**Методические указание**

# MDA

## Теоретическая часть

**UML** – унифицированный язык моделирования (Unified Modeling Language) – это система обозначений, которую можно применять для объектно-ориентированного анализа и проектирования.

Его можно использовать для визуализации, спецификации, конструирования и документирования программных систем.

Словарь UML включает три вида строительных блоков:

* Диаграммы.
* Сущности.
* Связи.

**Сущности** – это абстракции, которые являются основными элементами модели, связи соединяют их между собой, а диаграммы группируют представляющие интерес наборы сущностей.

**Диаграмма** – это графическое представление набора элементов, чаще всего изображенного в виде связного графа вершин (сущностей) и путей (связей). Язык UML включает 13 видов диаграмм, среди которых на первом месте в списке — диаграмма классов, о которой и пойдет речь.

**Диаграммы классов** показывают набор классов, интерфейсов, а также их связи

Большинство элементов UML имеют уникальную и прямую графическую нотацию, которая дает визуальное представление наиболее важных аспектов элемента.

**Класс** – это описание набора объектов с одинаковыми атрибутами, операциями, связями и семантикой.

Графически класс изображается в виде прямоугольника, разделенного на 3 блока горизонтальными линиями:

* имя класса
* атрибуты (свойства) класса
* операции (методы) класса.

**Архитектура, управляемая моделью (Model Driven Architecture, MDA)** - создаваемая консорциумом OMG разновидность концепции «Разработка, управляемая моделями»: модельно-ориентированного подхода к разработке программного обеспечения.

Его суть состоит в построении абстрактной метамодели управления и обмена метаданными (моделями) и задании способов её трансформации в поддерживаемые технологии программирования (Java, CORBA, XML и др.). Создание метамодели определяется технологией моделирования MOF (Meta Object Facility), являющейся частью концепции MDA.

**Мета моделирование** – способ описания моделей, определяющий механизмы построения конкретных моделей программных систем с помощью базового словаря и набора ограничений, налагаемых на создаваемые модели.

Понятие «управление моделью» подразумевает прямое использование модели при любых действиях по проектированию, разработке и развертыванию системы. Под архитектурой, управляемой моделью, понимается полная и законченная архитектура приложения, целиком формируемая с помощью модели.

**Модель приложения** – это взаимосвязанный набор визуальных диаграмм, наглядно описывающих внутреннюю структуру системы и принципы ее функционирования. Модель приложения не привязана к конкретному языку или конкретной среде программирования.

При разработке модели воспользуемся инструментом, который позволяет разрабатывать UML модель приложения и по ней генерировать код, в данном примере будет использоваться бесплатный инструмент: <http://argouml.tigris.org/>

**ArgoUML** — средство UML моделирования. ArgoUML является открытым программным обеспечением и распространяется под лицензией EPL.

Функциональность ArgoUML включает в себя:

* Поддержку спецификаций UML 1.3, 1.4, XMI 1.0, 1.1, 1.2
* 9 видов диаграмм UML (диаграммы классов, состояний, кооперации, последовательности, деятельности, прецедентов, объектов, компонентов, развёртывания)
* Поддержку OCL для классов
* Генерацию исходного кода Java, C++, C# и PHP
* Обратный инжиниринг из исходного кода и байткода Java
* Автоматическую верификацию модели UML (design critics)

## Практическая часть

Для примера разработаем приложение показывающее жизненный цикл Микроволновой печи. При анализе было выявлено 3 основных сущности:

* **Еда**, для неё мы будем хранить информацию о названии и времени приготовления.
* **Таймер**, в нем будет хранится информация об оставшемся времени приготовления и методы по запуску таймера, его приостановке и остановке таймера
* **Микроволновка**, в ней будет хранится информация о готовящейся еде или её отсутствию и внутреннем таймере,

Получим такую UML модель:

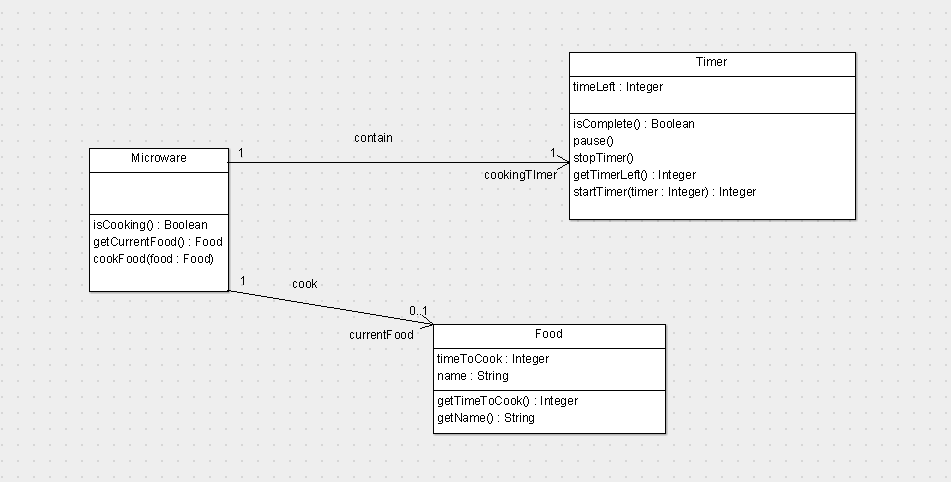


Рисунок 1. UML модель классов системы.

И с помощью средств AgroUML получим классы:

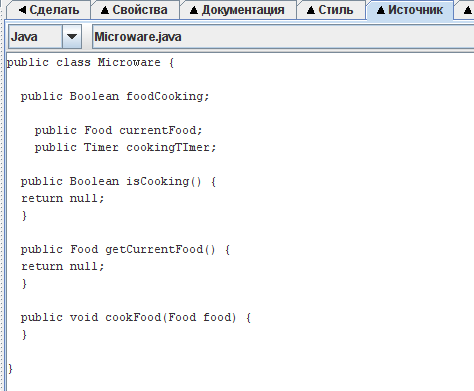


Рисунок 2. Авто сгенерированный класс Микроволновка

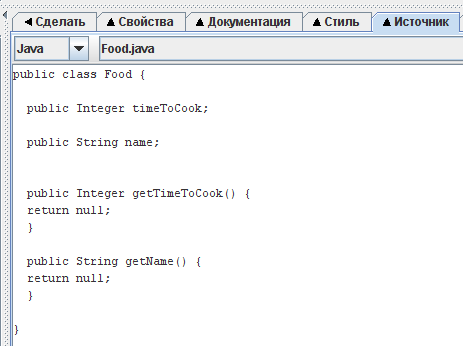


Рисунок 3. Авто сгенерированный класс Еда

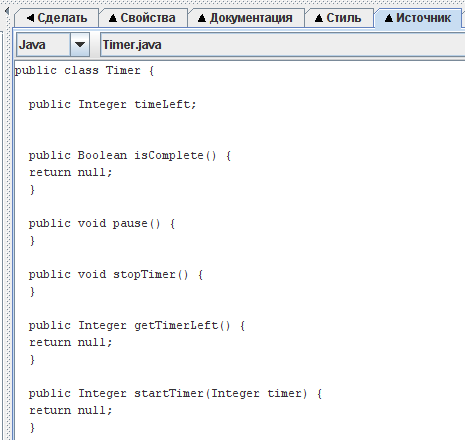


Рисунок 4. Авто сгенерированный класс Таймер

## Контрольные вопросы

1. Что такое UML и зачем он используется?
2. Какая основная суть использования MDA концепции?
3. Какая UML диаграмма наиболее часто используется для создания модели MDA?

# Машина состояний

## Теоретическая часть

**Автомат** – это элемент системы, который характеризуется двумя базовыми понятиями: состояние и переход.

**Диаграмма состояний (statechart diagram)** - объекты характеризуются поведением и состоянием, в котором находятся. Например, человек может быть новорожденным, младенцем, ребенком, подростком или взрослым. Другими словами, объекты что-то делают и что-то "знают". Диаграммы состояний применяются для того, чтобы объяснить, каким образом работают сложные объекты. Несмотря на то что смысл понятия "состояние" интуитивно понятен, все же приведем его определение в таком виде, в каком его дают классики и Zicom Mentor:

**Состояние (state)** - ситуация в жизненном цикле объекта, во время которой он удовлетворяет некоторому условию, выполняет определенную деятельность или ожидает какого-то события.

Состояние объекта определяется значениями некоторых его атрибутов и присутствием или отсутствием связей с другими объектами.

Диаграмма состояний показывает, как объект переходит из одного состояния в другое.

Очевидно, что диаграммы состояний служат для моделирования динамических аспектов системы (как и диаграммы последовательностей, кооперации, прецедентов и, как мы увидим далее, диаграммы деятельности).

Диаграмма состояний полезна при моделировании жизненного цикла объекта.

От других диаграмм диаграмма состояний отличается тем, что описывает процесс изменения состояний только одного экземпляра определенного класса - одного объекта, причем объекта реактивного, то есть объекта, поведение которого характеризуется его реакцией на внешние события. Понятие жизненного цикла применимо как раз к реактивным объектам, настоящее состояние (и поведение) которых обусловлено их прошлым состоянием. Но диаграммы состояний важны не только для описания динамики отдельного объекта. Они могут использоваться для конструирования исполняемых систем путем прямого и обратного проектирования. И они действительно с успехом применяются в таком качестве, вспомним существующие варианты "исполняемого UML", такие как UNIMOD, FLORA и др.

Для её практической реализации воспользуемся библиотекой, позволяющей реализовывать машины состояний на Java:

<https://github.com/Beh01der/EasyFlow>

**EasyFlow** обладает возможностями по:

* реализации сложной логики, при этом сохраняя код простым и чистым
* простой обработке асинхронных вызовов
* реализации событийного подхода к программированию
* защите от ошибки StackOverflow, с помощью нахождения рекурсий
* упрощение проектирования, программирования и тестирования сложных java-приложений

Для реализации машины состояний при помощи EasyFlow необходимо:

1. Реализовать набор состояний машины (java enum), реализующий интерфейс StateEnum
2. Реализовать набор событий машины (java enum), реализующий интерфейс EventEnum
3. Создать класс контекста машины состояний, реализующий интерфейс StatefulContext
4. Создать объект класса EasyFlow<StatefulContext> с помощью функций библиотеки from(), transit(), on().
5. Описать логику действий при переходах между состояниями с помощью функции whenEnter().
6. Запустить машину flow.start().

Для подключения библиотеки, присутствует наиболее простой способ с использованием Maven:

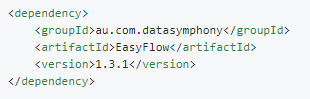


Рисунок 5. Листинг зависимости EasyFlow

**Apache Maven** — Фреймворк для автоматизации сборки проектов на основе описания их структуры в файлах на языке POM, являющемся подмножеством XML.

Проект Maven издаётся сообществом Apache Software Foundation, где формально является частью Jakarta Project.

Информация для сборки проекта, поддерживаемого Apache Maven, содержится в XML-файле с названием pom.xml. При запуске Maven проверяет, содержит ли конфигурационный файл все необходимые данные и все ли данные синтаксически правильно записаны.

В файле pom.xml задаются зависимости, которые имеет управляемый с помощью Maven проект. **Менеджер зависимостей** основан на нескольких основных принципах:

* **Репозитории.** Maven ищет необходимые файлы в локальных каталогах или в локальном Maven-репозитории. Если зависимость не может быть локально разрешена, Maven подключается к указанному Maven-репозиторию в сети и копирует в локальный репозиторий. По умолчанию Maven использует Maven Central Repository, но разработчик может конфигурировать и другие публичные Maven-репозитории, такие, как Apache, Ibiblio, Codehaus или Java.Net.
* **Транзитивные зависимости.** Необходимые библиотеки подгружаются в проект автоматически. При разрешении конфликта версий используется принцип «ближайшей» зависимости, то есть выбирается зависимость, путь к которой через список зависимых проектов является наиболее коротким.
* **Исключение зависимостей.** Файл описания проекта предусматривает возможность исключить зависимость в случае обнаружения цикличности или отсутствия необходимости в определённой библиотеке.
* **Поиск зависимостей.** Поиск зависимостей (open-source-библиотек и модулей) ведётся по их координатам (groupId, artifactId и version). Эти координаты могут быть определены с помощью специальных поисковых машин, например, Maven search engine. Например, по поисковому признаку «pop3» поисковая машина предоставляет результат с groupId="com.sun.mail" и artifactId="pop3".
* **Менеджеры репозиториев.** Репозитории реализуются с помощью менеджеров репозиториев Maven (Maven Repository Manager), таких как Apache Archiva, Nexus (ранее Proximity), Artifactory, Codehaus Maven Proxy или Dead Simple Maven Proxy.

## Практическая часть

Для использования внутри нашего MDA приложения машины состояний нам необходимо составить диаграмму состояний.

Состояниями системы-микроволной печи будут являться:

* Пустая
* Готова к приготовлению
* Идет приготовление
* Приготовление завершено
* Приготовление прервано

Событиями будут являться:

* Положили еду
* Нажали на дверцу
* Нажали кнопку
* Время таймера окончено

Соответственно, при проектировании получим следующую диаграмму состояний:

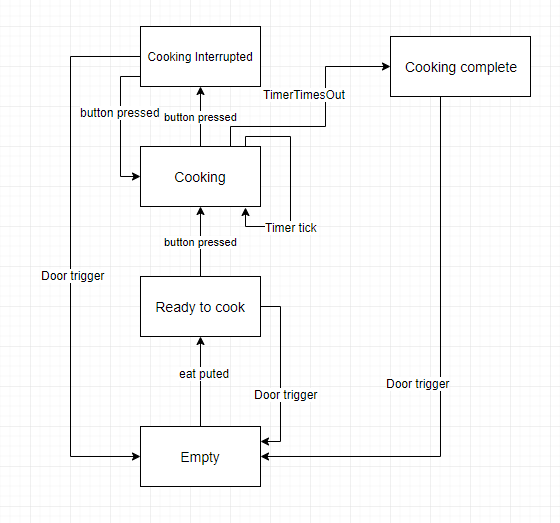


Рисунок 6. Диаграмма состояний системы

Для использования данной библиотеки, нам необходимо создать множество наших состояний, которое будет представлять из себя *enum* и наследоваться от *StateEnum*. И множество событий, которые будут наследоваться от *EventEnum.*

В нашем случае множество состояний: EMPTY, READY\_TO\_COOK, COOKING, COOKING\_INTERRUPTED, COOKING\_COMPLETE

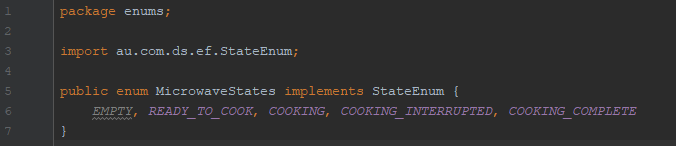


Рисунок 7. Множество состояний системы

В нашем случае множество событий: DOOR\_CLOSED, DOOR\_OPENED, BUTTON\_PRESED, TIMER\_TIMES\_OUT

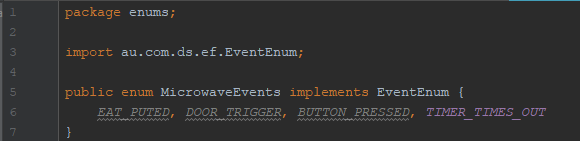


Рисунок 8. Множество событий системы

А также нам необходимо создать некоторую реализацию контекста, на котором будет реализована наша машина состояний. Для этого нам необходимо просто создать класс, наследующий *StatefulContext*

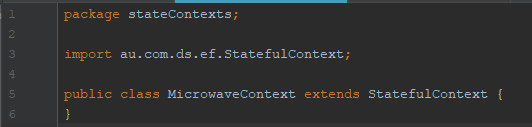


Рисунок 10. Пустой класс, представляющий контекст ЖЦ микроволновки

## Контрольные вопросы?

1. Что показывает диаграмма состояний?
2. Отличие диаграмм состояний от других диаграмм?
3. В каком файле описываются зависимости Maven?

# Реализация паттерна MVC

## Теоретическая часть

**MVC** — это фундаментальный паттерн, который нашел применение во многих технологиях, дал развитие новым технологиям и каждый день облегчает жизнь разработчикам.

Впервые паттерн MVC появился в языке SmallTalk. Разработчики должны были придумать архитектурное решение, которое позволяло бы отделить графический интерфейс от бизнеса логики, а бизнес логику от данных. Таким образом, в классическом варианте, MVC состоит из трех частей, которые и дали ему название.

1. **Модель**

Под Моделью, обычно понимается часть, содержащая в себе функциональную бизнес-логику приложения. Модель должна быть полностью независима от остальных частей продукта. Модельный слой ничего не должен знать об элементах дизайна, и каким образом он будет отображаться. Достигается результат, позволяющий менять представление данных, то как они отображаются, не трогая саму Модель.

Модель обладает следующими признаками:

* Модель — это бизнес-логика приложения;
* Модель обладает знаниями о себе самой и не знает о контроллерах и представлениях;
* Для некоторых проектов модель — это просто слой данных (DAO, база данных, XML-файл);
* Для других проектов модель — это менеджер базы данных, набор объектов или просто логика приложения;

1. **Представление (View)**

В обязанности Представления входит отображение данных полученных от Модели. Однако, представление не может напрямую влиять на модель. Можно говорить, что представление обладает доступом «только на чтение» к данным.

Представление обладает следующими признаками:

* В представлении реализуется отображение данных, которые получаются от модели любым способом;
* В некоторых случаях, представление может иметь код, который реализует некоторую бизнес-логику.

Примеры представления: HTML-страница, WPF форма, Windows Form.

Model-View-Controller

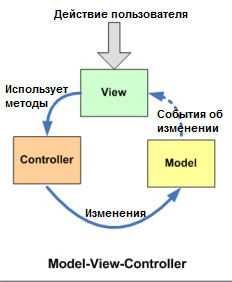


Рисунок 11. Схема работы MVC

Основная идея этого паттерна в том, что и контроллер, и представление зависят от модели, но модель никак не зависит от этих двух компонент.

1. **Контроллер**

Признаки контроллера  
Контроллер определяет, какие представление должно быть отображено в данный момент;

* События представления могут повлиять только на контроллер.контроллер может повлиять на модель и определить другое представление.
* Возможно несколько представлений только для одного контроллера;

**JavaFX**

Для визуализации работы программы будет использована платформа JavaFX.

JavaFX представляет инструментарий для создания кроссплатформенных графических приложений на платформе Java.

JavaFX позволяет создавать приложения с богатой насыщенной графикой благодаря использованию аппаратного ускорения графики и возможностей GPU.

С помощью JavaFX можно создавать программы для различных операционных систем: Windows, MacOS, Linux и для самых различных устройств: десктопы, смартфоны, планшеты, встроенные устройства, ТВ. Приложение на JavaFX будет работать везде, где установлена исполняемая среда Java (JRE).

JavaFX предоставляет большие возможности по сравнению с рядом других подобных платформ, в частности, по сравнению со Swing. Это и большой набор элементов управления, и возможности по работе с мультимедиа, двухмерной и трехмерной графикой, декларативный способ описания интерфейса с помощью языка разметки FXML, возможность стилизации интерфейса с помощью CSS, интеграция со Swing и многое другое.

Для создания View в JavaFX есть несколько способов:

* С помощью FXML
* С помощью кода на Java

Контроллеры и модель будут представлять из себя обычные Java классы, начиненные логикой согласно паттерну MVC.

## Практическая часть

В нашем примере будет использоваться только одна View, отображающая состояние микроволновки и имеющая несколько кнопок для её управления. В связи с этим будет проще создать один класс на Java отвечающий за эту View

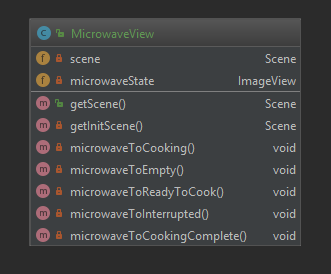


Рисунок 12. Схема класса View

Весь функционал нацелен на работу со сценой JavaFX:

* Начальная настройка сцены
* Перевод сцены между состояниями
* Выдача сцены

А также она имеет поля:

* Сцена с которой ведется работа
* Основной элемент – ImageView, который отображает саму Микроволновку

Моделью будет является единственный объект Microwave создаваемый при инициализации Контроллера. Его интерфейс был построен нами ранее в форме UML и сгенерирован в Java код, в нашем проекте осталось только дополнить его реализацией.



Рисунок 14. Описание класса Microwave

И аналогичные действия необходимо сделать для классов Food и Timer

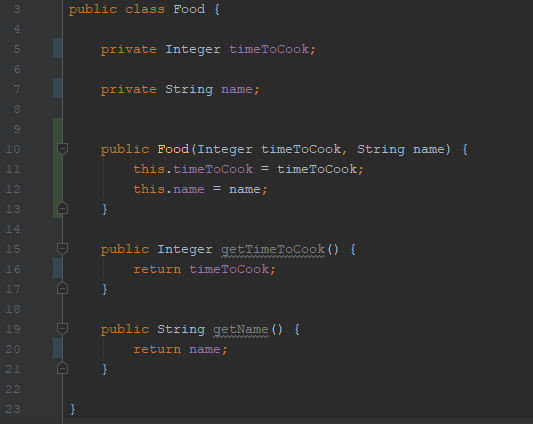


Рисунок 15. Описание класса Food

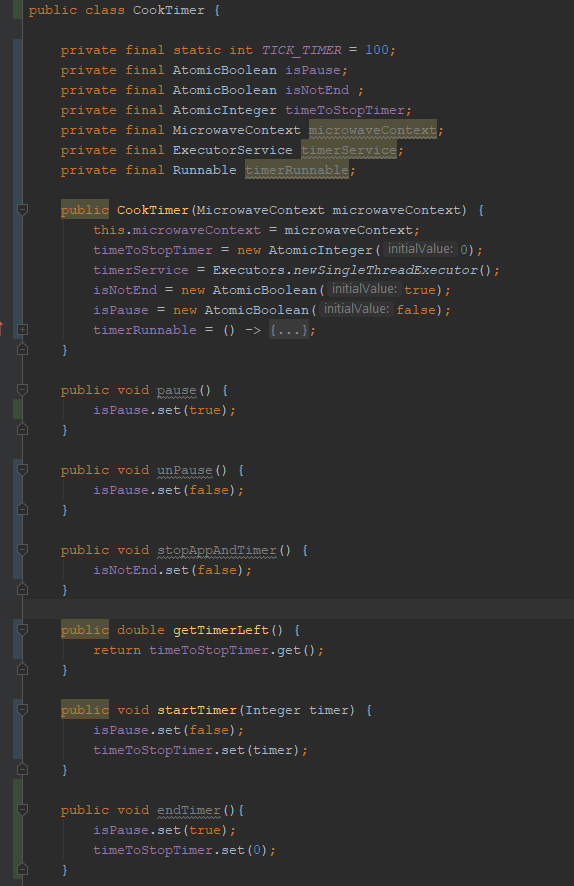


Рисунок 16. Схема класса CookTimer

В Таймер на моменте реализации мы добавили StateMachineContext в связи с тем, что нам необходимо вызывать события по таймеру при каждом тике и при завершении. Который ему передается при создании Microwave.

Контроллер будет работать с машиной состояний, внутри которой будет скрыта вся логика работы с View и Model.

Поэтому контроллер будет реализовать только две функции:

* Связывание интерфейса View и событий Машины состояний.
* Инициализация машины состояний (инициализация событий и инициализация действий).

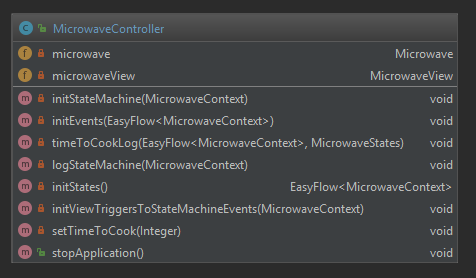


Рисунок 13. Схема класса Controller

Рассмотрим некоторые функции подробнее:

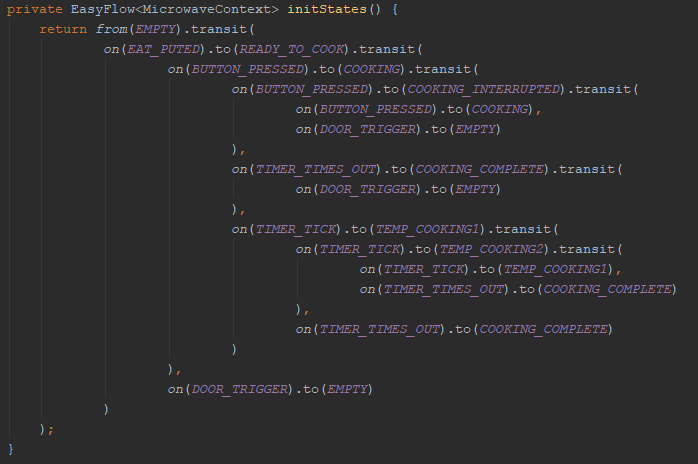


Рисунок 14. Логика инициализации основных состояний

В соответствии с графом состояний, нам необходимо описать его программную реализацию (возможные переходы между состояниями).

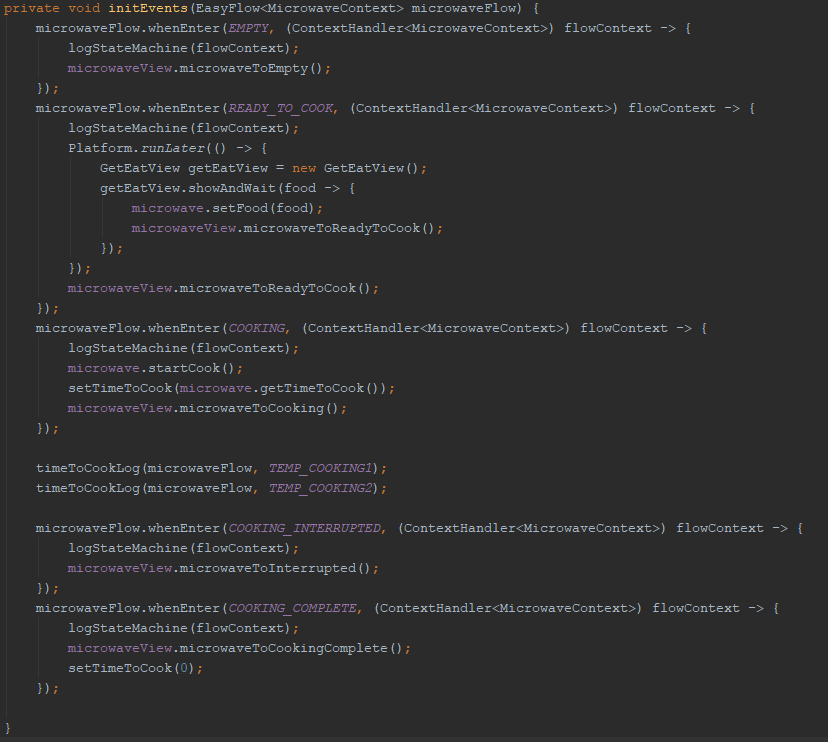


Рисунок 14. Логика инициализации основных событий

* При попадании в состояние Empty, нам необходимо очистить микроволновку.
* При попадании в ReadyToCook, пользователь обязан выбрать еду для приготовления, которая будет помещена в микроволновку.
* При попадании в состояние COOKING, необходимо начать готовку.
* При попадании в состояние COOKING\_INTERRUPTED, необходимо прервать готовку
* При попадании в состояние COOKING\_COMPLETE необходимо завершить готовку

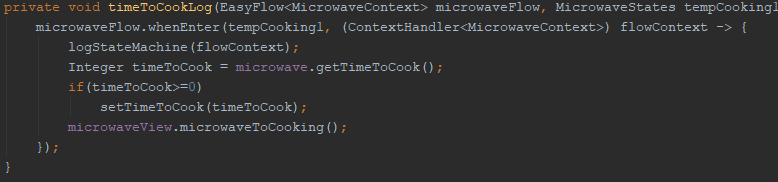


Рисунок 14. Логика опциональных событий

* В системе введены 2 опциональных состояния: TEMP\_COOKING1, TEMP\_COOKING2. Они необходимы для организации цикла при помощи которого будет обновляться таймер, посредством постоянного вызова события TIMER\_TICK.

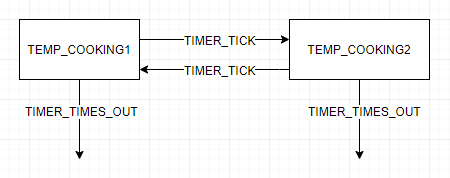


Рисунок 14. Метод организации циклических переходов

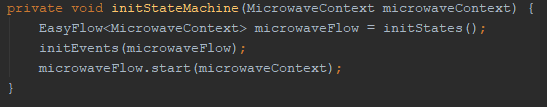


Рисунок 14. Инициализация машины состояний

Последним действием нам необходимо вызвать данные функции и запустить машину состояний.

## Контрольные вопросы?

1. В чем преимущества использования паттерна MVC?
2. В чем отличается функционал контроллера от модели?
3. Каким образом можно организовать архитектуру MVC в JavaFX?

# Задания для студентов:

Мы разработали задания для студентов:

1. Создать Информационную систему. В начале система находится в состоянии ожидания и может принимать на вход файлы, далее она переходит в состояние работы, читает файл и считает его хэш. Всю метаинформацию о файле и его хэш код необходимо сохранять в базу данных (если у файлов совпадает хэш, его не нужно пере сохранять).
2. Разработать систему массового обслуживания ресторана. Клиентов подаёт пользовалось. Люди случайным образом выбирают себе место между 3 столиками, за 1м столиком могут сидеть от 1 до 4 человек, также пользователь задает длительность обслуживания одного клиента. Система должна организовать оптимизированную работу обслуживания клиентов 1м официантом, с помощью машины состояний.
3. Разработать систему массового обслуживания парикмахерской. В парикмахерской есть 4 мастера, с случайно интенсивность работы (кто-то стрижет 20 минут, кто-то 15, также она зависит от сложности клиента). Система должна оптимизировать работу администратора, который распределяет людей между мастерами, если клиентов слишком много они помещаются в ограниченную очередь. Клиентов подает пользователь и задает их сложность.
4. Организовать работу охладительной системы на ядерном реакторе, имеется 4 грифельных стержня, в зависимости от интенсивности нагрева реактора, система должна выбрать количество стержней для его охлаждения (учесть, что после использования стержни необходимо некоторое время охлаждать и их использование недопустима). Система должна контролировать температуру реактора и стержней.
5. . Создать Информационную систему. В начале система находится в состоянии ожидания, пользователь вносит информацию о кредитах, размере желаемого кредита и срока. Система должна оптимально подбирать процентную ставку для клиентов в зависимости от размера оставшегося фонда банка (он также задается пользователем в начале, за основу взять экономическое состояние в стране на текущий момент). Вся информацию должна сохраняться в базу данных
6. Разработать систему массового обслуживания мак-дональдса. Пользователь добавляет заказы в систему и их объем (сложность приготовления), система обязана выдать им номер и организовать порядок обработки заказов 1м работником (система должна ориентировать на сложность заказа и время ожидание его в системе).
7. Разработать систему массового обслуживания гардероба. В гардеробе есть 3 работника (с разной эффективность и стоимостью), пользователь организует поток клиентов на какой-то промежуток времени, в зависимости от этого потока, система должна выбрать количество используемых работников, чтобы минимально затратить деньги и не создавать очередей. Система также должна записывать количество отработанных часов работниками в базу данных.
8. Организовать работу склада. Пользователь подает на склад грузы (с разным временем разгрузки и разной стоимостью). У системы имеет 2 разгрузочных аппарата (разной скорости). Система должна организовать порядок разгрузки на основании параметров грузов

# Вид конечного результата

Результат работы должен представлять из себя:

* Диаграмма классов приложения
* Авто сгенерированная основа классов приложения
* Оформленный отчет, включающий в себя ответы на контрольные вопросы
* Десктоп-приложение, написанное на Java

# Правила программного кода

* Визуализация приложения должна осуществляться с помощью некоторой платформы (JavaFX/Swing)
* Приложение должно реализовать паттерн MVC/MVVM/MVP
* Сборка проекта должна происходить автоматизировано с помощью некоторого Фреймворка (Gradle/Maven).
* Вся работа с моделью (бизнес логикой) должна осуществляться внутри машины состояний