Weiterentwicklung objektorientierter Softwaresysteme: Risiken und deren Vermeidung

Jens Uwe Pipka

Daedalos Consulting GmbH
Ruhrtal 5, D-58456 Witten, Germany
Tel.: (49) +2302 979 0
jens-uwe.pipka@daedalos.com

Mira Mezini

Universität-GH Siegen
Fachbereich Elektrotechnik und Informatik
Hölderlinstr. 3, D-57068 Siegen, Germany
Tel.: (49) +271 740 2316
mira@informatik.uni-siegen.com

Zusammenfassung. Vererbung gehört zu den wichtigsten Mechanismen objektorientierter Programmierung. Durch Vererbung können vorhandene Implementierungen spezialisiert und an bestimmte Aufgaben angepasst werden. Doch während der Zugriff auf Instanzen über eine klar definierte Schnittstelle erfolgt, existiert für die Vererbung kein äquivalenter Mechanismus. Deshalb können Änderungen innerhalb der Vererbungshierarchie nicht abschätzbare Auswirkungen auf das Verhalten des Gesamtsystems haben. Insbesondere Änderungen in Basiskomponenten können dazu führen, dass das Gesamtsystem syntaktisch korrekt und damit übersetzbar ist, aber semantische Konflikte ein Fehlverhalten des Gesamtsystems zur Folge haben.

In aktuellen Entwicklungsumgebungen wird die Weiterentwicklung objektorientierter Systeme durch Versionsverwaltungen unterstützt. Änderungen unterschiedlicher Entwicklungsstände können verfolgt werden, wobei allerdings ausschließlich einfache Werkzeuge zur Arbeit auf Quelltextebene zur Verfügung stehen. Die Zusammenhänge des Gesamtsystems bleiben dabei unberücksichtigt. Da dieser Ansatz zu restriktiv ist, stellen wir mit „JaMB für Java“ ein Analysewerkzeug zur Erkennung semantischer Konflikte vor. Durch Analyse des Java-Bytecodes werden Konfliktsituationen zur Integrationszeit erkannt. So wird der Entwickler auf mögliche Verhaltensänderungen in einem Softwaresystem hingewiesen, die sonst unerkannt bleiben.

0. Einleitung

Bei der Weiterentwicklung objektorientierter Softwaresysteme mit Hilfe der Vererbung können semantische Konfliktsituationen entstehen, die das Verhalten einer realisierten Anwendung in unvorhersehbarer Art und Weise beeinflussen und Fehler hervorrufen. Im Rahmen dieses Artikels werden im ersten Teil Konfliktsituationen und deren mögliche Auswirkungen vorgestellt, die durch den Austausch bereits spezialisierter Basisklassen entstehen. Im zweiten Teil untersuchen wir, welche Schritte zur Erkennung dieser semantischen Konflikte notwendig sind und welche Hilfsmittel klassische Programmierwerkzeuge dem Programmierer dazu zur Verfügung stellen.

Ausgehend von diesen Überlegungen wird im dritten Teil das Analysewerkzeug JaMB für Java vorgestellt, das innerhalb von Java-Projekten diese Konfliktsituationen

lokalisiert. Es wird dabei sowohl die Architektur als auch die Anwendung des Werkzeugs innerhalb des Software-Entwicklungsprozesses erläutert. Abschließend wird der aktuelle Stand von JaMB für Java im Kontext der Konflikterkennung bei der Weiterentwicklung objektorientierter Systeme betrachtet sowie auf mögliche Erweiterungen ausgeblickt.

1. Konfliktmöglichkeiten bei der Weiterentwicklung objektorientierter Softwaresysteme

Bei der Entwicklung von Softwaresystemen erhält die Weiterverwendung vorhandener Komponenten eine immer größere Bedeutung. Teile der Software müssen nicht neu implementiert werden, sondern können durch den Einsatz und die Spezialisierung vorhandener Implementierungen realisiert werden. Das Verhalten existierender Klassen wird durch Vererbung an die Anforderungen des neuen Systems angepasst.

Wir gehen von folgendem prinzipiellen Aufbau eines objektorientierten Systems aus: Ein Basismodul besteht aus einer oder mehreren Basiskomponenten, die extern oder intern entwickelt werden. Das Erbmodul stellt die anwendungsspezifische Implementierung für das Gesamtsystem bereit, indem mit Hilfe von Vererbung die im Basismodul enthaltenen Klassen spezialisiert werden. Daneben existieren weitere Klassen, die unabhängig vom Basismodul sind und weitere anwendungsspezifische Funktionalitäten realisieren. Für die im Rahmen dieser Arbeit betrachten Konflikttypen ist es nicht notwendig, diese Klassen näher zu untersuchen. Der Aufbau eines solchen Systems ist in Abbildung 1 dargestellt.

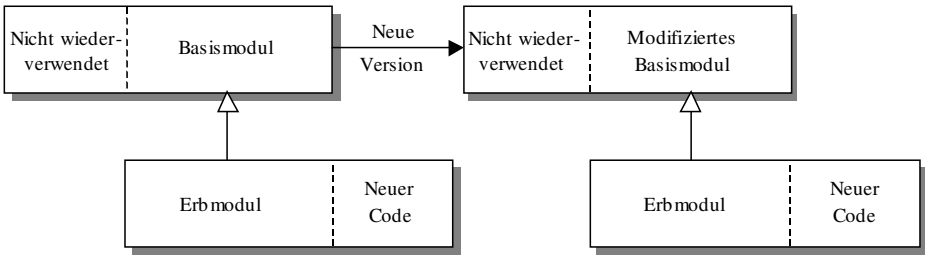

Abbildung 1: Weiterentwicklung objektorientierter Systeme

Bei der Weiterentwicklung, der Wartung und der Migration objektorientierter Softwaresysteme spielt die oben vorgestellte Struktur eine entscheidende Rolle: Sobald eine neue Version des Basismoduls vorliegt, wird die ursprüngliche Implementierung gegen die der neuen Version ausgetauscht. Dazu ist eine Integration des modifizierten Basismoduls in das Gesamtsystem notwendig. Aber auch wenn sich das Gesamtsystem ohne syntaktische Fehler übersetzen lässt und scheinbar erfolgreich ausgeführt werden kann, ist nicht garantiert, dass das derart realisierte Verhalten korrekt ist. Durch Änderungen innerhalb des Basismoduls können bei der Kombination mit dem Erbmodul semantische Fehler entstanden sein, die das Gesamtverhalten beeinflussen und zur Integrationszeit unentdeckt bleiben [KicLam92, Lam93, Mez97, MPDB99, Min96].

Man spricht in diesem Fall auch vom *Fragile Class Problem* [MiSe98]: Eine Oberklasse wird verändert, wohingegen die Unterklasse in ihrer ursprünglichen Form erhalten bleibt. Diese Kombination führt dazu, dass sich das Verhalten ändert. Im Folgenden werden Konfliktsituationen vorgestellt, die durch den Austausch des Basismoduls gegen eine modifizierte Variante entstehen.

1.1 Konflikte durch Änderung innerhalb des Aufrufgraphen

Zunächst werden semantische Konflikte untersucht, die durch Änderungen innerhalb des Aufrufgraphen (Calling Graph) zustande kommen. In einem Aufrufgraphen werden die Kommunikationsstrukturen zwischen Klassen und Aktivierungsreihenfolgen von Methoden untereinander festgehalten. Mit diesen Informationen kann überblickt werden, welche Teile des Systems miteinander kommunizieren und welche anderen Teile der Implementierung von Änderungen innerhalb einer Klasse bzw. Methode betroffen sind.

Änderungen innerhalb des Basismoduls können dazu führen, dass der Aufrufgraph verändert und damit die Abhängigkeiten von Klassen zueinander beeinflusst werden. Dadurch können semantische Konfliktsituationen ausgelöst werden, die wir nun näher betrachten.

1.1.1 Accidental Method Capture

Eine kritische Änderung ist das Hinzufügen einer neuen Operation in einer Klasse des Basismoduls, die innerhalb der modifizierten Klasse von anderen vorher existierenden Methoden aufgerufen wird. Ein Problem tritt auf, falls in einer Klasse des Erbmoduls, die direkt oder indirekt von der modifizierten Klasse des Basismoduls erbt, zufällig eine Methode mit gleicher Signatur bereits existiert. In diesem Fall wird die im Erbmodul definierte Methode an Stellen innerhalb des modifizierten Basismoduls einbezogen, an denen dies nicht vorgesehen ist. Die im Erbmodul definierte Methode wird unvorhergesehen in einem anderen Kontext verwendet, was dazu führen kann, dass das Verhalten des Gesamtsystems fehlerhaft wird. Dieser Effekt wird als das *Accidentally Captured Methods* Problem bezeichnet [SLMDH96].

Das folgende Beispiel macht die Problematik anhand der Realisierung eines einfachen Geldautomaten deutlich. Innerhalb des Basismoduls ist die Klasse **Bankomat** definiert. Diese enthält die Methoden **gibGeldeinheit()** und **gibGeldeinheit(int)**, über die ein bestimmter Geldbetrag abgehoben werden kann. Das Erbmodul spezialisiert die Klasse **Bankomat** durch die Definition der Unterklasse **BankomatX**. Diese sorgt dafür, dass ein Kunde nicht mehr als drei Mal Geld abheben darf und protokolliert zusätzlich die Abhebezeiten. Dazu werden die Methoden **gibGeldeinheit()** und **gibGeldeinheit(int)** überschrieben sowie die Methode **initialisiere()** neu eingeführt, die vom Konstruktor aufgerufen wird. Das Basismodul wird später geändert, um die Anschaltzeit des Automaten zu protokollieren und so etwaige Garantieansprüche eines Kunden prüfen zu können. Dazu wird in der Klasse **Bankomat** ebenfalls die Methode **initialisiere()** eingeführt, die neben der Anschaltzeit gleichzeitig auch den Ausgabebetrag auf Null setzt.

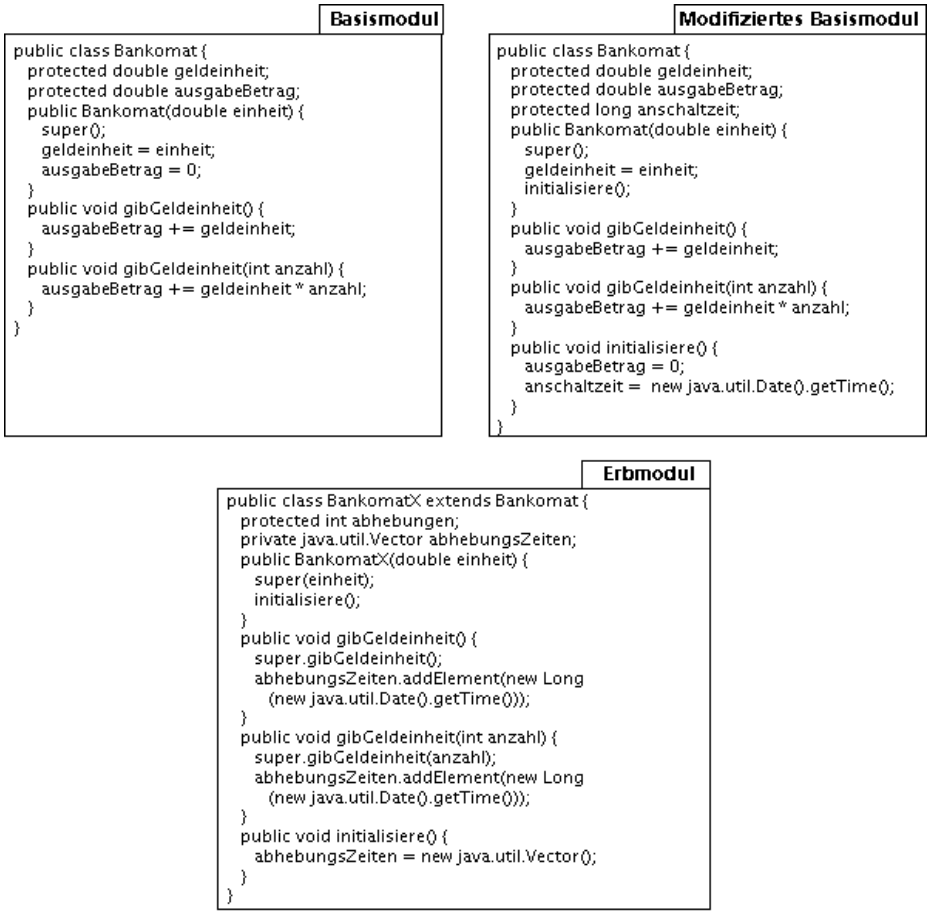

Tabelle 1: Beispiel für Accidental Method Capture

Die Kombination des modifizierten Basismodul mit dem Erbmodul beeinflusst das Verhalten von **BankomatX**. Grund dafür: Bei Einbeziehung des **Bankomat()** Konstruktors durch den **super-**Aufruf innerhalb des **BankomatX()** Konstruktors wird nicht die Methode **initialisiere()** aus der Klasse **Bankomat** aufgerufen, sondern die gleichnamige Methode aus der Klasse **BankomatX**. Dadurch wird weder der Ausgabebetrag auf null gesetzt noch die Anschaltzeit protokolliert. Damit befindet sich das System in einem nicht vorhergesehenen Zustand, was zu einem fehlerhaften Verhalten des Systems führt.

1.1.2 Regular Method Capture

Dieser Konflikttyp beruht darauf, dass eine Methode, die bereits innerhalb des ursprünglichen Basismoduls existiert, im modifizierten Basismodul an weiteren Stellen aufgerufen wird. Dies ist deshalb problematisch, da im Erbmodul Klassen existieren können, die diese Methode überschrieben haben. Durch Hinzufügen von Aufrufen wird die Definition der Methode im Erbmodul an Stellen innerhalb des modifizierten

Basismoduls einbezogen, an der dies nicht vorgesehen war. Diese Art von Konflikt wird als *Regular Method Capture* bezeichnet [SLMDH96]. Im Vergleich zum *Accidental Method Capture* tritt dieser Konflikt nur für Methoden auf, die bereits im ursprünglichen Basismodul vorhanden sind und innerhalb des Erbmoduls überschrieben werden.

Basismodul

```
public class Bankomat {
  protected double geldeinheit;
  protected double ausgabeBetrag;
  public Bankomat(double einheit) {
    super();
    geldeinheit = einheit;
    ausgabeBetrag = 0;
  }
  public void gibGeldeinheit() {
    ausgabeBetrag += geldeinheit;
  }
  public void gibGeldeinheit(int anzahl) {
    ausgabeBetrag += geldeinheit * anzahl;
  }
}
```

Modifiziertes Basismodul

```
public class Bankomat {
  protected double geldeinheit;
  protected double ausgabeBetrag;
  public Bankomat(double einheit) {
    super();
    geldeinheit = einheit;
    ausgabeBetrag = 0;
  }
  public void gibGeldeinheit() {
    ausgabeBetrag += geldeinheit;
  }
  public void gibGeldeinheit(int anzahl) {
    for (int i=0; i<anzahl; i++)
      gibGeldeinheit();
  }
}
```

Erbmodul

```
public class BankomatX extends Bankomat {
  protected int abhebungen;
  private static int maxAbhebungen = 3;
  public BankomatX(double einheit) {
    super(einheit);
  }
  public void gibGeldeinheit() {
    if (abhebungen < maxAbhebungen) {
      super.gibGeldeinheit();
      abhebungen += 1;
    }
  }
  public void gibGeldeinheit(int anzahl) {
    if (abhebungen < maxAbhebungen) {
      super.gibGeldeinheit(anzahl);
      abhebungen += 1;
    }
  }
}
```

Tabelle 2: Beispiel für Regular Method Capture

Um diese Konfliktsituation zu verdeutlichen, greifen wir wieder das Beispiel des Geldautomaten auf. Als Basismodul kommt erneut die Klasse **Bankomat** zum Einsatz. Das Erbmodul realisiert wiederum mit **BankomatX** eine Klasse, die einen Geldautomaten der Bank X darstellt und maximal drei Abhebungen zulässt. Dazu wird diesmal die Methode **gibGeldeinheit(int)** in **BankomatX** unter Einbeziehung der Definition der Elternklasse implementiert. Die ursprüngliche Methode **gibGeldeinheit(int)** aus der Klasse **Bankomat** ist im Basismodul unabhängig von der Methode **gibGeldeinheit()** definiert. Beim Austausch gegen das modifizierte Basismodul ändert sich folgendes: Die Methode **gibGeldeinheit(int)** ruft nun die Methode **gibGeldeinheit()** auf.

Bei der Kombination des modifizierten Basismoduls mit dem vorhandenen Erbmodul ergibt sich dadurch die fatale Situation, dass für die Klasse **BankomatX** das ursprünglich realisierte Verhalten verändert wird. Grund dafür ist, dass bei Aufruf von **gibGeldeinheit(int)** die in **BankomatX** überschriebene Methode **gibGeldeinheit()**

aufgerufen wird. Dadurch wird die Anzahl der Abhebungen nicht mehr wie vorgesehen um eins, sondern um den an **gibGeldeinheit(int)** übergebenen Parameter erhöht. Dies führt dazu, dass nach der dritten Geldeinheit die Auszahlung verweigert wird.

1.1.3 Unanticipated Recursion

Unter bestimmten Voraussetzungen kann *Regular Method Capture* dazu führen, dass eine unendliche Rekursion entsteht. Dazu muss im Basismodul innerhalb einer Methodendefinition der Aufruf einer Methode hinzukommen, die innerhalb des Erbmoduls überschrieben wurde und selbst wieder direkt oder indirekt auf die modifizierte Methode zugreift. Dadurch entsteht ein Zyklus, der beim Aufruf einer enthaltenen Methode nicht mehr terminiert.

Basismodul

```
public class Bankomat {
  protected double geldeinheit;
  protected double ausgabeBetrag;
  public Bankomat(double einheit) {
    super();
    geldeinheit = einheit;
    ausgabeBetrag = 0;
  }
  public void gibGeldeinheit() {
    ausgabeBetrag += geldeinheit;
  }
  public void gibGeldeinheit(int anzahl) {
    ausgabeBetrag += geldeinheit * anzahl;
  }
}
```

Modifiziertes Basismodul

```
public class Bankomat {
  protected double geldeinheit;
  protected double ausgabeBetrag;
  public Bankomat(double einheit) {
    super();
    geldeinheit = einheit;
    ausgabeBetrag = 0;
  }
  public void gibGeldeinheit() {
    gibGeldeinheit(1);
  }
  public void gibGeldeinheit(int anzahl) {
    ausgabeBetrag += geldeinheit * anzahl;
  }
}
```

Erbmodul

```
public class BankomatX extends Bankomat {
  protected int abhebungen;
  private static int maxAbhebungen = 3;
  public BankomatX(double einheit) {
    super(einheit);
  }
  public void gibGeldeinheit() {
    if (abhebungen < maxAbhebungen) {
      super.gibGeldeinheit();
      abhebungen += 1;
    }
  }
  public void gibGeldeinheit(int anzahl) {
    if (abhebungen < maxAbhebungen) {
      for (int i=0; i< anzahl; i++)
        super.gibGeldeinheit();
      abhebungen += 1;
    }
  }
}
```

Tabelle 3: Beispiel für Unanticipated Recursion

Dieser Konflikt wird in der Literatur als *Unanticipated Recursion* bezeichnet [SLMDH96]. Das gesamte System kann durch diesen Effekt zur Laufzeit zum Absturz gebracht werden, ohne dass dies zur Integrationszeit entdeckt wird. Ursache ist, dass sich Methoden aus modifiziertem Basis- und Erbmodul gegenseitig aufrufen, die ursprünglich unabhängig voneinander implementiert waren.

Zur Verdeutlichung wird wiederum auf das bekannte Basismodul zurückgegriffen. Das Erbmodul realisiert mit der Klasse **BankomatX** als Unterklasse von **Bankomat**

wiederum einen Geldautomaten, der maximal drei Abhebungen erlaubt. Dazu werden die Methoden **gibGeldeinheit()** und **gibGeldeinheit(int)** überschrieben, wobei beide die Methode **gibGeldeinheit()** der Elternklasse einbeziehen. Das modifizierte Basismodul enthält eine neue Definition der Klasse **Bankomat**, in der die Methode **gibGeldeinheit()** unter Einbeziehung der Methode **gibGeldeinheit(int)** realisiert ist. Dies führt bei der Kombination von modifiziertem Basismodul und Erbmodul für Instanzen der Klasse **BankomatX** nun zu einem Konflikt im Sinne der *Unanticipated Recursion*: Sobald die Methode **gibGeldeinheit()** oder **gibGeldeinheit(int)** ausgeführt wird, ruft diese die Methode **gibGeldeinheit()** aus der Elternklasse **Bankomat** aus. Dies wiederum hat zur Folge, dass die Methode **gibGeldeinheit()** aufgerufen wird, die in diesem Kontext an die Definition der Klasse **BankomatX** gebunden ist. Aufgrund dieses gegenseitigen Aufrufs ist die Anwendung blockiert.

1.1.4 Inconsistent Methods (Specialization Negation)

Eine andere problematische Änderung innerhalb des Basismoduls ist das Entfernen von Methodenaufrufen. Dies kann dazu führen, dass innerhalb des Erbmoduls vorgenommene Verhaltensanpassungen nach der Integration des modifizierten Basismoduls unberücksichtigt bleiben und somit das ursprünglich vorgesehene Verhalten nicht mehr gewährleistet ist. In der Literatur wird in diesem Zusammenhang von *Inconsistent Methods* gesprochen [SLMDH96]. Unserer Ansicht nach ist dieser Begriff zu allgemein gehalten. Deshalb sprechen wir von *Specialization Negation*. Dies bringt zum Ausdruck, dass durch die Änderung innerhalb des Basismoduls die vorgenommene Spezialisierung des Erbmoduls unberücksichtigt bleibt.

Basismodul

```
public class Bankomat {
  protected double geldeinheit;
  protected double ausgabeBetrag;
  public Bankomat(double einheit) {
   super();
    geldeinheit = einheit;
    ausgabeBetrag = 0;
  }
  public void gibGeldeinheit() {
    ausgabeBetrag += geldeinheit;
  }
  public void gibGeldeinheit(int anzahl) {
    for (int i=0; i<anzahl; i++)
      gibGeldeinheit();
  }
}
```

Modifiziertes Basismodul

```
public class Bankomat {
  protected double geldeinheit;
  protected double ausgabeBetrag;
  public Bankomat(double einheit) {
    super();
    geldeinheit = einheit;
    ausgabeBetrag = 0;
  }
  public void gibGeldeinheit() {
    ausgabeBetrag += geldeinheit;
  }
  public void gibGeldeinheit(int anzahl) {
    ausgabeBetrag += geldeinheit * anzahl;
  }
}
```

Erbmodul

```
public class BankomatX extends Bankomat {
  private static int maxEinheiten = 20;
  public BankomatX(double einheit) {
    super(einheit);
  }
  public void gibGeldeinheit() {
    if (ausgabeBetrag / geldeinheit < maxEinheiten) {
      super.gibGeldeinheit();
    }
  }
}
```

Tabelle 4: Beispiel für Specialization Negation

Diesmal wird innerhalb des Beispiels eine andere Implementierung des ursprünglichen Basismoduls verwendet. Dabei steuern die Methoden **gibGeldeinheit()** und **gibGeldeinheit(int)** wiederum die Geldausgabe des Geldautomaten. Die Methode **gibGeldeinheit(int)** ruft **gibGeldeinheit()** auf. Innerhalb des Erbmoduls ist ein Geldautomat realisiert, der einen maximalen Auszahlungsbetrag definiert. Dazu stellt das Erbmodul durch die geeignete Implementierung der Methode **gibGeldeinheit()** aus der Klasse **BankomatX** sicher, dass ein Kunde insgesamt maximal 20 Geldeinheiten abheben darf. Im modifizierten Basismodul wird die Methode **gibGeldeinheit(int)** derart verändert, dass der Ausgabebetrag direkt berechnet wird: Die Methode **gibGeldeinheit()** wird nicht mehr aufgerufen. Bei der Kombination mit dem Erbmodul ergibt sich die Situation, dass auch dann eine Auszahlung zustande kommt, wenn das Limit von maximal 20 Geldeinheiten bereits überschritten ist. Dies ist deshalb möglich, weil die Überwachung des Limits in der überschriebenen Methode **gibGeldeinheit()** realisiert ist, diese aber nicht mehr von der Methode **gibGeldeinheit(int)** aufgerufen wird, da der entsprechende Aufruf innerhalb des modifizierten Basismoduls entfernt wurde.

1.2 Konflikte durch Änderung von Methodensignaturen

Diese Art von Konflikt tritt nur in statisch typisierten Programmiersprachen wie Java auf. Ausgehend von Änderungen innerhalb der Methodensignaturen des Basismoduls bleiben Spezialisierungen bzw. Generalisierungen des Erbmoduls unberücksichtigt.

1.2.1 Reverse Method Capture

Dieser Konflikttyp beruht darauf, dass innerhalb einer Klasse des Basismoduls eine Methodensignatur generalisiert oder spezialisiert wird, d.h. mindestens einer der übergebenen Parameter der Methodensignatur generalisiert oder spezialisiert wird. Ist die ursprüngliche Methodendefinition innerhalb des Erbmoduls überschrieben worden, so führt die Kombination mit dem modifizierten Basismodul dazu, dass diese Definition unberücksichtig bleibt und somit das ursprünglich realisierte Verhalten nicht mehr gewährleistet ist. Es wird zwischen zwei unterschiedlichen Arten des Konflikttyps *Reverse Method Capture* unterschieden, nämlich der *Generalisierung* und der *Spezialisierung*.

Bei der *Generalisierung* wird eine Methodensignatur beim Übergang vom ursprünglichen auf das modifizierte Basismodul derart manipuliert, dass ein oder mehrere Parameter generalisiert werden. Wird das System neu übersetzt, so deckt die generalisierte Methode einen größeren Wertebereich ab als in der ursprünglichen Implementierung. Kritischer ist allerdings, dass die Implementierung der ursprünglichen Methode innerhalb des Erbmoduls möglicherweise nicht mehr berücksichtigt wird. Dies ist beispielsweise dann der Fall, wenn ein Objekt einer spezialisierten Klasse des Erbmoduls im Kontext der Elternklasse verwendet wird. Der gleiche Effekt ist ebenso bei der *Spezialisierung* einer Methodensignatur innerhalb des modifizierten Basismoduls möglich. Durch die Spezialisierung eines oder mehrerer Parameter bleibt eine innerhalb des Erbmoduls vorgenommene Spezialisierung der ursprünglichen Methode unberücksichtigt [MPDB99].

Basismodul

```
public class S {}
public class T extends S {}
public class Base {
  public void test(S s) {
    System.out.println("base.test(S)");
  }
  public void test(T t) {
    System.out.println("base.test(T)");
  }
}
```

Modifiziertes Basismodul

```
public class S {}
public class T extends S {}
public class Base {
  public void test(S s) {
    System.out.println("base.test(S)");
  }
}
```

Erbmodul

```
public class Sub extends Base {
  public void test(T t) {
    System.out.println("sub.test(T)");
  }
}
public class Tester {
  public static void main(String[] args) {
    Base tester = new Sub();
    tester.test(new T());
  }
}
```

Tabelle 5: Beispiel für Reverse Method Capture (Generalisierung)

Die durch die Generalisierung einer Methodensignatur entstandene Konfliktsituation wird im folgenden Beispiel deutlich. Im ursprünglichen Basismodul wird die Klasse **Base** implementiert, welche die Methoden **test(S)** und **test(T)** enthält. **S** und **T** sind dabei zwei Klassen mit **T** Unterklasse von **S**. Das Erbmodul spezialisiert die Klasse **Base** durch **Sub**, in der die Methode **test(T)** überschrieben ist. Die ausführbare Klasse **Tester** erzeugt ein Objekt von **Sub**, verwendet diese im Kontext von **Base** und schickt an dieses Objekt die Nachricht **test(T)**. Im Beispiel führt dies zur Ausgabe des Strings **„sub.test(T)“**, d.h. es wird die Methode **test(T)** aus der Klasse **Sub** verwendet. Im modifizierten Basismodul wird nun die Methodendefinition **test(T)** entfernt. Ansonsten bleibt die Implementierung gleich. Kombiniert man diese Version des Basismoduls mit dem Erbmodul, so wird nun **„base.test(S)“** statt **„sub.test(T)“** ausgegeben. Der Grund: Der Aufruf **test(T)** im Kontext von **Base** ist an die Methodendefinition **test(S)** gebunden. Wird nun ein **Sub**-Objekt in diesem Kontext verwendet, so bleibt aus diesem Grund die dort vorgenommene Spezialisierung von **test(T)** unberücksichtigt. Dies führt dazu, dass sich beim Austausch des Basismoduls das Verhalten ändert und nicht mehr den ursprünglich realisierten Anforderungen entspricht.

2. Realisierung eines Werkzeugs zur Konflikterkennung als Plug-In für aktuelle Entwicklungsumgebungen am Beispiel von JaMB für Java

Im Folgenden soll untersucht werden, ob heutige Entwicklungsumgebungen ausreichende Mechanismen zur Erkennung und Vermeidung semantischer Konflikte bei der Weiterentwicklung objektorientierter Systeme zur Verfügung stellen.

Ein zentraler Mechanismus im Hinblick auf die Weiterentwicklung von Softwareprojekten innerhalb einer Entwicklungsumgebung ist die Versionsverwaltung. Diese

ermöglicht es, unterschiedliche Entwicklungsstände zu verwalten und stellt darüber hinaus Werkzeuge zur Nachverfolgung des Entwicklungsprozesses an. Wird eine neue Version des Basismoduls in das Projekt integriert, so können mit Hilfe der in der Versionsverwaltung enthaltenen Werkzeuge die vorgenommenen Änderungen nachverfolgt werden.

Die Möglichkeiten dieser Werkzeuge reichen nicht aus, um den Entwickler bei der Weiterentwicklung objektorientierter Projekte ausreichend zu unterstützen. Speziell die Erkennung und Vermeidung der im ersten Teil vorgestellten semantischen Konflikte verlangen besondere Techniken und Werkzeuge. Anhand von JaMB für Java wird eine mögliche Realisierung eines geeigneten Analysewerkzeugs zur Erkennung dieser Konfliktsituationen vorgestellt.

2.1 Hilfsmittel klassischer Entwicklungsumgebungen beim Übergang auf eine neue Version einer Basiskomponente

Durch die Verwendung von Versionsverwaltungen wie Envy oder PVCS innerhalb des Entwicklungsprozesses wird die Entwicklung von Softwareprojekten unterstützt. Es können beliebige Entwicklungsstände verwaltet werden, wobei die unterschiedlichen Versionen der Projektdaten in einer gemeinsamen Datenbank abgelegt werden. Versionsverwaltungen bieten dabei die Möglichkeit, jederzeit wieder auf eine ältere Version zurück zu greifen sowie Änderungen in der Implementierung zu verfolgen. Für die Weiterentwicklung objektorientierter Systeme bedeutet dies, dass beim Austausch des ursprünglichen Basismoduls gegen eine neue Version innerhalb der Versionsverwaltung die alten Basismoduldateien durch die neuen ausgetauscht und innerhalb eines Projektes verwendet werden. Innerhalb der Datenbank sind allerdings weiterhin beide Versionen vorhanden, so dass der Austausch auch wieder rückgängig gemacht werden kann.

Versionsverwaltungen bieten über das reine Speichern und Verwalten der Projektdateien hinaus zusätzliche Funktionen an, um unterschiedliche Versionen miteinander vergleichen zu können. Allerdings wird dazu zumeist ausschließlich ein rudimentärer Vergleich zweier unterschiedlicher Implementierungen auf Quelltextebene durchgeführt. Grund dafür ist, dass Versionsverwaltungen ausschließlich Werkzeuge für den Vergleich reiner Textdaten unabhängig von der Implementierungssprache anbieten, die sich entsprechend nur für eine Quelltextanalyse eignen. Die einzelnen Projektdaten werden dabei wie einfache Textdaten behandelt und verarbeitet. Die unterschiedlichen Versionen werden ausschließlich auf Grundlage von textuellen Unterschieden miteinander verglichen. Auf dieser Ebene können nur syntaktische Analysen durchgeführt werden.

Zudem beschränkt die Sicht auf die Klassenebene den Blick für die Zusammenhänge des Gesamtsystems: Jede Klasse wird für sich betrachtet, ohne dass die vorhandenen Querbeziehungen mit einbezogen werden. Ob etwaige Vererbungs- oder Delegationskonflikte entstehen, muss der Entwickler selbst beurteilen. Abhängigkeiten wie Vererbungshierarchien und semantische Änderungen bleiben unberücksichtigt, obwohl die zugehörigen Informationen aus den innerhalb der Versionsverwaltung gespeicherten Daten extrahiert werden könnten.

Liegt eine modifizierte Komponente nicht als Quelltext vor, so ist bei dieser Vorgehensweise eine weitergehende Verfolgung der Änderungen nicht möglich. Diese

Einschränkung gilt insbesondere für Fremdkomponenten, die von einem externen Anbieter bezogen werden und zumeist ohne Quelltext vorliegen. Gerade hier ist aber eine zusätzliche Prüfung notwendig, um überhaupt Änderungen auf das Verhalten des Gesamtsystems erkennen zu können.

Dies zeigt, dass der Vergleich von Quelltextdaten auf Klassenebene mit Hilfe rein textbasierter Werkzeuge nicht ausreicht, um Entwicklern die benötigten Informationen zur Integration neuer Modulversionen zu verschaffen. Viel mehr ist es notwendig, dass zusätzlich eine Analyse im Hinblick auf Zusammenhänge des Gesamtsystems integriert wird. Nur so ist man in der Lage, auf mögliche Verhaltensänderungen durch semantische Konflikte zumindest hinzuweisen.

2.1.1 Integration einer neuen Version des Basismoduls am Beispiel von VisualAge für Java

Als Beispiel für die Möglichkeiten einer Entwicklungsumgebung wird im Folgenden VisualAge für Java verwendet. Die Erfahrungen sind auch auf andere Entwicklungsumgebungen respektive Versionsverwaltungen wie MS SourceSafe oder PVCS übertragbar. Anhand des bereits im ersten Teil vorgestellten Konflikts *Regular Method Capture* wird untersucht, welche Analysemöglichkeiten in VisualAge für Java bei der Integration einer neuen Version des Basismoduls zur Verfügung stehen. VisualAge für Java bietet dazu nach dem Einspielen des modifizierten Basismoduls innerhalb der Projekt- und Versionsverwaltung den Punkt „Compare with...“ an. Damit können zwei beliebige Versionen auf Projekt-, Paket-, Klassen- oder Methodenebene miteinander verglichen werden. Als Ergebnis des Vergleichs der beiden unterschiedlichen Versionen des verwendeten Basismoduls wird folgendes Ergebnis ausgegeben:

```
Differences:

        Bankomat
          gibGeldeinheit(int): Source changed
```

Die genauen Änderungen in der Methode **gibGeldeinheit(int)** werden dabei über einen textuellen Vergleich des Quelltextes ermittelt. Der Entwickler wird darauf hingewiesen, dass in der ursprünglichen Implementierung die Zeile

```
ausgabeBetrag += geldeinheit * anzahl;
```

durch

```
for (int i=0; i< anzahl; i++)
  gibGeldeinheit();
```

ersetzt wurde. Darüber hinaus werden keine weiteren Informationen oder Werkzeuge für eine umfassendere Analyse der unterschiedlichen Versionen des Basismoduls zur Verfügung gestellt.

An diesem Beispiel wird deutlich, dass der Vergleich ohne Berücksichtigung der Klassenhierarchie sowie des Aufrufgraphen nicht ausreicht. Die im Basismodul vorgenommenen Änderungen erscheinen für sich genommen unkritisch. Bei einem Integrationstest wird aufgrund dieser vorliegenden Informationen kein Fehler gefunden. Die von der Entwicklungsumgebung bereitgestellten Informationen reichen nicht aus, um das Problem zu erkennen. Die beim Vergleich der unterschiedlichen Versionen des Basismoduls ermittelten Unterschiede lassen die Eigenheiten des Gesamtsystems

unberücksichtigt. Der Konflikt kann deshalb nur von demjenigen erkannt werden, der mit der Struktur des Gesamtsystems vertraut ist. Im aufgeführten Beispiel ist der Konflikt nur dann erkennbar, falls zusätzlich zur Klasse **Bankomat** die Implementierung der Unterklasse **BankomatX** zusammen mit den enthaltenen Aufrufstrukturen betrachtet wird. Je komplexer ein System ist, um so unwahrscheinlicher ist es, dass solche Konflikte überhaupt während der Integration erkannt werden.

2.2 Notwendige Schritte zur Erkennung semantischer Konflikte

Um die Weiterentwicklung objektorientierter Softwareprojekte zu vereinfachen und die vorgestellten Konfliktsituationen zu vermeiden, müssen die Änderungen in der neuen Version einer Basiskomponente unter Berücksichtigung der bereits vorhandenen Implementierungen auf semantische Konflikte überprüft werden. Das dazu geeignete Vorgehen zur Erkennung von Konfliktsituationen kann in folgende Schritte gegliedert werden: Im ersten Schritt werden zunächst die für die Konfliktanalyse benötigten Informationen aus der ursprünglichen sowie der modifizierten Version des Basismoduls extrahiert. Zu diesen Informationen gehören Informationen zu den enthaltenen Klassen- und Methodendefinitionen und zum Aufrufgraph. Diese Daten werden aus den Java-Bytecode-Dateien ermittelt und in eine Metastruktur überführt, die eine Erweiterung der Java Reflection API darstellt.

Im Anschluss daran wird das Konfliktpotential ermittelt, indem die Implementierung des ursprünglichen Basismoduls mit der des modifizierten Basismoduls verglichen wird. Das Konfliktpotential enthält als Ergebnis alle Konfliktmöglichkeiten, die bei der Kombination des modifizierten Basismoduls mit einem beliebigen Erbmodul möglich sind. Zur Ermittlung des Konfliktpotentials werden unterschiedliche Analyseschritte auf Grundlage der Änderungen innerhalb der Klassen- und Schnittstellendefinitionen sowie des Aufrufgraphen des Basismoduls durchgeführt. Da je nach Konflikttyp zur Erkennung des Konfliktpotentials unterschiedliche Analyseschritte notwendig sind, wird für jeden Konflikttyp ein eigener Konfliktpotentialdetektor definiert, der die spezifischen Analyseschritte zur Erkennung des jeweiligen Konflikttyps enthält. Wir werden die Funktionsweise eines Konfliktpotentialdetektors im folgenden Abschnitt exemplarisch für *Regular Method Capture* erläutern.

Im danach folgenden Schritt werden die zur Konfliktanalyse benötigten Daten des Erbmoduls analog zum Basismodul aus den Java-Bytecode-Dateien gewonnen. Zusammen mit dem ermittelten Konfliktpotential werden diese Daten im Hinblick auf konkrete Konfliktsituationen untersucht. Dazu werden die Klassen- und Schnittstellendefinitionen sowie der Aufrufgraph des Erbmoduls im Hinblick auf die im Konfliktpotential enthaltenen Konfliktsituationen geprüft. Das Ergebnis dieses Schritts liefert die Konflikte, die bei der Kombination aus modifiziertem Basismodul und Erbmodul auftreten. Analog zu Konfliktpotentialen steht zur Erkennung konkreter Konflikte pro Konflikttyp ein eigener Konfliktdetektor zur Verfügung. Die Implementierung eines Konfliktdetektors beruht dabei auf der Konflikttyp-spezifischen Analyse des Konfliktpotentials in Verbindung mit dem Erbmodul.

Bei der Analyse von Java-Projekten werden die beschriebenen Konflikte unabhängig davon erkannt und gemeldet, ob sie tatsächlich das Verhalten des Gesamtsystems beeinflussen. Grund für das Auftreten eines Fehlalarms ist, dass eine tatsächliche Änderung des Verhaltens nicht zur Integrations-, sondern erst zur Laufzeit entdeckt

werden kann. Zur Integrationszeit ist teilweise keine eindeutige Zuordnung für das Ziel eines Methodenaufrufs möglich. Die Ursache liegt darin, dass Instanzen einer Klasse innerhalb des Kontext einer implementierten Schnittstelle bzw. einer Elternkasse verwendet werden können. Um zum einen sowohl Fehlalarme zu vermeiden als auch zum anderen alle Konfliktsituationen zuverlässig zu erkennen, sind Erweiterungen innerhalb des Analyseprozesses im Hinblick auf die exakte Auflösung der Ziele von Methodenaufrufen denkbar, wie sie z.B. in der Literatur von Palsberg und Schwartzbach diskutiert werden [PaSchw95]. Eine andere Möglichkeit wäre, zusätzliche Analyseschritte zur Laufzeit durchzuführen, um das Ziel kritischer Methodenaufrufe bei Ausführung eindeutig bestimmen zu können.

Als Beispiel für den Aufbau eines Analyseprozesses werden wir im nächsten Abschnitt auf die Erkennung des Konflikttyps *Regular Method Capture* eingehen. Danach wird anhand von JaMB für Java exemplarisch eine Umsetzung der Konfliktanalyse für Java Projekte vorgestellt. JaMB basiert auf den vorgestellten Analyseschritten und ermöglicht eine automatische Konfliktanalyse und -erkennung in Java Projekten. Basierend auf diesen Konzepten ist eine Umsetzung für andere objektorientierte Programmiersprachen wie Smalltalk oder Eiffel leicht möglich.

2.3 Ermittlung potentieller und konkreter Konflikte am Beispiel von Regular Method Capture

Bei einer konkreten Implementierung müssen als erstes Informationen z.B. über Klassen- und Schnittstellendefinitionen sowie Aufrufgraph aus den zu analysierenden Daten extrahiert werden. Die Vorgehensweise ist davon abhängig, in welcher Programmiersprache die zu analysierenden Daten vorliegen. Deshalb wird an dieser Stelle zunächst davon ausgegangen, dass die benötigten Informationen zur Verfügung stehen. Bei der Implementierung von JaMB beschäftigen wir uns in Abschnitt 3 am Beispiel von Java genauer mit dieser Problematik.

2.3.1 Ermittlung des Konfliktpotentials

Um das Konfliktpotential für *Regular Method Capture* zu erkennen, müssen alle Methodenaufrufe gefunden werden, die in der neuen Version des Basismoduls hinzugekommen sind. Dazu werden die Methoden der Klassen verglichen, die sich zwischen ursprünglichem und modifiziertem Basismodul geändert haben. Es wird untersucht, ob neue Aufrufe bereits vorhandener Methodendefinitionen hinzugekommen sind. Ist dies der Fall, so sind die Voraussetzungen für einen potentiellen Konflikt im Sinne von *Regular Method Capture* erfüllt.

In unserem Beispiel trifft dies für die Methode **gibGeldeinheit()** aus der Klasse Bankomat zu. Im modifizierten Basismodul erfolgt ein zusätzlicher Aufruf von **gibGeldeinheit()** in der Methode **gibGeldeinheit(int)**.

```
Konfliktpotential für Regular Method Capture

    ⇒ Zusätzliche Aufrufer von
       Bankomat.gibGeldeinheit():
            Bankomat.gibGeldeinheit(int)
```

2.3.2 Ermittlung der konkreten Konflikte

Die Ermittlung der konkreten Konflikte im Sinne von *Regular Method Capture* beruht auf der Analyse des Konfliktpotentials in Kombination mit einem vorhandenen Erbmodul. Ausgehend vom ermittelten Konfliktpotential wird für jede betroffene Klasse untersucht, ob sie eine direkte oder indirekte Unterklasse besitzt, die eine der im Konfliktpotential enthaltenen Methodendefinitionen überschreibt. In diesem Fall sind die im modifizierten Basismodul neu hinzugekommenen und von der Unterklasse geerbten Methodenaufrufe an die von der Unterklasse bereitgestellte Definition gebunden. Sobald die Implementierung basierend auf dem ursprünglichen Aufrufgraphen vorgenommen wurde oder implizit zusätzliche Methodenaufrufe als ausgeschlossen betrachtet wurden, verändert dieser Effekt das Verhalten des Systems. In unserem Beispiel ist die Klasse **BankomatX** betroffen, die von der Klasse **Bankomat** erbt. Analysiert man, welche Methoden dort überschrieben werden, so wird die Methode **gibGeldeinheit()** gefunden. Da diese Methode zum ermittelten Konfliktpotential gehört, liegt ein konkreter Konflikt im Sinne von *Regular Method Capture* vor.

```
Konkrete Konflikte für Regular Method Capture

    ⇒ BankomatX.gibGeldeinheit() überschreibt
       Bankomat.gibGeldeinheit()
       Zusätzliche Aufrufer:
          Bankomat.gibGeldeinheit(int)
```

Wir haben uns dafür entschieden, dass ein konkreter Konflikt auch dann gemeldet wird, falls die kritische Methodendefinition innerhalb des Systems nicht an eine Instanz der Klasse **BankomatX** gesendet wird. In diesem Fall hat der gefundene Konflikt keine Auswirkungen auf das Verhalten des Systems, sondern es handelt sich um einen Fehlalarm. Hier greift allerdings das schon beschriebene Problem der Typinterferenz: Um die Zahl möglicher Fehlalarme zu verringern ist es möglich, eine zusätzliche Prüfung im Hinblick darauf durchzuführen, ob die betroffene Methode überhaupt innerhalb des Systems an eine Instanz der Klasse **BankomatX** gesendet wird. Es werden dabei auch die Fehlersituationen ignoriert, die durch das Benutzen einer Instanz einer betroffenen Klasse innerhalb des Kontexts einer Schnittstelle bzw. Elternkasse entstehen.

3. Konflikterkennung in Java-Projekten: Das Werkzeug JaMB für Java als Plug-In für Integrierte Entwicklungsumgebungen

Die Implementierung von JaMB für Java orientiert sich an den im vorigen Abschnitt vorgestellten Schritten zur Konflikterkennung. Um auf neue Anforderungen eingehen und neue Konfliktsituationen erkennen zu können, wurde JaMB als erweiterbares Framework implementiert. So können zur Erkennung weiterer Konfliktsituationen neue Detektoren innerhalb des Frameworks implementiert werden, die automatisch in den Analyseprozess integriert werden.

3.1 Die Architektur von JaMB für Java

Als Eingabe benötigt JaMB folgende Daten:

- Ursprüngliches Basismodul (Java Bytecode Dateien bzw. JAR Archiv)
- Modifiziertes Basismodul (Java Bytecode Dateien bzw. JAR Archiv)
- Erbmodul (optional; Java Bytecode Dateien bzw. JAR Archiv)
- Aktivierte Konfliktdetektoren (optional)
- Gewünschte Ausgabe: Potentielle und/oder konkrete Konflikte (optional, Boolesche Werte)

Als Ausgabe stellt JaMB folgende Daten bereit:

- Potentielle Konflikte (JaMBStream, variabel)
- Konkrete Konflikte (optional, JaMBStream, variabel)

Das Framework ist in verschiedene Funktionseinheiten unterteilt, die jeweils einen bestimmten Aufgabenbereich abdecken (siehe Abbildung 2).

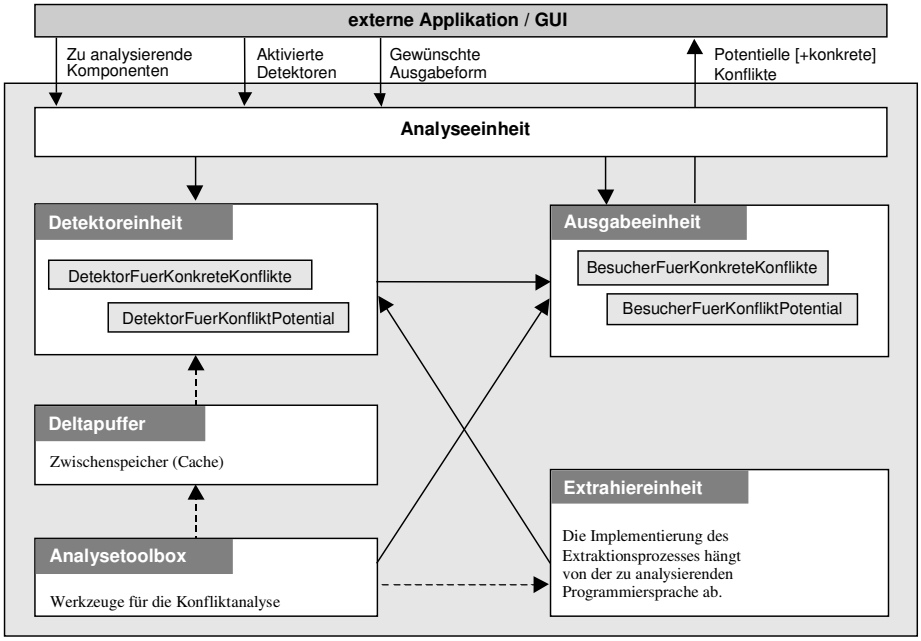

Abbildung 2: Architektur des JaMB für Java Framework

Die *Analyseeinheit* nimmt die zu analysierenden Daten entgegen und steuert den Analyseprozess unter Einbeziehung weiterer Funktionseinheiten. Über die Analyseeinheit können Daten auch direkt an JaMB übergeben werden: So kann eine Integration von JaMB in einer Entwicklungsumgebung erfolgen, indem die zu analysierenden Daten direkt aus der Versionsverwaltung heraus an JaMB übergeben werden.

Die Aufgabe, die zur Konfliktanalyse benötigten Daten aus den Eingabedateien zu gewinnen, übernimmt die *Extrahiereinheit*. Für Java gibt es dazu zwei Alternativen: Zum einen können diese Informationen aus den Quelltextdateien (.java-Dateien) entnommen werden. Zum anderen kann der Aufrufgraph durch eine Analyse der Bytecodedateien (.class-Dateien) aufgebaut werden, da die Spezifikation des Java

Bytecodeformats offen gelegt ist ([LinYel96], [GosSte96]). JaMB nutzt die zweite Alternative, da so auch Komponenten berücksichtigt werden, die ausschließlich als Bytecode vorliegen. Darüber hinaus können Referenzen direkt aus dem Java Bytecode extrahiert werden, da diese bereits bei der Übersetzung durch den Java Übersetzer aufgelöst werden. Bei einer Analyse der Quelltextdateien müssen hingegen die Mechanismen zur Auflösung von Referenzen selbst realisiert werden. Die durch die Extrahiereinheit gewonnenen Daten werden in einer Metastruktur abgelegt, die eine Erweiterung der Java Reflection API darstellt. Diese Erweiterung ist notwendig, da zusätzliche Informationen über den Aufrufgraphen notwendig sind, die nicht von der Java Reflection API zur Verfügung gestellt werden.

Die Erkennung potentieller und konkreter Konflikte übernimmt die *Detektoreinheit*. Es stehen zwei parallele Vererbungshierarchien für Konfliktpotential- und Konfliktdetektoren zur Verfügung. Durch Spezialisierung der dort definierten, abstrakten Oberklassen können neue Detektoren zur Erkennung weiterer Konfliktsituationen implementiert werden. Alle Detektoren, die innerhalb dieser Vererbungshierarchie definiert sind, werden automatisch von JaMB erkannt und können bei der Durchführung des Analyseprozesses entsprechend berücksichtigt werden. Dazu werden die von Java bereitgestellten Mechanismen zum dynamischen Laden von Klassen zur Laufzeit benutzt.

Wiederkehrende Teilschritte wie die Bestimmung geänderter Methodensignaturen einer Klasse sind in der *Analysetoolbox* implementiert und können damit allen Detektoren zur Verfügung gestellt werden. So können wiederkehrende Analyseschritte gemeinsam genutzt werden. Durch Nutzung der *Deltapuffereinheit* können die Ergebnisse der einzelnen Analyseteilschritte zwischengespeichert und dadurch die Verarbeitungsgeschwindigkeit gesteigert werden.

Die *Ausgabeeinheit* stellt die Ergebnisse der Analyse zusammen und gibt sie an den Aufrufer weiter. Sie stellt die zentrale Schnittstelle für die Integration in Fremdanwendungen dar. Über diese Schnittstelle können die ermittelten potentiellen und konkreten Konflikte aus JaMB an die aufrufende Anwendung übergeben werden. Dabei kann das Übergabeformat der Analysedaten mit Hilfe des Besucher-Musters anwendungsspezifisch definiert werden. Mit Hilfe der durch die Ausgabeeinheit realisierten, flexiblen Schnittstelle kann JaMB zur Unterstützung einer Versionsverwaltung in eine klassische Entwicklungsumgebung integriert werden. Dies ermöglicht die Durchführung der Konfliktanalyse direkt aus der Entwicklungsumgebung heraus.

3.2 Konflikterkennung in der Praxis: Das JaMB für Java GUI

Neben der Integration in Entwicklungswerkzeuge zur Unterstützung der Versionsverwaltung oder des Debuggers existiert JaMB auch als eigenständiges Werkzeug, das als Ergänzung zum Java Development Kit (JDK) eingesetzt werden kann. Die Konflikterkennung kann dabei zur Integrationszeit durchgeführt werden. Sobald eine neue Version des Basismoduls eintrifft, kann diese zusammen mit dem vorhandenen Erbmodul analysiert werden. Dazu kann JaMB über eine entsprechende GUI Anwendung mit einer graphischen Oberfläche aktiviert werden, die als Java Applikation in Swing realisiert ist (siehe Abbildung 3). Daneben steht ebenfalls noch eine Version für das Abstract Window Toolkit sowie eine Textvariante zur Stapelverarbeitung auf der Kommandozeile zur Verfügung.

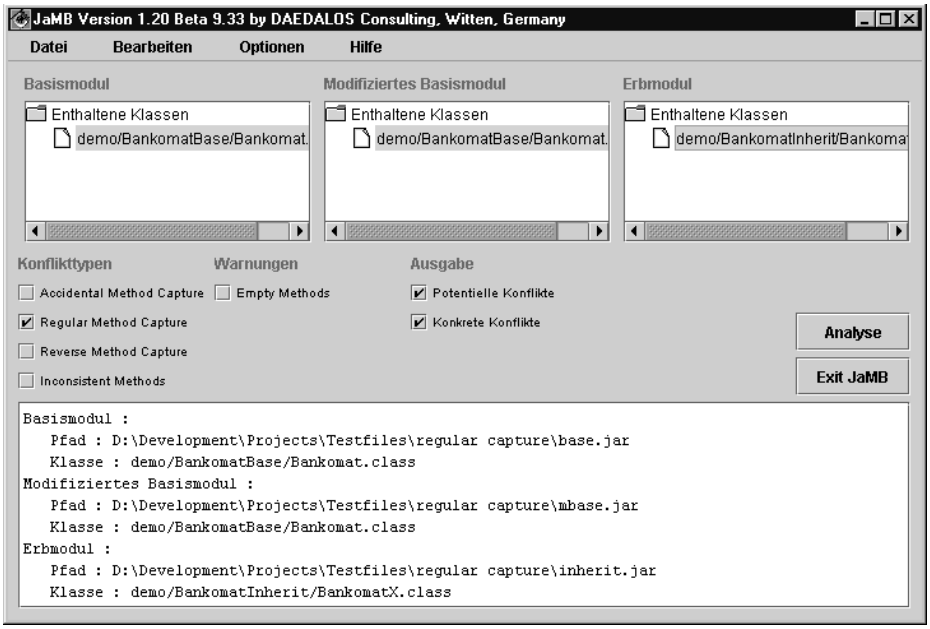

Abbildung 3: GUI Oberfläche für JaMB für Java

Diese GUI Anwendung übernimmt dabei die Aufbereitung der Ein- und Ausgabedaten, die von JaMB erwartet werden, und ersetzt somit die Anbindung an eine externe Entwicklungsumgebung. Es wird dabei über die gleiche Schnittstelle auf die Basisfunktionalität des JaMB Kerns zugegriffen, die im vorigen Abschnitt vorgestellt wurde. An der Funktionsweise des JaMB Kerns orientiert sich auch der Aufbau der GUI Anwendung: Zunächst werden die Eingabedaten in Form der zu untersuchenden Module ausgewählt. Es können dabei einzelne Bytecodedateien, Verzeichnisse oder JAR-Archive ausgewählt werden. Der Entwickler kann darüber hinaus die zu erkennenden Konflikttypen sowie die gewünschte Ausgabe als Konfliktpotential bzw. als konkrete Konflikte auswählen.

Nach Durchführung der Konflikterkennung werden die Ausgabedaten in Form des Analyseergebnisses ausgegeben. Die gefundenen Konflikte werden dabei durch die Oberflächenanwendung nach Klassen sortiert angezeigt. Für jeden Konflikt wird dabei erläutert, durch welche Zusammenhänge er ausgelöst wird. Dies ist in Abbildung 4 für das Beispiel *Regular Method Capture* dargestellt.

Mit Hilfe der durch JaMB zusammengestellten Informationen kann ein Entwickler schnell erkennen, in welchen Klassen Konflikte auftreten und welche Methoden betroffen sind. Zusammen mit der Konfliktbeschreibung geht daraus hervor, welche Verhaltensänderungen durch diese Konflikte möglich sind. Er ist somit in der Lage, das vorhandene Erbmodul an die neue Version des Basismoduls anzupassen und Konfliktsituationen gezielt zu korrigieren. Durch diese zusätzlichen Informationen wird die Gefahr reduziert, dass semantische Konflikte bei der Integration einer neuen Basiskomponente unentdeckt bleiben und erst dann bemerkt werden, wenn es bereits zu spät ist: Zur Laufzeit, wenn das System bereits im Einsatz ist und es dabei in einen nicht definierten Zustand gerät.

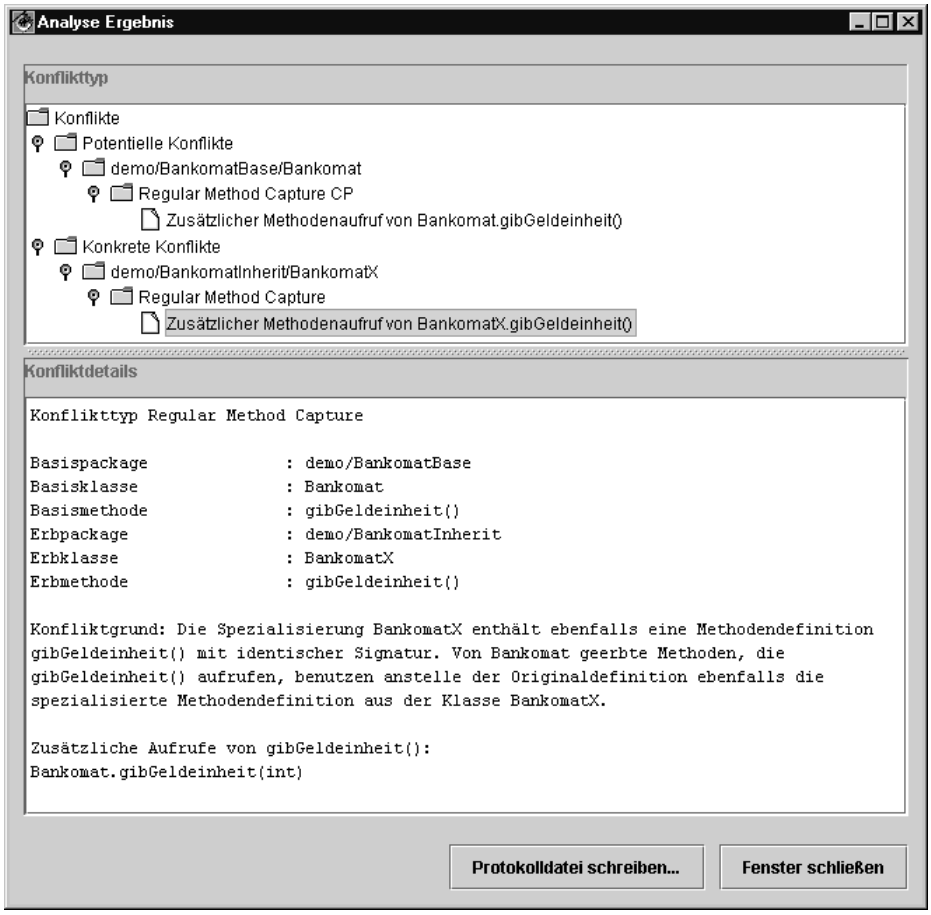

Abbildung 4: Ausgabe des Analyseergebnisses durch JaMB für Java

3.3 Fallstudien

Die Relevanz des vorgestellten Themas zeigt sich insbesondere bei der Analyse umfangreicher Klassenbibliotheken. Insbesondere bei Klassenbibliotheken, die am Anfang ihrer Entwicklung stehen und sich rasch weiterentwickeln, sind Änderungen die Regel, die zu potentiellen Konfliktsituationen führen. Inwieweit diese sich tatsächlich auf eine Anwendung auswirken hängt von der Struktur der Anwendung ab. Die Zahl möglicher konkreter Konfliktsituationen hängt auch davon ab, wie stark bei der Realisierung der Anwendung von Vererbung Gebrauch gemacht wird.

Mit Hilfe von JaMB für Java haben wir verschiedene Java Klassenbibliotheken bzgl. potentieller Konfliktsituationen untersucht, so z.B. das JDK 1.0.x im Vergleich zu den nachfolgenden Versionen (JDK 1.1.x und höher). Da die Analyse der kompletten Klassenbibliothek hohe Anforderung an Performance- und Speicher stellt, haben wir einzelnen Teile im Hinblick auf potentielle Konflikte geprüft. Um einen

kurzen Überblick zu bekommen sei hier als Beispiel die Analyse des Pakets **java.awt.*** beim Übergang vom JDK 1.0.x auf das JDK 1.1.x erwähnt. Zur Analyse der rund 50 Klassen benötigt JaMB für Java auf einem Arbeitsplatzrechner knapp 3 Minuten. Es werden insgesamt 498 potentielle Konflikte gefunden, davon 351x *Accidental Method Capture*, 39x *Regular Method Capture*, 108x *Specialization Negation* und 0x *Reverse Methode Capture*. Ob diese potentiellen Konflikte auch tatsächlich konkrete Konflikte verursachen, hängt davon ab, wie die jeweilige Anwendung unter Verwendung der betroffenen Klassen realisiert ist.

Nicht berücksichtigt in dieser Analyse sind Änderungen, die durch das Umbenennen einer Klasse oder das Verschieben in ein anderes Paket hervorgerufen werden, da diese bei der Umstellung der Anwendung auf die neue Version in jedem Fall vom Programmierer angepasst werden müssen. In zukünftigen Versionen von JaMB für Java ist vorgesehen, den Entwickler bei dieser Anpassung zu unterstützen und die betroffenen Klassen entsprechend in der Konfliktanalyse zu berücksichtigen.

4. Fazit und Ausblick

Bei der Weiterentwicklung objektorientierter Systeme stellen semantische Konflikte einen bedeutenden Risikofaktor dar. Unsere Untersuchungen haben gezeigt, dass bei der Verwendung von Basiskomponenten sowie der Weiterentwicklung bestehender Systeme eine Vielzahl an potentiellen Konflikten auftreten können, die bei Kombination mit existierendenden Anwendungen zu konkreten Fehlersituationen führen und das Verhalten in unabsehbarer Art und Weise verändern. Die Gefahr, die von semantischen Konflikten ausgeht, blieb lange Zeit unbeachtet, so dass aktuelle Entwicklungsumgebungen noch keine Werkzeuge anbieten, die den Entwickler bei der Suche nach semantischen Konflikten unterstützen. Auch die Möglichkeiten, die Versionsverwaltungen bieten, reichen bei weitem nicht aus, obwohl prinzipiell alle benötigten Daten durch zusätzliche Analyseschritte ermittelt werden können. Insbesondere die Gefahr, dass semantische Konflikte bei der Integration einer neuen Version einer Basiskomponente zunächst unerkannt bleiben und im laufenden Betrieb Schäden verursachen, erhöht die Notwendigkeit, entsprechende Werkzeuge zur Integration in den Entwicklungsprozess anzubieten.

Aus diesen Erfahrungen heraus ist JaMB für Java entstanden, ein Werkzeug zur Erkennung semantischer Konflikte in Java Projekten. Ausgehend von der Analyse des Java Bytecodes werden semantische Konfliktsituationen erkannt, die zu einem Fehlverhalten des Softwaresystems führen können. Aktuell werden von JaMB für Java alle im ersten Abschnitt beschriebenen Konfliktsituationen erkannt, allerdings sind dabei Situationen möglich, in denen eine erkannte Konfliktsituation nicht zu einem Fehlverhalten des Systems führt. Deshalb ist ein wichtiger Punkt für die zukünftige Weiterentwicklung, dass die Bindung eines Methodenaufrufs an genau eine Methodendefinition ermittelt wird. Darüber hinaus sind weitere Detektoren zur Erkennung neuer Konflikttypen geplant, die unter Verwendung der vorgegebenen Schnittstelle der Detektoreinheit implementiert und automatisch in den Analyseprozess mit einbezogen werden.

Für die Zukunft ist weiterhin eine Umsetzung von JaMB für Smalltalk anhand von VisualAge Smalltalk geplant, die direkt in die Entwicklungsumgebung integriert ist und auf die Daten der zugehörigen Versionsverwaltung Envy zugreifen kann.

5. Literaturverzeichnis

[GosSte96] Gosling J, Joy B. und Steele G. The Java Language Specification. Addison-Wesley, 1996.

[KicLam92] Kiczales G. und Lamping J. Issues in the Design and Documentation of Class Libraries. In Proceedings of OOPSLA '92, ACM SIGPLAN Notices, Vol. 27, No. 10, S. 435-451, 1992.

[Lam93] Lamping J. Typing the Specialization Interface. In Proceedings of OOPSLA '93, ACM SIGPLAN Notices, Vol. 28, No. 10, S. 201-214, 1993.

[LinYel96] Lindholm T. und Yelling F. The Java Virtual Machine Specification. The Java Series, Addison Wesley, 1997.

[Mez97] Mezini M. Maintaining the Consistency of Class Libraries During their Evolution. In Proceedings of OOPSLA '97, Sigplan Notices Vol. 29, No. 10, S.1-22, 1997.

[MPDB99] Mezini M., Pipka J.U., Dittmar T. und Boot W. Detecting Evolution Incompatibilities by Analyzing Java Binaries. In Proceeding of TOOLS USA '99, IEEE Press.

[MiSe98] Mikhajlov L. und Sekerinski E. A Study of The Fragile Base Class Problem. In Proceedings of ECOOP '98, LNCS 1445, S. 355-382, Springer Verlag.

[Min96] Minsky N. Law-Governed Regularities in Object Systems. In Theory and Practice of Object Systems (TAPOS), Vol. 2, No. 4, John Wiley, 1996.

[PaSchw95] Palsberg J. und Schwartzbach, M.I. Safety Analysis versus Type Interference. In Information and Computation 118(1), S.128-141, 1995.

[SLMDH96] Steyaert P., Lucas C., Mens K. und D'Hondt T. Reuse Contracts: Managing the Evolution of Reusable Assets. In Proceedings of OOPSLA '96, ACM SIGPLAN Notices, Vol. 31 No. 10, S. 268-286, 1996.

[Szy97] Szyperski C. Component Software. Beyond Object-Oriented Programming. Addison Wesley, 1997