DOI 10.37882/2223-2966.2024.7.33

ДЕКЛАРАТИВНЫЙ ПОДХОД К ГЕНЕРАЦИИ ГОТОВЫХ ПРИЛОЖЕНИЙ

DECLARATIVE APPROACH TO GENERATING READY-TO-USE APPLICATIONS

B. Rizoev

Summary. With modern development tools, project implementation has become a relatively easy task that can be automated. Considering that most applications involve input data, transforming data, enriching it with other data, and returning output data, we can take automation even further by making it declarative. This approach allows users to describe declaratively what they want to achieve, and then generate a ready-to-use application based on these descriptions. To accomplish this, it is necessary to create a domain-specific language (DSL) that has the necessary constructs to enable users to describe business tasks at a high level of abstraction. The generated description can be used as documentation for the application. This description is then transformed into a fully functional application using templates. This approach saves time in application development and eliminates the need for repetitive testing, as the templates only need to be tested once. Additionally, the proposed approach helps to unify the codebase, making it easy to change the technology stack, thus making the solution platform-independent. The second part of the paper presents the syntax of the proposed DSL and provides examples of templates used to create a typical application.

Keywords: code generation, templates, declarative approach, development automation.

Ризоев Бахтовар Махмадалиевич
НИУ Московский авиационный институт
mavlono.rb@gmail.com

Аннотация. С современными инструментами разработки реализация проектов превратилась в относительно легкую задачу, которую можно автоматизировать. Учитывая, что большая часть приложений — это входные данные, преобразование этих данных, обогащение другими данными и возвращение выходных данных, то можно пойти еще дальше и автоматизацию сделать декларативной — дать пользователю возможность декларативно описывать, что он хочет получить, а после генерировать готовое приложение. Для этого нужно создать специфический для предметной области язык, который будет обладать необходимыми конструкциями, позволяющими пользователям описывать бизнес задачи на высоком уровне абстракции. Сформированное описание можно использовать как документацию к приложению. После такое описание с помощью шаблонов превращается в полностью готовое приложение, которое можно использовать. Такой подход позволяет сэкономить время на написание приложений и избавляет от необходимости тестирования, так как нужно протестировать шаблоны всего один раз. Также предложенный подход помогает унифицировать кодовую базу, в которой к тому же легко поменять стек технологий — это делает решение платформонезависимым. Во второй части работы приведен синтаксис рассматриваемого специфического языка и приведен пример шаблонов — они используются для создания типичного приложения.

Ключевые слова: генерация кода, шаблоны, декларативный подход, автоматизация разработки.

Введение

Прогрессивное развитие информационных технологий и постоянное увеличение количества компьютерных систем в нашей повседневной жизни приводит к необходимости проектирования и разработки большого числа приложений, покрывающих различные аспекты работы предприятий, учреждений и организаций различных сфер и направлений [1]. Это подталкивает на создание большого количества однотипных приложений, которые выполняют сходные функции, но имеют различные вариации в связи с отсутствием общих принципов разработки. Одновременно с этим мы имеем высокую динамику развития инструментов разработки, фреймворков и библиотек, которая также создает цикл обновления приложений.

Задача разработчика в такой высокоразвитой области упрощается, но порождает рутину из написания одних и тех же блоков кода, их тестирования и эксплуатации. Также не стоит забывать, что приложения постоянно обновляются и дополняются новым функционалом. Если учитывать общие свойства проектов, то можно автоматизировать этот процесс, чтобы позволять разработчикам акцентироваться больше на архитектуре приложений, думая о том, как масштабировать и наиболее оптимально построить высоконагруженные и стабильные сервисы.

Целью данного исследования является разработка эффективного инструмента для автоматического создания готовых приложения с помощью декларативного подхода. Для достижения этой цели нужно выработать методологию декларативного описания приложений и разработать сам генератор.

Литературный обзор

В разработке приложений широко применяются различные подходы, среди которых можно выделить Model-

Driven Engineering (MDE) и Domain-Specific Languages (DSL).

Model-Driven Engineering (MDE) является парадигмой разработки приложений, в которой процесс создания программного обеспечения основывается на моделях различных аспектов его функционирования и архитектуры [2]. MDE подразумевает работу с высокоуровневыми абстракциями и автоматическую генерацию результатов на основе этих моделей. Этот подход делает акцент на повышении уровня абстракции в разработке ПО и позволяет снизить зависимость от конкретной технологии, упростить процесс разработки, сократить затраты и повысить эффективность.

Domain-Specific Languages (DSL) — это специально разработанные языки программирования, созданные для решения конкретных задач в определенной предметной области [3]. В сравнении с общими языками программирования, DSL обеспечивают более высокий уровень абстракции, что может сделать процесс разработки приложений проще и быстрее. Однако, ограниченность DSL может затруднять их применение в различных контекстах и при интеграции с другими технологиями.

Декларативное программирование основывается на задании требований и ожидаемой функциональности приложения, а не его конкретной реализации. Это позволяет автоматизировать генерацию кода, снижая трудозатраты на разработку и улучшая качество решений. На практике довольно часто применяется такой подход к генерации [4], а также проводятся исследования на предмет декларативных шаблонов проектирования [5].

Также существуют на коммерческой основе инструменты, позволяющие графически или в тексте создавать свои приложения без непосредственного написания кода, например, [6]. Однако их функциональность ограничена и чаще всего они без открытого исходного кода, что затрудняет их модификацию и использование для специфических задач.

Материалы и методы

Разумеется, из команды в команду различается процесс разработки, но объединяющие этапы присутствуют [7]. Так, сбор требований, их анализ, называемый процессом аналитики, мы никак не оптимизируем. Мы рассматриваем процесс написание документации, разработку и тестирование. Для первого нужно создать стандарт, который, с одной стороны, будет легко читаем, не сложен в написании и при этом поддерживает все необходимое для понимания и написания приложения. Это также решает проблему того, что даже в рамках одной команды обычно в разных проектах одни и те же действия описываются по-разному, что приводит к расхождениям.

Для документации был выбрал формат Emacs Org Mode [8]. Emacs — семейство многофункциональных расширяемых текстовых редакторов. Инструмент славится своими настраиваемостью и расширяемостью — в него легко можно добавить огромное количество плагинов и макросов, сделав его удобным под свои нужды. Org Mode — это текстовая версия ToDoList, это режим Emacs для содержания коротких заметок, TODO-списков, для планирования проектов, а также для организации любой информации, которую можно представить текстом и в виде дерева.

Декларативное описание — это спецификация сервиса (приложения), которую мы попытались максимально приблизить к документации сервиса. То есть нам нужен формат, в котором скрыты технические детали, но при этом есть достаточно инструментов для реализации бизнес-задач. Однако это не значит, что пользователь этого описания не должен иметь представление о многих технических деталях. Большинство приложений имеют структуру: входные данные, обработка этих данных, обогащение другими данными и выходные данные. Это позволяет нам определить модель приложения.

Описание имеет древовидную структуру, узлами которой являются логические модули, которые решают необходимую задачу. Пример таких модулей: обслуживание веб-запросов, работа с кафка-очередями. Модули в свою очередь состоят из следующих пунктов:

- тип
- конфигурация *
- название
- свойства
- блок кода *

Тип модуля определяет, какой шаблон использовать для генерации такого модуля, в то время как название является уникальным идентификатором. * обозначает, что данный пункт необязателен. Конфигурация определяет источник данных для некоторых модулей. Например, для баз данных — это файл конфигурации подключения к БД. Свойства определяются пользователем и используются непосредственно в шаблонах. Либо для их выбора, либо для кастомизации. Например, для модуля взаимодействия к БД — это свойство dbtype, которое определяет, какая БД используется и какой шаблон БД, соответственно, нужно выбрать. Блок кода описывает вставку пользовательского кода в шаблон. Например, это запрос БД.

Приложение — это множество модулей, которые выполняют свои функции. Главный модуль, объединяющий все остальные модули в приложение, находится в самой вершине дерева. После идут операции — это основная логика приложения, они определяют последовательность действий при получении входного запроса или со-

общения. Такие модули включают в себя другие. Отсюда появляется вложенность и древовидная структура.

Модули бывают двух видов:

- пользовательские — непосредственно описываются в документации
- служебные — генерируются на основе данных из документации, но не напрямую задаются в нем

Служебные модули необходимы для работы сервиса и скрывают в себе часть его реализации. Отвечают они, к примеру, за сбор логов, метрик, обработку исключений и последовательность выполнения действий. Пользовательские модули реализовываются разработчиками под определенный стек технологий. Генератор позволяет определять новые модули, которые написаны в определенном формате.

Сложно представить себе современное приложение без условий и различных веток выполнения кода. Для этого в документации есть специальный функционал, который позволяет после модуля перейти в любой другой внутри операции. По умолчанию последующий модуль будет следующим. Внутри этот механизм реализован посредством конечного автомата, где состояния — это различные модули, а условия переходов отвечают за условия между ними.

Перейдем к шаблонам — это структура или фрагмент кода в программировании, рассчитанный на повторное использование, в котором определенные части могут быть изменены или настроены при помощи маркеров-заполнителей (placeholders). Маркеры-заполнители — это специальные выражения или символы, которые служат в качестве заполнителей для будущих значений или данных. Такой инструмент является основой для генерации кода и широко используется, например, [9]. Для более свободного использования в шаблонах есть не просто метки, куда будут вставлены значения, а также механизм преобразования этих значений. Шаблоны, как и сам проект, написаны на языке программирования Java. В приведенных примерах используется фреймворк Spring [10]. Однако генератор позволяет писать шаблоны для любого языка и фреймворка, ведь он достает значения из документации по стандарту, а в шаблонах метки заменяются на значения — таким образом, шаблоны никак за исключением меток не преобразуются во время генерации, это и позволяет писать на любом языке и фреймворке шаблоны и сами приложения.

Шаблоны состоят из двух частей — самого шаблона и файла конфигурации шаблона. Шаблон описан выше, а конфигурация — это описание, откуда взять шаблон и куда его положить. Также в файле конфигурации есть возможность объединять файлы с помощью последнего механизма. Например, это нужно для файла pom.xml для Maven в Java, где указывается информация для программного проекта. Этот механизм позволяет указывать какие строки включать в конечный файл и как их сравнивать в случае, если такие строки уже существуют.

В свою очередь работа генератора заключается в синтаксическом анализе документации, которая была описаны выше, а потом подстановки значений в шаблоны, которые также описаны выше. На первый взгляд это легкая задача, но при детальном разборе появляется ряд проблем, если спроектировать что-то неправильно.

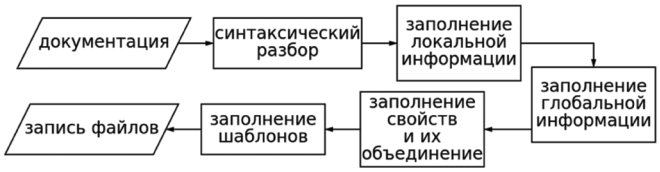

Рис. 1. Блок-схема генератора

После синтаксического разбора идет заполнение информации, которая находится внутри самого модуля — это и есть локальная информация. Затем следует заполнение глобальной информации: например, условные переходы, для заполнения которых нужна информация о всей операции. После идет заполнение свойств — тех, которые определяются пользователем, и тех, которые нужны для генерации. Некоторые свойства могут повторяться, поэтому нужно дополнительно объединять их. Теперь есть вся необходимая информация для заполнения шаблонов — это следующий шаг. И финальное действие — создание этих файлов и размещение их по папкам. Нужно учесть, что в ходе всех этих шагов могут возникнуть ошибки, которые приведут к остановке программы и прекращению генерации. Пользователю нужно исправить эти ошибки и заново попробовать сгенерировать приложение. Таким образом мы получаем готовое приложение, в которое нужно вставить файлы конфигурации. Для Spring Java — application файлы.

Результаты

Разработанное декларативное описание легче всего разбирать на примере. Рассмотрим следующую документацию:

```
 * приложение dec-test-app
 - contract_path: contract.yaml
 - entry: openapi
 - description: Сервис взаимодействия с коммуникациями
 - version: 0.0.1
 - url: /dec_app

 ** веб-операция getAllCommunications
 ** кафка-операция insertCommunications
  - consumer_type: json
  - topic_name: communication.input
  - auto_startup: true
  - retry: false
 ** веб-операция sendNewClientId
 ** функции
```

Рис. 2. Документация приложения без подробностей

Как видно, древовидная структура достигается с помощью *, что является стандартом для формата org. Одной * всегда определяется модуль всего сервиса — в нем перечислены самые верхнеуровневые свойства. Двумя ** определяется операция — это модуль, который объединяет в себя несколько модулей логики и определяет последовательность действий при получении входного запроса или сообщения. Также двум ** соответствует модуль функций — это область, где перечислены пользовательские функции, про них подробнее будет написано ниже. Три и более *** обозначают принадлежность модуля операциям.

Каждый модуль обладает своим синтаксисом, свойствами и всегда принимает данные и возвращает данные. Исключением для последнего являются модули кафка-операции и кафка-продюсера, которые только принимают данные.

В главном модуле — приложении — свойства description и version не влияют никак на приложение. Интересное свойство — это entry, которая в данном случае равна openapi. OpenAPI представляет собой стандарт для описания и документирования RESTful API, который обеспечивает универсальность, расширяемость и поддержку инструментов для создания, развертывания, интеграции и тестирования веб-сервисов [11]. В данном случае мы в нужном формате пишем контракт — документ, содержащий описание API с точки зрения входных параметров, вызываемых методов (операций), ответов и всех связанных деталей. После с помощью плагинов в готовом приложении создаются интерфейсы — именно их и переопределяет генератор, добавляя туда логику, которую пользователь в веб-операциях описал. Свойство contract_path определяет путь до контракта openapi, а url определяет корневой URL приложения.

Рассмотрим первую операцию более подробно (см. рис. 3).

Веб-операции является веб-сервером. Openapi по контракту генерирует интерфейс, который переопределяется шаблоном веб-операции. У модуля веб-операции нет никаких свойств, только вложенные модули. Конвертер — это модуль, который отвечает за преобразование данных. Например, в данном случае достает из query параметров запроса строку и присваивает другой строке. Также модуль позволяет применять над данными пользовательские функции, которые описаны в модуле функций. В отличие от привычно сгенерированных веб-серверов openapi в генератор заложены шаблоны, которые позволяют обращаться к данным запроса напрямую, что более удобно: bodyParam, queryParams и pathParams обозначают тело запроса, query и path параметры соответственно. Отдельно хочется обратить внимание на таблицы, которые используются в конвертере — это еще одно преимущества формата org, который «на лету» рисует такие таблицы. Название колонок таблицы говорят сами за себя.

Вложенный модуль «база» представляет модуль базы данных. В данном случае конфигурация БД, Communication, значит, что только один раз создается файл конфигурации, а дальше можно его переиспользовать. В приложении могут быть несколько видов БД, и, разумеется, работа с ними в коде тоже различается — именно поэтому в модуле БД есть свойство dbtype, которая обозначает тип БД. Свойство strategy имеет два значения: all, first — которые обозначают выборку всех результатов запроса или только первого соответственно. Далее идет блок кода, начало и конец которого ограничиваются соответствующими маркерами.

```
** веб-операция getAllCommunications
*** конвертер GetAllCommunicationsApiRequest -> DBSelectInput
| Input Name         | Input Type | -> | Output Name | Output Type | Описание |
|--------------------+------------+----+-------------+-------------+----------|
| queryParams.userId | String     |    | userId      | String      |          |
|--------------------+------------+----+-------------+-------------+----------|
 - MDC: queryParams.user_id:userId

*** база Communication GetCommunications
 - dbtype: postgres
 - strategy: all
#+BEGIN_SRC sql
SELECT * FROM communication WHERE user_id = :user_id
#+END_SRC

*** (end) конвертер DBSelectOutput -> GetAllCommunicationsApiResponse
| Input Name    | Input Type | ->                | Output Name                          | Output Type | Описание |
|---------------+------------+-------------------+--------------------------------------+-------------+----------|
| rows[].id     | Long       |                   | bodyParam.communications[].id        | Long        |          |
| rows[].name   | String     |                   | bodyParam.communications[].name      | String      |          |
| rows[].status | String     |                   | bodyParam.communications[].status    | String      |          |
|---------------+------------+-------------------+--------------------------------------+-------------+----------|
| rows[].time   | Timestamp  | timestampToString | bodyParam.communications[].time      | String      |          |
|---------------+------------+-------------------+--------------------------------------+-------------+----------|
```

Рис. 3. Подробности первой операции

```
@Slf4j
@Component
@RequiredArgsConstructor
public class ${#property methodId}Logic {
    private static final RowMapper<${#property mapperOutputType}> mapper = new RwMapper();
    private final NamedParameterJdbcTemplate namedParameterJdbcTemplate${#property config};

    public ${#property outputType} logic(${#property inputType} input) {
        var parameters = new MapSqlParameterSource();
        ${#template filling_parameters[idx]:###
        parameters.addValue("${#property \lowerCase{\camelusToSnake{daoInputFieldNames}}[__idx__]}",
                            input.get${#property \upFirst{daoInputFieldNames}[__idx__]}());###}
        ${#conditional [${#property strategy} == "first"]:###
        var response = namedParameterJdbcTemplate${#property config}
            .queryForObject("${#property codeBlock}", parameters, mapper);
        var response_json = new Gson().toJson(response);
        log.debug("${#property methodId} response: " + response_json);
        return response;###}

        ${#conditional [${#property strategy} == "all"]:###
        var list = new ArrayList<${#property mapperOutputType}>(
            namedParameterJdbcTemplate${#property config}.query(
                "${#property codeBlock}", parameters, mapper));
        var response_json = new Gson().toJson(list);
        log.debug("${#property methodId} response: " + response_json);
        var output = new ${#property outputType}();
        output.set${#property \upFirst{mapperOutputName}}(list);
        return output;###}
    }
}
```

Рис. 4. Шаблон базы данных

Если опустить импортирование зависимостей в файл, то шаблон выглядит так (см. рис. 4).

В приведенном шаблоне есть следующие маркеры за исключением Java кода: property methodId — название модуля; property outputType — название выходного объекта модуля и т.д.

Маркер template filling_parameters[idx] — начало цикла, шаблон, который цикл заполняет начинается с трех ### и заканчивается ими же. В данном случае это поля, которые отмечены в модуле конвертера после модуля базы в документации — то есть таблица в генераторе представлена списком этих полей.

Не менее интересным маркером является conditional, который завязывается на проверке какого-то свойства — в данном случае свойства стратегии. В зависимости от нее по-разному происходит обработка ответа базы данных.

Так выглядит файл конфигурации шаблона (см. рис. 5).

Как видно, присутствуют пути откуда брать шаблон и куда сохранить готовый файл. Также сама конфигурация является шаблонизируемым: в путь подставляются значения в соответствующие маркеры.

Само же приложение представляет из себя типичное Java приложения на фреймворке Spring. Каждая операция представляет собой конечный автомат, который содержит в себе словарь всех модулей операции, так как данные одного модуля могут быть использованы в другом модуле. Структура проекта тоже похожа на типичную (см. рис. 6).

Обсуждение

С помощью разработанного метода генерации готовых приложений можно значительно сократить время

```
transfers:
  - sourceLocation:
      origin: INTERNAL
      path: "/config_templates/datasourceConfig"
    targetLocation:
      path: "/src/main/java/${#property groupId}/${#property \withoutDashes{projectName}}/config/db/${#property config}DsConfig.java"

  - sourceLocation:
      origin: INTERNAL
      path: "/config_templates/hikariConfig"
    targetLocation:
      path: "/src/main/java/${#property groupId}/${#property \withoutDashes{projectName}}/config/db/${#property config}HikariConfig.java"
```

Рис. 5. Конфигурация шаблона базы данных

Рис. 6. Структура созданного проекта

для решения рутинных задач, а при должном внимании к данному подходу и вовсе обходиться без разработчика для создания простых сервисов.

Разумеется, нужна работа разработчика на начальном этапе, чтобы настроить правильно генератор, написать шаблоны под нужный стек технологий и протестировать их. Однако после такой начальной настройки, которая происходит один раз, даже аналитик без глубоких знаний разработки может сгенерировать приложение, так как ему всего лишь нужно написать документацию в нужном формате. Стоит подчеркнуть, что документацию аналитик пишет для любого сервиса, поэтому в данном случае это не является дополнительной работой, но при этом помогает поддерживать некий стандарт для написания документация.

В среднем время для разработки простых сервисов при должном понимании генератора и декларативного описания без учета начальной настройки может сократиться на более, чем 50%. Данная цифра подтверждается на практике и основана на генерации веб-сервисов и сервисов, поддерживающий кафка-очереди.

Ограничением декларативного генератора приложений является сложность генерации приложений с уникальной запутанной логикой. При написании больших приложений появляется сложность в виде значительного роста файла декларативного описания. Однако для большинства типовых приложений и стандартных интеграций разработанный генератор позволяет заметно сократить время на разработку.

Заключение

Таким образом, исходная цель достигнута — был разработан эффективный инструмент для автоматического создания готовых приложений. Выработана методология декларативного описания приложений, которая предоставляет широкие возможности для документирования. Также разработан генератор, позволяющий генерировать проекты на основе декларативного подхода с помощью шаблонов.

Декларативный генератор приложений показывает хороший потенциал для дальнейшей адаптации и расширения. В современных требованиях к стеку, который довольно часто меняется и обновляется, такой подход позволяет сэкономить приличное количество времени, так как при смене библиотек и фреймворков меняется не само описание, а только шаблоны.

Одним из направлений для дальнейшего улучшения декларативного генератора приложений является разработка графического пользовательского интерфейса, который позволит более наглядно видеть связи между модулями и проще редактировать документацию, которая сейчас только в текстовом виде.

Благодаря разработке и применению декларативного генератора приложений, данное исследование вносит вклад в упрощение процесса разработки ПО, способствуя сокращению затрат и времени на создание типовых приложений.

ЛИТЕРАТУРА

1. L. Yang, Q. Liu. Brief Talk about the Modern Application of Computer Software Technology and the Development Trend of Research // Journal of Physics Conference Series 1992. 2021.
2. M. Brambilla, J. Cabot, M. Wimmer. Model-Driven Software Engineering in Practice // Synthesis lectures on software engineering. 2012. Lecture No 4. pp. 2328–3327.
3. G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, S. Völkel. Design Guidelines for Domain Specific Languages // OOPSLA Workshop on Domain-Specific Modeling. 2009. Vol. 9.
4. Корытов П.В. Беляев С.А. Опыт создания программа автоматической генерации веб-приложения по формальным требованиям // Cloud of science. 2020. Т. 7. № 3. С. 559–576.
5. Mohan A., Jayaraman S., Jayaraman B. A declarative approach to detecting design patterns from Java execution traces and source code // Information and Software Technology. 2024. Vol. 171. P. 107457.
6. R. Barton. Talend Open Studio Cookbook. UK: Packt Publishing, 2013. 270 p.
7. Саяпин О.В., Тиханычев О.В., Безвесильная А.А. Организационные проблемы реализации гибких подходов в разработке прикладного программного обеспечения // Программные продукты и системы. 2022. Т. 35. № 4. С. 533–540.
8. C. Dominik, B. Guerry. Org Mode // Org Mode official site. [Электронный ресурс]. Режим доступа: https://orgmode.org/index.html: (дата обращения: 01.05.2024).
9. I. Ullah. I. Inayat. Template-based Automatic code generation for Web application and APIs Using Class Diagram // International Conference on Frontiers of Information Technology (FIT). 2022. pp. 332–337.
10. G. Walls. Spring in Action, Six Edition. NY: Manning Publications, 2022. 520 p.
11. J.S. Ponelat, L.L. Rosenstock. Designing APIs with Swagger and OpenAPI. NY: Manning Publications, 2022. 426 p.