JChassis User Guide

Sam Stainsby

Revision History

Revision version 0.1	22nd May, 2003
Revision version 0.2	20th March, 2004

Table of Contents

Introduction

JChassis is a component system for developing and maintaining Java applications. JChassis is alos a collection of useful components that can be used in Java applications. This document explains how to develop Java applications using JChassis and how to create your own JChassis components. If you are unfamiliar with the basic aims of JChassis, or just want a quick overview ("executive summary"), you should consult the document *JChassis Factsheet* first.

Prerequisites

You must know how to write simple Java programs to understand this guide. Some familiarity with XML is also needed. It will be an advantage if you can understand XML DTDs as well. You should probably read *JChassis Factsheet* first for a quick overview of what JChassis is about.

A note on scripts

The examples in this document that run scripts in the JChassis SDK are shown for the Linux operating system. You will find that there are similarly named batch files that you can use under Windows NT or XP.

The Basics

This section covers the basic concepts of the JChassis system, including how to compile and run the "Hello World" example application.

What's a component?

The term *component* in the software world can mean many different things to different people. Classes, packages, applications, JavaBeans, J2EE Entity Beans, whole APIs, etc. can all be thought of as components in some respect. The most useful components are those that have simple, cleanly defined interfaces with a "natural" interpretation in the domain they are used in.

Components are more useful too if they can deployed easily. Ideally, they are independent of other components, which means that moving one doesn't require taking a whole bunch of other components with it. In reality, components depend on other components — this is a good thing because components that don't depend on other ones are usually pretty boring. A good component though will have relatively few dependencies on other components.

A good collection of components will be "loosely coupled": the dependencies will be few and will usually be dependencies on abstract interfaces rather than internal implementations of other components. This second property — dependency on abstract interfaces — allows many alternative components to be slotted in where a particular interface is required by another component, as long as each of those alternative components supports the required interface. That approach gives rise to flexible, reusable systems of components: which is exactly what JChassis tries to achieve.

What's a context?

If you are already familiar with the JavaBeans BeanContext API, then you probably don't need to read this section, since you'll have a basic understanding of what contexts are used for. One thing to note before you skip on to the next bit is that JChassis uses its own context API and does not use Sun's more complex BeanContext API. The reason: JChassis' context API is much simpler and places fewer constraints on what can be a component. The JavaBeans Context API is also not supported on all of the platforms that JChassis can be deployed on. See "What about the JavaBean APIs?" in the appendices for more details.

A context, or more accurately, a *service context*, is a software entity that provides a number of services

to the components contained within it. A services is small, simple, highly intuitive interface. This is exactly what a good component should provided in the interfaces that it presents.

In JChassis, components provide services, through interfaces, that other components can choose to use. Sometime, components *require* the presence of certain services before that can operate. So components are put into contexts, and contexts in turn provide a convenient point for other components to find the services of that component. Those other components known little about the component that implements the service though.

Maybe an example will help. We'll consider components formed from just one Java class for simplicity. Let's suppose **Log** is a simple Java interface with one method **log(String message)**. We could make two components that implement this interface. Lets call the first one **LogA** and the other **LogB**. Perhaps one of these components logs to a file, where the other one logs to the console. Another component, **MyComponent**, wants to use a **Log** service for logging what is does. We place **LogA** in the same context as **MyComponent**. When **MyComponent** asks its context for a **Log** service, it gets back an instance of **LogA.** If we had placed **LogB** into the context instead of **LogA**, then **MyComponent** would get an instance of **LogB**. Note that **MyComponent** doesn't depend on which type of service it gets, **LogA** or **LogB** — it just knows that it gets a **Log** service of some kind. This allows anyone to write their own **Log** service and slot it in, without having to change **MyComponent**.

You might say to the above example "well isn't this just what the Java **interface** mechanism is used for anyway?". Well, yes ... and no. Yes, we are taking advantage of the usefulness of Java interfaces to provide flexibility and extensibility. In addition though, contexts are useful because they limit the scope of where an implementation is available, and aggregate the services provided by components into a convenient form, accessed through the context's API, as we shall see.

The "Hello World" Example

Let's look at an example straight away to clarify what was discussed so far.

In your SDK's **examples** directory, you can build and run all of the examples. List the contents of that directory and you will see a subdirectory for each example program that comes with the SDK.

To build an example, change to the **examples** subdirectory. Before attempting to build anything, you must edit **build_env** to suit your environment. Make sure you backup the original copy of **build_env** before you do this. The important variables are **JAVA_HOME** and **ANT_HOME.** You must have Java and Apache Ant installed somewhere. Java version 1.1 or above will be sufficient for this example.

Once **build_env** has been edited to your satisfaction, give the command:

```
> ./build -Dexample=hello_world_basic run
```

Note that the **example** property tells the build system which example to build. You should see "Hello World" appear in the output. Most errors occur at this point due to incorrect **JAVA_HOME** and **ANT_HOME** settings in the **build_env** file. Please check those values first.

So what have we done so far? Looking at the **Main.java** code in the **src** subdirectory of the **hello_world_basic** example, "not much" would appear to be the answer. The **Main** class extends something called **Application**: a class that is a convenient starting point for JChassis applications. We'll talk more about that later. Though it might not look like it, the **Main** instance is actually a trivial JChassis service in our example application. It uses the **Log** instance by getting it from the service context, using the **getService** method (in fact, **Application** has its own convenient **getService** method that avoids getting the service context each time, but using that would be less instructive). The **Log** instance that it retrieves is an instance of another service. So where is the context that these components live in configured? The answer to this question is found in the **services.xml** file in the **hello_world_basic** directory ...

Have a look at the **services.xml** file. You'll see that there's a **context** element there that contains two instances: the **Main** instance that runs the show, and a **Log** instance that is used for output. The syntax and semantics of this XML format is described in the **services.dtd** file in the doc/dtd/0.1 subdirectory of the JChassis SDK distribution — don't bother looking at that yet though unless you are particularly curious. Just understand that the **services.xml** file configures a single context that contains a **Main** service and another service that is responsible for logging. The implementation **Main** is declared to implement the **Application** interface. The **main** method in the **Application** superclass takes care of calling the **run** method on **Main**. The **Log** interface is implemented by **Log** (in the **org.jchassis.log.impl** package this time) — the default logging service that by default outputs to **stdout** (it can also be configured to output to a named file or to **stderr** if required — see "Configuring services" later).

Since **Main** only uses one other component, this example is a bit boring. That's not to say that there aren't other components at work here. Have a look in the directory **libs** created under **/tmp/jchassis/$USER/examples/hello_world_basic/** (where $USER is your username) by the build system. You'll see a bunch of files suffixed by "**.mar**". These are MAR files, which are just the usual JAR files with some extra information added into the metadata. Each MAR file represents a JChassis component, though many do not export services and are simply convenient vehicles for transporting code. These components are as used by the JChassis "basic" framework. For example, some of them are used for processing XML under Java 1.1. There seems to be a lot of files there: when using the alternative "core" framework, there are quite few less components required to run a "Hello World" program, although this comes at a cost in terms of developer convenience. We'll talk more about the alternative JChassis core framework and MAR files later on.

Note that we could take out that default Log implementation and replace it with a different **Log** implementation (say one that appends log entries onto a database table, maybe called **DbLog**, that implements the **Log** interface) and **Main** would be none the wiser. More importantly, we could do this by simply changing the XML configuration and replacing the appropriate MAR file, *without the need to modify or recompile the application code*. This is one of the key advantages of the JChassis approach.

Configuring Services

We have mentioned before that services can be configured. In this section we show you how to not only place services in a context, but to configure those services as well.

Make a backup copy of your **services.xml** file in the **hello_world_basic** directory. Now modify **services.xml** thus: add the following XML into the **Log instance** element, directly under the end tag for the Log **implementation** element.

```
<property>
    <name>logFileName</name>
    <value>mylog</value>
</property>
```

What this snippet does is set a property on the **Log** instance so that when it is created, the property **logFileName** is set to **mylog**. Run the example and you will see that no output occurs to the console. Instead you will find a file called **mylog** in your directory that contains the text "Hello World".

If this worked, "Congratulations!" (if not, you could check the validity of your XML against the DTD to see where you went wrong). You have just modified the application behaviour through configuration, again *avoiding the need to modify or recompile source code*.

In Summary

We hope you now have a bit of an idea of what components and contexts and their configuration are all about. If not, the example code in the following sections might help you gain a clearer understanding.

Creating JChassis applications

In this section, we describe how to write your own JChassis applications using the JChassis "basic" framework. Writing a non-trivial JChassis application invariably requires not only using some JChassis components but writing some of your own as well. If not, then you are probably missing the point: component-based programming is a more flexible and extensible way of developing your applications. Writing your own JChassis components is covered in the "Creating JChassis Components" section.

This section assumes that you know how to write simple Java applications. If not, consult the documentation that came with your Java Development Kit or on the Sun Microsystems Java Web Site [http://java.sun.com/].

The essentials

What are the essential elements needed by a JChassis application? JChassis applications can use either of two frameworks: the *core* framework or the *basic* framework. The core framework is for applications that need to operate in very constrained environments and require a minimal code size (the core framework add an overhead about 15KB) and is not covered until later (see "The core framework"). The basic framework is not quite as compact (it's overhead is about 60KB), but you will find it much more compact than other frameworks such as NetBeans Platform [http://www.netbeans.org/products/platform/] for example, which can add megabytes to your code size. The basic framework actually uses the core framework to bootstrap the core services that it requires.

To use the basic framework, you will need a certain set of modules (in the form of MAR files) in the classpath of your application. Looking at the **depends.properties** file in **hello_world_basic** example application directory, you will see what MAR files it requires. The **depends.properties** file is used by the build system for the examples, and is not needed for general JChassis applications. These are necessary for all basic framework applications, except for the MAR files whose names are prefixed by **jc_log_** — these are only needed if you intend to use a JChassis logging service. See the *JChassis Module Guide* for more information on what each module does. The MAR files for these and other modules in the SDK are found in the **modules** directory of the JChassis SDK.

For the basic framework, you will also require a **services.properties** file and a **services.xml** file. We mentioned that the basic framework uses the core framework: this is where the **services.properties** file comes in. The **services.properties** file contain the service configuration for the core framework, in much the same way the **services.xml** file contains the service configuration for the basic framework. The difference is that the core framework configuration is much more limited. For example, you cannot set properties on a service in the **services.properties** file. For our purposes, simply using a copy of the "Hello World" **services.properties** file is sufficient — you don't need to know what it does at this point. While you're at it, you'll need a **services.xml** file which you may as well copy from there as well and then modify to your desired configuration. Both configuration files should be in the classpath of your application.

The name and location of the **services.xml** file can be changed by setting the system property **org.jchassis.basic.config**, which is the resource name (as in a class's **getResource** method) of the XML file.

You now have the essentials to build a JChassis application. We recommend using Apache Ant [http://ant.apache.org/] for your build system. You can look at some code from the **build.xml** file in the **examples** directory to get some ideas, such as the **copylibs** target that can be used to automatically copy the MAR files you need into your build area.

Configuring a service context

We have seen in the section "The Basics" how the **services.xml** file is used to configure a service context using the **context** element. In fact, contexts can be nested in the same XML file by nesting **context** elements, but that is not a major concern for simple applications. Other entities that can occur within a

context are *instances* and *factories*. We have already seen **instance** elements in "The Basics". The root element of a **services.xml** file is always a **services** element.

An *instance* is a *singleton* service: only one instance of that service every exists within its enclosing context. This instance is created when the context is initialised. Each time the context is asked for that service, the same instance of the service implementation class is returned. This is ideal for services such as a log or printer service that represents a single object. Instances are represented by **instance** elements in the **services.xml** file.

A *factory* is a *source* of service instances. Services are only created by factories when they are requested from a service context. This construction is useful for services that are required on demand. For example, suppose we had a **ChatRoom** service: we would require an new instance for each new chat room that we created. Factories are represented by **factory** elements in the **services.xml** file.

Normally, only **context** elements are added under the **services** element. However, it is possible to add **instance** elements before the first **context** element. The instances are added to the *root* service context: to top-most service context, which in this case is provided by the JChassis core framework. Certain constraints apply to these instances because of the limitations in the core framework — see "The core framework" later.

Note that the DTD is very specific about the order of the children of a **context** element. The order is: **metadata**, **factory**, **instance**, and then nested **context** elements. So far we haven't mentioned **metadata** elements. These are discussed later.

We mentioned above that service contexts can be nested. What effect does this have? In general, it works like this: if a specified service cannot be found in a certain context, then its parent context is searched, and so on recursively through the context hierarchy until the service is found. In a sense, default services in higher-level contexts can be "overridden" by descendant contexts. This means we can have default services in the top-level context and more specialised services in lower-level (descendant) contexts, that might be used for different parts of your application. Most likely you will not need nested contexts for simple applications.

Configuring service instances

A *service instance* is a single instance of a service implementation class that provides a service to the users of a context. It is essential to specify a service interface class and a service implementation class as a minimum. This is done with **interface** and **implementation** elements inside each **instance** element in the **services.xml** file. The **interface** element must come first. The fully qualified class names are entered as text between the start and end tags.

For example:

```
<instance>
    <interface>MyService</interface>
    <implementation>MyServiceImpl</implementation>
</instance>
```

specifies that the service **MyService** is implemented by the **MyServiceImpl** class.

Once an instance is configured, the context can be asked for the service by the name of the interface, and the context in return will supply the corresponding implementation instance. Following the example above, an instance of **MyServiceImpl** will be created when the enclosing context is initialised, and that instance will be given out to users that request the **MyService** service.

Service instances are created when the context is initialised, not when the service is requested. To delay instantiation to request time, you will need to use a *service factory* (see below).

Configuring service factories

A *service factory* describes how a context will produce instances of a service. As with an **instance** element, a **factory** element can specify a service interface class and a service implementation class (although this is not the only way, as we'll see). This is done with **interface** and **implementation** elements inside a **factory** element in the **services.xml** file. Once this is done, the context can be asked for a service by the name of the interface, and the context in return will supply an instance of the corresponding implementation.

For example:

```
<factory>
    <interface>MyService</interface>
    <implementation>MyServiceImpl</implementation>
</factory>
```

specifies that the service **MyService** is implemented by the **MyServiceImpl** class.

Once a factory is configured, the context can be asked for a service by the name of the interface, and the context in return will supply an instance of the corresponding implementation. This may be the same instance each time or a different one as we'll discuss below. The important point is that the first instance is created at the time the services is first requested, not when the context is initialised.

Service factories are *singleton* multiplicity by default: they produce only one instance, which is cached and returned each time the service is requested from the context. Factories can also be *on-demand*, which means that a new instance is returned each time the service is requested. The **multiplicity** attribute of the **factory** element controls this behaviour: it can take the values **singleton** or **on-demand**. If the **multiplicity** attribute is omitted, the default value is, as mentioned above, **singleton**.

Configuring service properties

Just creating a service instance is pretty boring really. What we need is some way to configure that instance, to provide us with some flexibility. JChassis' services XML format provides a way to do this. It allows you to configure most JavaBean-style properties on a service implementation.

You might already know what a JavaBean property is. If not, in a nutshell, a JavaBean property is defined by a naming conventions for the class' "getters" and "setters" (**accessors** and **mutators** to be precise). For example, a **Log** implementation class might have methods **String getLogFileName** and **setLogFileName(String name)**. This means that **logFileName** is a valid property for that bean, and furthermore has type **String**. Actually, for our purposes, only the "setter" method needs to be present. Hopefully you see the method naming pattern here, otherwise see Sun Microsystems' Java web site [http://java.sun.com/] to find out more about JavaBeans and their properties.

The **property** element is used to configure the properties of factories and instances. For example, continuing our **instance** example above,

```
<instance>
    <interface>MyService</interface>
    <implementation>MyServiceImpl</implementation>
    <property>
        <name>myProperty</name><value>ABC123</value>
    </property>
</instance>
```

would ensure that the **setMyProperty** method on the instance of **MyServiceImpl** was called with the argument **"ABC123"** immediately after it was created, but before it was handed to the user requesting the **MyService** service. The same is kind of configuration is possible for **factory** elements. In the case of

an on-demand factory, each service will have this property set as it is created. The property element must come last in an **instance** or **factory** element. Note that if the **setMyProperty** method did not exist, or it did not take a single argument of type **String**, then an unchecked exception, **ServiceConfigurationError**, would be thrown.

Properties of primitive type (e.g. **int**, **byte**, etc.) and their "object" equivalents (e.g. **Integer**, **Byte**, etc.) can be set through configuration. The types **String** and **Class** are also catered for. The **value** element has a **type** attribute, that defaults to **String**, as in the case above.

For example, if the **setMyProperty** method took values of type **boolean** instead of **String**, the we might write:

```
<property>
    <name>myProperty</name><value type="boolean">false</value>
</property>
```

instead. If the supplied value cannot be parsed, then a **ServiceConfigurationError** is thrown.

Finally, arrays of the above allowed types are supported through the **arrayvalue** element where the **value** element would normally be used. An **arrayvalue** element has a child **item** element for each item in the array. For example,

```
<property>
    <name>myArrayProperty</name>
    <arrayvalue>
        <item>10</item>
        <item>20</item>
        <item>30</item>
        <item>40</item>
    </arrayvalue>
</property>
```

defines a property with type **Integer[]** and the values in that array are specified by the values enclosed in the **item** elements in the given order: 10, 20, 30 and 40. The **arrayvalue** element also has a **type** attribute for declaring the type of all of the array elements.

As many **property** elements can be applied to an entity as desired, provided they correspond to an existing "setter" methods. Any **property** elements, if configured, must come last within an **instance** or **factory** element.

Configuring metadata

It's often desirable to have more than one implementation of the same service within a context, or to have two or more different instances of the same service implementation within a context. How can we ask a context for a particular instance? The answer lies in service metadata as we will explain ...

Any service instance or service factory can have *metadata* associated with it (in fact, so can a context, but no use has been found for this yet). Metadata is information about an entity, that is external to that entity. The most important metadata in JChassis is "Name" metadata. Name metadata can be used to label an instance or a factory so that it can be referred to within a JChassis context. Metadata does not *affect* a service instance or factory, it is merely associated with it.

Metadata is associated with entities in the XML format using the **metadata** element. The metadata element has exactly the same format and usage as the **property** element, including the types that it supports. However, any **metadata** elements, if present, always come first within an entity such as an interface or factory.

Here is an example of naming two service instances:

```
<instance>
    <metadata>
        <name>Name</name><value>normal-log</value>
    </metadata>
    <interface>Log</interface>
    <implementation>StdoutLog</implementation>
</instance>

<instance>
    <metadata>
        <name>Name</name><value>error-log</value>
    </metadata>
    <interface>Log</interface>
    <implementation>StderrLog</implementation>
</instance>
```

where one instance is named "normal-log" and the other "error-log". Both have the interface **Log** but different implementations: say, **StdoutLog** sends log messages to sdtout, and **StderrLog** to stderr. The users of the enclosing context might use the two services in different roles.

In general, "Name" metadata is useful for distinguishing between services within the same context with the same interface but different implementations or properties. The basic context API allows the user to find services by names as well shall see later.

In the future, metadata other than "Name" may be used, for example to describe other aspects of an implementation. For the moment though, only "Name" with **String** values is reserved. Please do not use it for any other purpose or with any other type of value.

Configuring instances using factories

Instead of specifying an interface and implementation class, a service instance can refer to an existing service factory by specifying its XML *ID*. For example, if there is a **factory** element such as:

```
<factory id="factory1">
    <interface>MyService</interface>
    <implementation>MyServiceImpl</implementation>
</factory>
```

you can configure as many instances as you like by referring to the factory's **id** attribute in this way:

```
<instance factory="factory1"/>
```

This means that the factory is used to produce the instance in the normal way, except that the instance is created when the context is initialised.

Please note that if an **instance** element refers to a factory as above then it is constrained to have only **metadata** elements as children.

Configuring instances using other instances

The JChassis basic framework allows you to *alias* service instances, so that you can declare an alternative name for that instance, or even an alternative interface. Aliasing is achieved using an **instance** element once again.

For example, if we have an instance declared as:

```
<instance id="myimpl">
    <interface>ServiceA</interface>
    <implementation>MyServiceImpl</implementation>
</instance>
```

we can refer to the same instance (the same object in the Java VM) and declare that it has another interface by the following configuration:

```
<instance instance="myimpl">
    <interface>ServiceB</interface>
</instance>
```

Note that the aliasing configuration exposes another interface on **MyServiceImpl**, namely **ServiceB**. It refers to the original instance by using the original instance's ID in the **instance** *attribute*. No new instance is created here. The second **instance** element just provides an alternative way to look up the same object. Alternative "Name" metadata can be defined in the same way. Continuing our example above, we could append:

```
<instance instance="myimpl">
    <metadata>
        <name>Name</name><value>"Jim"</value>
    </metadata>
</instance>

<instance instance="myimpl">
    <metadata>
        <name>Name</name><value>"Bob"</value>
    </metadata>
</instance>
```

which would give the service instance two alternative names: "Jim" and "Bob".

Another example: continuing our logging example from "Configuring metadata" above, suppose we want all logging to go to the same logging service instance. Then we can do this:

```
<instance id="thelog">
    <metadata>
        <name>Name</name><value>normal-log</value>
    </metadata>
    <interface>Log</interface>
    <implementation>StdoutLog</implementation>
</instance>

<instance instance="thelog">
    <metadata>
        <name>Name</name><value>error-log</value>
    </metadata>
</instance>
```

so that there is only one instance, an instance of **StdoutLog**. Whether the context's user asks for "normal-log" or for "error-log" they always get the same instance.

Please note that if an **instance** element refers to another instance as above then it is constrained to have only **metadata** elements and **interface** elements as children.

The "Archiver" example

Before we go much further into the details of writing a JChassis application, lets look at another simple example that is not quite a trivial as the "Hello World" example. The "Archiver" example is a simple program to archive a set of files into a JAR file.

Change to the **examples** directory in the your JChasiss SDK. Run the "Archiver" example program thus:

```
> ./build -Dexample=archiver run
```

You should see an error message such as "**no archive file specified: use "-f" <filename>**". This is because the example program requires arguments - the name of the JAR file to produce and the files to place into a JAR file. The build system for the SDK examples has a slightly clumsy way of inserting command line arguments: you must define the property **args** to be the command line parameters. Lets archive the entire **print_numbers** subdirectory:

```
> ./build -Dexample=archiver -Dargs="-f tmp.jar print_numbers" run
```

You should now see that a new jar file is created called **tmp.jar**. You can check that it contains the contents of the **print_numbers** subdirectory using your Java SDK's **jar** tool.

Now lets look at how this application works. Looking at the **services.xml** file in the **archiver** directory, you'll see that the class **Main** is the application's entry point since it subclasses **Application**. One other service is used: the default command line parser. Because **Main** is a subclass of **Application**, its **run** method is called by the application's **main** method when the application runs. However, the **main** method in **Application** does something first: if a **CommandLineParser** service is available, it uses that service to parse the command line parameters.

Note that in the **services.xml** file, the command line parser's **rules** property is set. That's possible because the implementation class has a **setRules(String[])** method. The **String** array passed to this method contains only one element: **"-tag:file:-f"**. This defines a "tag" in the command line syntax called **"file"** so that a command line argument "**-f**" followed by another argument, say "**foo**", will have a special meaning. Once the command line is parsed, we can retrieve **"foo"** by calling **getParameter("file")** on the parser. After applying rules such as the " **file**" tag rule, the remaining command line arguments are interpreted as positional parameters that can be retrieved by calling **getPositionalParameter(int i)** on the parser, where **i** is the (zero-based) position on the command line. See the *JChassis Module Guide* for more information about the default command line parser service **jc_cmdline_impl**.

You can see in our command line example above how the "**file**" tag ("**-f**") is used to retrieve the file name "**tmp.jar**" from the command line. The remaining command line arguments become positional parameters and are interpreted as the names of the files to add to the JAR file. Refer to the source code in **archiver/src/Main.java**. The **Application** class has a shortcut for getting the command line parser service in the application's context: **getCommandLineParser**. (it also has a convenient **getLog** method too, but we don't use that in this example). The run method uses the parser's **getPositionalParameterCount** and **getPositionalParameter** methods to read in the names of the files to archive. The rest of the **run** method simply does the archiving using these parameters.

This example is not a particularly useful application, since we already have a **jar** tool that can do archiving and more, but hopefully it's useful in showing you how JChassis can be used to place configuration, such as command line parsing rules, outside of the code. The is preferable to hardwiring configuation into the code.

An important point to note is that the **setRules** method is part of the implementation class but not the service interface. That is, the **org.jchassis.cmdline.CommandLineParser** interface has no **setRules** method. Hence, we have factored the *implementation-specific* configuration out into the XML file. This is important because if we want to change to a different command line parser implementation, we can do so by changing the XML configuration and swapping the command line parser implementation's MAR file for another. Once again, we can do this *without having to modify or recompile the application's Java code*.

Putting it all together

Here is a check list for creating an application using the JChassis basic framework:

1. Use a good build system. We recommend the excellent Apache Ant build system.

2. Create a subclass of **org.jchassis.core.app.Application**.

3. Implement your subclass's **run** method to do whatever you application does. It may use one or more services provided by components in its enclosing context.

4. You may need to write some of your own components. it's encouraged that you try to make these as reusable as possible, both for you own sake, or the sake of your organisation, and also so you could contribute them back to the JChassis project or the open source community in general. Writing your own components is covered in the next section.

5. Make sure all of the relevant MAR files are in the classpath. You will at least need **jc_attribute.mar**, **jc_core.mar**, **jc_coreapp.mar**, **jc_coreloader.mar**, **jc_basic.mar**, **jc_basicloader_kdom2.mar**, **jc_resloc_if.mar**, **jc_resloc_impl.mar**, **kxml2.mar**, **kdom2.mar** and **xmlpull.mar**. These are found in the **modules** directory of your JChassis SDK. A shortcut is to use Ant to pick up all MAR files in the SDK. This is OK until you want to deploy your application, when you will want to include as few MAR files as possible.

6. You should probably use the **jcdep** tool to check that your MAR files' dependencies are all satisfied. See "The SDK Tools" section.

7. If you use any other services, such as logging or command line parsing, you'll need MAR files for those too. Consult the *JChassis Module Guide* for details.

8. Construct a **services.properties** file. Just use the same one as the "Hello World" example, namely:

```
loader.org.jchassis.basic.loader.kdom2.XmlServiceConfigurationLoader=

service.org.xmlpull.v1.XmlPullParser=\
    org.kxml2.io.KXmlParser
instance.1.org.jchassis.resloc.ResourceLocator=\
    org.jchassis.resloc.impl.ResourceLocator
```

9. Construct a **services.xml** file with the configuration that you desire to work with your code, as we have discussed in this section. it's easiest to leave the public and system IDs out of the **DOCTYPE** element, unless you are going to include the **jcservices.dtd** in your application.

10. Run your application. Enjoy!

The "Item List" example

A more complex example of a JChassis basic framework application is the "Item List" example in the **examples/item_list** subdirectory of your JChassis SDK. This simple application persistently stores a list of strings and allows the list to be displayed and manipulated in a user interface. This application can thus be used as a rudimentary TODO list, shopping list, etc.

The "Item List" application uses an JChassis interface called the JChassis "UI" interface. This interface has two different implementations for generating simple GUIs in AWT, or on ANSI text consoles (such as those used in GNU/Linux). This means that you can write your application to use the JChassis UI interface and then run it with one of the two implementations, without having to change a single line of code (you get two UIs for the price of one!). The example is set up to use AWT. You can change this by

editing the example's **services.xml** file to change the implementation of the **org.jchassis.ui.Display** interface.

When selecting the ANSI terminal UI, you should also set the Log service to a null log (see the comment in the **services.xml**) file) and use the supplied **run_rawio** to run the example. This script takes care of setting up the console to accept raw input and to not automatically echo characters.

More complex applications

You may wish to write a application that has different parts running in different threads. JChassis provides a convenient mechanism to do this by using instances of **Engine** (see the **jc_coreapp** module in the *JChassis Module Guide*) in your configuration. Unlike **Application**, there can be as many instances of **Engine** as you like in your configuration. Each **Engine** instance will be "started" when the application initialised, and will get the service context you embedded it in your **services.xml** file.

Creating JChassis Components

In this section, we describe how to create your own JChassis components. Here are a few good reasons why you would want to do this:

- you want your application to have a better structure: the use of loosely coupled components will lead to a more flexible and maintainable program;
- you want to reuse your components in different applications, so you want them to be easily transportable;
- you want your components to be reused in other parts of your organisation, or by other organisations and developers: again, they need to be easily transportable and also highly usable and understandable by others.

Defining a service

Defining a service in JChassis is easy enough. Simply define an interface that describes the service, and then define one or more implementations of that interface. It is even valid to just define an implementation class and make that class both the interface and the implementation (in the JChassis sense). We advise against this unless you are absolutely certain that you'll only ever need one implementation. Even then, you are polluting your interface every time you add or change a method in your implementation class. it's best that you have a separate interface class if possible, to future-proof your application.

The only restriction on a service implementation class is that it must have a constructor that takes no arguments. JavaBeams have the same constraint. Hence, any JavaBean can be a JChassis service. This constructor will be the one that is used to instantiate the service in the JChassis framework.

The next question you need to ask is "Does my component need to use any services?" If so, it needs to be able to get hold of its service context. To do that, your implementation class should implement the **ServiceContextual** interface in the **org.jchassis.core** package (see **jc_core** in the *JChassis Module Guide*). One convenient way to do this is to extend **BaseServiceContextual**. This means that your implementation can call its own **getServiceContext** method to get its service context. This service context will be set by the framework when your implementation instance is inserted into the service context. Once you have the **ServiceContext** instance, you can use that instance to obtain other services. If your implementation was inserted into a service context in the basic framework, then your service context will be a **BasicServiceContext** (see **jc_basic** in the *JChassis Module Guide*), and is equipped with more features, such as getting services by name.

If you don't want to package your service implementation in any way then you are finished. Otherwise you could put it in a JAR file and read no further in this section. Alternatively, you could package it as a JChassis module. We discuss JChassis modules and their advantages below.

JChassis Modules

Components in JChassis are called JChassis *modules*. These are distinct chunks of functionality with well-defined interfaces. Traditionally, Java uses JAR files to package up components and APIs — there is even a Sun standard to put version and other information (metadata) about the JAR contents into the JAR file. We find this approach to be too limited: What if the archive contains several resources that we wish to declare — their names and their versions? What about dependencies between a component and other components' interfaces or other resources. The dependencies will often relate to the versions of those other resources. How do we represent this version dependence?

Modules address the above questions by providing the information about a module in a convenient XML format: the *JChassis Module XML format*. Here is an example (we'll skip the XML headers):

```
<module>
    <name>jc_store_if</name>
    <version>0.1</version>
    <metadata><name>Author</name><value>Sam Stainsby</value></metadata>
    <metadata><name>License</name><value>LGPL 2.1</value></metadata>
    <metadata>
        <name>Description</name>
        <value>An interface for persistently storing and retrieving objects
        </value>
    </metadata>

    <interface>
        <name>org.jchassis.store.ObjectStore</name>
        <version>0.1</version>
    </interface>
</module>
```

This shows the module XML for the JChassis "Storage" service interface module. In JChassis, it is convenient to package service interfaces and default service implementations in separate modules to avoid the interface carrying around any excess baggage. Without going into too much detail, the above XML tells us that:

- this is a module, named "jc_store_if" with module version 0.1
- the author of the interface code is Sam Stainsby and the code's license is LGPL (Lesser GNU Public License) version 2.1
- the description is, well, the stated description
- the module contains one JChassis service interface, **org.jchassis.store.ObjectStore**, of version 0.1
- no dependencies are listed

Here is the module that contains the default implementation of the JChassis Storage service (again skipping the XML headers):

```
<module>
    <name>jc_store_impl</name>
    <version>0.1</version>
    <metadata><name>Author</name><value>Sam Stainsby</value></metadata>
```

```
    <metadata><name>License</name><value>LGPL 2.1</value></metadata>
    <metadata>
        <name>Description</name>
        <value>A simple storage implementation that uses Java serialisation.
        </value>
    </metadata>

    <implementation>
        <name>org.jchassis.store.impl.ObjectStore</name>
        <version>0.1</version>
        <implementation-of>
            <name>org.jchassis.store.ObjectStore</name>
            <version>0.1</version>
        </implementation-of>
    </implementation>
</module>
```

The items to note here are:

- there is one module called "jc_store_impl" of version 0.1
- the author and license are the same as for the service interface
- the module contains one JChassis service implementation called **org.jchassis.store.impl.ObjectStore** with version 0.1
- this implementation is an implementation of the **org.jchassis.store.ObjectStore** version 0.1 service interface: note that more than one interface can be declared in this way

So far, we have not seen any examples of dependencies. Or have we? In fact, in the last example, there is an implicit dependency on the **org.jchassis.store.ObjectStore** version 0.1 interface (in fact any storage interface version greater than or equal to 0.1 will suffice since backward compatibility of interfaces is the default — see later).

Dependencies can be listed explicitly too, as in this example (once again excluding the XML headers):

```
<module>
    <name>jc_coreloader</name>
    <version>0.1</version>
    <metadata><name>Author</name><value>Sam Stainsby</value></metadata>
    <metadata><name>License</name><value>LGPL 2.1</value></metadata>
    <metadata>
        <name>Description</name>
        <value>The configuration loader for the JChassis core framework.
            </value>
    </metadata>

    <requires>
        <module>
            <name>jc_core</name>
            <version>0.1</version>
        </module>
    </requires>
</module>
```

This module is the one that is used by JChassis to load the **services.properties** file in the JChassis core framework. There are no service interfaces or implementations declared here. Instead this module declares that it requires the module "**jc_core**", that contains the essential code for service contexts in the

core framework.

The Module XML format

The exact syntax of the JChassis Module XML format is provided in the DTD called **jcmodule.dtd** in the doc/dtd/0.1 subdirectory of your JChassis SDK. However, in this section we provide a more gentle overview.

The top-level element in a module XML file is the **module** element. This may contain nested module elements if you wish to package more than one module in a MAR file — we've not found a use for this yet though.

Generally, a module XML file declares a numbers of *resources* which include modules, services interfaces and implementations. Other resource types may be added in later versions. Every resource must have a name and a version, and hence must contain a **name** and a **version** element. These come first inside the element that represents the resource: **name** first, and then **version**. A resource version is a sequence of non-negative integers, separated by "**.**" characters. A resource name is any string.

Apart from modules, other resources are JChassis service interfaces, represented by **interface** elements, and JChassis service implementations, represented by **implementation** elements.

In some cases, you may want to declare some kind of implementation class, that is not necessarily a JChassis service implementation, in its own right. You can use an **implementation** element for this as well. This can be handy for setting up dependency relationships, as we'll see later.

After the **name** and **version** in a resource, there can be one or more optional **metadata** elements. These describe the resource. Unlike metadata values in **service.xml** files, only string metadata values are supported in these elements. That is, there is no "type" attribute in the **value** element.

The order of child elements within a **module** element is important. First comes the module's **name**, **version** and **metadata** elements, then the **interface** elements and then the **implementation** elements.

Within an **implementation** element, there can be zero or more **implementation-of** elements. Each **implementation-of** element declares a JChassis service interface that the implementation wishes to expose. The **implementation-of** element must contain a **name** element followed by a **version** element, expressing the name and version of the interface exposed. For example,

```
<implementation>
    <name>MyImpl</name>
    <version>2.1</version>

    <implementation-of>
        <name>InterfaceA</name>
        <version>1.0</version>
    </implementation-of>

    <implementation-of>
        <name>InterfaceB</name>
        <version>1.1</version>
    </implementation-of>

</implementation>
```

declares that the service implementation **MyImpl** (version **2.1**) implements two service interfaces: **InterfaceA** (version **1.0**) and **InterfaceB** (version **1.1**). Note that not every interface has to be exposed — it's up to the module developer to decide what should be seen by the module's users.

You now have the basics that enable you to declare JChassis modules and service interfaces and implementations. But what about the dependency information that we promised? Each element representing a

resource (**module**, **interface** or **implementation**) can optionally have a final **requires** element. This element declares the names and versions of the resources that the enclosing resource requires to operate. Compatible versions are normally specified as a range of versions.

The most simple version construct in a **requires** element is something like:

```
<requires>
    <interface>
        <name>MyInterface</name>
        <version>3.0</version>
    </module>
</requires>
```

which means that **MyInterface** version **3.0** or above are required.

A version element always encloses a single version number. How that version number is interpreted is controlled by an optional attribute on the **version** element, called **range**. Here is a summary of the effect of the **range** attributes:

Table 1. The compatible versions v implied by the range attribute on a version range <version range="...">V</version> (V must be a version number, e.g. 5.0).

value of "range"	compatible versions
ge	**v >= V**
le	**v <= V**
gt	**v > V**
lt	**v < V**
eq	**v == V**
ne	**v != V**
any	**any v is compatible**
none	**no values of v are compatible**

Instead of just stopping there with handy version ranges for your dependencies, JChassis also allows you to construct complex dependency relationships such as "resource X requires either resource Y or (resource Z together with resource W)". This is accomplished with the following elements that can occur as children of a **requires** elements and can also be nested inside each other:

- **all-of**: every contained resource is required
- **one-of**: one, and only one, of the contained resources is required
- **some-of**: one or more of the contained resources is required
- **none-of**: none of the contained resources must be present

These elements can nest as we've said, and can also contain resources, such as **module**, **interface** and **implementation** elements. For example "implementation X requires either interface Y or (interface Z together with interface W)", this can be expressed as:

```
<requires>
    <one-of>
```

```
            <interface><name>Y</name><version>1.0</version>
            <all-of>
                <interface><name>Z</name><version>1.0</version>
                <interface><name>W</name><version>1.0</version>
            </all-of>
        </one-of>
    </requires>
```

Note that a statement about the version of each required resource is mandatory. In this case we have taken "version 1.0 or greater" to be the rule for each required resource.

You should now have an idea of how to set up a module XML file to declare the contents of your module and what is depends on. Later we explain what do with those contents and the XML file to make a component that can be easily shared between developers and installed into JChassis applications (see "Module Archives").

Module naming conventions

The name space of modules is important because it's used when defining resource interdependencies as we have seen. Here we define a few conventions that you should use when naming modules:

1. Use the "**_**" character to separate name components.
2. If your module contains a service interface (and little else), use the suffix "**_if**".
3. If your module contains a default service implementation (and little else), use the suffix "**_impl**".
4. Use a prefix that relates to you or your group, team or organisation. For example, if you are building components as an employee of ABC Inc., you could use the prefix "**abc_**". Please use one prefix consistently throughout your whole organisation.
5. Please don't use the prefix "**jc_**". This is reserved for "official" JChassis components.

Metadata naming conventions

As you can see, any metadata you like can be attached to resources in a module. In general, we would like some of these to have consistent meanings. So far, we are reserving the following resource metadata names and their meanings:

Table 2. Reserved resource metadata names and meanings.

resource metadata name	meaning
Author	the person(s) or organisation that created the resource (from a copyright perspective)
License	the license that the resource is distributed under (we recommend an OSI approved license [http://opensource.org/licenses/])
Description	a short description of the module

Module Archives

In JChassis, *module archive* (MAR) files are simply Java archives (JARs) with some extra information added. As such, they can be used just as JAR files are, such as placing them in the classpath of your application. The standard **jar** tool in Sun's SDK can be used on them. Every JAR file contains a **META-INF** directory that contains metadata about the JAR. The extra information in a MAR file is the module XML described above, which is placed in the JAR's **META-INF** directory, in a file called **jcmodule.xml**.

The module XML in a MAR file can be listed using the **jcmar** tool. Even better, you can check that the dependencies between a group of MAR files are satisfied using the **jcdep** tool. These tools are described in the section "The SDK Tools" below.

Also included in your SDK are *source* MAR files for each module. These MAR files are suffixed with "**.src.mar**" and are found in the **src_modules** subdirectory in your JChassis SDK. These MAR files contain the source code for the modules, where applicable, and also the Apache Ant build files to build them.

Creating your own service context

There is nothing stopping you from implementing you own service context, for example by implementing the **ServiceContext** or **BasicServiceContext** interface, or easier, extending **DefaultServiceContext** or **DefaultBasicServiceContext** (see the **jc_core** and **jc_basic** modules in the *JChassis Module Guide*).

If you want to configure your custom context from a file, you may wish to write a *loader* for that context. You can extend from the loaders found in the **jc_coreloader**, **jc_basicloader** and **jc_basicloader_kdom2** modules (see the *JChassis Module Guide* for information about these loaders).

The SDK Tools

In this section, we describe how to use the tools that come with the JChassis SDK, **jcmar** and **jcdep**. By the way, both of these tools are JChassis applications — interested readers can find their configuration files in the separate directories under the **conf** directory in your JChassis SDK (note that the tools use a different loader and uses a validating XML parser). Both tools are started by scripts withing the **bin** directory of the SDK. You will need to define the **JAVA_HOME** environment variable (or edit it into the script called **common**).

IMPORTANT: The SDK tools need the jcmodule.dtd to validate module configuation. This is currenlty dowloaded by HTTP as required. Hence, the tools need to access the network to run. If you see a message such ass "External entity not found: http://jchassis.sourceforge.net/dtd/module/0.1/jcmodule.dtd" then the tools is failing because it cannot retrieve the DTD.

Sorry, but there are no Apache Ant tasks to wrap these tools (yet!).

The jcmar tool

The **jcmar** tool simply lists the **jcmodule.xml** file within a MAR file. This file is discussed in "Creating JChassis Components".

For example,

```
> jcmar mymodule.mar
```

will list the **jcmodule.xml** file in the **META-INF** directory stored in the **mymodule.mar** MAR file.

If more than one MAR file is listed on the command line, the output is just the concatenation of the contents of the **jcmodule.xml** files from each MAR file.

The jcdep tool

The **jcdep** tools is used to test a group of MAR files to see if their modules' dependencies are all mutually satisfied. This is typically used to determine if the MAR files that you are using for your application have any outstanding dependencies. This helps prevent accidental misconfigurations by inadvertently leaving out MAR files.

The **jcdep** tool simply takes a list of MAR files as it arguments. It prints out to the console any unsatisfied requirements for these MAR files. If there are no unsatisfied requirements, it prints nothing. For example,

```
> jcdep mymodule1.mar mymodule2.mar mymodule3.mar
```

will check the requirements in the **jcmodule.xml** files in each of the MAR files **mymodule1.mar**, **mymodule2.mar** and **mymodule3.mar** to find out what resources the modules provide and also what resources they require. It will then attempt to reconcile those requirements with the total set of resources provided by these MAR files. If any of those requirements are not satisfied, they will be printed to the console. Otherwise, nothing will be printed to the console.

The core framework

So far in this document, we have almost exclusively talked about the JChassis *basic* framework. In this section we discuss the JChassis *core* framework. The core framework is only used if your code size requirements are very constrained, such as a few tens of kilobytes. The core framework does not require XML, and instead uses the Java property file format for service configuration. This file is called **service.properties** and must be found in the classpath of your application. The features available on a core framework's service context are significantly reduced (compare **ServiceContext** in the module **jc_core** with **BasicServiceContext** in the module **jc_basic**, in the *JChassis Module Guide*).

The format of the **service.properties** file is fully explained in the javadoc for the **DefaultServiceConfigurationLoader** class in the **jc_coreloader** module (see the *JChassis Module Guide*). The location and name of the **service.properties** file can be changed by setting the system property **org.jchassis.core.config** which is the resource name (as in a class's **getResource** method) for the property file.

The **examples** section of your JChassis SDK contains some examples of the use of the core framework in the **core** subdirectory. By looking in the file named **depends.properties** in the **examples/hello_world** example application, you see that only the MARs **jc_core.mar**, **jc_coreapp.mar** and **jc_coreloader.mar** are needed. This is only about 15KB of byte code. The **print_numbers** example shows an alternative way of configuring services (see its **service.properties** file) without needing the basic framework, although it is less flexible.

An important point to note is that the "root" context in a basic framework application is actually a core service context (an instance of **ServiceContext**), which is why resources added directly under the **services** element in a **service.xml** file have constraints on their features: **factory**, **metadata** and **property** elements are not allowed and instance aliasing is not allowed. Normally such resources are placed in a child **context** element instead. The core context in a basic framework application is initially loaded from the **service.properties** file in the application's classpath. Other resources directly under the **services** element in the **services.xml** file are added after this. That is why the **services.properties** file always contains an XML parser and a "loader" entry — these two entities are used to load the **services.xml** file.

A. What about the JavaBean APIs?

JChassis is not a replacement for standard *JavaBeans* components. JavaBeans components can live quite happily within a JChassis context, and we have seen that any JavaBean can be a JChassis service.

JChassis does however provide an alternative to Sun's standard BeanContext API. The BeanContext API provides service contexts as does JChassis, but it also requires services and components within those contexts to implement a set of interfaces and to conform to a particular event model. We believe that these requirements are too onerous for simple components.

For example, the BeanContext API spends a lot of time worrying about services being added and removed during runtime. This requires contexts to keep track of how many entities are currently using the service. Entities within a context must explicitly acquire and release services so that a context knows when it can allow them to be removed. Services must send events when they are removed, and other entities must listen for those events and react accordingly. JChassis does not explicitly support adding and removing services during runtime, because in the vast majority of cases, that is not needed. That is not to say that a special class of services will allow this in the future, but the functionality will be provided in a flexible "plug-in" way rather than hard-coded so that all services and other entities must suffer the added complexity.

JChassis' lack of onerous requirements on service implementations means that integrating your existing application code into JChassis is much easier, compared to many components systems.