

Erkennung von Entwurfsmustern

Minh Cuong Tran
tranmh(at)studi.informatik.uni-stuttgart.de
Hauptseminar Programmanalysen
12. Februar 2006

Zusammenfassung

Im Jahre 1995 wurden Entwurfsmuster für objektorientierte Programmierung von Gamma et al. [GHJV95] vorgestellt. Seitdem wurden sie in der breiten Praxis angewendet. Die Forschung beschäftigt sich seit ca. 1996 bis heute mit der Frage, wie man aus Programmcode die Architektur der Software in Form von Entwurfsmustern und damit das Programmverständnis wiedergewinnen kann.

In diesem Artikel versuche ich dem Leser einen Überblick zu geben, welche Ansätze unternommen worden sind und auf welche Schwierigkeiten man gestoßen ist.

1 Einführung

Die Wartung des objektorientierten Programmcodes ist schwierig und beansprucht die meiste Zeit eines Wartungsprogrammierers. Der Wartungsprogrammierer verwendet mehr als 50% seiner Zeit dafür Programmcode zu lesen und zu verstehen [Kos04].

Die Entwicklung einer Software in der Praxis ist meistens nicht vorbildlich. Es existiert keine Dokumentation, die Dokumentation ist unvollständig oder weicht ab von der vorhandene Software. Über die Gründe kann man spekulieren: Es ist wichtig, als Erster auf dem Markt zu sein, die Programmierer stehen unter Zeitdruck und können nicht genügend testen, geschweige denn dokumentieren. Die Dokumentation weicht von der Software ab, da Entwickler die Firma verlassen oder die Software viele Jahre von verschiedenen Entwicklergenerationen gepflegt wird.

Gamma, Helm, Johnson und Vlissides (auch bekannt als *Gang of Four*) haben 1995 das Buch Design Patterns [GHJV95] herausgebracht. Sie stellten 3 Arten von Entwurfsmustern vor (Erzeugungs-, Struktur- und Verhaltensmuster), die die objektorientierte Softwareentwicklung danach entscheidend geprägt haben. Ein Entwurfsmuster stellt eine wiederverwendbare Vorlage zur Problemlösung dar, konkrete Ausführungen finden sich heute oft in Programmcode wieder.

Die Forschung versucht nun, aus gegebenem undokumentierten Programmcode Muster nach Gamma et al. wiederzuerkennen. Das so gewonnene Wissen trägt zum Programmverstehen und zur Architekturerkennung bei und hilft bei der Wartung einen Überblick über Softwaresysteme zu verschaffen.

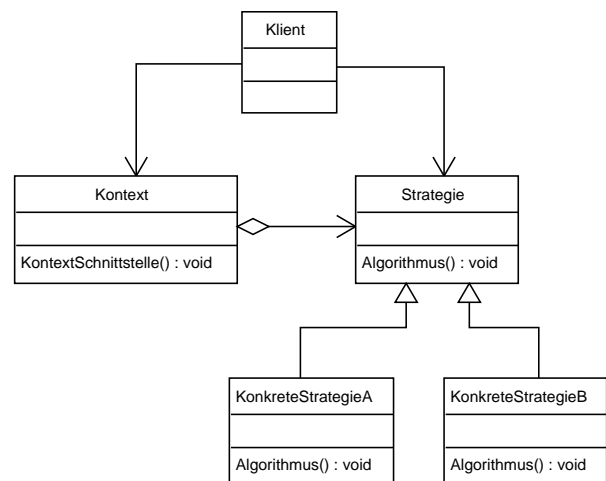


Abbildung 1. Strategie-Muster

Als Beispiel zeigt Abbildung 1 ein UML-Diagramm für das Strategie-Muster. Die abstrakte Klasse Strategie bietet ein Interface für einen Algorithmus an und versteckt die konkreten Implementierungen des Algorithmuses.

Im Abschnitt 2 versuche ich die verschiedenen Ansätze in chronologischen Reihenfolge zu erläutern. Anschließend vergleicht Abschnitt 3 diese Ansätze und Abschnitt 4 zieht ein Fazit.

2 Verschiedene Ansätze

2.1 Pat

Krämer und Prechelt waren die Pioniere in diesem Bereich und haben 1996 das System Pat [KP96] entwickelt.

Pat arbeitet folgendermaßen:

- Muster werden mittels UML-Diagrammen wie in Abbildung 1 eingegeben und dargestellt.
- Ein Programm P2prolog wandelt die UML-Diagramme in Prolog-Regeln um. Dabei entspricht ein Muster genau einer Prolog-Regel.
- Mittels struktureller Analyse von einem ooCASE-Werkzeug¹ werden C++ Header-Dateien in UML-Diagramme umgewandelt.
- Ein zweites Programm D2prolog wandelt die so entstandenen UML-Diagramme in Prolog-Fakten um.
- Die Prolog-Abfrage liefert auf Basis der Fakten und Regeln alle Klassentupel, die einem Muster entsprechen.

In Abbildung 2 ist ein Überblick über die Architektur des Pat-Systems skizziert

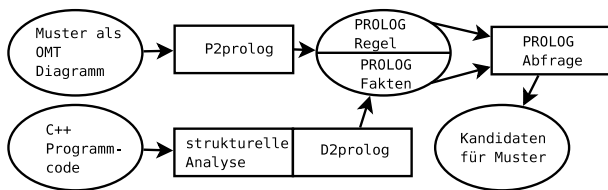


Abbildung 2. Architektur des Pat-Systems

2.1.1 Beispiel

Wir betrachten als Beispiel die folgende C++-Klasse:

```
class zPane:public zChildWin {
    zDisplay* curDisp;
    /* ... */

public:
    virtual void show(int = 1);
    /* ... */
}
```

Die daraus resultierenden Prolog-Fakten lauten:

```
class(concrete, zPane).
inheritance(zChildwin, zPane).
attribute(zPane, curDisp).
operation(virtual, selector, zPane,
    show, public, "int,",
    "void").
```

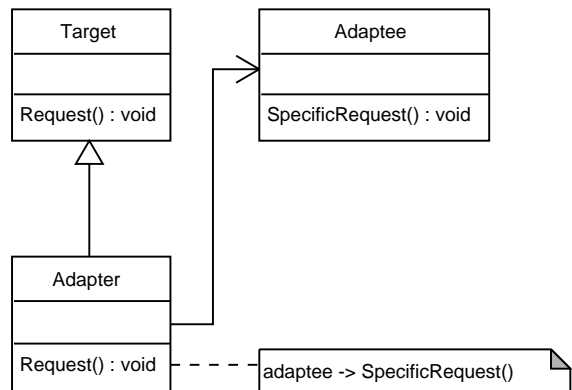


Abbildung 3. Adapter-Muster

Das Adapter-Muster in Abbildung 3 wird in folgende Prolog-Regel umgewandelt:

```
adapter(Target, Adapter, Adaptee) :-
    class(_, Target),
    class(concrete, Adapter),
    class(concrete, Adaptee),
    operation(_, _, Target, Request,
        _, _, _),
    operation(_, _, Adapter, Request,
        _, _, _),
    operation(_, _, Adaptee,
        SpecificRequest,
        _, _, _),
    inheritance(Target, Adapter),
    association(Adapter, Adaptee).
```

Nun erfolgt nur noch eine Prolog-Abfrage, um alle Tupel zu erhalten, die dem Adapter-Muster als Regel entsprechen:

```
?- adapter(X,Y,Z).
```

```
X = Parser
Y = Stack
Z = DoubleList
```

```
X = Main
Y = PrintDebug
Z = Printf
```

...

2.2 Teilgraph

Die Arbeit von Seemann und von Gudenberg im Jahr 1998 [SvG98] reduziert die Suche nach Entwurfsmuster

¹Object Oriented Computer Aided Software Engineering, Paradigm Plus v. 2.01

auf das Suchen eines gegebenen Mustergraphen M in einem großen Graphen G . Dabei versuchten sie den Graphen G mit möglichst vielen Informationen anzureichern, die man aus dem Quellcode ableiten kann. Zusätzliche Informationen in Form von Kanten wurden dann transitiv berechnet und dem Graphen G hinzugefügt.

Die formale Definition von G lautet:

$G = (V, E, \phi, \eta)$ mit
 $V = \text{CLASS} \cup \text{INTERFACE} \cup \text{METHOD}$
 $E \subseteq \text{CLASS} \times \text{CLASS} \cup \text{CLASS} \times \text{METHOD}$
 $\cup \text{CLASS} \times \text{INTERFACE}$
 $\cup \text{INTERFACE} \times \text{INTERFACE}$
 $\cup \text{METHOD} \times \text{METHOD} \cup \text{METHOD} \times \text{CLASS}$
 $\phi : V \rightarrow \text{TEXT} \times \text{TYPE}^n$
 $\eta : E \rightarrow \text{TEXT} \times \text{TEXT} \times \text{MULTIPLICITY}$

Klassen, Interfaces und Methoden sind die Knoten V des Graphen G . Kanten E werden zwischen Klassen und Klassen, Klassen und Methoden, Klassen und Interfaces, Interfaces und Interfaces, Methoden und Methoden, Methoden und Klassen gezogen. Die Funktion ϕ beschriftet die Knoten V mit ihren Namen, dem Rückgabewert und $n - 1$ Parametern. η beschriftet die Kanten mit:

- *extends* – Klasse \bar{A} erbt von der Klasse A oder Interface \bar{I} erbt von dem Interface I
- *implements* – Klasse A realisiert ein Interface I
- *references* – Klasse A_1 hat ein Attribut von Klasse A_2 oder Interface I
- *owns* – Klasse A besitzt eine Methode B
- *calls* – Methode B_1 ruft Methode B_2 auf
- *downcasts* – Klasse A_1 macht einen Downcast von Klasse A_2
- *creates* – Methode B erzeugt eine Instanz von Klasse A
- *assoc* – Klasse A_1 ruft und referenziert Klasse A_2
- *aggreg* – Klasse A_1 assoziiert und erzeugt Klasse A_2
- *delegates* – Methode B_1 ruft eine nicht lokale Methode B_2 auf

Wir gehen von einem Graphen G aus, den wir aus dem Frontend erhalten. Dieser Graph wird im nächsten Schritt mit mehr Informationen erweitert.

Für alle Methoden m_1, m_2 und alle Klassen c_1, c_2 gilt, falls m_1 m_2 aufruft und c_1 m_1 besitzt und c_2 m_2 besitzt,

so ziehen wir eine Aufrufkante von c_1 nach c_2 . Formal:

$$\forall m_1, m_2 \in \text{Methods} \forall c_1, c_2 \in \text{Classes} : \\ m_1 \text{ calls } m_2 \wedge c_1 \text{ owns } m_1 \wedge c_2 \text{ owns } m_2 \\ \Rightarrow c_1 \text{ calls } c_2$$

Die formale Schreibweise für weitere Regeln lauten:

$$\forall m_1 \in \text{Methods} \forall c_1, c_2 \in \text{Classes} : \\ m_1 \text{ creates } c_2 \wedge c_1 \text{ owns } m_1 \\ \Rightarrow c_1 \text{ creates } c_2$$

$$c_1 \text{ assoc } c_2 \iff c_1 \text{ calls } c_2 \wedge c_1 \text{ references } c_2 \\ c_1 \text{ aggreg } c_2 \iff c_1 \text{ assoc } c_2 \wedge c_1 \text{ creates } c_2$$

$$\forall c_1, c_2 \in \text{Classes} \forall m_1, m_2 \in \text{Methods} : \\ c_1 \text{ owns } m_1 \wedge c_2 \text{ owns } m_2 \wedge m_1 \text{ delegates } m_2 \\ \Rightarrow c_1 \text{ delegates } c_2$$

Mit dem angereicherten Graphen G ist es nun möglich, Kompositum-, Brücke- und Strategie-Muster zu finden. Wie schon erwähnt sucht man Subgraphen S in G , die für das Strategie-Muster beispielsweise folgende Bedingungen erfüllen müssen: Beginne mit Klasse C und Interface D , C delegiert dabei an D . Suche dazu alle Klassen B_i , die D implementieren.

2.3 Metriken

Einen anderen Ansatz verfolgten 1998 Antoniol, Fiutem und Cristoforetti [AFC98]. Das Neue an ihrer Arbeit ist die Benutzung von Metriken, um die potenziellen Kandidaten zu filtern und den Suchraum einzuschränken.

Im ersten Schritt wandelten sie den Programmcode und die Darstellung der Muster in Form von OMT Design² nach AOL³ um. AOL ist eine von ihnen eigens entwickelte Sprache zur Darstellung von Klassen und ihren Relationen; diese hat eine gemeinsame Semantik mit der Darstellung von Klassen in UML. Hier ein kurzes Beispiel für eine Klasse Proxy, die eine öffentliche Methode operation() besitzt:

```
CLASS    Proxy
        OPERATIONS
            PUBLIC operation();
```

Aus der AOL-Repräsentation werden mittels eines AOL-Parsers und eines Metriken-Werkzeugs die Metriken einzelner Klassen und ihre Relationen zueinander gewonnen. Hier die Metriken im Einzelnen:

²Object Modelling Technique, dt. Objektentwurfstechnik bei CASE

³Abstract Object Language

- Anzahl der öffentlichen, privaten und geschützten Attribute;
- Anzahl der öffentlichen, privaten und geschützten Methoden;
- Anzahl der Assoziationen, Aggregationen und Vererbungen;
- Gesamtzahl der Attribute, Methoden und Relationen.

In Abbildung 4 werden die nächsten Phasen veranschaulicht, um aus Klassen, ihren Metriken und Relationen Muster zu erkennen.

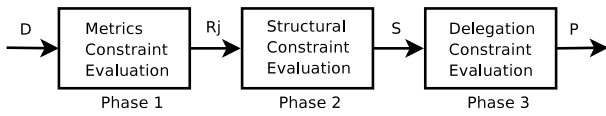


Abbildung 4. Mehrfachphasen-Filterung

2.3.1 Phase 1

Sei $p = (\langle e_1, \dots, e_k \rangle, \mathcal{R})$ ein Muster mit k Elementen (oder Klassen), \mathcal{R} eine Relation und sei $M_p = \langle m_1, \dots, m_k \rangle$ die zu p gehörige Matrix, wobei $m_i = (m_{i1} \dots m_{il})^T$ als Vektor die l Metriken in der Klasse m_i ausdrückt. Sei D ein Graph mit den Relationen als Kanten, Klassen als Knoten, der den Programmcode repräsentiert. Klassen in D , die für eine Klasse e_i im Muster in Frage kommen, werden wie folgt definiert für $i = 1 \dots k$:

$$C_i = \{x | x \in D \wedge \forall m_{ij} \in m_i : m_{ij_x} \geq m_{ij}\}$$

Durch die Bedingung $m_{ij_x} \geq m_{ij}$ werden nur Klassen in C_i aufgenommen, die in allen Metriken größer oder gleich die Klasse e_i sind. Zu beachten ist, dass die C_i nicht paarweise disjunkt sein müssen. Wir wählen aus der Menge der C_i die Menge C_{min} mit der minimalen Anzahl an Elementen, die dazu gehörige Klasse im Muster nennen wir e_{min} . Es folgt nun eine weitere Verfeinerung von C_i . Ausgehend von $j \neq min, j = 1, \dots, k$ und $y \in C_{min}$ gibt es einen kürzesten Weg von y nach $x \in C_j$. Kanten von y nach x können eine beliebige Beziehung sein. Nun definieren wir die Verfeinerung:

$$R_j(y) = \{x | x \in C_j \wedge K\u00fctWeg(x, y) = K\u00fctWeg(e_j, e_{min})\}$$

Die Bedingung garantiert, dass wir nur Knoten anschauen, die die gleiche Entfernung wie im Muster vorweisen. Zu nahe oder weiter entfernte Knoten scheiden dagegen aus. Um das Ganze zu veranschaulichen, haben wir in Abbildung 5 das Muster p dargestellt. Pro Klasse wurden die Metriken angegeben, e_{min} wurde durch C_{min} in Abbildung 6

bestimmt. Abbildung 6 zeigt, wie der Suchraum D durch die C_i aufgeteilt und eingegrenzt wurde. Aus Abbildung 6 gewinnen wir Informationen für eine weitere Eingrenzung $R_j(y)$ als zweidimensionales Array, welches in Tabelle 1 dargestellt wird.

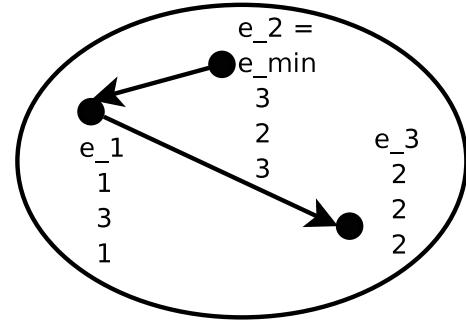


Abbildung 5. Muster p mit 3 Klassen

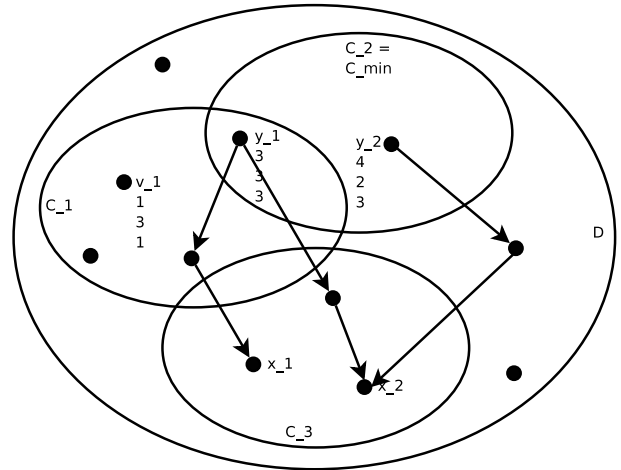


Abbildung 6. Graph D mit Klassen als Knoten

$R_j(y)$	y_1	y_2	\dots
$j = 1$	\dots	\dots	\dots
$j = 3$	$\{x_1, x_2\}$	$\{x_1\}$	\dots

Tabelle 1. Tabelle mit $R_j(y)$

2.3.2 Phase 2 und 3

Wir definieren nun R als eine Menge an k -Tupel, die tatsächlich als Kandidaten für das Muster p in Frage kämen. Anschaulich suchen wir für jede Spalte i und y_i aus Tabelle 1 k -Tupel, indem wir für jede Zeile in der Spalte i ein Ele-

ment aus dieser Zeile aussuchen. Dabei müssen alle Kombinationen betrachtet werden.

$$R = \{p | p = \langle r_1(y), \dots, r_k(y) \rangle \\ \wedge y \in C_{min} \\ \wedge \forall r_j(y) \in R_j(y)\}$$

Es sei \mathcal{R}_S die Teilmenge von \mathcal{R} , die nur strukturelle Beziehungen und z. B. keine Delegationsen enthält (und nehmen anschaulich z. B. alle Delegationskanten weg). S wird nun als eine Teilmenge von R definiert mit folgende Eigenschaften:

$$S = \{x | x = \langle x_1, \dots, x_k \rangle \in R \wedge \\ \forall r \in \mathcal{R}_S : r(e_p, e_s, \dots, e_t) \\ \Rightarrow r(x_p, x_s, \dots, x_t)\}$$

Die Menge S ist die Eingabe für den letzten Schritt. Zuerst werden alle strukturellen Anforderungen für das Muster geprüft. Kandidaten, die diesen Test bestanden haben, werden auf die im Muster vorkommende Delegationsen geprüft, um die Endkandidaten in der Menge P zu erhalten.

2.4 Mengenbildung

Tonella und Antoniol haben 1999 ein System entwickelt [TA99], welches *ohne* einer vordefinierten Muster-Bibliothek auskommt. Bemerkenswert an der Vorgehensweise ist, dass man generelle Informationen über das Zusammenspiel von Klassenmengen der Größe n bekommt, man erhält also gewollt auch schlechte Entwurfsmuster (sog. Anti-Pattern), die in der untersuchten Software vorliegen.

Die formale Schreibweise des Algorithmuses:
Anfangsschritt:

$$O_2 = \{(x, y) | (x, y)_l \in R\} \\ A_2 = \{(1, 2)_l | \exists (x, y)_l \in R\}$$

Schritt $k > 2$ (Schritt $k - 1$ sei gegeben):

$$O_k = \{(x_1, \dots, x_k) | (x_1, \dots, x_{k-1}) \in O_{k-1} \\ \wedge \exists j, 1 \leq j \leq k - 1, ((x_j, x_k)_l \in R \\ \vee (x_k, x_j)_l \in R)\} \\ A_k = A_{k-1} \cup \{(i, j)_l | \exists (x_1, \dots, x_k) \in O_k \\ \wedge ((i = k, 1 \leq j \leq k - 1) \vee (1 \leq i \leq k - 1, j = k)) \\ \wedge (x_i, x_j)_l \in R\}$$

R sei die Menge der Relationen. O_k ist eine Menge von Tupel aus Klassen/Objekte der Länge k . A_k stellt die Relationen zwischen einzelnen Objekten eines Tupels dar, die in O_k vorkommen. In jedem Schritt kommt eine weitere Klasse zum Tupel dazu. Die Idee dieses schrittweise Verfahren

ist ein systematisches Auffinden aller Klassenkonstellationen der Länge k .

Das Beispiel erläutert die Vorgehensweise. In der Abbildung 7 sind 9 Klassen mit Assoziationen und Vererbungen dargestellt.

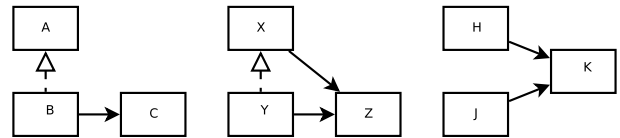


Abbildung 7. Struktur im Programmcode

Ausgehend von dieser Abbildung wird in der Tabelle 2 der Anfangsschritt des Algorithmuses vollzogen für O_2 und A_2 . Alle Paare mit irgendeiner Relation werden aufgelistet und sind in eine Zeile gruppiert, da sie gemeinsame Relationen haben.

	O_2 und A_2
c_1	$\{(B, A), (Y, X)\}, \{(1, 2)_e\}$
c_2	$\{(B, C), (Y, Z), (X, Z), (H, K), (J, K)\}, \{(1, 2)_a\}$

Tabelle 2. Anfangsschritt

Im nächsten Schritt werden aus 2er-Mengen $M_i \in O_2$ bzw. A_2 3er-Mengen gebildet, in dem man alle 2er-Mengen nimmt und ein Objekt $X \notin M_i$ hinzufügt, wobei eine Kante zwischen M_i und X existieren muss. Entsprechend werden sie wie im vorherigen Schritt gruppiert nach Relationen, die ebenfalls hinzugekommen sind. Tabelle 3 verdeutlicht diesen Schritt.

	O_3 und A_3
d_1	$\{(B, A, C), (Y, X, Z)\}, \{(1, 2)_e, (1, 3)_a\}$
d_2	$\{(B, C, A), (Y, Z, X)\}, \{(1, 2)_a, (1, 3)_e\}$
d_3	$\{(Y, X, Z)\}, \{(1, 2)_e, (1, 3)_a, (2, 3)_a\}$
d_4	$\{(X, Z, Y)\}, \{(1, 2)_a, (3, 1)_e, (3, 2)_a\}$
d_5	$\{(Y, Z, X)\}, \{(1, 2)_a, (1, 3)_e, (3, 2)_a\}$
d_6	$\{(X, Z, Y), (Y, Z, X), (H, K, J), (J, K, H)\}, \{(1, 2)_a, (3, 2)_a\}$

Tabelle 3. Schritt $k = 3$

Nach jedem Schritt muss noch eine Vereinfachung durchgeführt werden, da die Anzahl der Kombinationsmöglichkeiten explodieren würde. Die Zeilen d_1 und d_2 fallen zusammen wegen einer isomorphen Abbildung, genauso d_3, d_4 und d_5 . In d_6 fallen zwei Mengen weg, da die Relationen eine Symmetrie zulassen und die Mengen deshalb isomorph sind. Tabelle 4 stellt die Vereinfachung dar.

	O_3 und A_3 vereinfacht
d_1	$(\{(B, A, C), (Y, X, Z)\}, \{(1, 2)_e, (1, 3)_a\})$
d_3	$(\{(Y, X, Z)\}, \{(1, 2)_e, (1, 3)_a, (2, 3)_a\})$
d_6	$(\{(X, Z, Y), (H, K, J), \}, \{(1, 2)_a, (3, 2)_a\})$

Tabelle 4. Schritt $k = 3$ vereinfacht

Nach Schritt n (hier $n = 3$) erhält man Informationen über Musterarten, die Anzahl pro Muster und Hinweise darauf, welche Klassen konkret betroffen sind, veranschaulicht in Abbildung 8. Die linke Klassenkonstellation stellt eine typische Anordnung für das Adapter-Muster dar.

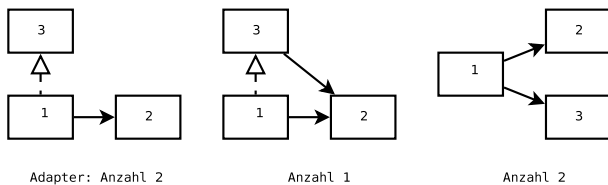


Abbildung 8. Gefundene Muster

Wir haben uns in unserem Beispiel auf Assoziationen und Vererbung beschränkt. Denkbar wären auch Relationen wie „Klasse A ruft eine Methode f von Klasse B auf“ oder „Klasse A besitzt eine Methode f “. Durch zusätzliche Relationen werden die gefundenen Muster feiner, jedoch sollte man auch die Explosion der Kombinationsmöglichkeiten bedenken.

2.5 Teilmuster

Einen ähnlichen und raffinierteren Ansatz als den von Seemann und von Guldenberg in 2.2 stellten Niere, Schäfer, Wadsack, Wendehals und Welsh 2002 [NSW⁺02] vor. Sie arbeiten direkt auf AST⁴ mit Knoten und Kanten und reichern diesen zusätzlich mit weiteren Knoten und Kanten nach bestimmten Regeln zu einem ASG⁵ an. Muster werden aus Teilmuster abgeleitet, die Herleitung eines Musters folgt einer Regel, die auf Teilregeln aufgebaut sind. Regeln in tieferen Ebenen basieren direkt auf Knoten des AST.

In Abbildung 9 wird das Kompositum-Muster dargestellt. Wir wollen anhand dieses Musters das Varianten-Problem einer 1:N Relation verdeutlichen (Kante zwischen Component und Composite).

Wir betrachten dazu den Programmcode in Listing 1. Hier sind drei Varianten einer 1:N Relation in Java implementiert. Die erste Variante benutzt Item als Array und bietet eine direkte Möglichkeit, Items zu lesen und zu verändern. Die zweite Variante benutzt eine Hash-Menge und die

⁴Abstract Syntax Tree

⁵Abstract Syntax Graph

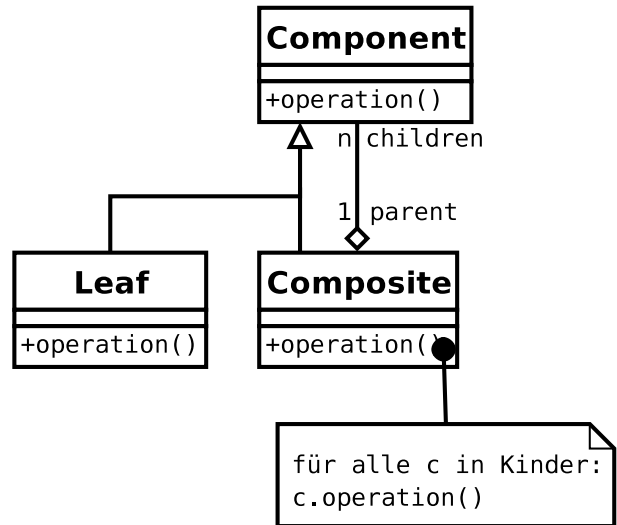


Abbildung 9. Kompositum-Muster

dritte einen Vektor. Alle drei Varianten würden zu einer formalen Beschreibung einer 1:N Relation in [GHJV95] passen. Um einen Knoten 1:N Relation zwischen Component und Composite hinzuzufügen zu können, ist es nötig für alle drei Varianten jeweils eine Regel aufzustellen, die direkt auf AST arbeitet. Leider sind diese drei Varianten nicht die einzige Möglichkeit eine 1:N Relation zu realisieren.

Listing 1. 1:N Relation in Java

```
// Variante 1
// (arrays mit Zugriffsmethoden)
public class Panel {
    private Item [] items;
    ...
    public void setItem
        (Item [] newValue);
    { ...
      this.item = newValue;
    }
    ...
}
public Item [] getItem () { ... }
}

// Variante 2
// (benutzt Container-Bibliothek)
public class Panel {
    public HashSet items;
}

// Variante 3
// (Standard-Vektor mit
// Zugriffsmethode)
public class Panel {
```

```

private Vector items
= new Vector (100);
...
public void addToItem (Item value)
{ ... }
public void removeFromItem
      (Item value)
{ ... }
public void draw ()
{ ...
  Enumeration e = item.elements ();
  while (e.hasMoreElements ()) {
    ((Item)e.nextElement()).draw ();
  }
  ...
}
}

```

Als Nächstes veranschaulichen wir uns die Vorgehensweise, um das Kompositum-Muster in Abbildung 9 zu erhalten. Die Abbildung 10 stellt einen Teil des ASG dar. Direkte AST-Knoten sind rechteckig, hinzugefügte Knoten sind oval. Falls wir eine solche Knoten- und Kantenkonstellation im ASG finden, so fügen wir den Assoziationsknoten hinzu.

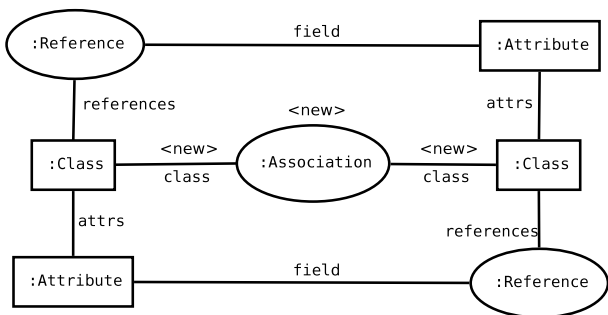


Abbildung 10. Teilmuster Assoziation

Die Regel einer 1:N-Delegation verdeutlicht die Abbildung 11. Dabei wird verlangt, dass die Aufruf- und die aufrufende Methode beide den gleichen Namen besitzen müssen. Ferner besitzt die Aufruf-Methode mehrere Zeiger zu mehreren Instanzen der aufrufende Klasse. Das mehrfache Aufrufen wird durch eine Schleife garantiert, die die Aufruf-Methode umschließt.

Wenn die Knoten- und Kantenkonstellation wie in Abbildung 12 gegeben sind, so fügen wir den Knoten für das Kompositum-Muster hinzu.

2.6 Statische und dynamische Analyse

Heuzeroth, Holl, Högström und Löwe stellten 2003 [HHHL03] einen neuen Ansatz vor, um Verhaltensmuster

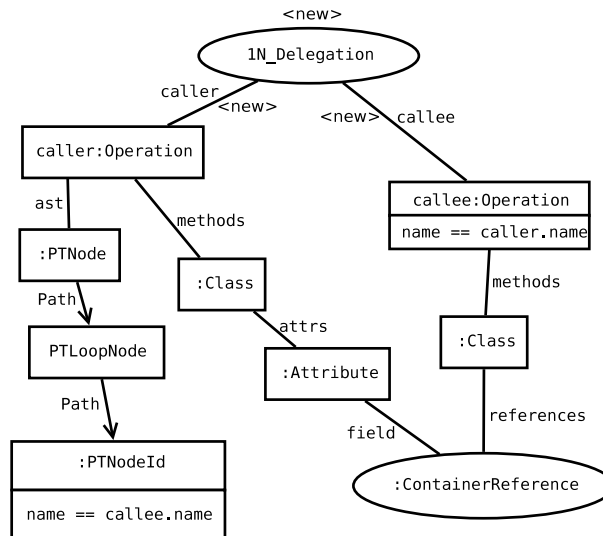


Abbildung 11. Teilmuster 1:N-Delegation

zu erkennen. Dabei verwendeten sie zwei Phasen (in der Abbildung 13 dargestellt). Neben der bisher bekannten statischen Analyse, die eine Menge an Kandidaten liefert, filtern sie die Kandidaten erneut durch Verhaltensbeobachtung bei der Programmausführung in der dynamischen Analyse. Dadurch wird es möglich, auch Verhaltensmuster zu erkennen, die statisch nicht im Programmcode erkennbar sind, sondern nur während einer Programmausführung ihr Verhalten zeigen.

Die statische Analyse läuft, wie bisherige Ansätze auch, mit einem Front-End eines Compilers namens Recorder. Dabei wird ein abstrakter Syntaxbaum aufgebaut, anschließend wird eine semantische Analyse durchgeführt.

Da die statische, aber vor allem die dynamische Analyse sehr stark vom Muster abhängt, schauen wir uns die Analyse im konkreten Beispiel des Beobachter-Musters (Abbildung 14) an.

Das Beobachter-Muster wird eingesetzt, wenn mehrere Objekte ihre Zustände und Verhalten in Abhängigkeit des Zustandes eines bestimmten Objekts ändern sollen.

Z. B. wenn man die Bildlaufleiste verschiebt, so soll sich der Text verschieben und die aktuelle Seitenangabe aktualisiert werden. Der Text und die aktuelle Seitenangabe sind zwei Objekte, die als Beobachter fungieren und das Objekt Bildlaufleiste beobachten. In Abhängigkeit vom Zustand der Bildlaufleiste verändern sich die zwei anderen Objekte. Um die ständige Abfrage des Zustandes zu vermeiden, melden sich die Beobachter beim zu beobachtenden Objekt an. Die Beobachter werden dann verständigt, falls der Zustand sich geändert hat. Daraufhin rufen die Beobachter den Zustand des Objektes ab und verändern ihr Verhalten entsprechend.

Die verfeinerte statische Analyse liefert für das

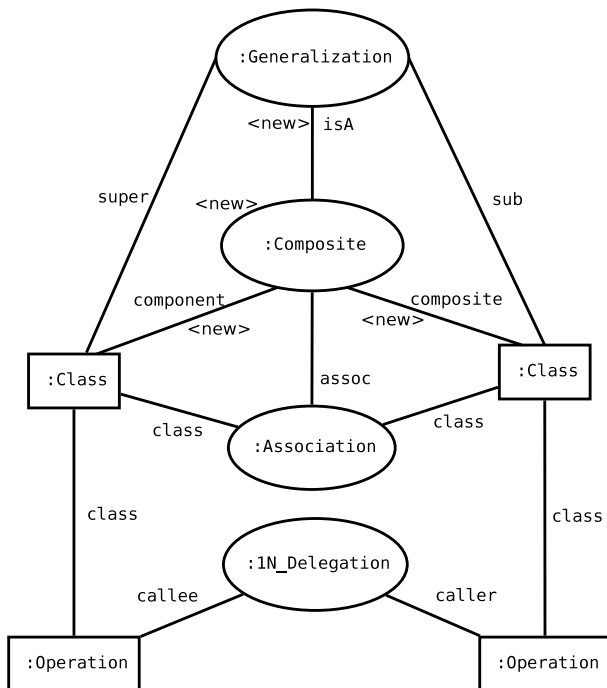


Abbildung 12. Gesamtmuster Kompositum

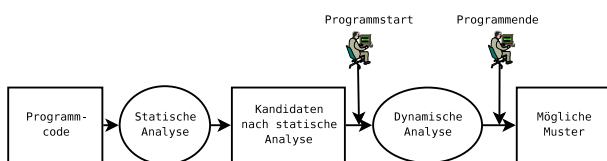


Abbildung 13. Analyseablauf

Beobachter-Muster eine Menge an Kandidaten zurück. Jeder Kandidat enthält zwei Klassen *S* und *L*, wobei die Klasse *S* drei Methoden, ohne eine Einschränkung auf Namenskonvention, *MeldeAn*, *MeldeAb* und *Benachrichtige*, und die Klasse *L* eine Methode *Aktualisiere* enthalten muss. *MeldeAn* und *MeldeAb* sollen jeweils einen Parameter haben. Der Parameter sollte nicht von Typ *S* der Klasse *S* sein, auch nicht Untertyp oder Mutterklasse von *S*. [Weitere Feinheiten der statischen Analyse in [HHHL03] wurden aus Einfachheitsgründen weggelassen.]

Bei der dynamischen Analyse müssen wir auf die $1 : 1$, $1 : n$ und $n : m$ Relationen zwischen den Instanzen achten. In unserem konkreten Fall verlangt das Beobachter-Muster eine $1 : n$ Relation zwischen einer Instanz des Subjekts und ihre Beobachter. Darüber hinaus müssen natürlich alle Instanzen einer Klasse beobachtet werden.

Wir beobachten und erweitern die zum Muster zugehörigen Methoden, um die Kandidaten für das Muster zu filtern. In unserem konkreten Fall erhält *MeldeAn* einen Beobachter als Argument. Der Beobachter wird in einer Liste

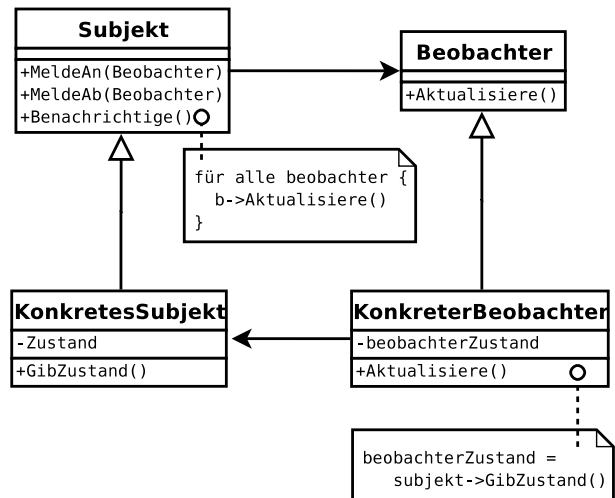


Abbildung 14. Beobachter-Muster

pro Instanz gespeichert. Bei *MeldeAb* wird der Beobachter als Argument wieder von der Liste genommen, welcher zuvor eingefügt worden ist. Jedoch kann dieses Aufrufen auch durch ein Programmierfehler auftreten, daher nehmen wir das nicht als Kriterium. Bei *Benachrichtige* müssen alle oder kein Beobachter über die Methode *Aktualisiere* benachrichtigt werden. Dazu werden alle Beobachter am Anfang der Methode als *nichtAktualisiert* markiert, am Ende müssen alle oder kein Beobachter markiert sein. Eine Nicht-Aktualisierung macht Sinn, wenn sich alle (internen) Zustände vom beobachtenden Objekt nicht geändert haben. *Aktualisiere* markiert den Beobachter als *aktualisiert*, falls *Aktualisiere* zuvor von *Benachrichtige* aufgerufen wurde. Ein Kandidat wird ausgefiltert, falls die Methode *Benachrichtige* *Aktualisiere* aufruft, der Beobachter jedoch nicht in der Liste der Beobachter enthalten ist.

Weitere Struktur- und Verhaltensmuster wie Kompositum, Vermittler, Zuständigkeitskette und Besucher [HHHL03] wurden vorgestellt, auf die ich hier im Einzelnen nicht eingehe.

2.7 Bitvektor-Algorithmus

Einen neuen interessanten Ansatz entwickelten 2006 Kaczor, Guéhéneuc und Hamel [KGH06]. Sie wandelten das Muster und die im Programmcode zu findenden Klassen mit ihren Beziehungen in zwei eindeutige Strings um. Darauf aufbauend kann man einen Bitvektor-Algorithmus anwenden, der typischerweise auf Strings arbeitet, um das Muster zu finden.

Wir sehen in Abbildung 15 das Kompositum-Muster und in Abbildung 16 den Programmcode als Graph dargestellt. Kantenbeschriftung *co* steht für Komposition und *in* für Inheritance. Darüber hinaus wurden Dummy-Kanten *dm* ein-

gefügt, um künstlich einen Eulerkreis im Muster-Graph und Programmcode-Graph zu erzeugen. Ein Eulerkreis existiert genau dann, wenn die Anzahl der eingehende wie ausgehende Kanten für jeden Knoten im Graphen ausgeglichen und der Graph zusammenhängend ist. Um also Dummy-Kanten hinzuzufügen, teilt man alle Knoten in zwei Hälften: die erste Hälfte besitzt zu viele eingehende Kanten, die zweite zu viele ausgehende Kanten. Man zieht nun entsprechend Dummy-Kanten von der einen Hälfte zur anderen Hälfte. Ausgeglichene Knoten müssen nicht weiter betrachtet werden. (Die Richtung der Inheritance-Kanten ist anders herum zu betrachten.)

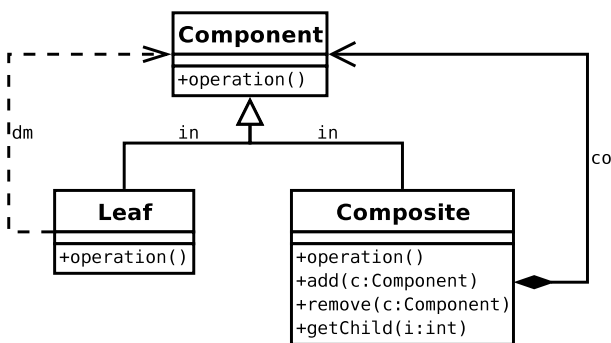


Abbildung 15. Kompositum-Muster

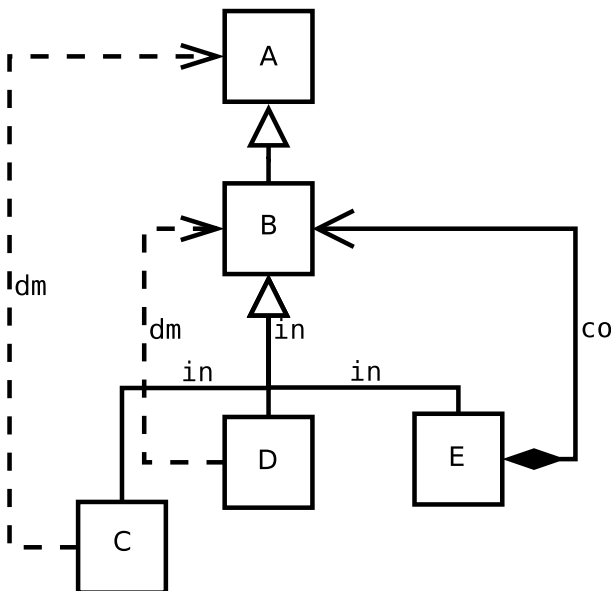


Abbildung 16. Programmcode als Graph

Als nächstes wird ein Linearalgorithmus [EGL94] angewendet, um die Eulerkreise zu finden. Der gefundene Eulerkreis wird in eine eindeutige Stringrepräsentation umgewandelt und wird in Abbildung 17 und 18 dargestellt.

Component in Leaf dm Component in Composite co Component

Abbildung 17. Muster als String

A in B in D dm B in E co B in C dm A

Abbildung 18. Programmcode als String

Das Bitmuster \mathcal{L} für eine gegebene Klasse L im Bezug auf den Programmcode als String $X = x_1 \dots x_n$ mit der Länge n wird wie folgt definiert mit $\mathcal{L} = l_1 \dots l_n$:

$$l_i = \begin{cases} 1 & \text{wenn } x_i = L \\ 0 & \text{sonst} \end{cases}$$

Für die Klasse B in Abbildung 18 ergibt sich folgende Darstellung:

$$B = 001000100010000$$

Auf Bitmuster funktionieren zweistellige Operationen wie \wedge und \vee sowie *shiftright* \rightarrow und *shiftright* \leftarrow . Da wir zuvor einen Eulerkreis gebildet haben, wird das „herausgeschobene“ Bit aus einer Shift-Operation wieder auf der anderen Seite hineingeschoben.

Nun gehen wir auf Mustersuche und bilden als allererstes die *before* und *after* Mengen für alle Kanten. Eine Kante ist in unserem konkreten Fall *in*:

$$before_{in} = \{A, B\}$$

$$after_{in} = \{B, C, D, E\}$$

Um das Teilmuster *Component in Leaf* zu finden muss jeweils ein Element aus den Mengen *before_{in}* und *after_{in}* genommen werden, wobei zwischen den zwei Elementen *in* auftauchen muss, also formal z. B. für $\langle B, C \rangle$:

$$\begin{aligned} & (\rightarrow B) \wedge (\rightarrow in) \wedge C \\ = & 000010001000100 \quad \wedge \\ & 001010001000100 \quad \wedge \\ & 000000000000100 \\ = & 000000000000100 \\ \neq & 000000000000000 \end{aligned}$$

Dummy-Kanten *dm* können im Muster ignoriert werden. Als nächstes kommt *Component in Composite*, da wir für $\langle B, C \rangle$ *B* als *Component* schon haben, wählen wir z. B. *E* aus der *after_{in}* Menge:

$$\begin{aligned}
& (\rightarrow \mathcal{B}) \wedge (\rightarrow in) \wedge \mathcal{E} \\
= & \quad 000010001000100 \quad \wedge \\
& \quad 001010001000100 \quad \wedge \\
& \quad 000000001000000 \\
= & \quad 000000001000000 \\
\neq & \quad 000000000000000
\end{aligned}$$

(Mit `Composite` `co` `Component` wird ähnlich verfahren.) Ein Teiltupel lautet also $\langle B, C, E \rangle$ für $\langle Component, Leaf, Composite \rangle$, da das Resultat der Berechnung nicht $0 \dots 0$ ist.

An der Tabelle 5 kann man das komplette Resultate der drei Kantenüberprüfungen sehen, die aufeinander aufbauen. Wie man leicht erkennen kann, explodiert die Anzahl der Möglichkeiten im 2. Schritt – Der Autor schlägt hier vor, Heuristiken einzubauen und z. B. mit seltensten Kanten im Graph anzufangen, in unserem Beispiel die Kante `co`, die nur einmal auftaucht.

1.	A	B	B	B						
(in)	B	C	D	E						
2.	A	B	B	B	B	B	B	B	B	B
(in)	B	C	C	C	C	C	C	C	C	C
	B	C	D	E	C	D	E	C	D	E
3.	B	B	B							
(co)	C	D	E							
	E	E	E							

Tabelle 5. Schrittweise Berechnung

3 Vergleich

3.1 Analyse und Schwierigkeiten

Um die sieben vorgestellten Ansätze im Punkt 2 vergleichen zu können, ist es zuerst nötig, das gegebene Problem zu analysieren.

Das Problem lässt sich auf die Subgraphensuche reduzieren. Dabei sind Klassen im Programmcode als Knoten vorgegeben. Die Kanten zwischen den Knoten sind Beziehungen zwischen den Klassen. Knoten und Kanten bilden einen großen Graphen G . Gegeben sei außerdem ein kleinerer Mustergraph M . Gesucht sind nun alle Teilgraphen T_i von G , für die es eine isomorphe Abbildung zwischen M und T_i gibt. Dieses Problem ist NP-vollständig⁶. Ebenfalls skaliert das Problem nicht für große Graphen, da die Kombinationsmöglichkeiten explodieren.

⁶<http://www.nada.kth.se/~viggo/wwwcompendium/node36.html>

Das zweite große Problem sind die Entwurfsmuster von [GHJV95], die hier komplett aufgelistet und sortiert sind nach drei Arten:

- Erzeugungsmuster: Abstrakte Fabrik, Erbauer, Fabrikmethode, Prototyp, Singleton.
- Strukturmuster: Adapter, Brücke, Dekorierer, Fassade, Fliegengewicht Kompositum, Proxy.
- Verhaltensmuster: Befehl, Beobachter, Besucher, Interpreter, Iterator, Memento, Schablonenmethode, Strategie, Vermittler, Zustand Zuständigkeitskette.

Strukturmuster können prinzipiell durch die Programmcode-Struktur mit statische Analyse erkannt werden. Generell werden Erzeugungsmuster und Verhaltensmuster „informell“ durch ihr Verhalten zur Laufzeit beschrieben. Ihre Struktur dagegen ist vergleichsweise identisch oder noch schlimmer, ein Muster kann als ein Teilmuster eines zweiten Muster angesehen werden. Das Varianten-Problem in Listing 1 erschwert die Arbeit zusätzlich bei der Suche nach Relationen zwischen Klassen.

3.2 Gemeinsamkeiten und Unterschiede

Als eine Gemeinsamkeit der Ansätze kann die Eingabe der Muster durch eine UML-ähnliche Notation betrachtet werden. In [AFC98] wurde AOL als Sprache und somit ein Pendant zu der grafischen UML-Methode entwickelt. Die untersuchten Sprachen waren C++ oder Java. Die Informationen aus dem Programmcode wurden in der Regel aus einem Front-End eines Compilers gewonnen. Zwei Ansätze arbeiteten direkt auf AST und bildeten einen Abstrakten Syntax-Graph für weitere Untersuchungen.

Erhebliche Unterschiede gibt es bei der Vorgehensweise für die eigentliche Suche. Zuerst gab es die „naive“ Methode mit Prolog – man kann sich leicht vorstellen, dass dieser Ansatz für große Systeme unter schlechter Performance leidet, da die oben erwähnte kombinatorische Möglichkeit voll zu Buche schlägt. Andere Ansätze benutzten Graphen, Metriken, Bitvektoren und andere Methoden, um den Suchraum so früh wie möglich einzuschränken.

Ein richtiges Vergleichen zwischen den sieben Ansätzen war jedoch nicht möglich. Alle Ansätze verteilten sich auf einem Zeitraum von 10 Jahren. Für die Implementierung der Ansätze wurden zwar einige Programme getestet, diese waren jedoch alle unterschiedlich. Ein Vergleichen über die Anzahl der Programmzeilen macht keinen Sinn. Zumindest für die gefundene Kandidaten schwierig war, sie auf Richtigkeit zu überprüfen, weil man das System nicht kennt und es zu aufwändig wäre, alles per Hand zu verifizieren. Nur ein oder zwei Ansätze bauten absichtlich eigene Klassen hinein, die einem Muster entsprechen – es wurde daraufhin geprüft, ob diese fremde Klassen wiedererkannt wurden.

3.3 Resultate

In Tabelle 6 ist ein Überblick über alle Ansätze, die vorgestellt worden sind. Nicht alle Informationen konnten aus den Veröffentlichungen gewonnen werden („-“). Falls mehrere *Software* für den jeweiligen Ansatz getestet worden sind, so habe ich eine Repräsentative gewählt – in der Regel die größte Software im Bezug auf *Anzahl der Klassen*, die in der Software enthalten sind oder *Anzahl der Programmzeilen (KLOC)*. Die untersuchten *Muster* werden für jeden Ansatz aufgelistet. Die erste Zahl im Klammer drückt die Anzahl der gefundenen Muster aus, diese wurden daraufhin per Hand überprüft – die zweite Zahl drückt die tatsächlichen Muster aus. Die *Laufzeit* sollte nicht als strenge Kriterium für einen Vergleich heran gezogen werden, da die Ansätze über einen Zeitraum von 10 Jahre stattfanden und die Schnelligkeit der *Rechner* im Laufe der Zeit sich geändert hat. Die angegebene Laufzeit „29 s“ für Bitvektor-Algorithmus bezieht sich nur auf das Muster Abstrakte Fabrik.

4 Fazit

Aufgrund der in 3.1 vorgestellten Schwierigkeiten müssen bei vielen Ansätze konservative Annahmen getroffen werden. Die Subgraphensuche ist schwer skalierbar und viele Muster sind über ihre Struktur unzureichend definiert. Die Ergebnisse in Tabelle 6 sind mittelmäßig, es gibt noch zu viele False Positives. Die Veröffentlichungen geben an: es können bei bestimmten Kandidaten keine Entscheidung getroffen werden. Es gibt keinen Ansatz, der für sich den Anspruch erhebt, der beste zu sein. Es ist auch fraglich bei dieser Fragestellung, ob man überhaupt einen „guten“ Ansatz finden kann.

Literatur

- [AFC98] G. Antoniol, R. Fiutem, and L. Cristoforetti. *Using Metrics to Identify Design Patterns in Object-Oriented Software*. Proc. 5th International Symposium on Software Metrics, 1998.
- [EGL94] H. A. Eiselt, M. Gendreau, and G. Laport. *Arc routing problems. part I: The Chinese Postman problem. Technical Report CRT-960*. Centre de Recherche sur les Transports, 1994.
- [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley Publishing Company, Reading, MA, 1995.
- [HHHL03] D. Heuzeroth, T. Holl, G. Högström, and W. Löwe. *Automatic Design Pattern Detection*. Proc. 11th IEEE International Workshop on Program Comprehension WPC, 2003.
- [KGH06] O. Kaczor, Y. Guéhéneuc, and S. Hamel. *Efficient Identification of Design Patterns with Bit-vector Algorithm*. CSMR, 2006.
- [Kos04] J. Koskinen. *Software maintenance costs*. Webseite, 2004.
- [KP96] C. Krämer and L. Prechelt. *Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software*. Proc. WCRE, 1996.
- [NSW⁺02] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Wels. *Towards Pattern-Based Design Recovery*. ACM ICSE, 2002.
- [SvG98] J. Seemann and J. W. von Gudenberg. *Pattern-Based Design Recovery of Java Software*. ACM SIGSOFT, 1998.
- [TA99] P. Tonella and Guilio Antoniol. *Object Oriented Design Pattern Inference*. ICSM, 1999.

Ansatz	Software	Anzahl Klassen	KLOC	Muster	Laufzeit	Rechner
Prolog Karlsruhe 1996	ACD ^a (...)	343	204	Adapter (150/69) Brücke (0/0) Kompositum (0/0) Dekorierer (0/0) Proxy (17/0)	36 s	P133 32 MB Win98
Teilgraph Würzburg 1998	Java AWT 1.0 ^b	-	-	Kompositum (-/-) Strategie (-/-) Brücke (-/-)	-	-
Metriken Povo (Trento) 1998	LEDA ^c (...)	187	115	Adapter (8/5) Brücke (7/0) Kompositum (0/0) Dekorierer (0/0) Proxy (0/0)	216 s	-
Mengenbildung Sannio 1999	mec 0.3 ^d	32	21	Adapter (34/27) 1721 Kandidaten der Länge 3	1,8 s	Sun SPARC 20 workstation
Teilmuster Paderborn 2002	Java AWT ^e (JGL)	429	114	Strategie (-/-) Brücke (-/-) Kompositum (-/-) (...)	22 min	Pentium III 933 MHz 1 GB JDK 1.3 Linux 2.4.5
Statische und Dynamische Analyse Karlsruhe 2003	Recorder ^f (SwingSet2)	1481	-	Beobachter (5/-) Mediator (18/6) Zustand Zuständig- keitskette (24/1) Besucher (349/2)	70 s (statisch)	Pentium III 500 MHz 256 MB Windows NT JDK 1.3.1
Bitvektor- Algorithmus Montreal 2006	QuickUML 2001 ^g (...)	373	-	Abstrakte Fabrik (57/13) Kompositum (0/22) (...)	149 s + 29 s	AMD Athlon 64 Bits 2 GHz

^aTelekommunikationssoftware Automatic Call Distribution

^bJava Abstract Window Toolkit 1.0

^cMax-Planck Institut Library of Efficient Data Types and Algorithms

^dTrace-And-Replay Program

^eJava Abstract Window Toolkit

^fFront-End eines Compilers

^gGrafischer Editor für UML-Diagramme

Tabelle 6. Überblick der Verfahren