auf `true` gesetzt und `sender` nicht beim dcopserver registriert ist.

Nach erfolgreichem Verbindungsaufbau wird der Slot `slot` des Objektes `receiverObj` aufgerufen, wann immer das spezifizierte signal vom `senderObj` des Senders `sender` emittiert wird.

Mit `disconnectDCOPSignal()` können Verbindungen wieder unterbrochen werden. Die Emission eines DCOP Signales ist in Listing 1.12 ersichtlich.

Listing 1.11: Eine Beispieldeklaration eines DCOP-Servants

```
class Desktop : public DCOPObject
{
    K_DCOP
k_dcop:
    QImage getBackgroundImage( int desktop );
    ASYNC startScreensaver();
k_dcop_signals:
    void screensaverStarted();
};
```

Neben des Entgegennehmens der Clientaufrufe und Weiterleitung an die Server fungiert der dcopserver noch als *Repository*. Er verwaltet Referenzen auf alle laufenden Anwendungen und vergibt diese an anfragende Clients. Mit einer Referenz auf eine Anwendung ist es einem Client möglich, alle DCOP Objekte dieser Anwendung sowie deren Methoden zu erfragen. Mit dem Tool 'dcop' oder der grafischen Variante 'kdcop' können alle laufenden Anwendungen auf DCOP Funktionalität inspiziert werden.

Listing 1.12: Versenden eines KonqHistoryEntry Objektes als DCOP Signal

```
class BrowserHistory : public DCOPObject
{
    K_DCOP

k_dcop_signals:
    void newHistoryEntry( KonqHistoryEntry entry );
public:
    void urlOpened( const KURL& url )
    {
        KonqHistoryEntry entry;
        entry.url = url;
        // ...
        emit newHistoryEntry( entry );
    }
};

// generiert von dcopidl2cpp:
void BrowserHistory::newHistoryEntry( KonqHistoryEntry arg0 )
{
    QByteArray data;
    QDataStream arg( data, IO_WriteOnly );
    arg << arg0;
    emitDCOPSignal( "newHistoryEntry(KonqHistoryEntry)", data );
}
```

1.5.3. Parameterserialisierung

Damit Methoden über Prozessgrenzen hinweg aufgerufen werden können, müssen sie als Nachricht serialisiert werden. D.h. es muss eine Nachricht erzeugt werden, die den Empfängerprozess sowie das Empfängerobjekt beinhaltet. Ebenso muss der Name der aufzurufenden Methode in die Nachricht aufgenommen werden. Dies scheint relativ einfach zu sein, da sowohl der vollständige Empfänger als auch die Methode als String angegeben und von der Middleware ausgewertet werden können.

Schwieriger wird es bei strukturierten Datentypen als Parametern von Methoden. Da C++ keine Introspektion unterstützt, können nicht generisch alle Attribute eines Objektes ausgelesen, serialisiert und in ein neues Objekt im Zielprozess deserialisiert werden.

Es muss daher für jeden Typ, der per DCOP transportiert werden soll, eine Se- und eine Deserialisierungsfunktion implementiert werden. Qt bietet einen solchen Mechanismus und stellt für viele Datentypen Serialisierungsfunktionen bereit.

Als Träger der Nachricht wird die Klasse `QByteArray` verwendet, die einen Container für Bytes (`char`) darstellt. Die Serialisierung in ein `QByteArray` erfolgt mit Hilfe der Klasse `QDataStream`. Sie bietet *Stream-Operatoren* für alle integralen Datentypen (`int`, `float,` `double`, `char`), mit denen diese in ein `QByteArray` geschrieben und daraus gelesen werden können.

Das Verwenden von Stream-Operatoren anstatt von überladenen Methoden, beispielsweise den fiktiven Methoden

QDataStream::write(**char** value)und
QDataStream::write(**float** value),

ermöglicht die Serialisierung weiterer Datentypen, ohne die Klasse `QDataStream` selbst zu ändern. Neben den „eingebauten" Operatoren wie

QDataStream& QDataStream::**operator**« (**double** f)

kann ein neuer Datentyp durch Implementierung eines globalen Operators

QDataStream& **operator**« (QDataStream& s, **const** SomeOtherType& e)

mittels `QDataStream` serialisiert werden. Als Beispiel sei in Listing 1.13 gezeigt, wie ein Verlaufseintrag des Webbrowsers Konqueror (de-) serialisiert wird. Zu beachten ist, dass die Kombination `QByteArray/QDataStream` wie ein FIFO[12] funktioniert und daher die Attribute in der gleichen Reihenfolge gelesen wie geschrieben werden müssen.

Ein `KonqHistoryEntry` Objekt lässt sich damit recht einfach z.B. als *DCOP Signal* versenden, wie in Listing 1.12 zu sehen ist.

[12]First In, First Out beschreibt die Reihenfolge in der etwas geschrieben und gelesen wird.

Listing 1.13: Serialisierungs- und Deserialisierungsoperatoren eines Konqueror-Verlaufsobjektes.

```
class KonqHistoryEntry
{
public:
    KonqHistoryEntry() : numberOfTimesVisited(1) {}
    KURL url;
    QString typedURL;
    QString title;
    Q_UINT32 numberOfTimesVisited;
    QDateTime firstVisited;
    QDateTime lastVisited;
};
QDataStream& operator<< (QDataStream& s, const KonqHistoryEntry& e)
{
    s << e.url;
    s << e.typedURL;
    s << e.title;
    s << e.numberOfTimesVisited;
    s << e.firstVisited;
    s << e.lastVisited;
    return s;
}
QDataStream& operator>> (QDataStream& s, KonqHistoryEntry& e)
{
    s >> e.url;
    s >> e.typedURL;
    s >> e.title;
    s >> e.numberOfTimesVisited;
    s >> e.firstVisited;
    s >> e.lastVisited;
    return s;
}
```

2. Aspektorientierte Software Entwicklung

Wie schon in der Einleitung erwähnt, ist eine Beherrschbarkeit großer, komplexer Softwaresysteme nur gegeben, wenn sie in kleine, verständliche Einheiten aufgeteilt werden. Erst diese Abgrenzung in möglichst unabhängige Teilmodule mit wohldefinierten Schnittstellen erlaubt die Verteilung der Softwareentwicklung auf Teams. Die Unabhängigkeit der Module senkt den Kommunikationsbedarf zwischen den Modulentwicklern und dadurch auch den Entwicklungsaufwand. Auch der Aufwand für die Qualitätskontrolle und Wartung ist geringer, wenn Module einzeln getestet und auf Fehler geprüft werden können, anstatt im gesamten System auf Fehlersuche gehen zu müssen.

Eine *günstige* Modularisierung zu ermöglichen ist daher ein wichtiges Ziel der Softwaretechnik. „Separation of concerns" heißt die Idee der Identifikation und Kapselung der „Dinge", die zusammengehören in Modulen, oder andersherum, die Separierung der Dinge, die nicht zusammengehören. Sowohl das funktionale, das imperative als auch das objektorientierte Paradigma bieten Mechanismen, um Systeme auf Einheiten des Verhaltens oder der Funktion herunterzubrechen. Diese Art der Modularisierung wird *funktionale Dekomposition* genannt [22].

Modularisierung kann jedoch auch zu weit getrieben werden: je unabhängiger und generischer die Einheiten sind, desto mehr Aufwand muss für deren Komposition/Integration betrieben werden. Die Verwendung von *Generizität* (*templates* in C++) beispielsweise ermöglicht es, Datenstrukturen und Algorithmen unabhängig von konkreten Datentypen zu entwickeln. Obwohl damit unabhängige, wiederverwendbare Module erstellt werden können, ist damit nicht in jedem Fall eine Reduktion der Komplexität verbunden. Der GNU C++ Compiler überrascht z.B. mit kaum nachvollziebaren Fehlermeldungen bei fehlerhafter Anwendung der Standard Template Library (STL), wie in Listing A.1 auf Seite 85 zu sehen ist.

Insbesondere die Objektorientierung bietet durch Kapselung von Daten und Funktionalität in Klassen sowie durch Vererbung und Polymorphie gute Voraussetzungen für eine Strukturierung in funktionale Einheiten. Es hat sich jedoch herausgestellt, dass sich einige Eigenschaften von Softwaresystemen nicht isolieren und in Module fassen lassen. Eine Analyse dieser oft nicht-funktionalen Eigenschaften hat ergegeben, dass sie quer zur Basisfunktionalität des Systems liegen und dies der Grund für deren schlechte Erfassung und Kapselung im Design ist. Diese Erkenntnis hat den Begriff *crosscutting Concern* für derartige Systemeigenschaften geprägt.

Die fehlende Dekompositionsmöglichkeit für solche crosscutting Concerns führt da-

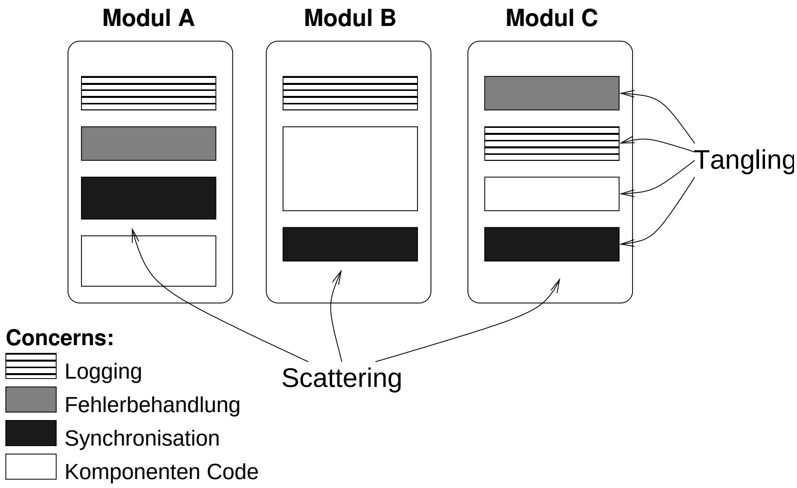

Abbildung 2.1.: Crosscutting Concerns, die zu einer *scattered* und *tangled* Implementierung führten

zu, dass deren Implementierungen über viele Module hinweg verstreut sind (*scattering*) und dadurch die einzelnen Module nicht nur einen Concern realisieren, sondern eine Vermischung mehrerer Concerns darstellen (*tangling*), was in Abbildung 2.1 dargestellt ist.

2.1. Neuer Modularisierungsmechanismus Aspektorientierung

Um auch crosscutting Concerns als modulare Einheiten erfassen und damit modellieren zu können, wurde eine Reihe von Ansätzen entwickelt, welche sich unter dem Begriff „Aspektorientierung" zusammenfassen lassen. Ein *Aspekt* ist keine Einheit der funktionalen Dekomposition, sondern eine Eigenschaft des Systems, die die Semantik oder Performanz der Komponenten beeinflusst. Mit Komponenten sind hierbei genau die Eigenschaften eines Systems gemeint, die sich in funktionalen, prozeduralen oder objektorientierten Sprachen sauber kapseln lassen, d.h. Funktionen, Methoden, Objekte[22].

Ziel der Aspektorientierung ist die Unterstützung der Separierung von Komponenten und Aspekten (mehrdimensionale Dekomposition), sowie die Bereitstellung von Mechanismen zu deren Abstraktion und Komposition. Im Folgenden werden einige Schlüsselkonzepte von aspektorientierten Sprachen vorgestellt.

2.1.1. Erfassen von Aspekten

Um Code, der sich sonst über mehrere Module erstrecken würde, in separaten Entitäten zu erfassen, sind zwei Dinge nötig:

1. Es muss ein Konzept geben, mit dem ein Aspekt aufgeschrieben werden kann.

So wie Komponentencode z.B. in Klassen strukturiert werden kann, muss auch Aspektcode als Entität erfasst werden können.

2. Die Wirkung eines Aspektes muss definiert werden können. Es muss eine Möglichkeit geben, die Stellen im Komponentencode zu spezifizieren, an denen ein Aspekt wirksam sein soll.

Mit AspectJ als Vorreiter in der aspektorientierten Programmierung sind folgende Begriffe und Konstrukte dafür eingeführt worden:

- Aspekte werden als Entitäten ersten Grades behandelt und lassen sich dementsprechend auch separat aufschreiben. Sie sind eine Erweiterung des Klassenkonzeptes und können neben Attributen und Methoden den eigentlichen Aspektcode (*advices*) sowie die Wirkungsstellen (*pointcuts*) beinhalten. Ein Aspekt wird mit dem Schlüsselwort *aspect* definiert.

- Eine Stelle im Komponentencode, an der ein Aspekt wirken soll, wird *join-point* genannt. Join-point deshalb, weil an diesem Punkt Komponentencode und Aspektcode zusammengeführt werden.

- Eine Menge von join-points kann zu einem *pointcut* zusammengefasst werden. Die Definition von pointcuts geschieht mittels pointcut-Ausdrücken, welche es erlauben join-points im Komponentencode anhand von Struktur oder Semantik zu adressieren.

- Ein *advice* beinhaltet den Code, der an den join-points eines angegebenen pointcuts ausgeführt werden soll. Der Code eines advices wirkt am jeweiligen join-point auf Methodenebene. Er kann die Methode des join-points entweder komplett ersetzen oder davor oder danach ausgeführt werden. Bezeichnet werden diese Arten als *around* (bzw. *replace*), *before* und *after*-advice. Bei einem around-advice ist es möglich, die überschriebene Methode vom advice-Code aufzurufen (*proceed()*).
  Ein advice kann auch neue Methoden, Attribute und Klassen einführen. Eine join-point-API erlaubt es, im advice-Code auf Kontextinformationen des join-points zuzugreifen.

- Das Anwenden eines Aspektes, d.h. das Einfügen dessen advice-Codes an die spezifizierten join-points wird als *Weben* (*weaving*) bezeichnet. Ein Werkzeug, das dies erledigt erhält den klanghaften Namen *Aspektweber*. Abbildung 2.2 veranschaulicht die Wirkungsweise eines solchen Tools.

2.1.2. Komposition von Aspekten und Komponenten

Um die vollständige Software aus dem Aspekt- und Komponentencode zusammenzusetzen, wird ein Aspektweber eingesetzt. Dieser vereint die Codeteile nach den vorgegebenen

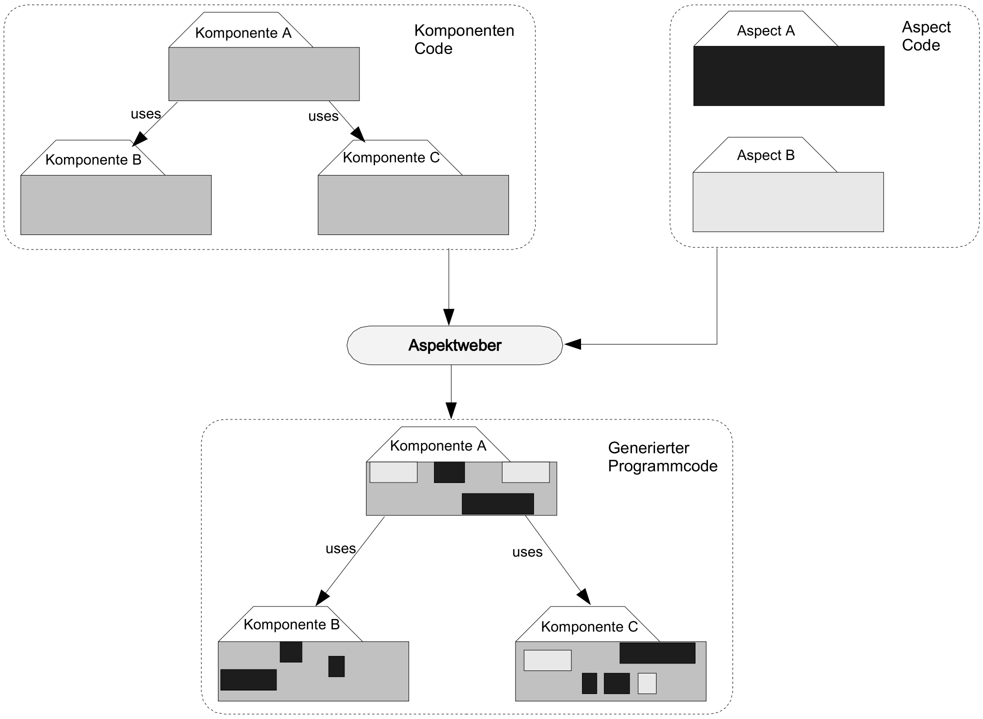

Abbildung 2.2.: Die Wirkung eines Aspektwebers

Regeln (pointcuts). Das Weben kann sowohl statisch auf Codeebene als auch dynamisch auf Objektebene erfolgen.

Beim dynamischen Weben zur Laufzeit wird nicht tatsächlich Code verwoben, sondern es werden Kompositionsbeziehungen zwischen „normalen" Objekten und „Aspekt-Objekten" hergestellt, wodurch sich die Aspekte auf die Anwendung auswirken. Der Aspekt ist dabei auch zur Laufzeit getrennt von der Basissoftware.

Durch diese Dynamik ist eine größere Flexibilität gegeben, als bei statischem Weben, welches hauptsächlich eine bessere Modularisierung und Strukturierung des Quellcodes bewirkt. Allerdings darf nicht verschwiegen werden, dass bei dynamischer Komposition aufgrund des zusätzlichen Ressourcenbedarfs durch die Aspekt-Objekte sowie der Kommunikation ein Laufzeit-Overhead entsteht.

2.1.3. Wiederverwendung von Aspekten

Da Aspektorientierung eine Erweiterung der Objektorientierung darstellt, erscheint es naheliegend, dass deren Merkmale auch für Aspekte gelten. In der Tat erlaubt AspectJ z.B. die Vererbung von Aspekten. Polymorphie wird durch *virtuelle* und *abstrakte* pointcuts und Methoden unterstützt, die in erbenden Aspekten (re-) implementiert werden können. Somit können wiederverwendbare Aspekte realisiert werden, die durch Erben und Implementierung eines abstrakten pointcuts sowie ggf. Reimplementierung von Methoden vervollständigt werden[1].

Die JoinPoint-API unterstützt die Erstellung von generischen aber auch spezialisierten Aspekten. Der Zugriff auf Kontextinformationen des join-points, wie der Referenz auf das Objekt, den Methodennamen, Parameter, Rückgabetyp eröffnet vielfältige Möglichkeiten. Einerseits können generische Aspekte realisiert werden, beispielsweise ein Aspekt, der Übergabeparameter auf Fehler (z.B. Nullpointer) überprüft. Andererseits kann ein Aspekt Spezialisierungen für bestimmte join-points beinhalten, welche anhand des Kontextes erkannt werden.

2.2. Object Teams

Object Teams ist ein neues aspektorientiertes Programmiermodell. Es kombiniert die besten Ideen verschiedener Ansätze zu einem vielseitigen und dabei kohärenten und schlanken Gesamtkonzept. Vereint werden ein Rollenmodell und ein Kollaborationsmodell mit der Möglichkeit, verfeinerbare und wiederverwendbare Module (*Teams*), größer als Klassen daraus zu erstellen. Instanziiert in eine Team-Instanz, kapseln sie eine Anzahl kollaborierender Rollenobjekte, welche innerhalb dieses Teams eine bestimmte Rolle für je ein zu adaptierendes Basisobjekt[2] spielen. Ein Team kann daher als Aspekt aufgefasst werden, der auf Objektebene wirkt. Object Teams fällt somit in die Kategorie der dynamischen Aspektweber.

Ein instanziiertes Team stellt einen Kontext dar, der aktiviert und deaktiviert werden kann. Der *aktuelle* Kontext mit seinen Rollenobjekten bestimmt das Verhalten der gebundenen Objekte. Bei mehreren aktivierten Teams entscheidet die Aktivierungsreihenfolge über die Reihenfolge, in der sie als Aspekte wirken.

Rolle - Basis-Beziehungen. Die Beziehung zwischen einem Rollenobjekt und seinem Basisobjekt ist bidirektional. Ein Methodenaufruf auf dem Basisobjekt kann dazu führen, dass eine Methode des Rollenobjektes aufgerufen wird, was *callin* genannt wird. Dabei kann die Rollenmethode nach Vorbild von CLOS in drei verschiedenen Arten aufgerufen werden:

- vor der Ausführung der Basismethode (*before*)
- nach der Ausführung der Basismethode (*after*)
- anstatt deren Ausführung (*replace*)

Bei letzterer besteht die Möglichkeit, die ursprünglichee Basismethode explizit aus der Rollenmethode heraus aufzurufen (*base-call*).

[1] Zu beachten ist, dass derzeit nur von abstrakten Aspekten geerbt werden kann. Um von einem Aspekt zu erben muss dieser also mindestens eine abstrakte Methode oder einen abstrakten pointcut besitzen.

[2] Um Verwirrung zu vermeiden, wird in der gesamten Arbeit der Begriff *Oberklasse* anstelle von *Basisklasse* verwendet, da Object Teams diesen Terminus also eine zu adaptierende Klasse der Basisanwendung definiert.

In der anderen Richtung kann ein Rollenobjekt per deklarativer Delegation eine Methode des Basisobjektes aufrufen, was als *callout* bezeichnet wird. Bei dieser Kommunikation ist zu beachten, dass die Objekte strikt innerhalb ihrer Domäne bleiben: evtl. übergebene Basisobjekte, die in das Team eintreten, werden zu Rollenobjekten *geliftet*, Rollenmethoden, die als Parameter an Basisobjekte übergeben werden, werden an der Teamgrenze zu Basisobjekten *gelowert* – was zwingend notwendig ist, da die adaptierte Basisanwendung höchstwahrscheinlich ohne Kenntnis von Rollenobjekten entwickelt wurde.

Rolle - Basis-Bindung. Zur Kompilationszeit eines Teams wird festgelegt, welche Rolle eine Basis im Team spielen soll. Es wird eine *Bindung* erstellt, mit der festgelegt wird, welche Methoden der Basis in einem Methodenaufruf der Rolle münden sollen (*callin*) und welche Basismethoden von der Rolle aufrufbar sein sollen (*callout*). Abstrakte Teams, d.h. Teams mit abstrakten Rollenklassen können mit Bindungen „vervollständigt" werden, indem abstrakte Rollenmethoden durch Deklaration von callout-Bindungen implementiert werden.

Vererbungsbeziehungen. Das Teamkonzept baut auf ein modulares Verfeinerungsmodell auf. Ein Team selbst ist eine Klasse, die Methoden und Attribute enthalten kann. Ein Team kann eine Vererbungsbeziehung mit einem anderen Team eingehen. Dabei werden die Rollenklassen eines Teams mitvererbt. Die Vererbung der Rollenfeatures beruht auf Namensgleichheit — existiert eine Rollenklasse `R` im erbenden Team wie im Oberteam, so erbt die Rollenklasse im erbenden Team sämtliche Features der anderen. Weiterhin können Rollenklassen reguläre Vererbungsbeziehungen mit anderen Klassen eingehen. Zusammen mit dem objektbasierten Erben (Delegation) der Basisklassen stellen dies drei Dimensionen von Vererbungsmöglichkeiten dar. Innerhalb dieser Dimensionen gibt es entsprechend drei Möglichkeiten, überschriebene Methoden von „Superklassen" aufzurufen:

- `super`: Mit dem von Java her bekannten Schlüsselwort wird die Methode der regulären Superklasse aufgerufen.
- `tsuper`: Analog zu super wird hier die Methode der implizit geerbten Rollenklasse aufgerufen.
- `base`: Hiermit wird das Aufrufen der durch einen replace-callin gebundenen Basismethode ermöglicht.

Modellierung. Die Team-Rollebeziehung hat einen Package-ähnlichen Charakter — mehrere Rollenklassen sind Teil eines bestimmten Teams. In [23] wird eine Notation vorgeschlagen, die diese Idee mit UML-Klassendiagrammen kombiniert.Teams werden als Packages modelliert, die jedoch wie Klassen Vererbungsbeziehungen eingehen können, und die Rollenklassen beinhalten. Die Rollenklassen innerhalb des Teams können wie normale Klassen in UML Beziehungen untereinander eingehen. Die Beziehung zwischen Team und Basispaket wird durch eine *adapt*-Relation dargestellt.

Außerdem können die Bindungen von Rollenobjekten an Basisobjekte sowie direkt die Methodenbindungen modelliert werden.

2.2.1. Fokus für diese Arbeit

Die für diese Arbeit interessantesten Ideen von Object Teams sind die *a-posteriori* Adaptionsfähigkeit zusammen mit dem Konzept der callins. Die damit ermöglichten objektbasierten Vererbungsbeziehungen, die zudem aktivierbar und deaktivierbar sind, erscheinen eine hochgradige Flexibilität zu bieten, insbesondere wenn beachtet wird, dass beliebig viele Beziehungen dieser Art gleichzeitig oder nacheinander dynamisch zu Laufzeit eingegangen werden können.

3. Aspektorientierung mit C++

Aspektorientierte Erweiterungen existieren für eine Vielfalt von Sprachen. Insbesondere die Erweiterungen für Java (AspectJ, [24], Object Teams [2], Hyper/J [25],) und Smalltalk (AspectS [26], Apostle [27]) haben eine ausgeprägte Entwicklergemeinde. Da der Fokus dieser Arbeit auf graphischen Benutzungsungsoberflächen mit Qt und KDE liegt, bin ich auf Unterstützung für die Sprache C++ angewiesen.

Trotz der weiten Verbreitung von C++ ist die Verwendung von aspektorientierten Techniken in dieser Community noch nicht besonders fortgeschritten. Bedarf sollte eigentlich reichlich vorhanden sein, da C und C++ eine breite Anwenderschar besitzen, sowohl im eingebetteten Bereich, wo mangels Ressourcen eine virtuelle Maschine nicht zum Einsatz kommen kann, als auch bei Betriebssystemen und „Desktop-Anwendungen”. Auf vielen Unices beispielsweise sind C und C++ die Standardsprachen.

Größtes Hindernis bei der Adoptierung aspektorientierter Techniken dürfte die fehlende Werkzeugunterstützung sein. Die einzigen bereits einsetzbaren Ansätze sind AspectC++[28, 29] und OpenC++ [30, 31], wobei letzterer eher als Framework zu verstehen ist, mit dem u.a. aspektorientierte Ideen umgesetzt werden können. Beide Tools werden im Folgenden vorgestellt.

3.1. AspectC++

AspectC++ ist eine aspektorientierte Spracherweiterung für C/C++. Aufgrund der Gemeinsamkeiten von Java und C++ im Hinblick auf die Grammatik und Semantik sowie der Verfügbarkeit und Popularität von AspectJ als Spracherweiterung von Java, diente dieses als Grundlage für die Entwicklung von AspectC++. Die Features und Konstrukte von AspectC++ sind daher denen von AspectJ sehr ähnlich. Lediglich die Syntax ist in einigen Details angepasst worden, um besser mit der „Host-Sprache” C++ zu harmonieren.

Wie AspectJ bietet AspectC++ einen *source-to-source* Übersetzer, der den Aspektcode statisch in den C++ Code hineinwebt[1]. Der Aspektweber `ac++` bekommt den AspectC++- und regulären C++-Code als Eingabe, webt diese zusammen, wie in Abbildung 2.2 auf Seite 38 zu sehen ist und produziert den gewobenen, wiederum regulären C++-Code als Ausgabe.

[1] AspectJ unterstützt mittlerweile auch das Einweben von Aspektcode in den kompilierten Java Bytecode, ohne den Quelltext zu modifizieren.

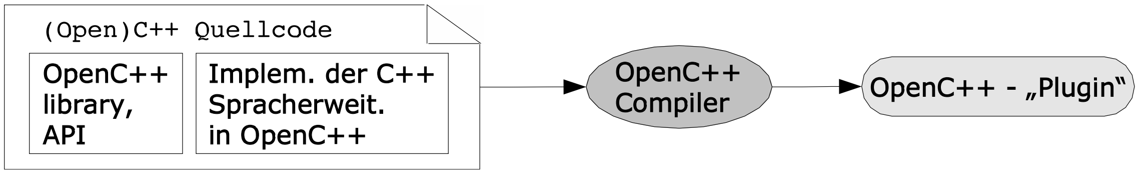

Abbildung 3.1.: Erzeugung der OpenC++ Spracherweiterungsplugins

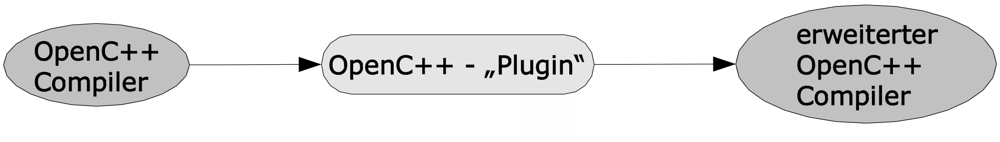

Abbildung 3.2.: Einbindung der Spracherweiterungsplugins in den OpenC++ Compiler

3.2. OpenC++

OpenC++ ist ein Toolkit zum Analysieren und Transformieren von C++ Quellcode. Es besteht aus einem Compiler `occ` und einer API zur Introspektion und Transformation des zu bearbeitenden Codes. Der Compiler selbst führt keine Transformationen durch, er bietet jedoch eine Schnittstelle für Introspektions- und Transformations-Plugins, die in den Compiler geladen und für den Kompilationsvorgang verwendet werden. Mit Hilfe dieser Plugins verarbeitet der Compiler nicht nur regulären C++ Code sondern auch um neue Sprachkonzepte erweiterten C++ Code (= OpenC++ Code). Die Abbildungen 3.1, 3.2 und 3.3 zeigen, wie der OpenC++ Compiler erweitert und eingesetzt wird.

Die Hauptanwendungsgebiete von OpenC++ sind C++ Spracherweiterungen z.B. zur Unterstützung von Parallelität, Nebenläufigkeit oder Persistenz auf Sprachebene sowie domänenspezifische Kompilationsoptimierung, z.B. durch Nutzen von domänenspezifischem Wissen, welches ein regulärer Compiler nicht besitzt [32].

3.2.1. Erweiterungsmöglichkeiten

OpenC++ ermöglicht das Einführen neuer Schlüsselwörter in die C++ Sprache. Die folgende Liste gibt einen Überlblick über die Stellen, an denen neue Schlüsselwörter verwendet werden dürfen:

- als Modifier (optional mit Parametern)

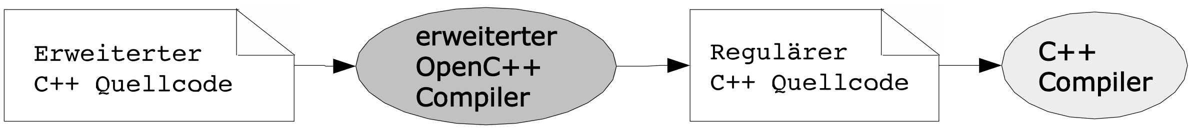

Abbildung 3.3.: Verwendung des erweiterten Compilers

  - vor der Deklaration einer Klasse, d.h. vor dem `class` Schlüsselwort
  - vor der Deklaration einer Variable / eines Methodenparameters / eines Attributes
  - vor dem `new`-Operator

- als Zugriffsspezifikator in der Klassendeklaration (anstelle von `public:`, `protected:` oder `private:`), wiederum optional mit Parametern
- als `while`-Schleifen-ähnliches Statement in der Form
`objekt . Schlüsselwort ( expression ) { statements }`
Dasselbe ist möglich mit Zeigern (`zeiger -> Schlüsselwort`) und Klassennamen (`Klassenname :: Schlüsselwort`)
- als `for`-Schleifen-ähnliches Statement, das 3 Expressions annimmt. Ansonsten ist es identisch mit der `while`-ähnlichen Variante.
- als Abschluss (closure) -Statement. Auch diese Variante ist dem while-ähnlichen Statement ähnlich. Der einzige Unterschied dazu besteht darin, dass anstatt der Expression eine Deklarationsliste erwartet wird.

3.2.2. Meta Object Protocols

Der zu bearbeitende Quellcode wird von OpenC++ in kleine Codeschnipsel zerteilt, die durch sog. *Metaobjekte* repräsentiert werden. Metaobjekte deshalb, weil sie zwar normale Objekte sind, jedoch einen Meta-Aspekt des eigentlichen Programms, nämlich einen Teil seines Quellcodes darstellen. Diese Metaobjekte — und dadurch auch der Code, den sie repräsentieren — können durch `Class`-*Metaobjekte* (Instanzen der OpenC++-*Metaklasse* `Class` bzw. deren Subklassen) transformiert werden. Metaklassen sind die vom Entwickler implementierten Analysatoren und Transformatoren sowie deren OpenC++ Oberklassen. Diese Metaklassen werden „normalen”, zu transformierenden Klassen zugeordnet, so dass beispielsweise `PersistentClass` die Metaklasse von `MyAddressbookEntry` sein könnte.

OpenC++ ist eine „selbstreflektive” Sprache [33]. Die vom Entwickler implementierten Metaklassen müssen selbst mit dem OpenC++ Compiler übersetzt werden, der einige Transformationen an ihnen durchführt. Diese Transformationen sind in der Metaklasse `Metaclass` definiert, welche wie alle anderen Transformatoren von `Class` erbt. Dies bedeutet, dass `Metaclass` die Metaklasse von `Class` sowie all ihren Subklassen und damit auch sich selbst ist. Skizziert sind diese Relationen in Abbildung 3.4, welche dankbar [31] entnommen ist.

Das Zusammenspiel von Objekten, Metaobjekten und `Class`-Metaobjekten wird *Meta Object Protocol* (MOP)[2] genannt. OpenC++ verwendet ein sog. Kompilationszeit-MOP (*CT-MOP*), das nur für den Kompilationsvorgang verwendet wird. Damit verursacht

[2]Das Wort „Protokoll” steht in diesem Kontext für die Schnittstellen von Metaobjekten, z.B. die Introspektions- und Transformationsschnittstellen und wie diese verwendet werden.

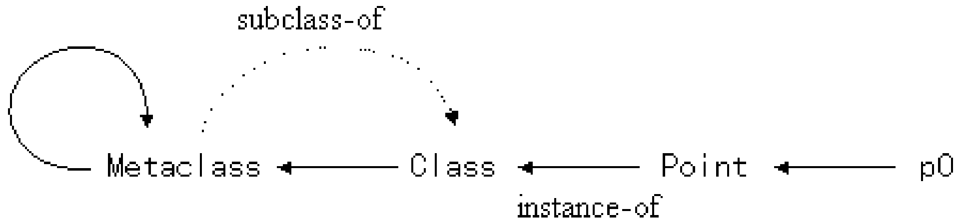

Abbildung 3.4.: Die Relationen von `Metaclass`, `Class`, einer Klasse `Point` und einer Instanz davon

es im Gegensatz zu Laufzeit-MOPs (*RT-MOPs*) keinen Overhead, da keine Metaobjekte zur Laufzeit des Programmes existieren [33]. Die Metaobjekte können daher jedoch nur Codeschnipsel, d.h. syntaktische, statische Aspekte des Programmes darstellen und verarbeiten [34]. Die Codeschnipsel sind als Parsierbäume (*parse trees*) strukturiert.

Im Gegensatz dazu stehen die Metaobjekte von zur Laufzeit existierenden RT-MOPs in Relation zu „richtigen" Objekten des Programms und können diese somit zustandsabhängig und dynamisch adaptieren.

Das OpenC++ MOP bietet strukturierten Zugriff auf den Quellcode (*compile time reflection*). Mit Hilfe der Transformatoren erlaubt es, neue Syntax und neues semantisches Verhalten in C++ einzubetten. Auch die Modifikation von Standardmechanismen, wie Methodenaufrufe, Attributzugriff oder Objekterzeugung wird ermöglicht.

Einstiegspunkt in das MOP ist die OpenC++ Metaklasse `Class`. Sie repräsentiert eine Klasse und bietet dementsprechend Zugriff auf deren Name, Namespace, Attribute, Methoden und Oberklassen. Neben Zugriffsmethoden für die Introspektion sind durch sie folgende Transformationsmöglichkeiten gegeben:

- Die Klasse kann vollständig entfernt werden.
- Der Name kann verändert werden.
- Die Liste der Oberklassen kann verändert / entfernt werden, inklusive der Spezifikatoren (`public`, `protected`, `private`, `virtual`).
- Attribute und Methoden können hinzugefügt, verändert und entfernt werden. Änderungen an existierenden Attributen und Methoden, z.B. das Ändern der Parameterliste können mittels der Klasse `Member` getätigt werden.

Weiterhin können beliebige Änderungen direkt auf Quelltextebene durchgeführt werden.

3.2.3. Metaklassen / Plugins

Beliebig viele Transformationsplugins können mit OpenC++ verwendet werden. Genau genommen stellt ein Plugin eine Metaklasse dar. Es erbt direkt oder indirekt von der Metaklasse `Class` und ist somit eine eigene Metaklasse.

Listing 3.1: Zuordnung einer Klasse Person zur Metaklasse LoggingClass.

```
1  #ifdef __opencxx
2  metaclass LoggingClass Person;
3  #endif
4
5  class Person
6  {
7  public:
8      Person( const std::string& name) : m_name( name ){}
9      void setName( const std::string& name ){ m_name = name; }
10     string getName() const { return m_name; }
11 private:
12     std::string m_name;
13 };
```

Listing 3.2: Implementierung der Metaklasse LoggingClass.

```
class LoggingClass : public Class
{
public:
    virtual Ptree* TranslateMemberCall( Environment* env, Ptree* object, Ptree*
                 op, Ptree* member, Ptree* arglist)
    {
        return Ptree::Make("(std::cerr << \"%p() \" << std::endl), %p)", member,
                 Class::TranslateMemberCall(env, object, op, member, arglist) );
    }
};
```

Damit der OpenC++ Compiler weiß, welches Plugin für die Kompilierung einer Klasse verwendet werden soll, muss die Klasse einer bestimmten Metaklasse zugeordnet werden. Dies geschieht durch Verwendung des Schlüsselwortes `metaclass` vor der Klassendeklaration[3]. Dadurch wird der Compiler informiert, welche Metaklasse für die Kompilation eingesetzt, d.h. welches Plugin geladen werden soll.

Im Beispiellisting 3.1 wird in Zeile 2 die Klasse `Person` der Metaklasse `LoggingClass` zugeordnet, welche sodann vom OpenC++ Compiler für die Übersetzung von `Person`

[3]Da ein normaler C++ Compiler die `metaclass`-Deklaration mit einer Fehlermeldung ahnden würde, wird sie typischerweise per `#ifdef __opencxx` vor diesem versteckt (der Präprozessor entfernt den gesamten Block, wenn `__opencxx` nicht definiert ist). Bei Kompilation mit dem OpenC++ Compiler ist dieses Makro definiert, sodass das `metaclass`-Statement effektiv ist.

Dies erlaubt die Verwendung der Klasse sowohl in „Reinform" mit beliebigen C++ Compilern als auch mit Transformationen mit Hilfe des OpenC++ Compilers. Ersteres macht selbstverständlich nur Sinn, wenn reguläres C++ verwendet und OpenC++ beispielsweise nur für Kompilationsoptimierung eingesetzt wird.

Listing 3.3: Ausgabe eines Programms welches die mit LoggingClass modifizierte Klasse Person verwendet.

```
int main()
{
    Person person( "Carsten" );
    std::string fullName = person.getName() + " Pfeiffer";
    std::cerr << fullName << std::endl;
    return 0;
}

Ausgabe:
getName()
Carsten Pfeiffer
```

verwendet würde. Die Implementierung der Metaklasse `LoggingClass` ist in Listing 3.2 zu bewundern. `LoggingClass` reimplementiert die Methode `Class::TranslateMemberCall()`, welche während des Kompiliervorganges für jeden gefundenen Methodenaufruf in der gerade kompilierten Klasse aufgerufen wird. Die Parameter haben dabei folgende Bedeutung:

- `Environment *env` stellt den aktuellen *Scope* dar.
- `Ptree *object` ist das Objekt, auf dem die Methode aufgerufen wird
- `Ptree *op` bezeichnet den Dereferenzierungsoperator ('`.`' oder '`->`', je nachdem, ob `object` ein Objekt oder ein Zeiger ist).
- `Ptree *member` beinhaltet den Namen der Methode.
- `Ptree *arglist` referenziert die Parameter des Methodenaufrufs.

Der Rückgabewert von `TranslateMemberCall()` wird in den Syntaxbaum eingefügt, wobei die Standardimplementierung `Class::TranslateMemberCall()` ein `Ptree` Metaobjekt erzeugt, das den Originalcode des Methodenaufrufs repräsentiert. Mit `Ptree::Make()` ist es möglich, neue Code repäsentierende Metaobjekte zu erzeugen. Code kann direkt als Zeichenkette angegeben werden, existierende `Ptree` Objekte können durch den Platzhalter `%p` hinzugefügt werden. Das resultierende Metaobjekt im Beispiel beinhaltet damit folgenden Code:

`(std::cerr << "[Name der Methode]()" << std::endl ), [ originaler Inhalt der Methode]`
wodurch eine simple Form des *Loggings*[4] gegeben ist, da vor jeder Ausführung einer Methode der Methodenname auf der Standardfehlerausgabe ausgegeben wird (siehe Listing 3.3).

[4]Logging bezeichnet das Protokollieren bestimmter Ereignisse, in diesem Fall aller Methodenaufrufe.

Listing 3.4: Delegation der Transformation eines PtreeTypedef Metaobjektes an den Visitor

```
Ptree* PtreeTypedef::Translate(Walker* w)
{
    return w->TranslateTypedef( this );
}
```

3.2.4. Parse Trees (Ptree-Metaobjekte)

Die interne Repräsentation von Code innerhalb eines Compilers ist typischerweise ein abstrakter Syntaxbaum (*abstract syntax tree*, *AST*). Dabei repräsentiert jeder Knoten des Baumes einen Operator und alle Kinder des Knotens die Operanden [35]. Im Unterschied zu Parsierbäumen oder konkreten Syntaxbäumen (*concrete syntax trees*, *CST*), stellen ASTs keine 1:1 Repräsentation, sondern nur eine semantisch äquivalente Form des Codes dar. Dinge, die für den Übersetzungsprozess nicht notwendig sind, erscheinen nicht in abstrakten Syntaxbäumen.

Da OpenC++ jedoch ein *source-to-source* Übersetzer ist, bei dem alle Informationen des Quelltextes für den Übersetzungsprozess genutzt werden können sollen (auch Kommentare), kommt dort kein AST, sondern ein konkreter Syntaxbaum zum Einsatz.

Der Quellcode einer parsierten Klasse wird im OpenC++ MOP auf `Ptree`-Metaobjekte abgebildet, welche hierarchisch in geschachtelten Listen strukturiert sind. `Ptree` ist eine abstrakte Oberklasse für konkrete Klassen, die sämtliche in C++ vorkommende Konstrukte (Schlüsselwörter, Expressions, Statements, Blöcke, Deklarationen etc.) repräsentieren.

Transformation. Die Transformation der Metaobjekte — und damit auch des Quellcodes, den sie darstellen — geschieht durch ein Plugin in einer dem *Visitor* Entwurfsmuster [6] ähnlichen Weise. Ein Transformationsvisitor (`ClassWalker`) wird erzeugt und als Parameter im Methodenaufruf `Translate()` dem `Ptree` Wurzelknoten des Syntaxbaumes übergeben. `Ptree::Translate( Walker * )` ist dabei das Äquivalent zum `Element::accept( Visitor * )` des Visitor-Musters. Jede konkrete `Ptree`-Klasse beinhaltet eine eigene Implementierung dieser Methode, in der die für diese Klasse spezialisierte Methode des Visitors aufgerufen wird, wie in Listing 3.4 beispielhaft für die Klasse `PtreeTypedef` demonstriert ist.

Die Einbindung eines Transformationsplugins in den Kompilationsprozess geschieht mittels einiger *Hook-Methoden [36]* der Metaklasse `Class`, welche ein Plugin reimplementiert. Diese werden an geeigneten Stellen vom `ClassWalker`-Visitor aufgerufen, wodurch das Plugin die Gelegenheit erhält, Analysen oder Transformationen durchzuführen. Einige wichtige dieser Transformations-Template-Methoden sind im folgenden beispielhaft aufgelistet:

- `virtual void Class::TranslateClass(Environment*)`
  Innerhalb dieses Hooks kann die Deklaration der Klasse (sowie *inline*-Implementierungen

von Methoden innerhalb der Deklaration) modifiziert werden.

- `virtual void Class::TranslateMemberFunction(Environment*, Member& member)`
  Hier kann die Implementierung der Methode `member` transformiert werden.

- `virtual Ptree* Class::TranslateMemberWrite(Environment*, Ptree* object, Ptree* access_op, Ptree* member_name, Ptree* assign_op, Ptree* expr)`
  Dieser Hook ermöglicht die Transformierung eines „Schreibzugriffs" auf das Object `object`. `access_op` spezifiziert, ob es sich dabei um ein Objekt oder einen Zeiger handelt ('.' oder '->'). `member_name` bezeichnet den Namen des Objektes. `assign_op` trägt den Zuweisungsoperator ('=', '+=', '-=', etc.) und `expr` beinhaltet den Ausdruck rechts davon.

Die vollständige Liste ist in der Klassendokumentation unter [37] zu finden.

3.2.5. Aspektorientierung mit OpenC++

OpenC++ ist selbst kein aspektorientiertes Werkzeug. Der Grund, aus dem es hier vorgestellt wurde liegt in seiner Eignung als Werkzeug zum *Implementieren* von aspektorientierten Funktionalitäten. Wer den Bereits mit der Basisfunktionalität lassen sich ohne große Probleme *before*- und *after-methods* implementieren.

Die offene Architektur von OpenC++ erlaubt es, tief in den Kompilations-/Adaptionsprozess einzugreifen. Entspricht der Standardkompilationsprozess nicht den eigenen Anforderungen, so können eigene *Walker* und *Driver* implementiert werden, um den Parsierbaum selbst aufzubauen oder zu traversieren.

4. Realisierung von Object Teams für C++

Im Rahmen dieser Diplomarbeit sollte untersucht werden, inwieweit die Anpassbarkeit und Erweiterbarkeit komplexer Software verbessert werden kann. Bei Verwendung von Klassenbibliotheken wie Qt (siehe Abschnitt 1.3.1 auf Seite 12) oder JFC[1], deren Quellcode selbst nicht geändert werden kann oder soll, können Anpassungen und Erweiterungen u.a. erreicht werden durch

- explizit in der Software vorgesehene Konfigurationsmöglichkeiten,
- Erben bestimmter Klassen und Überschreiben von Methoden,
- Implementieren von *Hooks*, die speziell für die Anpassung des Standardverhaltens bereitgestellt werden,
- Einhängen in die Ereignisbehandlungskette.

Oft ist es jedoch nicht vorgesehen, eigene Klassen zu instanziieren oder mittels einer *Factory* instanziieren zu lassen und so in existierende Kollaborationen zu integrieren. In dem Fall ist die Integration in die Ereignisbehandlung, d.h. das Abfangen und die Behandlung von Ereignissen, der einzige Weg, das Verhalten der Software zu ändern (vgl. Abschnitt 1.4.2 auf Seite 26). Jedoch ist man dabei der Voraussicht der Entwickler ausgeliefert: es können nur solche Ereignisse abgefangen und verarbeitet werden, die bei der Entwicklung der Softwarekomponente bedacht und implementiert wurden.

Das Object Teams-Konzept bietet die Möglichkeit mit Hilfe von callins Methodenaufrufe in existierende Methoden einzuschleusen (siehe Kapitel 2.2). Dies kann z.B. dazu verwendet werden um unvorhergesehene Ereignisbenachrichtigungen nachträglich einzubinden. Wird ein bestimmter Zustand einer Klasse nicht durch ein Ereignis signalisiert, so kann ein Interessent dieses Zustandes sich selbst eine Benachrichtigung über das Eintreten des Zustandes erzeugen.

Die Adaptivität, die dadurch erreicht wird, war Grund genug, eine Implementierung von Object Teams für C++ anzustreben, welche im Folgenden beschrieben wird.

[1] Java Foundation Classes

4.1. Allgemeines

Die Unterstützung von Object Teams für C++ erfordert die Fähigkeit, C++ Klassen zu analysieren und zu transformieren. Anfangs erschien es möglich, die Grundkonzepte mit einem einfachen Parser, z.B. einer Erweiterung des `moc` (siehe Abschnitt 1.3.2 auf Seite 19), einigen Makros sowie AspectC++ für die Transformationen zu realisieren. Dies hat sich jedoch als nicht praktikabel erwiesen, als die Probleme der Copy-Inheritance (siehe Abschnitt 4.4) erkannt wurden.

Eine Alternative war, C++ Code in eine XML-Repräsentation umzuwandeln und diese unter Verwendung eines Standard-XML-Parsers zu analysieren. Auch Transformationen sind auf XML-Ebene relativ einfach möglich. Die XML-Repräsentation des `gcc-xml`[2] enthält jedoch leider keine Methodenrümpfe, so dass auch dieser Ansatz verworfen wurde.

Nur ein ausgereiftes C++ Analyse-/Transformationstoolkit ist der Aufgabe gewachsen, wie die folgende Liste der zu beantwortenden Fragen deutlich macht:

- Zu welchem Team gehört eine Rollenklasse?
- Welche Superklassen hat ein Team, eine Rollenklasse?
- Existiert eine bestimmte Rollenklasse in einem Team?
- Welche Methoden und Attribute gibt es – sind sie öffentlich, geschützt, privat, virtuell, abstrakt, statisch?
- An welchen Stellen wird auf bestimmte Attribute und Methoden zugegriffen?
- Wie sehen die Signaturen gebundener Basismethoden aus?

Transformationsunterstützung wird benötigt für

- das Umbenennen von Attributen, Methoden,
- die Verwendung von Attributen und Methoden,
- die Erzeugung von Attributen, Methoden, Klassen.

Angeboten hat sich dazu das OpenC++ Toolkit, das bereits in Abschnitt 3.2 vorgestellt wurde.

Im Gegensatz zu ObjectTeams/Java [38, 39], das einige Object Teams-Details erst zur Ladezeit in der Java Virtual Machine realisiert, ist ObjectTeams/C++ vollständig als Compiler umgesetzt. Lediglich die Anwendung von callins benötigt Unterstützung des *runtime-linkers* (siehe Abschnitt 4.3).

[2] http://www.gccxml.org

4.2. Der Object Teams Compiler `otcpp`

Mit Hilfe von OpenC++ wurde ein Compiler `otcpp` entwickelt, der die Sprache C++ um die Object Teams Sprachelemente erweitert. Diese werden in regulären C++ Code transformiert, welcher die entsprechende Semantik umsetzt. Der resultierende Code wird von einem normalen C++-Compiler übersetzt. Für die Entwicklung wurde keine reguläre Version von OpenC++ verwendet, sondern stets die Entwicklerversion im *CVS-Repository*. Obwohl im Laufe dieser Arbeit mehrere Versionen veröffentlicht wurden, ist die aktuelle Version 2.6.1 nicht ausreichend, um `otcpp` zu kompilieren. Die nächste Version wird jedoch voraussichtlich als stabile Entwicklungsbasis für den `otcpp` dienen können.

4.2.1. Syntax

Die vom `otcpp` verarbeitete Syntax ist der ObjectTeams/Java Syntax angelehnt, differiert jedoch in einigen Details, um eine „rundere” Integration in die Sprache C++ zu erreichen. Ferner ist die Syntax an die Erweiterungsmöglichkeiten von OpenC++ gekoppelt. Im Folgenden werden die verwendeten Schlüsselworte und Konstrukte mit Beispielen dargestellt sowie ihre Auswirkungen erläutert.

Teams. Eine Teamklasse wird durch Voranstellen des Schlüsselwortes `team` vor das `class`-Spezifikationssymbol deklariert. Nur damit wird es als Team erkannt und erhält die gewünschte Unterstützung vom Compiler. Jedes Team, das nicht explizit von einem anderen Team erbt, erhält implizit die Oberklasse `ObjectTeams::Team`. Diese bietet öffentliche Methoden für die Aktivierung und Deaktivierung (`void setActive( bool )` sowie die äquivalenten Methoden `void activate()` und `void deactivate()`).

Die Kopplung von Rollen an ein Team geschieht im Team durch eine `roles:` Sektion, in der alle zu kapselnden Rollenklassen als `class RollenName;` aufgeführt werden müssen.[3]

Rollen. Die Deklaration von Rollenklassen geschieht durch das Schlüsselwort `role`, das wiederum vor dem `class`-Spezifikationssymbol stehen muss.

callin-Methoden werden mit einem eigenen Zugriffsspezifikator gekennzeichnet, welcher sinnvollerweise `callins:` lautet. Alle folgenden Methoden in dieser Sektion werden als callins behandelt. Diese Syntax verdeutlicht, dass callins weder öffentlich, geschützt, noch privat sind und nicht direkt aufgerufen werden können.

callout-Methoden werden wie normale Methoden deklariert, erhalten zusätzlich aber einen `callout` Modifier, in der Art: `callout void foo()`[4]. In ihrer Implementierung

[3]In einem ersten Ansatz wurde die Kapselung ähnlich ObjectTeams/Java durch *inner* (*nested* in C++) classes erreicht. Qts Meta-Object-Compiler `moc` kann jedoch nicht mit geschachtelten Klassen umgehen, was die Verwendung von Signalen und Slots verhindert (s. Abschnitt 1.3.2 auf Seite 14 ff. sowie Seite 19).

[4]Idealerweise sollte die Syntax der Deklaration von Slots ähneln und demnach `public callouts:` sein. Dies ist mit der aktuellen Version von OpenC++ noch nicht möglich.

wird das `base`-Attribut verwendet, auf dem die zu bindende Basismethode einfach aufgerufen wird. Dieses explizite Aufrufen löst gleichzeitig das Problem der Parametermappings, da der Entwickler die Übergabeparameter sowie deren Reihenfolge frei wählen kann. Diese Umsetzung steht im Kontrast zu der deklarativen Bindung von callouts in ObjectTeams/Java (*bindings).*

Jedes Rollenobjekt hat ein privates Attribut `m_team` vom Typ seines Teams, das es verwenden kann, um Team-level Methoden aufzurufen.

Bindungen. Eine Rollenklasse wird an eine Basisklasse gebunden, indem das Schlüsselwort `role` die Parameter `playedBy BasisKlasse` erhält. Beispiel:
`role ( playedBy DataObject ) class PersistentClass {...}`

Die Bindungen von Basismethoden an Rollenmethoden geschehen in einer Methode namens `bindings()`. In dieser sind die Aufruf `replace()`, `after()` und `before()` erlaubt, welche als ersten Parameter die Signatur der Basismethode als Zeichenkette und als zweiten Parameter das Schlüsselwort `callin` (bei `before`- und `after`-Bindungen) bzw. `with` (bei einer `replace`-Bindung) erwartet. Die Schlüsselwörter `callin` und `with` erwarten die Signatur der zu bindenden Rollenmethode als Zeichenkette.

Team-Methoden, die zu liftende Basisobjekte als Parameter haben, können dies mit dem Schlüsselwort `regard` verdeutlichen. Dieses Schlüsselwort wird vor die Deklaration des Rollenobjektes gestellt und muss als Parameter den Basistyp sowie das Schlüsselwort `as` enthalten muss:
`void someTeamMethod( regard(BasisKlasse as) RollenKlasse *rolle )`

Veranschaulicht ist die Syntax noch einmal in einem vollständigen Beispiel in Listing 4.1.

4.3. Callins

Das Hauptaugenmerk der Umsetzung von Object Teams für C++ lag auf der Unterstützung von callins — dem Aufrufen von Rollenmethoden aus Basismethoden heraus. Dieses Feature erfordert die Fähigkeit, (Basis-) Methoden existierender Klassen zu verändern. ObjectTeams/Java nutzt die Flexibilität der Java Virtual Machine (JVM) aus und modifiziert die zu ändernden Klassen durch Bytecodetransformationen während des Ladens.

Da C++-Code jedoch in der Regel direkt in Maschinencode übersetzt wird, besteht diese Option für ObjectTeams/C++ nicht. Anfangs wurde in Betracht gezogen, eine statische Adaption durchzuführen, d.h. den Quellcode der zu adaptierenden Basisklassen vorauszusetzen und diesen zu transformieren. Dies hätte jedoch die Rekompilation des Basiscodes nach sich gezogen und damit — sofern dieser Teil einer Bibliothek ist — *alle* Programme betroffen, die diesen Code nutzen. Da in Kapitel 5 die Adaption von Bibliotheksklassen thematisiert wird, erschien dies impraktikabel.

Listing 4.1: Beispiel einer Team-Definition mit gebundenen Rollenklassen

```
team class MeinTeam
{
roles:
    class RoleA;
    class RoleB;

public:
    void doSomething( regard(Base as) RoleA *role )){...} // Team-level Methode
private:
    int meinStatus;
};

role( playedBy Base )class RoleA
{
void bindings() {
    // before( "baseMethod( bool )", callin("hook( bool )") );
    replace( "baseMethod( bool )", with("hook( bool )") );
}

public:
    callout double calculate( int value, double factor )
    {
        return base->doIt( factor, value );
    }
callins:
    void hook( bool activate )
    {
        base->hook( false );
    }
};
```

4.3.1. Überschreiben auf Symbolebene

Nachdem die statische Adaption verworfen wurde, musste eine Möglichkeit gefunden werden, zur Linkzeit, Ladezeit oder zur Laufzeit Modifikationen durchzuführen. Nach der Kompilation einer Klasse ist es zwar kaum möglich, Methoden zu verändern, es können jedoch „Reimplementierungen" hinzugefügt werden.

Der Trick besteht darin, eine neue Methode zu erzeugen, die exakt die gleiche Signatur besitzt, wie die zu adaptierende Methode. Das „gelinkte" Programm erhält dadurch zwei Mal das gleiche Symbol, einmal direkt im Programm und einmal in der zu ladenden Bibliothek. Sämtliche Aufrufe dieser Methode, egal ob sie aus dem Programm oder der Bibliothek stammen, verwenden das erste Symbol und rufen somit die hinzugefügte Methode anstatt der Originalmethode auf, da die Symboltabelle erst mit den Symbolen des Programms selbst und darauf folgend mit denen der verwendeten Bibliotheken gefüllt wird.

Damit ist zunächst erreicht, dass der Kontrollfluss aus existierenden Methoden herausgeleitet werden kann. Um die Originalmethode aufrufen zu können, muss der *runtime linker* zu Rate gezogen werden. Aufgefordert durch den Bibliotheksaufruf
`dlsym(RTLD_NEXT, "symbol")`[5]
beschafft dieser einen generischen (`void *`) Zeiger auf das nächste Symbol in der Symboltabelle mit dem Namen `"symbol"`. Dieser muss zu einem Funktionspointer mit der vollständigen Signatur der Methode gecastet werden. Zu beachten ist, dass bei jeder Objektmethode ein implizites Argument vorhanden ist, nämlich der `this`-Zeiger, der als `struct Klasse *`, bzw. `const struct Klasse *` bei konstanten Methoden übergeben wird. Dieser ist immer der allererste Parameter.

Fortan kann dieser Funktionspointer benutzt werden, um die „überschriebene" Methode aufzurufen. Ein Beispiel, wie das Ersetzen einer Methode und das Aufrufen der Originalmethode funktioniert ist in Listing 4.2 zu sehen.

Der `dlsym()`-Aufruf zur Erfassung der Referenz auf die Originalimplementierung erfordert die Kenntnis des Symbolnamens der gewünschten Methode. Der folgende Abschnitt erläutert die Relation von Funktionsnamen im Quellcode und deren Symbolnamen im Binärcode.

Name Mangling. Jeder verfügbare Typ, jede Funktion und jede Methode erhält im kompilierten Code einen eindeutigen Namen, um referenziert werden zu können. In der Sprache C ist das ganz einfach der Name des Typs oder der Funktion. Da es weder *Overloading*[6], noch ein Klassenkonzept gibt, so dass sich alle Funktionen hierarchielos auf einer Ebene befinden, muss jede Funktion einen eindeutigen Funktionsnamen haben, der auch direkt als Symbolname verwendet wird. Daher „gruppieren" C-Programmierer Funktionen eines „Moduls" häufig durch geschachtelte Namensgebung, wie am Beispiel der GTK-Bibliothek in Listing 4.3 ersichtlich ist. Die Namen beginnend mit `gtk_` bezeichnen die Symbole der gleichnamigen Funktionen.

[5] `dlsym()` ist Bestandteil der GNU- sowie wenigstens der Solaris-Dynamic-Linking-Bibliothek.
[6] Overloading ist die Bezeichnung für das Konzept, mehrere Methoden gleichen Namens zu verwenden, die nur bezüglich ihrer Argumentliste differieren.

Listing 4.2: Das 'Überschreiben' der Methode QWidget::show() der Qt-Bibliothek

```
#include <qwidget.h>

// _ZN7QWidget4showEv ist der "gemanglete"Name von QWidget::show(void).
typedef void (*orig_QWidget_show_ptr) ( const struct QWidget * );
static orig_QWidget_show_ptr orig_QWidget_show =
      (orig_QWidget_show_ptr) dlsym( RTLD_NEXT, "_ZN7QWidget4showEv" );

void QWidget::show()
{
    std::cerr << "Gleich wird die Originalmethode aufgerufen..." << std::endl;
    orig_QWidget_show( this );
}
```

Listing 4.3: Ausschnitt der Ausgabe von 'nm /usr/lib/libgtk.so' einer C-Bibliothek

```
001106a0 T gtk_tooltips_disable
00110d10 t gtk_tooltips_draw_tips
00110630 T gtk_tooltips_enable
```

Das Klassenkonzept von C++ mit Klassenhierarchien, Namensräumen und Overloading erfordert im Vergleich zu C die Erfassung mehr als des Funktionsnamens im Symbolnamen. Neben Klassennamen und Namensraum muss u.a. die vollständige Signatur einer Methode, d.h. die Typen all ihrer Parameter in den Symbolnamen einfließen, um Eindeutigkeit zu gewährleisten.

Bemerkenswert ist, dass der Rückgabetyp einer Methode nicht in den Symbolnamen einfließt, da er keine Auswirkung auf dessen Eindeutigkeit hat — Overloading von Methoden nur anhand des Rückgabetyps ist nicht möglich.

Leider schreibt der C++-Standard nicht vor, wie das *Name Mangling* — das Kodieren dieser Informationen in einen Symbolnamen — auszusehen hat. Symbolnamen sind demnach compiler- und versionsspezifisch, was z.B. zur Folge hat, dass C++-Programme und die Bibliotheken, die sie benutzen, mit dem selben Compiler übersetzt werden müssen.

Glücklicherweise wurde jedoch kooperativ eine ABI-Spezifikation (*Application Binary Interface*) für C++-Compiler für den Intel Itanium von einer Reihe namhafter Firmen entwickelt [40]. Diese Spezifikation definiert u.a. auch das Name Mangling. Da dieses nicht prozessorspezifisch ist, hat die Spezifikation bereits jetzt einen standardisierenden Charakter. Der GNU C++ Compiler setzt diese Art des Manglings seit der Version 3.2 für alle unterstützten Architekturen um. Das Mangling einiger überladener Methoden ist in Listing 4.4 beispielhaft dargestellt.

Da der gcc-Distribution zwar ein Tool namens `c++filt` zum Dekodieren von Symbolnamen in die Quellcode-Repräsentation beiliegt, jedoch keines für den umgekehrten Fall, war es nötig, die Mangling-Spezfikation selbst umzusetzen, um die Adaptierung der in

Listing 4.4: Einige überladene Methoden und ihre 'gemangleten' Symbole

```
class Test;
class Overloading
{
public:
    void setSomething();
    virtual bool setSomething( int );
    void setSomething( int, bool );
    void setSomething( Test * );
};
// Nicht-Inline Implementierungen ...

// Ausgabe von nm:
00000000 T _ZN11Overloading12setSomethingEv
00000010 T _ZN11Overloading12setSomethingEi
00000020 T _ZN11Overloading12setSomethingEib
00000030 T _ZN11Overloading12setSomethingEP4Test
```

den Konnektoren angegebenen Basismethoden zu ermöglichen.

Nachdem damit die technischen Voraussetzungen für die Realisierung von callins erfüllt sind, wird im folgenden Abschnitt die Implementierung der überschreibenden Methoden vorgestellt, die die Funktionalität der before-, after- und replace-calls bereitstellen.

4.3.2. Dispatching

Ein callin lenkt den Kontrollfluss aus einer Basismethode in eine gebundene Rollenmethode (siehe Abschnitt 2.2). Folgende Möglichkeiten sind dabei zu beachten:

1. Abhängig vom *Binding* kann die Rollenmethode bevor, nach oder anstatt der Basismethode aufgerufen werden. Letztere Option ermöglicht das explizite Aufrufen der Basismethode aus der Rollenmethode (*base-call*).

2. Die Methode eines Basisobjekts kann an Methoden unterschiedlicher Rollenobjekte einer Teaminstanz gebunden sein.

3. Es kann mehrere Instanzen dieses Teams und damit der Rollenobjekte geben.

4. Das Basisobjekt kann weiterhin an Instanzen anderer Teams gebunden sein.

Abbildung 4.1 verdeutlicht diese Relationen in einer Art Kollaborationsdiagramm, das um das Teamkonzept nach UFA-Manier erweitert wurde.

Im Gegensatz zum Aufruf von Slots durch Emission eines Signals (vgl. Abschnitt 1.3.3 auf Seite 22) ist die Reihenfolge der aufzurufenden Rollenmethoden klar festgelegt: sie wird

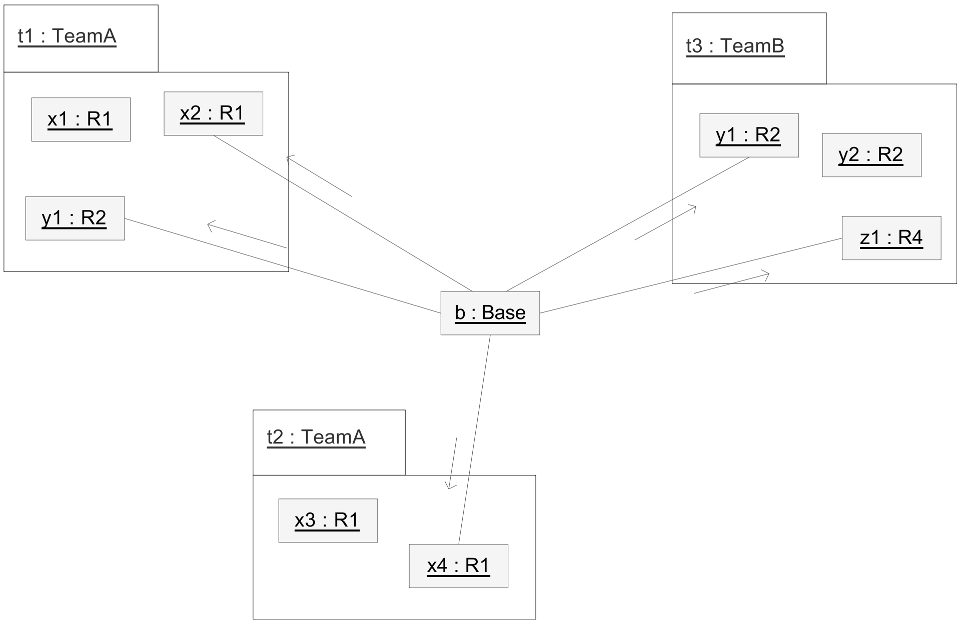

Abbildung 4.1.: Flexibilität bei callins: die Möglichkeiten der Bindungen von Basisobjekten an Rollenobjekte.

durch die Aktivierungsreihenfolge der betroffenen Teams festgelegt. Das zuletzt aktivierte Team hat dabei den größten Einfluss: dessen *before-gebundene* Rollenmethoden werden als allererstes und dessen *after-gebundene* Rollenmethoden werden als allerletztes aufgerufen.

Das dadurch notwendige *Dispatching* von callins — das Aufrufen der unterschiedlichen Rollenmethoden abhängig von der Teamaktivierung wird im Folgenden beschrieben.

Kontextaktivierung und callin-Aufrufe. Alle aktivierten Team-Instanzen werden in einer einzigen, sortierten Liste geführt, welche die Team-Priorisierung für callins festlegt. Sobald ein Team (de)aktiviert wird, entfernt die Implementierung von `Team::setActive( bool active )` das Team aus dieser Liste und hängt es anschließend ans Ende an, sofern `active true` ist. Im Falle einer geschachtelten Teamaktivierung, d.h. wenn ein bereits aktives Team erneut aktiviert wird, bleibt das Team am aktuellen Platz in der Liste.

Alternativ könnte es an den vordersten Platz der Liste wandern, aber dann sollte es konsequenterweise wieder an den alten Platz zurückwandern, sobald es wieder (*einmal*) deaktiviert wird. Da dies jedoch nicht eindeutig möglich ist, haben geschachtelte Teamaktivierungen keine Auswirkungen auf die Priorisierung.

Erst wenn das Team genauso oft deaktiviert wie aktiviert wurde, wird es *wirklich* deaktiviert, d.h. aus der Liste gelöscht. Hierfür wird der `active`-Status in einer Variablen gespeichert, die die Aktivierungen und Deaktivierungen zählt. Mit `bool Team::isActive()` kann der Status abgefragt werden, d.h. ob die Variable größer oder gleich 0 ist. Zu häufige Deaktivierungen (d.h. wenn die Zählvariable negativ würde) werden mit einer Laufzeitwarnung bestraft.

Listing 4.5: Die Verwendung von Zeigern für Rückgabewerte bei callins

```
bool BaseClass::isSomething() // Dies ist die generierte Methode
{
    bool * result = 0L;
    // Iteration für before-Methoden....
        result = &(role->roleMethod());

    // Iteration für replace-Methoden...
        result = &(role->roleMethod());
        if ( keinBaseCall )
            return *result;

    result = &(orig_BaseClass_isSomething( this ));
    // Iteration für after-Methoden
        result = &(role->roleMethod());

    return *result;
}
```

Der `otcpp`-Compiler generiert für jede callin-Bindung eine Methode, die den gleichen Namen und Klassennamen der Basismethode erhält und diese somit „überschreibt”, wie in Abschnitt 4.3.1 dargestellt. Die Implementierung dieser Methode enthält den Code zum Aufrufen sämtlicher gebundener callin-Rollenmethoden.

Es wird maximal 3 Mal über die Liste der aktiven Teams iteriert — einmal für *before*-, einmal für *replace*- und einmal für *after*-Bindungen. Um die Aktivierungsreihenfolge zu honorieren, wird für erstere beiden Bindungen rückwärts über die Liste iteriert. Da zur Kompilationszeit feststeht, welche Bindungen vorhanden sind, werden nur die Iterationen generiert, die auch wirklich benötigt werden.

Während der Iteration wird für das jeweils aktuelle Team dessen konkrete Teamklasse ermittelt und die generische Team-Referenz aus der Liste dahin gecastet. Damit ist es möglich, eine Referenz auf das Rollenobjekt durch Aufruf der passenden lift-Methode (siehe Abschnitt 4.3.3) zu erhalten. Auf dieser kann nun die gebundene Rollenmethode aufgerufen werden.

Der Rückgabewert der Aufrufe wird in einer vorher deklarierten Variablen gespeichert. Da diese vom Rückgabetyp der Basismethode sein muss, jedoch kein konkretes Objekt instanziiert werden soll, wird sie als Zeiger deklariert und mit der Adresse des (ggf. „gelowerten”) Rückgabewertes der Rollenmethodenaufrufe belegt. Schließlich wird der Zeiger für die Rückgabe dereferenziert. Nachzuvollziehen ist dies in Listing 4.5.

Nach der Abarbeitung der *before*-Methoden erfordert die Behandlung der *replace*-Methoden etwas mehr Aufwand. Nach jedem Aufruf wird überprüft, ob ein *base-call* stattfand. Dazu wird ein „versteckter” boolscher Rückgabeparameter der callin-Methoden ausgewertet, der genau dann automatisch auf `true` gesetzt wird, wenn ein base-call statt-findet. Dieser Rückgabeparameter mit dem Standardwert `false` wird vom Compiler in alle *replace*-callin-Methoden hineingeneriert. Bei Transformation des *base-calls* in den korrekten Aufruf wird zusätzlich ebendieser Rückgabeparameter auf `true` gesetzt.

Sobald der Wert nach einem Aufruf `true` ist, wird die Abarbeitung abgebrochen und das Ergebnis des letzten Aufrufes zurückgeliefert.

Anderenfalls werden alle weiteren replace-callins ausgeführt. Hat auch der letzte replace-callin die Ausführungskette nicht unterbrochen, so wird die Originalmethode aufgerufen. Schlussendlich haben eventuelle *after*-Methoden das Privileg, den Rückgabewert noch zu verändern, bevor dieser zurückgeliefert wird.

4.3.3. Lifting und Lowering

Der Lifting- und Lowering Mechanismus stellt die Verbindung zwischen Basisobjekt und Rollenobjekt her. Ein callin eines Basisobjektes verursacht einen Aufruf des „passenden" Rollenobjektes. Tritt ein Basisobjekt als Parameter eines callins in ein Team ein, so muss es zum passenden Rollenobjekt *geliftet* werden. In der anderen Richtung — bei einem callout — muss das passende Basisobjekt für ein Rollenobjekt verwendet werden, um darauf die gewünschte Basismethode aufzurufen. Ebenso müssen dabei ggf. Parameter *gelowert* werden. Konsequenterweise müssen Rückgabewerte von callin-Methoden unter Umständen gelowert und die von callout-Methoden geliftet werden.

Um zu gewährleisten, dass dabei jedes Basisobjekt für seine gesamte Lebensdauer mit dem gleichen Rollenobjekt verbunden ist, wird ein Cache verwendet, der die eindeutige Zuordnung von Basisobjekt zu Rollenobjekt bietet. Für die andere Richtung — das Lowering — ist kein Cache notwendig, da jedes Rollenobjekt selbst eine Referenz auf das zugehörige Basisobjekt verwaltet.

Generierte Methoden und Attribute. Eine Bindung legt fest, welche Klasse als Rolle für eine Basisklasse dienen soll. Anhand dieser Bindungsinformationen werden die folgenden Artefakte in das entsprechende Team generiert (angenommen, die Basisklasse trägt den Namen Basis, die Rollenklasse den Namen Rolle und das Team den Namen MeinTeam):

- ein privates Attribut `hash_map<Basis*, Rolle*> m_RolleCache`,
- eine private Methode `Rolle * liftToRolle( Basis *basis )`, deren Implementierung in Listing 4.6 ersichtlich ist,
- eine `friend class Basis;` Deklaration, damit die generierte callin-Dispatch-Methode (siehe Abschnitt 4.3.2 auf Seite 57) Zugriff auf diese private Methode hat.

Auch der gebundenen Rolle werden einige Artefakte hinzugeneriert:

- ein privates Attribut `Basis *base`, welches den im Konstruktor übergebenen Zeiger auf das Basisobjekt speichert, Listing 4.6: Implementierung der lifting- und lowering-Methoden

```
Rolle * MeinTeam::liftToRolle( Basis *basis )
{
    hash_map<Basis*, Rolle*>::iterator it = m_RolleCache.find( basis );
    if ( it == m_RolleCache.end() )// need to create a new role...
    {
        Rolle *role = new Rolle( this, basis );
        m_RolleCache[basis] = role;
        return role;
    }

    return *it; // already in cache
}

// Die generierte lowering-Methode liefert einfach das gebundene Basisobjekt zurück,
// welches im Konstruktor gesetzt wird.
Basis * Rolle :: lower()
{
    return base;
}
```

- ein privater Zeiger auf das zugehörige Team: `MeinTeam *m_team`,
- eine private `Basis * lower()`-Methode, die `base` zurückliefert,
- wiederum eine `friend class Basis;` Deklaration, damit die `lower()`-Methode aufgerufen werden kann.

Dadurch, dass die `liftToRole()`- und `lower()`-Methoden privat sind, treten keine Kovarianzprobleme bei Teamvererbung auf, da jedes instanziierbare Team in der Vererbungshierarchie unabhängige (da private) Methoden erhält.

4.4. Copy Inheritance

Rollenklassen können in zwei Dimensionen Vererbungsbeziehungen eingehen. Neben regulärem Erben von beliebigen Klassen, besteht die implizite Vererbung von Rollenklassen entlang der Team-Hierarchie (siehe Kapitel 2.2).[7] Da C++ Mehrfachvererbung — das Erben von mehr als einer Klasse auf einer Ebene — unterstützt, lag es nahe, diese für die beiden Vererbungsbeziehungen bei Rollen einzusetzen. Die Rollenklasse `Role` des Teams `TeamB` in Abbildung 4.2 würde dabei von der Klasse `List` erben sowie implizit

[7] Als dritte Dimension könnte die objektbasierte Vererbung — die Delegation von Aufrufen einer Rollenmethode an eine Basismethode — angesehen werden, was hier jedoch nicht betrachtet wird.

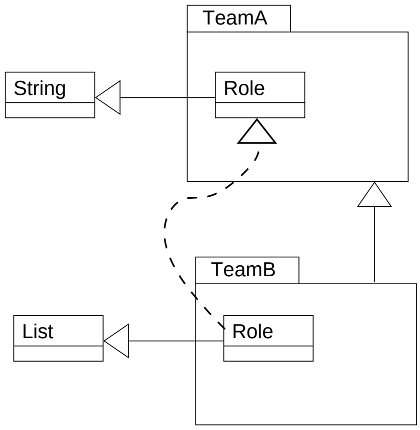

Abbildung 4.2.: Vererbungsbeziehungen von Teams und Rollen unter Verwendung der UFA-Notation [23]. Der gestrichelte Pfeil ist nicht Teil von UFA, sondern deutet nur die implizite Rollenvererbung an.

durch Teamvererbung von der Rollenklasse `Role` des Teams `TeamA`, wodurch sie weiterhin indirekt von deren Superklasse `String` erben würde.

Leider hat ein Typisierungsproblem diese Art der Realisierung jedoch verhindert. Seien die Klassen aus Abbildung 4.2 gegeben, wobei es in den beiden Rollenklassen `Role` eine Methode `doSomething( Role * )` gebe, die einen Zeiger ihrer selbst als Parameter habe. Die Methode hätte damit einen *kovarianten* Parameter, der in `TeamA` vom Typ `TeamA::Role`, in `TeamB` jedoch von `TeamB::Role` wäre.

Listing 4.7: Kovarianzproblem bei Rollenvererbung

```
void TeamA::Role::doSomething( Role *role )= 0;

void TeamB::Role::doSomething( Role *role )
{
    // just do it!
}
```

Da der C++ Standard dies leider nicht erlaubt, Kovarianz aber gerade ein ausgezeichnetes Feature von Object Teams ist, musste die Vererbungsbeziehung zwischen Rollenklassen anders gelöst werden.

Der Name der Lösung lautet: *Copy Inheritance*. Der Compiler kopiert sämtliche Attribute und Methoden von den Rollenklassen der „Super-Teams" in die Rollenklassen der abgeleiteten Teams. Damit wird die Vererbungshierarchie der Rollenklassen bezüglich Teamvererbung durch Zusammenführen der Rollenklassen auf Codeebene implementiert. Der Originalcode bleibt dabei selbstverständlich unangetastet, das Kopieren wird lediglich für die Kompilierung jedes einzelnen Teams durchgeführt.

Der Algorithmus, mit dem dies implementiert wurde, wird im Folgenden vorgestellt.

Vorgehen. Listing 4.8 stellt ein Beispiel der Vererbungsbeziehungen dar. Voraussetzung für die Anwendung von Copy-Inheritance ist die Verfügbarkeit des Quellcodes jeglicher in der Vererbungsbeziehung stehender Klassen. Ausgangspunkt ist die zu kompilierende Rollenklasse. Eine einzige neue Klasse wird erzeugt, die das Ziel aller Kopieraktionen ist und die schließlich kompiliert wird und die Ausgangsrollenklasse ersetzt.

Zuallererst wird das umschließende Team der zu kompilierenden Rollenklasse benötigt, um die Vererbungsbeziehungen festzustellen. Von diesem Team aus wird bis zur obersten Teamklasse nach Rollenklassen mit dem Namen der zu kompilierenden Rollenklasse gesucht und alle existierenden erfasst. Von der obersten Rollenklasse in der Vererbungshierarchie bis hin zur zu kompilierenden Rolle werden all diese Rollenklassen mit den folgenden Schritten abgearbeitet.

1. Alle privaten Methoden und Attribute (abgekürzt *Features*) müssen für jede Rollenklasse in der Teamhierarchie eindeutig sein. Daher werden sie beim Kopieren in die Zielklasse umbenannt und erhalten den voll qualifizierten Klassennamen als Postfix. Ebenso werden selbstverständlich alle Verwendungen dieser Features umbenannt.
   Beispiel: Die Rollenklasse `Rolle` ist Teil des Namensraumes `UnsereFirma`, gehört zum Team `UserTeam` und hat ein privates Attribute `m_data`. Dann würde das Attribute umbenannt werden in `m_data_UnserTeam_UnsereFirma`.

2. Alle öffentlichen und geschützten Attribute werden direkt kopiert.

3. Alle virtuellen Methoden werden nach bereits erläutertem Schema umbenannt, sofern eine Reimplementierung vorhanden ist. Anderenfalls wird der Name beibehalten. Verwendungen werden *nicht* umbenannt, damit die Reimplementierung aufgerufen wird. Explizite `tsuper()`-Aufrufe werden in Aufrufe der entsprechenden umbenannten Methode transformiert.

4. Alle nicht-virtuellen öffentlichen und geschützten Methoden werden ohne jegliche Änderung kopiert.

5. Behandlung abstrakter (*pure-virtual*) Methoden:

   a) Ist eine Implementierung einer pure-virtual Methode vorhanden, so ersetzt sie die pure-virtual Methode.
   b) Randfall: Existiert eine pure-virtual Methode mit einer Default-Implementierung, so wird diese wie eine „normale” virtuelle Methode umbenannt, sofern eine Reimplementierung vorhanden ist.

6. Vorsicht ist bei der Behandlung der Konstruktoren geboten:

   a) die Konstruktoren der „normalen” Oberklassen müssen aufgerufen werden,
   b) eventuelle Konstruktoraufrufe von Oberklassen, von denen nun implizit geerbt wird, müssen entfernt werden,

Listing 4.8: Copy-Inheritance – Ausgangssituation

```
class String
{
public:
    virtual std::string getValue() const { return m_string; }
private:
    std::string m_string;
};

team class TeamA
{
public:
    class Role : String, m_data( 42 )
    {
    public:
        Role() : String() { std::cout << "TeamA::Role()" << std::endl; }
        virtual int doSomething( int foo ) = 0;
    protected:
        int m_data;
    };
};

team class TeamB : public TeamA
{
public:
    class Role
    {
    public:
        Role() { std::cout << "TeamB::Role()" << std::endl; }
        virtual std::string getValue() const
        {
            std::string value = tsuper();
            std::cout << "TeamB::Role::getValue() << std::endl;
            return value;
        }
        virtual int doSomething( int foo ) {
            return i * m_data;
        }
    };
};
```

Listing 4.9: Copy-Inheritance – Das Resultat: die zusammengeführte Rollenklasse des TeamB

```
class Role
{
public:
    Role() : m_data( 42 )
    {
        std::cout << "TeamA::Role()" << std::endl;
        std::cout << "TeamB::Role()" << std::endl;
    }
    virtual std::string getValue_String_TeamA() const {
        return m_string_String_TeamA;
    }
    virtual std::string getValue() const
    {
        std::string value = getValue_String_TeamA();
        std::cout << "TeamB::Role::getValue() << std::endl;
        return value;
    }
    virtual int doSomething( int foo ) {
        return i * m_data;
    }
protected:
    int m_data;
private:
    std::string m_string_String_TeamA;
};
```

c) die Initialisierungslisten müssen zusammengefügt werden, die Member der obersten Klasse werden zuerst initialisiert,

d) der Code aus den Konstruktoren muss zusammengefügt werden, der Code aus der obersten Klasse erscheint wiederum zuerst.

7. Destruktoren müssen in exakt der umgekehrten Reihenfolge zusammengefügt werden — die der untersten Klasse zuerst. Die Destruktoren der „normal" geerbten Klassen werden automatisch danach aufgerufen.

Die Liste macht deutlich, dass die Änderung einer Rollenklasse in der Hierarchie die Rekompilation aller instanziierbaren, davon erbenden Rollenklassen erforderlich macht. instanziierbar bedeutet in diesem Fall, dass es ein vollständiges, d.h. nicht-abstraktes Team gibt, in dem sie verwendet werden.

4.5. Stand der Entwicklung

Die Implementierung von Object Teams für C++ ist noch nicht vollständig abgeschlossen. Im Folgenden werden Dinge beschrieben, die in zukünftigen Versionen umgesetzt werden sollen.

Robustheit. Der Compiler sollte sehr viel strikter sein, als er es in der aktuellen Version ist. Momentan liegt das Hauptaugenmerk darauf, die Anwendung eines Object Teams-Features zu erkennen und dieses umzusetzen. Fehlerhafte Anwendung wird derzeit kaum beachtet.

Die Robustheit könnte u.a. erhöht werden, indem explizite `setActive( false )`-Aufrufe innerhalb eines Teams abgefangen werden. Momentan kann sich ein Team selbst deaktivieren und dann z.B. einen callout ausführen. In Analogie zu `blockSignals( true )` (siehe Abschnitt 1.3.2 auf Seite 19), könnte dies zur Laufzeit abgefangen werden, durch Einfügen von `if ( !isActive() ) return;` in die callouts. Jedoch ist diese Möglichkeit nur bei rückgabelosen Methoden gegeben. Welcher Wert sollte hinter dem `return` stehen, wenn die Methode beispielsweise einen Integer zurückliefert? Qt-Signale sind generell `void`-Methoden, so dass dieses Problem dort nicht auftritt.

Ebenfalls bei callouts ist bisher die temporäre Team-*De*aktivierung für die Dauer des Basismethodenaufrufes noch nicht implementiert. Diese sollte automatisch vom Compiler in die callout-Methode eingefügt werden, um Reentranz und damit das Problem der „jumping aspects" zu vermeiden [2].

Ein weiteres vom Compiler abfangbares Problem ist explizites Aufrufen von callins. callin-Methoden dürfen nur von gebundenen Basismethoden aufgerufen werden — jegliche anderen Aufrufe sollten mit einem Compilerfehler geahndet werden.

Auch das explizite Aufrufen der `lower()`-Methoden ist verboten und sollte vom Compiler abgefangen werden. Ebenso sollte es unterbunden werden, dass Rollenklassen Zuweisungen an das `base`- oder an das `m_team`-Attribut tätigen.

Features. Unterstützung für folgende ObjectTeams-Konzepte ist bisher noch nicht implementiert:

- statische Teams,
- Konstruktor- und Destruktormerging bei der Copy-Inheritance,
- Behandlung von externalized Roles,
- Parametermappings für callins,

Des Weiteren ist das lifecycle-Management derzeit mangelhaft. Einmal erzeugte Rollenmethoden werden niemals zerstört. Ein automatisierter callin vom Destruktor des gebundenen Basisobjekts an den Destruktor des Rollenobjektes sollte bewirken, dass beide Objekte gemeinsam zerstört werden. Der Destruktor des Rollenobjektes sollte weiterhin seinen Eintrag im lift-Cache des Teams entfernen.

Die hier beschriebene Syntax von ObjectTeams/C++ sollte nicht als final, sondern bestensfalls als initiale Version aufgefasst werden. Da OpenC++ im Quellcode unter einer OpenSource Lizenz freigegeben ist, besteht die Option, OpenC++ dahingehend zu erweitern, dass noch mehr C++-Syntaxerweiterungen ermöglicht werden. Dadurch könnte die Syntax von ObjectTeams/C++ der von ObjectTeams/Java näher gebracht werden.

5. Anwendung von Object Teams für C++

Nachdem im vorigen Kapitel die Implementierung von ObjectTeams/C++ vorgestellt wurde, soll hier eine erste Anwendung beschrieben werden. Als Beispiel einer komplexen Software mit graphischer Oberfläche dient der Dateidialog des KDE-Projektes [8], an dessen Entwicklung der Autor dieser Diplomarbeit maßgeblich beteiligt ist. Dessen Funktionalität, Design und Probleme werden im Folgenden thematisiert. Im Anschluss daran wird ein neues Design mit Object Teams diskutiert.

5.1. KFileDialog Überblick

Die Klasse `KFileDialog` bietet die Funktionalität eines gewöhnlichen Dialoges zum Auswählen einer Datei, die da wäre:

- ein Fenster, das den Inhalt des aktuellen Verzeichnis darstellt
- die Visualisierung von Dateien und Verzeichnissen als Icons nebst deren Namen
- die Navigation durch Verzeichnisse per Aktivierung dieser durch Mausklick oder Tastatur
- die Selektion einer Datei
- das Sortieren der Dateien und Verzeichnisse nach verschiedenen Kriterien wie Name, Größe, Datum, Eigentümer...
- das direkte Eingeben eines Pfades/Dateinamens in ein Eingabefeld

Diese Standardfunktionalität wird ergänzt durch

- verschiedene Ansichten:
  - die bereits erwähnte Icon-basierte Ansicht
  - eine Listen-Darstellung mit Spalten für Dateieigenschaften, wie Name, Datum, Größe, ...
  - eine Vorschau, die den Inhalt der aktuellen Datei anzeigt

Abbildung 5.1.: Kombination mehrerer Ansichten im Dateidialog

- eine „Thumbnail"-Vorschau aller Dateien in einem Verzeichnis
- Selektion mehrerer Dateien
- Drag & Drop von Dateien bzw. deren Repräsentationen

Neben Dateinamenvervollständigung, Lesezeichen, Navigationshilfen, Herausfiltern von Dateien, Dateioperationen (Löschen, Umbenennen...), Netzwerktransparenz gibt es noch viele weitere Features, die hier jedoch nicht alle aufgezählt werden sollen.

Sinnvollerweise wird all die Funktionalität jedoch nicht einzig in der Klasse `KFileDialog` implementiert, sondern in einer Reihe kollaborierender Klassen.

Ziel war es, einen modularen Aufbau zu erhalten, um die einzelnen Teile auch außerhalb des Dateidialoges sinnvoll einsetzen zu können. Da nicht jede Anwendung die gleichen Anforderungen hat, sollte es möglich sein, die Klassen einzeln zu konfigurieren und z.B. durch Erben zu erweitern, aber trotzdem die gesamte Kollaboration zu erhalten. Weiterhin sollte der Benutzer zur Laufzeit zwischen den einzelnen Ansichten umschalten und diese einzeln konfigurieren können.

Die Klasse `KFileDialog` selbst wird hier eher am Rande betrachtet — sie ist nur ein mögliches „Frontend" für die dahinterliegende Funktionalität. Abbildung 5.1 zeigt den Dateidialog in einer Konfiguration mehrerer Ansichten (Vorschau sowie zwei Icon-Ansichten).

5.2. Aktuelles Modell

Nicht in Details gehend kann man die Funktionalität des Dateidialogs in zwei unabhängige Module teilen. Eine Komponente liest ein Verzeichnis ein und stellt dessen Einträge bereit. Die andere kümmert sich um die Darstellung der Verzeichniseinträge sowie der Interaktion mit dem Benutzer.

Daraus ergeben sich schon einmal die Klassen `KDirLister` sowie `KFileItem`. `KDirLister` liest Verzeichnisse auf Befehl hin ein und liefert deren Inhalt als Liste von `KFileItems`.

Da die Darstellung von `KFileItems` laut Anforderungen in unterschiedlichen Ansichten möglich sein soll, erscheint eine abstrakte Oberklasse für alle möglichen Ansichten sinnvoll. Diese Klasse namens `KFileView` spezifiziert die angeforderte Schnittstelle und implementiert einige Funktionalitäten, die allen Ansichten gemein ist.

An Klassen, die die Darstellung von `KFileItems` übernehmen, gibt es `KFileIconView`, `KFileDetailView`, `KFilePreview` sowie `KCombiView`. Erstere erlaubt die Darstellung von Icons zusammen mit dem Dateinamen. Als Icon kann auch ein Thumbnail der jeweiligen Datei verwendet werden.

Icon-Ansicht. `KFileIconView` ist in vielerlei Hinsicht konfigurierbar — neben den Icons inklusive der Icon-Größe kann die Scrollrichtung beeinflusst werden, die Anordnung der Icons, sowie der Selektionsmodus (einzelne oder mehrere Dateien selektierbar).

Listen-Ansicht. `KFileDetailView` übernimmt die listenhafte Darstellung. Sie kann beliebig viele Spalten für die Darstellung von Dateinamen, Dateigröße, Typ der Datei, etc. bereitstellen.

Vorschau-Ansicht. `KFilePreview` ist eine *compound*-View. Sie dient als Container für eine andere `KFileView`, welche dann neben einer Vorschau der gerade selektierten Datei dargestellt wird. Wie alle Views erbt sie von `KFileView`, leitet jedoch nur deren Schnittstelle an die Kind-`KFileView` weiter. Von der Kind-View wird sie über Änderungen des selektierten `KFileItems` informiert, worauf sie für dieses eine Vorschau erzeugt.

Kombinierte Ansicht. Auch `KCombiView` ist eine compound-View, welche jedoch keine weitere Funktionalität bietet, als zwei andere `KFileViews` horizontal zu kombinieren. In der linken Ansicht werden nur Verzeichnisse angezeigt, in der rechten nur Dateien, so dass eine getrennte Darstellung und Navigation gegeben ist (siehe Abbildung 5.1). Die Beziehungen zwischen `KCombiView` und `KFileView` sind in Abbildung 5.2 verdeutlicht.

Zusammensetzen der Bausteine. Was jetzt noch fehlt ist die Verbindung von `KDirLister` und den Ansichten. `KDirOperator` nennt sich die Klasse, die die Ansichten in ein Fenster einbettet und einen `KDirLister` zum Verzeichniseinlesen steuert. Dessen `KFileItems` werden vom `KDirOperator` an die Ansichten weitergeleitet, welche wiederum eigene „Items" für die Darstellung verwenden. Jedes `KFileItem`-Objekt wird in der `KFileIconView` durch ein `KFileIconViewItem` repräsentiert, in der `KFileDetailView` durch ein

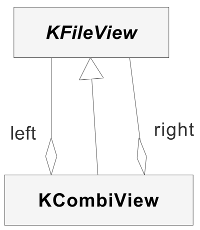

Abbildung 5.2.: Die Relationen von `KCombiView` mit `KFileView`.

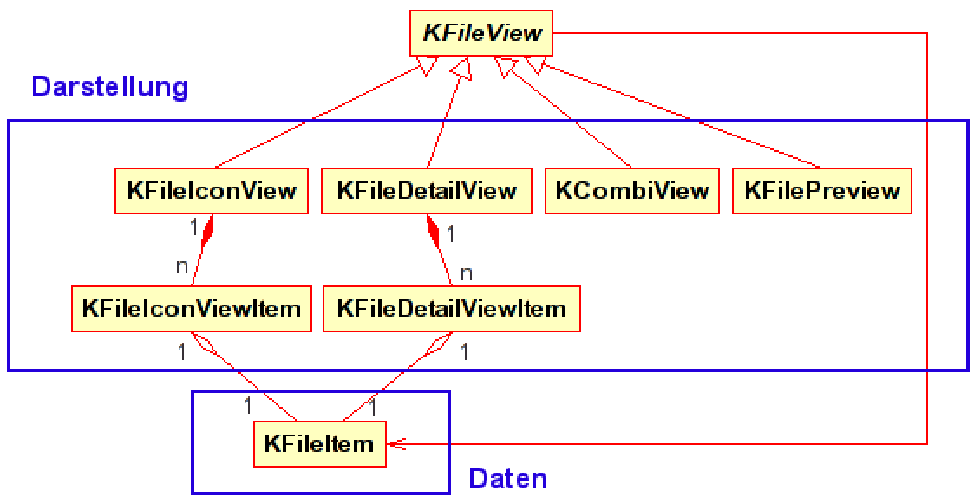

Abbildung 5.3.: KFileItems werden in den jeweiligen Views durch entsprechende View-Items repräsentiert

`KFileDetailViewItem`. Diese Relationen können anhand Abbildung nachvollzogen werden. Ein Detail am Rande sei erwähnt: da das Einlesen von Verzeichnissen relativ langsam ist — insbesondere von entfernten Verzeichnissen, z.B. eines ftp-Servers — verwaltet `KDirLister` einen Cache der `KFileItems`. Um Ressourcen zu sparen bekommen alle Interessenten eines Verzeichnislistings die selben `KFileItem`-Objekte. Dies bedeutet, dass ein `KFileItem`-Objekt gleichzeitig in verschiedenen Ansichten dargestellt werden kann.

`KDirOperator` ist *die* zentrale Klasse, die alle Einzelteile zusammenführt. Sie kapselt alle unterschiedlichen Ansichten, bietet graphische Elemente zu deren Konfiguration, zum Umschalten zwischen Ansischten sowie zum Sortieren der angezeigten Dateien nach unterschiedlichen Kriterien. Weiterhin wertet sie die Signale (siehe Abschnitt 1.3.2) der Ansichten aus und steuert damit z.B. den `KDirLister`: wenn ein Verzeichnis in der Ansicht ausgewählt wurde, wird veranlasst, dass der `KDirLister` dieses Verzeichnis einliest. Ebenso werden die Signale des `KDirLister` verarbeitet; beispielsweise wenn dieser mitteilt, dass eine Datei gelöscht wurde und daraufhin das dazugehörige `KFileItem` zerstört wird. In diesem Fall veranlasst der alarmierte KDirOperator das Entfernen des entsprechenden Eintrags in der Ansicht.

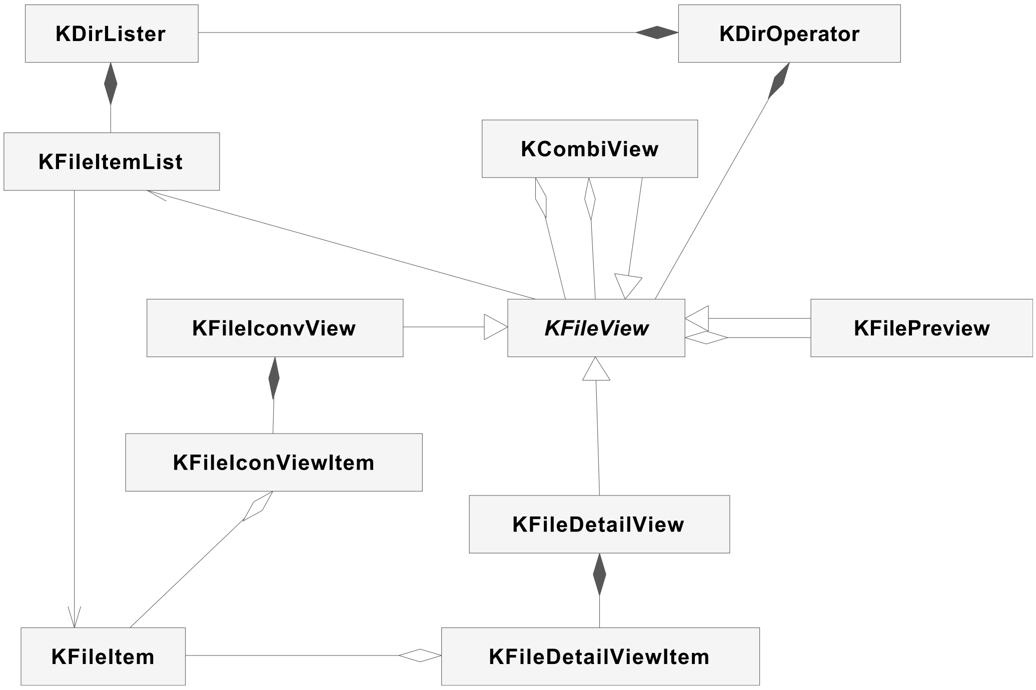

Abbildung 5.4.: `KDirOperator` fügt die einzelnen Komponenten zusammen, verdeutlicht durch die beiden Kompositionsbeziehungen.

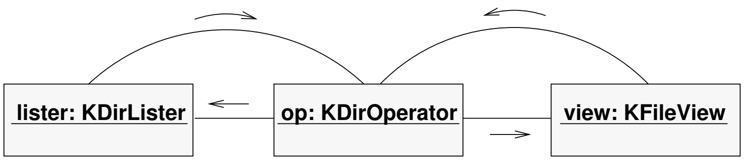

Abbildung 5.5.: Kollaboration von `KDirOperator`, `KDirLister` sowie `KFileView`

Kommunikationsfluss. Sowohl die Ansichten, als auch der `KDirLister` benötigen eine bidirektionale Kommunikation mit dem `KDirOperator`. Gesteuert werden sie durch normale Methodenaufrufe vom KDirOperator. In der Rückrichtung — Benachrichtigungen über das Eintreten von Ereignissen wie „`KFileItem` X wurde selektiert/markiert", „auf ein `KFileItem` wurde eine URL gezogen (drag&drop)", „ein Kontextmenü wurde für ein `KFileItem` angefordert" oder „neue Einträge wurden im Verzeichnis eingelesen" — wird mit Hilfe von Signalen kommuniziert. Empfänger ist der `KDirOperator`, der sie auswertet und entsprechend reagiert. Diese Kollaboration ist in Abbildung 5.5 ersichtlich, wobei die Kommunikation per Signalen durch gebogene Pfeile veranschaulicht wird.

5.3. Probleme der aktuellen Implementierung

Nachdem das aktuelle Design kurz vorgestellt wurde, werden in diesem Abschnitt einige problematische Details der `KFileDialog`-Klassen erörtert.

Objektverwaltung. Zunächst besteht das Problem der expliziten Verwaltung von Darstellungsobjekten (den jeweiligen View-Items) zu den Datenobjekten (`KFileItems`). Eine beidseitige Beziehung besteht hier. Es muss die Möglichkeit bestehen, zu einem `KFileItem` das passende View-Item einer bestimmten Ansicht abzufragen, beispielsweise beim Löschen einer Datei und dem darauf folgenden Zerstören des `KFileItems`, wie bereits im vorherigen Abschnitt beschrieben.

Ebenso muss zu einem gegebenen View-Item das dazugehörige `KFileItem` gefunden werden können, damit z.B. ein in der View-Domäne selektiertes View-Item nach außen signalisiert werden kann — nicht als View-Item, sondern als `KFileItem`.

Die manuelle Erzeugung von View-Items zu `KFileItems` für jede View sowie das Mapping sind zwar zufriedenstellend gelöst, eine allgemeinere Lösung, die nicht für jede Ansicht eine erneute Implementierung erfordert, wäre jedoch schön.

Enge Kopplung. Die Oberklasse `KFileView`, die die Schnittstelle für alle Ansichten definiert, erschwert die Implementierung der compound-Views `KCombiView` und `KFilePreview` unnötigerweise. Diese müssen die gesamte Schnittstelle implementieren, die Aufrufe jedoch nur an deren Kind-Views delegieren.

Weiterhin kommt das Problem der Binärkompatibilität hinzu. Das Hinzufügen neuer virtueller Methoden zu `KFileView`, um bestimmte Ereignisse oder Daten an die Views zu senden, ist binär inkompatibel (siehe Abschnitt 1.3.2 auf Seite 21) und erfordert daher die Rekompilation aller Klassen, die `KFileView` verwenden. Da der Dateidialog mitsamt aller vorgestellten Klassen Teil der KDE-Bibliotheken ist und damit von vielen Programmen verwendet wird, die nicht alle rekompiliert werden können/sollen, ist dies inakzeptabel.

Ein möglicher Ausweg aus diesem Dilemma wäre die Verwendung von Ereignissen zur Kommunikation. Die Erweiterung der Kommunikationsschnittstelle durch hinzufügen neuer Ereignisse ist unproblematisch. Darüber hinaus wäre es eine Erleichterung, die Ereignisse zentral an einer Stelle an die Kind-Views zu delegieren, anstatt jede einzelne virtuelle Methode.

Erweiterungsmöglichkeiten. Die Verwendung der `KDirOperator`-Kollaboration auch außerhalb des Dateidialoges macht es nötig, diese auf anwendungsspezifische Anforderungen anzupassen. Das Hinzufügen von neuen Ansichtsklassen, die entweder die existierenden erweitern oder eine vollständige Neuimplementierung sind, ist jedoch nicht ohne Weiteres möglich. Zwar kann eine Instanz einer neuen Ansichtsklasse mit der Methode

```
KDirOperator::setView( KFileView * )
```

als aktuelle Ansicht festgelegt werden, jedoch wird dieses Objekt zerstört, sobald der Benutzer die Ansicht wechselt. Es gibt jedoch keine Möglichkeit, automatisch eine erneute Erzeugung zu veranlassen. Es müsste zusätzlich eine Subklasse von `KDirOperator` erstellt werden, um die Methode

```
createView( QWidget *parent, KFile::FileView viewType)
```

zu überschreiben (`KFile::FileView` ist ein Aufzählungstyp, der die Standardviews beschreibt). Damit könnte die Erzeugung einer eingebauten Ansicht verhindert und stattdessen eine spezialisierte Ansicht integriert werden. Ferner müsste das Kontextmenü erweitert werden, wenn neue Ansichten hinzukommen.

Auch die Konfiguration einzelner Ansichten — ohne die Notwendigkeit des Erbens — ist durch die compound-Ansichten nicht gerade einfach. `KDirOperator` benachrichtigt Interessenten zwar über die Änderung der aktuellen Ansicht mittels des Signales `viewChanged( KFileView *view )`, jedoch müsste die übergebene `view` anhand ihres dynamischen Typs auf die konkrete Viewklasse gecastet werden, um daran Konfigurationen durchzuführen. Sollte der konkrete Typ eine compound-View sein, so müssten deren Kind-Views wiederum abgefragt werden — welche wie in Abbildung 5.1 sichtbar ist, auch noch geschachtelt sein können.

5.4. Einsatz von Object Teams

Unter Beibehaltung der existierenden Klassen, wurde einige neue Design erarbeitet, bei denen versucht wurde, das Rollenkonzept von Object Teams auszunutzen. Erhofft wurden dadurch eine bessere Struktur, eine bessere Erweiterbarkeit sowie eine partielle Lösung der im vorigen Abschnitt beschriebenen Probleme.

5.4.1. Bestandsanalyse und Modellierung

Zunächst einmal wurden die Fähigkeiten von Object Teams mit den Anforderungen des Dateidialogs verglichen. Mit dem Wissen um die Probleme des existierenden Designs wurde versucht, einen besseren Entwurf zu finden, der die Probleme umgeht oder Lösungen dafür bereitstellt.

Als Domänenmodell konnten eindeutig die Klasse `KFileItem` mitsamt dem Erzeuger und Verwalter ihrer Instanzen `KDirLister` identifiziert werden. Die unterschiedlichen Ansichten können als Rollenklassen des `KDirListers` aufgefasst werden; deren View-Items als Rollenklassen von `KFileItem`.

Die Modellierung von Ansichten als Rollenklassen restriktiert jedoch deren Sichtbarkeit nach außen. Anwendungsentwickler, die eine Icon-Ansicht konfigurieren wollen, z.B. zur Verwendung von 16x16 großen Icons, erhalten keine Möglichkeit dazu, da Rollenobjekte nur innerhalb des Teams sichtbar sind. Sie könnten zwar ein Subteam davon erstellen und so durch Rollenvererbung eine eigene Ansicht ins Spiel bringen, jedoch erscheint dies unnötig komplex (es wird ja gerade versucht, die Komplexität für alle Beteiligten zu verringern). Alternativ könnte die gesamte Konfigurationsschnittstelle der Ansicht in die Team-Schnittstelle dupliziert werden, so dass das Team die Konfigurationsaufrufe entgegennimmt und an das (einzige) Ansichts-Rollenobjekt delegiert.

Unter Vernachlässigung dieses nicht allzu gravierenden Problems sollen nun weitere Details der Modellierung erfasst werden. Angenommen wird dazu, dass nun jede Ansicht eine Rollenklasse eines `KDirListers` in einem entsprechenden Teams ist. Ferner kann jede Ansichtsklasse eine weitere Rollenklasse verwenden, nämlich ihre passende View-Item-Klasse, gebunden an `KFileItem`. Dies ermöglicht, mittels Teamvererbung

Spezialisierungen sämtlicher Ansichts-Rollenklassen sowie ggf. deren View-Item-Klassen zu implementieren und zu verwenden.

Dabei stellt sich die Frage, wie in diesem Fall die compound-Ansichten zu handhaben sind. Wenn auch die `KFilePreview`-Klasse eine Rollenklasse in einem dedizierten Team darstellt, dann benötigt sie eine Team-level Methode zum Konfigurieren ihrer Kind-Ansicht. Da alle Ansichten jedoch als Rollenklasse in einem Team gekapselt sind, müsste der Team-level Methode ein weiteres Team übergeben werden, wodurch *das übergebene Team als Rollenklasse* neben der `KFilePreview`-Rollenklasse bestehen würde. Diese Schachtelung von Teams als Rollenklassen erscheint ein wenig ungewöhnlich.

Als problematisch erweist sich in jedem Fall die statische Bindung der Teams an die Basis. Es ist nicht möglich, einen zentralen Dispatch-Mechnismus zu verwenden, der die callins der Basis an die aktuell konfigurierte Kombination von Ansichten delegiert. Die Teams müssen schon statisch gebunden sein und können maximal dynamisch aktiviert und deaktiviert werden. Dies erschwert bzw. verhindert insbesondere die individuelle Kombinierbarkeit durch Schachtelung von Teams.

5.4.2. Erweiterbarkeit durch callins

Unabhängig von einem neuen Entwurf stehen callins als Möglichkeit da, einzelne Klassen zu erweitern, ohne eine Subtypbeziehung einzugehen. Damit kann ein Entwickler die existierenden Ansichtsklassen erweitern, um beispielsweise eine spezielle Drag&Drop Implementierung zu integrieren. In den Ansichtsklassen existiert eine virtuelle Methode `QDragObject * dragObject()`, die standardmäßig ein URL-Dragobjekt zurückliefert. Ein Bildbetrachter könnte jedoch dieses URL-Dragobjekt beispielsweise um einen Thumbnail der Datei erweitern und somit in einer Reimplementierung dieser Methode ein ThumbnailDragObject zurückliefern.

Mittels Bindung an eine Thumbnail-Rolle, die als replace-callin die `dragObject()`-Methode ersetzt kann auf das Erben von KDirOperator und Überschreiben der createView()-Methode mit all den oben erörterten Problemen umgangen werden. Von Vorteil ist, dass die Bindung von Ansichtsklasse an die Thumbnail-Rollenklasse unabhängig davon ist, ob die Ansicht direkt angezeigt wird oder als Kind einer compound-Ansicht.

Fast noch mächtiger ist jedoch die Einmischung in den Kommunikationsfluss. Die Ereignisbehandlung, die normalerweise von den Ansichten unter Verwendung von Signalen an den `KDirOperator` delegiert wird, kann durch Bindung von callins an die Signale komplett an den Anwendungsentwickler übertragen werden. Signale können damit abgefangen und an fremder Stelle verarbeitet werden, was sonst nur bei Ereignisversendung möglich ist (vgl. Abschnitt 1.4.2 auf Seite 26).

Noch einen Schritt weiter gehend können sogar von der Anwendung aus durch callout-Bindungen Signale ausgelöst werden. Alle Ansichtsklassen verwenden zur Kommunikation mite dem `KDirLister` eine Instanz der Klasse `KFileViewSignaler`, welche dadurch ein ideales Ziel für Bindungen darstellt.

6. Zusammenfassung und Fazit

Zur Motivation der theoretischen Überlegungen bezüglich der Anpassbarkeit komplexer Software-Module mit graphischen Benutzungsschnittstellen untersucht diese Arbeit den KDE Dateidialaog. Da es sich bei dieser Software um einen Bestandteil einer Bibliothek handelt, sind erhöhte Ansprüche an die Flexibilität, Wiederverwendbarkeit sowie Erweiterbarkeit gestellt.[1] Nicht nur die Gesamtheit des Dateidialoges soll diesen Anforderungen entsprechen, sondern auch die einzelnen Bausteine sollen geeignet nutzbar sein. Dies erfordert eine angemessene Modularisierung, die es auch erlaubt, einzelne Bestandteile zu konfigurieren oder auszutauschen. Weiterhin soll die Kombinierbarkeit einzelner Module gewährleistet sein. Die Kapselung der einzelnen Funktionalitäten in unabhängige Module erfordert andererseits einen Kommunikationsmechnismus, um deren Kollaboration zu ermöglichen. Hierbei hat sich die ereignisbasierte Kommunikation aufgrund ihrer losen Kopplung der Beteiligten als geeignet herausgestellt.

Ereignisbasierte Kommunikation. Um die im KDE Dateidialog bereits einigermaßen zufriedenstellende Erfüllung obiger Kriterien weiter zu erhöhen, wurden deshalb zunächst unterschiedliche Technologien dieser Art der Kommunikation untersucht.

Die dabei deutlich gewordenen Unterscheidungskriterien dieser Benachrichtigungsmodelle können wie folgt zusammengefasst werden:

- Kardinalität der Kommunikationspartner: Broadcast (1 : n) vs. 1 : 1
- synchrone / asynchrone Kommunikation
- Kopplung der Kommunikationspartner zur Kompilationszeit bzw. zur Laufzeit (wer kennt wen?)
- Möglichkeit des a-posteriori Einhängens in die Ereignisbehandlungskette (Interception)
- Verwendung einer Ereigniswarteschlange, Ereigniskompression sowie -priorisierung unterstützend
- Erfassung von Ereignissen als Entitäten 1. Grades

[1] Im Gegensatz zu Bibliotheken sind Anwendungen in der Regel auf eine feste Domäne ausgerichtet und müssen daher weniger flexibel sein.

Die Betrachtung der unterschiedlichen Systeme Signals & Slots, Qt Events, Observer, DCOP und Object Teams callins hat ergeben, dass insbesondere bei einer Kombination von einem Ereignissystem mit der Adaptionsmöglichkeit des callin-Mechanismus von Object Teams eine hochgradige Flexibilität erreicht wird. Damit lässt sich auch in Kommunikationssysteme, die keine Interception unterstützen, solche integrieren. Gerade das Abfangen von Ereignissen zwischen existierenden Kollaborationspartnern, hat sich als sehr brauchbar zur Adaption derer erwiesen. Letztlich ist durch a-posteriori Ereignisgenerierung mit callins sogar die Adaption von bestehender Software möglich, die unzureichend oder gar keine Ereignisbenachrichtigung unterstützt.

Wichtig für eine gute Adaptierbarkeit der Ereignisbehandlung ist eine adäquate Granularität der versendeten Ereignisse. Zum Schluss soll noch ein praktischer Hinweis gegeben werden, der sich durch die Experimente mit der ereignisbasierten Kommunikation ergeben hat: Das Versenden von Ereignissen sollte wenn möglich am Ende von Methoden stattfinden, anstatt an deren Anfang oder in der Mitte. Dadurch wird sichergestellt, dass kein inkonsistenter Zustand entstehen kann. Das Versenden eines Ereignisses kann etwaige Empfänger dazu ermuntern, zustandsändernde Methoden auf dem Senderobjekt aufzurufen. Wird nach dem Versenden des Ereignisses noch Code ausgeführt, so könnte der gerade geänderte Zustand wieder rückgängig gemacht werden oder mit unwissentlich geänderten Daten Berechnungen getätigt werden.

Als überaus positives Feature von Object Teams hat sich die objektbasierte Vererbung erwiesen, die es ermöglicht, Erweiterungen an Klassen zu vollziehen, ohne dass eine reguläre Vererbungsbeziehung eingegangen werden muss (was nicht immer möglich ist). Selbst das „Überschreiben" nicht-virtueller Methoden ist damit möglich.

Das Object Teams Rollenkonzept. Das Rollenkonzept hat sich im Gegensatz zu den Object Teams callins für den beabsichtigten Einsatzzweck leider nicht geeignet. Für die hohen Anforderungen an Modularität, Erweiterbarkeit und Kombinierbarkeit hat sich das Rollenkonzept als nicht flexibel genug erwiesen.

6.1. Tools und Bugs

Für die Anfertigung dieser Arbeit wurde eine Reihe von Tools benutzt. Neben OpenC++ als Basis für den Compiler wurde LyX als Textverarbeitung eingesetzt und Umbrello zum Erstellen der UML-Diagramme. Keines der Programme ist perfekt und so wurden in allen dreien einige Bugs entdeckt. Glücklicherweise sind all diese Programme OpenSource, so daß die Probleme innerhalb kürzester Zeit gelöst werden konnten.

Ein LyX-Bug, der mir eine Leerzeile zwischen jeder Zeile eines Listings bescherte, wurde nach Problemdarstellung im Internet Relay Chat, #lyx in weniger als einer Minute behoben.

Auch Umbrello und OpenC++ haben rege und insbesondere sehr offene Entwicklergemeinden, die es mir ermöglichten, eine Reihe von Programmfehlern selbst zu beheben und in die nächste Version einfließen zu lassen. Vor allem auch die Diskussionen auf der OpenC++ Mailingliste haben sehr zum Verständnis des Toolkits beigetragen — ein herzliches Dankeschön an Grzegorz Jakacki an dieser Stelle.

Literaturverzeichnis

[1] C. Schulzki-Haddouti, “Desaster Inpol-neu: Das neue Polizei-Informationssystem: Viel zu teuer, viel zu langsam,” *c’t Magazin für Computertechnik*, vol. 24, pp. 108–116, November 2001.

[2] S. Herrmann, “Object Teams: Improving Modularity for Crosscutting Collaborations,” in *Proceedings of Net.Object Days 2002*, Technical University Berlin, 2002.

[3] *Object Teams Homepage*. http://www.objectteams.org.

[4] Y. Tao, “Component- vs. Application-Level MVC-Architecture.” 32nd ASEE/IEEE Frontiers in Education Conference, Boston, MA, November 2002.

[5] S. Burbeck, *Applications Programming in Smalltalk-80: How to Use Model-View-Controller MVC)*. 1986, 1987.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, New York, NY: Addison-Wesley Publishing Company, 1995.

[7] D. Parnas, “On the Criteria To Be Used in Decomposing Systems into Modules,” *Communications of the ACM*, vol. 15, pp. 1053–1058, December 1972.

[8] *Die Homepage des KDE Projektes*. http://www.kde.org.

[9] B. Stroustrup, *The C++ Programming Language*. Addison-Wesley Publishing Company, 3rd ed., 1997.

[10] Trolltech, AS, *Qt Whitepaper*. http://www.trolltech.com/products/qt/whitepaper/qt-whitepaper.html.

[11] M. K. Dalheimer, *Programming with Qt - Writing Portable GUI Applications on Unix and Win32*. O’Reilly, 2nd ed., March 2002.

[12] Trolltech, AS, *Signals and Slots*. http://doc.trolltech.com/3.1/signalsandslots.html.

[13] Trolltech, AS, *Why doesn’t Qt use templates for signals and slots?* http://doc.trolltech.com/3.1/templates.html.

[14] K. Sullivan and D. Notkin, "Reconciling Environment Integration and Software Evolution," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 1, pp. 229–268, July 1992.

[15] H. Herold, *Das Qt Buch.* SuSE Press, 2001.

[16] *Die CORBA Homepage der Object Management Group.* http://www.corba.org.

[17] *Java Remote Method Invocation (RMI).* http://java.sun.com/products/jdk/rmi/.

[18] *Microsoft Distributed COM (DCOM).* http://www.microsoft.com/com/tech/DCOM.asp.

[19] *DCOP Whitepaper.* http://webcvs.kde.org/cgi-bin/cvsweb.cgi/~checkout~/kdelibs/dcop/HOWTO.

[20] S. R. S. Gupta, J. M. Hartkopf, "Event Notifier, a Pattern for Event Notification," *Java Report*, vol. 3, no. 7, 1998. SIGS Publications.

[21] *DCOPRef Dokumentation.* http://developer.kde.org/documentation/library/cvs-api/classref/dcop/DCOPRef.html.

[22] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *Proceedings European Conference on Object-Oriented Programming*, vol. 1241, pp. 220–242, Berlin, Heidelberg, and New York: Springer-Verlag, 1997.

[23] S. Herrmann, "Composable designs with UFA," in *Workshop on Aspect-Oriented Modeling with UML at 1st Intl. Conference on Aspect Oriented Software Development*, 2002.

[24] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," *Lecture Notes in Computer Science*, vol. 2072, pp. 327–355, 2001.

[25] H. Ossher and P. Tarr, "Multi-Dimensional Separation of Concerns and The Hyperspace Approach," in *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, Kluwer, 2000.

[26] R. Hirschfeld, "Aspect-Oriented Programming with AspectS," in *Proceedings of Net.Object Days 2002*, DoCoMo Communications Laboratories Europe GmbH, 2002.

[27] B. de Alwis and G. Kiczales, *Apostle: Aspect Programming in Smalltalk.* http://www.cs.ubc.ca/labs/spl/projects/apostle/.

[28] A. Gal, W. Schröder-Preikschat, and O. Spinczyk, "AspectC++: Language Proposal and Prototype Implementation," in *Proceedings of OOPSLA 2001, Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 2001.

[29] O. Spinczyk, A. Gal, and W. Schröder-Preikschat, "AspectC++: An Aspect-Oriented Extension to the C++ Programming Language," in *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems*, 2002.

[30] *Die Homepage des OpenC++ Projektes.* http://opencxx.sourceforge.net/.

[31] S. Chiba, "OpenC++ Programmer's Guide for Version 2," Technical Report SPL-96-024, Xerox PARC, 1996.

[32] T. B. Dinesh, M. Haveraaen, and J. Heering, "An algebraic programming style for numerical software and its optimization," in *129*, p. 19, ISSN 1386-369X: Centrum voor Wiskunde en Informatica (CWI), 1998.

[33] S. Chiba, "A Metaobject Protocol for C++," in *Proceedings of OOPSLA '95*, vol. 30 of *ACM SIGPLAN Notices*, pp. 285–299, October 1995.

[34] R. Pawlak and L. Seinturier, "Implementation of an Event Based Run-Time Metaobject Protocol," Technical Report 9804, Laboratoire CEDRIC-CNAM, July 1998.

[35] A. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1986.

[36] W. Pree and E. Gamma, *Design Patterns for Object-Oriented Software Development.* Reading, MA: Addison-Wesley Publishing Company, 1995.

[37] *Die Klassendokumentation der OpenC++ Metaklasse Class.* http://opencxx.sourceforge.net/opencxx/html/class.html.

[38] C. Binder, "Aspectual Collaborations: Erweiterung des Java-Compilers für verbesserte Modularität durch aspekt-orientierte Techniken," Master's thesis, Technische Universität Berlin, 2002.

[39] C. Hundt, "Bytecode-Transformation zur Laufzeitunterstützung von aspektorientierter Modularisierung mit Object-Teams/Java," Master's thesis, Technische Universität Berlin, 2003.

[40] CodeSourcery, Compaq, EDG, Hewlett Packard, Intel, IBM, Red Hat and SGI, *Application Binary Interface Specification for C++ Programs.* http://www.codesourcery.com/cxx-abi/abi.html♯mangling.

A. Anhang

A.1. Credits

Schließlich möchte ich Dank aussprechen an alle, die mich bei dieser Arbeit unterstützt haben:

Stephan, Christine, Jan, Marco, Andrea und vor allen Dingen meine Eltern.

Index

Abbildungsverzeichnis

A.2. Kryptische Fehlermeldungen des gcc

Listing A.1: Komplexe Compiler-Fehlermeldungen bei Verwendung von Generizität

```
1  #include <string>
2  #include <algorithm>
3  #include <map>
4
5  using namespace std;
6
7  int main()
8  {
9      multimap<string, int> mapper;
10     multimap<string, int>::iterator it = unique( mapper.begin(), mapper.end() );
11
12     return 0;
13 }
```

Fehlermeldung bei Kompilation mit dem GNU C++ Compiler der Version 3.2. Die Version 2.95.2 produziert eine Ausgabe, die ungefähr 3 mal so lang ist!

```
gis@nevermind:~ [0] > g++ foo.cpp
/usr/include/g++/bits/stl_algo.h: In member function ‘std::pair<const
    std::string, int>& std::pair<const std::string, int>::operator=(const
    std::pair<const std::string, int>&)’:
/usr/include/g++/bits/stl_algo.h:1125:   instantiated from ‘_ForwardIter std::
        __unique_copy(_InputIter, _InputIter, _ForwardIter, std::
        forward_iterator_tag) [with _InputIter = std::_Rb_tree_iterator<std::
        pair<const std::string, int>, std::pair<const std::string, int>&, std::pair
        <const std::string, int>*>, _ForwardIter = std::_Rb_tree_iterator<std
        ::pair<const std::string, int>, std::pair<const std::string, int>&, std::
        pair<const std::string, int>*>]’
/usr/include/g++/bits/stl_algo.h:1160:   instantiated from ‘_OutputIter std::
        unique_copy(_InputIter, _InputIter, _OutputIter) [with _InputIter =
        std::_Rb_tree_iterator<std::pair<const std::string, int>, std::pair<
        const std::string, int>&, std::pair<const std::string, int>*>,
        _OutputIter = std::_Rb_tree_iterator<std::pair<const std::string, int
        >, std::pair<const std::string, int>&, std::pair<const std::string, int
        >*>]’
/usr/include/g++/bits/stl_algo.h:1273:   instantiated from ‘_ForwardIter std::unique
        (_ForwardIter, _ForwardIter) [with _ForwardIter = std::
        _Rb_tree_iterator<std::pair<const std::string, int>, std::pair<const
        std::string, int>&, std::pair<const std::string, int>*>]’
foo.cpp:10:   instantiated from here /usr/include/g++/bits/stl_algo.h:1125: non-
        static const member ‘const
```

std :: string std :: pair<**const** std::string, **int**>:: first ' , can't use **default** assignment **operator**