FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Automatisierung von Software-Mustern mittels Metaprogrammierung

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering / Internet Computing

eingereicht von

Andreas Schuh
Matrikelnummer 0726975

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Ao.Univ.Prof. Dipl.-Ing. Dr. Franz Puntigam

Wien, 3.12.2013

(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Andreas Schuh
Wienerstraße 4, Fels am Wagram

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Abstract

The topic of this thesis is the automation of software patterns. These software patterns occur during the development with conventional Java, without help of frameworks and generators. An indicator of such patterns is a frequent use of copy/paste/replace.

- Boilerplate-Code: Getter/Setter, Constructor, ...
- Design-Patterns: Proxy, Strategy, ...
- Abstraction-Code: Persistence (Data Access Object), ...
- Cross-Cutting-Concerns: Logging, ...

The problem with these patterns is, that they lead to a large amount of manually created code. This negatively affects the productivity, maintainability and extensibility of the software.

This thesis's interest is to find an answer to the following question: *Can the automation of software patterns improve the productivity, maintainability and extensibility for programming without frameworks?*

The approach is to specify the typical programming steps of the pattern in so-called metaprograms, so that the computer can execute them.
A functional prototype of an appropriate metaprogramming-system will be created.
The next step is to automate different patterns (boilerplate-code, design patterns, ...) They are randomly selected to get a statistical overview of whether patterns can be generated easily or only with difficulties.

In summary it can be said, that all selected patterns have been automated successfully and now can be programmed via annotations. The productivity, maintainability and extensibility has been improved due to reduced programming work, improved overview and central definition. Static code generation makes the function of metaprograms visible and helps debugging. The properties of dynamic metaprogramming were not missed. Furthermore the creation of metaprograms improve the understanding of patterns: Lots of common properties between patterns have been discovered. After a while, a core of common patterns could emerge, which could be used to build new programming languages.

All in all the developed solution is a mighty tool for the automation of software patterns, with lots of advantages and few disadvantages. The usage is flexible and platform-independent.

Kurzfassung

Das Thema dieser Diplomarbeit ist die Automatisierung von Software-Mustern, welche während der Entwicklung mit konventionellem Java, d.h. ohne die Hilfe von Frameworks und Generatoren, vorkommen. Das deutlichste Kennzeichen solcher Muster ist eine häufige Anwendung von Copy/Paste/Replace.

- Boilerplate-Code: Getter/Setter, Constructor, ...
- Design-Patterns: Proxy, Strategy, ...
- Abstraktions-Code: Persistenz (Data Access Object), ...
- Cross-Cutting-Concerns: Logging, ...

Diese Muster führen zu einer großen Menge an manuell erstelltem Code, welcher die Produktivität sowie Wart- und Erweiterbarkeit negativ beeinflusst.

Der Lösungsansatz ist, dass die typischen Programmierschritte des Musters in einem Metaprogramm spezifziert werden, damit sie der Computer generieren kann. Im Zuge dieser Diplomarbeit wird ein funktionaler Prototyp eines passenden Metaprogramming-Systems erstellt. Anschließend wird mit dessen Hilfe versucht verschiedene Muster (Boilerplate-Code, Design-Patterns,...) zu automatisieren. Diese werden stichprobenartig ausgewählt um einen statistischen Überblick zu erhalten, ob Muster prinzipiell gut oder schlecht generiert werden können.

Am Ende konnte auf folgende konkrete Forschungsfrage eine Antwort gegeben werden:
Kann die Automatisierung von Software-Mustern die Programmierung ohne Frameworks in Hinblick auf Produktivität, Erweiterbarkeit und Wartbarkeit verbessern?

Es konnten alle ausgewählten Muster sehr gut automatisiert werden, welche nun mittels deklarativer Programmierung (Annotationen) gesteuert werden. Dadurch konnte die Produktivität, Wart- und Erweiterbarkeit aufgrund von reduzierter Codierungsarbeit, besserer Übersicht und zentraler Verwaltung verbessert werden. Statische Codegenerierung macht die Funktion von Metaprogrammen sichtbar und erleichtert das Debugging. Eigenschaften von dynamischer Metaprogrammierung wurden während der Entwicklung nicht vermisst. Weiters führt die Erstellung von Metaprogrammen zu einem besseren Verständnis von Mustern: So konnten viele Gemeinsamkeiten zwischen Design-Patterns erkannt werden. Eventuell könnte sich nach einiger Zeit ein Kern an oft gebrauchten Mustern herauskristallisieren, welcher in die Konzeption neuer Programmiersprachen miteinbezogen werden könnte.

Insgesamt erwies sich die entwickelte Lösung als mächtiges Werkzeug zur Automatisierung von Code-Mustern, mit vielen Vorteilen und wenigen Nachteilen. Der Einsatz kann flexibel gestaltet werden und ist technologieunabhängig.

Inhaltsverzeichnis

KAPITEL 1

Einleitung

1.1 Problembeschreibung

Bei der Implementierung von Software werden oft Frameworks verwendet um den Aufwand zu reduzieren. Sie nehmen dem Entwickler Programmieraufgaben ab und ermöglichen so die Konzentration auf den eigentlichen Inhalt. Für viele Anwendungen existieren bereits etablierte Frameworks (z.B.: EJB3 für Web-Anwendungen), jedoch gibt es auch Anwendungen, für die es kein passendes Framework gibt und für die eine Anpassung zu aufwändig wäre.

Die Programmierung ohne Framework gibt sehr viel Freiheit bezüglich Architektur und ermöglicht somit die Lösung spezieller Problemstellungen. Jedoch erfordert dies viel zusätzlichen Code, welcher nichts mit der eigentlichen Funktionalität zu tun hat sondern nur das System zusammenhält. Beispielsweise muss bei der Erweiterung der Geschäftslogik oft an anderen Stellen etwas hinzugefügt und angepasst werden. Dadurch wird die Software schwer wartbar und fehleranfällig.

Bei genauerer Betrachtung fällt auf, dass dieser zusätzliche Programierungaufwand oft nach definierten Mustern abläuft. Damit gemeint sind nicht nur die bekannten Design-Patterns, sondern jeglicher Code, welcher nach einen bestimmten Muster programmiert wurde:

- Boilerplate-Code: Getter/Setter, Constructor, ...
- Design-Patterns: Strategy, Composite, ...
- Abstraktions-Code: Persistenz (Data Access Oject), ...
- Cross-Cutting-Concerns: Logging, ...

Viele solcher Muster sind genau definiert und sollten daher mit ausreichend Information auch automatisch generiert werden können. Ein Beispiel hierfür ist das Pattern *Proxy*: Durch die gemeinsame Schnittstelle mit dem Subjekt ist seine Funktion genau definiert. Trotzdem ist es schwer bis gar nicht möglich, dieses Pattern effizient in konventioneller objektorientierter Programmierung zu automatisieren. Bei jeder Änderung des Subjekts muss auch der Proxy manuell angepasst werden.

Ziel der Arbeit ist die Entwicklung einer Methode, mit dessen Hilfe Software, welche ohne Framework auskommen muss, effizienter umgesetzt werden kann.

Die Erwartungen sind:

- höhere Produktivität
- bessere Wart- und Erweiterbarkeit

durch:

- Automatisierung von Software-Mustern (Boilerplate-Code und Design-Patterns)
- Abstraktion von Technologien (DAOs,...)
- Lösungen für Cross-Cutting Concerns (Logging, ...)
- Teilweise Simulation fehlender Sprachfeatures (Delegation, Multimethoden, ...)
- Hohe Kompatibilität, da nur Sourcecode generiert wird (transparent für andere Technologien)

1.2 Methodisches Vorgehen

Zu Beginn wird gezielt nach themennaher Literatur recherchiert und versucht eigene Ideen einzuordnen und gegebenfalls bereits vorhandene Ideen in das Konzept einzubauen.

Danach erfolgt eine genaue Analyse der Problemstellung anhand konkreter Beispiele. Dadurch soll ermittelt werden, welche Anforderungen an die Problemlösung gestellt werden.

Anschließend folgt die Entwicklung eines konkreten Lösungsansatzes, welcher auch als Prototyp implementiert werden soll.

Danach wird mit Hilfe dieses Prototyps versucht verschiedene Muster (Boilerplate-Code, Design-Patterns,...) zu automatisieren. Diese werden stichprobenartig ausgewählt um einen statistischen Überblick zu erhalten, ob Muster prinzipiell gut oder schlecht generiert werden können.

Abschließend sollen alle Erfahrungen und Ergebnisse dokumentiert werden und auf folgende konkrete Forschungsfrage eine Antwort gegeben werden können:

Kann die Automatisierung von Software-Mustern die Programmierung ohne Frameworks in Hinblick auf Produktivität, Erweiterbarkeit und Wartbarkeit verbessern?

1.3 Überblick

- Kapitel 2 beschreibt alle benötigten Begriffe der Diplomarbeit.
- Kapitel 3 befasst sich mit der Literaturanalyse und dem State-of-the-Art.
- Kapitel 4 beschreibt das konkrete Umfeld der Diplomarbeit: Metaprogrammierung.
- Kapitel 5 sammelt Anforderungen durch Analyse konkreter Anwendungsbeispiele.
- Kapitel 6 erklärt den erarbeiteten Lösungsansatz.
- Kapitel 7 beschreibt die Implementierung des entwickelten Prototyps.
- Kapitel 8 bis 11 beschäftigt sich mit der praktischen Anwendung von Metaprogrammen.
- Kapitel 12 dokumentiert alle Ergebnisse und beantwortet die Forschungsfrage.
- Kapitel 13 fasst die Diplomarbeit in einem Fazit zusammen.
- Kapitel 14 gibt einen Ausblick auf künftige Arbeiten.

KAPITEL 2

Begriffserklärungen

In diesem Kapitel werden alle theoretischen Konzepte und Begriffe erklärt, welche in dieser Diplomarbeit verwendet werden. Diese Beschreibung umfasst eine allgemeine Diskussion mit Vor- und Nachteilen sowie die geschichtliche Entwicklung und konkrete Beispiele.

Es werden die folgenden Begriffe erklärt:

- Boilerplate-Code
- Design-Patterns
- Aspektorientiere Programmierung
- Java-Annotationen

2.1 Boilerplate-Code

Beschreibung

Als Boilerplate-Code werden Codefragmente bezeichnet, welche für eine relativ einfache Aufgabe verhältnismäßig viel trivialen Code benötigen. Ein deutliches Kennzeichen dafür ist, dass ein bestimmtes Codefragment in sehr ähnlicher Weise immer wieder vorkommt.

Geschichte

Boilerplate ist ursprünglich ein Begriff aus der Medienarbeit, welcher einen gleichbleibenden Text mit Informationen über das Unternehmen am Ende von Pressemitteilungen bezeichnet. Später wurde dieser Begriff aufgrund der ähnlichen Bedeutung in die Programmierung übernommen und auch in wissenschaftlichen Arbeiten verwendet [29].

Beispiel

Ein typisches Beispiel sind Getter und Setter, welche Variablen über Methoden öffentlich zugreifbar machen. Diese werden aufgrund einiger Vorteile standardmäßig in Java anstatt Public-Variablen eingesetzt.

Der einzige Zweck dieses Code-Abschnittes ist, dass die Klasse *Person* ein öffentliches Attribut *Name* besitzt. Die Methoden *getName* und *setName* sind Boilerplate-Codes.

```
public class Person
{
    private String name;

    public String getName(){
        return name;
    }

    public void setName(String name){
        this.name = name;
    }
}
```

Andere Sprachen haben dieses Problem mit Hilfe eigener Sprachkonstrukte erleichtert (z.B.: Properties in C#). Jedoch kommt Boilerplate-Code sehr häufig vor, wodurch nicht für alle Codewiederholungen ein eigenes Sprachkonstrukt erstellt werden kann.

Nachteile

Ein Nachteil ist die verringerte Produktivität, welche aufgrund der erhöhten Schreibarbeit entsteht. Dieser kann jedoch mit Hilfe von Entwicklungsumgebungen, welche einmalige Codegenerierung ermöglichen, reduziert werden.
Der viel größere Nachteil ist die verschlechterte Wartbarkeit. Bei einer Änderung von Typ oder Namen einer Variable müssen zusätzlich die zugehörigen Getter und Setter angepasst werden.

2.2 Design-Patterns

Beschreibung

Design-Patterns beschreiben bewährte Lösungen für bestimmte Entwurfsprobleme von objektorientierten Softwarearchitekturen. Sie stellen die gesammelten Erfahrungen von Experten dar, welche aufbereitet wurden, um auch in Zukunft davon profitieren zu können. Das Buch „Design Patterns: Elements of Reusable Object-Oriented Software“ [21] hat sich hierfür als Standardwerk etabliert. Darin werden Design-Patterns in natürlicher Sprache beschrieben und mit Hilfe von UML-Strukturdiagrammen dargestellt. Es wird für jedes Design-Pattern der konkrete Problemkontext, die Ziele, Vor- und Nachteile sowie Beispiele angegeben.

Im folgenden Abschnitt wird der geschichtliche Hintergrund geklärt und anschließend alle Design-Patterns, welche in dieser Diplomarbeit verwendet werden, einzeln beschrieben. Die Auswahl der umzusetzenden Design-Patterns erfolgte stichprobenartig, wobei oft verwendete Muster bevorzugt wurden. Das Ziel war einen statistischen Überblick zu erhalten, wie viele Design-Patterns sich prinzipiell gut oder schlecht automatisieren lassen.

Geschichte

Die Idee von wiederverwendbaren Patterns stammt ursprünglich aus dem Bereich der Architektur und wurde von Christopher Alexander im Buch „A Pattern Language: Towns, Buildings, Construction“ [3] beschrieben. Diese Idee wurde von Kent Beck and Ward Cunningham für den Entwurfs von grafischen User-Interfaces in die Softwareentwicklung übertragen [6]. Den Durchbruch erlangten Design-Patterns durch das Buch „Design Patterns: Elements of Reusable Object-Oriented Software“ [21], welches Design-Patterns für den Entwurf von Softwarearchitekturen beschreibt.

Observer-Pattern

Funktion

Observer-Objekte können sich bei *Subject*-Objekten registrieren um über Änderungen benachrichtigt zu werden.

Beschreibung

Die Beschreibung bezieht sich auf die Abb. 2.1 „Strukturdiagramm des Observer-Patterns [21]“ auf Seite 8.
Ein *Subject*-Objekt hält Referenzen zu einer Menge von *Observer*-Objekten, welche zur Laufzeit hinzugefügt und entfernt werden können (*Attach*, *Detach*). *Subject*-Objekte können Events auslösen (*Notify*), welche an die registrierten *Observer*-Objekte weitergeleitet werden (*Update*). Optional können *ConcreteObserver*-Objekte eine Referenz zu einem *ConcreteSubject*-Objekt haben um den Status auszulesen (*getState*).

Vorteile

- *Observer*-Objekte erhalten die Benachrichtung automatisch, wodurch kein ständiges Abfragen notwendig ist (Polling).
- Lose Kopplung zwischen *Observer*- und *Subject*-Objekt.

Nachteile

- Bei vielen *Observer*-Objekten kann eine Änderung viel Zeit benötigen, da alle *Observer*-Objekte benachrichtigt werden müssen.
- Es können Endlosschleifen auftreten, falls eine Benachrichtigung wiederum zu einer Änderung führt.

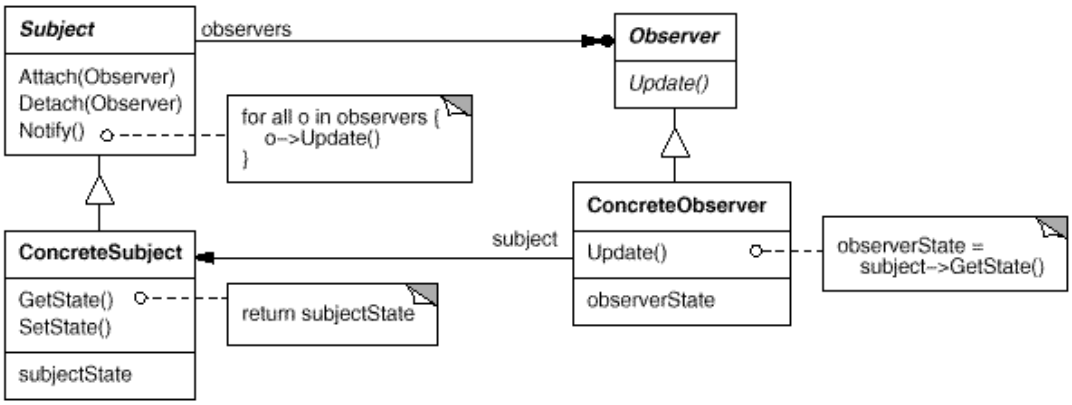

Abbildung 2.1: Strukturdiagramm des Observer-Patterns [21]

Beispiel

Das Beispiel (siehe Abb. 2.2 „Anwendungsbeispiel des Observer-Patterns [21]“ auf Seite 9) zeigt eine Anwendung des Observer-Patterns um Änderungsbenachrichtungen umzusetzen. Das *Subject*-Objekt speichert Daten, welche in drei verschiedenen Ansichten (*Observer*-Objekte) dargestellt werden. Bei einer Änderung der Daten sollen alle Ansichten aktualisiert werden.

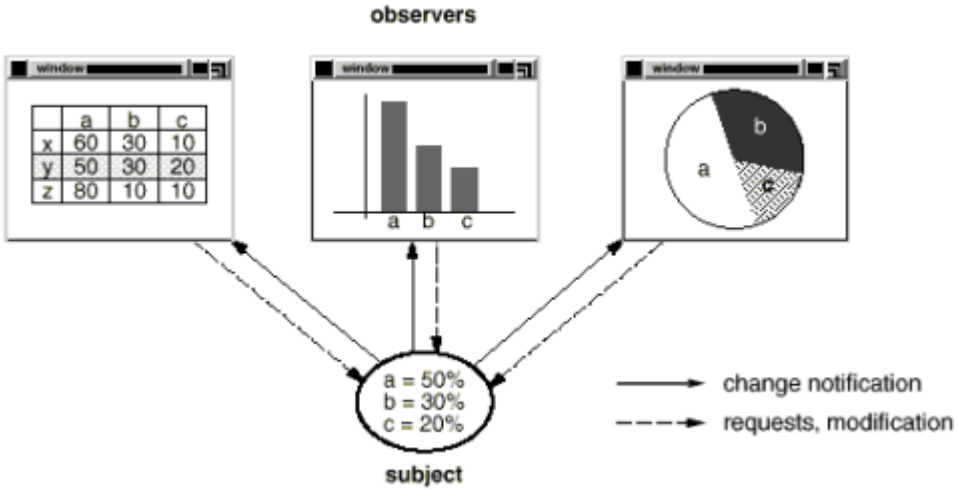

Abbildung 2.2: Anwendungsbeispiel des Observer-Patterns [21]

Strategy-Pattern

Funktion

Unterschiedliche Algorithmen werden in konkrete *Strategy*-Klassen ausgelagert und können zur Laufzeit gewechselt werden.

Beschreibung

Die Beschreibung bezieht sich auf die Abb. 2.3 „Strukturdiagramm des Strategy-Patterns [21]“ auf Seite 10.
Das *Context*-Objekt speichert eine Referenz auf ein *Strategy*-Objekt um dessen Methoden nutzen zu können (*AlgorithmInterface*). Das *Strategy*-Interface verbirgt die Implementierung von konkreten *Strategy*-Objekten vor dem *Context*-Objekt, wodurch sie zur Laufzeit einfach ausgewechselt werden können.

Vorteile

- Das Verhalten (Algorithmus) von *Strategy*-Objekten kann in verschiedenen *Context*-Objekten ohne Subtyping wiederverwendet werden.
- *Strategy*-Objekte können zur Laufzeit getauscht werden.
- Vermeidung von Fallunterscheidungen.

Nachteile

- Clients müssen konkrete *Strategy*-Implementierungen kennen.
- Erhöhte Klassenanzahl durch *Strategy*-Klassen.

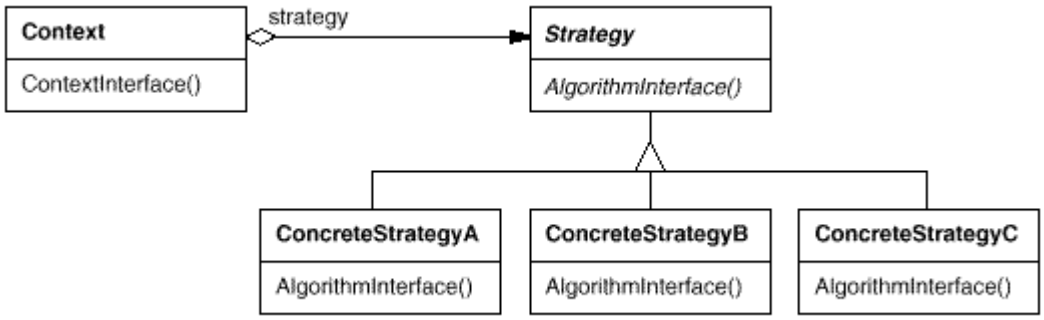

Abbildung 2.3: Strukturdiagramm des Strategy-Patterns [21]

Beispiel

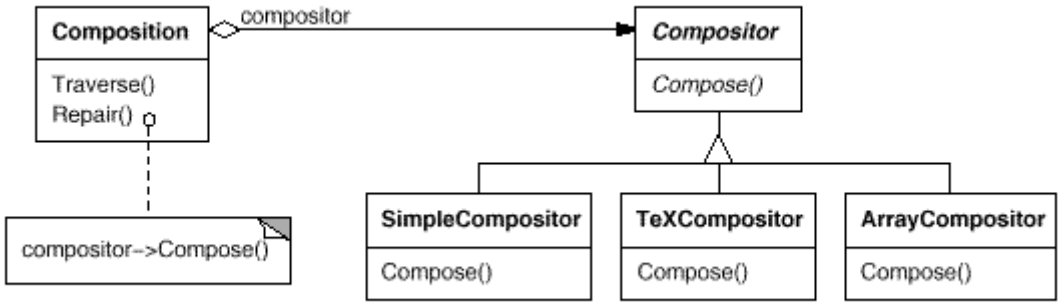

Abbildung 2.4: Anwendungsbeispiel des Strategy-Patterns [21]

In diesem Beispiel (siehe Abb. 2.4 „Anwendungsbeispiel des Strategy-Patterns [21]“ auf Seite 10) wird das Strategy-Pattern verwendet um das Verhalten wiederverwendbar in verschiedene Algorithmen zu kapseln. Der *Compositor*-Algorithmus kann auf drei verschiedene Arten umgesetzt werden (Simple, TeX, Array). Das *Composition*-Objekt kann durch eine Referenz auf ein *Compositor*-Objekt dessen Algorithmus verwenden und bei Bedarf zur Laufzeit austauschen.

Composite-Pattern

Funktion

Composite-Objekte ermöglichen eine einfache Erstellung und Verarbeitung von Objekthierarchien.

Beschreibung

Die Beschreibung bezieht sich auf die Abb. 2.5 „Strukturdiagramm des Composite-Patterns [21]“ auf Seite 11.
Die *Composite*-Klasse implementiert das *Component*-Interface und kann mehrere Referenzen (*Children*) auf andere *Component*-Objekte speichern (*Add*, *Remove*). Ausgewählte Methoden des *Component*-Interfaces (*Operation*) werden vom *Composite*-Objekt an alle *Children* weitergeleitet. Der Client kennt nur das *Component*-Interface, wodurch eine gute Entkopplung erreicht wird.

Vorteile

- Einheitliches und vereinfachtes Arbeiten mit Objekthierarchien.
- Stabiles Client-Interface, da neue Objekte zur Hierarchie ohne Änderung des Interfaces hinzugefügt werden können.

Nachteile

- Composite-Objekte definieren keine Einschränkungen, sodass auch semantisch ungültige Objekthierarchien aufgebaut werden können. Abhilfe schaffen nur Laufzeit-Typüberprüfungen.

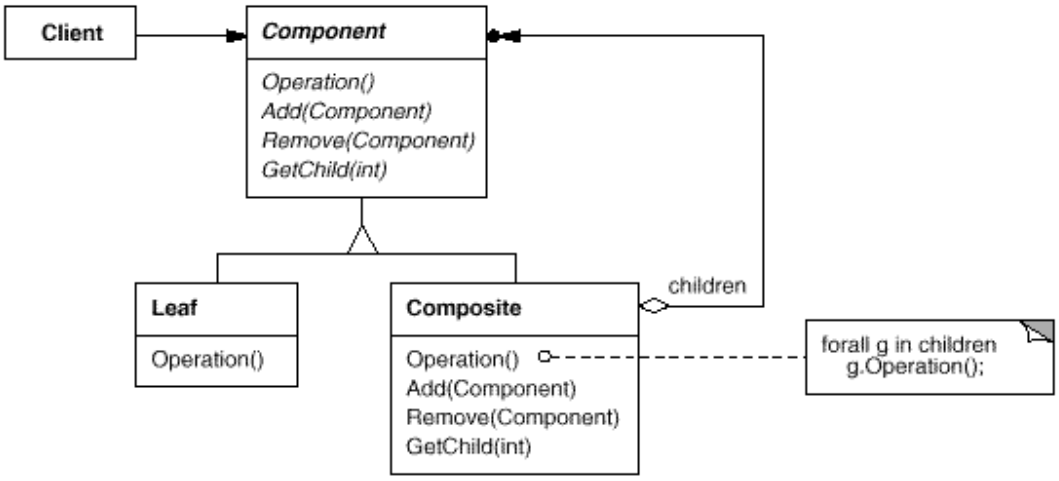

Abbildung 2.5: Strukturdiagramm des Composite-Patterns [21]

Beispiel

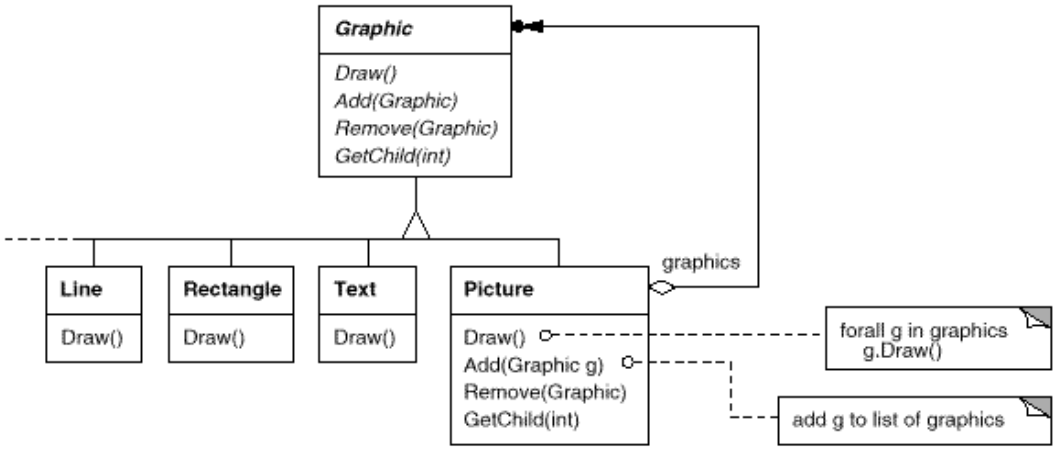

Abbildung 2.6: Anwendungsbeispiel des Composite-Patterns [21]

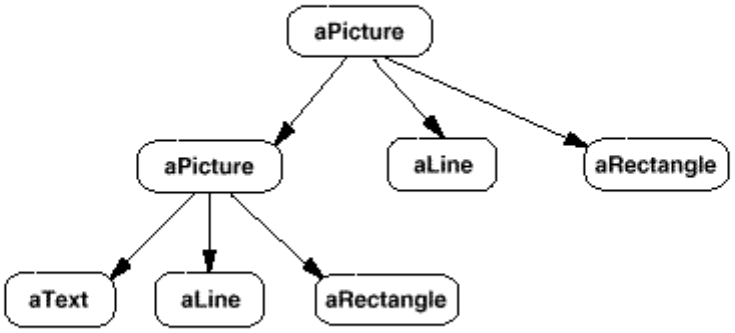

Abbildung 2.7: Beispiel für eine Objekthierarchie [21]

In diesem Beispiel (siehe Abb. 2.6 „Anwendungsbeispiel des Composite-Patterns [21]“ auf Seite 12) wird das Composite-Pattern verwendet um eine Hierarchie von Graphic-Elementen aufzubauen. Das *aPicture*-Objekt ist selbst ein *Graphic*-Object und kann außerdem mehrere *Graphic*-Objekte enthalten. Dadurch kann eine Hierarchie von *Graphic*-Objekten aufgebaut werden. Funktionen wie z.B. *draw* werden von *aPicture*-Objekt auf alle Objekte der Hierarchie ausge-

führt. Die folgende Abbildung (siehe Abb. 2.7 „Beispiel für eine Objekthierarchie [21]“ auf Seite 12) zeigt ein Beispiel einer Objekthierarchie.

Visitor-Pattern

Funktion

Visitor-Klassen können zusammenhängende Algorithmen über Objekthierarchien realisieren.

Beschreibung

Die Beschreibung bezieht sich auf die Abb. 2.8 „Strukturdiagramm des Composite-Patterns [21]“ auf Seite 14.
Die *Visitor*-Klasse hat für jede konkrete *Element*-Klasse (*ConcreteElement*) eine *Visit*-Methode (*VisitConcreteElement*). Ein konkretes *Visitor*-Objekt kann durch die *Accept*-Methode eines *Element*-Objektes ausgeführt werden. In dieser Methode ruft das konkrete *Element*-Objekt die *Visit*-Methode des *Visitor*-Objektes auf. Mit Hilfe eines dynamischen Dispatchs wird die passende *Visit*-Methode aufgerufen.

Vorteile

- Es bietet eine gute Alternative zur Fallunterscheidung anhand des dynamischen Typs.
- Es können zusammenhängende Algorithmen über Strukturen (Objekt-Hierarchien) definiert werden.
- Neue Algorithmen (Operationen) können durch Definition einer neuen *Visitor*-Klasse leicht hinzugefügt werden.

Nachteile

- Bei Änderungen der Objekthierarchie (neue Klasse hinzugefügt/entfernt) müssen alle *Visitor*-Klassen überarbeitet werden.

Beispiel

In diesem Beispiel (siehe Abb. 2.9 „Objekthierarchie für Node-Objekte [21]“ auf Seite 14) wird das Visitor-Pattern verwendet um Algorithmen über eine Hierarchie von Node-Elementen zu definieren. Die *NodeVisitor*-Klassen haben dazu eine Methode für jeden unterschiedlichen *Node*-Subtyp. Beim Durchlauf einer Hierarchie wird in den *NodeVisitor*-Klassen für jedes *Element*-Objekt jeweils die treffende Methode ausgeführt. Dadurch kann ein zusammenhängender Algorithmus wie *TypeChecking* oder *CodeGeneration* über *Node*-Klassen definiert werden.

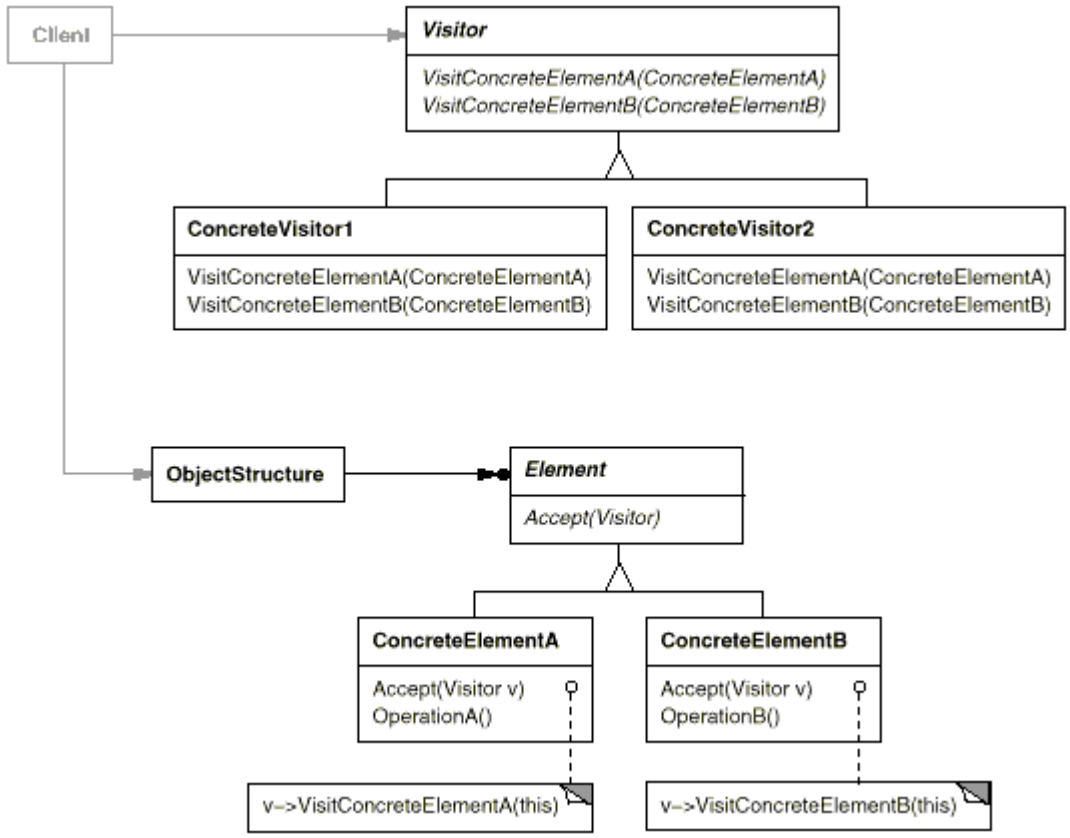

Abbildung 2.8: Strukturdiagramm des Composite-Patterns [21]

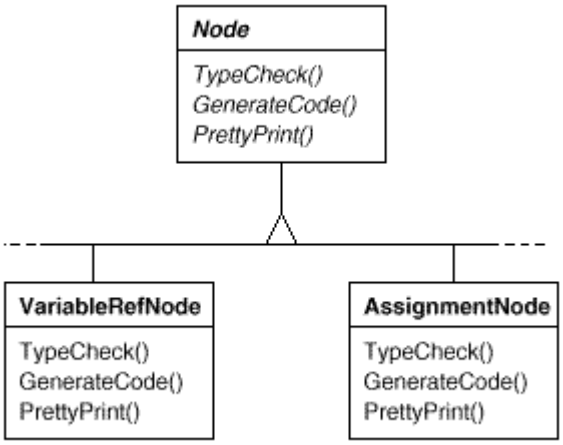

Abbildung 2.9: Objekthierarchie für Node-Objekte [21]

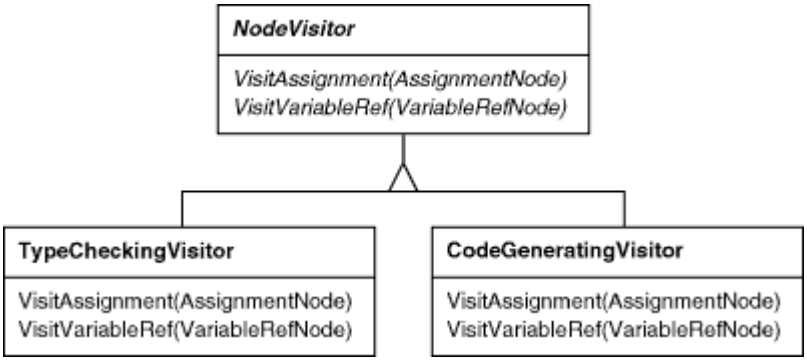

Abbildung 2.10: Anwendungsbeispiel des Visitor-Patterns [21]

Command-Pattern

Funktion

Methodenaufrufe werden in *Command*-Objekte gekapselt um besser verwaltet werden zu können (z.B.: Undo-Funktion).

Beschreibung

Die Beschreibung bezieht sich auf die Abb. 2.11 „Strukturdiagramm des Command-Patterns [21]“ auf Seite 16.
Die *Command*-Klasse deklariert ein Interface zur Ausführung einer Operation (*Execute*). Das konkrete *Command*-Objekt (*ConcreteCommand*) hat eine Referenz zum *Receiver*-Objekt um eine bestimmte Methode bei der Ausführung des Befehls starten zu können. Der Client erzeugt ein konkretes *Command*-Objekt und setzt dessen *Receiver*-Objekt. Anschließend kann mit Hilfe des *Invoker*-Objekts der Befehl gestartet werden.

Vorteile

- Lose Kopplung zwischen aufrufenden und ausführenden Objekt.
- *Command*-Objekte können wie normale Objekte gespeichert und manipuliert werden (z.B. Undo-Funktion).

Nachteile

- Erhöhte Klassenanzahl durch *Command*-Klassen.

Beispiel

Dieses Beispiel (siehe Abb. 2.12 „Anwendungsbeispiel des Command-Patterns [21]“ auf Seite 16) zeigt eine Anwendung des Command-Patterns um Undo-Funktionen umzusetzen. Es wird

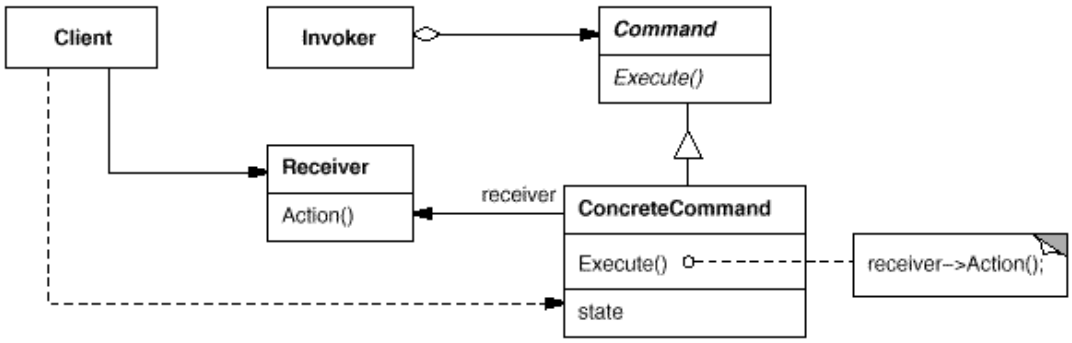

Abbildung 2.11: Strukturdiagramm des Command-Patterns [21]

ein sehr allgemeines Command-Interface implementiert, welches nur die Methode *execute* enthält. Der konkrete Befehl *PasteCommand* führt die *paste*-Methode in dem Receiver-Objekt (Document) aus. Der Vorteil ist, dass alle ausgeführten Befehle in einer Liste gespeichert werden und so gegebenenfalls rückgängig gemacht werden können.

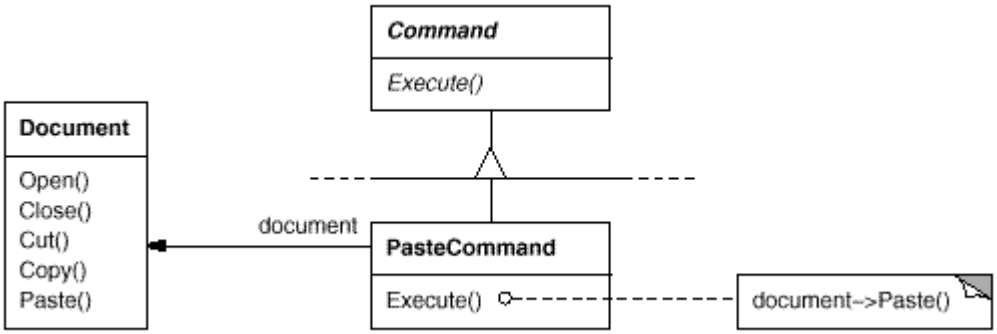

Abbildung 2.12: Anwendungsbeispiel des Command-Patterns [21]

2.3 Sonstige Code-Muster

Beschreibung

In diesem Abschnitt werden zwei weitere Code-Muster beschrieben, welche jedoch keine klassischen Design-Patterns aus den Buch „Design Patterns: Elements of Reusable Object-Oriented Software“ [21] sind.

Multimethoden in Java

Funktion

Die Methodenauswahl erfolgt anhand des dynamischen Typs von mehreren Objekten.

Beschreibung

Bei Multimethoden wird die auszuführende Methode dynamisch anhand des Parameters als auch des Empfängers bestimmt. Die Methode wird also anhand des dynamischen Typs von zwei Objekten ausgewählt. Java kann dies nur mittels zweifachem Dispatch simulieren.

Multimethoden können am Einfachsten durch ein konkretes Beispiel erklärt werden:

Beispiel

Dieses Beispiel zeigt eine typische Anwendung von Multimethoden bei Kollisionen von Objekten. Es wird das Spiel *Asteroids* beschrieben, in dem alle Objekte miteinander kollidieren können (Asteroid, Spaceship, Rocket). Jede Kollision führt zu einem unterschiedlichen Ereignis, welches durch Multimethoden umgesetzt werden soll. Das Ereignis wird anhand des dynamischen Typs von zwei kollidierenden Objekte ausgewählt.

Der Sourcecode zeigt nur die Implementierung der *Asteroid*-Klasse, jedoch sind alle anderen Objekte (Spaceship, Rocket) analog aufgebaut. Jede beteiligte Klasse muss eine First-Dispatch-Methode, sowie für jede beteiligte Klasse eine Second-Dispatch-Methode erstellen. Es ist offensichtlich, dass die Methodenanzahl mit den beteiligten Klassen enorm ansteigt. Wenn nun die *dispatchCollisionWith*-Methode mit einem Objekt als Parameter aufgerufen wird, erfolgt ein zweifacher Dispatch, welcher schlussendlich zur erwarteten Methode führt.

```
public class Asteroid implements Collidable{

   public void collisionWith(Collidable c) {
      System.out.println("Asteroid destroyed.");
   }

   public void dispatchCollisionWith(Collidable e) { //first dispatch
      e.dispatchCollisionWith(this);
   }

   public void dispatchCollisionWith(Asteroid e) {  //second dispatch
      e.collisionWith(this);
```

```
    }
    public void dispatchCollisionWith(Rocket e) {  //second dispatch
        e.collisionWith(this);
    }
    public void dispatchCollisionWith(Spaceship e) { //second dispatch
        e.collisionWith(this);
    }
}
```

Vorteile

- Java kann Multimethoden nutzen.
- Es werden Typabfragen zur Laufzeit vermieden.

Nachteile

- Die benötigte Anzahl an Dispatch-Methoden ist bei vielen beteiligten Klassen enorm.
- Schlechte Wart- und Erweiterbarkeit, da jede neue beteiligte Klasse eine Änderung aller beteiligten Klassen benötigt.

Data-Access-Object (DAO)

Funtion

Das Data-Access-Objekt (DAO) wird verwendet um den Zugriff auf Datenquellen technologieunabhängig zu kapseln.

Beschreibung

Es wird für jedes zu speichernde Objekt ein eigener DAO erstellt, welcher ein Interface für Create-Read-Update-Delete-Operationen (CRUD) bereitstellt. Dieses Interface versteckt die konkrete Technologie der verschiedenen Implementierungen (XML, Binary). Da der Client nur das abstrakte DAO-Interface verwendet, können Technologien einfach ausgetauscht werden.

Beispiel

Das Beispiel (siehe Abb. 2.13 „Anwendungsbeispiel des DAO-Patterns[21]“ auf Seite 19) zeigt die Struktur des DAO-Patterns anhand eines Anwendungsbeispiels.

- *Client*: Verwendet das allgemeine DAO-Interface.
- *PersonDAO*: Definiert allgemeine CRUD-Operationen.
- *XMLPersonDAO, BinaryPersonDAO*: Stellen zwei unterschiedliche Implementierungen des DAO-Interfaces dar, welche XML und Binary als Technologie verwenden.

Vorteile

- Der Zugriff auf Datenquellen ist für den Client transparent.
- Konkrete Technologien können einfach getauscht werden.

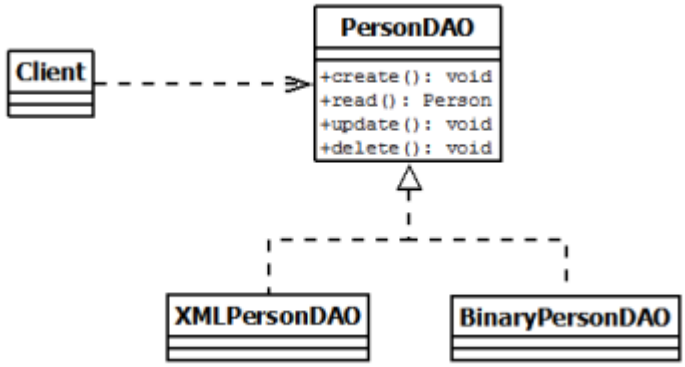

Abbildung 2.13: Anwendungsbeispiel des DAO-Patterns[21]

Nachteile

- Stark erhöhte Klassenanzahl durch *DAO*-Klassen und deren konkrete Implementierungen.
- Die Erweiterung einer neuen Klasse ist trivial, muss jedoch manuell durchgeführt werden (Boilerplate-Code).

2.4 Aspektorientierte Programmierung

Beschreibung

Die aspektorientierte Programmierung (AOP) ist ein Programmierparadigma, welches auf die objektorientierte Programmierung aufbaut und versucht das Problem der Cross-Cutting-Concerns zu lösen. Cross-Cutting-Concerns sind Anforderungen, welche nicht zentral definiert werden können und daher im Code verstreut werden müssen. Typische Beispiele hierfür sind Logging, Tracing und Security.

Geschichte

Die Wurzeln der aspektorientierten Programmierung liegen im Meta-Object-Protocol für Lisp, welches von Gregor Kiczales und seinen Kollegen entwickelt wurde [26]. Dieses ermöglichte neben vielen anderen Dingen auch aspektorientierte Programmierung. Später wurde daraus ein eigenständiges Konzept, welches als AspectJ für Java umgesetzt wurde [25].

Beispiel

Ein Beispiel für aspektorientiere Programmierung ist das Tracing von Methoden. Dabei wird beim Eintritt und Austritt einer Methode eine Nachricht ausgegeben. Wird Tracing in konventionellem Java umgesetzt führt dies zu einer hohen Streuung im Code und kann nicht zentral verwaltet werden.

```
public class ClassA
{
   public void methodA()
   {
      System.out.println("Entering methodA");

      //do something

      System.out.println("Leaving methodA");
   }
}
```

Aspektorientierte Programmierung ermöglicht die Definition von Aspekten, welche diese Probleme lösen können. Ein Aspekt kann dazu Pointcuts definieren, welche mit Hilfe einer eigenen Selektionssprache, eine bestimmte Menge an Methoden auswählen können. Für diese Auswahl an Methoden kann anschließend bei verschiedenen Ereignissen (*after*, *before*) Code ausgeführt werden (Advices). In diesem Beispiel wird ein Pointcut für alle Methoden der Klassen im Package *Project* definiert. Weiters werden Advices für den Eintritt und Austritt von Methoden erstellt.

```
public aspect Tracing
{
    pointcut trace() : call(* Project.*(..));

    before(): trace() {
        System.out.println("Entering \"" + thisJoinPoint + "\"");
    }
```

```
    after(): trace() {
        System.out.println("Leaving \"" + thisJoinPoint + "\"");
    }
}
```

Vorteile

- Wart- und Erweiterbarkeit: Cross-Cutting-Concerns können zentral definiert werden und dadurch die Wart- und Erweiterbarkeit stark verbessern.

Nachteile

- Seiteneffekte: Aspektorientierte Programmierung verwenden oft Bytecode-Manipulation, wodurch Änderungen unsichtbar sind. Erst spezielle Debugger ermöglichen eine gezielte Fehlersuche.

2.5 Java-Annotationen

Beschreibung

Annotationen ermöglichen die Definition von Metadaten für bestimmte Sprachelemente wie Klassen, Methoden oder Variablen. Metadaten sind zusätzliche Informationen, die direkt in den Quellcode eingetragen werden, um die Semantik von Programmiersprachen zu erweitern. Annotationen können durch verschiedene Werkzeuge ausgelesen und interpretiert werden. Die offizielle Version dafür ist die Java-Reflection-API, welche jedoch nur sehr eingeschränkte Möglichkeiten zur Manipulation bietet. Aus diesem Grund werden oft externe Werkzeuge wie Bytecode-Manipulationsbibliotheken [13][12] verwendet.

Geschichte

XDoclet war eines der ersten Werkzeuge, welches Annotationen in Form von speziell interpretierten Kommentaren in Java umsetzte [42]. Ab Java 1.5 wurden Annotationen als eigene Sprachelemente hinzugefügt und konnten mit der bereits vorhandenen Java-Reflection-API ausgelesen werden.

Beispiel

Es wird das Beispiel über Getter und Setter erneut verwendet (siehe Abschnitt 2.1 „Beispiel“ auf Seite 6). Wie bereits erwähnt ist der relevante Teil dieses Codes nur, dass die Klasse *Person* ein öffentliches Attribut *Name* hat. Annotationen können nun verwendet werden um diese Information kompakter zu machen. Mittels @GetSet über der Variable wird definiert, dass sowohl Getter als auch Setter generiert werden sollen.

```
public class Person
{
   @GetSet
   private String name;
}
```

Vorteile

- Übersichtlichkeit durch deklarative Programmierung: Annotationen können Funktionalität kompakter beschreiben.
- Annotationen bieten eine einfache und flexible Möglichkeit die Semantik einer Programmiersprache zu erweitern.

Nachteile

- Die Reflection-API von Java ist viel zu eingeschränkt, wodurch auf externe Werkzeuge zurückgegriffen werden muss.
- Die Implementierung von Java-Annotationen ist nicht optimal. Es ist nur umständlich möglich den selben Annotations-Typ mehrfach über ein Element zu schreiben.

KAPITEL 3

State-of-the-Art

In diesem Kapitel werden verschiedene wissenschaftliche Arbeiten vorgestellt, welche sich mit einer ähnlichen Zielsetzung beschäftigten.

3.1 Literaturrecherche

Zu Beginn mussten Stichworte gefunden werden, welche Arbeiten mit ähnlichen Zielsetzungen identifizieren können. Als gemeinsamer Nenner stellte sich die **Automatisierung und Implementierung von Design-Patterns** heraus. Die Recherche wurde online durchgeführt, wobei verwandte Arbeiten über Referenzen gefunden werden konnten.

Die Arbeiten konnten in folgende Gruppen eingeteilt werden:

Assistent für Design-Patterns

Es wurden Assistent-Programme (Wizards) entwickelt, welche semiautomatische Codegenerierung von Design-Patterns ermöglichen. Der Nutzer kann Design-Patterns parametrisieren (Tradeoffs) um ein passendes Grundgerüst zu generieren, welches anschließend manuell mit Business-Logik erweitert werden muss. Semiautomatische Code-Generierung bietet nur einmalige Reduzierung von Codierungsarbeit und führt auf langer Sicht zu keiner Verbesserung der Wart- und Erweiterbarkeit.

Relevante Arbeiten: UMLStudio [10], ModelMaker [10], Cogent [20]

Design-Patterns als Sprachelemente

Es wurden neue Sprachen entwickelt oder vorhandene Sprachen erweitert um Design-Patterns als explizite Sprachelemente zu unterstützen. Das hat den Vorteil, dass die Architektur explizit

im Code sichtbar ist und dadurch besser eingebunden werden kann. Die Zielsetzung unterscheidet sich jedoch zu stark, wodurch die neuen Programmiersprachen viel zu unflexibel sind um beliebige Muster automatisieren zu können.

Relevante Arbeiten: PaL [18][7], LayOM [9][8]

Design-Patterns mittels aspektorientierter Programmierung

Aspektorientierte Programmierung ermöglicht effizientes Abfangen von Methoden, wodurch Cross-Cutting-Concerns zentraler definiert werden können. Die gefundenen Arbeiten geben zwar gute Beispiele, wie Design Patterns einfacher (kürzer) implementiert werden können, jedoch müssen diese aufgrund der eingeschränkten Funktionalität (Abfangen von Methoden) teilweise stark angepasst werden.

Relevante Arbeiten: AspectJ [23][24][35]

Design-Patterns mittels Model-Driven-Development

Beim Model-Driven-Development wird Software durch Modelle (meist UML) abgebildet um anschließend automatisch Sourcecode zu generieren. Dieser Ansatz ist jedoch sehr schwergewichtig, da auch für einfache Muster verhältnismäßig aufwendige Modelle entwickelt werden müssten.

Relevante Arbeiten: Eclipse Modeling Framework [15], Pattern Modeling Framework [16]

Design-Patterns mittels Metaprogrammierung

Es wurde gezeigt wie erweiterte Sprachfeatures (z.B. Reflection, Metaprogrammierung) verwendet werden können um Design Pattern zu implementieren. Die Arbeiten sind eine eindrucksvolle Machtdemonstration von Lisp und Scheme, jedoch nicht mit konventioneller, objektorientierter Programmierung vergleichbar, weshalb die Ansätze nicht verwendet werden können. C++ Templates werden in Kombination mit konventioneller Objektorientierung verwendet, wirken jedoch kompliziert und umständlich.

Relevante Arbeiten: C++ Templates [5], Lisp [41], Scheme [38]

3.2 Fazit

Die Literaturrecherche hat gezeigt, dass zwar viele Forschungsarbeiten in diesem Bereich existieren aber sich keine mit den selben Zielsetzungen wie diese Diplomarbeit beschäftigt. Jedoch konnte die Metaprogrammierung als vielversprechende Richtung eingegrenzt werden, welche aus diesem Grund im folgenden Kapitel genauer beleuchtet wird.

KAPITEL 4

Umfeld: Metaprogrammierung

In diesem Kapitel wird der Begriff Metaprogrammierung ausführlich anhand von Definition und Geschichte erklärt. Im Anschluss folgt eine Aufzählung der aktuellen Werkzeuge (State-of-the-Art) in diesem Bereich.

4.1 Definition

Für den Begriff Metaprogrammierung gibt es keine eindeutige Definition, sondern abhängig vom Kontext unterschiedliche Bedeutungen. Aus diesem Grund wurde sogar ein Buch [37] geschrieben, welches sich mit der Definition und Taxonomie von Metaprogrammierung ausführlich beschäftigt.

Eine besonders kurze und treffende Definition wurde jedoch in [27] gefunden:

> „Meta-programs are programs that analyze, transform or generate other programs. Ordinary programs work on data; meta-programs work on programs.“ [27]

D.h. Metaprogramme sind normale Programme, welche Sourcecode als Daten verarbeiten.

4.2 Geschichte

Der Grundstein für die Metaprogrammierung wurde in den 1940er Jahren gelegt als sich die Von-Neumann-Architektur, welche Code und Daten im selben Speicherbereich ablegt, durchsetzte [37]. Dadurch wurde die Entwicklung der ersten Metaprogramme in den 1950er Jahren ermöglicht: Compiler [43]. Diese wurden verwendet um höhere Sprachen in Assemblersprache zu übersetzen und waren für lange Zeit die einzige Form der Metaprogrammierung [43]. Erst 1966 wurde die Programmiersprache PL/1 vorgestellt, welche über einen Präprozessor verfügte

[37]. Dieser ermöglichte die Definition von einfachen Makros, welche die Programmierarbeit reduzieren konnten und vor allem in prozeduralen Sprachen wie C noch heute eingesetzt werden. Im Bereich der statischen Programmiersprachen wurde danach nur wenig Metaprogrammierung eingesetzt. Erst in den 1980er wurde für C++ nachträglich ein Template-System entwickelt, welches anschließend auch für die Metaprogrammierung verwendet wurde [4]. Aktuelle statische Sprachen wie Java oder C# bieten nur sehr eingeschränkte Möglichkeiten für Metaprogrammierung durch Reflection, jedoch gibt es eine Vielzahl an Werkzeugen oder Bibliotheken, welche diese Sprachen um Code-Generierung/Metaprogrammierung erweitern.

In den 1950er Jahren gab es eine alternative Entwicklung zum Compiler, welche die Idee der Metaprogrammierung an die Spitze trieb: Lisp. Diese Sprache folgt einem einfachen aber sehr mächtigen Prinzip: Daten und Code sind gleichwertig [31]. Dadurch ist es möglich die Sprache beliebig zu verändern: So wurde in den 1980er Jahren Common Lisp Object System (CLOS) entwickelt, welches das objektorientierte Paradigma ohne Änderung des Interpreters zur Sprache hinzufügte [30]. Später wurde CLOS durch das Meta Object Protocol (MOP) erweitert, welches das Objektsystem für den Entwickler offenlegte und einfacher anpassbar machte [26]. Diese Art von Metaprogrammierung ist heute in dynamischen Sprachen wie Ruby [34] und Python [2] sehr oft vertreten. Der Entwickler des MOP hat sich später von Lisp distanziert und hat begonnen ähnliche Konzepte in Form von aspektorientierter Programmierung für Java (AspectJ) zu entwickeln [25].

Zusammengefasst ist in der Geschichte der Metaprogrammierung eine Trennung von statischen und dynamischen Sprachen erkennbar. Während in dynamischen Sprachen Metaprogrammierung häufig bereits in der Sprache vorhanden ist, ist diese bei statischen Sprachen nur selten vertreten und wird erst durch zusätzliche Werkzeuge ermöglicht.

4.3 State-of-the-Art

Reflexion

Reflexion ermöglicht das Auslesen von Programmstrukturen zur Laufzeit, wie z.B. das Auflisten der Methoden einer Klasse. Manipulationen sind nur äußerst eingeschränkt bzw. gar nicht möglich.

Bekannte Vertreter: Java [19], C# [17]

Visitorbasierte Metaprogrammierung

Bei visitorbasierter Metaprogrammierung können Listener für bestimmte Methoden, Klassen, Annotationen definiert werden. Anschließend können verschiedene Elemente (meist dieselbe Klasse) manipuliert werden. Es können jedoch nicht mehrere Klassen zusammengehörig modifiziert werden.

Bekannte Vertreter: Spoon [33], Jasper [32]

Templatebasierte Metaprogrammierung

Bei templatebasierter Metaprogrammierung werden Templates erstellt, welche Sourcecode-Dateien mit speziellen Schlüsselworten sind. Diese Schlüsselworte werden später ausgewertet und erstellen eine spezialisierte Version der Sourcecode-Datei. Es kann jeweils nur eine Klasse zusammengehörig modifiziert werden.

Bekannte Vertreter: XDoclet [42], Apache Velocity [22]

Offene Compiler

Bei offenen Compilern können Klassen bestimmten Metaklassen zugeordnet werden. Diese Metaklassen können diese Klasse anschließend modifizieren. Es können jedoch nicht mehrere Klassen zusammengehörig modifiziert werden.

Bekannte Vertreter: OJ (OpenJava) [39], OpenC++ [11]

Bytecode-Manipulation

Hierbei handelt es sich um Java-Bibliotheken, welche kompilierte Java-Klassen (.class-Dateien) vor dem Classloading modifizieren oder erstellen können. Der Vorteil ist, dass auf diese Weise mehrere Klassen zusammengehörig modifiziert werden könnten. Jedoch hat die Manipulation von Bytecode zwei große Nachteile. Einerseits wird die Manipulation erst zur Laufzeit durchgeführt (schlechtere Performance) andererseits sind die Änderungen für den Entwickler nicht sichtbar (Bytecode). Somit können leicht unbemerkt Seiteneffekte auftreten, welche mit statischem Debugging nicht auffindbar sind.

Bekannte Vertreter: Javassist [12], ASM [28], BCEL [13]

Dynamische Metaprogrammierung

Dynamische Metaprogrammierung ist oft fester Bestandteil dynamischer Sprachen und sehr mächtig. Sie ist ähnlich der Reflexion, jedoch mit dem Unterschied, dass die ausgelesenen Programmstrukturen auch vielfältig modifiziert werden können. So können z.B. Klassenmethoden während der Laufzeit geändert oder hinzugefügt werden. Jedoch können aufgrund der enormen Manipulationsmöglichkeiten sehr leicht unerwartete Seiteneffekte auftreten, welche nur durch Laufzeit-Tests gefunden werden können.

Bekannte Vertreter: Ruby [34], Phyton [2]

4.4 Probleme der Metaprogrammierung

Die Recherche nach Metaprogrammierung hat den Eindruck hinterlassen, dass sie nur von Experten in besonderen Anwendungsfällen eingesetzt wird. Daraufhin wurde versucht die Ursache

zu finden. Als ersten Bezugspunkt wurde mit Hilfe von Google Trends versucht einen Hinweis auf die Verbreitung der unterschiedlichen Metaprammierungsarten zu finden.

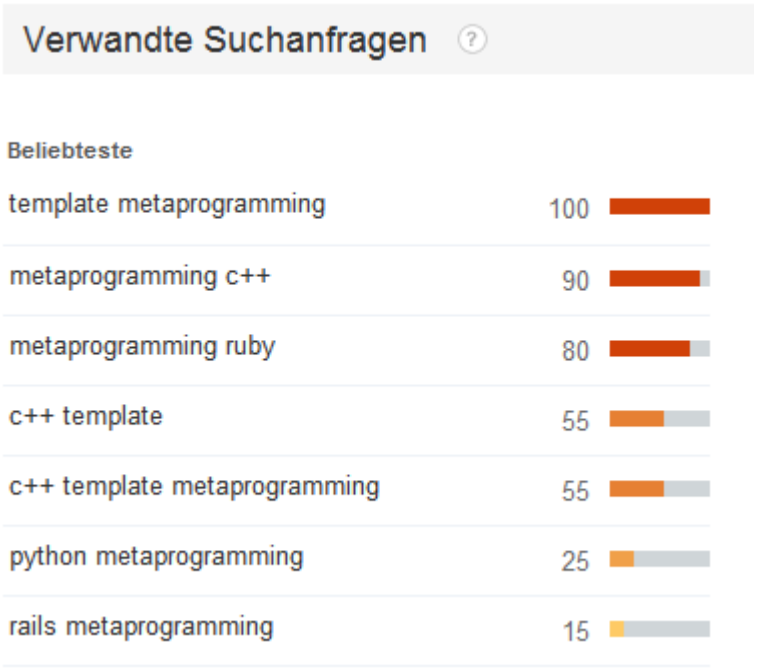

Abbildung 4.1: Verwandte Suchanfragen zum Begriff „metaprogramming“ [40]

Die Abbildung 4.1 zeigt, dass C++ Template-Metaprogrammierung mit Abstand am meisten gesucht wird (Platz 1,2,4,5). An zweiter Stelle stehen Vertreter der dynamischen Metaprogrammierung (Ruby (-Rails), Python). Die Abwesenheit von Sprachen wie Java oder C# liegt wahrscheinlich daran, dass in diesem Bereich vor allem Frameworks eingesetzt werden, welche Metaprogrammierung in Form von Bytecode-Manipulation nutzen und somit vor den ProgrammiererInnen verstecken. Das führt zu dem Schluss, dass Metaprogrammierung in diesen Sprachen kein verbreiterter Begriff ist.

Problem 1: Metaprogrammierung ist kompliziert

Für dieses Problem ist vermutlich die hohe Verbreitung von C++ Template-Metaprogrammierung verantwortlich, welche als eines der kompliziertesten Metaprogrammierungssysteme angesehen werden kann. Dies liegt daran ist, dass das Template-System ursprünglich zur Erstellung von generischen Klassen entwickelt wurde und erst später für die Metaprogrammierung zweckentfremdet wurde. Dadurch besitzt die C++ Template-Metaprogrammierung nicht nur eine eigene Syntax, sondern folgt sogar einem anderen Programmierparadigma, nämlich der funktionalen

Programmierung [4]. Die beste Möglichkeit diesen Umstand aufzuheben ist, dass Metaprogrammierung in derselben Sprache durchgeführt wird.

Problem 2: Metaprogrammierung ist gefährlich
Dafür verantwortlich ist vermutlich die hohe Verbreitung dynamischer Metaprogrammierung, welche aufgrund der enormen Manipulationsmöglichkeiten zur Laufzeit sehr leicht zu unerwarteten Seiteneffekten führen kann. Bereits ein einfacher Methodenaufruf kann dazu führen, dass die Methodenimplementierung des übergebenen Objektes in der Methode verändert wird. Wenn dies unabsichtlich durch Namensgleichheit von Methoden passiert, kann dies zu äußerst schwierigen Fehlern führen. Debugging ist in diesem Fall nur eingeschränkt möglich, wodurch der Fehler durch Laufzeit-Tests gefunden werden muss. Die beste Möglichkeit um diese Gefahren zu reduzieren ist statische Codegenerierung, welche zu sichtbaren Ergebnissen führt und Debugging erleichtert.

4.5 Fazit

Zu Beginn wurde der Begriff Metaprogrammierung und dessen Geschichte näher beschrieben. Anschließend wurden aktuelle Ansätze der Metaprogrammierung vorgestellt und erklärt, warum sie nicht die Anforderungen erfüllen können. Danach wurden Probleme der Metaprogrammierung beschrieben und Lösungsansätze aufgezeigt, welche als nichtfunktionale Anforderungen in den Anforderungskatalog aufgenommen werden.

Anforderungskatalog

Nichtfunktionale Anforderungen

Komplexitätsreduktion
- Programmier- und Metaprogrammierungssprache sollten sich kaum unterscheiden.

Gefährlichkeitsreduktion
- Statische Codegenerierung führt zu sichtbaren Ergebnissen, macht Seiteneffekte auffindbar und erleichtet somit das Debugging.

KAPITEL 5

Analyse

In diesem Kapitel wird die Problemstellung anhand konkreter Beispiele näher beschrieben und anschließend analysiert, welche Funktionen benötigt werden um das konkrete Problem lösen zu können. Am Ende soll ein Anforderungskatalog für das zu entwickelnde Werkzeug entstehen. Die konkreten Probleme werden in der Programmiersprache Java beschrieben, kommen aber in den meisten anderen (statischen) objektorientierten Sprachen ebenfalls vor.

5.1 Design Pattern: Composite

Beschreibung

Ein *Composite*-Objekt hält Referenzen zu einer Menge von *Component*-Objekten, welche zur Laufzeit hinzugefügt und entfernt werden können. Das *Composite*-Objekt hat dasselbe Interface wie *Component*-Objekte, wodurch eine Hierarchie von *Component*-Objekten aufgebaut werden kann. Weiters leiten *Composite*-Objekte alle Methoden des *Component*-Interfaces an alle Child-Objekte weiter. Child-Klassen implementieren das *Component*-Interface.

Eine ausführlichere Beschreibung befindet sind im Abschnitt 2.2 „Composite-Pattern“ auf Seite 11.

Generierung einer Composite-Klasse

Das Graphic-Interface bildet den gemeinsamen Obertypen für *Child*- und *Composite*-Klassen.

```
public interface Graphic
{
    public void draw();
}
```

Die Klasse *GraphicComposite* wird mittels Annotation als *Composite*-Klasse für Untertypen des *Graphic*-Interface deklariert.

Es sollen die Methoden *add*, *remove*- und *get* und eine List-Variable für das *Composite*-Klasse generiert werden. Weiters soll für jede Methode des *Component*-Interfaces eine Weiterleitung zu den Child-Objekten generiert werden (*draw*-Methode).

```
@Composite("Graphic")
public class GraphicComposite implements Graphic {
    //************** begin of generated code ****************
    private List<Graphic> childrenOfGraphic = new ArrayList<Graphic>();

    public void addChild(Graphic element) {
        childrenOfGraphic.add(element);
    }
    public void removeChild(Graphic element) {
        childrenOfGraphic.remove(element);
    }
    public List<Graphic> getChilds() {
        return (List<Graphic>)java.util.Collections.unmodifiableList(childrenOfGraphic);
    }

    public void draw() {
        for(Graphic child : childrenOfGraphic)
            child.draw();
    }
    //************** end of generated code ****************
}
```

Anforderungen

Folgende Anforderungen sind notwendig um das beschriebene Beispiel generieren zu können:

- Lesen: Annotationen über Klassen (@*Composite*)
- Suchen: Klasse über Interface-Name (String *Graphic* von Annotation @*Composite*)
- Lesen: Methoden einer Klasse (alle Methoden von *Graphic*)
- Lesen: Name und Attribute eine Methode (für Delegationsmethode *draw*)
- Lesen: Typ und Name von Parametern (für Delegationsmethode *draw*)
- Erstellen: Beliebige Variablen (List)
- Erstellen: Beliebige Methoden (add, remove, get, Delegationmethoden)

5.2 Design Pattern: Command

Beschreibung

Eine *Command*-Klasse kapselt den Aufruf einer Methode in eine eigene Klasse. Die *execute*-Methode der *Command*-Klasse führt die ursprüngliche Methode auf des als Parameter übergebenen *CommandReceiver*-Objekt aus. Der Vorteil ist, dass dadurch Methodenaufrufe als Objekte behandelt werden können. Beispielsweise können *Command*-Objekte gepspeichert und später auf verschiedene *CommandReceiver*-Objekte ausgeführt werden.

Eine ausführlichere Beschreibung befindet sind im Abschnitt 2.2 „Command-Pattern“ auf Seite 15..

Deklaration eines *CommandReceiver*-Typs mit *Command*-Methoden

Das *Driveable*-Interface wird mittels Annotation als *CommandReceiver*-Typ deklariert, welches die *Command*-Klassen in das Package *commands* generieren soll. Weiters werden alle Methoden als *Command*-Methoden deklariert. D.h. für diese Methoden sollen *Command*-Klassen generiert werden.

```
@CommandReceiver("commands")
public interface Driveable {
   @Command
   public void setSpeed(int speed);

   @Command
   public void turnRight();

   @Command
   public void turnLeft();
}
```

Generierung der *Command*-Klassen

Es soll für jede mit *@Command* annotierte Methode eine eigene *Command*-Klasse generiert werden, welche das gemeinsame *Command*-Interface implementiert. Für jeden Parameter der *Command*-Methode soll eine Variable in die Command-Klasse generiert werden. Weiters soll eine *execute*-Methode generiert werden, welche eine Delegation auf das übergebene *CommandReceiver*-Objekt durchführt.

```
//************** begin of generated code *****************
public class SetSpeed implements DriveableCommand
   @Constructor
   private int speed;

   public void execute(Driveable driveable) {
      driveable.setSpeed(speed);
   }

   public SetSpeed(int speed) {
```

```
        this.speed = speed;
    }
}
//*************** end of generated code ****************
```

```
//************* begin of generated code ****************
public class TurnLeft implements DriveableCommand {

    public void execute(Driveable driveable) {
        driveable.turnLeft();
    }
}
//*************** end of generated code ****************
```

```
//************* begin of generated code ****************
public class TurnRight implements DriveableCommand {

    public void execute(Driveable driveable) {
        driveable.turnRight();
    }
}
//*************** end of generated code ****************
```

Anforderungen

Folgende Anforderungen sind notwendig um das beschriebene Beispiel generieren zu können:

- Lesen: Annotationen über Klassen (@*CommandReceiver*)
- Lesen: Methoden einer Klasse (*CommandReceiver*-Klasse)
- Lesen: Annotationen über Methoden (@*Command*)
- Lesen: Typ und Name einer Methode (*CommandMethoden*: *setSpeed*, *turnLeft* und *turnRight*
- Lesen: Parameter einer Methode (*setSpeed*)
- Erstellen: Beliebige Klassen (*Command*-Klassen: *SetSpeed*, *TurnLeft* und *TurnRight*)

5.3 Cross-Cutting-Concerns: Tracing

Beschreibung

Eine *Aspect*-Klasse kann *Listener*-Methoden definieren, welche beim Methodenaufruf von Instanzen ebenfalls aufgerufen werden. Die zu überwachenden Methoden werden mittels regulärer Ausdrücke von Typ- und Methodennamen definiert. Weiters kann festgelegt werden, ob sie beim Eintritt oder Austritt der Methode aufgerufen werden sollen. *Aspect*-Klassen können verwendet werden um Cross-Cutting-Concerns wie Logging, Tracing und Profiling zu realisieren.

Eine ausführlichere Beschreibung befindet sind im Abschnitt 2.4 „Aspektorientierte Programmierung“ auf Seite 20.

Deklaration einer Aspect-Klasse

Die *Main*-Klasse wurde als *Aspect* deklariert und soll dadurch *Listener*-Methoden definieren können. Die *traceBefore*-Methode soll am Beginn jeder Methode, welche mit *say* beginnt, aufgerufen werden. Die *traceAfter*-Methode soll am Ende jeder Methode, welche mit *say* beginnt, aufgerufen werden.

```
@Aspect
public class Main {
    public static void main(String[] args) {
        new ClassA().sayHelloB();
        new ClassB().sayHelloA();
    }

    @Before(method=".*say.*", type=".*")
    public static void traceBefore(Object a, String method) {
        System.out.println("Before: " + method);
    }

    @After(method=".*say.*", type=".*")
    public static void traceAfter(Object a, String method) {
        System.out.println("After: " + method);
    }
}
```

Generierung des Aufrufcodes

Es sollen für alle Methoden, welche in das Muster der *Listener*-Methode passen, einen Aufrufcode generiert werden. Für die *@Before*-Methode soll eine einfache Aufrufzeile am Beginn der Methode eingefügt werden. Für die *@After*-Methode muss ein Trick angewendet werden um vorzeitige Returns behandeln zu können: Der gesamte Methodeninhalt soll in einen try-Block und der Aufrufcode in einen finally-Block gelegt werden.

```
public class ClassA {
    public void sayHelloB(){
        //************** begin of generated code *****************
        try { //from after
```

```
        Main.traceBefore(this,"sayHelloB"); //from before
        //*************** end of generated code *****************

        System.out.println("Hello B!");

        //*************** begin of generated code *****************
        } finally {
           Main.traceAfter(this,"sayHelloB");
        }
        //*************** end of generated code *****************
    }
}
```

Anforderungen

Folgende Anforderungen sind notwendig um das beschriebene Beispiel generieren zu können:

- Lesen: Annotationen über Klassen (*@Aspect*)
- Lesen: Methoden einer Klasse (*Aspect*-Klasse)
- Lesen: Annotationen über Methoden (*@Before* und *@After*)
- Lesen: Typ und Name einer Methode (*Listener*-Methoden: *traceBefore* und *traceAfter*
- Suchen: Alle Klassen
- Lesen: Type und Name einer Klasse (zu überwachende Klasse)
- Lesen: Typ und Name einer Methode (zu überwachende Methode)
- Bearbeiten: Code am Anfang einer Methode hinzufügen (zu überwachende Methode)
- Bearbeiten: Code am Ende einer Methode hinzufügen (zu überwachende Methode)

5.4 Fazit

Die Problemstellung wurde anhand konkreter Beispiele näher beschrieben. Es wurde gezeigt, wie Annotationen verwendet werden können um bestimmte Anforderungen deklarativ zu programmieren um sie anschließend generieren zu lassen. Weiters wurde analysiert, welche Funktionen benötigt werden um das konkrete Problem lösen zu können. Im folgenden Anforderungskatalog wurden alle gesammelten Anforderungen zusammengefasst und gruppiert.

Anforderungskatalog

Funktionale Anforderungen

Das gesuchte Werkzeug muss folgende Funktionen in beliebiger Reihenfolge und Anzahl durchführen können:

Suchen
- Alle Klassen auflisten
- Klasse über Subtypenname finden

Lesen
- Annotationen über Klassen, Variablen, Methoden auslesen
- Variablen und Methoden einer Klasse auslesen
- Name und Parameter einer Methode auslesen
- Typ und Name von Variablen oder Parametern auslesen

Erstellen
- Klassen oder Interfaces erstellen

Bearbeiten
- Variablen oder Methoden zu Klassen hinzufügen
- Obertypen oder Interfaces zu Klassen hinzufügen
- Code am Anfang oder Ende einer Methode hinzufügen

KAPITEL 6

Lösung

Dieses Kapitel setzt sich noch einmal konkret mit der Problemstellung auseinander und beschreibt anschließend die Schlussfolgerungen zur Lösungsfindung. Danach folgt eine detaillierte Beschreibung der konkreten Lösung.

6.1 Problemstellung

Das Ziel dieser Diplomarbeit ist die Entwicklung eines Werkzeugs, mit dessen Hilfe Muster in der Programmierung (Boilerplate-Code, Design Patterns, ...) automatisiert werden können. In den vorherigen Kapiteln wurde mittels Problemanalyse und Recherche-Arbeit folgender Anforderungskatalog erstellt:

Anforderungskatalog

Funktionale Anforderungen

(siehe Abschnitt 5.4 „Anforderungskatalog“ auf Seite 38)

Nichtfunktionale Anforderungen

(siehe Abschnitt 4.5 „Anforderungskatalog“ auf Seite 29)

6.2 Lösungsfindung

Zum Anforderungskatalog soll nun mittels logischer Schlussfolgerungen ein passender Lösungsansatz gefunden werden.

Schlussfolgerungen

1. Die funktionalen Anforderungen sind sehr anspruchsvoll, besonders, das die Funktionen in beliebiger Reihenfolge und Anzahl durchgeführt werden sollen. Dadurch können unflexible Ansätze wie template- und visitorbasierte Metaprogrammierung bereits im Vorhinein ausgeschlossen werden. Am passendsten erscheint eine Lösung wie sie bei Bytecode-Manipulation oder dynamischer Metaprogrammierung verwendet wird: Eine Bibliothek, welche Sourcecode-Dateien wie Daten liest und Funktionen zur Suche und Bearbeitung bereitstellt.

2. Die erste nichtfunktionale Anforderung „Programmier- und Metaprogrammierungssprache sollten sich kaum unterscheiden“ wird aufgrund der Umsetzung als Bibliothek bereits erfüllt. D.h. die Metaprogramme werden in derselben Programmiersprache mit Hilfe der Bibliothek geschrieben.

3. Die zweite nichtfunktionale Anforderung „Statische Codegenerierung“ steht in keinem Widerspruch zu den bisher getroffenen Annahmen. Eine Bibliothek kann sowohl Bytecode- als auch Sourcecode generieren.

Somit konnten alle Anforderungen in einer Lösung vereint werden:

> Eine Bibliothek in derselben Programmiersprache, welche Sourcecode-Dateien wie normale Daten lesen und manipulieren kann.

Aus diesem Ansatz wurde ein konkreter Lösungsweg entwickelt, welcher im folgenden Abschnitt genau erklärt wird.

6.3 Lösungsbeschreibung

Bezeichnung

Um die Lösung im Text besser referenzieren zu können, musste ein Name gefunden werden. Die Wahl auf den Namen *Code-Monkey* (CM) hatte zwei Gründe:

- Da Metaprogrammierung nicht den besten Ruf hat (siehe Abschnitt 4.4 „Probleme der Metaprogrammierung“ auf Seite 27), wird auf einen expliziten Bezug dieses Begriffs im Namen verzichtet.
- Es ist eine treffende Bezeichnung für die Funktion des Werkzeugs: Ein *Code-Monkey* ist eine ProgrammiererIn, welche triviale und repetitive Codierungsarbeit durchführen muss [40].

Beschreibung

Code-Monkey ist ein Werkzeug, welches Programmierung durch Automatisierung von Software-Mustern unterstützen soll. Dazu kann der Entwickler Metaprogramme in derselben Programmiersprache entwickeln, welche mit Hilfe einer Bibliothek Sourcecode wie normale Daten lesen und manipulieren kann. Dadurch können beliebige Codierungsmuster beschrieben als auch

anschließend automatisiert generiert werden.

Funktionsprinzip

Kopieren und Einfügen des Sourcecodes

Der erste Schritt umfasst die Vorbereitung der Arbeitsumgebung. Metaprogramme dürfen nicht am Sourcecode direkt eingreifen, da sich dadurch die Ausgangsbedingungen ändern würden. Aus diesem Grund dürfen sie immer nur eine Kopie bearbeiten. Daher wird vor jedem Aufruf der Metaprogramme eine Kopie der Sourcecode-Basis erstellt, welche von nun an als Sourcecode-Generat bezeichnet wird. (siehe Abb. 6.1 „Kopieren und Einfügen des Sourcecodes“ auf Seite 41).

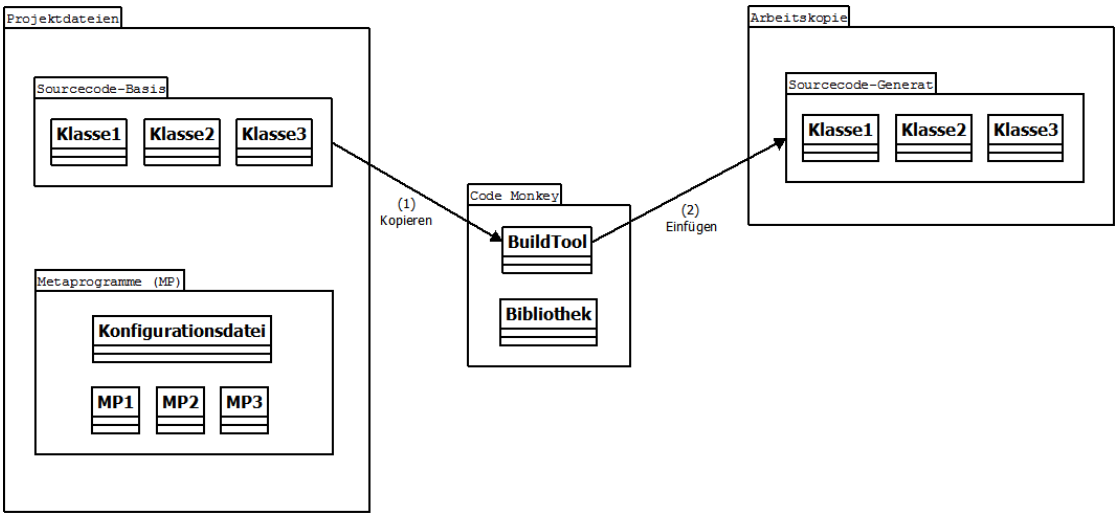

Abbildung 6.1: Kopieren und Einfügen des Sourcecodes

Lesen und Ausführen der Metaprogramme

Der zweite Schritt ist das Auslesen der Konfigurationsdatei, welche die Reihenfolge der auszuführenden Metaprogramme enthält. Diese wird im Anschluss sequentiell abgearbeitet, wobei alle Metaprogramme das Sourcecode-Generat bearbeiten. Der Output eines Metaprogrammes ist der Input des nächsten Metaprogrammes, wodurch deren Ergebnisse kombiniert werden können. (siehe Abb. 6.2 „Lesen und Ausführen der Metaprogramme“ auf Seite 42).

6.4 Fazit

Es wurde eine konkrete Lösung vorgestellt, welche durch Schlussfolgerungen aus den Anforderungen der Problemanalyse und Recherche-Arbeit erarbeitet wurde. Im Folgenden werden nun

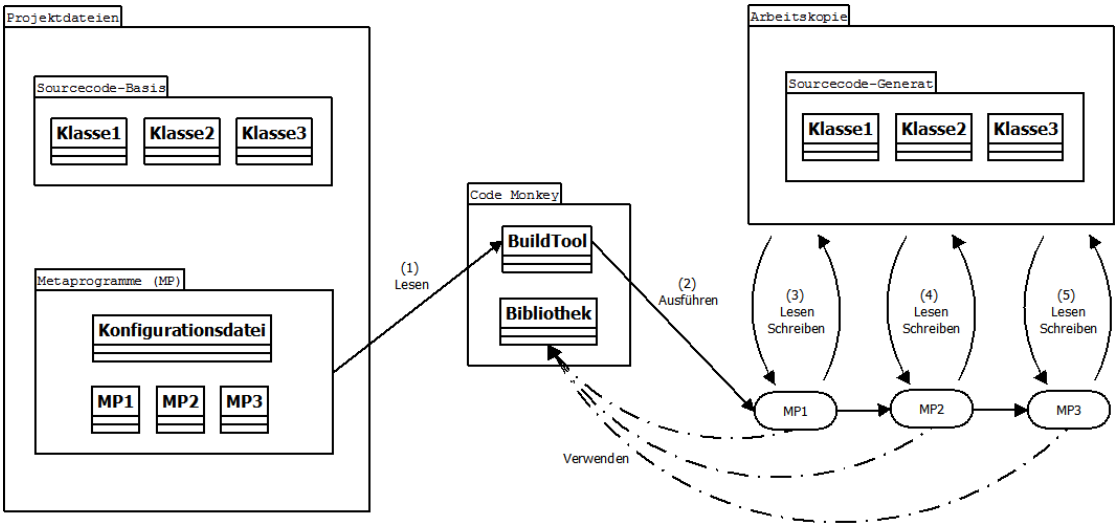

Abbildung 6.2: Lesen und Ausführen der Metaprogramme

jene Punkte angeführt, die sich von den alternativen Lösungsansätzen unterscheiden.

Mächtigkeit
CM-Metaprogramme verzichten auf Einschränkungen von template- oder visitorbasierten Ansätzen und nutzen eine Bibliothek, welche die Manipulationsfunktionen in beliebiger Anzahl und Reihenfolge durchführen können.

Vertraute Arbeitsumgebung
CM-Metaprogramme werden in der gleichen Programmiersprache geschrieben, somit können alle vertrauten Funktionen und Methoden benutzt werden.

Sichtbare Ergebnisse
Die CM-Bibliothek ermöglicht die Bearbeitung von Sourcecode-Dateien wie normale Daten, daher sind die Ergebnisse in den manipulierten Dateien klar ersichtlich. Die „Zauberei“ von ähnlichen Ansätzen wie Bytecode-Manipulation oder dynamischer Metaprogrammierung, wo die Änderungen nur an den Effekten zur Laufzeit ersichtlich ist, wird vermieden. Dadurch kann das Funktionsprinzip einfacher verstanden und debuggt werden.

Sprachunabhängig
Das Funktionsprinzip von CM kann bei fast allen objektorientierten Programmiersprachen angewendet werden. Von Vorteil sind Sprachen, welche Annotationen unterstützen. Jedoch können alternativ interpretierte Kommentare verwendet werden (vgl. XDoclet [42]).

KAPITEL 7

Umsetzung

In diesem Kapitel wird eine konkrete Umsetzung der zuvor beschriebenen Lösung dargestellt.

7.1 Beschreibung

Java-Code-Monkey (JCM) ist eine konkrete Umsetzung des zuvor beschriebenen Lösungsansatzes *Code-Monkey* für die Programmiersprache Java. JCM ermöglicht die Erstellung und Ausführung von Metaprogrammen.

Ein JCM-Metaprogramm kann folgende Aktionen beliebig durchführen:

- Projekt-Dateien suchen und auflisten
  - Java- und XML-Dateien
- Java-Elemente lesen
  - Klassen, Methoden, Variablen, Annotationen, ...
  - Java-Elemente liegen als Metaobjekte vor (ähnlich Reflection-API).
- Beliebige Dateien lesen
  - z.B.: XML-Dateien
- Java-Elemente manipulieren
  - Klassen um Obertypen, Interfaces, Variablen, Methoden, Annotationen erweitern
  - Interfaces um Obertypen, Methoden, Annotationen erweitern
  - Methoden um Annotationen, Exceptions, Code (davor, danach) erweitern
  - Variablen um Annotationen erweitern
- Java-Dateien erstellen
  - Klassen
  - Interfaces

7.2 Technologie

JCM wurde als Plugin für die Eclipse-IDE umgesetzt. Es ist eine Kombination aus Eclipse-Plugin und Java-API:

CM-BuildTool = Eclipse-Plugin

Das Eclipse-Plugin bereitet die Projektstruktur vor und führt anschließend die Metaprogramme in der definierten Reihenfolge aus.
Die Organisation eines JCM-Eclipse-Projekts erfolgt durch vier Ordner:

- *src*- und *xml*-Ordner: Input der Metaprogramme (Java- bzw. XML-Dateien)
- *meta*-Ordner: Metaprogramme und dessen Konfigurationsdatei
- *gen*-Ordner: Output der Metaprogramme (generierter, lauffähiger Sourcecode)

Das Plugin wird mittels Button gestartet und führt folgende Aktionen durch:

1. Kopieren des Inhalts des *src*-Ordners.
2. Löschen des *gen*-Ordners und anschließendes Einfügen des kopierten Inhalts.
3. Auslesen der Reihenfolge der Metaprogramme aus der *MetaConfig.xml*-Datei.
4. Für jeden Metaprogramm-Eintrag:
    - Dynamisches Laden des Metaprogramms durch Classloader
    - Erstellen einer Metaprogramm-Instanz
    - Aufruf der *run*-Methode und Übergabe der Java-API-Instanz
    - Speichern der Änderungen

CM-Bibliothek = Java-API

Die API wird von dem Metaprogrammen zum Lesen und Schreiben von Source-Code eingesetzt. Es ist ein Wrapper der Module *JavaModel* und *AST* der Eclipse-API *Java Development Tools*.

- Modul *JavaModel*: Auslesen von Java-Elementen, Erstellen von Java-Dateien, Hinzufügen von Methoden und Variablen („grobgranular“)
- Modul *AST*: Hinzufügen von Annotationen, Obertypen, Code,... („feingranular“)

7.3 Beispiele

Konfigurationsdatei: MetaConfig.xml

Die Konfigurationsdatei enthält eine Liste von Metaprogrammen, welche sequentiell ausgeführt werden. Die Reihenfolge ist für das Endergebnis relevant. Zum Beispiel verwendet das *Strategy*-Pattern eine *@Constructor*-Variable um das Setzten des konkreten *Strategy*-Objektes zu realisieren. Das Metaprogramm, welches *@Constructor*-Annotationen ausliest, darf erst nach

dem *Strategy*-Metaprogramm ausgeführt werden.

Dadurch ergibt sich folgende Reihenfolge:

MetaStrategy > MetaConstructor

Aufgrund dieser Abhängigkeiten ergeben sich Gruppen von Metaprogrammen, welche eine definierte Reihenfolge einhalten müssen. Diese Gruppen müssen in der Konfigurationsdatei in Einklang gebracht werden. Falls es hierbei zu zyklischen Abhängigkeiten kommt, hilft oft die Teilung eines Metaprogrammes um eine sequentielle Abhängigkeit zu schaffen.

```
<meta>
   <metaprogrammer class="meta.patterns.multimethod.programmer.MetaMultimethod"/>
   <metaprogrammer class="meta.patterns.publisher.programmer.MetaPublisher"/>
   <metaprogrammer class="meta.patterns.logger.programmer.MetaLogger"/>
   <metaprogrammer class="meta.patterns.command.programmer.MetaCommand"/>
   <metaprogrammer class="meta.patterns.visitor.programmer.MetaVisitor"/>
   <metaprogrammer class="meta.patterns.composite.programmer.MetaComposite"/>
   <metaprogrammer class="meta.patterns.strategy.programmer.MetaStrategy"/>

   <metaprogrammer class="meta.basics.getset.programmer.MetaGetSet"/>
   <metaprogrammer class="meta.basics.access.programmer.MetaListAccess"/>
   <metaprogrammer class="meta.basics.constructor.programmer.MetaConstructor"/>

   <metaprogrammer class="meta.aop.programmer.MetaAspect"/>

</meta>
```

Metaprogramm: MetaExample.java

Das Metaprogramm gibt den Namen aller gelesenen Java-Dateien aus. Es ist eine normale Java-Klasse, welche das *MetaProgrammer*-Interface implementiert. Dieses enthält die Methode *run* mit dem Parameter des Typs *MpManager*, welche vom JCM-Plugin aufgerufen wird. Der *MpManager* ist ein Facade-Objekt, welches Zugriff auf die Funktionen der JCM-API ermöglicht.

```
public class MetaExample implements MetaProgrammer
{
   public void run(MpManager mp) {
      for(MpClass clazz : mp.getClasses())
      {
         System.out.println(clazz.getName()));
      }
   }
}
```

7.4 Fazit

Es wurde die konkrete Umsetzung *Java-Code-Monkey* des zuvor beschriebenen Lösungansatzes *Code-Monkey* entwickelt und beschrieben. Sie wurde für die Programmiersprache Java auf Basis

von Eclipse entwickelt und ermöglicht die Erstellung von Metaprogrammen. In den folgenden Kapiteln wird dieses Werkzeug verwendet um verschiedene Codierungsmuster zu automatisieren.

KAPITEL 8

Metaprogramme für Boilerplate-Code

Das Kapitel beschreibt Metaprogramme, welche zur automatisierten Programmierung von Boilerplate-Code verwendet werden. Als Boilerplate-Code werden Codefragmente bezeichnet, welche für eine relativ einfache Aufgabe verhältnismässig viel trivialen Code benötigen, z.B. Getter und Setter. Eine ausführlichere Beschreibung befindet sind im Abschnitt 2.1 „Boilerplate-Code“ auf Seite 6.

Der Hauptzweck dieser Metaprogramme ist die Reduzierung der Komplexität von aufbauenden Metaprogrammen. Viele benötigen Boilerplate-Code für die Umsetzung von Variablen- und Listenzugriffen, sowie einfache Konstruktoren.

Diese Metaprogramme können auch direkt für die Programmierung selbst verwendet werden. Jedoch sollte beachtet werden, dass z.B. Getter und Setter generell zugunsten von Klassenkopplung und Datenkapselung vermieden werden sollten. Aus diesem Grund wurde auch auf komplexere Sonderfälle (z.B. MultiSetter), welche aufbauende Metaprogramme nicht benötigen, verzichtet.

8.1 GetSet

Funktion

Das *GetSet* -Metaprogramm ermöglicht eine deklarative Programmierung von Getter und Setter.

Beschreibung

Getter und Setter werden in Java verwendet um Variblenzugriffe über Methoden zu kapseln. Dies wird z.B. vom Strategy-Pattern benötigt um das konkrete Strategy-Objekt auswechselbar zu machen.

Verwendung

- Deklaration eines Getters
  - Klassenvariable mit @*Get* annotieren.
- Deklaration eines Setters
  - Klassenvariable mit @*Set* annotieren.
- Deklaration von Getter und Setter
  - Klassenvariable mit @*GetSet* annotieren.

Beispiel vor Ausführung der Metaprogramme (Sourcecode-Basis)

Deklaration von Getter und Setter

Variable A: Nur Lesen, Variable B: Nur Schreiben, Variable C: Lesen und Schreiben

```
public class ClassA {
   @Get
   private int a;

   @Set
   private int b;

   @GetSet
   private int c;
}
```

Beispiel nach Ausführung der Metaprogramme (Sourcecode-Generat)

Generierung von Getter und Setter

Das Metaprogramm hat für Variable A einen Getter, für Variable B einen Setter und für Variable C Getter und Setter generiert.

```
public class ClassA
{
   @Get
   private int a;

   @Set
```

```
    private int b;

    @GetSet
    private int c;

    //************** begin of generated code *****************
    public int getA(){
        return a;
    }

    public void setB(int b){
        this.b = b;
    }

    public int getC(){
        return c;
    }
    public void setC(int c){
        this.c = c;
    }
    //**************** end of generated code *****************
}
```

Sourcecode des JCM-Metaprogramms

Das *GetSet*-Metaprogramm liest *@Get*-,*@Set*- und *@GetSet*-Annotationen aus und generiert daraus Getter- und Setter-Methoden.

```
public class MetaGetSet implements MetaProgrammer {
    public void run(MpManager mp) {
        for(MpClass clazz : mp.getClasses()) {
            String annotationPath = "meta.basics.getset.annotations.";

            for(MpField f : clazz.getFields()) {
                String type = f.getType().getName();
                String name = f.getName();

                //if field has @Get oder @GetSet => generate getter
                if(f.getAnnotation(annotationPath + "Get") != null || f.getAnnotation(annotationPath + "
                        GetSet") != null) {
                    clazz.createMethod(mp.createCode()
                        .line("public " + type + " get" + MpString.firstToUpper(name) + "()")
                        .startBlock().line("return " + name + ";").endBlock());
                }

                //if field has @Set or @GetSet => generate setter
                if(f.getAnnotation(annotationPath + "Set") != null || f.getAnnotation(annotationPath + "
                        GetSet") != null ) {
                    clazz.createMethod(mp.createCode()
                            .line("public void set" + MpString.firstToUpper(name) + "(" + type + " " + name + ")")
                            .startBlock().line("this." + name + " = " + name + ";").endBlock());
                }
} } } }
```

Evaluierung

Der Zeitbedarf des JCM-Metaprogramms bei 80 Klassen/Interfaces in der Sourcecode-Basis liegt bei ca. **421** Millisekunden.

Der Anteil des generierten Codes wird durch die Gegenüberstellung der Zeichenanzahl vor und nach der Generierung (ohne Kommentare) aller Java-Dateien (Ausnahme: Main.java) berechnet.

Anzahl Zeichen vor Generierung	86
Anzahl Zeichen nach Generierung	232
Anteil generierter Code	62,9%

Fazit

- Reduziert die Komplexität von Metaprogrammen, welche Getter und Setter benötigen.
- Deklarative Programmierung verbessert Überblick und Verständlichkeit.

8.2 Constructor

Funktion

Das *Constructor*-Metaprogramm ermöglicht eine deklarative Programmierung von einfachen Constructor-Methoden.

Beschreibung

Einfache Constructor-Methoden schreiben jeden übergebenen Parameter direkt in eine Variable des Objekts. Diese Variablen werden als @*Constructor*-Felder deklariert.

Verwendung

- Deklaration einer *Constructor*-Methode
  - Ein oder mehrere Klassenvariable mit @*Constructor* annotieren.
  - Sie stellen die Parameter der einfachen *Constructor*-Methode dar.

Beispiel vor Ausführung der Metaprogramme (Sourcecode-Basis)

Deklaration einer *Constructor*-Methode
Die Variablen A und B bilden gemeinsam eine einfache *Constructor*-Methode.

```
public class ClassA
{
   @Constructor
   private int a;

   @Constructor
   private int b;
}
```

Beispiel nach Ausführung der Metaprogramme (Sourcecode-Generat)

Generierung einer *Constructor*-Methode
Das Metaprogramm hat für alle annotierten Variablen einen gemeinsamen einfachen Constructor definiert, welcher die übergebenen Parameter direkt in die Klassenvariablen überträgt.

```
public class ClassA
{
   @Constructor
   private int a;

   @Constructor
   private int b;

   //************** begin of generated code *****************
   public ClassA(int a, int b)
   {
      this.a = a;
      this.b = b;
```

```
    }
    //*************** end of generated code ****************
}
```

Sourcecode des JCM-Metaprogramms

Das *Constructor*-Metaprogramm liest *@Constructor*-Annotationen aus und generiert daraus einfache Constructor-Methoden.

```
public class MetaConstructor implements MetaProgrammer {
    public void run(MpManager mp) {
        for(MpClass clazz : mp.getClasses()) {

            //save all fields with @Constructor in array
            MpParameterArray parameters = mp.createParameterArray();
            for(MpField f : clazz.getFieldsByAnnotation("meta.basics.constructor.annotations.Constructor")) {
                parameters.addParameter(f);
            }

            if(parameters.size() > 0) {
                //create constructor-method
                MpCode constructorCode = mp.createCode();
                constructorCode.startMethod("public " + clazz.getName() + "(" + parameters.
                        codeOfParameterTypeAndNameList() + ")");
                for(MpParameter p : parameters.getParameters())
                    constructorCode.line("this." + p.getName() + " = " + p.getName() + ";");
                constructorCode.endMethod();
                clazz.createMethod(constructorCode);
            }
} } }
```

Evaluierung

Der Zeitbedarf des JCM-Metaprogramms bei 80 Klassen/Interfaces in der Sourcecode-Basis liegt bei ca. **326** Millisekunden.

Der Anteil des generierten Codes wird durch die Gegenüberstellung der Zeichenanzahl vor und nach der Generierung (ohne Kommentare) aller Java-Dateien (Ausnahme: Main.java) berechnet.

Anzahl Zeichen vor Generierung	78
Anzahl Zeichen nach Generierung	136
Anteil generierter Code	42,6%

Fazit

- Reduziert die Komplexität von Metaprogrammen, welche darauf aufbauen.
- Deklarative Programmierung verbessert Überblick und Verständlichkeit.

8.3 ListAccess

Funktion

Das *ListAccess*-Metaprogramm ermöglicht eine deklarative Programmierung von Zugriffsmethoden für List-Variablen.

Beschreibung

Oft müssen List-Variablen von anderen Objekten verändert werden. Z.B. Hinzufügen und Entfernen von Observern zu Publishern. Um die Variable nicht direkt offenlegen zu müssen, werden *add-*, *remove-* und *get*-Methoden für den Zugriff verwendet.

Verwendung

- Deklaration einer *ListAccess*-Variable
  - Ein oder mehrere List-Variablen mit *@ListAccess(type=<String>,name=<String>)* annotieren.
  - Der Parameter *type* ist der vollständige Name der Element-Klasse.
  - Der Parameter *name* ist die Bezeichnung des Elements. (z.B.: Parent)

Beispiel vor Ausführung der Metaprogramme (Sourcecode-Basis)

Deklaration einer *ListAccess*-Variable

Für die List-Variable *observers* sollen Zugriffsmethoden erstellt werden. Der Typ der Elemente ist *Observer* und die Bezeichnung *EventObserver*.

```
public class Publisher {
   @ListAccess(type="Observer",name="EventObserver")
   private List observers = new LinkedList();

}
```

Beispiel nach Ausführung der Metaprogramme (Sourcecode-Generat)

Generierung der Zugriffsmethoden

Das Metaprogramm generiert für die annotierte List-Variable die *add-*, *remove-* und *get*-Zugriffsmethoden

```
public class Publisher
{
   @ListAccess(type="Observer",name="EventObserver")
   private List observers = new LinkedList();

   //************** begin of generated code ****************
   public void addEventObserver(Observer element) {
      observers.add(element);
   }
```

```
    public void removeEventObserver(Observer element) {
        observers.remove(element);
    }

    public List<Observer> getEventObservers() {
        return (List<Observer>)java.util.Collections.unmodifiableList(observers);
    }
    //*************** end of generated code ****************
}
```

Sourcecode des JCM-Metaprogramms

Das *ListAccess*-Metaprogramm liest *@ListAccess*-Annotationen aus und generiert daraus Zugriffsmethoden für die List-Variable.

```
public class MetaListAccess implements MetaProgrammer {
    public void run(MpManager mp) {
        for(MpClass clazz : mp.getClasses()) {
            for(MpField field : clazz.getFields()) {
                //for each field with @ListAccess-annotation
                MpAnnotation listAccess = field.getAnnotation("meta.basics.access.annotations.ListAccess");
                if(listAccess != null) {
                    String type = listAccess.getString("type");
                    String name = listAccess.getString("name");
                    String var = field.getName();
                    clazz.createImport(type);

                    //create add-method
                    clazz.createMethod(mp.createCode()
                        .line("public void add" + MpString.firstToUpper(name) + "(" + type + " element)")
                        .startBlock().line(var + ".add(element);").endBlock());
                    //create remove-method
                    clazz.createMethod(mp.createCode()
                        .line("public void remove" + MpString.firstToUpper(name) + "(" + type + " element)")
                        .startBlock().line(var + ".remove(element);").endBlock());
                    //create getter-method, which return unmodifiable list
                    clazz.createMethod(mp.createCode()
                        .line("public List<" + type + "> get" + MpString.firstToUpper(name) + "s()")
                        .startBlock().line("return (List<" + type + ">)java.util.Collections.unmodifiableList("+var+");").
                            endBlock());
} } } } }
```

Evaluierung

Der Zeitbedarf des JCM-Metaprogramms bei 80 Klassen/Interfaces in der Sourcecode-Basis liegt bei ca. **633** Millisekunden.

Der Anteil des generierten Codes wird durch die Gegenüberstellung der Zeichenanzahl vor und nach der Generierung (ohne Kommentare) aller Java-Dateien (Ausnahme: Main.java) berechnet.

Anzahl Zeichen vor Generierung	118
Anzahl Zeichen nach Generierung	396
Anteil generierter Code	70,2%

Fazit

- Reduziert die Komplexität von Metaprogrammen, welche darauf aufbauen.
- Deklarative Programmierung verbessert Überblick und Verständlichkeit.

KAPITEL 9

Metaprogramme für Design-Patterns

Das Kapitel beschreibt Metaprogramme, welche zur automatisierten Programmierung von Design-Patterns verwendet werden. Der Großteil der beschriebenen Design-Patterns stammt aus dem Buch „Design Patterns: Elements of Reusable Object-oriented Software“ [21].

Die Auswahl der umzusetzenden Design-Patterns erfolgte stichprobenartig, wobei oft verwendete Muster bevorzugt wurden. Das Ziel war einen statistischen Überblick zu erhalten, wie viele Design-Patterns sich prinzipiell gut oder schlecht automatisieren lassen.

9.1 Publisher

Funktion

Das *Publisher*-Metaprogramm ermöglicht eine deklarative Programmierung des *Observer*-Patterns.

Beschreibung

Ein *Publisher*-Objekt hält Referenzen zu einer Menge von *Subscriber*-Objekten, welche zur Laufzeit hinzugefügt und entfernt werden können. *Publisher*-Objekte können Events auslösen, welche an die registrierten *Subscriber*-Objekte weitergeleitet werden.

Eine ausführlichere Beschreibung befindet sind im Abschnitt 2.2 „Observer-Pattern“ auf Seite 7.

Verwendung

- Deklaration einer oder mehrerer *Subscriber*-Typen
  - Methoden von Klassen oder Interfaces mit *@Subscribe* annotieren.
  - Klassen oder Interfaces werden dadurch implizit zu *Subscriber*-Typen
- Deklaration eines *Publisher*-Typen
  - Interface oder Klasse mit *@Publisher(<String[]>)* annotieren.
  - Der Parameter beschreibt eine Menge von *Subscriber*-Typen, welche das *Publisher*-Objekt benachrichten kann. (z.B.: *path.IChatSubscriber*)
- Verwendung
  - Konkrete *Subscriber*-Klassen implementieren ein oder mehrere *Subscriber*-Typen.
  - *Subscriber*-Objekte werden über *add*-Methoden zum *Publisher*-Objekt hinzugefügt.
  - *Subscriber*-Objekte werden über *remove*-Methoden vom *Publisher*-Objekt entfernt.
  - Durch Aufruf privater Benachrichtigungsmethoden kann das *Publisher*-Objekt alle registrierten *Subscriber*-Objekte benachrichtigen.

Beispiel vor Ausführung der Metaprogramme (Sourcecode-Basis)

Deklaration eines *Subscriber*-Typen

Die *loginEvent*- und *messageEvent*-Methoden werden als Empfänger für Events festgelegt. *IChatSubscriber* wird somit implizit zu einen Subscriber-Typen.

```
public interface IChatSubscriber
{
   @Subscribe
   public void loginEvent(String username);
   @Subscribe
   public void messageEvent(String username, String message);
}
```

Deklaration eines Publisher-Typen

Das *IChatPublisher*-Interface wird als *Publisher* für Events von *IChatSubscriber*-Objekten festgelegt.

```
@Publisher( { "IChatSubscriber" })
public interface IChatPublisher
{

}
```

Benachrichtigung der Subscribers im Publisher

Die Klasse *Chatroom* implementiert den *Publisher*-Typ und erhält dadurch dessen *Publisher*-Funktionalität. Durch Aufruf von privaten Methoden können Events ausgelöst werden, welche alle registrierten *Subscriber*-Objekte benachrichtigen.

```
public class Chatroom implements IChatPublisher
{
    public void login(String name) {
        //use private method(generated) to notify subscribers
        loginEvent(name);
    }
    public void chat(String name, String message) {
        //use private method(generated) to notify subscribers
        messageEvent(name, message);
    }
}
```

Empfangen der Events im Subscriber

Durch Implementierung des *Subscriber*-Typs können Events des *Publisher*-Objekts empfangen werden.

```
public class Person implements IChatSubscriber
{
    @Constructor
    @Get
    private String name;

    public void messageEvent(String username, String message){
        //called by publisher
        System.out.println(username + ": " + message);
    }
    public void loginEvent(String username){
        //called by publisher
        System.out.println("login: " + username);
    }
}
```

Verwendung

Subscriber-Objekte werden mittels *add*-Methoden zum *Publisher*-Objekt hinzugefügt. Durch Aufruf der privaten Benachrichtigungsmethoden, welche Events auslösen (*login* und *chat*), werden die Empfängermethoden in allen registierten *Subscriber*-Objekten aufgerufen.

```
public class Main {
    public static void main(String[] args) {
        Chatroom chatroom = new Chatroom();

        Person person1 = new Person("Person1");
        Person person2 = new Person("Person2");

        chatroom.addIChatSubscriber(person1);
        chatroom.addIChatSubscriber(person2);

        chatroom.login(person1.getName());
        chatroom.login(person2.getName());

        chatroom.chat(person1.getName(), "Hello " + person2.getName());
        chatroom.chat(person2.getName(), "Hello " + person1.getName());
    }
}
```

Beispiel nach Ausführung der Metaprogramme (Sourcecode-Generat)

Erweiterung des Publisher-Interfaces

Das Metaprogramm generiert für den angegebenen *Subscriber*-Typ eine *add*, *remove*- und *get*-Methode.

```
@Publisher( { "IChatSubscriber" })
public interface IChatPublisher
{
    //************** begin of generated code ****************
    public void addIChatSubscriber(IChatSubscriber element);

    public void removeIChatSubscriber(IChatSubscriber element);

    public List<IChatSubscriber> getIChatSubscribers();
    //************** end of generated code ****************
}
```

Erweiterung der Publisher-Implementierung

Das Metaprogramm generiert Implementierungen für die *add*, *remove*- und *get*-Methode. Außerdem wird für jede *Subscriber*-Methode eine private Benachrichtungsmethode erstellt, welche ein Event an alle *Subscriber*-Objekte weiterleitet.

```
public class Chatroom implements IChatPublisher{
    //************** begin of generated code ****************
    private List<IChatSubscriber> iChatSubscriberSubscribers = new ArrayList();
    //************** end of generated code ****************

    public void login(String name) {
        //use private method(generated) to notify subscribers
        loginEvent(name);
    }
    public void chat(String name, String message) {
        //use private method(generated) to notify subscribers
        messageEvent(name, message);
    }
```

```
    //************** begin of generated code ****************
    public void addIChatSubscriber(IChatSubscriber element) {
        iChatSubscriberSubscribers.add(element);
    }
    public void removeIChatSubscriber(IChatSubscriber element) {
        iChatSubscriberSubscribers.remove(element);
    }
    public List<IChatSubscriber> getIChatSubscribers() {
        return (List<IChatSubscriber>)java.util.Collections.unmodifiableList(
                iChatSubscriberSubscribers);
    }

    private void loginEvent(String username) {
        for(IChatSubscriber _o : iChatSubscriberSubscribers)
            _o.loginEvent(username);
    }
    private void messageEvent(String username, String message) {
        for(IChatSubscriber _o : iChatSubscriberSubscribers)
            _o.messageEvent(username, message);
    }
    //*************** end of generated code ****************
}
```

Sourcecode des JCM-Metaprogramms

Das Metaprogramm generiert für *Publisher*-Objekte Methoden für das Speichern, Hinzufügen und Entfernen von *Subscriber*-Objekte. Weiters werden private Methoden für die Benachrichtigung von *Subscriber*-Objekten generiert.

```
public class MetaPublisher implements MetaProgrammer {
    public void run(MpManager mp) {
        //for all classes with @Publisher-annotation
        String annotationName = "meta.patterns.publisher.annotations.Publisher";
        for(MpClass publisher : mp.getClassesByAnnotation(annotationName))
        {
            MpAnnotation annotation = publisher.getAnnotation(annotationName);

            //find subscribers by strings from annotation and save in subscriber-list
            List<MpClass> subscriberList = new LinkedList<MpClass>();
            for(String s : annotation.getStrings()) {
                subscriberList.add(mp.findClass(s));
            }

            for(MpClass publisher_subtype : mp.getAllSubClasses(publisher.getQualifiedName())) {
                for(MpClass subscriber : subscriberList) {

                    if(publisher_subtype.isInterface()) {
                        //create add, remove and get method for subscriber-list
                        MetaListAccess.createListAccessInterfaceMethods(mp, publisher_subtype,
                                subscriber.getName(), subscriber.getName());
                    }
                    //if publisher_subtype is not interface
                    else {
                        publisher_subtype.createImport("java.util.List").createImport("java.util.ArrayList").
                                createImport(subscriber.getQualifiedName());

                        //create subscriber-list by using subscriber
```

```
                String listName = MpString.firstToLower(subscriber.getName()) + "Subscribers";
                publisher_subtype.createField(mp.createCode().line("private List<" + subscriber.
                    getName() + "> " + listName + " = new ArrayList();"));

                //create add, remove and get method for subscriber-list
                MetaListAccess.createListAccessImplementationMethods(mp, publisher_subtype,
                    listName, subscriber.getName(), subscriber.getName());

                //for all methods with @Subscribe-annotation of subscriber
                for(MpMethod m : subscriber.getMethodsByAnnotation("meta.patterns.publisher.annotations.
                    Subscribe")) {
                  //create private copy of methodheader with delegation to subscribers
                  publisher_subtype.createMethod( mp.createCode()
                      .line(m.codeOfMethodHeader().replace("public", "private")).startBlock()
                        .line("for(" + subscriber.getName() + " _o : " + listName + ")")
                        .line(" _o." + m.getName() + "(" + m.codeOfParameterNameList() + ");").endBlock
                            ());
} } } } } } }
```

Evaluierung

Der Zeitbedarf des JCM-Metaprogramms bei 80 Klassen/Interfaces in der Sourcecode-Basis liegt bei ca. **325** Millisekunden.

Der Anteil des generierten Codes wird durch die Gegenüberstellung der Zeichenanzahl vor und nach der Generierung (ohne Kommentare) aller Java-Dateien (Ausnahme: Main.java) berechnet.

Anzahl Zeichen vor Generierung	858
Anzahl Zeichen nach Generierung	1762
Anteil generierter Code	51,3%

Fazit

- Reduzierter Codierungsaufwand.
- Deklarative Programmierung verbessert Überblick und Verständlichkeit.
- Erspart die Verwendung abstrakter Klassen um Boilerplate-Code von *Publisher*-Klassen wiederzuverwenden. *Publisher*-Klassen können dadurch von wichtigeren Klassen erben. (z.B.: JFrame)
- Benachrichtigungsmethoden werden automatisch verwaltet, d.h. bei einer Änderung müssen keine *Publisher*-Typen angepasst werden.

9.2 Strategy

Funktion

Das *Strategy*-Metaprogramm ermöglicht eine deklarative Programmierung des *Strategy*-Patterns.

Beschreibung

Das *StrategyContext*-Objekt delegiert bestimmte Methoden an ein konkretes *Strategy*-Objekt, welches zur Laufzeit ausgewechselt werden kann. Dadurch kann sich das Verhalten eines *StrategyContext*-Objekts dynamisch zur Laufzeit ändern.

Eine ausführlichere Beschreibung befindet sind im Abschnitt 2.2 „Strategy-Pattern“ auf Seite 9.

Verwendung

- Deklaration einer oder mehrerer *Strategy*-Typen
  - Interfaces und Klassen können implizit als *Strategy*-Typ verwendet werden.
- Deklaration eines StrategyContext-Typen
  - Interface oder Klasse mit *@StrategyContext(<String[]>)* annotieren.
  - Der Parameter beschreibt eine Menge von *Strategy*-Typen, welche der *StrategyContext* auswechseln kann (z.B.: *path.ISortStrategy*).
- Verwendung
  - Konkrete *Strategy*-Klassen implementieren ein oder mehrere *Strategy*-Typen.
  - Das erste konkrete *Strategy*-Objekt wird über den Konstruktor des StrategyContext gesetzt.
  - Der Wechsel des konkreten *Strategy*-Objekts erfolgt über Setter-Methoden.
  - Das *StrategyContext*-Objekt implementiert das gleiche Interface wie das *Strategy*-Objekt, wobei dessen Methoden weitergeleitet werden.

Beispiel vor Ausführung der Metaprogramme (Sourcecode-Basis)

Erstellung eines *Strategy*-Interfaces

Das *ISortStrategy*-Interface definiert das gemeinsame Interface der Strategy-Implementierungen. Es ist keine explizite Deklaration notwendig.

```
public interface ISortStrategy
{
    public void sort();
}
```

Erstellung eines *StrategyContext*-Interfaces

Das *ISortedList*-Interface wird als *StrategyContext*-Typ für *ISortStrategy*-Typen festgelegt.

```
@StrategyContext({ "ISortStrategy" })
public interface ISortedList
{

}
```

Erstellung einer konkreten *Strategy*-Klasse

Die Klasse *QuickSort* implementiert das *ISortStrategy*-Interface und kann dadurch mit anderen Implementierungen von *ISortStrategy* ausgewechselt werden.

```
public class QuickSort implements ISortStrategy
{
   public void sort()
   {
      System.out.println("Apply QuickSort.");
   }
}
```

Verwendung

Über den Konstruktor wird die erste konkrete Strategy dem *StrategyContext*-Objekt übergeben. Später wird diese über den Setter des *StrategyContext*-Objekts zur Laufzeit gewechselt.

```
public class Main {
public static void main(String[] args) {
      //create concrete strategycontext with concrete strategy
      ISortedList list = new ConcreteSortedList(new MergeSort());
      list.sort();

      //changy strategy
      list.setISortStrategy(new QuickSort());
      list.sort();
   }
}
```

Beispiel nach Ausführung der Metaprogramme (Sourcecode-Generat)

Erweiterung der StrategyContext-Interfaces

Das Metaprogramm generiert eine Setter-Methode für das *Strategy*-Interface *ISortStrategy*. Anschließend wird für alle Methoden des *Strategy*-Interface eine Delegationsmethode generiert (*sort*-Methode).

```
@StrategyContext({ "ISortStrategy" })
public interface ISortedList{
   //************** begin of generated code ****************
   public void setISortStrategy(ISortStrategy strategy);
   public void sort();
   //************** end of generated code ******************
```

```
}
```

Erweiterung der StrategyContext-Implementierung

Das Metaprogramm generiert ein Feld in dem die aktuelle *Strategy*-Instanz gespeichert wird und macht es über Konstrutor und Setter veränderbar. Anschließend wird für alle Methoden des *Strategy*-Interface eine konkrete Delegationsmethode generiert (*sort*-Methode).

```
public class ConcreteSortedList implements ISortedList {
    //************** begin of generated code ****************
    @Set
    @Parameter
    private ISortStrategy iSortStrategy;

    public void sort() {
        iSortStrategy.sort();
    }

    public ConcreteSortedList(ISortStrategy iSortStrategy) {
        this.iSortStrategy = iSortStrategy;
    }

    public void setISortStrategy(ISortStrategy iSortStrategy) {
        this.iSortStrategy = iSortStrategy;
    }
    //************** end of generated code ****************
}
```

Sourcecode des JCM-Metaprogramms

Das Metaprogramm generiert für *StrategyContext*-Objekte Methoden für das Speichern und Setzen der *Strategy*-Objekte sowie Delegationsmethoden zu den *Strategy*-Objekten.

```
public class MetaStrategy implements MetaProgrammer {
    public void run(MpManager mp) {
        //for all classes with @Stratgy−annotation
        String annotationName = "meta.patterns.strategy.annotations.StrategyContext";
        for(MpClass context : mp.getClassesByAnnotation(annotationName))
        {
            MpAnnotation annotation = context.getAnnotation(annotationName);
            //find strategies with strings from annotation and save in list
            List<MpClass> strategyList = new LinkedList<MpClass>();
            for(String s : annotation.getStrings()) {
                strategyList.add(mp.findClass(s));
            }

            //for each subtype of context−class
            for(MpClass context_subtype : mp.getAllSubClasses(context.getQualifiedName())) {
                //and for each strategy
                for(MpClass strategy : strategyList) {
                    if(context_subtype.isInterface()) {
                        context_subtype.createImport(strategy.getQualifiedName());
                        //create stub for setter method
                        MetaGetSet.createSetterInterfaceMethod(mp, context_subtype, strategy.getName(),
                                strategy.getName());
```

```
                for(MpMethod m : strategy.getMethods()) {
                    //createImprot for each parameter of method
                    for(MpParameter p : m.getParameters()) {
                        context_subtype.createImport(p.getType().getQualifiedName());
                    }
                    //create stub of delegation-method
                    context_subtype.createMethod(mp.createCode().line(m.codeOfMethodHeader() + ";")
                        );
                }
            }
            //if context_subtype is not interface
            else {
                context_subtype.createImport(strategy.getQualifiedName()).createImport("meta.
                    basics.constructor.annotations.Constructor");
                //create field for saving the strategy with constructor
                String fieldName = MpString.firstToLower(strategy.getName());
                context_subtype.createField(mp.createCode().line("@Constructor").line("private " +
                    strategy.getName() + " " + fieldName + ";"));

                //create setter for field
                MetaGetSet.createSetterImplementationMethod(mp, context_subtype, fieldName,
                    strategy.getName());

                //create implementations for delegations methods
                for(MpMethod m : strategy.getMethods())
                {
                    //createImport for each parameter of method
                    for(MpParameter p : m.getParameters()) {
                        context_subtype.createImport(p.getType().getQualifiedName());
                    }
                    //create delegation-method to strategy
                    context_subtype.createMethod(mp.createCode().line(m.codeOfMethodHeader()).
                        startBlock()
                        .line(m.getReturn() + " " +fieldName + "." + m.getName() + "(" + m.
                            codeOfParameterNameList() + ");")
                        .endBlock());
                }
} } } } } }
```

Evaluierung

Der Zeitbedarf des JCM-Metaprogramms bei 80 Klassen/Interfaces in der Sourcecode-Basis liegt bei ca. **331** Millisekunden.

Der Anteil des generierten Codes wird durch die Gegenüberstellung der Zeichenanzahl vor und nach der Generierung (ohne Kommentare) aller Java-Dateien (Ausnahme: Main.java) berechnet.

Anzahl Zeichen vor Generierung	410
Anzahl Zeichen nach Generierung	784
Anteil generierter Code	47,7%

Fazit

- Reduzierter Codierungsaufwand.
- Deklarative Programmierung verbessert Überblick und Verständlichkeit.

- Erspart die Verwendung abstrakter Klassen um Boilerplate-Code von *StrategyContext*-Klassen wiederzuverwenden. Diese können dadurch von wichtigeren Klassen erben (z.B.: IFrame).
- Delegationsmethoden werden automatisch verwaltet, d.h. bei einer Änderung müssen keine *StrategyContext*-Typen angepasst werden.

9.3 Composite

Funktion

Das *Composite*-Metaprogramm ermöglicht eine deklarative Programmierung des *Composite*-Patterns.

Beschreibung

Ein *Composite*-Objekt hält Referenzen zu einer Menge von *Component*-Objekten, welche zur Laufzeit hinzugefügt und entfernt werden können. Das *Composite*-Objekt hat dasselbe Interface wie *Component*-Objekte, wodurch eine Hierarchie von *Component*-Objekten aufgebaut werden kann. Weiters leiten *Composite*-Objekte alle Methoden des *Component*-Interfaces an alle Child-Objekte weiter. Child-Klassen implementieren das *Component*-Interfaces und werden von nun an als *Leaf*-Klassen bezeichnet.

Eine ausführlichere Beschreibung befindet sind im Abschnitt 2.2 „Composite-Pattern“ auf Seite 11.

Verwendung

- Deklaration eines *Component*-Typen
  - Interfaces und Klassen können implizit als *Component*-Typ verwendet werden.
- Deklaration eines *Composite*-Typen
  - Klasse mit *@Composite(<String>)* annotieren.
  - Der Parameter beschreibt den gemeinsamen *Component*-Typ(z.B.: *path.Graphic*).
- Verwendung
  - Konkrete *Leaf*-Klassen implementieren den gemeinsamen *Component*-Typ.
  - *Component*-Instanzen werden mittels *add*-Methode zu einer *Composite*-Objekt hinzugefügt.
  - *Component*-Instanzen werden mittels *remove*-Methode von einer *Composite*-Objekt entfernt.
  - Methoden des gemeinsamen *Component*-Typs werden von *Composite*- Objekten zu den zugehörigen Child-Instanzen weitergeleitet.

Beispiel vor Ausführung der Metaprogramme (Sourcecode-Basis)

Deklaration eines *Component*-Typen

Das Graphic-Interface bildet den gemeinsamen Obertypen für *Leaf*- und *Composite*-Klassen. Eine explizite Deklaration ist nicht notwendig.

```
public interface Graphic
{
    public void draw();
}
```

Deklaration eines *Composite*-Typen

Die Klasse *GraphicComposite* wird mittels Annotation als *Composite*-Klasse für Untertypen des *Graphic*-Interface deklariert.

```
@Composite("Graphic")
public class GraphicComposite
{

}
```

Erstellung einer konkreten *Leaf*-Klasse

Die Klasse *Circle* implementiert das *Graphic*-Interface und kann daher als *Leaf*-Klasse eingesetzt werden.

```
public class Circle implements Graphic
{
    public void draw(){
        System.out.println("Draw Circle");
    }
}
```

Verwendung

Instanzen von *Circle* und *Rectangle* werden zu einem *GraphicComposite*-Objekt hinzugefügt, welche wiederum in eine übergeordnete *GraphicComposite*-Objekte hinzugefügt wird. Dadurch entsteht eine Hierarchie von *Leaf*- und *Composite*-Instanzen. Der Aufruf der *draw*-Methode wird von der *Composite*-Instanz an alle Child-Instanzen weitergeleitet.

```
public class Main {
    public static void main(String[] args) {
        GraphicComposite composite = new GraphicComposite();
        GraphicComposite composite2 = new GraphicComposite();

        composite.add(new Rectangle());
        composite.add(new Circle());

        composite2.add(composite);
        composite2.add(new Rectangle());

        composite2.draw();
    }
}
```

Beispiel nach Ausführung der Metaprogramme (Sourcecode-Generat)

Erweiterung der *GraphicComposite*-Klasse

Das Metaprogramm generiert Implementierungen für die *add*, *remove*- und *get*-Methode mit Hilfe des *ListAccess*-Metaprogramms. Weiters wird für jede *Component*-Methode eine Delegationsmethode erstellt, welche den Methodenaufruf an alle Child-Objekte weiterleitet.

```
@Composite("Graphic")
public class GraphicComposite implements Graphic {
   //************** begin of generated code ****************
   private List<Graphic> childrenOfGraphic = new ArrayList<Graphic>();

   public void addChild(Graphic element) {
      childrenOfGraphic.add(element);
   }
   public void removeChild(Graphic element) {
      childrenOfGraphic.remove(element);
   }
   public List<Graphic> getChilds() {
      return (List<Graphic>)java.util.Collections.unmodifiableList(childrenOfGraphic);
   }

   public void draw() {
      for(Graphic child : childrenOfGraphic)
         child.draw();
   }
   //************** end of generated code ****************
}
```

Sourcecode des JCM-Metaprogramms

Das Metaprogramm generiert für *Composite*-Klassen Methoden für das Speichern, Hinzufügen und Entfernen von *Component*-Objekten. Weiters werden Delegationsmethoden für das Weiterleiten an die Child-Objekte generiert.

```
public class MetaComposite implements MetaProgrammer
{
   public void run(MpManager mp) {
      //for all classes with @Composite−annotation
      String annotationName = "meta.patterns.composite.annotations.Composite";
      for(MpClass composite : mp.getClassesByAnnotation(annotationName))
      {
         MpAnnotation annotation = composite.getAnnotation(annotationName);
         MpClass component = mp.findClass(annotation.getString());

         if(component != null) {
            //insert component−interface to composite−class
            composite.insertInterface(component);

            //some imports & label definition
            composite.createImport(component.getQualifiedName()).createImport("java.util.List").
                  createImport("java.util.ArrayList");
            String fieldName = "childrenOf" + component.getName();
            String typeName = component.getName();

            //create field to store children with component−interface
            composite.createField(mp.createCode().line("private List<" + typeName + "> " + fieldName + "
                  = new ArrayList<" + typeName + ">();"));

            //create add−, remove− and get−method for list
            MetaListAccess.createListAccessImplementationMethods(mp, composite, fieldName, "
                  Child", typeName);

            //for each method of component
```

```
            for(MpMethod m : component.getMethods()) {
                //create delegation-method to children
                composite.createMethod(mp.createCode().line(m.codeOfMethodHeader())
                        .startBlock()
                            .line("for(" + typeName + " child : " + fieldName + ")")
                            .line(" child." + m.getName() + "(" + m.codeOfParameterNameList() + ");")
                        .endBlock());
            }
} } } }
```

Evaluierung

Der Zeitbedarf des JCM-Metaprogramms bei 80 Klassen/Interfaces in der Sourcecode-Basis liegt bei ca. **751** Millisekunden.

Der Anteil des generierten Codes wird durch die Gegenüberstellung der Zeichenanzahl vor und nach der Generierung (ohne Kommentare) aller Java-Dateien (Ausnahme: Main.java) berechnet.

Anzahl Zeichen vor Generierung	311
Anzahl Zeichen nach Generierung	755
Anteil generierter Code	58,8%

Fazit

- Reduzierter Codierungsaufwand.
- Deklarative Programmierung verbessert Überblick und Verständlichkeit.
- Erspart die Verwendung abstrakter Klassen um Boilerplate-Code von *Composite*-Klassen wiederzuverwenden. *Composite*-Klassen können dadurch von wichtigeren abstrakten Klassen erben.
- Delegationsmethoden werden automatisch verwaltet, d.h. bei einer Änderung müssen keine *Composite*-Typen angepasst werden.

9.4 Visitor

Funktion

Das *Visitor*-Metaprogramm ermöglicht eine deklarative Programmierung des *Visitor*-Patterns.

Beschreibung

Der *Visitor*-Typ hat für jeden *Element-Typ* eine *visit*-Methode, welche von der *accept*-Methode einer *Element*-Klasse aufgerufen wird. Die *accept*-Methode wird für eine Menge von *Element*-Objekten sequentiell mit dem gleichen Visitor-Objekt aufgerufen. Das *Visitor*-Klasse hat dadurch die Möglichkeit Algorithmen über Strukturen auszuführen. Das Visitor-Pattern bietet sich vor allem in Kombination mit dem Composite-Pattern an.

Eine ausführlichere Beschreibung befindet sind im Abschnitt 2.2 „Visitor-Pattern“ auf Seite 13.

Verwendung

- Deklaration eines *Element*-Typen
  Interfaces und Klassen können implizit als *Element*-Typ verwendet werden.
- Deklaration eines *Visitor*-Typen
  - Interface oder Klasse mit *@Visitor(<String>)* annotieren.
  - Der Parameter beschreibt den *Element*-Typ, welche das *Visitor*-Objekt verarbeiten kann (z.B.: *path.Graphic*)
- Verwendung
  - Die *accept*-Methode wird für jedes *Element*-Objekt einer Struktur mit dem gleichen *Visitor*-Objekt ausgeführt.
  - Die konkrete *Visitor*-Klasse implementiert eine *visit*-Methode für jedes Element.
  - Das *Visitor* funktioniert gut in Kombination mit dem *Composite*-Pattern: Das *Composite*-Interface kann eine *accept*-Methode definieren, welche in allen Child-Objekten weitergeleitet wird.

Beispiel vor Ausführung der Metaprogramme (Sourcecode-Basis)

Erstellung eines *Visitor*-Interfaces

Das *GraphicVisitor*-Interface wurde mittels Annotation als *Visitor* für Subtypen von *Graphic* deklariert.

```
@Visitor("Graphic")
public interface GraphicVisitor
{
}
```

Erstellung einer *Visitor*-Klasse

Die *GraphicCounterVisitor*-Klasse implementiert das *GraphicVisitor*-Interface und kann daher in *accept*-Methoden von *Graphic*-Subtypen ausgeführt werden. Die GraphicCounterVisitor-Klasse muss für jeden *Element*-Typ eine *visit*-Methode implementieren.

```
public class GraphicCounterVisitor implements GraphicVisitor {
   @Get
   private int circleCount;
   @Get
   private int rectangleCount;

   public void visit(Circle circle) {
      circleCount++;
   }
   public void visit(Rectangle rectangle) {
      rectangleCount++;
   }

   public void visit(Graphic graphic) { }

   public void visit(GraphicComposite graphicComposite) { }
}
```

Verwendung

Instanzen von *Circle* und *Rectangle* werden zu einem *GraphicComposite*-Objekt hinzugefügt, welches wiederum in ein übergeordnetes *GraphicComposite*-Objekt hinzugefügt wird. Dadurch entsteht eine Hierarchie von *Leaf* - und *Composite*-Instanzen. Das *GraphicCounterVisitor*-Objekt wird über diese Struktur mittels Aufruf der *accept*-Methode ausgeführt.

```
public class Main {
   public static void main(String[] args) {
      GraphicComposite composite = new GraphicComposite();
      GraphicComposite composite2 = new GraphicComposite();

      composite.add(new Rectangle());
      composite.add(new Circle());

      composite2.add(composite);
      composite2.add(new Rectangle());

      GraphicCounterVisitor counter = new GraphicCounterVisitor();

      composite2.accept(counter);
      System.out.println("Rectangles: " + counter.getRectangleCount());
      System.out.println("Circles: " + counter.getCircleCount());
   }
}
```

Beispiel nach Ausführung der Metaprogramme (Sourcecode-Generat)

Erweiterung der *Visitor*-Interfaces

Das Metaprogramm fügt für jeden *Element*-Typ eine *visit*-Methode hinzu.

```
@Visitor("Graphic")
public interface GraphicVisitor {
    //************** begin of generated code ****************
    public void visit(Circle circle);
    public void visit(Graphic graphic);
    public void visit(Rectangle rectangle);
    //************** end of generated code ****************
}
```

Erweiterung der *Element*-Interfaces

Das Metaprogramm fügt dem *Element*-Interface eine *accept*-Methode hinzu.

```
public interface Graphic
{
    public void draw();
    //************** begin of generated code ****************
    public void accept(GraphicVisitor visitor);
    //************** end of generated code ****************
}
```

Erweiterung der *Element*-Implementierungen

Das Metaprogramm generiert für jede *Element*-Implementierung (Circle, Rectangle, ...) eine *accept*-Methode, welche die *visit*-Methode des übergebenen *Visitor*-Objekts aufruft.

```
public class Circle implements Graphic
{
    public void draw(){
        System.out.println("Draw Circle");
    }
    //************** begin of generated code ****************
    public void accept(GraphicVisitor visitor){
        visitor.visit(this);
    }
    //************** end of generated code ****************
}
```

Erweiterung der *Element*-Implementierungen in Kombination mit @Composite

Das *Visitor*-Metaprogramm muss vor dem *Composite*-Metaprogramm ausgeführt werden. Der Grund hierfür ist, dass die vom *Visitor*-Metaprogramm hinzufügte *accept*-Methode des *Graphic*-Interface von dem *Composite*-Metaprogramm als weiterzuleitende Methode interpretiert wird.

```
@Composite("Graphic")
public class GraphicComposite implements Graphic {
    //************** begin of generated code ****************
    private List<Graphic> childrenOfGraphic = new ArrayList<Graphic>();
```

```
    public void add(Graphic child) {
        childrenOfGraphic.add(child);
    }
    public void remove(Graphic child) {
        childrenOfGraphic.remove(child);
    }

    public void draw() {
        for(Graphic child : childrenOfGraphic)
            child.draw();
    }
    public void accept(GraphicVisitor visitor) {
        for(Graphic child : childrenOfGraphic)
            child.accept(visitor);
    }
    //************** end of generated code *****************
}
```

Sourcecode des JCM-Metaprogramms

Das Metaprogramm generiert für das *Visitor*-Interface eine visit-Methode für jede konkrete *Element*-Klasse. Gleichzeitig wird zu jeden *Element*-Typ eine passende *accept*-Methode hinzugefügt.

```
public class MetaVisitor implements MetaProgrammer {
    public void run(MpManager mp) {
        //for all classes with @Visitor−annotation
        String annotationName = "meta.patterns.visitor.annotations.Visitor";
        for(MpClass visitor : mp.getClassesByAnnotation(annotationName))
        {
            MpAnnotation annotation = visitor.getAnnotation(annotationName);

            //for each subtype of element
            for(MpClass element : mp.getAllSubClasses(annotation.getString())) {

                System.out.println(element.getName());

                //create import for visitor
                element.createImport(visitor.getQualifiedName());

                //if element is interface
                if(element.isInterface()) {
                    //create stub of accept−method into element
                    element.createMethod(mp.createCode().line("public void accept(" + visitor.getName() + "
                            visitor);"));
                }
                //if element is not interface
                else {
                    //create accept method implementation into element
                    element.createMethod(mp.createCode().line("public void accept(" + visitor.getName() + "
                            visitor)")
                            .startBlock().line("visitor.visit(this);").endBlock());
                }

                //create stub of visit−method into visitor
                visitor.createImport(element.getQualifiedName());
```

```
            visitor.createMethod(mp.createCode().line("public void visit(" + element.getName() + " " +
                MpString.firstToLower(element.getName() + ");")));
} } } }
```

Evaluierung

Der Zeitbedarf des JCM-Metaprogramms bei 80 Klassen/Interfaces in der Sourcecode-Basis liegt bei ca. **537** Millisekunden.

Der Anteil des generierten Codes wird durch die Gegenüberstellung der Zeichenanzahl vor und nach der Generierung (ohne Kommentare) aller Java-Dateien (Ausnahme: Main.java) berechnet.

Anzahl Zeichen vor Generierung	714
Anzahl Zeichen nach Generierung	1547
Anteil generierter Code	53,8%

Fazit

- Stark verbesserte Wart- und Erweiterbarkeit, da die *accept*-Methode zu jeder *Element*-Klasse einzeln hinzugefügt werden müsste. Wiederverwendung durch abstrakte Klassen ist aufgrund des notwendigen dynamischen Dispatchs nicht möglich.
- Deklarative Programmierung verbessert Überblick und Verständlichkeit.
- Visitmethoden werden automatisch verwaltet, d.h. das *Visitor*-Interface muss nicht bei jedem Hinzufügen eines neues Element angepasst werden.

9.5 Command

Funktion

Das *Command*-Metaprogramm ermöglicht eine deklarative Programmierung des *Command*-Patterns.

Beschreibung

Eine *Command*-Klasse kapselt den Aufruf einer Methode in eine eigene Klasse. Die *execute*-Methode der *Command*-Klasse führt die ursprüngliche Methode auf das als Parameter übergebenen *CommandReceiver*-Objekt aus. Der Vorteil ist, dass dadurch Methodenaufrufe als Objekte behandelt werden können. Beispielsweise können *Command*-Objekte gepspeichert und später auf verschiedene *CommandReceiver*-Objekte ausgeführt werden.

Eine ausführlichere Beschreibung befindet sind im Abschnitt 2.2 „Command-Pattern“ auf Seite 15.

Verwendung

- Deklaration eines *CommandReceiver*-Typen
  - Ein Interface oder eine Klasse mit *@CommandReceiver(<String>)* annotieren.
  - Der Parameter gibt den Pfad an, in dem die *Command*-Klassen generiert werden sollen.
- Deklaration ein oder mehrerer *Command*-Methoden
  - Methoden des *CommandReceiver*-Typs mit *@Command* annotieren.
  - Für jede annotierte Methode wird eine *Command*-Klasse erstellt.
- Verwendung
  - Die benötigten Parameter der Methode werden den *Command*-Objekten mittels einfachem Konstruktor übergeben.
  - Die *execute*-Methode des *Command*-Objekts benötigt eine Instanz des *CommandReceiver*-Typs als Parameter. Anschließend wird die *Command*-Methode dieser Instanz aufgerufen.

Beispiel vor Ausführung der Metaprogramme (Sourcecode-Basis)

Deklaration eines *CommandReceiver*-Typs mit *Command*-Methoden

Das *Driveable*-Interface wird mittels Annotation als *CommandReceiver*-Typ deklariert, welches die *Command*-Klassen in das Package *commands* generiert. Weiters werden alle Methoden als *Command*-Methoden deklariert.

```
@CommandReceiver("commands")
public interface Driveable {
   @Command
   public void setSpeed(int speed);
```

```
    @Command
    public void turnRight();

    @Command
    public void turnLeft();
}
```

Definition von *CommandReceiver*-Klassen

Die Klasse *Car* und *Plane* können durch die Implementierung des *CommandReceiver*-Typs als Empfänger von *Command*-Objekten verwendet werden.

```
public class Car implements Driveable {

    public void setSpeed(int speed) {
        System.out.println("Car set speed to: " + speed);
    }
    public void turnRight() {
        System.out.println("Car turn right.");
    }
    public void turnLeft() {
        System.out.println("Car turn left.");
    }
}
```

```
public class Plane implements Driveable{

    public void setSpeed(int speed) {
        System.out.println("Plane set speed to: " + speed);
    }
    public void turnRight() {
        System.out.println("Plane turn right.");
    }
    public void turnLeft() {
        System.out.println("Plane turn left.");
    }
}
```

Verwendung

Es wird eine Liste von *Command*-Objekten erstellt, welche anschließend auf eine *Car*- und eine *Plane*-Instanz ausgeführt werden.

```
public class Main {
    public static void main(String[] args) {
        //define a set of commands
        List<DriveableCommand> commands = new LinkedList<DriveableCommand>();
        commands.add(new SetSpeed(100));
        commands.add(new TurnLeft());
        commands.add(new TurnRight());

        //execute commands on car-instance
        for(DriveableCommand c : commands)
            c.execute(new Car());
        //execute commands on plane-instance
        for(DriveableCommand c : commands)
```

```
        c.execute(new Plane());
    }
}
```

Beispiel nach Ausführung der Metaprogramme (Sourcecode-Generat)

Generierung des *Command*-Interfaces

Das Metaprogramm generiert ein gemeinsames *Command*-Interface um die Speicherung von unterschiedlichen *Command*-Klassen zu ermöglichen.

```
//************** begin of generated code ****************
public interface DriveableCommand {
    public void execute(Driveable driveable);
}
//*************** end of generated code ****************
```

Generierung der *Command*-Klassen

Das Metaprogramm generiert für jede mit *@Command* annotierte Methode eine eigene *Command*-Klasse, welche das gemeinsame *Command*-Interface implementiert. Für jeden Parameter der *Command*-Methode wird eine Variable in die Command-Klasse generiert, für die mittels *@Constructor*-Annotation ein einfacher Constructor erstellt wird. Die *execute*-Methode führt eine Delegation auf das übergebene *CommandReceiver*-Objekt durch.

```
//************** begin of generated code ****************
public class SetSpeed implements DriveableCommand
    @Constructor
    private int speed;

    public void execute(Driveable driveable) {
        driveable.setSpeed(speed);
    }

    public SetSpeed(int speed) {
        this.speed = speed;
    }
}
//*************** end of generated code ****************
```

```
//************** begin of generated code ****************
public class TurnLeft implements DriveableCommand {

    public void execute(Driveable driveable) {
        driveable.turnLeft();
    }
}
//*************** end of generated code ****************
```

```
//************** begin of generated code ****************
public class TurnRight implements DriveableCommand {
```

```
    public void execute(Driveable driveable) {
        driveable.turnRight();
    }
}
//*************** end of generated code ****************
```

Sourcecode des JCM-Metaprogramms

Das Metaprogramm generiert für jede mit *@Command* annotierte Methode des *CommandReceiver*-Typs eine eigene *Command*-Klasse, sowie ein gemeinsames *Command*-Interface.

```
public class MetaCommand implements MetaProgrammer {
    public void run(MpManager mp) {
    //for all classes with @Publisher−annotation
    String annotationName = "meta.patterns.command.annotations.CommandReceiver";
    for(MpClass receiver : mp.getClassesByAnnotation(annotationName)) {
        MpAnnotation annotation = receiver.getAnnotation(annotationName);
        if(annotation != null) {
            String path = annotation.getString();
            String fieldName = MpString.firstToLower(receiver.getName());
            //create common interface for all commands
            MpType commandInterfaceType = mp.createType(path + "." + receiver.getName() + "Command");
            mp.createClass(commandInterfaceType.getQualifiedName(),mp.createCode()
                    .line("public interface " + commandInterfaceType.getName())
                    .startBlock().line("public void execute(" + receiver.getName() + " " + fieldName + ");").
                        endBlock())
                    .createImport(receiver.getQualifiedName());

            //for each method with @Command−Annotation
            for(MpMethod m : receiver.getMethodsByAnnotation("meta.patterns.command.annotations.Command")) {
                MpType commandType = mp.createType(path + "." + MpString.firstToUpper(m.getName()));
                //create command−class for method
                MpClass command = mp.createClass(commandType.getQualifiedName(), mp.createCode()
                        .classBody("public class " + commandType.getName() + " implements " +
                                commandInterfaceType.getName()));
                command.createImport("meta.basics.constructor.annotations.Constructor").createImport(receiver.
                        getQualifiedName());
                command.createImport(commandInterfaceType.getQualifiedName());

                //for each parameter of method
                for(MpParameter p : m.getParameters()) {
                    receiver.createImport(p.getType().getQualifiedName());
                    //create field for storage via constructor
                    command.createField(mp.createCode().line("@Constructor")
                            .line("private " + p.getType().getName() + " " + p.getName() + ";"));
                }
                //create delegation−method to receiver−instance
                command.createMethod(mp.createCode().line("public void execute(" + receiver.getName() + " " +
                        fieldName + ")")
                        .startBlock()
                            .line(fieldName + "." + m.getName() + "(" + m.codeOfParameterNameList() + ");")
                        .endBlock());
} } } } }
```

Evaluierung

Der Zeitbedarf des JCM-Metaprogramms bei 80 Klassen/Interfaces in der Sourcecode-Basis liegt bei ca. **634** Millisekunden.

Der Anteil des generierten Codes wird durch die Gegenüberstellung der Zeichenanzahl vor und nach der Generierung (ohne Kommentare) aller Java-Dateien (Ausnahme: Main.java) berechnet.

Anzahl Zeichen vor Generierung	703
Anzahl Zeichen nach Generierung	1230
Anteil generierter Code	42,8%

Fazit

- Stark reduzierter Codierungsaufwand, da für jede *CommandReceiver*-Methode eine eigene *Command*-Klasse erstellt werden müsste.
- Verbesserte Wartbarkeit, da *Command*-Methoden und *Command*-Klassen automatisch synchronisiert werden.
- Deklarative Programmierung verbessert Überblick und Verständlichkeit.

9.6 Multimethod

Funktion

Das *Multimethod*-Metaprogramm ermöglicht eine deklarative Programmierung von Multimethoden.

Beschreibung

Bei Multimethoden wird die auszuführende Methode dynamisch anhand des Parameters (*Dispatch*-Typ) als auch des Empfängers (*Multimethod*-Klasse) bestimmt. Die Methode wird also dynamisch anhand des dynamischen Typs von 2 Objekten ausgewählt. Da Java dies nur mittels zweifachen Dispatch simulieren kann, werden erhebliche Codemengen dafür benötigt.

Eine ausführlichere Beschreibung befindet sind im Abschnitt 2.3 „Multimethoden in Java“ auf Seite 17.

Verwendung

- Deklaration einer Multimethode
  - Methode eines Interfaces oder Obertypen mit *@Multimethod(<String>)* annotieren.
  - Der Parameter gibt den vollständigen Namen des *Dispatch*-Typs an.
- Erstellung von überladenen Methoden
  - Durch Überladung der Multimethode mit einem spezielleren Typ kann die Fallunterscheidung in konkreten Untertypen implementiert werden.
- Verwendung
  - In *Multimethod*-Klassen wurde eine weitere *dispatch*-Methode generiert. Welche als Suffix den Namen der Multimethode hat (z.B. *dispatchCollide*).
  - Diese *dispatch*-Methode kann als indirekter Aufruf der Multimethode mit dem selben Parameter angesehen werden, jedoch wird hierbei der dynamische Typ des Parameter beachtet.

Beispiel vor Ausführung der Metaprogramme (Sourcecode-Basis)

Deklaration einer Multimethode

Das *Collidable*-Interface deklariert die *collisionWith*-Methode als Multimethode, welcher das *Colliadable*-Interface als *Dispatch*-Parameter verwendet.

```
public interface Collidable {

   @Multimethod("Collidable")
   public void collisionWith(Collidable c);
}
```

Erstellung von überladenen Methoden

Die Klasse *Asteroid* und *Spaceship* erstellen jeweils eine Default-Methode, welche bei Kollision mit einen beliebigen Objekt auftritt. Die *Rocket*-Klasse hat neben der Default-Methode auch eine überladene Methode, welche bei Kollision mit einem *Asteroid*-Objekt Punkte hinzuzählt.

```
public class Asteroid implements Collidable{
    public void collisionWith(Collidable c) {
        System.out.println("Asteroid destroyed.");
    }
}
```

```
public class Spaceship implements Collidable{
    public void collisionWith(Collidable c) {
        System.out.println("Gameover.");
    }
}
```

```
public class Rocket implements Collidable{
    public void collisionWith(Collidable c) {
        System.out.println("Rocked destroyed.");
    }

    public void collisionWith(Asteroid c) {
        collisionWith((Collidable)c);
        System.out.println("+100 points.");
    }
}
```

Verwendung

Alle beteiligen Objekte wurden statisch als *Collidable*-Objekte deklariert um den dynamischen Dispatch zu testen. Zu beachten ist, dass im *Multimethod*-Objekt immer die *dispatch*-Methode aufgerufen werden muss. Würde die *collideWith*-Methode direkt aufgerufen werden, würde kein dynamischer Dispatch erfolgen.

```
public class Main {
    public static void main(String[] args) {
        Spaceship ship = new Spaceship();
        Rocket rocket = new Rocket();
        Asteroid asteroid = new Asteroid();

        collision(rocket,asteroid);
        collision(asteroid,asteroid);
        collision(ship,asteroid);
    }

    private static void collision(Collidable c1, Collidable c2) {
        System.out.println("Collision:");
        c1.dispatchCollisionWith(c2);
        c2.dispatchCollisionWith(c1);
    }
}
```

Beispiel nach Ausführung der Metaprogramme (Sourcecode-Generat)

Generierung der erst- und zweitrangigen *Dispatch*-Methoden im Interface

Das Metaprogramm generiert eine erstrangige *Dispatch*-Methode für den *Dispatch*-Typ. Danach wird für jeden Subtyp des *Multimethod*-Klasse eine zweitrangige *Dispatch*-Methode erstellt.

```
public interface Collidable {

   @Multimethod("Collidable")
   public void collisionWith(Collidable c);

   //************** begin of generated code ****************
   public void dispatchCollisionWith(Collidable e); //first dispatch

   public void dispatchCollisionWith(Asteroid e); //second dispatch
   public void dispatchCollisionWith(Rocket e); //second dispatch
   public void dispatchCollisionWith(Spaceship e); //second dispatch
   //*************** end of generated code ****************
}
```

Generierung der erst- und zweitrangigen *Dispatch*-Methoden in den Implementierungen

Als Beispiel wird nur eine Klasse angeführt, da hier sehr viel Code generiert wird. In allen anderen Implementierungen befinden sich exakt dieselben *Dispatch*-Methoden.
Das Metaprogramm generiert eine erstrangige *Dispatch*-Methode für den *Dispatch*-Typ. Danach wird für jeden Subtyp der *Multimethod*-Klasse eine zweitrangige *Dispatch*-Methode erstellt.

```
public class Asteroid implements Collidable{

   public void collisionWith(Collidable c) {
      System.out.println("Asteroid destroyed.");
   }

   //************** begin of generated code ****************
   public void dispatchCollisionWith(Collidable e) {
      e.dispatchCollisionWith(this);
   }
   public void dispatchCollisionWith(Asteroid e) {
      e.collisionWith(this);
   }
   public void dispatchCollisionWith(Rocket e) {
      e.collisionWith(this);
   }
   public void dispatchCollisionWith(Spaceship e) {
      e.collisionWith(this);
   }
   //*************** end of generated code ****************
}
```

Sourcecode des JCM-Metaprogramms

Das Metaprogramm generiert für jede mit *@Multimethod* annotierte Methode *dispatch*-Methoden in Interfaces und konkreten Klassen des *Dispatch*-Typs als auch der *Multimethod*-Klasse.

```
public class MetaMultimethod implements MetaProgrammer {
   public void run(MpManager mp) {
      for(MpClass receiverType : mp.getClasses()) {
         //for all methods with @Multimethod-annotation
         String annotationName = "meta.patterns.multimethod.annotations.Multimethod";
         for(MpMethod multimethod : receiverType.getMethodsByAnnotation(annotationName)) {

            MpClass dispatchType = mp.findClass(multimethod.getAnnotation(annotationName).
                  getString());
            String dispatchMethodName = "dispatch" + MpString.firstToUpper(multimethod.getName());

            Collection<MpClass> receiverSubtypes = mp.getAllSubClasses(receiverType.
                  getQualifiedName());
            Collection<MpClass> dispatchSubtypes = mp.getAllSubClasses(dispatchType.
                  getQualifiedName());

            //create first dispatch-methods for interfaces and implementations of receiverType
            for(MpClass receiverSubtype : receiverSubtypes) {
               if(receiverSubtype.isInterface()) {
                  receiverType.createMethod(mp.createCode()
                        .line("public void " + dispatchMethodName + "(" + dispatchType.getName() + " e);"));
               } else {
                  receiverSubtype.createMethod(mp.createCode()
                     .line("public void " + dispatchMethodName + "(" + dispatchType.getName() + " e)")
                     .startBlock().line("e." + dispatchMethodName + "(this);").endBlock());
               }
            }
            //create second dispatch-methods for interfaces and implementations of dispatchType
            for(MpClass dispatchSubtype : dispatchSubtypes) {
               for(MpClass receiverSubtype : receiverSubtypes) {
                  if(dispatchSubtype.isInterface()) {
                     dispatchType.createMethod(mp.createCode()
                           .line("public void " + dispatchMethodName + "(" + receiverSubtype.getName() + " e
                                 );"));
                  } else {
                     dispatchSubtype.createMethod(mp.createCode()
                           .line("public void " + dispatchMethodName + "(" + receiverSubtype.getName() + " e
                                 )")
                           .startBlock().line("e." + multimethod.getName() + "(this);").endBlock());
                  }
               }
}}}}}
```

Evaluierung

Der Zeitbedarf des JCM-Metaprogramms bei 80 Klassen/Interfaces in der Sourcecode-Basis liegt bei ca. **480** Millisekunden.

Der Anteil des generierten Codes wird durch die Gegenüberstellung der Zeichenanzahl vor und nach der Generierung (ohne Kommentare) aller Java-Dateien (Ausnahme: Main.java) berechnet.

Anzahl Zeichen vor Generierung	617
Anzahl Zeichen nach Generierung	1772
Anteil generierter Code	65,2%

Fazit

- Stark reduzierter Codierungsaufwand, da das zweifache Dispatching automatisch generiert wird.
- Stark verbesserte Wartbarkeit, da das Hinzufügen neuer Elemente nur mehr kleine Änderungen bestehender Klassen benötigt.

Anmerkung

Es wurde erst im Nachhinein erkannt, dass die vermeintlich schlechtere Lösung mit Hilfe eines umfangreichen *switch-case*-Blocks als einfachere und auch teilweise bessere Lösung mit Hilfe von Metaprogrammen umgesetzt werden könnte. Aufgrund des automatisch generierten *switch-case*-Blocks bleiben Wartung und Überblick gleich, jedoch wird die enorme Anzahl an zusätzlichen *public*-Methoden vermieden. Dies dient jedoch als gutes Beispiel dafür, wie Metaprogramme auch sehr komplexe Muster umsetzen können, aber auch viele neue Möglichkeiten bieten, welche ebenfalls berücksichtigt werden sollten.

KAPITEL 10

Metaprogramme zur Abstraktion von Technologien

Das Kapitel beschreibt wie Metaprogramme zur Abstraktion von konkreten Technologien eingesetzt werden können.

Als Basis dienen Interfaces, welche von Metaprogrammen automatisch generiert werden um dadurch konkrete Implementierungen zu abstrahieren. Diese Interfaces produzieren erhebliche Mengen an Boilerplate-Code, wodurch die manuelle Erstellung viel Zeit in Anspruch nehmen würde. Ein weiterer Vorteil ist, dass Metaprogramme wohldefinierte Interfaces erstellen. Dadurch können weitere Metaprogramme an diese anschließen und konkrete Mappings zu Technologien erstellen. Dadurch ermöglichen sie eine bessere Abstraktion und Einbindung von Technologien.

10.1 Data Access Object

Funktion

Als Data Access Object (DAO) werden Klassen bezeichnen, welche den Zugriff auf unterschiedliche Arten von Datenquellen kapseln. Das *DAO*-Metaprogramm ermöglicht eine deklarative Programmierung von DAOs mittels Entity-Klassen.

Beschreibung

DAOs ermöglichen die Abstraktion von konkreten Technologien zum Speichern und Wiederfinden von Entitäten. Dazu wird für jede Entität ein eigener DAO erstellt.

Eine ausführlichere Beschreibung befindet sind im Abschnitt 2.3 „Data-Access-Object (DAO)“ auf Seite 18.

Verwendung

- Deklaration einer *Entity*-Klasse
  - Klasse mit *@Entity(<String>* annotieren.
  - Der Parameter *name* gibt den Pfad an, an dem der DAO generiert werden soll.
- Deklaration einer *FindBy*-Variable
  - Variable mit *@FindBy(<String[]>* annotieren.
  - Der Parameter ist eine Liste von Strings, welche die möglichen Suchtypen angeben (z.B.: Equal, Greater,...)

Beispiel vor Ausführung der Metaprogramme (Sourcecode-Basis)

Deklaration einer *Entity*-Klasse mit *FindBy*-Variablen

Die Klasse *Person* wird als *Entity*-Klasse deklariert. Weiters werden die Variablen *name* und *age* mittels Annotationen als Suchkriterium hinzugefügt.

```
@Entity(path="persistence")
public class Person
{
   @Constructor
   @Get
   @FindBy( {"Equal"} )
   private String name;

   @Constructor
   @Get
   @FindBy( {"Greater", "Smaller"} )
   private int age;

}
```

Beispiel nach Ausführung der Metaprogramme (Sourcecode-Generat)

Generierung des *DAO*-Interfaces

Das Metaprogramm generiert für die *Entity*-Klasse ein DAO-Interface, welches die Default-Methoden *save*, *delete* und *getAll* definiert. Weiters wird für jeden Suchtyp der *FindBy*-Variable eine Suchmethode erstellt.

```
//************** begin of generated code ****************
public interface IPersonDAO
{
    public void save(Person entity);

    public void delete(Person entity);

    public List<Person> getAllPersons();

    public List<Person> findAllWithNameEqual(String value);

    public List<Person> findAllWithAgeGreater(int value);

    public List<Person> findAllWithAgeSmaller(int value);
}
//*************** end of generated code ****************
```

Sourcecode des JCM-Metaprogramms

Das *MetaDAOBase*-Metaprogramm liest *@Entity*- und *FindBy*-Annotationen aus, definiert den Methoden-Header und ruft protected-Methoden auf, welche von einem konkreten Untertyp implementiert werden müssen.

```
public abstract class MetaDAOBase implements MetaProgrammer {
    protected MpManager mp;
    protected MpClass entity;
    protected MpClass dao;

    protected abstract void implDao(String path);
    protected abstract void implSaveMethod(MpCode code);
    protected abstract void implDeleteMethod(MpCode code);
    protected abstract void implGetAllMethod(MpCode code);
    protected abstract void implFindByMethod(MpCode code, String type);

    public void run(MpManager mp) {
        this.mp = mp;
        //for all classes with @Publisher-annotation
        String annotationName = "meta.abstractions.dao.annotations.Entity";
        for(MpClass entity : mp.getClassesByAnnotation(annotationName)) {
            MpAnnotation entityAnnotation = entity.getAnnotation(annotationName);
            String path = entityAnnotation.getString("path");
            this.entity = entity;

            //create dao instance in subtype
            implDao(path);

            dao.createImport(entity.getQualifiedName()).createImport("java.util.List");
            String type = entity.getName();
```

```
        //create header for all default methods and call subtype
        MpCode save = mp.createCode().text("public void save(" + type + " entity)");
        MpCode delete = mp.createCode().text("public void delete(" + type + " entity)");
        MpCode getAll = mp.createCode().text("public List<" + type + "> getAll" + type + "s()");

        implSaveMethod(save);
        implDeleteMethod(delete);
        implGetAllMethod(getAll);

        dao.createMethod(save);
        dao.createMethod(delete);
        dao.createMethod(getAll);

        //create header for all findby methods and call subtype
        String fieldAnnotation = "meta.abstractions.dao.annotations.FindBy";
        for(MpField field : entity.getFieldsByAnnotation(fieldAnnotation)) {
           for(String searchtype : field.getAnnotation(fieldAnnotation).getStrings()) {
              MpCode findBy = mp.createCode().text("public List<" + type + "> findAllWith" + MpString.
                    firstToUpper(field.getName()) + MpString.firstToUpper(searchtype) + "(" + field.
                    getType().getName() + " value)");
              implFindByMethod(findBy, searchtype);
              dao.createMethod(findBy);
           }
        }
} } }
```

Das *MetaDAOInterface*-Metaprogramm erbt vom *MetaDAOBase*-Metaprogramm. Es erhält die Methoden-Header und generiert daraus Interfaces. Andere Implementierung von *MetaDAOBase* können z.B. ein Mapping zu einer konkreten Technologie realisieren, wobei die passenden Methodenheader bereits definiert wurden.

```
public class MetaDAOInterface extends MetaDAOBase {
   protected void implDao(String path){
      //create interface for entity dao
      MpType daoType = mp.createType(path + ".I" + entity.getName() + "DAO");
      dao = mp.createClass(daoType.getQualifiedName(),mp.createCode()
            .line("public interface " + daoType.getName()).startBlock().endBlock());
   }
   protected void implSaveMethod(MpCode code) {
      code.line(";");
   }
   protected void implDeleteMethod(MpCode code) {
      code.line(";");
   }
   protected void implGetAllMethod(MpCode code) {
      code.line(";");
   }
   protected void implFindByMethod(MpCode code, String type) {
      code.line(";");
   }
}
```

Evaluierung

Der Zeitbedarf des JCM-Metaprogramms bei 80 Klassen/Interfaces in der Sourcecode-Basis liegt bei ca. **638** Millisekunden.

Der Anteil des generierten Codes wird durch die Gegenüberstellung der Zeichenanzahl vor und nach der Generierung (ohne Kommentare) aller Java-Dateien (Ausnahme: Main.java) berechnet.

Anzahl Zeichen vor Generierung	182
Anzahl Zeichen nach Generierung	480
Anteil generierter Code	62,1%

Fazit

- Interfaces werde unmittelbar nach Definition der *Entity*-Klassen generiert. Der Programmierer kann sie anschließend sofort verwenden.
- Das Mapping von *Entity*-Klasse zu DAOs ist klar definiert.
- Ein DAO-Entwickler erhält ein wohldefiniertes Interface, welches er implementieren muss.
- Deklarative Programmierung verbessert Überblick und Verständlichkeit.

Anmerkung

Das klassische DAO-Pattern ist nur für die Speicherung von einfachen Objekt-Bäumen geeignet. Ein komplexes Entity-Relationship-Modell kann aufgrund einer fehlenden Umsetzung von Relationen nicht zufriedenstellend implementiert werden.

Kommentar:
Ein Konzept zur effizienten Übertragung einer Teilmenge eines Entity-Relationship-Modells aus Datenbank und Netzwerk sehe ich als eines der wichtigsten aber auch komplexesten Themen der Software-Entwicklung an. Das Problem derzeitiger Implementierungen sehe ich in der Umsetzung von Relationen. Bei JPA können Relationen über Methoden in Objekten abgefragt werden, welches 1:N Abfragen fördert und Entitäten schlecht übertragbar macht.

Meiner Meinung nach darf die Komplexität von Relationen nicht vor dem Benutzer verborgen werden. Eine Query-Klasse, welche eine Teilmenge von auszulesenden Entitäten beschreibt und die konkrete Technologie abstrahiert, sollte angestrebt werden. Eine weitere Grundvoraussetzung ist die explizite Trennung von Entitäten und Relationen, wodurch Entitäten einfacher über Netzwerke übertragen werden. D.h. sie dürfen keine Methoden, wie z.B. *getChildren()* besitzen. Dafür können Relationen über das Query-Objekt abgefragt werden (z.B.: getChildrenFor(Object)).

Metaprogramme würden sich zur Umsetzung solcher Query-Objekte sehr gut eignen. Jedoch konnte leider aufgrund der Komplexität des Themas im Zuge dieser Diplomarbeit nicht näher darauf eingegangen werden.

KAPITEL 11

Metaprogramme für AOP

Das Kapitel beschreibt wie Metaprogramme dazu eingesetzt werden können Elemente der aspektorientierten Programmierung (AOP) umzusetzen. Die aspektorientierte Programmierung (AOP) ist ein Programmierparadigma, welches auf die objektorientierte Programmierung aufbaut und versucht das Problem der Cross-Cutting-Concerns zu lösen.

Eine ausführlichere Beschreibung befindet sind im Abschnitt 2.4 „Aspektorientierte Programmierung“ auf Seite 20.

11.1 Aspect

Funktion

Das *Aspect*-Metaprogramm ermöglicht aspektorientierte Programmierung durch globale *Listener*-Methoden.

Beschreibung

Eine *Aspect*-Klasse kann *Listener*-Methoden definieren, welche beim Methodenaufruf von Instanzen ebenfalls aufgerufen werden. Die zu überwachenden Methoden werden mittels regulärer Ausdrücke von Typ- und Methodennamen definiert. Weiters kann festgelegt werden, ob sie beim Eintritt oder Austritt der Methode aufgerufen werden sollen. *Aspect*-Klassen können verwendet werden um Cross-Cutting-Concerns wie Logging, Tracing und Profiling zu realisieren.

Verwendung

- Deklaration einer *Aspect*-Klasse
  - Eine Klasse mit *@Aspect* annotieren.
- Deklaration von *Listener*-Methoden bei Methodeneintritt
  - Statische Methode mit *@Before(method=<String>, type=<String>)* annotieren.
  - Der *method*-Parameter definiert den Typnamen mittels regulären Ausdrucks.
  - Der *type*-Parameter definiert den Methodennamen mittels regulären Ausdrucks.
- Deklaration von *Listener*-Methoden bei Methodenaustritt
  - Statische Methode mit *@After(method=<String>, type=<String>)* annotieren.
  - Der *method*-Parameter definiert den Typnamen mittels regulären Ausdrucks.
  - Der *type*-Parameter definiert den Methodennamen mittels regulären Ausdrucks.
- Verwendung
  - Die ersten beiden Parameter der *Listener*-Methode sind für die Aufrufer-Instanz und den Methodennamen reserviert.
  - Bei weiteren Parametern in der *Listener*-Methode, werden die alle Parameter der zu überwachenden Methode übergeben. Vorraussetzung dafür ist, dass alle zu überwachenden Methoden die selbe Signatur besitzen.

Beispiel vor Ausführung der Metaprogramme (Sourcecode-Basis)

Deklaration einer Aspect-Klasse

Die *Main*-Klasse wurde als *Aspect* deklariert und kann dadurch *Listener*-Methoden definieren. Die *traceBefore*-Methode wird am Beginn jeder Methode, welche mit *say* beginnt, aufgerufen. Die *traceAfter*-Methode wird am Ende jeder Methode, welche mit *say* beginnt, aufgerufen.

```
@Aspect
public class Main {
    public static void main(String[] args) {
        new ClassA().sayHelloB();
        new ClassB().sayHelloA();
```

```
    }

    @Before(method=".*say.*", type=".*")
    public static void traceBefore(Object a, String method) {
        System.out.println("Before: " + method);
    }

    @After(method=".*say.*", type=".*")
    public static void traceAfter(Object a, String method) {
        System.out.println("After: " + method);
    }
}
```

Erstellung passender Methoden

Die *ClassA*- und *ClassB*-Klasse implementieren jeweils eine eigene Variante einer *sayHello*-Methode, welche in das Muster der *Listener*-Methode passt.

```
public class ClassA {
    public void sayHelloB(){
        System.out.println("Hello B!");
    }
}
```

```
public class ClassB {
    public void sayHelloA(){
        System.out.println("Hello A!");
    }
}
```

Beispiel nach Ausführung der Metaprogramme (Sourcecode-Generat)

Generierung des Aufrufcodes

Das Metaprogramm generiert für alle Methoden, welche in das Muster der *Listener*-Methode passen, einen Aufrufcode. Für die *@Before*-Methode wurde eine einfache Aufrufzeile am Beginn der Methode eingefügt. Für die *@After*-Methode musste ein Trick angewendet werden um vorzeitige Returns behandeln zu können: Der gesamte Methodeninhalt wird in einen try-Block gelegt und der Aufrufcode in einen finally-Block.

```
public class ClassA {
    public void sayHelloB(){
        //************** begin of generated code ****************
        try { //from after
        Main.traceBefore(this,"sayHelloB"); //from before
        //************** end of generated code ****************

        System.out.println("Hello B!");

        //************** begin of generated code ****************
        } finally {
            Main.traceAfter(this,"sayHelloB");
        }
        //************** end of generated code ****************
```

```
    }
}
```

```
public class ClassB {
    public void sayHelloA(){
        //************** begin of generated code ****************
        try { //from after
        Main.traceBefore(this,"sayHelloA"); //from before
        //*************** end of generated code ****************

        System.out.println("Hello A!");

        //************** begin of generated code ****************
        } finally {
            Main.traceAfter(this,"sayHelloA");
        }
        //*************** end of generated code ****************
    }
}
```

Sourcecode des JCM-Metaprogramms

Das Metaprogramm generiert für jede *@Before*- und *@After*-Methode der *Aspect*-Klassen einen statischen Aufruf in die zu überwachenden Methoden.

```
public class MetaAspect implements MetaProgrammer {
    public void run(MpManager mp){
        //for all classes with @Aspect−annotation
        for(MpClass aspect : mp.getClassesByAnnotation("meta.aop.annotations.Aspect")){

            //for each method with @Before− or @After−annotation
            for(MpMethod method : aspect.getMethods()){
                MpAnnotation b = method.getAnnotation("meta.aop.annotations.Before");
                if(b != null) generateCode(mp, aspect, method, b);

                MpAnnotation a = method.getAnnotation("meta.aop.annotations.After");
                if(a != null) generateCode(mp, aspect, method, a);
    } } }

    private void generateCode(MpManager mp, MpClass aspect, MpMethod method, MpAnnotation
        annotation)
    {
        //read values for annotation
        String methodName = annotation.getString("method");
        String type = annotation.getString("type");

        //create parameter array without first and second parameter(caller−object and methodname)
        MpParameterArray array = mp.createParameterArray();
        for(int i = 2 ; i < method.getParameters().length ; i++)
            array.addParameter(method.getParameters()[i]);

        //for each class, which:
        for(MpClass clazz : mp.getClasses()) {
            if( !clazz.isInterface() //is no interface
                && clazz.getAnnotation("meta.aop.annotations.Aspect") == null //is no aspect
                && clazz.getQualifiedName().matches(type)) //matches the search−pattern
                {
```

```
            //for each method which matches the search−pattern
            for(MpMethod m : clazz.getMethods()){
                if(m.getName().matches(methodName)) {

                    MpCode callCode = mp.createCode();
                    //method parameters are optinal
                    if(method.getParameters().length > 2)
                        callCode.line(aspect.getName() + "." + method.getName() +
                                "(this,\"" + m.getName() + "\"," + array.codeOfParameterNameList() +");");
                    else
                        callCode.line(aspect.getName() + "." + method.getName() + "(this,\"" + m.getName() + "\");
                            ");

                    //annotation is @Before
                    if(annotation.getName().equals("Before"))
                        clazz.insertFirst(m,callCode);
                    //annotation is @After
                    else
                        clazz.insertLast(m,callCode);
} } } } } }
```

Evaluierung

Der Zeitbedarf des JCM-Metaprogramms bei 80 Klassen/Interfaces in der Sourcecode-Basis liegt bei ca. **410** Millisekunden.

Der Anteil des generierten Codes wird durch die Gegenüberstellung der Zeichenanzahl vor und nach der Generierung (ohne Kommentare) aller Java-Dateien (Ausnahme: Main.java) berechnet.

Anzahl Zeichen vor Generierung	164
Anzahl Zeichen nach Generierung	428
Anteil generierter Code	61,7%

Fazit

- Effizientes Werkzeug zur Analyse eines Programms.
- Extreme Codestreuung wird durch Metaprogramme beherrschbar.
- Durch statische Codegenerierung sind mögliche Seiteneffekte besser ersichtlich.

11.2 Logging

Funktion

Das *Logging*-Metaprogramm baut auf dem *Aspect*-Metaprogramm auf und ermöglicht eine gute Abstraktion des individuellen Loggings von Klassen.

Beschreibung

Klassen verwenden eine private *log*-Methode für das Logging. Mit Hilfe einer *Listener*-Methode werden diese *log*-Methoden abgefangen und verarbeitet.

Verwendung

- Deklaration einer *Logging*-Klasse
  - Eine Klasse mit *@Logging* annotieren.
- Deklaration einer *Aspect*-Klasse zum Empfangen der Log-Nachrichten
  - Eine Klasse mit *@Aspect* annotieren.
  - Eine statische Methode mit *Before(type=„.*“,method=„log“)* annotieren.
- Verwendung
  - Die *Logger*-Klasse verwendet *log*-Methode für Log-Nachrichten.
  - Die *Listener*-Methode leitet die Log-Nachricht weiter. (Z.B.: System.out)

Beispiel vor Ausführung der Metaprogramme (Sourcecode-Basis)

Deklaration von *Logging*-Klassen

Die *ClassA*- und *ClassB*-Klasse werden als *Logging*-Klasse definiert. Dadurch können sie auf eine private *log*-Methode zugreifen.

```
@Logging
public class ClassA {
   public void sayHelloWorld() {
      log("HelloWorldA");
   }
}
```

```
@Logging
public class ClassB {
   public void sayHelloWorld() {
      log("HelloWorldB");
   }
}
```

Deklaration einer *Aspect*-Klasse zum Empfangen der Log-Nachrichten

Die *Main*-Klasse wird als *Aspect*-Klasse deklariert und definiert eine *Listener*-Methode für den Empfang für Log-Nachrichten.

```
@Aspect
public class Main
{
   public static void main(String[] args)
   {
      new ClassA().sayHelloWorld();
      new ClassB().sayHelloWorld();
   }

   @Before(method="log", type=".*")
   public static void printLog(Object a, String method, String message) {
      System.out.println(a + ": " + message);
   }
}
```

Beispiel nach Ausführung der Metaprogramme (Sourcecode-Generat)

Generierung des Aufrufcodes

Das *Logging*Metaprogramm generiert für alle annotierten Klassen eine private *log*-Methode. Das *Aspect*Metaprogramm generiert anschließend für alle *log*-Methoden einen Aufrufcode, welcher den Parameter der *log*-Methode an die *Listener*-Methode übergibt.

```
@Logging
public class ClassA
{
   public void sayHelloWorld() {
      log("HelloWorldA");
   }

   //************** begin of generated code ****************
   private void log(String message) {
      Main.printLog(this,"log",message);
   }
   //************** end of generated code ****************
}
```

```
@Logging
public class ClassB
{
   public void sayHelloWorld() {
      log("HelloWorldB");
   }

   //************** begin of generated code ****************
   private void log(String message) {
      Main.printLog(this,"log",message);
   }
   //************** end of generated code ****************
}
```

Sourcecode des JCM-Metaprogramms

Das Metaprogramm generiert für eine Klasse mit @*Logging*-Annotation eine private *log*-Methode

```
public class MetaLogger implements MetaProgrammer
{
    public void run(MpManager mp) {
        for(MpClass clazz : mp.getClasses())
        {
            //for each class with @Logging−Annotation
            MpAnnotation annotation = clazz.getAnnotation("meta.patterns.logger.annotations.Logging");
            if(annotation != null) {
                //create private log−method
                clazz.createMethod(mp.createCode()
                        .line("private void log(String message)")
                        .startBlock().endBlock());
            }
        }
    }
}
```

Evaluierung

Der Zeitbedarf des JCM-Metaprogramms bei 80 Klassen/Interfaces in der Sourcecode-Basis liegt bei ca. **240** Millisekunden.

Der Anteil des generierten Codes wird durch die Gegenüberstellung der Zeichenanzahl vor und nach der Generierung (ohne Kommentare) aller Java-Dateien (Ausnahme: Main.java) berechnet.

Anzahl Zeichen vor Generierung	166
Anzahl Zeichen nach Generierung	308
Anteil generierter Code	46,1%

Fazit

- Konkrete Logger-Implementierung wird nicht im Code verstreut.
- Zum Ausführen der Klassen muss keine Logger-Implementierung vorhanden sein. (Ergebnisloser Aufruf der *log*-Methode)
- Durch die @*Logging*-Annotation ist klar ersichtlich, welche Klassen Log-Nachrichten produzieren.
- Durch die *Listener*-Methode wird automatisch eine Referenz des Senders übertragen.

KAPITEL 12

Zusammenfassung der Ergebnisse

Entwicklung von JCM-Metaprogrammen

JCM-Metaprogramme können sehr schnell entwickelt werden und werden praktisch nur während der Anwendung getestet. Sie gleichen einfachen, imperativen Skripts, welche sehr geradlinig programmiert werden können. Planung wird jedoch für die Konzeption der deklarativen Programmierung (Annotationen) benötigt, es gibt oft mehrere Alternativen wie Annotationen gesetzt werden können.

Beispiel:
Das Observer-Pattern kann in zwei Varianten umgesetzt werden. Die Erste ist, dass der Publisher explizit annotiert wird und mittels privater Methoden Events auslösen kann. Die Zweite ist, dass Observer explizit annotiert werden und Methoden unsichtbar überwachen.

Weiters muss immer auf die mögliche Zusammenarbeit von unterschiedlichen Metaprogrammen geachtet werden. Dabei haben sich zwei Richtlinien als nützlich erwiesen:

- Annotationen sollten auch für Interfaces gesetzt werden können.
- Generierte Methoden müssen unterschiedliche Namen haben.

Entwicklung mit Hilfe von JCM-Metaprogrammen

Die Entwicklung mit JCM-Metaprogrammen funktionierte sehr gut. Besonders die Programme für Design-Patterns finden oft und schnell Anwendung, so müssen nur ein paar Annotationen gesetzt werden um eine Publish/Subscribe-Kommunikation umzusetzen. Multimethoden wären ohne Metaprogramme fast unwartbar, auch die gleichzeitige Verwendung unterschiedlicher Design-Patterns in einer Klasse war kein Problem. Oft ergänzen sie sich sogar sehr gut, z.B. Composite- und Visitor-Pattern.
Ein weiterer Vorteil ist, dass JCM-Metaprogramme vollkommen technologieunabhängig sind, so können Metaprogramme in Kombination mit jedem Framework verwendet werden. Die einzige Vorraussetzung ist, dass mit Java-Klassen gearbeitet wird.

Mustererkennung durch Metaprogramme

Während der Entwicklung von Metaprogrammen für Design-Patterns wurden viele Gemeinsamkeiten erkannt. Oft sind Patterns Kombinationen von einfacheren Mustern:

- Publisher: ListAccess + Delegation
- Strategy: Setter + Constructor + Delegation
- Composite: ListAccess + Delegation
- ...

Wenn viele Programme mit Hilfe von Metaprogrammen entwickelt werden würden, könnte sich nach einiger Zeit ein Kern an oft gebrauchten Mustern herauskristallisieren, welcher in die Konzeption neuer Programmiersprachen miteinbezogen werden könnte (siehe Abschnitt 14 „Ausblick“ auf Seite 109).

Design-Patterns wurden schon in einigen Arbeiten ([1], [36]) hinsichtlich ihrer Bestandteile analysiert, jedoch basieren diese Erkenntnisse auf zeitaufwendigen Überlegungen. Mit Hilfe von Metaprogrammen ergeben sich die Bestandteile aufgrund des Refactoring-Vorgangs (Auslagerung und Wiederverwendung durch Methoden, Klassen, ...) von selbst.

Statische vs. Dynamische Metaprogrammierung

Die in [14] aufgestellte Behauptung, dass die erweiterten Eigenschaften dynamischer Metaprogrammierung nicht oft benötigt werden, konnte überwiegend bestätigt werden. Während der gesamten Entwicklung mit statischer Metaprogrammierung ist es zu keinem Fall gekommen, wo die Umsetzung eines Metaprogramms aufgrund der statischen Code-Generierung eingeschränkt wurde. Überraschenderweise war oft das Gegenteil der Fall. Die Möglichkeiten zur Umsetzung waren so vielfältig, dass es schwierig war die passendste Lösung auszuwählen. Der Grund hierfür liegt darin, dass sich auch statisch generierter Code dynamisch verhalten kann.
Der Aufrufcode des Listeners wird zwar statisch in die zu überwachende Methode eingefügt, jedoch kann er mit einer zusätzlichen if-Anweisung dynamisch aktiviert und deaktiviert werden. Ein zusätzlicher Vorteil ist, dass dadurch die Nebeneffekte explizit im Source-Code dargestellt werden, was z.B. bei AspectJ nur mittels spezieller Debugger während der Laufzeit möglich ist.

JCM-Metaprogramme als normale Java-Programme mit Bibliothek

JCM-Metaprogramme als normale Java-Klassen mit spezieller API-Anbindung (Bibliothek) umzusetzten erwies sich überwiegend als gute Entscheidung. Der Grund hierfür liegt darin, dass die bestehende Infrastruktur von Java verwendet wurde, so können mit Hilfe von Methoden und Vererbung Metaprogramme besser organisiert werden. Auch die gewohnte Umgebung der Sprache ermöglicht eine sehr schnelle und intuitive Programmierung von Metaprogrammen.
In speziellen Anwendungsfällen (viel Codegenerierung, kaum Steuerbefehle) ist die Übersichtlichkeit und der Umfang im Vergleich zu templatebasierten Ansätzen schlechter, da in JCM-Metaprogrammen der zu generierende Code zwischen vielen Bibliotheksmethoden untergeht.
Z.B.: *startBlock().line("CODE").endBlock()*.
Hier muss jedoch angemerkt werden, dass templatebasierte Ansätze wesentlich geringere Mög-

lichkeiten bieten und bei weitem nicht für alle Anwendungsfälle ausreichen würden. JCM-Metaprogrammen sind aber so flexibel, dass sie auch templatebasierte Ansätze umsetzen könnten indem sie als Interpreter für XML-basierte Templates verwendet werden. So könnte für diese Anwendungsfälle die Übersichtlichkeit verbessert werden.

Bewertung der Implementierung von JCM

Mit der derzeitigen Implementierung von JCM können Metaprogramme großteils fehlerfrei erstellt und ausgeführt werden. Jedoch gibt es in einigen Bereichen Verbesserungspotential. Die durchschnittliche Laufzeit bei 80 Klassen und 12 JCM-Metaprogrammen liegt bei ca. 5 bis 6 Sekunden (siehe Tab. 12.1 „Evaluierung: Zeitbedarf der JCM-Metaprogramme bei 80 Klassen in der Sourcecode-Basis“ auf Seite 103). JCM-Metaprogramme führen oft Schleifen über alle Klassen und Methoden aus, wodurch jede weitere Klasse oder Methode dazu führt, dass jedes einzelne Metaprogramm langsamer wird. Dadurch steigt die Laufzeit exponentiell mit jeder weiteren Klasse/Methode/Variable. Dieser Umstand könnte durch intelligentere Suchalgorithmen (anstatt Schleifen) reduziert werden.

Weiters werden statische Fehler (wie z.B. Schreibfehler) erst nach der Metaprogrammdurchführung angezeigt. Auch die eingebauten Refactoring-Tools der IDE funktionieren nicht mehr. Diese Fehler könnten durch verbesserte Integration in die IDE behoben werden.

Name des Metaprogramms	Zeit in Millisekunden
GetSet	421
Constructor	326
ListAccess	633
Publisher	325
Strategy	331
Composite	751
Visitor	537
Command	634
Multimethod	480
Data Access Object	638
Aspect	410
Logging	240
Gesamt	5726

Tabelle 12.1: Evaluierung: Zeitbedarf der JCM-Metaprogramme bei 80 Klassen in der Sourcecode-Basis

Beantwortung der Forschungsfrage

Auf folgende konkrete Forschungsfrage sollte am Ende eine Antwort gegeben werden können:
Kann die Automatisierung von Software-Mustern die Programmierung ohne Frameworks in Hin-

blick auf Produktivität, Erweiterbarkeit und Wartbarkeit verbessern?

Produktivität
Folgende Erkenntnisse lassen auf eine Verbesserung der Produktivität schließen:

- Metaprogramme reduzieren die manuelle Codierungsarbeit teilweise enorm (siehe Tab. 12.2 „Evaluierung: Anteil an generierten Code“ auf Seite 105). Dadurch können Programme schneller geschrieben und getestet werden (Rapid-Prototyping).
- Metaprogramme ermöglichen deklarative Programmierung, welche den Überblick und die Verständlichkeit von Programmen fördert.

Wart- und Erweiterbarkeit
Folgende Erkenntnisse lassen auf eine Verbesserung der Wart- und Erweiterbarkeit schließen:

- Metaprogramme ermöglichen eine zentrale Generierung von Code. Für eine Änderung muss oft nur mehr eine Stelle im Code verändert werden.
- Metaprogramme lösen Erweiterbarkeitsprobleme, bei denen beim Hinzufügen eines neuen Elements bestehende Klassen angepasst werden müssen, durch automatische Synchronisation (z.B. Command- und Visitor-Pattern).
- Metaprogramme ermöglichen Abstraktionen von konkreten Technologien, wodurch diese bei Software-Evolution besser getauscht werden können.

Vorsicht gilt jedoch bei übermässiger Verwendung von Metaprogrammen. Die Mächtigkeit verleitet gern dazu zu viel Zeit in Automatisierungen zu investieren.
Besser ist die Vorgangsweise traditionell mit der Programmierung zu beginnen und bei wiederholter Verwendung von Copy/Paste/Replace den Einsatz eines Metaprogrammes in Betracht zu ziehen.

Name	Anzahl Zeichen vor Generierung	Anzahl Zeichen nach Generierung	Anteil generierter Code
GetSet	86	232	62,9
Constructor	78	136	42,6
ListAccess	118	396	70,2
Publisher	858	1762	51,3
Strategy	410	784	47,7
Composite	311	755	58,8
Visitor	714	1547	53,8
Command	703	1230	42,8
Multimethod	617	1772	65,2
Data Access Object	182	480	62,1
Aspect	164	428	61,7
Logging	166	308	46,1
Gesamt	4407	9830	55,2

Tabelle 12.2: Evaluierung: Anteil an generierten Code

KAPITEL 13

Fazit

Zu Beginn wurden alle theoretischen Konzepte und Begriffe dieser Diplomarbeit erklärt. Diese umfassten Boilerplate-Code, Design-Patterns, aspektorientierte Programmierung und Java-Annotationen. Anschließend folgte eine Literaturrecherche zum State-of-the-Art, welche zeigte, dass zwar viele Forschungsarbeiten in diesem Bereich existieren, aber sich keine mit denselben Zielsetzungen wie diese Diplomarbeit auseinandersetzt. Jedoch konnte die Metaprogrammierung als vielversprechende Richtung eingegrenzt werden.

Im nächsten Kapitel wurde der Begriff Metaprogrammierung und dessen Geschichte ausführlich beschrieben. Anschließend wurden aktuelle Ansätze der Metaprogrammierung vorgestellt und erklärt, warum sie nicht die Anforderungen erfüllen können. Danach wurden Probleme der Metaprogrammierung beschrieben und Lösungsansätze aufgezeigt, welche als nichtfunktionale Anforderungen in den Anforderungskatalog aufgenommen werden.

In einer Analyse wurde die Problemstellung anhand konkreter Beispiele näher beschrieben. Es wurde gezeigt, wie Annotationen zur deklarativen Programmierung eingesetzt werden können. Gleichzeitig wurde analysiert, welche Funktionen benötigt werden, um das konkrete Problem lösen zu können.

Im Anschluss wurde eine konkrete Lösung namens *Code-Monkey* (CM) vorgestellt, welche durch Schlussfolgerungen aus den Anforderungen der Problemanalyse- und Recherche-Arbeit erarbeitet wurde. Der Name bezieht sich auf die Funktion des Werkzeugs: Die Durchführung von trivialer und repetitiver Codierungsarbeit. Folgende Punkte machen den Unterschied zu alternativen Lösungsansätzen aus:

Mächtigkeit: Funktionen werden mittels Bibliothek bereitgestellt und können in beliebiger Anzahl und Reihenfolge durchgeführt werden.

Vertraute Arbeitsumgebung: Metaprogramme werden in der gleichen Programmiersprache geschrieben.

Sichtbare Ergebnisse: Metaprogramme bearbeiten Sourcecode-Dateien, die Ergebnisse sind klar ersichtlich.

Sprachunabhängig: Das Funktionsprinzip von CM kann bei fast allen objektorientierten Programmiersprachen angewendet werden.

Anschließend wurde die konkrete Umsetzung *Java-Code-Monkey* des zuvor beschrieben Lösungansatzes entwickelt und beschrieben. Sie wurde für die Programmiersprache Java auf Basis von Eclipse entwickelt und ermöglicht die Erstellung von Metaprogrammen.

Danach wurde mit Hilfe dieses Prototyps versucht verschiedene Muster (Boilerplate-Code, Design-Patterns,...) zu automatisieren. Diese wurden stichprobenartig ausgewählt um einen statistischen Überblick zu erhalten, ob Muster prinzipiell gut oder schlecht generiert werden können.

Abschließend wurden alle Erfahrungen und Ergebnisse dokumentiert und auf folgende konkrete Forschungsfrage eine Antwort gegeben:

Kann die Automatisierung von Software-Mustern die Programmierung ohne Frameworks in Hinblick auf Produktivität, Erweiterbarkeit und Wartbarkeit verbessern?

Es konnten alle ausgewählten Muster sehr gut automatisiert werden, welche nun mittels deklarativer Programmierung (Annotationen) gesteuert werden. Dadurch konnte die Produktivität, Wart- und Erweiterbarkeit aufgrund von reduzierter Codierungsarbeit, besserer Übersicht und zentraler Verwaltung verbessert werden. Statische Codegenerierung macht die Funktion von Metaprogrammen sichtbar und erleichtert das Debugging. Eigenschaften von dynamischer Metaprogrammierung wurden während der Entwicklung nicht vermisst. Weiters führt die Erstellung von Metaprogrammen zu einem besseren Verständnis von Mustern: So konnten viele Gemeinsamkeiten zwischen Entwurfsmustern erkannt werden. Eventuell könnte sich nach einiger Zeit ein Kern an oft gebrauchten Mustern herauskristallisieren, welcher in die Konzeption neuer Programmiersprachen miteinbezogen werden kann.

KAPITEL 14

Ausblick

Einer der Motivationsgründe für diese Diplomarbeit war die besorgniserregende Entwicklung von Frameworks in statischen Sprachen wie Java oder C#. Die Entwicklung von Anwendungen wird zunehmend auf Frameworks ausgelagert, wodurch die ProgrammiererInnen immer abhängiger werden und verlernen Probleme selbst zu lösen.

Die Situation erinnert an ein chinesisches Sprichwort:

> „Gib einem Hungernden einen Fisch, und er wird einmal satt, lehre ihn Fischen, und er wird nie wieder hungern.“ - nach Laotse

Übertragen auf die Programmierung könnte es wie folgt lauten:

> „Gib einem Programmierer ein Framework, und er wird ein Problem lösen können, lehre ihn sich selbst zu helfen und er wird jedes Problem lösen können.“

Die Metaprogrammierung ist eine Alternative die dem Programmierer mehr Möglichkeiten gibt sich selbst zu helfen. Leider waren die bisherigen Ansätze sehr kompliziert oder konnten zu undurchsichtigen Seiteneffekten führen. Daher war es ein Ziel dieser Diplomarbeit die Metaprogrammierung zu vereinfachen um sie so für den Mainstream interessanter zu machen.

Das Faszinierendste an der Softwareentwicklung ist die Tatsache, dass sie nur eine sehr, sehr hohe Abstraktion von 1er und 0er sowie der Logik-Gattern *And*, *Or* und *Not* ist. Aus Logikgatter wurden Rechenfunktionen, der *Goto*-Befehl wurde zu *If*-Abfragen und *For*-Schleifen und erste Wiederverwendung wurde mittels Methoden ermöglicht. Alles entwickelte sich intuitiv und schrittweise zu immer höher werdenden Abstraktionen. Jedoch geriet diese Entwicklung nach der objektorientierten Programmierung ins Stocken. Seitdem Frameworks verwendet werden, haben sich die statischen Programmiersprache Java und C# nur wenig weiterentwickelt.

Metaprogrammierung wäre eine Möglichkeit der Programmiersprache wieder die Macht zu geben sich schrittweise zu einer höheren Programmiersprache weiterzuentwickeln:

- Als ersten Schritt müssten Metaprogramme in vielen praktischen Projekten eingesetzt werden um Erfahrungen zu sammeln. Allein in dieser Diplomarbeit konnten bereits viele Gemeinsamkeiten zwischen Design-Patterns erkannt werden (siehe Abschnitt 12 „Mustererkennung durch Metaprogramme“ auf Seite 102).

- Nach einiger Zeit könnte sich ein Kern an oft gebrauchten Mustern herauskristallisieren, welcher in die Konzeption neuer Programmiersprachen miteinbezogen werden kann. Besonders interessant wäre eine leichtgewichtige imperative Sprache, welche bereits auf Metaprogrammierung ausgelegt ist und sich dadurch selbst weiterentwickeln kann.

Literaturverzeichnis

[1] Ellen Agerbo and Aino Cornils. How to preserve the benefits of design patterns. In *Proceedings of OOPSLA*, pages 134–143. ACM Press, 1998.

[2] Marty Alchin. *Pro Python*. Apress, Berkely, CA, USA, 1st edition, 2010.

[3] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)*. Oxford University Press, August 1978.

[4] Andrei Alexandrescu. *Modernes C++ Design*. mitp-Verlag, 2003.

[5] Philipp Bachmann. Static and metaprogramming patterns and static frameworks: a catalog. an application. In *Proceedings of the 2006 conference on Pattern languages of programs*, PLoP '06, pages 17:1–17:33, New York, NY, USA, 2006. ACM.

[6] Kent Beck and Ward Cunningham. Using Pattern Languages for Object-Oriented Programs. Technical Report CR-87-43, Apple Computer, Inc. and Tektronix, Inc., 1987.

[7] Stefan Bünnig, Peter Forbrig, Ralf Lämmel, and Normen Seemann. A programming language for design patterns. In *Proceedings of the GI-Jahrestagung 1999, Informatik 99, Reihe Informatik Aktuell*, pages 400–409. Springer-Verlag, 1999.

[8] Jan Bosch. Design patterns frameworks: On the issue of language support. In *Object-Oriented Technologys*, volume 1357 of *Lecture Notes in Computer Science*, pages 133–136. Springer Berlin Heidelberg, 1998.

[9] Jan Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 11:18–32, 1998.

[10] Andy Bulka. Design pattern automation. In *Proceedings of the 2002 conference on Pattern languages of programs - Volume 13*, CRPIT '02, pages 1–10, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[11] Shigeru Chiba. Open c++ programmer's guide. University of Tokyo, Faculty of Science, Department of Information Science, 1993.

[12] Shigeru Chiba. Javassist - A Reflection-based Programming Wizard for Java. In *International Business Machines Corp*, page http://www.javassist, 1998.

[13] Markus Dahm. Byte code engineering. In *JIT99*, pages 267–277. Springer, 1999.

[14] Christof Lutteroth Dirk Draheim. An analytical comparison of generative programming technologies technical report b-04-02. Institute of Computer Science, Freie Universität Berlin, 2004.

[15] Ghizlane El Boussaidi and Hafedh Mili. A model-driven framework for representing and applying design patterns. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 1, pages 97–100. IEEE, 2007.

[16] Maged Elaasar, LionelC. Briand, and Yvan Labiche. A metamodeling approach to pattern specification. In *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 484–498. Springer Berlin Heidelberg, 2006.

[17] Frank Eller. *Visual C# 2008*. Addison Wesley, 2008.

[18] Peter Forbrig and Ralf Lämmel. Programming with patterns. In *TOOLS-USA 2000. IEEE*, pages 159–170. IEEE, 2000.

[19] Ira R. Forman, Nate Forman, Dr. John Vlissides Ibm, Ira R. Forman, and Nate Forman. Java reflection in action, 2004.

[20] Marilyn Finnie Frank Budinsky, Patsy Yu, and John Vlissides. Automatic code generation from design patterns. *IBM Systems Journal*, pages 151–171, 1996.

[21] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[22] Joseph D Gradecki and Jim Cole. *Mastering Apache Velocity*. Wiley.com, 2003.

[23] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. *SIGPLAN Not.*, 37(11):161–173, November 2002.

[24] Robert Hirschfeld, Ralf Lämmel, and Matthias Wagner. Design patterns and aspects modular designs with seamless run-time integration. In *Univercity of Essen*, page 3, 2003.

[25] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[26] Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[27] Paul Klint, Tijs Storm, and Jurgen Vinju. Easy meta-programming with rascal. In *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Lecture Notes in Computer Science*, pages 222–289. Springer Berlin Heidelberg, 2011.

[28] Eugene Kuleshov. Using the asm framework to implement common java bytecode transformation patterns. *Aspect-Oriented Software Development*, 2007.

[29] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *ACM SIGPLAN Notices*, volume 38, pages 26–37. ACM, 2003.

[30] Jo A. Lawless and Molly M. Miller. *Understanding CLOS: the Common LISP object system*. Digital Press, Newton, MA, USA, 1991.

[31] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960.

[32] Dmitry Nizhegorodov. Jasper: Type-safe compile-time reflection language extensions and mop-based templates for java. In *Proceedings of ECOOP 2000 Workshop on Reflection and Metalevel Architectures*, 2000.

[33] Renaud Pawlak. Spoon: Compile-time annotation processing for middleware. *IEEE Distributed Systems Online*, 7(11), November 2006.

[34] Paolo Perrotta. *Metaprogramming Ruby*. Pragmatic Bookshelf, 1st edition, 2010.

[35] Eduardo Kessler Piveta and Luis Carlos Zancanella. Observer pattern using aspect-oriented programming. Universidade Federal de Santa Catarina, 2003.

[36] Johannes R Sametinger and Rudolf K Keller. Compositional design reuse. In *VIII Congreso Argentino de Ciencias de la Computación*, 2002.

[37] Vytautas Stuikys and Robertas Damasevicius. *Meta-Programming and Model-Driven Meta-Program Development: Principles, Processes and Techniques (Advanced Information and Knowledge Processing)*. Springer, 2012.

[38] G.T. Sullivan. Advanced programming language features for executable design patterns. technical report aim-2002-005. *MIT Artificial Intelligence Laboratory*, 2002.

[39] Michiaki Tatsubori, Shigeru Chiba, Marc Killijian, and Kozo Itano. Openjava: A class-based macro system for java. In *Reflection and Software Engineering*, pages 117–133. Springer-Verlag, 2000.

[40] Google Trends. Metaprogramming. http://www.google.at/trends/explore?q=metaprogramming, 2013. [Online; letzter Zugriff am 16.11.2013].

[41] Daniel von Dincklage. Making patterns explicit with metaprogramming. In *Proceedings of the 2nd international conference on Generative programming and component engineering*, GPCE '03, pages 287–306, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[42] Craig Walls and Norman Richards. *XDoclet in action*. Manning, 2003.

[43] Horst Zuse. Geschichte der Programmiersprachen. Technische Universität Berlin, Fachbereich Informatik, 1999.