The EzcContainer: Reference and User's Guide

Revision History			
Revision	**Datum**	**Desription**	**By**
DRAFT	2004-08-17	Created	Jens Wyke

1. Introduction

The EzcContainer is an implementation of the popular Inversion-of-control (or IOC) design pattern. Others and more well-known IOC implementations include PicoContainer, Spring Framework, Apache HiveMind and Apache Avalon.

When developing an application, the EzcContainer can be used as a mechanism through which the different functional components of the application are instantiated, configured (e.g. by reading parameters from a configuration file) and invoked. The container also manages the dependencies between components through dependency injection. Dependency injection means that the container provides, at instantiation time, each registered component with access only to those components (which are also registered in the container) with which it needs to collaborate. In the IOC pattern the responsibility for knowing how to gain access to dependencies does not lie with the individual components but is a job for the *container*.

This document is both a documentation of the EzcContainer's overall design and implementation and a guide on how it can be used when developing applications.

The document does not however, particularly go into describing and explaining the reasons for using IoC/Dependency Injection as these are nicely described elsewhere, please see resources.

The EzcContainer source can be found at: FIXME

1.1 Resources

[1] Inversion of Control Containers and the Dependency Injection pattern (Martin Fowler)
http://martinfowler.com/articles/injection.html

[2] The Spring Framework
http://www.springframework.org

[3] PicoContainer
http://picocontainer.codehaus.org/

[4] Apache Avalon
http://avalon.apache.org/

[5] Apache HiveMind
http://jakarta.apache.org/hivemind/

1.2 Terminology

Definition of terms used in this text.

- **Business Application**
  Throughout this text, the term *Business Application* refers informally to a software system which may potentially make use of the EzcContainer as a way to achieve its goals. Example: "A Business Application may use an EzcContainer to manage initialization, configuration and component dependencies".

2. The EzcContainer

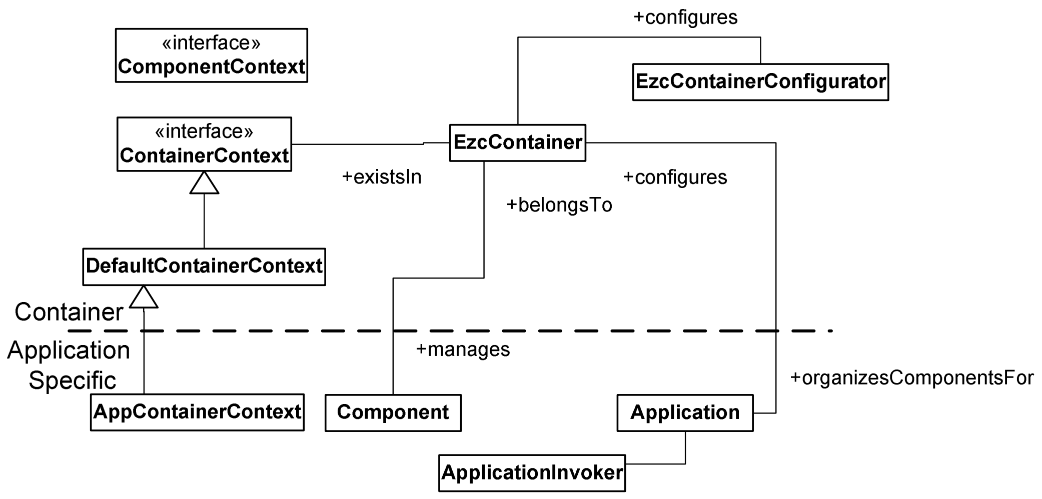
This chapter describes the overall design of EzcContainer. Key features and principles are highlighted. Each public class and interface is described as are important concepts such as component dependencies, sub-containers, instantiation of components etc. In addition to the descriptions of classes and interfaces (static view) there are also some illustration on how they work together (dynamic view) during initialization and application runtime.

2.1 Key features

- Agnostic about what constitutes a component, except that each component MUST provide its functionality through one or more Java Interfaces and one Java Class
- Simple API (one method: `addComponent`) for initialization using Groovy, plain Java or your custom-built mechanism.
- Obvious rules for component dependencies – the EzcContainer does not try to apply any intelligence in “resolving” component dependencies. This makes configuration straight-forward.
- Easy and non-intrusive to use in Business Applications – only one application-specific class need to be aware of the fact that EzcContainer is used at all.
- Does component registration, initiation, and dependency injection *only*; EzcContainer does not provide any framework to aid with issues specific to J2EE or other development environments.
- Supports the use of nesting containers.
- Few dependencies to external tools and libraries (except for configuration using Groovy, which is optional, no dependencies except J2SE 1.3 exists).
- Less than 1000 lines of code, compiled jar less than 15Kb

2.2 Important Abstractions

The diagram below illustrates the main parts of the EzcContainer design and how they are statically related. In the following subsections, each of them is described. This section concentrates on explaining the aspects of the design which are *visible to the user* (the Business Application where EzcContainer is used). The inner workings of the class EzcContainer, which is responsible for most part of the functionality, are not covered here.



2.2.1 ContainerContext and DefaultContainerContext

A `ContainerContext` provides an `EzcContainer` with callbacks which are invoked at certain events in the EzcContainer runtime. Most importantly, there are callbacks to handle creation and initiation of component instances according to the *component definition* used by the application in question. Other callbacks include methods to allow the `EzcContainer` to perform its logging using the logging toolkit choosen by your application.

The class `DefaultContainerContext` is a default implementation of the `ContainerContext` interface, it is used by the `EzcContainer` unless another `ContainerContext` is specified. The `DefaultContainerContext` sends logging output to `System.out` and assumes that a component is any Java Bean (i.e: No-arg constructor and dependency injection and configuration through java bean style setter methods).

2.2.2 EzcException

Whenever something fails within the EzcContainer an `EzcException`, which is a `RuntimException`, will be thrown providing a message detailing what the problem.

2.2.3 EzcContainer

This is the main class of the EzcContainer implementation. Components are added to the container, through the method `addComponent()`, typically (but not necessarily) during application initialization, and are used, explicitly, by the application through the `getComponent` method of the container during application runtime, or implicitly as a result of being a dependency to another component.

An `EzcContainer` is responsible for:

- holding a registry of components, where each component is uniquely identified using a *key* (`java.lang.String`).
- creating and initializing component instances (For this it depends on having access to the application specific callbacks located in a `ContainerContext`).
- Wire components together based upon their declared dependencies. (For this it depends on having access to the application specific callbacks located in `ContainerContext`)
- Providing the Business Application, via a *Container Holder,* with access to components not marked as *private*.

2.2.3.1 The `addComponent` method

The only way to add a component to an `EzcContainer` is through the `addComponent` method. Once a component is added to the container it can be used by either non-component application code via the `getComponent` method (for public components) or by private components which have declared a dependency upon it – and therefore will be able to reference it via a `ComponentContext`. After each call to `addComponent`, a consistency check is performed to assure that all specified interfaces are in fact implemented and that all declared dependencies are satisfiable by components added prior to this component.

If the component cannot be added, an `EzcException` will be thrown. Reasons for not being able to add a component include:

Duplicate or invalid key

Unsatisfiable dependencies

The table below describes the parameters accepted by the `addComponent` method.

`key` (`java.lang.String`)	The key is a unique identifier for the component in this container.
`implClass` (`java.lang.Class`)	Class providing the implementation for the component
`features` (`String`)	Features are a way of controlling various aspects of a components behavior. The features of a component are specified by concatenating together the desired combination of the following constants, which are defined in the EzcContainer class (e.g. FEATURE_PRIVATE+FEATURE_SINGLETON+FEATURE_LAZY) If no features are to be specified, either `null` or an empty String may be entered. FEATURE_PRIVATE Only public (=non-Private) components may be accessed by code not managed by the container. If `getComponent(`*`key`*`)` is called for a component which was marked as private, no component will be returned and an EzcException will be thrown. Private components are only accessible as dependencies for other components (via a `ComponentContext`) FEATURE_SINGLETON By default, a new instance of a component is created on each request (requests can be explicitly via method `getComponent` or implicitly via a dependency exposed by a `ComponentContext`). By specifying this feature, the behavior changes so that one single instance is reused for

	all requests. The component will act as a singleton relative to its container. FEATURE_LAZY If this feature is specified, instantiation of actual component instances will be done in lazy mode. This means that instantiation is deferred until the first request for one of the components service methods is made. A client will hold a reference to a Proxy which instantiates the real component when it is first needed.
`dependencies` (`java.util.Map` of (`String,Object`))	Declares the dependencies for this component. The dependencies Map consists of (`localKey`, `globalKeyOrDependantObject`) pairs representing the dependencies for the component. The dependencies declared here are used by the `EzcContainer` when generating a `ComponentContext` for use by the `ContainerContext`, on behalf of the `EzcContainer,` at component instantiation. *`localKey`* is the name through which the `EzcContainer`, via `ComponentContext`, will expose a dependency to a component. *`globalKeyOrDependantObject`* refers to another component (in the same container) by its *key* **or** provides configuration to the component in the form of an arbitrary Java Objects (Objects not managed by the container). If *`globalKeyOrDependantObject`* is a String, it is interpreted as the key of a component in the container and is used to provide access to that component. If *`globalKeyOrDependantObject`* is *anything but a String*, it is used as a dependency on an arbitrary Object. *It is true that this “magic” meaning of String makes it impossible to declare a dependency to a String object since it will be interpreted as a component reference. The decision was made in order to simplify the API needed for adding components to the container, and is assumed to have little or no negative effects in practice.* The following examples illustrates how different kinds of dependencies can be expressed: An entry: `(“userManager”,”LdapUserManager”)` will cause the component with key `LdapUserManager` to be accessible as the local key `userManager`. Note how the LdapUserManager could be replaced by e.g. a `DbUserManager` without changing the component implementation. Entry`(“config”,this.getClass().getResourceAsStream(“config.props))` will cause the contents of the file `config.props` to be accessible for local key `config`. When the component gets hold of the `InputStream` it can read and parse it into e.g. a `java.util.Properties` instance. The use of a local key provides a level of indirection between the component implementation class and the component keys in the container. Component implementations must know which local keys they require to have as dependencies, but need no knowledge about which components they are mapped to. Nothing stops the same component to be referred twice, by different local keys. Consider, for example, a component modeling a network connection, lets call it

	`NetworkConnection`. Assume that the network connection has primary and secondary DNS servers but that there currently only exists one physical server (which is modeled by a class `DnsServer` and registered as a component with `key=theOnlyDns`.
	
	DNS dependencies for the network connection can be declared as follows:
	
	`("primaryDns", "theOnlyDns")`
	
	`("secondaryDns", "theOnlyDns")`
	
	This scheme will make it possible to start using different DNS servers once they exist – without having to change the `NetworkConnection` class.
	
	As the example above illustrates, the `localKey` can be thought of as stating the role or association name played by the dependent object relative to the component.

`interfaces` (`Collection` of `java.lang.Class`)	One or more interfaces which are to be exposed for the component. All interfaces given here must also be implemented by `implClass`. When a client uses a component from the container he will not interact directly with the component instance, but via a Dynamic Proxy which exposes only the declared interfaces listed here. This makes it possible to have setter methods on the implementation class used for injecting dependencies but avoid exposing those methods by not including them in one of the interfaces.

2.2.4 EzcContainerUtils

EzcContainerUtils provides static methods to facilitate configuration of an EzcContainer using Groovy scripting. Initiation using Groovy is covered in chapter 3, "Configuring the EzcContainer using Groovy".

2.2.5 ComponentContext

For each registered component, the container provides instances of the component with access to dependencies (other component instances) and configuration (regular Java objects, such as instances of `java.lang.Properties`) through the `ComponentContext` interface.

ComponentContext is the mechanism used by the container to expose (via the ContainerContext) to individual components instances of those and only those other components (or ordinary objects) it depends on.

Depending on the `ContainerContext` implementation in use, component instances may not have direct references to their `ComponentContexts` but rather be provided with their dependent components via some other mechanism – such as JavaBean property setters.

2.2.6 Component

The EzcContainer does not impose or assume any definition of what constitutes a component in the context of a particular Business Application except that each component must **have it's services exposed via one or more *interfaces* in the Java programming language and must be able to represent by one Java class**. That is, a component may consist of any number of classes and interfaces, but it can only expose functionality (to code outside of the component) via one specific class which acts as a *facade* for the component. The strict requirement for using Interfaces is caused by the fact that all component instances handled by the EzcContainer will make use of the Java's dynamic proxy feature when creating component instances – each component instance will be accessed via a dynamic proxy.

Your applications definition of what a component is will need to be reflected in the implementation of the method `createInstance` of the `ContainerContext` supplied at container creation. For example: If you have decided for your application that all components must extend the (fictive) class `MyAppComponentBase`, which has a method `init(Properties)` and have a no-argument constructor, then your `createInstance` method can safely assume these things to be true about the classes it receives as input.

When designing EzcContainer, the anticipated kind of component to be used with the container has been the kind used to divide functionality of an application into coherent and loosely coupled units. (example component names may be: InventoryManagement, UserRespository, OrderManagement).

Although not technically impossible, the focus has not been to create a container where each and every class participating in an application is to be treated as components. For example, the main intention is not that domain objects, such as, for example, *UserAccount*, *Order*, *OrderItem*, *Product* are made components of the container. Instead, [instances of] such classes are meant to be managed (created, located, updated, deleted) by components such as *InventoryManagement*, *UserRespository* and *OrderManagement*.

2.2.7 Container Holder and Container Invoker

The *Container Holder* and *Container Invoker* are not classes provided by the EzcContainer tool. They rather represent roles played by classes belonging to a Business Application. The existence of an Container Holder or Container Invoker are not in any way required in order to use EzcContainer but are presented here as a way of illustrating how `EzcContainer` can be used in an application.

- The *Container Holder* represents the part of a Business Application functionality which are managed by the EzcContainer and allows code not belonging to a class or component managed by the container to invoke components in the container. The Container Holder typically will have methods for accessing the public components of an EzcContainer and thus provide an entry-point to the web of components managed by the container. The Container Holder is responsible for creating, initializing and holding the EzcContainer. It also has the responsibility of invoking the public components of the container on behalf of the Container Invoker. The Application is typically the only object with a reference to the EzcContainer.
- The *Container Invoker* represents some piece of a Business Application that cannot be a component managed by the EzcContainer. One reason for not being able to treat a certain object as a component of the EzcContainer is that it has its creation managed by some other framework or container. This, for example, is the case for Java Servlets. If you are building a web application using the EzcContainer, and an MVC framework such as Jakarta Struts, the Container Invoker may be the *Struts Actions* being executed when users make HTTP requests. In the Struts case (or equivalently if other MVC framework is used), the Actions will receive the request and as soon as possible hand over to a component in the container to perform the requested function.

2.3 Adding and Using Components (Dynamic View)

This section provides a walkthrough illustrating the steps typically involved in initiation and use of an EzcContainer within a Business Application.

In the example below, the same Java class and exposed interface (`TestComponentImpl.class` and `TestComponent.class`) are used for all components. The interface `TestComponent` has only one method, called `invoke`, which does nothing except recursively invokes all its dependent components. Although not very useful in a real application, this behavior is useful as it can be used to produce a trace printout listing the order in which component instances are created and invoked.

Also, it is probably uncommon in a real application for all components to be instantiated using the same class, as is the case in this example. However, this stresses the fact that a component is not identified by its class or interface, but by the *key* with which it was added to the container.

In Listing 1 contains java code used in the illustration of the course of events taking place when initiating and using an EzcContainer.

Listing 1 Java code showing creation, initialization and use of an EzcContainer

```
// PART 1 - Creating container and container context
// ==================================================================
ContainerContext ctx = new MyContainerContext();
EzcContainer c = new EzcContainer(ctx);

// PART 2 - Adding components to the container
// ==================================================================

// Tracker is an example of a non-component object made available to
// (all) components in the container as a dependency
ArrayList tracker = new ArrayList();

// All components expose the same interface
final List interfaces = new ArrayList();
interfaces.add(TestComponent.class);

{ // ====================================================
        final HashMap dependencies = new HashMap();
        dependencies.put("tracker", tracker);
        c.addComponent("comp5", TestComponentImpl.class, interfaces, ""
                        + EzcContainer.FEATURE_PRIVATE
                        + EzcContainer.FEATURE_SINGLETON, dependencies);
}{ // ====================================================
        final HashMap dependencies = new HashMap();
        dependencies.put("tracker", tracker);
        c.addComponent("comp3", TestComponentImpl.class, interfaces,
                        EzcContainer.FEATURE_PRIVATE, dependencies);
}{ // ====================================================
        final HashMap dependencies = new HashMap();
        dependencies.put("tracker", tracker);
        dependencies.put("a", "comp5");
        dependencies.put("b", "comp3");
        c.addComponent("comp1", TestComponentImpl.class, interfaces,
                        EzcContainer.FEATURE_SINGLETON +
                        EzcContainer.FEATURE_PRIVATE, dependencies);
}{ // ====================================================
        final HashMap dependencies = new HashMap();
        dependencies.put("tracker", tracker);
        dependencies.put("a", "comp1");
        dependencies.put("b", "comp3");
        dependencies.put("c", "comp5");
        c.addComponent("comp2", TestComponentImpl.class, interfaces,
                        EzcContainer.FEATURE_LAZY +
                        EzcContainer.FEATURE_PRIVATE,dependencies);
}{ // ====================================================
        final HashMap dependencies = new HashMap();
        dependencies.put("tracker", tracker);
        dependencies.put("a", "comp3");
        dependencies.put("b", "comp3");
        dependencies.put("c", "comp2");
        c.addComponent("comp4", TestComponentImpl.class, interfaces, "",
                        dependencies);
}
// PART 3 - referencing and invoking component
// ==================================================================
TestComponent comp = (TestComponent) c.getComponent("comp4");
comp.invoke();
```

2.3.1 Starting the application

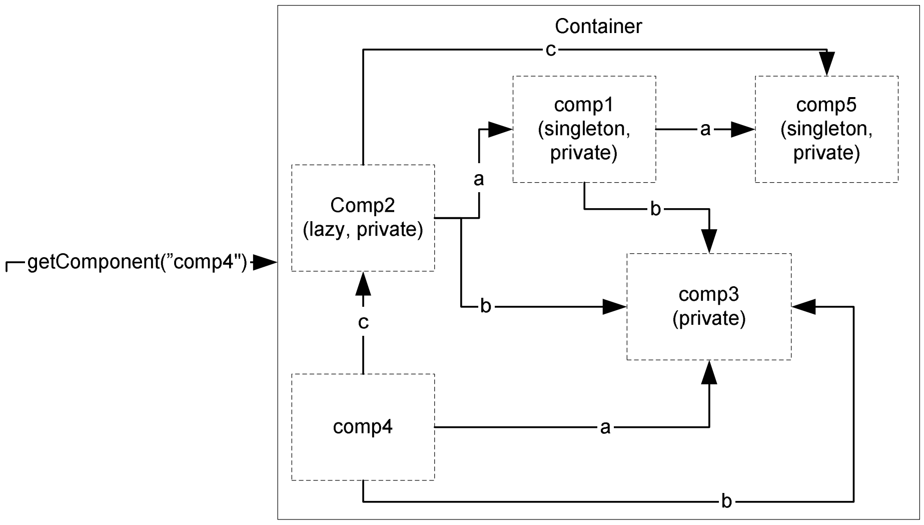
This section shows what happens during startup of the business application.

At some point in the overall initiation of the Business Application, a *Container Holder*, (as defined in section 2.2.7) is created. The Container Holder is responsible for creating, adding components to, and holding an EzcContainer and for providing an application-specific ContainerContext to the EzcContainer.

After executing Part 1 and Part 2 of Listing 1, there will be a container holding metadata for a component model as shown in Figure 1. The Java code for adding of components to the container (Part 2 of Listing 1) is a bit impractical and clumsy – configuring the container using Groovy, see chapter 3, is much more convenient.

Note that no component instances will be created yet. Components are created when they are referenced (or when they are *invoked*, if lazy instantiation is used) for the first time (either by being called upon via `getComponent` or by virtue of being a dependency to another component).

Figure 1 Metadata for the example component model

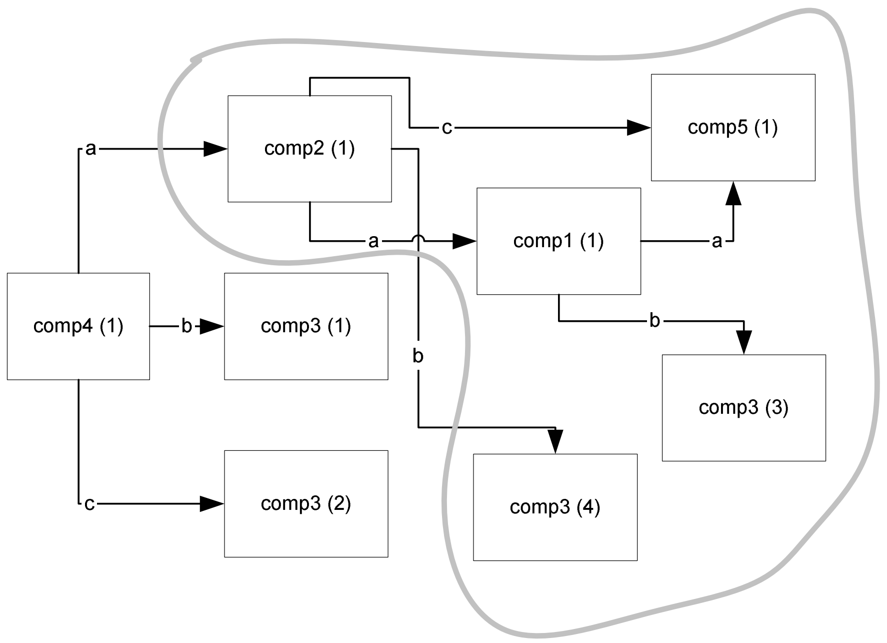


2.3.2 Invoking components in the container

This section shows how components are instantiated as a result of referring and invoking components in the container.

Part 3 of Listing 1 above, shows how `comp4` is referenced and subsequently invoked. Obtaining a reference to a component, will trigger the creation of a graph of component instances for all its dependencies, both direct and indirect (X has an indirect dependency on Z if X is dependent on Y and Y is dependent on Z). The graph in Figure 2 shows the instance graph resulting from referencing and invoking `comp4` for the first time.

Figure 2 Component instance graph resulting from referencing and invoking `comp4`.



Note that, since `comp2` is using lazy instantiation (see Listing 1, line 41|), the component instances in the grey area will not be created until `comp2` is invoked for the first time. This fact can be further illustrated by Listing 2 which shows all component instantiations and invocation in the order they occur when executing the getComponent/invoke operations.

Listing 2 The order in which component instances are created and invoked.

```
Creating Instance: comp3-1
Creating Instance: comp4-0
  Invoking: comp4-0
      Invoking: comp3-0
Creating Instance: comp3-2
Creating Instance: comp5-0
Creating Instance: comp1-0
Creating Instance: comp3-3
Creating Instance: comp2-0
      Invoking: comp2-0
          Invoking: comp1-0
              Invoking: comp5-0
              Invoking: comp3-2
          Invoking: comp5-0
          Invoking: comp3-3
      Invoking: comp3-1
```

2.4 Component naming, hierarchy and sub-containers

The EzcContainer supports the concept of sub-containers. An EzcContainer (`subContainer`) may be added to another container (`topContainer`) using the method `addContainer(prefix,container)`. This will cause all (non-private) components of subContainer to be accessible as components of topContainer by prepending a prefix to their component keys.

If `subContainer` contains the components `foo` and `bar` and are added to `topContainer` using `topContainer.addContainer("sub",subContainer)` the `foo` and `bar` components may be retrieved from `topContainer` using the prefix "`sub`".

```
getComponent("sub-foo")

getComponent("sub-bar")
```

Note how the character "-" is used as a special delimiter for container prefix and component key. As a result of this convention, the "-" characted are not allowed in component keys.

Components in `topContainer` may of course specify dependencies to components located in subContainers, just as they would to components in the same container. In order for this to work, the subContainer must be configured and added to topContainer *before* adding any components (to topContainer) which depends on subContainer components, since all stated dependencies are validated directly.

It is not possible for components in subContainer to access topContainer components.

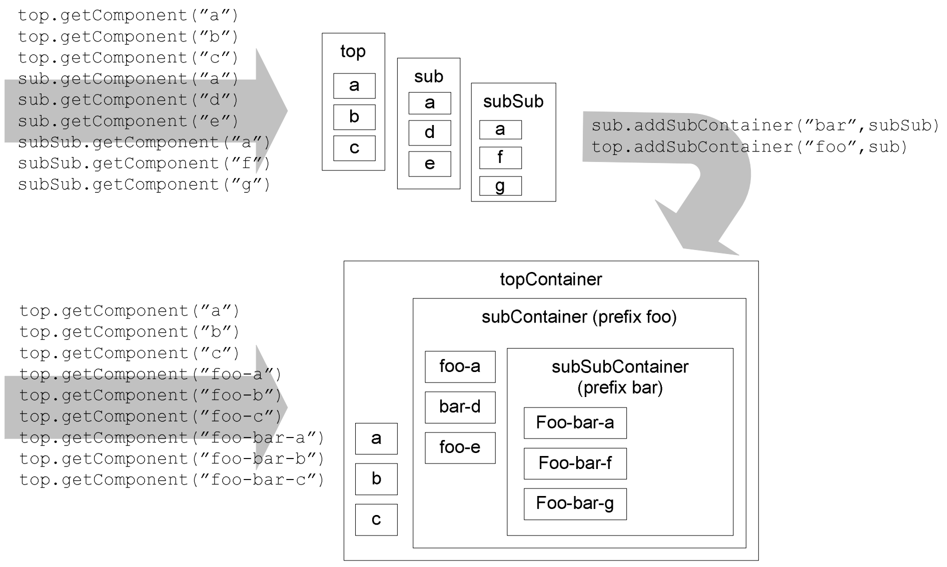
Containers may be nested in any number of levels, but not so that cycles are created. For example, the following would be illegal:

```
top.addContainer("sub,sub);

sub.addContainer("top",top); // ILLEGAL
```

Figure 3 shows how three separate containers are "merged" into one, which is nested in two levels, using multiple calls to `addContainer`. The left part of figure also shows how the code needed to access each components looks, before and after merging their containers.looking.

Figure 3


```
top.getComponent("a")
top.getComponent("b")
top.getComponent("c")
sub.getComponent("a")
sub.getComponent("d")
sub.getComponent("e")
subSub.getComponent("a")
subSub.getComponent("f")
subSub.getComponent("g")
```

```
sub.addSubContainer("bar",subSub)
top.addSubContainer("foo",sub)
```

```
top.getComponent("a")
top.getComponent("b")
top.getComponent("c")
top.getComponent("foo-a")
top.getComponent("foo-b")
top.getComponent("foo-c")
top.getComponent("foo-bar-a")
top.getComponent("foo-bar-b")
top.getComponent("foo-bar-c")
```

The use of sub-containers can be a convenient way of supporting the partitioning of a Business Application into subsystems or layers. One could for instance place all "business logic" in one container while using another for presentation components.

Note also that if nested EzcContainers are used in a Business Application, they may employ different component definitions (achieved by using different ContainerContexts).

3. Configuring the EzcContainer using Groovy

Groovy is a scripting language for the JVM. It is undergoing standardization under the Java Community Process (JSR-241). The following description of Groovy is found at the Groovy Home Page (http://groovy.codehaus.org):

> *Groovy is a new agile dynamic language for the JVM combining lots of great features from languages like Python, Ruby and Smalltalk and making them available to the Java developers using a Java-like syntax.*
>
> *Groovy is designed to help you get things done on the Java platform in a quicker, more concise and fun way - bringing the power of Python and Ruby inside the Java platform.*
>
> *Groovy can be used as an alternative compiler to javac to generate standard Java bytecode to be used by any Java project or it can be used dynamically as an alternative language such as for scripting Java objects, templating or writing unit test cases.*

EzcContainer uses Groovy as it preferred mechanism for configuration (however it may of course also be configured through plain Java). The main reason to use a scripting language for this kind of information is to enable configuration type information to be changed without recompiling the application.

By using Groovy (or another "Turing complete" scripting language) rather than inventing our own (XML) format, a lot of work (defining, coding and documenting yet another language is not that inspiring a task) can be saved. But laziness is not the only reason: the use of a real scripting language makes it possible for users to extend and adapt the initialization procedure as needed. Although there is only one way (the method `addComponent`) to add components to the container, it is an easy task defining your own helper functions using Groovy. Groovy is a powerful way of customizing the syntax of initialization to personal needs and preferences.

3.1 Simple Example

The listing below shows how a very simple groovy script for initializing an EzcContainer may look. Note how the object called "top" is a reference to an EzcContainer placed in the context of the Groovy Script by the invoking Java code (the method `initContainer` of `EzcContainerUtils`). From the view-point of the Groovy script "top" is the container to which components will be added.

Listing 3 Contents of (fictive) file my-app-config.groovy

```
top.addComponent("userManager", LdapUserManager.class, [UserManager.class],
SINGLETON+LAZY, [ "ldapServer": "primaryLdap"])
top.addComponent("primaryLdap", IbmDirectoryServer.class, [LdapServer.class],
PRIVATE+SINGLETON+LAZY, [ "config": [ "port": 389,
                                      "host": "foo.bar.com"]])
```

The following snippet shows how an application is initialized based upon the component definitions in the listing above:

Listing 4 Code for initiating and accessing and incoking components

```
public static void main(String[] argv) {
 // MyAppContainerContext is an application-specific class
 MyAppContainerContext containerCtx = new MyAppContainerContext();
 EzcContainer c = new EzcContainer(containerCtx);
 EzcContainerUtils.initContainer(c,”my-app-config.groovy”);
 // use container to access components and invoke their services
 UserManager userMgr = (UserManager)c.getComponent(“userManager”);

String user = argv[0];
String password = argv[1];
         if(userMgr.validateUser(user,password)) {
         System.out.println(”user validated!”);

} else {
         System.out.println(”failed validating user...”);

}
}
```

The code above reads component definitions and configuration from a file called `my-app-config.groovy` and with the help of `EzcContainerUtils` an EzcContainer with the correct component model is created. The code also shows how the main method obtains a reference to the component “userManager”. After casting this reference to the correct class, it may be used to invoke `validateUser` which does something useful.

Note how the main method can be considered being the Container Holder (as defined in section 2.2.7) and the lack of a Container Invoker.

3.2 Larger Example

This sections show a larger example of a groovy script used for container initialization.

```
FIXME

```

4. Integraing the EzcContainer in a Business Application

This chapter describes the easy steps which must be performed when adopting the EzcContainer in a Business Application.

1. components has to live by

Component Definition <-> ContainerContext

Good principle to let your own application provide at

5. Notes/Ideas on how the EzcContainer can be used

Some unsorted tips and ideas on the use of the EzcContainer.

- Use more than one container in one application.
- It is worth mentioning that using more than one container is containers

- As a basis for building more advanced containers.
- If using a Command pattern architecture, each Command may be added as a component to an EzcContainer and have commands relying on the execution of other commands may get those as
- Use sub-containers to integrate subsystems
- As with any framework, it may be worthwhile to minimize the concrete dependencies on it in your application code.