Санкт-Петербургский Государственный Университет
Математико-механический факультет

Кафедра системного программирования

Разработка декомпилятора языка Java SE 6

Курсовая работа студента 345 группы
Михайлова Дмитрия Петровича

Научный руководитель Шафиров М.Г.

Санкт-Петербург
2011

Оглавление

Введение

Процесс декомпиляции является важной, а при решении некоторых задач (таких, как поддержка программного обеспечения без возможности использования исходного кода или обратная разработка) и неотъемлемой, частью разработки программного обеспечения.

Быстрое развитие и практически полная обратная совместимость платформы Java SE позволяет обновлять виртуальную машину и переходить на новые версии языка и стандартных библиотек непосредственно во время разработки приложения, что способствует распространению новых языковых особенностей и конструкций в старых проектах и даже в унаследованном коде. Именно поэтому возможность декомпиляции в актуальные версии языка Java SE весьма востребована.

1 Постановка задачи

Наиболее мощными и значимыми особенностями современного языка Java SE являются добавленные в версии J2SE 5.0 [1] параметризованные типы [2] и аннотации (метаданные) [3].

Возможность корректной декомпиляции параметризованных типов в ряде случаев (к примеру, когда после компиляции отсутствует информация о параметрах типов возвращаемых методами значений) значительно облегчила бы взаимодействие со сторонними библиотеками с отсутствующей или неполной документацией, что является одним из ключевых случаев использования декомпиляции [4] и в некоторых странах разрешено законодательно [5].

Отображение аннотаций может в большой степени упростить понимание алгоритма работы декомпилированной программы и её бизнес-логики. При использовании таких фреймворков как Spring, она может быть практически полностью описана метаданными [6] и не быть в явном виде выражена в коде.

2 Обзор существующих решений

На данный момент существует несколько основных программных продуктов, обладающих функциональностью декомпилятора языка Java:

2.1 Популярные декомпиляторы

2.1.1 Mocha

Mocha (автор — Hanpeter van Vliet) — это, вероятно, один из первых выпущенных декомпиляторов Java. Предоставляет консольный пользовательский интерфейс. Его релиз состоялся в 1996-ом году, ещё до того, как появился Java Development Kit версии 1.1, поэтому говорить о поддержке в нём каких-либо особенностей Java 6 не приходится. К его недостаткам так же стоит отнести тот факт, что доступна только версия Mocha для Windows, которую можно скачать с сайта декомпилятора [7].

Известность же и популярность Mocha среди Java-разработчиков, помимо возраста декомпилятора, могут быть объяснены тем, что на нём основан декомпилятор, встроенный в среду разработки Embarcadero® JBuilder® (ранее — Borland® JBuilder®).

2.1.2 JAva Decompiler

JAva Decompiler, JAD (автор — Pavel Kouznetsov) — по всей видимости, самый популярный декомпилятор Java. Как и Mocha, этот декомпилятор предоставляет консольный интерфейс, давно не обновляется и не поддерживается, но большое число графических инструментов для работы с ним, в том числе плагин JadClipse для среды разработки Eclipse, делают его и по сей день используемым в качестве подручного средства для декомпиляции небольших классов.

Помимо декомпиляции, JAva Decompiler обладает возможностью дизассемблирования .class-файлов.

Срок аренды доменного имени официального сайта истёк, но можно скачать последнюю выпущенную версию для разных платформ с зеркала [8].

2.1.3 DJ Java Decompiler

DJ Java Decompiler (автор — Atanas Neshkov) — долгое время вопреки названию являлся лишь графической оболочкой для предыдущего декомпилятора, позволявшей легко и удобно выбрать аргументы командной строки для вызова JAD.

В текущей версии добавлена поддержка аннотаций, но декомпилятор стал условно-бесплатным (необходима покупка после 10 пробных использований).

Внешний вид графического пользовательского интерфейса[1]:

[1]Скриншоты взяты с официального сайта декомпилятора [9]

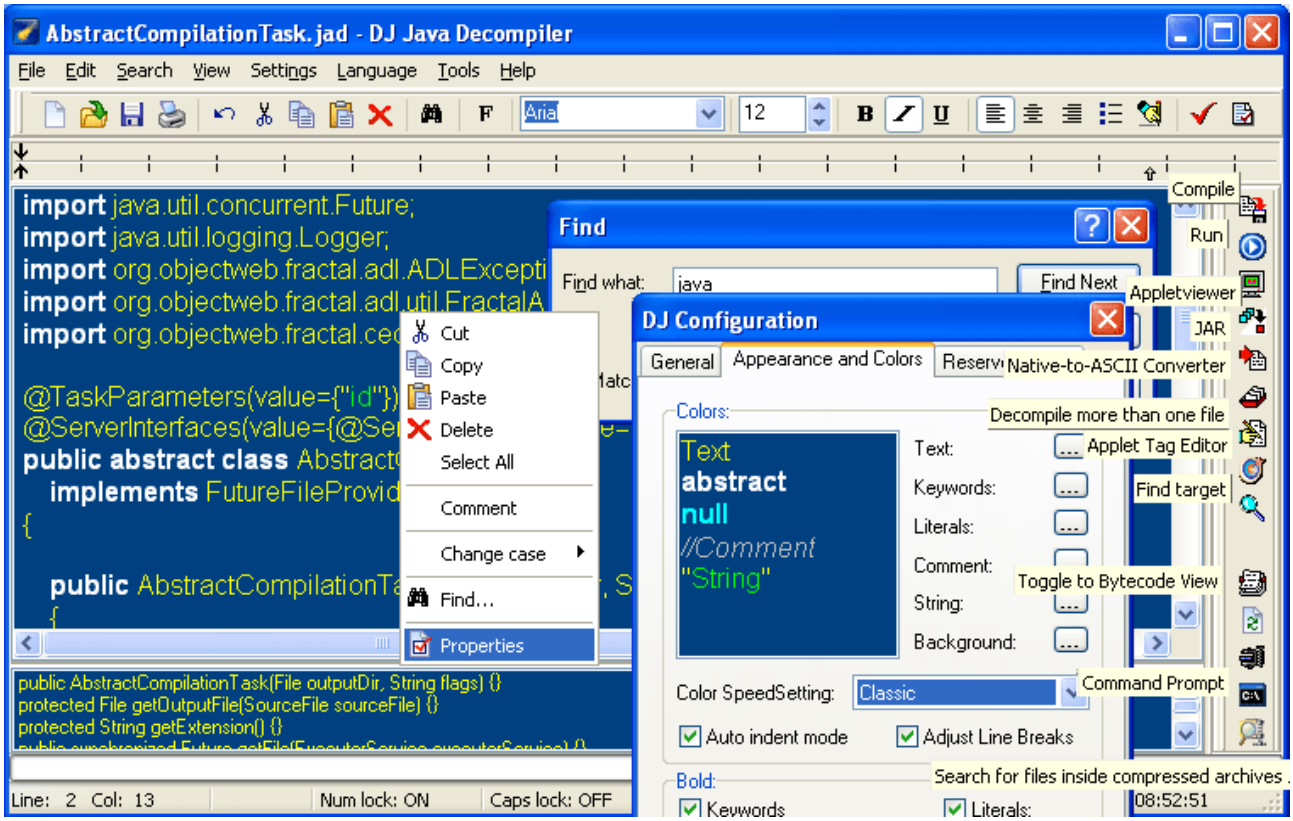

Рис. 1: Общий вид интерфейса

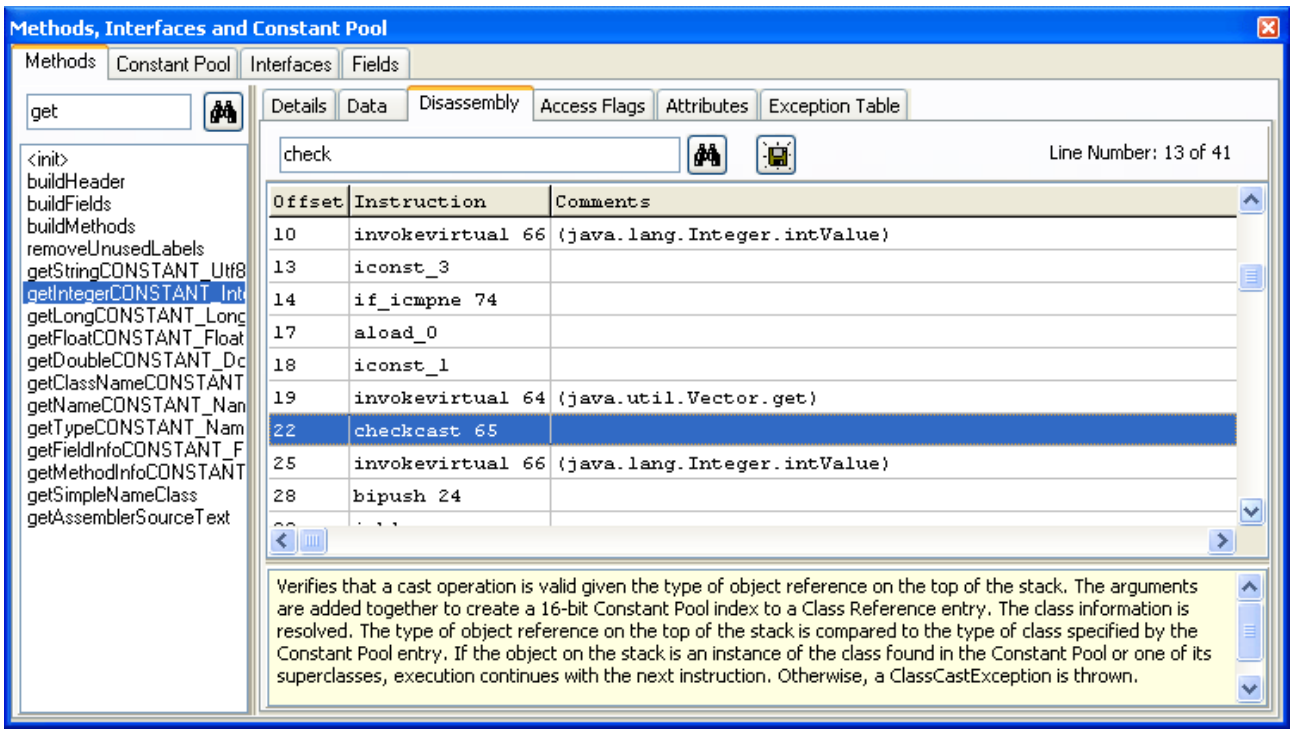

Рис. 2: Дизассемблирование

2.2 JD-Core

JD-Core (автор — Emmanuel Dupuy) — очень мощная и функциональная библиотека для декомпиляции и анализа байткода Java, разработанная в рамках «Java Decompiler project».

Имеет следующие особенности:

- Полностью написана на языке C++, что делает декомпиляцию необычайно быстрой
- Не требует для работы Java Runtime Environment и поэтому не требует специальной установки
- Корректно декомпилирует .class-файлы, сгенерированные большинством компиляторов
- Поддерживает следующие возможности Java 5:
  - Генерики
  - Аннотации
  - Перечислимые типы

Пожалуй, к недостаткам JD-Core можно отнести лишь то, что она распространяется как часть самостоятельного графического приложения JD-GUI, также разработанного на C++ и прилинкованного к ней статически, или плагина JD-Eclipse для среды разработки Eclipse, что делает практически невозможным её использование в стороннем некоммерческом проекте, особенно разработанном на языке Java. Использование библиотеки в коммерческих программных продуктах запрещено автором.

Внешний вид приложения JD-GUI[2]:

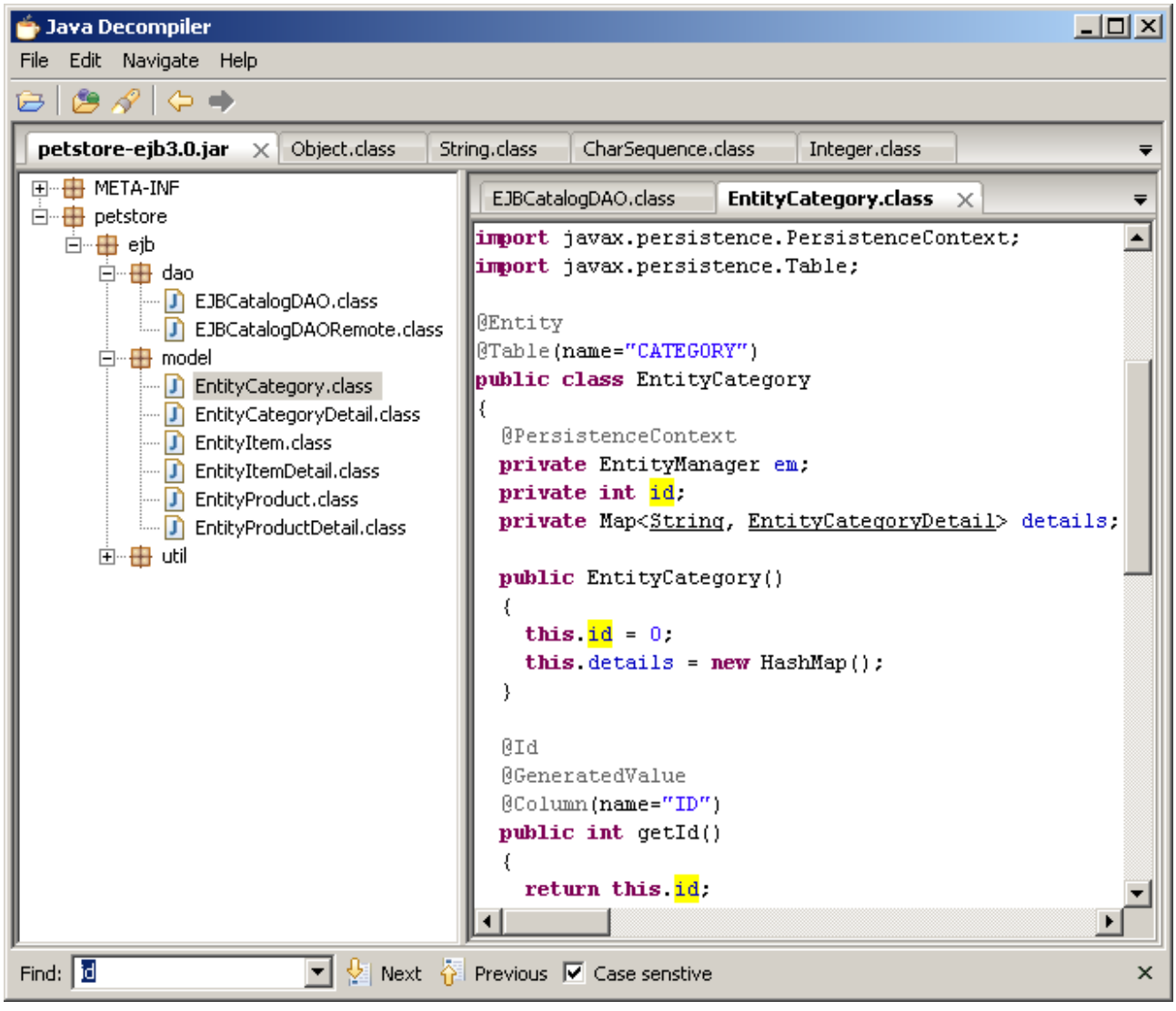

Рис. 3: Демонстрация декомпиляции аннотаций и генериков

2.3 Fernflower

Fernflower — один из лучших декомпиляторов языка программирования Java на сегодняшний день.

Обладает следующими возможностями:

- Поддерживает разнообразные языковые конструкции:
  - Параметрические типы
  - Аннотации
  - Перечислимые типы
  - Утверждения
- Корректно декомпилирует байткод, сгенерированный вследствие некоторых известных багов компиляторов
- Частично деобфусцирует код

Единственным существенным недостатком Fernflower являеется тот факт, что он доступен исключительно в виде веб-интерфейса на сайте разработчика [11], что вызывает определённые трудности с его использованием:

- Необходимость подключения к Интернету
- Большая длительность декомпиляции
- Сложность использования Fernflower из кода
- Правовые трудности, связанные с невозможностью передачи копии бинарного файла третьим лицам

Внешний вид интерфейса:

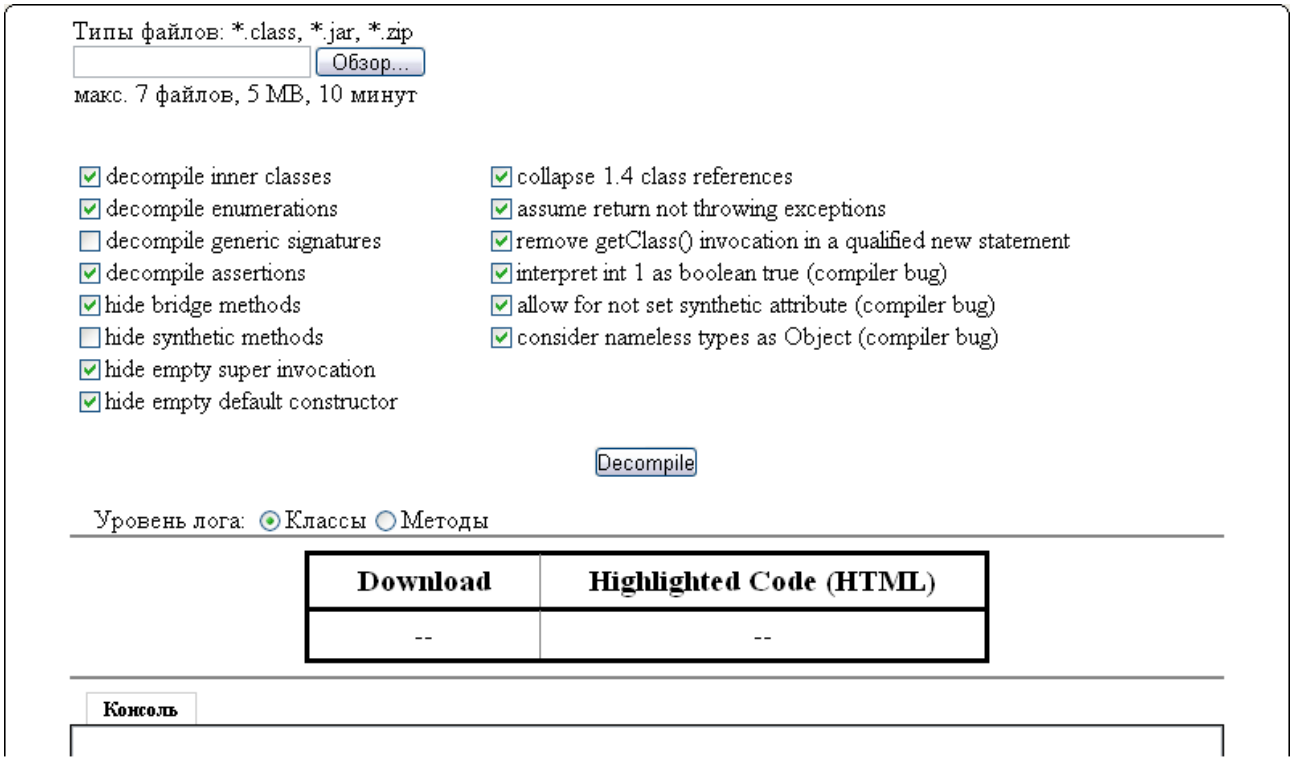

Рис. 4: Веб-интерфейс декомпилятора Fernflower

3 Реализация

Разработка велась с использованием библиотек ObjectWeb ASM для парсинга метаданных в .class-файлах и Sable Research Group Soot для построения промежуточных представлений методов и генерации кода из них.

3.1 ObjectWeb ASM

ObjectWeb ASM [12] — многофункциональная библиотека, предоставляющая мощный интерфейс для манипулирования байткодом вплоть до генерации классов во время исполнения и их последующей загрузки.

Одна из версий ASM включена в Java Development Kit 6 в пакете под названием com.sun.xml.internal.ws.org.objectweb.asm.

Библиотека состоит из двух API схожей функциональности, первое из которых основано на использовании паттерна Visitor для обхода всех элементов классов, а второе предоставляет представление полученных им данных в виде дерева [13].

При разработке декомпилятора применялся второй вариант API, поскольку перед генерацией кода необходимо построить полное представление входного класса.

К использованным возможностям ObjectWeb ASM следует отнести:

- Парсер заголовков бинарных файлов с целью извлечения метаданных
- Парсер сигнатур параметризованных типов (TraceSignatureVisitor) [14]:

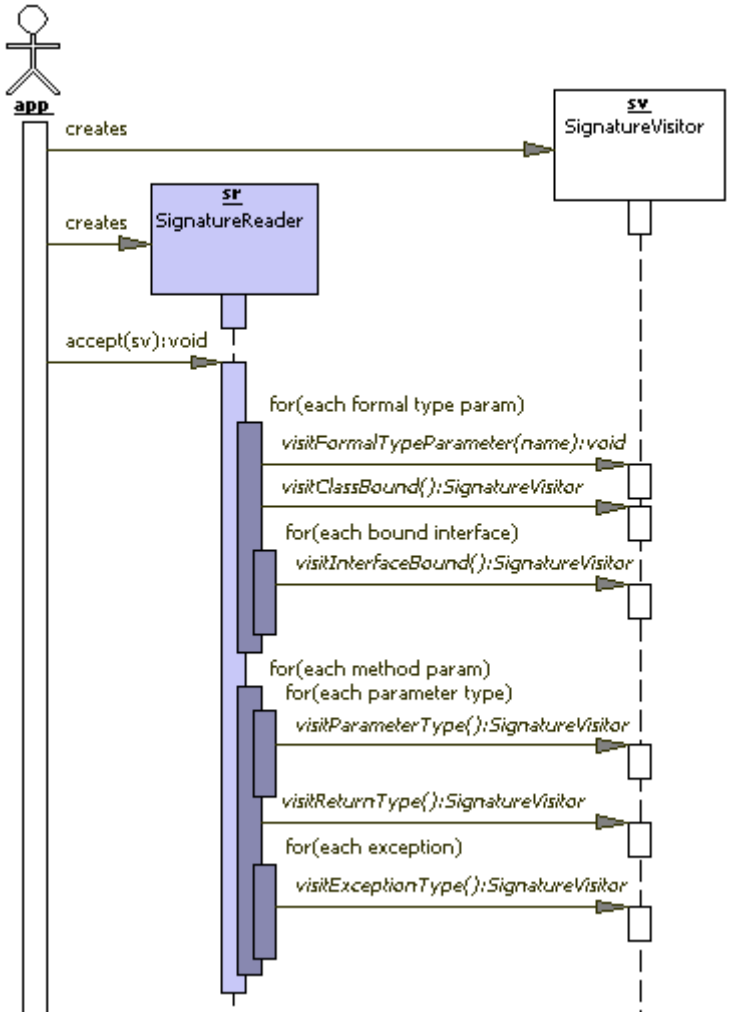

Рис. 5: Схема последовательности вызовов методов при разборе сигнатуры

3.2 Soot

Soot [15] — фреймворк для анализа байткода и его преобразований, содержащий несколько различных промежуточных представлений, и одновременно с этим консольный инструмент для осуществления этих преобразований.

Предоставляет возможность последовательного преобразования байткода в более высокоуровневые представления [16] и добавления произвольных пользовательских трансформаций на любом этапе [17]:

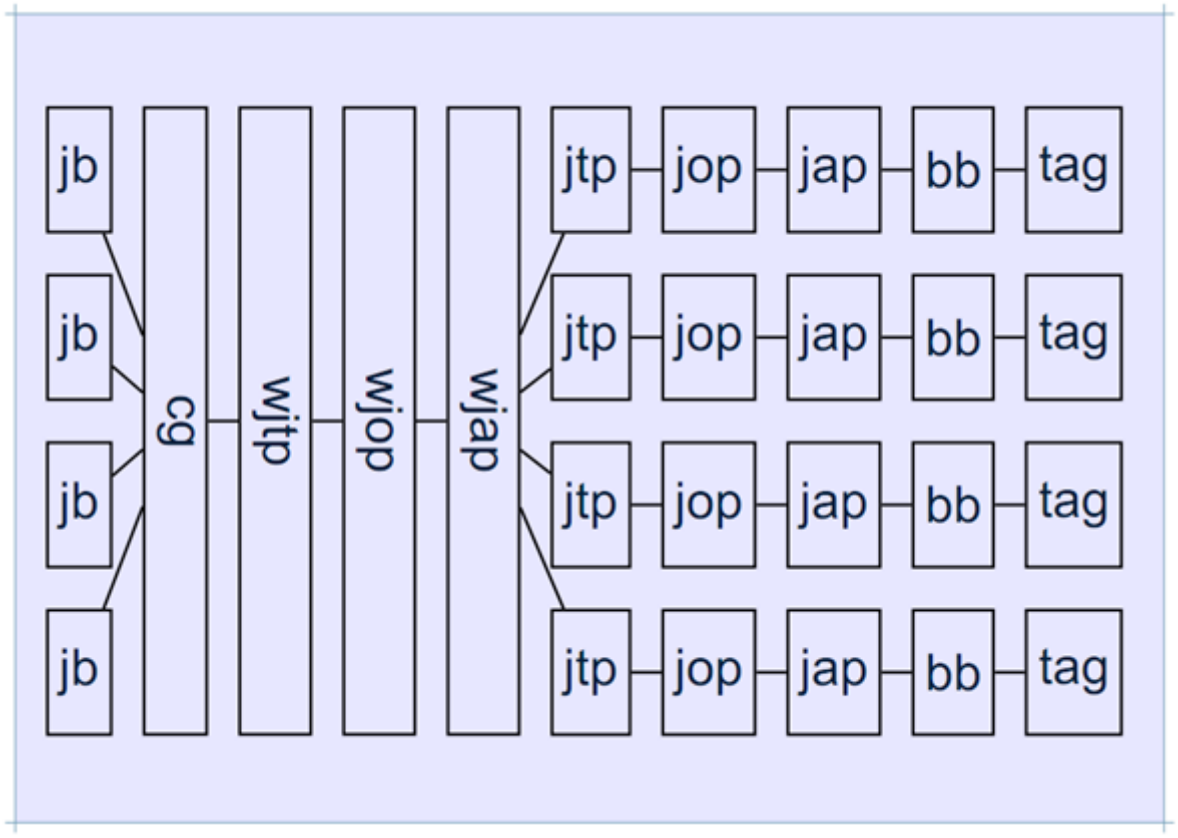

Рис. 6: Последовательность выполнения преобразований тел методов в Soot

Для демонстрации возможностей фреймворка в Soot добавлен созданный на его основе декомпилятор Dava. Он соответствует классическому подходу к разработке декомпиляторов, основанному на построении графа базовых блоков и его анализе [18].

Документировано только использование Dava в качестве консольного инструмента, поэтому для выяснения способа его правильного использования как библиотеки потребовалось обратиться к почтовой рассылке Soot [19]. Dava был использован в качестве ядра декомпилятора для генерации кода конкретных (не-абстрактных) методов.

3.3 Процесс декомпиляции

Процесс декомпиляции .class-файла разбит на два последовательных этапа:

1. Построение полного промежуточного представления класса и всех его членов, включая внутренние классы
2. Генерация кода из построенного на предыдущем этапе представления

3.3.1 Построение промежуточного представления

Первый этап работы начинается с построения дерева метаданных при помощи библиотеки ASM. Для узлов дерева метаданных, соответствующих классам, полям и методам созданы агрегирующие их классы, которые самостоятельно (посредством использования шаблона проектирования Template method в конструкторе их общего абстрактного предка) устанавливают особенности их описания в коде и модификаторы доступа в соответствии с предполагаемой актуальной спецификацией виртуальной машины Java [20]. Именно из этих классов в конечном итоге будет построено дерево представления класса.

Помимо полученных из ASM метаданных, экземпляры класса метода, соответствующие конкретным методам классов, на этом этапе агрегируют в себе и промежуточные представления методов из библиотеки Soot, прошедшие весь стек преобразований, необходимых для генерации их кода, включая преобразования Dava, но сохранившие последнее абстрактное представление, непосредственно из которого может быть сгенерирован код.

Для корректного отображения типов возвращаемого значения и аргументов метода в случае, если некоторые из них окажутся параметризованными, предусмотрен wrapper для представления типов из Soot, делегирующий всю логику экземпляру типа внутри, но использующий свой метод toString(), печатающий название указанным способом с генериками. Для получения корректного строкового отображения типа использован парсер параметризованных сигнатур из ASM, а для принятия решения о «заворачивании» типа и выдачи получившейся «обёртки», если она нужна, и исходного типа, если «обёртка» не требуется, реализована специальная фабрика.

Декомпиляция внутренних классов производится рекурсивным обходом по дереву вложенности в глубину в предположении, что .class-файлы внутренних классов находятся в той же директории (которое часто оказывается верно, поскольку внутренние классы, очевидно, относятся к тому же пакету). Для хранения разобранных классов и проверки на то, что класс встретился в первый раз, используется хешированный словарь.

Для получения правильного дерева вложенности классов при каждой попытке спуска по ребру делается проверка, что вершина на ранг выше указана у вершины ниже, как внешний класс. Обусловлено это тем, что генерируемые стандартным компилятором Java синтетические классы могут быть указаны как внутрение сразу у нескольких классов и даже сами у себя, но внешний класс всегда указан один и всегда верно.

На демонстрирующей это схеме чёрными квадратами обозначены пользовательские классы, фиолетовым квадратом — синтетический класс, стрелка с красным началом означается физическое содержание описания одного класса внутри описания другого, стрелки с зелёным началом означают, что класс содержит другой в списке внутренних, а стрелки с синим началом — что класс-конец указан у класса-начала, как внешний:

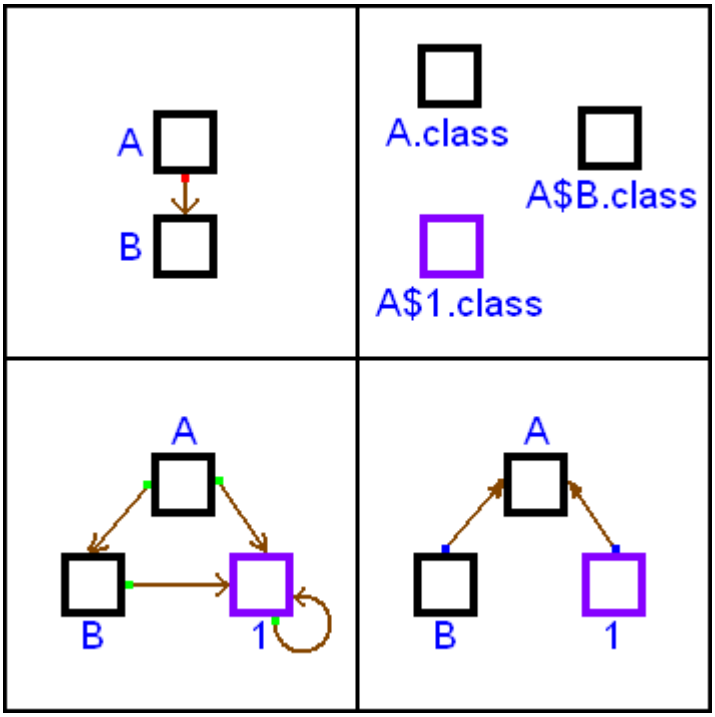

Рис. 7: Схема некорректных пометок о внутренних классах

3.3.2 Генерация кода из промежуточного представления

Генерация кода осуществляется рекурсивным обходом дерева промежуточного представления в глубину, поскольку иерархия этого дерева в точности соответствует вложенности описаний сущностей в исходном коде, а про каждый объект верно, что код из него должен начать генерироваться раньше, чем из всех содержащихся в нём, а закончить после окончания генерации из них.

Алгоритм генерации кода из каждого конкретного объекта выбирается внутри него из соображений информации о способе его описания (фактически, эта информация представляет собой набор флагов, выставленный с учётом логики битовых флагов в заголовке .class-файла, но не повторяющий его в силу неудобства последнего), полученной в начале предыдущего этапа, и информации о способе описания внешнего объекта, для чего в дереве хранятся обратные рёбра (рёбра «снизу вверх»).

Для генерации описаний классов, интерфейсов и аннотаций вновь используется парсер сигнатур из ASM, осуществляющий строковое преобразование сигнатуры из того вида, в котором она хранится в бинарном файле, в тот, в котором она должна быть в исходном коде. Он же используется при генерации типов полей.

Для генерации аннотаций реализовано несколько отдельных классов со статическими методами, поскольку она, фактически, происходит независимо от генерации как того объекта, к которым аннотации относятся, так и его внешнего объекта, если он существует.

Заключение

Результаты

На данный момент разработан прототип декомпилятора, доказывающий возможность однозначного распознавания в скомпилированных классах без эвристических алгоритмов использования современных особенностей языка Java SE 6 и последующей генерации соответствующего им кода.

К поддерживаемым в текущей версии возможностям языка относятся:

- Параметризованные типы в:
  - Описаниях классов
  - Сигнатурах методов
  - Возвращаемых значениях методов
  - Бросаемых методами исключениях
  - Типах полей
- Аннотации:
  - Классов
  - Полей
  - Методов
  - Аргументов методов
- Описания пользовательских аннотаций

Декомпиляция этих возможностей языка не зависит от свойств использованного ядра декомпилятора, что предоставляет возможность его замены.

Дальнейшее развитие

Данная разработка может быть продолжена по следующим направлениям:

- Улучшение работы ядра декомпилятора:
  - Разработка собственного ядра с поддержкой параметризованных типов и современных языковых конструкций:
    - Циклов for-each
    - Перечислимых типов
    - Методов с переменным числом аргументов
    - Утверждений
  - Модификация или расширение библиотеки Soot:
    - Расширение системы типов и решение задачи о нахождении наиболее узкого общего предка для двух параметризованных типов (с целью добавления полной поддержки параметризованных типов)
    - Добавление новых преобразований кода с целью улучшения его качества и читаемости, а также поддержки высокоуровневых языковых конструкций
- Повышение качества генерации кода параметрических типов и аннотаций:
  - Отказ от полных имён типов:
    - Аккумулирование общей таблицы импорта
    - Реализация проверки на совпадение имён в разных пакетах
  - Использование «синтаксического сахара» (к примеру, при присвоении элементу аннотации типа коллекция коллекции из одного элемента, можно присвоить сам этот элемент)
- Исследование возможности декомпиляции новых особенностей Java 7 [21]:
  - Строковые аргументы в switch-конструкциях
  - Выражение try-with-resources
  - Multicatch и rethrow для исключений

Список литературы

1. http://www.oracle.com/technetwork/java/javase/compatibility-137541.html
2. JSR 14, http://www.jcp.org/en/jsr/detail?id=14
3. JSR 175, http://www.jcp.org/en/jsr/detail?id=175
4. http://www.program-transformation.org/Transform/WhyDecompilation
5. Directive 2009/24/EC of the European Parliament and of the Council of 23 April 2009
6. Seema Richard, Annotation based configuration in Spring
7. http://www.brouhaha.com/~eric/software/mocha/
8. http://www.varaneckas.com/jad
9. http://members.fortunecity.com/neshkov/dj.html
10. http://java.decompiler.free.fr/
11. http://www.reversed-java.com/fernflower/
12. http://asm.ow2.org/
13. http://download.forge.objectweb.org/asm/asm-guide.pdf
14. http://asm.ow2.org/doc/tutorial-asm-2.0.html
15. http://www.sable.mcgill.ca/soot/
16. http://www.sable.mcgill.ca/soot/tutorial/phase/index.html
17. Eric Bodden, Packs and phases in Soot
18. http://www.backerstreet.com/decompiler/decompiler_main.htm
19. http://www.sable.mcgill.ca/mailman/listinfo/soot-list/
    Отдельную благодарность хочется выразить Laurie Hendren (Professor, School of Computer Science, McGill University) и Nomair A. Naeem (PhD Candidate, Programming Languages Group, School of Computer Science, University of Waterloo)
20. Draft of the Java VM Specification, Third Edition
21. http://openjdk.java.net/projects/coin/