C++/CF – Composition Filters

Marc Lörner

Gerd.loerner@t-online.de

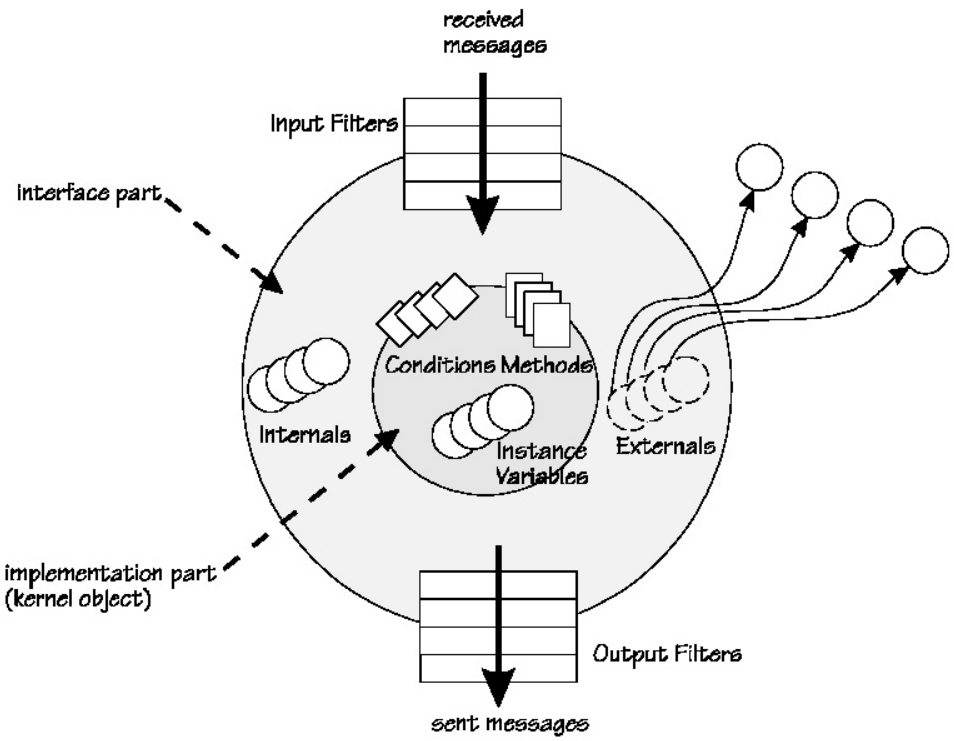
1. Das Composition-Filter Objektmodell

1.1. Motivation

Die Schwächen des objektorientierten Konzepts, was Synchronisation, Tracing und Wartbarkeit eines Systems angeht, versucht man mit verschiedensten Mitteln auszubessern, ein Lösungsansatz ist der des Composition-Filter Objektmodells, welches im folgenden theoretisch, wie auch praktisch in Form der Programmiersprache C++/CF erläutert wird.

1.2. Genereller Überblick

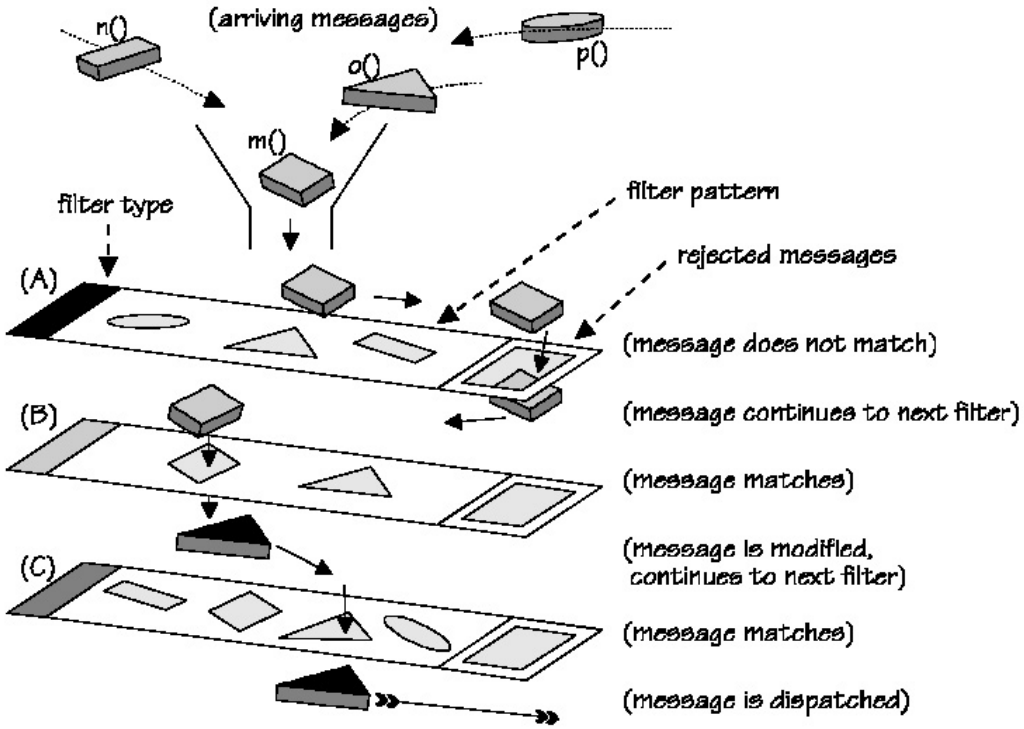
Das Objektmodell der Composition-Filter basiert auf dem objektorientierten Konzept des „Message Passings“, bei dem von einem Objekt zu einem Anderen Daten mittels Nachrichten ausgetauscht werden können. Filter greifen dabei genau beim Verschicken mittels Output-Filtern und beim Empfangen mittels Input-Filtern ein und können Nachrichten auf verschiedene Weisen manipulieren.



Im Interface-Teil eines Objekts werden Filter, Internals und Externals gebündelt, welche das herkömmliche Objektkonzept erweitern. Filter können dynamisch ausgeführt werden, mit Hilfe von Conditions. Conditions sind Methoden, die keine Parameter haben und einen bool-Wert zurückliefern, sie sollten nur zum Abfragen

des Objektstatus verwendet werden und dabei das Objekt nicht verändern. Vordefinierte Filter sind z.B.: Wait, Error, Send, Meta und Dispatch.
Filter bestehen hierbei aus Target- und Selector-Referenzen. Das Target bezeichnet das Zielobjekt der Nachricht, und der Selector eine Methode des Targets. Hierbei kann man sogenannte Wildcards verwenden, welche es dem Programmierer erlauben, z.B. alle Klassen eines Packages, oder alle Methoden einer Klasse in den Filter miteinzubeziehen. Filter können diese Targets, oder Selectors manipulieren, so dass eine Nachricht plötzlich nicht mehr zum ursprünglichen Objekt, sondern wo anders hin geleitet wird. Eine weitere Eigenschaft von Filtern ist ihre Reihenfolgegebundenheit, sie werden so abgearbeitet, wie sie der Programmierer angegeben hat.
Weiterhin bestehen CF-Objekte aus Internals und Externals, das sind Instanzen oder Referenzen von Objekten. Internals werden hier beim Instanziieren des Objekts selbstständig erzeugt und beim Zerstören des Objekts vernichtet. Externals sind nur Referenzen auf Objekte, die mit anderen Objekten geteilt werden können. Folglich bestehen diese Referenzen schon vor dem Erzeugen des Objekts und müssen auch über die Lebensdauer desselben hinaus weiter bestehen.

1.3. Arbeitsweise der Filter



Im obigen Bild kann man die Arbeitsweise verschiedener Filter gut nachvollziehen. Oben kommen zuerst die ganzen Nachrichten am Filter (A) an. (Hier Konzentrieren wir uns nur auf Nachricht m()). Diese Nachricht kann in diesem Filter nicht verarbeitet werden, also wird sie weitergeleitet zu Filter (B). In diesem findet schließlich ein „Match“ statt, d.h. der Filter erlaubt genau diese Konstellation von Selector, Target und Bedingung. Er substituiert die Target-Referenz und leitet die Nachricht an Filter (C) weiter. Filter (C) ist schließlich der letzte Filter in diesem Filter-Set, und so wird die Nachricht ins innere des Objekts geleitet, und kann verarbeitet werden.

2. C++/CF im Detail

Die C++/CF-Implementation erweitert und benutzt die Programmiersprache C++, um die Fähigkeit verschiedene Input-, oder Output-Filter benutzen zu können.

2.1. Definition eines Filters

```
aFilterIdentifier : aFilterClass =
 {aCond=>aTarget.aSelector(aParameter,…),…};
```

aFilterIdentifier:	Bezeichner, mit dem man im Programm Bezug auf den Filter nehmen kann
aFilterClass:	benutzter Filter (z.B. Dispatch, Wait, ...)
aCond:	Bedingung für die Ausführung des Filters
aTarget:	das Zielobjekt, hierbei sind Wildcards erlaubt
aSelector:	die Zielmethode, hierbei sind auch Wildcards erlaubt
aParameter:	Parameter der Zielmethode(n)

Der "=>"-Operator bewirkt in diesem Ausdruck, dass alle Methoden ausgeführt werden, die rechts dahinter angegeben wurden. Das Gegenteil dieses Operators ist der „~>"-Operator, der bewirkt, dass alle Methoden, bis auf die angegebenen ausgeführt werden.

2.2. Ableiten mit C++/CF

2.2.1. Statisches Ableiten

Das Prinzip der Vererbung kann mit C++/CF durch die Erstellung eines Internal-Objekts der Superklasse und deren Einbindung in den Dispatch-Filter realisiert werden.

Schematisches Beispiel hierzu:

```
Class Klasse
     internals
          instance : Superklasse
     methods
          ...
     inputfilters
          disp : Dispatch = {TRUE=>Superklasse.* ,
                             TRUE=>inner.*};
End;
```

2.2.2. Dynamisches Ableiten

Es besteht auch die Möglichkeit dynamisch abzuleiten, d.h. zur Laufzeit zu bestimmen, von welchem Objekt ein anderes Objekt abgeleitet ist, indem man statt TRUE eine Bedingung im Filter angibt.

```
Class Person
     internals
          atHome : Home;
```

```
          atWork : Work;
     conditions
          nineTILfive;
     inputfilters
          disp : Display = {nineTILfive=>atWork.* ,
                           TRUE=>atHome.*};
End;
```

2.2.3. Mehrfachvererbung

Bei Mehrfachvererbung muss man einfach von jeder Superklasse eine Instanz erzeugen und diese dann im Filter bekannt machen. Diese Vererbungsart kann auch wieder dynamisch erfolgen, wobei man wiederum nur statt der Condition TRUE eine Condition-Methode deklarieren muss.

```
Class Klasse
     internals
          instance  : Superklasse
          instance2 : Superklasse2
     methods
          ...
     inputfilters
          disp : Dispatch = {TRUE=>Superklasse.* ,
                             TRUE=>Superklasse2.* ,
                             TRUE=>inner.*};
End;
```

2.3. Pseudo-Variablen

Folgende Pseudo-Variablen stehen dem C++/CF-Programmierer zur Verfügung:

- Client : Referenz auf das Objekt, von dem die Nachricht stammt
- Server : Referenz auf das Objekt, das die Nachricht zuerst erhalten hat
- Outer : Referenz auf das C++/CF-Objekt selbst
- Inner : Referenz auf den C++-Teil des Objekts
- Message : Referenz auf das Nachrichten-Objekt, Funktionen können dadurch Informationen über die Nachricht erhalten

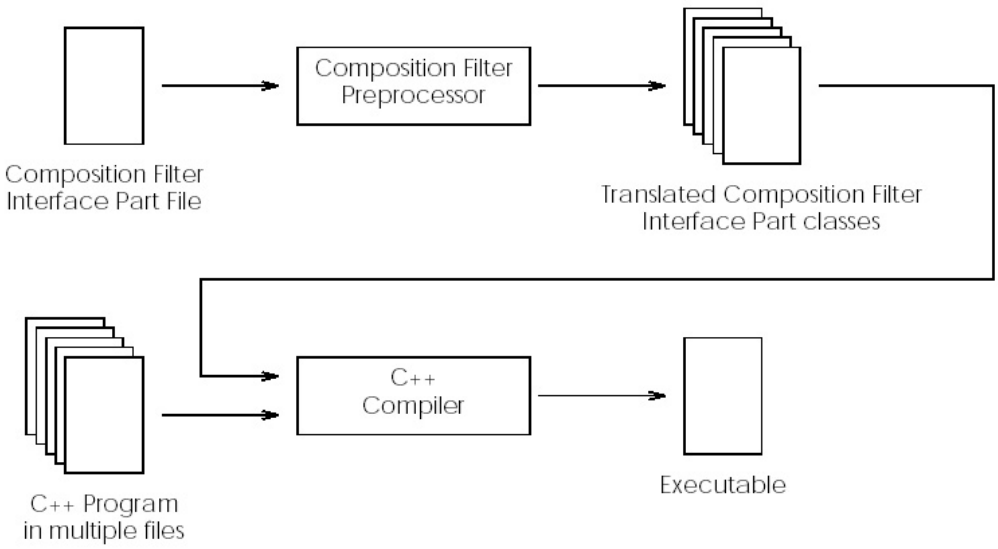
2.4. Aufteilung von C++/CF-Programmen und Übersetzungsvorgang

C++/CF-Programme lassen sich in drei Dateiarten gliedern, zuerst die CF-Datei, welche die komplette Definition der Composition-Filter beinhaltet. Dann hat man noch die C-Datei, welche die C++-Implementation einer jeden Klasse beinhaltet, und zum Schluss hat man noch die Headerdatei mit der Endung .h.

Der Übersetzungsvorgang startet damit, dass man die erstellten CF-Dateien durch den eigens dafür entwickelten Präprozessor schickt, welche diese dann vorübersetzt in eine vom C++-Compiler verständliche Syntax. Die Header- und C++-Dateien

lassen sich nun gemeinsam mit den übersetzten CF-Dateien zu einem funktionierenden Programm übersetzen.
Die Vorteile an diesem Verfahren sind:

- Möglichkeit einen bestehenden Compiler zu verwenden und die damit verbundene Aufwandsreduzierung
- man kann mit geeigneter Implementierung dafür sorgen, dass das Programm mit und ohne Composition-Filter funktioniert



2.5. Performanz Betrachtung C++ - C++/CF

Generell lässt sich sagen, dass C++/CF-Programme langsamer sind, was damit zusammenhängt, dass für jede Nachricht erst einmal die komplette Filterschlange für den Input, sowie den Output, abgearbeitet werden muss. Weiterhin müssen mehrere Objekte erzeugt werden, damit ein C++/CF-Objekt richtig funktioniert, eins für jeden verwendeten Filter, der Objektmanager, sowie der CF- und der C++-Schicht. Der Speicherplatzbedarf kann hierbei natürlich auch nicht vernachlässigt werden, da jedes Objekt eine gewisse Menge an Speicherplatz braucht. Zusätzliche Kosten verursachen hierbei noch die Nachrichten, welche auch einen Teil des Speichers belegen.

Literatur:

- Doktorarbeit „Extending C++ using the concepts of composition filters“ von M.II.J. Glandrup: http://trese.cs.utwente.nl/publications/msc_theses/glandrup.thesis.pdf
- “The Composition-Filters Object Model” von Lodewijk M.J. Bergmans: http://trese.cs.utwente.nl/publications/reports/cf.pdf