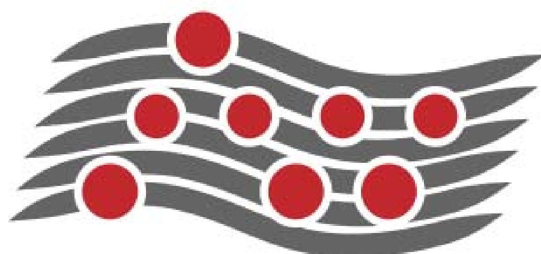


AtomWeaver

AtomWeaver Guide

Version 1.3.0

isomeris

AtomWeaver Guide

The AtomWeaver Guide teaches you how to get the most out of AtomWeaver.

AtomWeaver is a *Freemium* product (see definition in Wikipedia), that is, you can use some of its features for free for an unlimited period of time, and have the choice to unlock all features by purchasing a license. See the Licensing page for more information.

Introduction

AtomWeaver

AtomWeaver is an Integrated Development Environment (IDE) that implements the ABSE development automation framework. ABSE, which stands for *Atom-Based Software Engineering*, relies heavily on reuse. Learn more about ABSE here (guide) and here (reference).

If you are currently using another IDE, AtomWeaver is not meant to replace it. Instead, AtomWeaver works in conjunction with your current tool set, speeding up the development of your source code. You will continue to use your favorite IDE and tools to compile, test and debug.

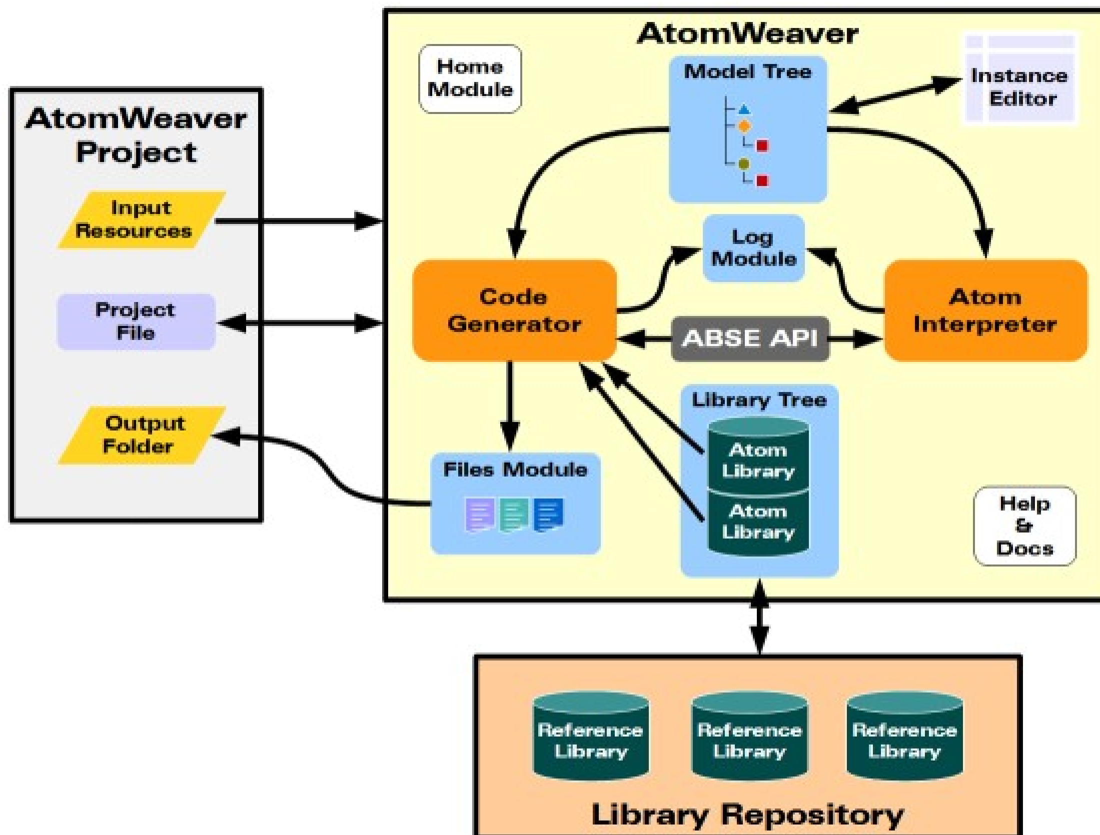
But AtomWeaver is not a "fire and forget" code generator. It is meant to support your application's complete life-cycle. From the moment you create a model, until your application life ends, AtomWeaver can support its evolution: defining requirements, generating the first code, making changes to the model, refactoring code and Atoms, managing issues, etc.

By implementing ABSE, AtomWeaver has some distinguishing factors over a regular IDE:

- It helps you cut repetitive coding, automatically decreasing maintenance costs, since the resulting code base is smaller: there's no need to maintain several instances of the same code.

- The abstraction level is higher: You can work mostly at the concept level and not at the code level.

AtomWeaver's Architecture



The above diagram gives you a quick overview of AtomWeaver.

Its two most important sub-systems are the *Atom Interpreter* and the *Code Generator*. The Atom Interpreter helps you build your project's model, along with the *Instance Editor*. The Instance Editor builds automatic editors for your Atom Templates.

AtomWeaver keeps a project as a folder that contains, besides its project file (".awp"), one folder for input resources, and another for generation results. Your Atom trees are stored on the project file.

The *ABSE API* is also at the core. Your reusable assets (Atom Templates), stored and organized into Atom Libraries, rely on it to generate code. The Code Generator loads your Model Tree, which drives the execution of your Atom Templates. By calling the ABSE API, generated artifacts are obtained. These artifacts are kept on the *Files Module*, and then saved by your command to the project's *Output Folder*.

The *Atom Interpreter* and *Code Generator* (as well as other sub-systems) send their messages to the Log module, that can be checked for interpretation or generation problems.

Finally, the *Library Repository* is a separate storage location where you can save your Atom Libraries for later reuse on other projects.

Starting AtomWeaver

After starting AtomWeaver, you are shown the Home Page, on the Home Module.

If it's the first time you are trying AtomWeaver, you are strongly advised to go through the introductory tutorials. You can access them right from the Home Page.

You should also keep the *In-Context Help* system activated, as it was made for newcomers to get an on-the-fly explanation of the AtomWeaver features you are looking at. The *In-Context Help* system is on by default, and you can activate it again through the *Help - Activate In-Context Help* menu command.

Only a handful of commands are available at this time. Your next logical step is to start a new project, or continue working on an existing one.

The AtomWeaver IDE

The AtomWeaver IDE (Integrated Development Environment) is a meta-IDE. As such, it has been developed to be as neutral as possible to any project you commit yourself to.

An AtomWeaver work session is composed of several modules.

Each module is responsible for a specific aspect of the project. For each project, you will find the following modules:

Home : Shows general aspects of the current project, as well as news and help resources
Library : Holds and manages the Atom Libraries used in the current project
Model : Holds the project model
Files : Holds generation results
Log : Registers what happens on the project during the current session

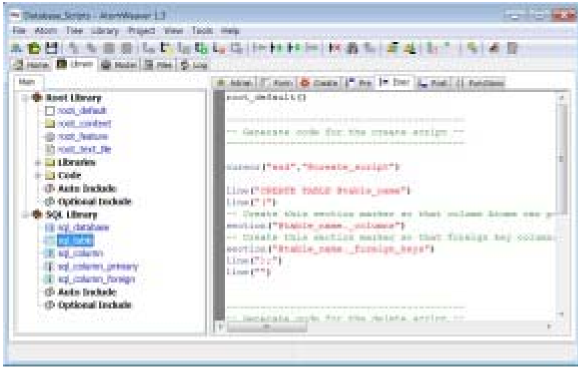
---

The Home Module

The Home module shows you the Home Page.

The Home Page is your start page for every project. Here you will find news and resources about ABSE and AtomWeaver, as well as other general tasks.

The Library Module

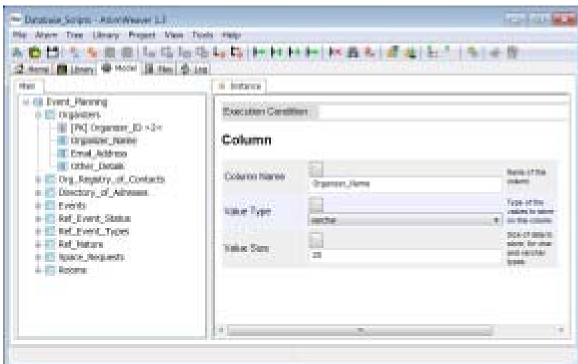


The Library Module holds and manages all the Atom Libraries you are using on your project. Some Atom types like Atom Templates and Atom Organizers can only be created on this module.

An Atom Library contains a group of reusable assets (Atom Templates) that you will use to build your projects. For example, the Root Library contains pre-defined Atom Templates for code scaffolding that can be instantiated on the Model Module.

The Root Library is always loaded into the tree.

The Model Module

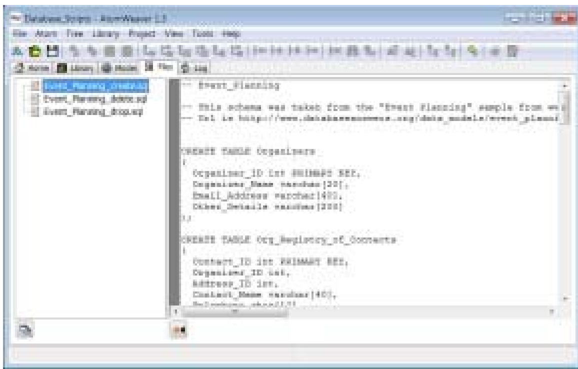


The Model Module is the project's main module. It's on this module that you implement your project's ABSE tree. You can generate software applications, fragments of larger projects, websites, data files, or a combination of these.

You will create and combine Atom Instances to build a correct model of your project, and then you can use the Generate command to generate the desired code for your project. The tree is recursively executed by the generator, so you can easily know in which order will Atom Instances be executed.

Before adding Atoms to the model's tree, if you have first to create some assets you can reuse. You create your reusable assets (Atom Templates) on the Library Module.

The Files Module



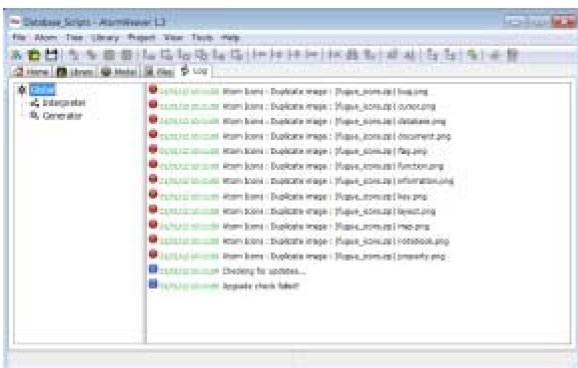
The Files Module is an auxiliary module: It does not manage an ABSE tree. On the Files Module you can browse the artifacts that were created as a result of a Generation command. The results will remain here until a new Generation command is run.

To see the contents of a generated artifact, select it on the left pane. The right pane will show the artifact's contents.

Traceability is supported: every generated line remains connected to its generator Atom. If you find a source line that is not correct, you can quickly jump to that Atom (or its Template) and make the necessary changes.

Use the *Jump to Atom Instance* button (premium feature) to find out what Atom generated a given line. Place the cursor on the generated line you want to inspect, and then press this button. AtomWeaver then takes you to the generating Atom. You can further select *Go To Atom Template* on the Instance's context menu to inspect the generator code.

The Log Module



The Log Module is an auxiliary module: It does not manage an ABSE Tree. Instead, it holds all messages issued by AtomWeaver. These messages can be notifications, warnings, or errors, and are classified on an hierarchical tree, which is not an ABSE Tree.

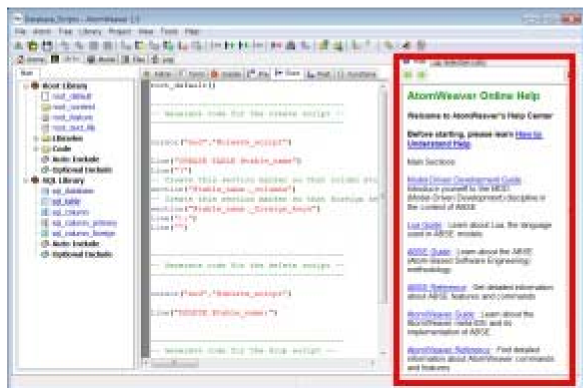
You can double-click the Main category on the left pane to expand its sub-category logs.

When a message is related to a specific Atom on the project, you can jump directly to that Atom through a pop-up menu

---

The Auxiliary Panel

The Auxiliary Panel is a collapsible pane that is located on the right side of AtomWeaver's window. If you are reading this page through AtomWeaver, you are looking at it right now.



The Auxiliary panel can show you additional information while you work. Its common use is to display the Help Sub-system. Additionally, it also manages Atom Selection Lists.

Projects

An AtomWeaver project is composed of at least two ABSE trees: The Library tree (managed by the Library module) and the Model tree (managed by the Model module).

The project is saved on a file-based database, using SQLite, with the "awp" extension. You can open this file on any SQLite managing application. Only one file is needed to store your project.

Generated files are not stored in the project. Therefore, when you reload your project, the Files module, the one that holds your generated artifacts, will be empty. You need to generate code again so that the Files module is populated with the resulting generated artifacts.

Starting a New Project

You can use the following methods to start a new project:

CTRL-N

File > New Project...

AtomWeaver has a default folder for all your new projects. You can change it on the general preferences dialog, through the *Tools* - *Options* command.

Your new project is saved as soon as it is created.

The first thing you should do is to make a rough plan of your project and decide what Atom Libraries are going to used. You should then load them into the Library module. If you need to build an Atom Library, you should do it at the beginning, even if you don't have yet a definitive idea of what concepts are you going to model. At least you can create a couple of Atom Templates that represent the top-level concepts. Because modeling with ABSE is a very

iterative process, you can first build a Library mock-up that will be refined later.

If you are going to use external libraries and/or frameworks, you don't need to include them in the AtomWeaver workspace. You can do it as usual and integrate them with your usual development IDE.

Opening an existing Project

You can use the following methods to load an existing project:

File > Open Project...

AtomWeaver loads a project in two steps: In the first step, it loads all Atoms from the project file; In the second step, AtomWeaver rebuilds the project.

Due to the fact that AtomWeaver only stores the Atom's persistent data, rebuilding a project implies that all "live links" between Atoms must be rebuilt from scratch. On large projects, and on slow machines, the loading process may take some time to complete.

Saving a Project

You have the following ways to save a project:

CTRL-S
File > Save Project...

Because the AtomWeaver's project file is actually a database, only the Atoms that have changed need to be saved. This is good for large projects because saves are much faster.

Unfortunately, having a database as storage also means that AtomWeaver's project file will be always growing, even after you delete Atoms from your project. An SQLite manager application might be helpful to purge previously deleted data from the project file.

Project Libraries

ABSE has a strong focus on reuse. Therefore, Atom Libraries are one of the most fundamental and important aspects of your ABSE projects. It's important that you master the use of libraries because it's on reuse that you'll mostly benefit from.

About Libraries

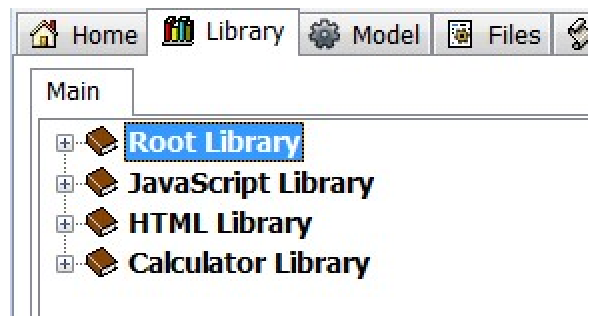
Libraries hold Atom Templates and reusable Atom Instances.

Each library has a short prefix, and this prefix is also used on all its Atom Templates. For

instance, on the Root Library, all Atom Templates will begin with *root_*. One example is *root_default*, the Base Template for all templates. This works like a namespace, and is a common technique in software development, which will make it very easy to know to which Library a given Atom Template belongs to.

Library Dependencies

In the Library Tree, Atom Libraries are ordered by dependency: The lowest-level Libraries are at the top. Any other Library that relies on those Libraries has to be placed after them.



In the above example, taken from one of the tutorials, shows the Calculator project's Atom Libraries. Because the *Calculator* Library reuses Atoms from the *HTML* Library, it must be placed after it. Likewise, because the *HTML* Library includes JavaScript code, it was placed after the *JavaScript* Library. Because the *Root* Library contains the *root_default* Atom Template, it is always the first Atom Library in the tree.

The Library Repository

The Library Repository is a central location where the latest version of a library can be stored and retrieved. Each team or AtomWeaver site has a single Library Repository.

Although the latest version of any library is stored on the Library Repository, each project has its own copy of an Atom Library, which can be an older version. This allows libraries to be individually upgraded without breaking existing projects.

Adding a Library to the Project

You create a new Atom Library through the *Library - New...* menu command.

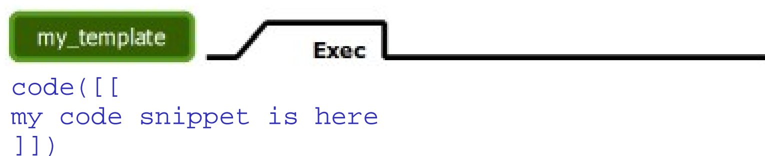
You'll be prompted to set the name of the new library, as well as a prefix to use on all Atom Templates in that library. After pressing OK, the new Atom Library is created and appended to the Library module's tree. The two special Library Include Organizers are also created.

The newly created Atom Library starts empty.

Creating Atom Templates

Atom Templates are the workhorse of ABSE: they capture your knowledge, define your project's architecture, and generate the resulting code or other artifacts.

The simplest code generator you can build with an Atom Template is something like this:



Translating into plain English: You create an Atom Template named *my_template*, and on the *Exec Section* of the Template, you append the code above. The code is written in Lua and contains a single call to a code-generating command named *code*, which uses Lua's long brackets to generate a multi-line block of code (commonly called a *snippet*).

Don't understand the diagrams shown here? Check this page!

Adding Atom Instances

There are two types of Atom Instances you can create on an Atom Library: Regular and Auto-Generated.

Regular Atom Instances

Regular Atom Instances, which have exactly the same behavior as Atom Instances that reside on a project model, must be created under the Library's Include Atom Organizers, labeled *Auto Include* and *Optional Include*. These Atom Organizers work as virtual roots of a project model, and therefore you can create portions of *ready-made projects* that you can later include to a full project.

Atoms on these two virtual branches usually model common parts of projects that are used over and over again. For instance, if you work on a specific domain, and have a set of classes that you use on all your projects, then you can model them on the *Auto Include* branch of an Atom Library. When starting a new project, you just need to add the Include Atoms through dedicated commands or Atom Templates. All the Atoms under the *Auto Include* branch, and those under the *Optional Include* branch that are selected for inclusion are executed and will generate code just like the rest of your project model.

Auto-Generated Atom Instances

When you create an Atom Instance under an Atom Template, that Instance becomes an *Auto-Generated Master Instance*. Being an Auto-Generated Instance means that every time you instantiate the Template, that Instance is automatically created under it. Being a Master Instance means that all Instances that are automatically created are clones of the Master. These Instances are called *Slave Instances*. Any changes to the Master Instance will automatically be replicated on all its Slave Instances.

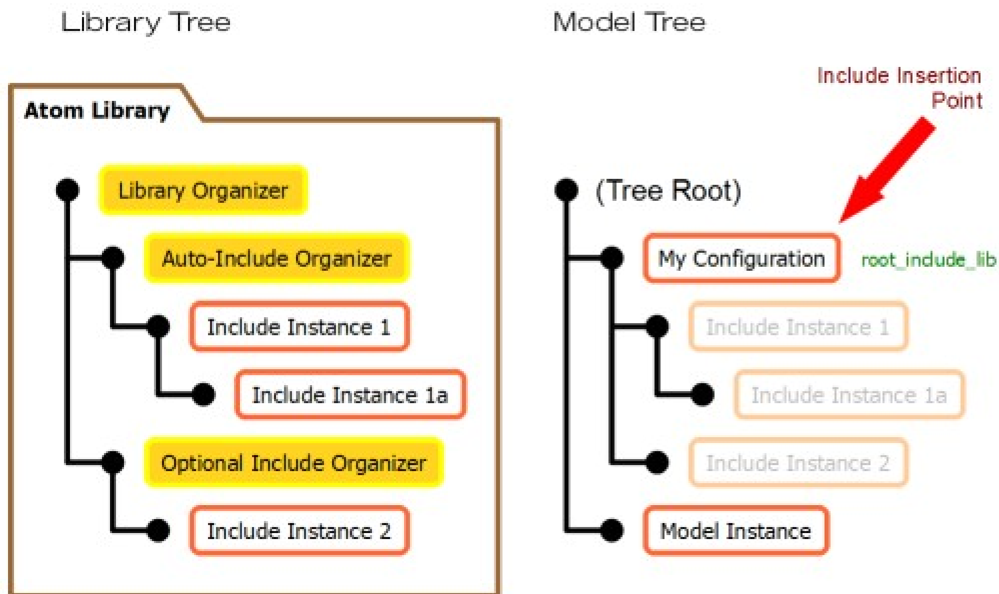
Library Include Atoms

In an Atom Library, you can include ready-made models that will make part of your project, and which will be automatically generated along with your project-specific models. However, you may wish that the way these ready-made models are generated could be influenced by

settings in your own project.

Because Library include Atoms reside under a special *Include Atom Organizer*, they will not have access to the model's variables unless you explicitly include these Atoms in some point on your project model.

For this you can use the include_lib() or include_all_libs() commands, or simply instantiate the *root_include_lib* or *root_include_all_libs* Atom Template somewhere in your project model. These commands will "virtually insert" the Include Instances under itself.

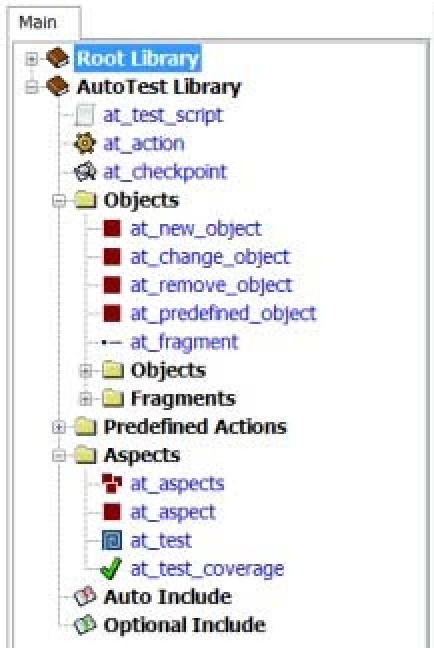


Then, when an Library include Atom reaches the *Include Atom Organizer* while trying to access a variable, AtomWeaver will automatically jump to the *Include Insertion Point* (the Atom Instance) and continues its variable search as if the Library include Atoms were actually under the *Include Insertion Point* Atom.

Don't understand the diagrams shown here? Check this page!

Trees and Atoms

An ABSE model is always a tree. AtomWeaver represents ABSE trees using the operating system's native tree widget:



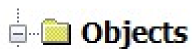
Starting from a virtual root Atom, the model grows by adding new Atoms that are always children of others that already exist. The tree's virtual root Atom is never shown.

Atom Labels

The label of an Atom gives you plenty of information about itself and its state.

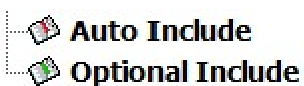
Atom Organizers

Atom Organizer labels are shown in black, in bold letters. You cannot choose an icon for an Atom Organizer. It's icon is set according to its purpose. A Folder Organizer looks like this:



A Library Organizer looks like this:

And the Include Organizers look like this:



Atom Templates

Atom Template labels are shown in blue. Their label gets the Template's own name.

If the Atom Template is invalid, because an error has occurred while running its Admin Section, its label turns red until the problem is solved:

Atom Instances

Atom Instance labels are shown in black, and their content is defined by their label's formula, set by the label() command. Their icon is set by the icon() command.

If the Atom Instance is an Auto-Generated Slave Instance, it will be shown with a green label. If the Slave Instance has the *blocked* qualifier, then the "[]" symbol is prefixed:

If the Slave Instance has been relocated, then the "<>" symbol is also prefixed:

Just like an invalid Atom Template, an invalid Atom Instance will have a red label:

If it's an invalid Auto-Generated Instance Slave, then its label will be orange:

If the Atom Instance is excluded by an Execution Condition, then its label turns grey:

If it's an Auto-Generated Instance Slave that is excluded by an Execution Condition, then its label turns light green:

If an Atom Instance is referenced by others, the total number of referencing Atoms is appended to its label, using the symbol ">n<" where *n* is the number of referencing Atoms:

The Atom Instance Form

Each Atom Instance on your model will help define a part of your system and will then generate the necessary code. To support variability, each Atom Instance may require zero, one, or more variables in order to feed their values into its Atom Template. For each variable you need to fill, there is a corresponding Parameter on the Instance's Template.

Then, when an Atom Instance is selected, AtomWeaver automatically builds a form to edit the Instance's variables:

Instance | Areas

Execution Condition :

<table>

Columns	2	Number of columns on this table
Rows	2	Number of rows on this table
Align	center	Table alignment on page
Background Color	#eaccaa	Background color for this table
Table Width	35%	Table width. Add units accordingly
Border Width	4	Table border width, in pixels

Code

Title	Calculator table	Code title
Description		Code description

A typical Atom Instance form

The order by which these variables are shown depends on the following factors:

- The order of the Parameter within the Template's Admin Section.
- The order of the Group from where the Parameter belongs.
- The Template inheritance chain: Parameters from derived Templates appear first within their Group.

Common variable buttons

This button is placed before each variable and allows you to set that variable using an expression. If you set one, its input box becomes read-only, and the variable's content begins to be set by the result of that expression. When you press this button, the expression editor is shown:

Expression Editor

Expression :

OK | Cancel

Set your one-line expression and press OK. AtomWeaver will evaluate the expression by feeding it to the eval() function. If you are setting a text-type variable, the expression should result in a string. If you are setting a link-type variable, then the expression should result in an Atom.

After you set an expression, the previous button becomes colored, indicating that an expression is set. You can press this button to edit the current expression. If you remove the expression, it will also be removed from the Atom and you can then set the variable by a direct value.

This button is shown next to a link-type variable (set on the Template by the param_link()

command), and lets you select a target Atom using the Atoms List Explorer.

This button is also shown next to a link-type variable, and lets you jump directly to the Atom that is currently being linked by its variable.

Setting values on the Atom Instance's variables

You can set a variable by typing a value into its edit box. If it's a list-type variable, you will use a drop-down list to select a pre-defined value from.

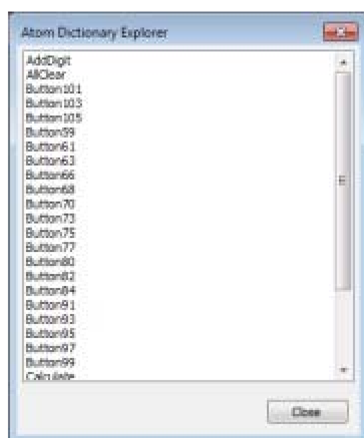
If it's a link-type variable, you can use the selection button pictured above and shown next to the edit box. If the target Atom has an entry on the Atom Dictionary, then you can just type the entry's text on the edit box.

Your variables are only set after the Atom Instance is updated. This happens whenever you start any AtomWeaver menu or tool bar command, if you select another Atom, or if you press F5.

Organizing and Accessing Your Atoms

Soon you'll find your project with hundreds or even thousands of Atoms that you need to search and manage. For that matter, AtomWeaver provides features that allow you to organize, classify, and find Atoms in your projects.

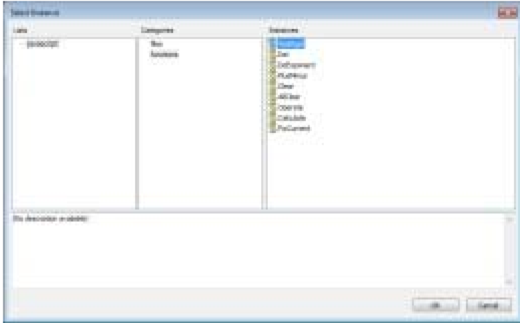
The Atom Dictionary Explorer



The Atom Dictionary is a list of Atom Instances that allows a particular Instance to be referenced by name.

The Atom Dictionary is composed of unique names, i.e. there are no two entries with the same name. AtomWeaver warns you when a dictionary entry is already assigned. Each entry can only link to a single Atom Instance. However, you can have two entries pointing to the same Atom Instance.

Atom Lists Explorer



Lists are useful to categorize Atom Instances into meaningful groups, allowing you to find a particular Atom much more easily. This is particularly useful when establishing links to other Atoms: The *param_link* command lets you specify an Atom list that will be used to restrict the selection to just a few Atoms.

Lists are classified in a tree-like fashion. Then, each list can be split into one or more categories. Each of these categories will then hold one or more Atom Instance references.

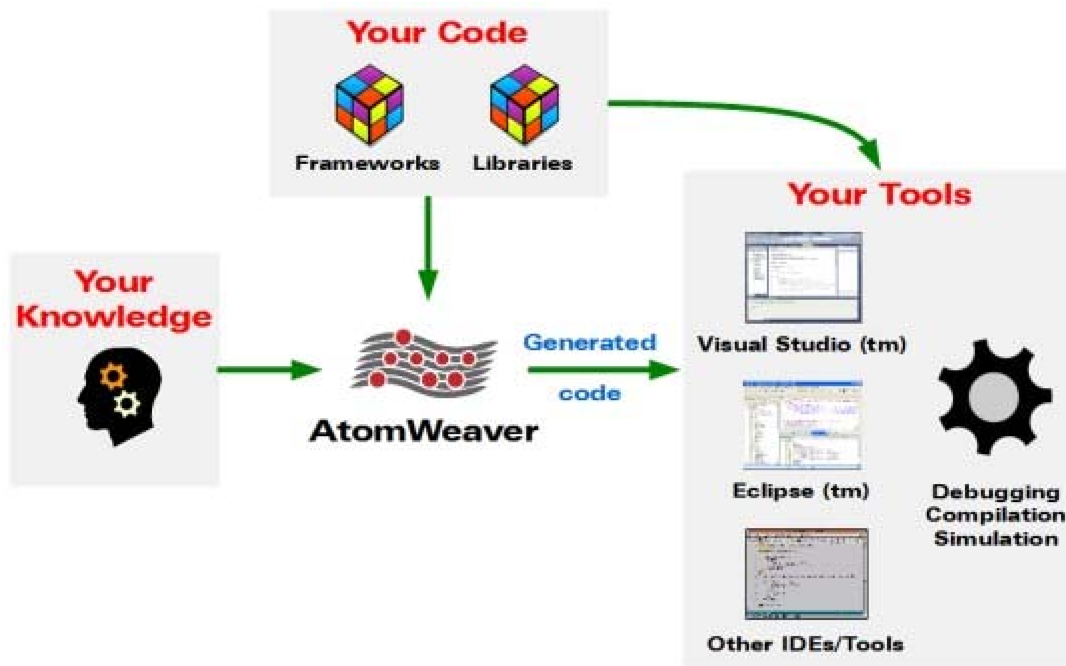
Code Generation

AtomWeaver has a built-in code generator. The code generator builds a Lua script by weaving the Atoms' transformation code. This script is composed and executed on the fly.

The generated files will be saved to the Output Folder as defined by the current project settings.

Integration with Other Development Tools

AtomWeaver is a standalone application. For some scenarios this might be seen as a disadvantage. But for most scenarios, this means that whatever tools you need to use on your development environment, you can use AtomWeaver to apply code generation and model-driven development on your projects.



As such, it can easily be integrated in any development setup you currently use. You can have AtomWeaver and your favorite IDE like Visual Studio or Eclipse running side-by-side, using AtomWeaver to generate your code, and your IDE for debug and build tasks.

Al the libraries and frameworks you currently use don't need to change. You can create support for those on your project Atoms, being then able to generate code using those libraries and/or frameworks.

Remember that AtomWeaver's model is now the source, not the source code itself. Therefore, any changes you make to your system have to be done on the model.

Integration with Microsoft Visual Studio

Because Visual Studio can automatically detect external changes to a file, you don't need any special preparations.

If you are generating the complete Visual Studio project, or just a part of it, the approach is the same:

Set a common project directory
Start AtomWeaver and Visual Studio
Create and edit your models in AtomWeaver
Generate and save the generated files
Compile, run and debug in Visual Studio

Remember that the model is the source, and as such you make your changes on the model and not on the generated code.

Project files

Visual Studio project files can be automatically generated by AtomWeaver: You can create an Atom Template that defines the project's main configuration, and that accepts child Atoms that specify each of the project's assets, usually source files.

Debugging

Debugging is performed as usual, by the Visual Studio's own debugging capabilities. Once the ABSE model is translated into code, you can compile and debug as usual.

Making changes to code under Visual Studio

The current version of AtomWeaver does not support source code round-tripping. The main reason for this is that the ABSE model is the source, and not the traditional source code of the application. Because the line to change was most certainly generated by an Atom Template, you will have to change its transformation code. Therefore, you should always make your changes to the model and then regenerate code to obtain the desired changes.

AtomWeaver Uses

ABSE is an universal modeling and code generation solution. It can support many uses. Let's see some simple and direct examples of what AtomWeaver can be used for:

User Interface Design

There is no specific AtomWeaver support to user interface design. However, it's not hard to create a simple Atom Template library of the user interface elements you want to implement, along with the construction constraints needed to create a consistent interface.

The bonus side of this flexibility is that you can build a GUI editor for any user interface, from window-based rich interfaces to single-line displays on embedded devices.

Data Access Layer

You can easily create a few Atoms to model your database schema, and generate the corresponding scripts for its creation, cleaning and deletion. If you want to go a bit further, you can implement a data access layer for your application, in the language of your choice, and using an ORM of your choice if you want to.

A great outcome of this solution is that you need just a few minutes to change your database schema, and automatically obtain the updated code.

Licensing

AtomWeaver uses an external license file to control access to its features. This file is named *license.txt* and is located on the application's installation directory. You can read and check this license file from inside AtomWeaver using the *Help* - *License* menu command.

==== AtomWeaver modes of operation ====

AtomWeaver can run in two modes: *Free Mode* and *Full Mode*. The *Free Mode* lets you use the application's free features for an unlimited period of time. The *Full Mode* unlocks all application features for a limited (trial license) or unlimited (permanent license) period of time.

Free Mode

The *Free Mode* lets you use those AtomWeaver's features directly dealing with ABSE. That is, *Free Mode* supports the full ABSE feature set. These are permanent features: You don't need to purchase any license to use them.

Full Mode

The *Full Mode* lets you use all available AtomWeaver features, including premium features. You always start using AtomWeaver in *Full Mode* anyway because you get a free 60-day trial license during software registration. After this period your license expires and AtomWeaver reverts to *Free Mode*. If you want to keep using all AtomWeaver features after the trial period you need to purchase a *Permanent License*.

==== Available license types ====

Trial License

A trial license allows you to use AtomWeaver in *Full Mode* for a limited period of time (60 days). Trial Licenses are valid for a single release, but are valid for all versions within that release. For instance, if you requested a trial license for version 1.3 and version 1.4 gets released within the trial period, you can install the new version and use the same license.

When the trial period expires, AtomWeaver will automatically revert to *Free Mode*. The projects you may have developed within the trial period remain fully functional and you can regain full application features by purchasing a Permanent License.

Permanent License

A permanent license allows you to use AtomWeaver in *Full* mode for an unlimited period of time for that release: If you purchase a license during release 1, you'll be entitled to upgrade to all 1.x versions of AtomWeaver. However, you'll have to purchase an upgrade to versions 2.x, unless you are under a maintenance contract.

Click here to buy a permanent license, or visit http://www.atomweaver.com/buy.html

---

AtomWeaver Premium Features

AtomWeaver premium features are only available in *Full Mode*. You can run AtomWeaver in *Full Mode* using a Trial License or a Permanent License.

AtomWeaver premium features

Atom Dictionary Explorer : Lets you list all dictionary entries and directly jump to any associated Atom.

Atom Lists Explorer : Lets you check all existing Atom Lists and directly jump to any associated Atom.

Icon Manager : Lets you see all available Atom Icons as a gallery and copy the chosen icon's name to the clipboard.

Atom Template Wizard : Let's you easily create an Atom Template, helping you with all the presentation, constraints, parameters and discovery aspects. See also the final list of a Template's parameters through its inheritance chain.

Find & List Atoms : Search Atoms using several criteria and build a selection list with the results. Then, you can directly jump to any listed Atom.

Traceability Support : On the Files module you can select a generated line and jump directly to the Atom that generated it.

AtomWeaver Glossary

You'll find here some terms that are used throughout this guide regarding AtomWeaver:

Atom Interpreter

The Atom Interpreter evaluates all Atoms in your project so that AtomWeaver can create and maintain a "live" version of your model.

The Atom Interpreter evaluates the Atom Template's Admin and Form Sections, and runs the Create Section for all Atom Instances in your model and loaded libraries.

When the Atom Interpreter detects a problem with your model, it adds a message to the *Interpreter* log on the Log module.

Constrained Atom Instance

Atom Instances whose Templates are constrained through the command

under()

are considered Constrained Atom Instances because they cannot be freely placed on an ABSE Tree.

Free Atom Instance

Atom Instances whose Templates are not constrained by an under() command are considered to be Free Atom Instances.

Therefore, Free Atom Instances can be created anywhere on an ABSE Tree.

Icon Library

A folder on AtomWeaver's installation that contains all the icons that can be used on Atoms. The name of the icon corresponds to the name of the file (without extension).

Meta-IDE

A meta-IDE is an IDE (Integrated Development Environment) that can be adapted and specialized for a specific context and target. AtomWeaver is a Meta-IDE because it lets you configure it through Atom Libraries that you choose to include. Those libraries will contain customizations, configuration and the implementation of custom commands.

Work Folder

AtomWeaver's Work Folder is where all user-created or user-editable files are kept. This folder is set at installation time. If you want to change AtomWeaver's working folder, edit the *workdir.txt* file on the installation folder (the folder that contains *atomweaver.exe*).