ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет Санкт-Петербургская школа
физико-математических и компьютерных наук

Смирнов Игорь Андреевич

РАЗРАБОТКА И РЕАЛИЗАЦИЯ МОДЕЛИ ДЛЯ ОБРАБОТКИ АВТОМАТИЧЕСКИ СГЕНЕРИРОВАННЫХ ОТЧЕТОВ ОБ ОШИБКАХ

Выпускная квалификационная работа - БАКАЛАВРСКАЯ РАБОТА
по направлению подготовки *01.03.02 Прикладная математика и информатика*
образовательная программа «Прикладная математика и информатика»

Рецензент
А.В. Черняев

Руководитель
д-р. физ.-мат. наук, проф.
Б.А. Новиков

Консультант
Р.А. Васильев

Санкт-Петербург 2021

Оглавление

Системы мониторинга ошибок оперируют с разными видами отчётов: исключения из различных языков программирования, зависания пользовательского интерфейса, падения (сбои) виртуальных машин. Для добавления обработки нового типа отчётов требуется писать новый код. В то же время, большая часть обработки ошибки может быть выполнена языконезависимо (сравнение трассировок стека, кластеризация ошибок, поиск по ошибкам, генерация представления, навигация по коду и другие части обработки), потому что многие алгоритмы не зависят от происхождения ошибки. В современных системах есть языконезависимые модели для исключений, но их подход плохо обобщается на случай нелинейных структур ошибок. В этой работе представлена модель, которая может быть использована для языконезависимой обработки большинства известных видов отчётов и описана реализация всего пайплайна обработки отчётов в рамках этой модели в Exception Analyzer (внутренняя система мониторинга ошибок JetBrains). Также в работе показано, как языкоспецифичная логика может быть реализована в рамках модели.

Ключевые слова: модель отчёта, отслеживание ошибок, мониторинг ошибок, обработка отчётов, обработка сбоев, зависания пользовательского интерфейса

Error monitoring systems operate with different types of reports: exceptions from various programming languages, UI freezes, virtual machine crashes. Every new type of report requires writing new code. At the same time, most actions in report processing can be done in a language-agnostic way (stack trace comparison, errors' clustering, search, presentation generation, code navigation and others), because many algorithms do not depend on the origin of the error. Modern systems have language-agnostic models for exceptions, but their approach is not suitable for non-linear error structures. In this paper we present a model that can be used for language-agnostic processing of most popular report types and describe the process of implementing a full report processing pipeline within this model in the Exception Analyzer (JetBrains internal error monitoring system). We also show how language-dependent logic can be implemented within the framework of this model.

Keywords: report model, bug tracking, error monitoring, error processing, crash processing, UI freeze

Введение

Актуальность и релевантные работы

Отчёты об ошибках являются популярным средством для получения разработчиками обратной связи о работе программного обеспечения. Отчёты можно разделить на две категории: отправляемые вручную и сгенерированные автоматически.

Отправляемые вручную отчёты требуют много действий от пользователя: зайти в трекер ошибок, описать проблему, приложить файлы, которые помогут разработчикам определить проблему. У этого подхода есть несколько проблем: не все пользователи готовы совершать все эти действия [14] и часто пользователи не прикладывают достаточно информации для решения проблемы [13].

Поэтому существуют автоматически сгенерированные отчёты об ошибках. Для отправки такого отчёта пользователю требуется нажать несколько кнопок, а в отчёте содержится информация об ошибке, которую можно получить без участия пользователя. Таким способом отправляют исключения из разных языков программирования, сбои виртуальных машин, зависания пользовательского интерфейса, ошибки нехватки памяти с дампами памяти, отчёты профилировщиков и другие ошибки [14] .

Так как автоматически сгенерированных отчётов об ошибках очень много, требуется их автоматически предобрабатывать: группировать похожие, назначать ответственных за исправление и так далее. Эту работу выполняют системы мониторинга ошибок [3]. Exception Analyzer — внутренняя система мониторинга ошибок JetBrains. В Exception Analyzer сейчас поддерживаются исключения из языков для JVM[1], .NET[2] исключения, отчёты зависания пользовательского интерфейса, сбои JBR[3].

Сейчас в Exception Analyzer отчёты отправляются как текст ошибки

[1]JVM — Java Virtual Machine — виртуальная машина Java

[2].NET — кроссплатформенная платформа для разработчиков с открытым исходным кодом для создания различных типов приложений

[3]JBR — JetBrains Runtime — среда выполнения для запуска продуктов на базе платформы IntelliJ в Windows, Mac OS X и Linux

и вспомогательная информация (плагин, где произошла ошибка, версия, сообщение от пользователя и так далее). Поэтому, для добавления нового вида отчётов, программист должен с нуля писать код по их обработке. В то же время, многие алгоритмы по работе с отчётами языконезависимы [4, 12], а значит добавление новых видов отчётов можно упростить, если работать с каждым видом отчётов отдельно, а с языконезависимым представлением ошибки. Кроме того, при переносе формирования отчёта с серверной части на клиентскую, можно получать дополнительную информацию, которую невозможно получить по тексту отчёта (например, версии библиотек, которые использовались, когда произошла ошибка).

Такие модели успешно используются в популярных системах мониторинга ошибок: Sentry [10], Rollbar [7], Airbrake [1] и других. Однако модели в этих системах поддерживают только необработанные исключения, где ошибка — это последовательность кадров стека. В Exception Analyzer посылаются и ошибки с более сложной структурой и используются алгоритмы, которые требуют больше информации, чем та, что доступна в перечисленных выше системах.

Цель и задачи

Цель данной работы — разработать модель для языконезависимой обработки отчётов и внедрить её в Exception Analyzer.
Для этого ставятся следующие задачи:

- Разработать модель, в рамках которой возможно реализовать всю текущую функциональнось Exception Analyzer.
- Реализовать на сервере обработку отчётов в языконезависимом формате.
- Реализовать библиотеку-клиент для JVM-языков, с использованием которой возможна отправка отчётов в языконезависимом формате из IntelliJ IDEA.

Достигнутые результаты

В рамках данной работы предложена языконезависимая модель отчёта, поддерживающая исключения из, как минимум, 30 языков программирования, а также отчёты других видов, например, сбои виртуальных машин или зависания пользовательского интерфейса. Модель основана на модели для исключений из системы мониторинга Sentry, но при этом более гибкая и поддерживает обработку нескольких ошибок внутри отчёта.

В рамках модели был реализован языконезависимый пайплайн обработки отчётов в Exception Analyzer. Удалось поддержать всю имеющуюся функциональность, а также улучшить обработку сбоев JBR, которая ранее была неполной. В частности, скорость сравнения сбоев JBR увеличилась в 4-8 раз и стала доступна функциональность, которая раньше для них не поддерживалась (выделение важных элементов, визуализация сравнения).

Была реализована библиотека-клиент, позволяющая отправлять отчёты из JVM языков. Отправка отчётов с помощью этой библиотеки будет внедрена в одну из ближайших версий IntelliJ IDEA.

Структура работы

В главе 1 представлен обзор существующих моделей ошибок в системах мониторинга и рассмотрены алгоритмы для работы с ошибками.

В главе 2 описана разработанная модель.

В главе 3 пошагово описан процесс реализации языконезависимого пайплайна обработки отчётов в Exception Analyzer.

В главе 4 описана библиотека-клиент для JVM языков.

В заключении проанализирована проделанная работа и представлены дальнейшие планы по развитию модели.

1. Обзор литературы

Алгоритмы сравнения ошибок — самая наукоёмкая часть систем мониторинга. В этой главе описаны современные алгоритмы для сравнения ошибок, с целью сформировать требования к модели.

Также в данной главе представлены модели ошибок в популярных системах мониторинга, описаны их преимущества и недостатки.

1.1. Алгоритмы для сравнения ошибок

Большая часть алгоритмов в этой области создавались для работы с исключениями, поэтому оперируют трассировками стека и кадрами стека.

Чаще всего, алгоритмы рассматривают трассировку стека как строку, а кадры стека в трассировке как буквы, то есть неделимые символы, которые можно сравнивать между собой. Самый простой способ сравнения таких трассировок — по префиксу [2], но он не даёт хороших результатов. Другой подход, основанный на теории вероятности, предложили Brodie и другие в статье 2005 года [5]. В этой работе кадр стека — тоже неделимая сущность и важен порядок кадров в трассировке. В работе [2] было предложено использовать алгоритмы на строках для определения похожести ошибок, в частности, редакционное расстояние (расстояние Левенштейна). Ещё один алгоритм, который работает с трассировкой как со строкой — ReBucket [6].

Другая большая группа алгоритмов рассматривает трассировку стека как мешок слов (bag of words). Неделимыми словами всё так же являются кадры стека. Самый простой подход — измерить косинусное сходство между двумя трассировками [8]. К нему применяются различные модификации. Хороший результат обычно даёт использование TF-IDF, вместо единичных весов кадров стека. В работе Lerch [4] приведена модификация TF-IDF, которая даёт прирост результата по сравнению со стандартным TF-IDF.

Также используются сочетания этих идей. Например, в работе [12] представлен алгоритм, в котором считается взвешенное редакционное

расстояние, где весами является TF-IDF кадра стека.

Однако не всегда кадр стека считается неделимой сущностью. В той же статье [12] описывается подход, который сейчас используется в Exception Analyzer для обработки зависаний пользовательских исключений: IDF считается не по кадрам стека, а по пакетам, а сравниваются при этом всё ещё кадры стека. Интуиция такого распределения весов в том, что в одном пакете методы выполняют примерно одну и ту же работу и примерно одинаково важны. Кроме того, при рефакторинге методы часто перемещаются в другой файл, а с таким подсчётом весов IDF будет «стабильнее». Также это помогает уменьшить размер словаря для применения более сложных методов, что описано в статье DURFEX [9].

Для непоследовательных видов ошибок всё ещё применимы методы, рассматривающие кадры стека из ошибки как мешок слов. В частности, для обработки зависаний пользовательского интерфейса, ищется косинусное сходство между векторами кадров, где вес кадра определяется как логарифм времени этого кадра (аналог TF) помноженный на IDF пакета кадра.

Для определения похожести сбоев виртуальных машин, сравниваются их трассировки, а также можно сравнивать такую информацию как значения регистров.

1.2. Модели ошибок в системах мониторинга

Популярные системы мониторинга ошибок, поддерживающие исключения из многих языков программирования, имеют общие модели для исключений. К сожалению, у многих систем код сервера закрыт, поэтому об их устройстве приходится узнавать из открытого кода соответствующих библиотек-клиентов.

Sentry [10]

Sentry — одна из самых популярных систем мониторинга. У них есть библиотеки-клиенты для более чем 30 языков программирования[4].

Модель в Sentry — JSON-объект с информацией об исключении.

Пример отчёта о Java-исключении в модели Sentry можно найти в приложении 1.

Более подробные примеры исключений из разных языков в модели Sentry можно увидеть по ссылке[5].

Вся работа с отчётом происходит как с нетипизированным JSON-объектом.

Основная часть исключения — трассировка стека. В модели Sentry это просто массив. Трассировка стека может быть только одна. В частности, не отправляются исключения-причины (caused by) и отправляется только последняя трассировка[6].

Основная часть информации внутри кадра стека языкозависима и используется при автоматической обработке только как идентификатор равенства кадров стека (два кадра считаются равными, если информация в них совпадает полностью) и при пользовательской настройке правил сравнения[7].

Также вместе с исключением может передаваться вспомогательная информация. Например, идентификационные данные пользователя, библиотеки в окружении с их версиями, операционная система. Помимо этого, может передаваться и похожая на исключения последовательная информация: состояния потоков (с трассировками стеков и значениями регистров), навигационные цепочки (breadcrumbs), информация о контексте. При этом, добавление каждого нового типа информации выполнено языкозависимо и требует написания отдельного кода на сервере для его обработки.

[4] https://sentry.io/platforms/

[5] https://github.com/getsentry/sentry/tree/master/src/sentry/data/samples

[6] https://github.com/getsentry/sentry-java/blob/main/sentry/src/main/java/io/sentry/SentryExceptionFactory.java#L90

[7] https://docs.sentry.io/platforms/unity/data-management/event-grouping/fingerprint-rules/

Стоит отметить, что в объекте JSON, соответствующем кадру стека, содержится довольно много информации. Более того, для алгоритмов Exception Analyzer для каждого кадра будет требоваться ещё больше информации (например, подсистемы функции, указание на место в приложенном файле).

Airbrake [1]

В модели системы мониторинга Airbrake так же, как и в Sentry. отправляется JSON-файл. К сожалению, код сервера здесь закрыт и невозможно проанализировать, как происходит работа с ним. Здесь, в отличие от Sentry, посылаются и исключения-причины тоже. Но информации посылается значительно меньше чем в Sentry: для каждого кадра записывается имя функции, номер строки и имя файла[8].

Помимо этого посылаются:

- Контекст: язык исключения, операционная система, архитектура
- Параметры: пары ключ-значение, которые могут задаваться пользователем
- Сессия: пары ключ-значение; информация о сессии работы, задаваемая пользователем
- Окружение: пары ключ-значение; информация об окружении, задаваемая пользователем

Rollbar [7]

Модель Rollbar очень похожа на модель Airbrake за некоторыми исключениями.

Здесь, помимо информации о кадре стека, которая есть в Airbrake, посылается ещё и имя класса. Все основные параметры окружения

[8] `https://github.com/airbrake/javabrake/blob/master/src/main/java/NoticeStackFrame.java`

здесь фиксированы, их больше чем в Airbrake, но при этом пользователь может добавлять свои[9].

Periscop [11]

Periscop — система мониторинга, специализирующаяся на отчётах из микросервисов. Отчёты здесь передаются с помощью Protocol Buffers. Передаётся только класс исключения, сообщение об ошибке и построчная трассировка стека. Также поддерживаются исключения-причины[10]

Эта модель отчётов не расширяется на случай добавления языкозависимой информации для каких-либо исключений.

1.3. Выводы

По результатам изучения работ в области алгоритмов сравнения ошибок и в области моделей в системах мониторинга были сделаны следующие выводы:

- Классические алгоритмы сравнения ошибок можно разделить на две группы: оперирующие с последовательностями (трассировками) кадров стека и оперирующие с мультимножествами кадров стека. Вторая группа алгоритмов подходит для языконезависимой обработки ошибок.

- Для алгоритмов сравнения ошибок требуется информация для идентификации кадра стека. Некоторые алгоритмы требуют информацию о подсистемах: классах, пакетах.

- Больше всего информации об ошибке отправляется в Sentry. Но модель Sentry рассчитана только на одну ошибку в отчёте, не позволяет добавлять исключения-причины и алгоритмы сравнения на сервере работают только с трассировками стека. Но даже

[9] `https://github.com/rollbar/rollbar-java/blob/master/rollbar-api/src/main/java/com/rollbar/api/payload/data/Data.java`

[10] `https://github.com/soundcloud/periskop/blob/master/representation/errors.proto`

несмотря на то, что в Sentry передаётся много информации в отчёте, её всё равно не хватает для Exception Analyzer.

- В системах Rollbar и Airbrake передаётся значительно меньше информации об окружении, чем в Sentry. Информации о каждом кадре стека тоже меньше (например, не передаётся имя файла и принадлежит ли этот кадр стека проекту). В отличие от Sentry, здесь передаются исключения-причины. Подробно рассмотреть обработку исключений на сервере в этих системах невозможно, поскольку код сервера закрыт. И Rollbar, и Airbrake, и Sentry передают отчёт в виде JSON-файла. Такой формат удобен для добавления языкоспецифичных свойств.

- Система Periscop передаёт ещё меньше разобранной информации, чем Rollbar и Airbrake. Periscop использует для передачи отчётов Protocol Buffers, что, с одной стороны, делает отчёты более типизированными, а с другой стороны, затрудняет добавление языкоспецифичных свойств.

2. Разработка модели

В данной главе поэтапно представлена разработка модели, начиная от выработки требований, заканчивая подробным описанием разработанной модели. Также в этой главе представлена первая версия модели, в рамках которой была реализована часть функциональности Exception Analyzer, но затем модель была признана неудачной.

2.1. Требования к разрабатываемой модели

Основные требования к модели, разработанные после анализа аналогичных моделей в других системах мониторинга, алгоритмов сравнения отчётов и текущей обработки отчётов в Exception Analyzer:

- В рамках модели должно быть возможно записать ошибку в последовательной структуре (например, трассировку стека)
- В рамках модели должно быть возможно записать ошибку в древовидной структуре (например, зависание пользовательского интерфейса)
- В рамках модели должно быть возможно выразить отображение (например, отображение из регистров в память)
- Модель должна позволять указывать опциональные языкозависимые поля (например, время кадра стека в зависании интерфейса)
- В модели должны поддерживаться отчёты, в которых содержится более одной ошибки (например, исключения-причины (caused by))
- У элементов ошибки (например, кадров стека) в модели должен быть идентификатор, позволяющий сравнивать их между собой
- В рамках модели должно быть возможно выразить информацию об источниках элементов ошибки (например, из какой строки какого файла элемент)

- В рамках модели должно быть возможно выразить связь элементов ошибки с файлами-вложениями, которые показываются пользователю (какая строка файла какому элементу ошибки соответствует)

- При работе с моделью должно быть возможно выразить как языконезависимую, так и языкозависимую обработку

- В рамках модели должно быть возможно передавать информацию об окружении, в котором произошла ошибка (плагины, операционная система, библиотеки с версиями и так далее) и дополнительную метаинформацию (идентификатор пользователя, версия модели и так далее)

2.2. Первая версия модели

Основная идея при разработке первой версии модели состояла в том, что ошибку любой структуры из рассмотренных выше, можно выразить в виде дерева, где элементы ошибки записаны на рёбрах (в частности, последовательность — это граф-бамбук[11], а отображение — граф-звезда[12]). В эту структуру очень хорошо и без дублирования информации вписываются исключения, поскольку исключения с исключениями-причинами как раз образуют дерево. Более того, в одном дереве можно сохранить и несколько ошибок.

Отчёт представляется в формате JSON.

Ошибки записываются в виде сжатого дерева, то есть на ребре может быть записано сразу несколько элементов ошибки. Это сделано, во-первых, чтобы работа с деревом была проще (например, если знаем, что в отчёте есть всего одна ошибка и она последовательная, можно получить массив её элементов за одно обращение к дереву), а во-вторых, для экономии памяти (иначе для каждого элемента создавалась бы отдельная вершина и ребро).

[11]Граф-бамбук — дерево с одним листом

[12]Граф-звезда — дерево, у которого все вершины кроме корня являются листовыми

Рис. 1: Представление отчёта в модели 1

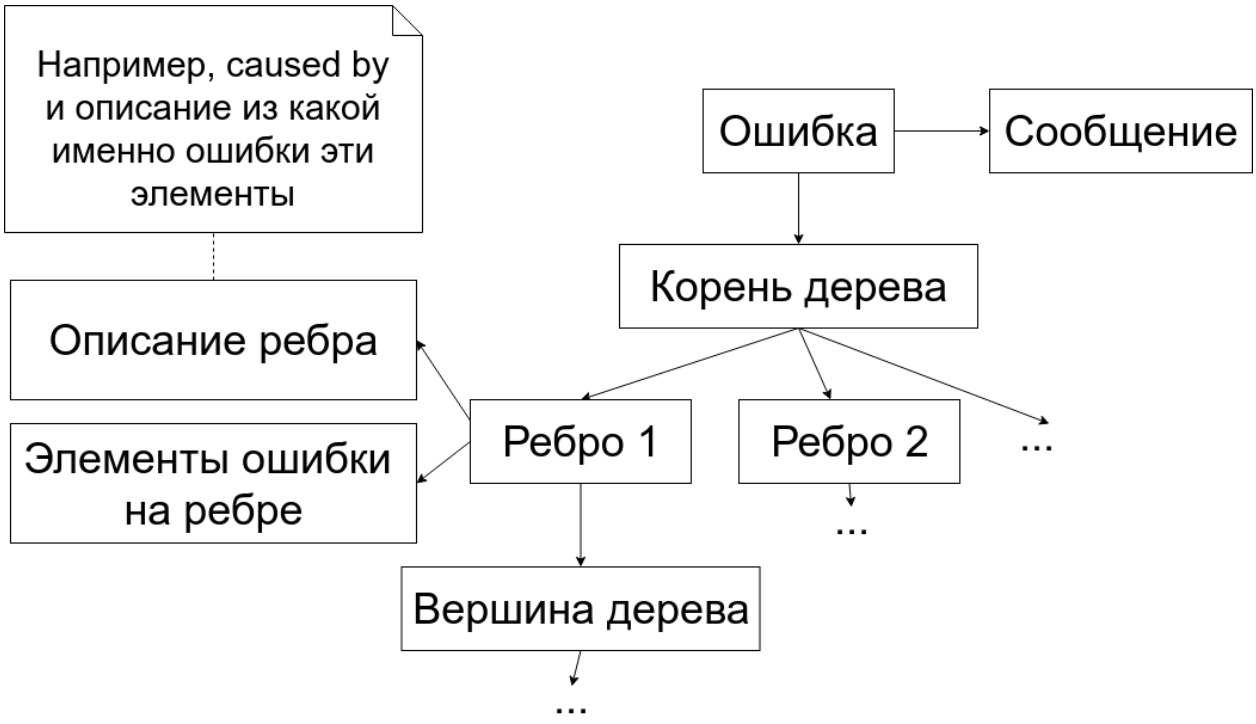

Рис. 2: Представление ошибки в модели 1

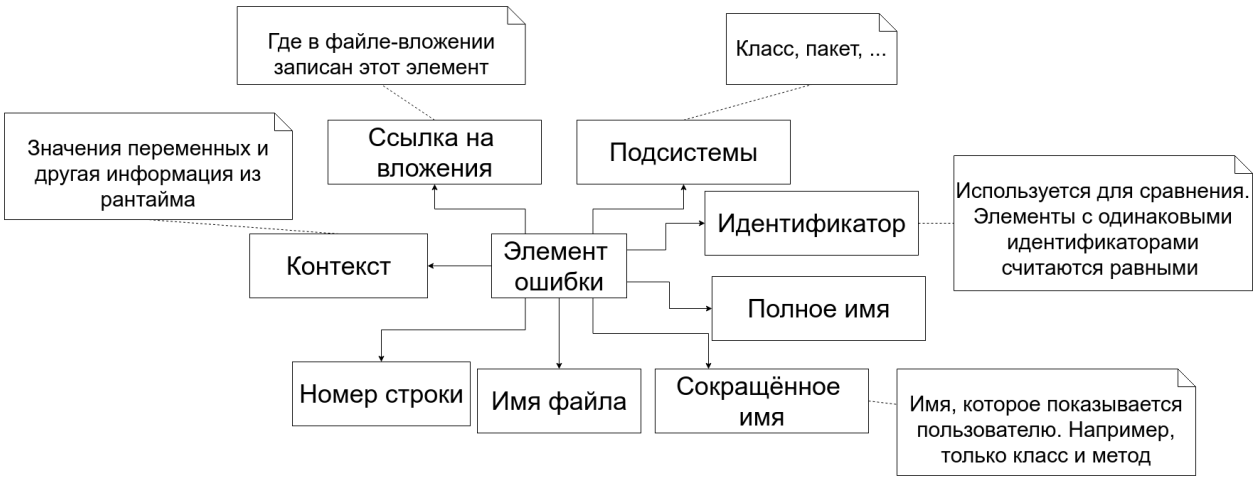

Рис. 3: Представление элемента ошибки в модели 1

Элемент ошибки, записанный на ребре, содержит всю информацию о себе: идентификатор для сравнения, ссылка на вложения, ссылка на исходный код (имя файла, номер строки), подсистемы (класс, пакет и так далее) и дополнительные языкозависимые данные.

В рамках этой модели была реализована часть обработки сбоев JBR в Exception Analyzer и были выявлены следующие недостатки:

- С деревом работать неудобно, особенно когда заранее известно, что ошибка представляется как последовательность (трассировка)
- Слишком обобщённое представление ошибок повысит порог входа к использованию клиент-библиотеки или изменению сервера
- Объект элемента ошибки избыточно нагружен. Информацию о связи с исходным кодом лучше вынести отдельно, поскольку она используется лишь в небольшой части мест на сервере

Поэтому модель было решено переделать, учитывая эти недостатки.

2.3. Вторая версия модели

Во второй версии модели ошибки были разделены. У каждой ошибки указана структура (последовательность, дерево, взвешенная последовательность или любая другая структура, выразимая в JSON), в которой она записана.

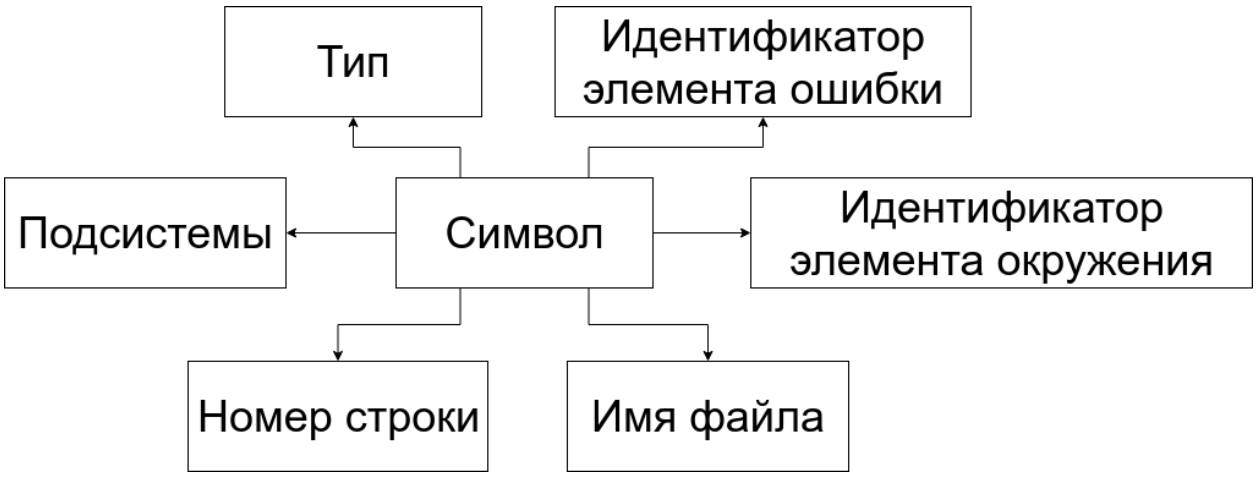

Рис. 4: Представление символа в модели 2

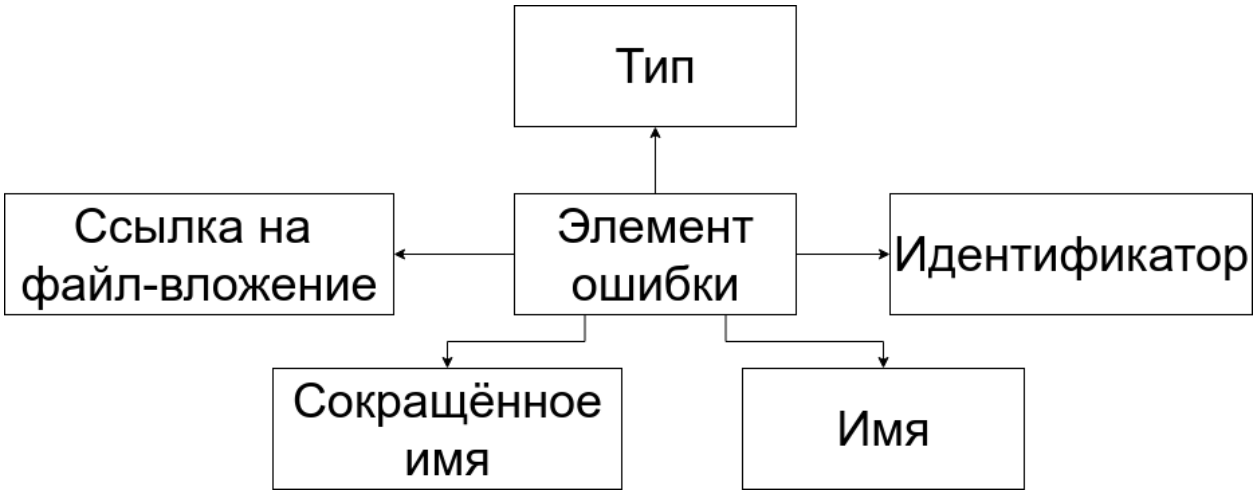

Рис. 5: Представление элемента ошибки в модели 2

Добавлена новая сущность — символы. Символ — это связь с исходным кодом; то есть ссылка на исходный код, а также вспомогательная информация, например, подсистемы. Символ, в свою очередь, ссылается на элемент ошибки и на элемент окружения. Благодаря этому, JSON ошибки стал более читаемым.

У ошибки теперь есть три поля: имя, метка и тип. Имя — это короткое описание ошибки. Например, в случае исключений, это класс исключения. Метка — обозначение ошибки, чтобы их различать между собой. Например, `main` для главного исключения или `caused1` для первого исключения-причины. Тип — описание, что это за ошибка. Например, `jvm.stacktrace` для трассировки стека JVM. Вместе эти три поля образуют **идентификатор ошибки**. При сравнении будут

сопоставляться только ошибки с одинаковыми идентификаторами.

Пример отчёта в JSON можно найти в приложении 2.

2.4. Выводы

Были разработаны две версии модели, которые удовлетворяют перечисленным выше требованиям. Первая версия модели оказалась неудобной в использовании, поэтому была переработана. В итоговой модели отчёта можно выразить всё, что представимо в модели Sentry, а значит она подходит для исключений из, как минимум, 30 языков.

3. Обработка отчётов на сервере в рамках модели

В этой главе описан пошаговый процесс реализации поддержки отчётов в формате разработанной модели на сервере Exception Analyzer.

3.1. Реализация базовой функциональности Exception Analyzer для сбоев JBR

Первым шагом, было решено реализовать базовую функциональность Exception Analyzer для сбоев JBR. Во-первых, потому что обработка сбоев JBR в Exception Analyzer неполная (например, они не индексируются), а поэтому реализовать всю уже имеющуюся функциональность проще, а затем можно будет улучшить обработку сбоев, добавив стадии обработки, используемые для исключений. Во-вторых, потому что недавно для сбоев JBR, помимо стандартного файла, описывающего сбой, начали приходить файлы с дополнительной информацией, которую тоже нужно учитывать при сравнении отчётов; это показалось хорошей возможностью опробовать модель в нетривиальной ситуации.

Раньше процесс работы со сбоями JBR был таким:

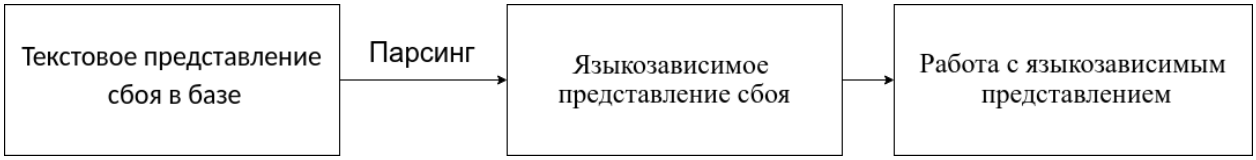

Рис. 6: Процесс обработки в языкозависимом формате

Теперь в эту цепочку добавилось ещё одно звено:

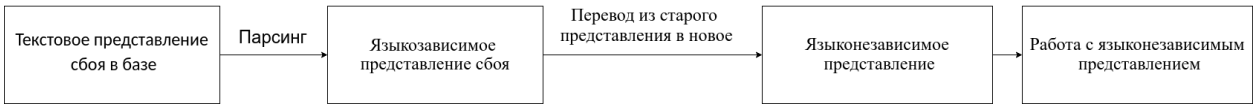

Рис. 7: Процесс обработки в языконезависимом формате

Можно было бы упростить этот пайплайн и сразу перевести сбои в

JSON-формат новой модели, но это бы затруднило дальнейшую модификацию модели (при любой модификации, все отчёты из базы пришлось бы заново переводить в новый формат).

Важно заметить, что работа с моделью на сервере происходит не с JSON-объектом, а с обычными объектами классов, представляющих разные части модели. JSON удобен для передачи и хранения информации, но оперировать удобнее с типизированным отображением в объекты (например, из-за дополнительных проверок на стадии компиляции, статического анализа и автодополнения).

Трассировка стека в рамках модели представляется как последовательность, где у всех элементов одинаковый тип. Отображение из регистров в память представляется тоже как последовательность, но тип каждого элемента зависит от имени регистра, чтобы при сравнении сравнивались только одинаковые регистры между собой.

Затем, так как в дальнейшем в языконезависимой модели предполагается одинаково обрабатывать все отчёты, всю языкозависимую логику было решено вынести в адаптер модели, специфичный для каждого вида отчётов.

Адаптер модели для сбоев изначально обладал следующей функциональностью (в дальнейшем функциональность к адаптеру будет добавляться):

- Генерация заголовка отчёта
- Генерация сигнатуры (короткое человекочитаемое описание ошибки)
- Получение хеша отчёта (для последующего быстрого поиска одинаковых отчётов)
- Выделение ошибок для сравнения (подробно описано ниже)

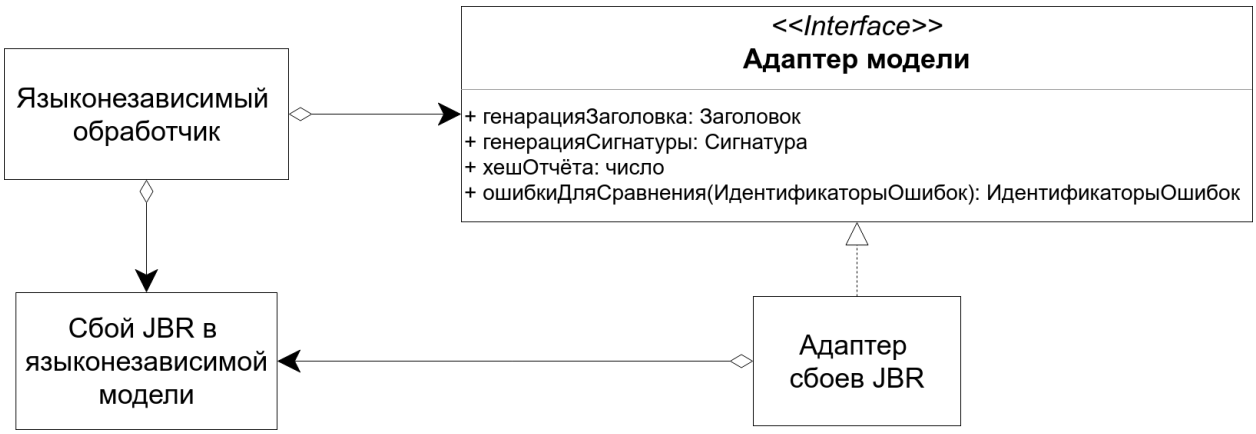

Рис. 8: Обработка отчёта с помощью адаптера

Генерация заголовка отчёта в рамках языконезависимой модели:

```
private val os = crash.environment.getType(OsType)
private val crashTrace = crash.errors.getType(TraceType)
private val siginfo = crashTrace.flatMap(_.name)
private val problematicFrame = crashTrace.flatMap(_.byKeyOpt(ProblematicFrame))

override def title: String = {
    os.map(_ + ": ").getOrElse("") +
      siginfo.map(_ + " at ").getOrElse("") +
      problematicFrame.map(_.name).getOrElse("Unknown") +
      crash.errors.getType(JbrNativeTraceType)
        .flatMap(_.byKeyOpt(JbrMessage))
        .map(" in " + _)
        .getOrElse("")
  }
```

Для сравнения ранее использовался простой алгоритм на основе косинусного сходства. Сравнивались кадры стека из трассировок, значения в регистрах и сигнатуры. Этот же алгоритм был реализован над языконезависиимой моделью. Реализация была успешно протестирована на отчётах из базы: вызывалась старая, языкозависимая, реализация сравнения и новая, и похожести проверялись на равенство. Время работы сравнения от использования новой модели увеличилось незначительно, поскольку время на перевод сбоя из старого представления в новое пренебрежительно мало по сравнению со временем, затрачиваемым на сравнение.

Так как в дальнейшем планируется ускорение обработки сбоев с помощью перевода ошибок в бинарное представление и индексирова-

ния, было решено переписать этот алгоритм на более похожий на тот, что будет использоваться далее. Поэтому был реализован в языконезависимом стиле алгоритм из статьи Lerch [4], который используется в Exception Analyzer для сравнения исключений, и было решено отказаться от сравнения отображений из регистров в память, поскольку это сравнение вносит в итоговый результат незначительный вклад. Алгоритму для работы требуются значения IDF кадров стека, но их можно получить только после индексирования всех отчётов, поэтому временно IDF считалась равной единице.

Далее я реализовал поддержку расширенных сбоев JBR: к некоторым сбоям, помимо основного файла с информацией о сбое, прикладывается файл с дополнительной информацией, в котором может содержаться JVM-трассировка стека и нативная трассировка стека.

Для поддержки этих сбоев я реализовал парсер таких файлов и добавил соответствующие ошибки при переводе сбоя в языконезависимую модель. Работа парсера была протестирована на всех сбоях такого вида из базы.

Для определения сравниваемых ошибок был добавлен соответствующий метод в адаптер. Метод принимает список идентификаторов «главного» отчёта при сравнении и возвращает список идентификаторов, ошибки, соответствующие которым, надо сравнивать. Например, если у одной из ошибок нет дополнительного файла, то ошибки будут сравниваться только по трассировкам из основного файла. Если же дополнительный файл есть, то будут сравниваться только ошибки, которые есть в дополнительных файлах, а трассировки из основного файла будут игнорироваться (так посчитали нужным сравнивать эти отчёты разработчики, которые занимаются разбором сбоев JBR).

Ранее отчётов с несколькими ошибками в Exception Analyzer не было, поэтому необходимо было продумать логику работы системы в такой ситуации. Я реализовал это следующим образом: алгоритм сравнения на вход принимает, помимо функции определения схожести и сравниваемых ошибок, ещё и реализацию интерфейса для сравнения нескольких ошибок. Этот интерфейс предоставляет метод, который на

вход принимает функцию схожести и последовательность пар из сравниваемых ошибок и возвращает результат. Одной из реализаций такого интерфейса было взятие минимума из результатов по всем сравнениям, что и было применено для сбоев JBR. Такой интерфейс позволит в будущем гибко настраивать сравнение отчётов, состоящих из нескольких ошибок и, при необходимости, внедрять языкозависимую логику.

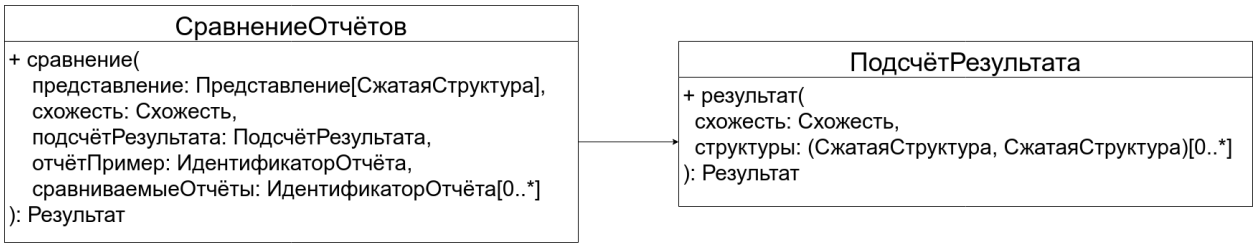

Рис. 9: Обработка отчётов с несколькими ошибками

Таким образом, вся прежняя функциональность для сбоев JBR была реализована с использованием модели, а также была добавлена обработка отчётов, состоящих из нескольких ошибок.

3.2. Улучшение обработки сбоев JBR

Следующим шагом было решено реализовать весь пайплайн обработки исключений для сбоев JBR, а именно:

- Генерацию бинарного представления, индексирование и поиск

  Чтобы работа с отчётами проходила максимально быстро, нужно работать с сжатыми данными. Для этого в Exception Analyzer кадры стека исключений индексируются: кадру стека в соответствие ставится число. С помощью такого отображения становится возможным перевести в бинарный вид весь отчёт и использовать эту информацию при сравнении. Также важной задачей является поиск похожих отчётов. Для исключений поиск реализован с помощью обратного индекса.

- Выделение (маркировку) важных элементов

В Exception Analyzer у разработчиков есть возможность помечать, какие кадры стека являются причиной возникновения данного исключения. Такие кадры будут подсвечиваться в плагине и будут учитываться при сравнении: если выделенный кадр есть в обоих отчётах, то их схожесть будет увеличена, а если кадр есть только в одном отчёте, схожесть будет уменьшена.

- Быстрые алгоритмы сравнения

  То есть алгоритмы сравнения, которые работают с бинарным представлением ошибки.

- Визуализацию сравнения

  Чтобы объяснить разработчику, почему система считает, что два отчёта похожи, надо визуально выделить их общие части. Сейчас в Exception Analyzer для исключений с этой целью используется алгоритм нахождения взвешенного редакционного расстояния. Нужно реализовать его в языконезависимом стиле и обобщить на случай нескольких ошибок в одном отчёте.

Индексирование, генерация бинарного представления и поиск

Ранее пайплайн индексирования выглядел следующим образом:

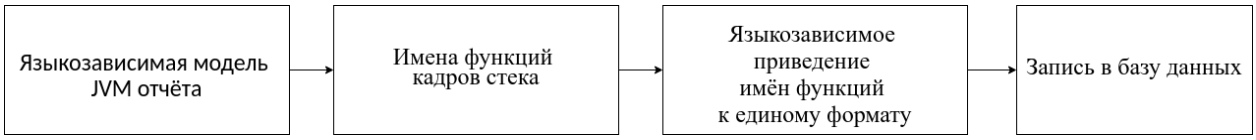

Рис. 10: Пайплайн индексирования раньше

Приведение имён функций к единому формату включает в себя унификацию имён лямбда-функций (например, `com.group.IssuesController$8.ancestorAdded` заменится на `com.group.IssuesController$0.ancestorAdded`) и другие языкозависимые преобразования.

Часть приведения имён к общему формату языкозависимая, поэтому в языконезависимой модели она была переведена на фазу генерации

модели, поскольку идентификатор элемента как раз по задумке и должен быть одинаковым для равных по смыслу элементов.

Таким образом, удалось переиспользовать, за незначительными изменениями, индексирование отчётов JVM. Сейчас сбои JBR индексируются в одну таблицу в базе с отчётами JVM. В будущем будет протестировано, как это сказывается на правильности подсчёта IDF (он насчитывается при индексировании тоже), и, при надобности, индексирование будет разнесено по разным таблицам, в зависимости от типа отчёта.

После того, как элементы проиндексированы, можно строить бинарное представление ошибки. Для сбоев JBR используются только ошибки-последовательности, поэтому было достаточно реализовать тривиальный перевод последовательности идентификаторов в последовательность чисел с помощью уже имеющегося функционала индексирования.

Далее бинарное представление требовалось сохранить в базу, но переиспользовать эту функциональность от JVM-отчётов не получилось, поскольку там все сжатые представления хранились в одной таблице и одному отчёту могла соответствовать только одна ошибка, что неверно для сбоев JBR. Поэтому хранение бинарных представлений было полностью переработано.

Рис. 11: Хранение сжатых структур отчётов в базе данных

Идентификатор ошибки находится в развязочной таблице, поскольку так можно будет находить номера нужных структур по идентификаторам ошибки, не подгружая при этом саму структуру и обращаясь только к одной таблице.

С помощью обратного индексирования был реализован поиск похожих отчётов. Он похож на языкозависимую реализацию, но поддерживает несколько ошибок для одного отчёта. Алгоритм работы:

1. Загружается полное представление отчёта-шаблона (того отчёта, похожие отчёты на который ищутся)
2. С помощью адаптера получаются идентификаторы ошибок, которые сравниваются
3. Загружаются бинарные представления соответствующих ошибок
4. Выделяются значимые элементы этих ошибок

   Выделение значимых элементов — это алгоритм, уже имеющийся в Exception Analyzer. Выделяются элементы с высоким значением IDF и некоторые из префикса или суффикса ошибки, если ошибка последовательная
5. С помощью обратного индекса ищутся отчёты, в которых есть значимые элементы

Выделение (маркировка) важных элементов

В плагине разработчик может отмечать важные элементы, которые являются значимыми для этой ошибки. Наличие/отсутствие выделенных элементов влияет на результат сравнения двух ошибок.

Чтобы поддержать текущий протокол выделения важных элементов, требовалось сделать сопоставления между элементами ошибок и строками выводимого пользователю текста (текстового представления отчёта). Сложность здесь снова в том, что для отчётов JBR пользователю показываются склеенные тексты двух файлов-вложений и надо правильно произвести сопоставление элементов.

Первым шагом решения этой задачи было сохранение сопоставления строк и элементов при парсинге сбоя.

Затем в адаптер было добавлено два метода:

1. Получить текстовое представление отчёта

2. Получить отображение из строк текстового представления отчётов в строки файлов-вложений

Таким образом, имея отображение из элементов в строки файлов-вложений и отображение из строк текстового представления в строки файлов-вложений было построено отображение из элементов в строки текстового представления и обратное к нему, что позволило переиспользовать имеющийся протокол выделения важных элементов.

Быстрые алгоритмы сравнения

Были реализованы языконезависимые алгоритмы сравнения на основе прошлой реализации и использования бинарного представления ошибки. Также, с помощью индексирования, стало возможно считать IDF элементов.

Скорость работы данной реализации была протестирована на отчётах из базы данных. По результатам измерений, скорость сравнения обычных отчётов JBR увеличилась приблизительно в 4 раза, а отчётов с дополнительными файлами приблизительно в 8 раз. Замеры проводились на группах из тысячи отчётов одного вида. В первом случае выполнялось ≈ 1800 операций сравнения в секунду (при ≈ 450 ранее), во втором случае ≈ 1200 (при ≈ 150 ранее).

Визуализация сравнения

С помощью построенного ранее отображения из элементов в строки текстового представления стало возможно поддержать протокол визуализации алгоритмов сравнения, который уже используется для исключений.

Для генерации визуализации алгоритмов сравнения в языконезависимом стиле и с учётом нескольких ошибок был переписан существующий алгоритм на основе взвешенного редакционного расстояния.

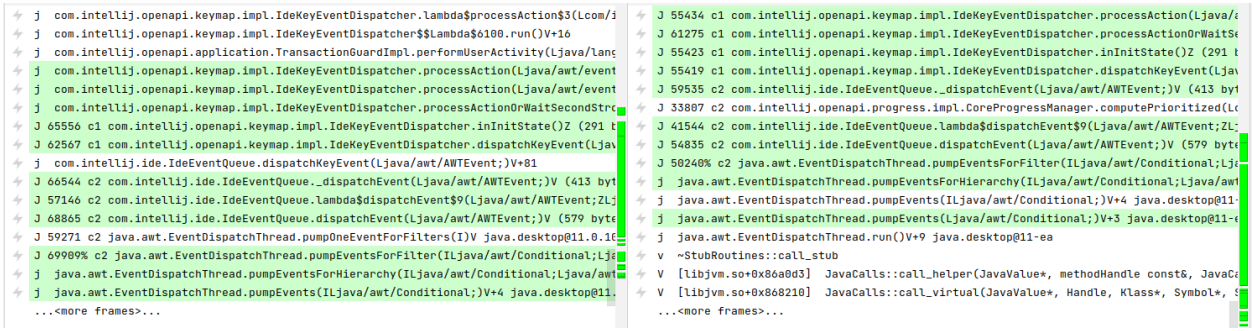

Рис. 12: Визуализация сравнения двух сбоев JBR

3.3. Обработка исключений и зависаний

Следующим шагом было добавление обработки исключений и зависаний.

У обработки исключений и зависаний есть особенности, которых нет у обработки сбоев JBR:

- Удаление рекурсии (учитывается в генерации и текстового, и бинарного представлений)
- Для обработки зависаний используется языкозависимый алгоритм, учитывающий время у кадра стека
- Используются более сложные алгоритмы кластеризации отчётов
- Для исключений реализовано взаимодействие с исходным кодом (в большей части языконезависимое, но есть и языкозависимые части)

Первым шагом исключения и зависания были переведены в языконезависимую модель по тому же пайплайну, который описан выше: из языкозависимого формата, который использовался ранее, отчёт переводится в языконезависимый формат, а языкозависимая логика реализуется с помощью адаптера модели.

Для сравнения зависаний используется «трассировка зависания» — небольшая (по сравнению со всем деревом зависания) последовательность кадров стека из зависания с их временами. Ошибка, содержащая «трассировку зависания» является главной ошибкой всего отчёта.

Поскольку получать всё дерево зависания долго, есть два режима перевода зависания в языконезависимый формат: с сохранением времён кадров стека (в этом случае грузится всё дерево зависания и в «трассировке зависания» проставляются времена кадров стека) и без сохранения времён кадров стека. Второй вариант используется, когда нужно выполнить работу, не связанную со сравнением или индексированием. Например, при генерации заголовка отчёта.

Для представления зависаний была реализована структура **взвешенная последовательность** и её бинарное представление.

Далее был реализован в языконезависимом стиле алгоритм, используемый для сравнения зависаний.

Алгоритм удаления рекурсии уже был реализован в языконезависимом стиле, поэтому было достаточно только удалить выбранные алгоритмом элементы из генерации текстового и бинарного представления.

Для сравнения исключений была переиспользована реализация языконезависимого алгоритма для сбоев JBR. Реализация была протестирована на 10000 отчётах из базы и результаты сравнения отчётов оказались такими же, как и в языкозависимой реализации (поскольку алгоритм из языкозависимой реализации сравнения сбоев и был взят за основу языконезависимого алгоритма), что свидетельствует о корректности реализации.

3.4. Выводы

Таким образом, в рамках предложенной в предыдущей главе модели, удалось реализовать языконезависимый пайплайн обработки отчётов в Exception Analyzer и перевести на этот пайплайн работу с .NET, JVM исключениями, сбоями JBR и зависаниями пользовательского интерфейса. Благодаря этому удалось значительно улучшить обработку сбоев JBR: скорость их сравнения увеличилась в 4-8 раз, стало возможно выделять важные элементы и появилось визуальное сравнение отчётов.

4. Клиент-библиотека для отправки отчётов

В данной главе описана реализация клиент-библиотеки для JVM языков, которая в дальнейшем будет использована для отправки отчётов из IntelliJ IDEA.

4.1. Разработка библиотеки

Так как в дальнейшем планируется писать и библиотеку для JavaScript исключений, было решено писать библиотеку для JVM исключений с помощью технологии `Kotlin Multiplatform`, которая позволит переиспользовать большую часть кода.

Текущая реализация отправки отчётов в языкозависимой модели может быть найдена по ссылке[13].

Новая библиотека разделена на две части: общую, написанную на Kotlin Multiplatform, и языкоспецифичную, написанную на Kotlin JVM.

Общая часть включает в себя объектное представление модели, перевод этой модели в JSON и отправку по сети. Языкоспецифичная часть включает в себя разбор исключения (то есть работу с объектом класса `java.lang.Throwable`) и перевод его в модель из общей части.

Отправка по сети реализована с помощью библиотеки `Ktor`. Отправляются все вложения. В частности, сгенерированный по модели JSON, тоже отправляется как отдельное вложение. Отправка происходит с помощью POST-запроса, всё тело запроса сжимается с помощью `gzip`.

Сериализация в JSON и десериализация из JSON реализованы с помощью библиотеки `Kotlinx.serialization`[14].

[13] `https://github.com/JetBrains/intellij-community/blob/master/platform/platform-impl/src/com/intellij/diagnostic/ITNProxy.java`

[14] `https://github.com/Kotlin/kotlinx.serialization`

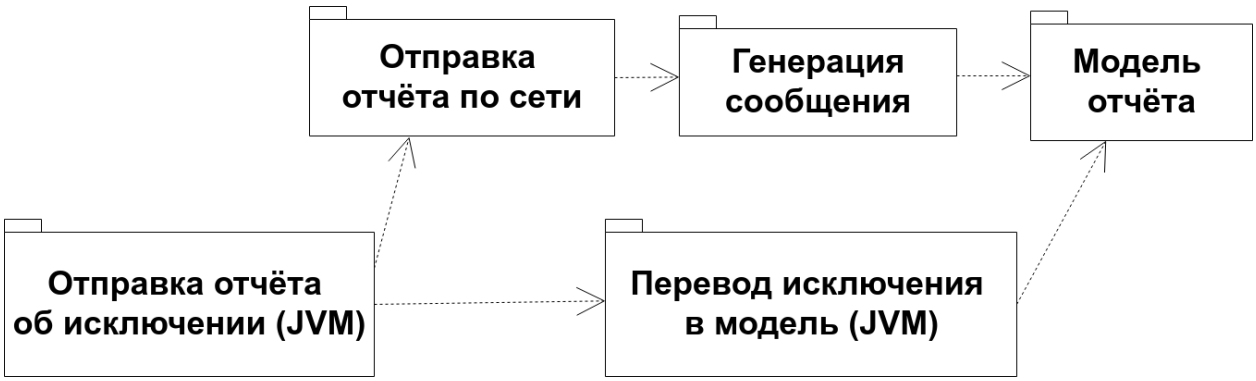

Рис. 13: Взаимодействие пакетов внутри библиотеки

В дальнейшем, эту библиотеку планируется расширять. В частности, в неё могут быть перенесены адаптеры модели, которые сейчас реализованы на сервере. Это позволит разработчикам самим настраивать то, как будут обрабатываться отчёты, не затрагивая код сервера.

Отправка отчёта с помощью этой библиотеки будет внедрена в одну из ближайших версий IntelliJ IDEA.

4.2. Выводы

Была реализована библиотека для отправки отчётов о необработанных исключениях из языков для JVM. Часть библиотеки будет переиспользована для отправки отчётов о необработанных исключениях из JavaScript. В дальнейшем, библиотека будет расширяться и отправка отчётов с помощью неё будет внедрена в одну из ближайших версий IntelliJ IDEA.

Заключение

В результате данной работы предложена языконезависимая модель для обработки и отправки отчётов в системах мониторинга ошибок. Модель подходит для описания исключений из, как минимум, 30 популярных языков программирования, сбоев виртуальных машин, зависаний пользовательского интерфейса и других типов отчётов.

Обработка отчётов в рамках модели была реализована в Exception Analyzer. 4 языкозависимых пайплайна обработки были заменены на языконезависимый. Это позволило улучшить обработку сбоев JBR: скорость их сравнения увеличилась в 4-8 раз, добавилась возможность выделения важных кадров стека и появилось визуальное сопоставление отчётов. В одном отчёте поддерживается несколько ошибок, что позволило добавить обработку нового вида отчётов о сбоях JBR.

Была реализована клиент-библиотека для отправки отчётов об исключениях из языков для JVM. Отправка исключений из IntelliJ IDEA с помощью этой библиотеки будет внедрена в одной из ближайших версий. Часть кода библиотеки будет переиспользована для работы с JavaScript исключениями.

В дальнейшем планируется продолжать работу с моделью в нескольких направлениях:

- Реализовывать клиент-библиотеки для разных языков программирования.

  Уже есть запрос на клиент-библиотеки для Python и JavaScript.

- Переносить языкозависимые адаптеры модели в отдельные библиотеки.

  Это позволит избавить код сервера от языкозависимых частей и облегчит внесение изменений в обработку для сторонних разработчиков.

- Добавлять поддержку новых типов отчётов.

  Например, отчётов о нехватке памяти, отчётов профилирования.

Список литературы

[1] Airbrake: Frictionless error monitoring and performance insight for your app stack. — 2021. — May. — Access mode: `https://airbrake.io/`.

[2] Automatically Identifying Known Software Problems / Natwar Modani, Rajeev Gupta, Guy Lohman et al. // 2007 IEEE 23rd International Conference on Data Engineering Workshop. — 2007. — P. 433–441.

[3] Debugging in the (Very) Large: Ten Years of Implementation and Experience / Galen Hunt, Kirk Glerum, Kinshuman Kinshumann et al. // Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09). — 2009. — October.

[4] Lerch Johannes, Mezini Mira. Finding Duplicates of Your Yet Unwritten Bug Report // 2013 17th European Conference on Software Maintenance and Reengineering. — 2013. — P. 69–78.

[5] Quickly Finding Known Software Problems via Automated Symptom Matching / M. Brodie, Sheng Ma, G. Lohman et al. // Second International Conference on Autonomic Computing (ICAC'05). — 2005. — P. 101–110.

[6] ReBucket: A method for clustering duplicate crash reports based on call stack similarity / Yingnong Dang, Rongxin wu, Hongyu Zhang et al. // Proceedings - International Conference on Software Engineering. — 2012. — 06. — P. 1084–1093.

[7] Rollbar. Rollbar: Error Tracking Software for JavaScript, PHP, Ruby, Python and more. — Access mode: `https://rollbar.com/`.

[8] Runeson Per, Alexandersson Magnus, Nyholm Oskar. Detection of Duplicate Defect Reports Using Natural Language Processing. — 2007. — 06. — P. 499–510.

[9] Sabor Korosh Koochekian, Hamou-Lhadj Abdelwahab, Larsson Alf. DURFEX: A Feature Extraction Technique for Efficient Detection of Duplicate Bug Reports // 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS). — 2017. — P. 240–250.

[10] Sentry: Application Monitoring and Error Tracking Software. — Access mode: `https://sentry.io/welcome/`.

[11] SoundCloud. — Access mode: `https://developers.soundcloud.com/blog/periskop-exception-monitoring-service`.

[12] TraceSim: A Method for Calculating Stack Trace Similarity / Roman Vasiliev, Dmitrij V. Koznov, George A. Chernishev et al. // CoRR. — 2020. — Vol. abs/2009.12590. — 2009.12590.

[13] What Makes a Good Bug Report? / Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg et al. // IEEE Transactions on Software Engineering. — 2010. — Vol. 36, no. 5. — P. 618–643.

[14] A dynamical quality model to continuously monitor software maintenance / Valentina Lenarduzzi, Alexandru Cristian Stan, Davide Taibi et al. // The European Conference on Information Systems Management / Academic Conferences International Limited. — 2017. — P. 168–178.

Приложения

1. Пример Java-исключения в модели Sentry


```
{
  "platform": "java",
  "release": "3.0.2",
  "sdk": {
    "client_ip": "172.56.7.13",
    "name": "sentry-java",
    "version": "1.4.0-3ded0"
  },
  "sentry.interfaces.Breadcrumbs": {
    "values": [
      {
        "level": "debug",
        "message": "Querying for user.",
        "timestamp": 1501799139,
        "type": "default"
      },
      {
        "level": "debug",
        "message": "User found: user@sentry.io",
        "timestamp": 1501799139,
        "type": "default"
      }
    ]
  },
  "sentry.interfaces.Exception": {
    "exc_omitted": null,
    "values": [
      {
        "mechanism": null,
        "module": "io.sentry.example",
        "raw_stacktrace": null,
        "stacktrace": {
          "frames": [
            {
              "abs_path": "Thread.java",
              "filename": "Thread.java",
              "function": "run",
              "in_app": false,
              "lineno": 748,
              "module": "java.lang.Thread"
            },
            ...,
```

```
          {
            "abs_path": "ApiRequest.java",
            "filename": "ApiRequest.java",
            "function": "perform",
            "in_app": true,
            "lineno": 8,
            "module": "io.sentry.example.ApiRequest"
          }
        ],
        "frames_omitted": null,
        "registers": null
      },
      "thread_id": null,
      "type": "ApiException",
      "value": "Authentication failed, token expired!"
    }
  ]
},
"sentry.interfaces.Http": {
  "data": "{\"logged_in\":[\"1\"]}",
  "env": {
    "AUTH_TYPE": null,
    "LOCAL_ADDR": "0:0:0:0:0:0:0:1",
    ...
  },
  "headers": [
    ["Accept-Language","en-US,en;q=0.8"],
    ["Host", "localhost:8080"],
    ...
  ],
  "method": "GET",
  "query_string": "logged_in=1",
  "url": "http://localhost:8080/"
},
"sentry.interfaces.Message": {
  "message": "Authentication failed, token expired!"
},
"sentry.interfaces.User": {
  "data": {
    "account_level": "premium"
  },
  "email": "user@sentry.io",
  "ip_address": "0:0:0:0:0:0:0:1",
  "username": "user"
},
"tags": [
```

```
    [
      "os",
      "Mac OS X 10.12.6"
    ],
    [
      "browser",
      "Chrome 60.0"
    ]
  ],
  "type": "error",
  "version": "6"
}
```

2. Пример Java-исключения во второй версии модели

```
{
  "meta": { "protocol": "0.0.1_jvm", "reporter": "42" }, // метаинформация
  "properties": { "attachments": ["stacktrace.txt", "message.txt"] },
  "environment": [   // окружение: ОС, среда выполнения, библиотеки, плагины...
    {
      "type": "IDE",              "id": "IntelliJ IDEA 212.2345",
      "name": "IntelliJ IDEA",    "version": "212.2345"
    }, ...
  ],
  "symbols": [ // символы: связь с исходным кодом
    {
      "type": "jvm.frame",
      "elementId": "com.intellij.openapi.SomeClass.someMethod", // ссылка на элемент ошибки
      "environmentItemId": "IntelliJ IDEA 212.2345", // ссылка на элемент окружения
      "fileName": "SomeClass.java",
      "line": 251,
      "subsystems": [ // подсистемы
        "com.intellij.openapi.SomeClass", "com.intellij.openapi", "com.intellij", "com
      ]
    },
    ...
  ],
  "errors": [
    {
      // тип, метка и имя – идентификатор
      "type": "jvm.exception.trace", "label": "main", "name": "some.Exception",
      "attachment": "stacktrace.txt",
```

```
      "structure": {
        "type": "sequence", // тип структуры. Последовательность, дерево...
        "elements": [
          { // только самая важная информация об ошибке: сравнение и навигация
            "type": "jvm.frame",
            // идентификатор (для сравнения)
            "id": "com.intellij.openapi.diff.impl.dir.DirDiffTableModel.lambda$reloadM
            "name": "DirDiffTableModel.lambda$reloadModel$0", // человекочитаемое
имя
            "textReference": { "start": 2, "end": 3 } // ссылка на вложение
          },
          ...
        ]
      }
    },
    { // ошибок может быть несколько
      "type": "jvm.exception.trace", "label": "caused1", "name": "java.lang.NullPointe
      "attachment": "stacktrace.txt",
      "structure": {
        "type": "sequence",
        "elements": [
          ...
        ]
      }
    }
  ]
}
```