

How to install MISTA:

MISTA is a standalone executable JAR. [Get it from the official website](#).

We also recommend you use an IDE such as [Eclipse](#) to import, run and test your code. There are two different Java projects you'll need to inspect, one per exercise.

Exercise 1: Understanding a model

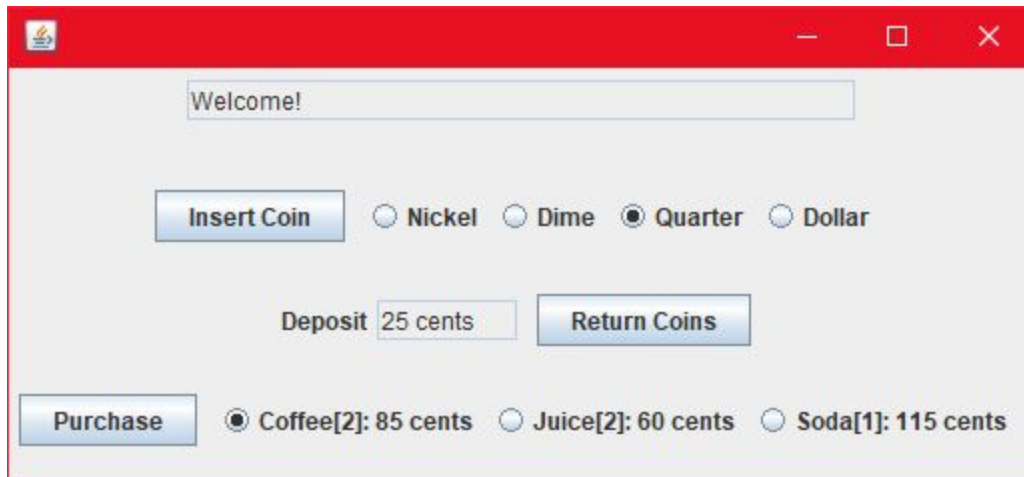
MISTA is a lightweight academic program that allows you to *model behaviours as a state machine through a GUI*. The models can either be a regular Finite State Machine with transitions between reachable states, or can be a Function Net graph, which are state machines where tokens of data travel through transitions between states to represent changes in a system.

And finally, it allows us to *automatically generate a large number of test suites based on the model*, on a number of languages and test engines.

On these exercises, we'll be working with the Function Net model.

First, we'll be looking at a complete Function Net, and try to understand how it works and what the resulting program should be doing. Then, we'll fix the problems with the program the tests find.

VendingMachine is a simple program written in Java Swing, with a self-descriptive purpose. It is nearly complete. You can insert coins, and in turn, receive a drink and some change. You should find 3 packages, within the project. *vending* is where you can find the program's logic as 3 class that abstract the concepts of Drinks, Coins and the Vending Machine itself. *gui*, on the other hand, has two files, one if the controller behind rendering the program in a window, while the other is the program's entry point. By running *VendingMain.java* as a Java application, you should see a window come up, which is the **VendingMachine** program proper.

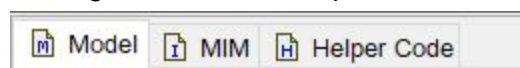


And finally there's the [test](#) package. Within, you should see a java file called [VendingTest.java](#). This is a handmade jUnit test suite with a single unit test. However it doesn't cover a lot of the program's behaviour.



Here's where the other files in the test package come into play, *we'll be using them to generate more test files*. You may notice that these are not Java files but rather a binary *xmid* file and a *xml* file describing a model. To open these you will require **MISTA**. It's not recommended you move these from these location as the program will eventually generate the jUnit files in the same directory the model file is found.



With **MISTA** opened, do File -> Open and then choose the [VendingMachine.xmid](#) file. Right away the program should show a model's graph, in function net form. With that done, let's describe a bit of MISTA's workflow.

The program has 3 main Views, which are sequentially the basis of how it works. Views can be accessed through the tabbed navigation bar at the top.



The *Model View* is our first stopping point, and where all of the model is described. It has 4 main elements we need to know about, all of which you can already see placed on the existing model. These are:

- [Places](#)  - Places are discrete conditions in the model, or more accurately thought of as the equivalent of data holders for the program. These will hold tokens, which represent objects, and which move either around between other places as events occur, or change properties between each other.
- [Transitions](#)  - These represent Events, things that may occur, or in terms of a program, they are functions that you call. In the model, they show in which scenarios tokens can be moved between two places.

- **Arcs**  - These connect between a Place and a Transition, and never between two of the same. Depending on whether the Place is on one end or the other, the Place will be an Input or Output place for the transition. Ultimately, these let you give transitions a direction, as well as define what types of tokens they transit.
- **Annotations**  - These come in two varieties. A functional, bordered in Yellow, and a nonfunctional one, bordered in Black. There's a functional annotation placed in the model that begins with **INIT** - this tells the model which tokens are found on the model on the starting condition. Other valid commands exist, such as **ASSERTION**, which let you tell the model conditions that should be true at any point in the model's operation.

But this might be a bit confusing as a general overview, so rather than try to understand the model on its own, why not observe the model in motion? So here's your first **tasklist** for the exercise:

- ☐ Run a simulation on the model.
- ☐ Understand the at least one transition of the model.

So let's run down an explanation of what we're doing. We'll be trying to understand how the **insertCoin(v)** transition works. Before beginning, we should look at the model for important elements. These are: The **coins place**, the **deposit place** and the **insertCoin(v) transition + arcs**. From the INIT annotation and the arcs, we can tell what tokens are relevant to each.

The **coins place** represents the number of coins the user has in hand. Its tokens have the format $\langle v, c \rangle$, for value and quantity.

The **deposit place** represents the total value inserted into the machine. It has a single token with the format $\langle d \rangle$.

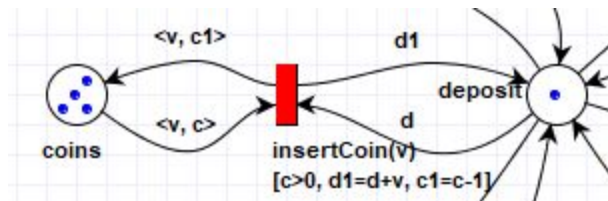
And finally **insertCoin(v)** is a transition that receives two tokens, a $\langle v, c \rangle$ and a $\langle d \rangle$, and outputs a $\langle v, c1 \rangle$ and a $\langle d1 \rangle$. If you double click the transition using the **selection tool**, you should be able to more clearly read that the guard clauses define $d1$ as the sum of d and v , and $c1$ as c minus 1. Additionally, that c must be larger than 0.

To simulate the model, you should either click Analysis -> Simulate, or click the Simulate button directly. A simulation control panel should have opened, and the model should have changed.



So, let's take a look at what that did.

There should be 4 **blue** tokens within coins, and 1 within deposit. These are the tokens defined in INIT, and you can read the current state on the left and confirm that it matches up.



But now, let's try to fire up an event.

Events that you can currently fire up are shown in **red** on the model, and currently only **insertCoin(v)** is available. On the right side of the panel, you can confirm on this Event list. As well as that, you can choose what parameter this event will fire with. As all 4 coin tokens are valid, and only one deposit value is valid, you should have 4 options.

Choose the largest option: $v/100$, to send a 100 cents coin (a dollar) through to the machine. Then press the **Play** button once.

So what changes did this do to the model's current state? On the coins place, the $(100,1)$ token should have become a $(100,0)$ token, while the deposit place saw its token go from (0) to (10) . While the resulting two tokens are completely new by the model's perspective, we could view the transition as the original coin and deposit token being updated in value. Also, since the deposit have a value larger than 0, more events may now be fired.

The rest of the model follows a similar logic of a dual-input, dual-output event that returns the tokens to the original place with the values changed. This is enough to create tests that cover the program's functions.

If you want to, for fun, you can change the **INIT** annotation and give it a larger quantity of coins, an initial deposit value, an extra coin token with value 1 (a penny). You can also set the simulation on **Random Simulation** and watch it fire random events and reset. When you're done, I recommend returning everything to its initial state before moving on.

The *Model-Implementation Mapping (MIM)* View is where you tell **MISTA** how to translate between model elements, and their standard language (Java) implementation. This is important as *the program doesn't actually look at any Java code to generate the test suites*.

There's 4 submaps on this View:

- **Objects** - Values found in tokens may not be necessarily integers. Should you give a token a value such as String1, you could tell MISTA here what String1 should resolve to (for example: "Foobar")
- **Methods** - Methods represent the Transition Events as they occur. For example, insertCoin, the event we saw previously, translates to vendingmachine.insertCoin(Coin.X) in Java.
- **Accessors** - Accessors are called whenever the current state needs to be obtained, usually to verify assertions. These can be thought of as Getters, in general, but other checks may be built into the list.
- **Mutators** - Mutators, conversely, can be thought of as Setters. They are called by, for example, the INIT annotation to setup the initial state of the test suite.

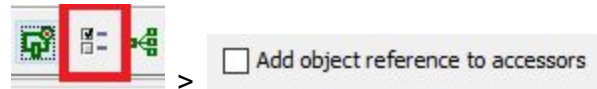
You should also notice that several methods have variables after an interrogation mark in the parameters. This is used to state that the Model Level object found next to the ? should be *internally translated*. For example, in the Accessor deposit(?d), the ?d on the left side is a model level integer that should appear as a Java integer on the right side ?d.

And finally the *Helper Code* View is where you write down implementation-specific code that isn't a result of model mapping, such as Imports, or helper functions. You can take a quick look at it if you wish (you can scroll down).

But onto the actual test code generation. Here's the next **tasklist**:

- ☐ Make sure the Test Generation Options are correct.
- ☐ Select the correct target test type, standard language and engine.
- ☐ Run the tests on the Java project.

Before generating the test suite, let's address a little quirk with this model that is mentioned on a comment annotation in the model view. While the other implementations use the tested class' object as the reference for the invoked functions, the accessors use a function defined in Helper Code. To go around this issue, find the Test options button and [unchecked "Add object reference to accessors"](#). You should leave the other references checked.



Next up, let's make sure we're [actually generating jUnit test cases](#). Simply look in the top right corner and choose the following options.



And finally: [Generate the test code](#)! You can either click Test -> Generate Test Code, or click on the button directly.



A new View should open up, which shows the contents of the new Java file, [VendingMachineTester_RT](#). This file is directly saved/overwritten on the same directory the xmid file was opened from, and as such, it should already be placed in your java project's Test package.

Good news, we won't need **MISTA** anymore on this exercise. Try running as a jUnit test suite. Well, Bad news, it seems that there might be some bugs with the project. The program is incomplete!

Here's the opportunity to [brush up on some Test-Driven Development](#). Here's the final task list for now:

- ☐ Using the tests, identify what are the problems.
- ☐ Fix the problems on VendingMachine.java.

Exercise 2: Creating a model

Now we'll be implementing our own function net of a system and generate junit tests from it. Don't worry, we'll be keeping things simple.

Lists is another Java project, with a single class, **Main**, where you can find two ArrayLists of strings. There's two functions to transfer strings between one list and the other, and one function that counts the total strings on the lists. Let's pretend this simple class is the end result of a model you wanted to implement. Given the similarities between the class's behaviour and a state machine, we can draw some parallels to help the modelling.

The **ArrayLists** will be **Places** in the model.

The **Functions** will be **Transitions** in the model.

The **Strings** will move through **Arcs** in the model.

With this in mind, let's open up a new instance of **MISTA** and load the file **src\test\Lists.xml**.

You might right away notice that there's some work done on this file. There's functional annotations on the model view, and a couple of entries unfinished on the Model-Implementation Mapping view. Here's what we can try to figure out from what's available to us:

- A **State** called onList should exist in the model with accessors and mutators
- There's **Objects** labeled T1 through T4 that are instantiated on the onList
- An **Event** add() should also exist somewhere in the model, capable of transporting a token
- There's also an offList reference in similar fashion to onList on the Assertions.

```
INIT onList(T1), onList(T2), onList(T3),
onList(T4)
```

```
ASSERTION onList(x) => not offList(x)
```

```
ASSERTION offList(x) => not onList(x)
```

The implementation level equivalence of each element present on the model should be easy to understand. With this said, the first step is to create the model. Go to the **Model View**. Here's a **tasklist**:

- ☐ Create **two places** - Don't forget to give them the correct model domain names
- ☐ Create **two transitions** - One also already has a defined name, so make sure to give it that. You should give the second one a fitting name.
- ☐ Create **four arcs** connecting the places and transitions - Don't forget that each arc can transport one token, so you should double click them and add a single x to the label. And don't accidentally put the arcs in reverse either!

With the model defined, we may still not be able to generate tests, but if everything is correctly set up, you should be able to open and run the simulation



control panel. It's recommended that that you do and verify that 4 tokens (represented as blue dots) start on the onList place and travel one at a time between it and the other place.

But with the model correctly set up, we still can't create tests until we tell **MISTA** how to translate from the model to standard code. It's recommended to pay close attention to the Main class for this part, as the program and its generated code is agnostic to the project's code (It prints out whatever you write as-is), so spelling mistakes will propagate to the junit test cases.

Here's the next **tasklist**, this time open the *MIM* (Model-Implementation Mapping) view, and write the model level instances on the left, and the equivalent Java code on the right:

- ☐ The program needs to know what is the **name of the class** we're working on.
- ☐ Create **the 4 object tokens** - Here's a suggestion: a string for each of the words in TVVS.
- ☐ Implement the **existing event** method, and **create another** - One for each transition. Reminder, since there's a ?x on the left, you might want to have a ?x on the right side too.
- ☐ Implement the **existing Accessor and Mutator** - Remember, accessors should allow you to tell what's the condition of the List, while mutators should allow you to add tokens to a List. You might want to look up the following Java methods related to the ArrayLists: add, contains, remove.
- ☐ **One more Accessor** - There's another thing missing. What are the functional parts of the model? Is there something in described in the annotations that also needs to be implemented in the tests?

Finally, **MISTA** should be able to know how to translate the model to standard code, and should compile its internal syntax properly. We can press test tree button to get an overview of what steps the testing will take, and test code button to generate the files.

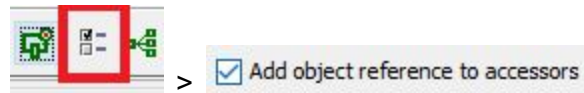


What the tests should be doing is moving Strings between the ArrayLists, and between each step it should be verifying if the assertions are correct. **But**, something *should* go wrong with the code you generate. If you try to run it on your project, it has several errors. Despite knowing how to translate the model, there's still some implementation specifics that need to be defined.

So here's one more **tasklist**, this time look into the Helper Code View:

- ☐ **Package Code** - The tests are generated in the same place the model files were on, which should be inside a folder called test. Java uses directory structure to define packages. Write the line of code that will rest at the top of the junit file and will make it a part of the test package.
- ☐ **Import Code** - While the tests are in the test package, the class we're testing is not. One line should be enough to import it into the test suite.

- ❑ As part of exercise 1, you might have had to tell **MISTA** not to include a reference to the object when using accessors. Since we're not using our own extra code segment in test suite, and DO want to reference the Main class, make sure to turn that option back on.



- ❑ Make sure we're actually generating JUnit tests by selecting the right language and testing engine on the top right corner.



With that done, close the Generated code tab and re-generate new code. Your files will be overwritten and the test suite should run within your project without any issues.

Since you have successfully generated a test suite from a model you've created, you can consider yourself done 😊.

However, if you want to fine tune one detail, you may want to complete this next **task**:

- ❑ The test should make sure that the number of Strings doesn't change

There's a couple ways this can be achieved through the model, such as for example, using a tokenCount guard on our events. However what we're going to use is the numStrings() function in the **Main** class. We want our test to actually make the verification itself. What we need is way of telling our model events to fire a verification every time they occur.

If you double click a **transition** on the model view, you should be able to edit more fields than just its name. The field you're looking for is **Effect**. The effect of a transition provides a way to define Test oracles, in other words, if you write down a predicate method here, you can then map it to implementation level code as an Accessor or Mutator.

Choose a name for the predicate and write it down as the effect of both the transitions in your model. Then, on the MIM view, map that predicate to a valid JUnit verification. For example, an AssertTrue() that encapsulates the number of tokens and the class function. Should this be done correctly, you will be able to see that the verification occurs on generated tests after every step of adding or removing strings from a list.

If you're confused as to what you're supposed to do, you can open a separate instance of **MISTA** and check the **VendingMachine** model from the previous exercise, where a similar approach was taken to verify that the change was handed out correctly.