**Universitatea de Stat din Moldova**

**Facultatea MATEMATICA ŞI INFORMATICA**

**DEPARTAMENTUL DE INFORMATICA**

**PRIJILEVSCHI EVGHENI**

**ГЕНЕРАЦИЯ ПРОГРАММИНОГО КОДА**

**НА ОСНОВЕ ДИНАМИЧЕСКИХ МОДЕЛЕЙ**

**Informatica - 32**

**Teză de licență**

Sef Catedră **\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

(semnătura)

Conducător ştiinţific: **\_\_\_\_\_\_\_\_\_\_\_\_\_ Croitor Mihail, Lector universitar**

(semnătura)

Autorul: \_\_\_\_\_\_\_\_\_\_\_\_\_\_

(semnătura)

**Chişinău-2014**

Содержание

[Введение 4](#_Toc389669908)

[Цели и задачи работы 6](#_Toc389669909)

[Глава 1: Динамические модели информационных систем 7](#_Toc389669910)

[Генерация кода на основе динамических моделей 10](#_Toc389669911)

[Глава 2: Исследование предметной области 16](#_Toc389669912)

[Глава 3: Описание разработки приложения 19](#_Toc389669913)

[Библиография 25](#_Toc389669914)

Введение

a.​ Уникальность работы (15-20%)

b.​ Актуальность работы (70%)

c.​ Цели и задачи работы

Цели – теоретические: исследование, анализ, сравнительный анализ

Задачи – практические: разработка, внедрение, реализация

2)​ Динамические модели информационных систем

a.​ Событийная модель информационных систем

b.​ Динамические модели (Конечные автоматы)

c.​ Генерация кода на основе динамических моделей

d.​ Концепции статического и динамического порождающего программирования

3)​ Существующие генераторы кода

a.​ Примеры приложений генерации кода с помощью конечных автоматов

b.​ Сравнение вышеописанных примеров

c.​ Сценарии использования генерации кода на основе динамических моделей

d.​ Практические примеры

4)​ Описание разработки приложения

a.​ Формулировка проблемы

b.​ Определение функциональных требований (список)

c.​ Инструменты разработки

IDE, язык программирования, специальные библиотеки

d.​ Проектирование информационной системы

e.​ Описание пользовательского интерфейса

f.​ Резюме

5)​ Вывод

a.​ Актуальность (30%)

b.​ Уникальность (80-85%)

c.​ Результаты по целям и задачам

d.​ Дальнейшее развитие проекта

​ •Библиография (книжная + электронная)

​ •Приложение – код программы

# Введение

Предметом этой дипломной работы является генерация кода на основе динамических моделей, называющихся ещё конечными автоматами. Конечные автоматы представляют собой спецификацию последовательности состояний, через которые проходит объект или взаимодействие в ответ на события своей жизни, а также ответные действия объекта на эти события. Конечные автоматы применяются в автоматизации проектирования электронных приборов, протоколах передачи данных, синтаксическом анализе. Между тем, применение конечных автоматов в программировании является привлекательной с точки зрения использования компонентов конечных автоматов. Эти компоненты на практике способствуют минимизации дублирования кода и обеспечивают столько вариантов соединения, сколько потребуется для оптимизации повторного использования.

Несмотря на наличие похожего программного обеспечения, многие из них предназначены для решения узко поставленной задачи в рамках определённого языка программирования. В отсутствии удобного пользовательского интерфейса и богатого инструментария ставится вопрос о разработки нового программного обеспечения, которое оставалось бы актуальным помощником, как для начинающих программистов, так и для опытных разработчиков. Как следствие, уникальность данной работы кроется в сочетании нескольких факторов.

* Во-первых, пользовательский интерфейс представлен в качестве веб-приложения, где пользователь строит динамическую модель на клиентской стороне, а сервер генерирует код на основе полученных данных. Данная клиент-серверная модель делает программу платформонезависимой, доступной и удобной для пользователя.
* Во-вторых, пользователю представлена возможность выбрать для генерации один из нескольких современных языков программирования с реализованной объектно-ориентированной архитектурой. Локализация поведения каждого состояния происходит в каждом отдельном классе благодаря единой модели, которая получается либо проектированием диаграммы состояний, либо импортом/экспортом данных в формате XML.
* В-третьих, генерируемый код реализует паттерн Состояние, что, в отличие от других реализаций, позволит избежать многих хлопот: код будет соответствовать принципу открытости/закрытости (классы должны быть открыты для расширения, но закрыты для реализации). Данный принцип проектирования осуществляется благодаря инкапсулированию переменных аспектов этой архитектуры и конкретизации переменных переходов между состояниями. При огромных масштабах такие состояния не будут «прятаться» в множестве условных конструкций. Таким образом, дальнейшие изменения не приведут к ошибкам в готовом коде.

В наши дни сложно переоценить роль UML-диаграмм в контексте планирования информационных систем. Многочисленные средства разработки позволяют строить графическое описание объектных моделей. Основная причина их появления состоит в том, что языки программирования не обеспечивают нужный уровень абстракции, способный облегчить процесс проектирования. Унифицированный язык моделирования (UML 2.0) описывает 13 официальных типов диаграмм. Хотя эти виды диаграмм отражают различные подходы многих специалистов к моделированию систем, важно понимать, что UML не является реализацией программы, а лишь служит эскизом для дальнейшей разработки. Структуру классов, вариантов использования, компонент можно экспортировать в абстрактный код с реализованным скелетом приложения и пустой реализацией. Особое внимание стоит обратить на диаграмму автоматов, на которой отражены отношения между состояниями и переходами между ними. Диаграмма состояний представляет собой ориентированный граф для конечного автомата, вершины которого обозначают состояния системы, а дуги - события (реакцию), происходящую при переходе из одного состояния в другое. В частности, при разработке сложных систем, основанных на событиях, вызванных действиями пользователя (щелчок мыши, нажатие клавиши), конечные автоматы приходятся как нельзя кстати.

Когда разработчик сталкивается с необходимостью построения контекстно-свободной грамматики, у него есть несколько вариантов работы. Вполне возможно использование формальной грамматики на основе некоторого конечного алфавита, детерминированного конечного автомата или регулярных выражений. Первичной целью для моей работы является обеспечение возможностью построения чётких и ясных переходов между состояниями в доступной форме одновременно как для опытных специалистов, так и для начинающих программистов.

# Цели и задачи работы

Одним из важнейших условий разработки успешного приложения служит понимание теоретических принципов, служащих основой программы. В данном случае, генерация осуществляется за счёт конечных автоматов. Для того, чтобы все компоненты органично взаимодействовали между собой, передо мной была поставлена цель ознакомиться со структурой и теоретической базой динамических моделей, а также иметь чёткое представление о концепциях статического и порождающего программирования. Сочетание данных знаний позволит лучше понять событийную модель информационных систем, а также генерацию кода на основе динамических моделей.

По мере развития сети Интернет, технологии программирования не стоят на месте, постоянно расширяясь и дополняясь новыми фреймворками. Пришло время и нам идти в ногу со временем, поэтому мой взгляд привлекли Java EE технологии, а точнее Java-сервлеты вместе с контейнером сервлетов (также именуемым веб-контейнером). Следующей целью служит ознакомление с современными методами разработки веб-приложений, взяв за основу известный и активно используемый фреймворк Spring MVC, а также следуя принятым методам разработки на основе системы контроля версий, Apache Maven для автоматизации сборки проектов, а также применяя паттерны проектирования в проекте. Конечная задача данной дипломной работы – использовать возможные преимущества на практике и уменьшить многократно ручной труд, заменив его на автоматизированный, посредством использования конечных автоматов. На основе вышеупомянутых целей можно выделить задачи перед разрабатываемым приложением:

* Строить графическую модель конечного автомата, где множество состояний описывает состояние системы в момент бездействия пользователя, а дуги – события и реакцию системы.
* Преобразовывать графическую модель конечного автомата в программное представление в виде xml-документа.
* Преобразовывать динамическую модель в необходимый конкретный язык.
* Обеспечить использование паттернов проектирования в генерируемый код при необходимости.
* Возможность загружать/сохранять как весь проект, так и xml-структуру динамической модели в отдельности.
* Обеспечить возможность добавления новых языков.
* Добавить необходимый справочный материал для быстрого ознакомления пользователя с информационной системой.

# Глава 1: Динамические модели информационных систем

**Событийная модель информационных систем**

В этой главе мы рассмотрим парадигму событийно-ориентированного программирования, её применение и варианты использования в информационных системах.

С того момента, как приходится сталкиваться с написанием графического пользовательского интерфейса, мы предоставляем программе среду, в которой могут произойти различные события: пользователь может нажать кнопку, ввести данные в поле формы и нажать Enter, выбрать radio button или закрыть окно, а также тысячи других возможностей. Какое действие произойдёт при этом, и в каком порядке они сработают, зачастую нельзя предсказать. Это означает, что мы должны писать событийно-ориентированный код, который будет срабатывать на действия пользователя. Таким образом, первичной задачей подобной парадигмы программирования, именуемой событийно-ориентированной, является реакция на приём семантически важных сигналов (т.е. событий). Такие сигналы могут исходить из любого зарегистрированного источника, к примеру, от Button, CheckBox, Choice, MenuItem, ScrollBar, Frame, Panel и других компонентов.

Для обработки события необходимо знать интерфейсы прослушивания событий (Listeners). Когда возникает событие, источник события вызывает соответствующий метод, определённый блоком прослушивания и передаёт ему объект события в качестве параметра. Блоки прослушивания событий создаются путём прослушивания интерфейсов событий. Чтобы обработать события, необходимо реализовать интерфейс в блоке прослушивания, чтобы он принимал события желаемого типа, а затем реализовать код для регистрации блока прослушивания в качестве получателя уведомления событий.

Источник может генерировать неcколько типов событий, тогда каждому событию нужно регистрироваться отдельно. Объект тоже может регистрироваться, чтобы принимать несколько типов событий, но он должен тогда реализовать все интерфейсы. Данный механизм получения извещений при изменении состояния получил название паттерна Наблюдатель.

Поведенческий шаблон поведения или паттерн Наблюдатель определяет отношение «один-ко-многим» между объектами таким образом, что при изменении состояния одного объекта происходит автоматическое оповещение и обновление всех зависимых объектов.

**Динамические модели (Конечные автоматы)**

В данной главе рассматриваются два наиболее распространенных способа конечного задания формального языка: грамматики и автоматы. Рассматриваются конечные автоматы, соответствующие в иерархии Хомского праволинейным грамматикам, определяются понятия конечного автомата, недетерминированного конечного автомата и распознаваемого конечным автоматом языка. Приведены практические примеры и рисунки с целью ознакомления.

Два наиболее распространенных способа конечного задания формального языка - это грамматики и автоматы. Автоматами в данном контексте называют математические модели некоторых вычислительных устройств. В этой лекции рассматриваются конечные автоматы, соответствующие в иерархии Хомского праволинейным грамматикам.

Прежде чем непосредственно перейти к понятию динамических моделей, необходимо познакомиться с методами описания языков. Языки бывают разговорными (естественными) и формальными (искусственными). Формальные языки обладают правилами понимания предложений. В логике информатики это множество слов над алфавитом. Формальный язык может быть определён:

а) Простым перечислением слов, входящих в данных алфавит;

б) Словами, порождёнными некоторой формальной грамматикой;

в) Словами, порождёнными регулярным выражением;

г) Словами, порождёнными некоторым конечным автоматом.

У формального языка много общего с естественным языком, множество которого строится в соответствии с грамматическими правилами. Различают порождающие грамматики, а также распознающие (аналитические) грамматики. Первые создают правила, с помощью которых можно построить любое слово языка. Вторые позволяют по написанному слову определить, входит ли оно в правило или нет.

Динамическая модель или конечный автомат – это альтернативный способ определения языков. Для регулярных грамматик допускается описание при помощи автоматов. Этот вид также можно описать при помощи регулярных выражений. Существует несколько способов описания конечного автомата:

а) Аналитический. Конечный автомат описывается как объект, состоящий из 5 элементов:

КА = (Σ, Q, q0, δ, F), где

Σ - допустимый входной алфавит автомата. (конечное множество);

Q = { q0,q1, …, qn} – множество состояний автомата (конечное множество);

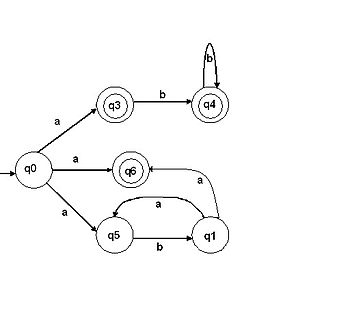
q0 – первоначальное состояние автомата, q0 ∈ Q;

δ – множество всех переходов автомата, δ : Q x Σ -> P(Q) (например, δ(q0, a)= {q1} или δ(q0, a)={q0, q1, q2} );

F – множество всех финальных состояний автомата, F ∈ Q.

Действия конечного автомата просты: он должен принять действия с входной ленты с состояния qi в qj. Цель состоит в том, чтобы принять слово, написанное на входной ленте. Работа автомата всегда начинается с состояния q0, считывая по одному символу из входной строки. Входной символ переводит автомат в состояние q с переходом функции перехода δ. Если в результате считывания входящего слова автомат окажется в одном из состояний множества F, то слово принимается, и тогда говорят, что данное слово принадлежит языку данного автомата. В противном случае, слово отвергается автоматом.

b) Диаграмма состояний (граф перехода):

Графическое представление автомата и функции перехода представляет собой взвешенный граф, где вершины – состояния конечного автомата, рёбра-переходы. Нагрузка (вес) – символы, при которых осуществляется данный переход. Если переход из qi в qj осуществляется при помощи нескольких символов, то все они будут описаны над данной дугой. В графе перехода все состояния обозначены одной окружностью, а все финальные состояния – двумя окружностями.

c) Таблица переходов – матричное представление функции δ. В таблице каждой строке соответствует одно состояние автомата, а каждому столбцу – входной символ. В ячейке – пересечении строки и столбца – записывается действие, которое должен выполнить автомат.

|  |  |  |
| --- | --- | --- |
| **Q\∑** | **a** | **b** |
| q0 | q3, q5, q6 | 0 |
| q1 | q5, q6 | 0 |
| q3 | 0 | q4 |
| q4 | 0 | q4 |
| q5 | 0 | q1 |
| q6 | 0 | 0 |

**Генерация кода на основе динамических моделей**

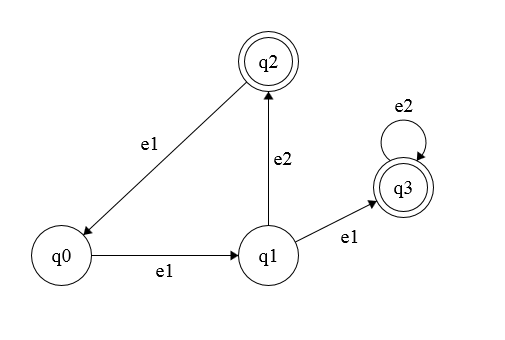
Код для программы, состоящей из множества состояний и событий, вызывающих переходы, можно автоматически сгенерировать с помощью динамических моделей. С помощью системы диаграмм переходов можно определить все возможные состояния перехода информационной системы и условия перехода между ними. Подобный шаблон можно реализовать двумя способами, используя:

1) Таблицу переходов конечного автомата

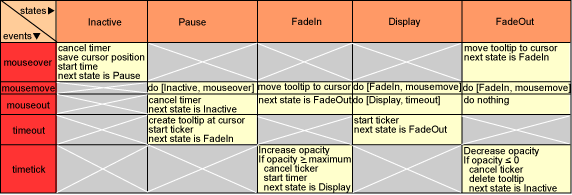
2) Switch-технология

3) Шаблон проектирования Состояние

Представим, что нам дана динамическая модель одного приложения. Её графическое приложение представлено на рисунке



Попробуем вручную реализовать код, с помощью каждого из методов:

В теории автоматов **таблица переходов** представляет собой аналитическое описание связей между состояниями, являясь наиболее прямым и удобным вариантом для компьютерной интерпретации. На основе предоставленной информации уже можно получить скелет информационной системы. Методология генерации кода заключается в следующем:

1. Выделяются все состояния компоненты ИС.
2. Выделяются все возможные события, обрабатываемые компонентой ИС.
3. Определяются прототипы функций, описывающих реакции на события.
4. Строится таблица переходов состояний для компоненты ИС.
5. Строится таблица реакций на события.

Пример реализации кода:

**Switch-технология** представляет собой метод проектирования конечных автоматов, названный так в силу того, что при использовании языков программирования высокого уровня это наиболее рационально выполнять с помощью конструкции switch языка Си либо ее аналогов в других языках программирования. Была предложена А. А. Шалыто в 1991 году.

Предлагаемый вариант технологии характеризуется следующими особенностями:

− В качестве базового термина используется понятие «автомат», а

не «класс», «объект», «алгоритм» или «агент», как это имеет место при других подходах;

− В общем случае автоматы рассматриваются не изолированно, а как составные части взаимосвязанной системы — системы взаимосвязанных автоматов, поведение которой формализуется с помощью системы взаимосвязанных графов переходов;

− В качестве основной применяется модель смешанного автомата, для описания поведения которого используется соответствующий граф переходов, содержащий только «простые» состояния;

− На этапе изучения предметной области на основе технического задания, которое при автоматизации технологических процессов обычно выдается заказчиком в словесной форме в виде совокупности сценариев и случаев использования, строится структурная схема системы, позволяющая получить общее представление об организации

управления, применяемой аппаратуре и интерфейсе объекта управления;

− На этапе анализа на основе технического задания выделяются сущности, каждая из которых называется автоматом (например, автомат управления насосом или автомат контроля температуры);

− Состояния каждого автомата первоначально определяются

по выделенным состояниям объекта управления или его

части, а при большом их количестве — по алгоритму

управления, построенному в другой нотации;

− В автоматы также могут быть введены и другие состояния,

связанные, например, с неправильными действиями оператора;

− Каждый автомат, при необходимости, может быть

разбит на составные части;

− Итеративный процесс анализа может выполняться многократно и завершается созданием перечня автоматов и перечня состояний для каждого автомата.

Пример реализации switch-технологии на базе вышеописанной динамической модели:

**Поведенческий шаблон проектирования Состояние** (State) управляет изменением поведения объекта при изменении его внутреннего состояния. Внешне это выглядит так, словно объект меняет свой класс.

Стандартный способ моделирования состояния: в объекте создаётся переменная экземпляра, в которой хранятся возможные коды состояния, а условные конструкции в методах обрабатывают разные состояния.

Для того, чтобы преобразовать диаграмму состояний в программный код, необходимо описать краткий процесс реализации конечных автоматов:

1. Собрать все состояния
2. Создать переменную экземпляра для хранения текущего состояния, определить значение для каждого возможного состояния.
3. Собрать все действия, которые могут выполняться в системе.

Преимущества объектной реализации:

* Локализация поведения каждого состояния в отдельном классе
* Исключение многочисленных конструкций if или switch, усложнявших сопровождение кода.
* Каждое состояние закрыто для изменения, однако сам автомат открыт для расширения посредством добавления новых классов состояний.
* Создание кодовой базы и структуры классов, приближенной к диаграмме переходов автомата, а, следовательно – более простой для чтения и понимания.

Пример реализации паттерна Состояние:

**Концепции статического и динамического порождающего программирования:**

В современной индустрии информационных технологий концепция порождающего программирования привлекает всё больше внимания благодаря перспективе производства большего многообразия продуктов при меньших времени и затратах. Порождающее программирование – это автоматизированное производство программных продуктов из отдельных компонентов, реализуя идею перехода от одноразовых программных систем к полуавтоматическому производству самых разнообразных продуктов. С точки зрения информационных технологий, порождающее программирование представляет собой парадигму технологии разработки программного обеспечения, основанную на моделировании семейства программных систем, используя которые можно по конкретным техническим требованиям автоматически получить специализированный и оптимизированный промежуточный или конечный программный продукт из элементарных, многократно используемых компонентов реализации…

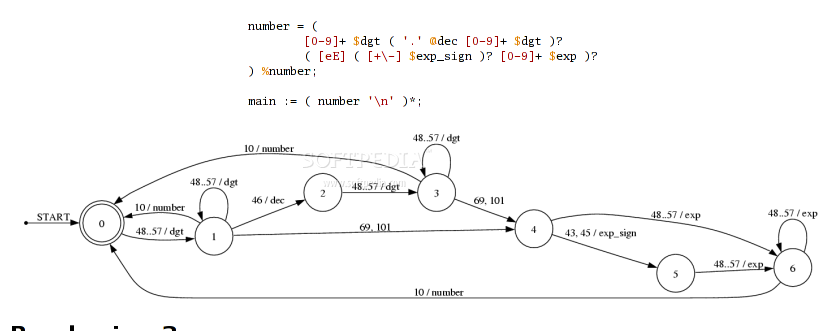
# Глава 2: Исследование предметной области

Сегодня ни одна из существующих IT-компаний не может обходиться без инструментов проектирования систем различного назначения и различных уровней сложности. Внедрение проектирования в жизненный цикл разработки является одним из важнейших условием динамичного развития.

Однако процессу генерации кода не уделяется должного внимания. Среди факторов, препятствующих использованию инструментов, порождающих код, наиболее значимыми являются следующие: нежелание компаний подстраивать команду разработчиков под отдельно взятую программу, несоответствие со специфичными требованиями разработчика, а также отсутствием возможности повторного использования сгенерированного кода. Существующие генераторы кода на основе динамических моделей можно разделить по следующим требованиям, являющимися критериями оценки выбранных приложений:

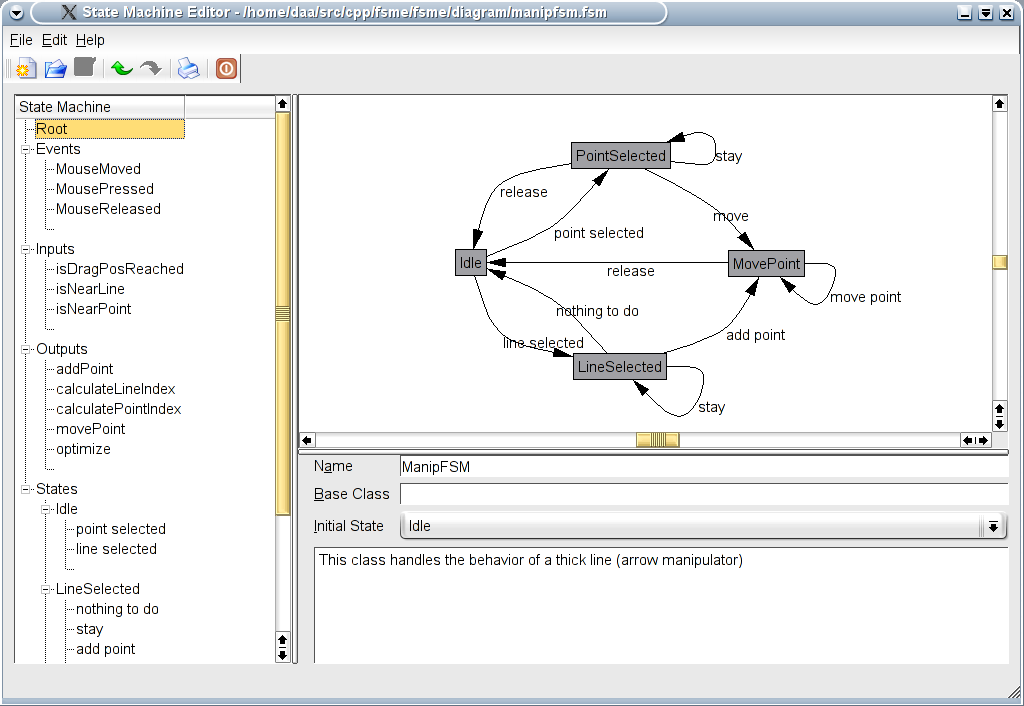
1. **Удобство интерфейса.** На какую целевую аудиторию нацелено данное приложение, какой оно требует квалификации, насколько оно накладывает дополнительные требования на скорость работы интерфейса.
2. **Универсальность.** Предназначено ли данное приложение для решения узкого круга задач или является общим для обширного рода задач на разных языках программирования.
3. **Возможность хранения и и повторного использования данных.** Загрузка и сохранение данных позволяет повторно использовать наработанные материалы.
4. **Поддержка.** Как часто проводятся обновления, существует ли руководство пользователя, является ли приложение платным/бесплатным.

В качестве примеров существующих приложений возьмём несколько примеров и сравним их по вышеуказанным характеристикам:

**Ragel State Machine Compiler.** 

Ragel компилирует выполняемые динамические модели из регулярных выражений. Ragel ориентирован на C, C++, Objective-C, C#, D, Java, Ruby, OCaml and Go. Ragel применяется для построений трудоёмких реализаций протоколов, парсинга данных, лексического анализа грамматик языков программирования, проверки ввода с клавиатуры.

**FSME**

****

FSME (Finite State Machine Editor) – графическое приложение для разработки конечных автоматов. Созданные динамические модели хранятся в файлах совместимых с Qt Designer IDE, а также могут быть конвертированы в код на C++.

**QM**

QM Modeler – это бесплатный, кросс-платформенный графический инструмент для UML моделирования

<http://www.state-machine.com/qm/index.php>

FSM Creator

<http://www.fsmcreator.de.gg/>

Stateflow

<http://www.mathworks.com/products/stateflow/>

Rational Rose RealTime

<ftp://ftp.software.ibm.com/software/rational/docs/documentation/manuals/rosert.html>

Enterprise Architect

<http://www.sparxsystems.com/support/faq/code_generation.html>

<http://www.sparxsystems.com/enterprise_architect_user_guide/9.2/software_engineering/code_generated_from_state_mach.html>

Visio2SWITCH

<http://www.softcraft.ru/auto/switch/v2s.shtml>

Visio2Switch - инструментальное средство [Visio2Switch](http://is.ifmo.ru/progeny/visio2switch/) позволяет по графу переходов, построенному в определенной нотации и изображенному с помощью редактора Visio, автоматически реализовать его в виде изоморфной программы на языке С.

MetaAuto - инструментальное средство [MetaAuto](http://is.ifmo.ru/projects/metaauto/) позволяет по графу переходов, построенному в той же нотации и изображенному с помощью того же редактора, автоматически реализовать его в виде изоморфной программы на любом языке программирования, для которого предварительно построен шаблон.

UniMod - инструментальное средство [UniMod](http://unimod.sourceforge.net/) предназначено для поддержки автоматного программирования и построения и реализации не только автоматов, но и программ в целом.

# Глава 3: Описание разработки приложения

**Формулировка проблемы**

Для того, чтобы разработать программу, генерирующую код на основе динамических моделей, необходимо выделить предпосылки, приведшие меня к выбору именно этой задачи. Несмотря на то, что существует много экземпляров готовых генераторов кода, ниже я постараюсь описать тезисы, которые обуславливают создание данного проекта:

* Необходимость адаптации к изменяющимся средам и областям применения информационных технологий.
* Необходимость реализации улучшений и усовершенствования существующих проектов с актуальными дополнениями и учётом предыдущих ошибок.
* Необходимость разработки удобного приложения для широкой аудитории.

Необходимо построить сайт, используя фреймворк Spring MVC, для генерации кода из динамической модели, построенной с помощью диаграммы состояний. Для построения графа перехода используется Javascript, с помощью которого также информация, записанная в формате JSON, поступает в контроллер. Серверная часть реализована на Java, а контейнером сервлетов является Jetty. Сервер получает необходимые данные о конечном автомате и строит несколько файлов, используя паттерн проектирования Состояние. В конце все файлы заносятся в архив и передаются, в качестве, на сторону клиента.

**Определение функциональных и нефункциональных требований**

Функциональные требования:  
1)      Строить графическую модель конечного автомата, где множество состояний описывает состояние системы в момент бездействия пользователя, а дуги – события и реакцию системы.  
2)      Преобразовывать графическую модель конечного автомата в программное представление в виде xml-документа.  
3)      Обеспечить использование паттернов проектирования в генерируемый код при необходимости

Нефункциональные требования:  
1)      Возможность загружать/сохранять xml-структуру динамической модели  
2)      Обеспечить возможность добавления новых языков  
3)      Добавить необходимый справочный материал для быстрого ознакомления пользователя с информационной системой

**Инструменты разработки**

* **Eclipse IDE for Java EE developers** - расширяемаяинтегрированная среда разработки (IDE) со свободным исходным кодом. Eclipse был создан в 2003 году и используется и поныне в различных областях, в частности, таких, как среда разработки для динамических веб-приложений на Java, используя технологию сервлетов.
* **Git** – открытая распределённая система контроля версий с приоритетом скорости. git изначально был разработан Линусом Торвальдсом для разработки ядра linux, сейчас же он управляется Джунио Хамано. Каждая рабочая директория git содержит окончательно готовый репозиторий с полной историей и отслеживания пути правок, не зависящей от доступа к сети или центральному серверу.
* **gitext.pngGit Extensions** представляет собой удобный графический пользовательский интерфейс для Git, который позволяет управлять системой контроля версий без командной строки.
* **Apache Maven** – это инструмент разработки, предназначенный для автоматизации сборки проектов. Основан на концепции объектной модели проекта(файл pom.xml). Maven может управлять сборкой проекта, компиляцией, внедрением зависимостей, версий пакетов, сборок. В Maven принято соглашение по структуре проектов. Он сам подкачивает все нужные зависимости через доступ в Интернет.

Жизненный цикл Maven-проекта:

1. validate – проект настроен корректно
2. compile - компиляция
3. test – с помощью JUnit-тестов
4. package – создание .jar/.war или .ear
5. integration-test – интеграционное тестирование исполняемых файлов
6. verify – пакет является корректным
7. install – копирование .jar в локальный репозиторий
8. deploy – публикация в удалённый репозиторий

Отдельные фазы:

clean – удаление всех созданных в процессе сборки артефактов

site – создание документации (javadoc + сайт описания проекта)

**Spring Framework** – это открытый фреймворк для разработки приложений на языке Java. Он предоставляет средства и функциональность для разработки приложений, включая основной контейнер инверсии управления IoC, независимую поддержку доступа к базам данных, слой MVC (модель-представление-контроллер), всестороннюю поддержку AOP (аспектно-ориентированного программирования), удалённого доступа, средства аутентификации и авторизации и многие другие модули для облегчения процесса разработки.

**Log4J** **-** библиотека журналирования Java программ, часть общего проекта «Apache Logging Project». По сути он представляет собой фреймворк для скрытия реализации рутинных операций по логгированию (журналированию) некоторых событий, которые происходят во время работы приложения, написанного на Java. Log4J позволяет сохранить трассировку выполнения конструкторов, методов, блоков обработки критических ситуаций, а также выполнить следующие операции:

* Выбор хранилища – консоль, файл, СУБД;
* Конфигурация хранилища – именование хранилища, объем хранилища, путь к хранилищу, сохранение конфигурации;
* Форматирование записей журнала – дата/время, класс/метод и т.д., гибкое форматирование.

Уровни вывода класса Logger (в возрастающем порядке):

* TRACE
* DEBUG
* INFO
* WARN
* ERROR
* FATAL
* OFF

**JavaScript** – это динамический, объектно-ориентированный, прототипно-ориентированный, слабо типизированный язык программирования, использующийся для написания скриптов в веб-браузерах. Несмотря на своё название, он не связан с языком программирования Java и похож на него лишь поверхностно. Созданный компанией Netscape в 1995 году как расширение HTML для браузера Netscape Navigator 2.0, главным назначением JavaScript было манипулирование HTML документами и валидацией форм. До приобретения своего известного имени, JavaScript назывался Mocha, затем LiveScript и, наконец, когда он был выпущен Sun Microsystems, тогда и начал называться по тому имени, которым он известен сегодня.

**jQuery**, ранее известная как jQuery Core, является библиотекой для JavaScript, написанной Джоном Ресигом, которая облегчила трафаретные действия, такие как прохождение и манипулирование DOM, перехват событий, анимация и передача данных через ajax.



**Ajax** - Asynchronous Javascript and XML. Не являясь технологией как таковой, ajax описывает «новый» подход к передаче данных, используя ряд технологий, таких как XMLHttpRequest для асинхронного обмена данными с сервером; JavaScript(или jQuery)/DOM, для вывода/взаимодействия с информацией; CSS – для стилизации данных; и XML(или JSON), использующиеся как формат для передачи данных.

Схема передачи данных посредством Ajax:

**Браузер**

Происходит событие

* Создаётся объект XMLHttpRequest
* Отправляется Http Request

**Сервер**

* Обрабатывается Http Request
* Создаётся ответ и данные отправляются обратно к браузеру

**Браузер**

* Обрабатываются возвращённые данные, используя JS
* Обновляется страница с контентом.

# Библиография

**Книжные источники:**

1. К. Чарнецки, У. Айзенекер – Порождающее программирование: методы, инструменты, применение. Для профессионалов. – СПб.: Питер,2005
2. Фримен Эр., Фримен Эл., Сьерра К., Бейтс Б. - Паттерны проектирования – Питер, 2011
3. Walls C. - Spring in Action. Third Edition – 2011
4. David Flanagan - JavaScript The Definitive Guide 6th Edition - 2011
5. Шлеер С., Меллор С. - Объектно-ориентированный анализ моделирование мира в состояниях – 1993
6. Рамбо Дж., Блаха М. - UML 2.0. Объектно-ориентированное моделирование и разработка (Библиотека программиста) – 2007
7. Switch-технология — автоматный подход к созданию программного обеспечения «реактивных» систем – А. А. Шалыто, Н. И. Туккель – 2000
8. Программирование с явным выделением состояний – А. А. Шалыто, Н. И. Туккель – 2000

**Электронные источники:**

1. <http://www.developer.com/design/article.php/2238131/State-Diagram-in-UML.htm> - Mandar Chitnis, Pravin Tiwari, & Lakshmi Ananthamurthy - State Diagram in UML – July, 2003
2. <http://cyberleninka.ru/article/n/ispolzovanie-porozhdayuschego-programmirovaniya-pri-realizatsii-yazyka-opisaniya-diagramm> - Новиков Ф. А., Степанян К. Б. - Использование порождающего программирования при реализации языка описания диаграмм - 2008
3. <http://stackoverflow.com/questions/1422480/graphical-finite-state-machine-editor> - Graphical Finite State Machine Editor
4. <http://c2.com/cgi/wiki?EventDrivenProgramming> Concept of Event-Driven Programming