**Теоретическая часть**

**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. Название концепции не совсем удачно, так как она определяет вовсе не архитектуру, а именно метод разработки программного обеспечения.

**Практическая часть**

**Создание MDA модели:**

При разработке модели воспользуемся инструментом, который позволяет разрабатывать UML модель (<https://habr.com/post/150041/> ) приложения и по ней генерировать код, в данном примере будет использоваться бесплатный инструмент:

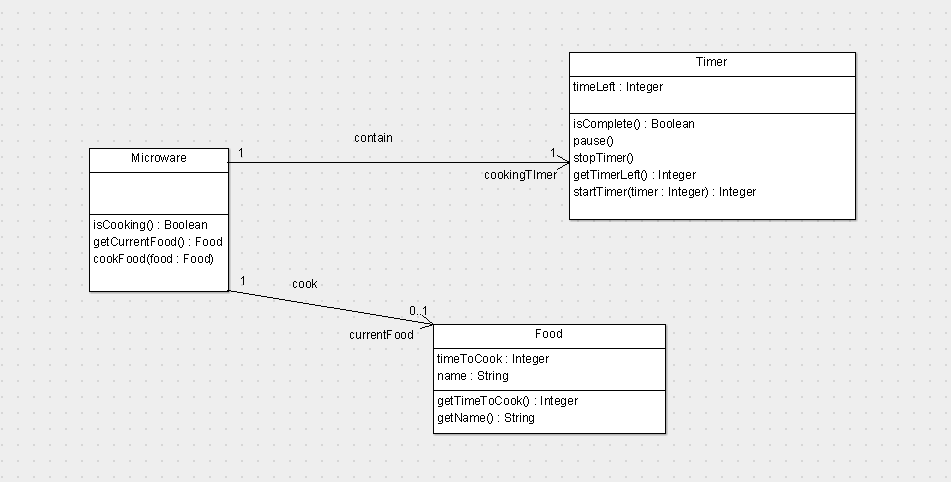
<http://argouml.tigris.org/>

Для примера разработаем приложение показывающее жизненный цикл Микроволновой печи.

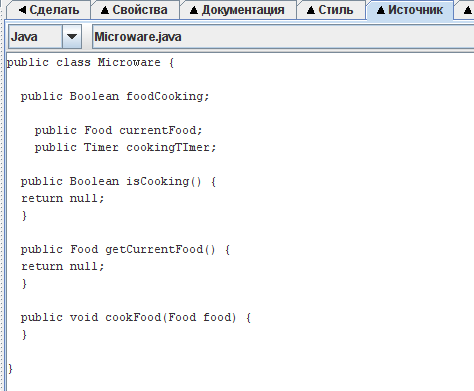
Следовательно, у нас будет 3 основных сущности:

* 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,

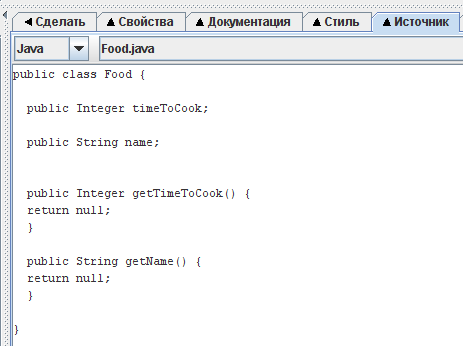
Получим такую UML модель:



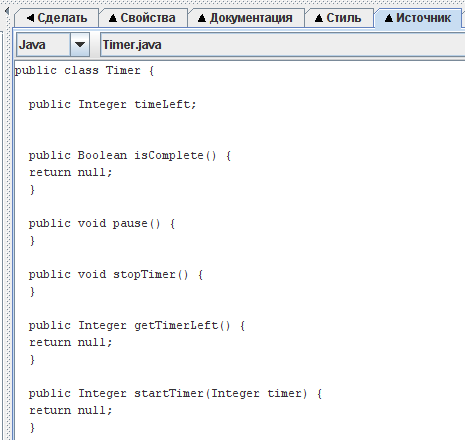
И с помощью средств AgroUML получим классы:



*Класс Микроволновка*



*Класс Еда*



*Класс Таймер*

**Теоретическая часть**

**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). Часто можно услышать, что диаграмма состояний показывает автомат, но об этом мы поговорим подробнее чуть позже. Диаграмма состояний полезна при моделировании жизненного цикла объекта (как и ее частная разновидность - диаграмма деятельности, о которой мы будем говорить далее).

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.

Но поговорим об обозначениях на диаграммах состояний. Скругленные прямоугольники представляют состояния, через которые проходит объект в течение своего жизненного цикла. Стрелками показываются переходы между состояниями, которые вызваны выполнением методов описываемого диаграммой объекта. Существует также два вида псевдосостояний: начальное, в котором находится объект сразу после его создания (обозначается сплошным кружком), и конечное, которое объект не может покинуть, если перешел в него (обозначается кружком, обведенным окружностью)..

**Практическая часть**

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

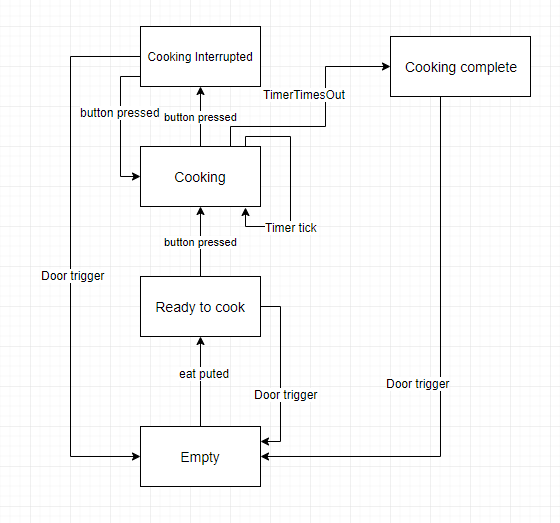
Состояниями системы-микроволной печи будут являться:

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

Событиями будут являться:

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

Получим диаграмму состояний:



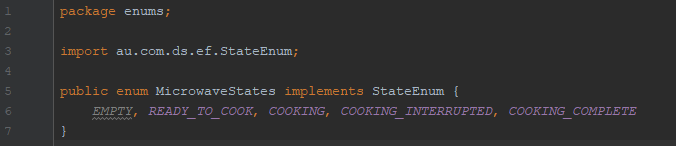
Для её практической реализации воспользуемся библиотекой, позволяющей реализовывать машины состояний на Java:

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

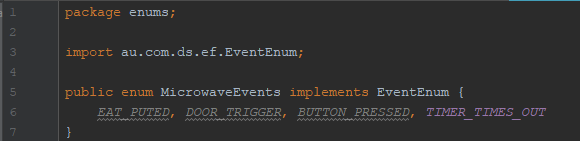
**Подготовка к использованию машины состояний**

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.*

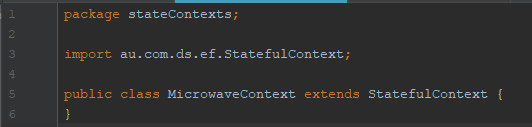
В нашем случае множество состояний: *EMPTY, READY\_TO\_COOK, COOKING, COOKING\_INTERRUPTED, COOKING\_COMPLETE*



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



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



**Реализация патерна MVC**

**Теоретическая часть**

Начнем с первого главного – Model-View-Controller. 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 бизнес логики, а бизнес логику от данных. Таким образом, в классическом варианте, MVC состоит из трех частей, которые и дали ему название.

**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;

**Представление (View)**

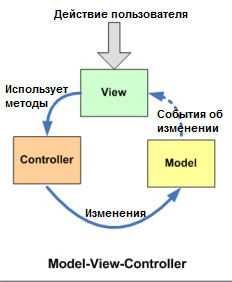
В обязанности Представления входит отображение данных полученных от Модели. Однако, представление не может напрямую влиять на модель. Можно говорить, что представление обладает доступом «только на чтение» к данным.

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**

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

**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 использован 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.

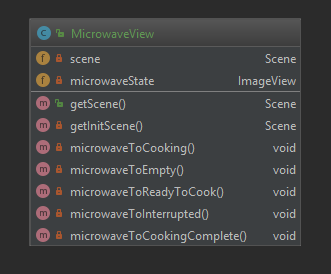
История JavaFX фактически началась в первой половине 2000-х годов, когда разработчик по имени Крис Оливер (Chris Oliver), будучи работником компании SeeBeyond, разработал для создания графических интерфейсов новый язык F3 (Froms Follows Functions). Впоследствии в 2005 году SeeBeyond была приобретена компанией Sun Microsystems (которая на тот момент развивала язык Java до покупки компанией Oracle). F3 был переименован в JavaFX, а Крис Оливер продолжил работу над новой платформой уже в рамках компании Sun. И в мае 2007 года Sun Microsystems публично анонсировала новую платформу для создания графических приложений. А 4 декабря 2008 года вышел JavaFX 1.0 SDK.

Для создания View в JavaFX есть несколько способов:

* С помощью FXML
* С помощью кода на Java

**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



Весь функционал нацелен на работу со сценой JavaFX:

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

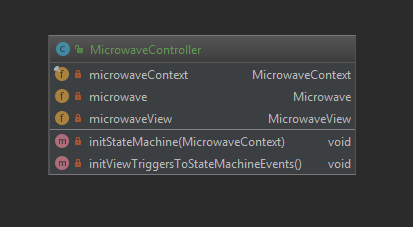
А также она имеет поля:

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

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

Поэтому контроллер будет реализовать только две функции:

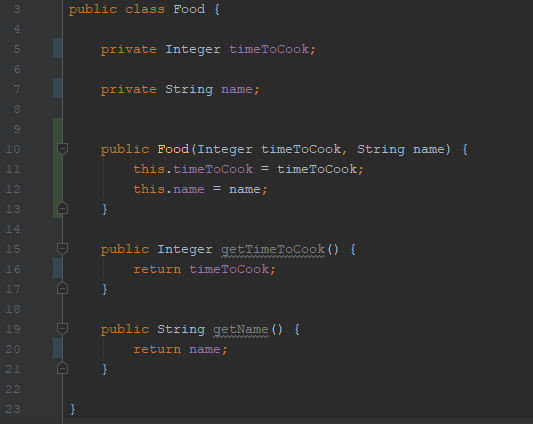
* Связывание интерфейса View и событий Машины состояний.
* Инициализация машины состояний

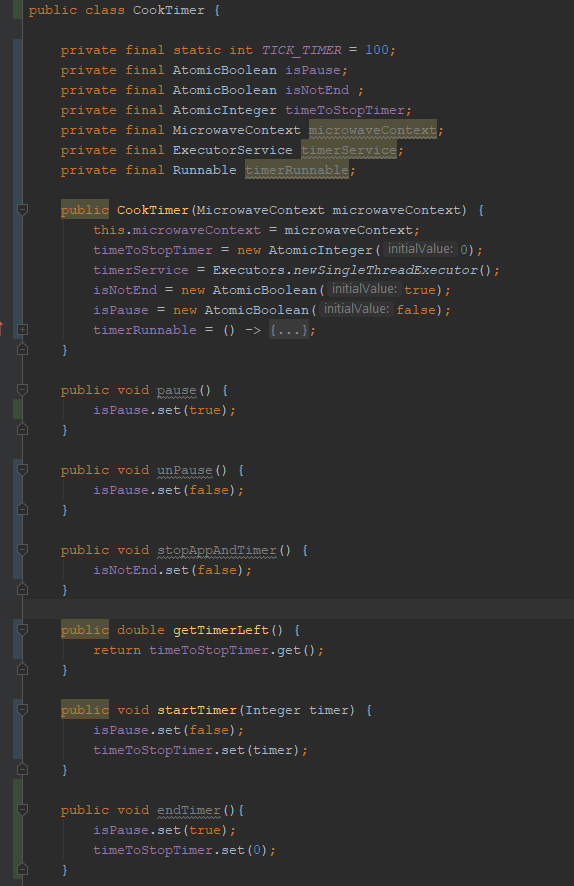


Моделью будет является единственный объект Microwave создаваемый при инициализации Контроллера. Его интерфейс был построен нами ранее в форме UML и сгенерирован в Java код, в нашем проекте осталось только дополнить его реализацией.



И аналогичные действия необходимо сделать для классов Food и Timer





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.