ECSE 539 - Advanced Software Language Engineering
Project Report - Bixi Language

November 28th 2018

P539-2

Frédéric Ladouceur - 260690491
Alec Parent - 260688035
Mushfique Rahman - 260576385

Table of Contents

Project Proposal

The selected domain will encapsulate Montréal's "BIXI Bike Rental System". It will include abstractions for the service's bicycles, bicycle racks, employees (those who maintain the bicycles and racks, and those who manage the systems) and employee responsibilities. The primary purpose would be to model the domain to determine how many bikes are needed and how often the bikes would have to be moved from one station to another in order to meet the customers need. We would be able to change the usage rate of the system, the number of bikes used and the frequency of re-stocking. Also, we would be able to change the percentage of the bikes that would need repair (due to usage) after a given period of time and see how that would affect the overall system.

This domain-specific language will be created using the JetBrains MPS transformation environment. Specifically, the abstract syntax will be generated using JetBrains' integrated "structure" modules, wherein a concept's alias, properties, attributes, et cetera can be specified as per requirements and specifications. The concrete syntax will be created using other modules integrated in JetBrains, such as the behavior generator in which a constructor can be specified as well as object methods, and the constraints generator where limitations can be placed upon a concept's child and parent nodes, thus unifying construction.

We will use the built-in generator to gradually transform the Bixi domain-specific model into a model represented in Java. The resulting model will then be further transformed into textual source files, which will be fed into traditional compilers to generate runnable binaries.

Note: This is only the initial project proposal. As the project developed, many of these aspects were altered. You may read these later on in the report. Specifically, the bicycle percentages, the way concrete syntax was specified, and even our implementation of semantics as a whole, are aspects that were completely overhauled.

Workload Specifications

Specifications on what each of these modules represent can be found below in the "Language Engineering and Transformation Environment" section of the report.

Frédéric:
- Constraint Module;
- Typesystem Module;
- Backend behaviour Module;

Alec:
- Generator Module;
- Frontend behaviour Module;

Mushfique:
- Editor Module;
- Action Module;

All:
- Structure Module;
- Report;
- Tutorial;
- Presentation;

Language Engineering and Transformation Environment

As specified in the project proposal, the chosen language engineering environment is JetBrains MPS. The following is a short tutorial on how to specify a language in this environment, as well as an experience report relating the progress of our project.

For the PowerPoint tutorial on how to specify the following concepts in MPS, please see the included file *ECSE539.pdf,* as this written version of our MPS tutorial functions best in conjunction with those slides. Below, you will find a textual explanation of how these concepts work in MPS.

Intro to JetBrains MPS

JetBrains MPS is an open-source language workbench focusing on domain-specific languages (DSLs). It is currently being utilized in domains spanning from electrical engineering and data mining to government organizations and insurance industries.

Using MPS, you may choose to either build your own language from scratch using the tools well-organized workbench and layout, or extend existing languages in order to better fulfill your DSL's domain or subdomain. As you build your language, you will benefit from many integrated tools into the workbench, such as auto-completion, typesystem checking, static code analysis, debugging, and many more. These intelligent systems then allow future developers to begin working with your language quickly, and reduce errors in not only their code, but yours as well. The internal code generator will transform your code into a target platform of your choice including, but not limited to, C, Java, and JavaScript.

MPS displays a modern-day mastery of projectional editing. Unlike the text editors we've all come to be accustomed to, it avoids text parsing. This allows the use of unconventional coding mechanics, such as a decision table directly in the editor to better visualize control flow. Form-like notations in the use of your DSL provide guidance to amateur programmers, and the use of diagrams combined with textual code (either in a diagram's node or to accompany them), just may be the most intuitive option for your particular domain. By switching among these various visualizations and notations, a developer can easily choose the best approach for the task without being burdened by the language engineer's intentions.

MPS also provides the possibility of using multiple languages in your programs, allowing for the combination and reuse of languages, which just may turn out to be convenient once your languages become larger.

How to Specify Abstract Syntax in MPS

As specified above, MPS is not a text editor, but a projectional editor. The exact structure it is projecting is called the *Abstract Syntax Tree*, and every language has its own tree that is edited as alterations are brought within the editor. Additionally, every tree's projection is organized into a collection of models, all of which serve to alter different aspects of the tree. Below is an overview of each of them, by the end of this section of the report, a reader should be familiar with the inner workings of MPS from a language designer's perspective.

Structure models define each concept used within the language. That is, each concept defined under the "Structure" header defines a type of node present in the abstract syntax tree. Each concept specifies within its structure model three aspects of its existence within the system:

- The properties of the concept, parallel to a class's attributes in Java, specify potentially volatile aspects of the concept. Furthermore, property types can only be primitive or enumerations;
- The children of the concept, that is, a list (or some number of lists) of concepts which compose this concept;
- The concepts references, or some number of other concepts that this concept may view as a dependency or association. Reference types can only be some other structure defined in this language.

Additionally, a structure can specify if a concept extends some super-concept or implement some *ConceptInterface*. The ability to name a concept, for example, is implemented via the *INamedConcept* interface. It is also within the structure models that properties such as the concepts ability to be a root node, an alias, and a short description of the concept are specified.

It should also be noted that it is under the structure models that enumeration types can be specified. Every entry of an enumeration type will have both an internal value (how the interpreter interprets the enum value) and a presentation value (how the programmer specifies the enum value).

With each defined structural concept, the option to create a constraint model tied to the structure is present. Constraint models are used in order to restrict the relationships between nodes in ways that aren't necessarily possible within the structure nodes, as well as potentially limiting the values of certain properties. Typically, a constraint model will define limitations such as a collection of allowed nodes that a reference can point to, a property's "getters" and "setters", and a situation definition where a node is allowed to be extend or be extended by some other node.

Just like constraints, each node may define a behaviour model. The behaviour model defines any static or non-static method that can be polymorphically invoked the node. It is by the behaviour models that a node may carry run-time behaviour as well as its properties and relationships.

How to Specify Concrete Syntax in MPS

Every concept that is desired to be used by the programmer should have an editor model created for it. Editor nodes define a projectional editor that the programmer may use to alter the abstract syntax tree directly. The editor model will allow the language designer to create either a UI for the programmer (in the form of a table, for example), or specify some context-free grammar which must be adhered to in order for the concept to be utilized.

Intentions are utilized in order to bring the MPS editor that much closer in terms of quality to mainstream IDEs. If a node has an editor model created for it, it is recommended to also create an intention model. These serve to give instant code manipulation (using the alt + enter key combination) to help the programmer, as well as providing context-sensitive hints for the programmer.

In a textual editor, some actions can be rather trivial (we've all grown accustomed to the convenience of copy/pasting, for example). Since MPS is a projectional editor and directly manipulates an underlying data structure, this option becomes more complex. A concept's action model can be created in order to specify how such editing actions are to be handled by the editor. In other words, how such actions will modify the abstract syntax tree.

How to Specify Semantics

In MPS, semantics are defined operationally through what are known as *Structures* (not to be confused with a language's structures). We say that it is operational due to having to define all reactions of the machine in response to the execution of the models defined in these structures. Structures serve to store any number of potentially cooperating programs, all of which specified by some imported language(s), which are to be specified in the structure's module properties, as well as its dependencies to other structures (a functionality parallel to package importing in Java). It is in these structures that programs using your DSLs are created, using the syntax defined in the editor aspect of one of the language's root nodes. In the generation of a project, the option is given to create a default "sandbox" structure, which serves to test the code periodically as it develops, and does not create its own package like any other structure would.

Since the definitions and interrelations of the concrete and abstract syntaxes are so thoroughly laid out in MPS, the interpreter is able to aid the programmer in more ways than simply what is defined in the Intentions defined above. The *dataflow* modules inside the language designer can design the flow using these modules, and thus let the interpreter warn the programmer when issues such as unreachable code or null-pointer exceptions occur. While such modules are not utilized in the Bixi Language, they become rather useful in languages that contain conditional flow or loops.

In a further attempt to allow the language designer to fully aid the programmer, *typesystem* modules can be defined to provide type-checking. This serves to detect errors early on in the programming portion of any user project.

Certainly not among the least important aspects of MPS, the generator defines not only the main method in the execution of any program, but also provides the rules for translating concepts and setting their proper ordering. Additionally, it serves to bridge the gap between different languages if a certain program is written using more than one, as we must be able to have these numerous languages cooperate. The generator achieves this cooperation by abstracting the defined concepts until some lowest common denominator is reached. The generator can, for example, create reduction rules in order to define how a certain concept should act in run-time once it is read by the interpreter (such as how to draw it on a JFrame).

Finally, while this aspect of MPS was not utilized in this project, MPS does have a built-in Model-to-Text translator, as implemented in the TextGen modules. Once the Generator (defined above) has fully abstracted the abstract syntax tree, the defined TextGen modules will translate the model into its textual representation and save it to disk, wherever on disk the programmer specifies.

Experience Report

Getting to grips with MPS was a bit of a steep hill at first for all three of us, as none of us were familiar with projectional editors. Having the editor auto-complete some aspects as we were writing on auto-pilot was a new approach we had to keep an eye out for. For example, in the Java block statements written for behaviour modules, simply writing out:

```
public void get()
```

Which is a method signature, and a text sequence, all programmers write out instinctively in classic text editors, could not simply be written sequentially as seen above. Once public is written out, the projectional editor recognizes what you are trying to accomplish, and automatically searches to retrieve the method name before the method's return type, so we were often surprised to see in its place:

```
public get void
```

By the time the error was recognized, any number of changes could have occured on the abstract syntax tree, or even no progress at all. Of course, such an example may be a little silly or even inconsequential when considering the scope of the average MPS project, but the idea remains: writing out our code while perpetually considering the data structure underlying the editor was a task which took a little bit of adapting to get used to.

Additionally, the large and precise structures of MPS took some understanding and studying before a project could truly begin, as seen in tutorial above. However, the creators of MPS have provided a very thorough 10-step program on their website, which takes aspiring language designers such as ourselves and brings them from being entirely oblivious about MPS all the way to creating custom UI languages (in a mere 40 hours). Given this extreme learning curve, learning as a group and pigeonholing our responsibilities and specialties became an unavoidable requirement. However, once the approach to projectional editing was well understood, it became just like any text editor in some ways: just write your code so it works. Whether or not we would return to projectional editing in the future depends on who you ask in the team.

In our use of Git, we were surprised to see that MPS does not appreciate git branch merges, and we occasionally ran into issues where the compiler would simply give up its operations and report an error. In order to counter this error, we were forced to create a new project and manually introduce everyone's accomplished individual work. While tedious, it's the only functional solution to this problem we could muster.

Furthermore, we were also troubled by the glitches in MPS as we opened multiple projects at the same time on separate windows. It mixes up the references between the opened projects. It also does not support the copy paste function between the projects which we learned the hard way, as we ended up with numerous compile errors.

When it came time to design the language, and how we would implement the project, our biggest issue was seeing too large, especially since we were all new to the world of language design. What exactly is a small language? Is this implementation too ambitious? How long should we expect this to take? Coming to terms with a final project and deciding how we would implement said project (now manifesting in the form of the metamodel, as seen below) was a monumental task in and of itself. Once that was decided, all that was left was doing some research within MPS to discover *how* exactly such a project could be implemented in the context of the language engineering environment.

Finally, and this sort of ties in with the previous point, once specialties were dealt and deadlines were set for specific tasks, it was difficult to decide whether those deadlines were reasonable, or even if far too much time was being allocated to a certain task. However, all that was required to surmount that was a bit of communication and hard work being done in advance.

Language Definition

The chosen modeling domain is Montréal's BIXI service. Presented below is a specification of the domain specific language's encapsulated domain, as well as a description on how to use the language editor and specify custom BIXI simulations.

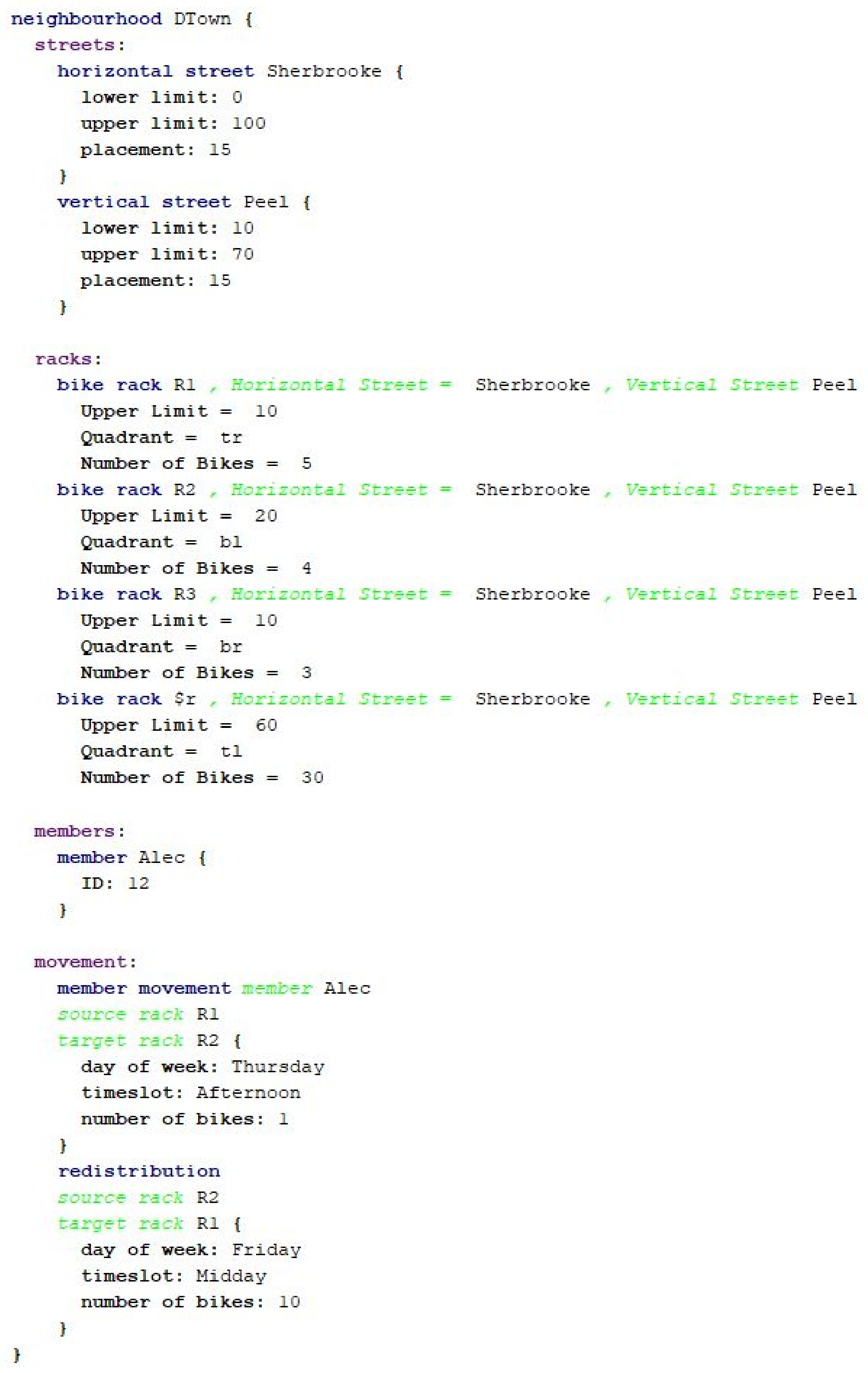
Description of Domain-Specific Language

BIXI Montréal is a bicycle sharing system whose name is a portmanteau of the words "Bicycle" and "Taxi", thus evidencing the nature of the service as citizens using bicycles as taxis. The service is showcased mostly by the bike racks sprinkled throughout the city. Each bike rack is composed of a collection of bike docks, allowing users to take or return bicycles and a pay station, where members swipe their member cards, thus allowing them to remove a bicycle from a dock, and all of these are fitted onto modular platforms powered by solar panels. A bike rack can be set up in about half an hour, and removed in about an equal amount of time without need for any preparation or excavation, making the installation of a BIXI rack a must-have addition to any public parking lot.

The developed language, named Bixi, aims to encompass this system and provide a simulation of its daily and weekly functions and usage, evaluating bicycle redistribution strategies given a neighbourhood's model. The language allows for the easy generation of any (grid-based) neighbourhood along with a bicycle rack repopulation strategy for that neighbourhood's bike racks. The UI to be generated by the node structures shows a simulation of the neighbourhood, and allows the user to evaluate if their repopulation strategy is effective given certain randomness. The randomness is provided by the implemented ability for non-member citizens to partake in the service. That is, members provide some concrete usage patterns for each bike rack, while non-members are not expected to partake in the activity on a regular basis, and are thus interpreted as randomly taking and moving bikes, thus potentially concretizing the potential of a redistribution model.

How to Use the Language Editor

Every project in JetBrains MPS is created with two primary modules: a structure and a language (in this case, the structure is not to be confused with the structure modules found in the language definition, the structures do not define the abstract syntax tree, but rather create programs using the language). In this project, the module properties of the Sandbox structure have already been configured to recognize Bixi as a used language. All that needs be done is create a Neighbourhood node in the Sandbox, and specify your programs using Bixi's concrete syntax. An example program utilizing this syntax can be seen in Figures 1 and 3 below, with their respective outputs in figures 2 and 4..



```
neighbourhood DTown {
  streets:
    horizontal street Sherbrooke {
      lower limit: 0
      upper limit: 100
      placement: 15
    }
    vertical street Peel {
      lower limit: 10
      upper limit: 70
      placement: 15
    }

  racks:
    bike rack R1 , Horizontal Street =  Sherbrooke , Vertical Street Peel
      Upper Limit =  10
      Quadrant =  tr
      Number of Bikes =  5
    bike rack R2 , Horizontal Street =  Sherbrooke , Vertical Street Peel
      Upper Limit =  20
      Quadrant =  bl
      Number of Bikes =  4
    bike rack R3 , Horizontal Street =  Sherbrooke , Vertical Street Peel
      Upper Limit =  10
      Quadrant =  br
      Number of Bikes =  3
    bike rack $r , Horizontal Street =  Sherbrooke , Vertical Street Peel
      Upper Limit =  60
      Quadrant =  tl
      Number of Bikes =  30

  members:
    member Alec {
      ID: 12
    }

  movement:
    member movement member Alec
    source rack R1
    target rack R2 {
      day of week: Thursday
      timeslot: Afternoon
      number of bikes: 1
    }
    redistribution
    source rack R2
    target rack R1 {
      day of week: Friday
      timeslot: Midday
      number of bikes: 10
    }
}
```

Figure 1: An example program, DTown. This program is designed to be a faulty setup, the results of this are shown in the next figure.

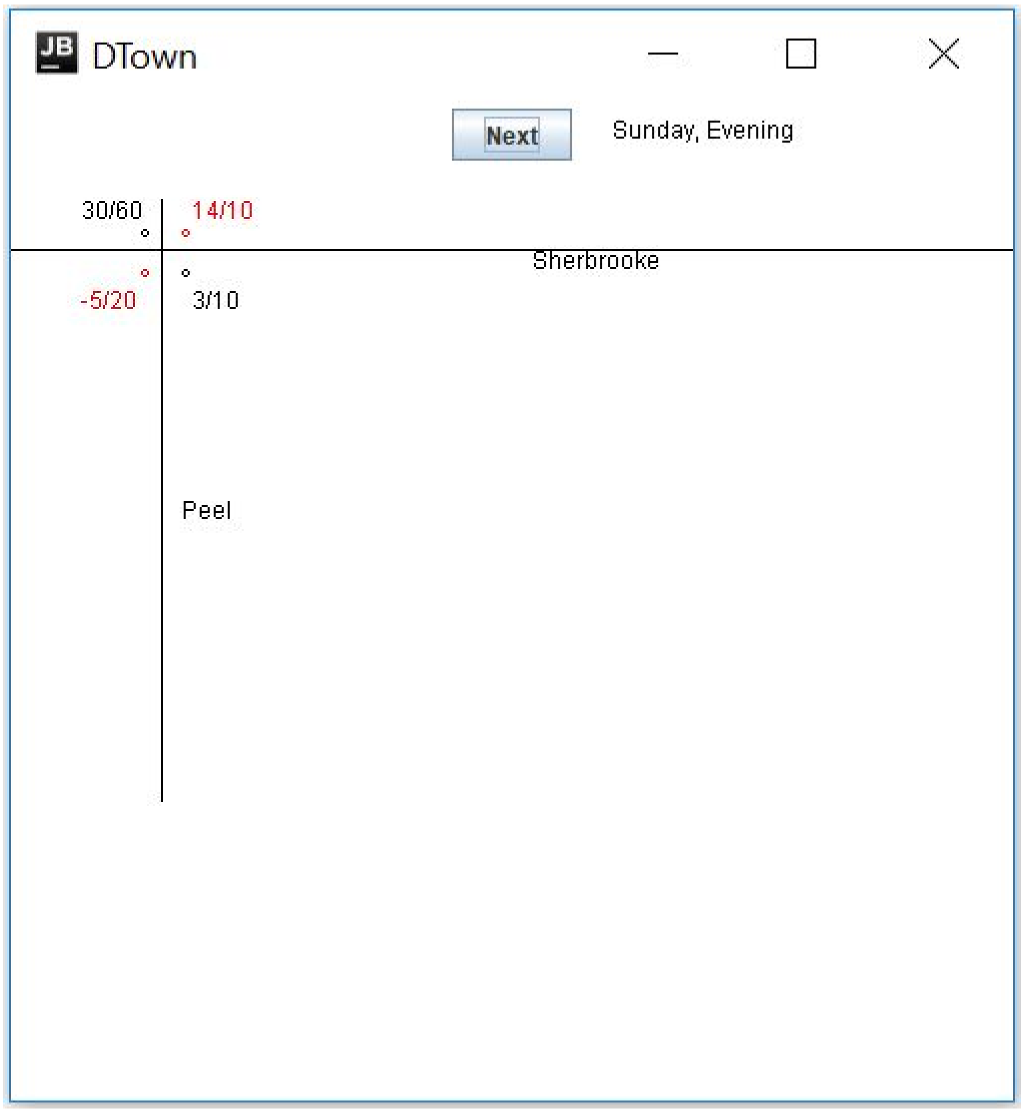
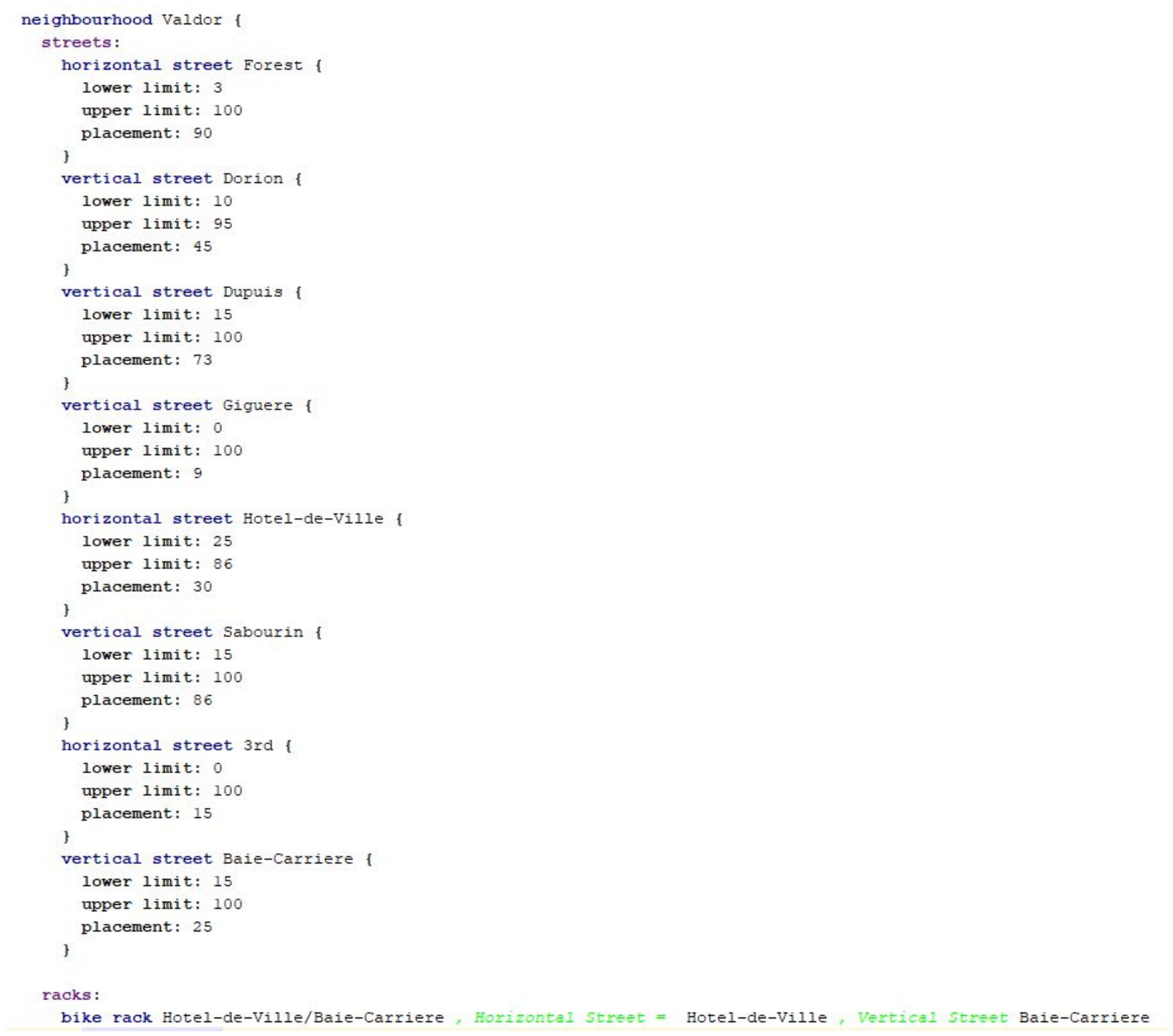


Figure 2: DTown's execution result, after 2 "Next" button clicks. Notice how the model was designed to be faulty, thus informing the user that the design of their map needs to be re-worked.



```
neighbourhood Valdor {
  streets:
    horizontal street Forest {
      lower limit: 3
      upper limit: 100
      placement: 90
    }
    vertical street Dorion {
      lower limit: 10
      upper limit: 95
      placement: 45
    }
    vertical street Dupuis {
      lower limit: 15
      upper limit: 100
      placement: 73
    }
    vertical street Giguere {
      lower limit: 0
      upper limit: 100
      placement: 9
    }
    horizontal street Hotel-de-Ville {
      lower limit: 25
      upper limit: 86
      placement: 30
    }
    vertical street Sabourin {
      lower limit: 15
      upper limit: 100
      placement: 86
    }
    horizontal street 3rd {
      lower limit: 0
      upper limit: 100
      placement: 15
    }
    vertical street Baie-Carriere {
      lower limit: 15
      upper limit: 100
      placement: 25
    }

  racks:
    bike rack Hotel-de-Ville/Baie-Carriere , Horizontal Street =  Hotel-de-Ville , Vertical Street Baie-Carriere
```

Figure 3: An example program, Valdor (see Appendix A for complete model)

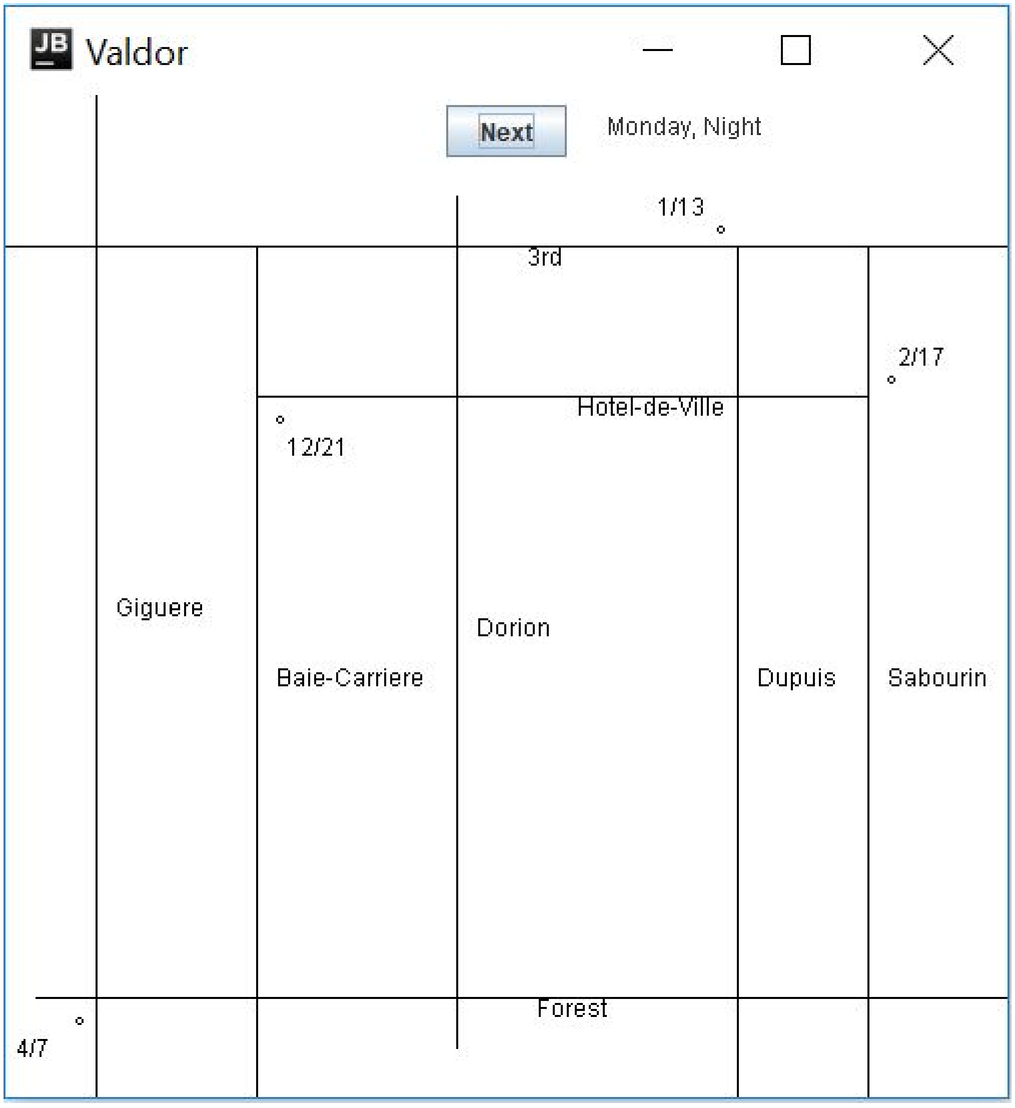


Figure 4: Valdor's execution result

List of Manually Changed Source Files

While MPS does have some source files that are readily accessible to the language designer, they are all saved in read-only mode and, as far as we can tell, not to be messed with. These files serves to specify the concrete and abstract syntax that MPS uses to help language designers specify their language (one level higher in the metamodel layering, if you will). Therefore, no source files were altered.

Metamodel & Metamodel Design Decisions

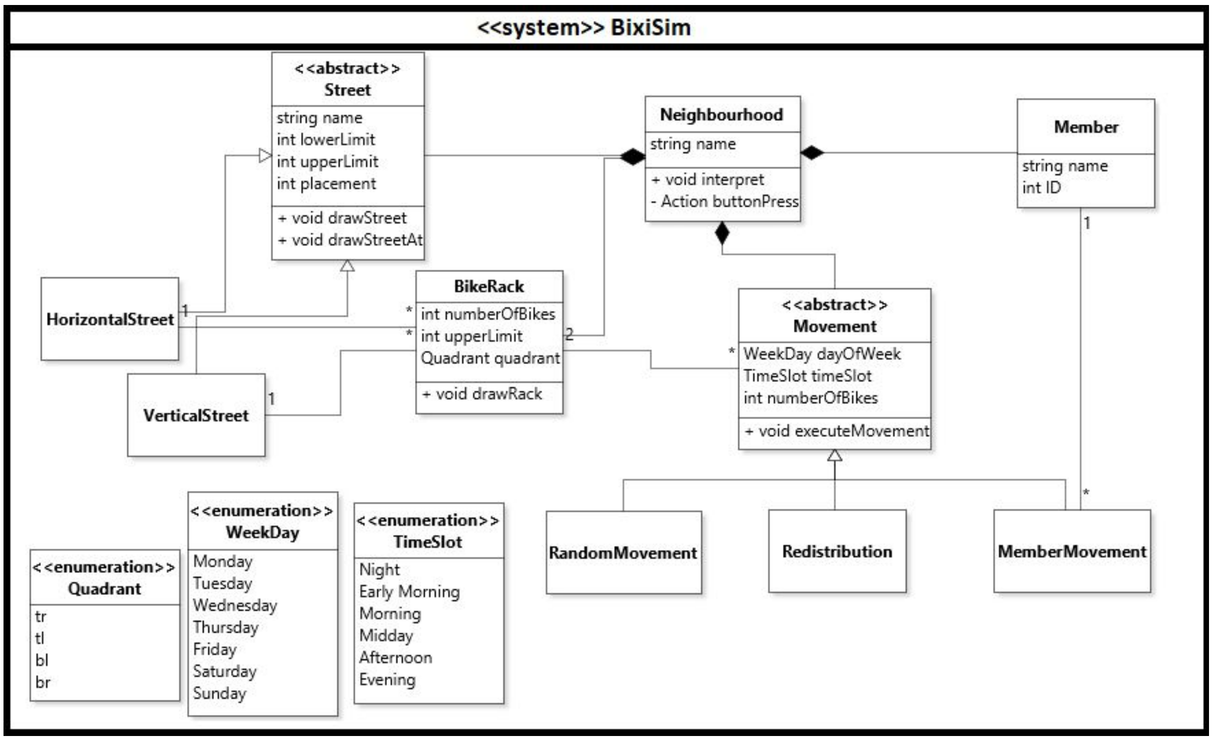


Figure 5: The BixiSim metamodel

The above metamodel outlines the design of the language, and how the final structure was decided. Here are some of the design decisions that were made in its creation:

- Street was made abstract and is extended by both HorizontalStreet and VerticalStreet. This decision was made over making Street an instantiable class with the additional attribute of StreetDirection, which takes on the either the value "Horizontal" or "Vertical", both of which specified by an enum. This abstraction of street was made in order to remove a constraint. With this model, we must store one street of each direction, whereas if Streets had been stored in a length 2 Street array, a constraint would have had to be imposed on said array to enforce the orthogonal nature of the Streets, and even potentially leading to runtime errors. This design circumvents the issue.
- The TimeSlot enumeration has exactly 6 potential values (internally, these values represent 4 hour long intervals, with "Night" starting at midnight, the internal ascending order is the same as what is shown in the metamodel above). This choice of 6 4-hour intervals was made in order to strike a balance between efficiency and accuracy. More intervals would reduce efficiency, while less intervals would create timespans far too large for a single TimeSlot, which would be difficult to justify in simulation.
- Movement was rendered abstract and separated into 3 subclasses, each with their own function:
    - MemberMovement is associated with a Member, and is used to model the consistent schedule of a member, is will thus always occur. MemberMovements always have their numberOfBikes attribute equal to 1;
    - RandomMovement is used to model the stochastic travels of people without memberships (and thus, RandomMovements are not always expected to occur as specified), this randomness was incorporated in order to potentially solidify the model specified by the programmer;
    - Redistribution represents the mass movements of bicycles from one rack to another by employees, essentially serving as a repopulation mechanism.
- Neighbourhood serves as the root, and is what is originally instantiated. This is done in order to ensure that the map is always created before Movements are specified.

Semantics

Implementation of Semantics

As explained above, semantics in JetBrains MPS defined operationally through several different modules of the language. The most important module to this end is the generator. Upon the creation of an MPS project, the generator module is created with a default main folder, in which the language designer will outline how each concept of the language gets mapped or reduced when a program is run.

The most important of the generators functions for this project are the root mapping and reduction rules. The root mapping rule was used in order to call the defined *map_Neighbourhood* class. Therefore, when a node using *Neighbourhood* as its root (which is the only option in this project) is run, the generator automatically detects the nature of the run program and creates the java code to properly run a *Neighbourhood* node. The rule reductions are used within this root map. Every so often, the *$COPY_SCR$* macro can be seen in the mapping rule. This calls the reduction rule which applies best to the affected concept (for example, if the macro is used on a bike in quadrant 1 of an intersection, then the rule *reduce_BikeRack_1* is invoked), which automatically generates more code when called. In our language, these are used exclusively to draw objects to the JPanel.

Once the generator is defined and functional, the behaviour module must be completed. For each of the created structure nodes, an associated behaviour node was created, containing java code which specifies all behavioural actions required of each node. Upon construction of each node, these behavioural methods can be forced to execute, thus implementing their operational semantics.

Overall, the semantics are dependent on the execution of Java Swing, thus the Swing framework had to be included in the language dependencies. Once the proper generator was defined and accompanying behaviours specified, the application is expected to run as defined.

How to Analyze & Execute the Bixi Language

Using the concrete and abstract syntax as specified above, and in further detail within the source code files, a user may create a program in the structure module (specifically, the one containing the sandbox model, not the one containing the abstract syntax tree node specifications). The sandbox model is provided to give a place for a programmer to create their own program using the Bixi language. Right-clicking on sandbox and selecting the option *New->Neighbourhood* will generate a template containing *Neighbourhood*'s concrete syntax, where the user can now generate code to specify the other aspects of the neighbourhood under their respective sections (Hint: hitting *ctrl* + *space* while inside one of those sections will auto-generate concrete syntax of that concept, making the coding process much easier). Once the code is complete, right-clicking on the created program in the package explorer on the right, and selecting *Run* will execute the Bixi program.

Manually Changed Source Files

While MPS does have some source files that are readily accessible to the language designer, they are all saved in read-only mode and, as far as we can tell, not to be messed with. These files serves to specify the concrete and abstract syntax that MPS uses to help language designers specify their language (one level higher in the metamodel layering, if you will). Therefore, no source files were altered.

Appendix

Appendix A - *Valdor* complete code

```
neighbourhood Valdor {
  streets:
    horizontal street Forest {
      lower limit: 3
      upper limit: 100
      placement: 90
    }
    vertical street Dorion {
      lower limit: 10
      upper limit: 95
      placement: 45
    }
    vertical street Dupuis {
      lower limit: 15
      upper limit: 100
      placement: 73
    }
    vertical street Giguere {
      lower limit: 0
      upper limit: 100
      placement: 9
    }
    horizontal street Hotel-de-Ville {
      lower limit: 25
      upper limit: 86
      placement: 30
    }
    vertical street Sabourin {
      lower limit: 15
      upper limit: 100
      placement: 86
    }
    horizontal street 3rd {
      lower limit: 0
      upper limit: 100
      placement: 15
    }
    vertical street Baie-Carriere {
      lower limit: 15
      upper limit: 100
      placement: 25
    }

  racks:
    bike rack Hotel-de-Ville/Baie-Carriere , Horizontal Street =  Hotel-de-Ville , Vertical Street Baie-Carriere
      Upper Limit =  21
      Quadrant =  br
      Number of Bikes =  12
    bike rack Hotel-de-Ville/Sabourin , Horizontal Street =  Hotel-de-Ville , Vertical Street Sabourin
      Upper Limit =  17
      Quadrant =  tr
      Number of Bikes =  2
    bike rack 3rd/Dupuis , Horizontal Street =  3rd , Vertical Street Dupuis
      Upper Limit =  13
      Quadrant =  tl
      Number of Bikes =  1
    bike rack Forest/Giguere , Horizontal Street =  Forest , Vertical Street Giguere
      Upper Limit =  7
      Quadrant =  bl
      Number of Bikes =  4

  members:
    member Alec Parent {
      ID: 12
    }
    member Frederic Ladouceur {
      ID: 107
    }
    member Mushfique Rahman {
      ID: 55
    }
    member Gunter Mussbacher {
      ID: 90
    }
    member Chewbacca {
      ID: 11
    }
    member John Snow {
      ID: 45
    }
    member Doctor Who {
      ID: 88
    }
    member Ghandi {
      ID: 3
    }

  movement:
    member movement member Alec Parent
    source rack 3rd/Dupuis
    target rack Forest/Giguere {
      day of week: Monday
      timeslot: Midday
      number of bikes: 1
    }
    member movement member Alec Parent
```

```
source rack Forest/Giguere
target rack Forest/Giguere {
  day of week: Tuesday
  timeslot: Afternoon
  number of bikes: 1
}
member movement member Alec Parent
source rack Forest/Giguere
target rack 3rd/Dupuis {
  day of week: Wednesday
  timeslot: Morning
  number of bikes: 1
}
member movement member Chewbacca
source rack Hotel-de-Ville/Baie-Carriere
target rack Hotel-de-Ville/Baie-Carriere {
  day of week: Monday
  timeslot: Night
  number of bikes: 1
}
member movement member Chewbacca
source rack 3rd/Dupuis
target rack Hotel-de-Ville/Sabourin {
  day of week: Tuesday
  timeslot: Midday
  number of bikes: 1
}
member movement member Chewbacca
source rack Hotel-de-Ville/Sabourin
target rack Hotel-de-Ville/Baie-Carriere {
  day of week: Wednesday
  timeslot: Evening
  number of bikes: 1
}
member movement member Doctor Who
source rack Hotel-de-Ville/Sabourin
target rack Hotel-de-Ville/Baie-Carriere {
  day of week: Monday
  timeslot: Morning
  number of bikes: 1
}
member movement member Doctor Who
source rack Forest/Giguere
target rack 3rd/Dupuis {
  day of week: Tuesday
  timeslot: Midday
  number of bikes: 1
}
member movement member Doctor Who
source rack 3rd/Dupuis
target rack Forest/Giguere {
  day of week: Wednesday
  timeslot: Midday
  number of bikes: 1
}
member movement member Frederic Ladouceur
source rack Hotel-de-Ville/Baie-Carriere
target rack Hotel-de-Ville/Sabourin {
  day of week: Monday
  timeslot: Evening
  number of bikes: 1
}
member movement member Frederic Ladouceur
source rack Hotel-de-Ville/Baie-Carriere
target rack Forest/Giguere {
  day of week: Tuesday
  timeslot: Early Morning
  number of bikes: 1
}
member movement member Frederic Ladouceur
source rack Hotel-de-Ville/Baie-Carriere
target rack 3rd/Dupuis {
  day of week: Wednesday
  timeslot: Night
  number of bikes: 1
}
member movement member Ghandi
source rack Hotel-de-Ville/Sabourin
target rack Forest/Giguere {
  day of week: Monday
  timeslot: Afternoon
  number of bikes: 1
}
member movement member Ghandi
source rack Hotel-de-Ville/Baie-Carriere
target rack Hotel-de-Ville/Sabourin {
  day of week: Tuesday
  timeslot: Night
  number of bikes: 1
}
member movement member Ghandi
source rack 3rd/Dupuis
target rack Forest/Giguere {
  day of week: Wednesday
  timeslot: Evening
  number of bikes: 1
}
member movement member Gunter Mussbacher
source rack Hotel-de-Ville/Baie-Carriere
target rack Hotel-de-Ville/Sabourin {
  day of week: Monday
  timeslot: Early Morning
  number of bikes: 1
}
member movement member Gunter Mussbacher
source rack Hotel-de-Ville/Baie-Carriere
target rack Hotel-de-Ville/Baie-Carriere {
  day of week: Tuesday
  timeslot: Early Morning
  number of bikes: 1
}
member movement member Gunter Mussbacher
source rack Hotel-de-Ville/Baie-Carriere
target rack 3rd/Dupuis {
  day of week: Wednesday
  timeslot: Midday
  number of bikes: 1
}
member movement member John Snow
source rack Hotel-de-Ville/Baie-Carriere
target rack 3rd/Dupuis {
  day of week: Monday
  timeslot: Early Morning
  number of bikes: 1
}
member movement member John Snow
source rack 3rd/Dupuis
target rack Hotel-de-Ville/Baie-Carriere {
  day of week: Tuesday
  timeslot: Evening
  number of bikes: 1
}
member movement member John Snow
source rack Hotel-de-Ville/Baie-Carriere
target rack 3rd/Dupuis {
  day of week: Wednesday
  timeslot: Midday
  number of bikes: 1
}
member movement member Mushfique Rahman
source rack Hotel-de-Ville/Sabourin
target rack Hotel-de-Ville/Baie-Carriere {
  day of week: Monday
  timeslot: Early Morning
  number of bikes: 1
}
member movement member Mushfique Rahman
source rack Hotel-de-Ville/Sabourin
target rack Hotel-de-Ville/Baie-Carriere {
  day of week: Tuesday
  timeslot: Midday
  number of bikes: 1
}
```

```
member movement member Mushfique Rahman
source rack 3rd/Dupuis
target rack Forest/Giguere {
  day of week: Wednesday
  timeslot: Evening
  number of bikes: 1
}
random movement
source rack Forest/Giguere
target rack Hotel-de-Ville/Sabourin {
  day of week: Monday
  timeslot: Night
  number of bikes: 5
}
random movement
source rack Forest/Giguere
target rack 3rd/Dupuis {
  day of week: Monday
  timeslot: Night
  number of bikes: 2
}
random movement
source rack Forest/Giguere
target rack Hotel-de-Ville/Sabourin {
  day of week: Monday
  timeslot: Evening
  number of bikes: 3
}
random movement
source rack Hotel-de-Ville/Baie-Carriere
target rack Forest/Giguere {
  day of week: Monday
  timeslot: Midday
  number of bikes: 2
}
random movement
source rack Forest/Giguere
target rack Hotel-de-Ville/Baie-Carriere {
  day of week: Tuesday
  timeslot: Afternoon
  number of bikes: 2
}
random movement
source rack 3rd/Dupuis
target rack Hotel-de-Ville/Baie-Carriere {
  day of week: Tuesday
  timeslot: Night
  number of bikes: 6
}
random movement
source rack Hotel-de-Ville/Sabourin
```

```
  target rack Hotel-de-Ville/Baie-Carriere {
    day of week: Tuesday
    timeslot: Afternoon
    number of bikes: 5
  }
  random movement
  source rack Hotel-de-Ville/Baie-Carriere
  target rack Forest/Giguere {
    day of week: Wednesday
    timeslot: Afternoon
    number of bikes: 4
  }
  random movement
  source rack 3rd/Dupuis
  target rack Hotel-de-Ville/Baie-Carriere {
    day of week: Wednesday
    timeslot: Evening
    number of bikes: 2
  }
  random movement
  source rack 3rd/Dupuis
  target rack Hotel-de-Ville/Baie-Carriere {
    day of week: Wednesday
    timeslot: Evening
    number of bikes: 3
  }
  redistribution
  source rack Hotel-de-Ville/Sabourin
  target rack 3rd/Dupuis {
    day of week: Monday
    timeslot: Evening
    number of bikes: 4
  }
  redistribution
  source rack 3rd/Dupuis
  target rack Forest/Giguere {
    day of week: Monday
    timeslot: Evening
    number of bikes: 5
  }
  redistribution
  source rack Forest/Giguere
  target rack Hotel-de-Ville/Baie-Carriere {
    day of week: Monday
    timeslot: Early Morning
    number of bikes: 3
  }
  redistribution
  source rack Hotel-de-Ville/Baie-Carriere
  target rack 3rd/Dupuis {
    day of week: Tuesday
    timeslot: Morning
    number of bikes: 4
  }
  redistribution
  source rack Hotel-de-Ville/Baie-Carriere
  target rack 3rd/Dupuis {
    day of week: Tuesday
    timeslot: Afternoon
    number of bikes: 5
  }
  redistribution
  source rack Forest/Giguere
  target rack Hotel-de-Ville/Baie-Carriere {
    day of week: Tuesday
    timeslot: Midday
    number of bikes: 6
  }
  redistribution
  source rack Hotel-de-Ville/Baie-Carriere
  target rack Hotel-de-Ville/Sabourin {
    day of week: Wednesday
    timeslot: Morning
    number of bikes: 3
  }
  redistribution
  source rack 3rd/Dupuis
  target rack Hotel-de-Ville/Sabourin {
    day of week: Wednesday
    timeslot: Morning
    number of bikes: 5
  }
  redistribution
  source rack Forest/Giguere
  target rack Hotel-de-Ville/Sabourin {
    day of week: Wednesday
    timeslot: Midday
    number of bikes: 6
  }
}
```