DOI: 10.15514/ISPRAS-2025-37(4)-7

Обзор методов миграции программных интерфейсов приложений для объектно-ориентированных языков

[1,2] *Я.А. Чуркин, ORCID: 0009-0000-5044-4249 <yan@ispras.ru>*
[1] *Д.М. Мельник, ORCID: 0000-0002-0177-1558 <dm@ispras.ru>*
[1] *Р.А. Бучацкий, ORCID: 0000-0001-8522-1811 <ruben@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.*

[2] *Московский государственный университет имени М.В. Ломоносова, Россия, 119991, Москва, Ленинские горы, д. 1.*

Аннотация. Миграция приложений – это процесс переноса программного обеспечения с одной платформы или версии прикладного интерфейса на другую. В условиях быстрого развития технологий и постоянных изменений в пользовательских предпочтениях эффективная миграция интерфейсов становится необходимостью для поддержания конкурентоспособности приложений. В статье представлен обзор современных методов миграции программных интерфейсов приложений, акцентирующий внимание на важности адаптации программного обеспечения к изменяющимся условиям и требованиям пользователей. Статья также классифицирует существующие подходы к миграции, включая использование автоматизированных инструментов, методов адаптации и рефакторинга кода на объектно-ориентированных языках программирования. Рассматриваются преимущества и недостатки различных методов, таких как адаптация пользовательских интерфейсов к новым платформам, миграция на основе шаблонов и использование адаптеров для обеспечения совместимости между устаревшими и новыми интерфейсами. Обсуждаются вызовы, с которыми сталкиваются разработчики при миграции, включая проблемы семантического преобразования и необходимость учета специфики целевых платформ. Данный обзор будет полезен как исследователям, так и практикам, работающим в области разработки программного обеспечения, предоставляя знания о методах и подходах к успешной миграции программных интерфейсов приложений.

Ключевые слова: программный интерфейс приложения; миграция программного кода; миграция интерфейса API; статический анализ; трансформация программного кода; миграция библиотеки.

Для цитирования: Чуркин Я.А., Мельник Д.М., Бучацкий Р.А. Обзор методов миграции программных интерфейсов приложений для объектно-ориентированных языков. Труды ИСП РАН, том 37, вып. 4, часть 1, 2025 г., стр. 111–146. DOI: 10.15514/ISPRAS–2025–37(4)–7.

Survey of Application Program Interface Migration Methods for Object-Oriented Languages

[1,2] *Y.A. Churkin, ORCID: 0009-0000-5044-4249 <yan@ispras.ru>*
[1] *D.M. Melnik, ORCID: 0000-0002-0177-1558 <dm@ispras.ru>*
[1] *R.A. Buchatskiy, ORCID: 0000-0001-8522-1811 <ruben@ispras.ru>*
[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.*
[2] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia.*

Abstract. Application migration is the process of moving software from one platform or API version to another. With rapid technological development and constant changes in user preferences, effective interface migration is becoming a necessity to maintain the competitiveness of applications. This article provides an overview of modern methods of application programming interface migration, focusing on the importance of adapting software to changing conditions and user requirements. The article also classifies existing approaches to migration, including the use of automated tools, adaptation and refactoring methods for code on object-oriented languages. The advantages and disadvantages of various methods, such as adapting user interfaces to new platforms, template-based migration, and using adapters to ensure compatibility between legacy and new interfaces are considered. The challenges faced by developers during migration are discussed, including semantic transformation issues and the need to take into account the specifics of target platforms. This review will be useful for both researchers and practitioners working in the field of software development, providing knowledge about methods and approaches to successful migration of application programming interfaces.

Keywords: application interface API; code migration; API migration; static analysis; code transformation; library migration.

For citation: Churkin Y.A., Melnik D.M., Buchatskiy R.A. Survey of Application Program Interface Migration Methods for object-oriented languages. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 4, part 1, 2025. pp. 111-146 (in Russian). DOI: 10.15514/ISPRAS-2025-37(4)-7.

1. Введение

В настоящее время различные операционные системы (ОС) быстро эволюционируют, что приводит к внедрению новых возможностей, элементов пользовательских интерфейсов и каркасов для разработки программных приложений, позволяющих использовать новейшую функциональность. Эти изменения неизбежно влекут за собой обновления в программных интерфейсах приложений (API – Application Programming Interface) [1]. В таких условиях важным становится сохранение совместимости приложений, разработанных для более старых версий операционных систем. Адаптация программных приложений к различным программным средам при изменении API, на основе которого они написаны, приобретает особую значимость.

Разработчики приложений обязаны учитывать не только изменения в API операционных систем, но и изменения в API библиотек, используемых в процессе разработки, изменения интерфейса системы сборки (например, CMake [2]) а также обновления или дополнения в языках программирования, на которых написаны эти приложения. Применение нового функционала библиотек или конструкций языка программирования может существенно повысить эффективность работы приложения. Однако изменения в API библиотеки могут привести к полной несостоятельности старого кода, написанного под предыдущие версии API.

Миграция программного кода – это процесс адаптации программного кода к обновлениям одного или нескольких используемых им программных компонентов путем его трансформации. В зависимости от сложности миграции, ее ручное осуществление может

быть достаточно долгим и трудозатратным процессом, особенно для больших кодовых баз. В связи с этим остро встает вопрос о способах автоматизации этого процесса.

Целями исследования является классификация типов миграции кода на объектно-ориентированных языках (раздел 2), а также обзор инструментов и методов автоматической и автоматизированной миграции кода (раздел 3).

Доступность программных решений и исследований, касающихся адаптации программных приложений под различные версии конкретного API, остается ограниченной. В данной работе будут рассмотрены основные из них.

Основные результаты исследования могут быть использованы разработчиками приложений для определения возможности осуществления необходимых миграций их кодовых баз автоматическим или полуавтоматическим способом и помочь с выбором инструмента для осуществления необходимых миграций.

2. Классы миграции программных интерфейсов приложений

Согласно опубликованной статистике [3] менее 40% разработчиков приложений под операционную систему Android полностью адаптировали свои приложения под версию 14 (версия API 34). Под актуальную версию Android 15 (версия API 35) адаптация приложений составляет менее 2%. Изменение версии Android и его API происходят ежегодно и политика выпуска новых версий API в основном направлена на добавление новых функциональных возможностей, улучшение безопасности и энергоэффективности работы приложений и не гарантирует обратной совместимости. Это означает, что рано или поздно приложение может перестать корректно работать или работать не так эффективно на новых версиях Android. Разработчики, которые долго не обновляют версию API своих приложений подвергают риску пользователей. Так как приложений, написанных для Android достаточно много и смена версий API происходит часто, проблема миграции Android приложений под новые версии API стоит особенно остро [4]. Согласно статистике [5], Java является наиболее популярным языком для написания Android приложений. Поэтому было принято решение отбирать примеры миграции из приложений, написанных под Android на языке Java. Эти примеры могут быть обобщены для применения к любому объектно-ориентированному языку программирования, так как являются обособленными и не привязаны к особенностям языка Java.

Отбор примеров миграции исходного кода проводился из открытых репозиториев популярных приложений и программных платформ, написанных на языке Java с открытым исходным кодом на платформе GitHub [6]. Поиск осуществлялся по записям изменений кода (commits), содержащим в заголовке или содержании информацию о переходе с одной версии Android API на другую, например, с двадцать шестой версии на двадцать восьмую и другие. В результате данного процесса было найдено порядка 60 различных примеров изменений программного кода.

После завершения отбора примеры были подвергнуты анализу с целью выявления схожих признаков изменений в программном коде. В результате все примеры были разделены на 2 основных класса миграции API:

1. Миграция с сохранением семантической эквивалентности (низкоуровневый класс миграции API).
2. Миграция без сохранения семантической эквивалентности (высокоуровневый класс миграции API).

Первый класс был дополнительно разделен на 5 подклассов из-за возможности группировки по схожести найденных примеров миграции, принадлежащих данному классу:

1. замена вызова метода или функции последовательностью инструкций;
2. замена библиотеки на схожую по функциональным возможностям;

3. переименование метода класса или функции;
4. замена сигнатуры метода класса или функции;
5. замена нестатического метода статическим и наоборот.

Примеры миграции, принадлежащие второму классу, не были дополнительно сгруппированы из-за уникальности каждого найденного примера. В следующих подразделах будет представлено более подробное описание каждого из этих классов миграции API.

2.1 Миграция с сохранением семантической эквивалентности (низкоуровневый класс миграции API)

Данный класс миграции API относится к ситуации, когда гарантируется семантическая эквивалентность кода до и после миграции. Далее будет дано подробное описание подклассов данного класса миграции.

2.1.1 Замена вызова метода/функции/части кода последовательностью инструкций

Данный подкласс миграции API описывает ситуацию, когда вызов метода класса, функции или часть кода должны быть заменены последовательностью инструкций с сохранением семантической эквивалентности.

На листинге 1 представлен пример замены вызова метода `getAllNetworkInfo` Java-класса `ConnectivityManager` на метод `getAllNetworks` того же класса. Новый метод имеет иной возвращаемый тип объекта, который сохраняется в переменной с другим идентификатором. Использование старого идентификатора также необходимо заменить на новый с учетом его особенностей, что и происходит в следующем цикле. В данном примере миграция осуществляется просто, учитывая, что у данных методов одинаковое количество аргументов и из возвращаемого типа возможно извлечь необходимую информацию, чтобы сохранить семантическую корректность.

```
ConnectivityManager connectivity =
  (ConnectivityManager)context.getSystemService(Context.CONNECTIVITY_SERVICE);
if (connectivity != null) {
  NetworkInfo[] info = connectivity.getAllNetworkInfo();
  for (int i = 0; i < info.length; i++)
    if (info[i].getState() == NetworkInfo.State.CONNECTED)
      return true;
}
```

```
ConnectivityManager connectivity =
  (ConnectivityManager)context.getSystemService(Context.CONNECTIVITY_SERVICE);
if (connectivity != null) {
  Network[] networks = connectivity.getAllNetworks();
  for (int i = 0; i < networks.length; i++) {
    NetworkInfo info = connectivity.getNetworkInfo(networks[i]);
    if (info.getState().equals(NetworkInfo.State.CONNECTED))
        return true;
    }
}
```

Листинг 1. Пример простой замены метода класса. Код до (сверху) и после (снизу) замены.
Listing 1. An example of a simple class method replacement. Code before (top) and after (bottom) replacement.

В дополнение, к этому подклассу относятся ситуации, в которых осуществляется трансформация блоков с циклами, в которых происходит доступ и изменение элементов массива или контейнера (в том случае, когда в API расширился интерфейс доступа к контейнеру и цикл можно сделать короче и эффективнее), например, с `while` на `for` или `foreach`, а также, в общем случае, трансформации операторов управления на иные с сохранением семантической эквивалентности.

В большинстве случаев преобразования программного кода, относящиеся к данному классу миграции API являются тривиальными, так как предполагают учет небольшого локального контекста места трансформации. Соответственно, такой тип трансформации можно осуществить автоматическим способом.

2.1.2 Замена библиотеки на схожую по функциональным возможностям

Данный подкласс миграции API относится к ситуации, когда метод класса, функция или часть исходного кода должны быть заменены другим методом, функцией или частью кода из новой библиотеки или программной платформы, обеспечивая сохранение семантической эквивалентности.

На листинге 2 представлен пример миграции программного кода, использующего библиотеку Java «Apache Network», предоставляющую функциональные возможности для работы с сетевыми протоколами, на библиотеку «Java Internal Network» являющуюся стандартной библиотекой языка Java и также предоставляющую функциональность для работы с сетевыми протоколами. Видно, что миграция сопровождается значительными изменениями в программном коде, поскольку функциональные возможности этих библиотек различаются.

```
HttpClient client = new DefaultHttpClient();
HttpGet request = new HttpGet(url);
HttpResponse response = client.execute(request);
InputStream content = response.getEntity().getContent();
```

```
URL urlObj = new URL(url);
HttpURLConnection conn = (HttpURLConnection)urlObj.openConnection();

conn.setRequestMethod("GET");
conn.setDoInput(true);
conn.connect();

InputStream content = conn.getInputStream();
```

Листинг 2. Пример замены метода класса. Код до (сверху) и после (снизу) замены.
Listing 2. Example of replacing a class method. Code before (top) and after (bottom) replacement.

Преобразования программного кода, относящиеся к данному подклассу миграции API, возможно осуществить автоматическим способом с помощью задания специального механизма соответствия (например, при помощи специализированного DSL) API одной библиотеки другой. Однако построение такого соответствия требует от разработчика, осуществляющего миграцию, углубленных знаний как новой, так и старой библиотеки, что существенно усложняет процесс миграции. Без задания специального механизма соответствия, учитывающего все особенности мигрируемых библиотек, автоматическая миграция невозможна.

2.1.3 Переименование метода класса или функции

Данный подкласс миграции API относится к ситуации, когда метод класса или функция должны быть переименованы при сохранении семантической эквивалентности.

Преобразования программного кода, относящиеся к данному подклассу миграции API, можно осуществить автоматическим способом. При осуществлении трансформаций необходимо учитывать замену как определений метода, так и его использований.

2.1.4 Замена сигнатуры метода класса или функции

Данный подкласс миграции API относится к ситуации, когда сигнатура метода класса или функции должна быть изменена при сохранении семантической эквивалентности.

На листинге 3 представлен пример исходного кода, в котором необходимо выполнить замену сигнатуры метода путем удаления аргумента `mContext` у метода `setTextAppearance`.

```
if (b.getTag().equals(mSelectedNumber)) {
  b.setTextAppearance(mContext, android.R.style.TextAppearance_Large);
  b.getBackground().setColorFilter(null);
}
```

```
if (b.getTag().equals(mSelectedNumber)) {
  b.setTextAppearance(android.R.style.TextAppearance_Large);
  b.getBackground().setColorFilter(null);
}
```

Листинг 3. Пример замены сигнатуры метода класса. Код до (сверху) и после (снизу) замены.
Listing 3. Example of replacing a class method signature. Code before (top) and after (bottom) replacement.

Преобразования программного кода, относящиеся к данному подклассу миграции API, можно осуществить автоматическим способом в тривиальных случаях, когда количество параметров до и после преобразования совпадает или для вычисления новых параметров достаточно использовать небольшой локальный контекст вызова. При осуществлении трансформаций необходимо учитывать замену как определений метода, так и его использований.

2.1.5 Замена нестатического метода статическим и наоборот

Данный подкласс миграции API относится к ситуации, когда необходимо заменить тип вызова метода с нестатического на статический при сохранении семантической эквивалентности. Преобразования программного кода, относящиеся к данному подклассу миграции API, можно осуществить автоматическим способом. При выполнении трансформаций необходимо учитывать замену как определений метода, так и его использований.

2.2 Миграция без сохранения семантической эквивалентности (высокоуровневый класс миграции API)

К данному классу миграции API были отнесены найденные примеры миграции, которые изменили семантические свойства кода после ее проведения.

На листинге 4 представлен пример замены создания и использования объекта Java-класса `ListView` на создание и использование объектов другого Java-класса `RecyclerView`. После проведения трансформации семантические свойства кода будут отличаться, так как объекты представленных классов не обладают эквивалентным поведением.

Преобразования программного кода, относящиеся к данному классу миграции API, сложно осуществить автоматическим способом. Сложность миграции может заключаться в изменении иерархии типов классов, преобразовании методов и функций в другие методы и функции при наличии более одного подходящего кандидата на замену, а также в других аспектах.

```
ListView listView = (ListView) findViewById(R.id.list_view_announcement);
listView.setAdapter(new ListAdapter(AnnouncementActivity.this, list));
listView.setOnItemClickListener(new AdapterView.OnItemClickListener() {
    @Override
    public void onItemClick
        (
            AdapterView <?> parent,
            View view,
            int position,
            long id
        ) {
            startActivity(new Intent(AnnouncementActivity.this,
                AnnouncementContentActivity.class));
        }
});
```

```
RecyclerView mRecyclerView =
    (RecyclerView) findViewById(R.id.recycler_view_announcement);
mRecyclerView.setLayoutManager(
    new LinearLayoutManager(this, LinearLayoutManager.VERTICAL, false));
mRecyclerView.addItemDecoration(
    new DividerItemDecoration(this, LinearLayoutManager.VERTICAL));
mRecyclerView.setAdapter(new ListAdapter(this, list));
```

Листинг 4. Пример замены использования объектов класса.
Listing 4. Example of replacing the use of class objects.

2.3 Выводы

В табл. 1 представлено ранжирование найденных примеров кода по выделенным классам и подклассам миграции API.

Из таблицы видно, что большинство примеров (**>90%**) возможно осуществить с помощью автоматического (с использованием программных средств миграции) и полуавтоматического (с использованием программных средств миграции и небольшим участием разработчика, которое заключается в верификации результатов автоматической миграции или ручная доработка сложных случаев) способа миграции. Оставшиеся **8%** найденных примеров, относящиеся ко высокоуровневому классу и ко второму подклассу низкоуровневого класса миграции API, требуют прямое участие разработчика или гибко настраиваемые инструменты, с помощью которых возможно задать семантическое сопоставление кода до или после миграции.

3. Обзор существующих инструментов и исследований

По рассматриваемой проблеме адаптации программных приложений под различные версии определенного API существует ограниченное количество программных решений и исследований, которые опубликованы и могут быть найдены в открытом доступе. Инструментов, применимых для миграции Java кода еще меньше, поэтому помимо инструментов миграции Java кода будет рассмотрены как мультиязычные инструменты (не привязанные для миграции кода на определенном языке), так и для таких языков как C/C++/JavaScript и т.д. Подходы, используемые каждым из инструментов, можно обобщить как на язык Java, так и на любой другой объектно-ориентированный язык. Каждое программное решение или подход будет рассмотрено в порядке значимости (количества цитирований работ, описывающих его), а также с точки зрения возможности осуществления описанных в предыдущем разделе классов миграций. При обзоре особое внимание будет уделено удобству предлагаемых инструментом способов описания трансформаций кода для конечного пользователя.

Табл. 1. Ранжирование найденных примеров по выделенным классам и подклассам миграции API.
Table 1. Ranking of found examples by selected API migration classes and subclasses.

Класс миграции API	**Доля от общего числа найденных миграций API**	**Способ миграции**	**Проблемы, возникающие в ходе миграции**
Низкоуровневый: Замена метода класса/функции/ части кода	***30%***	Автоматический	Отсутствуют
Низкоуровневый: Замена библиотеки	***4%***	Полуавтоматический или ручной	Отсутствие соответствия между старым и новым кодом
Низкоуровневый: Переименование метода класса или функции	***15%***	Автоматический	Отсутствуют
Низкоуровневый: Замена сигнатуры метода класса или функции	***43%***	Автоматический или полуавтоматический	Нетривиальные замены параметров метода или функции
Низкоуровневый: Замена нестатического метода статическим	***4%***	Автоматический	Отсутствуют
Высокоуровневый	***4%***	Полуавтоматический или ручной	Отображение метода многие ко многим, изменения иерархии типов и др.

Предполагается, что описание трансформаций на целевых языках или на схожих с ними DSL несет меньше нагрузки на пользователя, так как нет необходимости в глубоком изучении внутреннего устройства и механизмов работы инструмента или предполагаемого авторами инструмента подхода.

3.1 Обзор программных инструментов

3.1.1 Coccinelle и Coccinelle4J

Coccinelle [7] – это открытый инструмент для анализа и преобразования программ, написанных на языке C, который широко используется при разработке и выпуске новых версий ядра операционной системы Linux с 2006 года. Инструмент имеет открытый исходный код и доступен в различных версиях операционной системы Linux.

Целью разработки данного инструмента было повышение эффективности обновления кода ядра Linux при изменении API некоторых библиотек, используемых в программном коде ядра, путем выполнения определенных преобразований кода в полуавтоматическом режиме. Правила для преобразования программного кода записываются в Coccinelle с помощью скриптового языка SmPL (Semantic Patch Language – язык семантических преобразований), который имеет синтаксис, подобный синтаксису языка C. В языке SmPL итоговые преобразования целевого кода выражаются с использованием строк кода, представленных с префиксами «−» и «+». Первый префикс предназначен для удаления сопоставленного кода, а второй – для добавления нового кода. Coccinelle нечувствителен к комментариям и пробелам, а также учитывает поток управления и информацию о типах переменных, что существенно повышает возможности для описания сложных трансформаций на SmPL.

На листинге 5 приведен пример описания семантического преобразования для инструмента Coccinelle на SmPL.

Представленное описание семантического преобразования предполагает удаление объявления переменной только в том случае, когда переменная инициализируется константой и используется только в операторе возврата (оператор `return` языка C). Описание преобразования состоит из двух частей: блока с объявлениями метапеременных (в терминах текущего исследования метапеременной называется символьный идентификатор шаблона или правила трансформации, который хранит в себе информацию о синтаксической конструкции языка), каждая из которых задает тип узла и его символьное имя для сопоставления, располагающегося между двумя парами символов «`@@`», и блока с правилами преобразования кода после сопоставления.

```
@@
type T;
constant C;
identifier ret;
@@
- T ret = C;
... when != ret
when strict
return
- ret
+ C
```

Листинг 5. Пример описания семантического преобразования инструмента Coccinelle.
Listing 5. An example of a semantic patch of the Coccinelle tool.

В описанном примере преобразования объявлены три метапеременных с уникальными типами – `T`, `C` и `ret`. В блоке с правилами преобразования описано удаление объявления переменной `ret` в том случае, когда отсутствует ее использование до оператора `return`, а также замену возвращаемого выражения в операторе `return` на константу `C` из объявления переменной `ret`. Непосредственно в коде, переменная языка C может иметь любой символьный идентификатор, значение константного инициализатора, а также тип. Идентификаторы `T`, `C` и `ret` необходимы лишь для ссылки на эти сущности в правилах преобразования.

Одним из недостатков инструмента Coccinelle является то, что он может преобразовывать только код внутри тела какой-либо функции языка C, что сильно сужает его возможности, касающиеся полной миграции исходного кода между различными версиями API. Подводя итог, можно заключить, что инструмент подходит для осуществления большинства миграций, принадлежащих низкоуровневому класса миграции из раздела 2, в том случае, если трансформации необходимо осуществить в пределах контекста отдельных определений функций.

Coccinelle4J – это инструмент, описанный в работе [8], являющийся надстройкой над инструментом Coccinelle. Он предназначен для трансформации кода на языке Java и обладает такими возможностями как переименование метода или изменения его сигнатуры путем добавления параметров. Coccinelle4J использовался для миграции программного кода Java между различными устаревшими API операционной системы Android.

Синтаксис языка Java схож с синтаксисом языка C, что и позволило разработать Coccinelle4J на основе Coccinelle. Таким образом, Coccinelle4J использует тот же язык семантических преобразований – SmPL. Кроме того, разработчики добавили возможность работы с блоками обработки исключений (блоки `try-catch`) и с подтипами классов.

Инструмент Coccinelle4J, как и Coccinelle, может осуществлять преобразования кода только в контексте определения отдельных методов и не поддерживает трансформацию полей

классов. Вышеперечисленное ограничивает возможность использования данного инструмента для высокоуровневой миграции исходного кода между различными версиями API, но делает пригодным использование инструмента для осуществления большинства низкоуровневых миграций из классификации раздела 2.

3.1.2 PATL

PATL (Patch-Like Transformation Language) [9] – это язык трансформаций, предназначенный для описания преобразований программного кода, написанного на языке Java. Его синтаксис схож с синтаксисом языка семантических преобразований SmPL, используемого в инструменте Coccinelle, описанном в предыдущем разделе.

На листинге 6 представлено описание правила трансформации кода на PATL.

```
// rule rButton
(jb : JButton -> Button, parent : JPanel -> Composite) {
- jb = new JButton();
- parent.add(jb);
+ jb = new Button(parent, SWT.PUSH);
}
```

Листинг 6. Правила трансформации кода на PATL.
Listing 6. Rules for transforming PATL code.

Данная трансформация подразумевает замену композиции создания объекта класса `JButton` и вызова метода `add`, помеченных знаком «-», на вызов конструктора класса `Button`, помеченного знаком «+».

Перед применением трансформаций выполняется преобразование кода в SSA-форму, в которой каждой переменной значение присваивается только один раз. Код, представленный в SSA-форме, обладает необходимыми свойствами для выполнения оптимизаций и упрощает анализ потока данных. Эта нормализация используется для проверки правил преобразования на согласованность друг с другом перед проведением трансформаций, а также для более точного сопоставления с правилами преобразования. PATL применяет правило сопоставления «многие ко многим»: последовательность вызовов из старого API сопоставляется с последовательностью вызовов из нового API.

Основным преимуществом PATL по сравнению с Coccinelle4J является нормализация исходного кода, что позволяет выполнять более точные сопоставления с правилами трансформации путем семантического анализа для вывода типов. Однако, с другой стороны, исходный код программы полностью преобразуется в трехадресную SSA-форму, что влечет за собой дополнительные сложности для разработчика, связанные с необходимостью обратной трансляции преобразованного представления в исходный код.

Подводя итог, можно заключить, что инструмент подходит для осуществления как низкоуровневых, так и низкоуровневых миграций из классификации раздела 2, учитывая тот факт, что трансформации происходят над преобразованным представлением, а не над исходным кодом.

3.1.3 Twinning и SWIN

В работе [10] представлен язык Twinning для описания правил трансформаций программного кода, написанного на Java. Общий вид правил трансформации на языке Twinning представлен в листинге 7. В приведенном фрагменте показано, что тип `T1i` должен быть сопоставлен типу `T2i` для всех индексов `i`, где каждому типу `T1i` сопоставляется его тип-пара `T2i`. Каждый идентификатор `xi` задает метапеременную, которая имеет тип `T1i` в исходном коде и отображается в экземпляр типа `T2i` – `yi`. Шаблон выражения, которое необходимо заменить

– `JavaExp1` является выражением типа `T10`, которое может использовать метапеременные с идетификаторами `x1...xn`. `JavaExp2` задает выражение после трансформации с типом `T20`, в котором метапеременные `yi` являются результатом отображения метапеременных с идентификаторами `x1...xn`.

Пример правила трансформации на языке Twinning, представлен в листинге 8.

Данное правило задает правила замены всех вызовов метода `elements()` класса `Hashtable` и на вызовы метода `values().iterator()` класса `HashMap`. Например, на листинге 9 представлен фрагмент кода (слева), который преобразуется во фрагмент кода (справа).

Несмотря на то что Twinning учитывает правила вывода Java-типов и требует отображения типов один в один, он все еще небезопасен с точки зрения преобразования типов. Пример на листинге 10 иллюстрирует это.

```
[
T10(T1i xi, ..., T1n xn) { return JavaExp1; }
T20(T2i yi, ..., T2n yn) { return JavaExp2; }
]
```

Листинг 7 Общий вид правила трансформации кода на Twinning.
Listing 7. General view of the code transformation rule on Twinning.

```
 [
Enumeration (Hashtable x)
{ return x. elements ();}
Iterator (HashMap x)
{
    return x.values().iterator ();
}
]
```

Листинг 8. Правило трансформации кода на Twinning.
Listing 8. Twinning code transformation rule.

```
void f(Hashtable t ) {
 Enumeration e = t.elements () :
 ...
}
```

```
void f( HashMap t) {
 Iterator e = t.values().iterator() :
 ...
}
```

Листинг 9. Пример преобразования метода. Код до (слева) и после (справа) преобразования.
Listing 9. Example of method transformation. Code before (left) and after (right) conversion.

```
 [
Container() { return new Container();}
Composite() { return new Composite ( new Shell(), 0);}
]
[
JList() { return new JList(); }
List() { return new List(); }
]
```

Листинг 10. Пример небезопасной трансформации типов.
Listing 10. An example of unsafe type transformation.

Описанные выше правила трансформации при применении к коду `Container x = new JList();` приведут к ошибке типов после трансформации: `Composite x = new List();`. Полученный фрагмент кода содержит ошибку, так как `JList` является подклассом класса

`Container`, но `List` таковым не является, что делает невозможным присвоение созданного объекта класса `List` переменной типа `Composite`.

Рассмотренный пример показывает, что, несмотря на то что Twinning обеспечивает корректную замену выражений и преобразование отдельных типов, применение этих замен приведет может привести к нарушению семантической корректности результирующего кода. Также инструмент не позволяет менять последовательности инструкций и блоки кода, а только лишь отдельные выражения. Этот фактор имеет существенное значение при выборе инструмента для миграции программного кода под различные версии API.

В работе [11] представлен безопасный с точки зрения преобразования типов язык **SWIN**, предназначенный для описания трансформаций программного кода, написанного на Java, который расширяет функциональные возможности инструмента Twinning. Одним из главных преимуществ SWIN по сравнению с Twinning является безопасность преобразований типов объектов.

Программа на языке SWIN состоит из трех частей: правила преобразования, шаблона исходного кода и шаблона кода после преобразований. Пример использования SWIN описан ниже. Исходный код на Java:

На листинге 11 представлена программа преобразования SWIN, где 1 и 2 – это правила преобразования.

На листинге 12 представлен исходный код на языке Java до (слева) и после (справа) преобразования с применением правила из листинга 7.

```
P = [1, 2]
where
1 = () [ new A() : A -> new B() : B ]
2 = (x : A -> B, u : A -> B)
[ x.h(u) : A -> x.k(u, new B()) : B ]
```

Листинг 11. Пример программы преобразования SWIN.
Listing 11. Example of SWIN conversion program.

`(new A ()). h (new A ())`	`(new B()).k(new B(), new B())`

Листинг 12. Пример трансформации выражения на языке Java. Код до (слева) и после (справа) преобразования.
Listing 12. Example of transforming an expression in Java. Code before (left) and after (right) conversion.

Данное правило трансформации описывает замену объектов класса `A` на объекты класса `B`, а также замену метода `h` класса `A` на метод `k` класса `B` в программном коде.

Несмотря на все преимущества языка SWIN, как и Twinning, он позволяет трансформировать лишь отдельные выражения. Это существенно ограничивает его применимость для миграций больших кодовых баз под различные версии API. Также, в отличие от Twinning, описания трансформаций для SWIN обладают гораздо более нетривиальным синтаксисом, что создает дополнительную нагрузку на пользователя.

Подводя итог, можно заключить, что SWIN, как и Twinning, подходят для осуществления тривиальных миграций из первого, третьего и четвертого подклассов низкоуровневого класса миграций API, так как с помощью них возможно осуществить лишь трансформацию вызовов методов, но не их определений.

3.1.4 Nobrainer

Инструмент Nobrainer [12] разработан на основе программного обеспечения компилятора Clang [13] из компиляторной инфраструктуры LLVM [14] в Институте системного программирования им. В.П. Иванникова РАН. Он предназначен для модификации кода,

написанного на языках C/C++, с использованием специальных правил преобразования кода. Правила для Nobrainer пишутся на C/C++ без использования специализированных языков (DSL, Domain-Specific Language – предметно-ориентированный язык). На листинге 13 представлен пример правил преобразования для инструмента Nobrainer.

```
int NOB_BEFORE_EXPR(ChangeOrder)(Agent a, char *x, char *y) {
    return a.compose(x, y);
}
int NOB_AFTER_EXPR(ChangeOrder)(Agent a, char *x, char *y) {
    return a.compose(y, x);
}
```

Листинг 13. Пример правил преобразования Nobrainer.
Listing 13. An example of Nobrainer transformation rules.

В данном примере показано правило преобразования, которое изменяет порядок аргументов внутри вызова метода `compose` класса `Agent`.

Шаблоны трансформации, представляющие фрагмент целевого кода до и после преобразования, пишутся как отдельные функции. Для сопоставления кода при проведении трансформаций Nobrainer использует AST-сопоставители (Abstract Syntax Tree – абстрактное синтаксическое дерево) Clang [15] для поиска подходящих шаблонов по дереву абстрактного синтаксиса.

Nobrainer применяет все замены или создает файл в формате YAML [16] с описанием этих замен. Процесс работы Nobrainer состоит из трех основных этапов:

1. Анализ всех единиц трансляции кода, переданных инструменту командами компиляции, и фильтрация шаблонов сопоставления на действительные и недействительные.
2. Проверка каждого правила на предмет согласованности и его обработка для создания внутреннего представления шаблона преобразования.
3. Сопоставление каждого правила с исходным кодом программы.
4. Генерация замен по шаблону.

Nobrainer также поддерживает написание шаблонизированных правил (см. листинг 14).

```
template <class T> T *before() {
    return (T *)malloc(sizeof(T));
}
template <class T> T *after() {
    return new T;
}
```

Листинг 14. Пример шаблонизированных правил Nobrainer.
Listing 14. An example of Nobrainer template rules.

Данный пример демонстрирует ситуацию, когда разработчику необходима частичная миграция, связанная с динамическим выделением памяти, со стиля языка C на C++.

Несомненным плюсом Nobrainer является формат описания преобразований непосредственно на целевых языках (C/C++), что упрощает использование данного инструмента целевым пользователем. Однако, несмотря на все функциональные возможности инструмента Nobrainer, его формат описания правил преобразования в виде функций и их возвращаемых значений не охватывает значительную часть конструкций целевых языков C/C++, а также не позволяет менять блоки кода единым правилом трансформации (для этого придется писать множество косвенно связанных отдельных правил). Например, с помощью данного инструмента невозможно заменять конструкции

циклов `for(…)` на `while(…)`, что является ограничением при выборе инструмента для миграции кода между различными версиями API.

Подводя итог, можно заключить, что инструмент Nobrainer подходит для осуществления тривиальных миграций из первого, третьего и четвертого подклассов низкоуровневого класса миграций API, так как с помощью него возможно осуществить лишь трансформацию вызовов методов, но не их определений.

3.1.5 MARTINI

MARTINI [17] – это инструмент для анализа и преобразования программ на языках C/C++, который использует подход инструмента Nobrainer. В MARTINI задаются пары шаблонов поиска (`match`) и шаблонов трансформации (`replace`) соответственно. В отличие от Nobrainer, который позиционируется как безопасный инструмент с точки зрения преобразования типов, разработчики MARTINI используют менее строгий синтаксис описания шаблонов. Например, символьные имена внутри шаблона не воспринимаются как реально существующие в исходном коде. Это означает, что в Nobrainer (см. раздел 3.1.4) невозможно заменить тип произвольного выражения с произвольными именами переменных, тогда как подход MARTINI это позволяет.

На листинге 15 представлен пример кода шаблонов поиска и трансформации, которые заменяют инициализацию указателя произвольного типа с `0` на `nullptr`.

```
template <typename T>
[[clang::matcher("nullptr-decl"]]
auto null_match()
  [[clang::matcher_block]]
  T *var = 0;
}

template <typename T>
[[clang::matcher("nullptr-decl"]]
auto null_replace()
  [[clang::matcher_block]]
  T *var = nullptr;
}
```

Листинг 15. Пример шаблона трансформации MARTINI.
Listing 15. Example of MARTINI transformation template.

На данный момент инструмент поддерживает только ограниченное подмножество стандартов C/C++ и находится в активной стадии разработки, что является существенным ограничением при выборе инструмента для миграции кода между различными версиями API. Подводя итог, можно заключить, что MARTINI подходит для осуществления тривиальных миграций из первого, третьего и четвертого подклассов низкоуровневого класса миграций API, так как с помощью него возможно осуществить лишь трансформацию вызовов методов и функций, но не их определений.

3.1.6 Refaster

Refaster [18] – это открытый инструмент, разработанный компанией Google (впоследствии переименованный в Error-Prone), предназначенный для рефакторинга Java кода с помощью шаблонов кода «до» и «после» его сопоставления. Шаблоны записываются на языке Java без специализированного DSL.

В качестве примера рассмотрим перехода на новый API библиотеки кодирования `Base64`. На листинге 16 представлен код до и после преобразования.

Правило преобразования кода инструмента Refaster, позволяющее осуществить данную миграцию, представлено на листинге 17.

Правила (шаблоны) преобразования, применяемые инструментом Refaster, включают в себя класс, содержащий один или несколько аннотированных методов с пометкой `@BeforeTemplate` и один метод, аннотированный как `@AfterTemplate`. Каждый из методов шаблона может содержать аргументы, именуемые переменными выражений, которые соответствуют выражениям определенного типа в пользовательском коде; например, это могут быть байты, связанные с кодированием `Base64`. Тело метода шаблона представляет собой выражение, возвращаемое оператором `return`, аналогично подходу, используемому в инструменте Nobrainer (см. раздел 3.1.4). Шаблоны преобразования Refaster описывают замену любого выражения в исходном коде, которое соответствует одному из выражений, возвращаемых методами `@BeforeTemplate`, на выражение, возвращаемое единственным методом `@AfterTemplate`.

Однако одним из основных недостатков Refaster является отсутствие семантического анализа (хотя бы для корректного вывода типов) кода перед сопоставлением. Приведенный на листинге 18 фрагмент кода не будет преобразован в соответствии с установленным правилом.

```
System.out.println(Base64.encodeWebSafe(Files.toByteArray(file), false));
```

```
System.out.println(BaseEncoding.base64Url().omitPadding().encode(Files.toByteArray(file));
```

Листинг 16. Пример миграции библиотеки кода, использующего библиотеку Base64. Код до (сверху) и после (снизу) миграции.
Listing 16. An example of migrating a code library that uses the Base64 library. Code before (top) and after (bottom) migration.

```
class BaseEncodingMigration {
    @BeforeTemplate public String before(byte[] bytes) {
        return Base64.encodeWebSafe(bytes, false /* no padding */);
    }
    @AfterTemplate public String after(byte[] bytes) {
        return BaseEncoding.base64Url().omitPadding().encode(bytes);
    }
}
```

Листинг 17. Пример правила трансформации Refaster.
Listing 17. An example of a Refaster transformation rule.

```
doPadding = false;
System.out.println(Base64.encodeWebSafe(Files.toByteArray(file), doPadding));
```

Листинг 18. Пример правила трансформации Refaster.
Listing 18. An example of a Refaster transformation rule.

В данном случае инструмент Refaster не способен определить, что переменная `doPadding` имеет булевый тип. Так как он учитывает только типы стандартной библиотеки языка Java (примитивные и классы), а также классы из Java библиотек от компании Google.

Более того, Refaster иногда стремится переписать реализацию нового API. Например, если метод `BaseEncoding.base64Url(…)` в приведенных ранее правилах Refaster использует в своей реализации метод `Base64.encodeWebSafe(…)`, это может привести к возникновению бесконечного цикла, когда Refaster попытается переопределить `BaseEncoding.base64Url(…)` в терминах самого себя. Таким образом, инструменту Refaster необходима помощь разработчика для разрешения определенных ситуаций, связанных с преобразованием кода, которые он не в состоянии решить самостоятельно.

Указанные выше ограничения инструмента Refaster препятствуют выполнению сложных и масштабных преобразований кода, требующих семантический анализ сопоставленных конструкций. В дополнение к этому, Refaster не способен трансформировать цепочки операторов, а лишь отдельные выражения. В связи с этим, Refaster подходит для осуществления тривиальных миграций из первого, третьего и четвертого подклассов низкоуровневого класса миграций API, так как с помощью него возможно осуществить лишь трансформацию вызовов методов, но не их определений.

3.1.7 Proteus

Proteus [19] – это проприетарный инструмент, предназначенный для анализа и преобразования программ на языке C/C++, учитывающий пробельные символы и комментарии. Инструмент использует промежуточное представление программы LL-AST (Literal-Layout AST), которое учитывает количество пробельных символов, табуляции, пользовательские комментарии, а также отдельные литералы, такие как « (», «) » в качестве самостоятельных элементов абстрактного синтаксического дерева.

Шаблоны поиска в Proteus описываются с помощью разработанного DSL YATL (Yet Another Transformation Language), который синтаксически схож со Stratego [20].

Разработанный язык позволяет захватывать узлы LL-AST и сопоставлять их внутренним переменным языка YATL для дальнейшего семантического анализа. Язык YATL позволяет использовать конструкции целевого языка в задании типов узлов AST, которые необходимо сопоставить и трансформировать, а также ссылаться в шаблонах трансформации на нетипизированные переменные языка YATL, хранящие в себе значения AST узлов.

Пример шаблона поиска и трансформации вызова функции `boo` на вызов функции `foo` представлен на листинге 19.

```
foreach-match(FunctionCall:{‘boo,=*p}) {
  on *p {
    $_ = new(Id:{‘foo});
  }
}
```

Листинг 19. Пример правила трансформации Proteus.
Listing 19. An example of a Proteus transformation rule.

Фактически, трансформации кода, выполняемые Proteus представляют собой интеллектуальные макроподстановки, а сам инструмент представляет собой препроцессор, учитывающий комментарии, пробельные символы и отступы, но, несмотря на это, инструмент не поддерживает трансформации, требующие семантического анализа, например, с помощью него невозможно осуществить предварительную проверку типов перед сопоставлением и заменой и прочее. Также Proteus обладает нетривиальным синтаксисом языка описания правил трансформации, что может потребовать от пользователя, осуществляющего трансформацию, дополнительных навыков и времени на его изучение.

Инструмент подходит для осуществления тривиальных миграций из первого, третьего и четвертого подклассов низкоуровневого класса миграций API, так как с помощью него возможно осуществить лишь трансформацию вызовов методов и функций, но не их определений.

3.1.8 SPOON

В работе [21] представлена открытая библиотека SPOON, разработанная на языке программирования Java. Эта открытая библиотека предназначена для анализа и преобразования кода программ, написанных на языке Java, и может быть использована для

различных преобразований, включая изменения классов, методов и типов переменных. В процессе анализа исходного кода библиотека SPOON создает абстрактное синтаксическое дерево (AST), использующее информацию о метамодели объектов SPOON, являющейся расширением стандартной метамодели языка Java, которая задает иерархию сущностей (например, метод, класс, переменная и т. д.) Java с их атрибутами и является фактически структурой Java AST. Метамодель SPOON включает специально разработанные классы, имеющие префикс `Ct`. Например, `CtElement` является базовым (корневым) классом метамодели SPOON, `CtClass` представляет абстракцию класса Java, а `CtMethod` служит абстракцией для объявления метода Java.

Преобразование программы в контексте библиотеки SPOON предусматривает комбинацию её анализа и последующей трансформации. Эта концептуальная пара реализована в классе-обработчике SPOON, который фокусируется на анализе элементов программы в метамодели. В качестве иллюстрации, на листинге 20 приведен код, демонстрирующий процессор, который анализирует программу с целью поиска и замены метода `getText` класса `A` на новый метод `testReplace`.

```
public class MethodProcessor extends AbstractProcessor<CtElement> {
    public void process(CtElement el) {
        if (el instanceof CtMethod<?>) {
            // Замена объявления метода
            CtMethod<?> method = (CtMethod<?>)el;
            if (... && method.getSimpleName().equals("getText")) {
                method.setSimpleName("testReplace");
            }
            // Замена вызовов метода
            if (el instanceof CtInvocation<?>) {
                ...
            }
}
```

Листинг 20. Пример класса-обработчика SPOON.
Listing 20. An example of SPOON processor.

В данном примере захватываются все узлы AST, так как каждый узел является производным от базового класса метамодели SPOON – `CtElement`. Метод `process` класса `MethodProcessor` принимает запрошенный элемент в качестве входных данных и выполняет анализ, в результате чего происходит замена имени метода `getText` класса `A` при объявлении и во всех вызовах. Для использования обработчика требуется его добавление в библиотеку SPOON. В библиотеке допускается использование нескольких обработчиков одновременно, при этом порядок их применения определяется пользователем.

Тестирование показало, что библиотека SPOON хорошо распознает цепочки вызовов функций. Например, она может распознать и изменить метод `getText` Java-класса `A` с помощью описанного выше обработчика в ситуации на листинге 21.

```
Class c = new C();
c.getA().getText()
```

Листинг 21. Пример трансформируемого Java кода.
Listing 21. An example of Java programming code

SPOON способен распознать, что возвращаемое значение метода – `c.getA()` является экземпляром класса `A`.

Однако данная библиотека не предоставляет функциональности для работы со статическими методами классов, а также не учитывает поток управления программы при трансформациях

кода (анализ потока управления при использовании данного инструмента можно провести вручную, обходя дерево AST самостоятельно).

Среди других возможностей библиотека SPOON предоставляет возможности для модификации не только исходного пользовательского кода, но и пользовательских комментариев. Ограничениями SPOON являются классы Java, которые определены во внешних пакетах Java. В контексте простых примеров трансформации кода, таких как преобразование классов, методов и переменных, библиотека SPOON демонстрирует хорошие результаты.

Библиотека SPOON используется во многих инструментах восстановления и преобразования Java-кода, включая:

1. Genesis [22] – инструмент для автоматического исправления кода Java (добавление или замена предикатов в блок с условием оператора `if`, замена операторов возврата `return`, а также других типов ошибок);
2. Elixir [23] – инструмент для поиска дефектов в программном обеспечении, написанном на языке программирования Java;
3. Cap-Gen [24] – инструмент для автоматической генерации некоторого ограниченного класса контекстно-зависимых преобразований;
4. MuCrash [25] – инструмент для выявления условий сбоя выполнения программы и генерации тестов для воспроизведения этих сбоев;
5. Nopol [26] – инструмент для автоматического исправления блоков `if-then-else`, который учитывает поток управления программы, написанной на языке программирования Java;
6. Astor [27] – библиотека, предназначенная для поиска дефектов в программах, написанных на языке программирования Java.

Представленные в списке выше инструменты и библиотеки имеют ограниченную сферу применения, косвенно связанную с миграцией программного кода, соответственно, инструментами трансформации программного кода они не являются.

SPOON подходит для осуществления большинства миграций низкоуровневого класса миграций API из раздела 2.

3.1.9 Comby

Comby [28] – это инструмент для синтаксической трансформации кода, который использует концепцию парсеров-комбинаторов (Parser Combinators) для многократной обработки синтаксических структур языков программирования. В работе авторы подробно описывают архитектуру инструмента, его возможности и применяемые методы.

Comby предоставляет контекст для выполнения трансформаций на основе синтаксических шаблонов и обладает поддержкой множества языков программирования. Инструмент поддерживает описание правил трансформации на языке, схожим с целевым.

Функциональные возможности инструмента Comby включают в себя системы поиска и замены, что позволяет разработчикам эффективно выполнять рефакторинг, обновления API и другие виды преобразования программного кода.

В листинге 22 снизу показан пример замены цикл `for` на языке Python, который вычисляет сумму, на функцию `np.sum()` из API библиотеки NumPy. Преобразование описывается правилом, показанным в листинге 8, в котором левая часть задает шаблон «до», а правую шаблон «после». Обе стороны правила содержат операторы Python с переменными шаблона (например, `:[[v0]]`), которые связываются с узлами AST фактического исходного кода (например, `:[[v0]]` связывается с результатом вычисления). Эти правила могут быть использованы для преобразования любого целевого кода, который имеет аналогичную

структуру AST с кодом, представленным в листинге 8, независимо от различий в именах переменных.

Хотя Comby поддерживает множество языков и обладает синтаксисом шаблонов трансформации схожим с целевыми языками (что повышает удобство его использования), его эффективность снижается при работе с проектами, обладающими крупными кодовыми базами. Так как Comby не проверяет синтаксическую и семантическую корректность кода после подстановки, то в результате преобразований может получиться некорректный код, например, в шаблон поиска цикла из листинга 22 может попасть цикл, итерируемый по строке. Это приведет к тому, что в `numpy.sum` окажется строка из переменной шаблона – `:[[v2]]`, и возникнет ошибка несоответствия интерфейсов типов времени выполнения интерпретатора python. Соответственно, чем больше кодовая база, тем больше будет таких ошибок. В таком случае потребуется ручная верификация результатов преобразований, что не сильно проще осуществления преобразований вручную. Также не получится задать дополнительные семантические проверки (например, типа) переменных шаблона, содержащих сопоставляемые синтаксические конструкции, так как Comby рассчитан лишь на проведение простых трансформаций в виде поиска и замены. Переменные шаблона представляют собой лишь найденные синтаксические конструкции в исходном коде, а не являются типизированными AST узлами.

```
:[[v0]] = 0
for :[[v1]] in :[[v2]]:
    :[[v0]] = :[[v0]] + :[[v1]]
```

```
:[[v0]] = numpy.sum(:[[v2]])
```

```
- result = 0
- for elem in elements:
-     result = elem + result
+ result = numpy.sum(elements)
```

Листинг 22. Пример использования Comby.
Правило перезаписи «до» (слева-сверху) и «после» (справа-сверху), Итоговое преобразование (снизу).
Listing 22. An example of Comby tool usage.
The rewriting rule "before" (left-top) and "after" (right-top), the final transformation (bottom).

Из-за перечисленных недостатков Comby подходит для осуществления тривиальных миграций из первого, третьего и четвертого подклассов низкоуровневого класса миграций API, так как с помощью него возможно осуществить лишь трансформацию вызовов методов и функций, но не их определений.

3.1.10 OpenRewrite

OpenRewrite [29] – это инструмент с открытым исходным код, предназначенный для рефакторинга и миграции программного кода на языке Java. Правила миграции можно записывать как в декларативном, так и в императивном стиле.

Декларативный стиль предполагает использование существующих правил миграции с параметризацией, которые описывается в файле конфигурации на языке YAML. В листинге 23 представлен пример использования такого файла конфигурации, предполагающего миграцию с библиотеки Java ApacheClient [30] версии 4 на версию 5. Такой тип миграции предполагает существенное изменение описаний классов и их методов, соответственно используются разных правила миграции, каждая из которых осуществляет определенный ее вид.

OpenRewrite также позволяет описывать правила миграции выражений или отдельных операторов в императивном стиле с помощью шаблонов `@BeforeTemplate` и `@AfterTemplate` библиотеки инструмента Refaster (см. раздел 3.1.6).

Инструмент поддерживает также более интеллектуальный тип миграций с использованием обходчиков LST (Lossless Semantic Tree – семантического дерева без потерь) [31], который представляет собой вариант реализации AST с расширенной информацией о типе каждого узла дерева и сохранением информации о форматировании исходного кода (наличие отступов и комментариев). На листинге 24 представлен пример описания правила миграции с помощью Java шаблонов.

```
---
type: specs.openrewrite.org/v1beta/recipe
name: org.openrewrite.java.apache.httpclient5.UpgradeApacheHttpClient_5
displayName: Migrate to ApacheHttpClient 5.x
description: >
  Migrate applications to the latest Apache HttpClient 5.x release.
recipeList:
  - org.openrewrite.java.apache.httpclient4.UpgradeApacheHttpClient_4_5
  - org.openrewrite.java.apache.httpclient5.UpgradeApacheHttpClient_5_ClassMapping
  - org.openrewrite.java.apache.httpclient5.UpgradeApacheHttpClient_5_DeprecatedMethods
  - org.openrewrite.java.apache.httpclient5.UpgradeApacheHttpClient_5_TimeUnit
  - org.openrewrite.java.apache.httpclient5.StatusLine
---
```

Листинг 23. Пример декларативного шаблона миграции OpenRewrite.
Listing 23. An example of a declarative OpenRewrite migration pattern.

```
@Value
@EqualsAndHashCode(callSuper = false)
public class SayHelloRecipe extends Recipe {
    ...
    public class SayHelloVisitor extends JavaIsoVisitor<ExecutionContext> {
      private final JavaTemplate helloTemplate =
         JavaTemplate.builder("public String hello() { return \"Hello from #{}!\"; }")
                     .build();
        @Override
        public J.ClassDeclaration visitClassDeclaration(...) {
          ...
          // Проверка того, что в классе нет метода "hello"
          boolean helloMethodExists = classDecl.getBody().getStatements().stream()
                  .filter(statement -> statement instanceof J.MethodDeclaration)
                  .map(J.MethodDeclaration.class::cast)
                  .anyMatch(methodDeclaration ->
                    methodDeclaration.getName().getSimpleName().equals("hello"));
          ...
          // Добавление метода в описание класса
          classDecl = classDecl.withBody(helloTemplate.apply(...));
          return classDecl;
        }
    }
}
```

Листинг 24. Пример императивного шаблона миграции OpenRewrite.
Listing 24. An example of an imperative OpenRewrite migration pattern.

В примере выше описывается правило, добавляющее в каждый найденный класс метод `hello`, если метод с таким наименованием в этом классе отсутствует. Правило миграции записывается в виде Java класса, производного от `Recipe`, который предоставляет интерфейс, позволяющий задать имя шаблону миграции, текстовое описание, способ сериализации, а также набор обходчиков LST. В данном примере зарегистрирован всего один обходчик – `SayHelloVisitor`, в котором представлен обработчик объявлений класса,

проверяющий, что в классе присутствует метод с именем `hello` и добавляющий его в противном случае.

Созданный таким образом шаблон затем можно использовать в файлах конфигурации YAML вместе с другими шаблонами для проведения более сложных трансформаций.

OpenRewrite предоставляет заранее разработанные шаблоны миграции Java кода в виде библиотеки. Однако результат его работы требует дополнительного анализа, так как OpenRewrite не гарантирует семантическую корректность преобразованного кода. Также эффективность работы инструмента для определенного кода или проекта зависит от качества и полноты доступных в библиотеки шаблонов миграции. Для нетривиальных миграций имеющихся шаблонов не будет достаточно, что потребует от разработчика изучения библиотеки OpenRewrite для написания новых и интеграции их с существующими. Учитывая необходимость верификации результатов трансформации, изучения библиотеки шаблонов и написания новых, затраты времени на процесс миграции могут быть существенны для больших кодовых баз, что ставит под сомнение возможность использования OpenRewrite вместо осуществления миграций вручную.

Несмотря на это библиотека OpenRewrite позволяет описывать сложные трансформации с помощью семантического анализа, используя информацию о типах и взаимосвязь узлов из LST, что автоматически делает его использование пригодным для осуществления большинства миграций низкоуровневого класса миграций API из раздела 2, а также с помощью него возможно осуществлять высокоуровневые миграции.

3.1.11 GoPatch

Открытый инструмент GoPatch [32] представляет собой утилиту командной строки для простого рефакторинга кода на языке Go. Правила трансформации описываются с помощью шаблонов (см. листинг 25), с использованием собственного DSL, похожего на формат Unified Diff [33].

```
# Replace time.Now().Sub(x) with time.Since(x)
@@
var x identifier
@@

-time.Now().Sub(x)
+time.Since(x)
# Not a description comment
```

Листинг 25. Пример правила трансформации GoPatch.
Listing 25. Example of a GoPatch transformation rule.

Правило на листинге 25 позволяет найти и заменить вызовы `time.Now().Sub(x)` вызовами `time.Slice(x)`, где `x` является метапеременной шаблона трансформации с типом идентификатора. GoPatch предоставляет два доступных типа метапеременных для использования в шаблонах – идентификатор и выражение. Поиск и сопоставление осуществляется по AST исходного кода на языке Go. Ищутся строки, помеченные «−», при этом осуществляется попытка проверки соответствия типов, вложенных в строку метапеременных. При успешном сопоставлении генерируются строки, помеченные «+», в которых раскрываются излеченные значения метапеременных в качестве замены сопоставленным строкам шаблона. После обработки всего AST генерируется описание трансформации кода в формате Unified Diff.

Использование такого формата DSL является удачным решением, так как шаблоны трансформации описываются достаточно просто с использованием целевого языка. Однако инструмент GoPatch не позволяет производить трансформации, требующие семантического

анализа сопоставленных узлов перед их заменой. Например, трансформации, требующие проверку переменных на соответствие определенному типу до применения преобразований к коду, использующему их. Соответственно он подходит для осуществления большинства миграций только подклассов (кроме второго) низкоуровневого класса миграций API из раздела 2.

3.1.12 CodeTurn

Astadia CodeTurn [34] является инструментом с закрытым исходным кодом, который осуществляет миграции программного кода на языке Cobol на объектно-ориентированные (ООП) языки высокого уровня – Java и C#. Инструмент был разработан для осуществления миграций старых кодовых баз на языке Cobol на более распространенные языки с целью упрощения их поддержки и развития. CodeTurn позволяет мигрировать код, используя как процедурные, так и объектно-ориентированный стиль целевого языка программирования, например, Java. Примеры миграции кода на языке Cobol с использованием CodeTurn представлены на листинге 26.

COBOL	`SELECT DATFILE ASSIGN TO "test.dat"`
	`* …`
	` OPEN OUTPUT DATFILE`
	` ACCEPT W-TIME FROM TIME`
	` MOVE SS OF W-TIME`
	`  TO BREAK-VAL`
	` WRITE DAT-REC`

Процедурная Java	`cobol.open(datfile, "test.dat", FileOpenMode.OUTPUT);`
	`cobol.acceptTime(wTime);`
	`cobol.move(wTime.ss, datfile.datrec.breakVal); cobol.write(datfile);`

ООП Java	`datfile.open("test.dat", FileOpenMode.OUTPUT);`
	`wTime.setValue(cobol.getTimeAsString());`
	`datfile.datrec.breakVal.setValue(wTime.ss); datfile.write();`

Листинг 26. Миграция кода на COBOL с использованием CodeTurn. Исходный код на COBOL (сверху), Java в процедурном стиле после миграции (посередине), Java в ООП стиле после миграции (снизу).
Listing 26. Migrating code to COBOL using CodeTurn. COBOL source code (top), Procedural style Java after migration (middle), OOP style Java after migration (bottom).

Несмотря на широкую функциональность инструмента, CodeTurn не позволяет мигрировать произвольный код на Cobol, так как поддерживаются не все виды операторов данного языка, что потребует их ручной миграции после проведения автоматической трансформации.

CodeTurn подходит для осуществления большинства миграций низкоуровневого класса миграций API из раздела 2.

3.1.13 TXL

TXL [35] – это специализированный язык программирования, предназначенный для быстрого создания прототипов описаний грамматик языков программирования, а также для выполнения различных преобразований с данными языками. С помощью него можно описывать правила преобразования в том числе и объектно-ориентированных языков.

Программы на языке TXL состоят из трех частей:

1. Не зависящей от контекста базовой грамматики для языка, с которым необходимо выполнять преобразования;
2. Набора не зависящих от контекста грамматических преобразований (расширений или изменений) базовой грамматики языка;
3. Корневого набора преобразований в базовый язык.

Пример правила преобразования операции синтаксически длинного присваивания в операцию синтаксически короткого присваивания, описанного на языке TXL представлен на листинге 27.

Применимость TXL к различным классам миграции из раздела 2 существенно зависит от того или иного инструмента, которые реализует, описанные в работе авторов TXL механизмы. Но, подводя итог, можно заключить, что с помощью подхода авторов TXL возможно осуществлять лишь тривиальные миграции только из низкоуровневого класса, так как язык ориентирован на грамматические преобразования без учета семантического смысла конструкций целевого языка, а также их типов. Подводя итоги, TXL подходит для описания лишь тривиальных миграций из первого, третьего и четвертого подклассов низкоуровневого класса миграций API.

```
rule simplifyAssignments
  replace [statement]
  V [reference] := V + E [term]
by
  V += E
end rule
```

Листинг 27. Пример правила трансформации присваивания на TXL.
Listing 27. Example of assignment transformation rule in TXL.

3.1.14 JScodeshift

JScodeshift [36] – это открытый инструмент, разработанный компанией Facebook, предназначенный для анализа программного кода, написанного на языке JavaScript. Инструмент работает с AST представлением кода и использует JavaScript библиотеку Reacast [37] для анализа и изменения AST дерева программы.

Еще одной особенностью JScodeshift является использование модуля AST-Types, который предоставляет абстрактную иерархию типов синтаксического дерева (AST), впервые примененную в Mozilla Parser API [38].

Правила миграции программного кода для инструмента описываются на языке программирования JavaScript и оформляются в виде функций. На листинге 28 приведен пример миграции программного кода JavaScript библиотеки Avajs4 [39] средствами JScodeshift.

На листинге 28 свойство класса `ok` заменяется свойством `truthy`, а `notOk` свойством `falsy`.

Другие программные платформы и библиотеки, такие как React5 [40] и Lodash6 [41], используют JScodeshift для миграции между версиями.

После выполнения преобразований инструмент JScodeshift генерирует AST дерево в формате ESTree [42], которое может быть транслировано обратно в высокоуровневый программный код с использованием API компилятора языка TypeScript [43]. Ограничением инструмента является отсутствие возможности добавления семантических проверок свойств узлов AST (например, проверки корректности преобразования объекта к определенному типу) до и после сопоставления, соответственно он подходит для осуществления лишь тривиальных миграций из первого, третьего и четвертого подклассов низкоуровневого класса миграций API.

```
function renameAssertion(name, newName, j, ast) {
ast
.find(j.CallExpression, {callee: { object: {name: 't'}, property: {name}}})
  .forEach(p => p.get('callee')
  .get('property')
  .replace(j.identifier(newName)));
```

```
}
export default function (file, api) {
  const j = api.jscodeshift;
  const ast = j(file.source);
  renameAssertion('ok', 'truthy', j, ast);
  renameAssertion('notOk', 'falsy', j, ast);
  return ast.toSource();
}
```

Листинг 28. Пример правила миграции библиотечного кода средствами JSCodeShift.
Listing 28. An example of a library code migration rule using JSCodeShift.

3.1.15 Rascal

Rascal [44] – это интерпретируемый DSL (Domain-Specific Language – предметно-ориентированный язык), предназначенный для описания трансформации кода программ, написанных на Java, который имеет свой собственный набор примитивных типов данных, таких как `boolean`, `int` и других. Rascal имеет множество возможностей: сопоставление с образцом исходного кода по его AST, обработка исключений, функции, правила перезаписи и другие.

Сопоставление с образцом – это механизм поиска фрагментов кода по шаблону в Rascal. Например, следующее выражение соответствия может использоваться (см. листинг 29), чтобы находить оператор языка Java `while`.

```
whileStat (Exp cond, list [ Stat ] body) := stat
```

Листинг 29. Выражение соответствия для оператора `while`.
Listing 29. Match expression for the `while` statement.

Переменные шаблона, такие как `cond` и `body`, могут быть объявлены как в этом примере, или они могут быть объявлены в контексте, в котором будет сопоставляться шаблон. Также в Rascal определены правила перезаписи, которые являются единственным неявным механизмом управления в языке и используются для поддержки инвариантов во время вычислений. На листинге 30 представлено правило, которое меняет структуру операторов `if` языка Java.

```
rule
  ifStat (neg(Exp cond), list [ Stat ] thenPart,
    list [ Stat ] elsePart)
    -> ifStat (cond, elsePart, thenPart);
```

Листинг 30. Пример правила трансформации операторов if на Rascal.
Listing 30. Example of a rule for transforming if operators in Rascal.

Если блок с условием оператора `if` содержит выражение инверсии, то инверсия из данного выражения удаляется и ветви меняются местами. Это правило будет срабатывать каждый раз, когда обнаружится оператор `if`, соответствующий левой части правила.

Но, несмотря на все преимущества языка Rascal, его синтаксис достаточно далек от синтаксиса Java. Соответственно, пользователю, осуществляющему миграцию, потребуется время для его изучения. Другим недостатком Rascal являются отсутствие возможностей проведения семантического анализа (например, проверка эквивалентности типов) перед сопоставлением, что может потребоваться при проведении высокоуровневых миграций для объектно-ориентированных языков программирования, так как Rascal ориентирован на синтаксическое сопоставление по AST узлов и не предоставляет возможности для анализа

потока данных. Соответственно, Rascal подходит для проведения большинства низкоуровневых трансформаций, описанных в разделе 2.

3.1.16 ClangMR

ClangMR [45] – это инструмент, разработанный в компании Google для упрощения миграций внутренней кодовой базы. Реализованный в инструменте подход основан на использовании в шаблонах трансформации AST сопоставителей компилятора Clang, как и в случае рассмотренного ранее инструмента Nobrainer. Основное отличие состоит в том, что написание шаблонов трансформации для ClangMR требует знаний внутреннего API Clang и структуры абстрактного синтаксического дерева. На листинге 31 приведен пример сопоставления и замены вызовов метода `Bar` класса `Foo` методом `Baz` из того класса.

Для осуществления миграции кода путем его трансформации, пользователь определяет один или несколько сопоставителей Clang AST (в примере на листинге 31 описан сопоставитель, предназначенный для поиска всех вызовов с именем `Bar` из пространства имен `Foo`, обладающих одним аргументом). Также пользователю необходимо определить обработчики результатов сопоставления по AST (в примере описана функция, которая проверяет, что сопоставленный вызов является вызовом метода в терминах языка C++, а затем производит замену имени вызова на `Baz`, путем тривиальной замены лексической единицы трансляции (`replaceToken`)).

```
StatementMatcher matcher =
  callExpr(allOf(
    argumentCountIs(1),
    callee(functionDecl(hasName(
      "::Foo::Bar")))))
  .bind("call");
```

```
void Refactor(const MatchFinder::MatchResult& res) {
  Clang::CXXMemberCallExpr* call =
    res.Nodes.getStmtAs<clang::CXXMemberCallExpr>(
      "call");
  const clang::MemberExpr *member_expr =
    llvm::dyn_cast<clang::MemberExpr>(
      call->getCallee());
  EditState state(res, call) ;
  state.ReplaceToken(member_expr->getMemberLoc(),"Baz");
  Report(&state);
}
```

Листинг 31. Сопоставитель для поиска вызовов метода Bar (сверху).
Пример реализации правила трансформации вызовов метода Bar с использованием ClangMR (снизу).
Listing 31. Matcher for finding Bar method calls (top).
An example of a handler for transforming Bar method calls using ClangMR (bottom).

Ограничениями такого подхода являются: привязанность к языку (C/C++), необходимость наличия углубленных знаний AST Clang и его API (в отличие от Nobrainer, в котором трансформации осуществляются на целевом языке), нетривиальность проведения сложных трансформаций, что влечет за собой увеличение времени процесса миграции больших кодовых баз. Несмотря на то, что библиотека Clang предоставляет широкие возможности для семантического анализа по AST (включая проверку типов и моделирование потока управления), который является необходимым при описании высокоуровневых миграций, все же у Clang есть ограничения, связанные с невозможностью анализа зависимостей между разными единицами трансляции.

Подводя итог, можно заключить, что инструмент подходит для осуществления большинства миграций, принадлежащих низкоуровневому классу, описанному в разделе 2, а также с помощью него возможно осуществлять высокоуровневые миграции, используя функциональные возможности библиотеки Clang.

3.1.17 Другие инструменты

В обзоре не был представлен фреймворк Cubix [46], предназначенный для описания мультиязычных трансформаций. Также отсутствует упоминание о недавно представленном инструменте PoliglotPirahna [47], основанном на фреймворке языковых подзапросов tree-sitter [48]. Этот инструмент развивает идеи своего предшественника – Piranha [49], который опирается на технологию Error-Prone (см. раздел 3.1.6).

Кроме того, DMS [50] – инструмент с закрытым исходным кодом, который предлагает фронтенд для множества языков и позволяет описывать правила трансформации в декларативно-процедурном стиле с использованием собственного языка описания (DSL), обладающего сложным синтаксисом. Ограничением инструмента является необходимость уточнения и реализации семантики целевого языка для осуществления сложных трансформаций.

Отдельного внимания заслуживают инструменты, представленные в виде расширений программных средств разработки (IDE), например, такие как SemDiff [51] и Catchup! [52], которые помогают в трансформации и анализе программного кода. SemDiff является плагином для Eclipse IDE [53] и уведомляет разработчиков об изменениях API Java библиотек, используемых в проекте. В свою очередь, Catchup! также выступает в качестве плагина для Eclipse IDE и предлагает функциональные возможности для рефакторинга исходного кода Java, включая такие операции как переименование типов, перемещение элементов, изменение сигнатур методов, переименование и изменение типов переменных, а также переименование полей.

Далеко не все инструменты рефакторинга и трансформации ориентированы непосредственно на исходный код. Например, в работе [54] описан инструмент ASM, который позволяет манипулировать и трансформировать классы в бинарных файлах с байткодом Java.

Также, существуют инструменты, предназначенные исключительно для поиска и сопоставления языковых конструкций без применения замен или трансформаций. К таким инструментам относятся ast-grep [55] и Semgrep [56]. Оба инструмента поддерживают множество языков и осуществляют сопоставление программного кода на уровне AST. Кроме того, язык запросов CodeQL [57] и его инструментарий позволяют выполнять поиск и сопоставление конструкций в программном коде с использованием декларативных запросов без его непосредственной замены.

3.2 Обзор исследований

В данном подразделе представлен обзор исследований на тему миграции API, описывающих проблемы, возникающие как при определении необходимости миграции, так и при осуществлении самой миграции. Рассматриваемые результаты исследований не нашли практическое применение в виде полноценных инструментов (не прототипов) или программных решений, соответственно нет возможности полноценно протестировать их для оценки возможности осуществления определенных классов миграций API.

3.2.1 Исследование практического применения методов автоматизированного переноса API Android

В работе [58] авторы обсуждают свой опыт миграции Android API, включая подробное рассмотрение документации. Основное внимание уделяется выявлению проблем,

возникающих при миграции программного кода, а также определению случаев, когда миграция оказывается необходимой.

Авторы выделяют несколько проблем, связанных с миграцией API, которые рассматриваются как неразрешимые:

1. Проблема миграция методов API «N-to-N», которая выражается в необходимости поиска соответствия методов из разных наборов API друг другу.
2. Проблема определения ложноположительных случаев среди множества возможных миграций.
3. Проблема определения временных интервалов между необходимостью добавления и удаления API.
4. Проблема определения миграции кода (значительное количество (около 66,3%) исследованных модификаций API не содержало миграций кода, то есть замены отсутствовали или находились в труднодоступном для поиска состоянии).

Также в работе представлены способы поиска информации об истории миграции API в проектах и описаны основные проблемы, которые могут возникнуть во время этого поиска.

3.2.2 Полуавтоматическое обновление приложений в ответ на изменения API библиотеки

В статье [59] предлагается метод распространения правил миграции кода через заголовочные файлы библиотеки. Правила интегрируются в грамматику языка, используемого для реализации приложения, что требует генерации нового синтаксического анализатора. Миграция API выполняется на уровне абстрактного синтаксического дерева (AST) с использованием инструментов PCCTS (Purdue Compiler Construction Tool Set) [60] и Sorcerer [61]. Текущая реализация данного подхода представлена в виде прототипа инструмента для обновления кода на языке C, который способен:

1. Добавлять и удалять включаемые файлы или изменять их имена;
2. Добавлять, удалять или переименовывать имена функций;
3. Перемещать функции между включаемыми файлами;
4. Добавлять, удалять и переупорядочивать параметры функций;
5. Модифицировать аргументы по умолчанию;
6. Изменять типы возвращаемых значений функций;
7. Изменять типы параметров функций;
8. Изменять значения входящих параметров функций;
9. Удалять, добавлять, переименовывать и изменять поля структур;
10. Удалять, добавлять, переименовывать и изменять глобальные переменные.

Статья также представляет низкоуровневый язык AST-DSL, позволяющий описывать миграции API в виде набора правил для разработанного прототипа инструмента. Например, правило для миграции сигнатуры функции от `assign(char*, char*, int=NPOS)` к `assign(char*, char*, int=0, int=NPOS)` представлено на листинге 32.

В поле `FNAME` указана функция, которая была изменена разработчиком. А в поле `PATTERN` задана синтаксическая структура вызова функции, которая может использоваться для сопоставления с любым вызовом. Если фрагмент целевого кода сопоставим с полем `PATTERN`, инструмент выполняет код, указанный в поле `ACTION`. В данном случае добавляется четвертый аргумент к функции со значением по умолчанию равным 0.

Разработанный прототип инструмента далек от завершения и способен выполнять лишь простейшие миграции программного кода, что ограничивает его использование для высокоуровневых миграций.

```
/*
* _ASE_BEGIN_
* FNAME = assign
* PATTERN = #(fc:FCALL_assign id:ID lp:L_PAREN arg1:gen_expr
    co1:COMMA arg2:gen_expr co2:COMMA arg3:gen_expr rp:R_PAREN)
* ACTION = #gen_logical_expr =
    #(#fc, #id, #lp, #arg1, #co, #arg2, #co, 0, #co2, #arg3, #rp);
* _ASE_END_
*/
```

Листинг 32. Правило трансформации на AST-DSL.
Listing 32. Transformation rule in AST-DSL.

3.2.3 Swing to SWT и обратно: шаблоны для миграции API с помощью переноса программного кода

В работе [62] рассматривается миграция между платформами Swing [63] и SWT Java GUI [64] с применением шаблона «Адаптер» [65]. Авторы выделяют несколько основных проблем, возникающих при адаптации одного API к другому:

1. Нетривиальность отображения методов (например, «один ко многим» и «многие ко многим»);
2. Различия в иерархиях классов;
3. Поиск соответствия между идентификаторами объектов;
4. Проблемы использования нового API целевой платформой.

Авторы приводят описание потенциальной реализации инструмента автоматической миграции, который позволяет создавать адаптационные оболочки для старого API в целях совместимости с новым.

3.2.4 Автоматизированное обновление API для приложений Android

В работе [66] представлен метод AppEvolve, предназначенная для автоматических обновлений использования Android API. Метод заключается в обновлении использования Android API в коде целевого приложения, путем заимствования опыта миграции программного кода других разработчиков на новые версии API. Авторы выделяют 4 основных этапа обновления приложения под новую версию Android API:

1. Анализ использования API: на этом этапе выполняется анализ исходного кода приложения с целью определения мест, требующих изменения.
2. Поиск примеров обновлений: здесь информация с первого этапа соединяется со спецификацией изменений API для поиска актуальных примеров обновлений.
3. Обновление примеров анализа: найденные примеры миграции API обрабатываются, обобщаются и ранжируются по степени соответствия нуждам пользователя.
4. Обновление использования API: на этом этапе используются примеры, отобранные на предыдущем этапе, для внесения обновлений в API, а также производится тестирование обновленного кода с помощью дифференциального тестирования.

Результаты, полученные в ходе исследования, носят теоретический характер и не нашли полноценное применения в программном решении обновления приложений при миграции Android API.

3.2.5 Влияние переноса API на качество и понимание программного обеспечения

В работе [67] исследуется влияние переноса API на качество и понимание программного обеспечения. Подход, разработанный авторами, основывается на:

- Правилах адаптации программного кода;
- Методах отображения старого API на новый;
- Адаптационном рефакторинге;
- Принципе связности;
- Принципе сплоченности.

Правила адаптации программного кода содержат описание пар «Источник» – «Цель», где «Источник» представляет исходную библиотеку или класс, а «Цель» – целевую библиотеку или класс, к которым требуется адаптация. Например, правило `ClassSrc – ClassDst` означает переноса класса `ClassSrc` в новый класс `ClassDst`.

Методы отображения определяются как процесс замены одного или нескольких классов или методов из исходной библиотеки классами целевой библиотеки.

Рефакторинг определяется как процесс изменения программного приложения, который улучшает качество приложения, не изменяя его поведения [68-69]. Рефакторинг является одним из наиболее распространенных методов повышения качества программного приложения [69-70]. Существуют различные операции рефакторинга, которые могут быть использованы для улучшения качества программного приложения, включая изменение типов параметров, перемещение атрибутов и методов, переименование переменных, извлечение методов и классов и другие.

Принцип связности подразумевает измерение уровня взаимосвязи между модулями программного приложения, где желательной является слабая связь (то есть меньшая зависимость между модулями) [71]. Для вычисления этой метрики можно использовать связь между объектами (Coupling Between Objects, CBO), соединённую с модулями. Чем выше CBO, тем выше класс связности приложения.

Принцип сплоченности заключается в необходимости измерения взаимосвязей внутри отдельных модулей программного приложения. Сильное взаимодействие между элементами повышает удобство сопровождения кода [71]. Для оценки сплоченности классов достаточно использовать одну метрику – нормализованное отсутствие сплоченности методов (Normalized Lack of Cohesion of Methods, LCOM) [72-73]. Метрика LCOM показывает, насколько методы связаны друг с другом через атрибуты. Если все методы обращаются к одинаковым атрибутам, значение LCOM равно 0, что свидетельствует о высокой связности методов внутри класса, в противном случае высокое значение LCOM указывает на слабую связь, что увеличивает вероятность ошибок при проектировании и разработке программного приложения.

Результаты, полученные в ходе исследования, носят теоретический характер и не нашли свое применения в полноценном инструменте обновления приложений при миграции API.

3.2.6 Адаптация программных приложений к новым версиям API от компании Google

Работа [74] содержит рекомендации от Google, направленные на перенос программного кода приложений, написанных под операционную систему Android, с устаревших версий API на более новые. Авторы охватываю ключевые изменения в API, а также новые функции и улучшения. Однако эти рекомендации касаются только ручной адаптации кода под новые версии API.

3.2.7 Другие исследования

Существует ряд статистических исследований, подчеркивающих актуальность инструментов миграции API и опыт их использования. Однако воспроизведение их результатов зачастую бывает затруднительным.

В работе [75] рассматриваются аспекты устаревшего Android API, в то время как работа [76] сосредоточена на устаревании API платформы Java Development Kit. Исследование [77] охватывает общие проблемы, связанные с API, включая масштаб изменений, время реагирования на изменения, и способы, которыми системы справляются с изменениями API. В работе [78] представлен обзор стабильности версий API и особенностей миграции в экосистеме Android, тогда как работа [79] исследует возможности миграции программного кода между языками программирования C# и Java. В работе [80] описывается метод миграции между реализациями функционально схожих библиотек для разных программно-аппаратных платформ с использованием языка программных аннотаций (Program Annotation Language – PanLang) собственной разработки. С помощью данного языка можно описать каждый функциональный элемент каждой библиотеки (например, глобальные переменные и функции), чтобы затем сопоставить их по семантическому смыслу при миграции программного кода, который их использует. В качестве примера авторы описывают миграцию кода, использующего сетевую библиотеку BSD сокетов в код, использующий библиотеку Winsock. Работа [81] содержит статистические данные о деструктивных изменениях API.

Также проводятся исследования трансформаций программного кода с использованием методов искусственного интеллекта и машинного обучения, разбор которых выходит за рамки данной работы – [82] и [83].

4. Выводы

Применимость рассмотренных инструментов обусловлена сложностью реализованных в них подходов. Каждый из рассмотренных инструментов будет полезен при осуществлении миграций, принадлежащих определенным классам из раздела 2.

Например, такие инструменты как ast-grep (см. раздел 3.1.16), Comby (см. раздел 3.1.9) или GoPatch (см. раздел 3.1.11) предоставляют функциональные возможности для простого поиска и замены языковых конструкций (выражений или отдельных операторов) как по исходному коду, так и по его внутреннему представлению (например, AST) без проведения семантических проверок. Эти инструменты будут полезны при решении простых задач по миграции или трансформации кода, таких как, переименование переменной или функции, или замены сигнатуры метода или функции. Таким образом, с помощью данных инструментов возможно производить автоматические миграции программного кода первого, третьего, четвертого и пятового подклассов низкоуровневого класса миграции API из таблицы 1 (см. раздел 2.6) для кодовых баз различного размера.

Инструменты, предоставляющие возможности для семантического анализа конструкций языка программирования перед их сопоставлением и трансформацией, обычно обладают DSL собственного проектирования и разработки, например, инструменты Coccinelle (см. раздел 3.1.1), Coccinelle4J (см. раздел 3.1.1) и язык PATL (см. раздел 3.1.2), или предоставляют библиотеку и инструментарий для этого, например, SPOON (см. раздел 3.1.8) или OpenRewrite (см. раздел 3.1.10). С помощью инструментов данной категории возможно проводить большинство миграций программного кода первого, третьего, четвертого и пятового подклассов низкоуровневого класса миграции, а также определенные виды сложных миграций программного кода второго подкласса низкоуровневого класса миграции и высокоуровневого класса миграций из таблицы 1. Но, автоматическая трансформация программного кода произвольной сложности без верификации и поддержки разработчиков

не осуществима из-за ограничений данных инструментов. В частности, ряд рассматриваемых инструментов, таких как Coccinelle и Coccinelle4J, не учитывает поток управления программы или граф наследования классов. Кроме того, некоторые из этих решений не предлагают достаточно выразительные языки для описания сложных правил семантического преобразования, что требуется дополнительной квалификации конечного пользователя для использования полного набора их возможностей. Это относится, например, к решениям Nobrainer (см. раздел 3.1.4), Refaster (см. раздел 3.1.6), Twining (см. раздел 3.1.3), SWIN (см. раздел 3.1.3), Proteus (см. раздел 3.1.7), Stratego (см. раздел 3.1.7) и Rascal (см. раздел 3.1.15).

Для различных задач миграции и трансформации программного кода, также может быть важно представление кода, с которым работает тот или иной инструмент. Не все анализируемые инструменты функционируют на уровне исходного кода программы. Например, инструмент ASM ориентирован на работу с байт-кодом, а не с исходным. В то же время Coccinelle предлагает интуитивно понятный для пользователя язык семантических преобразований SmPL, а Nobrainer реализует инновационный подход на основе механизма сопоставления абстрактного синтаксического дерева (AST) компилятора Clang.

Рассмотренные исследования на тему миграции API (подраздел 3.2) разделяются на работы, исследующие проблемы, связанные с миграцией API и работы, предлагающие методы решения проблемы миграции для конкретных случаев (например, миграция приложений при обновлении Android API (подраздел 3.2.4). Общей характеристикой всех рассмотренных исследований является преобладание достигнутых авторами теоретических результатов над практическими.

5. Заключение

В ходе проведенного исследования были отобраны и классифицированы примеры миграций приложений на языке Java под различные версии Android API. По отобранным примерам было выделено 2 типовых класса миграции API – низкоуровневый и высокоуровневый. Низкоуровневый класс миграции API был дополнительно классифицирован на 5 подклассов. Выделенные классы миграции носят общий характер и не привязаны к миграции Android API или языку Java. Их можно обобщить на случай миграции произвольной библиотеки или произвольного объектно-ориентированного языка. Далее были рассмотрены программные решения, предназначенные для решения задач автоматического и автоматизированного переноса программного кода в процессе миграции API для различных языков программирования, включая C, C++, Java и JavaScript, а также исследования, посвященные проблеме миграции API. Критерии рассмотрения инструментов включали возможность осуществления миграций из выделенных классов и подклассов миграции API, а также удобство использования их конечным пользователем, исходя из предположения, что правила миграций предпочтительнее описывать на целевых языках или на схожих с ними DSL.

Подводя итог, важно подчеркнуть, что ни одно из рассмотренных программных средств не предоставляет возможности для простого и полнофункционального автоматического переноса произвольного программного кода на конкретном языке программирования с точки зрения примеров произвольной сложности из каждого выделенного класса миграции API. Это свидетельствует о необходимости дальнейших исследований и разработок в данной области, направленных на создание более универсальных и эффективных инструментов для автоматизации миграции кода.

В табл. 2 представлен сравнительный список рассмотренных программных инструментов миграции API.

Табл. 2. Сравнительный список программных инструментов для миграции API.
Table 2. Comparison List of API Migration Software Tools.

Инструментарий	Тип лицензии	Поддерживаемые языки	Комментарий
Coccinelle	открытая	C	поддерживает трансформации только в пределах отдельных функций / подходит для осуществления большинства миграций, принадлежащих **низкоуровневому** классу миграций
Coccinelle4J	закрытая	Java	поддерживает трансформации только в пределах отдельных методов / подходит для осуществления большинства миграций, принадлежащих **низкоуровневому** классу миграций
Nobrainer	закрытая	C/C++	поддерживает трансформацию ограниченного подмножества конструкций целевого языка / подходит для осуществления миграций из **первого**, **третьего** и **четвертого подклассов низкоуровневого** класса миграций
MARTINI	открытая	C/C++	находится в разработке и поддерживает трансформацию ограниченного подмножества конструкций целевого языка/подходит для осуществления миграций из **первого**, **третьего** и **четвертого подклассов низкоуровневого** класса миграций
Refaster	открытая	Java	отсутствуют семантические проверки при сопоставлении/подходит для осуществления миграций из **первого**, **третьего** и **четвертого подклассов низкоуровневого** класса миграций
Proteus	закрытая	C/C++	отсутствуют семантические проверки при сопоставлении/подходит для осуществления миграций из **первого**, **третьего** и **четвертого подклассов низкоуровневого** класса миграций
SPOON	открытая	Java	поддерживает трансформацию ограниченного подмножества конструкций целевого языка / подходит для осуществления большинства миграций, принадлежащих **низкоуровневому** классу миграций
Comby	открытая	мультиязычный	пригоден лишь для проведения простых трансформаций/подходит для осуществления миграций из **первого**, **третьего** и четвертого **подклассов низкоуровневого** класса миграций
OpenRewrite	открытая	Java	сложность описания правил трансформации/подходит для осуществления большинства миграций, принадлежащих **низкоуровневому** классу миграций
GoPatch	открытая	Go	отсутствуют семантические проверки при сопоставлении/подходит для осуществления миграций из **первого**, **третьего** и четвертого **подклассов низкоуровневого** класса миграций
CodeTurn	закрытая	COBOL	поддерживает трансформацию ограниченного подмножества конструкций целевого языка/подходит для осуществления большинства миграций, принадлежащих **низкоуровневому** классу миграций
Rascal	открытая	Java	отсутствуют семантические проверки при сопоставлении/подходит для осуществления большинства миграций, принадлежащих **низкоуровневому** классу миграций
JSCodeshift	открытая	JavaScript	отсутствуют семантические проверки при сопоставлении/подходит для осуществления миграций из **первого**, **третьего** и четвертого **подклассов низкоуровневого** класса миграций
ClangMR	открытая	C/C++	требуются знания API Clang для проведения сложных трансформаций/ подходит для осуществления большинства миграций, принадлежащих **низкоуровневую** классу, а также с помощью него возможно осуществлять **высокоуровневые** миграции

Список литературы / References

[1]. What is an API? (Application Programming Interface). https://www.mulesoft.com/resources/api/what-is-an-api (accessed 01.09.2024).

[2]. CMake Documentation and Community. https://cmake.org/documentation/ (accessed 01.09.2024).

[3]. Android API levels. https://apilevels.com (accessed 01.09.2024).

[4]. Lamothe M., Shang W., Chen T. H. P. A3: Assisting android api migrations using code examples. IEEE Transactions on Software Engineering, 2020, vol. 48, issue 2, pp. 417-431, DOI: 10.1109/TSE.2020.2988396.

[5]. Top Programming Languages for Android App Development in 2023. https://medium.com/agileinsider/top-programming-languages-for-android-app-development-in-2023-3eb5d (accessed 01.09.2024).

[6]. Github. https://github.com/ (accessed 01.09.2024).

[7]. Rodriguez L. R. and Lawall J. Increasing Automation in the Backporting of Linux Drivers Using Coccinelle. In 11th European Dependable Computing Conference (EDCC), Paris, France, 2015, pp. 132-143, DOI: 10.1109/EDCC.2015.23.

[8]. Kang H. J., Thung F., Lawall J., Muller G., Jiang L., Lo D. Semantic Patches for Java Program Transformation (Experience Report). In 33rd European Conference on Object-Oriented Programming (ECOOP 2019). Leibniz International Proceedings in Informatics (LIPIcs), vol. 134, pp. 22:1-22:27, Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2019), DOI: 10.4230/LIPIcs.ECOOP.2019.22.

[9]. Chenglong Wang, Jiajun Jiang, Jun Li, Yingfei Xiong, Xiangyu Luo, Lu Zhang, and Zhenjiang Hu. Transforming Programs between APIs with Many-to-Many Mappings. In 30th European Conference on Object-Oriented Programming (ECOOP 2016). Leibniz International Proceedings in Informatics (LIPIcs), vol. 56, pp. 25:1-25:26, Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2016), DOI: 10.4230/LIPIcs.ECOOP.2016.25.

[10]. M. Nita and D. Notkin. Using twinning to adapt programs to alternative APIs. 2010 ACM/IEEE 32nd International Conference on Software Engineering, Cape Town, South Africa, 2010, pp. 205-214, DOI: 10.1145/1806799.1806832.

[11]. Jun Li, Chenglong Wang, Yingfei Xiong, Zhenjiang Hu. SWIN: Towards Type-Safe Java Program Adaptation between APIs. Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation, 2015, pp. 91-102, DOI: 10.1145/2678015.2682534.

[12]. Savchenko V. V., Sorokin K. S., Pankratenko G. A., Markov S., Spiridonov A., Alexandrov I., Volkov A. S., and Sun, K.-W. Nobrainer: An example-driven framework for C/C++ code transformations. Perspectives of System Informatics: 12th International Andrei P. Ershov Informatics Conference, PSI 2019, Novosibirsk, Russia, July 2–5, 2019, Revised Selected Papers 12, Springer International Publishing, 2019, pp. 140-155, DOI: 10.1007/978-3-030-37487-7_12.

[13]. Clang: a C language family frontend for LLVM. https://clang.llvm.org/ (accessed 01.09.2024).

[14]. The LLVM Compiler Infrastructure. https://llvm.org/ (accessed 01.09.2024).

[15]. AST Matcher Reference. https://clang.llvm.org/docs/LibASTMatchersReference.html (accessed 01.09.2024).

[16]. The Official YAML Web Site. https://yaml.org/ (accessed 01.09.2024).

[17]. Johnson A., Coti C., Malony A.D., Doerfert J. MARTINI: The Little Match and Replace Tool for Automatic Application Rewriting with Code Examples. In: Cano, J., Trinder, P. (eds) Euro-Par 2022: Parallel Processing. Euro-Par 2022. Lecture Notes in Computer Science, vol 13440. Springer, Cham., 2022, vol. 7, issue 76, pp. 4590, DOI: 10.1007/978-3-031-12597-3_2.

[18]. Wasserman L. Scalable, example-based refactorings with refaster. WRT'13: Proceedings of the 2013 ACM Workshop on refactoring tools, pp. 25-28, DOI: 10.1145/2541348.2541355.

[19]. Waddington D., Yao B. High-fidelity C/C++ code transformation. Science of Computer Programming, 2007, vol. 68, issue 2, pp. 64-78, DOI: 10.1016/j.scico.2006.04.010.

[20]. Visser E. (2001). Stratego: A Language for Program Transformation Based on Rewriting Strategies System Description of Stratego 0.5, 2001, pp. 357-361. Springer, Berlin, Heidelberg. DOI: 10.1007/3-540-45127-7_27.

[21]. Pawlak R. et al. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. Software: Practice and Experience, 2016, vol. 46, issue 9, pp. 1155-1179, DOI: 10.1002/spe.2346.

[22]. Long F., Amidon P., Rinard M. Automatic inference of code transforms for patch generation. Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017, pp. 727-739, DOI: 10.1145/3106237.3106253.

[23]. Saha R. K., Lyu Y., Yoshida H. and Prasad M. R. Elixir: Effective object-oriented program repair, 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), Urbana, IL, USA, 2017, pp. 648-659, DOI: 10.1109/ASE.2017.8115675.
[24]. Wen M., Chen J., Wu R., Hao D. and Cheung S.-C., Context-Aware Patch Generation for Better Automated Program Repair, 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), Gothenburg, Sweden, 2018, pp. 1-11, DOI: 10.1145/3180155.3180233.
[25]. Xuan J., Xie X., Monperrus M. Crash Reproduction via Test Case Mutation: Let Existing Test Cases Help. ESEC/FSE 2015 - 10th Joint Meeting on Foundations of Software Engineering, NIER Track, Aug 2015, Bergamo, Italy. pp.910-913, DOI: 10.1145/2786805.2803206.
[26]. Xuan J. et al. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. IEEE Transactions on Software Engineering, 2016, vol. 43, issue 1, pp. 34-55, DOI: 10.1109/TSE.2016.2560811.
[27]. Martinez M., Monperrus M. Astor: A program repair library for java. Proceedings of the 25th international symposium on software testing and analysis, 2016, pp. 441-444, DOI: 10.1145/2931037.2948705.
[28]. Van Tonder R., Le Goues C. Lightweight Multi-Language Syntax Transformation with Parser Parser Combinators. Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2019, pp. 363-378, DOI: 10.1145/3314221.3314589.
[29]. OpenRewrite: Semantic code search and transformation tool. Moderne Inc., https://docs.openrewrite.org/ (accessed 01.09.2024).
[30]. Introduction to Apache HttpClient. https://blog.knoldus.com/introduction-to-apache-httpclient/ (accessed 01.09.2024).
[31]. Lossless Semantic Trees (LST). https://docs.openrewrite.org/concepts-explanations/lossless-semantic-trees (accessed 01.09.2024).
[32]. go-patch: Structured code diffs and refactors. Uber Technologies, Inc. GitHub repository. https://github.com/uber-go/gopatch (accessed 01.09.2024).
[33]. Detailed Description of Unified Format. https://www.gnu.org/software/diffutils/manual/html_node/Detailed-Unified.html (accessed 01.09.2024).
[34]. CodeTurn: Automated Mainframe Code Conversion. https://www.astadia.com/products/codeturn (accessed 01.09.2024).
[35]. Cordy J. R. The TXL source transformation language. Science of Computer Programming, 2006, vol. 61, issue 3, pp. 190-210, DOI: 10.1016/j.scico.2006.04.002.
[36]. Jscodeshift. https://github.com/facebook/jscodeshift (accessed 01.09.2024).
[37]. Recast. https://github.com/benjamn/recast (accessed 01.09.2024).
[38]. AST-types. https://github.com/benjamn/ast-types (accessed 01.09.2024).
[39]. AVA. https://github.com/avajs/ava (accessed 01.09.2024).
[40]. React-codemod. https://github.com/reactjs/react-codemod (accessed 01.09.2024).
[41]. Lodash-codemods. https://github.com/jfmengels/lodash-codemods (accessed 01.09.2024).
[42]. ESTree. https://github.com/estree/estree (accessed 01.09.2024).
[43]. Using the Compiler API. https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API (accessed 01.09.2024).
[44]. Klint P., van der Storm T., Vinju J. Rascal: A Domain Specific Language for Source Code Analysis and Manipulation. 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, Edmonton, AB, Canada, 2009, pp. 168-177, DOI: 10.1109/SCAM.2009.28.
[45]. Wright H. K., Jasper D., Klimek M., Carruth C. and Wan Z. Large-Scale Automated Refactoring Using ClangMR, 2013 IEEE International Conference on Software Maintenance, Eindhoven, Netherlands, 2013, pp. 548-551, DOI: 10.1109/ICSM.2013.93.
[46]. Koppel J., Premtoon V., Solar-Lezama A. One tool, many languages: language-parametric transformation with incremental parametric syntax. Proceedings of the ACM on Programming Languages, 2018, vol. 2, issue OOPSLA, pp. 1-28, DOI: 10.1145/3276492.
[47]. Ketkar A. et al. A Lightweight Polyglot Code Transformation Language. Proceedings of the ACM on Programming Languages, 2024, vol. 8, issue PLDI, article No.: 199, pp. 1288-1312, DOI: 10.1145/3656429.
[48]. Max Brunsfeld Tree-sitter. GitHub. https://tree-sitter.github.io/tree-sitter/ (accessed 01.09.2024).
[49]. Ramanathan M. K., Clapp L., Barik R. and Sridharan M. Piranha: Reducing Feature Flag Debt at Uber, 2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), Seoul, Korea (South), 2020, pp. 221-230, DOI: 10.1145/3377813.3381350.

[50]. Baxter I. D. DMS: Program transformations for practical scalable software evolution. IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution, pp. 48-51, DOI: 10.1145/512035.512047.
[51]. Dagenais B., Robillard M. P. SemDiff: Analysis and recommendation support for API evolution. 2009 IEEE 31st International Conference on Software Engineering, Vancouver, BC, Canada, 2009, pp. 599-602, DOI: 10.1109/ICSE.2009.5070565.
[52]. Henkel J., Diwan A. CatchUp! Capturing and replaying refactorings to support API evolution. Proceedings, 27th International Conference on Software Engineering, 2005. ICSE 2005., St. Louis, MO, USA, 2005, pp. 274-283, DOI: 10.1109/ICSE.2005.1553570.
[53]. Eclipse. https://www.eclipse.org/ (accessed 01.09.2024).
[54]. Bruneton E., Lenglet R., Coupaye T. ASM: a code manipulation tool to implement adaptable systems. In Adaptable and extensible component systems, 2002, vol. 30, issue 19.
[55]. ast-grep: write code to match code. Open-source. https://ast-grep.github.io (accessed 01.09.2024).
[56]. Semgrep. Open-source. https://github.com/semgrep/semgrep (accessed 01.09.2024).
[57]. CodeQL: the libraries and queries that power security researchers around the world, as well as code scanning in GitHub Advanced Security. GitHub, Inc., https://codeql.github.com (accessed 01.09.2024).
[58]. Lamothe M., Shang W. Exploring the use of automated API migrating techniques in practice: an experience report on android. Proceedings of the 15th International Conference on Mining Software Repositories, 2018, pp. 503-514, DOI: 10.1145/3196398.3196420.
[59]. Chow, Notkin. Semi-automatic update of applications in response to library changes. 1996 Proceedings of International Conference on Software Maintenance, Monterey, CA, USA, 1996, pp. 359-368, DOI: 10.1109/ICSM.1996.565039.
[60]. Parr T. J., Dietz H. G., Cohen W. E. PCCTS reference manual: version 1.00. ACM SIGPLAN Notices, 1992, vol. 27, issue 2, pp. 88-165, DOI: 10.1145/130973.130980.
[61]. Parr T. J. An Overview of SORCERER: A Simple Tree-Parser Generator. Poster paper; International Conference on Compiler Construction 1994; Edinburgh, Scotland; April 1994.
[62]. Bartolomei T. T., Czarnecki K., Lämmel R. Swing to SWT and back: Patterns for API migration by wrapping. 2010 IEEE International Conference on Software Maintenance, Timisoara, Romania, 2010, pp. 1-10, DOI: 10.1109/ICSM.2010.5610429.
[63]. Java Swing Tutorial. https://www.javatpoint.com/java-swing (accessed 01.09.2024).
[64]. SWT: The Standard Widget Toolkit. https://www.eclipse.org/swt/ (accessed 01.09.2024).
[65]. Adapter Pattern. https://www.geeksforgeeks.org/adapter-pattern/ (accessed 01.09.2024).
[66]. Fazzini M., Xin Q., Orso A. Automated API-usage update for Android apps. ISSTA 2019: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 204-215, DOI: 10.1145/3293882.3330571.
[67]. Alrubaye H., Alshoaibi D., Alomar E., Mkaouer M. W., and Ouni A. How Does API Migration Impact Software Quality and Comprehension? An Empirical Study. 2019, DOI: 10.48550/arXiv.1907.07724.
[68]. Opdyke W. F. Refactoring object-oriented frameworks. University of Illinois at Urbana-Champaign, 1992.
[69]. Fowler M. Refactoring: improving the design of existing code. Addison-Wesley Professional, 2018.
[70]. Stroggylos K., Spinellis D. Refactoring – Does It Improve Software Quality? Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007), Minneapolis, MN, USA, 2007, pp. 10-10, DOI: 10.1109/WOSQ.2007.11.
[71]. Stevens W. P., Myers G. J., Constantine L. L. Structured design. In IBM Systems Journal, vol. 13, issue 2, pp. 115-139, 1974, DOI: 10.1147/sj.132.0115.
[72]. Pantiuchina J., Lanza M., Bavota G. Improving Code: The (Mis) Perception of Quality Metrics, 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), Madrid, Spain, 2018, pp. 80-91, DOI: 10.1109/ICSME.2018.00017.
[73]. Chávez A., Ferreira I., Fernandes E., Cedrim D., Garcia A. SBES '17: Proceedings of the XXXI Brazilian Symposium on Software Engineering, pp. 74-83, DOI: 10.1145/3131151.3131171.
[74]. Migrating your apps to Android 9. https://developer.android.com/about/versions/pie/android-9.0-migration (accessed 01.09.2024).
[75]. Li L. et al. Characterising deprecated Android APIs. MSR '18: Proceedings of the 15th International Conference on Mining Software Repositories, pp. 254 – 264, DOI: 10.1145/3196398.3196419.
[76]. Sawant A.A., Robbes R. & Bacchelli A. On the reaction to deprecation of clients of 4 + 1 popular Java APIs and the JDK. Empirical Software Engineering, vol. 23, issue 4, 2158–2197 (2018). DOI: 10.1007/s10664-017-9554-9.

[77]. Hora A., Robbes R., Anquetil N., Etien A., Ducasse S., Valente M. T. How Do Developers React to API Evolution? The Pharo Ecosystem Case, 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), Bremen, Germany, 2015, pp. 251-260, DOI: 10.1109/ICSM.2015.7332471.
[78]. Itsykson V., Zozulya A. Automated program transformation for migration to new libraries, 2011 7th Central and Eastern European Software Engineering Conference (CEE-SECR), Moscow, Russia, 2011, pp. 1-7, DOI: 10.1109/CEE-SECR.2011.6188463.
[79]. Brito A., Xavier L., Hora A., Valente M. T. Why and how Java developers break APIs, 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Campobasso, Italy, 2018, pp. 255-265, DOI: 10.1109/SANER.2018.8330214.
[80]. McDonnell T., Ray B., Kim M. An Empirical Study of API Stability and Adoption in the Android Ecosystem. 2013 IEEE International Conference on Software Maintenance, Eindhoven, Netherlands, 2013, pp. 70-79, DOI: 10.1109/ICSM.2013.18.
[81]. Pandita R., Jetley R. P., Sudarsan S. D., Williams L. Discovering likely mappings between APIs using text mining. 2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM), Bremen, Germany, 2015, pp. 231-240, DOI: 10.1109/SCAM.2015.7335419.
[82]. Li Y., Wang S., Nguyen T. N. DLFix: Context-based code transformation learning for automated program repair. ICSE '20: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 602 – 614, DOI: 10.1145/3377811.3380345.
[83]. Thongtanunam P., Pornprasit C., Tantithamthavorn C. Autotransform: Automated Code Transformation to Support Modern Code Review Process. ICSE '22: Proceedings of the 44th International Conference on Software Engineering, pp. 237 – 248, DOI: 10.1145/3510003.3510067.

Информация об авторах / Information about authors

Ян Андреевич ЧУРКИН – младший научный сотрудник отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ программ, компиляторные технологии, оптимизации.

Yan Andreevich CHURKIN – researcher in Compiler Technology department at ISP RAS. Research interests: static analysis, compiler technologies, optimizations.

Дмитрий Михайлович МЕЛЬНИК – старший научный сотрудник отдела компиляторных технологий. Научные интересы: компиляторные технологии, оптимизации.

Dmitry Mikhailovich MELNIK – Senior Researcher in Compiler Technology department. Research interests: compiler technologies, optimizations.

Рубен Артурович БУЧАЦКИЙ – кандидат технических наук, научный сотрудник отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ программ, компиляторные технологии, оптимизации.

Ruben Arturovich BUCHATSKIY – Cand. Sci. (Tech.), researcher in Compiler Technology department at ISP RAS. Research interests: static analysis, compiler technologies, optimizations.