Universität Bremen
Bibliothekstraße 1
28359 Bremen

Fachbereich 3
Informatik
AG Datenbanken

Diplomarbeit

Technologieübergreifende Verifikation von Laufzeit-Annahmen mit UML- und OCL-Modellen

Erweiterung des USE-Monitors um eine CLR-Schnittstelle

Daniel Honsel

5. August 2013

Erstprüfer:	Prof. Dr. Martin Gogolla
Zweitprüfer:	Prof. Dr. Hans-Jörg Kreowski
Betreuer:	Prof. Dr. Martin Gogolla M. Sc. Lars Hamann

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1. Einleitung

In der heutigen Softwareentwicklung nehmen Modelle eine zentrale Rolle ein. Neben den Prozessmodellen sind insbesondere Software-Modelle von großer Bedeutung. Sie eigenen sich besonders, um Software objektorientiert zu entwerfen. Unter modellgetriebener Software-Entwicklung versteht man einen Prozess, bei dem eine Technologie unabhängige Spezifikation der fachlichen Anforderungen durch spezielle Werkzeuge zu Implementierungsmodellen (z. B. Java oder .NET Code) transformiert wird (siehe [LL07]). Wenn ein Modell einen hohen Stellenwert in der Softwareentwicklung hat, folgt daraus, dass dessen Korrektheit ebenfalls von großem Interesse ist. Um Modelle basierend auf der Unified Modeling Language (UML) und Object Constraint Language (OCL) zu spezifizieren und zu validieren, steht das Programm USE zur Verfügung.

Die Weiterentwicklung von USE schreitet seit dem Jahr 1998 stetig voran. Eines der neusten Features von USE ist das seit 2011 enthaltene Monitor-Plug-in [HGK11]. Der Monitor ermöglicht die Analyse von Java Programmen zur Laufzeit. Dafür wird eine Spezifikation, die sich zwischen der Plattform unabhängigen Spezifikation des zu analysierenden Programms und dem Plattform abhängigen Modell (in diesem Fall der Java Byte Code) befindet, benötigt. Die gewünschte „Zwischen-Spezifikation“ beinhaltet nur einen kleinen Ausschnitt der in der Plattform unabhängigen spezifizierten Klassen und Assoziationen und wird von USE geladen. Anschließend ist der Monitor in der Lage sich mit dem zu analysierenden Programm zu verbinden, die Daten auszulesen und von USE validieren zu lassen. Untersucht werden von USE statische Systemzustände und das dynamische Verhalten sowie die mit letzterem verbundenen Änderungen am Systemzustand des Programms.

Um den Monitor auch für andere Technologien, die der Virtuellen Maschine von Java ähneln, verwenden zu können, wurde das Design des Monitors in [HGH12] überarbeitet. Dort ist ein neue Schicht eingeführt worden – das Metamodell für Virtuelle Maschinen. Es beinhaltet genau die Informationen, welche USE benötigt, um ein Modell validieren zu können. Es muss für konkrete Virtuelle Maschinen implementiert werden. Für Java ist das bereits erfolgreich geschehen (siehe [HGH12]).

Aufgrund der bisherigen Entwicklung stellt sich die Frage: Ist der Monitor auf beliebige Plattformen, beruhend auf einer ähnlichen Technologie wie die der Java Virtuellen Maschine, erweiterbar? Genau an diesem Punkt setzt die vorliegende Diplomarbeit an und versucht die Frage für die Common Language Runtime (CLR) von

Microsoft zu beantworten. Dabei spielt die Interoperabilität zwischen den verschiedenen Technologien eine bedeutende Rolle.

1.1. Ziel der Arbeit

Ein Ziel dieser Diplomarbeit ist es, die für einen CLR-Adapter benötigten Informationen aus einem von der CLR geladenen Programm korrekt auslesen zu können. Diese müssen nach dem Erhalt so verwaltet werden, dass sie dem Monitor in einer für ihn verständlichen Form übergeben werden können. Da sich die zum Auslesen der Daten benötigte CLR Debug API aus COM-Komponenten zusammensetzt, wird dieser Teil in C++ geschrieben.

Ein aus der Verwendung unterschiedlicher Technologien resultierendes Ziel ist der Datenaustausch zwischen dem Teil des CLR-Adapters auf der Java Seite, der das Metamodell für eine Virtuelle Maschine implementiert, und dem Teil auf der C++-Seite. Für die Interoperabilität zwischen Java und C++ wird JNI verwendet.

Sind die beiden Zwischenziele erreicht, soll USE durch die neue CLR-Schnittstelle in der Lage sein, .NET Programme zur Laufzeit anhand von UML- und OCL-Modellen validieren zu können. Dieses stellt das erklärte Ziel der vorliegenden Arbeit dar.

1.2. Aufbau der Arbeit

Die vorliegende Diplomarbeit ist wie folgt aufgebaut: Zuerst wird in die theoretischen Grundlagen eingeführt, danach folgt eine detaillierte Beschreibung des Designs und der Implementierung des CLR-Adapters und abschließend folgen die Auswertung mit verschiedenen Testszenarien sowie das Fazit mit Zukunftsaussichten.

In Kapitel 2 werden die theoretischen Grundlagen erläutert, auf denen diese Diplomarbeit basiert. Hierbei wird auf Softwaremodelle und dessen Modellierungssprachen UML und OCL eingegangen, um anschließend besser auf die aktuellen Möglichkeiten zur Überprüfung von OCL Invarianten, Vor- und Nachbedingungen zur Laufzeit eingehen zu können. Anhand der vorgestellten Grundlagen wird in das Testsystem „AnimalWorld“ für den CLR-Adapter eingeführt. Dieses Testprogramm wird in C# implementiert und soll zur Laufzeit von USE unter Verwendung des CLR-Adapters verifiziert werden. Danach wird USE mit dem Monitor Plug-in beschrieben, um die in dieser Arbeit entwickelte Erweiterung des Monitors um eine CLR-Schnittstelle erläutern zu können. Darauf folgend wird in Kapitel 2 noch die Struktur des .NET Frameworks von Microsoft und der Unterschied zu Java näher beschrieben. Abschließend wird auf die Interoperabilität zwischen Java und nativem Code via JNI eingegangen, um die spätere Kommunikation beider Technologien untereinander verständlich machen zu können.

Die CLR Debug API und dessen COM-Schnittstellen werden in Kapitel 3 beschrieben. Dabei wird nur auf die vom CLR-Adapter benötigten Komponenten eingegangen und dessen Verantwortlichkeiten für den Adapter werden dargelegt. Dies ist für das Verständnis der Implementierung des CLR-Adapters von großer Bedeutung.

Kapitel 4 stellt den Hauptteil der vorliegenden Abschlussarbeit dar. Es beschreibt den Entwurf und die Implementierung des CLR-Adapters. Begonnen wird mit einer Übersicht der Komponenten, die für den in dieser Arbeit präsentierten Ansatz des Monitorings von .NET Programmen verwendet werden. Danach wird das Design des C++-Teils des Adapters in seinen Einzelheiten ausführlich erklärt. Der CLR-Adapter ist für das Auslesen der Daten eines laufenden, verwalteten .NET Programms verantwortlich. Es werden neben möglichen Optimierungsstrategien auch alternative und nicht funktionierende Ansätze beschrieben. Dabei wird ebenso auf vorherige Entwurfsideen des Adapters eingegangen. Das Kapitel schließt mit der Erläuterung des in Java implementierten Teils des CLR-Adapters ab. Dabei werden auch die für die Kommunikation zwischen Java und C++ benötigten JNI-Methoden und deren Funktionen erläutert.

In Kapitel 5 werden die Ergebnisse dieser Diplomarbeit dargestellt und ausgewertet. Für die Ergebnisgenerierung werden verschiedene Testfälle konstruiert, die insbesondere unter den Aspekten Korrektheit und Performance entworfen werden. Für die Testfälle wird unter anderem das Testsystem „AnimalWorld“ verwendet. Des Weiteren wird USE unter Verwendung des CLR-Adapters mit einem Open Source Programm verbunden, um dessen Laufzeitverhalten zu untersuchen.

Die Zusammenfassung der einzelnen Arbeitsschritte, das Fazit und die Zukunftsaussichten des implementierten CLR-Adapters beinhaltet Kapitel 6. Dabei werden auch verschiedene Verbesserungsvorschläge und Erweiterungsmöglichkeiten des CLR-Adapters diskutiert, die den Rahmen der vorliegenden Diplomarbeit gesprengt hätten. Abschließend werden weitere Ideen bezüglich der Erweiterung des Monitors auf andere Plattformen angeschnitten.

Dem Anhang sind unter anderem die im Rahmen dieser Arbeit erstellten USE Spezifikationen, ein Ausschnitt der Metadaten des .NET Programms „AnimalWorld“ und der für den CLR-Adapter generierte JNI-Header zu entnehmen. Des Weiteren wird dort die physikalische Struktur des CLR-Adapters erklärt. Auf der dieser Arbeit beigelegten CD sind die Implementierungen des CLR-Adapters und der Testprogramme enthalten.

2. Grundlagen

In diesem Kapitel werden die Technologien und die theoretischen Grundlagen vorgestellt, die dieser Arbeit zu Grunde liegen.

Zum einen werden Aspekte der Modellierung behandelt, wie sie in einem Entwicklungsprozess von Software vorkommen. Zum anderen werden die Technologien beschrieben, welche für den *CLR-Adapter* benötigt werden. Des Weiteren wird das Programm *USE* beschrieben. Dessen Monitor-Plug-in wird in der vorliegenden Arbeit um eine CLR-Schnittstelle, den CLR-Adapter, erweitert.

2.1. Modelle in der Softwareentwicklung

In den meisten Naturwissenschaften werden Modelle erstellt, um allgemeingültige Aussagen zu treffen, Vermutungen zu konkretisieren und sich ein Bild (das Modell) der Realität zu machen. Der Bedeutung von Modellen ist in der Informatik noch ein größerer Stellenwert einzuräumen, da die Endpunkte der Arbeit ebenfalls ein Modell darstellen. Die Spezifikation einer Software, das fertige Programm und auch die Prozessmodelle der Projektorganisation stellen Modelle der Softwareentwicklung dar. Die theoretischen Überlegungen zum Begriff des Modells in diesem Abschnitt gehen auf [LL07] zurück.

Im Folgenden werden nun die für diese Arbeit wichtige graphische Modellierungssprache UML sowie deren Bestandteil, die textuelle Sprache OCL, näher erläutert. Mit Hilfe von UML und OCL wird während der Beschreibung der Sprachen beispielhaft das Programm, auf welchem die Auswertungen (siehe Kapitel 5 auf Seite 57) basieren, eingeführt und im Detail erläutert.

2.1.1. Unified Modeling Language (UML)

Die Unified Modeling Language (UML) [OMG13] ist eine graphische Modellierungssprache, die unterschiedliche Sichten zur Verfügung stellt. Jede Sicht ist eine Teilmenge der UML und repräsentiert jeweils einen Aspekt eines Systems [RJB05].

Nach [RJB05] kann man die verschiedenen Sichten in vier Hauptgruppen unterteilen, die im Folgenden mit den wichtigsten Bestandteilen zusammengefasst vorgestellt werden.

- **Strukturelle Spezifikation.** Die Gruppe beinhaltet die statische Sicht (Klassendiagramm) und die Design-Schicht (Komponentendiagramm, Kollaborationsdiagramm). Des Weiteren ist das Use-Case-Diagramm Bestandteil der strukturellen Spezifikation.
- **Dynamisches Verhalten.** Sie enthält Sichten für den Zustand und die Interaktion mit den Diagrammen Zustandsautomat und Sequenzdiagramm.
- **Physikalischer Aufbau.** In dieser Gruppe ist nur ein Element, das Verteilungsdiagramm, enthalten.
- **Modellverwaltung.** Diese Gruppe enthält nur das Paketdiagramm, das in zwei Sichten enthalten ist.

Im weiteren Verlauf wird nur noch näher auf die für diese Arbeit wichtige Sicht, die statische aus der strukturellen Spezifikation, eingegangen.

Ein Klassendiagramm gibt detailliert Aufschluss darüber, wie die Daten und das Verhalten eines Systems strukturiert sind [RQZ07]. Es enthält statische Eigenschaften sowie ihre Beziehung untereinander in Form von Klassen, Attributen, Operationen und Assoziationen. Für objektorientierte Software dient das Klassendiagramm typischerweise als das Modell, welches es zu implementieren gilt. Die Testsoftware „AnimalWorld“ für die Auswertung ist nach dem Klassendiagramm in Abbildung 2.1 auf der nächsten Seite implementiert und wird im folgenden Abschnitt näher erläutert.

Entwurf der Testsoftware „AnimalWorld“

Das Programm „AnimalWorld“ ist entworfen worden, um den CLR-Adapter zu testen und beispielhaft zu zeigen, welche Funktionalität er im Rahmen der vorliegenden Arbeit erhalten hat.

Wie man Abbildung 2.1 auf der nächsten Seite entnehmen kann, besteht das Programm aus fünf Klassen, zwei Assoziationen und einer Generalisierung.

Die Klasse *Pet* repräsentiert die Basisklasse für ein Haustier. Durch die Assoziation *ParentHood* kann ein Tier in der Rolle des Vaters mit mehreren Tieren in der Rolle eines Kindes verbunden werden und somit eine Verwandtschaft modelliert werden. *ParentHood* stellt eine reflexive Assoziation dar, weil – ähnlich wie bei einem Kreis – Anfangs- und Endpunkt zusammen liegen. Das Attribut *Gender* repräsentiert die möglichen Geschlechter eines Tieres, dabei steht die 1 für männlich und die 2 für weiblich. Gender ist ein Beispiel für eine Eigenschaft, dessen Typ sehr gut durch einen Aufzählungsdatentypen repräsentiert werden kann. Diese werden allerdings in der aktuellen Version des CLR-Adapters nicht unterstützt.

Mit Hilfe der Klasse *Color*, welche eine Farbe darstellt, und der Assoziation *PetColor* wird einem Objekt der Klasse *Pet* genau eine Farbe zugeordnet. Eine Farbe kann allerdings mehreren Tieren zugeordnet sein.

Die eigentlichen Haustiere sind Spezialisierungen der Klasse *Pet*. Im einzelnen sind dies die Klassen *Mouse*, *Cat* und *Dog*. Bei der Klasse *Cat* ist noch die Operation `Cat::Eat(m : Mouse)` erwähnenswert, mit der modelliert werden soll, dass eine Katze eine Maus verspeist. Die USE (siehe Abschnitt 2.2 auf Seite 13) Spezifikation zum Erstellen des UML-Modells kann Anhang B.1 entnommen werden.

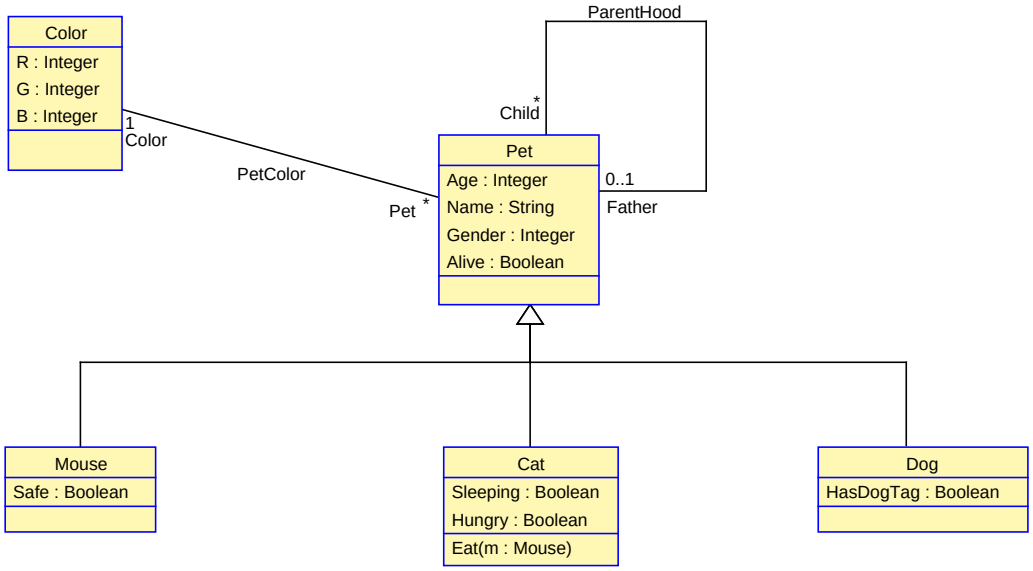

Abbildung 2.1.: Klassendiagramm des Programms „AnimalWorld“

Objekte sind in der UML nichts weiter als Ausprägungen einer Klasse. Eine Klasse repräsentiert den Typ eines Objekts [RQZ07]. Objektdiagramme stellen dementsprechend den Zustand eines Systems zu einem bestimmten Zeitpunkt dar, indem sie alle Instanzen der Klassen mit ihren Attribut-Werten und Links abbilden. Ein Beispiel für ein Objektdiagramm zeigt Abbildung 2.2 auf der nächsten Seite. Es wurde mit USE (siehe Abschnitt 2.2 auf Seite 13) erstellt und repräsentiert einen Snapshot zu dem in Abbildung 2.1 dargestellten Klassendiagramm. Die USE Spezifikation dafür ist dem Anhang B.2 zu entnehmen.

Das Objektdiagramm auf Abbildung 2.2 auf der nächsten Seite zeigt eine Katzenfamilie, deren Vater Tom ist, zwei Mäuse und einen Hund. Die Katze Ada hat auf dem Diagramm keine Verwandten. Alle Tiere sind einer Farbe, davon gibt es vier, zugeordnet.

2.1.2. Object Constraint Language (OCL)

Ein Klassendiagramm, wie z. B. das in Abschnitt 2.1.1 auf Seite 5 beschriebene, stellt die Struktur eines Programms dar. Die dort enthaltenen Annahmen sind jedoch

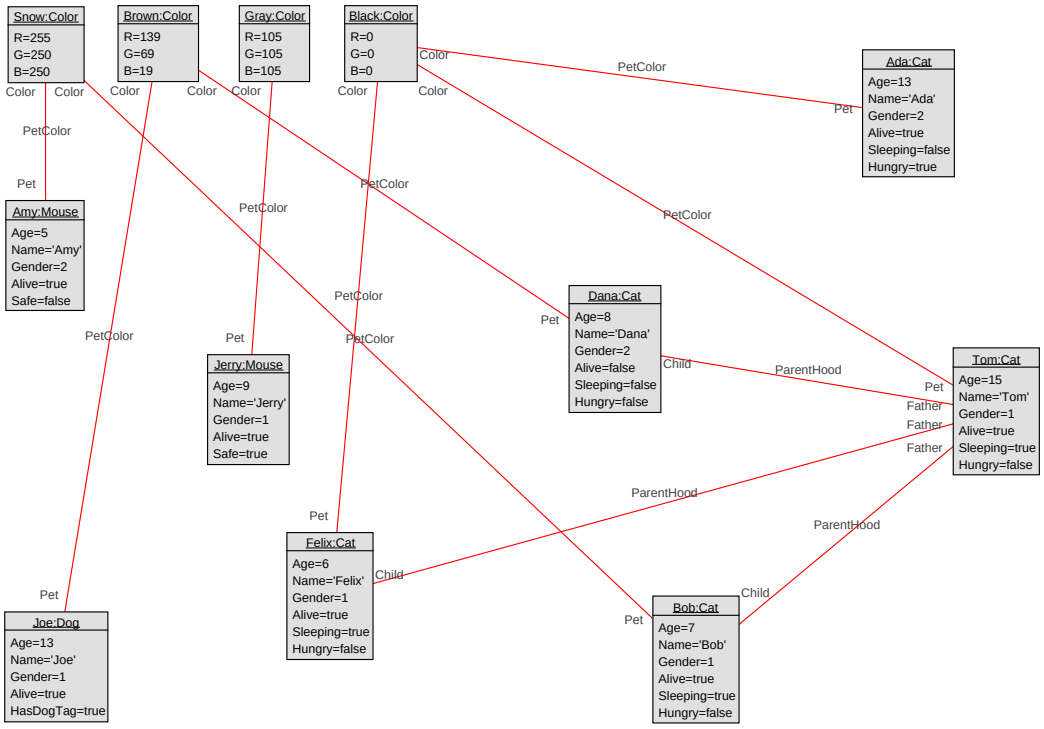

Abbildung 2.2.: Objektdiagramm der „AnimalWorld“

meistens nicht ausreichend, um ein Modell validieren zu können. Dafür muss das UML-Modell um weitere Annahmen über das Laufzeitverhalten erweitert werden. Da sich das mit der graphischen UML-Notation nicht darstellen lässt, verwendet man dazu die textuelle Object Constraint Language (OCL) [OMG10]. Sie ist Teilmenge der UML. Im Folgenden werden nur die Teile der OCL erläutert, welche für die aus dem Vertragsmodell [LL07] bekannten Zusicherungen notwendig sind.

Vertragsmodell (nach Hoare, 1969): Das sogenannte Hoare-Tripel (siehe Gleichung (2.1)*) bildet den Grundbaustein des Vertragsmodells und besagt Folgendes: Wenn vor der Ausführung der Operation S die Vorbedingungen P gelten und S terminiert, gelten nach der Ausführung von S die Nachbedingungen Q.* (Definition sinngemäß [LL07, S. 438] entnommen.)

$$\{P\}S\{Q\} \tag{2.1}$$

Klasseninvarianten Klasseninvarianten sind Bedingungen, die *während* des gesamten Lebenszyklus eines Objekts einer Klasse eingehalten werden müssen. Eine Ausnahme bilden laufende Operationen. Während der Ausführung einer Operation darf die Prüfung einer Klasseninvarianten fehlschlagen. Sie haben jedoch vor einem Operationsaufruf sowie auch danach zu gelten.

Vorbedingungen Wird auf einem Objekt eine Operationen ausgeführt, so müssen diese Bedingungen *vor* der Ausführung gültig sein.

Nachbedingungen Wird auf einem Objekt eine Operationen ausgeführt, so müssen diese Bedingungen *nach* der Ausführung gültig sein.

Annahmen und Zusicherungen der Testsoftware „AnimalWorld“

In diesem Abschnitt werden ein paar Annahmen über das Testprogramm, die „AnimalWorld“ (siehe Abschnitt 2.1.1 auf Seite 6), besprochen und mit Hilfe der OCL formal beschrieben. Die OCL Beschreibung definiert Invariante, Vor- und Nachbedingungen.

1. Annahme: Das Alter eines Haustiers ist nicht negativ.

Listing 2.1: OCL Invariante: Alter nicht negativ

```
context Pet inv AgeNotNegative:
  Age >= 0
```

2. Annahme: Der Name eines Tieres identifiziert es eindeutig.

Listing 2.2: OCL Invariante: Eindeutiger Name

```
context p1:Pet inv NameIsUnique:
  Pet.allInstances->forAll(p2 |
    p1.Name = p2.Name implies p1 = p2)
```

3. Annahme: Wenn ein Pet die Rolle eines Vaters einnimmt, so kann es an der Assoziation Fatherhood nicht gleichzeitig als Child teilnehmen.

Listing 2.3: OCL Invariante: Ein Vater darf nicht sein Kind sein

```
context p1:Pet inv FatherNotChild:
  not(p1.Father = p1) and
    p1.Child->excluding(p1) = p1.Child
```

4. Annahme: Wenn es einen bekannten Vater gibt, so muss dieser männlich sein.

Listing 2.4: OCL Invariante: Ein Vater muss männlich sein

```
context p1:Pet inv FatherIsMale:
  p1.Father = null or p1.Father.Gender = 1
```

5. Annahme: Die Attribute der Klasse Color stellen RGB-Werte dar und dürfen deshalb jeweils den Wertebereich [0,255] nicht verlassen.

Listing 2.5: OCL Invariante: Einhaltung eines Wertebereichs

```
context Color inv ColorRange:
  R >= 0 and R <= 255 and
  G >= 0 and G <= 255 and
  B >= 0 and B <= 255
```

6. Annahme: Damit die Methode Cat::Eat() ausgeführt werden kann, muss folgendes gelten: Die Maus darf sich nicht in Sicherheit befinden, die Katze darf nicht schlafen und muss hungrig sein. Nachdem die Methode ausgeführt wurde, soll die Katze nicht mehr hungrig sein und die Maus soll, nachdem sie verspeist wurde, nicht mehr am Leben sein.

Listing 2.6: OCL Vor-/Nachbedingungen einer Methode

```
context Cat::Eat(m : Mouse)
  pre UnsafeMouse:       not(m.Safe)
  pre CatIsHungry:       Hungry
  pre CatIsNotSleeping:  not(Sleeping)
  pre CatIsAlive:        Alive
  post CatNotHungry:     not(Hungry)
  post VictimIsDead:     not(m.Alive)
```

Die oben getroffenen Annahmen sind nur eine Auswahl aus der Menge der möglichen Annahmen, die man für die „AnimalWorld“ treffen kann. Sie sollten ausreichend genug sein, um OCL anhand von leichtverständlichen Beispielen zu erläutern und später die Funktionalität des CLR-Adapters zu veranschaulichen.

Alle Codebeispiele sind der textuellen Spezifikation der „AnimalWorld“ entnommen (siehe Anhang B.1).

2.1.3. Verifikation und Validierung von Modellen zur Laufzeit

Dieser Abschnitt beschäftigt sich mit Möglichkeiten UML- und OCL-Modelle zu validieren und zu verifizieren. Zuerst werden die Begriffe Verifikation und Validierung frei nach [ZGK04] definiert. Anschließend wird in momentan übliche Praktiken eingeführt.

- **Verifikation**: Verifikation ist die Überprüfung, ob die Spezifikation oder der Entwurf eines Systems mit der Implementierung übereinstimmt. Mit anderen Worten wird die Frage beantwortet: „Wurde das System korrekt entwickelt?“ (siehe [ZGK04, S. 356])
- **Validierung** Validierung bezeichnet die Untersuchung, ob ein fertiges System den zu Anfang spezifizierten Anforderungen entspricht. Mit anderen Worten

wird die Frage beantwortet: „Wurde das richtige System entwickelt?“ (siehe [ZGK04, S. 356])

Wurde nun ein UML-Klassendiagramm mit OCL-Bedingungen erweitert, sodass es valide erscheint, braucht man eine Möglichkeit, diese Zusicherungen in der Implementierung sicherzustellen. Dieser Abschnitt erläutert eine paar Ansätze, wie man die an ein System gestellten Bedingungen zur Laufzeit überprüfen kann.

Die Artikel [Avi+10; Fro+07] geben einen Überblick über gängige Methoden und Werkzeuge um OCL-Bedingungen zur Laufzeit zu testen. Die Wichtigsten werden im Folgenden zusammenfassend dargestellt.

1. Bedingungen manuell in der benutzten Programmiersprache schreiben Dieser Ansatz ist recht einfach umzusetzen. Hierfür hat man nichts weiter zu tun, als vor der eigentlichen Ausführung des Codes einer Methode die Vorbedingungen mit Hilfe von `if`-Anweisungen zu überprüfen und nach der Ausführung des eigentlichen Methoden-Codes, bevor die Methode zurückkehrt, die Nachtbedingen ebenso zu überprüfen.

Wenn eine Programmiersprache `assert`-Anweisungen unterstützt, kann man diese anstelle von `if`-Anweisungen verwenden. Dieser Ansatz hat den Vorteil, dass man die Überprüfung via Compiler-Schalter ein- und ausschalten kann. Ein Beispiel dafür ist dem Listing 2.7 zu entnehmen.

Bei diesem Ansatz kann man zwar an Ort und Stelle beim Nichteinhalten einer Bedingung angemessen auf die entsprechende Verletzung reagieren, jedoch ist der Code schwer zu warten. Bei sich ändernden Annahmen ist die Konsistenz zwischen Implementierung und Modell nicht leicht zu gewährleisten, insbesondere da die Möglichkeit besteht, dass die gleiche Bedingung an mehreren Stellen im Code zu überprüfen ist.

Des Weiteren bietet Microsoft für .NET Sprachen eine spezielle Möglichkeit das Vertragsmodell im Code umzusetzen. Sie nennt sich „Code Contract“ [Mic13g] und stellt Schlüsselwörter für Vor- und Nachbedingungen sowie Invarianten zur Verfügung. Unterstützung durch den Visual Studio Assistenten und die Testumgebung ist für diese Technologie vorhanden.

2. Bedingungen in Klassen verpacken Hierbei wird jede OCL-Bedingung in eine eigene Klasse geschrieben. Dadurch wird die Validierung von der Logik gekapselt. Des Weiteren hat man so die Möglichkeit, alle Bedingungen in einer dafür geeigneten Datenstruktur abzulegen und zentral in einem Programm zu verwalten. Ein solcher Ansatz wäre auch für die Wiederverwendbarkeit von Bedingungen von Vorteil, da man so eine Bedingung an mehreren Programmstellen verwenden könnte. Ein weiterer Vorteil liegt in der Möglichkeit, Bedingungen sogar zur Laufzeit ein- und auszuschalten.

Listing 2.7: Vor- und Nachbedingungen der Methode Cat::Eat(m : Mouse), ausgedrückt in C#-Programmcode mit Hilfe von assert-Anweisungen

```
public void Eat(Mouse m)
{
  Debug.Assert(!m.Safe, "UnsafeMouse");
  Debug.Assert(this.Hungry, "CatIsHungry")
  Debug.Assert(!this.Sleeping, "CatIsNotSleeping");
  Debug.Assert(!this.Alive, "CatIsAlive");

  // do some work
  this.Hungry = false;
  m.Alive = false;

  Debug.Assert(!this.Hungry, "CatNotHungry");
  Debug.Assert(!m.Alive, "VictimIsDead");
}
```

Dieser Gedanke ist – nach Meinung des Autors – bei genauer Betrachtung nur eine mögliche Erweiterung einer der oben aufgeführten Ansätze. Man könnte ihn auch als ein Architekturmuster für die Überprüfung von Bedingungen im Allgemeinen ansehen. Einige der anderen Ansätze lassen sich mit diesem kombinieren.

3. Verwendung von Kompiler-Werkzeugen, die das Vertragsmuster unterstützen Eine Möglichkeit, das Vertragsmuster umzusetzen, ist die Java Modeling Language (JML) [Lea13] oder deren nicht ganz so mächtiges Pendant auf der .NET-Seite Spec# [Mic13f].

Um OCL-Bedingungen mit JML sicherzustellen, wird der Code mit speziellen Kommentaren annotiert. Diese Kommentare werden vom JML-Compiler übersetzt und zur Laufzeit ausgewertet.

JML bietet weitaus mehr Möglichkeiten als nur die Bedingungsüberprüfung mittels des Vertragsmusters. So kann z. B. mit Hilfe von Datengruppen von privaten Implementierungsdetails abstrahiert werden [Cha+06].

4. Verwendung von aspektorientierten Werkzeugen oder Sprachen Als solche sind unter anderem auf der Java-Seite AspectJ [Col+13] und auf der .NET-Seite PostSharp [Pos13] als prominente Beispiele zu erwähnen.

Aspektorientierte Programmierung ist ein Paradigma, welches es erlaubt, *Aspekte* an sogenannten *Pointcuts* in den Programmfluss einzubauen oder dazwischen zuschieben. So ist eine OCL-Bedingung in diesem Beispiel als ein Aspekt aufzufassen.

Da die Aspekte zusätzliches Verhalten via Codeinjektion in das Programm einpflegen, ist eine Änderung des originalen Quellcodes nicht notwendig. Die Kenntnis des Quellcodes, insbesondere von den Namen der privaten Felder, ist beim Erstellen der Aspekte jedoch von Nöten. Der Programmablauf ist allerdings nicht mehr der des originalen Programms, da durch Codeinjektion das Programm manipuliert wird.

Alle der vorgestellten Möglichkeiten, OCL-Bedingungen zu validieren und zu verifizieren, setzten Veränderungen am Quellcode oder zumindest eine gute Kenntnis der Implementierungsdetails der zu untersuchenden Software voraus.

2.2. USE: UML-based Specification Environment

Das UML- und OCL-Werkzeug USE [GBR07; 13c] wird von der Arbeitsgruppe Datenbanken der Universität Bremen veröffentlicht. Die erste Version ist bereits 1998 erschienen und seitdem wird USE kontinuierlich weiterentwickelt.

Mit USE ist es möglich, UML-Modelle zu definieren, diese mit OCL-Bedingungen zu erweitern und zu validieren. Die für das Testprogramm „AnimalWorld“ (siehe Abschnitt 2.1.1 auf Seite 5) erstellten Modelle und Graphiken wurden mit USE erstellt.

Seit der Version 3.0.0 wurde die Kommandozeilen-Sprache von USE durch SOIL (Simple OCL-based Imperative Language) [BG11], einer eigenen Sprache, ersetzt. In SOIL wurde auch die Generierung des Snapshots für die „AnimalWorld“ beschrieben (Spezifikation siehe Anhang B.2).

Eine weitere Möglichkeit, einen Snapshot einer UML/OCL-Spezifikation zu erstellen, bietet USE mit der Sprache ASSL (A Snapshot Sequenze Language) [GBR03]. Mit Hilfe von ASSL lässt sich die Erstellung eines Snapshots automatisieren. Damit hat der Anwender die Möglichkeit unterschiedliche Testfälle zu entwickeln, diese leicht zu variieren und das Modell zu validieren. Für die vorliegende Arbeit wurde ASSL nicht verwendet.

Einen Überblick über USE gibt Abbildung 2.3 auf Seite 15. Ihr kann man das bereits bekannte Klassen- und Objektdiagramm entnehmen, zusätzlich noch eine Übersicht über die definierten Klassen, Assoziationen, Invarianten sowie Vor- und Nachbedingungen. Des Weiteren sind unter dem Klassendiagramm noch ein Fenster mit den Auswertungen der definierten Invarianten und ein Fenster, das die Anzahl der Objektinstanzen angibt, zu erkennen. Unter dem Objektdiagramm sind noch drei Fenster mit zusätzlichen Informationen dargestellt. Sie zeigen die Anzahl der Links sowie detaillierte Eigenschaften eines oder aller instantiierter Objekte.

Zusammenfassend lässt sich sagen, dass man mit USE die Korrektheit von UML- und OCL-Modellen, die Einhaltung von Multiplizitäten, statische Bedingungen (Klasseninvarianten) und dynamische Bedingungen (Vor- und Nachbedingungen) validieren

kann. Damit stellt USE seinen Plug-ins wertvolle Funktionalitäten zur Verfügung, von denen auch die vorliegende Arbeit profitiert.

2.2.1. USE-Monitor-Plug-in

Der Monitor [HGK11; Ham+12], der in dieser Arbeit um eine Schnittstelle zum Microsoft .NET Framework erweitert wird, ist ein in Java geschriebenes Plug-in für USE. Dadurch steht dem Monitor mit USE ein seit Jahren stabiles Werkzeug für die Validierung zur Verfügung und er kann sich die Erweiterungen von USE zunutze machen. Eine dieser Erweiterungen ist der Protokollzustandsautomat [HHG12], der für den dynamischen Teil des Monitors von Interesse ist.

So wie der Monitor in [HGK11; Ham+12] eingeführt wird, ist er nur für Programme zu gebrauchen, die von der Java Virtual Machine (JVM) ausgeführt werden. Um ihn mit beliebigen Plattformen kommunizieren zu lassen, wurde das in Abbildung 2.4 auf Seite 16 dargestellte Metamodell einer Virtuellen Maschine entworfen [HGH12]. Dieses Metamodell wurde bereits erfolgreich für die JVM in Form eines JVM-Adapters implementiert [HGH12]. Die vorliegende Arbeit konzentriert sich auf eine Erweiterung dieser Schnittstelle um eine neue Technologie. Hierfür wurde die CLR von Microsoft gewählt. Die im Folgenden erläuterte Funktionsweise des JVM-Adapters ist jedoch grundlegend für alle weiteren Adapter-Implementierungen.

Da der dynamische Teil des Adapter den Rahmen einer Diplomarbeit sprängen würde, wird nur der für die vorliegende Arbeit benötigte statische Teil des vorhandenen JVM-Adapters detailliert erläutert.

Für den statischen Teil des Monitorings sind die in Abbildung 2.4 auf Seite 16 links oben abgebildete Adapter, der `JVMAdapter` für Java oder für die CLR der in dieser Arbeit entwickelte `CLRAdapter` verantwortlich. Ein Adapter muss die Schnittstelle `VMAdapter` implementieren. Sie definiert die benötigten Methoden um aktiv einen aktuellen Snapshot des zu untersuchenden Programms abzurufen. Der Adapter ist nicht in der Lage den Monitor zu benachrichtigen. Für diesen dynamischen Teil ist der `MonitorNotifier` verantwortlich.

Unten auf der Graphik sind USE-Klassen mit ihren Beziehungen zu den korrespondierenden Klassen des Metamodells abgebildet. Um einen Snapshot zu erstellen werden die Klassen `VMType`, `VMObject` und `VMField` verwendet. All diese Klassen haben eine eindeutige ID, mit der sie innerhalb der Virtuellen Maschine zu identifizieren sind. Des Weiteren ist der Adapter dafür zuständig, primitive Typen in Datentypen umzuwandeln, die von USE verstanden werden.

Um den Monitor mit einem in einer Virtuellen Maschine laufenden Programm zu verbinden, sind je nach Technologie unterschiedliche Daten notwendig. Für Java sind dies der Host und der Port. Für die CLR wird die Prozess ID benötigt. Nachdem eine Verbindung hergestellt wurde, wird das zu untersuchende Programm gestoppt

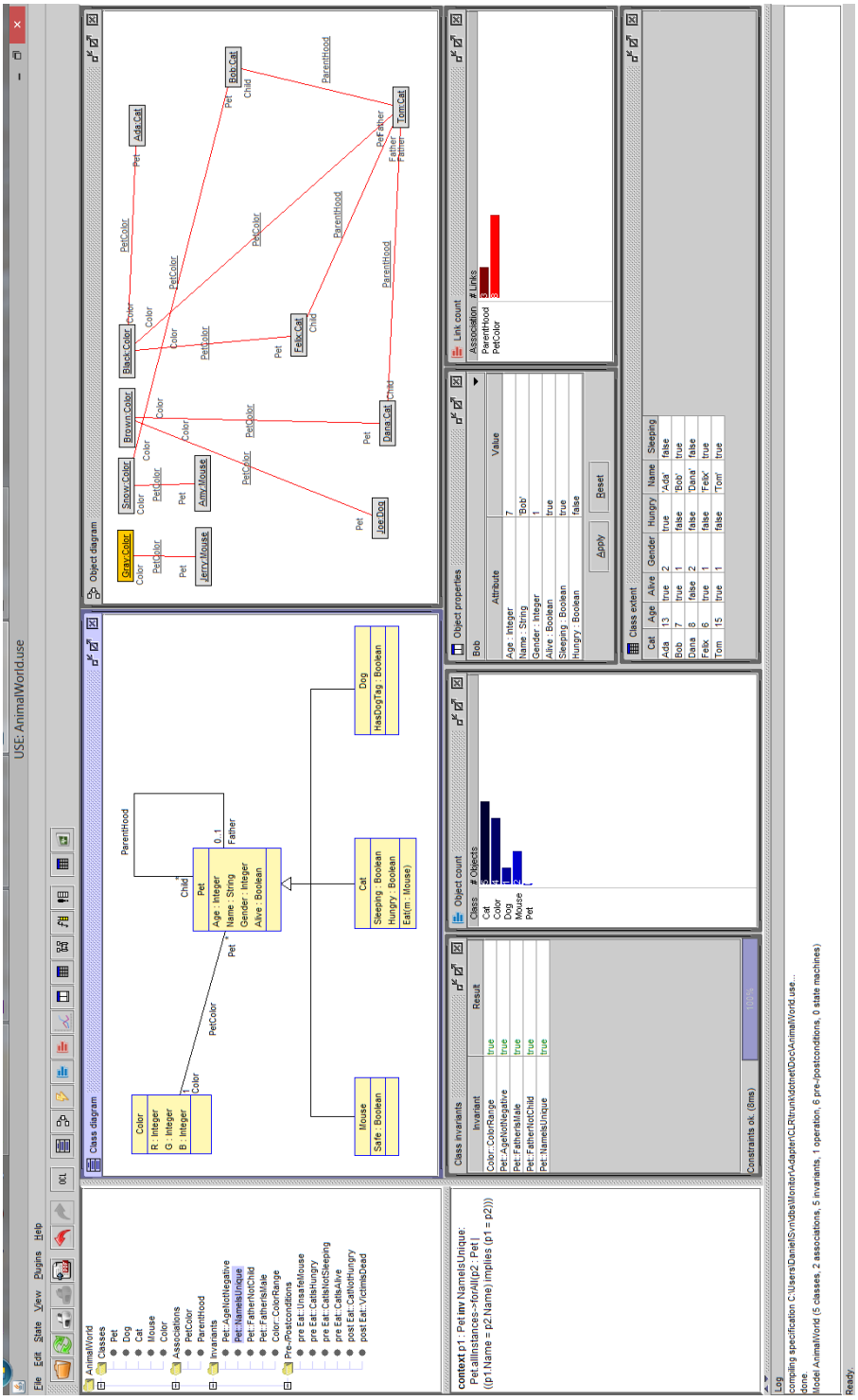

Abbildung 2.3.: USE – eine Übersicht

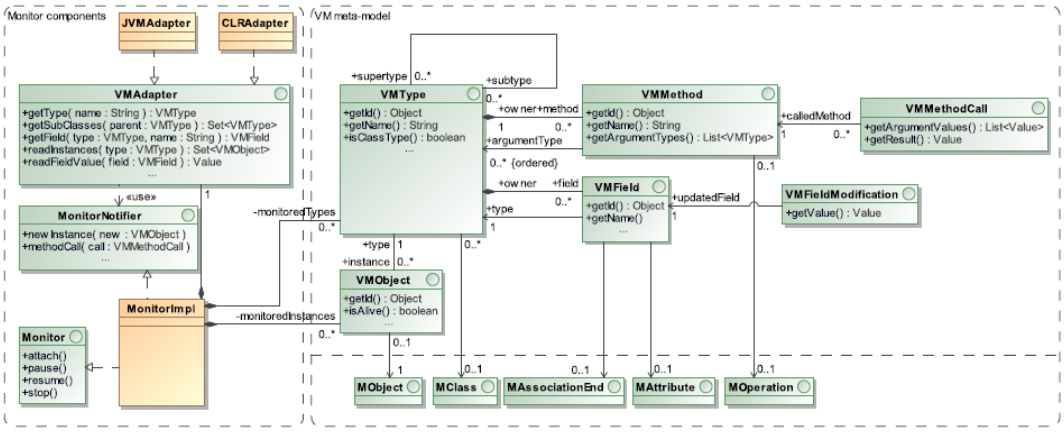

Abbildung 2.4.: Metamodell der Virtuellen Maschine mit den Komponenten das Monitors (entnommen [HGH12])

und der Monitor lässt sich durch den Adapter einen Snapshot erstellen. Dazu werden folgende Schritte der Reihe nach ausgeführt (siehe auch [HGK11]).

1. Für die Typen, die bei dem Monitoring-Durchgang von Interesse sind, werden alle Instanzen geladen. Welche Typen das sind, wird in einer USE-Spezifikation definiert. Diese Spezifikation ist ein UML-Modell mit speziellen Annotationen für den Monitor, wie z. B. Namensräume oder Attribute zum Ignorieren. In den meisten Fällen definiert dieses Modell auch nur einen Ausschnitt aus einem Gesamt-Modell des zu untersuchenden Programms. Für die Testsoftware „AnimalWorld“ ist es jedoch das komplette Modell (siehe Anhang B.1). Geladene Objekte werden in einer geeigneten Datenstruktur vom Monitor gespeichert.

2. Für jedes in Schritt 1 erstellte Objekt werden nun die Eigenschaftswerte gelesen. Für primitive Typen werden einfach die entsprechenden OCL-Typen erstellt. Referenztypen, dessen Klassen im USE-Modell spezifiziert sind, werden in der im ersten Schritt erstellten Datenstruktur gesucht und referenziert.

3. Zum Schluss werden noch für alle Assoziationen die entsprechenden Links erzeugt.

Nachdem der Monitor aktiv gewesen ist und einen Snapshot vom Adapter geliefert bekommen hat, wird das zu untersuchende Programm fortgesetzt und der Monitor wechselt in den passiven Modus. Nun lauscht er auf Statusveränderungen. Der dynamische Teil ist nicht im Rahmen dieser Arbeit enthalten.

2.3. Microsoft .NET Framework

Das .NET Framework [Mic13h] ist eine von Microsoft entwickelte Klassenbibliothek mit Laufzeitumgebung. Diese Laufzeitumgebung wird als *Common Language Runtime (CLR)* bezeichnet und ist der *JVM* ähnlich. Die erste Version wurde im Jahr 2002 veröffentlicht. Seitdem gab es mehrere Updates und zu jeder neuen Version des .NET Frameworks gab es fast auch immer eine neue Version des Visual Studios. Das Visual Studio (VS) ist die Entwicklungsumgebung für .NET von Microsoft. Aktuell sind das .NET Framework 4.5 mit dem VS 2012, die auch für diese Arbeit verwendet werden.

Jede neue Version des .NET Frameworks stellt eine Erweiterung des Funktionsumfangs zur Verfügung, beinhaltet jedoch nicht automatisch eine neue CLR. Abbildung 2.5 zeigt diesen Versionsverlauf.

Wie aus [Mic13i] hervorgeht, ist das .NET Framework 4.5 nur mit Windows Vista und aufwärts kompatibel, was den in dieser Arbeit entwickelten CLR-Adapter auf diese Plattformen beschränkt.

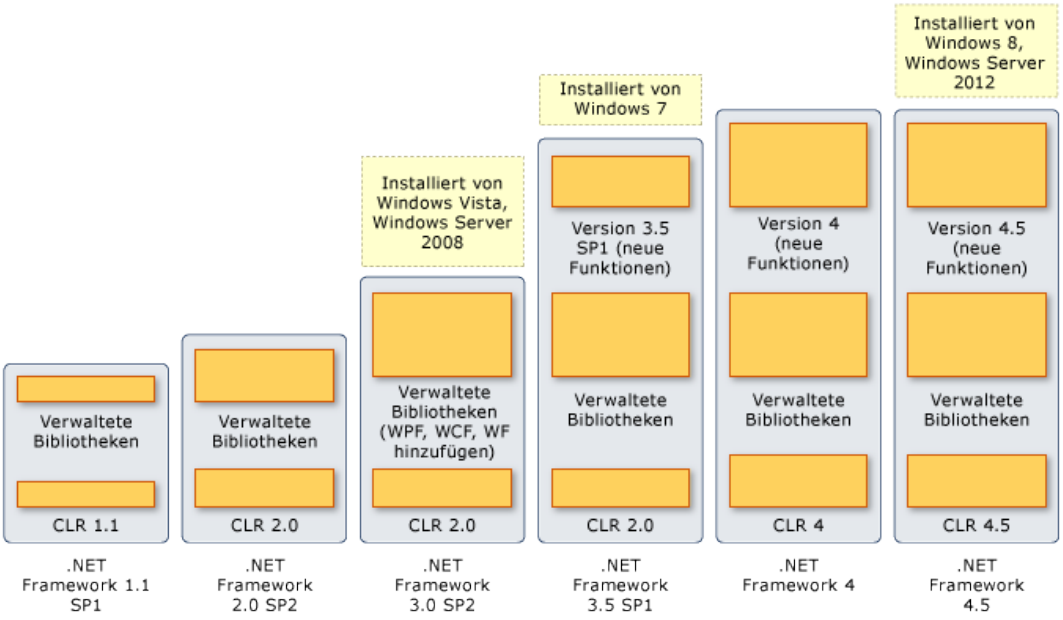

Abbildung 2.5.: Versionsverlauf .NET Framework (entnommen [Mic13i])

2.3.1. Spezifikation (Common Language Infrastructure, CLI)

Der Standard, nach welchem das .NET Framework und die CLR implementiert wurden, nennt sich Common Language Infrastructure (CLI) und wurde von Microsoft

bei der ECMA als ECMA-335 [Mic12] eingereicht. Aktuell ist er in der 6. Version vom Juni 2012. Genau spezifiziert sind dort unter anderem die folgenden Punkte, die in [Koe05; Ric10] ausführlich beschrieben sind.

Allgemeines Typ-System (Common Type System, CTS)

Das Herzstück der CLI ist das Common Type System (CTS). Hier werden alle verfügbaren Typen, die in einer CLI-Hochsprache verwendet werden können, definiert. Dieses System garantiert die Typsicherheit der CLI.

Definiert sind dort unter anderem Typen sowie dessen Zugriffsmodifizierer, Regeln für Vererbung und die Lebenszyklen von Objekten. Nicht alle im CTS definierten Typen oder Regeln werden von den CLI-Hochsprachen verwendet. Damit ist das CTS die Menge aller möglichen Sprachmittel der CLI, das kann man auch Abbildung 2.6 entnehmen. Durch das CTS wird auch erreicht, dass verschiedene Programmiersprachen der CLI untereinander kommunizieren können.

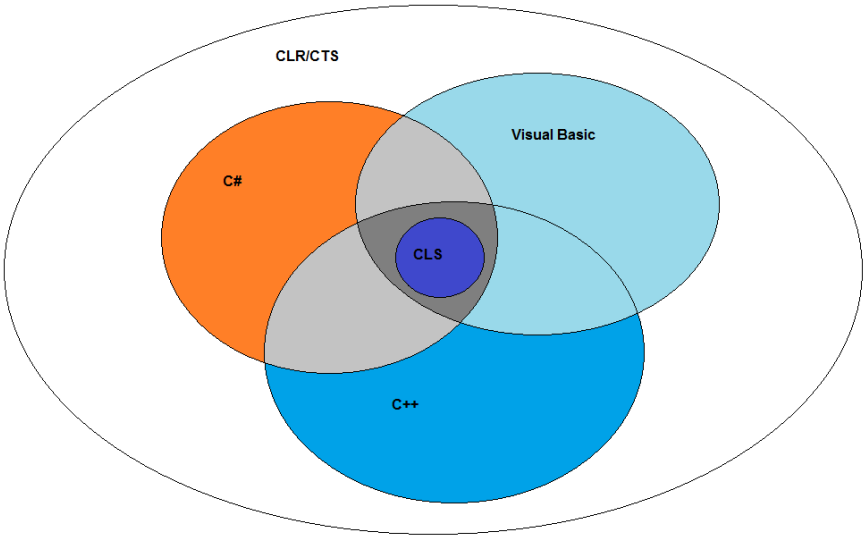

Abbildung 2.6.: Das CTS als Menge aller möglichen Elemente der CLR-Sprachen und die CLS als kleinste gemeinsame Menge aller Sprachen

Metadaten

Metadaten enthalten Typinformationen in einer von der Programmiersprache unabhängigen Form. Dadurch sind sie von allen CLI-kompatiblen Tools zu lesen und zu interpretieren. Dies gilt auch für die Debugging-Tools, welche einen essentiellen Bestandteil dieser Arbeit bilden.

Die Metadaten enthalten sämtliche Informationen über Typen und Methoden und sind zur Laufzeit des Programms verfügbar. Einen Ausschnitt der Metadaten des für die Auswertung erstellten C#-Programms kann dem Anhang C entnommen werden.

Allgemeine Sprachspezifikation (Common Language Specification, CLS)

Die CLS bildet die kleinste gemeinsame Menge aller möglichen Sprachelemente (Regeln, Typen und Funktionen) der CLI und fungiert als Vertrag zwischen der CLI und einer anderen Sprache. Dies ist auch Abbildung 2.6 auf der vorherigen Seite zu entnehmen. Jede Sprache der CLI muss also mindestens diesen Teil implementieren.

Virtuelles Ausführungssystem (Virtual Execution System, VES)

Das VES führt den verwalteten Code, bestehend aus Modulen, aus. Module sind einzelne Dateien – die kleinsten Elemente – die ausführbaren Code enthalten. Typen, die in einem geladenen Modul definiert sind, können durch dessen Metadaten eingelesen werden.

Wenn man nach [Mic13c] geht, ist das VES der CLR gleichzusetzen. Die CLR ist die proprietäre Microsoft-Implementierung der CLI. Auch Open Source Implementierungen wie z. B. Mono [Xam13a] setzten Teile des CLI Standards um.

Im letzten Absatz ist von Modulen die Rede und in [Mic13c] wird dargelegt, dass das VES Assemblies lädt. Assemblies bestehen aus Modulen und haben die unter Windows bekannten Dateiendungen EXE für ein ausführbares Programm oder DLL für eine Bibliothek. Wenn ein Programm mit dem Visual Studio kompiliert wird, besteht eine Assambly immer aus einem Modul. Nur wenn das Programm mit dem Compiler auf der Konsole mit bestimmten Flags kompiliert wird, kann ein Assambly aus mehreren Modulen gebaut werden [Ric10]. Ein Assembly enthält zudem noch landesspezifische Informationen, eine Version und Sicherheitsanforderungen.

Die CLR lädt nicht nur den Code, sie ist auch verantwortlich für die automatische Speicherverwaltung mit Hilfe des Garbage Collectors und die Ressourcenverwaltung.

Auf Abbildung 2.7 auf der nächsten Seite kann man den Weg einer C#-Datei bis zum ausführbaren Programm verfolgen. Zuerst wird aus der Codedatei durch den C#-Compiler ein ausführbares Assembly, in welchem das Modul enthalten ist, erstellt. Der dabei generierte Common Intermediate Language (CIL)-Code wird von der CLR geladen und ausgeführt. Durch Just-In-Time (JIT)-Kompilierung der CIL abstrahiert die CLR dabei von maschinenspezifischen Details.

Der vom C#-Compiler generierte CIL-Code ist für alle Sprachen, die der CLI-Spezifikation entsprechen, identisch. Somit ist es theoretisch möglich, dass ein Assembly mehrere Module enthält, welche in unterschiedlichen Sprachen geschrieben sind. Diese Sprachinteroperabilität ist ein interessantes Feature der CLR, wodurch erreicht

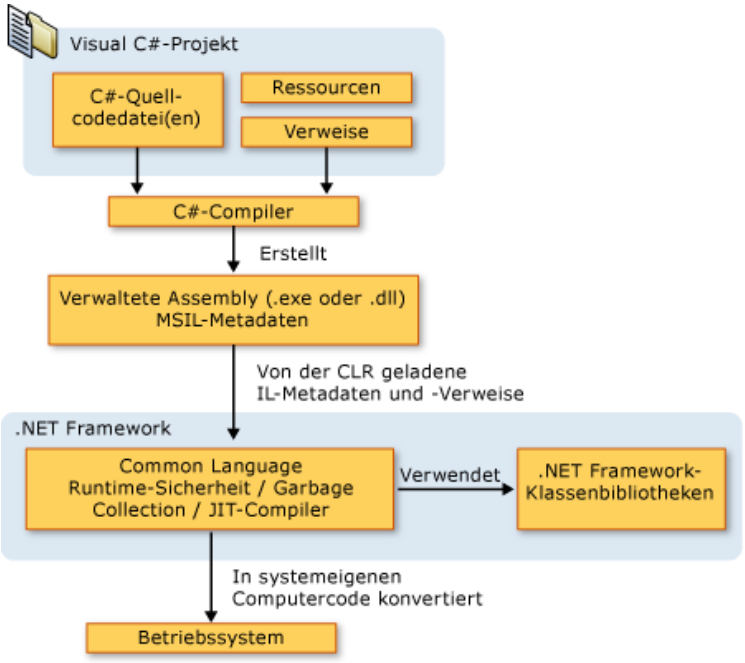

Abbildung 2.7.: Von einer C#-Quellcodedatei zum ausführbaren Programm (entnommen [Mic13c])

wird, dass ein Programm aus mehreren Modulen – geschrieben in unterschiedlichen CLI-Sprachen – bestehen kann.

Als *Managed Code* oder auch als *Verwalteter Code* bezeichnet man alles, was im VES ausgeführt wird. Dieser verwaltete Code liegt meistens als Modul in CIL vor. Um mit *Unmanaged Code* oder *Nativen Code*, der im Gegensatz zu dem Bytecode der CIL als Maschienencode einer spezifischen Rechnerarchitektur vorliegt, interagieren zu können, werden von der CLR COM[1]-Schnittstellen benutzt. Im weiteren Verlauf dieser Arbeit wird von nun an ausschließlich von verwalteten oder nativen Code gesprochen und alle Untersuchungen und Ergebnisse beziehen sich auf Programme, die aus verwaltetem Code bestehen.

2.3.2. Unterschiede zu Java

Java und das .NET Framework verfolgen im Prinzip ähnliche Konzepte und stellen beide eine Virtuelle Maschine zur Verfügung, welche den Java Bytecode bzw. den Code in der .NET Common Intermediate Language (CIL) ausführt. Java geht allerdings etwas mehr in die Richtung eine Sprache für viele Plattformen zu bieten, wohingegen .NET eine Plattform für viele Sprachen zur Verfügung stellt. Im

[1] Componont Object Model: Eine von Microsoft für Windows entwickelte Technologie als Schnittstelle zwischen verschiedene Programmiersprachen zur Erzeugung dynamische Objekte.

Jahr 2010 gab es bereits .NET-Compiler für über 60 Sprachen, darunter auch Funktionale wie z. B. Haskell oder ältere wie z. B. Fortran und Cobol [STZ10]. Beide Klassenbibliotheken bieten mehr als 10 000 Elemente.

Bei der Plattformunabhängigkeit hat eindeutig Java einen Vorsprung, da es Versionen für Windows, Unix/Linux und Mac OS gibt [STZ10]. Die von Microsoft veröffentlichte CLR gibt es hingegen nur für Windows und andere Implementierungen der CLI, die es für mehrere Plattformen gibt, wie z. B. das Open Source Projekt Mono [Xam13a], laufen der aktuellen CLR meistens ein oder zwei Versionen hinterher.

Als Entwicklungsumgebung kommen für Java Eclipse und NetBeans häufig zum Einsatz. Davon ist Eclipse die am weitesten verbreitete und modular erweiterbar. Für .NET-Programme, besonders unter Windows für die CLR, kommt primär das Visual Studio (VS) von Microsoft in Frage. Neue Versionen des Visual Studios erscheinen meistens zusammen mit einer neuen Version des .NET Frameworks. Dadurch ist die optimale Unterstützung neuer Features des Frameworks, wie z. B. WPF oder dem Entity Framework, durch Designer und Assistenten gewährleistet. Auch das VS lässt sich, über Add-ins mit, aus der Open-Source Welt bekannten, Tools wie z. B. Subversion, erweitern [STZ10].

Im Folgenden werden kurz die Unterschiede zu Java erläutert. Zum einen wird auf die Sprachelemente eingegangen und zum anderen werden die Debug-Schnittstellen verglichen.

Unterschiede bezüglich der Sprache

Wenn man die sprachlichen Mittel von .NET-Sprachen mit Java vergleicht, fällt zunächst auf, dass primitive Datentypen (int, bool, etc.) unterschiedlich verwaltet werden. In Java sind sie echte Datentypen und werden auf dem Stack verwaltet [Ora13]. Es existieren „Wrapperklassen“, welche die primitive Datentypen repräsentieren und auf dem Heap verwaltet werden. Die Umwandlung erfolgt durch automatisches Autoboxing (siehe [Sch05]). Dies ist wichtig, wenn man eine Datenstruktur verwenden möchte, die nur Referenzen verwaltet. In .NET werden die primitiven Datentypen auf die korrespondierenden Typen der Klassenbibliothek gemappt, z. B. `int` auf `System.Int32`. Diese Typen erben von `System.ValueType`, welcher von `System.Object` abgeleitet ist. Dadurch ist es auch leicht möglich, eigene Wertetypen zu definieren, indem von `System.ValueType` abgeleitet wird. Wertetypen sind allerdings keine Objekte im eigentlichen Sinn, wie es aus [Wie12] hervorgeht. Sie werden auf dem Stack gespeichert, wohingegen Referenzwerte auf dem Heap verwaltet werden. Für den Programmierer wirken sie allerdings wie Objekte, da Instanz-Methoden wie z. B. `123.ToString()` für ein `Int32` aufgerufen werden können. Wenn Referenzen von Wertetypen benötigt werden, ist auch hier ein automatisches Boxing dafür verantwortlich (siehe [Ric10]).

In .NET stellen auch Referenzen auf Funktionen spezielle Typen (Events und Delegates) dar, diese gibt es in Java nicht. Des Weiteren sind in Java alle Methoden virtuell, dies muss in .NET-Sprachen über entsprechende Schlüsselwörter gekennzeichnet werden [Wie12].

Ab dem .NET Framework 3.5 stellt Microsoft eine SQL ähnliche Abfragesprache, die Language-Integrated Query (LINQ) [Mic13e], in den .NET-Sprachen zur Verfügung. Damit ist es möglich in einheitlicher Syntax verschiedene Datenspeicher, wie z. B. Objektmengen aus dem Hauptspeicher, Datenbanken oder XML-Dokumente abzufragen. Dank der Integration von LINQ-Befehlen in die Sprachsyntax von .NET-Sprachen kann bereits der Compiler die Ausdrücke überprüfen [STZ10]. Ein vergleichbares Konzept ist in Java nicht zu finden.

Software wird häufig in Module mit speziellen Funktionen unterteilt. Um modulare Abhängigkeiten, vor allem auch zwischen verschiedenen Versionen eines Moduls, einfach auflösen zu können, stehen Informationen über Abhängigkeiten und Versionen im .NET Framework in den Metadaten einer Assembly. Dies löst die Probleme, die auch als „DLL-“ oder „JAR-Hell“ bekannt geworden sind, für das .NET Framework [STZ10].

Unterschiede bezüglich der Debug API

Java bietet mit der Java Platform Debugger Architecture (JPDA) [Ora11] mächtige sowie auch einfach zu benutzende Tools für das Debuggen von Software an, welche in einer Java Virtual Machine (JVM) läuft. Diese Tools können aus einem anderen Java Programm heraus benutzt werden.

Die CLR-Debug-Schnittstelle setzt sich, wie in Kapitel 3 auf Seite 25 zu sehen ist, aus COM-Komponenten zusammen. Um diese zu verwenden, stehen nun zwei Möglichkeiten zu Verfügung. Entweder man benutzt diese Schnittstelle aus einem nativen C++-Programm heraus oder man stellt die Schnittstelle einem verwalteten C#-Programm über Interop-Wrapper zur Verfügung. Für diese Arbeit wurde ein Debugger in C++ geschrieben.

Ein Beispiel für einen in C# geschriebenen, verwalteten Debugger liefert Microsoft für das .Net Framework 4.0 [Mic13a]. Anhand dieses Beispiels ist zu erkennen, dass die Interop-Wrapper einen Großteil des Programms ausmachen.

Beide Debug-Schnittstellen erlauben es, dass sich der Debugger mit einem laufenden Programm verbindet.

2.4. Interoperabilität zwischen Java und nativen C++ Bibliotheken

Um zwischen Java-Programmen und in C++ geschriebenen Bibliotheken Daten austauschen zu können, kann das Java Native Interface (JNI) [Gor98; Ora12] verwendet werden.

Wenn ein Java-Programm Methoden enthält die in einer in C++ geschriebenen Bibliothek implementiert werden sollen, so müssen die Methoden, die auf der C++-Seite implementiert werden sollen, mit dem Schlüsselwort `native` gekennzeichnet werden. Nachdem das Java-Programm kompiliert wurde, kann mit Hilfe des Programms `javah.exe` eine Header-Datei generiert werden. Listing 2.8 zeigt den Aufruf für den CLR-Adapter. Der generierte JNI-Header kann dem Anhang E entnommen werden.

Listing 2.8: Automatische Generierung des JNI-Headers

```
javah -jni -o CLRAdapter.h _
      org.tzi.use.monitor.adapter.clr.CLRAdapter
```

JNI ist im Java JDK enthalten und kann von einem beliebigen C++-Programm verwendet werden. Dazu wird der Header `jni.h`[2] eingebunden und der aus der Java-Klasse generierte Header implementiert. Der C++-Teil bietet die folgenden Möglichkeiten: Erstellung beliebiger Java-Objekte, Methodenaufruf beiderseitig mit beliebigen Parametern und Rückgabewerten sowie Fehlerverwaltung.

Die C++-Bibliothek, in Form einer DLL, muss jetzt noch von der Java-Klasse, in der die nativen Methoden deklariert sind, geladen werden. Dies passiert, wie in Listing 2.9 gezeigt, mittels statischer Initialisierung.

Listing 2.9: Statische Initalisierung der nativen Bibliothek

```
public class CLRAdapter extends AbstractVMAdapter {
        static {
                System.loadLibrary("clradapter");
        }
        // ...
}
```

Damit die DLL vom Java-Programm gefunden werden kann, muss bei dieser Variante noch der Pfad zur DLL bekannt gemacht werden. Das passiert mittels JVM-Argument `-Djava.library.path` beim Start des Programms.

JNI ermöglicht es alle für den Monitor benötigten Java-Objekte aus dem C++-Programm heraus zu erstellen und dem Monitor zu übergeben.

[2] Enthalten im JDK Installationsordner unter „include“.

3. Die CLR Debug API

Dieses Kapitel beschreibt zuerst die grundlegende Arbeitsweise der CLR-COM-Schnittstellen und geht danach detailliert auf den in der vorliegenden Arbeit verwendeten Teil der CLR Debug API ein.

3.1. Grundlegende Funktionalität

Um die CLR Debug API besser verstehen zu können, ist es sinnvoll ein paar Grundlagen über die CLR und deren Schnittstellen zu wissen. Bevor tiefer in die Debug API eingestiegen wird, werden diese erläutert.

Die CLR ist implementiert als Bündel von COM-Komponenten [BS02]. Genauer gesagt ist sie ein COM-Server. Jede Windowsanwendung kann als Host für die CLR dienen. Dazu muss sie nur die Funktion `CLRCreateInstance`, die in *MSCorEE.dll*[1] implementiert ist, aufrufen [Ric10]. Dieser Aufruf kann ein Objekt, welches die Schnittstelle `ICLRMetaHost` implementiert, zurückliefern. Nun kann der Host mit Hilfe der Funktion `GetRuntime` aus `ICLRMetaHost` die gewünschte Version der CLR laden. Nachdem die CLR erfolgreich geladen ist, wird eine sogenannte *AppDomain* erstellt, welche als Container für die zu ladenden Assemblies dient.

Ein Programm, das die CLR hosted, bietet viele Möglichkeiten die CLR speziellen Bedürfnissen anzupassen. Man kann z. B. Einfluss auf die Speicherverwaltung der CLR nehmen oder detailliert darüber bestimmen, wie Assemblies geladen werden sollen. Auf diese Thematik wird in dieser Arbeit nicht näher eingegangen. Eine ausführliche Beschreibung, wie man eine CLR hosted und welche Vorteile man dadurch geboten bekommen, liefert [Pra09].

Bevor die vom CLR-Adapter verwendeten Debug API Schnittstellen detailliert beschrieben werden, wird zusammenfassend erläutert, wie ein beliebiges Windowsprogramm in der Lage ist, die CLR zu hosten (siehe [Ric10]).

Wenn ein Programm – bestehend aus verwaltetem Code – hinsichtlich der instantiierten Objekte, derer Zustände und Zustandsveränderungen untersucht werden soll, so stehen zwei COM-Schnittstellen zur Auswahl. Zum einen gibt es den Profiler [Mic13j] und zum anderen den Debugger [Mic13j]. Für diese Arbeit, die sich im CLR-Teil

[1] Gewöhnlich liegt die Datei im Verzeichnis: C:\Windows\System32.

hauptsächlich mit der Erstellung eines Heapsnapshots beschäftigt, wird der Debugger verwendet, weil er mehr Kontrolle über das zu untersuchende Programm bietet. So kann er z. B. das Programm pausieren und fortlaufen lassen. Diese Eigenschaften sowie die Möglichkeit, sich an ein laufendes Programm anzuhängen, sind essentiell für den in Abschnitt 2.2.1 auf Seite 14 beschriebenen Monitor.

Der Debugger hat zwei Möglichkeiten Informationen über ein anderes Programm zu bekommen. Zum einen kann er im gleichen Prozess wie das andere Programm laufen, dazu lädt der Debugger selbst den *Debuggee* (so wird das zu überwachende Programm im folgenden Text meistens genannt). Zum anderen können Debugger und Debuggee in zwei Prozessen laufen. Letztere Variante wird für diese Arbeit gewählt und dazu hängt sich der CLR-Adapter, der bei dieser Variante in einem eigenen Prozess läuft, an den Prozess des Debuggees an.

3.2. Verwendete API-Elemente

Die für den CLR-Adapter wichtigsten COM-Schnittstellen der Microsoft CLR Debug API [Mic13j] sowie ein paar Schnittstellen und Typen für die Auswertung der Metadaten (siehe auch Abschnitt 2.3.1 auf Seite 18) werden nun genauer beschrieben. Es wird nur die Funktionalität der jeweiligen Schnittstelle erläutert, welche für die vorliegende Arbeit von Belang ist.

ICLRMetaHost Die Schnittstelle `ICLRMetaHost` wird benutzt, um alle Versionen von Laufzeitumgebungen aufzuzählen, die in einem gegebenen Prozess laufen. Diese werden repräsentiert durch die Schnittstelle `ICLRRuntimeInfo`. Der Prozess wird als Handle übergeben, das von der Windows API-Funktion `OpenProcess` zurückgegeben wird. `OpenProcess` benötigt nur die Prozess-Id (PID) als Parameter von außen.

ICLRRuntimeInfo Objekte, welche die Schnittstelle `ICLRRuntimeInfo` implementieren, repräsentieren eine spezielle Common Language Runtime. Hat man das gewünschte `ICLRRuntimeInfo` in der Liste aus dem `ICLRMetaHost` gefunden, so kann man mittels dessen Methoden das eigentliche Debugobjekt `ICorDebug` bekommen.

ICorDebug Die Schnittstelle `ICorDebug` repräsentiert das eigentliche Debug-Objekt und ist der Startpunkt für alle weiteren Debug-Aktionen. Es stellt Funktionen zur Verfügung, um dem Debugger zu sagen, dass der aktive Prozess zu untersuchen ist und gibt das `ICorDebugProcess`-Objekt zurück. Mit Hilfe des `ICorDebug`-Objekts werden die verwalteten Callbacks initialisiert und andere benötigte Prozess-Objekte geholt. Für diese Arbeit ist insbesondere noch ein Objekt wichtig, welches die Schnittstelle `ICorDebugProcess5` implementiert.

ICorDebugProcess Die Schnittstelle `ICorDebugProcess` repräsentiert einen Prozess, der verwalteten Code ausführt. Als Unterklasse von `ICorDebugController` stellt es auch die Funktionen zur Verfügung um das in der CLR ausgeführte Programm zu stoppen bzw. wieder fortzusetzen. Des Weiteren beinhaltet es auch die Funktion um den Debugger vom Debuggee zu trennen, was ein unproblematisches Wiederverbinden des CLR-Adapter mit dem Debuggee ermöglicht.

ICorDebugProcess5 Die Schnittstelle `ICorDebugProcess5` stellt eine Erweiterung von `ICorDebugProcess` dar und enthält Funktionen für die Untersuchung des verwalteten Heaps. Diese Schnittstelle ist neu im .NET Framework 4.5 und dadurch mit verantwortlich für die Einschränkungen an das Betriebssystem (der CLR-Adapter ist mit Windows Vista aufwärts kompatibel).

Mit Hilfe dieser Schnittstelle ist es möglich den Heapsnapshot für den CLR-Adapter zu erzeugen. Leider hat diese Schnittstelle auch einige Fehler, die noch genauer in Kapitel 4 auf Seite 31 beschrieben werden.

ICorDebugManagedCallback Die Schnittstelle `ICorDebugManagedCallback` muss implementiert und dem `ICorDebug`-Object übergeben werden um Debugger Callbacks zu verarbeiten. Wichtig für den CLR-Adapter sind momentan die Callback-Methoden `LoadModule` und `UnloadModule`, welche den Debugger darüber informieren, wenn ein Modul vom Debuggee geladen oder entladen wird. Ein für den CLR-Adapter vorteilhaftes Feature der Methoden ist, dass sie zu dem Zeitpunkt, wenn der Debugger sich an den Debuggee anhängt, vergangene Benachrichtigungen erneut aufrufen. Somit kennt der Debugger, dank dieser „Speicherung“ von Benachrichtigungen, alleine mittels des `ICorDebugManagedCallback`, immer die aktuell geladenen Module.

Des Weiteren werden für eine spätere Betrachtung der dynamischen Aspekte des Monitors auch noch die Callbacks, welche Breakpoints behandeln, von Interesse sein.

Wenn eine Callback-Methode aufgerufen wird, pausiert der Debuggee, bis er manuell wieder fortgesetzt wird. Dies bedeutet, dass jede implementierte Callback-Methode dem Programm einen Befehl zum fortlaufen geben muss, bevor sie zurückkehrt.

ICorDebugManagedCallback2 Die Schnittstelle `ICorDebugManagedCallback2` erweitert des `ICorDebugManagedCallback`. Die Schnittstelle muss nur implementiert werden, wenn .NET Framework 2.0 Programme untersucht werden sollen.

ICorDebugModule Die Schnittstelle `ICorDebugModule` repräsentiert ein Modul (siehe auch Abschnitt 2.3.1 auf Seite 19). Unter Verwendung der Metadaten kann man die in einem Modul definierten Typen sowie dessen Felder und Methoden herausbekommen.

IMetaDataImport Die Schnittstelle `IMetaDataImport` importiert die Metadaten (siehe auch Abschnitt 2.3.1 auf Seite 18) aus einem Modul. Für den CLR-Adapter sind insbesondere die Token, welche einen Typ eindeutig innerhalb eines Moduls identifizieren, von großer Bedeutung. Die Token werden vom CLR-Adapter gespeichert um Auskunft über geladene Typen geben zu können.

Des Weiteren werden auch Klasseninformationen, Feldinformationen und konkrete Namen der Typen und Felder aus den Metadaten gewonnen.

ICorDebugHeapEnum Die Schnittstelle `ICorDebugHeapEnum` stellt einen Iterator für Objekte auf dem verwalteten Heap bereit. Er wird erzeugt durch die Funktion `EnumerateHeap`, welche durch die Schnittstelle `ICorDebugProcess5` zur Verfügung gestellt wird.

Dies ist die zugrunde liegende Schnittstelle zur Erstellung des Heapsnapshots. Wie auch `ICorDebugProcess5` wurde sie erst mit dem .NET Framework 4.5 eingeführt und trägt somit zu den im oberen Teil erwähnten Einschränkungen an des Betriebssystem bei.

ICorDebugValue Die Schnittstelle `ICorDebugValue` repräsentiert einen vom Debugger zu untersuchenden Wert, welche lesenden oder schreibenden Zugriff erlaubt. Dieser Typ stellt die Basis für die weiter unten beschriebenen konkreten Debug-Typen dar.

ICorDebugObjectValue Objekte, welche die Schnittstelle `ICorDebugObjectValue` implementieren, repräsentieren ein von der CLR geladenes Objekt. Die Schnittstelle `ICorDebugObjectValue` ist eine logische Erweiterung von `ICorDebugValue`.

Bei der Iteration über den Heap erhält man unter anderem die Adressen der Objekte als Rückgabewert. Mit Hilfe der Funktion `GetObjectW` aus der Schnittstelle `ICorDebugProcess5`, der die Adresse übergeben wird, bekommt man dann ein Objekt des hier beschrieben Typs zurück.

Nachdem man die Objekte, welche die Schnittstelle `ICorDebugObjectValue` implementieren, bekommen hat, ist man in der Lage dessen Felder und Methoden zu untersuchen.

ICorDebugReferenceValue Als eine Erweiterung von `ICorDebugValue` repräsentiert `ICorDebugReferenceValue` eine Referenz zu einem anderen Objekt und stellt Funktionen bereit, um an das eigentliche Objekt zu gelangen oder zu erkennen, ob es sich um eine NULL-Referenz handelt. Die Schnittstelle wird in der vorliegenden Arbeit hauptsächlich bei der Untersuchung der Feldwerte benötigt, um Assoziationen erstellen zu können.

ICorDebugGenericValue Die Schnittstelle `ICorDebugGenericValue` ist eine Erweiterung von `ICorDebugValue` um *Get*- und *Set*-Methoden. Objekte des Typs werden vom CLR-Adapter benutzt, um die Werte der sogenannten primitiven Typen zu erhalten.

ICorDebugStringValue Die Schnittstelle `ICorDebugStringValue` erweitert die beiden Schnittstellen `ICorDebugValue` und `ICorDebugHeapValue` um Funktionen, welche die Länge eines Strings sowie dessen eigentlichen Inhalt zurückliefert.

ICorDebugArrayValue Ein Objekt des Typs `ICorDebugArrayValue` repräsentiert als eine Erweiterung der Schnittstellen `ICorDebugValue` und `ICorDebugHeapValue` eine ein- oder mehrdimensionale Reihung. Mit Hilfe des Typs kann man die Anzahl der in der Reihung vorhandenen Elemente sowie dessen Inhalt herausbekommen.

ICorDebugClass Diese Schnittstelle `ICorDebugClass` bildet einen nicht instantiierten generischen Typen ab. Bei nicht generischen Typen stellt die Schnittstelle einfach nur den Typen dar. Sie bietet eine Funktion an, um an das Modul zu gelangen, in welchem die repräsentierte Klasse definiert ist.

Mittels der Objekte, welche die Schnittstelle `ICorDebugClass` implementieren, kann man an die statischen Felder der Klasse gelangen.

ICorDebugType Die Schnittstelle `ICorDebugType` repräsentiert einen instantiierten generischen Typen. Bei nicht generischen Typen stellt die Schnittstelle einfach nur den Typen dar.

Mittels der Objekte, welche die Schnittstelle `ICorDebugType` implementieren, kann man an die statischen Felder der Klasse gelangen.

CorElementType Die Aufzählung `CorElementType` repräsentiert den konkreten CLR-Typ von geladenen Typen. Einige der oben vorgestellten Schnittstellen bieten Funktionen an, um an diesen Typen zu gelangen.

4. Aufbau und Implementierung des CLR-Adapters

Dieses Kapitel beschreibt das Design des CLR-Adapters für das USE Monitor-Plug-in (siehe Abschnitt 2.2.1 auf Seite 14). Es wird auf Entwurfsentscheidungen eingegangen und fehlgeschlagene Ansätze sowie Alternativen werden diskutiert.

Das Gesamtprojekt *CLR-Adapter* besteht aus Java- und C++-Code. Der Hauptteil des CLR-Adapters ist in C++ geschrieben und wird in Abschnitt 4.2 auf Seite 33 detailliert beschrieben. Der in Java geschriebene Teil ist hauptsächlich für die Kommunikation zwischen Monitor und dem C++-Teil (siehe Abschnitt 4.3 auf Seite 52) verantwortlich. Bevor die Details der Implementierung ausführlich erklärt werden, stellt der folgende Abschnitt den Aufbau des CLR-Adapters mit all seinen Komponenten vor.

Der Aufbau der physikalischen Struktur des CLR-Adapters wird im Anhang A dargestellt. Dort wird auch erläutert, was nötig ist, um den CLR-Adapter zu kompilieren und wie eine Verbindung zu einem von der CLR gehosteten .NET Programm hergestellt werden kann.

In diesem Kapitel wird nicht weiter auf die Ausführung von USE in Kombination mit der in C++ geschrieben Bibliothek des CLR-Adapters eingegangen, weil dieses Thema ausführlich bei dem Aufbau der Testumgebung behandelt wird (siehe Abschnitt 5.1.1 auf Seite 57).

Bei allen Klassen des CLR-Adapters auf der C++-Seite sind die einfachen Eigenschaften öffentlich, um auf Getter und Setter verzichten zu können. Dies gilt allerdings nur solange sie keine eigene Logik implementieren.

4.1. Komponenten des CLR-Adapters

Um ein von der Microsoft CLR gehostetes .NET Programm mit USE und dem USE-Monitor-Plug-in analysieren zu können, sind mehrere Komponenten notwendig. Auf der Java-Seite muss der Adapter für das Monitor-Plug-in erstellt werden (siehe Abschnitt 2.2.1 auf Seite 14) und auf der C++-Seite muss eine Bibliothek entworfen werden, welche die benötigten Schnittstellen der CLR Debug API (siehe Kapitel 3 auf Seite 25) implementiert. Für die Kommunikation zwischen beiden Technologien wird JNI (siehe Abschnitt 2.4 auf Seite 23) verwendet.

Damit die implementierten Schnittstellen der CLR Debug API leichter getestet werden können, wird eine native Anwendung implementiert, welche sich direkt mit einem in der CLR laufenden .NET Programm verbinden kann.

Die für diesen Ansatz zur Analyse eines verwalteten .NET Programms verwendeten Komponenten können Abbildung 4.1 entnommen werden.

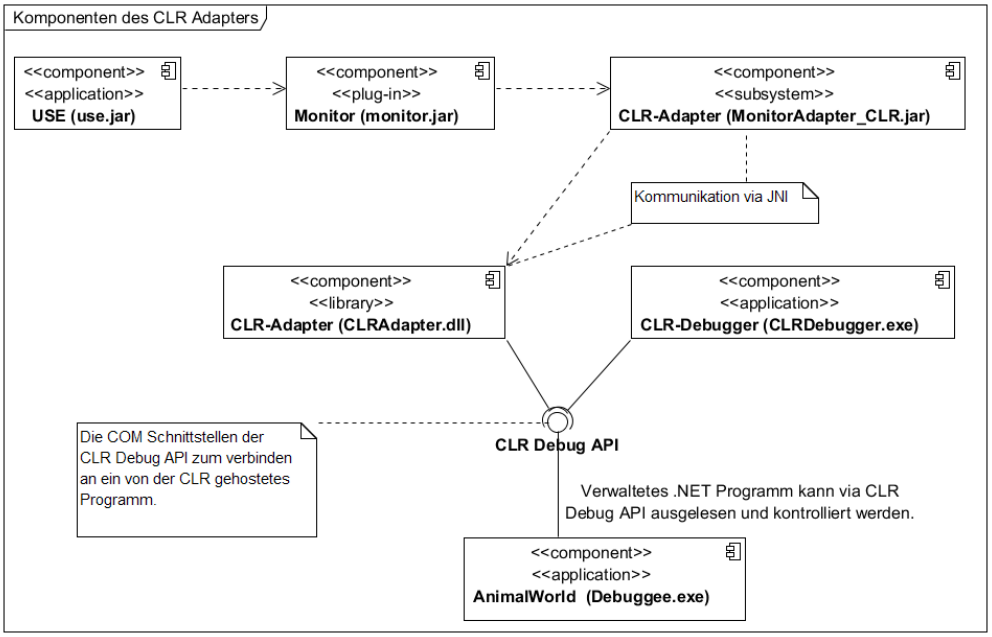

Abbildung 4.1.: Benötigte Komponenten um ein .NET Programm mit USE zu analysieren

Im Folgenden werden die Komponenten aus Abbildung 4.1 sowie dessen Verantwortlichkeiten für den in dieser Arbeit präsentierten CLR-Monitoring-Ansatz detailliert erläutert.

- **USE (Java)**: Ein für diese Arbeit verwendetes Werkzeug mit dem UML- und OCL-Modelle validiert werden können (siehe Abschnitt 2.2 auf Seite 13). Das Programm überprüft den vom Monitor erzeugten Snapshot anhand einer UML- und OCL-Spezifikation.
- **Monitor (Java)**: Der Monitor ist verantwortlich für die Verbindung mit dem .NET Programm, dessen Laufzeitverhalten untersucht werden soll. Des Weiteren lässt er sich nach der Verbindung einen Snapshot vom Debuggee mittels des Adapters liefern (siehe Abschnitt 2.2.1 auf Seite 14) und verwaltet die vom Debuggee geladenen Typen sowie dessen Vererbungshierarchien. Der Monitor ist bereits vorhanden und wird wie USE für diese Arbeit verwendet.
- **CLR-Adapter (Java)**: Der Adapter für den Monitor. Eine Java-Version ist bereits vorhanden (siehe Abschnitt 2.2.1 auf Seite 14) und das CLR Pendant

wird in dieser Arbeit entwickelt.

Der CLR-Adapter ist verantwortlich für die Erstellung eines Snapshots und implementiert dafür die Schnittstelle `VMAdapter` (siehe Abbildung 2.4 auf Seite 16). Er ist auch für die Umwandlung der Ausgelesenen Daten in von USE verständliche OCL-Datentypen verantwortlich.

Der CLR-Adapter auf der Java-Seite definiert die JNI-Methoden, welche von seinem C++ Pendant implementiert werden. Damit wird die Kommunikation zwischen Java und C++ umgesetzt. Alle Methoden zur Generierung des Heap-Snapshots sind als JNI-Methode definiert. Eine detaillierte Beschreibung ist Abschnitt 2.2.1 auf Seite 14 zu entnehmen.

- **CLR-Adapter (C++)**: Der CLR-Adapter auf der C++-Seite ist eine native Bibliothek und wird im Rahmen der vorliegenden Arbeit entwickelt. Sie implementiert die benötigten CLR Debug API Schnittstellen um einen Snapshot zu erstellen sowie Typinformationen zu erhalten. Details der Implementierung und verwendete Optimierungsstrategien sind in Abschnitt 4.2 beschrieben.

  Um mit dem CLR-Adapter auf der Java-Seite kommunizieren zu können, implementiert dieser Adapter die dort definierten JNI-Schnittstellen. Mit Hilfe dieser Methoden kann der Debuggee vom Monitor verwaltet (gestoppt, pausiert, getrennt) werden.

- **CLR-Debugger (C++)**: Der CLR-Debugger ist eine Windowsanwendung, welche die gleichen CLR Debug API Schnittstellen und Optimierungsstrategien wie der CLR-Adapter implementiert. Sie ist für Testzwecke erstellt und kann sich mittels der Prozess ID des Debuggees mit ihm verbinden.

- **AnimalWorld (C#)**: Ein verwaltetes .NET Testprogramm, geschrieben in C#. Es ist nach dem Entwurf aus Abschnitt 2.1.1 auf Seite 6 implementiert und wird für die Auswertungen Abschnitt 5.1.1 auf Seite 57 benutzt um den CLR-Adapter zu testen.

Der Aufbau der einzelnen Komponenten wird nun in den folgenden Abschnitten detailliert erläutert. Da der CLR-Adapter und der CLR-Debugger in Bezug auf die implementierten CLR Debug API Schnittstellen identisch sind, werden sie im weiteren Verlauf der Arbeit unter dem Begriff *Debugger* beide angesprochen.

4.2. Aufbau des CLR-Adapters auf der .NET-Seite

Der CLR-Adapter setzt sich aus Klassen für zwei unterschiedliche Zwecke zusammen. Auf der einen Seite werden Typen benötigt, die ausgelesene Daten für die Weiterleitung an USE repräsentieren. Auf der anderen Seite stehen die Klassen, die das eigentliche Programm – oder die Logik – enthalten.

Zunächst werden in Abschnitt 4.2.1 die Metadaten-Klassen, die nichts weiter sind als Datenhaltungsklassen für Informationen über den Debuggee (ausgelesen via CLR Debug API), beschrieben. Anschließend wird in Abschnitt 4.2.2 auf Seite 38 der eigentliche Kern des CLR-Adapters sowie dessen Funktionsweise und Konfigurationsmöglichkeiten detailliert erläutert.

Die Java-Seite des CLR-Adapters wird näher in Abschnitt 4.3 auf Seite 52 behandelt. Die Beschreibung der Schnittstellen der CLR Debug API kann Kapitel 3 auf Seite 25 entnommen werden.

4.2.1. Klassen für die Metadaten

Die Metadaten des CLR-Adapters repräsentieren Informationen, die vom Adapter über den Debuggee gesammelt werden, um sie zu einem bestimmten Zeitpunkt an USE zu übergeben. Sie können je nach Bedarf und Einstellung vom Adapter in geeigneten Datenstrukturen intern gespeichert oder bei jedem Abruf neu geladen werden. Jede der CLR-Metadaten-Klassen, mit Ausnahme der konkreten Felder, korrespondiert zu einer Klasse aus der Metadaten-Ebene für die Virtuelle Maschine des Adapters (siehe Abschnitt 2.2.1 auf Seite 14).

Eine Übersicht der einzelnen Klassen sowie ihre Vererbungshierarchie ist Abbildung 4.2 auf der nächsten Seite zu entnehmen. In der nun folgenden, detaillierten Beschreibung jedes Typs werden Schnittstellen der CLR Debug API verwendet, welche in Kapitel 3 auf Seite 25 ausführlich erklärt sind.

CLRInfoBase Der Typ `CLRInfoBase` stellt mit dem Namen und dem innerhalb eines Moduls eindeutigen Metadaten-Token minimale Informationen bereit. Diese Klasse repräsentiert den Basistypen für alle Metadaten-Klassen.

Die virtuelle Methode `CLRInfoBase::Print()` gibt den Namen des Typs auf der Konsole aus. Sie wird ggf. von den abgeleiteten Klassen um weitere Attribute erweitert.

CLRType Die Klasse `CLRType` repräsentiert einen Typen, der vom Debuggee zur Laufzeit geladen ist. Die eindeutige ID dieses Typen ist das Metadaten-Token vom Typ `mdTypeDef`. Diese Eindeutigkeit gilt nur innerhalb eines Moduls. Der Typ stellt die Vererbungshierarchie durch die Attribute `baseClass` und `subClasses` öffentlich zur Verfügung. Diese Eigenschaften beinhalten Zeiger auf die entsprechenden `CLRType`-Objekte. Des Weiteren werden Informationen über die Felder des Typen sowie ein Zeiger auf das Modul, das den Typen beinhaltet, bereitgestellt.

Mit Hilfe des Attributs `typeAttr` vom .NET Typ `CorTypeAttr` werden detaillierte Typinformationen herausgefunden. Dabei handelt es sich unter Anderem um die Folgenden: Zugriffsmodifizierer (public, protected und private); Information darüber,

Abbildung 4.2.: Klassen für die Metadaten des CLR-Adapters (erstellt mit dem Visual Studio von Microsoft, richtet sich nicht streng nach der UML)

ob es sich um eine Klasse, eine abstrakte Klasse oder eine Schnittstelle handelt; Aufschluss darüber, ob der Typ in anderen Typen geschachtelt ist. Die Eigenschaft `typeAttr` wird vom `TypeInfoHelper` (siehe Abschnitt 4.2.2 auf Seite 40) ausgewertet.

Die Datenstruktur `instances` dient als Speicher für die geladenen Instanzen eines Typs. An dieser Stelle wird allerdings nur der eindeutige Schlüssel eines Objekts – die Speicheradresse – gespeichert. Das Auslesen und Verwalten der Objekte übernimmt der `ObjectInfoHelper` (siehe Abschnitt 4.2.2 auf Seite 42).

Ein gewünschtes Feld in Form eines `CLRMetaField`-Objekts wird mit Hilfe der überladenen Methode `CLRType::GetField()` zurückgegeben. Als Parameter wird entweder der Name oder das Token des Feldes benötigt.

CLRMetaField Der Typ `CLRMetaField` ist eine Erweiterung von `CLRInfoBase`, der zusätzlich noch das eindeutige Metadaten-Token des Feldes im Attribut `fieldDef` sowie mehr Informationen über das Feld – z. B. ob es sich um ein statisches Feld handelt – in der Eigenschaft `fieldAttr` bereitstellt.

Der Eigenschaft `mdTypeDef` wird das Metadaten-Token übergeben, das die Klasse identifiziert, in der das Feld deklariert ist.

Ein Objekt dieses Typs repräsentiert eine Feldbeschreibung. Instanzen von `CLRType` speichern ihre Felder vom Typ `CLRMetaField` in einer dafür geeigneten Datenstruktur. Genau wie die Typinformationen werden auch die Felder vom `TypeInfoHelper` aus den Metadaten extrahiert. Damit sind Klassen und Eigenschaften ausreichend genug beschrieben, um ihre Instanzen für einen Snapshot auslesen zu können.

CLRObject Die Klasse `CLRObject` stellt eine Instanz eines geladenen Typen dar. Eindeutig identifiziert wird ein `CLRObject`-Objekt an seiner Speicheradresse. Das Attribut `debugValue` zeigt auf die CLR Repräsentation des Objekts, welche die Schnittstelle `ICorDebugValue` implementiert.

Die Eigenschaft `fields` ist eine Datenstruktur, dessen Elemente die konkreten Felder des Objekts sind. Diese Felder müssen von der Klasse `CLRFieldBase` abgeleitet sein.

Objekte des Typs `CLRObject` werden vom `ObjectInfoHelper` verwaltet (siehe Abschnitt 4.2.2 auf Seite 42).

CLRFieldBase Der Typ `CLRFieldBase` ist die Basisklasse für alle Felder einer Instanz von `CLRObject`. Um das Feld einem `CLRMetaField` zuordnen zu können, wird auch hier das Metadaten-Token gespeichert.

Um an den Inhalt und den genauen Datentypen des Feldes zu gelangen, werden in den Attributen `debugValue` und `corType` das eigentliche CLR-Objekt vom Typ

`ICorDebugValue` sowie der Typ als `CorElementType` gespeichert (siehe Kapitel 3 auf Seite 25).

Das Attribut `info` dient ausschließlich zu Testzwecken. Ob ein Feld einen eigentlichen Wert hat oder einen Nullreferenz repräsentiert, wird in der Eigenschaft `isNull` gespeichert.

Da es in C++ kein Sprachkonstrukt für das aus Java bekannte *instanceof* (oder *typeof* in C#) gibt, wird hier der spezielle Feldtyp anhand der Enumeration `FieldType` in der Eigenschaft `type` festgehalten.

Für die Verwaltung der Felder und ihrer Werte ist ebenfalls der `ObjectInfoHelper` verantwortlich, da sie sich einem Objekt zuordnen lassen.

CLRFieldValue Die Klasse `CLRFieldValue` repräsentiert ein Feld, dessen Wert dem eines primitiven Typen (int, bool, etc.) oder dem eines Strings entspricht.

Um auf die eigentlichen Daten des Feldes zugreifen zu können, wird im Attribut `genericDebugValue` ein Zeiger auf die entsprechende Instanz vom Typ `ICorDebugGenericValue` der CLR (siehe Abschnitt 3.2 auf Seite 28) gespeichert. Eine einfache String-Repräsentation des Wertes ist der Eigenschaft `valueAsString` zu entnehmen.

CLRFieldReference Der Typ `CLRFieldReference` ist eine Erweiterung der Klasse `CLRFieldBase` um eine Darstellung einer Referenz zu einer anderen, geladenen Instanz vom Typ `CLRObject`.

Um das Objekt eindeutig zu identifizieren und ggf. in der Liste der geladenen Objekte wiederfinden zu können, wird die Eigenschaft `address` verwendet, die die Speicheradresse der Instanz beinhaltet. Des Weiteren wird der Name des Typen im Attribut `typeName` festgehalten.

CLRFieldList Der Typ `CLRFieldList` repräsentiert eine Spezialisierung der Klasse `CLRFieldBase`. Die Klasse stellt Felder dar, die auf Datenstrukturen zeigen. Dieser Typ ist zur Zeit noch als experimentell einzustufen und funktioniert ansatzweise mit Reihungen, dessen Elemente vom Typ `CLRObject` sein müssen. Die Rückgabewerte der CLR Debug API scheinen hier nicht immer zu stimmen. Auf mögliche Ursachen dafür wird in Abschnitt 4.2.3 auf Seite 50 eingegangen.

4.2.2. Programmaufbau des Adapters

In diesem Abschnitt werden die einzelnen Klassen des CLR-Adapters vorgestellt, ihre Verantwortlichkeiten erläutert und die Funktionsweisen detailliert beschrieben. Wertvolle Informationen über die Arbeitsweise des Debuggers wurden neben der MSDN Dokumentation [Mic13j] auch [Sta13; Shu05; Mic13b; Pel02] entnommen und fließen in Designentscheidungen mit ein. Verwendete Entwurfsmuster richten sich nach [Gam+09].

Eine Übersicht über die Komponenten des CLR-Adapters, mit Ausnahme der in Abschnitt 4.2.1 auf Seite 34 erläuterten Metadaten, liefert Abbildung 4.3 auf der nächsten Seite. Auf ihr sind im oberen Teil die Klassen, in der Mitte die Strukturen und Aufzählungsdatentypen und im unteren Teil verwendete Typ-Definitionen abzulesen. Die einzelnen Komponenten der Abbildung sowie ihre Verbindungen untereinander werden im Folgenden näher betrachtet und erläutert.

Abbildung 4.3.: Komponenten des CLR-Adapters (erstellt mit dem Visual Studio von Microsoft, richtet sich nicht streng nach der UML)

DefaultCallback Die Klasse `DefaultCallback` stellt dem Adapter eine Standardimplementierung der beiden CLR-Schnittstellen `ICorDebugManagedCallback` und `ICorDebugManagedCallback2` bereit. Standard bedeutet in diesem Fall, dass jede Callback-Methode den Wert `E_NOTIMPL` zurückliefert. Dieser Wert lässt den Debuggee nach dem Aufruf einer Callback-Methode automatisch weiterlaufen (siehe Abschnitt 3.2 auf Seite 27).

Jede Spezialisierung von `DefaultCallback` muss dadurch nur noch die Methoden überschreiben, die für ihren Einsatzzweck von Belang sind.

CLRDebugCallback Die Klasse `CLRDebugCallback` ist von `DefaultCallback` abgeleitet und repräsentiert die Implementierung der beiden CLR-Schnittstellen `ICorDebugManagedCallback` und `ICorDebugManagedCallback2` für die Projekte *CLRAdapter* und *CLRDebugger*.

Die Klasse implementiert die Methoden `LoadModule`, `UnloadModule` und `CreateAppDomain`. Die beiden Erstgenannten benachrichtigen den `TypeInfoHelper` genau dann, wenn ein Modul geladen oder entladen wird. Die letztere Methode dient zu Testzwecken.

Um mit dem `TypeInfoHelper` kommunizieren zu können, wird das Entwurfsmuster *Dependency Injection* benutzt. Konkret wird bei diesem Ansatz eine Instanz der Klasse `TypeInfoHelper` dem Konstruktor von `CLRDebugCallback` übergeben – und somit bekanntgemacht. Dem `TypeInfoHelper` stehen dadurch immer die aktuell geladenen Module zur Verfügung und er kann sich um deren Verwaltung kümmern.

CLRDebugCore Ein Objekt der Klasse `CLRDebugCore` ist für die Verbindung zum Debuggee verantwortlich, initialisiert die Debug-Prozesse und stellt sie öffentlich zur Verfügung. Damit bietet es die Schnittstellen für die Kontrolle des Debuggees sowie die Untersuchung des Heaps an. Diese Funktionalität macht `CLRDebugCore` zum Herzstück des CLR-Adapters.

Um sicherzustellen, dass genau ein Exemplar der Klasse instantiiert wird und ein globaler Zugriffspunkt existiert, ist `CLRDebugCore` nach dem Entwurfsmuster *Singleton* entworfen.

Nachdem ein Objekt des Typs `CLRDebugCore` erstellt wurde, muss es initialisiert werden bevor, es verwendet werden kann. Damit ist der erste Funktionsaufruf des CLR-Adapters üblicherweise der in Listing 4.1 dargestellte.

Listing 4.1: Initialisierung des CLR-Adapters

```
CLRDebugCore::theInstance()->
        InitializeProcessesByPid(pid,
                new CLRDebugCallback(typeInfo));
```

Der Methode `InitializeProcessesByPid` wird die Prozess ID des Debuggees sowie ein Callback Objekt übergeben. Für die Instantiierung des `CLRDebugCallback` Objekts ist eine zuvor erstellte Instanz vom `TypeInfoHelper` notwendig. Nach dem Methodenaufruf werden folgende Schritte zur Initialisierung durchgeführt (siehe Kapitel 3 auf Seite 25).

1. Mit Hilfe der WINAPI-Funktion `OpenProcess`, welche die PID als Parameter bekommt, das Prozess-Handle initialisieren.
2. Initialisierung eines `ICLRMetaHost` durch die Funktion `CLRCreateInstance`.
3. Über die in dem Prozess geladenen CLR-Laufzeiten iterieren und für die erste passende CLR das `ICLRRuntimeInfo` Objekt initialisieren. Als passend gelten nur Laufzeiten der .NET Version 4.
4. Das `ICorDebug` Objekt aus `ICLRRuntimeInfo` laden und initialisieren.
5. Das übergebene DefaultCallback an die `ICorDebug`-Instanz mittels der Methode `SetManagedHandler` weiterleiten.
6. Die öffentlichen Schnittstellen `ICorDebugProcess` und `ICorDebugProcess5` initialisieren.

Nach einer erfolgreichen Initialisierung können die öffentlich zugänglichen Objekte vom gesamten Programm aus verwendet werden.

Bevor der Adapter beendet wird, muss die Methode `Release` aufgerufen werden. Dort werden alle Ressourcen geschlossen, der Speicher freigegeben und der Debugger wird korrekt vom Debuggee getrennt.

TypeInfoHelper Eine Instanz der Klasse `TypeInfoHelper` ist verantwortlich für die Verwaltung der Metadaten-Klassen `CLRType` und `CLRMetaField`. Die Vererbungshierarchie der geladenen Typen, detaillierte Informationen über einen gewünschten Typen sowie Statusinformationen (z. B. die Anzahl geladener Typen und Module) werden von dieser Instanz zur Verfügung gestellt. Üblicherweise wird pro Anwendung genau ein Objekt vom `TypeInfoHelper` erstellt.

Damit der `TypeInfoHelper` die Verwaltung der Typen beginnen kann, müssen ihm Module mittels der Methode `AddModule` hinzugefügt werden. Der Aufruf der Methode erfolgt durch das `CLRDebugCallback` beim Start des Debuggers für alle bisher geladenen Module des Debuggees sowie zur Laufzeit bei aktuellen Änderungen. Beim Entladen eines Moduls wird auf die gleiche Art und Weise die Methode `RemoveModule` aufgerufen. Wenn beide Methoden ausschließlich vom Callback aus aufgerufen werden, ist gewährleistet, dass der `TypeInfoHelper` zu jedem Zeitpunkt die aktuell geladenen Module kennt.

Jeder Aufruf der Methode `AddModule` stellt einen neuen Zustand des `TypeInfoHelper` her. Dieses passiert indem der Reihe nach die folgenden Schritte ausgeführt werden.

1. Mit Hilfe der Metadaten-Funktion `IMetaDataImport::EnumTypeDefs` wird über alle in dem Modul definierten Typen iteriert. Dabei wird für jeden einzelnen Typen ein Objekt der Klasse `CLRType` erstellt und in einer privaten Datenstruktur als Schlüssel-Wert-Paar (Schlüssel = Token, Wert = Objekt) gespeichert.

   Des Weiteren werden während der Iteration für jede `CLRType` Instanz die Feldinformationen erstellt. Dazu wird die Funktion `IMetaDataImport::EnumFields` benötigt, mit der über die Felder eines Typs iteriert wird. Während des Durchlaufs wird für jedes Feld ein `CLRMetaField`-Objekt erstellt und einer öffentlichen Datenstruktur von `CLRType` hinzugefügt.

2. Nun wird noch einmal über alle Typen mit der Methode `IMetaDataImport::EnumTypeDefs` iteriert. Dabei wird die Vererbungshierarchie aufgebaut, indem man sich über die den Metadaten bekannte Basisklasse immer weiter nach oben in der Hierarchie jedes einzelnen Typen hangelt. Der zweite Schritt ist notwendig, damit jetzt auf die im ersten Schritt erstellten Typen verwiesen werden kann.

3. Speicherung eines Zeigers auf das gerade ausgelesene Modul in einer privaten Datenstruktur der `TypeInfoHelper`-Instanz.

Bei der – wie vorher beschrieben – aufgebauten Vererbungshierarchie ist zu beachten, dass jeder Typ nur seine eigenen Felder beinhaltet. Dies bedeutet, dass ein Feld einer abgeleiteten Klasse nur die in ihr definierten Felder und nicht die der Basisklasse kennt. Deshalb müssen Felder rekursiv über die Basisklassen gesucht werden, was sich die Methode `CLRType::GetField` zu Nutzen macht.

Die überladene Methode `TypeInfoHelper::GetType` gibt den gewünschten `CLRType` zurück und benötigt als Parameter entweder den Namen des Typen oder das Metadaten-Token. Die Suche nach dem Token ist effizienter, da das Token der Schlüssel der Einträge in der Datenstruktur ist.

Der Methode `TypeInfoHelper::GetTypeInfo` wird das `CorTypeAttr` eines `CLRType` übergeben und sie gibt zurück, ob es sich bei dem Typen um eine Klasse, eine abstrakte Klasse oder eine Schnittstelle handelt. Der Rückgabewert wir repräsentiert durch den Aufzählungsdatentypn `TypeInfo`.

Um Informationen darüber zu erhalten, ob der `TypeInfoHelper` geladene Typen enthält, stehen die beiden Methoden `HasAny` und `IsInitialized` zur Verfügung. Die Erstgenannte liefert einen wahren Wert zurück, wenn der `TypeInfoHelper` mindestens einen Typen geladen hat. In der zweiten Methode wird die Anzahl der geladenen Module mit einer in den Einstellungen (siehe Abschnitt 4.2.2 auf Seite 45) zu definierenden Mindestanzahl von Modulen verglichen und ein wahrer Wert genau dann zurückgegeben, wenn mindestens das Minimum erreicht ist.

Bevor eine korrekte Trennung des Debuggers vom Debuggee erfolgen kann, ist ein Aufruf der Methode `TypeInfoHelper::Detach` notwendig. Damit wird sichergestellt, dass die Datenstrukturen mit den gespeicherten Modulen und Typen geleert werden und somit bei einer neuen Verbindung keine alten Daten mehr vorhanden sind.

ObjectInfoHelper Eine Instanz der Klasse `ObjectInfoHelper` ist hauptsächlich für die Erstellung und Verwaltung der `CLRObject`-Objekte der vom `TypeInfoHelper` geladenen Typen verantwortlich. Da der `ObjectInfoHelper` an einer Stelle Zugriff auf die geladenen Typen benötigt, muss er den `TypeInfoHelper` kennen. Diese Abhängigkeit wird unter Zuhilfenahme des Entwurfsmusters *Dependency Injection* aufgelöst.

Aufgrund der in Abschnitt 2.2.1 auf Seite 14 beschriebenen Arbeitsabläufe des Monitors wird bei der Implementierung der Klasse `ObjectInfoHelper` davon ausgegangen, dass als erstes alle Instanzen eines bestimmten Typs zurückgeliefert werden sollen. Daraus folgt, dass zur Laufzeit zunächst die Methode `ObjectInfoHelper::GetInstances` – als eine Art Initialisierung – aufgerufen werden muss, bevor andere Daten zurückgeliefert werden können. Dies ist entscheidend für die Umsetzung der in den Einstellungen (siehe Abschnitt 4.2.2 auf Seite 45) konfigurierbaren Speicherstrategien, die im Folgenden näher erläutert werden.

Um Objekte eines bestimmten Typs im Heap zu finden, muss über alle geladenen Instanzen iteriert werden und für jede einzelne Instanz ein Typ-Vergleich stattfinden. Dies ist bei großen Projekten – insbesondere mit vielen Klassen, wovon nur ein kleiner Teil für Monitoring-Durchlauf von Interesse ist – eine teure Operation. Zu Optimierungszwecken stehen dem CLR-Adapter darum drei unterschiedliche Strategien zur Verfügung.

1. Keine Optimierung. Das bedeutet, es werden keine Objekte des Heaps vom `ObjectInfoHelper` gespeichert und bei jedem Aufruf wird erneut auf den Speicher zugegriffen.

2. Speicherung zurückgegebener Objekte in einer geeigneten Datenstruktur bei jedem Aufruf von `ObjectInfoHelper::GetInstances`. Bei erneuter Verwendung der Objekte, z. B. zur Erstellung eines neuen Snapshots oder um die Felder zu laden, werden sie in der Datenstruktur gesucht und zurückgegeben.

3. Speicherung aller Instanzen von allen den Anwendenden interessierenden Typen beim ersten Aufruf der Methode `ObjectInfoHelper::GetInstances`. Dazu müssen alle Typen, die von USE geladen werden sollen, noch einmal in den CLRAdapter-Einstellungen festgehalten werden. Diese Einstellung setzt voraus, dass Variante zwei ebenfalls gewählt ist.

Eine detaillierte Erläuterung der Einstellungen kann der Lesende dem Abschnitt 4.2.2 auf Seite 45 entnehmen. Ob – und falls ja, wie – sich die Optimierungen auf die Performance des CLR-Adapters auswirken, ist Teil der Auswertung (siehe Kapitel 5 auf Seite 57).

Je mehr Informationen der Objekte aus dem Heap vom CLR-Adapter zwischengespeichert werden, umso komplexer wird die Implementierung. Es sind zur Zeit nur einfache Verfahren implementiert, die nicht die volle Funktionalität des nicht optimierten Adapters bieten und auch nicht komplett fehlerfrei arbeiten. Um zu erkennen, wie sich verschiedene Verfahren auf die Performance auswirken können, reichen sie allerdings aus. Keine der Optimierungen geht zu Zeit auf Veränderungen im Speicher, z. B. durch den Garbage Collector der CLR, ein.

Nun wird darauf eingegangen, wie genau die Untersuchung des Heaps vonstatten geht und wie Objekte des Typs `CLRObject` instantiiert und ggf. gespeichert werden. Wie bereits erwähnt, werden zuerst alle Instanzen eines bestimmten Typs gesucht, weshalb damit begonnen wird.

Nachdem die Methode `ObjectInfoHelper::GetInstances` für einen bestimmten `CLRType` aufgerufen wurde, folgt eine Untersuchung des Heaps auf Vorkommen dieses Typs. Dieser Arbeitsschritt wird mit Hilfe der in .NET 4.5 neu eingeführten Schnittstelle `ICorDebugProcess5` erledigt. Bevor mit der Funktion `EnumerateHeap` über den Heap iteriert werden kann, muss sichergestellt sein, dass der Heap sich in einem validen Zustand befindet. Dies lässt sich mit der Funktion `GetGCHeapInformation` überprüfen. Ein Kriterium für einen validen Heap ist, dass der Debuggee pausiert. Ist die Validität des Heaps gegeben, so kann mit der Iteration begonnen werden. Dabei wird jede Instanz durch ein `COR_HEAPOBJECT` repräsentiert, das unter anderem die Speicheradresse und den Typen in Form einer Type-ID des Objekts enthält. Nur Instanzen, die einer Klasse entsprechen, sind im weiteren Verlauf von Interesse. Stimmen nun die Metadaten-Token des Typen der aktuellen Instanz und des gewünschten Typen überein und beide stammen aus dem selben Modul, so wird ein `CLRObject` erstellt. Alle dafür benötigten Informationen werden einem `ICorDebugObjectValue`-Objekt entnommen, welches für eine Speicheradresse von der Funktion `GetObjectW` initialisiert wird. Die Speicheradresse des gerade erstellten Objekts wird in `CLRType::instances` eingetragen. Je nach Konfiguration erfolgt auch ein Eintrag des Objekts in einer privaten Datenstruktur des `ObjectInfoHelper` als Schlüssel-Wert-Paar mit der Speicheradresse als Schlüssel.

Ist der Adapter so eingestellt, dass er beim ersten Aufruf von `ObjectInfoHelper::GetInstances` alle Instanzen der zu analysierenden Typen speichert, so entstehen kleine Abweichungen von der obigen Vorgehensweise. Bei dieser Variante wird genau einmal über den Heap iteriert, wohingegen sonst für jeden gewünschten Typen iteriert wird. Bei einer einmaligen Iteration ist ein Typ-Vergleich wie oben allerdings nicht möglich, weshalb Typ-Namen verglichen werden. Darum ist es bei dieser Optimierungsstrategie notwendig, dass alle Typen, die von USE geladen werden sollen, in den Einstellungen vorhanden sind.

Durch die Methode `GetCLRObject` stellt der `ObjectInfoHelper` die Möglichkeit, einzelne `CLRObject`-Instanzen auf Basis ihrer Speicheradresse zurückzubekommen, zur Verfügung. Dazu wird, je nach Konfiguration, in der privaten Datenstruktur die entsprechende Instanz gesucht oder durch die CLR-Funktion `ICorDebugProcess5::GetObjectW` ein `ICorDebugObjectValue` erstellt, dessen Informationen ausreichen, um eine Instanz von `GetCLRObject` zu erstellen.

Einzelne Felder und dessen aktuelle Werte lassen sich mit Hilfe der Methode `ObjectInfoHelper::GetField` zurückgeben. Ihr werden als Parameter der `CLRType`, die Speicheradresse des Objekts sowie das Metadaten-Token für das gewünschte Feld übergeben. Die eigentliche Arbeit erledigt dann aber der als nächstes beschriebene `FieldValueHelper`.

Ähnlich wie beim `TypeInfoHelper` muss auch für den `ObjectInfoHelper` die Methode `Detach` aufgerufen werden, bevor der Debugger vom Debuggee korrekt getrennt werden kann. In diesem Fall muss dies nur dann passieren, wenn Objekte gespeichert worden sind. Ruft man `Detach` jedoch immer auf, führt dies nicht zu einem Fehlverhalten des Adapters.

FieldValueHelper Die Hilfsklasse `FieldValueHelper` enthält nur eine einzige öffentliche Methode. Sie heißt `FieldValueHelper::GetField`, ist statisch und ihr werden der `CLRType`, das `CLRObject` und das Metadaten-Token für das gewünschte Feld als Parameter übergeben. Ihr Rückgabewert ist eine Spezialisierung der Klasse `CLRFieldBase`.

Von den der Methode übergebenen Parameter bis zu einem speziellen `CLRFieldBase`-Objekt sind einige Arbeitsgänge notwendig, die im Folgenden schrittweise beschrieben sind.

1. Bestimmung des `CLRMetaField` aus dem als Parameter übergebenen Feld-Token mithilfe der Methode `CLRType::GetField`.

2. Bestimmung des `ICorDebugObjectValue` aus dem im `CLRObject` gespeicherten `ICorDebugValue`.

3. Erstellung der CLR-Klassenrepräsentation in Form eines `ICorDebugClass`-Objekts mit Hilfe des im `CLRType` enthaltenen Moduls. Dabei ist wichtig, dass die Klasse genau derjenigen entspricht, in welcher das Feld deklariert ist und nicht einer anderen aus der Vererbungshierarchie stammenden.

4. Einen Zeiger auf das eigentliche Feld als `ICorDebugValue` erstellen. Dieser Zeiger ergibt sich aus der Funktion `ICorDebugClass::GetStaticFieldValue` für statische Felder und `ICorDebugValue::GetFieldValue` für „Member“ eines Objekts. Die Information, ob ein Feld statisch ist oder nicht, stammt aus `CLRMetaField:fieldAttr`.

5. Erstellung eines speziellen, von `CLRFieldBase` abgeleiteten Objekts für das aus Schritt 4 stammende `ICorDebugValue`-Objekt. Folgende drei Typen werden momentan unterstützt.

   - **CLRFieldValue**: Eine Klasse für primitive Typen und Strings. Enthält eine String-Repräsentation seines Werts sowie einen Zeiger auf ein `ICorDebugGenericValue` Objekt. Letzteres ermöglicht den Zugriff auf die primitiven Datentypen und ist für einen String einfach ein Null-Zeiger.

   - **CLRFieldReference**: Ein Typ für eine Referenz zu einem anderen geladenen Objekt. Um die Möglichkeit zu haben, das Objekt eindeutig zu identifizieren und zu finden, wird die Speicheradresse gespeichert.

   - **CLRFieldList**: Eine Klasse für Referenzen zu mehreren anderen geladenen Objekten. Hierfür wird eine Liste von Speicheradressen gespeichert.

6. Erstellung der eigentlichen Werte der Felder. Dazu wird für jedes in Schritt 5 erzeugte Feld der jeweils geforderte Wert, abhängig vom konkreten Typ, ausgelesen und gespeichert.

Der `FieldValueHelper` ist teilweise noch etwas experimentell und es hat mehrere Ansätze gegeben, worauf in Abschnitt 4.2.3 auf Seite 50 noch näher eingegangen wird.

Settings Die Klasse `Settings` ist nach dem Entwurfsmuster Singleton entworfen. Sie lädt eine Einstellungsdatei und stellt die dort enthaltenen Einstellungen dem Debugger global zur Verfügung. Durch diese Einstellungsklasse ist es möglich Konfigurationen am Debugger vorzunehmen, ohne die Projekte neu kompilieren zu müssen.

Die zu ladende Datei hat den Namen *clr_adapter_settings.xml*, liegt im XML-Format vor und wird mittels des Open Source Tools pugixml [Kap13] eingelesen und geparst. Um gefunden zu werden, muss sie für das Projekt *CLRDebugger* im Verzeichnis *Doc*[1] liegen und sich für das Projekt *CLRAdapter* im USE Plug-in-Verzeichnis im Ordner *monitor_adapter* befinden. Die Konfigurationsdatei für den aktuellen Debugger kann Anhang D entnommen werden.

Die folgenden Konfigurationsmöglichkeiten für ausgewählte Komponenten des Debuggers werden momentan angeboten:

- **TypeInfoHelper**: Die Einstellung *MinNumberOfModules* gibt die minimale Anzahl von geladenen Modulen an, die geladen sein muss, bevor ein Objekt vom Typ `TypeInfoHelper` als initialisiert gilt.

[1] Befindet sich auf einer Ebene mit dem Src Verzeichnis, das die Projektmappe enthält.

- **ObjectInfoHelper**: Hierfür sind zwei boolesche Einstellungen vorhanden. Als erstes gibt die Option *InMemoryInstanceMap* an, ob geladene Instanzen im `ObjectInfoHelper` gespeichert werden sollen. Als zweites wird mit *CacheAtStartUp* festgelegt, ob alle Instanzen beim ersten Aufruf von `GetInstances` geladen werden sollen. Wenn die zweite Einstellung *true* ist, so muss auch die erste *true* sein und es müssen Einträge unter dem Einstellungspunkt *TypesOfInterest* zu finden sein.

- **TypesOfInterest**: Hier werden alle Typen angegeben, die geladen werden sollen. Dies ist nur genau dann notwendig, wenn *CacheAtStartUp true* ist (siehe Abschnitt 4.2.2 auf Seite 42).

- **ModulesToIgnore**: Weil der `TypeInfoHelper` ohne Konfiguration alle aktuell geladenen Module lädt, kann man an dieser Stelle welche definieren, die er ignorieren soll. In den seltensten Fällen interessieren einen .NET-Systemmodule oder GUI-Module.

- **Debugger**: Unter diesem Punkt ist einstellbar, welche Ausgaben auf der Konsole vom *CLRDebugger* zur Laufzeit getätigt werden sollen. Auf die einzelnen Einstellungen wird an dieser Stelle nicht näher eingegangen, da die Namen selbsterklärend sind.

Jede der genannten Einstellungen wird im Debugger durch ein öffentliches Attribut der Klasse `Settings` repräsentiert.

InfoBoard Die Instanz der Klasse `InfoBoard` stellt den Debugger global Datenstrukturen und Variablen zur Verfügung. Da es nur ein Vorkommen des Typs im gesamten Programm geben soll, ist die Klasse nach dem Singleton Entwurfsmuster implementiert.

Die Häufigkeit der Benutzung dieser Klasse nahm mit verschiedenen Ansätzen mal zu, mal ab. Momentan besitzt sie nur eine globale Variable. Der Aufzählungsdatentyp `AppType` wird durch das gleichnamige Attribut repräsentiert und gibt zur Laufzeit des Debuggers Aufschluss darüber, ob es sich um das Projekt *CLRDebugger* oder *CLRAdapter* handelt.

JNIHelper Die Klasse `JNIHelper` stellt statische Methoden zur Verfügung, die vom JNI-Header benötigt werden. Hauptsächlich handelt es sich hierbei im Methoden, die Java-Objekte erstellen oder aus von Java übergebenen Objekten Informationen auslesen. Diese Klasse wird nur vom Projekt *CLRAdapter* verwendet.

HelperMethods Die Klasse `HelperMethods` stellt dem Debugger statische Hilfsmethoden zur Verfügung. Sie ist in der Header-Datei *CommonTypes.h* deklariert. Hauptsächlich sind dort Methoden zur Typ-Umwandlung enthalten, die verschiedene Datentypen oder Aufzählungen als String-Repräsentation zurückliefern. Hier werden auch die primitiven Feldwerte in Zeichenketten umgewandelt, damit sie z. B. in Debug-Ausgaben leichter zu verwenden sind.

DebugBuffer Bei der Klasse `DebugBuffer` handelt es sich um eine Hilfsklasse, die von den COM-Komponenten zurückgelieferte „Char-Reihungen“ einfach und komfortabel verwalten kann. Beim Konstruieren wird eine „Char-Reihung“ mit der dem Konstruktor übergeben Größe erstellt und die Größe gespeichert. Im Destruktor wird diese Reihung wieder freigegeben. Des Weiteren stellt ein `DebugBuffer` eine *CString*-Repräsentation seines Inhalts zur Verfügung.

Allgemeine Typ-Definitionen Konstrukte wie z. B. Aufzählungsdatentypen und eigene Typen erleichtern dem Programmierenden die Arbeit und verbessern die Lesbarkeit des Codes. Der Abbildung 4.3 auf Seite 38 können in der Mitte die Strukturen und Aufzählungsdatentypen sowie unten die selbst definierten Typen des Debuggers entnommen werden. Die Wichtigsten davon finden sich im Folgenden näher erklärt.

Die Struktur `CStringHash` stellt eine Hashfunktion für *CString*s bereit. Dadurch ist es möglich, CStrings in Hash-Datenstrukturen als Schlüssel zu verwenden.

Für den Debugger wurden die folgenden drei Aufzählungsdatentypen definiert:

- **AppType**: gibt Aufschluss darüber, ob es sich beim Debugger zur Laufzeit um das Projekt *CLRAdapter* oder *CLRDebugger* handelt. Diese Information ist standardmäßig auf den Wert *Adapter* gesetzt und muss kurz nach dem Start des *CLRDebuggers* auf den entsprechenden Wert gesetzt werden.
- **FieldType**: identifiziert den speziellen Feldtypen. Dies ist eine einfache Lösung um bei polymorphen Typen den genauen Typ herauszufinden. Damit wird im C++-Teil das Fehlen des aus Java bekannten Schlüsselworts *instanceof* kompensiert.
- **TypeInfo**: sagt aus, ob es sich bei einem Typen um eine normale oder abstrakte Klasse oder eine Schnittstelle handelt.

Für die C++-Datenstrukturen `unordered_set` und `unordered_map` wurden, je nach Verwendungszweck, eigene Typen mit den – falls notwendig – dazugehörigen Typen für die Einträge definiert.

So gibt es Mengen für Module und CStrings, in denen ein und derselbe Eintrag genau einmal vorkommen darf. Für Typen, Objekte und Felder sind Datenstrukturen definiert, deren Elemente ein ebenfalls definiertes Schlüssel-Wert-Paar darstellen.

Implementierung des JNI-Headers Die, wie in Abschnitt 2.4 auf Seite 23 beschrieben, generierten Methoden des JNI-Headers bilden die Schnittstelle zwischen dem C++- und dem Java-Teil des *CLRAdapters*.

Nachdem die Bibliothek *CLRAdapter* vom Java-Programm geladen worden ist, sind beide Seiten in der Lage diese Methoden aufzurufen. Momentan gibt es nur Methodenaufrufe von der Java-Seite aus und zwar genau dann, wenn der Monitor Daten zur Generierung des Snapshots benötigt oder der Adapter initialisiert werden soll. Die Implementierung der Methoden wird nun näher erklärt.

Der JNI-Header ist von der Datei *CLRAdapter.cpp* implementiert. Global werden dort jeweils eine Instanz vom `TypeInfoHelper` sowie vom `ObjectInfoHelper` definiert. Um den Adapter zu initialisieren, muss zunächst, nachdem der Monitor gestartet wurde, die Methode `attachToCLR` aufgerufen werden. Danach ist der Debugger mit dem Debuggee verbunden. Diese Methode hat folgende Signatur.

Listing 4.2: Signatur einer JNI-Methode

```
JNIEXPORT jint JNICALL
 Java_org_tzi_use_monitor_adapter_clr_CLRAdapter_attachToCLR
 (JNIEnv* env, jobject adapter, jlong pid)
```

Die ersten beiden Parameter sind für alle JNI-Methoden des CLR-Adapters identisch. Nummer 1 ist ein Zeiger auf die Java-Umgebung, mit der z. B. neue Java-Objekte erzeugt oder als Parameter übergebene Objekte ausgelesen werden können. Nummer 2 ist die Java-Klasse, in der die JNI-Methoden deklariert stehen. Der Parameter `pid` übergibt dem *CLRAdapter* die Prozess-ID des Debuggees. Innerhalb der Methode wird die Initialisierung von `CLRDebugCore` mit Hilfe der PID und eines neu erstellten `CLRDebugCallback` durchgeführt. Ist das `CLRDebugCore`-Objekt erfolgreich initialisiert worden, so gibt die Methode 0 zurück, ansonsten beträgt der Rückgabewert -1.

Wie ein Java-Objekt als Parameter an C++ übergeben und dessen Zustand ausgelesen werden kann, zeigt das nächste Codebeispiel, in welchem auf den Namen eines `CLRType`-Objekts (Java-Klasse) zugegriffen wird. Auf den Code für die Fehlerbehandlung wird in dieser Darstellung der Übersicht halber verzichtet.

Wie Listing 4.3 zu entnehmen ist, wird als erstes die Klasse des übergebenen Parameters bestimmt. Danach kann man mit Hilfe der eben bestimmten Klasse, des Methodennamens und der Signatur der Methode die Methoden-ID von `GetName` herausbekommen. Diese ID wird nun für den Aufruf der Methode benutzt und der Rückgabewert wird als Objekt gespeichert. Da es sich bei dem zurückzugebenden Wert um einen String handelt, wird er in einen für C++ verständlichen Datentypen umgewandelt und kann anschließend benutzt werden. Am Ende der Methode ist noch an die Freigabe des allozierten Speichers zu denken.

Listing 4.3: Auslesen eines übergebenen Java-Objekts mit JNI

```
jclass typeClass = env->GetObjectClass(clrType);
jmethodID typeGetName = env->
  GetMethodID(typeClass, "getName", "()Ljava/lang/String;");
jobject typeName = env->
  CallObjectMethod(clrType, typeGetName);
const char* strTypeName = env->
  GetStringUTFChars((jstring)typeName, NULL);

// do something with strTypeName ...

// release string
env->ReleaseStringUTFChars((jstring)typeName, strTypeName);
```

Listing 4.4: Erstellung einer Java-Datenstruktur mit JNI

```
jobject hashSet = env->NewGlobalRef(NULL);

// create java HashSet
jclass setClass = env->FindClass("java/util/HashSet");
jmethodID setConstructor = env->
  GetMethodID(setClass, "<init>", "()V");
hashSet = env->NewObject(setClass, setConstructor);

// add method to add objects
jmethodID setAdd = env->
  GetMethodID(setClass, "add", "(Ljava/lang/Object;)Z");
```

Mit JNI ist es auch möglich, beliebige Java-Datenstrukturen zu erstellen und zu verwalten. Beispielhaft zeigt das Listing 4.4. Dort wird ein *HashSet* erstellt, indem zuerst die Klasse mit Hilfe des genauen Pfades gesucht und mit Hilfe der Methoden-ID des Konstruktors ein neues Objekt erstellt wird. Danach ist es möglich die Datenstruktur mittels der Methode `add` zu füllen. Alle Methoden eines *HashSet* sind in dieser Weise aufrufbar. Das so erstellte Objekt kann als Rückgabewert einer JNI-Methode dienen und somit nach dem Erstellen aus dem Java-Programm heraus wie gewohnt verwendet werden.

In den Methodensignaturen sind große Buchstaben zu finden, über dessen Bedeutung Tabelle 4.1 auf der nächsten Seite Aufschluss gibt. Sie alle repräsentieren einen bestimmten Datentypen [Gor98].

Mit Hilfe der hier gezeigten JNI-Features ist es möglich, alle Objekte und Daten-

Typ-Signatur	Java-Typ
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double
L<fully-qualified-class>	fully-qualified-class: Eine Klasse mit komplettem Pfad.
[<Typ>	Reihung von <Type>
(<Arg.-Type Liste>)<Rückgabe-Typ>	Methodensignatur

Tabelle 4.1.: JNI Signaturen

strukturen aus dem C++-Teil dieser Arbeit heraus zu erzeugen und zu verwalten. Daraus folgt ein relativ schlanker Java-Abschnitt für den CLR-Adapter, in dem die benötigten JNI-Methoden einzeln besprochen werden (siehe Abschnitt 4.3 auf Seite 52).

4.2.3. Alternative und nicht funktionierende Ansätze

Bis zur aktuellen Version des CLR-Adapters waren einige Ansätze notwendig. Ein Problem bei der Implementierung des CLR-Adapters war die für MSDN Verhältnisse recht dürftige Dokumentation der CLR Debug API [Mic13j]. Einige dort enthaltene Funktionen sind (noch?) nicht implementiert oder liefern fehlerhafte Werte zurück. Dieser Absatz beschreibt zwei alternative Ansätze, die ausprobiert wurden, bevor der aktuelle Stand des Debuggers erreicht worden ist.

Der erste Versuch des Debuggers hat nicht, wie jetzt implementiert, die Metadaten beim Programmstart eingelesen, sondern startete mit der Iteration über den Heap mittels der Methode `ICorDebugProcess5::EnumerateHeap`. Dabei wurde für jedes Objekt der Typ – via String-Vergleich der Namen – daraufhin geprüft, ob er zu denen gehört, die analysiert werden sollen. Wenn dies der Fall war, wurde das Objekt in einer globalen Datenstruktur gespeichert und die Felder mit Hilfe der Methode `ICorDebugProcess5::GetTypeFields` untersucht. Mit der daraus resultierenden Datenstruktur war es problemlos möglich, über die Felder eines Objekts zu iterieren und deren Metadaten zu bekommen. Als nun versucht wurde die Feldwerte auszulesen, traten Probleme auf. Beim Versuch die `ICorDebugValue` Objekte für die einzelnen Felder mittels der Methode `ICorDebugValue::GetFieldValue` zu erstellen, wurden teilweise Fehler geworfen. Der erste Verdacht, dass die Methode

`ICorDebugProcess5 ::GetObjectW`, mit der auch bei diesem Ansatz die Objekte aus dem Speicher mittels ihrer Adresse erstellt wurden, fehlerhaft sei, hat sich leider bewahrheitet [Fal12] – für den aktuellen Adapter wird sie noch verwendet, allerdings mit Daten aus anderen Schnittstellen.

Weitere Versuche, den Speicher direkt anhand der Speicheradresse des Objekts und den aus `ICorDebugProcess5::GetTypeFields` bekannten Offsets auszulesen, waren zeitaufwändig und schlugen fehl.

Der nächste Ansatz, an die Feldwerte zu gelangen, war das Auslesen der Metadaten anstelle des Aufrufs der Methode `ICorDebugProcess5::GetTypeFields`. Die Metadaten lieferten die korrekten Typen und Token zurück um die Feldwerte erfolgreich auszulesen. Seitdem wurde bei der Iteration über den Heap für die Metadaten jedes einzelne Objekt geladen, was funktioniert, aber noch nicht optimal war. Das Problem, dass so gelöst wurde, scheint eine Kombination der beiden Methoden `GetObjectW` und `GetTypeFields` aus der CLR-Schnittstelle `ICorDebugProcess5` zu sein.

Nach dieser Änderung konnten Feldwerte zwar gelesen werden, jedoch wurden die Metadaten bei jeder Iteration über den Heap neu erstellt und der Debugger musste beim Start bereits wissen, welche Typen er zu untersuchen hat. Dieses System war ähnlich der jetzigen dritten Optimierungsstrategie des `ObjectInfoHelper`. Um das zu verbessern, wurde die zweite größere Änderung vorgenommen.

Da nun bekannt war, dass die Metadaten leicht auszulesen sind und mit ihnen alle nötigen Informationen zur Verfügung stehen, um die vom CLR-Adapter benötigten Daten zurückzuliefern, sollte ihnen eine größere Bedeutung zukommen. Der `TypeInfoHelper` wurde so umgebaut, dass er beim Start die Metadaten der geladenen Module einliest und anhand derer Informationen eine Typ-Struktur aus Klassen, Feldern und Vererbungshierarchien speichert. Durch das neue Design ist es auch nicht mehr notwendig, dass der Debugger die Typen, die er laden soll, vor dem Start mitgeteilt bekommt. Diese Datenstruktur des `TypeInfoHelper` liegt allen weiteren Aktionen zu Grunde. Das ist jetzt der Entwurf des Debuggers, der den Endpunkt dieser Arbeit darstellt.

Die Feldwerte sind auszulesen für Strings, primitive Datentypen und Referenzen. Experimentell sind Reihungen mit Referenzen bereits im Debugger enthalten, liefern jedoch zur Zeit noch die falschen Speicheradressen zurück. An diesem Punkt muss als nächstes überprüft werden, ob auch dieses Problem mit den gewählten Ansätzen zu lösen ist.

Falls sich mehr Probleme ergeben sollten – oder auch nur das jetzige nicht gelöst werden kann – gibt es die Möglichkeit, die dem jetzigen Ansatz zugrundeliegende Funktion `ICorDebugProcess5::GetObjectW` nicht mehr zu benutzen.

Dann wäre ein Weg z. B. die Funktionen `ICorDebugProcess5::GetTypeForTypeID` zu verwenden, um mit Hilfe der Typinformationen zu wissen, wie groß ein Objekt

und wie der Speicher zu interpretieren ist [Fal12]. Dazu werden die Informationen benötigt, die in der beim Iterieren über den Heap zurückgelieferten Struktur `COR_HEAPOBJECT` stehen. Für die Feldwerte ist dann die – nicht mit `GetObjectW` kombiniert hoffentlich korrekt arbeitende – Funktion `ICorDebugProcess5::GetTypeFields` zu benutzen. Aus der daraus resultierenden Struktur `COR_FIELD` für ein Feld können ebenfalls mittels `ICorDebugProcess5::GetTypeForTypeID` die Informationen, wie groß ein Feld ist und wie der Speicher zu interpretieren ist, gewonnen werden. Die Speicheradresse eines Feldes ergibt sich aus der Adresse des Objekts addiert mit dem Offset des Feldes. Der Speicher kann dann mit der Funktion `ReadProcessMemory` ausgelesen werden.

Ein ähnlicher Ansatz, komplett auf die Methode `ICorDebugProcess5::GetObjectW` zu verzichten, wurde bereits ohne Erfolg ausprobiert. Ein weiteres ausprobieren dieses Ansatzes hätte den Rahmen einer Diplomarbeit gesprengt. Bis jetzt erschient das Auslesen und das korrekte Interpretieren des Speichers als relativ aufwendig im Vergleich zum in dieser Arbeit vorgestellten Ansatz.

Alle in diesem Absatz diskutierten Änderungen am CLR-Adapter oder alternativen Ansätze sind mit hohem Zeitaufwand verbunden. Dies hat sich während der Arbeit mit der CLR Debug API herausgestellt. Es gibt in dieser API viele Wege, um ein Ziel zu erreichen. Ein überdurchschnittlich hoher Aufwand liegt in der Ermittlung der für ein Ziel richtigen Kombination von CLR-Schnittstellen, insbesondere da einige von ihnen nicht korrekt funktionieren oder noch nicht implementiert sind.

4.3. Aufbau des CLR-Adapters auf der Java-Seite

Der Java Teil des CLR-Adapters ist im Vergleich zum C++-Teil simpel gehalten und orientiert sich an der Implementierung des bereits fertigen Java-Adapters. Seine Hauptaufgabe besteht darin, als Schnittstelle zwischen Java und C++ zu fungieren. Sie stellt die im C++-Teil ausgelesenen Daten über den Debuggee dem Monitor in einer ihm verständlichen Art und Weise zur Verfügung.

Diese Arbeit implementiert auf der C++-Seite ausschließlich den statischen Teil des Monitors, weshalb auch in Java nur die dafür verantwortlichen Klassen implementiert werden. Das sind im einzelnen der Adapter, die Klassen für Typen, Objekte und Felder (siehe Abbildung 2.4 auf Seite 16) sowie noch „Wrapper-Klassen“ für die Feldwerte. Eine detaillierte Beschreibung findet sich auf den kommenden Seiten.

4.3.1. Implementierung der Metadaten-Klassen

Die Implementierung dieser Klassen orientiert sich an den Beispielen aus dem Java-Adapter und beinhaltet nur den statischen Teil. Eine Übersicht gibt Abbildung 4.4 auf der nächsten Seite.

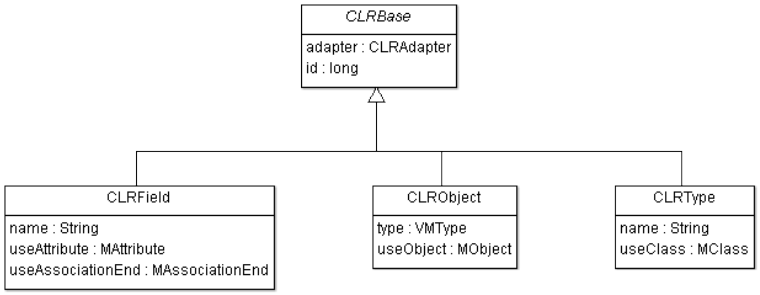

Abbildung 4.4.: Java CLR-Metadaten der Virtuellen Maschine.

Die Basisklasse `CLRBase` beinhaltet die beiden für jede Spezialisierung obligatorischen Attribute `adapter` und `id`. Das erste ist der `CLRAdapter` und das zweite ist die eindeutige ID mit der die Instanzen der jeweiligen Klasse eindeutig identifiziert werden können. Diese ID ist je nach Typ ein CLR-Metadaten-Token oder eine Speicheradresse. Welche das im Einzelnen ist, und welche Metadaten-Klasse mit welcher Klasse aus dem CLR-Adapter korrespondiert, kann Tabelle 4.2 entnommen werden. Dort aufgelistet sind auch die jeweiligen Metadaten-Schnittstellen, die von den einzelnen Klassen implementiert werden.

Java-Klasse	Schnittstelle	CLR-Klasse	ID
CLRField	VMField	CLRMetaField	Token: mdFieldDef
CLRObject	VMObject	CLRObject	Speicheradresse
CLRType	VMType	CLRType	Token: mdTypeDef

Tabelle 4.2.: Metadaten-Klassen, ihre Schnittstellen, ihre IDs und ihre Gegenstücke auf der CLR-Seite

Die hier dargestellten Java-Klassen werden auf Anfrage des Adapters auf der C++-Seite erstellt und danach von der Java Seite in einer dem Monitor verständlichen Repräsentation weitergeleitet.

4.3.2. Java Wrapper-Klassen für Felder

Um Java die CLR-Feldwerte verständlich zu machen, sind Wrapper-Klassen für unterschiedliche Arten von Feldern entworfen worden. Zur Zeit werden Felder der folgenden Form unterstützt: primitive Datentypen, Referenzen, Reihungen mit Referenzen als Einträgen (experimentell). Siehe Abbildung 4.5 auf der nächsten Seite.

Welche Java-Klasse mit welcher CLR-Klasse korrespondiert, kann Tabelle 4.3 auf der nächsten Seite entnommen werden.

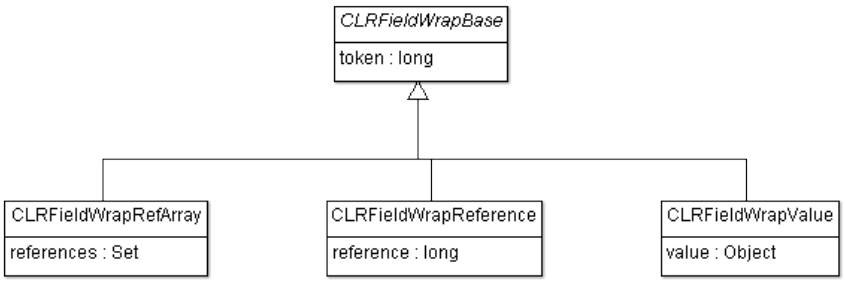

Abbildung 4.5.: Java Wrapper-Klassen für Felder.

Java-Klasse	CLR-Klasse	Wert
CLRFieldWrapValue	CLRFieldValue	Object (enthält den Wert)
CLRFieldWrapReference	CLRFieldReference	Speicheradresse
CLRFieldWrapRefArray	CLRFieldList	Liste mit Speicheradressen

Tabelle 4.3.: Wrapper-Klassen, ihre Gegenstücke auf der CLR-Seite und ihr eigentlicher Wert

Dieses System kann beliebig für andere Arten von Feldwerten erweitert werden. Eine sinnvolle Erweiterung für zukünftige Systeme wäre ein Wrapper für Listen mit primitiven Datentypen als Einträgen.

4.3.3. CLR-Adapter mit JNI-Methoden

Der CLR-Adapter wird auf der Java-Seite durch die Klasse `CLRAdapter` repräsentiert. Sie erweitert die abstrakte Klasse `AbstractVMAdapter`, die wiederum die Schnittstelle `VMAdapter` implementiert. Ausführlicher findet der Lesende eine Beschreibung dieser Typen in Abschnitt 2.2.1 auf Seite 14. An dieser Stelle wird nur auf Implementierungsdetails und die vom `CLRAdapter` deklarierten JNI-Methoden eingegangen.

Der `CLRAdapter` implementiert alle Methoden die notwendig sind, um dem Monitor einen Heap-Snapshot liefern zu können. Bis auf die Methode `getUSEValue` rufen alle anderen von ihnen JNI-Methoden auf um die gewünschten Werte zurückgeben zu können.

Feldwerte werden durch `getUSEValue` in einen für USE verständlichen Wert umgewandelt. Dazu bekommt die Methode eine Instanz von `CLRFieldWrapBase` als Parameter. Die Werte dieses Parameters werden je nach genauem Typ ausgelesen und dementsprechende USE-Objekte werden erstellt.

Im Folgenden werden die JNI-Methoden und ihre Aufgaben näher erläutert.

- **attachToCLR**: Verbindet den Adapter mittels der PID mit dem Debuggee. Es ist erst sinnvoll weitere JNI-Methoden aufzurufen, wenn der CLR-Adapter vollständig initialisiert ist. Ist das nicht der Fall, können die Rückgabewerte der Methoden fehlerhaft sein. Dies kann mit Hilfe der Methode `isCLRAdapterInitialized` herausgefunden werden.
- **resumeCLR**: Setzt einen pausierenden Debuggee fort.
- **suspendCLR**: Pausiert einen laufenden Debuggee.
- **stopCLR**: Stoppt das Monitoring, trennt den Adapter vom Debuggee und gibt verwendeten Speicher frei.
- **isCLRAdapterInitialized**: Gibt genau dann *true* zurück, wenn eine definierte Mindestanzahl an Modulen geladen ist.
- **isCLRClassType**: Liefert *true* zurück, wenn der ihr als Parameter übergebene Typ eine Klasse repräsentiert.
- **getCLRSuperClasses**: Liefert eine Liste mit Typen zurück, welche die Basistypen des ihr als Parameter übergebenen Typs repräsentieren.
- **getCLRSubClasses**: Liefert eine Liste mit Typen zurück, welche die Subtypen des ihr als Parameter übergebenen Typs repräsentieren.
- **getFieldByName**: Gibt eine Feldbeschreibung anhand des als Parameter übergebenen Namens zurück.
- **getWrappedField**: Gibt einen Feldwert für eine Feldbeschreibung eines Objekts zurück.
- **getInstances**: Liefert eine Liste mit allen aktuell geladenen Instanzen eines bestimmten Typs zurück.
- **getCLRType**: Gibt eine Typ-Beschreibung auf Basis seines als Parameter übergebenen Namens zurück, sofern ein entsprechender Typ gefunden werden konnte.

All diese Methoden rufen Methoden aus dem in Abschnitt 4.2.2 auf Seite 38 beschriebenen C++-Teil auf, weshalb dort auch die meiste Arbeit investiert wurde. In Java reicht lediglich die Deklaration aus, um die im nativen Teil erstellten Objekte als Rückgabewert zu erhalten und mit ihnen wie gewohnt weiterarbeiten zu können. Aus der Sicht von Java ist JNI dementsprechend einfach zu benutzen.

Um sicherzustellen, dass der Snapshot erst erstellt wird, nachdem der CLR-Adapter vollständig initialisiert ist, wird nach dem Aufruf `attachToCLR` solange gewartet, bis `isCLRAdapterInitialized` einen wahren Wert zurückliefert oder eine bestimmte

Zeit (momentan 750 ms) abgelaufen ist. Ist die Zeit vorüber, wird eine Fehlermeldung geworfen und der Verbindungsversuch abgebrochen.

5. Ergebnisse und Auswertung

In diesem Kapitel wird der CLR-Adapter unter den Aspekten Korrektheit und Performance evaluiert. Dazu wird der implementierte CLR-Adapter anhand dafür ausgedachter Testfälle überprüft. Die dabei gewonnenen Ergebnisse werden anschließend ausgewertet.

Weil mit USE ein stabiles System für die Validierung zugrunde liegt, ist es ausreichend genug zu zeigen, dass USE die korrekten Daten vom CLR-Adapter geliefert bekommt.

5.1. Ergebnisse

Um an Daten für die Auswertung zu gelangen, wurde eine Testumgebung für den CLR-Adapter gebaut. In dem nun folgenden Abschnitt wird zuerst diese Umgebung beschrieben und danach die Ergebnisse verschiedener Tests anschaulich dargestellt. Es wird die Korrektheit anhand eins bekannten Snapshots, die Effizienz sowie die Verbindung mit einem beliebigen Open Source Programm – bestehend aus verwaltetem C#-Code – getestet.

5.1.1. Aufbau der Testumgebung

Um die Implementierung des CLR-Adapters zu testen, gibt es den bereits bekannten C#-Debuggee, dessen Entwurf in Abschnitt 2.1.1 auf Seite 6 ausführlich beschrieben ist. Des Weiteren ist nach dem gleichen Entwurf ein Java-Programm erstellt worden, mit dem ein Performance-Vergleich der beiden Technologien möglich ist. Es wird auch kurz darauf eingegangen, wie USE in Verbindung mit dem CLR-Adapter zu starten ist, um sich mit einem CLR-Programm zu verbinden.

Debuggee „AnimalWorld“ in C#

Der Debuggee ist nach dem Plattform unabhängigen Klassendiagramm in Abbildung 2.1 auf Seite 7 implementiert. Die plattformabhängige Umsetzung ist Abbildung 5.1 auf der nächsten Seite zu entnehmen.

Den beiden Diagrammen kann entnommen werden, dass die Assoziation *ParentHood* durch die Eigenschaft `Pet::Father` abgebildet ist. Die Assoziation *PetColor* ist dargestellt mittels der Eigenschaft `Pet::Color`. Alle weiteren Attribute eines Typs sind als Eigenschaften der entsprechenden Klasse wiederzufinden.

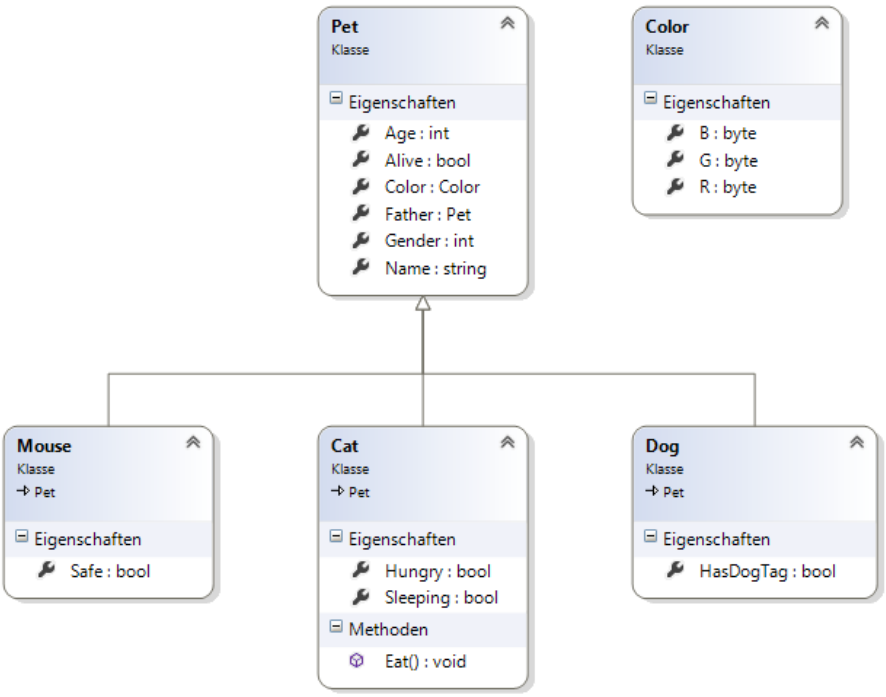

Abbildung 5.1.: Klassendiagramm des in C# geschriebenen Debuggees für die „AnimalWorld“ (erstellt mit dem Visual Studio von Microsoft, richtet sich nicht streng nach der UML)

Für Auswertungszwecke stehen dem Debuggee zwei Möglichkeiten zur Verfügung. Die Methode `CreateBasicSnapshot` erzeugt genau die Objekte, die auch für das Objektdiagramm in Abbildung 2.2 auf Seite 8 durch die USE-Spezifikation im Anhang B.2 erzeugt worden sind. Um eine große Anzahl von Objekten zu erzeugen, gibt es die Methode `CreateLargeSnapshot`.

Beide Methoden erzeugen darüber hinaus eine bestimmte Anzahl von Objekten des Typs `ClassWithoutInterest`. Diese Instanzen interessieren den Monitor nicht bei der Erzeugung eines Snapshots. Sie dienen der Simulation beliebiger Programme, die üblicherweise aus wesentlich mehr Klassen bestehen als diejenigen, welche vom Monitor analysiert werden sollen.

Der Debuggee für die „AnimalWorld“ ist in C# als .NET 4.0-Konsolen-Anwendung geschrieben und für eine 32-Bit Plattform kompiliert.

Debuggee „AnimalWorld“ in Java

Das Java-Programm, welches die „AnimalWorld“ implementiert, ist nach dem gleichen plattformabhängigen Klassendiagramm (Abbildung 5.1 auf der vorherigen Seite) aufgebaut wie sein C#-Pendant. Es setzt die Assoziationen ebenso identisch um, wie es auch die Methoden `CreateBasicSnapshot` und `CreateLargeSnapshot` nach gleicher Art und Weise implementiert.

Besonders interessant für die Auswertung ist die Generierung vieler Objekte, damit die Performance zweier identisch aufgebauter Programme mit großen Objektmengen getestet werden kann.

Verbindung von USE mit Monitor-Plug-in und CLR-Adapter zu einem .NET-Programm

Damit USE den Monitor mit dem CLR-Adapter benutzen kann, müssen die Dateien *MonitorAdapter_CLR.jar*, *CLRAdapter.dll* und *clr_adapter_settings.xml* im USE-Ordner *lib/pluins/monitor_adapter* vorhanden sein.

Um die Bibliothek des CLR-Adapters finden zu können, muss USE der Pfad beim Programmstart als JVM-Argument wie folgt übergeben werden:
`-Djava.library.path=<path_to_lib_folder>`.

Für eine erfolgreiche Verbindung muss der Debugge als 32-Bit-Kompilat vorliegen und aus verwaltetem Code bestehen. Die meisten Open Source Programme sind nicht für 32-Bit kompiliert, sodass sie vorher neu zu erstellen sind.

Nachdem USE gestartet und das Modell geladen wurde, kann als nächstes der Monitor gestartet werden. Dazu öffnet sich der Kontroll-Dialog (siehe Abbildung 5.2 auf der nächsten Seite), in welchem die PID des Debuggees eingegeben werden muss. Danach ist der Anwendende über diesen Dialog in der Lage den Monitor mit dem Debugge zu verbinden, ihn pausieren/fortlaufen zu lassen und den Monitor vom Debuggee zu trennen.

Nach einer erfolgreichen Verbindung wird vom Monitor ein Snapshot auf Basis der am Anfang geladenen USE-Spezifikation erstellt. Dieser kann danach wie üblich in USE validiert und graphisch dargestellt werden (siehe Abschnitt 2.2 auf Seite 13).

5.1.2. Snapshot der „AnimalWorld“

Um zu testen, ob der CLR-Adapter die korrekten Daten ausliest, erzeugt der Debuggee die identischen Objekte und Assoziationen, die auch von USE für das Objektdiagramm (siehe Abbildung 2.2 auf Seite 8) erstellt worden sind.

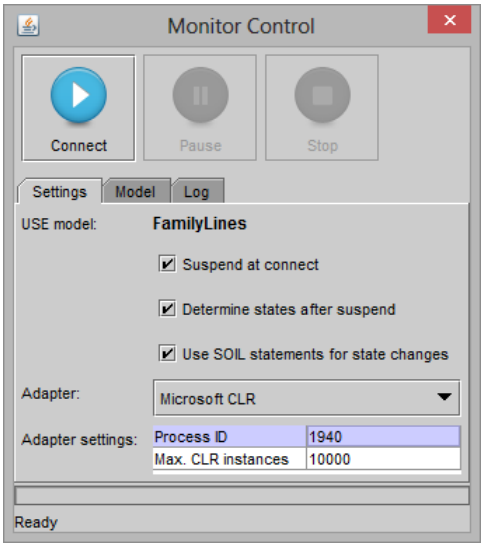

Abbildung 5.2.: Monitor-Kontrolldialog zur Übergabe der Parameter und Kontrolle des Debuggees

Der vom CLR-Adapter erstellte Snapshot ist der Abbildung 5.3 auf der nächsten Seite zu entnehmen. Das Ergebnis zeigt, dass Objekte, Attribute und Assoziationen korrekt ausgelesen werden.

Um die Vererbung zu testen, wurde in der USE-Spezifikation temporär die Klasse *Dog* auskommentiert. Dann sollte das Objekt mit dem Namen „Joe“ vom Typ *Dog* als Pet dargestellt werden.

Der Abbildung 5.4 auf der nächsten Seite ist zu entnehmen, dass das Objekt mit dem Namen „Joe“ als Pet mit den Attributen der Basisklasse dargestellt wird. Vererbung wird dementsprechend korrekt erkannt.

5.1.3. Performance des CLR-Adapters

Um die Performance des CLR-Adapters bewerten zu können, wird der Debuggee bei seinem Start eine große Datenmenge erzeugen. Als erstes werden die Optimierungsstrategien des CLR-Adapters miteinander vergleichen. Als zweites folgt ein Vergleich der besten Optimierungsstrategie mit dem Java-Debuggee[1].

[1] Alle Testdaten sind mittels eines Notebooks mit Intel® Core™ 2 Duo (2.53 GHz) und 4 GB Arbeitsspeicher erfasst worden.

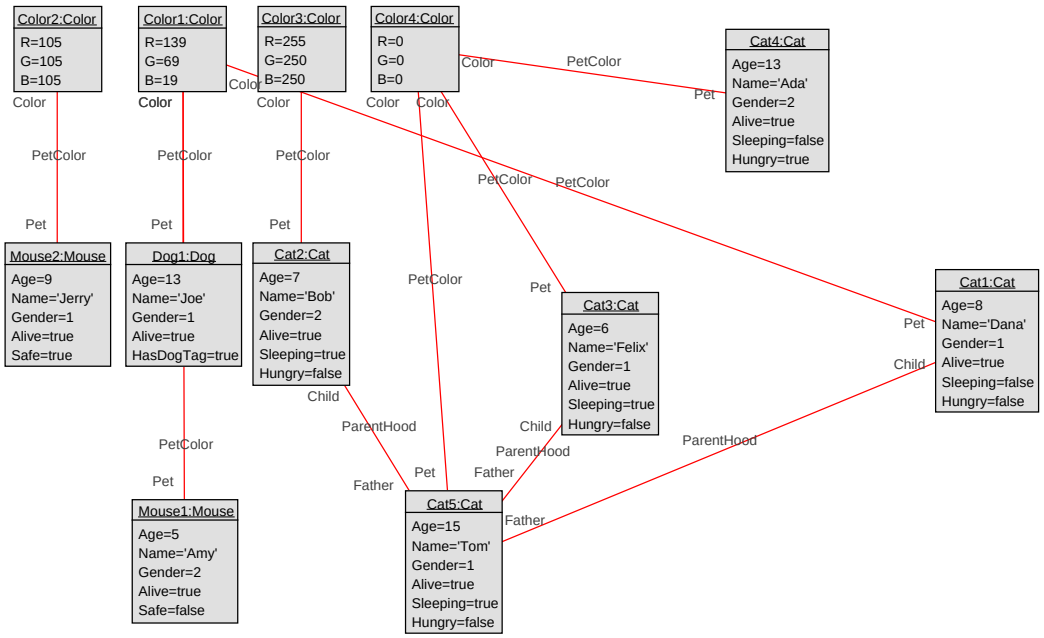

Abbildung 5.3.: Objektdiagramm des Debuggees

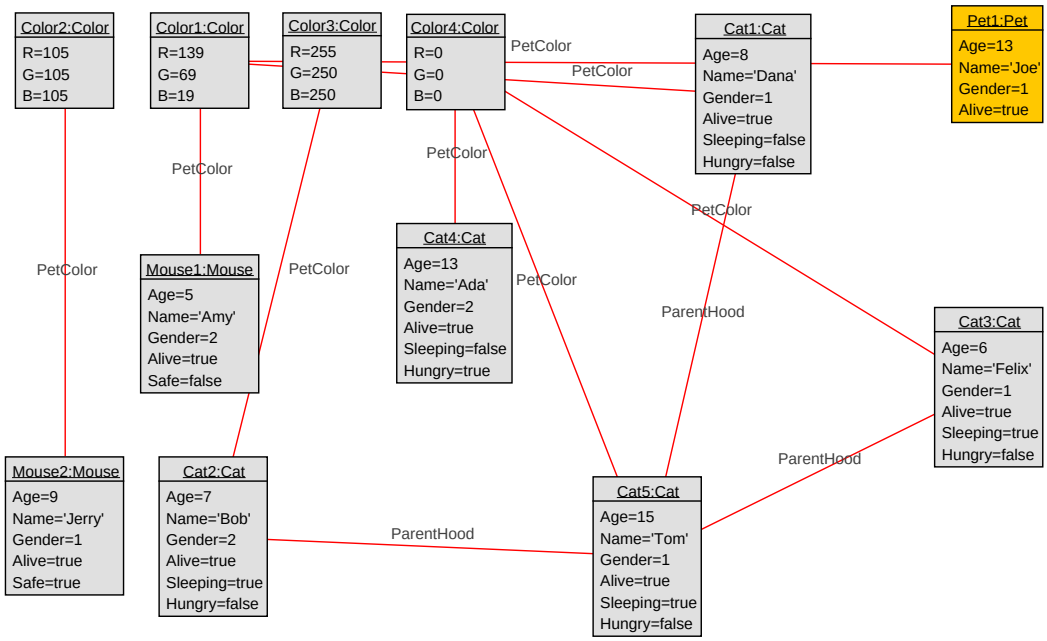

Abbildung 5.4.: Objektdiagramm des Debuggees mit Vererbung

Vergleich der Optimierungsstrategien des CLR-Adapters

Für diesen Test erzeugt der Debuggee 500 mal die bekannte Datenmenge der „AnimalWorld“ und USE verbindet sich mit jeder Optimierungsstrategie jeweils zu einer neuen Instanz des Debuggees.

Im Folgenden sind die drei möglichen Optimierungsstrategien noch einmal knapp zusammengefasst. Wie in Abschnitt 4.2.2 auf Seite 42 bereits erwähnt, dienen die Optimierungen 2 und 3 nur einem Performance-Vergleich und bieten nicht die volle Funktionalität. Die *Links* werden in der bisherigen Implementierung nach einer Neuverbindung nicht korrekt zurückgeliefert.

- *Optimierungsstrategie 1*: Es wird nichts optimiert.
- *Optimierungsstrategie 2*: Beim jedem Aufruf von *readInstances* werden alle Instanzen des aktuellen Typs gespeichert.
- *Optimierungsstrategie 3*: Beim *ersten* Aufruf von *readInstances* werden alle Instanzen von Typen, die der Adapter haben will, gespeichert. Es wird also nur einmal über den Heap iteriert.

Die folgenden Tabellen präsentieren die benötigten Zeiten um alle Instanzen zu erstellen und die Attribute auszulesen. Auf Links wird verzichtet, da sie in den Optimierungen, nachdem der Debuggee pausiert hat, nicht korrekt funktionieren. Die Zeiten wurden direkt nach der Verbindung sowie nachdem das Programm ein- bzw. zweimal nach einer Pasuse fortgesetzt wurde genommen.

	keine Optimierung	Optimierung 2	Optimierung 3
Verbindung	1803 instances/s	1756 instances/s	2923 instances/s
1. Fortsetzung	1906 instances/s	1920 instances/s	60 289 instances/s
2. Fortsetzung	–	1870 instances/s	56 864 instances/s

Tabelle 5.1.: Performance des CLR-Adapters beim Erstellen der Instanzen

In Tabelle 5.1 gibt es für die zweite Fortsetzung des Programms keine Daten. Das liegt daran, dass jedes Iterieren über den Heap sehr viel Speicher benötigt. Er wird trotz der Freigabe der Ressourcen anscheinend nicht komplett freigegeben. Das bedeutet, dass dem Monitor nach einigen wenigen Verbindungen ohne Optimierung der Speicher ausgeht. Gleiches gilt ebenso für Tabelle 5.2 auf der nächsten Seite.

Für die Auswertung der Attribute ist zu beachten, dass Felder nicht explizit gespeichert werden. Allerdings profitieren sie ggf. von gespeicherten Objekten, welche die Felder enthalten.

	keine Optimierung	Optimierung 2	Optimierung 3
Verbindung	4585 attributes/s	6878 attributes/s	6878 attributes/s
1. Fortsetzung	4585 attributes/s	5502 attributes/s	5502 attributes/s
2. Fortsetzung	–	5502 attributes/s	5502 attributes/s

Tabelle 5.2.: Performance des CLR-Adapters beim Auslesen der Attribute

Die für den Test erstellten 5000 Instanzen wurden experimentell bestimmt. Sie repräsentieren die obere Grenze für das Testsystem, bei welcher der CLR-Adapter, auf der verwendeten Testumgebung, reproduzierbar stabil läuft.

Performance-Vergleich: CLR-Adapter vs. Java

Um Daten für einen Vergleich mit Java zu bekommen, wird die in Java geschriebene „AnimalWorld“ benutzt. Sie erstellt ebenfalls 5000 Instanzen. Die Ergebnisse können der Tabelle 5.3 entnommen werden.

	Verbindung	1. Fortsetzung	2. Fortsetzung
Instanzen	5234 instances/s	9065 instances/s	9131 instances/s
Attribute	9171 attributes/s	9171 attributes/s	9171 attributes/s
Links	6500 links/s	6500 links/s	6500 links/s

Tabelle 5.3.: Performance des Java-Adapters zum Vergleich

Der Vergleich zeigt deutlich, dass der Java-Adapter den Snapshot schneller erstellt. Die Attribute werden in etwa doppelt so schnell ausgelesen und die Instanzen – im Vergleich mit dem CLR-Adapter ohne Optimierung – bis zu vier mal so schnell erstellt.

Bei der Erzeugung der Links erreicht ein optimierter CLR-Adapter identische Zeiten wie der Java-Adapter. Diese Werte sind jedoch nicht aussagekräftig, da die Links nach einer Neuverbindung nicht neu erstellt werden. Die nicht optimierte Version des CLR-Adapters benötigte die etwa doppelte Zeit für die Erstellung der Links.

Das Potential des CLR-Adapters wird bei der Erstellung der Instanzen mit der Optimierungsstrategie 3 für Neuverbindungen deutlich. Dort ist er um ein sechsfaches schneller als sein Pendant auf der Java-Seite.

5.1.4. Verbindung des CLR-Adapters mit einem beliebigen C#-Open Source-Programm

Als Open Source C#-Programm ist die Entscheidung auf *Family Lines* [13a] gefallen, da es die graphisch unterstützte Erstellung eines Familienstammbaums ermöglicht und somit perfekt zum Kontext des Testprogramms „AnimalWorld“ passt.

Nach dem Start der Family Lines ist der Anwendende in der Lage einen Familienstammbaum aus der Sicht einer Person anzulegen. Dabei ist das Programm intuitiv zu bedienen.

Um auf die „AnimalWorld“ einzugehen, wurde die Katzenfamilie des Vaters Tom aus Sicht des Sohns Bob angelegt. Der Abbildung 5.5 sind die Verwandtschaftsverhältnisse der angelegten Katzen zu entnehmen.

Abbildung 5.5.: Das laufende Programm FamilyLines

Bevor sich der Debugger mit den Family Lines verbindet, können die Metadaten mit Hilfe des Programms *ildasm.exe* betrachtet werden. Dieses Tool wird mit dem Visual Studio ausgeliefert und kann über dessen Eingabeaufforderung gestartet werden.

Den Metadaten können Klassendefinitionen mit ihren Feldern entnommen werden. So gibt es die Klasse `Person`, die unter anderem die Felder `firstName` und `lastName` besitzt. Das klingt nach der Klasse, welche die erstellte Personen repräsentiert. Ob das wirklich der Fall ist, lässt sich durch den Debugger herausfinden.

Verbindung mit dem CLR-Debugger

Wird der CLRDebugger etwas modifiziert, sodass er zuerst den Typen für die Person in den geladenen Typen eines Moduls sucht und anschließend dessen Instanzen sowie die Werte der gewünschten Felder ausgibt, dann erhält man die folgende Ausgabe.

```
Loaded instance of type: KBS.FamilyLinesLib.Person
         Address: 52978952
         Vorname: Unknown        Nachname:
Loaded instance of type: KBS.FamilyLinesLib.Person
         Address: 60852764
         Vorname: Bob    Nachname: Cat
Loaded instance of type: KBS.FamilyLinesLib.Person
         Address: 60856760
         Vorname: Tom    Nachname: Cat
Loaded instance of type: KBS.FamilyLinesLib.Person
         Address: 60860444
         Vorname: Ada    Nachname: Cat
Loaded instance of type: KBS.FamilyLinesLib.Person
         Address: 60863440
         Vorname: Felix  Nachname: Cat
Loaded instance of type: KBS.FamilyLinesLib.Person
         Address: 60866152
         Vorname: Dana   Nachname: Cat
```

Der Konsolen-Ausgabe des CLRDebuggers sind die in Abbildung 5.5 auf der vorherigen Seite dargestellten Katzen (Personen) zu entnehmen sowie eine weitere ohne Namen. Daraus folgt, dass der CLRDebugger bei diesem Test die korrekten Daten zurückliefert.

Auffällig an diesem Test sind zwei Punkte. Zum einen braucht die Iteration über den Heap sehr lange (etwa 1 s) und zum anderen benötigt der Debugger etwa 20 s, um sich vom Debuggee zu trennen.

Verbindung mit USE und dem CLR-Adapter

Um die Family Lines mit USE validieren zu können, wird eine Spezifikation (siehe Listing 5.1) benötigt. Diese enthält nur die Klasse Person. Da die Assoziationen in Family Lines mit Datenstrukturen implementiert wurden, die der CLR-Adapter nicht unterstützt.

Listing 5.1: USE Spezifikation für das Programm Family Lines

```
@Monitor(defaultPackage="KBS.FamilyLinesLib")
model FamilyLines
class Person
  attributes
    firstName : String
    lastName : String
end
```

Wird die Spezifikation in USE geladen und der Monitor via CLR-Adapter mit dem Programm FamilyLines verbunden, so wird der in Abbildung 5.6 dargestellte Snapshot erstellt.

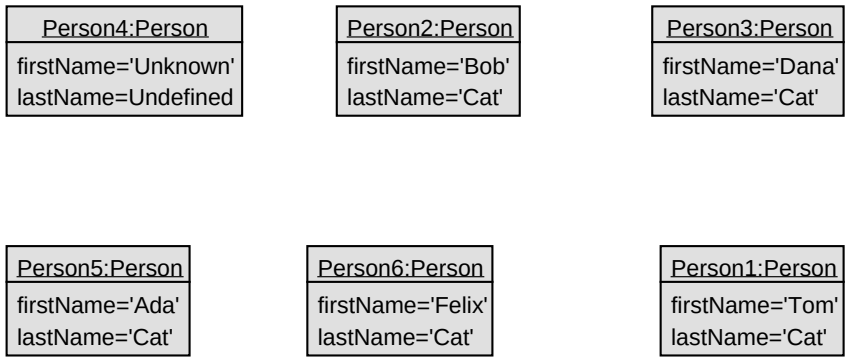

Abbildung 5.6.: Der Snapshot des Programms FamilyLines als USE Objektdiagramm

Der Abbildung 5.6 kann entnommen werden, dass die in Abbildung 5.5 auf Seite 64 erzeugten Instanzen vom Type `Person` korrekt eingelesen werden.

Des Weiteren ist in Abbildung 5.6 zu erkennen, dass eine Instanz vom Typ `Person` ohne vom CLRAdapter auslesbare Feldwerte erzeugt worden ist. Sie ist nicht in dem Beispiel aus Abbildung 5.5 auf Seite 64 vom Anwendenden erzeugt und somit vom Programm *Familiy Lines* intern instantiiert. Die Ursache kann eine fehlerhafte Implementierung sein.

Wie anhand dieses Tests zu erkennen ist, ist es mit Hilfe des Monitors möglich eventuelle Implementierungsfehler bzw. Abweichungen von der Spezifikation oder Speicherlecks aufzudecken.

5.2. Auswertung

Die bisherigen Ergebnisse zeigen, dass der CLR-Adapter in der Lage ist, die Typen, deren Feldstruktur sowie die Vererbungshierarchie der Typen untereinander, eines

geladenen Moduls auszulesen. Die geladenen Instanzen der einzelnen Typen und dessen Feldwerte – mit den bereits erläuterten Einschränkungen – können ausgelesen werden. Als Feldwerte werden primitive Datentypen und Referenzen vollständig unterstützt.

Mit diesen Funktionalitäten ist der CLR-Adapter in der Lage einen Snapshot für USE zu erstellen. Dieser kann die entsprechenden Objekte, Feldwerte und Assoziationen enthalten. Des Weiteren werden die Vererbungshierarchien korrekt an USE weitergegeben.

Ein überraschendes Ergebnis liefert die Performance-Analyse. Der nicht optimierte CLR-Adapter ist deutlich langsamer als sein Java-Gegenstück. Dies scheint vor allem an der sehr teuren Operation zur Iteration über den Heap zu liegen, da sich im Speicher befindliche Instanzen bis zu zehnfach schneller als bei der Java-Version zurückgeben lassen (Optimierungsstrategie 3). Damit ist der Ansatz für die Kommunikation zwischen nativem C++ und Java via JNI wie erwartet schnell genug für das Monitoring von .NET-Programmen.

Die bisherigen Optimierungen sind noch nicht ausgereift genug, um als benutzbar gelten zu können. Für einen Einsatz des CLR-Adapters bei größeren Projekten ist es nach jetzigen Erkenntnissen allerdings sinnvoll, einen optimierten Adapter zu verwenden. Darum empfiehlt es sich, in dieser Richtung nach weiteren Lösungen zu suchen.

Des Weiteren wurde gezeigt, dass das in [HGH12] eingeführte Metamodell für Virtuelle Maschinen leicht um weitere Technologien zu erweitern ist. Der Java-Teil des CLR-Adapters – unabhängig von der C++-Bibliothek – war relativ unkompliziert zu implementieren.

6. Zusammenfassung und Ausblick

Dieses Kapitel fasst zunächst die Vorgehensweise und Ergebnisse der vorliegenden Diplomarbeit nochmal zusammen. Anschließend werden die Zukunftsaussichten des entwickelten CLR-Adapters und einer technologieübergreifenden Erweiterung des USE Monitor-Plug-ins vorgestellt. Dies geschieht auf Grundlage der in Abschnitt 5.2 auf Seite 66 ausgewerteten Ergebnisse.

6.1. Zusammenfassung

In Kapitel 2 der vorliegenden Diplomarbeit wurde in die Grundlagen der Modelle für die Softwareentwicklung und deren graphischer Modellierungssprache UML eingeführt. Danach wurde die textuelle Object Constraint Language (OCL) als Teilmenge der UML erläutert, mit der aus dem Vertragsmodell bekannte Bedingungen definiert werden können. Beispielhaft erklärt wurden UML und OCL anhand des Entwurfs für die „AnimalWorld“, die als Testprogramm für den CLR-Adapter entworfen und benutzt wurde. Der aktuelle Stand der Wissenschaft in Bezug auf Validierung und Verifikation von UML/OCL Modellen zur Laufzeit wurde betrachtet und erörtert, bevor die wichtigsten Funktionen von USE und dem Monitor-Plugin detailliert beschrieben wurden. Als neue Technologie, in Bezug auf das in Java geschriebene USE, wurde das .NET Framework mit der CLR von Microsoft grundlegend eingeführt und Vergleiche zu Java gezogen. Da die CLR Debug API – der für den CLR-Adapter wichtige Teil wurde in Kapitel 3 beschrieben – aus COM-Schnittstellen besteht, wurde der Debugger in C++ geschrieben. Für die Interoperabilität zwischen Java und C++ wurde das Java Native Interface (JNI) benutzt. Dieses Zusammenspiel unterschiedlicher Technologien bildet den Hauptbestandteil der vorliegenden Arbeit.

Nach Einführung in die benötigten theoretischen Grundlagen wurde der Entwurf und die Implementierung des CLR-Adapters erläutert. Dabei wurde auf die für diesen Monitoring-Ansatz verwendeten Komponenten, die Verantwortlichkeiten der einzelnen Klassen und auf benutzte Entwurfsmuster näher eingegangen. Des Weiteren ist auch die Erklärung verschiedener Optimierungs- und Konfigurationsmöglichkeiten Bestandteil des praktischen Teils dieser Arbeit.

Abschließend wurde der implementierte CLR-Adapter einigen Tests unterzogen. Dabei wurde besonders die Korrektheit der ausgelesenen Werte, die Performance und

die Anwendbarkeit auf beliebige, verwaltete C#-Programme untersucht und ausgewertet.

6.2. Fazit

Die Implementierung hat gezeigt, dass der Monitor dank dem Metamodell für Virtuelle Maschinen leicht um eine weitere Technologie zu erweitern ist. Zumindest gilt diese Aussage für den Teil der Arbeit, der auf der Java Seite zu implementieren ist und beschränkt sich auf die statischen Elemente des Monitoring-Ansatzes.

Den CLR-Adapter auf der nativen Seite zu implementieren war dagegen zeitaufwändiger als erwartet. Dies liegt zum einen an der nicht so umfangreichen Dokumentation der CLR Debug API – wie man es sonst nicht von der MSDN gewohnt ist – und zum anderen an teilweise nicht oder nur fehlerhaft implementierten Schnittstellen der CLR Debug API. Dadurch musste viel ausprobiert werden, bevor die gewünschten Informationen über den Debuggee mit Hilfe der CLR Debug API ausgelesen werden konnten.

Die Auswertung des CLR-Adapters hat gezeigt, dass eine Erweiterung des USE Monitors um eine CLR-Schnittstelle möglich ist. Damit der CLR-Adapter voll genutzt werden kann, ist noch weitere Arbeit zu investieren. Die Funktionalität gilt es noch zu erweitern und die Performance zu optimieren. Einige Ideen und Ansätze hierzu werden in Abschnitt 6.3 genauer erläutert.

6.3. Ausblick

Der CLR-Adapter besitzt die grundlegende Funktionalität einen Heap-Snapshot zu erstellen, ist allerdings im jetzigen Zustand noch nicht reif für eine Übernahme in USE. Dazu fehlen das dynamische Verhalten sowie andere Features. Welche das im einzelnen sind und wie mögliche Lösungsansätze aussehen, wird im Folgenden präsentiert. Für einige Probleme werden mehrere Ansätze erläutert, welche diskutiert werden sollten.

Die Auswertung hat gezeigt, dass der CLR-Adapter ohne Optimierung um bis zu ein Vierfaches langsamer ist als sein Pendant auf der Java-Seite. In der optimierten Version wird allerdings das Potential der Interoperabilität via JNI deutlich. Als Optimierung wird hier in Ansätzen eine Art Repository mit Zeigern auf die CLR-Instanzen aus dem Heap erzeugt. Dieses ist noch um Feldwerte zu ergänzen. Das Problem ist nicht das Erstellen einer solchen Datenstruktur, sondern die Aktualisierung bei einer Umorganisation der Speicherstruktur durch den CLR Garbage Collector. Hier sind zwei unterschiedliche Lösungsansätze diskussionswürdig. Ein Ansatz ist, in einem zweiten Thread in bestimmten zeitlichen Abständen über den Heap zu iterieren und

das Repository zu aktualisieren. Der andere, etwas erfolgversprechendere Ansatz, ist die Benutzung der CLR Profiling API [Mic13j]. Seit der .NET Version 4.5 ist es möglich einen Profiler, genau wie einen Debugger, an einen laufenden Prozess anzuhängen. Beide können gemeinsam an einen Prozess angehängt werden und diesen parallel untersuchen. Da der Profiler Callback-Funktionen zur Verfügung stellt, die bei Heap-Veränderungen aufgerufen werden [Mic13d], kann anhand dieser Informationen das Repository verwaltet werden. Diese Variante erscheint vielversprechender, da nicht in regelmäßigen Abständen erneut über den Heap iteriert werden muss.

Der in dieser Arbeit nicht berücksichtigte dynamische Teil des Monitors ist noch zu implementieren. Um die Veränderung von Feldwerten zu erkennen, kann in den zum Auslesen der Werte verwendeten `ICorDebugValue` Objekten ein Breakpoint gesetzt werden. Sobald sich der Wert danach verändert, wird eine Callback-Funktion aufgerufen. Um Vor- und Nachbedingungen von Methodenaufrufen zu überprüfen gibt es zwei Ansätze. Zum einen ist ebenfalls die Verwendung von Breakpoints möglich. Zum anderen kann die Profiling API [Mic13j] benutzt werden. Sie kann mittels Callback-Methoden Informationen darüber liefern, wenn Methoden betreten und verlassen werden.

Sollten die ersten beiden Punkte erfolgreich umgesetzt werden, so gibt es weitere Features, dessen Implementierung sinnvoll erscheint. Diese sind z. B. die Unterstützung von Aufzählungsdatentypen sowie die Möglichkeit Feldwerte auszulesen, die auf Datenstrukturen zeigen. Des Weiteren müssen zu analysierende Typen zur Zeit aus einem Modul kommen. Das könnte erweitert werden, indem man einen zusammengesetzten Schlüssel für Typen aus Metadaten-Token und Modul verwendet.

Im Rahmen dieser Diplomarbeit ist eine weitere Idee entwickelt worden. Um .NET zu unterstützen, muss nicht zwingend die CLR von Microsoft benutzt werden. Eine weitere Möglichkeit bietet Mono. Die Debug-Schnittstelle von Mono [Xam13b] ist an die von Java angelehnt. Dadurch sollte sie leicht zu benutzen sein, da man sich bei der Implementierung am fertigen Java-Adapter orientieren kann und somit bereits Erfahrungen auf diesem Gebiet bestehen. So würde für einen eventuellen Mono-Adapter noch das Problem der Interoperabilität zwischen Mono und Java bestehen. Dafür bietet sich das Werkzeug IKVM [13b] an. Mit diesen Mitteln ist auch dieses eine Option für einen weiteren Adapter und somit eine Diskussion wert.

A. Physikalische Struktur des CLR-Adapters

Der CLR-Adapter besteht aus vier Komponenten. Auf der Java-Seite gibt es das Eclipse-Projekt *use-monitor-adapter-clr* und auf der .NET-Seite existiert die Visual Studio Projektmappe *CLR-Monitoring* mit den Projekten *CLRAdapter*, *CLR-Debugger* und *Debuggee*. Die Verantwortlichkeiten der einzelnen Projekte wird im folgenden genauer erläutert.

- **use-monitor-adapter-clr**: Das Java-Projekt implementiert den `VMAdapter` (siehe Abschnitt 2.2.1 auf Seite 14), der dem Monitor die gewünschten Daten (Snapshot) zur Verfügung stellt und deklariert die nativen Methoden für die Kommunikation mit der C++-Bibliothek *CLRAdapter*.

- **CLRAdapter**: Das Projekt *CLRAdapter* ist das Herzstück dieses Monitoring-Ansatzes. Die C++-DLL implementiert den aus dem Projekt *use-monitor-adapter-clr* generierten JNI-Header (siehe Abschnitt 2.4 auf Seite 23) und übernimmt das eigentliche Debuggen des zu untersuchenden .NET-Programms.

- **CLRDebugger**: Dieses Projekt erstellt eine native Windowsanwendung (EXE) mit der gleichen Funktionalität wie der *CLRAdapter*. Sie dient zum Testen neuer Features des Adapters oder zum ersten Auslesen eines zu untersuchenden Programms. Dieser Konsolen-Anwendung übergibt man als Parameter die PID des gewünschten Programms, sie verbindet sich mit dem Debuggee und kurz danach gibt sie dem Anwendenden die geladenen Module und Typen zurück.

- **Debuggee**: Das Projekt ist ein verwaltetes C# 4.0-Programm. Es implementiert die in Abschnitt 2.1.1 auf Seite 6 vorgestellte „AnimalWorld“ und dient als Testprogramm für die Projekte *CLRDebugger* und *CLRAdapter*.

Wie die Projektmappe strukturiert ist und wie die ersten Schritte mit Visual Studio aussehen, wird im Folgenden beschrieben. Eine Projektmappe ist nichts weiter als eine Struktur, in der verschiedene Projekte und Ordner zusammengefasst sind und die vom Visual Studio komplett geladen wird.

A.1. Verzeichnisstruktur des CLR-Adapters

Nachdem öffnen der Projektmappe *CLR-Monitoring.sln* kann dem Projektmappen-Explorer die in Abbildung A.1 dargestellte Projektstruktur entnommen werden.

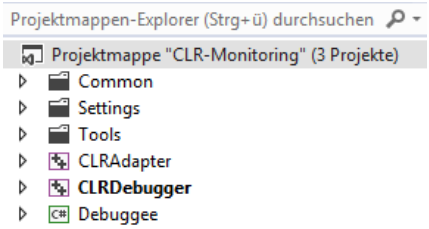

Abbildung A.1.: Visual Studio Projektstruktur der Projektmappe *CLR-Monitoring*

Die Projekte *CLRAdapter*, *CLRDebugger* und *Debuggee* sind die im vorherigen Abschnitt beschriebenen. Die Bedeutung der drei Projektmappenordner *Common*, *Settings* und *Tools* wird nun erläutert.

- **Common**: Das Verzeichnis enthält Dateien mit Klassen, die in beiden Projekten – also *CLRAdapter* und *CLRDebugger* – benötigt werden. Dadurch sind spätere Änderungen nur an einer Stelle einzubauen, was das Debuggen und spätere Warten erheblich vereinfacht.
- **Settings**: Der Ordner enthält die Einstellungs-/Konfigurationsdatei für die Projekte *CLRAdapter* und *CLRDebugger*. Ohne die Software neu kompilieren zu müssen ist es dadurch möglich, dass Verhalten des Debuggers anzupassen.
- **Tools**: In diesem Ordner sind Klassen oder Bibliotheken anderer Anbieter enthalten, die von einem oder mehreren Projekten der vorliegenden Arbeit benutzt werden.

Wie man Abbildung A.1 an der Hervorhebung ebenfalls entnehmen kann, ist *CLR-Debugger* als Standardprojekt festgelegt. Das bedeutet nichts weiter, als dass dieses Projekt beim Debuggen durch das Visual Studio geöffnet wird. Ein Startprojekt kann ganz einfach im Kontextmenü eines Projekts festgelegt werden. Mehr zu diesem Thema im nächsten Abschnitt.

A.2. Kompilieren und starten des CLR-Debuggers

Um alle Projekte der Projektmappe *CLR-Monitoring.sln* kompilieren zu können, ist noch etwas Vorarbeit notwendig. Voraussetzung an das System ist ein installiertes Windows Vista oder höher mit dem .NET Framework 4.5 und einem Visual Studio

2012. Ist das System wie eben beschrieben konfiguriert, so sind noch die folgenden Projekteigenschaften[1] festzulegen.

Zunächst ist für die Projekte *CLRAdapter* und *CLRDebugger* eine zusätzliche Abhängigkeit auf die Bibliotheken *corguids.lib* und *mscoree.lib* hinzuzufügen. Dies ist unter Projekteigenschaften → Konfigurationseigenschaften → Linker → Eingabe zu konfigurieren.

Danach müssen nun noch die JNI-Abhängigkeiten dem Projekt *CLRAdapter* hinzugefügt werden. Dazu wird das Verzeichnis *include* der JDK Installation den Includeverzeichnissen des Adapters hinzugefügt. Dies ist unter Projekteigenschaften → Konfigurationseigenschaften → VC++-Verzeichnisse → Includeverzeichnisse einzustellen.

Die oben genannten Abhängigkeiten sind bereits in den vorliegenden Projektmappen konfiguriert, es ist allerdings durchaus möglich, dass der Anwendende die Pfade seiner lokalen Installation anzupassen hat.

Nachdem vorherige Arbeitsschritte erledigt sind, ist es möglich, die komplette Projektmappe zu kompilieren. Die dabei erzeugten, ausführbaren Dateien befinden sich alle in dem Ordner *Build*[2].

Nun ist der Anwendende in der Lage, erste Schritte mit dem Projekt zu unternehmen. Am einfachsten ist es den Debuggee zu starten und danach den Debugger mit ihm zu verbinden. Die dazu notwendigen Schritte sind im Folgenden näher erläutert.

1. Die Konsolen-Anwendung *Debuggee* wird mittels einer Eingabeaufforderung gestartet. Nach dem Start gibt der Debugge folgende Zeilen aus:

   ```
   Running under .NET 4.0.30319.18046
   My PID: 1940
   Press enter to exit!
   ```

   Die erste Zeile gibt Aufschluss über die .NET-Version, unter der das Programm läuft. Der zweiten Zeile ist die aktuelle Prozess ID (PID) zu entnehmen und mit Betätigung der Enter-Taste wird das Programm beendet (dies sollte der Anwendende nicht tun, damit eine Verbindung mit dem laufenden Debuggee möglich ist).

2. Wie in Abbildung A.2 auf der nächsten Seite dargestellt, wird dem CLRDebugger die PID als Argument übergeben.

3. Starten des lokalen Windows-Debuggers von Visual Studio aus. Dadurch wird das Startprojekt, der *CLRDebugger* mit den in Schritt 2 eingestellten Befehlszeilenargumenten, gestartet. Nach dem Starten des Debuggers wird dieser an

[1] Befindet sich im Kontextmenü eines Projekts.
[2] Befindet sich auf einer Ebene mit dem Src Verzeichnis, der die Projektmappe enthält.

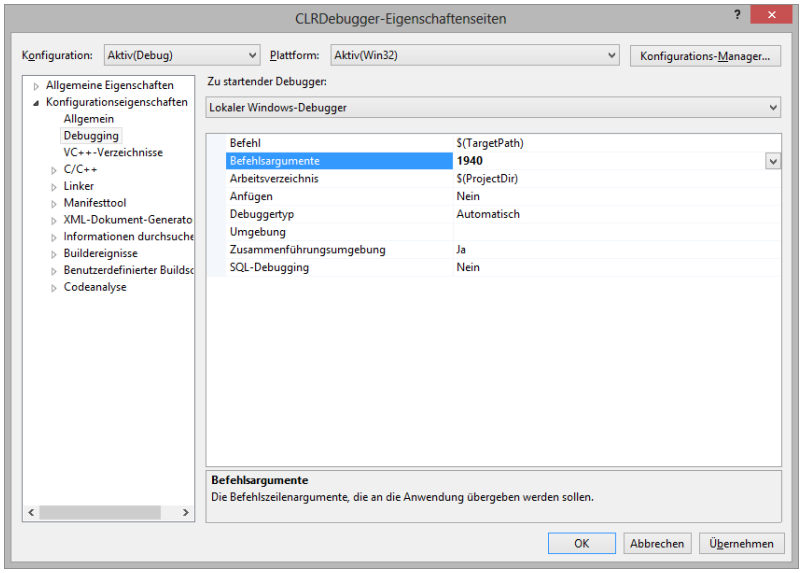

Abbildung A.2.: Übergabe der PID des Debuggees an den Debugger.

den Debuggee angehängt und gibt Basisinformationen über geladene Module, Typen und Vererbungshierarchien auf der Konsole aus.

Durch Modifikation der Einstellungen kann Einfluss auf die zu ladenden Module, und damit auch auf die geladenen Typen genommen werden. Wie das im Detail funktioniert, wird in Abschnitt 4.2.2 auf Seite 45 beschrieben.

Diese schrittweise Einführung zeigt, dass es nicht viel Arbeit erfordert den *CLRDebugger* an einen verwalteten .NET Debuggee anzuhängen und dessen Informationen auszulesen. Wie das funktioniert und welche Komponenten dabei eine zentrale Rolle spielen, wird im Kapitel 4 auf Seite 31 näher erläutert.

B. USE Spezifikationen für das Auswertungsprogramm

B.1. UML Modell

```
@Monitor(defaultPackage="Debuggee")

model AnimalWorld

-- classes

class Pet
  attributes
    Age : Integer
    Name : String
    Gender : Integer
    Alive : Boolean
end

class Dog < Pet
  attributes
    HasDogTag : Boolean
end

class Cat < Pet
  attributes
    Sleeping : Boolean
    Hungry : Boolean
  operations
    Eat(m : Mouse)
    begin
      self.Hungry := false;
      m.Alive := false;
    end
end
```

```
class Mouse < Pet
  attributes
    Safe : Boolean
end

class Color
  attributes
    R : Integer
    G : Integer
    B : Integer
end

-- associations

association PetColor between
  Pet[0..*] role Pet
  Color[1] role Color
end

association ParentHood between
  Pet[0..1] role Father
  @Monitor(ignore="true")
  Pet[0..*] role Child
end

-- constraints

constraints

context Pet inv AgeNotNegative:
  Age >= 0

context p1:Pet inv NameIsUnique:
  Pet.allInstances->forAll(p2 |
    p1.Name = p2.Name implies p1 = p2)

context p1:Pet inv FatherNotChild:
  not(p1.Father = p1) and
    p1.Child->excluding(p1) = p1.Child

context p1:Pet inv FatherIsMale:
  p1.Father = null or p1.Father.Gender = 1

context Color inv ColorRange:
```

```
  R >= 0 and R <= 255 and
  G >= 0 and G <= 255 and
  B >= 0 and B <= 255

context Cat::Eat(m : Mouse)
  pre UnsafeMouse:      not(m.Safe)
  pre CatIsHungry:      Hungry
  pre CatIsNotSleeping: not(Sleeping)
  pre CatIsAlive:       Alive
  post CatNotHungry:    not(Hungry)
  post VictimIsDead:    not(m.Alive)
```

B.2. Snapshot

```
-- create cats

!create Tom : Cat
!set Tom.Name := 'Tom'
!set Tom.Age := 15
!set Tom.Gender := 1
!set Tom.Hungry := false
!set Tom.Sleeping := true
!set Tom.Alive := true

!create Ada : Cat
!set Ada.Name := 'Ada'
!set Ada.Age := 13
!set Ada.Gender := 2
!set Ada.Hungry := true
!set Ada.Sleeping := false
!set Ada.Alive := true

!create Bob : Cat
!set Bob.Name := 'Bob'
!set Bob.Age := 7
!set Bob.Gender := 1
!set Bob.Hungry := false
!set Bob.Sleeping := true
!set Bob.Alive := true

!create Dana : Cat
!set Dana.Name := 'Dana'
```

```
!set Dana.Age := 8
!set Dana.Gender := 2
!set Dana.Hungry := false
!set Dana.Sleeping := false
!set Dana.Alive := false

!create Felix : Cat
!set Felix.Name := 'Felix'
!set Felix.Age := 6
!set Felix.Gender := 1
!set Felix.Hungry := false
!set Felix.Sleeping := true
!set Felix.Alive := true

-- create mice

!create Jerry : Mouse
!set Jerry.Name := 'Jerry'
!set Jerry.Age := 9
!set Jerry.Gender := 1
!set Jerry.Alive := true
!set Jerry.Safe := true

!create Amy : Mouse
!set Amy.Name := 'Amy'
!set Amy.Age := 5
!set Amy.Gender := 2
!set Amy.Alive := true
!set Amy.Safe := false

-- create one dog

!create Joe : Dog
!set Joe.Name := 'Joe'
!set Joe.Age := 13
!set Joe.Gender := 1
!set Joe.Alive := true
!set Joe.HasDogTag := true

-- create some colors

!create Black : Color
!set Black.R := 0
!set Black.G := 0
```

```
!set Black.B := 0

!create Gray : Color
!set Gray.R := 105
!set Gray.G := 105
!set Gray.B := 105

!create Brown : Color
!set Brown.R := 139
!set Brown.G := 69
!set Brown.B := 19

!create Snow : Color
!set Snow.R := 255
!set Snow.G := 250
!set Snow.B := 250

-- assign color to pet

!insert (Tom, Black) into PetColor
!insert (Jerry, Gray) into PetColor
!insert (Amy, Snow) into PetColor
!insert (Joe, Brown) into PetColor
!insert (Bob, Snow) into PetColor
!insert (Dana, Brown) into PetColor
!insert (Felix, Black) into PetColor
!insert (Ada, Black) into PetColor

-- assign cat childs to father Tom

!insert (Tom, Bob) into ParentHood
!insert (Tom, Dana) into ParentHood
!insert (Tom, Felix) into ParentHood
```

C. Ausschnitt der Metadaten des Testprogramms

```
TypeDef #1 (02000002)
-------------------------------------------------------
TypDefName: Debuggee.Pet  (02000002)
Flags     : [NotPublic] [AutoLayout] [Class] [AnsiClass] [BeforeFieldInit]
            (00100000)
Extends   : 01000001 [TypeRef] System.Object
Field #1 (04000001)
-------------------------------------------------------
Field Name: Father (04000001)
Flags     : [Public]  (00000006)
CallCnvntn: [FIELD]
Field type:  Class Debuggee.Pet

Field #2 (04000002)
-------------------------------------------------------
Field Name: <Age>k__BackingField (04000002)
Flags     : [Private]  (00000001)
CallCnvntn: [FIELD]
Field type:  I4
CustomAttribute #1 (0c000012)
-------------------------------------------------------
CustomAttribute Type: 0a000012
CustomAttributeName: System.Runtime.CompilerServices
.CompilerGeneratedAttribute :: instance void .ctor()
Length: 4
Value : 01 00 00 00                  >               <
ctor args: ()


Field #3 (04000003)
-------------------------------------------------------
Field Name: <Name>k__BackingField (04000003)
Flags     : [Private]  (00000001)
CallCnvntn: [FIELD]
```

```
Field type:  String
CustomAttribute #1 (0c000014)
-------------------------------------------------------
CustomAttribute Type: 0a000012
CustomAttributeName: System.Runtime.CompilerServices
.CompilerGeneratedAttribute :: instance void .ctor()
Length: 4
Value : 01 00 00 00                 >                <
ctor args: ()


Field #4 (04000004)
-------------------------------------------------------
Field Name: <Alive>k__BackingField (04000004)
Flags     : [Private]  (00000001)
CallCnvntn: [FIELD]
Field type:  Boolean
CustomAttribute #1 (0c000016)
-------------------------------------------------------
CustomAttribute Type: 0a000012
CustomAttributeName: System.Runtime.CompilerServices
.CompilerGeneratedAttribute :: instance void .ctor()
Length: 4
Value : 01 00 00 00                   >                <
ctor args: ()


Field #5 (04000005)
-------------------------------------------------------
Field Name: <Gender>k__BackingField (04000005)
Flags     : [Private]  (00000001)
CallCnvntn: [FIELD]
Field type:  I4
CustomAttribute #1 (0c000018)
-------------------------------------------------------
CustomAttribute Type: 0a000012
CustomAttributeName: System.Runtime.CompilerServices
.CompilerGeneratedAttribute :: instance void .ctor()
Length: 4
Value : 01 00 00 00                   >                <
ctor args: ()


Field #6 (04000006)
```

```
-------------------------------------------------------
Field Name: <Color>k__BackingField (04000006)
Flags     : [Private]  (00000001)
CallCnvntn: [FIELD]
Field type:  Class Debuggee.Color
CustomAttribute #1 (0c00001a)
-------------------------------------------------------
CustomAttribute Type: 0a000012
CustomAttributeName: System.Runtime.CompilerServices
.CompilerGeneratedAttribute :: instance void .ctor()
Length: 4
Value : 01 00 00 00                     >               <
ctor args: ()


Method #1 (06000001)
-------------------------------------------------------
MethodName: get_Age (06000001)
Flags     : [Public] [HideBySig] [ReuseSlot] [SpecialName]
(00000886)
RVA       : 0x000020d0
ImplFlags : [IL] [Managed]  (00000000)
CallCnvntn: [DEFAULT]
hasThis
ReturnType: I4
No arguments.
CustomAttribute #1 (0c000001)
-------------------------------------------------------
CustomAttribute Type: 0a000012
CustomAttributeName: System.Runtime.CompilerServices
.CompilerGeneratedAttribute :: instance void .ctor()
Length: 4
Value : 01 00 00 00                     >               <
ctor args: ()


Method #2 (06000002)
-------------------------------------------------------
MethodName: set_Age (06000002)
Flags     : [Public] [HideBySig] [ReuseSlot] [SpecialName]
(00000886)
RVA       : 0x000020e7
ImplFlags : [IL] [Managed]  (00000000)
CallCnvntn: [DEFAULT]
```

```
hasThis
ReturnType: Void
1 Arguments
Argument #1:  I4
1 Parameters
(1) ParamToken : (08000001) Name : value flags: [none] (00000000)
CustomAttribute #1 (0c000011)
-------------------------------------------------------
CustomAttribute Type: 0a000012
CustomAttributeName: System.Runtime.CompilerServices
.CompilerGeneratedAttribute :: instance void .ctor()
Length: 4
Value : 01 00 00 00            >                <
ctor args: ()
```

D. Einstellungen des CLR-Adapters

```
<?xml version="1.0" encoding="UTF-8"?>
<Settings>
  <TypeInfoHelper MinNumberOfModules="1"></TypeInfoHelper>
  <ObjectInfoHelper InMemoryInstanceMap="1"
                    CacheAtStartUp="0">
  </ObjectInfoHelper>
  <TypesOfInterest>
    <TypeOfInterest Name="Debuggee.Cat"></TypeOfInterest>
    <TypeOfInterest Name="Debuggee.Dog"></TypeOfInterest>
    <TypeOfInterest Name="Debuggee.PetColor"></TypeOfInterest>
  </TypesOfInterest>
  <ModulesToIgnore>
    <ModuleToIgnore Name="mscorlib.dll"></ModuleToIgnore>
    <ModuleToIgnore Name="System.Core.dll"></ModuleToIgnore>
    <ModuleToIgnore Name="System.dll"></ModuleToIgnore>
  </ModulesToIgnore>
  <Debugger PrintSettings="1"
            PrintAllModules="0"
            PrintLoadedModules="1"
            PrintLoadedTypes="1"
            PrintLoadedTypesFields="0"
            PrintInheritance="1">
  </Debugger>
</Settings>
```

E. Generierter JNI-Header

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class org_tzi_use_monitor_adapter_clr_CLRAdapter */

#ifndef _Included_org_tzi_use_monitor_adapter_clr_CLRAdapter
#define _Included_org_tzi_use_monitor_adapter_clr_CLRAdapter
#ifdef __cplusplus
extern "C" {
#endif
#undef org_tzi_use_monitor_adapter_clr_CLRAdapter_SETTING_PID
#define org_tzi_use_monitor_adapter_clr_CLRAdapter_SETTING_PID 0L
#undef org_tzi_use_monitor_adapter_clr_CLRAdapter_SETTING_MAXINSTANCES
#define org_tzi_use_monitor_adapter_clr_CLRAdapter_SETTING_MAXINSTANCES 1L
/*
 * Class:     org_tzi_use_monitor_adapter_clr_CLRAdapter
 * Method:    getInstances
 * Signature: (Lorg/tzi/use/plugins/monitor/vm/mm/clr/CLRType;)
 * Ljava/util/Set;
 */
JNIEXPORT jobject JNICALL
  Java_org_tzi_use_monitor_adapter_clr_CLRAdapter_getInstances
    (JNIEnv *, jobject, jobject);

/*
 * Class:     org_tzi_use_monitor_adapter_clr_CLRAdapter
 * Method:    getCLRType
 * Signature: (Ljava/lang/String;)
 * Lorg/tzi/use/plugins/monitor/vm/mm/clr/CLRType;
 */
JNIEXPORT jobject
  JNICALL Java_org_tzi_use_monitor_adapter_clr_CLRAdapter_getCLRType
    (JNIEnv *, jobject, jstring);

/*
 * Class:     org_tzi_use_monitor_adapter_clr_CLRAdapter
 * Method:    attachToCLR
```

```
 * Signature: (J)I
 */
JNIEXPORT jint JNICALL
  Java_org_tzi_use_monitor_adapter_clr_CLRAdapter_attachToCLR
    (JNIEnv *, jobject, jlong);

/*
 * Class:     org_tzi_use_monitor_adapter_clr_CLRAdapter
 * Method:    resumeCLR
 * Signature: ()I
 */
JNIEXPORT jint JNICALL
  Java_org_tzi_use_monitor_adapter_clr_CLRAdapter_resumeCLR
    (JNIEnv *, jobject);

/*
 * Class:     org_tzi_use_monitor_adapter_clr_CLRAdapter
 * Method:    suspendCLR
 * Signature: ()I
 */
JNIEXPORT jint JNICALL
  Java_org_tzi_use_monitor_adapter_clr_CLRAdapter_suspendCLR
    (JNIEnv *, jobject);

/*
 * Class:     org_tzi_use_monitor_adapter_clr_CLRAdapter
 * Method:    stopCLR
 * Signature: ()I
 */
JNIEXPORT jint JNICALL
  Java_org_tzi_use_monitor_adapter_clr_CLRAdapter_stopCLR
    (JNIEnv *, jobject);

/*
 * Class:     org_tzi_use_monitor_adapter_clr_CLRAdapter
 * Method:    getFieldByName
 * Signature: (Lorg/tzi/use/plugins/monitor/vm/mm/clr/CLRType;
 * Ljava/lang/String;)
 * Lorg/tzi/use/plugins/monitor/vm/mm/clr/CLRField;
 */
JNIEXPORT jobject
  JNICALL Java_org_tzi_use_monitor_adapter_clr_CLRAdapter_getFieldByName
    (JNIEnv *, jobject, jobject, jstring);
```

```
/*
 * Class:     org_tzi_use_monitor_adapter_clr_CLRAdapter
 * Method:    getWrappedField
 * Signature: (Lorg/tzi/use/plugins/monitor/vm/mm/clr/CLRType;
 * Lorg/tzi/use/plugins/monitor/vm/mm/clr/CLRObject;
 * Lorg/tzi/use/plugins/monitor/vm/mm/clr/CLRField;)
 * Lorg/tzi/use/plugins/monitor/vm/wrap/clr/CLRFieldWrapBase;
 */
JNIEXPORT jobject JNICALL
  Java_org_tzi_use_monitor_adapter_clr_CLRAdapter_getWrappedField
    (JNIEnv *, jobject, jobject, jobject, jobject);

/*
 * Class:     org_tzi_use_monitor_adapter_clr_CLRAdapter
 * Method:    isCLRAdapterInitialized
 * Signature: ()Z
 */
JNIEXPORT jboolean JNICALL
  Java_org_tzi_use_monitor_adapter_clr_CLRAdapter_isCLRAdapterInitialized
    (JNIEnv *, jobject);

/*
 * Class:     org_tzi_use_monitor_adapter_clr_CLRAdapter
 * Method:    isCLRClassType
 * Signature: (Lorg/tzi/use/plugins/monitor/vm/mm/clr/CLRType;)Z
 */
JNIEXPORT jboolean JNICALL
  Java_org_tzi_use_monitor_adapter_clr_CLRAdapter_isCLRClassType
    (JNIEnv *, jobject, jobject);

/*
 * Class:     org_tzi_use_monitor_adapter_clr_CLRAdapter
 * Method:    getCLRSuperClasses
 * Signature: (Lorg/tzi/use/plugins/monitor/vm/mm/clr/CLRType;)
 * Ljava/util/Set;
 */
JNIEXPORT jobject JNICALL
  Java_org_tzi_use_monitor_adapter_clr_CLRAdapter_getCLRSuperClasses
    (JNIEnv *, jobject, jobject);

/*
 * Class:     org_tzi_use_monitor_adapter_clr_CLRAdapter
 * Method:    getCLRSubClasses
 * Signature: (Lorg/tzi/use/plugins/monitor/vm/mm/clr/CLRType;)
```

```
 * Ljava/util/Set;
 */
JNIEXPORT jobject JNICALL
  Java_org_tzi_use_monitor_adapter_clr_CLRAdapter_getCLRSubClasses
    (JNIEnv *, jobject, jobject);

/*
 * Class:     org_tzi_use_monitor_adapter_clr_CLRAdapter
 * Method:    getNumOfInstances
 * Signature: ()I
 */
JNIEXPORT jint JNICALL
  Java_org_tzi_use_monitor_adapter_clr_CLRAdapter_getNumOfInstances
    (JNIEnv *, jobject);

/*
 * Class:     org_tzi_use_monitor_adapter_clr_CLRAdapter
 * Method:    getNumOfTypes
 * Signature: ()I
 */
JNIEXPORT jint JNICALL
  Java_org_tzi_use_monitor_adapter_clr_CLRAdapter_getNumOfTypes
    (JNIEnv *, jobject);

/*
 * Class:     org_tzi_use_monitor_adapter_clr_CLRAdapter
 * Method:    getNumOfModules
 * Signature: ()I
 */
JNIEXPORT jint JNICALL
  Java_org_tzi_use_monitor_adapter_clr_CLRAdapter_getNumOfModules
    (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Literaturverzeichnis

[13a] *CodePlex – Family Lines*. Webseite. 2013. URL: `http://familylines.codeplex.com/` (besucht am 20. 06. 2013).

[13b] *IKVM.NET Home Page*. Webseite. 2013. URL: `http://www.ikvm.net/` (besucht am 26. 06. 2013).

[13c] *The UML-based Specification Environment*. Webseite. 2013. URL: `http://sourceforge.net/apps/mediawiki/useocl/index.php?title=The_UML-based_Specification_Environment` (besucht am 02. 06. 2013).

[Avi+10] Carmen Avila u. a. »Runtime Constraint Checking Approaches for OCL, A Critical Comparison«. In: *SEKE*. 2010.

[BG11] Fabian Büttner und Martin Gogolla. »Modular Embedding of the Object Constraint Language into a Programming Language«. In: *Formal Methods, Foundations and Applications*. Hrsg. von Adenilso Simao und Carroll Morgan. Bd. 7021. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, S. 124–139. ISBN: 978-3-642-25031-6.

[BS02] D. Box und C. Sells. *Essential. NET: The common language runtime*. Bd. 1. Addison-Wesley Professional, 2002. ISBN: 0-201-73411-7.

[Cha+06] Patrice Chalin u. a. »Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2«. In: *Formal Methods for Components and Objects*. Hrsg. von Frank S. Boer u. a. Bd. 4111. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, S. 342–363. ISBN: 978-3-540-36749-9.

[Col+13] Adrian Colyer u. a. *The AspactJ Project*. Webseite. 2013. URL: `http://www.eclipse.org/aspectj/` (besucht am 02. 06. 2013).

[Fal12] Noah Falk. *Problems reading ICorDebugStringValue's returned from ICorDebugProcess5*. 2012. URL: `http://social.msdn.microsoft.com/Forums/en-US/netfxtoolsdev/thread/771ed9f3-b00e-49bd-a6d0-68057a53da33` (besucht am 17. 06. 2013).

[Fro+07] Lorenz Froihofer u. a. »Overview and Evaluation of Constraint Validation Approaches in Java«. In: *Proc. of ICSE '07*. Washington, DC, USA: IEEE Computer Society, 2007, S. 313–322. ISBN: 0-7695-2828-7.

[Gam+09] Erich Gamma u. a. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Professionelle Softwareentwicklung. Addison Wesley Verlag, 2009. ISBN: 9783827328243.

[GBR03] Martin Gogolla, Jörn Bohling und Mark Richters. »Validation of UML and OCL Models by Automatic Snapshot Generation«. In: *Proc. 6th Int. Conf. Unified Modeling Language (UML'2003)*. Hrsg. von Grady Booch, Perdita Stevens und Jonathan Whittle. Springer, Berlin, 2003.

[GBR07] Martin Gogolla, Fabian Büttner und Mark Richters. »USE: A UML-Based Specification Environment for Validating UML and OCL«. In: *Science of Computer Programming* 69 (2007), 27–34.

[Gor98] Rob Gordon. *Essential JNI: Java Native Interface*. Prentice-Hall, Inc., 1998. ISBN: 0136798950.

[Ham+12] Lars Hamann u. a. »Abstract Runtime Monitoring with USE«. In: *Proc. CSMR 2012*. 2012, S. 549–552.

[HGH12] Lars Hamann, Martin Gogolla und Daniel Honsel. »Towards supporting multiple execution environments for UML/OCL models at runtime«. In: *Proceedings of the 7th Workshop on Models@run.time*. MRT '12. Innsbruck, Austria: ACM, 2012, S. 46–51. ISBN: 978-1-4503-1802-0.

[HGK11] Lars Hamann, Martin Gogolla und Mirco Kuhlmann. »OCL-Based Runtime Monitoring of JVM Hosted Applications«. In: *Proc. WS OCL and Textual Modelling*. ECEASST. 2011.

[HHG12] Lars Hamann, Oliver Hofrichter und Martin Gogolla. »OCL-Based Runtime Monitoring of Applications with Protocol State Machines«. In: *ECMFA*. 2012, S. 384–399.

[Kap13] Arseny Kapoulkine. *pugixml – Light-weight, simple and fast XML parser for C++ with XPath support*. Webseite. 2013. URL: `http://code.google.com/p/pugixml/` (besucht am 12. 06. 2013).

[Koe05] Peter Koen. *Verwalteter Code hinter den Kulissen*. Website. Aug. 2005. URL: `http://msdn.microsoft.com/de-de/library/bb978892.aspx` (besucht am 13. 05. 2013).

[Lea13] Gary T. Leavens. *The Java Modeling Language (JML)*. Webseite. 2013. URL: `http://www.eecs.ucf.edu/~leavens/JML//index.shtml` (besucht am 02. 06. 2013).

[LL07] Jochen Ludewig und Horst Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. Heidelberg: dpunkt, 2007. ISBN: 3898642682.

[Mic12] Microsoft. *Standard ECMA-335: Common Language Infrastructure (CLI)*. Juni 2012. URL: `http://www.ecma-international.org/publications/standards/Ecma-335.htm` (besucht am 13. 05. 2013).

[Mic13a] Microsoft. *CLR Managed Debugger (mdbg) Sample 4.0*. Online. 2013. URL: `http://www.microsoft.com/en-us/download/details.aspx?id=2282` (besucht am 22. 05. 2013).

[Mic13b] Microsoft. *Debugging a Runtime Process*. Webseite. 2013. URL: `http://msdn.microsoft.com/en-us/library/bb384636` (besucht am 15. 06. 2013).

[Mic13c] Microsoft. *Einführung in die Programmiersprache C# und in .NET Framework*. Website. 2013. URL: `http://msdn.microsoft.com/de-de/library/z1zx9t92(v=vs.110).aspx` (besucht am 14. 05. 2013).

[Mic13d] Microsoft. *Garbage Collection in the Profiling API*. Webseite. 2013. URL: `http://msdn.microsoft.com/en-us/library/bb384688.aspx` (besucht am 26. 06. 2013).

[Mic13e] Microsoft. *LINQ (Language-Integrated Query, sprachintegrierte Abfrage)*. Webseite. 2013. URL: `http://msdn.microsoft.com/de-de/library/vstudio/bb397926.aspx` (besucht am 19. 05. 2013).

[Mic13f] Microsoft. *Microsoft Research – Spec#*. Webseite. 2013. URL: `http://research.microsoft.com/en-us/projects/specsharp/` (besucht am 02. 06. 2013).

[Mic13g] Microsoft. *MSDN – Code Contracts*. Webseite. 2013. URL: `http://msdn.microsoft.com/en-us/library/dd264808.aspx` (besucht am 02. 06. 2013).

[Mic13h] Microsoft. *.NET Framework 4.5*. 2013. URL: `http://msdn.microsoft.com/en-us/library/w0x726c2.aspx` (besucht am 13. 05. 2013).

[Mic13i] Microsoft. *.NET Framework-Versionen und -Abhängigkeiten*. 2013. URL: `http://msdn.microsoft.com/de-de/library/vstudio/bb822049.aspx` (besucht am 13. 05. 2013).

[Mic13j] Microsoft. *Unmanaged API Reference*. Website. 2013. URL: `http://msdn.microsoft.com/en-us/library/ch59zxfc.aspx` (besucht am 15. 05. 2013).

[OMG10] OMG. *Object Constraint Language 2.2*. Feb. 2010. URL: `http://www.omg.org/spec/OCL/2.2/` (besucht am 31. 05. 2013).

[OMG13] OMG. *Unified Modeling Language™ (UML)*. Webseite. 2013. URL: `http://www.omg.org/spec/UML/` (besucht am 28. 05. 2013).

[Ora11] Oracle. *Java™ Platform Debugger Architecture - Structure Overview*. 2011. URL: `http://download.oracle.com/javase/6/docs/technotes/guides/jpda/architecture.html`.

[Ora12] Oracle. *Java™ Native Interface*. 2012. URL: `http://docs.oracle.com/javase/1.4.2/docs/guide/jni/` (besucht am 24. 07. 2012).

[Ora13] Oracle. *The Java Tutorials – Primitive Data Types*. Webseite. 2013. URL: `http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html` (besucht am 22. 07. 2013).

[Pel02] Mike Pellegrino. *Improve Your Understanding of .NET Internals by Building a Debugger for Managed Code*. Webseite. 2002. URL: `http://msdn.microsoft.com/en-us/magazine/cc301510.aspx` (besucht am 05. 12. 2012).

[Pos13] PostSharp. *AOP on .NET – PostSharp*. Webseite. 2013. URL: `http://www.postsharp.net/aop.net` (besucht am 02. 06. 2013).

[Pra09] S. Pratschner. *Customizing the Microsoft®. NET Framework Common Language Runtime*. Microsoft Press, 2009. ISBN: 0-7356-1988-3.

[Ric10] Jeffrey Richter. *CLR via C#*. 3. Aufl. Redmond und WA: Microsoft Press, 2010. ISBN: 978-0-7356-2704-8.

[RJB05] James Rumbaugh, Ivar Jacobson und Grady Booch. *The Unified Modeling Language reference manual: [covers UML 2.0]*. 2. ed. The Addison-Wesley object technology series. Boston [u.a.]: Addison-Wesley, 2005. ISBN: 0321245628.

[RQZ07] Chris Rupp, Stefan Queins und Barbara Zengler. *UML 2 glasklar: Praxiswissen für die UML-Modellierung*. 3., aktualisierte Aufl. München [u.a.]: Hanser, 2007. ISBN: 3446411186.

[Sch05] Reinhard Schiedermeier. *Programmieren mit Java – Eine methodische Einführung ; [Java Version 5.0]*. Informatik, Programmierung. XVI, 452 S. ; 240 mm x 165 mm : graph. Darst. München [u.a.]: Pearson Studium, 2005. ISBN: 3827371163.

[Shu05] Jon Shute. *How the .NET Debugger Works*. Webseite. Apr. 2005. URL: `http://www.developerfusion.com/article/4692/how-the-net-debugger-works/` (besucht am 15. 06. 2013).

[Sta13] Mike Stall. *Mike Stall's .NET Debugging Blog*. Webseite. 2013. URL: `http://blogs.msdn.com/b/jmstall/` (besucht am 15. 06. 2013).

[STZ10] Holger Schwichtenberg, Stefan Toth und Stefan Zörner. »Aus der Vogelperspektive«. In: *iX – Magazin für professionelle Informationstechnik* 4 (2010), 62–72.

[Wie12] Thomas Wieland. *Strategievergleich: .NET versus Java*. Online. 2012. URL: `http://www.cpp-entwicklung.de/downld/Strategievergleich.pdf` (besucht am 18. 05. 2013).

[Xam13a] Xamarin. *Mono Projekt*. Website. 2013. URL: `http://www.mono-project.com/` (besucht am 14. 05. 2013).

[Xam13b] Xamarin. *Mono Soft-Mode Debugger*. Webseite. 2013. URL: `http://www.mono-project.com/Mono:Runtime:Documentation:SoftDebugger` (besucht am 26. 06. 2013).

[ZGK04] Wolfgang Zuser, Thomas Grechenig und Monika Köhle. *Software Engineering mit UML und dem Unified Process*. 2., überarb. Aufl. München [u.a.]: Pearson Studium, 2004. ISBN: 3827370906.

Ehrenwörtliche Erklärung

Ich erkläre hiermit, dass ich die vorliegende Diplomarbeit mit dem Titel

Technologieübergreifende Verifikation von Laufzeit-Annahmen mit UML- und OCL-Modellen

selbständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Ich versichere außerdem, dass die vorliegende Arbeit noch nicht einem anderen Prüfungsverfahren zugrunde gelegen hat.

Bremen, den 5. August 2013

Daniel Honsel