Постольник: Ультра-легкий процессуальный движок

Дмитрий Писаренко, d.a.pisarenko@bk.ru

Зачем еще один процессуальный движок?

За что мы любили Камунду 7?

Кроме прочего за то, что с ее помощью можно сначала

1. сделать набросок той или иной функциональности (диаграмма BPMN), а потом
2. обогатить эту диаграмму техническими подробностями.

Это дает ряд преимуществ. Можете смеяться, но рисование квадратиков помогает понять, как подступиться к той или иной задаче.

А если речь про большую компанию, то такие диаграмму повышают автобусный фактор – шансы разобраться как работает тот или иной кусок кода появляются не только у разработчика, но и у аналитика или тестировщика.

Возникает вопрос: Если визуальное программирование а ля Камунда 7 так хорошо, почему мы не используем такой подход всегда?

Здесь ограничение чисто техническое. Камунду 7 делали для определенных задач. Судя по всему, она никогда не задумывалась как средство о-BPMN-ить все, что движется (а что не движется – по-CMMN-ить).

Представьте себе, что мы хотим написать развесистый шелл-скрипт, а чтобы его было легче поддерживать подумываем об использовании Камунды 7 ради диаграмм.

Сразу возникает ряд проблем.

Во-первых, для Камунды 7 нужна база. Что, если для нашей задачи не нужно сохранять процессы в базе?

Во-вторых, Камунду 7 обычно используют в рамках спрингового приложения. Это значит, что шелл-скрипт на базе Камунды может стартовать несколько минут. Конечно, можно заменить Спринг на Микронавт и тогда запуск станет быстрее. Но все равно это костыль, а зная Камунду 7 можно предположить, что при работе с Микронавтом возникнут неочевидные проблемы.

В-третьих, если в вашей команде рецензирование кода делают по гамбургскому счету, а не для галочки, гарантированно возникнет критик, который скажет: Вы ради рисования диаграммок прицепили кучу кода с ненужными зависимостями (база, веб-приложения, аудит и прочее), функциональностью (автоматические повторы активностей в случае ошибок) и, самое главное, уязвимостями?

Критик будет прав.

Ультралегкий процессуальный движок нужен для того, чтобы

1. можно было отображать функциональность приложения визуально
2. для тех вариантов использования, где Камунда 7 непригодна из-за избыточной функциональности.

То есть, чтобы были диаграммы, как в Камунде 7, но не было ненужного функционала. Так, чтобы можно было вместе с аналитиками шелл-скрипты рисовать и разрабатывать.

Ничего подходящего я не нашел, поэтому решил сделать сам.

Сразу встал вопрос:

Где взять визуальный редактор?

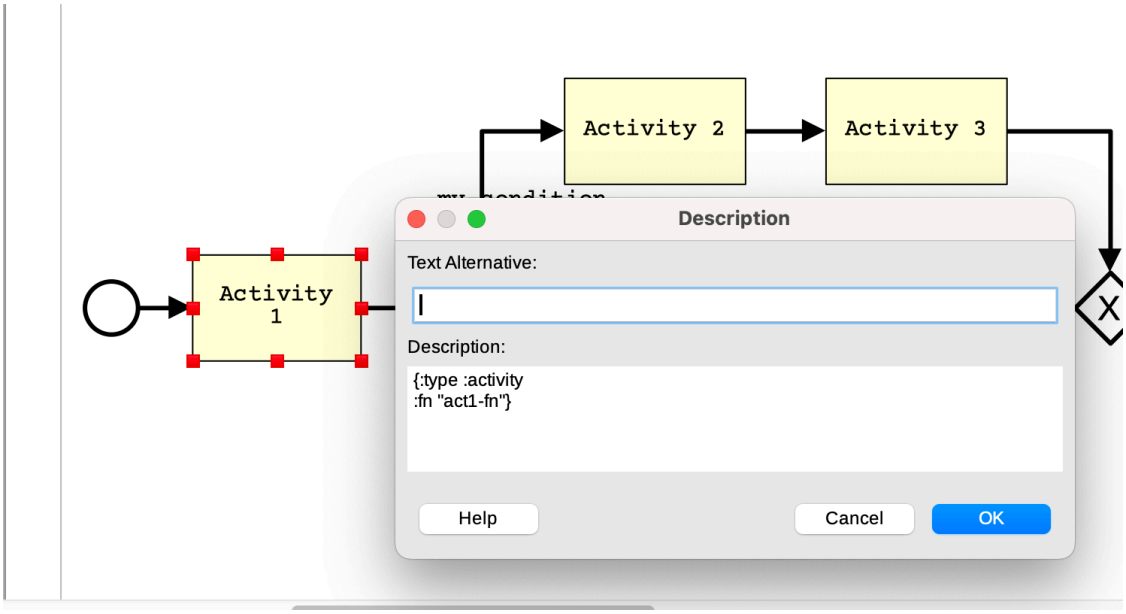
Камунда Моделер использовать не хочется, т. к. доступ к нему могут в любой момент закрыть из-за неправильной формы носа пользователя.

Нужно нечто такое, что

1. может понравится аналитикам и
2. где можно добавлять свои данные к фигуркам.

Я решил использовать плоский формат XML открытого офисного приложения *LibreOffice Draw*. По умолчанию LibreOffice Draw сохраняет данные в файлах ODG, которые являются ничем иным, как сжатым файлом XML. Можно тот же файл сохранить в формате FODG и тогда это будет обычным XML-файлом.

А еще я обнаружил, что если щёлкнуть правой кнопкой по фигуре, то можно ввести текст в поле `Description`.



Этот текст сохранится в XML-файле (тег `<svg:desc>`).

```
  <!-- ... -->
  <office:drawing>
   <draw:page draw:name="page1" draw:style-name="dp2" draw:master-
page-name="Default">
    <draw:custom-shape draw:name="process1" draw:style-name="gr1"
draw:text-style-name="P2" xml:id="id1" draw:id="id1"
draw:layer="layout" svg:width="3.027cm" svg:height="1.905cm"
svg:x="3.37cm" svg:y="7.985cm">
     <svg:desc>{:type :activity
:fn &amp;quot;act1-fn&amp;quot;}</svg:desc>
  <!-- ... -->
```

Читаем данные из диаграмм

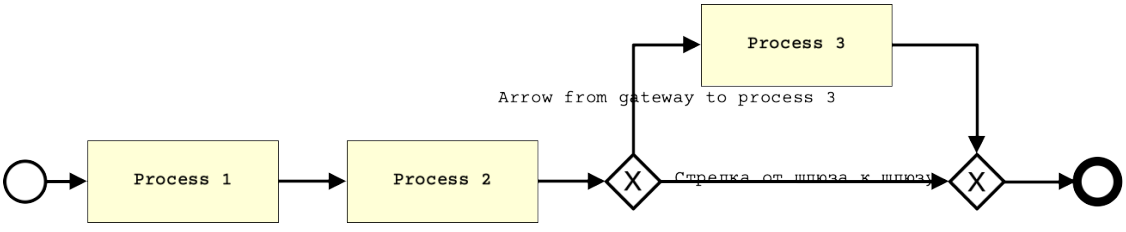
Теперь мы можем прочитать эти данные с помощью XPath. В библиотеке lo-draw-reader (Libre Office Draw reader) есть тест демонстрирующий ее работу.

Разберем его по частям.

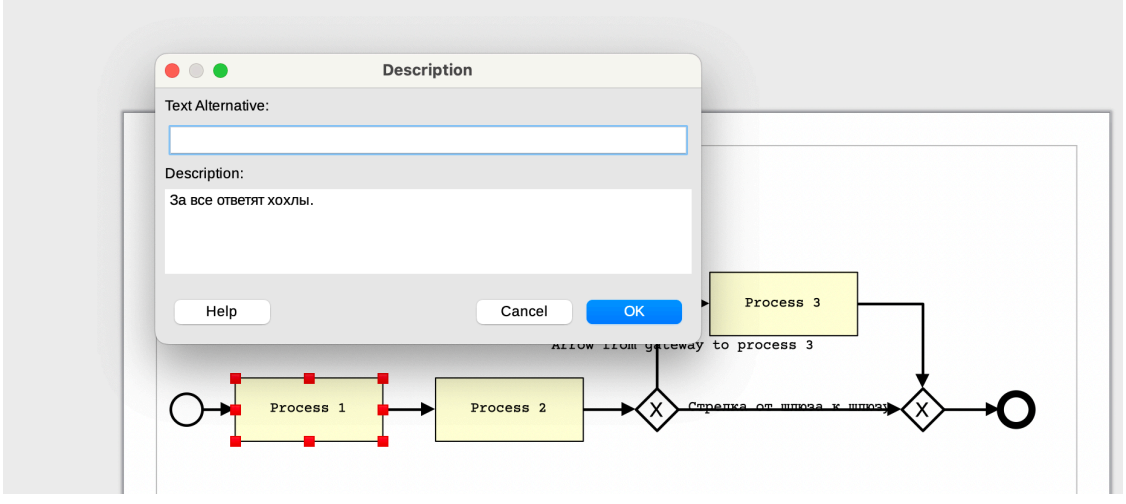
Сначала создаем поток для файла с диаграммой и передаем его в метод для извлечения из нее данных.

```
try (final InputStream is =
             FileUtils.openInputStream(new File("src/test/resources" +
                     "/process_diagram.fodg.xml"))) {
    // Given
    final LibreOfficeDrawParser sut = new LibreOfficeDrawParser();
    // When
    final LibreOfficeDrawParsingResult actualResult = sut.read(is);
```

Диаграмма выглядит так:



В поле `Description` прямоугольника `Process 1` есть текст:



В результате работы библиотеки мы получаем перечень

1. вершин (`actualResult.getVertices()`) и
2. ребер (`actualResult.getEdges()`)

нашей диаграммы.

У вершин в свойстве `description` сохраняется текст, введенный в LibreOffice Draw.

```
  final LibreOfficeDrawParsingResult actualResult = sut.read(is);

// ...

    final List<Vertex> vertices = actualResult.getVertices();

// ...

     assertTrue(vertices.contains(Vertex.builder()
             .id("id1")
             .name("process1")
             .description("За все ответят хохлы.")
         .build()));
```

Для ребер в свойстве `label` сохраняется текст на стрелке.

```
final List<Edge> edges = actualResult.getEdges();

// ...

assertTrue(edges.contains(Edge.builder()
   .source("id3")
   .target("id4")
   .label("Стрелка от шлюза к шлюзу")
   .build()));
assertTrue(edges.contains(Edge.builder()
   .source("id3")
   .target("id5")
   .label("Arrow from gateway to process 3")
   .build()));
```

Что это дает?

Поле `description` позволяет вводить данные активностей (как в моделере, только текстом), а надписи на стрелках – условия.

Пришла пора определить подробности нашего визуального языка. Назовём его ППМН (Postolnik Process Modeling Notation).

Визуальный язык ППМН

Все элементы языка можно разделить на вершины и рёбра.

Типы вершин

Вершины бывают пяти видов:

1. Начало экземпляра процесса
2. Конец экземпляра процесса
3. Активность (аналог служебной задачи, service task)
4. Исключительный шлюз
5. Вызов подпроцесса

Начало процесса

Началом процесса может быть любая фигура, у которой в поле `description` записан следующий текст:

```
{:type :start}
```

Это – определение словаря с одной записью (ключ – `:type`, значение `:start`) в формате EDN (Extensible Data Notation).

Конец процесса

У вершины, обозначающий конец процесса текст в описании такой:

```
{:type :end}
```

Активность

У активностей в описании нужно указать функцию, которая будет выполняться:

```
{:type :activity
:fn "parent.act1-fn"}
```

Здесь мы вызываем функцию с идентификатором `parent.act1-fn`. Функции, которые указываются здесь, должны реализовывать интерфейс ActivityFunction:

```
package com.dpisarenko.postolnik.api;

import java.util.Map;
import java.util.function.Function;
```

```
public interface ActivityFunction extends Function<Map<String,
    Object>, Map<String, Object>> {
}
```

То есть: На входе эта функция берет старое состояние экземпляра процесса в виде словаря `Map<String, Object>`. Потом эта функция что-то внутри себя делает и возвращает новую версию состояния.

Это может работать только в отсутствии многопоточности. По этой причине в ППМН нет паралельных шлюзов.

Вызов подпроцесса

Для вызова подпроцесса в описании надо написать правильный тип, а также идентификатор процесса, который надо вызвать.

```
{:type :call-subprocess
:process "sub-process"}
```

Здесь мы вызываем процесс с идентификатором `sub-process`.

Исключительный шлюз

У открывающего исключительного шлюза описание

```
{
  :type :gateway-open
}
```

У закрывающего:

```
{
  :type :gateway-close
}
```

Рёбра

У ребёр, исходяших из открывающего шлюза должно быть прописано условие.

Условия бывают двух видов:

1. `false`
2. Идентификатор функции

Выполнение пойдет по ребру с условием `false`, если не было найдено ни одного исходящего ребра, для которого функция, указанная в надписи на ребре вернула бы `true`.

Функции для условий реализуют интерфейс ConditionFunction:

```
public interface ConditionFunction extends Function<Map<String,
        Object>, Boolean> {
}
```

Здесь на входе – текущее состояние экземпляра процесса, на выходе – булевое значение.

Делаем движок

Сердце Постольника – метод `EngineImpl.runGraphWithSubprocesses`. Он принимает ряд параметров:

1. `processGraphsByProcessIds`: Словарь, где ключ – идентификатор процесса, а значение – граф этого процесса. У корневого процесса ключ – пустая строка.
2. `initCtx`: Словарь с первоначальным контекстом. Вызов каждой активности будет менять этот контекст.
3. `activityFns`: Словарь, где ключ – идентификатор функции, а значение – сама функция.
4. `conditionFns`: Аналогично для условий.
5. `processId`: Идентификатор процесса, который нужно выполнить.

Возвращает этот метод метод контекст после преобразований во всех активностях.

Общая схема выглядит так: В методах-обертках переданные данные диаграмм преобразуются в граф в методе `EngineImpl.turnXmlFilesIntoGraphs`. В результате получаем граф из библиотеки JGraphT.

Дальше в методе `EngineImpl.runGraphWithSubProcessses` работаем так.

Сначала получаем нужный нам граф (их может быть много из-за подпроцессов).

```
final DefaultDirectedGraph processGraph =
       processGraphsByProcessIds.get(processId);
```

Далее находим стартовую вершину. Если не находим, прекращаем обработку.

```
final Optional<Vertex> startNodeOpt =
    findStartNode(processGraph);

if (startNodeOpt.isEmpty()) {
    LOGGER.error("No start node found");
    return initCtx;
}
```

Потом формируем состояние обхода графа:

```
GraphTraversalState state = GraphTraversalState
        .builder()
        .fnBindings(activityFns)
        .conditionFns(conditionFns)
        .ctx(initCtx)
        .nextNodeToProcess(startNodeOpt.get())
        .continueToWalkThroughGraph(true)
        .curProcessId(processId)
        .processGraphsByProcessIds(processGraphsByProcessIds)
        .build();
```

А дальше в цикле обходим граф, пока не дойдем до конечной вершины.

```
while (state.isContinueToWalkThroughGraph()) {
    final Vertex curNode = state.getNextNodeToProcess();
    final Map<String, Object> curNodeData =
            parseClojureMap(curNode.getDescription());
    final Keyword type = (Keyword) curNodeData
            .get(Keyword.intern("type"));
    final NodeProcessor nodeProcessor =
            nodeProcessorsByTypes.get(type);
    state = nodeProcessor.apply(NodeProcessingInput.builder()
            .curNode(curNode)
            .curNodeData(curNodeData)
            .graph(processGraph)
            .state(state)
            .build());
}
```

В теле цикла делаем следующие вещи.

Сначала определяем вершину, которую нам надо обработать.

```
final Vertex curNode = state.getNextNodeToProcess();
```

Преобразуем описание в формате EDN в словарь.

```
final Map<String, Object> curNodeData =
        parseClojureMap(curNode.getDescription());
```

Из этого словаря берем тип вершины по ключу `:type` (`Keyword.intern("type")` в Джаве равнозначен `:type` в EDN).

```
final Keyword type = (Keyword) curNodeData
        .get(Keyword.intern("type"));
```

По типу вершины определяем обработчик этой вершины.

```
final NodeProcessor nodeProcessor =
        nodeProcessorsByTypes.get(type);
```

Код обработчиков вершин находится в пакете `com.dpisarenko.postolnik.impl.nodeprocessors`.

Передаем в обработчик вершины входные данные и обновляем состояние обхода графа.

```
state = nodeProcessor.apply(NodeProcessingInput.builder()
        .curNode(curNode)
        .curNodeData(curNodeData)
        .graph(processGraph)
        .state(state)
        .build());
```

Когда выходим из цикла, возвращаем крайний контекст.

```
return state.getCtx();
```

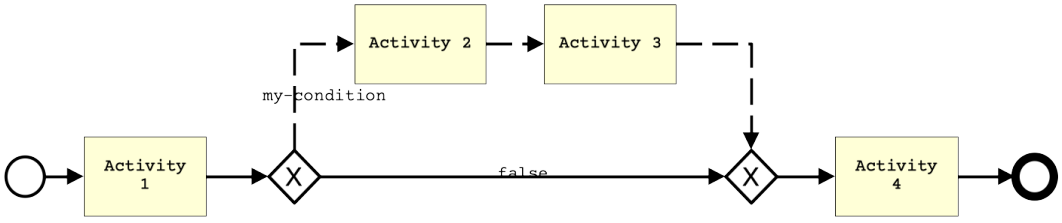
Тесты

Условия

В классе `EngineImplTest` есть ряд тестов.

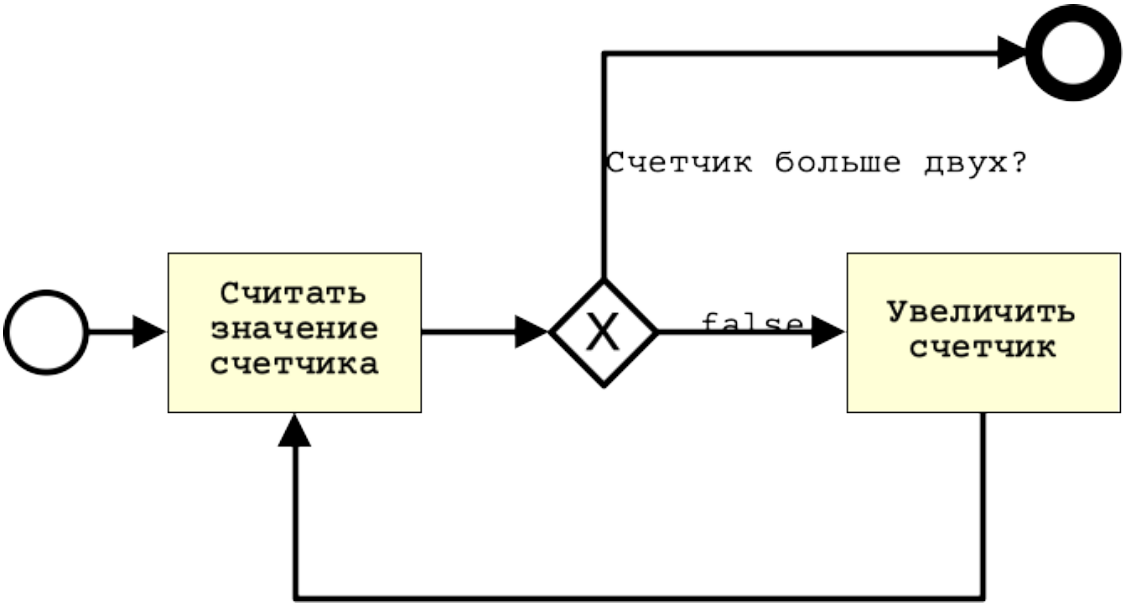
Тесты `givenMyConditionTrue_whenRun_thenExecAct234` и `givenMyConditionTrue_whenRun_thenExecAct4` демонстрирует как в зависимости

от значения, которая возвращает функция `my-condition` выполняются разные наборы активностей.



Пунктирной линией показан путь выполнения в тесте `givenMyConditionTrue_whenRun_thenExecAct234`, плотной – в `givenMyConditionTrue_whenRun_thenExecAct4`.

Цикл



В тесте `givenCycle_whenRun_thenExecuteCorrectActivities` показана работа Постольника в диаграмме с циклом.

Подпроцессы

Работу с подпроцессами демонстрирует тест `givenSubprocess_whenRun_thenExecuteRightActivities`.

Заключение

П. С.: Если Вам понравилась статья, подумайте о том, чтобы поддержать тех, благодаря кому вы можете читать ее на свободе и в безопасности, а также их близких.