
# Ciera User Guide

Levi Starrett

2020-01-31

# Table of Contents

# Preface

Welcome to the Ciera User Guide. This guide will give you a comprehensive overview of Ciera, how it works, the features it supports, and how to use them. Ciera is an open source project and depends on the community to move forward.

## History

I started writing Ciera in early 2017 just as a personal project — a sort of playground to experiment with model-based model compiler concepts. At times I envisioned it to be a replacement model compiler for MC-Java to be the architecture on which the BridgePoint project rests (I haven't completely lost that vision, but it is more blurry than it once was). At times I tried to make it everything to everyone. I had a fascination with making the generated OAL as readable as possible, which sometimes compromised in other areas that should not have been compromised. It has always been a goal to replace MC-3020 as the primary model compiler used for compiling model compilers. Over the years it has morphed and changed and now it stands as a somewhat complete general purpose Java model compiler with a handful of features useful to model compiler developers and a handful of experimental features. Instead of starting with the xtUML spec and a set of test models, as Ciera has grown, it has implemented just enough of xtUML to support the current project and therefore is somewhat skeletal.

## What Ciera is

- A personal project that turned into a semi feature complete model compiler
- A generic Java model compiler
- A tool for building other model compilers
- A fully self-building, self-supporting model compiler
- An open source project

## What Ciera is not

- A commercial project
- A thoroughly tested/verified architecture
- An architecture suited to real time or low memory environments

## Relationship to Maven

As a Java project, I chose Maven as the preferred build tool for Ciera. As will be explained later, Ciera does not strictly depend on Maven, but it is almost not worth talking about using Ciera without talking about it in the context of a Maven build environment.

# Versioning policy

Ciera adheres to the Maven versioning convention in the following format:

```
<major>.<minor>.<incremental>[-SNAPSHOT]
```

`<major>` is the major version. Major versions are not compatible with one another.

`<minor>` is the minor version. Minor version gets incremented with feature updates that do not break backwards compatibility with prior releases in the same major version

`<incremental>` is the incremental version. It is updated any time a new release is published that does not have a significant new feature set (most if not all bug fixes)

`-SNAPSHOT` is appended for unstable development versions. Non-snapshot versions may not be re-released without incrementing the version, but snapshot versions may change, so use caution.

# Jump Start

In this section, we will go start to finish on how to create, build, and run an xtUML project from scratch. This guide assumes some basic knowledge of how to use BridgePoint. For help with BridgePoint and xtUML, please visit xtUML.org.

## Example projects

If you have not already, go to the examples page of the repository. That is a great place to start if you find yourself wanting to learn by viewing existing projects.

## Dependencies

- BridgePoint latest nightly
- Maven
- pyxtuml

On Ubuntu linux:

```
sudo apt-get update && sudo apt-get install -y maven python-pip wget unzip
pip install pyxtuml
wget https://s3.amazonaws.com/xtuml-releases/nightly-build/org.xtuml.bp.product-
linux.gtk.x86_64.zip
unzip https://s3.amazonaws.com/xtuml-releases/nightly-build/org.xtuml.bp.product-
linux.gtk.x86_64.zip
```

On macOS:

```
brew install maven wget unzip
pip install pyxtuml
wget https://s3.amazonaws.com/xtuml-releases/nightly-build/org.xtuml.bp.product-
macosx.cocoa.x86_64.zip
unzip https://s3.amazonaws.com/xtuml-releases/nightly-build/org.xtuml.bp.product-
macosx.cocoa.x86_64.zip
```

## Download the Ciera runtime library

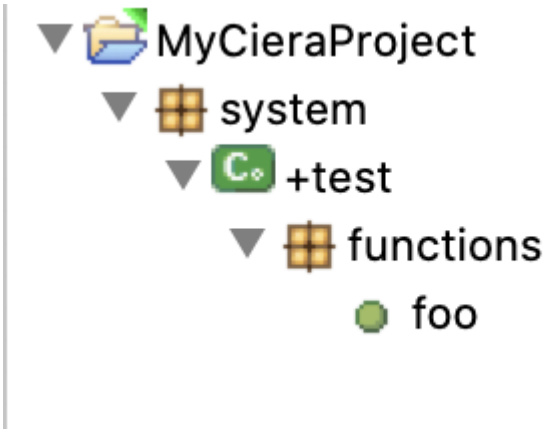
This step is not necessary if you have built a Ciera project before (including the examples). It simply downloads the runtime library (including modeled artifacts).

In a terminal window:

```
mvn -DgroupId=io.ciera -DartifactId=runtime -Dversion=2.0.0 dependency:get
```

# Creating an xtUML project

1. Open BridgePoint and create a new xtUML project. Call the project "MyCieraProject"
2. Create a new package, component, package, and function in the project. Name them "system", "test", "functions", and "foo" respectively. Your tree should now look something like this:




3. Enable inter-project references. Right click on project > Properties > xtUML Project > Inter-project References.
4. Import the Ciera runtime library. Import > General > Existing Projects into Workspace. Tick the "Select archive file" and navigate to `~/.m2/repository/io/ciera/runtime/2.0.0/runtime-2.0.0.jar`. Select the runtime project and click "Finish".
5. Open the "foo" function and enter the following code:

```
LOG::LogInfo(message:"Hello, World!");
control stop;
```

# Setting up the Maven build

1. Navigate to the Eclipse project location
2. Create a directory called `gen/`
3. Create a file called `gen/features.mark` with the following contents:

```
*,ApplicationName
*,ApplicationPackage
*,AsyncApplication
*,NonPersistentInstanceIds
*,RootPackage
*,SortComparator
*,TemplateDir
Association,Exclude
Attribute,NonPersistent
Component,EnableSimulatedTime
Component,InitFunction
Component,InstanceLoading
Component,Version
Model Class,Exclude
Model Class,NonPersistent
Model Class,UseKeyLettersForName
Package,DoNotSerialize
Port,BaseClass
Port,HttpEndpoint
```

4. Create a file called `gen/application.mark` with the following contents:

```
*,RootPackage,*,MyCieraProject::system
system::test,InitFunction,Component,foo
```

5. Create a file called `pom.xml` with the following contents:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>io.ciera</groupId>
  <artifactId>MyCieraProject</artifactId>
  <packaging>jar</packaging>
  <version>1.0.0-SNAPSHOT</version>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
  <dependencies>
    <dependency>
      <groupId>io.ciera</groupId>
      <artifactId>runtime</artifactId>
      <version>2.0.0</version>
    </dependency>
  </dependencies>
```

```
  <build>
    <plugins>
      <plugin>
        <groupId>io.ciera</groupId>
        <artifactId>ciera-maven-plugin</artifactId>
        <version>2.0.0</version>
        <executions>
          <execution>
            <id>pre-build</id>
            <goals>
              <goal>pyxtuml-pre-build</goal>
            </goals>
          </execution>
          <execution>
            <id>ciera-core</id>
            <goals>
              <goal>core</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
    <resources>
      <resource>
        <directory>${project.basedir}</directory>
        <filtering>true</filtering>
        <includes>
          <include>models/**/*.xtuml</include>
          <include>.project</include>
        </includes>
      </resource>
      <resource>
        <directory>${project.build.directory}/generated-sources/java</directory>
        <filtering>true</filtering>
        <includes>
          <include>**/*.properties</include>
        </includes>
      </resource>
    </resources>
  </build>
</project>
```

# Building the project

1. In the project directory:

```
mvn install
```

# Running the project

1. In a terminal:

```
java -cp $HOME/.m2/repository/io/ciera/runtime/2.0.0/runtime-
2.0.0.jar:$HOME/.m2/repository/io/ciera/MyCieraProject/1.0.0-
SNAPSHOT/MyCieraProject-1.0.0-SNAPSHOT.jar mycieraproject.MyCieraProjectApplication
```

# Congrats!

You've built and run your first Ciera project.

< | next: Features >

# Features

Ciera is a full model compiler which can generate xtUML models top to bottom from system modeling all the way to action language. The following is a summary of the features of Ciera and how to use them.

# System modeling

## Multi domain support

Ciera supports generating code for single and multi-domain projects. The compilation unit for Ciera is a package (referred to as the "root" package). Any component or component reference in the root package will be generated by Ciera. Any port satisfactions in the root package will also be generated into the system. To specify the root package, the `RootPackage` mark must be specified. See Marking: Root Package for more detail on marking. It is possible to maintain a project with multiple system configurations by modeling several packages that can act as the "root" package. Simply change the application mark and regenerate to switch bewteen two configurations. There is no support for generating code for multiple configurations simultaneously.

Ciera only generates model elements contained within components in the root package. Any types, EEs, or other elements required by the system must be imported into the component using a package reference. See more about package references in Package references.

## JSON serialized message passing

Ciera was designed with deploying components across networks in mind. For this reason, the message passing mechanism between components comes with built-in JSON serialization and deserialization support. For every message sent from a port, an instance of the `IMessage` interface is created for each message and passed to the port of the peer component. There, the values are unpacked and passed to the implementation within the component. The standard implementation of `IMessage` is simply an ordered list of values which correspond to the interface message parameters.

There are two ways to use the JSON serialization of messages to implement custom interface transports.

### Hand written "half" components

A user can create a hand written implementation of the component itself which "forwards" incoming messages across some transport to the other "half" of the component. In this scheme an instance of the component exists on both sides of the network and the component implementation itself is responsible for passing messages internally. The `serialize` method on the `IMessage` instances can be used to produce well formed JSON before sending across the chosen transport.

This is the simplest way to implement interfaces across some network, however the downside is that it is difficult to have real modeled behavior in the component cleanly without having to extensively modify generated code and continue to keep it up to date when the model changes.

This is the scheme that the GPS watch UI component uses. It works well since all of the behavior of the UI component is hand written graphical interface details.

#### Custom `IPort` implementation

A user can write a new Java class that implements the `IPort` interface and specify this class as the base for a generated port. In this scheme, the transport mechanism can be embedded directly into the port itself with no need for overriding generated code with a hand written class. This mechanism is much cleaner, but much more involved. This method would be suggested if a particular transport mechanism is expected to be widely used throughout the model (and not just a single case as in the GPS GUI example).

The details on how to mark a custom class as the base class for port can be found at Marking: Port implementation class.

This mechanism is considered advanced usage and there is not much documentation to aid in implementing it. If a user would go attempt to implement a custom port class, he should look closely at the default implementation in the `Port` class. This class should also be used as the supertype for any custom implementations.

### Component versioning

Ciera supports tagging components with version identifiers. By default, the version is the Maven artifact version plus a date and timestamp (if using the maven build plugin). A user can replace the maven artifact version with any string using an application mark. See details on marking at Marking: Component version.

## Class modeling

Ciera supports all types of class modeling constructs and relationship types. Derived attributes, referential attributes, class and instance operations, identifiers, and imported classes are all supported.

### Exclusions

By default, Ciera generates an interface definition and an implementation class for every modeled xtUML class. All relationships are also generated. Classes and associations can be marked for exclusion. See details about how to mark them at Marking: Element exclusions. No code will be generated for excluded classes and associations.

If you exclude a class, be sure to exclude all associations in which it participates; if you exclude an association, be sure to exclude all classes that participate in it. It is undefined behavior to leave associations with "orphaned" ends.

### Use key letters for name

By default, Ciera uses the name of the class to generate the Java class name (camel case with spaces removed and capitalized first letter). Sometimes, this can cause issues with name conflicts. The class key letters can be used *as is* for the class name. See Marking: Key letters for generated class name for details on how to apply this mark.

## State modeling

Ciera supports basic state modeling. All features necessary to achieve any behavior is supported in Ciera, however many features are not present that would have to be replaced with more involved (and sometimes clunky) patterns. For state modeling, it makes more sense to discuss it from the perspective of what is *not* supported. For that, see Restrictions/Limitations: State Machines.

## OAL action modeling

Ciera supports the majority of the OAL specification. Like state modeling, it is better to discuss in terms of what is missing. See Restrictions/Limitations

## Package references

Ciera supports package references within components. Any elements defined in a package referred to in a component will be translated as if they were defined within the referring package.

## Automatic selection sorting

For some applications, it makes sense to always sort selections by a specific attribute (e.g an integer identifier or a name). For model compilers, this allows output code to be diffable without explicit sorting in the application.

Ciera allows users to mark an attribute to be the global default sort key. Details on how to mark this can be found at Marking: Sort comparator

Ciera also provides more complex sorting with the `SORT` external entity. See Ciera specific utilities for more information.

If a sort key is marked, every selection of a class with that attribute will be sorted. Consider how this may effect application performance.

## Simulated time

Ciera supports executing models in simulated time. In simulated time, after initialization, the system clock is advanced to the expected generation time of the next delayed event. This event is generated and the system is allowed to run until no more events are waiting. The system clock is once again advanced to the next delayed event and so on.

Simulated time allows applications to be tested without waiting for wall clock time events, while

maintaining timing rules.

The details on how to mark a system to use simulated time can be found at Marking: Simulated time.

# Integration with hand written Java

Ciera provides the ability to integrate with external libaries or legacy code with hand written Java.

## Basic principle

Ciera allows any generated Java file to be overridden by a handwritten implementation by placing a file with the identical path in the source folder.

For example, the default output location for a Maven based Ciera build is `target/generated-sources/java` and the default source folder is `src/main/java`. A generated Java class `FooBar` located at `target/generated-sources/java/foo/bar/FooBar.java` could be replaced by installing a hand written class at `src/main/java/foo/bar/FooBar.java`. In this case, the original generated class will be renamed to `FooBar.java.orig` and the hand written class will be compiled into the binary package by the Java compiler.

## External Entities

Although any file can be overridden by a hand written implementation, it is not recommended since generated changes need to be merged in any time the model changes. External entities provide a clean interface to external code. Simply generate the empty EE once, copy the skeleton class into the source folder and fill in the implementation.

## Running a Ciera model

Ciera generates an "Application" class for every system deployment. This class contains a "main" method and can be used as the entry point for the application, however, it can also be imported and launched by a different hand written class that serves as the entry point for the application. The name of this class can be specified by a mark. See Marking: Application name for more detail.

## Running a Ciera model in a subordinate thread.

It is also possible to run an entire Ciera model in its own thread. This can be useful when working with legacy code where the xtUML model is only a part of a larger application.

This may look something like the following:

```
ExampleApplication app = new ExampleApplication();
app.setup(args, logger);
app.initialize();
Thread t = new Thread(app);
t.start();
```

Then to send a message to a port of a modeled component:

```
app.Component1().getRunContext().execute(new ReceivedMessageTask() {
     @Override
     public void run() throws XtumlException {
         app.Component1().Port1().message(param1, param2);
     }
});
```

It is required to execute the port message in this way to ensure that any modeled code is running in the context of the component's thread (to avoid data synchronization issues).

Arbitrary code can be run in the context of a modeled component the following way:

```
app.Component1().getRunContext().execute(new GenericExecutionTask() {
    @Override
    public void run() throws XtumlException {
        // arbitrary code here
        // ...
    }
});
```

# Built-in utilities

Ciera provides default implementations for useful external entities. Unlike other model compilers, with Ciera, it is *not* supported or recommended to overwrite the built-in external entities with hand written implementations (although this feature may be supported in the future). If a user would like to write a custom implementation of these external entities, a completely new EE with unique key letters should be created to do so.

The built-in EEs are included in the Ciera runtime library, so they should *not* be added to individual projects. Read more about this in the Build section.

## xtUML standard bridges

Ciera provides a standard implementation for the following EEs built into xtUML.

- Architecture (ARCH)
- Logging (LOG)
- Time (TIM)

Ciera does not implement the "State Save" external entity.

## Ciera specific utilities

Ciera provides implementations for the following additional utilities. Detailed usage and descriptions can be found in the API docs for each utility.

- Math library (MATH)
- String library (STRING)
- Command line parsing (CMD)
- Selection sorting (SORT)

< prev: Jump Start | next: Restrictions/Limitations >

# Restrictions and Limitations

Since Ciera was first developed pragmatically as a personal side project, its design and implementation was not driven by the same requirements as a "production" model compiler — pieces were implemented little by litte as needed by the few applications it was used to translate. This chapter describes the main restrictions and limiations of Ciera. Other restrictions are peppered throughout the user guide where it was natrual to discuss them.

## Coverage analysis

A more formal analysis of meta-model coverage has been performed and can be accessed online Engineering Documentation: 11844_Coverage.

## Interfaces

Ciera only supports asynchronous interface singals. Interface operations are not supported. Modelers must design their applications to be asynchronous and operate without relying on synchronous operations across component boundaries.

## State Machines

State machines are the area of greatest weakness for Ciera. While technically any desired behavior can be achieved with the current support, many model constructs are missing.

### Class-based (assigner) state machines

Ciera does not support class-based state machines. To achieve the same result, a modeler must create a new singleton class to act as assigner. Corollary to this, signal events are not supported. OAL actions must be written to "forward" events to the appropriate instance state machine to emulate this pattern

### Creation events

Ciera does not support creation events. Instances must always be created synchronously. It is trivial to achieve the same effect by creating an instance with a create statement and then subsequently generating an event to the new instance. Sometimes it is necessary to create a dummy start state to represent the non-existent state.

### Polymorphic events

Ciera does not support polymorphic events. Modelers must manually "pass down" events by having a switch like statement in the super type state machine to generate an event to the correct subtype.

### Other

- Ciera does not support transition actions bodies

- Ciera does not support designating states as final states

## Types

- Ciera does not support structured data types
- Ciera does not support declaring ranges on user defined types

## Deployments

Deployments are an alternative style system modeling scheme introduced to meet the needs of the MASL community. Ciera supports the component system modeling paradigm and does not provide any support for deployments.

## Miscellaneous

- Ciera does not support derived associations
- Ciera does not support delegations
- Ciera does not support exceptions
- Ciera does not honor default values on attributes (attributes are always initialized with the type default value)
- Ciera does not support baseless referential attributes
- Ciear does not support passing parameters by reference

## Known bugs

If you are running into a problem, be sure to check the issue tracker for known bugs.

< next: Features | next: Marking >

# Marking

The following describes the available marks and how to use them.

## Concepts

Marks are defined using comma separated values in the `application.mark` file. Available features are defined using comma separated values in the `features.mark` file. This file must be present and populated for marks to be loaded properly. The master version of this file can be found at features.mark.

Marks in `application.mark` are in the format:

```
<path>,<mark_name>,<markable_type>,<value>
```

where `<path>` is a unique identifier for the element you desire to mark, `<mark_name>` is the name of the feature, `<markable_type>` is the xtUML element type marked, and `<value>` is the value passed in to the feature.

A `*` for the path denotes all elements of that type. A `*` for the path and for the markable type denotes a system-wide mark.

BridgePoint includes a marking editor for editing `application.mark` files. Unfortunately, BridgePoint does not include the wildcard (`*`) which the Ciera marking mechanism uses. Perhaps BridgePoint can be extended in the future to support Ciera style marks.

## Application name

By default the application Java class (entry point for the application) is named `<project_name>Application` where `<project_name>` is the name of the xtUML project with spaces removed and each word capitalized. For example the application name for a project called "my project" would be "MyProjectApplication". If you would like to change this default behavior, add the following to your `application.mark` file.

```
*,ApplicationName,*,<app_name>
```

```
ex:
*,ApplicationName,*,CoreTool
```

where `<app_name>` is your custom name.

# Application package

By default the application Java class is generated into a top level package called `<project_name>` where `<project_name>` is the name of the xtUML project with spaces removed and all lower case. For example the package name for a project called "my project" would be "myproject". If you would like to change this default behavior, add the following to your `application.mark` file.

```
*,ApplicationPackage,*,<package_name>
```

```
ex:
*,ApplicationPackage,*,io.ciera.tool
```

where `<package_name>` is your custom package location.

# Root package

The root package is the package that is the translation unit for the compiler. This mark is required. If it is missing, no code will be generated. Add the following to your `application.mark` file.

```
*,RootPackage,*,<package_name>
```

```
ex:
*,RootPackage,*,MicrowaveOven::components
```

where `<package_name>` is the double colon delimited path to the xtUML package you want to translate (including the project name as the first path segment).

# Sort comparator

Ciera does not guarantee order of selected instance sets, however an attribute can be configured as the global sort comparator. For any selection, if the class has an attribute by that name, it will be used to sort the result set in ascending order. This is mostly useful for model compilers when you want elements to have a consistent order on each code generation. Ciera itself uses this feature to support all named elements by the attribute "name". Add the following to your `application.mark` file:

```
*,SortComparator,*,<attribute_getter>
```

```
ex:
*,SortComparator,*,getName
```

where <attribute_getter> is the name of the generated getter method for the attribute. Note that this mark can be configured to be any method on the generated classes — it could be configured to sort based on the return value of an instance operation.

## Simulated time

Ciera supports simulated time. For more detail about how simulated time affects application behavior, see Features: Simulated time. To enable simulated time for a component, add the following to your application.mark file:

```
<component_name>,EnableSimulatedTime,Component,true
```

```
ex:
components::MicrowaveOven,EnableSimulatedTime,Component,true
```

where <component_name> is the double colon delimited path to the xtUML component (not including the project name).

although this is a per component feature, Ciera does not yet support separate tasks for individual components. Therefore, if any component in a system is marked with simulated time, they will all operate in simulated time.

## Initialization function

Components can be marked with exactly one initialization function that will execute after the system is set up, but before any events, messages, or timers are handled. This is typically used to "kickstart" an application, however components do not necessarily need one if they are designed without any instance population setup needed or are configured at the time of the first external stimulus. To mark an initialization function, add the following to your application.mark file:

```
<component_name>,InitFunction,Component,<function_name>
```

```
ex:
components::MicrowaveOven,InitFunction,Component,init
```

where <component_name> is the double colon delimited path to the xtUML component (not including the project name), and <function_name> is the name of the xtUML domain function. The function name must be unique within the component — Ciera does not support domain functions with identical names even in different packages.

## Component version

Ciera provides a default scheme for versioning generated components when using the Maven build

plugin, however the version identifier can be overridden. To configure a custom version identifier, add the following to your `application.mark` file:

```
<component_name>,Version,Component,<version>
```

```
ex:
components::MicrowaveOven,InitFunction,Component,v1.0-pre-release
```

where `<component_name>` is the double colon delimited path to the xtUML component (not including the project name), and `<version>` is the custom version identifier.

Ciera will still include a generated timestamp with the version, even if the main version identifier is overridden

# Element exclusions

Ciera allows unused model elements (classes and associations) to be marked for exclusion. This can be useful if you are including a subsystem from another project as a package reference and are not using part of the model. See Features: Exclusions to read about the behavior and limitations of this feature. To configure exclusions, add the following to your `application.mark` file:

```
<path>,Exclude,Model Class,true
<path>,Exclude,Association,true
```

```
ex:
ooaofooa::Domain::Enumerator,Exclude,Model Class,true
ooaofooa::Domain::R20,Exclude,Association,true
```

where `<path>` is the double colon delimited path to the xtUML class or association (not including the project name).

# Key letters for generated class name

Ciera uses the modeled class name to generate the Java class name for each generated class. Spaces are removed and the first letter of each word is capitalized. If a class name is not suitable (e.g. it would clash with a Java reserved name), the key letters can be used as the generated class name. To confiugre a class to use the key letters as the name, add the following to your `application.mark` file:

```
<path>,UseKeyLettersForName,Model Class,true
```

```
ex:
architecture::statement::Break,UseKeyLettersForName,Model Class,true
```

where `<path>` is the double colon delimited path to the xtUML class (not including the project name).

## Port implementation class

Ciera allows users to provide their own implementations for ports. See Features: Custom IPort implementation to read about the behavior and limitations of this feature. To configure exclusions, add the following to your `application.mark` file:

```
<path>,BaseClass,Port,<class_name>
```

```
ex:
Tracking::Tracking::UI,BaseClass,Port,HttpPort
```

where `<path>` is the double colon delimited path to the xtUML port (not including the project name), and `<class_name>` is the name of the base class for generated ports.

The marked base class must either be a fully qualified class name or it must be defined in the same package into which the port will be generated.

## Other marks

Marks specifically related to instance loading, templating, and experimental features will be covered in Persistence, Templating, and Experimental Features respectively.

< next: Restrictions/Limitations | next: Build >

# Build

## Quick word about Maven

Ciera itself is built using Maven and it is used extensively to build other Ciera-based applications. Maven is an incredibly useful tool, but it can be frustrating if you do not understand it. I recommend taking a couple hours to read and learn the basics.

What is Maven?

## Requirements for a Ciera build

Ciera requires three things for a successful build:

1. An input model file with parsed OAL
2. A set of application marks (and feature specification)
3. An output location

The rest of the topics in this chapter will expound on how each of those three elements is configured and provided to the compiler.

## Pre-build

First, Ciera requires clean, parsed model data with all proxies resolved, in a single file. This has historically been a task handled for model compilers by the BridgePoint tool itself. Ciera supports pre-built output from BridgePoint.

Ciera also supports ouptut from the `pyxtuml` pre-builder. `pyxtuml` is a Python based dynamic xtUML tool used as the model backend by the `pyrsl` RSL generator. Details about `pyxtuml`, its author and its history can be seen in the pyxtuml documentation. Specific details about the pre-builder feature can be seen in the OAL prebuilder section. Using `pyxtuml` allows Ciera to be free of build dependencies on BridgePoint. Ciera projects can be built and executed entirely on a system with no BridgePoint installation, making it much easier to integrate into server builds.

The `pyxtuml` pre-builder is significantly faster for small-medium sized projects, since it does not suffer the weight of Eclipse, however for very large projects, the BridgePoint pre-builder is actually faster. It should also be noted that the BridgePoint pre-builder remains the gold standard implementation for OAL parsers.

The `pyxtuml` pre-builder is the preferred pre-build solution for Ciera projects because of its light weight and portability, however it does introduces an external dependency. `pyxtuml` must be installed on the system:

```
pip install pyxtuml
```

# Components of the `pom.xml` file

Let's take a look at the `pom.xml` file for the `MicrowaveOven` example:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>io.ciera</groupId>
  <artifactId>MicrowaveOven</artifactId>
  <packaging>jar</packaging>
  <version>1.0.0-SNAPSHOT</version>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
  <dependencies>
    <dependency>
      <groupId>io.ciera</groupId>
      <artifactId>runtime</artifactId>
      <version>2.0.0</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>io.ciera</groupId>
        <artifactId>ciera-maven-plugin</artifactId>
        <version>2.0.0</version>
        <executions>
          <execution>
            <id>pre-build</id>
            <goals>
              <goal>pyxtuml-pre-build</goal>
            </goals>
          </execution>
          <execution>
            <id>ciera-core</id>
            <goals>
              <goal>core</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
    <resources>
      <resource>
        <directory>${project.basedir}</directory>
```

```
        <filtering>true</filtering>
        <includes>
          <include>models/**/*.xtuml</include>
          <include>.project</include>
        </includes>
      </resource>
      <resource>
        <directory>${project.build.directory}/generated-sources/java</directory>
        <filtering>true</filtering>
        <includes>
          <include>**/*.properties</include>
        </includes>
      </resource>
    </resources>
  </build>
</project>
```

Let's break this down section by section:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>io.ciera</groupId>
  <artifactId>MicrowaveOven</artifactId>
  <packaging>jar</packaging>
  <version>1.0.0-SNAPSHOT</version>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
```

This is the basic setup for a Maven project. We have group and artifact identifers, packaging scheme, version identifier and some properties which define our character set and Java compiler version.

```
<dependencies>
  <dependency>
    <groupId>io.ciera</groupId>
    <artifactId>runtime</artifactId>
    <version>2.0.0</version>
  </dependency>
</dependencies>
```

Next we have our dependency section. All Ciera-based projects depend on the Ciera runtime library included here. In this case, it is the only dependency.

```
<build>
  <plugins>
    <plugin>
      <groupId>io.ciera</groupId>
      <artifactId>ciera-maven-plugin</artifactId>
      <version>2.0.0</version>
      <executions>
        <execution>
          <id>pre-build</id>
          <goals>
            <goal>pyxtuml-pre-build</goal>
          </goals>
        </execution>
        <execution>
          <id>ciera-core</id>
          <goals>
            <goal>core</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
```

The first part of the build section defines that this project uses the `ciera-maven-plugin`. This is the worker that actually handles the execution of the compiler. There are two execution units that we make use of: the pre-build and the core generation. The pre-build is done by pyxtuml and the core generation is done by the core model compiler. Other projects might also have executions for generating instance loaders/dumpers or template utilities. The pre-build section can be used to configure the BridgePoint pre-builder if that is preferred. See the `ciera-maven-plugin` in detail section for more on BridgePoint pre-builder.

```
<resources>
  <resource>
    <directory>${project.basedir}</directory>
    <filtering>true</filtering>
    <includes>
      <include>models/**/*.xtuml</include>
      <include>.project</include>
    </includes>
  </resource>
```

This section is standard for Ciera-based projects. It simply indicates to maven that all `.xtuml` files should be packaged into the output artifact (JAR). This allows Ciera-based xtUML "library" projects to be distributed as dependencies. See the next section for detail.

```
      <resource>
        <directory>${project.build.directory}/generated-sources/java</directory>
        <filtering>true</filtering>
        <includes>
          <include>**/*.properties</include>
        </includes>
      </resource>
    </resources>
  </build>
</project>
```

This final section is also standard for Ciera-based projects. It assues that `.properties` files are included as resources in the output artifact. This is used to store the version information for components.

# Ciera dependency strategy

Ciera leans into the dependency mechanism of Maven and therefore utilizes the Maven dependency mechanism to specify other xtUML projects that need to be included for inter-project references. A major problem with including other projects is dealing with fragile filesystem relative paths to locate model elements. Eclipse solves this using their workspace model (virtual filesystem) to bring all imported projects together. Ciera needs to be independent of BridgePoint and Eclipse.

The `ciera-maven-plugin` automatically invokes the pyxtuml pre-builder and passes the path to artifacts listed as Maven dependencies. If an xtUML project was built and installed in the local Maven repository or exists on an accessible remote repository, it can be accessed and passed directly to pre-builder.

Consider the GPS Watch example:

```
<dependencies>
  <dependency>
    <groupId>io.ciera</groupId>
    <artifactId>runtime</artifactId>
    <version>2.0.0</version>
  </dependency>
  <dependency>
    <groupId>io.ciera</groupId>
    <artifactId>HeartRateMonitor</artifactId>
    <version>1.0.0-SNAPSHOT</version>
  </dependency>
  <dependency>
    <groupId>io.ciera</groupId>
    <artifactId>Location</artifactId>
    <version>1.0.0-SNAPSHOT</version>
  </dependency>
  <dependency>
    <groupId>io.ciera</groupId>
    <artifactId>Tracking</artifactId>
    <version>1.0.0-SNAPSHOT</version>
  </dependency>
  <dependency>
    <groupId>io.ciera</groupId>
    <artifactId>UI</artifactId>
    <version>1.0.0-SNAPSHOT</version>
  </dependency>
  <dependency>
    <groupId>com.googlecode.lanterna</groupId>
    <artifactId>lanterna</artifactId>
    <version>3.0.1</version>
  </dependency>
</dependencies>
```

The GPS Watch example is comprised of five separate projects. Each has its own `pom.xml` in which its `.xtuml` files are zipped up in the output JAR. The system deployment project that is translated with Ciera declares Maven dependencies on each of the four "library" projects and all of their xtUML modeled elements are pulled in automatically to the pre-build.

I see a future in which a widely used data model like the `mcooa` project is built with Maven and published and model compiler projects can easily access it in this way without even needing the project on their machine much less in their development folder.

(I see a further future where Ciera can reuse compiled implementations of components directly without re-translation, but that is not a reality now.)

## `ciera-maven-plugin` in detail

Each of the "goals" the `ciera-maven-plugin` provides are documented below.

### pyxtuml-pre-build

The pyxtuml pre-build goal takes input files and executes the pyxtuml pre-build utility to produce clean SQL input for a Ciera compiler. It has the following configuration parameters:

#### outputFile

The location of the output SQL file. The default value is <project_name>.sql in the project build directory (target/ is the default build directory for Maven, but can be configured differently in the main build section of the pom.xml file).

#### modelDirs (array)

This list of directories to look for input models. This option is typically only used to specify model dependencies by path instead of included as a dependency (described above). This has some benefit in flexibility especially of required projects are not built with Maven, but suffers in its reliance on consistent filesystem paths. This method of including dependency models is not recommended.

#### includeDependencyModels

If "true", models dependency artifacts are searched for model files. This is typically false if "modelDirs" is used. Default value is "true".

#### includeLocalModel

If "true", the models/ directory in the current project is searched for model files. This is typically false only if "modelDirs" is being used. Default value is "true".

#### pythonExecutable

Specifies the name/path of the Python executable to use for pyxtuml. The default value is "python" which will execute the default Python interpreter installed on the system. This value can be changed to use an alternate interpreter. For example, the Ciera projects themselves use the pypy interpreter because it parses the OAL almost three times faster than standard CPython.

If you use a different Python interpreter, you may need to install pyxtuml again for that specific interpreter (e.g. pip3 install pyxtuml for python3)

#### Example

In addition to the MicrowaveOven example given above, consider the following pom.xml snippet from the Ciera core model compiler.

```
<execution>
  <id>pre-build</id>
  <goals>
    <goal>pyxtuml-pre-build</goal>
  </goals>
  <configuration>
    <includeDependencyModels>false</includeDependencyModels>
    <modelDirs>
      <param>${project.basedir}/../runtime/models</param>
      <param>${bpLoc}/src/org.xtuml.bp.ui.marking/models</param>
      <param>${mcLoc}/model/mcooa/models</param>
    </modelDirs>
  </configuration>
</execution>
```

In this example, models are not included from the dependency list, but paths are specified in the configuration of the pre-build itself.

This is done because Ciera is built with itself, and as such, the Ciera runtime library it requires as a runtime dependency is not the same as the one it needs as a build dependency (self-building compilers are confusing!).

### bridgepoint-pre-build

The BridgePoint pre-build goal uses the BridgePoint CLI to pre-build a project in a pre-defined Eclipse workspace. It requires that BridgePoint be installed and a workspace be set up, however it does not require a window manager (it can still be part of a server build).

#### bpHome

The location of the BridgePoint installation. If no value for "bpHome" is specified, the build will fail.

#### workspace

The location of the BridgePoint workspace where the models are imported. The name of the Maven project is used to determine which project in the workspace to build. This implies that the name of the Maven project and the name of the BridgePoint project *must* match. If no value for "workspace" is specified, the build will fail. All models required for the build must be imported into the workspace including the Ciera runtime library. An easy way to check this is to run a "parse all" in the BridgePoint workspace and verify that there are no parse errors due to missing model elements.

#### Environment variables

The previous two options can be specified by environment variables BPHOME and WORKSPACE respecitively. This can be useful if you are working on multiple Ciera projects in the same workspace or if you prefer not to couple development workspace and tool paths with your project source code.

#### Example

```
<execution>
  <id>pre-build</id>
  <goals>
    <goal>bridgepoint-pre-build</goal>
  </goals>
  <configuration>
    <bpHome>/Users/levi/xtuml/m6190.2019-12-18-1004_nightly-
build/BridgePoint.app/Contents/Eclipse</bpHome>

<workspace>/var/folders/6n/ybm_82hn3wq4w972zjl90q9w0000gp/T/tmp.issCz64J</workspace>
  </configuration>
</execution>
```

Since BridgePoint manages the acutal pre-build, the location of the output model file is determined by BridgePoint. You will have to refer to the BridgePoint project to confirm what this location is. As of this writing, BridgePoint outputs pre-built model files to `gen/code_generation/<project_name>.sql` where `<project_name>` is the name of the BridgePoint project.

### `core`

The core goal is the main code generation tool. For most Ciera projects, one of the pre-build goals and the core goal are all that is necessary to build the project.

#### input

The location of the input pre-built model file for translation. The default value is the same as the default outputFile of the `pyxtuml` pre-build goal.

#### output

The location of an output file where instances of OOA of OOA and Ciera architectural models will be dumped. The default value is an empty string (which causes the compiler to skip dumping output). This option is used if there is a downstream compiler that you need to load the instance population to generate additional Java source files.

#### genDir

The location where the generated Java source will be output. The default value is `generated-sources/java` in the project build directory.

#### Example

```
<execution>
  <id>ciera-core</id>
  <goals>
    <goal>core</goal>
  </goals>
</execution>
```

### `sql`

The SQL goal generates a SQL insert statement loader/dumper for the model. For more information, see the Persistence chapter. The configuration parameters are identical to the `core` goal.

### `template`

The template goal parses a set of RSL templates and generates a template registry. It also processes RSL substitutions in literal strings within OAL. For more information, see the Templating chapter. The configuration parameters are identical to the `core` goal.

## Building without Maven

Although it is not recommended, Ciera is not strictly dependent on Maven. This section will demonstrate step by step how to download the Ciera artifacts, build and run the MicrowaveOven example project.

### Download the artifacts

The Ciera runtime library and core generation tool are needed. Download directly from Maven central with:

```
wget https://search.maven.org/remotecontent?filepath=io/ciera/runtime/2.0.0/runtime-
2.0.0.jar -O runtime.jar
wget https://search.maven.org/remotecontent?filepath=io/ciera/tool-core/2.0.0/tool-
core-2.0.0.jar -O core.jar
wget https://search.maven.org/remotecontent?filepath=org/antlr/antlr4-
runtime/4.7.1/antlr4-runtime-4.7.1.jar -O antlr.jar
```

### Pre-build

Pre-build the model with:

```
python -m bridgepoint.prebuild -o MicrowaveOven.sql runtime.jar models/
```

Alternatively, import the project into a BridgePoint workspace and import existing projects from the `runtime.jar` archive and run a pre-build within BridgePoint.

### Generate code

Generate Java source with:

```
mkdir src-gen
java -cp runtime.jar:antlr.jar:core.jar io.ciera.tool.CoreTool -i MicrowaveOven.sql
--gendir src-gen --cwd .
```

### Compile the Java code

Compile Java with:

```
find src-gen/ -name *.java > sources.txt
javac -cp runtime.jar -d bin @sources.txt
```

### Run the application

Run with:

```
java -cp runtime.jar:bin microwaveoven.MicrowaveOvenApplication
```

### Ciera core generator CLI

As demonstrated above, it is possible to use Ciera without Maven — in fact, the Ciera compiler itself is only a small piece in a longer build chain including pre-build, generation, compilation, and execution. The code generator has its own command line interface and Maven simply maps configuration from the `pom.xml` file to existing CLI options.

The following is the output of passing `-h` to the Ciera tool:

```
$ java -cp runtime.jar:antlr.jar:core.jar io.ciera.tool.CoreTool -h
Usage:
  --cwd <root_dir>     : base working directory
  --gendir <gen_dir>   : generated output directory
  -i <input_file>      : input file
  -o <output_file>     : output file
  --use-version <use_version> : version identifier for generated components
  -h, --help           : Print usage information.
```

# Running projects

When running projects generated with Ciera, in addition to the the compiled Java classes, some libraries must be in the classpath.

- For all Ciera generated projects, the Ciera runtime library must be in the classpath. In general,

the runtime library must have the same major version as the version of the code generation tool used to generate the code (for more details on the Maven versioning policy, see Maven version number policy).

- For projects using JSON serialization for message passing, the JSON library found here is required.
- For projects using SQL instance loading, the Antlr 4.7.1 runtime library is required.
- Any other external libraries referenced in hand written code must be on the classpath.

If you are using Maven as your build tool, all of these dependencies will be downloaded in your local Maven repository which is a nice consistent place to reference them. The Ciera example projects each have a "run" bash script which builds the classpaths from the JARs installed in the local Maven repository.

# Using the Ciera "nightly build"

Release versions of Ciera will be published to the Maven central repository. This is the default repository for Maven and artifacts hosted here will be downloaded automatically with no extra configuration.

Snapshot versions of Ciera (i.e. development versions/nightly builds) are hosted in the Sonatype snapshot repository. To access development versions of Ciera, you must add the following `<repository>` and `<pluginRepository>` definitions in your project `pom.xml` or in your Maven `settings.xml`. More information on declaring additional repositories can be found here.

```
<repository>
  <id>snapshot-repo</id>
  <url>http://oss.sonatype.org/content/repositories/snapshots</url>
  <releases><enabled>false</enabled></releases>
  <snapshots><enabled>true</enabled></snapshots>
</repository>
```

```
<pluginRepository>
  <id>snapshot-repo</id>
  <url>http://oss.sonatype.org/content/repositories/snapshots</url>
  <releases><enabled>false</enabled></releases>
  <snapshots><enabled>true</enabled></snapshots>
</pluginRepository>
```

< prev: Marking | next: Persistence >

# Persistence

Ciera was built as a model compiler to build model compilers. Therefore one of the key design requirements was for it to be able to load and dump instances of the xtUML meta-model in the BridgePoint persistence format (SQL). Ciera has a built-in SQL loader/dumper generator, but also supports other types of loaders.

# SQL loader/dumper

Projects can be generated with a SQL insert statement loader/dumper. Instances can be loaded from standard input or a file or set of files and be dumped to standard output or a file.

The loader/dumper can serialize and reload entire instance populations including stateful classes. In flight events and pending timers can be serialized and re-loaded.

## Enabling the SQL loader/dumper

To enable the SQL tool, first add the `InstanceLoading` mark with "Sql" as the value. See Instance loading for mark details. Next add the `sql` tool as an execution step to the project `pom.xml`. You will need to define an ouptut location for the core code generator so the instances can be loaded by the `sql` tool to generate the loader/dumper. This file can be anything, but simply serves as an intermediate landing location between the two code generators:

```
<execution>
  <id>ciera-core</id>
  <goals>
    <goal>core</goal>
  </goals>
  <configuration>
    <output>${project.build.directory}/b.xtuml</output>
  </configuration>
</execution>
<execution>
  <id>ciera-sql</id>
  <goals>
    <goal>sql</goal>
  </goals>
  <configuration>
    <input>${project.build.directory}/b.xtuml</input>
  </configuration>
</execution>
```

## The `SQL` external entity

To use the SQL loader/dumper, you must make calls to the SQL external entity.

```
SQL::load();
```

will load instances from standard input.

```
SQL::serialize();
```

will write the instance population to standard output. More detailed documentation can be found in the API docs

### Limitations

#### Types

The SQL loader/dumper tool can only generate loaders and dumpers for class models with attributes of the core types and enumeration types. User defined types and structured types are currently not supported, however attributes typed with user defined types can be marked as non-persistent and therefore the instance population can still be loaded excluding those attributes.

#### Formalized associations

The SQL loader/dumper tool only supports class models with all formalized associations. Supporting unformalized relationships is a goal for the future, however it may never be part of the SQL loader/dumper and may be part of a different loader/dumper utility.

#### Complete population

Although multiple files can be loaded, the loader expects all related instances to be loaded at once. Because of this, multiple file support is not as useful because the populations held by the individual files would have to be disjoint.

Similarly, serialization only supports dumping the entire population at once (to standard output or a file).

## Generic loader interface

Ciera provides an API for writing extremely customizable population loaders. This feature is intended to support any type of loaders, but especially loaders based on parsing a natural modeling language.

The API consists of two parts. An external entity call that can be executed in an OAL action to hook into a hand written Java class and pass an argument list, and a set of bridges that allow hand written code to create and manipulate instances from hand written code.

The details of this API can be found in the LOAD external entity API docs

# Support for other loader/dumpers

Ciera has been written in a way to support implementation of other types of loaders in the future. Ciera could even support multiple types of instance loaders for a single project.

# Persistence related marks

## Instance loading

The instance loading mark registers an instance loader for a component. For SQL instance loading, this is required with the value "Sql"

```
<component_name>,InstanceLoading,Component,<loader_id>
```

```
ex:
pei::pei,InstanceLoading,Component,Sql
```

where `<component_name>` is the double colon delimited path to the xtUML component (not including the project name), and `<loader_id>` is the identifier of the specific instance loader/dumper. Currently the only supported value is "Sql".

## Non persistent instance IDs

By default, SQL loader/dumpers will be generated to load and dump architectural instance IDs. The architectural IDs are required to load instance populations with pending timers and in flight events. To disable architectural ID persistence, add the following mark:

```
*,NonPersistentInstanceIds,*,true
```

## Non persistent elements

To exclude an attribute or class from persistence, add the following mark:

```
<path>,NonPersistent,Attribute,<exclusion_type>
<path>,NonPersistent,Model Class,<exclusion_type>
```

```
ex:
ooaofooa::Instance::Timer::expiration,NonPersistent,Attribute,load_only
ooaofmarking::Markable Element Type,NonPersistent,Model Class,true
```

where `<path>` is the double colon delimited path to the xtUML class or attribute (not including the project name), and `<exclusion_type>` determines how it is excluded. For classes "true" is the only

acceptable value. For attributes, "true" causes the attribute to be completely excluded from the schema. "load_only" keeps the attribute in the schema, but excludes it from load. At load time, the attribute will be assigned the default value of its type and at serialization, the attribute's value will be serialized normally.

< prev: Build | next: Templating >

# Templating

Another design requirement for Ciera as a model compiler tool was to support RSL templates. RSL is a scripting/templating language that has been used for many years to build interpreted model compilers. `pyrsl` appeared several years ago as a modern interpreter for the language

The Ciera implementation of RSL templating parses the template files at build time and generates a "template registry" of Java methods that take a set of symbols as input and produce a string as output.

## RSL template tool

### Enabling the template tool

To enable the SQL tool, first add the `TemplateDir` mark with the path to your templates as the value. See Template directory for mark details. Next add the `template` tool as an execution step to the project `pom.xml`. You will need to define an ouptut location for the core code generator so the instances can be loaded by the `template` tool to generate the template registry. This file can be anything, but simply serves as an intermediate landing location between the two code generators:

```
<execution>
  <id>ciera-core</id>
  <goals>
    <goal>core</goal>
  </goals>
  <configuration>
    <output>${project.build.directory}/b.xtuml</output>
  </configuration>
</execution>
<execution>
  <id>ciera-template</id>
  <goals>
    <goal>template</goal>
  </goals>
  <configuration>
    <input>${project.build.directory}/b.xtuml</input>
  </configuration>
</execution>
```

### The `T` external entity

To use templates, you must make calls to the T external entity. Set the output directory with

```
T::set_output_directory(dir:"folder");
```

Include a template with

```
T::include(file:"t.mytemplate.java");
```

More detailed documentation can be found in the API docs

## Limitations

### RSL support

The Ciera template implementation does not support all of RSL.

Ciera supports the following:

- buffers
  - blobs of text
  - substitution variables
  - escaped characters (as defined in `pyrsl`)
- format characters in substitution variables
- if/elif/else statements
- all expressions (conditional expressions for if statements)
- literals
  - boolean
  - integer
  - real
  - string
- substitution variables in string literals

Ciera does not support the following:

- select statements
- create/delete statements
- relate statements
- variables
- for/while loops
- fragments
- etc, etc, etc...

Ciera only supports the core *templating* constructs and not any of the *query* constructs.

You can decide for yourself whether or not this is a limitation or a feature — it forces users to write true templates. There will be no future plan to implement the rest of the RSL specification in Ciera.

# Templating marks

## Template directory

The location of your template files must be declared through a mark. Any invocations to `T::include` in OAL will be relative paths to this directory. If this mark is not included, the build will fail. Configure the mark as follows:

```
*,TemplateDir,*,<location>
```

```
ex:
*,TemplateDir,*,templates
```

where `<location>` is the path to the directory where the templates are located.

< prev: Persistence | next: Experimental Features >

# Experimental Features

There are a handful of experimental/incomplete features lingering around Ciera. Most of these had to do with a demo produced in 2018 where Ciera was used to generate code for the AWS cloud architecture. Activities were run in AWS Lambdas, DynamoDB was used for instance population persistence, and AWS API Gateway was used to plumb components together through HTTP routes.

The demo presentation can be found here and demo video here

This chapter will be brief as there is no point in detailed documentation on features that don't work — the main point is to know that they exist.

## Asynchronous applications

Partial support for generating asynchronous applications is available through a mark. Typically an xtUML architecture will have some sort of long running dispatch loop (whether it is system-wide or more granular) that handles incoming stimuli (port messages and delayed events). Each time an outside stimulus is handled, the resulting generated events are handled until the system quiesces.

An async application responds to these outside stimuli by dynamically loading to handle a single request and then terminating after the transaction is completed instead of being a long running process. Thousands of instances of these applications could be deployed in a distributed architecture and perfectly respond to fluctuations in system load without any overhead during quiet times.

To turn on generation of async applications, configure the following mark

```
*,AsyncApplication,*,true
```

## Amazon DynamoDB instance loader/dumper

A basic implementation of a DynamoDB instance loader/dumper was implemented. It was not fast and it had many bugs. It was essentially a port of the SQL instance loader/dumper with one key difference. It was transactional, meaning each serialization only dumped the delta between the final instance population and what was loaded at the beginning.

This project has not been maintained, but the old source code can be found in `src-other` at the root of the main repository.

## HTTP endpoints

Another feature related to distributed asynchronous applications was HTTP endpoints. This mark allowed ports to specify an HTTP endpoint in lieu of a satisfaction with another port. Outbound messages would be serialized to JSON format and sent to the marked HTTP endpoint where they would be routed to a component to handle them.

To enable this feature for a port, add the following mark:

```
<path>,HttpEndpoint,Port,<url>
```

```
ex.
Tracking::Tracking::LOC,HttpEndpoint,Port,https://7t6vbnkhn4.execute-api.us-east-
1.amazonaws.com/test
```

# Ciera Maven archetype

A non-AWS related experimental feature is the Ciera Maven archetype project. Maven provides a mechanism for generating "template" projects to help users get started with a plugin or other tool. This would allow a user to run a maven command, enter a few parameters and have a new Ciera project set up and ready to build without having to mess with the dirty details of the `pom.xml` file all the time (or more likely, just copy and paste from an existing project!)

I do intend to finish this project, however, it is not ready at the moment. The existing source can be found in `src-other/maven/ciera-maven-archetype` at the root of the main repository.

< next: Templating | >