Background reading:

General background reading:

250

Previous applications:

It was noted in the proposal that insufficient attention had been paid to existing solutions in software for the problem posed. I hope to briefly rectify that oversight and present some reading. As programming languages are Turing Complete – as a Turing Machine can be written in Java:

**State state = start;**

**while (true) {**

**char c = tape.readSymbol();**

**tape.write(state.symbolToWrite(c));**

**state = state.next(c);**

**if (state.isLeft()) tape.moveLeft();**

**else if (state.isRight()) tape.moveRight();**

**else if (state.isHalt()) break;**

**}**

(<http://www.cs.princeton.edu/courses/archive/spring13/cos234/www/lib/exe/fetch.php?media=cos233-234-lecture9-universality-comput.ppt.pdf>)

And any language capable of simulating a Turing Machine can equally simulate a push-down automaton. The code above demonstrates an advantage of Object-Orientated languages.

With respect to Java implementations of Pushdown Automaton, I will take three examples, and draw conclusions about the merits of these with respect to the underlying PDA model of my application.

The first is the BRICS Automaton package. Although this is a Finite state machine the underlying representation of states are instructive. The application treats states as effectively treated as a set:

public Set<State> getStates()

It then associated a 2D array of transitions, the first dimension being states, the second the transitions that state has. As shown, on a logical level the implementation of the automata is relatively straightforward, and involves simple references. However the use of fixed arrays and indices to entries in the array instead of object references, and a focus on the logical structure of the automaton, instead of a diagrammatic focus suggest a different pattern can be used that would be appropriate for a model of a GUI application.

<http://www.brics.dk/automaton/doc/index.html>

Architecture:

The decision was made first to use Java. This was primarily because of Java’s scope, in terms of the systems it could be used with, the ease of finding and applying libraries and the open source documentation. For the sort of application that this is, that is a pseudo-enterprise solution, there is a large and helpful community online to deal with problems and for examples and ideas. My familiarity also played a role. I had used Java extensively at work, was familiar with the JDK and its libraries. Java was also the language for the group project of the Object Orientated Programming course at Birkbeck, and the group project I was involved in was written with it. Given the focus of the project was finding a software solution and not learning a new language (which would have been superfluous for this task); Java was the only sensible choice of language.

More specifically I choose the JavaFX framework for the graphics because it offers several powerful features such as ‘binding the UI to properties in a model and change listeners that reduce the need for setter methods, and you have a combination that will help restore Java to the client side of the RIA equation. (Kindle Locations 459-461). Apress. Kindle Edition.)’ Actually, ultimately the model was so decoupled from the application that the binding of properties was rendered unnecessary except in ‘bookkeeping’ controller options, but nonetheless, the features of JavaFX from the outside made it clear it would be cleaner than Swing. More importantly was the fact that it came with mark-up language support in the form of FXML. In particular I wished to take advantage of the fact that ‘FXML is particularly useful if you want to be able to treat your user interface as data. This may be the case if you need to load portions of the scene graph dynamically from disk or network without restarting.’(Kindle Locations 9105-9106) in such environments as dynamically displaying details about parts of an automaton. It also allowed me to use the Scene builder software, which may designing an effective layout both expeditious, but also transparent, insofar as the outputted FXML was easily understandable and modifiable. In addition, my proposal submitted an MVC design. Swing, whilst supporting MVC implicitly, was not directly designed to handle it. JavaFX was ‘JavaFX was developed from the ground up with MVC in mind such as the “Designer-Developer Workflow“’. (<http://carlfx.wordpress.com/2009/07/29/javafx-forms-framework-part-1/>) It also has richer and more immediate support for directly drawing objects to the screen and providing a geometric framework in which the user isn’t constrained by having to follow one layout over another. It is also the most cutting-edge framework for graphics, and given the need to create a visually pleasing application, JavaFX’s support for interal CSS specification of the native application (<http://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm>) which includes such graphical ornaments as linear gradients, it seemed a natural choice, and would also be a learning experience.

Database:

After further background reading relating to the implementation of the database on an architectural level, the decision to move from a proposed purely XML system to a traditional database relation model based on the outcome of some initial background reading relating to the implementation of these databases in real applications. Firstly, DDL and non-selection DML (gloss) support is poor in XQuery implementation in Java, specifically insert and update which requires external packages (<http://download.oracle.com/otn-pub/jcp/xq_api-0.9-prd-oth-JSpec/xquery-0_9-pr-spec.pdf?AuthParam=1378512161_7d8a23a2b9e7ef594a0baf7031045e6a> 97). More importantly the Java library does not support inserts as a pre-packaged functionality, instead requiring plugins. This was a concern for a system whose database was to be heavily weighted towards CRUD (gloss) operations rather than hierarchical query results, which is more the goal and aim of XQuery’s FLWOR query structure(<http://www.w3schools.com/xquery/xquery_flwor.asp>). As the internal structure of the XML is not used by the database (which is mapped to objects), my reading turned instead towards the existing use of XML in relational databases –specifically that many databases allow the storage of XML as ‘clob’ objects or as XML type objects. (h2 documentation) This would be a relatively simple way to implement a nonetheless complete CRUD system for the automata. Another reason for using relational databases was that I wished to have an embedded database that could be portably built with the application itself. Specifically the database for this system will be tightly coupled with application in question, with the use of the database invisible to the user and segregated from other applications (<http://www.cs.virginia.edu/~son/cs851/papers/embedded.computer00.pdf>) but retaining the logging, failure recovery and concurrency features of a standard relational database. Most XML databases are not sufficiently mature to offer embedded support, and certainly not within the context of a Java project. By contract the H2 system comes with a jar which can be built with the project as a JDBC driver, and has database file that can be used and manipulated with nothing more than the library embedded in the classpath. Moreover, it is directly linked to the Java classpath, and so can use Java classes in the project such as classes defined by an interface for triggers (<http://www.h2database.com/html/features.html>). As a consequence this was selected as the database.

System Analysis and design

Introduction:

System Purpose:

The purpose of the desire system is to enable the user to have a toolkit with which to explore push down automata. The aim is to take what are abstract mathematical entities and to allow them to be used as dynamic objects, things the user can personally design, manipulate and explore. It should allow the user more than just an animation, the user must be able to quickly design an automata that can then be experimented with.    
    1.2 Scope of the system

The system’s core ‘must-have’ scope is specifically focused on deterministic push-down automata, and will ignore the more complex case of Turing Machines, although finite state machines can be trivially simulated with the more complex pushdown automata. It will be an internally consistent program, which will run in its own space and not have interfaces with external systems, except insofar as an XML format will be sufficiently standardised that other applications could, in theory use the data generated for their own processing. The system have three overarching use cases: the simulation, the design and the persistence of these machines.

‘Nice-to-have’ aspects of the scope include batch processing, automatic arrangement of graphical elements in the design canvas, non-determinism and help features.

    1.3 Objectives and success criteria of the project

After having spoken to the supervisor of the project, one clear metric of success was identified: aesthetic quality. A successful application, to be used in an educational environment must specifically be visually satisfying. Implicitly this suggests a clear and well balanced GUI design that is easy to use and can be taught relatively quickly.

As a corollary to this, the system should faithfully represent pushdown automata in a manner fully consistent with their mathematical description.

Other success criteria include the limitation of bugs in the system – in an educational environment testing is important as the users are likely to be distracted and to be negatively affected by obvious or careless bugs.

In addition, the system should be self-contained and easy to deploy without any dependencies on external programs, and should involve minimal effort on the user’s part in setting the system up.

As-is system:

The current ‘as-is’ system is in fact non-existent, as an IT product. The system is not intended to replace to actual process of learning about the system through lecture slides and their presentation in-class. This will remain as is. What it is to enable is an extension of the present system of the student’s reading of textbook descriptions of PDAs. Often these give only a few classical examples (see textbook sipser) and such static methods of learning to not take advantage of the implicitly dynamic computational nature of these machines. As such, they are to act as a compliment to the textbook study of PDAs, forming a study-aid.

To-be system:

The new system then is to be a computer program, run natively in the user’s operating system, that acts as an automatous executable for the user to avail themselves of privately. It is envisaged to be a small program, one that can be downloaded say from a Virtual Learning Environment (such as Moodle, used by Birkbeck College), where it would accord with the same social constructivist pedagogical school of thought as Moodle itself, specifically the principal that ‘Knowledge is strengthened if you can use it successfully in your wider environment. You are not just a memory bank passively absorbing information, nor can knowledge be "transmitted" to you just by reading something or listening to someone.’ (<http://docs.moodle.org/24/en/Philosophy>)

Functional Requirements:

The following requirements have been identified as ‘must-have’:

* There should be an underlying data structure representing a Push Down Automaton, and it must replicate faithfully the mathematical definition defined in the background reading section.
* It must be capable of succeeded on both a final state, and when the stack is empty, according to how the user has defined the automaton.
* The Push Down Automaton must have a GUI interface.
* The GUI interface must be dynamically animated, showing transitions as clear graphical transitions.
* The GUI should be aesthetically pleasing, in so far as it will be clean and make use of modern 2d graphical techniques.
* The GUI should report to the user to errors, successes and other events that take place in the operation of an automaton.
* The system must allow the user to create their own push-down automata in a straightforward and intuitive manner.
* The system must allow the user to persist automata between uses of the program preferable using structured data such as XML.
* The system must support error handling and graceful fail conditions on the automata regardless of user input.
* The system must allow the editing of existing automata
* The system must come packages with a very small library of standard textbook examples of PDAs.

‘Nice-to-have’ requirements include:

* A system to handle batch processing file, to take in full inputs of comma separated strings, which get processed individually.
* An automated node analyser, which takes a user drawn PDA, and arranges the nodes in an order specified by a standard graph drawing algorithm, which in the proposal was specifically recommended to be the Sugiyama algorithm.
* The pushdown automata should allow for epsilon transitions between nodes, generating alternative paths through the system, thus replicating non-determinism.
* A help system whereby the user is guided towards the correct use of the system some form of packaged help files that can be viewed from the application itself.
* The program is to be written in Java for maximal cross-platform support, and is to use the JavaFX library for the GUI
* The animation should have variable speed limits
* There could be sub-routines defined as automaton within automaton

Nonfunctional Requirements:

The following non-functional requirements have been identified:

* The disk-space footprint of the program should be low, so that it can be downloaded from a Virtual Learning Environment.
* The program should be easy to run and require no pre-configuration.
* The program should operate without noticeable update latency, no action should block the main drawing thread of the JavaFX program (<http://java.dzone.com/articles/javafx-sockets-and-threading>) and the graphical actions should appear instantaneous.

Documentation:

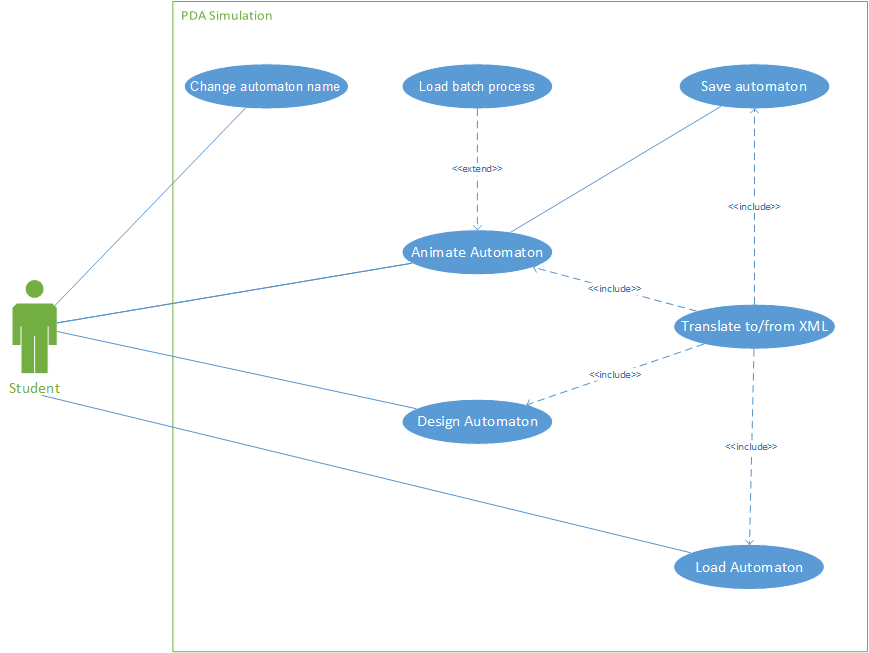
In this section the primary aim is to present the system as an analytic entity, both statically and dynamically. This serves both as the user-eye’s perspective of the system, and thus delineates the logical interactions of the user with this system, and the logical totality of the system taken as a pure domain without reference to implementation features, which should be driven by the demands of this analytical exercise.

Whilst the system was developed in an agile manner – using test driven development techniques and extreme programming methodology, the presentation of these documents is designed to demonstrate the analytic rigour of the final document. It has also been created to justify the decisions made in the implementation section, from a more high-level perspective. Finally, as Alistair Cockburn has suggested: ‘The main success scenario of each use case provides everyone involved with an agreement as to what the system will basically do, also, sometimes more importantly, what it will not do. It provides the context for each specific line item requirement, a context that is very hard to get anywhere else… The full use case set shows that the investigators have throught through every user’s needs, every goal they have with respect to the system, and every business variant involved’ (<http://alistair.cockburn.us/Why+I+still+use+use+cases>) To use a specific example, it became clear that there needed to be clarification on the action of the system after an automata had been saved in the designer tab. By constructing a use-case view of the system along with an activity diagram, the analysis pointed to extra logical requirement for the user to be given the option to reload the automata back into the animation tab.

The diagrams used have been built using Microsoft Visio under the DreamSpark license.

Use case model

The following diagram is an overview of the interaction of the use cases, specified in more detail below:



As can be seen, the user is the student and is the sole creator and animator. Whilst a lecturer may potentially create automata for student use, this is not considered a primary use-case as the user typically is excepted to be ‘learning by doing’.

Note that the translation from/to XML is at the centre of the use-cases in the diagram. The static data of the XML is the logical hub by which different portions of the system interact through the static definition of an automata

Use case specification:

The model used follows Alistair Cockburn’s recommendations (<http://alistair.cockburn.us/Basic+use+case+template>), one of the guiding spirits of the agile movement and the Latex plugin provided by Tom Desair (<http://www.tomdesair.com/blog/2012/04/latex-template-for-use-cases/>).

Use Case: 1 Animate Automaton

Goal in Context: The user runs an input through the automaton and achieved a result

Scope: The PDA runner tab

Primary Actor: Student

Priority: Critical

Frequency: Regular

**Trigger:** User loads an existing automaton from the database

**MAIN SUCCESS SCENARIO**

1. User inserts input into the queue.
2. The user selects run
3. The next unprocessed string is taken up
4. The automata transitions to a new state based on the current stack and the transition symbol. The stack is modified according to the specification of the transition – a single symbol can be removed, an arbitrary number of symbols can be added.
5. The system returns to step 2.

EXTENSIONS

4a. The system encounters errors it goes into a fail state and the automata cancels

4b The system ends, it calculates based on the type of automata in question whether or not it has succeeded in processing the string

SUB-VARIATIONS

1. The user may use a CSV file to input entries
2. The user presses play. The rest of the steps until the string is fully processed are now automatically processed by the computer.

4a. The automata is nondeterministic and has addition states it can return to on the string. It returns to step 2 at this new state

Use Case: 2 Design Automaton

Goal in Context: The user creates a set of nodes and transitions to build an automaon

Scope: The PDA designer tab

Primary Actor: Student

Priority: Critical

Frequency: Regular

**Trigger:** User selected design automata tab

**MAIN SUCCESS SCENARIO**

1. User creates a series of nodes and transitions
2. User edits the nodes and transitions so they have the correct values.
3. User saves PDA.

EXTENSIONS

2a. The user creates an infinite loop of epsilon transitions, which could crash the system. The attempted transition is cancelled

2b. The user creates an illegal transition by creating two transitions pointing from the same node with the same character. The transition is cancelled

3a. The user attempts to save, but save fails as no start node is selected.

SUB-VARIATIONS

1. The user loads an existing automata, and so skips to step two

Use Case: 3 Translate to/from XML

Goal in Context: The user requests a save/load from the database and requires the XML is serialised or deserialised

Scope: The save/load process

Primary Actor: Student

Priority: Critical

Frequency: Periodic

**Trigger:** User selected design automata tab

**MAIN SUCCESS SCENARIO**

1. User selects save
2. Object is translated to XML from designer
3. Object passed to database

EXTENSIONS

2a. An illegally defined PDA is loaded, if generated by a system bug or incorrectly inserted XML fragment, will trigger XSL validation error

SUB-VARIATIONS

1. User loads XML

1. XML translated to designer and animation model

Use Case: 4 save automata into database

Goal in Context: The user saves an automaton, and its XML fragment is persisted in the database

Scope: The database

Primary Actor: Application

Priority: Critical

Frequency: Periodic

**Trigger:** XML fragment passed on by use case 3

**MAIN SUCCESS SCENARIO**

1. User is given option of name for automaton, and an option of known directories
2. XML is persisted in database
3. User view of database is refreshed
4. User given option to refresh automata screen

SUB-VARIATIONS

1. User saves an existing file, no prompt is given for name or directory.

4 a. User does not select, nothing happens

4 b. User does select, automaton loaded into automata view with use case 5

Use Case: 5 Load Automataon to screen

Goal in Context: The user requests a load of an automaton from the database

Scope: load process

Primary Actor: Student

Priority: Critical

Frequency: Periodic

**Trigger:** User selects an automaton in the database view

**MAIN SUCCESS SCENARIO**

1. User selects a persisted automaton
2. Object is passed to the XML translator

SUB-VARIATIONS

1. User has loaded an automaton that was just saved; the XML is specified to only pass on to the animation tab

Use Case: 6 Change automaton database details

Goal in Context: The user requests a load of an automaton from the database

Scope: Database tab

Primary Actor: Student

Priority: Unimportant

Frequency: Rarely

**Trigger:** User selects an automaton in the database view for change

**MAIN SUCCESS SCENARIO**

1. User selects an automaton
2. User changes name of automaton

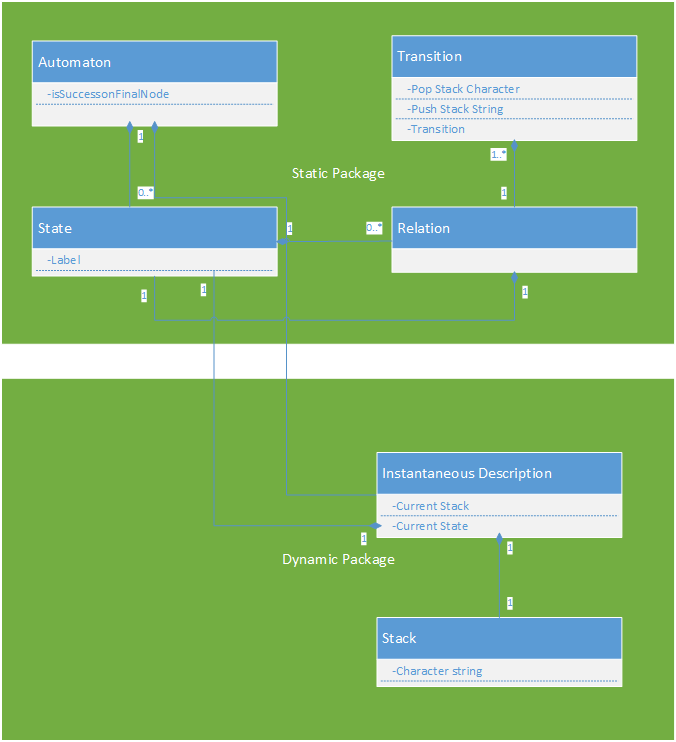
SUB-VARIATIONS

1a. User selects a directory

2a. User changes name of directory

* + 1. Analysis object model

The class diagram here represents the domain model. This is one of the patterns of enterprise specified in Patterns of Enterprise Application Architecture by Martin Fowler (pg. 116) The domain model captures, as a class, the fundamental entities that make up the static structure of the objects that the system hopes to represent in a structured form. <http://www.aptprocess.com/whitepapers/DomainModelling.pdf> This is a fundamental aspect of object-orientated programming (the paradigm the Java language used in this project is a part of), as it ‘concentrat[es] on entities first, and letting the behaviour follow the entities4’. Each object representing an entity in the domain must be well specified, so the behaviour of the implemented objects is cohesive and natural to the programmer. This specifically results in placing responsibilities between them.

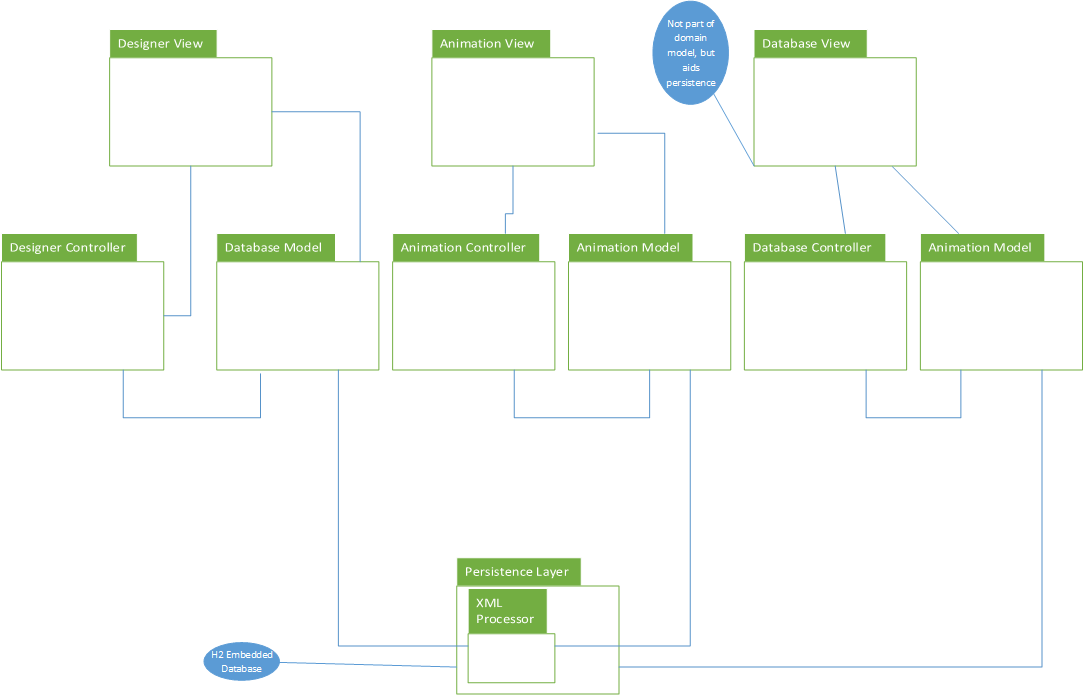
More specifically ‘A domain model is the abstraction of what … the system does14’. This is contrasted by the class model, which is a description of the actual classes in the system, and will be specified in the implementation. However, there is an important overlap between the two. The domain model now specified will see itself instantiated in four separate areas of the system. To wit: the model of the animation, the model of the designer screen, the XML fragments of the automata in the database and the staging area. By being mindful of the basic domain model, and its instantiations throughout the system, one can ensure that the individual models underlying different parts of the system are isomorphic to the original domain model. This ensures that they can be easily converted to each other between different layers of the system, such as the view, the persistence model and so forth, by use of mapping objects and methods and there is no a concern that the final interface between packages will be inconsistent. The following diagram explicates the problem domain in its essence:

As explained beforehand, because of the varied nature of the system, the problem domain layer will be split between three packages. The package is an ill-defined term, but generally means either a layer of software or a set of closely coupled classes that frequently communicate, or define each other. (Wilxcom or whatever 251) In my case it is a useful abstraction, as different parts of the system will use different aspects of the domain model, and thus seeing the fundamental relations of classes is important to ensure that the classes written are cohesive and use coupling only where necessarily demanded by the domain.

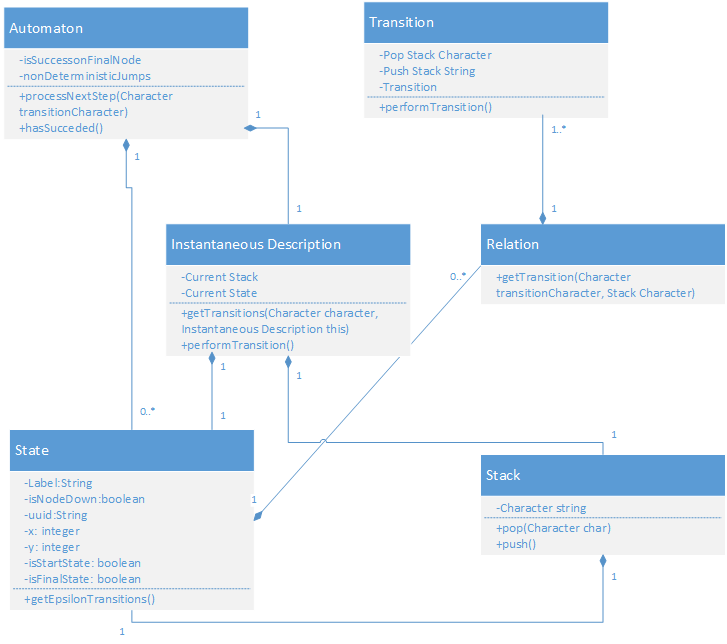
As can be seen, the essential domain model consists of two related by importantly distinct parts: the static definition of the automaton, and the dynamic aspect of it. The static definition is in essence simply the stages that constitute it, and their relations, defined in a one-to-many set of transitions, which define permissible characters and stack operations. This static aspect of the domain model will find itself reified in all aspects of the system where the domain model impinges it.

By contrast, the dynamic aspect is represented by the instantaneous description, and the stack. These are elements that define an automaton in action (along with the input, which is a part of the view). Thus they form a related, but distinct set of actions which are unique to the dynamic aspect of the system, the animation.

By factoring this domain model into the system requirements aforementioned, and by virtue of the Model View Controller paradigm (as well as the persistence layer) that will be used in this application (as justified in the implementation), I will have an application that will have a class model that takes the above domain and splits it into further packages by a process of mapping the domain to the logical layers of the system factoring out methods. (wilxcom) To this end the initial package diagram, a diagram of ways in which the broad areas of the application interact, displays the overall architecture of the designed system:

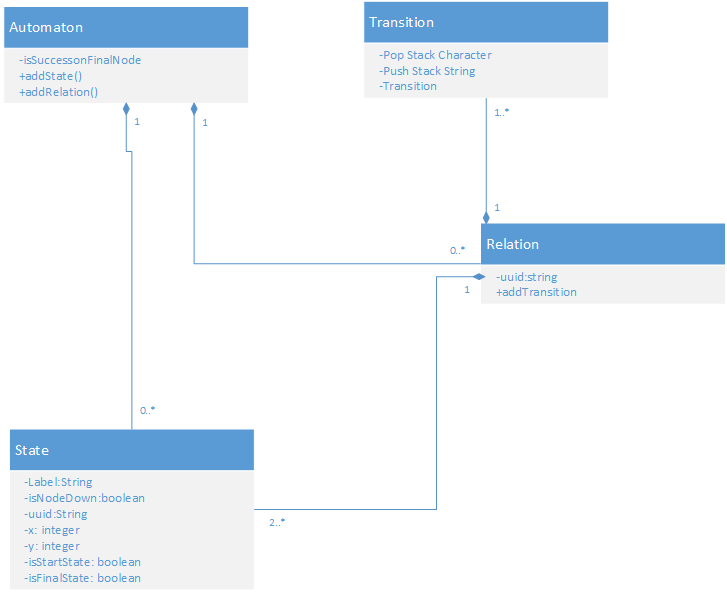


As can be seen, the three screens are their own individual MVC loops, and the persistence layer (mediated by XML) is the underlying connector between them. Most of these packages are fairly unenlightening from a design perspective, and for the sake of brevity their details will be ignored. However, the animation model and design model packages, on which so much hang, are an important component of the system, and their design was an important step in the development of the application, and ensuring it operated according to the specified requirement to replicate the mathematical model.



The design here deserves explication. An important change is the direct composition between the instantaneous description and the automaton – in the domain model these are logical separate entities, but a real system will need a composition between them to send messages. Whilst it broadly represents the classes specified in the domain model, it equally expands them with methods that define their action. The action of a pushdown automaton can be simply described in the diagram specified in the dynamic model, figure 1. The following class works to implement this logic in the following way:

* The get transitions on the instantaneous description is called every time the instantaneous description changes.
* Epsilon jumps are added by virtue of a call from the instantaneous description to the associated state
* An appropriate transition is found by way of the relations and transitions classes (including matching the character to be popped and the top symbol of the stack)
* A transition is made and the Instantaneous description is updated
* The automaton is notified of the progress of the above and falls into success or error state depending on the result of the above, the currently processed string and the exceptions that may be triggered by any of the classes below it.

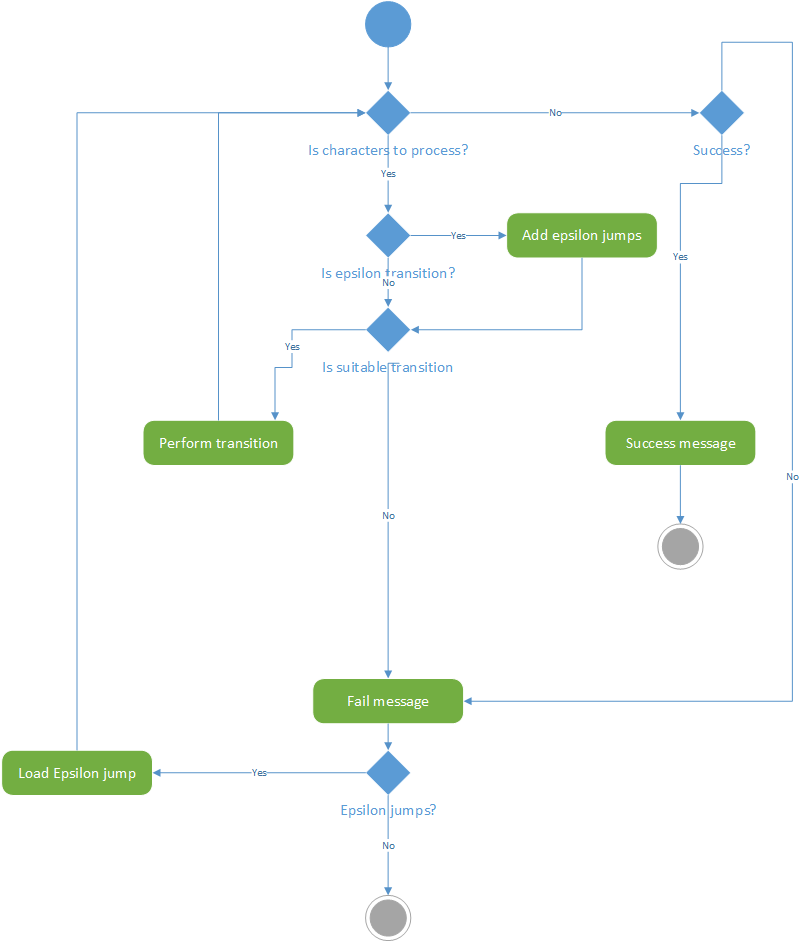


This figure details the inner working of the drawing model package. As one can tell, it only replicates the static elements of an automaton. More importantly the automaton is defined directly with references to relations and states. This is to ensure that relations can be processed as a separate array of items, which aids the clean design of many of the controller functionality, including drawing. However, in the XML process they are flattened into the state, and this is enforced by the XSL drawing. Most of the model is static and definitional, unlike in the previous document, most of the activity occurs in the methods defined at the automata model, triggered by the view. It can be said that the designer is a view-driven model, whereas the animation is a model-driven view.

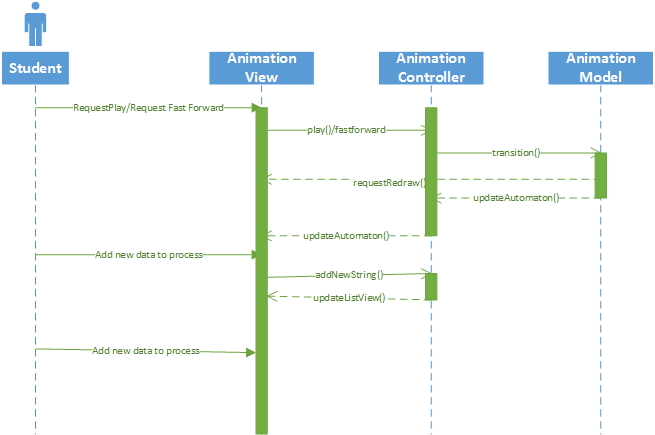
        3.4.4 Dynamic model

The dynamic model here specifies the ways in which the system operates over time. As the system is a state machine, it was decided to avoid using the UML state machine diagram – as it may not be especially enlightening!

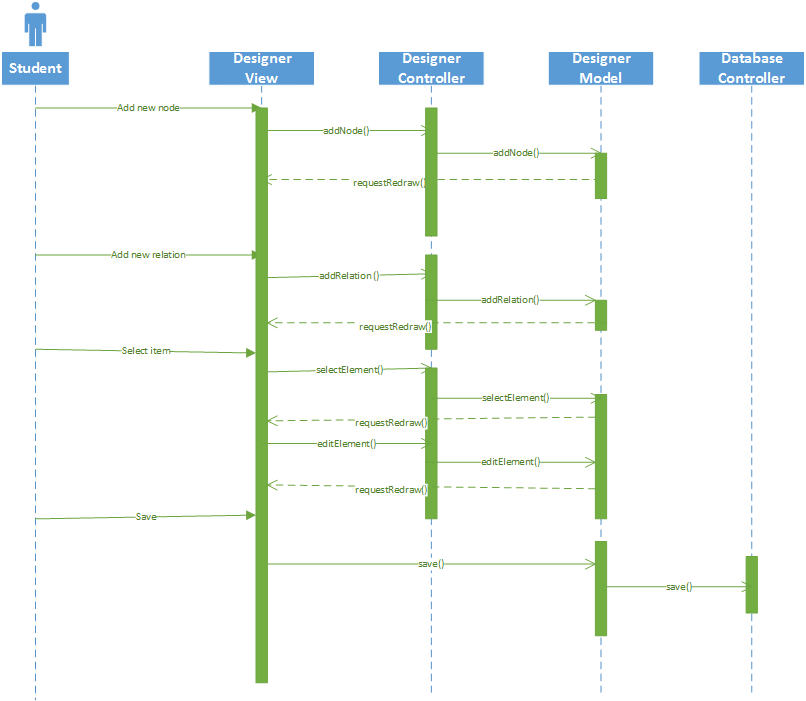
The following explicates the essential process of running an automaton, intended to support the decisions made in the class model designed in the previous section:



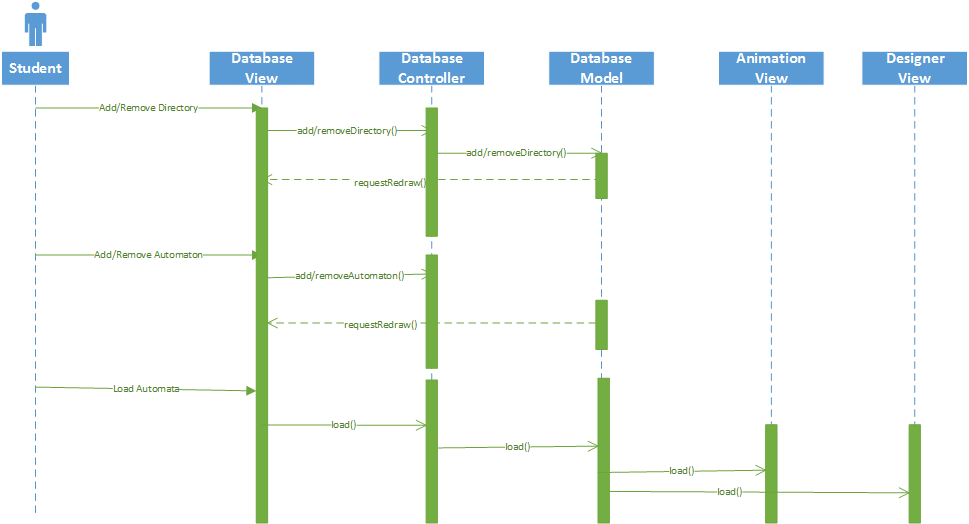
More importantly are the sequence diagrams I present below representing various relations in the system. The solution to developing a system to manage the potentially frightening degree of complexity in this, and to limit coupling where possible, is presented in the implementation section, however this gives an overview of the kind of actions that are necessarily required of it in any system matching the requirements:



This is a typical user interaction with the animation view. The user requests a play or a fast forward on a pre-existing animation with some data in the animation view. The model must be updated, and both the state of the computational context (controlled by the controller) – that is what is being processed and what has been processed – and the state of the automaton itself must be updated, and the view informed of this change. After this the use selects a new item of data to the listview. As the listview is a controller held collection, this communication does not reach the model. As seen, there is a great deal of communication between different parts of the system, and a hierarchy of (possibly mutually reinforcing) messages. Indeed, some of the messaging above could take the form of a loop, when the animation is run in fast-forward mode. Clearly a proper messaging framework is required for this - as with other parts of the system.



This sequence diagram represents typical user behaviour in the designer view. It consists of the user creating both nodes and relations, and then selecting and editing them, before finally saving the item (which gets passed to the database controller). This is a simpler MVC loop than before, but the great number of possible types of messages (both displayed here and subtypes and exception conditions of above), again suggest the need for a centralised messaging handling system. And reinforcing this is the pattern outside of the MVC loop to the database.

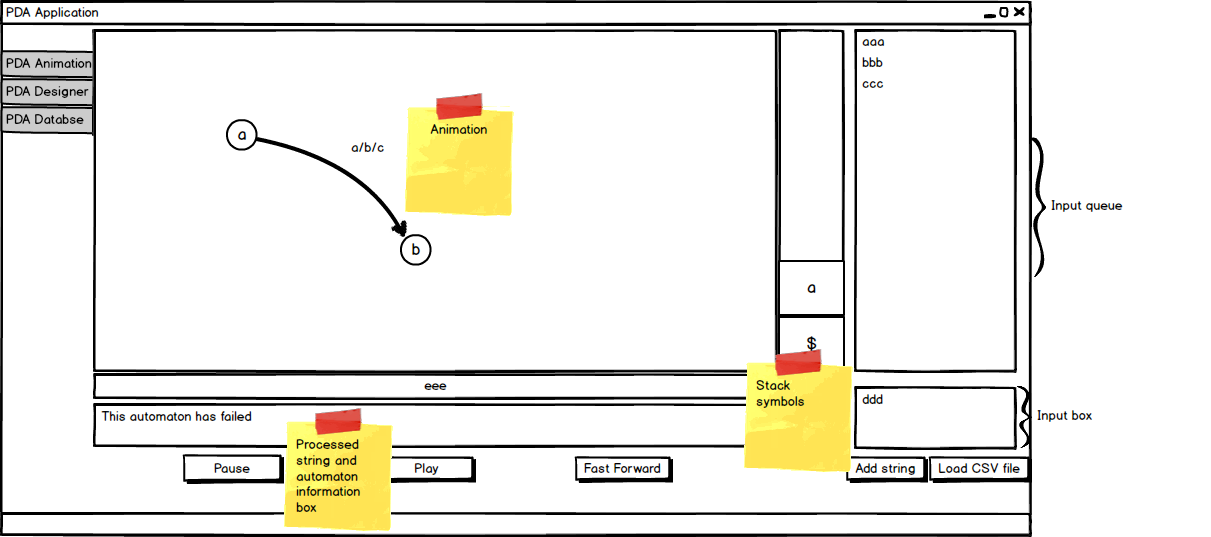


The database sequence is relatively simple, consisting of edits to the tree view and a final load request which gets passed to the animation view and designer view. Again, the volume and the range of the messages should be noted.

  3.4.5 User interface screen mock-ups

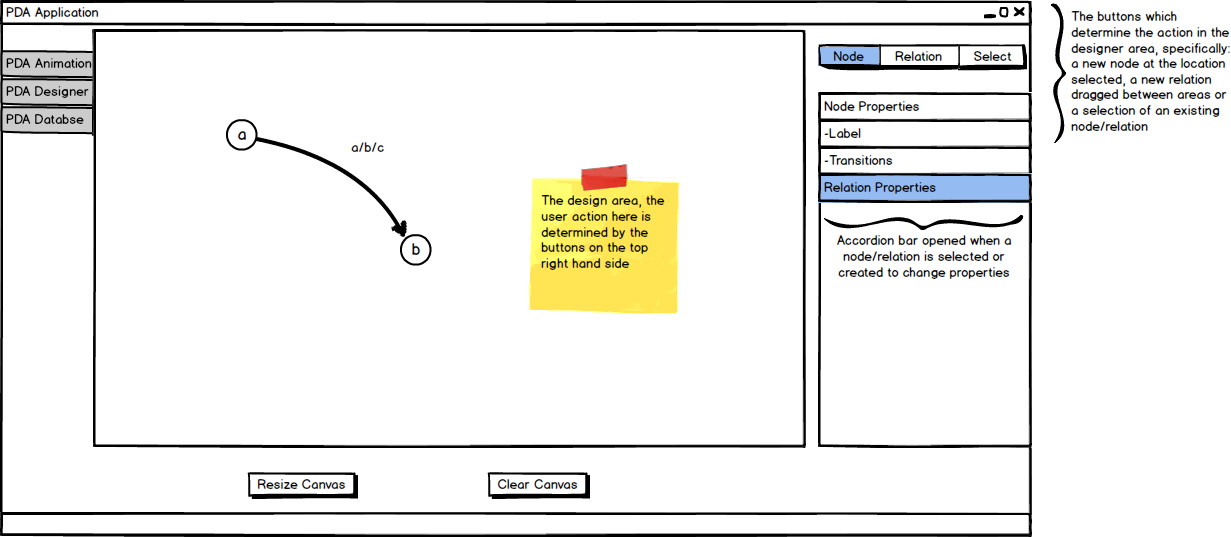
The following mock-ups, created by the Balsamiq program, serve as indications for the design of the GUI. These represent, in effect, a specification for the view of the system.

Animator screen:

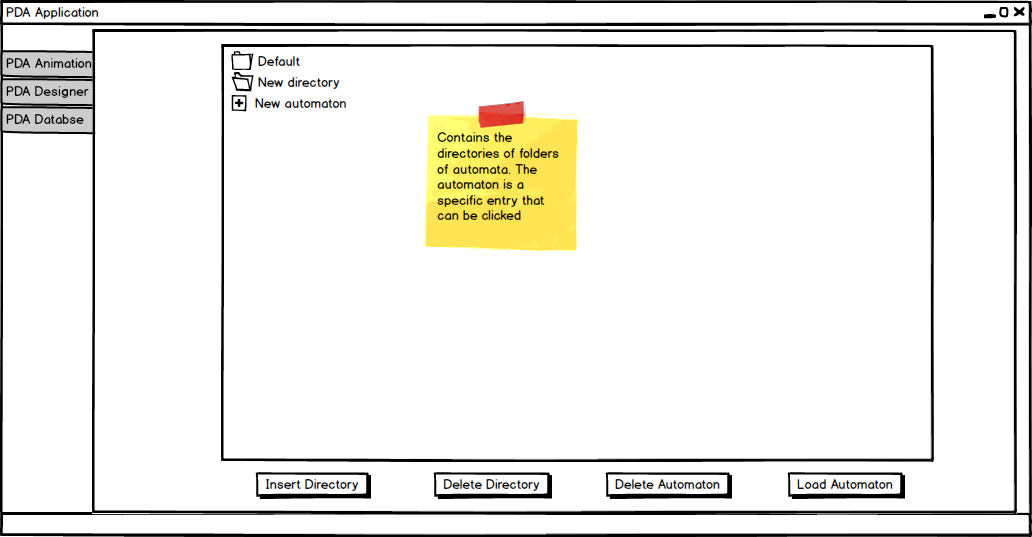


The design of this screen is based on a bilateral split between the user input on the right, and the animation of the left. The user enters strings of text into the list view on the right, and then clicks the action buttons at the button to set the automaton in motion. This may seem confusing at first, and I did reconsider the design to simplify it. However, it has the virtue that the loading stage and the operation stage are visually separated and the output animation has its own screen, whilst retaining the dynamic functionality of the queue to the side. The system allows a new user to quickly set up a set of conditions and run them sequentially, gaining information on the response of the automaton to the various items in the queue.

Designer screen:



The following screen demonstrates the design of the designer console. The user selects one of the buttons in the top right hand corner. Every node or relation entered, or selected expands the accordion (controlled by the application), and reveals a set of properties that can be set by the user. This allows the user to flexibly set properties without invoking pop-ups.

Database screen:

The database is viewed by the user through a tree view as specified in this image. The user can click on any directory and open up its contained list of automata. These automata can then be loaded as desired. Other utility functions are performed, including editing the items by directory editing the tree view nodes through a dynamic textbox.

Note:

The analysis and design above defines the salient elements of any system that fulfils the requirement. It does not define the controllers, because the task of communicating between both is an implementation issue, depending a great deal on the GUI framework involved. It does not also define the persistence format (although this has been selected in background reading), because again this is heavily implementation specific, depending on the environment and tools used. The eventual shape of these elements of the system will be demonstrated in the implementation.

Implementation:

General Principles:

The overriding goal of the implementation has naturally been to build the system described up till now. However, given the nature of the project, and the desire for extensibility and the possibility of expansion, I have been mindful of the need to make the project maintainable. To this end interfaces have been sought where possible. For example, the automaton implements the IAutomata interface, which whilst designed from experience of building the system, forms an aid to refactoring. Should it be wished to program the automaton in a different form, say utilising some of the data structures mentioned in other versions of the implementation, the interface could be used to define the function necessary to operate it - some of which could, as part of the refactoring processed, be discovered to be best placed in a joint abstract class - and built without requiring changes from other parts of the system. At a higher level, I have made use of Design Patterns, which take these interfaces and adapt them in ways which, whilst preserving the benefits described above, also form what effectively amounts to a flexible computation structure of their own, for example a strategy pattern which allows a class to dynamically modify its behaviour at runtime, but remain under the control of a normal program flow. Whilst the book ‘Design Patterns Elements of Reusable Design’ correctly emphasises the toolkit nature of design patterns (gof pg 12), as they say ‘The design patterns… are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context’, they actually also form means of abstracting an algorithmic domain at the architectural level, providing logic with extensibility.

System wide frameworks:

JavaDoc:

I have attempted to use the JavaDoc notation (which is automatically generated a set of HTML files by Intellij, the IDE the project was written in) as a reference for as many methods in the system as possible, to aid maintenance and to increase the possibility of the system being successfully extended. They also form a subsidiary document to the topics discussed here.

Lombok:

The Lombok library was used throughout. This can be seen in any code marked with ‘@Getter’, for example. It simply was an attempt to reduce ‘boilerplate’ code in the application so that the core logic was easier to find.

Dependency Injection:

One of the more fundamental design patterns is the factory method pattern. In order to make use of the large number of interfaces in my system, it has been necessary to abstract away the creation of new objects to factory methods. As stated, in the Gang of Four book, ‘The framework must instantiate classes, but it only knows about abstract classes, which it cannot instantiate ‘. Actually, in this application, as all classes do know about all the subclasses this is not necessarily an issue, and yet it is highly inelegant and error-prone to utilise the ‘new’ keyword for each instantiation and not easy to refactor. However, pure factory methods would, by themselves, constitute only a partial improvement, abstracting the classes but not forming a coherent container or interface through which to do this. This would be a contained class that automatically resolves the dependencies asked for through reference to a configuration, rather than a hardcoded factory class. This is known as the Hollywood Principle (Manning book pg. 13). Another benefit of DI is that by defining dependencies in a separate module the code itself is interface based, and is therefore behaviourally focused’, or as said in the book Depedency Injection ‘the crux… is the fact that all you code is freed from constructing dependencies. Code written with DI in mind is behaviourally focused without distracting clutter’ (Manning pg 21). This therefore aids modularity, which, given the complexity of any one of the modules of my system is important, and of course aids testability.

As part of the process of developing the system, there is therefore the need for dependency injection – the name of this process, and a framework which encapsulates this. However, the dependency levels in the system were not sufficiently involved or deep that a heavy-weight Depedency Injection framework such as Spring was necessary, and the bulky XML files that attend such complex object graph construction and all the other features that come with Spring. For many of the classes in my system a light-weight system that is not permanently visible context but one that disappears with fluent API to bind objects in a maintainable way. In Google’s Guice framework, which after some consideration was chosen, it was possible to specify a whole package of modules which, via the overriding of a single method could construct injections in a compact way within Java itself, which provides the benefits of type checking classes.

The typical Guice module extends an Abstract Module and implements a set of bindings within the overridden configure method, which define the injector’s behaviour towards any inject tags placed at fields, setters or constructors of a target class. For example, for the IAutomata interface the following configure method is called:

@Override

protected void configure() {

bind(IAutomata.class).to(Automata.class);

}

The start method of the PDAGUILoader, the starting class of the whole application, creates the following object:

Injector injector = Guice.createInjector(…

new AutomataModule());

injector.injectMembers(this);

Which ends up injecting a field, annotated to accept Depedency Injection:

@Inject

private IAutomata animationModel;

The real power of this method in this particular application was felt with the Eventbus objects (defined later) and the database connections where static classes were required. Static objects were defined within the module, such as:

private static IConnectionContext CONNECTION\_CONTEXT;

This could then be implemented in specific classes to inject the instance without requiring the proliferation of a static object references outside the DI framrwork, or the creation of a static factory method. Interestingly the same functionality could have been provided using the Guice Singleton scope, so this is a target of refactoring. In addition, multiple such static objects (such as different types of Eventbus) could be instantiated using separate dependencies, often aided by the use of customised annotations (defined in the application) such as the annotation ‘@Drawing’ which instructed the instantiation of the drawing Eventbus object. The code for this custom annotation is appended below:

@Retention(RetentionPolicy.RUNTIME)

@Target({ElementType.FIELD, ElementType.PARAMETER})

@BindingAnnotation

public @interface Drawing {}

In addition constructor injection is available, which is often useful when a dependencies need to be resolved within an object constructed by the Eventbus module, for example from the ConnectionContext to the Connection and Eventbus it requiress. The ConnectionContext is defined as SingletonScope with the relevant annotation, so only one instance ever gets injected in any particular injection, but the actual resolution of its dependencies is handled by virtue of constructor injection taking advantage of the static instances constructed in the DI module and the argument of the constructor which ties these bindings to the constructed class.

One final aspect should be mentioned in the use of DI: it was possible to inject automatically generated factories to customise the injection of a class so that certain fields can be resolved by DI whilst the rest are left alone. For example there was a generic interface for the Drawing Template classes (detailed later) – which was instaniated as DrawingTemplate module, whose constructor required a pane Pane and an EditTemplate module whose constructor required the vector graphics context. The solution to ensure both could be injected but allow the class to be associated with a specific type of drawing object was to define interface factories that simply instantiated the generic interface, e.g.:

public interface IGUIFactory<T,U> {

T create(U guiTemplate);

}

Implemented by:

public interface IDrawingFactory extends IGUIFactory<IDrawingTemplate, GraphicsContext> {

}

And a module that built the factory:

install(new FactoryModuleBuilder()

.implement(IEditTemplate.class, ConcreteEditTemplate.class)

.build(IEditFactory.class));

The constructor for this class had a field marked as assisted:

ConcreteDrawingTemplate(@Assisted GraphicsContext gc)

This would inject the factory into the class which required the drawing object, with the rest of its dependencies resolved, and resolve the one remaining dependency through the automatically generated factory method called in the class which ultimately needed this object (the relevant controllers), i.e.:

Injector injector = Guice.createInjector(new EventBusServerModule(this), new DrawingTemplateModule());

injector.injectMembers(this);

drawingStrategy = drawingFactory.create(canvas.getGraphicsContext2D());

Thus the factory is injected, and created in such a manner that it can create a commonly defined interface for the class which implements the algorithm for drawing.

Eventbus:

The analysis section pointed out the need for handling message passing between classes. Traditionally in a GUI application, including one that makes use of JavaFX such as this, this would be done through coupling between controllers, and between controllers and the model. The size and complexity of the models, as well as the extensive nesting of controllers would render this a tangle of coupling. As a result a more suitable means of communicating was sought.

After some research I came across the EventBus library, produced by Google as part of the broader set of utilities under the Guice framework of utilities. This implements what they describe as a ‘publish-subscribe’ API (<https://code.google.com/p/guava-libraries/wiki/EventBusExplained>). It simply requires objects register themselves with the Eventbus. Once registered the event bus can post event objects. These are then picked up by other objects registered with the Eventbus that implement a method of the same type (or superclass of) as the posted object, and marked as subscribed and are processed accordingly. For example, the following demonstrated such a chain of events.

As part of the process of abstracting the events, I separated the EventBus by context (animation, drawing and database), primarily so that they could be tested independently on in a testing framework- by registering the test case classes and their declared dependencies to specific instances built up in the unit test - away from the rest of the system defined here: <http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/eventbus/package-summary.html> It is also hoped that it makes the code more maintainable. I have created an IEventBus Server, which has the dependencies of the three Eventbuses mapped in. As well as registering and deregistering the Eventbuses to objects, it also switches between them by virtue of an EventObject interface that I have created, which forces all objects to define an enum. As can be seen, this enum allows a particular Eventbus to be selected, enhancing the maintainability of the events by their references:

switch(eventObject.getInstanceToken())

{

case ANIMATION:

animationEventBus.post(eventObject);

break;

case DRAWING:

drawingEventBus.post(eventObject);

break;

case DATABASE:

databaseEventBus.post(eventObject);

break;

}

It can also allow for the future implementation of multi-threaded event handling by application context. Whilst at present this requirement is not needed, should a more intense use case be implemented in say the database (such as continuous updating), it would be nice to process this on a separate thread. The ‘EventBusServer’ class would allow the database bus to be used in this way without blocking other communications.

The registering of the event buses has been handled automatically by Guice, the module which injects the IEventBus server to the object also registers the object. It can also be deregistered, automatically from all three, for example when an automaton is replaced, the old one needs to deregister itself to be garbage collected by the application.

As an example of how Event bus solves one of the communications mentioned above, it will be demonstrated how a node is drawn. The controller strategy for drawing nodes (explained later) uses the following line of code:

eventBus.post(new NodeCreationEvent(x,y));

This created a NodeCreationEvent, which stores the x and y co-ordinates of the object.

The DrawingList objects (the model for the drawing part of the application) has a method:

@Subscribe

public void handleNodeCreationEvent(NodeCreationEvent nodeCreationEvent)

{

String newId = UUID.randomUUID().toString();

addNode(new DrawingNode(nodeCreationEvent.getX(),nodeCreationEvent.getY(), newId));

eventBus.post(new RequestNodeDetailsEvent(newId, ""));

}

This takes the new node, adds it to its list of nodes, and fires a request to the controller. This is taken up by the controller of the drawing screen and the accordion panel of the application is opened with the relevant details:

@Subscribe

public void handleRequestNodeDetailsEvent(RequestNodeDetailsEvent requestNodeDetails)

{

nodeAccordionExpand(requestNodeDetails.getEditID(), requestNodeDetails.getLabel(),

false, false, false, null);

}

The node details are eventually passed back as required.

The model has been updated and the addNode method defined in handleNodeCreationEvent also sends the controller a redraw message:

@Override

public void addNode(DrawingNode drawingNode)

{

nodes.put(drawingNode.getId(), drawingNode);

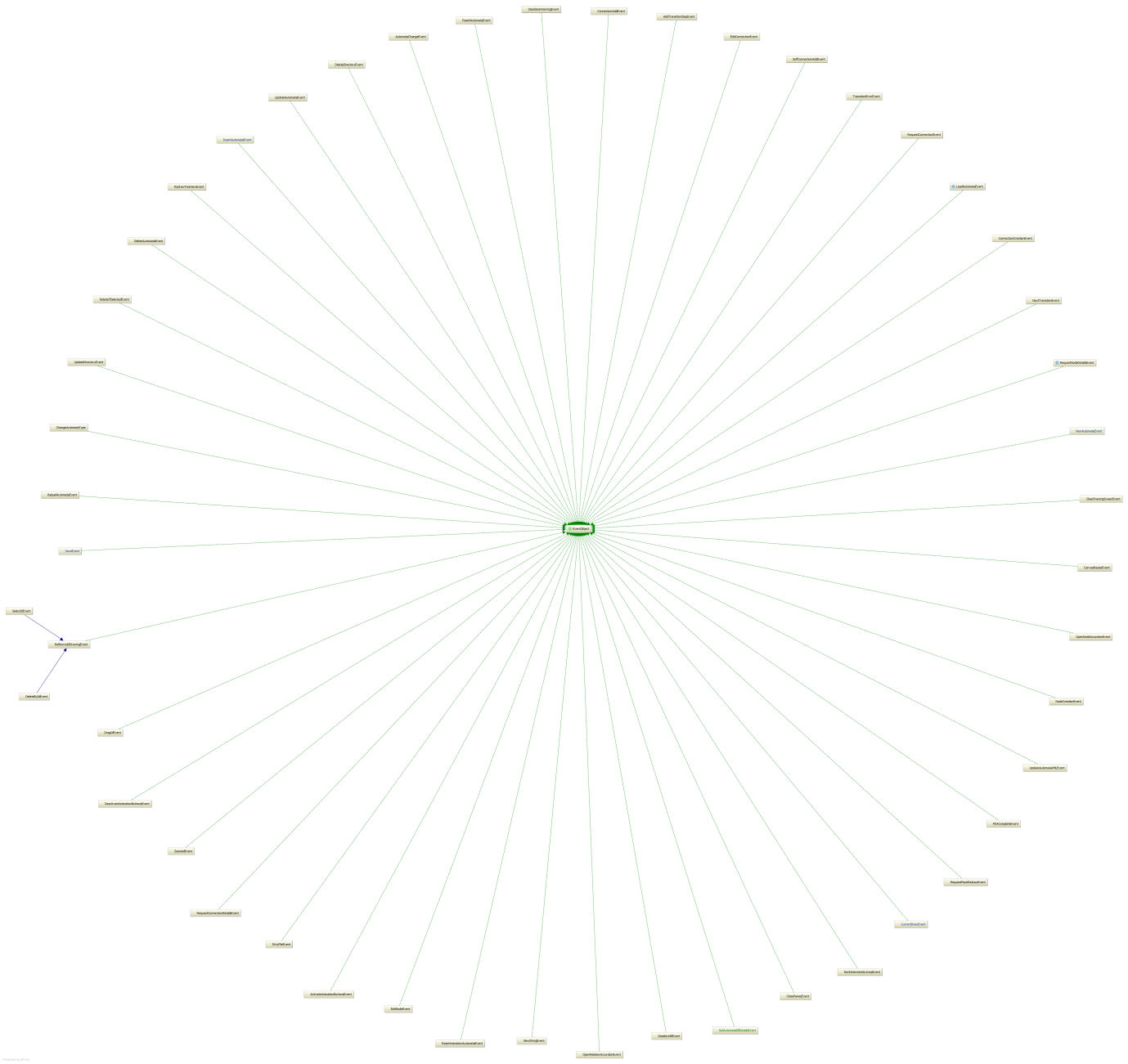
eventBus.post(new CurrentDrawEvent(this));

}

As a consequence the main pane of the panel is redrawn using the actual model. Both graphical parts of the system use this message, by passing itself as part of the composition of an event object, the drawing algorithm can dynamically apply itself to the model every time the model requests a redrawing of itself.

Another consequence of this is that the system is entirely decoupled from the database. Only the ConnectionContext class knows about the database and has reference to the relevant SQL (defined in a properties class). The rest of the application abstractly requests persistence events through the database eventbus.

The following page gives all the events that are handled by the eventbus pattern and the a sense of the considerable complexity and coupling which would otherwise infect the code:



Console:

The console was defined primarily as a means of testing the application during development, where a command-line system is beneficial. It is not intended to be used by the general user, however there is nothing put in place to prevent it being used by someone who is interested and can access the application code directly, and could be used by developers extending the product to quickly test new features or computation models that might be added in (say a Turing Machine extension). To create a reasonably sophisticated environment for console testing I made use of the JCommander library. This essentially works on the basis of a parameter class, which automatically sets up help dialog options such as when the user enters the incorrect commands, and automatically handles the arguments. For example:

@Getter

@Parameter(names = "-input", description = "Input string of data", required = true)

private String inputString;

Defines a parameter whose help file will give pointers to what should be entered and whether it is required. These parameters can then be referenced by the holding object.

The rest of the command line simply takes a requested automaton from the database and processes a user string one by one till the automaton completes, posting any exceptions back. There is no mitigation for nondeterministic machines that grow too large, so this should be used with caution.

A common pattern throughout this application is that CSS for the items is specified inline, so as background colour. I ran out of time to implement an externalised CSS file, which references classes and could define variant skins for the application. This would be an area of possible refactoring.

Dialog boxes:

It is a major flaw of JavaFX at the moment that dialog boxes are not ready for use. Developers must use third party libraries to enable this functionality - or create their own boxes as was the case in this project for the more involved pop-ups. The current standard seems to be ControlsFX <http://fxexperience.com/controlsfx/> - however this requires the JavaFX8. Version 8 of JavaFX, unlike JavaFX2 – which was used for this project – requires the eighth edition of the JDK. This is not supported by Lombok. As Lombok was a more crucial part of the application it was decided not to use this library. (Further mention of JDK8 will be made in the evalution). Instead I made use of MonologFX, a small user-made library available online. <https://blogs.oracle.com/javajungle/entry/monologfx_floss_javafx_dialogs_for> This has served the purpose although strange results are sometimes seen when the system is under load (such as during the warning for excess nondetermism). Ideally it would be refactored to ControlsFX, which is more visually pleasing, when Lombok supports JDK8. In any case such refactoring will be simple. The control boxes are defaulted across the application, held in a couple of static methods in a DialogUtils class. This DialogUtils defines a standard warning message, which is just a void function, and a modal dialog, which gives user choice, as a Boolean, and so can fit unobtrusively into the program flow.

The View:

The view was constructed using FXML files, built in the SceneBuilder package and then modified by hand in the IDE. The FXML view structure forms a general hierarchy of views, which are loaded as one view in the main application, as pda.fxml. It essentially represents a form of dependency injection specifically tailored for the JavaFX objects. The PDAGuiloader parses and loads this file and generates the full view. The code that does this is as simple as:

FXMLLoader fxmlloader = new FXMLLoader(getClass().getResource("pda.fxml") );

This is then set as the main stage of the program.

The PDA view defines three tabs, each of which includes the other views. The advantage of this is that the FXML for each view can be adapted individually and more importantly separate controllers can be defined for each. An example tab is shown below:

<Tab fx:id="runTab" text="PDA Runner">

<content>

<fx:include fx:id="pdaRunner" source="pdarunner.fxml" maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity" minWidth="-Infinity" prefHeight="791.0" prefWidth="1299.0" />

</content>

</Tab>

Each tabs in itself defines its own views, one for the animation, one for the designer and one for the database.

It is also here that the menu bar is defined, and the controller to handle this.

Animation view:

The actual animations are separated into independent views with their own controllers. This is so the respective containers are not too large and that the logic for animation is separate from the user-action logic.

The actual image of the FXML was defined in runcanvas.fxml. This was a simple class containing a vector graphic canvas that was acted upon by its canvas. The stack animation and the processed string animation – that is the text indicating what has been processed so far by the automata, by virtue of processed sections of the string being rendered red, and unprocessed sections rendered -black – are simple panes whose layout and values are set in the FXML. They are given controllers too.

The rest of the view was designed in accordance with the analysis section, and can be examined in the FXML file for it. See the screenshot below for the starting configuration thereby generated. One other aspect is that the buttons have images as description text. The FXML was designed in such a way to reference the image file in the same package and automatically loads the image onto it.

Designer View:

The designer view was constructed in much the same way as before. A simple pane at the start of the application is the main area upon which actions are added. A radio button and toggle button group are also specified for some of the items; this logically connects these elements to one of the two groups and enables special logic on the group. A noticeable addition is that the accordion references two views, which originally were popups (and thus defined in the pop-up folder). These are not visible at the start and are only made visible when a node or relation on the automata being designed has been added to the pane.

Database View:

The database view demonstrates the use of a tree view, which is type of scene node with hierarchical elements. This allows a direct relationship between the database model and the view.

Another important feature of this view, which is replicated elsewhere, is the usage of a HBox. These are elements that unify a set of objects together, say a set of buttons. This means that a visually arresting proportion between buttons that serve a common subset of operations can be intuitively linked in the GUI design.

Help View and About View and their controllers:

As a late edition it was decided (extraneous to the analysis), that to aid with the potentially confusing aspects of the design to new users, that a help file should be added. I wanted these linked together in such a way that they could be simply navigated without a proliferation of views. A solution to this was found in the Webview. The controller for this is associated with a webengine, JavaFX’s default HTML parsing and rendering engine. As a consequence I was able to call the following line of code:

webEngine.load(file.toURI().toURL().toString());

And thus generate the image seen below, specifically an HTML representation of the help file.

These were stored in a separate folder and designed as relatively simple HTML files with a toolbar at the top and some descriptive details on how to use each item of the view. There is nothing, however, preventing their extension or improvement with no implication on the main program.

An about view was also created, to give a professional field and to give some information on the default running conditions.

Controllers:

The controller part of the application is defined as the logic of the view, the controller between what the view updates and the model representing the underlying business model. Traditionally (at least according to the MVC patterns defined ‘the model notifies views that depend on it’ (GOF book, pg?). In this respect I was forced to diverge from the MVC pattern and create something which is somewhat closer to the Model View Presenter pattern (<http://www.wildcrest.com/Potel/Portfolio/mvp.pdf>) as the controllers enforce the logic of drawing to view strategy patterns, or as originally described by Potel ‘the presenter then represents the traditional "main" or "event loop" part of the application, creating the appropriate models’. Mine is a somewhat attenuated version of this as there are several main loops, and the creation of the model is rather under the control of the dependency injection framework and the database/XML serialiser, which can communicate with the model itself. Despite this, the use of FXML makes this formalism unavoidable, and whilst it has the disadvantage of not encapsulating view updates in a cleanly separated area from control logic I have sought to avoid this issue by encapsulating drawing responses into strategy patterns and nested controllers.

Main Controller:

The main controller primarily controls access to the menu bar options, which includes saving the existing automaton in the designer tab, closing the application, the help view and the about view, the last two being described before. These buttons utilise mnemonics, specified in the FXML, so that a user can use ‘alt shortcuts’ to navigate around them so that expert users can begin to utilise it in a more efficient manner.

The save and close button are closely related. The close button simply closes the application as expected. But it also prompts the user to save their automaton, and call the save logic if the user chooses to do so, before saving. This is just an implication of a standard helpful feature.

Saving generates a save event, which communicates first with the drawing model and then with the database through the connection context – which is simply a class which holds all the necessary data for database actions. First the drawing model serialises itself as XML and determines if it is an update to an old model or a new one, and then askes as to whether to animation model should be updated. This means that by saving a designed automaton, the user can instantly reload and test the newly designed automaton, making the application more rapidly responsive to the user.

If the automaton is new it can be saved into one of several directories. If it is an old automaton that has been updated, the drawling list calls an event which asks the update SQL request to simply update the stored XML record for that particular update.

In contrast if the automaton is entirely new, it must have its directory specified. Directories are given by an event call to the connection context, which handles database, which receives a request event to add a new automaton. To loads a pop-up as a view which has a combobox of possible directories to save to, and a textbox to define a name. The AddAutomatonDialogBoxController handles the logic for this. When done it calls an event back to the connection context which adds the new automaton to the database.

Animation Controller:

The animation controller is at the top of a hierarchy of three other controllers, which control the automaton display, the stack and the processed string. These will be described separately. The animation controller deals with user interactions.

The most fundamental parts of this controller deal with adding events to the queue. The queue is a list view – a JavaFX class which contains a vertical set of cells containing displayed objects - which contains strings in the order that they will be processed by the automaton. As with all JavaFX nodes that contain data, the actual ListView is bound to an ObservableArray class – a JavaFX wrapper. The controller logic actually changes this object, not the list view itself which remains an inviolable object derived from the view. The application starts with no automaton loaded here, and there is logic which disables these till one is added. In any case these buttons are defined by handlers which control two buttons – a button which adds an inputted string to the queue, and one which adds a batch file. The behaviour of both is defined in a strategy pattern called ActionStrategy. Essentially the controller simply takes the current value of the textbox (if there are some) from the textbox and clears it. More interesting is the batch controller, which loads a file dialog box which can only select Comma Separated Value files. It takes each comma separated entity, and using the split method of the standard Java strings sends all the results to the list view. This meant that the batch processing requirement was easily met. See the code sample below (inputItems is the ObservableArray):

while ((readIn = selectedReader.readLine()) != null) {

inputs = readIn.split(CSVSPLITTER);

for(String input : inputs)

{

inputItems.add(input);

}

}

An additional feature to this list view that I added was to modify the cells to be customised. To do so I defined a new class which extended the standard cell called RightClickableCell. As the code below demonstrates, it creates a ‘context menu’ – a small menu which briefly pops up – with one option, delete. The cell deletes its own entry from the underlying array and thus replicates a delete functionality. See the code before.

this.addEventHandler(MouseEvent.MOUSE\_CLICKED,

new EventHandler<MouseEvent>() {

@Override public void handle(MouseEvent e) {

if (e.getButton() == MouseButton.SECONDARY)

deleteMenu.show(RightClickableRow.this, e.getScreenX(), e.getScreenY()); }

}

);

The association of this special cell type is effected by a common interface which is the CallBack class, as shown by the code here:

inputStrings.setCellFactory(new Callback<ListView<String>, ListCell<String>>() {

@Override

public ListCell<String> call(ListView<String> stringListView) {

return new RightClickableCell(inputItems);

}

});

As one can see, the class associates a list view with a particular ListCell, in such a manner that the List View interface is able to call (using internal JavaFX code) the creation of particular list cells. It should be noted a reference to the actual Observable List is passed into the cell, this enables the cell to modify the underlying list view data.

The controller is injected with Guice not just with the eventbus, but also with two JavaFX timeline events. These deserve some explanation. The two animation events created, and which get called by the buttons (through the Action Strategy class) are played or stopped by the user. To start with they are merely stopped. The two static animation events are described in AnimationModule. Essentially they are built up of key frames which call eventbus events which first of all updates the model with a new transition taken from the list view. The first keyframe calls a draw event to the vector graphic drawing strategy that is requested to have no colouring of the last selected event. This uses an Enum called Blink which when passed to the controller triggers some logic which removes the reference to the last selected transition in the model as created by the last transition event before requesting a redraw which draws the automaton as if it had no previous transition. A subsequent key frame is called 400 milliseconds later, which invokes a request for a transition event which will call a redraw with the last selected transition highlighted as well as the newly selected node. This has the effect of make the transition obviously blink with every transition, a feature which makes the transitions of the automaton obvious to the user. This is especially the case on transitions to and from the same node where otherwise the user may not have been away of the transition. This thus takes advantage of the flexibility of the JavaFX animation class and the eventbus publish/subscribe method to abstract away a series of complex operations to one time-controlled event which can then be triggered by the user interface. The only materially difference between the fastforward and play animations is that the former is set to be in an indefinite cycle.

Automaton Runner Controller:

Underlying the visible output of the above controls is an automaton runner vector graphics canvas that receives update requests and redraws requests from the model.

The choice of vector graphics may seem strange, but it was motivated by a desire to have buttons which zoom the view in and out, which whilst not part of the original analysis, seemed an easy to add and desirable property of the display screen. The vector canvas defines simple methods to use typical vector graphic scale transform matrices (which are wrapped in even simpler functions) which scale the vectors of currently drawn objects. These are of course called by the relevant buttons. One difficulty I found with this was that (for reasons explained later), the degree of scaling had to be recorded by the controller itself, for future reference.

The main heart of the controller is the handleAutomataChange method. This takes changes in the model, triggered by any source (including a transition and a load) and fully redraws the vector canvas, by way of the graphics context which is a component of the vector canvas JavaFX node and has an internal stack which specifies both the list of vector items and a set of global property changes associated with entries that are associated with objects that are placed later stack and affect their visual properties.

The method encapsulates the broad algorithm by requesting the drawing of the nodes one by one, and then the relations between the nodes individually. It actually enforces this through a separate template class called IDrawingTemplate. The template method is an abstraction of steps of an algorithm (gang of four pp. 325–330), allowing the skeleton of an algorithm to be controlled at the highest level, with the implementation of the steps elsewhere. In this case the abstract algorithm is simply the order in which items get drawn to the object.

The most important template pattern methods are to draw the nodes and the relations. The nodes are relatively straight forward – the object is simply called and some code generates a circle on the frame based on the node’s co-ordinates, and using some of the defined static classes, as below:

gc.fillOval(node.getX(),node.getY(),NODEDIAMETER,NODEDIAMETER);

Some simple modifications to this add extra items onto nodes that are current selected, or are start/final nodes in the automaton.

More complex is the drawing of the relations between the nodes. The template method has to achieve three things: draw a convincing line between two nodes, include an arrowhead, and then add a textbox describing the nature of the transitions, which is in turn passed back to the parent controller that can, if there are more than one transition, expand the list on highlighting. To do so, it first calls an object called the TransitionDrawingObject that wraps the points of the two nodes to be connected as Point2D classes. This is defined as an extension of an Abstract Transition object that generates the necessary co-ordinates for drawing. The underlying controller logic knows if the relation is between two nodes that are different, or a transition between the same node. It calls different constructors on the TransitionDrawingObject dependent on the type of relation – a self-loop utilises the node’s property ‘isDownNode’, and so it is the passing in of this Boolean which defines a differential construction. The abstract object implements two different interfaces for each type of loop, so that the null values of the other kind of co-ordinates are not visible to the rest of the application as long as the programmer codes by way of the interfaces themselves (which is good practice). Once this is decided, the abstract object then constructs the co-ordinates of a Bezier curve to model the connection. The two cases are defined as following

1. A relation between two different nodes is a quadratic Bezier curve, representing the ‘bulge’ between two nodes. First of all the angle between the two nodes is calculated by way of the atan2 function, called on the two points. Then the co-ordinates of a bulge is calculated by way of a properties file which defines both the standard ‘bulge’ size of a node and the ration between this and the distance between nodes, in order to preserve a visually pleasing effect. The co-ordinates of the bulge are halfway between the two points at the edge of the node’s radius, plus an addition path on an orthogonal direction from this whose length is the ‘bulge’. The Bezier curve is attracted to this point. The function describing a Bezier curve is [FUNCTIONOFBEZ] (<http://mathworld.wolfram.com/BezierCurve.html>) As can be seen, the value of t parametrically defines the curve, which starts at the original co-ordinate and moves to the last one whose relative importance on the combined co-ordinate linearly increases. The relative importance of the middle co-ordinate grows quadratically to its maximum effect in the middle where it gets close to the co-ordinate. The code below shows how the x co-ordinate is defined. It should be noted that the use of trigonometric functions enables this curve to be defined against the border of the node and the yslope is just a convenience variable which defines whether the first node is vertically ‘up’ (i.e. with values 1) in relation to the second node or down (value -1).

xBulge = (xAngleOnFirstCircle + xAngleOnSecondCircle)/2 + ySlope\*ARCRADIUS

\*LOOPBULGERATIO\*start.distance(end);

1. A relation is defined between two nodes of the same type as a cubic Bezier curve. This is the same mathematical entity as above, except there are now two middle co-ordinates, which gets approaches cubically, one after the other according to the formula [CUBICBEZCURVE]. The relative importance of the (1-t) and the t factors ensures they are approached at different points in the curve. For this particular example I set the start and the end of the curve to be the same co-ordinate. The loop outwards is constructed by way of a set vertical distance from the node, that is then horizontally nudges one way or the other for each of the node. The code calling this is:

firstX = start.getX() + LOOPRADIUS;

secondX = start.getX() - LOOPRADIUS;

firstY = start.getY() + upCoefficient\*NODEDIAMETER/2 + upCoefficient\*LOOPRADIUS;

secondY = start.getY() + upCoefficient\*NODEDIAMETER/2 + upCoefficient\*LOOPRADIUS;

Once these co-ordinates are entered it draws to the screen in the following steps:

1. The Bezier curve is drawn using the co-ordinates defined above.
2. An arrowhead is created using a method which creates an equilateral triangle at the end of point of the line, which then rotates it to the correct angle. Unfortunately JavaFX does not support adding arrowheads to lines automatically yet, so this has to be done manually. The results are tolerable but not always maximally aesthetically effective.
3. A textbox with the transitions is created. Only the first is displayed if there is more than one. If there is more than one then the transitions are placed in an arraylist of NodeConnections. These are then passed back to the controller and define the bounding boxes of the displayed transition. A mouse controller detects movement over the canvas, if the bounding box is entered by the mouse then the full list of transitions is displayed and the list is highlighted yellow. This was in response to a specific concern of my supervisor who felt that too many transitions could make the screen unreadable. By solving this issue in this way I retain the capacity to display all the transitions in an uncluttered manner, but retain the information inside the canvas without reference to any other part of the screen, such as an external list of transitions would impose, thus aiding usability.

The controller also defines functions and methods for determining the size of the area bounded by the automaton – typically by taking the maximum radius of any node in the x or y axis and adding on a ‘buffer’ term. The canvas is wrapped in a JavaFX Scrollpane which, when faced with a certain sized child object automatically adds a scrollbar. The consequence of this is that when the canvas resizes itself to be larger than the underlying scrollpane a scrollbar is automatically generated. And this includes when the automaton is sized by the zoom buttons.

Each draw event is able to process these details because the AutomataChangeEvent contains the parts of the automaton that are necessary to effect the above algorithm.

Stack Runner Controller:

The stack runner controller is a fairly simple controller. It takes the same change event as above (so gets fired concurrently) and uses the instantaneous description’s current stack to draw a visual representation of it. This is done by simply drawing a square block with text inside, with each square stacked on the other. This is wrapped in a scrollpane too, which is automatically set to scroll to the top every time a change is made, as the user might expect so that they can see the new entries to the stack.

Processed String Controller:

This, as with the last controller, takes a string value. It dynamically generates an HBox with two colours of code and places them in the centre, as this suggests:

HBox coloredTextBox = HBoxBuilder.create()

.minHeight(scrollPane.getHeight() -10)

.minWidth(scrollPane.getWidth() -10)

.alignment(Pos.CENTER)

.spacing(0)

.children(

LabelBuilder.create().text(proc).textFill(Color.RED).font(font)

.build(),

LabelBuilder.create().text(unProc).textFill(Color.BLACK).font(font)

.build()

).build();

The scrollpane’s scrollbar, if active, is constantly being centred on the border between last processed item and the first unprocessed one so the user can always see the actively processed part of the string.

Drawing Controller:

The drawing controller is primarily concerned with the shuttling of events between the view and the incremental build up of the automaton, which is defined in the model class drawing list. To enable these interactions there are several methods used.

The three buttons at the top are, as mentioned in the view section, joined by a group, which has a collective change listener. A consequence of this is that custom code was added in which enabled only one button at a time to be selected. More importantly each listener assigns the interface IDesignStrategy with a custom implementation. The three button’s associated implementations of this class behave in different ways. Essentially every time the mouse button is pressed, dragged or released on the pane, an event handler picks up the responsibility for the appropriate action. It should be noted that the elements on the screen have to re-fire the event to the pane below it, as JavaFX automatically sets the target as the element with the highest z-index.

The three buttons perform different roles. The node button utilises the node strategy, which on a click simply checks the pane for existing node radii, so that there can be no intersections between them. As will be explained later the drawn nodes have ids associated with them that link the elements dynamically placed on the view with the elements below, overlapping nodes would cause ID conflicts and look aesthetically unpleasing. Once this check is performed then the node is added, with its x and y co-ordinates to the model by an eventbus call to a NodeCreationEvent.

The line controller is more complex. Rather than simply being called on the down press of the mouse it is activated between all three events. A class called TransitionMousePressContext records the details of the event in progress. When the mouse is pressed down the static class is activated and the start co-ordinates of the mouse action are entered. When it is closed the end co-ordinates, as well as the start ones are passed to the model, which can workout whether there are nodes at either of these start and end points and can draw a relation between them if necessary. This allows a user to drag a relation between two separate nodes. The real power of it though comes in the temporary drag controller. As seen in the code below:

@Override

public void fireDragEvent(double x, double y, IEditTemplate editTemplate, Pane drawingBoard) {

eventBus.post(new RequestPaneRedrawEvent());

if(TRANSITIONMOUSEPRESSCONTEXT.getIsActive())

{

Line line = LineBuilder

.create()

.startX(TRANSITIONMOUSEPRESSCONTEXT.getStartX())

.startY(TRANSITIONMOUSEPRESSCONTEXT.getStartY())

.endX(x)

.endY(y)

.stroke(Color.LIGHTGREY)

.build();

Double diffX = x - TRANSITIONMOUSEPRESSCONTEXT.getStartX();

Double diffY = y - TRANSITIONMOUSEPRESSCONTEXT.getStartY();

Polygon arrowHead = ArrowHeadBuilder

.create(x, y, diffX, diffY);

arrowHead.setFill(Color.DARKGREY);

drawingBoard.getChildren().addAll(line, arrowHead);

}

}

The drag mouse event handler continually re-evaluates the x and y co-ordinates of the drag and draws a grey line with an arrowhead to mark the progress of the user’s drag. This makes the process of creating nodes highly usable and intuitive, helping match the requirement for a well designed GUI.

The final button is the select button. As expected this simply calculates where in the down press of the mouse within a pane it finds itself either within the bounding box of a line between two nodes (defined by JavaFX), or within the radius of a node. When either of these is selected the graphical element’s ID is taken up and passed into the accordion pane which is populated with that element’s details.

When the model updates itself, it sends a redraw pane event. This simply uses the same co-ordinate details as the IDrawingTemplate before, however the IEditTemplate acts upon them by creating JavaFX Nodes which are dynamically added. The two nodes – the PDA Node and the PDA Relation are actually subclasses of the JavaFX Circle Node and the JavaFX Bezier curve node. They are subclassed so as to retain an ID value, which is in the case of the node a UUID, and in the case of the relation the concatenation of the two UUIDs of the two nodes which are related. This value is also stored in the model, and thus items selected or edited by the view can be automatically updated. Like the animation only the first transition defined for a relation is shown, however there is no pop up displaying this because of course the relation and its transitions can be seen by selecting it.

Two other simple buttons require brief mention. There is a button called canvas size which doubles the width and height of the canvas within the scrollpane that is given. This is so the user can create an arbitrarily large automaton. This information is retained in the model and the XML so that when it is reloaded the canvas is the same size and available to edit as before it was persisted. There is a clear button which sends a request to the model to mark itself as a new automaton (and therefore it will not overwrite an existing one on a save event) it will also reset the canvas size.

The accordion views use default JavaFX accordions but they have been hard coded to remain impossible to edit and the event handler for clicks on this element redirects any user actions to simply reflect the will of the two Booleans in this controller which determine which is open. The controllers for these will be described below:

Node accordion controller:

The node accordion view is primarily to shuttle data to and from the model items to the view elements, by either being set by the parent controller, or by passing new values onto the model by an event bus call and setting the label to the top. The one complex aspect of the controller is that it uses a list view to set transitions to and from the same node in the controller. These have special cells that, like the list view in the animation controller, can delete elements, although the user also has the option of using the delete button. The elements of the list view are also specialised in that when they are doubled clicked a text view is created that then allows the editing of an existing transition, sample code of how this is achieved can be seen in the database controller.

There is an add button which creates a popup view with three textboxes. It was necessary with this pop-up to find online a subclass that has been created to restrict the entries to the character fields, so that only one character is entered. It is a weakness of JavaFX that this hasn’t been entered. The apply button shuttles all this information to the model. The apply button fires the eventbus with these details (which matches the sequence diagram for the drawing events in the analysis).

Transition accordion controller:

The transition controller is a subset of the controls the node controller with the same list view described above. There are however several special classes focused on the issues of validation rules for these relations:

1. The system must no produce relations with no transitions. If the user creates a relation which is null by clicking apply before creating a relation, or a relation is created which is invalid and thus removes all transitions the model removes the newly created relation and informs the draw pane of this event.
2. The system must check for epsilon loops, as were discussed in the proposal. An epsilon loop is where a Push Down Automata has a loop of epsilon transitions, which would mean the automaton would be infinite. To detect this the adapter pattern is used to call upon methods in the JGraphT library which is a standard graph algorithm library. The adapater pattern is defined (headfirstdesignpatterns pg 244) as a wrapper which takes one interface – in this case the Java library – and wraps it in an adapted form for use by the new native library. To do this I was required to construct a class that converted between the model’s structure for the drawing automaton – filtering out only those relations with epsilon transitions - and a standard Vertex-Edge data structure used by GraphT. By doing so it can detect epsilon loops and send a dialog box message informing the user of the failure to add the relation because of this condition
3. It is also a rule of the system, as noted in the analysis phase, that no one node can have more than one transition per character. This rule is enforced in the node and can cause the proposed relation to fail and a dialog box informing the user of this to be produced.

Database Controller:

The database controller largely relies upon the already described ConnectionContext class to perform its actions. The Database controller shuttles the results produced from ConnectionContext’s search of all directories and automata to a tree view where the directories are placed in a top level and the automata below them. JavaFX’s tree view node automatically deals with the expansion and closing of these elements.

Using the aforementioned call-back functionality the tree views also allow the cells which, when double clicked, create a child textbox which then updates the database if the entries is changed. The following code adapted from the code in the Oracle tutotial (<http://docs.oracle.com/javafx/2/ui_controls/tree-view.htm>) overrides the cell’s norm start edit function to create a textbox:

@Override

public void startEdit() {

super.startEdit();

if(getItem().getTreeNodeType() != TreeNodeType.ROOT

&& !getItem().getName().equals("Default"))

{

if (textField == null) {

createTextField();

}

setText(null);

setGraphic(textField);

textField.selectAll();

}

}

Each cell has been subclassed to express an Enum which declared whether it is an automaton cell or a directory cell. An event handler on the tree view detects changes to the list view and then passed them onto the ConnectionContext for updates:

treeList.onEditCommitProperty().setValue(new EventHandler<TreeView.EditEvent<TreeNodeObject>>() {

@Override

public void handle(TreeView.EditEvent<TreeNodeObject> treeNodeObjectEditEvent) {

String name;

Integer automataId;

Integer directoryId;

switch(treeNodeObjectEditEvent.getNewValue().getTreeNodeType())

{

case DIRECTORY:

name = treeNodeObjectEditEvent.getNewValue().getName();

directoryId = treeNodeObjectEditEvent.getNewValue().getId();

eventBus.post(new UpdateDirectoryEvent(name, directoryId));

break;

case AUTOMATA:

name = treeNodeObjectEditEvent.getNewValue().getName();

automataId = treeNodeObjectEditEvent.getNewValue().getId();

eventBus.post(new UpdateAutomataEvent(name, automataId));

break;

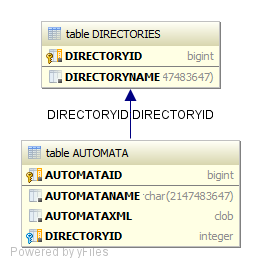
}

}

});

The connection context itself is constructed as a Singleton at run time. It has a static eventbus server reference, so it can pass information to other classes and receive it, and it has a connection passed in a start up to the H2 database. It utilises SQL to perform a set of standard actions on the database and can pass these results back as lists or values to the database controller. The connection itself, and the defined SQL are all held in a database properties file, so they are easily reusable and can be edited without tampering with live code. The H2 database itself resides as a driver that is called from a jar and is loaded as a connection. As an embedded database the actual contents of the database – including its schema – are defined in files that reside in the project file structure (or outside, say a jar). This means a default starting database can be packaged with the software.

To support the tree view above it was necessary to have a database with a simple data model. The directories are stored in one table with their own auto-incremented ids, and the automata are stored in a separate table with their respective auto-generated ids. The automata have a foreign key reference to the directory table. See the generated H2 data model below:



This fulfils but the requirement for an easy to use system for the user, in terms of being able to keep track of automata (and name them appropriately), and to allow for a coherent persistence policy to be implemented in the application. Not that the actual XML that is streamed and stored in saved in the data model at this point.

The other parts of the database controller simply call functions on the ConnectionContext object to manipulate the tree view – delete automaton, delete directories, add directories, move directory (which calls a pop up) and load the automaton into the other views. The latter is done so by then passing on the retrieved XML to be deserialised, as described elsewhere. The buttons are disabled or enabled depending on the kind of cell selected, again using the inherent Enum defined for each cell and an event handler.

To move automata to and from a directory the above methods were employed. However it was also necessary to employ a new trio of event handlers called setOnDragDetected, setOnDragOver and setOnDragDropped within the text cell. The cumulative result of defining these was to have a automaton tree view cell that could be dragged into a directory cell thus changing the directory of the automaton. This was made possible in these handlers by a DragBoard class, which is part of the JavaFX library, which holds state information about the dragged object as it is being transferred.

One rule that was implemented, to ensure consistency in the application was that there was always at any one time a directory called Default. The directory called default cannot be edited nor can it be deleted, thus ensuring new automaton always have a directory in which to be saved.

Model:

Animation Model:

Drawing Model:

XStreamer: