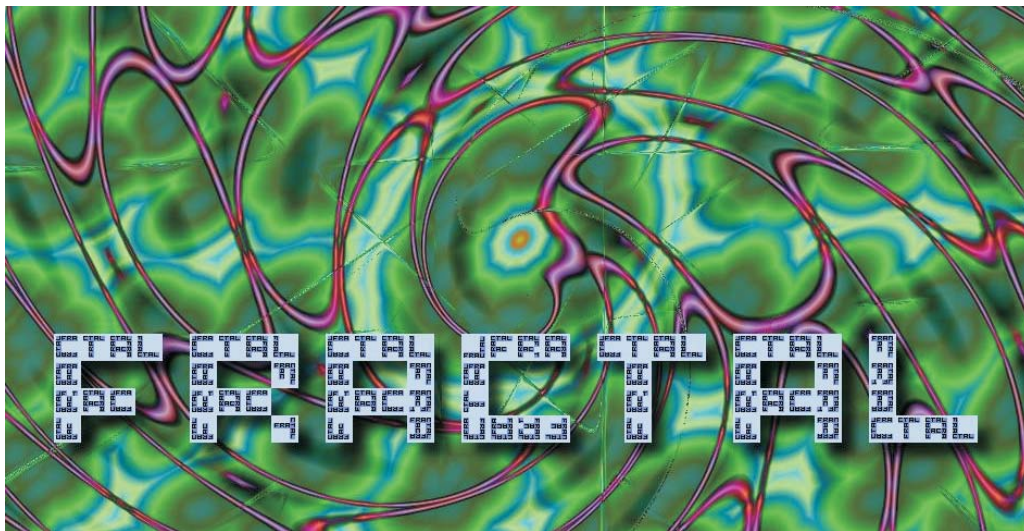


Universität Passau  
Fakultät für Mathematik und Informatik  
Lehrstuhl für Programmierung (Prof. Christian Lengauer)

## Das Komponentenmodell Fractal



Hauptseminar Moderne Programmiermethoden

Betreuer: Dipl. Inf. Armin Größlinger

von  
Christoph Zengler  
Matrikelnummer: 41712  
SS 2006

## Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>2</b>
<b>2</b>	<b>Motivation und Überblick</b>	<b>3</b>
2.1	Entwurf . . . . .	3
2.1.1	Finden der Komponenten und Definieren der Gesamtstruktur . . . . .	3
2.1.2	Definieren der Komponentenverträge . . . . .	4
2.2	Implementierung . . . . .	5
2.2.1	Implementierung der Interfaces . . . . .	5
2.2.2	Implementierung der Komponenten . . . . .	5
2.3	Konfiguration . . . . .	6
2.3.1	ADL basierte Konfiguration . . . . .	6
2.3.2	GUI basierte Konfiguration . . . . .	7
2.4	Rekonfiguration . . . . .	7
2.4.1	Lifecycle Management . . . . .	7
2.4.2	Introspektion . . . . .	8
<b>3</b>	<b>Die Spezifikation</b>	<b>9</b>
3.1	Gesamtkonzept . . . . .	9
3.1.1	Externe Struktur einer Komponente . . . . .	10
3.2	Instantierung von Komponenten . . . . .	10
3.2.1	Factories . . . . .	11
3.2.2	Templates . . . . .	11
3.2.3	Bootstrap . . . . .	12
3.3	Typsistem . . . . .	12
3.3.1	Zusicherungsart und Kardinalität . . . . .	12
3.3.2	Typsistem . . . . .	13
3.3.3	Subtyping . . . . .	13
<b>4</b>	<b>Implementierungen</b>	<b>13</b>
4.1	Julia . . . . .	14
4.1.1	Grundkonzept . . . . .	14
4.1.2	Mixin Klassen . . . . .	15
4.1.3	Optimierungen . . . . .	15
4.2	AOKell . . . . .	16
4.3	Think . . . . .	17
<b>5</b>	<b>Bewertung</b>	<b>18</b>
<b>A</b>	<b>Sourcecode des WebServer</b>	<b>19</b>
<b>B</b>	<b>ADL des WebServers</b>	<b>22</b>

---

# 1 Einleitung

Das Komponentenmodell *Fractal*<sup>1</sup> ist ein modulares und erweiterbares Komponentenmodell, für das es Implementierungen in verschiedenen Programmiersprachen (C, C++, SmallTalk, Java) gibt. Es ist ange-dacht damit Anwendungen wie Betriebssysteme, Middleware Plattformen oder graphische Oberflächen zu entwerfen, zu implementieren, einzusetzen und zu warten. Die Entwickler wollen damit erreichen, dass die Kosten für Entwurf, Entwicklung und Wartung von Software reduziert werden können.

Fractal ist nur eines von vielen Projekten des *ObjectWeb Konsortiums*<sup>2</sup>, das 2002 in Frankreich als Zusammenarbeit von Bull/Evidian, INRIA und der France Telecom R&D gegründet wurde. In diesem Konsortium befinden sich Unternehmen, die im Bereich der Middleware forschen und entwickeln. Unter Middleware versteht man eine Softwareschicht in verteilten Systemen, die zwischen Betriebssystem und den Anwendungen auf jeder Seite des Systems liegt. Middleware arbeitet auf einer sehr hohen Ebene im Schichtenmodell. Typische Einsatzbereiche von Middleware sind z.B. der Austausch komplexer Daten, Funktionsaufrufe zwischen Komponenten (Remote Procedure Calls) oder Sicherstellen der Transakti-onssicherheit von ansonsten unabhängigen Teilsystemen. Das ObjectWeb Konsortium bezeichnet als die Hauptfunktionen der Middleware, dass Komplexität verborgen wird, so z.B. die lokale Verteilung einzelner verbundener Komponenten einer Anwendung. Ebenso soll von der Heterogenität verschiedener Hardwa-rekomponenten, Betriebssysteme oder Kommunikationsprotokollen abstrahiert werden. Außerdem sollen den Anwendungsentwicklern einheitliche high-level Schnittstellen zur Verfügung gestellt werden, so dass Anwendungen leichter erstellt, portiert und zur Zusammenarbeit gebracht werden können.

Das Fractal Komponentenmodell basiert sehr stark auf dem *separation of concerns* Entwurfsmuster (*SoC Pattern*). Die Idee hinter diesem Pattern ist, die einzelnen *Concerns* einer Applikation in verschiedene Code- oder Laufzeitkomponenten aufzuspalten. Ein Concern definiert sich dabei recht frei: Es ist ein bestimmter Interessenschwerpunkt in einem Programm. Dies kann z.B. ein Stück Source-Code sein, oder auch ein Entwurfsmuster, ein UML Diagramm oder eine bestimmte Funktion sein. Doch gibt es immer wieder so genannte nicht-funktionale Anforderungen an Software, wie z.B. Synchronisierung, Konfigurati-on oder Logging, deren Code über viele einzelne Module eines Softwareprojekts verteilt ist. Außerdem ist somit auch ein einzelnes Modul (z.B. Klasse) für mehrere Dinge zuständig (z.B. Berechnungen ausfüh-ren und auch loggen). Daher sinken Verständlichkeit, Wiederverwendbarkeit und Anpassbarkeit des Codes.

Die Idee der Fractal-Entwickler ist es nun, diese einzelnen Aufgaben alle aufzuspalten in einzelne Kompo-nenten, bei denen jede nur noch für einen bestimmten Concern zuständig ist. Um dies zu erreichen benutzt das Fractal Komponentenmodell drei Spezialisierungen des SoC Patterns: *Trennung von Interface und Implementierung*, *komponentenorientierte Programmierung* und *Umkehrung der Kontrolle (Inversion of Control, IoC)*. Das erste Entwurfsmuster ist auch bekannt als so genanntes Bridge Pattern, dessen Einhal-tung sicherstellt, dass die Schnittstelle einer Komponente und ihre konkrete (z.B. plattform-spezifische) Implementation getrennt sind. Unter komponentenorientierter Programmierung versteht man, dass größe-re Concerns in kleinere trennbare Teile aufgespalten werden, deren Implementierung man als Komponente bezeichnet. Das letzte Pattern sorgt für die Trennung von funktionalen und nicht-funktionalen Concerns: Fractal Komponenten müssen zum Beispiel nicht nach ihrer Konfiguration und ihren Ressourcen suchen, sondern bekommen sie von außen zugewiesen.

Das SoC Pattern wird auch auf eine einzelne Komponente angewendet, die immer aus zwei Teilen be-steht: einem Inhalt (*content*) für die funktionale Seite und einem *Controller*, der die die nicht-funktionalen Concerns (Introspektion, Konfiguration, Sicherheit, Logging) verwaltet. Der Content besteht wiederum aus Fractal Komponenten, d.h. Fractal Komponenten können verschachtelt werden, können aber auch von mehreren Komponenten verwendet werden (*shared*).

Um einen besseren Überblick zu bekommen, wird im ersten Teil der Arbeit die Funktionsweise von Fractal

---

<sup>1</sup>fractal.objectweb.org

<sup>2</sup>www.objectweb.org

an einem kleinen Beispiel illustriert, bevor im zweiten Teil genauer auf die Spezifikation des Komponentenmodells eingegangen wird. Der dritte Teil beschäftigt sich dann mit konkreten Implementierungen dieses Modells. Abschließend folgt eine kurze Bewertung. ([1],[5])

## 2 Motivation und Überblick

### 2.1 Entwurf

Am Anfang eines Projektes mit Fractal steht immer der Entwurf des Systems mit einzelnen Komponenten. Dieser Entwurf ist zuerst einmal völlig unabhängig von der nachfolgenden komponentenorientierten Programmierung. Man kann natürlich auch alle Komponenten in ein großes Stück Code schreiben, was jedoch nicht sehr sinnvoll wäre, da damit wieder viele Vorteile der Komponentenorientierung verloren gehen. Wir betrachten in diesem Abschnitt das konkrete Beispiel eines minimalistischen WebServers (der nur die GET-Methode unterstützt).(siehe [1])

Wir folgen dabei dem ganz klassischen Aufbau eines Webservers: Der Server akzeptiert eingehende Verbindungen eines Clients auf einem bestimmten Socket und startet für jede eingehende Verbindung einen Thread, der diese Anfrage bearbeitet. In diesem Thread findet die Verarbeitung in zwei Schritten statt: Zuerst wird die Anfrage geparkt und auf Richtigkeit überprüft und dann wird im positiven Fall die angeforderte Seite an den Client zurückgeschickt. Der komplette Sourcecode des WebServers ist im Anhang A zu finden.

In einer komponentenbasierten Anwendung unterscheidet man prinzipiell zwischen zwei verschiedenen Arten von Komponenten: zum einen dynamische Komponenten, welche dynamisch erzeugt und zerstört werden können und zum anderen statische Komponenten, deren Lebenszeit an die des Programms gebunden ist. Bei dynamischen Komponenten spricht man von Daten (*datas*) und bei statischen Komponenten von Diensten (*services*).

#### 2.1.1 Finden der Komponenten und Definieren der Gesamtstruktur

Am Anfang des Entwurfs beginnt man am besten mit der Identifikation der Dienste. Im Falle unseres WebServers können wir leicht die zwei Hauptdienste identifizieren: Ein Dienst zum Empfangen der Anfragen (*request receiver*) und ein anderer Dienst zum Verarbeiten und Ausführen der Anfrage (*request processor*). Der *request receiver* benötigt einen weiteren Dienst, eine Thread-Fabrik, also eine Möglichkeit für jede einkommende Anfrage einen neuen Thread zur Bearbeitung zu erstellen. Dafür benötigt er also einen Scheduler-Dienst, der auf verschiedene Art implementiert sein kann (sequentiell, multi-threaded, parallel).

Der *request processor* benutzt seinerseits zwei Dienste: den *request analyzer*, um die Anfrage zu parsen, und den *Logger* um die Anfrage mitzuprotokollieren. Um die Anfrage schließlich auszuführen und die gewünschte Seite an den Client zurückzuschicken benötigt der *request processor* noch zwei Dienste zum Öffnen der gewünschten Datei oder zum Zurückschicken einer Fehlermeldung falls die Datei nicht gefunden wurde. Diese beiden letzten Dienste kann man zusammenfassen zu einem *request dispatcher*, also einem Dienst, der die Anfrage der Reihe nach an verschiedene Abteilungen (*request handler*) weiterleitet, bis eine die Anfrage bearbeiten kann. (In unserem Fall wird also zuerst versucht die Datei zu öffnen und an den Client zu schicken (*file request handler*), klappt dies nicht, so wird sie weitergegeben um eine Fehlermeldung zu schicken (*error request handler*)).

Dynamische Komponenten müssen nicht zwangsweise gesucht werden, denn da es sich dabei immer um Datenstrukturen handelt, ist es kein Nachteil, diese gleich als Objekte (in einer objektorientierten Programmiersprache) zu implementieren. Denn bei Daten oder Datentypen machen Features wie dynamische Rekonfiguration oder Introspektion oft keinen Sinn. So könnte man bei unserem WebServer z.B. Sockets, HTTP Anfragen, Dateien, Streams oder sogar Threads als Daten bezeichnen, doch wir wollen diese nicht konkret als dynamische Komponenten implementieren.

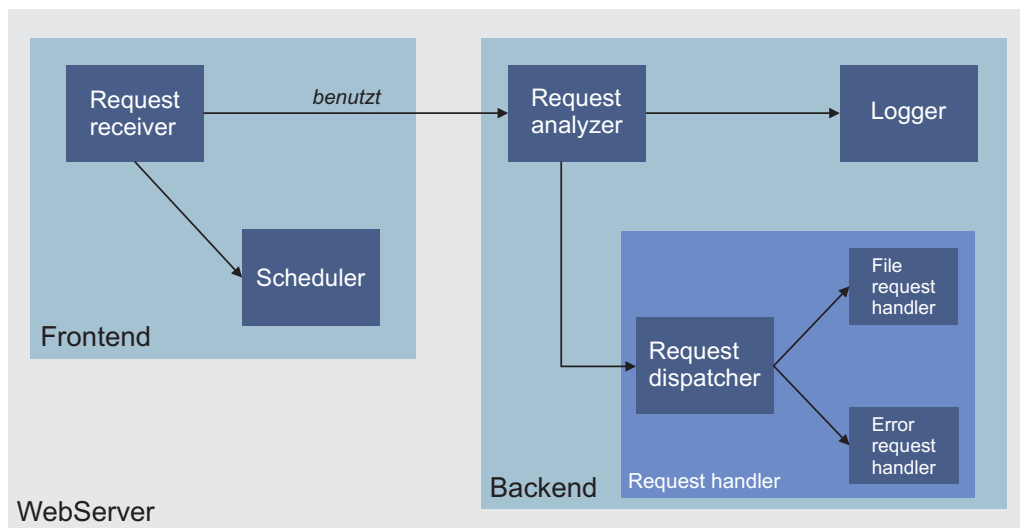


Abbildung 1: Gesamtarchitektur des Webservers

Hat man nun also alle Dienste und Daten identifiziert so müssen diese noch auf Komponenten verteilt werden. Solange zwei Dienste (oder Daten) nicht sehr stark miteinander verbunden sind, macht es keinen Sinn sie zu einer Komponente zusammenzufassen, da man dann wieder Probleme beim einfachen Austausch von Komponenten hat. Wird z.B. bei der Komponente für den *request analyzer* auch gleich ein *Logger* miteingefügt, so müsste das - sollte man diesen Analyzer einmal austauschen wollen - ein neuer Analyzer auch tun. Man sieht also, dass es sinnvoller ist solche strikt trennbaren *concerns* auch wirklich in eigene Komponenten zu programmieren. Man sollte also im Normalfall eine Komponente pro Service vorsehen. Bei uns haben wir also sieben Komponenten: *request receiver*, *request analyzer*, *request dispatcher*, *file request handler*, *error request handler*, *scheduler* und *logger*.

Nachdem man diese Komponenten gefunden hat, ist es nicht mehr schwer ihre Abhängigkeiten festzustellen und sie in einer Hierarchie anzuordnen. Aus der obigen Beschreibung der Komponenten kann man das Schema in Abbildung 1 erstellen.

### 2.1.2 Definieren der Komponentenverträge

Nach der erfolgreichen Identifikation und Hierarchisierung der Komponenten bleibt nun noch eins zu tun: Es müssen Verträge zwischen den einzelnen Komponenten geschlossen werden. D.h. was leistet eine bestimmte Komponente und was will sie als Gegenleistung dafür. Der *file request handler* will zum Beispiel die Datei, die er zurückgeben soll, ist sie vorhanden, gibt er sie auch zurück.

Die Verträge müssen dabei besonders bedacht erstellt werden. Man muss bedenken, dass es im Nachhinein im Allgemeinen mehr Aufwand erfordert einen Vertrag zu ändern (und somit alle betroffenen Komponenten) als eine einzelne Komponente, die einen bestimmten Vertrag erfüllt. Außerdem sollte man das SoC Konzept nicht aus dem Auge verlieren. Verträge beziehen sich ausschließlich auf die funktionale Seite der Komponenten. Nicht-funktionale Aufgaben wie die Konfiguration von Komponenten haben in einem Vertrag nichts zu suchen.

Im Fall unseres Webservers beschränken wir uns auf die notwendigen Verträge, die wie folgt wären:

- Der *Logger* Dienst stellt eine `log` Methode mit einem `String` Parameter zur Verfügung
- Die *Scheduler* Komponente enthält eine `schedule` Methode mit einem `Runnable` Parameter (Dieses `Runnable` wird nach Aufruf dieser Methode gestartet, möglicherweise auch in einem anderen Thread)
- Der *request analyzer*, der *request dispatcher* und der *file* und *error handler* stellen alle eine `handleRequest` Methode bereit, die als Argument einen `Request` bekommt.

Dieses `Request` beinhaltet folgende Informationen: den Socket, den Eingabe- und Ausgabestream dieses Sockets und die angeforderte URL. In jedem Dienst wird diese `handleRequest` Methode anders implementiert:

- Der *request analyzer* generiert den Eingabe- und Ausgabestream und parst den Eingabestream um die angeforderte URL zu erhalten
- Der *request dispatcher* leitet jeden Request der Reihe nach weiter an seine assoziierten *request handler* bis einer die Anfrage erfolgreich bearbeiten kann
- Der *file request handler* versucht die angeforderte Datei zu öffnen und sie an der Empfänger zurückzuschicken
- Der *error request handler* schickt eine Fehlermeldung an den Client zurück und bearbeitet den Request immer erfolgreich.

## 2.2 Implementierung

Wie bereits erwähnt müssen die Komponenten, die im Entwurf erstellt wurden, nicht unbedingt auch so als Komponenten implementiert werden. Es kann aus Performancegründen durchaus sinnvoll sein bestimmte Komponenten im Code zusammenzufassen (z.B. bei unserem WebServer den *request dispatcher* und seine beiden assoziierten *request handlers*). Sollte man sich jedoch nicht sicher sein, bei welchen Komponenten eine solche Zusammenfassung sinnvoll ist, so soll man lieber jede Komponente als eigenes Objekt implementieren.

### 2.2.1 Implementierung der Interfaces

Gemäß dem Bridge Pattern muss bei der Implementierung darauf geachtet werden, dass, bevor die eigentlichen Komponenten implementiert werden, ihre Interfaces programmiert werden. Dies gewährleistet die strikte Trennung von Interface und Implementation. Dadurch fällt es hinterher leichter eine spezielle Implementationen einer Komponente zu ändern.

Die Implementation der einzelnen Interfaces in unserem Beispiel fällt leicht, da die ganze Arbeit bereits im Entwurf mit den Verträgen getätigt wurde. Dementsprechend sehen unsere Interfaces z.B. wie folgt aus:

```
public interface Logger { void log(String msg); }  
public interface Scheduler { void schedule (Runnable task); }  
public interface RequestHandler { void handleRequest (Request r)  
    throws IOException; }
```

Wie man sieht werden *requests* als Instanz der Datenklasse `Request` dargestellt. Man kann also konkrete Klassen in Interfaces verwenden, was jedoch nicht empfohlen wird. Auch für Datenstrukturen sollte man Interfaces benutzen, um auch ihre Implementation leicht ändern zu können, oder sie als Komponente implementieren zu können.

### 2.2.2 Implementierung der Komponenten

Nachdem alle Interfaces implementiert wurden können nun die Komponenten programmiert werden. Alle Komponenten, die keinerlei Abhängigkeiten zu anderen Komponenten haben, können wie gewöhnliche Javaklassen programmiert werden. So zum Beispiel die Implementation des Loggers:

```
public class BasicLogger implements Logger {  
    public void log (String msg) {  
        System.out.println(msg);  
    }  
}
```

Komponenten, die Abhängigkeiten zu anderen Komponenten besitzen, müssen jedoch auf spezielle Art und Weise implementiert werden. Betrachten wir als Beispiel den *request receiver*, dieser benötigt zum einen einen *Scheduler*, zum anderen einen *request analyzer*. Es ist klar, dass man nicht - wie in klassischer objektorientierter Programmierung - diese beiden Komponenten einfach als statische Felder des *request receivers* implementieren kann. Dadurch geht die Modularität verloren, d.h. eine Änderung des Schedulertyps (single-, multithreaded) kann nur durch eine Änderung am Sourcecode bewerkstelligt werden.

Wendet man die IoC Pattern an, so würde man den Konstruktor der Klasse *RequestReceiver* so abändern, dass dieser den gewünschten *Scheduler* und *request analyzer* als Parameter bekommt. Dies erlaubt eine gewisse Modularität, jedoch nicht zur Laufzeit. D.h. ist ein solcher *request receiver* über seinen Konstruktor einmal konkret instantiiert worden, so besteht zur Laufzeit keine Möglichkeit mehr seinen verwendeten *Scheduler* zum Beispiel zu verändern.

Da Fractal die dynamische Rekonfigurierbarkeit der einzelnen Komponenten unterstützen will, muss das IoC Konzept noch verbessert werden. Dafür gibt es in Fractal das Konzept der *Bindings*. Komponenten mit Abhängigkeiten müssen das Interface *BindingController* implementieren, welches die Methoden *listFc*, *lookupFc*, *bindFc* und *unbindFc* zur Verfügung stellt. Dabei werden die Methoden so implementiert, dass die *listFc* Funktion die Namen der Abhängigkeiten einer Komponente zurückgibt. Mit der Methode *lookupFc* kann man das zugehörige *Binding* auslesen (d.h. es gibt den Namen der Komponente an, die dieser Abhängigkeit zugewiesen ist. Mit *bindFc* und *unbindFc* kann man diese Komponenten zuweisen und entfernen.

Die Klasse *RequestReceiver* implementiert jetzt also zwei Interfaces: *BindingController* und *Runnable*. Hieran kann man sehr gut das dritte Konzept, nämlich die Trennung von *content* und *controller* sehen: Die Methoden des *BindingController* Interfaces gehören zum *controller*, während die funktionale Seite (*content*) durch die *run* Methode des *Runnable* Interfaces repräsentiert wird. Des Weiteren gibt es noch das Interface *AttributeController*, welches implementiert werden muss, wenn man Attribute einer Komponente verändern will, z.B. den Text oder die Farbe eines Buttons ändern will.

## 2.3 Konfiguration

Nachdem nun alle Komponenten implementiert wurden stellt sich eine Frage: Wer konfiguriert die ganzen *Bindings*, d.h. wer sagt den einzelnen Komponenten mit wem sie genau assoziiert sind, mit wem sie kommunizieren müssen. Der einfachste Ansatz wäre es, dies direkt zu implementieren: Eine Hauptfunktion setzt sämtliche *Binding* manuell, z.B.: `requestAnalyzer.bindFc("1", basiclogger)` - sage dem *request receiver*, dass er zum loggen die konkrete Komponente *basiclogger* verwenden soll. Diese Methode hat jedoch einige Nachteile: Zum einen verliert man bei größeren Systemen schnell den Überblick und vergisst einzelne *Bindings* zuzuweisen. Zum anderen geht die Gesamtarchitektur des Systems verloren, bzw. ist nicht mehr ersichtlich. Und am wichtigsten: Dieser Ansatz vermischt zwei unterschiedliche Konzepte, nämlich die Architekturbeschreibung und die Entwicklung, was heißen soll, es ist nicht möglich eine gegebene Architektur auf mehrere Arten zu implementieren ohne dass man Code neu schreiben muss.

### 2.3.1 ADL basierte Konfiguration

Um letzteres Problem zu umgehen wurde eine eigene Sprache zur Beschreibung der Architektur eingeführt, die *ADL (Architecture Description Language)*. Diese auf XML basierende Sprache beschreibt ausschließlich die Architektur, d.h. es entstehen keine Vermischungen mehr mit der Implementation. Diese *ADL* ist streng getypt, und der *ADL Parser* hat einige grundlegende Verifikationsmöglichkeiten zur Fehlervermeidung. Der *ADL Code* des *WebServers* ist im Anhang B zu finden.

Der erste Schritt um eine Komponenten Architektur zu beschreiben ist es, die Typen der Komponenten zu definieren. Hierfür muss man spezifizieren, was die Komponente zur Verfügung stellt (z.B. ein Interface), und was sie von anderen Komponenten benötigt. Schaut man sich z.B. den Typ des *file* und

*error request handler* an, so erkennt man, dass diese beiden Komponenten ein `RequestHandler` Interface zur Verfügung stellen, und keine Abhängigkeiten haben. Solche Typdefinitionen können auch vererbt werden, d.h. ein Typ kann von einem übergestelltem Typ die zur Verfügung gestellten Interfaces und die benötigten Abhängigkeiten erben.

Nach den Komponententypen können die Komponenten selber definiert werden. Hierbei unterscheidet die ADL von Fractal zwischen primitiven Komponenten, die nur aus Ihren zugehörigen Java Klassen bestehen, und zusammengesetzten Komponenten, die mehrere Subkomponenten und Abhängigkeiten zwischen diesen enthalten können.

Ist die Architekturbeschreibung vollständig kann sie entweder kompiliert werden, was in einer Java Klasse resultiert, oder sie kann direkt interpretiert werden. In beiden Fällen vollzieht der ADL Parser zuvor eine Verifikation, die verhindern soll, dass Bindings übersehen wurden, oder falsch gesetzt wurden.

### 2.3.2 GUI basierte Konfiguration

Der Nachteil der bei der ADL Beschreibung noch bleibt ist, dass man immer noch keine rechte Vorstellung hat, wie die Architektur genau aussieht. Um auch diesen Nachteil noch zu beseitigen gibt es ein Tool namens *Fractal GUI*, in dem man das Modell visuell bearbeiten kann und als ADL speichert.

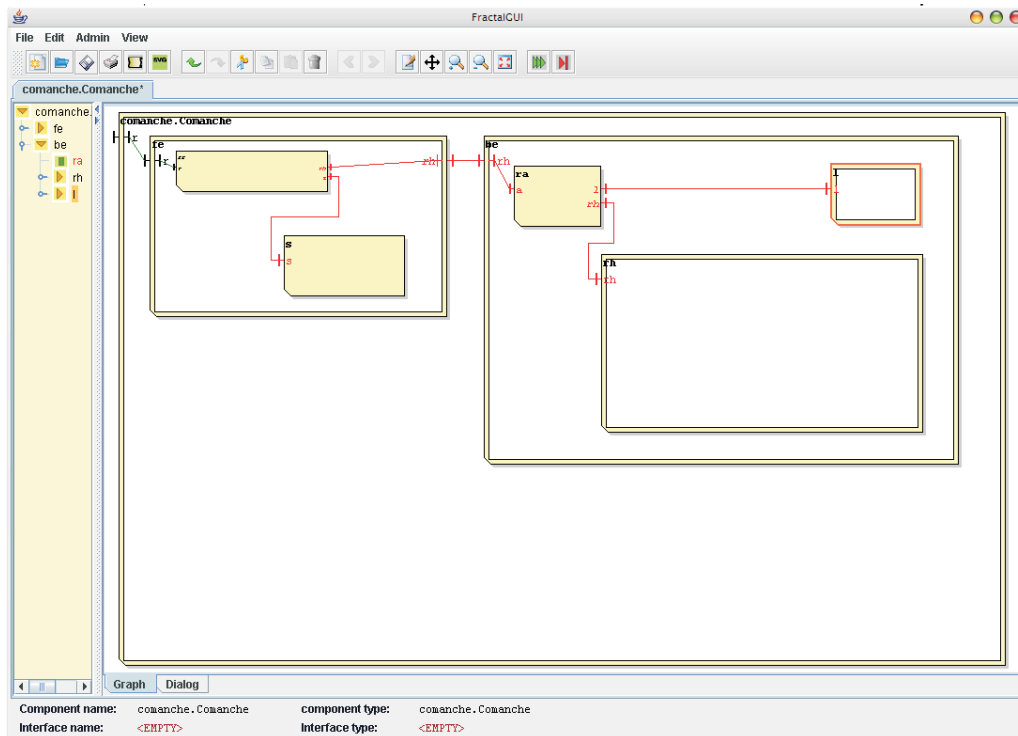


Abbildung 2: Fractal GUI

## 2.4 Rekonfiguration

In diesem Abschnitt soll gezeigt werden, wie man Fractal Komponenten dynamisch rekonfigurieren kann.

### 2.4.1 Lifecycle Management

Der einfachste Weg eine Anwendung zu rekonfigurieren ist sicherlich, die Anwendung zu stoppen, ihre ADL zu ändern, und neu zu starten. Dies ist jedoch nicht immer möglich, so gibt es z.B. Dienste die nicht unterbrochen werden dürfen. Dies erfordert eine Rekonfigurationsmöglichkeit zur Laufzeit. Doch hier gibt es eine Gefahren: Wenn die Synchronisation nicht genau abgestimmt ist, kann es sein dass der Zustand

der Applikation inkonsistent wird, oder sie einfach abstürzt.

Ein Ansatz für diese Synchronisation ist es, nur den (hoffentlich kleinen) Teil der Applikation in Ruhezustand zu setzen, der gerade rekonfiguriert werden soll. In Fractal nennt man dies Lebenszyklus Management (*life cycle management*).

Stellen wir uns ein einfaches Beispiel vor: Wir möchten in unserem WebServer den *file request handler* austauschen, unsere neue Version soll nicht nur Dateien sondern auch Verzeichnisse unterstützen. Nehmen wir an, im *request dispatcher* sei der zugehörige *request handler* in dem Feld `h0` gespeichert. Dann sieht unser Code wie folgt aus:

```
requestDispatcher.unbindFc("h0");
requestDispatcher.bindFc("h0", newSuperFileHandler);
```

Der *request dispatcher* verwaltet seine *request handler* intern in einer `TreeMap`. Dies ist eine Datenstruktur die keinen konkurrierenden Zugriff erlaubt, d.h. der obige Code kann zu einem Fehlverhalten der Anwendung führen. Unsere Vorgehensweise muss also wie folgt aussehen: 1) Setze den *request dispatcher* kurzfristig in den Ruhezustand 2) vollziehe obige Änderungen 3) reaktiviere den *request dispatcher*.

Um dieses Starten und Stoppen einer Komponente einfach zu gewährleisten gibt es ein weiteres Interface: `LifeCycleController`. Dieses Interface stellt die Methoden `startFc` und `stopFc` zum starten und stoppen einer Komponente, sowie `getFcState` zum Überprüfen des aktuellen Zustandes zur Verfügung. Implementiert unser *request dispatcher* dieses Interface ist ein einfaches Anhalten und Wiederstarten der Komponente möglich.

Die Implementation dieses Interfaces bringt jedoch einen gewissen Overhead mit sich, der vielleicht gar nicht nötig ist, wenn wir bestimmte Komponenten nicht dynamisch rekonfigurieren wollen. Daher ist es besser das Lebenszyklus Management komplett vom funktionalen Teil zu trennen. Dies kann durch einen *Interceptor* geschehen, den man - dank Trennung von Interface und Implementation möglich - zwischen *request analyzer* und dem ungeänderten *request dispatcher* schaltet. Dieser *Interceptor* kann ebenfalls das `BindingController` Interface implementieren und delegieren. Dann kann man diesen *Interceptor* praktisch als *request dispatcher* mit Lebenszyklus Management betrachten, der einen *request dispatcher* ohne Lebenszyklus Management wie in Abbildung 3 kapselt. Hier sehen wir die Vorteile der Trennung

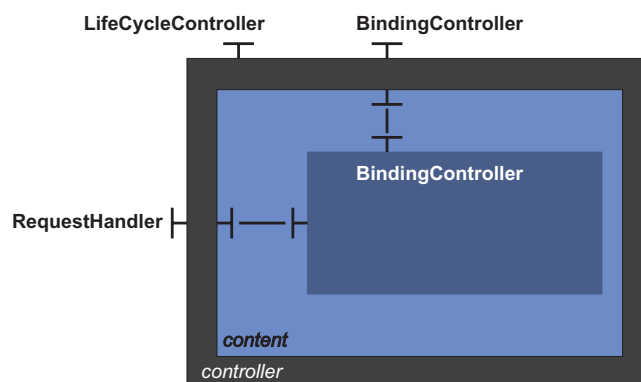


Abbildung 3: Kapselung einer Komponente

von Architekturbeschreibung und Instantierung: Obwohl die Beschreibung für die Gesamtarchitektur nur einmal getätigt wurde, kann sie nun mit oder ohne *automatisch generierten* kapselnden Komponenten wie z.B. den *Interceptors* instantiert werden.

## 2.4.2 Introspektion

Will man eine Anwendung dynamisch rekonfigurieren, so braucht man zu erst eine Referenz auf die Komponente, die rekonfiguriert werden soll. Im obigen Beispiel brauchten wir also eine Referenz auf den *request dispatcher* um ihn zu rekonfigurieren. Wie kommt man an eine solche Referenz? Die naive Methode ist

bei einer bekannten Komponente (z.B. der Hauptanwendung) zu starten und sich über die `lookupFc` Methode „durchzunavigieren“. Dies ist aber nur bei kleinen Applikationen möglich. Bei großen Anwendungen mit hunderten Komponenten kann dieses Verfahren schnell ausufern, da man sich durch sehr viele Komponenten „hangeln“ muss. Man braucht also einen alternativen Lösungsansatz. Ein solcher Ansatz ist es, eine komplette Beschreibung der Komponenten Architektur - inklusive der zusammengesetzten Komponenten - zu erstellen und während der Laufzeit zu aktualisieren. Dies nennt man dann die *Introspektion* der Architektur.

Um diese Introspektion in Fractal zu gewährleisten braucht es zwei Dinge: Erstens müssen neben den primitiven Komponenten auch die zusammengesetzten Komponenten (wie in der ADL beschrieben) instantiiert werden und zweitens müssen alle Komponenten die nötigen Interfaces implementieren um Informationen über sich selbst zur Verfügung stellen zu können. Eines dieser Interfaces ist das schon bekannte `BindingController`, aber auch zwei neue Interfaces, namens `Component` und `ContentController`. Ersteres wird benötigt um die Interfaces, die eine Komponente implementiert, nach außen sichtbar zu machen, zweites um die Subkomponenten einer zusammengesetzten Komponenten zu erfahren.

### 3 Die Spezifikation

Im obigen Beispielablauf der Entwicklung einer Fractal Anwendung sind schon einige Konzepte klar geworden. Diese sollen nun erst noch einmal in einem großen Kontext zusammengefasst werden. Dann wird auf einige Aspekte der Fractal Spezifikation (siehe [2]) eingegangen, die im Beispiel noch nicht vorkamen. Die komplette Fractal Spezifikation kann auf der *Fractal/Objectweb* Homepage eingesehen werden.

Bestehende komponentenorientierte Frameworks (z.B.: das Container Konzept in Java Beans, CORBA Component Model (CCM) oder Microsoft .Net) und ADLs bieten nur mangelnde Unterstützung für Erweiterungen und Adaptionen. Das bringt einige Nachteile mit sich: Zum einen fällt die Erweiterung der Kontrollmöglichkeiten für Komponenten (wie die nicht-funktionalen Aspekte) schwerer oder ist sogar nicht möglich. Zum anderen haben Entwickler und Programmierer oft keine Möglichkeit einen Tradeoff zwischen dem Grad der Konfigurierbarkeit und der Performance bzw. Speicherverbrauch einzugehen. Als letztes sei auch noch erwähnt, dass ein Einsatz solcher Systeme in verschiedenen Umgebungen - z.B. einem eingebetteten System - zu Schwierigkeiten führen kann.

Fractal umschiffet diese Problematik indem es dem Entwickler und Programmierer freie Hand bei der Gestaltung der Kontrollmöglichkeiten einzelner Komponenten lässt. Wie wir gesehen haben ist es z.B. nicht nötig, für jede Komponente ein Lifecycle Management zu implementieren. Mit anderen Worten heißt das: Komponenten in Fractal sind reflektiv, und der Grad der Reflektion kann vom Entwickler selbst bestimmt werden. Unter Reflektion versteht man in der Programmierung, dass ein Programm weitergehende Informationen über seine eigene Struktur oder Aufgabe hat, die normalerweise bei der Kompilation verloren gehen würden. Eine Java Klasse „weiß“ z.B. bei normaler Programmierung nach der Kompilation oder Interpretation nicht mehr, welche Interfaces sie implementiert. Diese Unwissenheit kann bei Fractal - wie wir bereits gesehen haben - durch Implementierung einzelner Interfaces zur Introspektion umgangen werden.

#### 3.1 Gesamtkonzept

Das Hauptziel des Fractal Komponentenmodells ist es, den Entwicklern und Programmierern das Entwerfen, Implementieren, Überwachen und dynamische Rekonfigurieren von komplexen Softwareprodukten so einfach wie möglich zu gestalten. Dies spiegelt sich in den Strukturen des Fractal Modells wieder:

- *zusammengesetzte Komponenten (composite components)* für eine einheitliche Sicht auf die Applikation von verschiedenen Abstraktionsebenen aus
  - *geteilte Komponenten (shared components)* um Ressourcen zu modellieren
  - *Möglichkeiten zur Introspektion* um ein laufendes System zu überwachen
-

- *Konfigurations und Rekonfigurationsmöglichkeiten* um eine Anwendung auch dynamisch rekonfigurieren zu können

Jedoch sollte das Fractal Modell auch auf viele Softwarebereiche anwendbar sein, also von Servern hin zu eingebetteten Systemen. Die obigen Vorteile von Fractal bringen jedoch leider auch immer einen gewissen Overhead an Kosten mit. Dies kann auf kleineren oder eingebetteten Systemen zu Problemen mit den Ressourcen führen. Um dies zu verhindern ist das Fractal Komponentenmodell keine große, unabänderbare Spezifikation, die von allen Fractal Komponenten erfüllt werden muss, sondern vielmehr eine Ansammlung zusammenhängender Konzepte, die einzelne Komponenten implementieren *können*, jedoch nicht *müssen*. So kann man Komponenten nach ihrem Kontrolllevel, also ihren reflektiven Möglichkeiten, sortieren.

Auf unterster Kontrollebene ist eine Fractal Komponente eine Laufzeiteinheit, die keinerlei Kontrollmöglichkeiten zur Interaktion mit anderen Komponenten besitzt. Ein klassisches Objekt in der objektorientierten Programmierung ist eine solche Komponente ohne Kontrollmöglichkeiten. Auf diese Weise kann man auch vorhandene, objektorientierte Software in das Komponentenmodell einbinden. Eine solche Komponente heißt dann Basiskomponente.

In der nächsten Kontrollebene - externe Introspektion genannt - implementiert eine Fractal Komponente ein Interface, das erlaubt, sämtliche externen Interfaces einer Komponente, also seine Anbindung zur Umwelt, zu erfahren.

Das nächste Kontrolllevel ist die Konfigurationsebene. Hier implementieren die Komponenten Interfaces, die die Veränderung des Inhalts einer Komponente erlauben. Der Inhalt von Komponenten sind wiederum Subkomponenten und ihre Bindings, also ihre Abhängigkeiten. Eine Komponente kann also verschiedene Interfaces implementieren, die die Veränderung dieser Subkomponenten und/oder ihrer Abhängigkeiten erlauben.

Diese drei Kontrollebenen - keinerlei Kontrolle, externe Introspektion und Konfiguration - haben wir in Kapitel 2 bereits behandelt. Der Logger z.B. ist eine Komponente der ersten Kontrollebene. Das Interface `Component` stellt Methoden zur Verfügung die für die externe Introspektion benötigt werden, mit den Interfaces `BindingController`, `LifeCycleController` oder `ContentController` stehen Schnittstellen zur Konfiguration zur Verfügung. All diese Interfaces können von Komponenten implementiert werden. Die Fractal Spezifikation sieht für diese Kontrollebenen verschiedene *conformance levels* vor, beginnend bei level 0 (einfache Java Objekte oder Java Beans) bis zu level 3.3 (volle Introspektion und Rekonfiguration auf *content* und *controller* Ebene).([2])

Neben diesen Kontrollmöglichkeiten stellt das Fractal Komponentenmodell auch ein Framework für die Instantierung von Komponenten und ein einfaches Typsystem zur Verfügung. Ebenso wie die obigen Schnittstellen sind auch diese beiden Konzepte optional. Im folgenden wird etwas genauer auf die Instantierung von Komponenten und das Typsystem eingegangen, da diese zwei Konzepte im WebServer Beispiel noch nicht erläutert wurden. Zuvor wollen wir jedoch noch kurz einen etwas genaueren Blick auf die externe Struktur einer Komponente werfen.

### 3.1.1 Externe Struktur einer Komponente

Je nach der Kontrollebene in dem sich eine Komponente befindet kann man sie als *black* oder *white* Box bezeichnen. Bei einer black box ist die innere Struktur nicht sichtbar, es gibt nur einige von außen sichtbare Schnittstellen (*external interfaces*) zur Komponente. Der Logger oder der Scheduler sind solche Beispiele, da sie keinerlei Interfaces zur Introspektion implementieren. Jedes Interface in einer Komponente braucht einen eindeutigen Namen, um es von den anderen Interfaces unterscheiden zu können. Man unterscheidet zwischen zwei Arten von Interfaces: dem *client* oder *notwendigem Interface* und dem *server*. Der client sendet Anfragen für Operationen oder Methoden, die ein server entgegen nimmt.

## 3.2 Instantierung von Komponenten

Bisher haben wir gesehen wie man mit bestehenden Komponenten umgeht, wie man sie konfigurieren und benutzen kann. Nun wollen wir betrachten, wie man solche Komponenten erzeugen kann. Fractal benutzt

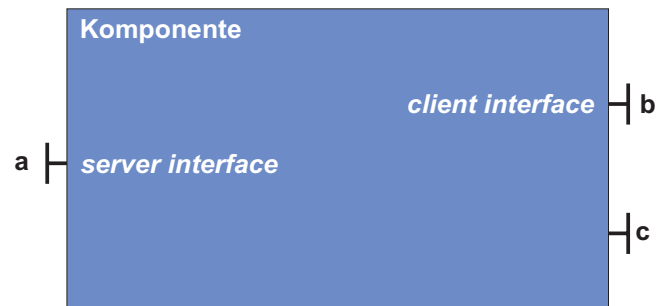


Abbildung 4: Externe Sicht auf eine Komponente

dazu das so genannte *Factory Pattern*, d.h. es gibt Komponenten die als „Fabrik“ dienen und die neue Komponenten erzeugen können.

### 3.2.1 Factories

Fractal unterscheidet grundlegend zwischen zwei verschiedenen Arten von Factories: Die generischen Factories (*generic component factories*), die verschiedene Arten von Komponenten erzeugen können, und den Standard Factories (*standard component factories*) für die Erzeugung einer bestimmten Komponententart - alle vom selben Typ. Zwei funktionale Interfaces *GenericFactory* und *Factory* stehen hierzu zur Verfügung.

Das *GenericFactory* Interface stellt eine einzige Methode zur Verfügung, *newFcInstance*. Diese Methode erstellt eine neue Komponente und gibt ihr *Component* Interface zurück. Als Parameter zur Beschreibung der neuen Komponente bekommt sie den Typ der neuen Komponente, sowie eine Beschreibung des *controllers* und des *contents* der neuen Komponente. Im Gegensatz dazu hat das Interface *Factory* vier Methoden: die erste, *newFcInstance* erstellt analog zu obiger Methode eine neue Komponente, nimmt jedoch keine Parameter, da eine Standard Factory nur einen Typ von Komponente instantiieren kann. Um zu erfahren, welche Komponententart eine Factory produziert, gibt es die drei Methoden *getFcInstanceType*, *getFcControllerDesc* und *getFcContentDesc*, die Auskunft darüber geben.

Für beide Interfaces gilt: Der Typ der Komponente darf nur die funktionalen Interfaces der jeweiligen Komponente beschreiben, die Interfaces zur Kontrolle der Komponenten (z.B. das *LifeCycleManagement* Interface) müssen in der jeweiligen Beschreibung des *controllers* spezifiziert werden.

Anzumerken bleibt noch, dass beide Factories nicht unbedingt jedes Mal eine neue Instanz einer Komponente erzeugen müssen. Gemäß dem *Singleton Pattern* können sie auch jedes Mal die selbe Instanz einer Komponente zurückgeben. Außerdem werden die Komponenten einer Factory im selben Adressraum erzeugt, in dem sich auch die Factory Komponente befindet.

### 3.2.2 Templates

Ein *Template* ist eine spezielle Komponente, die eine Komponente erzeugt, die praktisch zu ihr isomorph ist. D.h. die Komponenten, die von einer Template Komponente erzeugt werden, müssen die selben funktionalen Client und Server Interfaces (außer dem *Factory* Interface selber) wie die Template Komponente besitzen, können aber beliebige Kontroll Interfaces haben. Ebenso haben die Komponenten die von einer Template Komponente erzeugt wurden die selben Attribute. Hat eine Template Komponente weitere durch Bindings verbundene Templates als Subkomponenten, so haben auch die daraus resultierenden Komponenten so viele Subkomponenten mit den selben Bindings. Wenn einige dieser Subtemplates geteilt sind, so sind es auch die daraus resultierenden Subkomponenten.

Templatekomponenten machen nur in einem Fall Sinn, nämlich wenn aus einer textuellen Beschreibung (z.B. ADL) mehrere identische Komponenten erstellt werden müssen. Hier kann es mehr Sinn machen die Beschreibung nur einmal zu parsen und ein entsprechendes Template zu erstellen, als bei jeder Instantierung einer solchen Komponente die Beschreibung neu zu parsen. In allen anderen Fällen ist die

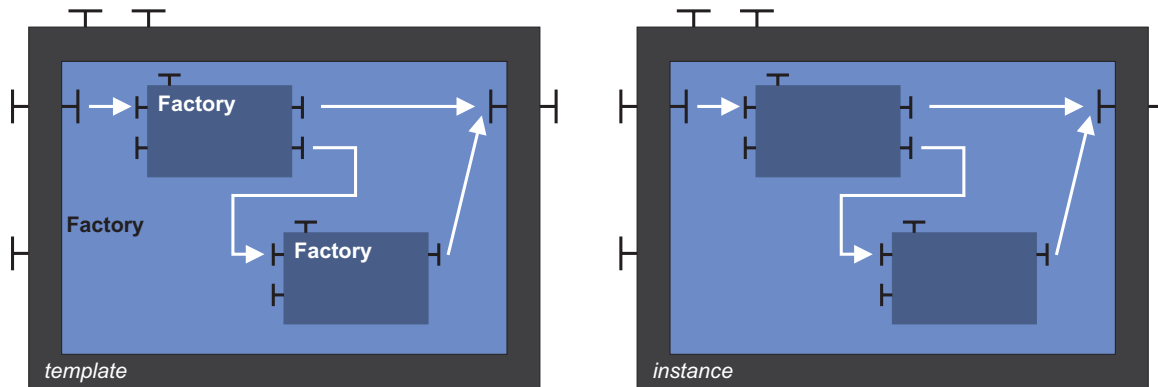


Abbildung 5: Template und Instanz des Templates

Verwendung von Templates zwar möglich, aber nicht effizient.

### 3.2.3 Bootstrap

Nun haben wir gesehen wie Komponenten durch eine Factory erzeugt werden können. Doch wie werden die Factories erzeugt? Sie können wieder durch Factories erzeugt werden, doch das gäbe eine unendliche Rekursion. Also gibt es eine sogenannte *Bootstrap Factory*, also eine erste Factory, die nicht erzeugt werden muss. Dies ist verständlicherweise eine generische Factory und kann so verschiedene Arten von Komponenten, darunter dann auch andere Factories, erzeugen.

## 3.3 Typsystem

### 3.3.1 Zusicherungsart und Kardinalität

Die *Zusicherungsart* (*contingency*) eines Interfaces gibt Auskunft darüber, ob die Funktionalität, die durch dieses Interface zur Verfügung gestellt wird, während der Laufzeit einer Komponente garantiert wird, oder nicht. Man unterscheidet zwischen *notwendigen* (*mandatory*) und optionalen Interfaces. Die Methoden eines notwendigen Interfaces stehen zur Laufzeit einer Komponente *garantiert* zur Verfügung. Ein solche Semantik ist notwendig für ein Server Interface. Ein solches Client Interface, das selber keine Funktionalität hat, *muss* an ein notwendiges Interface gebunden sein. Eine Komponente mit notwendigem Client Interface kann nicht gestartet werden, bis all diese Interfaces an andere notwendige Interfaces gebunden wurden.

Die Methoden eines optionalen Interfaces stehen *nicht garantiert* zur Verfügung. Das kann z.B. ein Server Interface sein, dessen zugehöriges interne Interface nicht an eine Subkomponente gebunden ist, oder ein Client Interface dessen Interface gar nicht gebunden ist.

Die *Kardinalität* eines Interfaces gibt an wie viele Interfaces eines bestimmten Typs eine Komponente besitzen kann. Bei einer *singleton* Kardinalität ist dies genau ein Interface. Bei der *collection* Kardinalität dürfen die Komponenten beliebig viele Interfaces dieses Typs haben, wobei alle Interfaces des selben Typs einen Namen mit dem selben Präfix haben müssen. Da es also eine unbegrenzte Anzahl solcher Interfaces geben kann, werden diese nicht alle sofort erstellt. Hier erfolgt eine *lazy* Auswertung. D.h. erst wenn ein Interface konkret gebunden wird (mit `bindFc`), wird es auch - falls es noch nicht existiert - erstellt. Gelöscht wird es automatisch, wenn es keinerlei Verbindungen mehr mit diesem Interface gibt.

Notwendige und optionale Interfaces sind sehr nützlich wenn man Komponenten hat, die auf ein Zusammenspiel mit anderen Komponenten vertrauen, die aber auch noch mit anderen Komponenten zusammenarbeiten können. Eine Parser Komponente zum Beispiel benötigt notwendigerweise eine Lexer Komponente, kann jedoch mit oder ohne Logger Komponente arbeiten. Die *collection* Kardinalität ist sinnvoll, wenn man sich zum Beispiel eine Menü Komponente vorstellt. Diese kann beliebig viele Untermenü Komponenten und Menüpunkte haben.

### 3.3.2 Typsystem

Bei dem von Fractal verwendeten Typsystem ist ein Komponententyp nur eine Menge von Interfacetypen. Ein Interfacetyp besteht aus seinem Namen (der gleich dem Namen seiner Komponenten Interfaces ist), einer Signatur, einer Rolle (client oder server), der Zusicherungsart und der Kardinalität. Für den Komponententyp steht das Interface `ComponentType` zur Verfügung, welches die Methode `getFcInterfaceType` zur Verfügung stellt. Diese Methode gibt ein Array aller Interfacetypen dieser Komponente zurück. Desweiteren gibt es das Interface `InterfacyType` welches sich aus obigen Daten zusammensetzt. Komponententypen und Interfacetypen können durch Typ Factories erzeugt werden, die das Interface `TypeFactory` implementieren. Diese Typ Factories werden wiederum durch die Bootstrap Factory instantiiert.

### 3.3.3 Subtyping

Zwischen einzelnen Typen von Komponenten kann auch eine Subtyp Beziehung bestehen, diese Beziehung ist hinreichend (aber nicht notwendig), dass folgende Substituierbarkeitsregel gilt:

*Wenn ein Komponententyp  $T_1$  ein Subtyp von  $T_2$  ist, dann kann eine Komponente vom Typ  $T_2$  in jeder Umgebung durch eine Komponente vom Typ  $T_1$  ersetzt werden, wobei andere Komponenten und ihre Bindings nicht verändert werden.*

Dabei ist ein Interfacetyp  $I_1$  Subtyp eines *Server Interfaces* des Typs  $I_2$ , wenn folgendes gilt:

- $I_1$  hat den selben Namen und die selbe Rolle wie  $I_2$
- Ist die Zusicherungsart von  $I_2$  notwendig, muss auch die Zusicherungsart von  $I_1$  notwendig sein
- Ist die Kardinalität von  $I_2$  eine collection, so muss auch die von  $I_1$  eine collection sein

Ein Interfacetyp  $I_1$  ist Subtyp eines *Client Interfaces* vom Typ  $I_2$ , wenn gilt:

- $I_1$  hat den selben Namen und die selbe Rolle wie  $I_2$
- Ist die Zusicherungsart von  $I_2$  optional, muss auch die Zusicherungsart von  $I_1$  optional sein
- Ist die Kardinalität von  $I_2$  eine collection, so muss auch die von  $I_1$  eine collection sein.

Eine Komponententyp  $T_1$  ist schließlich Sybtyp eines Komponententyps  $T_2$ , genau dann wenn jeder Client Interfacetyp von  $T_1$  ein Subtyp eines Interfacetyps in  $T_2$  ist, und jeder Server Interfacetyp von  $T_2$  ein Supertyp eines Interfacetyps in  $T_1$  ist.

## 4 Implementierungen

Wie bereits erwähnt, existieren vom Fractal Komponentenmodell verschiedene Implementierungen für verschiedene Programmiersprachen. Wir wollen im Folgenden drei Implementierungen etwas näher betrachten: Julia<sup>3</sup> und AOKell<sup>4</sup>, zwei Java Implementierungen, sowie Think<sup>5</sup>, eine C Implementierung. Desweiteren existieren noch Implementierungen für .Net (*FractNet*)<sup>6</sup>, Smalltalk (*FractTalk*)<sup>7</sup>, C++ (*Plasma*)<sup>8</sup> und eine Implementierung für verteilte Systeme (*ProActive*)<sup>9</sup>

---

<sup>3</sup>[fractal.objectweb.org/tutorials/julia](http://fractal.objectweb.org/tutorials/julia)

<sup>4</sup>[fractal.objectweb.org/tutorials/aokell](http://fractal.objectweb.org/tutorials/aokell)

<sup>5</sup>[think.objectweb.org](http://think.objectweb.org)

<sup>6</sup>[www-adele.imag.fr/fractnet/](http://www-adele.imag.fr/fractnet/)

<sup>7</sup>[csl.ensm-douai.fr/FracTalk](http://csl.ensm-douai.fr/FracTalk)

<sup>8</sup>[www.inria.fr/rapportsactivite/RA2004/sardes/uid43.html](http://www.inria.fr/rapportsactivite/RA2004/sardes/uid43.html)

<sup>9</sup>[www-sop.inria.fr/oasis/ProActive/](http://www-sop.inria.fr/oasis/ProActive/)

## 4.1 Julia

Julia ist eine Java Implementierung des Fractal Komponentenmodells der France Telecom. Die Entwickler haben sich dabei vier Ziele gesetzt: Ihr Hauptziel ist es, dem Programmierer ein Framework zum einfachen Entwickeln von Kontrollelementen zur Verfügung zu stellen. Diese Kontrollelemente übernehmen dann den *controller*-Part einer Komponente. Das zweite Ziel ist, diese Kontrollelemente möglichst flexibel kombinieren zu können. D.h. der Programmierer muss zum einen statische, aber dafür effiziente Komponenten, zum anderen aber auch nicht so effiziente Komponenten mit einem hohen Grad an Rekonfigurierbarkeit generieren können. Auch verschiedene Optimierungsstufen zwischen diesen beiden Extremen sollen leicht programmiert werden können. Das dritte Ziel ist es, dass diese Kontrolleinheiten so effizient wie möglich implementiert sind, d.h. einen möglichst geringen Overhead an Zeit bei der Ausführung der betroffenen Komponenten mit sich bringen. Auf den Mehrverbrauch von Arbeitsspeicher durch die zusätzlichen Kontrolleinheiten wurde hingegen nicht geachtet, da in der Praxis die Anzahl der Komponenten fast immer unter 1000 sein wird, und der zusätzliche Speicher so nicht schwer ins Gewicht fällt. Das letzte Ziel ist es, dass das Framework auf sämtlichen JVMs (*Java Virtual Machine*) und JDKs (*Java Development Kit*) benutzt werden kann. Zu diesen Zielen kommen noch zwei Anforderungen, die die Julia Entwickler stellen: Zum einen darf nur ein (Re)konfigurations Thread zu einer Zeit laufen, und zum anderen sind die internen Datenstrukturen von Julia nicht vor schadhafte Benutzerkomponenten geschützt. [3]

### 4.1.1 Grundkonzept

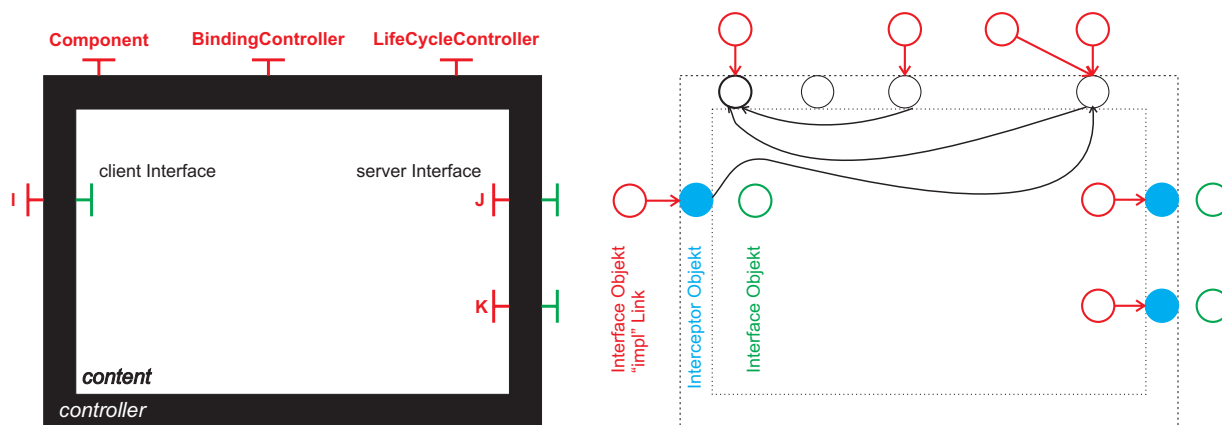


Abbildung 6: links: Fractal Komponente, rechts: mögliche Implementierung in Julia

Eine Fractal Komponente wird generell durch viele Java Objekte repräsentiert. Diese Objekte kann man wie in Abbildung 6 in drei Gruppen aufteilen:

1. Objekte, die die Komponenteninterfaces implementieren (in der Abbildung in rot und grün gekennzeichnet). Es gibt ein Objekt pro Komponenteninterface, wobei jedes Objekt einen „impl“ Link zu dem Objekt, das das Java Interface implementiert und zu dem alle Methodenaufrufe delegiert werden, hat. Für Client Interfaces ist dies eine `null` Referenz.
2. Objekte, die den *controller* Teil einer Komponente implementieren (blau und schwarz in der Abbildung), wobei ein Kontrollobjekt null oder mehr Kontrollinterfaces implementieren kann. Diese Objekte können nochmals in zwei Gruppen unterteilt werden: Zum einen Objekte, die die Kontrollinterfaces implementieren (in schwarz) und zum anderen (optionale) Interceptor Objekte (in blau), die eingehende und/oder ausgehende Funktionsaufrufe abfangen. Kontrollobjekte können sich gegenseitig referenzieren (schwarze Pfeile).
3. Objekte, die den *content* Teil der Komponente implementieren (in der Abbildung nicht dargestellt). Diese Objekte können zum einen Subkomponenten (bei zusammengesetzten Komponenten) sein, zum anderen aber auch Benutzerobjekte (bei primitiven Komponenten).

Da die Komponenteninterfaces getypt sind (d.h. ein Komponenteninterface implementiert sowohl `Interface` als auch das zugehörige Java Interface) muss jedes Komponenteninterface durch ein eigenes Java Objekt repräsentiert werden.

#### 4.1.2 Mixin Klassen

Wie in Kapitel 3 gesehen, ist eine der großen Stärken der Fractal Komponentenmodells die hohe Optionallität. D.h. für jede Komponente kann der Kontrolllevel individuell gewählt werden. Das bedeutet jedoch, dass Julia für jedes *conformance level* eine Implementierung zur Verfügung stellen muss. So muss Julia z.B. den Standard `BindingController` implementieren, jedoch auch Versionen davon, wenn das Standardtypsystem benutzt wird, wenn ein Lifecycle Controller vorhanden ist oder wenn zusammengesetzte Komponenten benutzt werden. Darüber hinaus sind hier auch noch verschiedene Kombinationen denkbar, d.h. es müssen des Weiteren Implementierungen für `BindingController` mit Standardtypsystem und vorhandenem Lifecycle Controller, oder zusätzlich noch vorhandenen zusammengesetzten Komponenten verfügbar sein. Man sieht, dass hier die Anzahl der nötigen verschiedenen Implementierungen schnell ansteigt. Außerdem sollen ja auch noch benutzerdefinierte Kontrollobjekte miteingebunden werden können, was dazu führen würde, dass sämtliche vorhandenen Implementationen wiederum abgeändert werden müssten.

Ein erster Gedanke, wie man dieses Problem lösen könnte, wären Klassenvererbungen. Doch sind diese hier nicht anwendbar. Sie würden einerseits zu einer kombinatorischen Explosion (wie oben gezeigt) führen und andererseits sehr viel Codeduplikation mit sich bringen. In obigem Beispiel müssen nämlich nicht nur die drei *concerns* „Typsystem“, „Lifecycle Controller“ und „zusammengesetzte Komponenten“ sondern auch alle möglichen Kombinationen implementiert sein, also  $2^3$  und damit 8 Klassen für 3 *concerns*.

Eine mögliche Lösung für dieses Problem wäre die aspektorientierte Programmierung z.B. mit AspectJ (ein Ansatz, den AOKell verfolgt). Als die Entwickler von Julia diesen Ansatz in Betracht gezogen haben, bestand jedoch noch als Problem, dass bei einer Implementierung mit AspectJ die Aspekte zur Compilezeit hinzugefügt werden müssen, und das deshalb der Sourcecode nötig ist. Daher konnte Julia nicht in einer kompilierten Version als JAR-Datei zur Verfügung gestellt werden, da so Programmierer ihre eigenen Kontrollobjekte nicht mehr hinzufügen könnten. Dieses Problem wurde inzwischen bei AspectJ behoben, so dass auch hier der Sourcecode als Basis genügt.

Um dieses Problem der Modularität und der Erweiterbarkeit also zu lösen, bedarf es einer Abwandlung der Aspekte, so dass diese auch zur Lade- oder Laufzeit eines Programms benutzt werden können. Julia benutzt zu diesem Zweck *Mixin Klassen*. Eine Mixin Klasse ist eine Klasse mit einer abstrakten Superklasse, die ein Minimum an Feldern und Methoden spezifiziert, die vorhanden sein müssen. Eine Mixin Klasse kann auf jede Superklasse angewendet werden (d.h. Overriding und Hinzufügen zusätzlicher Methoden), die mindestens die benötigten Felder und Methoden definiert. Diese Eigenschaft löst sowohl das kombinatorische Problem als auch das AspectJ Problem, denn die in Julia verwendeten Mixin Klassen können zur Laufzeit angewendet werden.

#### 4.1.3 Optimierungen

Bei der Implementation von Julia wurden zwei hauptsächliche Optimierungen vorgenommen, zum einen Optimierungen innerhalb einer Komponente, und zum anderen Optimierungen beim Zusammenspiel mehrerer Komponenten.

Um die Kontrollobjekte, die den *controller* einer Fractal Komponente bilden, zu (re)konfigurieren, könnte man Fractal selber benutzen: Jedes Kontrollobjekt wäre dann eine Komponente mit z.B. einem `BindingController` Interface um andere Kontrollobjekte zu binden. Mit diesem Ansatz könnte man auch zusammengesetzte Komponenten innerhalb des *controller* Parts einer Komponente benutzen. Die aktuelle Version von Julia benutzt jedoch einen anderen, effizienteren Ansatz: Es ist immer möglich die Kontrollobjekte zu einem einzigen Kontrollobjekt zusammenzuführen. Dabei kann jedes Kontrollobjekt

null oder mehr Java Interfaces benötigen oder zur Verfügung stellen. Die zur Verfügung gestellten Interfaces müssen von dem Objekt implementiert werden, und es muss ein Feld pro benötigtem Interface existieren. Der Name dieses Feldes hängt davon ab, ob es ein notwendiges oder ein optionales Interface ist. Jede Kontrollklasse, die mindestens ein Interface benötigt muss außerdem das `Controller` Interface implementieren. In einer gegebenen Konfiguration kann ein gegebenes Interface nur von einem Objekt zur Verfügung gestellt werden (außer das `Controller` Interface), da ansonsten ein Zusammenfügen der Objekte nicht möglich wäre, da ein Objekt nicht ein Interface auf verschiedene Arten implementieren kann. Die Bindings zwischen Objekten einer gegebenen Konfiguration werden nun automatisch in zwei Schritten erstellt:

1. Jedes Kontrollobjekt der Konfiguration wird in einem *naming service* registriert, der in der Praxis das Interface `InitializationContext` ist
2. Jedes Kontrollobjekt initialisiert sich dann selber unter Verwendung dieses *naming services*, um die Interfaces zu erhalten, die es benötigt

Wenn wir z.B. davon ausgehen, dass wir vier Kontrollinterfaces *I, J, K* und *L* haben und drei Kontrollklassen *IImpl, JImpl* und *KImpl* dann würden im unoptimierten Fall zuerst je eine Instanz von *IImpl, JImpl* und *KImpl* erzeugt werden, dann die resultierenden Objekte beim `InitializationContext` Objekt registriert werden und schließlich jedes Kontrollobjekt mit diesem Kontext initialisiert werden. Im optimierten Fall wird eine Klasse durch Zusammenfügen von *IImpl, JImpl* und *KImpl* dynamisch generiert (Oder vom Classpath geladen falls sie vorher schon statisch generiert wurde) und diese Klasse schließlich instanziiert. Auf diese Weise kann überflüssiger Arbeitsspeicherverbrauch vermindert werden.

Zusätzlich zu der Optimierung innerhalb einer Komponente gibt es bei Julia auch eine Optimierung zwischen den Komponenten, nämlich einen Algorithmus um *shortcut* Links zwischen Komponenten zu erstellen und zu aktualisieren. Wie bei dem Modell in Abbildung 6 sichtbar, besitzt jedes Objekt einer Komponente einen „impl“ Link zu einem Objekt, das dieses Komponenteninterface de facto implementiert. Im Falle eines Serverinterfaces *s* zeigt dieser Link grundsätzlich auf ein Interceptor Objekt, welches wiederum zu demjenigen Serverinterface zeigt, an das das jeweilige Clientinterface von *s* gebunden ist. Dies bringt bei einem großen System sehr viele Indirektionsstufen bei Funktionsaufrufen mit sich. Julia speichert in diesem Fall - wenn möglich - Abkürzungslinks, um überflüssige Delegationen zu vermeiden. Wenn z.B. ein Interceptor bestimmte Funktionsaufrufe nur delegiert, so kann Julia den Methodenaufruf eines Clients gleich an die gekapselte Komponente leiten und muss nicht den Weg über den Interceptor gehen. Diese Optimierung dient dem Geschwindigkeitsvorteil, da eine Vielzahl von delegierenden Methodenaufrufen entfallen.

## 4.2 AOKell

AOKell ist ebenfalls eine Implementation des Fractal Komponentenmodells für Java, wird im Gegensatz zu Julia, dessen aktuellste Version vom März 2005 stammt, aktuell weiterentwickelt (Version 2.0 vom Februar 2006). AOKell basiert auf einer aspekt-orientierten Implementation ist aber vom Grundaufbau Julia sehr ähnlich. Dies zeigt sich auch darin, dass von 142 JUnit Tests, die mit Julia ausgeliefert werden, 138 erfolgreich mit AOKell laufen. Im Vergleich zu anderen Implementierungen des Fractal Modells gibt es zwei Besonderheiten an AOKell: Zum einen stellt es - wie bei Julia bereits angedacht - einen komponentenorientierten Ansatz zur Implementation von *controllern* zur Verfügung. D.h. die Kontrollmembran einer Komponente ist selbst wiederum ein Zusammenschluss von Komponenten. Die zweite Besonderheit ist, dass AOKell Ideen der aspektorientierten Programmierung benutzt um die Anwendungs- und Kontrollleben stärker miteinander zu verbinden. Jede Kontrollkomponente ist mit einem Aspekt assoziiert, der die Ausführung der Anwendungskomponente überwacht und der Kontrollfunktionalitäten an die Kontrollkomponente delegiert. Dieser Ansatz fördert Modularität, da hier eine strikte Entkopplung von Implementation einer Kontrollkomponente und ihrer Integration in die Anwendungen vollzogen wird. Im Moment stehen zwei verschiedene Implementierungen dieser Verbindungsschicht für AOKell zur Verfügung. Die erste basiert auf *Spoon*, einem schnellen Java Compiler, die zweite basiert auf der aspektorientierten Programmiersprache *AspectJ*. Standardmäßig wird die *Spoon* Variante benutzt, da diese eine bessere

Performance zeigt.

Die Entwickler von AOKell haben einige Performance Tests gemacht (siehe [4]), um Julia mit AOKell zu vergleichen. Das Ziel dabei war es den Overhead, der entsteht, wenn eine Fractal Anwendung läuft, zu messen. Es wurde also nicht die Zeit zur Kompilation einer AOKell Anwendung oder die Zeit für Kontrollvorgänge gemessen, sondern die Zeit, die ein Benutzer erfährt, wenn er eine Anwendung benutzt, die mit AOKell implementiert ist. Die folgende Tabelle zeigt dir Ergebnisse. Für AOKell wurde jeweils die Version mit AspectJ und mit Spoon getestet, bei Julia die unoptimierte Version, und die optimierte Version, bei der wie oben beschrieben, Kontrollobjekte zusammengefasst werden (MERGEALL). Das Benchmark besteht aus einer zusammengesetzten Komponente und zwei primitiven Komponenten: einem Client und einem Server. Die Serverkomponente implementiert ein Interface mit 8 verschiedenen Methoden, jede mit einer anderen Signatur. Die Clientkomponente führt eine Million Aufrufe einer jeden Methode des Serverinterfaces durch, also insgesamt acht Millionen Funktionsaufrufe. Wie oben bereits

Implementation	Zeit in ms
AOKell 2.0 (AspectJ)	443 / 212
AOKell 2.0 (Spoon)	<b>221</b> / 212
Julia 2.1.1	484 / 234
Julia 2.1.1 (MERGEALL)	396 / 234

Tabelle 1: Performancevergleich zwischen Julia und AOKell Zeiten: (mit/ohne lifecycle Management)

erwähnt ist die AOKell Implementierung mit Spoon effizienter als die mit AspectJ. Der Zeitunterschied zwischen AOKell und Julia bei aktiviertem Lifecycle Management basiert vor allem auf einer Tatsache: Der Semantik, die der Lifecyclecontroller implementiert. Bei AOKell weist dieser Controller alle Anfragen auf gestoppte Komponenten zurück. Bei Julia blockiert der Lifecyclecontroller diese Aufrufe auf gestoppte Komponenten und zählt die Anzahl der ausführenden Methoden (um zu vermeiden, dass eine Komponente entfernt wird, die noch Anfragen ausführt). Diese Semantik bringt einigen Mehraufwand mit sich, u.a. das Management der gemeinsam benutzten Daten, die synchronisiert werden müssen.

Um einen Vergleich zu haben: Das selbe Benchmark benötigt in reinem Java 1.5 ohne Fractal Komponenten 172 ms mit einer vor dem Server geschalteten Proxyinstanz um Interception Kosten zu simulieren, und 122 ms ohne diese Proxyinstanz.

### 4.3 Think

Think ist eine C Implementierung des Fractal Komponentenmodells, die sich vor allem an Entwickler im Bereich der eingebetteten Systeme richtet. Betriebssystemarchitekten können Systemkomponenten beliebigen Umfangs erstellen, so dass Architektur Aspekte strikt von funktionalen Aspekten getrennt werden können. Dies führt auch zu einer klaren Aufgabenteilung: Architekten spezifizieren die Architektur, während Entwickler den funktionalen Code schreiben. Komponenten können von beliebiger Größe sein, von einfachen Semaphoren bis hin zu kompletten Kernels. Dieser Ansatz bringt viele Vorteile in der Entwicklungsphase mit sich: Code Wiederverwendung, Code Verbesserung oder gemeinsames Entwickeln reduzieren die Entwicklungszeit. Außerdem wird durch die Möglichkeiten der dynamischen Rekonfiguration eine hohe Flexibilität zur Laufzeit gewährt.

Die eindeutige Auslegung von Think für Kernels erkennt man auch daran, dass es dedizierte Sprachen für das Kerneldesign zur Verfügung stellt: Eine Beschreibungssprache für die Architektur (ADL - Architecture Description Language) und für die Interfaces (IDL - Interface Description Language). Desweiteren gibt es mit KORTX eine Komponentenbibliothek, die dem Kernelentwickler viele vorgefertigte Komponenten anbietet: Komponenten zur Hardwareabstraktion, Netzwerkkomponenten oder Dateisystemkomponenten. Hardwareabhängige Komponenten existieren für die gebräuchlichen Prozessorarchitekturen: PowerPC, ARM (StrongArm, XScale) und x86.

## 5 Bewertung

In den letzten vier Kapiteln haben wir gesehen wie das Fractal Komponentenmodell aufgebaut ist, wie man damit eine beispielhafte Anwendung entwickelt und welche konkreten Implementierungen es davon gibt. Es soll nun noch eine kurze Bewertung des Komponentenmodells aus meiner Sicht folgen.

Die Idee des komponentenorientierten Programmierens ist sicherlich ein guter Ansatz um einen nächsten Schritt von der objektorientierten Programmierung zu wagen. Durch eine strikte Trennung von concerns lässt sich - wie beim Beispiel des WebServers - eine Anwendung klar strukturieren und eine klare Struktur erkennen. Das Fractal Komponentenmodell bietet Features die meines Wissens viele andere Komponentenmodelle wie .Net oder CORBA Component Model nicht bieten. Der größte Vorteil ist sicherlich die hierarchische Verschachtelung von Komponenten, also die Möglichkeit, dass Komponenten sich aus anderen Komponenten zusammensetzen. Ebenso die gemeinsam verwendetes Komponenten (*shared*) zeichnen dieses Komponentenmodell aus.

Jedoch besteht immer das Problem, dass solche zusätzlichen Features nur schwierig ohne Spracherweiterungen in eine bestehende Programmiersprache eingebunden werden können. So auch bei den Implementierungen des Fractal Komponentenmodells. Die Möglichkeiten des dynamischen Rekonfigurierens und der Introspektion sind nur durch einen Mehraufwand an Code und durch „unschöne“ Programmierung zu bewerkstelligen. So funktioniert die Zuweisung von Bindings auf Basis von `Strings` und `Objects`. D.h. um eine Komponente an eine andere zu binden wird bei der `bindFc` Methode der Name der Komponente als `String`, und die Komponente selber als `Object` übergeben. Hierfür gibt es in Java natürlich keine Sprachkonstrukte. Jedoch ist diese Methode zum einen umständlich und zum anderen vor allem unsicher. Erstens müssen sämtliche dieser Komponenten dann immer gecastet werden auf den Komponententyp, der sie tatsächlich sind (da auch dies Java nicht wissen kann), zum zweiten treten durch die Vergleiche auf `Strings` zwangsläufig oft Laufzeitfehler auf. Es gibt keinerlei Mechanismus, zu überprüfen, was der Programmierer an Bindings vollzieht. Im Falle des WebServers könnte er z.B. anstelle eines Schedulers auch einen Logger an den *request receiver* binden. Erst bei einem Cast würde das System einen Laufzeitfehler bringen.

Dies ist sicher der Hauptkritikpunkt an dem Modell und seinen Implementierungen, der sich jedoch auch nur schwer ohne Spracherweiterung umgehen lässt. Außerdem scheint es, dass die von Fractal bezeichnete Referenzimplementierung Julia, aktuell nicht weiterentwickelt wird. Die aktuellste Version ist vom März 2005. Trotz ausgiebiger Recherchen scheint es keine größeren Anwendungen zu geben, die mit dem Fractal Komponentenmodell realisiert wurden. Wohl nicht zuletzt wegen der obigen Kritikpunkte.

Das Fractal Komponentenmodell ist also ein guter Ansatz, bei dem es jedoch an der praktischen Ausführung noch einige Probleme zu beseitigen gibt. Nicht zuletzt die Implementierung ProActive für verteilte Systeme könnte jedoch vor allem für das Grid interessant werden und vielleicht so zu weiteren Arbeiten in diese Richtung führen.

## A Sourcecode des WebServer

```
/* ===== Component interfaces ===== */

public interface RequestHandler {
    void handleRequest (Request r) throws IOException;
}

public interface Scheduler {
    void schedule (Runnable task);
}

public interface Logger {
    void log (String msg);
}

public class Request {
    public Socket s;
    public Reader in;
    public PrintStream out;
    public String url;
    public Request (Socket s) { this.s = s; }
}

/* ===== Component implementations ===== */

public class BasicLogger implements Logger {
    public void log (String msg) { System.out.println(msg); }
}

public class SequentialScheduler implements Scheduler {
    public synchronized void schedule (Runnable task) { task.run(); }
}

public class MultiThreadScheduler implements Scheduler {
    public void schedule (Runnable task) { new Thread(task).start(); }
}

public class FileRequestHandler implements RequestHandler {
    public void handleRequest (Request r) throws IOException {
        File f = new File(r.url);
        if (f.exists() && !f.isDirectory()) {
            InputStream is = new FileInputStream(f);
            byte[] data = new byte[is.available()];
            is.read(data);
            is.close();
            r.out.print("HTTP/1.0 200 OK\n\n");
            r.out.write(data);
        } else { throw new IOException("File not found"); }
    }
}

public class ErrorRequestHandler implements RequestHandler {
    public void handleRequest (Request r) throws IOException {
        r.out.print("HTTP/1.0 404 Not Found\n\n");
        r.out.print("<html>Document not found.</html>");
    }
}
```

---

```

public class RequestDispatcher implements RequestHandler, BindingController {
    private Map handlers = new TreeMap();
    // configuration concern
    public String [] listFc () {
        return (String []) handlers.keySet().toArray(new String [handlers.size()]);
    }
    public Object lookupFc (String itfName) {
        if (itfName.startsWith("h")) { return handlers.get(itfName); }
        else return null;
    }
    public void bindFc (String itfName, Object itfValue) {
        if (itfName.startsWith("h")) { handlers.put(itfName, itfValue); }
    }
    public void unbindFc (String itfName) {
        if (itfName.startsWith("h")) { handlers.remove(itfName); }
    }
    // functional concern
    public void handleRequest (Request r) throws IOException {
        Iterator i = handlers.values().iterator();
        while (i.hasNext()) {
            try {
                ((RequestHandler)i.next()).handleRequest(r);
                return;
            } catch (IOException _) { }
        }
    }
}

```

```

public class RequestAnalyzer implements RequestHandler, BindingController {
    private Logger l;
    private RequestHandler rh;
    // configuration concern
    public String [] listFc () { return new String [] { "l", "rh" }; }
    public Object lookupFc (String itfName) {
        if (itfName.equals("l")) { return l; }
        else if (itfName.equals("rh")) { return rh; }
        else return null;
    }
    public void bindFc (String itfName, Object itfValue) {
        if (itfName.equals("l")) { l = (Logger)itfValue; }
        else if (itfName.equals("rh")) { rh = (RequestHandler)itfValue; }
    }
    public void unbindFc (String itfName) {
        if (itfName.equals("l")) { l = null; }
        else if (itfName.equals("rh")) { rh = null; }
    }
    // functional concern
    public void handleRequest (Request r) throws IOException {
        r.in = new InputStreamReader(r.s.getInputStream());
        r.out = new PrintStream(r.s.getOutputStream());
        String rq = new LineNumberReader(r.in).readLine();
        l.log(rq);
        if (rq.startsWith("GET_")) {
            r.url = rq.substring(5, rq.indexOf('_', 4));
            rh.handleRequest(r);
        }
        r.out.close();
    }
}

```

```
    r.s.close();
  }
}

public class RequestReceiver implements Runnable, BindingController {
    private Scheduler s;
    private RequestHandler rh;
    // configuration concern
    public String[] listFc () { return new String[] { "s", "rh" }; }
    public Object lookupFc (String itfName) {
        if (itfName.equals("s")) { return s; }
        else if (itfName.equals("rh")) { return rh; }
        else return null;
    }
    public void bindFc (String itfName, Object itfValue) {
        if (itfName.equals("s")) { s = (Scheduler)itfValue; }
        else if (itfName.equals("rh")) { rh = (RequestHandler)itfValue; }
    }
    public void unbindFc (String itfName) {
        if (itfName.equals("s")) { s = null; }
        else if (itfName.equals("rh")) { rh = null; }
    }
    // functional concern
    public void run () {
        try {
            ServerSocket ss = new ServerSocket(8080);
            while (true) {
                final Socket socket = ss.accept();
                s.schedule(new Runnable () {
                    public void run () {
                        try { rh.handleRequest(new Request(socket)); } catch (IOException _) { }
                    }
                });
            }
        } catch (IOException e) { e.printStackTrace(); }
    }
}
```

---

## B ADL des WebServers

```

<!-- ===== Component types ===== ->

<definition name="webServer.RunnableType">
  <interface name="r" signature="java.lang.Runnable" role="server"/></provides>
</definition>

<definition name="webServer.FrontendType" extends="webServer.RunnableType">
  <interface name="rh" signature="webServer.RequestHandler" role="client"/>
</definition>

<definition name="webServer.ReceiverType" extends="webServer.FrontendType">
  <interface name="s" signature="webServer.Scheduler" role="client"/>
</definition>

<definition name="webServer.SchedulerType">
  <interface name="s" signature="webServer.Scheduler" role="server"/>
</definition>

<definition name="webServer.HandlerType">
  <interface name="rh" signature="webServer.RequestHandler" role="server"/>
</definition>

<definition name="webServer.AnalyzerType">
  <interface name="a" signature="webServer.RequestHandler" role="server"/>
  <interface name="rh" signature="webServer.RequestHandler" role="client"/>
  <interface name="l" signature="webServer.Logger" role="client"/>
</definition>

<definition name="webServer.LoggerType">
  <interface name="l" signature="webServer.Logger" role="server"/>
</definition>

<definition name="webServer.DispatcherType" extends="webServer.HandlerType">
  <interface name="h"
    signature="webServer.RequestHandler" role="client" cardinality="collection"/>
</definition>

<!-- ===== Primitive components ===== ->

<definition name="webServer.Receiver" extends="webServer.ReceiverType">
  <content class="webServer.RequestReceiver"/>
</definition>

<definition name="webServer.SequentialScheduler" extends="webServer.SchedulerType">
  <content class="webServer.SequentialScheduler"/>
</definition>

<definition name="webServer.MultiThreadScheduler" extends="webServer.SchedulerType">
  <content class="webServer.MultiThreadScheduler"/>
</definition>

<definition name="webServer.Analyzer" extends="webServer.AnalyzerType">
  <content class="webServer.RequestAnalyzer"/>
</definition>

<definition name="webServer.Logger" extends="webServer.LoggerType">

```

---

```
<content class="webServer.BasicLogger"/>
</definition >

<definition name="webServer.Dispatcher" extends="webServer.DispatcherType">
  <content class="webServer.RequestDispatcher"/>
</definition >

<definition name="webServer.FileHandler" extends="webServer.HandlerType">
  <content class="webServer.FileRequestHandler"/>
</definition >

<definition name="webServer.ErrorHandler" extends="webServer.HandlerType">
  <content class="webServer.ErrorRequestHandler"/>
</definition >

<!-- ===== Composite components ===== -->

<definition name="webServer.Handler" extends="webServer.HandlerType">
  <component name="rd" definition="webServer.Dispatcher"/>
  <component name="frh" definition="webServer.FileHandler"/>
  <component name="erh" definition="webServer.ErrorHandler"/>
  <binding client="this.rh" server="rd.rh"/>
  <binding client="rd.h0" server="frh.rh"/>
  <binding client="rd.h1" server="erh.rh"/>
</definition >

<definition name="webServer.Backend" extends="webServer.HandlerType">
  <component name="ra" definition="webServer.Analyzer"/>
  <component name="rh" definition="webServer.Handler"/>
  <component name="l" definition="webServer.Logger"/>
  <binding client="this.rh" server="ra.a"/>
  <binding client="ra.rh" server="rh.rh"/>
  <binding client="ra.l" server="l.l"/>
</definition >

<definition name="webServer.Frontend" extends="webServer.FrontendType">
  <component name="rr" definition="webServer.Receiver"/>
  <component name="s" definition="webServer.MultiThreadScheduler"/>
  <binding client="this.r" server="rr.r"/>
  <binding client="rr.s" server="s.s"/>
  <binding client="rr.rh" server="this.rh"/>
</definition >

<definition name="webServer.webServer" extends="webServer.RunnableType">
  <component name="fe" definition="webServer.Frontend"/>
  <component name="be" definition="webServer.Backend"/>
  <binding client="this.r" server="fe.r"/>
  <binding client="fe.rh" server="be.rh"/>
</definition >
```

## Abbildungsverzeichnis

1	Gesamtarchitektur des Webservers . . . . .	4
2	Fractal GUI . . . . .	7
3	Kapselung einer Komponente . . . . .	8
4	Externe Sicht auf eine Komponente . . . . .	11
5	Template und Instanz des Templates . . . . .	12
6	links: Fractal Komponente, rechts: mögliche Implementierung in Julia . . . . .	14

---

## Literatur

- [1] *Fractal - Getting Started*, E. Bruneton (France Telecom R&D), <http://fractal.objectweb.org/tutorial>
  - [2] *Fractal - Specification*, E. Bruneton, T. Coupaye (France Telecom R&D), J.B. Stefani (INRIA) <http://fractal.objectweb.org/specification>
  - [3] *Julia - Documentation*, E. Bruneton (France Telecom R&D), <http://fractal.objectweb.org/current/doc/javadoc/julia/overview-summary.html>
  - [4] *AOKell - Documentation*, L. Seinturier, N. Pessemier, T. Coupaye, <http://fractal.objectweb.org/tutorials/aokell>
  - [5] *Recursive and Dynamic Software Composition with Sharing*, E. Bruneton, T. Coupaye, J.B. Stefani, WCOP02, <http://fractal.objectweb.org/current/fractalWCOP02.pdf>
-