Interactive Session 1 – Fundamentals

Download and Install

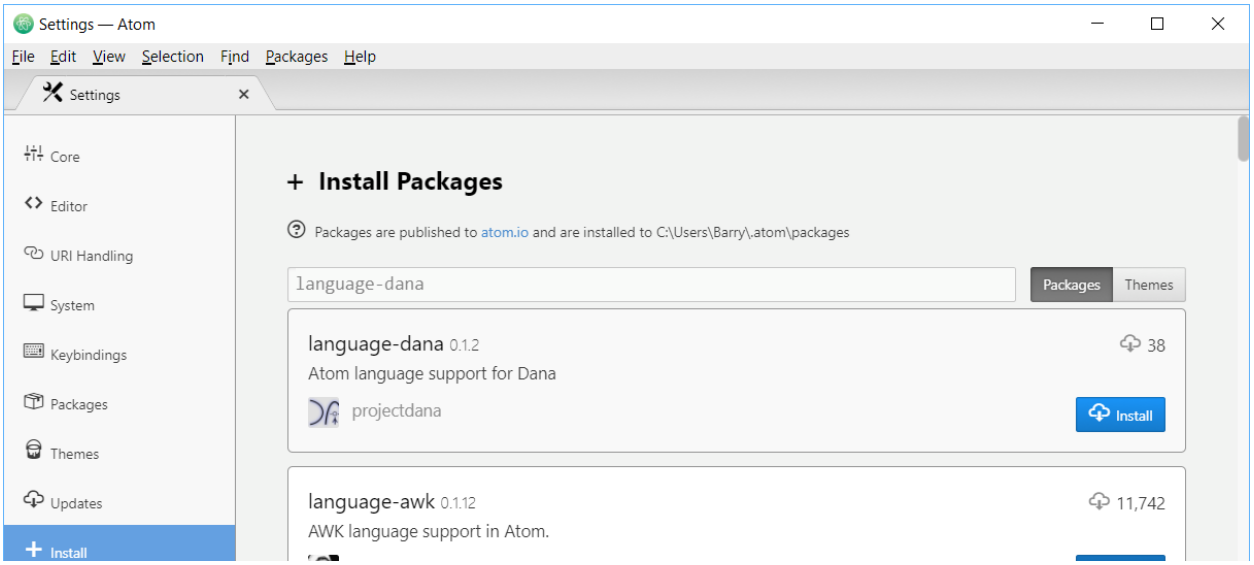
Start by downloading the latest version of Dana from http://www.projectdana.com

Select the version which matches your operating system, then follow the appropriate instructions for your platform at http://www.projectdana.com/dana/guide/doku.php?id=installation to get things working (this is about 3 steps).

Configuring Atom

You can use any text editor you like for this tutorial, but we suggest using Atom which you can get from http://www.atom.io

To configure syntax highlighting and auto-completion, open Atom, then go to File → Settings → Install, search for language-dana then click Install:



Your first program

We start by creating a new directory for our first project. Make a new directory (anywhere you like) with any name. Now open Atom and create a new file. Save the file in your new directory as `Main.dn`.

Now we're going to write our first Dana program. We'll write a console application which reads some user input and writes it back out.

Type this program into your open file in Atom, then save your changes:

```
component provides App requires io.Output out, io.Input in {

        int App:main(AppParam params[])
                {
                out.print("Type something: ")

                char txt[] = in.readln()

                out.println("You typed '$txt'")

                return 0
                }
}
```

To compile the program, open a terminal window in your new directory and type the command:

`dnc Main.dn`

Assuming that there were no compile errors, this will generate a file Main.o

You can now run the program with the command:

`dana Main.o`

You should now be asked to enter some text and have it printed back to you on the terminal.

This program demonstrates the use of standard library APIs via `requires` directives, and also shows the way in which strings can be printed out within other strings using the $ notation. In Dana a string is just a character array and is an example of the principle of separating *pure data* from *behaviour*, such as an API to tokenise the string into separate fragments (see the `data.StringUtil` API for this capability).

Your first adaptive program

Next we'll write a very simple adaptive system, showing stateful components and the separation of application logic from compositional logic.

Create a new project directory, and inside this directly create another folder called resources. Within the resources folder create a new file called `Thing.dn`. This will be the interface used by the application logic of our system, which should have the code:

```
interface Thing {

	transfer int value

	void doThing()

	}
```

In the root directory of your project, create three more files: `Main.dn`, `Thing.dn`, and `ThingB.dn` – these three components are the **application logic** of our system. Add code to these files as follows:

Main.dn

```
component provides Thing requires io.Output out, data.IntUtil iu {

	void Thing:doThing()
		{
		value ++
		out.println("value: $value")
		}

	}
```

Thing.dn

```
component provides Thing requires io.Output out, data.IntUtil iu {

	void Thing:doThing()
		{
		value += 10
		out.println("value: $value")
		}

	}
```

ThingB.dn

```
component provides App requires io.Output out, time.Timer timer, Thing {

	int App:main(AppParam params[])
		{
		Thing myThing = new Thing()

		while (true)
			{
			myThing.doThing()
			timer.sleep(300)
			}

		return 0
		}

	}
```


This application logic is self-describing in the sense that we have defined an interface type and explicitly expressed which components require and provide this interface. Because we have two components implementing the same interface (`Thing`), we have a reason to adapt the system at runtime.

We next implement the composition logic which decides how to wire these components together and then how to adapt those wirings at runtime. We usually write this kind of composition logic in a highly generalised way, but here we use a minimal example for this specific system. Create a new file called `Composer.dn` and enter the code:

```
component provides App requires composition.RecursiveLoader loader, time.Timer timer,
composition.Adapter adapter {

	int App:main(AppParam params[])
		{
		IDC comA = loader.load("Main.o").mainComponent
		IDC comB = loader.load("Thing.o").mainComponent
		IDC comC = loader.load("ThingB.o").mainComponent

		App q = new App() from comA :< App

		asynch::q.main(null)

		bool mode = false
		while (true)
			{
			if (mode)
				adapter.adaptRequiredInterface(comA, "Thing", comC)
				else
				adapter.adaptRequiredInterface(comA, "Thing", comB)
			mode = !mode
			timer.sleep(1000)
			}

		return 0
		}

	}
```

This code loads the three specific components we created above, launches the “main” function of the application logic in a separate thread, and then adapts between the two implementations of `Thing`.

Now we compile the entire project, using the command:

`dnc .`

We can then run the system either by itself, or via the composer that we wrote. To run the system by itself we use:

`dana Main.o`

Or to run it though our composer (which continuously adapts the system) it’s:

`dana Composer.o`

Summary

This session has covered the fundamental mechanics of writing components, implementing multiple version of an interface, and writing a simple programmatic composer which loads other components and adapts their wirings. Everything else that we do is based on these concepts. In the next session we'll cover automatically discovering components (instead of hard-coding them as in this composer), then automatically discovering entire compositions, perceiving information about the running system, and using machine learning to select entire compositions based on the current environment.

Using APIs in the standard library

In completing the above exercises you have already used four different interfaces in Dana's standard library. This is simply a set of open-source interfaces and components in Dana's installation folder, which you can browse in your own time. The standard library is just a project like any other, with its own resources folder containing all interface type definitions, and other sub-folders of the root directory which contain corresponding implementation components of those interfaces.

The standard library is documented at http://www.projectdana.com/dana/api/ which is generated entirely from annotations in the source code within the central resources directory. Becoming familiar with browsing this API documentation will help with more advanced topics in the rest of the course.