I was expecting a paradigm shift,
and all I got was a lousy constructor!

PicoContainer 1.2 documentation

For developers extending PicoContainer
and for developers Using PicoContainer

Preamble

These pages have been exported from Codehaus's documentation system, Confluence. Refer http://docs.codehaus.org/

There are some slight glitches to this method of documentation:

- Sometimes the text reads like it is still online.

- The Header, sub-header, sub-sub-header separations are not always clear.

- Links (that are clear in HTML) do not map well to printed forms where they just appear as blue phrases. Some PDF viewers (like Adobe's) will allow you to click through to the original web page though.

- There are still some comments inline that do not make much sense for a pseudo-printed media like PDF.

- We have artificially added '1. ' , '2. ' style prefixes to the page names to force a sort of order.

Either way, we hope you enjoy our **User Documentation** PDF

User Documentation

This page last changed on Sep 05, 2005 by paul.

Most useful

One minute description
Two minute tutorial
Five minute introduction

Full Content

1. Overview

This page last changed on Jan 25, 2006 by mauro.

Overview

PicoContainer is a lightweight and highly embeddable container for components that honour Dependency Injection.

This User Documentation is structured as follows.

We start from an analysis of patterns and anti-patterns, which highlight the reasons which prompted the development of PicoContainer.

We then introduce the container proper.

License

This page last changed on Jan 26, 2006 by mauro.

```
Copyright (c) 2003-2006, PicoContainer Organization
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

  Redistributions of source code must retain the above copyright notice, this
  list of conditions and the following disclaimer.

  Redistributions in binary form must reproduce the above copyright notice,
  this list of conditions and the following disclaimer in the documentation
  and/or other materials provided with the distribution.

  Neither the name of the PicoContainer Organization nor the names of its
  contributors may be used to endorse or promote products derived from this
  software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.
```

Sister Projects

This page last changed on Jan 25, 2006 by mauro.

NanoContainer

NanoContainer is essentially a scripting front end to PicoContainer. Script languages supported: XML, Groovy, BeanShell, Jython (Python for JVM) and Rhyno (JavaScript for JVM).

NanoContainer also adds to PicoContainer features such as:

- AOP
- Classloader management

- Integration to web and persistence frameworks

- Remoting (EJB, JMX, ...)

- ...and much more

See http://www.nanocontainer.org.

MicroContainer

MicroContainer is a micro-kernel that allows deployment of drop-in of self-contained archives of Pico-esque applications. The archive is similar in concept to an EAR file, and automatic JMX and publication is possible.

See http://www.microcontainer.org.

Note MicroContainer is not to be confused with "JBoss Microcontainer" - which came later.

2. Patterns and Anti-patterns

This page last changed on Jan 27, 2006 by paul.

Patterns and Anti-patterns of software developent are much the rage.

Pico tries to solve some anti-patterns, promote some patterns, but there are cases where people may go too far and introduce anti-patterns that are specific to those chasing Inversion of Control and Dependency Injection.

See Patterns and Antipatterns

2.1 Patterns

This page last changed on Jan 25, 2006 by mauro.

Key Design Patterns & Architectural Matters

IoC / Inversion of Control

the idea that an application is controlled from the top down

SoC / Separation of Concerns

the idea that a class (aspect) should do one job and do it well

SAI / SoAI / Separation of API from Implementation

the idea that you define and code to work interfaces

AOP / Aspect Oriented Programming

mostly lightweight nowadays where you add a chain of interceptors around a method call that can handle orthogonal concerns

COP / Component Oriented Programming

the idea that you decompose your software into components

DecP / Declarative Programming

where you use a declarative-style language (usually xml) to determine things like component wiring (i.e. your average tomcat config file, generalized)

EBP / Event Based Programming

basically making the inter-object method call asynchronous and encapsulating such a call into some kind of event object that can be queued, modified, etc

Patterns details here

- Good Citizen

Patterns detailed elsewhere

Separate Interfaces From Implementation

http://c2.com/cgi/wiki?SeparateInterfacesFromImplementation

Good Citizen

This page last changed on Feb 24, 2004 by rinkrank.

Least surprise, least paranoia

Authors: Dan North, Aslak Hellesoy

Imagine a software system where there is no need for you to spend your time programming defensively; your objects will be used responsibly, and your methods will always be passed sensible arguments.

This low-friction utopia can be approached by establishing some simple programming rules so that every class acts as a 'good citizen' in the society of classes collaborating at runtime.

This page outlines some rules that we, and others, believe lead to good citizenship. All are aimed at improving clarity, reducing surprise, and promoting basic consistency.

As a good citizen, I...

- Keep a consistent state at all times - init() or populate() is a code smell.
- Have no static fields or methods
- Never expect or return null.
- FailFast - even when constructing.
- Am Easy to test- all dependent object I use can be passed to me, often in my constructor (typically as Mock Objects).
- Accept dependent object that can easily be substituted with Mock Objects (I don't use Concrete Class Dependency).
- Chain multiple constructors to a common place (using this(...)).
- Always define hashCode() alongside equals()
- Prefer immutable value objects that I can easily throw away.
- Have a special value for 'nothing' - e.g. Collections.EMPTY_SET.
- Raise checked exceptions when the caller asked for something unreasonable - e.g. open a non-existant file.
- Raise unchecked exceptions when I can't do something reasonable that the caller asked of me - e.g. disk error when reading from an opened file.
- Only catch exceptions that can be handled fully.
- Only log information that someone needs to see.

Classes that are designed for Constructor Injection are better citizens than those that are not.

Interface Implementation Separation

This page last changed on Oct 09, 2004 by rinkrank.

Also see http://c2.com/cgi/wiki?SeparateInterfacesFromImplementation

Inversion of Control

This page last changed on Jan 25, 2006 by mauro.

Authors: Paul Hammant

Overview

Inversion of Control (IoC) is a design pattern that addresses a component's dependency resolution, configuration and lifecycle. Note to confuse things slightly, IoC is also relevant to simple classes, not just components, but we will refer to components throughout this text. The most significant aspect to IoC is dependency resolution and most of the discussion surrounding IoC dwells solely on that.

Types of IoC

There are many types of IoC, but we'll concentrate on the type of IoC that PicoContainer introduced to the community. Formerly known as type-3, now known as Constructor Injection. We'll ignore Setter Injection and Contextualized Lookup as they are described in the IoC Types page.

IoC Synonyms

DIP

One well-known synonym for IoC is DIP - described in Robert C. Martin's excellent Dependency Inversion Principle paper.

Hollywood Principle

A second nickname for IoC is The Hollywood Principle (Don't call us we'll call you).

IoC History

Some detail about the history of Inversion of Control - IoC History

Component Dependencies

It generally favors loose coupling between components. Loose coupling in turn favours:

- More reusable classes
- Classes that are easier to test
- Systems that are easier to assemble and configure

Explanation

Simply put, a component designed according to IoC does not go off and get other components that it needs in order to do its job. It instead *declares* these dependencies, and the container supplies them. Thus the name IoC/DIP/Hollywood Principle. The control of the dependencies for a given component is inverted. It is no longer the component itself that establishes its own dependencies, but something on the outside. That something could be a container like PicoContainer, but could easily be normal code instantiating the component in an embedded sense.

Examples

Here is the simplest possible IoC component :

```
public interface Orange {
  // methods
}
public class AppleImpl implements Apple {
  private Orange orange;
  public AppleImpl(Orange orange) {
    this.orange = orange;
  }
  // other methods
}
```

Here are some common smells that should lead you to refactor to IoC :

```
public class AppleImpl implements Apple{
  private Orange orange;
  public Apple() {
    this.orange = new OrangeImpl();
  }
  // other methods
}
```

The problem is that you are tied to the OrangleImpl implementation for provision of Orange services. Simply put, the above apple cannot be a (configurable) component. It's an application. All hard coded. Not reusable. It is going to be very difficult to have multiple instances in the same classloader with different assembly.

Here are some other smells along the same line :

```
public class AppleImpl implements Apple {
  private static Orange orange = OrangeFactory.getOrange();
  public Apple() {
  }
  // other methods
}
```

Component Configuration

Sometimes we see configuration like so ...

```
public class BigFatComponent {
  String config01;
```

```
    String config02;
    public BigFatComponent() {
      ResourceFactory resources = new ResourceFactory(new File("mycomp.properties"));
      config01 = resources.get("config01");
      config02 = resources.get("config02");
    }
    // other methods
  }
```

In the IoC world, it might be better to see the following for simple component designs :

```
  public class BigFatComponent {
    String config01;
    String config02;
    public BigFatComponent(String config01, String config02) {
      this.config01 = config01;
      this.config02 = config02;
    }
    // other methods
  }
```

Or this for more complex ones, or ones designed to be more open to reimplementation ..

```
  public interface BigFatComponentConfig {
    String getConfig01();
    String getConfig02();
  }
  public class BigFatComponent {
    String config01;
    String config02;
    public BigFatComponent(BigFatComponentConfig config) {
      this.config01 = config.getConfig01();
      this.config02 = config.getConfig02();
    }
    // other methods
  }
```

With the latter design there could be many different implementations of BigFatComponentConfig. Implementations such as:

1. Hard coded (a default impl)
2. Implementations that take config from an XML document (file, URL based or inlined in using class)
3. Properties File.

It is the deployer's, embeddor's or container maker's choice on which to use.

Component Lifecycle

Simply put, the lifecycle of a component is what happens to it in a controlled sense after it has been instantiated. Say a component has to start threads, do some timed activity or listen on a socket. The component, if not IoC, might do its start in its contructor. Better would be to honor some start/stop functionality from an interface, and have the container or embeddor manage the starting and stopping when they feel it is appropriate:

```
  public class SomeDaemonComponent implements Startable {
    public void start() {
```

```
        // listen or whatever
      }
      public void stop() {
      }
      // other methods
    }
```

Notes

The lifecycle interfaces for PicoContainer are the only characterising API elements for a component. If Startable was in the JDK, there would be no need for this. Sadly, it also menas that every framework team has to write their own Startable interface.

The vast majority of components do not require lifecycle functionality, and thus don't have to implement anything.

IoC Exceptions

Of course, in all of these discussions, it is important to point out that logging is a common exception to the IoC rule. Apache has two static logging frameworks that are in common use: Commons-Logging and Log4J. Neither of these is designed along IoC lines. Their typical use is static accessed whenever it is felt appropriate in an application. Whilst static logging is common, the PicoContainer team do not recommend that developers of reusable components include a logging choice. We suggest instead that a Monitor component interface is created and default adapters are provided to a number of the logging frameworks are provided.

Subpages

- Contextualized Lookup
  - Avalon Framework
- IoC History
- IoC Types
  - Dependency Injection
    - Constructor Injection
    - Setter Injection

Contextualized Lookup

This page last changed on Jan 25, 2006 by paul.

Authors: Paul Hammant

Overview

Contextualized Dependency Lookup is the type of dependency resolution that Avalon, Loom, Keel and alike do. It is where the container or embedder hands dependencies after instantiation and via an interface/method that the component implements to indicate that it has dependencies. The method in question typically hands in an reference to the component that has a lookup mechanism. Only the component dependencies declared will be returned on lookup. Typically XML is used to describe the dependancies for a particular component.

Example

```
import org.apache.avalon.framework.ServiceManager;
import org.apache.avalon.framework.Serviceable;
import org.apache.avalon.framework.ServiceException;

public class Shop implements Serviceable, Initializable {
    StockManager stockManager;
    String shopZipCode;
    public void service(ServiceManager sm) throws ServiceException {
        stockManager = (StockManager) sm.lookup("StockManager");
    }
    public void initialize() {
        // all service()ing done.
    }
}
```

A component has to have service (component) declarations in an external file. The popular Loom (forked from Phoenix) container has .xinfo files for such needs. All Avalon container have some mechanism for storing configuration and assembly externally to the class. Cross referenced against the xinfo files, Loom's assembly.xml defines the implementations to be used for component types. This all Avalon components must be interface/implementation separated. This in itself is not a bad thing in my (Paul) opinion.

Using Contextualized Dependency Lookup Components Without a Container.

The downside of the this design is that components can only be used without the container with great difficulty. If at all. This a proper container is needed at all times, and you have to choose one for different purposes (that not withstanding the efforts of the Avalon team to make a single all-purpose container). If you do manage to instantiate components without a container, you might miss one of the essential service dependencies. The component-using class will continue to compile, but at run time it will be apparent that there are missing dependencies.

Container support

The Avalon and JContainer's Loom are the best example of containers that support contextulaized lookup. PicoContainer does not directly support this type of dependency resultion. See Avalon Framework page to see how this works.

This page last changed on Jan 25, 2006 by mauro.

Authors: Paul Hammant

Overview

Apache's Avalon framework specification is enshrined in a number of interfaces. What this means is that the component writer has to implement them to designate their class as an Avalon component. PicoComponents require adaptation to fit the Avalon Framework (contextualzed lookup) design.

A simple example component

Example component APIs

```
public interface Engine {
  void runEngine();
}
public interface Persistor {
  void persist(String key, Object data);
}
```

A simple Pico implementation of the hypothetical engine.

```
public class EngineImpl implements Engine {
  Persistor persistor;
  String persistenceKey;
  Object persistable;
  public void EngineImpl(Persistor persistor, String persistenceKey) {
    this.persistor = persistor;
    this.persistorKey = persistorKey;
    persistable = new Object(); // not very 'heavy' we appreciate.
  }
  public void runEngine() {
  {
    persistor.persist(persistorKey, persistable);
  }
}
```

The same component natively written for Apache Avalon

```
public class AvalonEngine implements Engine, Servicable, Configurable, Initializable {
  Persistor persistor;
  String persistenceKey;
  Object persistable;
```

```
    public void service (ServiceManager sm) throws ServiceException {
      this.persistor = (Persistor) sm.lookup("Persistor");
    }
    public void configure(Configuration conf) {
      this.persistorKey = conf.getAttribute("persistorKey");
    }
    public void initialize() {
      persistable = new Object(); // not very 'heavy' we appreciate.
    }
    public void runEngine() {
    {
      persistor.persist(persistorKey, persistable);
    }
  }
```

An alternate wrapping strategy for Apache Avalon compatability.

```
  public class AvalonEngine implements Engine, Servicable, Configurable, Initializable {
    private Engine engine;
    // temporary
    private Persistor persistor;
    private String persistenceKey;
    public void service (ServiceManager sm) throws ServiceException {
      this.persistor = (Persistor) sm.lookup("Persistor");
    }
    public void configure(Configuration conf) {
      this.persistorKey = conf.getAttribute("persistorKey");
    }
    public void initialize() {
      engine = new EngineImpl(persistor persistenceKey);
    }
    public void runEngine() {
    {
      engine.runEngine();
    }
  }
```

IoC History

This page last changed on Jan 25, 2006 by paul.

Authors: Paul Hammant

The history of PicoContaner is tied to that of IoC (Inversion of Control) itself. There are three main types of IoC. The names for these have been recently evolved by Martin Fowler and friends and publicised here.

1. Contextualized Lookup
2. Setter Injection
3. Constructor Injection

One, two and three were roughly evolved in that order. One has been dominant for a number of years. PicoContainer authors have either formerly promoted it, or avoided it because it felt overly complicated.

Two and three came into being within a short space of each other in the last year or so. Both were created as part of a quest for transparency and simplicity.

IoC Types

This page last changed on Jan 25, 2006 by paul.

Overview

In recent years different approaches have emerged to deliver an IoC vision. Latter types, as part of a 'LightWeight' agenda have concentrated on simplicity and transparency.

IoC Types - Family Tree

Devised at ThoughtWorks' London offices in December of 2003. Present at the "Dependency Injection" meeting were Paul Hammant, Aslak Hellesoy, Jon Tirsen, Rod Johnson (Lead Developer of the Spring Framework), Mike Royle, Stacy Curl, Marcos Tarruela and Martin Fowler (by email).

Inversion of Control

- Dependency Injection (former type 3)
  - Constructor Dependency Injection
    Examples: PicoContainer, Spring Framework, (EJB 3.1 ?)
  - Setter Dependency Injection (former type 2)
    Examples: Spring Framework, PicoContainer, EJB 3.0
  - Interface Driven Setter Dependency Injection
    Examples: XWork, WebWork 2
  - Field Dependency Injection
    Examples: Plexus (to be confirmed)
- Dependency Lookup
  - Pull approach (registry concept)
    Examples: EJB 2.x that leverages JNDI, Servlets that leverage JNDI
  - Contextualized Dependency Lookup (former type 1)
    AKA Push approach
    Examples: Servlets that leverage ServletContext, Apache's Avalon, OSGi (to be confirmed), Keel (uses Avalon)

See also Constructor Injection, Setter Injection, Contextualized Lookup for more information.

Note Field Injection was provisionally known as 'type 4'. There was really no interest 'type 4' until EJB3.0. Getter Injection flourished for a while, but did not take and was never supported by the PicoContainer team.

Examples of Common Types

Constructor Dependency Injection

This is where a dependency is handed into a component via its constructor :

```
public interface Orange {
  // methods
}

public class AppleImpl implements Apple {
  private Orange orange;
  public AppleImpl(Orange orange) {
    this.orange = orange;
  }
  // other methods
}
```

Setter Dependency Injection

This is where dependencies are injected into a component via setters :

```
public interface Orange {
  // methods
}

public class AppleImpl implements Apple {
  private Orange orange;
  public void setOrange(Orange orange) {
    this.orange = orange;
  }
  // other methods
}
```

Contextualized Dependency Lookup (Push Approach)

This is where dependencies are looked up from a container that is managing the component :

```
public interface Orange {
  // methods
}

public class AppleImpl implements Apple, DependencyProvision {
  private Orange orange;
  public void doDependencyLookup(DependencyProvider dp) throws DependencyLookupExcpetion{
    this.orange = (Orange) dp.lookup("Orange");
  }
  // other methods
}
```

Terms: Service, Component & Class

Component is the correct name for things managed in an IoC sense. However very small ordinary classes are manageable using IoC tricks, though this is for the very brave or extremists

A component many have dependencies on others. Thus dependency is the term we prefer to describe the needs of a component.

Service as a term is very popular presently. We think 'Service' dictates marshaling and remoteness. Think of Web Service, Database service, Mail service. All of these have a concept of adaptation and transport. Typically a language neutral form for a request is passed over the wire. In the case of the Web Service method requests are marshaled to SOAP XML and forward to a suitable HTTP server for processing. Most of the time an application coder is hidden from the client/server and marshaling ugliness by a toolkit or API.

Obsoleted Terms

Types 1, 2 and 3 IoC were unilaterally coined earlier in 2003 by the PicoContainer team and published widely.

Type 1 becomes Contextualized Dependency Lookup
Type 2 becomes Setter Dependency Injection
Type 3 becomes Constructor Dependency Injection

Dependency Injection versus Contextualized Lookup

Dependency Injection is non-invasive. Typically this means that components can be used without a container or a framework. If you ignore life cycle, there is no import requirements from an applicable framework.

Contextualized Dependency Lookup is invasive. Typically this means components must be used inside a container or with a framework, and requires the component coder to import classes from the applicable framework jar.

Note that Apache's Avalon (and all former type-1 designs) are not Dependency Injection at all, they are Contextualized Dependency Lookup.

What's wrong with JNDI ?

With plain JNDI, lookup can be done in a classes' static initialiser, in the constuctor or any method including the finaliser. Thus there is no control (refer C of IoC). With JNDI used under EJB control, and concerning only components looked up from that bean's sisters (implicitly under the same container's control), the specification indicates that the JNDI lookup should only happen at a certain moment in the startup of an EJB application, and only from a set of beans declared in ejb-jar.xml. Hence, for EJB containers, the control element should be back. Should, of course, means that many bean containers have no clue as to when lookups are actually being done, and apps work by accident of deployment. Allowing it for static is truly evil. It means that a container could merely be looking at classes with reflection in some early setup state, and the bean could be going off and availing of remote and local services and components. Thus depending whether JNDI is being used in an Enterprise Java Bean or in a POJO, it is either an example of IoC or not.

Dependency Injection

This page last changed on Jan 25, 2006 by paul.

Overview

See Martin Fowlers's "Inversion of Control Containers and the Dependency Injection pattern" article for a thorough description.

Constructor Dependency Injection

See Constructor Injection

Setter Dependency Injection

See Setter Injection

Field Dependency Injection

TODO

Constructor Injection

This page last changed on Sep 10, 2004 by rinkrank.

Authors: Paul Hammant, Aslak Hellesoy

Overview

Constructor Injection is a Dependency Injection variant where an object gets all its dependencies via the constructor. This is PicoContainer's most important feature.

The most important benefits of Constructor Injection are:

- It makes a strong dependency contract
- It makes testing easy, since dependencies can be passed in as Mock Objects
- It's very succinct in terms of lines of code
- Classes that rely on Constructor Injection are generally Good Citizens
- A dependency may be made immutable by making the dependency reference *final*

Martin Fowler explains Constructor Injection in more detail.
PicoContainer also supports Setter Injection.

Origin

Rachel Davies, while reviewing Joe's book, left a Fermat-like margin note when view a snippet like the above. "Why not use constructors ?". Brilliant and simple.

Example

```
public class Shop {
    private final StockManager stockManager;
    private final String shopZipCode;
    public Shop(StockManager stockManager, String shopZipCode) {
        this.stockManager = stockManager;
        this.shopZipCode = shopZipCode;
    }
 }
```

Note, for this there is no need to declare needs in any other way. No interfaces, no doclet tags, no external XML. Just your simple component(s) and PicoContainer. No need for post assembly/config initialization either. If it is constructed (not withstanding some asserts on nulls) it has its needs satisfied. Components need not be interface/implementation separated. This is the coder's choice.

Using Constructor Injector Components Without a Container.

The component can be used directly, without any container. The missing dependency scenario is not an issue since it is impossible to instantiate an object without all dependencies being satisfied.

```
Shop shop = new Shop(myStockManager);
```

Container support

PicoContainer was the first lightweight container to support and popularize this for of dependency injection. Spring Framework has been retrofitted with constructor injection capability, but its primary focus is still setter injection. Even Avalon's reference container has been upgraded to have compatibility with constructor injection components.

This page last changed on May 21, 2004 by rinkrank.

Authors: Paul Hammant, Aslak Hellesoy, Philipp Meier

Overview

Setter Injection is where the container or embedder hands dependencies to a component via setter methods after instantiation.

Example

Joe Walnes whist working on Java Open Source Programming with other luminaries, started a Setter Injection IoC design. This is marked up with doclet tags (though that is not hard and fast) :

```
public class Shop {
   StockManager stockManager;
   String shopZipCode;
   /**
    * @service name="StockManager"
    */
   public void setStockManager(StockManager stockManager) {
       this.stockManager = stockManager;
   }
   /**
    * @config name="shopZipCode"
    */
   public void setStockManager(String shopZipCode) {
       this.shopZipCode= shopZipCode;
   }
   // TODO - Joe - how does setter injector do config ? Same way?
   public void initialize() {
       // all setXXXs are now done :-)
   }
}
```

The container use the meta-information to resolve all the dependencies. Components need not be interface/impl separated. Developer's choice.

Using Setter Injector Components Without a Container.

Setter Injection components can be used directly, without any container. The component-using class will continue to compile, but at run time it will be apparent that there are missing dependencies. The downside of this is that a developer may miss a setXXX(..) method invocation if they are using the component directly. That is fairly small as a risk as it would clearly be caught in the development cycle. Caught in the development cycle, but maybe obscurely so with a NullPointerException.

```
Shop shop = new Shop();
shop.setStockManager(myStockManager);
```

Container support

The Spring Framework project is the best example of a container that supports setter injector. PicoContainer does too, but we really believe that constructor injector is superior.

Refs + Comparison

Setter Injection is a Dependency Injection variant where an object gets all dependencies via setter methods. PicoContainer support this with SetterInjectionComponentAdapter, but the PicoContainer team recommends Constructor Injection.

The advantage of Constructor Injection is that the setting is atomic in a sense that either all or none of the dependencies are set and that it can occur once and only once. With Setter Injection there is the possibility to forget to set some of the dependencies

2.2 Antipatterns

This page last changed on Jan 25, 2006 by mauro.

An anti-pattern is a design solution which creates more problems than it solves.

Best design practices try to avoid anti-patterns. PicoContainer's design is driven by adherence to best practices.
Some of these anti-patterns are not implementable using PicoContainer but some actually are. We strongly discourage
them and are listed below as a reminder of pitfalls to avoid.

- Concrete Class Dependency
- Container Dependency
- Container Instantiation
- Instance Registration
- Long Constructor Argument List
- Propagating Dependency
- Singleton

Concrete Class Dependency

This page last changed on Oct 09, 2004 by rinkrank.

Symptoms

A class depends on other concrete classes. In order to favour decoupling (and thereby testability) we recommend depending on interfaces instead.

```
public class A {
    private final B b;

    public A(B b) {
        this.b = b;
    }
}

public class B {
}
```

Causes

Laziness

What to do

In order to reduce A's tight coupling, split B in an Interface Implementation Separation.

```
public interface B {
}

public class BImpl implements B {
}
```

Container Dependency

This page last changed on Apr 01, 2005 by rinkrank.

Symptoms

Classes that depend on the container.

Consider the following example. We have a class BImpl that requires an A instance. It declares the dependency on the container so it can look up that A:

```
public interface A {
}

public class AImpl implements A {
}

public class BImpl implements B {
    private final A a;

    BImpl(PicoContainer pico) {
        a = (A) pico.getComponentInstanceOfType(A.class);

        /*
        alternatively:
        a = (A) pico.getComponentInstance("a");
        */
    }
}
```

It can be used in the following way:

```
MutablePicoContainer pico = new DefaultPicoContainer();
pico.registerComponentImplementation("a", AImpl.class);
pico.registerComponentImplementation("b", BImpl.class);
pico.registerComponentInstance(pico);

...
B b = (B) pico.getComponentInstance("b");
```

This will work, but it's an antipattern.

The reasons why the above implementation of BImpl is an antipattern are:

- It introduces an unneeded dependency from BImpl to the container.
- This makes BImpl hard to test without a container.
- B assumes that the container has a registered an A. As a result, B won't FailFast if it hasn't. Instead, a will reference null, and BImpl will FailLate.

Causes

Not sure. Poor understanding of how PicoContainer works? Not being able to think "simple"?

What to do

The simple and elegant solution to this antipattern is not to complicate the world more than it is. Here is how it should be:

```
public class BImpl implements B {
    private final A a;

    BImpl(A a) {
        this.a = a;
    }
}
```

PicoContainer will figure out that BImpl needs an A instance, and will pass in the AImpl, as this is an implementation of A.

Container Instantiation

This page last changed on Jan 25, 2006 by mauro.

Symptoms

A class is directly instantiating a PicoContainer and registering components within it.

Causes

This smell is most often found in unit tests. It could be as a result of Container Dependency. The container is supplied to the class being tested, which then uses it to locate its dependencies.

Another cause is using the container to build the object we are testing, which itself may have several dependencies. For example::

```
public void testCocktailWithVodkaIsAlcoholic() {
    DefaultPicoContainer container = new DefaultPicoContainer();
    container.registerComponentImplementation(Banana.class);
    container.registerComponentImplementation(Vanilla.class);
    container.registerComponentImplementation(Vodka.class);
    container.registerComponentImplementation(Cocktail.class);

    Cocktail cocktail = (Cocktail) container.getComponentInstance(Cocktail.class);

    assertTrue(cocktail.isAlcoholic());
}
```

What To Do

For unit tests like this, the class being tested should be instantiated directly by the test. Mock Objects should be used to "mock" the dependent objects, i.e. supplying a fake implementation that can have expectations set and verified on it, rather than a real implementataion. So, the test becomes:

```
public void testCocktailWithVodkaIsAlcoholic() {

    Banana banana = createMockBanana();
    Vanilla vanilla = createMockVanilla();
    Vodka vodka = createMockVodka();

    // set expectations on banana, vanilla and vodka here.

    Cocktail cocktail = new Cocktail(banana, vanilla, vodka);
    assertTrue(cocktail.isAlcoholic());

    // verify expectations on banana, vanilla and vodka here.
}
```

The implementation details of creating a mock object and setting and verifying expectations have been left out of the example, as the details depend on which mock object library/technique is used.

Exceptions

The container has to be instantiated somewhere!

Bootstrappers

A common place to instantiate containers is in some bootstrap class in the application.

Functional Tests

There may be a requirement to write a high-level "functional" test, in which you wish to test the interactions between a set of real components (not mocks). In this case, you may wish to create a container with the appropriate components registered for your test.

NanoContainer

If you are using NanoContainer, you can use NanoContainer's Standalone class (a bootstrapper) to start the application. All the container configuration can live in a script and NanoContainer (and thereby PicoContainer(s)) will be instantiated.

Instance Registration

This page last changed on Feb 19, 2004 by rinkrank.

Too much use of registerComponentInstance(Object key, Object componentInstance) is an antipattern. When you are using this method, you are not taking advantage of PicoContainer's Dependency Injection mechanism. It should only be used as a last resort.

Long Constructor Argument List

This page last changed on Jun 19, 2004 by rinkrank.

Symptoms

The constructor for an object takes a long list of arguments.

Causes

- Can be a result of Propagating Dependency.
- Class is concerned with doing more than one thing and therefore has many dependencies.

What To Do

If the class is doing too much, it should be refactored into a set of smaller classes. Each class should have a clearly defined responsibility and therefore a smaller set of dependencies.

Propagating Dependency

This page last changed on Feb 19, 2004 by rinkrank.

Symptoms

A dependency exists solely to be propagated to another class, but no methods are called upon the dependent object itself. For example:

```
public class AControllerImpl implements AController {

    private SomeService service;
    private int someId;

    public AControllerImpl(SomeService service) {
        this.service = service;
        this.someId = generateRandomNumber();
    }

    public launchAnotherController() {
        AnotherController anotherController = new AnotherControllerImpl(service, fooId);
        anotherController.launch();
    }

    // ...
}
```

In this example, no method-calls are made upon 'service', it is simply propagated to the constructor of 'AnotherControllerImpl'. Therefore, it is not a valid dependency for 'AControllerImpl'.

Causes

DependencyInjection has been partially applied to a hierarchy of classes. This could be because some classes in the hierarchy depend upon instance state not available at container-registration time.

What To Do

Apply DependencyInjection to 'AControllerImpl' by replacing the dependency on 'SomeService' with a dependency on 'AnotherController'. If, as in the example above, 'AnotherControllerImpl' has a dependency upon some state that is not available at container-registration time, then we need to introduce a factory for creating 'AnotherController' as follows:

TODO: This is maybe a little contrieved for this example. Maybe remove or simplify. (AH).

```
public class AControllerImpl implements AController {

    private AnotherControllerFactory anotherControllerFactory;
    private int someId;

    public AControllerImpl(AnotherControllerFactory anotherControllerFactory) {
        this.anotherControllerFactory = anotherControllerFactory;
        this.someId = generateRandomNumber();
    }
```

```
    public launchAnotherController() {
        AnotherController anotherController =
anotherControllerFactory.createAnotherController(someId);
        anotherController.launch();
    }

    // ...
}
```

'AnotherControllerFactory' is an interface:

```
public interface AnotherControllerFactory {
    AnotherController createAnotherController(int someId);
}
```

It can be implemented as follows::

```
public class AnotherControllerFactoryImpl implements AnotherControllerFactory {
    private SomeService service;

    public AnotherControllerFactoryImpl(SomeService service) {
        this.service = service;
    }

    public AnotherController createAnotherController(int someId) {
        return new AnotherControllerImpl(service, someId);
    }
}
```

Now we can register both 'AControllerImpl' and 'AnotherControllerFactoryImpl' in the container. When 'AControllerImpl' is instantiated, it is supplied with an implementation of 'AnotherControlFactory' that it can use to create an 'AnotherController' instance.

Exceptions

When Migrating from executors to services, it can sometimes be difficult to avoid introducing a Propagating Dependency. In these cases, the Propogating Dependency can be considered as a good first step towards PicoFication of a set of classes. An effort should be made to complete PicoFication at some stage by making a series of further steps as described above.

Singleton

This page last changed on Jan 25, 2006 by mauro.

The singleton pattern was detailed in the GoF "Design Patterns" book. Because of its static nature and public availability, it allows component writers to obscurely reference other components. Overuse makes for bad solutions. At the enterprise level, it makes for very very bad solutions.

We claim that the GoF Singleton pattern is in fact an antipattern. The downside of the singleton antipattern is that classes depending on it often end up depending on everything and the kitchen sink. Singletons cannot be replaced with Mock Objects for the sake of easy unit testing.

With PicoContainer we would replace this with a container managed single instance, possibly in a container hierarchy (see Five minute introduction and Caching).

Quite often with J2EE solutions, the component model is honored to a degree for the sake of external APIs, but is sidestepped for the sake of the internals of the application. Session beans Foo and Bar are likely to leverage a hairball of singletons to achieve their ends. Systems developed along these lines, rapidly become entangled and unmaintainable. These entangled systems can also be referred to as

- Raymen Noodle Design
- Big Ball of Mud http://www.laputan.org/pub/foote/mud.pdf
- Spaghetti Code

In case its not clear, Singletons as a core feature of a component design are mutually exclusive with Inversion of Control.

3. PicoContainer

This page last changed on Jan 25, 2006 by mauro.

3.1 Container Overview

This page last changed on Jan 25, 2006 by mauro.

The container in PicoContainer has the following responsibilities:

- instantiating and assembling components through Dependency Injection,
- storing component instances accesible by keys,
- maintaining a components lifecycle,
- delegating to the parent for components not found locally.

Assembling components

The container instantiates a new instance of a component when it is requested. If the component have any dependencies the container looks for and instantiates all of them too.

This behaviour can be customized using component adapters, see Component Adapters and Factories for more information.

Storing component instances

The container stores a reference to the component instance as they are instantiated. You can access this instance using the key that the component was registered with.

This behaviour can also be customized using component adapters, see Component Adapters and Factories for more information.

Maintaining lifecycle

The container can also enforce a simple lifecycle (start, stop, dispose) on components contained within it. It is further described on 3.3 Lifecycle.

Container hierarchies

Each container can have a parent and a container can also be a component in another container. This effectively creates an hierarchy of containers.

The parent is used for resolving components not found locally. So if a container is looking for a component to resolve a dependency it looks first in locally in it's own container and then looks for the same component in the parent container. The opposite is not true though, a parent container never looks for a component in any of it's child-containers.

By placing a container in a child-container you override the same component if it exists in a sub-container. This is only true for local components or components in child-containers, components placed in a parent-container do not use the overridden component.

API Overview

Class	Description
PicoContainer	Is an interface that gives you read-only access to components in the container.
MutablePicoContainer	Is also an interface that extends PicoContainer and also allows components to be registered.
DefaultPicoContainer	This is the default implementation in the PicoContainer project.
ImplementationHidingPicoContainer	This is an alternate implementation of MutablePicoContainer. It hides implementations of components where it can
NanoContainer implementations	There are additional implementations in NanoContainer

3.2 Container Concepts

This page last changed on Jan 25, 2006 by mauro.

Containers have a number of consituent parts to their operation. Ordinary use of PicoContainer does not require much knowledge of those parts, nor the concepts behind them:

```
MutablePicoContainer pico = new DefaultPicoContainer();
```

But the advanced user may exploit the full power of PicoContainer but specifying custom constituent parts:

```
MutablePicoContainer pico = new DefaultPicoContainer(new FooComponentAdapterFactory(), new
BarComponentMonitor());
```

Constituent Parts of PicoContainer

- Component Adapters and Factories
- Component Monitors
- Lifecycle Managers and Strategies
- Parent and Child Containers

Caching

By default PicoContainer caches all registered component. See Caching for more details.

Caching

This page last changed on Sep 03, 2005 by paul.

Caching is where a container, if asked for a component a second time, gives the same instance. If you had design relying on Singletons and wanted to move to the DI age, this concept for you. Once interface/impl separated, with public static methods, and under caching Pico control, they become managed single instances.

CachingPicoContainer

This is a convenience class that implements all of the functionality of DefaultPicoContainer, but with the guaranteed caching of instances of components:

```
CachingPicoContainer pico = new CachingPicoContainer();
pico.registerComponentImplementation(List.class, ArrayList.class);
// other resitrations

Object one = pico.getComponentInstanceOfType(List.class);
Object two = pico.getComponentInstanceOfType(List.class);

assertSame("instances should be the same", one, two);
```

It does not matter if you choose another ComponentAdapterFactory (CAF), the combination of CachingPicoContainer and that CAF will still cache:

```
CachingPicoContainer pico = new CachingPicoContainer(new
ConstructorInjectionComponentAdapterFactory());
pico.registerComponentImplementation(List.class, ArrayList.class);
// other resitrations

Object one = pico.getComponentInstanceOfType(List.class);
Object two = pico.getComponentInstanceOfType(List.class);

assertSame("instances should be the same", one, two);
```

DefaultPicoContainer and CachingComponentAdapterFactory

DefaultPicoContainer (DPC) also caches by default:

```
DefaultPicoContainer pico = new DefaultPicoContainer();
pico.registerComponentImplementation(List.class, ArrayList.class);
// other resitrations

Object one = pico.getComponentInstanceOfType(List.class);
Object two = pico.getComponentInstanceOfType(List.class);

assertSame("instances should be the same by default", one, two);
```

But if passed a specific ComponentAdapterFactory that does not cache, no caching will occur:

```
DefaultPicoContainer pico = new DefaultPicoContainer(new
```

```
    ConstructorInjectionComponentAdapterFactory());
    pico.registerComponentImplementation(List.class, ArrayList.class);
    // other resitrations

    Object one = pico.getComponentInstanceOfType(List.class);
    Object two = pico.getComponentInstanceOfType(List.class);

    assertNotSame("instances should NOT be the same", one, two);
```

Caching with DPC can be done the hard way:

```
    DefaultPicoContainer pico = new DefaultPicoContainer(
                    new CachingComponentAdapterFactory(new
    ConstructorInjectionComponentAdapterFactory()));
    pico.registerComponentImplementation(List.class, ArrayList.class);
    // other resitrations

    Object one = pico.getComponentInstanceOfType(List.class);
    Object two = pico.getComponentInstanceOfType(List.class);

    assertSame("instances should be the same", one, two);
```

This introduced the secret of CachingPicoContainer. The CachingComponentAdapterFactory that is first is a chain of CAFs. For CachingPicoContainer is is secretly added to be first in the chain, for the snippet above for DPC, it was explicitly added. By this mechanism, any CAF for DPC can be instructed to cache. The default CAF for DPC to use when one is not explicitly injected, is DefaultComponentAdapterFactory which caches via CachingComponentAdapterFactory, and otherwise uses ConstructorInjectionComponentAdapters for components.

This page last changed on Jan 09, 2006 by joehni.

Authors: Jörg Schaible

The code snippets use the classes from the Five Minute Introduction.

Basic Building Blocks of PicoContainer

Looking at a PicoContainer you can compare it to a simple hash map, that delivers a component instance when requesting one with a special key. While this is the simplistic view from the user's point of view, it is necessary to understand the internal building blocks of such a PicoContainer, that allows it to create new instances and resolving their dependencies and to be able to support all kind of components necessary to build a complex application.

A PicoContainer is actually very simple. All it knows about a single component is

- the key
- the general type if the key is a type
- the real type of the component

With this information the PicoContainer can be requested for a component with a special key, for a component of a special type or for all components with a special type. Although the request is directed at the PicoContainer, it cannot instantiate any component by itself. This task is actually carried out by a ComponentAdapter. Such an adapter is created every time you register a component in a PicoContainer (actually in a MutablePicoContainer, since the PicoContainer interface is immutable):

```
MutablePicoContainer picoContainer = new DefaultPicoContainer();
picoContainer.registerComponentImplementation(Juicer.class);
picoContainer.registerComponentImplementation("My Peeler", Peeler.class);
picoContainer.registerComponentInstance(new Apple());
```

Internally the PicoContainer has created two instances with the DefaultComponentAdapterFactory and one instance of a InstanceComponentAdapter (since the last component is an already existing object). You can also register such an adapter directly in the PicoContainer:

```
picoContainer.registerComponent(new InstanceComponentAdapter("Another Apple", new Apple()));
```

This code snippet also demonstrates, that the ComponentAdapter in fact also stores the component's key.

ComponentAdapter

As already explained it is the task of a ComponentAdapter to deliver an instance of a component. This might be an existing object as delivered by the InstanceComponentAdapter or in the standard case a created object. If a PicoContainer is requested for a component it will request in fact a component adapter with a component of the appropriate key or type to do so. The ComponentAdapter is also responsible to resolve all the dependencies of the component i.e. it will ask the requesting PicoContainer in return for any component it needs itself to resolve the components its own component is dependend on. This separation allows the implementation of different instantiation strategies as realized by the following ComponentAdapter implementations from the PicoContainer core package:

- ConstructorInjectionComponentAdapter
- SetterInjectionComponentAdapter
- and others in the add on packages

Aside form the instantiation strategy a lot of the ComponentAdapter implementations are delegators that are used for orthogonal functionality:

- CachingComponentAdapter
- SynchronizedComponentAdapter
- ImplementationHidingComponentAdapter
- and a lot more

These adapters have a constructor taking another ComponentAdapter instance as argument and you might build a complex chain. Looking at the registry process in the PicoContainer, the following calls are equivalent:

- Convenient:

```
picoContainer.registerComponentImplementation(Juicer.class);
```

- At length:

```
picoContainer.registerComponent(
        new CachingComponentAdapter(
                new ConstructorInjectionComponentAdapter(Juicer.class, Juicer.class, null)));
```

ComponentAdapterFactory

The last code snippet has shown, that the PicoContainer automatically uses a cached ConstructorInjectionComponentAdapter. This is a reasonable default for service-oriented components using dependency injection. As you have already learned you can register a component with a different adapter (chain). But what if most of your service components have bean-style and need synchronization? This is a reasonable situation to change the ComponentAdapterFactory of the PicoContainer:

```
MutablePicoContainer picoContainer = new DefaultPicoContainer(
        new SynchronizedComponentAdapterFactory(
                new CachingComponentAdapterFactory(
                        new SetterInjectionComponentAdapterFactory())));
```

Following code is executed now, if a component is registered:

```
picoContainer.registerComponent(
        new SynchronizedComponentAdapter(
                new CachingComponentAdapter(
                        new SetterInjectionComponentAdapter(
                                JuicerBean.class, JuicerBean.class, null))));
```

Other Container Elements

- ComponentMonitor
- Lifecycle

Usage patterns

Multiple ComponentAdapterFactories

In complex applications are build often with a lot of components, that can be grouped in different ways. Some will act as singletons in your application and might be registered using the default ComponentAdapterFactory of the PicoContainer, others must be created for each thread, the next are exposed with JMX and another group may be some EJBs to use. In such a case it is useful to create different ComponentAdapterFactory chains, that matches the needs of every group. Use these factories to create the ComponentAdapter instances and register them directly in the PicoContainer.

Component Monitors

This page last changed on Sep 11, 2005 by paul.

PicoContainer provides a mechanism for developers to be informed of key events in the lifecytcle of components. If you pass in a ComponentMonitor DefaultPicoContainer as you instantiate it, you can be kept informed of these events:

```
new DefaultPicoContainer(myComponentMonitor);
```

ComponentMonitor implementations have six methods. From its interface:

```
void instantiating(Constructor constructor);
void instantiated(Constructor constructor, long duration);
void instantiationFailed(Constructor constructor, Exception e);
void invoking(Method method, Object instance);
void invoked(Method method, Object instance, long duration);
void invocationFailed(Method method, Object instance, Exception e);
```

As components are instantiated, or methods on them are invoked (with some limits), the methods on the CM implementation are invoked as appropriate.

Instantiation

If a component is instantiated, by one of the default ComponentAdapters, there be two calls to methods in CM. The first to 'instantiating' just before the instantiation of the component, the second to either 'instantiated' or 'instantiationFailed' as appropriate after the attempted instantiation.

Invocation

Where the ComponentAdapter in use can intercept arbitrary method invocations for a component, calls to the CM implementation will happen before component method invocation ('invoking') and after ('inkoked' or 'invocationFailed').

There are some limitations to this:

One limitation is the ComponentAdapter, as mentioned, has
to be able to intercept method invocations. ImplementationHidingComponentAdapter is one such impl, but
it is only able to intercept methods declared
on the public interfaces it is exposing via reflection.

The second limitation is that the methods invocations being monitored are only thos invoked from outside of the component from one of the other components depending on it. This may well be monitored:

```
fooComponent.doSomething();
```

Whereas, this may not:

```
this.doSomething();
```

ComponentMonitor Implementations.

There are a number of implementations of CM that are supplied by the PicoContainer team:

NullComponentMonitor - As per the NullObject pattern, does nothing, and is the default.

CommonsLoggingComponentMontor - Logs instantiations and invocations to a Log, implemented by Apache's
Commons-Logging framework (in the PicoContainer gems jar)
Log4JComponentMontor - Logs instantiations and invocations to a Log, implemented by Apache's
Log4J framework (also in the PicoContainer gems jar)
ConsoleLoggingComponentMonitor - Logs to the console.
NullComponentMonitor - does nothing. As per NullObject pattern, and the default for
DefaultPicoContainer.
WriterComponentMonitor - writes to a

Chaining

Some CM implementations are chainable. That is, you can get one impl to invoke the same method in another, subject to filtering or modification.

Implementations yet to be written

RegexFilterComponentMonitor - one that filters instantiations and invocations according to one or
more reular expression.
PasswordHidingComponentMonitor - one that for methods called 'login' obscures the second string
parameter before passing on the invocation. This means that passwords do not appear in log files
(amongst other things).

This page last changed on Sep 18, 2005 by mauro.

Lifecycle

PicoContainer handle the lifecycle aspects of components using two complementary concepts: Lifecycle Strategy and Lifecycle Manager.
The lifecycle strategy acts on a component instance and applies the lifecycle methods, while the lifecycle manager actos on container
instances and uses the container to get the component instance (resolving any dependencies if may require) and the delegates to the
injected lifecycle strategy to task of applying the lifecycle methods.

Lifecycle Strategies

The lifecycle strategy is represented by the interface LifecycleStrategy which, when implemented, handles the lifecycle aspects of components that a particular PicoContainer instance is managing. There are three methods that require implementation:

```
void start(Object component);

void stop(Object component);

void dispose(Object component);
```

PicoContainer will call start, stop, dispose as appropriate to start, stop or dispose when it may be appropriate for a component.

DefaultLifecycleStrategy

This class provides lifecycle functionality for components implementing the Startable and Disposable interfaces that ship with PicoContainer. The lifecycle methods are only invoked if the component implements Startable and Disposable.

```
public class Foo implements Startable {
  public void start() {
    // something
  }
  public void stop() {
    // something
  }
}
```

```
public class Bar implements Disposable {
  public void dispose() {
  }
}
```

ReflectionLifecycleStrategy

This class provides lifecycle functionality for components that expose the start/stop/dispose methods but do not
implement the Startable and Disposable interfaces.

```
public class Foo {
  public void start() {
    // something
  }
  public void stop() {
    // something
  }
}
```

```
public class Bar {
  public void dispose() {
  }
}
```

ReflectionLifecycleStrategy is part of the Picocontainer Gems package.

Lifecycle Managers

LifecycleManager is an interface implemented by various ComponentAdapterFactories, ComponentAdapters, and PicoContainers.
Similarly to LifecycleStrategy, it has three methods but the argument passed in is the container rather than the component instance.

```
void start(PicoContainer container);

void stop(PicoContainer container);

void dispose(PicoContainer container);
```

A ComponentAdapter may or may not implement the LifecycleManager interface. If it does implement it, it will typically have
an injected LifecycleStrategy to delegate invocation of start/stop/dispose events.

Parent and Child Containers

This page last changed on Sep 05, 2005 by paul.

3.3 Lifecycle

This page last changed on Jan 25, 2006 by mauro.

Authors: Paul Hammant, Jon Tirsen

Overview

Lifecycle is one third of Inversion of Control. Lifecycle concerns the post composition life of the component. Start, stop and dispose are the most commonly encountered lifecycle concepts.

No Lifecycle

For really simple PicoContainer compatible components, you would not bother with a lifecycle beyond instantiation and garbage collection - both of which are consequences of basic use of say DefaultPicoContainer. This is the vast majority of Pico components in the vast majority of applications.

```
public class Foo {
    public Foo() {
         // yippee, I am alive, active, going, steaming ahead, strutting my stuff...
     }
}
```

Aggregated View

It might be common to see a tree of components and containers in a advanced application. If one of the lifecycle methods is invoked on the root container, it is in turn invoked on all child containers and components where is is appropriately implemented. The starting of components is handled breadth The stopping and disposing of containers and components is handled depth first, and in reverse order to instatiation.

Simple lifecycle

PicoContainer provides two very simple interfaces for lifecycle:
Startable and Disposable.

These interfaces honours the classic lifecycle concepts for components. These are `start()`, `stop()` and `dispose()`. If any of the components implements one of the applicable interfaces, the calling of the same method on the container will find that the call is percolated through to it. The container is forgiving, as it is just fine if some or none of the components contained are Startable etc.

Example

```
public class Peach implements Startable, Disposable {
    public Peach() {
        // no wait, I'm fully composed but should wait for further instruction
     }

    public void start() {}
    public void stop() {}
    public void dispose() {}
}

public class Kiwi implements Startable {
    public void start() {}
    public void stop() {}
}
```

Adaption

Components can sometime implement start/stop, but not those mandated by an the PicoContainer lifecycle interface:

```
public interface Banana {
    public void banana();
}

public class BananaImpl implements Banana {
    public void banana() {}
    public void start() {}
    public void stop() { }
}
```

In this scenario, a simple extension can make the component honour the PicoContainer lifecycle interface:

```
public BananaExtender extends BananaImpl implements Startable {}
```

Delegation is also a neat way of adapting the component:

```
public class BananaDelegate implements Banana, Startable {
    private Banana realBanana;

    public BananaDelegate(Banana banana) {
        this.realBanana = realBanana;
    }

    public void banana() {
        realBanana.banana();
    }

    public void start() {
        realBanana.start();
    }

    public void stop() {
        realBanana.stop();
    }
}
```

Custom lifecycles

The following functionality has been moved to NanoContainer-Proxytoys

Custom lifecycle management is a common requirement for components. Probably because it is so crucial, there are many competing implementations. Each has its own passionate group of advocates.

PicoContaioner tries to avoid controversy by being partially agnostic about the lifecycle/lifecycles that it supports. As detailed previously, PicoContainer does have an implementation of a simple lifecycle and this is supported in the default container.

Instead of mandating PicoContainer's lifecycle interface, we provide a plug-in that should grant compatibility with other lifecycle concepts. Consider this custom interface:

```
public interface QuantumLeapable {
    void leap()
}
```

```
PicoContainer pico = ...

QuantumLeapable quantumLeapable = (QuantumLeapable) Multicaster.object(pico, true, new
StandardProxyFactory());

// This will call leap() on all QuantumLeapable components inside pico.
quantumLeapable.leap();
```

3.4 Components

This page last changed on Jan 25, 2006 by mauro.

Authors: Paul Hammant, Aslak Hellesoy, Jon Tirsen

Overview

Cannot resolve external resource into attachment.
A pico compatible component is a public class that, for PicoContainer at least, is characterized in the following ways:

- Does not have to implement or extend anything (unless they want to participate in the default lifecycle, see 3.3 Lifecycle).
- Declares one or more public constructors where the arguments are the component's dependencies and configuration.
- Reusable in many different deployment scenarios (root, Servlet, Applet, ClassLoaders etc).

This page last changed on Jan 25, 2006 by mauro.

Authors: Paul Hammant

Overview

Configuration for PicoContainer components requires some explanation. The basic idea is that a component should not be tied to a single configuration design.

There are two ways a developer may declare configuration needs for a component. The first is as seperate parameters in the constructor, the second is as a bespoke configuration pseudo-component, also passed in through constructor.

Constructor parameters

Consider a component a that requires configuration :

```
class Foo {
      public Foo(DependantComp dComp, String fooName, Integer barNumber) {
      }
    }
```

Clearly the string and the integer are not components. What we need is a way of passing in those parameters at runtime. Possibily interleaved with real components.

```
MutablePicoContainer pico = new DefaultPicoContainer();
    pico.registerComponent(DefaultDependantComp.class);
    pico.registerComponent(Foo.class, new Parameter[]{...});
```

We're trying to illustrate the intermingling of components and configuration. Well perhaps we are if you consider the following component :

```
class Foo {
      public Foo(Wilma wilma, String fooName, FredImpl fred, Integer barNumber) {
      }
    }

    ....

    MutablePicoContainer pico = new DefaultPicoContainer();
    pico.registerComponent(Foo.class, new Parameter[]{...});
    pico.registerComponent(Wilma.class, WilmaImpl.class);
    pico.registerComponent(FredImpl.class);
```

Pseudo-component

```
class Foo {
      public Foo(DependantComp dComp, FooConfig fooConfig) {
      }
    }

    interface FooConfig {
      String getFooName();
      int getBarNumber(); // note this is int not Integer (restriction lifted).
    }
```

Without going into the how, many implementations of the FooConfig are possible.

```
MutablePicoContainer pico = new DefaultPicoContainer();
    pico.registerComponent(DefaultDependantComp.class);
    pico.registerComponent(Foo.class);
    pico.registerComponent( DefaultFooConfig.class );
```

Clearly a person needs to write an adaptor, but any adaptor can be written. The developer who uses the component can do anything - they are not forced to fit one configuration design.

Anti-patterns

In non IoC designs, a component may hard code its configuration in one of a number of ways.

True hard coding:

```
class MyWebServer {
      ServerSocket socket;
      public MyWebServer() {
      }
      public void start() {
        // listen on port 80
        socker = new ServerSocket(80);
      }
    }
```

Bound to a specific properties file:

```
class MyWebServer {
      ServerSocket socket;
      public MyWebServer() {
      }
      public void start() {
        ResourceBundle rb = new ResourceBundle("MyWebServer.properties");
        socker = new ServerSocket(rb.getIntProperty("port.number");
      }
    }
```

There are IoC anti-patterns as the embedor can't choose their own configuration mechanism. An application comprising a number of components may have to include multiple xml and properties files to control the configuration.

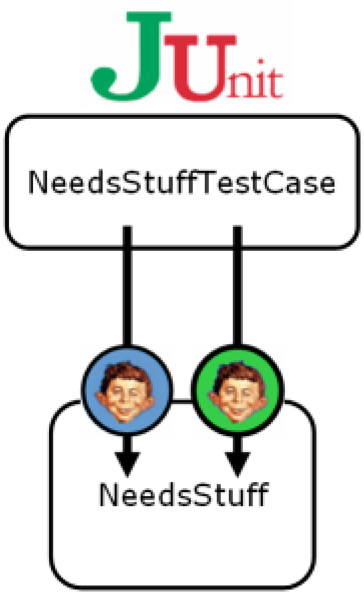
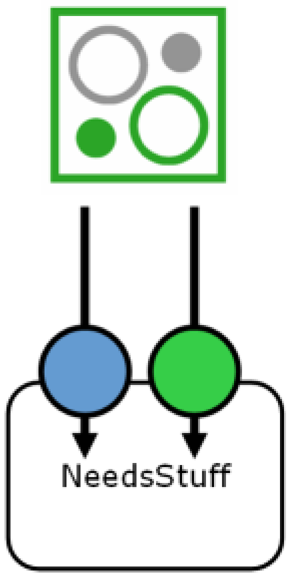
Mock Objects

This page last changed on Jan 25, 2006 by mauro.

If you have had it with codebases that drag along everything and the kitchen sink, (possibly using the Singleton) you must read this page. Classes that look up or instantiate heavyweight classes themselves are such beasts. As you might have experienced, they are a pain to test. (And ample proof that the authors didn't follow TDD). Dependency Injection with PicoContainer and Mock Objects to the rescue!

Mock Objects are special objects used during testing. Mock Objects and classes that honour Constructor Injection are a **perfect match**, since such classes can be handed mocks during testing and "the real thing" when the application is run.

This illustration shows how:

Unit-Test Time	Deployment Time
	

The class

Here is what NeedsStuff might look like:

```
public class NeedsStuff {
    // These are both interfaces.
    private final BlueStuff bs;
    private final GreenStuff gs;

    public NeedsStuff(BlueStuff bs, GreenStuff gs) {
        this.bs = bs;
        this.gs = gs;
    }

    public String doIt() {
        return bs.jump() + gs.beatIt();
    }
}
```

During test time we'll give NeedsStuff some mocks.

During prime time (when the final application is running), the NeedsStuff class will be instantiated with a SuperHeavyBlueStuff and a NuclearGreenStuff instance. These require some really heavy infrastructure such as database connections and network access. We don't want to drag along that when we test NeedsStuff! (It can wait till the integration test).

Test Time

Our test becomes like this:

```
public class NeedsStuffTestCase extends junit.framework.TestCase {
    public void testNeedsStuffDoesStuff() {
        BlueStuff bs = createBlueStuffMockThatReturnsBlahOnJump();
        GreenStuff gs = createGreanStuffMockThatReturnsHuhOnBeatIt();

        NeedsStuff ns = new NeedsStuff(bs, gs);
        assertEquals("BlahHuh", ns.doIt());

        // verify mocks.
    }
}
```

We are testing the doIt() method without having to drag along any heavy dependencies 🙂

ⓘ We won't go into further technical details about mocks, as there are many libraries to choose from. Check out JMock, MockObjects and EasyMock.

Prime Time

It is left to PicoContainer to instantiate NeedsStuff. In order for it to succeed, we must also configure the container with some real BlueStuff and GreanStuff:

```
public class AppBootstrapper {
    public void runapp() {
        MutablePicoContainer pico = new DefaultPicoContainer();
        pico.registerComponentImplementation(NeedsStuff.class);
        pico.registerComponentImplementation(SuperHeavyBlueStuff.class);
        pico.registerComponentImplementation(NuclearGreenStuff.class);
    }
}
```

ⓘ It is even possible to do this component wiring using a soft scripting language. See Nano Container

3.5 Advanced Topics

This page last changed on Jan 25, 2006 by mauro.

- Arrays, Collections and Maps
- Implementation hiding and Hot swapping

TODO - Multiple Components of the same Type

Arrays, Collections and Maps

This page last changed on Jul 25, 2005 by mauro.

Authors: Aslak Hellesoy, Jörg Schaible

Supported Collective Types

PicoContainer supports injection of collective types. These are native Arrays, Collections and Maps. Components depending on types implementing these interfaces can be provided automatically with such an instance. Since for native arrays the type can be determined at runtime, this can be done quite automatically, for the other types a special parameter must be provided. For the examples we use following classes (just ignore the fact that the classes are static):

```
public static interface Fish {
}

public static class Cod
        implements Fish {
}

public static class Shark
        implements Fish {
}
```

Arrays

PicoContainer can create a native array of components of a specific type automatically. Example code for the Bowl class in use:

```
public static class Bowl {
    private final Fish[] fishes;
    private final Cod[] cods;

    public Bowl(Fish[] fishes, Cod[] cods) {
        this.fishes = fishes;
        this.cods = cods;
    }

    public Fish[] getFishes() {
        return fishes;
    }

    public Cod[] getCods() {
        return cods;
    }
}
```

Example usage:

```
pico.registerComponentImplementation(Shark.class);
pico.registerComponentImplementation(Cod.class);
pico.registerComponentImplementation(Bowl.class);

Bowl bowl = (Bowl) pico.getComponentInstance(Bowl.class);
```

PicoContainer will instantiate the arrays and populate them with all components that matches the array type. Behind the scenes something similar to the following is happening:

```
Shark shark = new Shark();
Cod cod = new Cod();

Fish[] fishes = new Fish[]{shark, cod};
Cod[] cods = new Cod[]{cod};

Bowl bowl = new Bowl(fishes, cods);
```

Collections

PicoContainer supports automatically generated Collection types. Example code for the Bowl class in use:

```
public static class Bowl {
    private final LinkedList fishes;
    private final Collection cods;

    public Bowl(LinkedList fishes, Collection cods) {
        this.fishes = fishes;
        this.cods = cods;
    }

    public Collection getFishes() {
        return fishes;
    }

    public Collection getCods() {
        return cods;
    }
}
```

Unfortunately there is no way of detecting the type of the components beeing part of the collection as it is done for native arrays. Therefore you must use a special constructor of ComponentParameter and define the component's type:

```
pico.registerComponentImplementation(Shark.class);
pico.registerComponentImplementation(Cod.class);
pico.registerComponentImplementation(Bowl.class, Bowl.class, new Parameter[]{
        new ComponentParameter(Fish.class, false), new ComponentParameter(Cod.class, false)});
```

The boolean argument defines, that the Collection cannot be empty. Also look at the constructor of the Bowl class. You can use a specific class implementing the Collection interface or just the interface itself and PicoContainer will provide a matching Collection instance.

Maps

PicoContainer also supports automatically generated Map types. Example code for the Bowl class in this case:

```
public static class Bowl {
    private final TreeMap fishes;
    private final Map cods;
```

```
    public Bowl(TreeMap fishes, Map cods) {
        this.fishes = fishes;
        this.cods = cods;
    }

    public Map getFishes() {
        return fishes;
    }

    public Map getCods() {
        return cods;
    }
}
```

As for Collection types, PicoContainer cannot detect the type of the components, that should be part of the collection on its own. Again you must use a special constructor of [ComponentParameter]() and define the component's type:

```
pico.registerComponentImplementation("Shark", Shark.class);
pico.registerComponentImplementation("Cod", Cod.class);
pico.registerComponentImplementation(Bowl.class, Bowl.class, new Parameter[]{
    new ComponentParameter(Fish.class, false),
    new ComponentParameter(Cod.class, false)
});
```

The boolean argument defines, that the Map cannot be empty. Also look at the constructor of the Bowl class. You can use a specific class implementing the Map interface or just an interface itself and PicoContainer will provide a matching Map instance. A special feature is available due to the nature of the Map. The component adapter's key is used also as key in the injected Map.

Use Cases

While the usage of this collective types is straight forward, there are some special cases to consider. These special use cases are explained here.

Empty Collective Instances

Normally the dependency resolution for a collective type will fail, if no component of the specific component type is registerd in the PicoContainer. With the constructors of ComponentParameter you have also the possibility to accept an empty collective type as a satisfying argument. Example code for an Array:

```
Parameter parameter = CollectionComponentParameter.ARRAY_ALLOW_EMPTY;
pico.registerComponentImplementation(Bowl.class, Bowl.class, new Parameter[]{parameter,
parameter});

Bowl bowl = (Bowl) pico.getComponentInstance(Bowl.class);
```

Example code for a Collection (Map is ananlogous):

```
pico.registerComponentImplementation(Bowl.class, Bowl.class, new Parameter[]{
        new ComponentParameter(Fish.class, true), new ComponentParameter(Cod.class, true)});

Bowl bowl = (Bowl) pico.getComponentInstance(Bowl.class);
```

Note that in both examples no other component was registered. This behaviour is useful if you have a monitor with listeners, that can be registered by configuration. Even if no listener is configured, the monitor component is still instantiatable.

Precedence

PicoContainer will only generate a collective type on the fly, if no such type was registered before. So you can overwrite the dependency resolution (see example for a native Array):

```
pico.registerComponentImplementation(Shark.class);
pico.registerComponentImplementation(Cod.class);
pico.registerComponentImplementation(Bowl.class);
pico.registerComponentInstance(new Fish[]{});

Bowl bowl = (Bowl) pico.getComponentInstance(Bowl.class);
```

Demonstrated with this unit test:

```
List cods = Arrays.asList(bowl.getCods());
assertEquals(1, cods.size());

List fishes = Arrays.asList(bowl.getFishes());
assertEquals(0, fishes.size());
```

See same example code for a Collection (Map is again analogous):

```
final Set set = new HashSet();
pico.registerComponentImplementation(Shark.class);
pico.registerComponentImplementation(Cod.class);
pico.registerComponentImplementation(Bowl.class, Bowl.class, new Parameter[]{
        new ComponentParameter(Fish.class, false), new ComponentParameter(Cod.class, false)});
pico.registerComponentInstance(set);

Bowl bowl = (Bowl) pico.getComponentInstance(Bowl.class);
```

```
Collection cods = bowl.getCods();
assertEquals(0, cods.size());
assertSame(set, cods);

Collection fishes = bowl.getFishes();
assertEquals(2, fishes.size());
```

But how can you circumvent such a situation and ensure that the collective type is generated even if one of the same type was registered? Make usage of the CollectionComponentParameter. Example code for an Array:

```
pico.registerComponentImplementation(Shark.class);
pico.registerComponentImplementation(Cod.class);
Parameter parameter = new CollectionComponentParameter();
pico.registerComponentImplementation(Bowl.class, Bowl.class, new Parameter[]{parameter,
parameter});
pico.registerComponentInstance(new Fish[]{});
pico.registerComponentInstance(new Cod[]{});

Bowl bowl = (Bowl) pico.getComponentInstance(Bowl.class);
```

And here the example code for a Collection (Map is still analogous):

```
pico.registerComponentImplementation(Shark.class);
pico.registerComponentImplementation(Cod.class);
pico.registerComponentImplementation(Bowl.class, Bowl.class, new Parameter[]{
        new CollectionComponentParameter(Fish.class, false), new
CollectionComponentParameter(Cod.class, false)});
// This component will match both arguments of Bowl's constructor
pico.registerComponentInstance(new LinkedList());

Bowl bowl = (Bowl) pico.getComponentInstance(Bowl.class);
```

Scope

Any collective types will collect its components from the complete container hierarchy. See example code for Map types as a unit test:

```
MutablePicoContainer parent = new DefaultPicoContainer();
parent.registerComponentImplementation("Tom", Cod.class);
parent.registerComponentImplementation("Harry", Cod.class);
MutablePicoContainer child = new DefaultPicoContainer(parent);
child.registerComponentImplementation("Dick", Cod.class);
child.registerComponentImplementation(Bowl.class, Bowl.class, new Parameter[]{
    new ComponentParameter(Fish.class, false),
    new ComponentParameter(Cod.class, false)
});
Bowl bowl = (Bowl) child.getComponentInstance(Bowl.class);
assertEquals(3, bowl.fishes.size());
assertEquals(3, bowl.cods.size());
```

All Cods have been found and pu tinto the bowl. If two components are registered with the same key, the first found will be considered. This is even true, if it means that the component is not part of the collection. See example code again as unit test:

```
MutablePicoContainer parent = new DefaultPicoContainer();
parent.registerComponentImplementation("Tom", Cod.class);
parent.registerComponentImplementation("Dick", Cod.class);
parent.registerComponentImplementation("Harry", Cod.class);
MutablePicoContainer child = new DefaultPicoContainer(parent);
child.registerComponentImplementation("Dick", Shark.class);
child.registerComponentImplementation(Bowl.class, Bowl.class, new Parameter[]{
    new ComponentParameter(Fish.class, false),
    new ComponentParameter(Cod.class, false)
});
Bowl bowl = (Bowl) child.getComponentInstance(Bowl.class);
assertEquals(3, bowl.fishes.size());
assertEquals(2, bowl.cods.size());
```

Dick the Shark took precedence over *Dick the Cod*.

Filter based on Key Type

A generated map automatically deliver the component adapter's key as key of the map entry. As for the genric types in Java 5.0 this key may be of a specific type. Just define it using the constructors of ComponentParameter. See the example code:

```
pico.registerComponentImplementation(Shark.class);
pico.registerComponentImplementation("Nemo", Cod.class);
pico.registerComponentImplementation(Bowl.class, Bowl.class, new Parameter[]{
    new ComponentParameter(String.class, Fish.class, false),
    new ComponentParameter(Cod.class, false)
});

Bowl bowl = (Bowl) pico.getComponentInstanceOfType(Bowl.class);
```

The unit test demonstrates that only **named** fishes are in the fish bowl:

```
Cod cod = (Cod) pico.getComponentInstanceOfType(Cod.class);
Map fishMap = bowl.getFishes();
Map codMap = bowl.getCods();
assertEquals(1, fishMap.size());
assertEquals(1, codMap.size());
assertEquals(fishMap, codMap);
assertSame(cod,fishMap.get("Nemo"));
```

But this key type selection does not only work with Map types, but also with Collection types and even with a native Array! See the unit test for the Array (although the ValueType parameter will be ignored for a native Array):

```
List fishes = Arrays.asList(bowl.getFishes());
List cods = Arrays.asList(bowl.getCods());
assertEquals(1, fishes.size());
assertEquals(1, cods.size());
assertEquals(fishes, cods);
```

Individual Filter

Although filtering on component type or key type is useful, it might not be enough. Therefore you can add an additional and completely individual filter by overloading CollectionComponentParameter.evaluate(ComponentAdapter). See the example code to filter out the Cod named *Tom*:

```
MutablePicoContainer mpc = new DefaultPicoContainer();
mpc.registerComponentImplementation("Tom", Cod.class);
mpc.registerComponentImplementation("Dick", Cod.class);
mpc.registerComponentImplementation("Harry", Cod.class);
mpc.registerComponentImplementation("Sharky", Shark.class);
mpc.registerComponentImplementation(Bowl.class, Bowl.class, new Parameter[]{
    new CollectionComponentParameter(Fish.class, false),
    new CollectionComponentParameter(Cod.class, false) {
        protected boolean evaluate(ComponentAdapter adapter) {
            return !"Tom".equals(adapter.getComponentKey());
        }
    }
});
Cod tom = (Cod) mpc.getComponentInstance("Tom");
Bowl bowl = (Bowl) mpc.getComponentInstance(Bowl.class);
assertTrue(bowl.fishes.values().contains(tom));
assertFalse(bowl.cods.values().contains(tom));
```

An even more advanced filter mechanism is using the Constraint extension as provided in the picocontainer-gems.

Implementation hiding and Hot swapping

This page last changed on Oct 05, 2005 by joehni.

Authors: Aslak Hellesoy

⚠ ⚠ ⚠ ⚠ ⚠ ⚠ WARNING! CODE IS PARTLY OUT OF DATE. NEEDS WORK. ⚠ ⚠ ⚠ ⚠ ⚠ ⚠

Basics

Circular dependencies and transparent hot swapping is supported by ImplementationHidingComponentAdapter. In order to achieve this, your components must honour Interface Implementation Separation. It is then done simply by instantiating the container with a CachingComponentAdapterFactory around a ImplementationHidingComponentAdapterFactory.

Note: The ImplementationHidingComponentAdapter was removed from PicoContainer and is now located in the component NanoProxytoys.

Example

Consider two classes, Wife and Husband:

```
public class Wife implements Woman {
    public final Man man;

    public Wife(Man man) {
        this.man = man;
    }

    public Man getMan() {
        return man;
    }
}
```

```
public class Husband implements Man {
    public final Woman woman;

    public Husband(Woman woman) {
        this.woman = woman;
    }

    public int getEndurance() {
        return 10;
    }
}
```

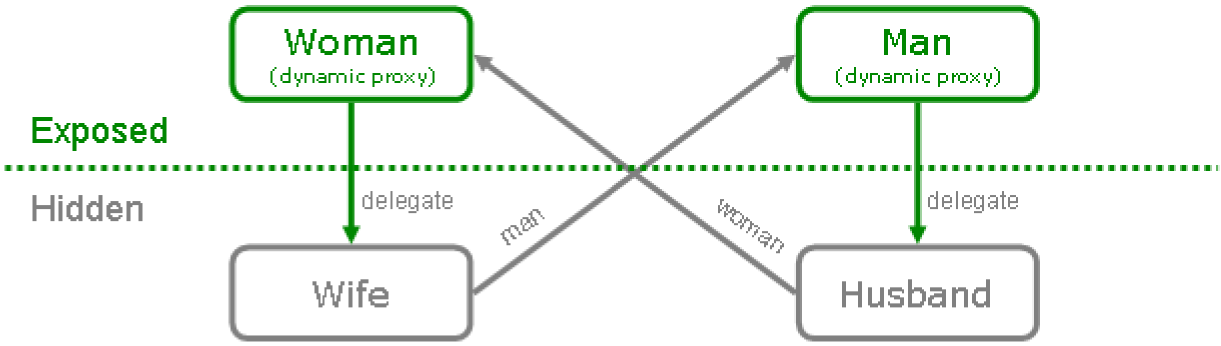
```
public interface Woman {
    Man getMan();
}
```

```
public interface Man {
    int getEndurance();
}
```

We can register them both in PicoContainer, and thanks to ImplementationHidingComponentAdapter's lazy materialisation, we can have mutual dependencies:

[Unable to download
https://svn.codehaus.org/picocontainer/java/picocontainer/trunk/container/src/test/org/picocontainer/doc/hotswappin
:
https://svn.codehaus.org/picocontainer/java/picocontainer/trunk/container/src/test/org/picocontainer/doc/hotswappin

Note that we have to cast to the interface they implement, since what the container gives us back is actually dynamic proxies that forward to the real subjects (that are hidden from you). Here is a UML object diagram that describes the relationships between the objects in the system.



When a component is requested via getComponentInstance(), only a dynamic proxy for the real subject is created. Nothing more. This means that after line 7 is executed, the only object that exists is the proxy for Woman. The hidden delegates aren't instantiated until a method is called on the proxy. (The delegates are lazily instantiated).

After line 8 is executed, all 4 objects will exist. The call to getMan() will cause the real Wife subject to be instantiated. This will in turn instantiate the Man proxy to satisfy Wife's constructor. Finally, the call to getEndurance() will instantiate the Husband subject.

In addition to supporting mutual dependencies, ImplementationHidingComponentAdapterFactory also lets you hotswap the delegates. This can be done by casting the component instance to Swappable and pass in the new subject via the hotswap() method. (All proxy objects created by ImplementationHidingComponentAdapter implement Swappable).

[Unable to download
https://svn.codehaus.org/picocontainer/java/picocontainer/trunk/container/src/test/org/picocontainer/doc/hotswappin
:
https://svn.codehaus.org/picocontainer/java/picocontainer/trunk/container/src/test/org/picocontainer/doc/hotswappin

```
public class Superman implements Man {
    public int getEndurance() {
        return 1000;
    }
}
```

This will swap out the Man proxy's current delegate with a new one. Completely transparently to the Wife object. When the wife now calls a methods on her Man, it will be delegated to a new instance. Scary?

3.6 Why use PicoContainer

This page last changed on Jan 26, 2006 by mauro.

Direct Usage Scenarios

While we have always said that embracing Constructor Injection within an Inversion of Control design is far more important than using a container, there are many reasons you'd use PicoContainer.

Factors that are common to just about all *direct* usage scenarios are:-

Mutiple implementations of a type

You have a need to support multiple implementations of some component type (as simple as a single class) with potentially very different constructors. The only assumption PicoContainer makes is that all component implementations are designed using Dependency Injection (Constructor Injection or Setter Injection). It could be that all of the implementations are in your codebase already. It could be that some are in your codebase and some are going to be written by others (refer to the Plugin pattern)

No forced choice of metadata for plugins

It would typically also be true that you did not want to expose your plugin community to an API that is not part of your product. That may include the metadata (XML, properties etc) that could be used to specify the plugin details. You're free to define your own metadata (XML or other) to define how the plugin behaves in your design. Also whether you have to go to the level of defining what dependencies are needed in that XML or not, which could be quite tedious.

Extensions & participation in component infrastructure

You want to make some simple extensions to, or optionally participate in, the component infrastructure.

Aspects, Monitoring, Instrumentation, Publishing etc are things that interest you. All transparent of course.

Indirect Usage Scenarios

You're using reusable tools that some other team has written/designed that has a plugin architecture that secretly uses PicoContainer to handle some of its open plugin needs. It's most likely not obvious to you when you use that tool's API.

Refer to Jetbrain's Intellij IDEA, the best Java IDE.

4. Other Containers

This page last changed on Jan 25, 2006 by mauro.

4.1 Container Comparison

This page last changed on Jan 25, 2006 by mauro.

Authors: Paul Hammant, Mauro Talevi

Overview

This document aims to compare PicoContainer to other IoC containers, both lightweight and not.

There are a number of published API specifications since Java was initially released that proport to be container/component designs. We discuss then here, suggesting there are goals that some have met, and some not. This may help you make better design choices for your own components.

IoC Containers

Spring Framework

The Spring Framework is a J2EE framework. As such, Dependency Injection and 3.3 Lifecycle is only one of its concerns. PicoContainer, on the other hand, is concerned only with Dependency Injection and 3.3 Lifecycle.

Apache Avalon and its containers

Apache hosts a project that has been running for years called Avalon. It has many components that fit that its design and many containers is writtern in Java. Avalon components are characterised by implementation of many optional interfaces. Avalon components are distributed with meta-information in XML in the jar file. More XML is required to assemble components together for the same of a application. Avalon Phoenix, Excalibur Component Manager (ECM), Avalon Fortress and Avalon Merlin are the pertinent containers.

The Avalon Framework requires implementing components to implement a number of interfaces. This has proven historically to be a bit of a turn-off for component writers. Luckily there are no abstract classes that have to be extended by component writers. Those interfaces are :-

Avalon Interface	PicoContainer equivalent
LogEnabled	Logging agnostic
Contextualizable	n/a
Serviceable (used to be called Composable)	Arguments in Constructor
Configurable	Arguments in Constructor
Parameterizable	n/a
Initializable	Constructor is eqivalent lifecycle place
Startable	Startable

Suspendable	n/a
Recontextualizable	n/a
Recomposable	n/a
Reconfigurable	n/a
Reparameterizable	n/a
Disposable	Disposable

Avalon is a Contextualized Lookup IoC design. Its flaw is that components written for it cannot be used without an Avalon Framework compatible container. Many say that the XML that has to accompany each component is also a flaw. Unit testing of Avalon components is very difficult because it is difficult to manage the components from JUnit.

Sun specified nearly-IoC Containers & Component designs

Sun have specified several container/component designs

Enterprise Java Beans

Clearly Entity and Session beans run inside a container. The API is well defined, and to varying degrees of success one can deploy EJB appllications to WebLogic, WebSphere, Orion and JBoss etc. For assembly and configuration, there is high use of element-normal XML . There are some mandated parent objects and interfaces for various to extend and/or implement. Resolution is done by the components themselves via JNDI more often than not.

PicoComponents are simpler in they they do not force an extensive XML markup, nor require the implementing of certaing interfaces or extening base classes. Quite importantly the relationship between factory (home), implementation (bean) and interface (remote) parts is much more real in PicoComponents. Lastly, EJB components are nearly impossible to unit-test without much effort.

Servlets

Not so obvious - Servlets are contained by a servlet container. They are generally bundled with (or replaced by) value added propositions like JSP, but it is still a container/component design. High use of XML for assembly and configuration. Servlets have no concept of parent container or the conatainer above that (sometimes EJB) and its provision of components, which is very unfortunate. Servlets have a number of interfaces to honor, none of which is too malignent. Servlets typically deal with external (or parent) components via RMI or JNDI. In more recent releases of EJB, local interfaces rather than RMI may be the mechanism for connection the parent components. WebLogic have always provided an optimizing mechanism for this interoperation

As with EJB, PicoComponents are far simpler. This is probably because they offer no web experience, apart from anything else. Servlets again are not that unit-testable.

Applets

Applets, though presently not so often used, are a good example of Container/Component separations. There is very little XML in use by Applets. Configuration is typically delivered in applet tags in HTML. Applets are granted some access to the parent container, the brower, and its DOM model for pages and other applets. There very little standardisation for Browser as a container.

As with EJB, PicoComponents are far simpler. Applets are unit-testable but with a little effort. Complex DOM interoperation is impossible under unit testing.

Mainable (Not actually a container)

```
public static void main(String[] args) {}
```

Familiar? Hopefully not 🙂 Static plays no part in a good IoC container/component design. This includes static launching of Java Webstart (JNLP) applications. If you have to keep mainable functionaility separate your components away from the main() class so they may be instntiated separately. In .NET you'll have to make sure that the application assembly is a small bootstrap to a component one.

JNDI (Java API)

A huge map of clunkily access components via a very non-IoC mechanism. It has to be strapped with much XML to prevent inappropriate access. This is not IoC because the component reaches out for external component dependancies whenever it feels like. This last fact clouds Serlvets and EJB use.

AWT, Swing, SWT (Java graphic toolkits)

Nice container/component designs. In the case of Swing, perhaps a little difficult for coders to easily assemble applications.

Eclipse (Java graphical application platform)

The Eclipse platform is very compelling. It supports the notion of a pluggable application concept. Each component statically accesses other components via a factory (which at least Paul does not like), though it is clear that some complex classloader magic is going on. The underpinning set of graphical components, SWT , are a simple and elegant design.

4.2 Defining Lightweight

This page last changed on Jan 25, 2006 by mauro.

Overview

Much conversation has been had online about heavyweight Java versus lightweight Java. Here we attempt to add our perspective.

Martin Fowler said much with this article : http://www.martinfowler.com/articles/injection.html (Inversion of Control Containers and the Dependency Injection pattern)

Marker interfaces

```
public class Foo implements OurComponent {
  // fields, ctors, methods, etc ...
}
public interface OurComponent {
}
```

We don't particularly like interfaces that are there simply to mark a class as part of particular component framework. Its not very transparent. It also is not something you can easily add to another team's codebase if you want to use their component as one of your components.

extends, implements, throws, @attributes and prefixedOrPostfixedMethods()

```
public class Foo extends BaseComponent {
  // fields, ctors, methods, etc ...
}
```

Extending a base class is pretty obviously a heavy thing to do. So is the mandatory implementing an interface for a forced design. This is mentioned above for marker interfaces, but also true for interfaces with methods that have to be mandatarily implemented.

Similarly, and a lot less common is suggesting that component's methods need to throw something or have some other clue that helps the framework:

```
public class Foo {

  // fields, ctors,
```

```
    public void foo() throws FrameworkException {
      // ...
    }

    public void xyzFrameworkInitBar(Bar bar) {
      // ...
    }

    @XyzFramework
    public Blortable blort() {
      // ...
    }

  }
```

EJB 2.0 versus EJB 3.0

EJB had a very heavyweight model. The beans, as such, did not even directly implement their interfaces..

```
public interface Foo extends EJBObject {
    void foo() throws RemoteException;
}

public class MyBean implements SessionBean {

    protected SessionContext sessionContext = null;

    public void ejbCreate(String oneOr, String moreArgs) {
    }

    public void ejbActivate() {
    }

    public void ejbPassivate() {
    }

    public void ejbRemove() {
    }

    public void setSessionContext(SessionContext sessionContext) {
        this.sessionContext = sessionContext;
    }

    public void foo() {
      // ..
    }
```

Wheras EJB 3.0 is a little more lightweight ..

```
@remote
public class MyBean {
    public void foo() {
      // ..
    }
```

From the client usage point of view

TODO

From the implementors point of view.

TODO

4.3 Future of Dependency Injection

This page last changed on Jan 25, 2006 by mauro.

Overview

In an ideal world we see PicoContainer principles being applied to a lot of different container/component aspects of Java:

Servlets

Currently the servlet spec advises that a servlet should:

1. Have a public empty contructor
2. On instantiation, not spawn threads, or mount its own socket servers.
3. Only be able to load classes distributed in its WAR file, present in the Servlet API, or J2SE (parent classloaders).

It would be quite cool, if the servlet container could determine the needs of a servlet as a component before instantiation. The servlet was allowed to get dependant component instances via one of its constructors. The need for servlet developers to break (2) above would be diminished because there is some place hier up the container tree for fatter components, that can provide services to request oriented servlets. The class visibity advice (for servlet container writers), would be relaxed too, allowing classes not supplied in the WAR file to be compiled against and delivered by the containing environment.

```
public class PicoServletExample extends HttpServlet {
    private StockQuoteService stockQuoteService;
    public PicoServletExample(StockQuoteService sqs) {
        stockQuoteService = sqs
    }
    public void doGet(HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><body>");
        out.println("Microsoft's Current Price =" + stockQuoteService.getQuote("MSFT"));
        out.println("</body></html>");
    }
}
```

This may seem far fetched, but it is entirely possible given the number of Open Source Servlet containers.

Applets

As for the servlet example, it would be nice for Applet to require that their container resolves component dependencies. This rather than have to do a fairly open ended JNDI or RMI lookup of a remote service.

The implemetation is a component that may have several manifestations. A RMI Remote real example is where you might be headed as you write the application In the interim, the team may code and deliver various quick win solutions - a) Hard coded fake StockQuoteService, b) some real implementation that steals data from http://quotes.nasdaq.com for display in the applet, c) some centralized/proxied version of (b), d) a MockStockQuteService for Unit testing (not withstanding the fact that AppletUnit has not been written yet).

```
public class PicoAppletExample extends Applet {
    private StockQuoteService stockQuoteService;
    public PicoAppletExample(StockQuteService sqs) {
         stockQuoteServic = sqs;
    }
    public void init() {
        // lay out trading GUI etc.
    }
    public void start() {
        //whatever
    }

    public void stop() {
        // whatever
    }
    public String getAppletInfo() {
        return "Hello";
    }
}
```

Sadly, this remains far fetched, due to a couple of reasons:

- The dearth of Open Source Applet containers
- the marginal status of Applets since Microsoft froze support for Java in Internet Explorer at the JDK 1.1 stage.

Enterprise Java Beans

The same story as above:

```
public interface StockQuoteService {
    void BigDecimal getQuote(String stockTicker);
}
public class DefaultStockQuoteService implements StockQuoteService {
    QuoteRecorder quoteRecorder;
    public DefaultStockQuoteService(QuoteRecorderquoteRecorder) {
        this.quoteRecorder =quoteRecorder;
    }
    public void BigDecimal getQuote(String stockTicker) {
      quoteRecorder.recordQuote(stockTicker);
      return new BigDecimal()// TODO - make real
    }
}
```

Desktop Operating System

Getting pedictable now:

```
public class QuotePanel extends JPanel implements {
    StockQuoteService stockQuoteService;
    public QuotePanel(StockQuoteService sqs) {
        this.stockQuoteService = sqs;
        this.add(new JTextField(""), BorderLayout.CENTER);
        //etc
    }
}
```

Yo you Eclipse guys, why'd ya have to adopt the OSGi stuff rather than CDI ?

5. Getting Started

This page last changed on Jan 26, 2006 by mauro.

5.1 Tutorials

This page last changed on Jan 26, 2006 by mauro.

In the best time-honoured tradition of HelloWorld, novice users should try out our simple tutorials to get a feel for PicoContainer.

First try out the One Minute Description;
Then move up to the Two Minute Tutorial;
Finally try your hand at the Five Minute Introduction.

Done? Welcome to the Pico-world!

Five minute introduction

This page last changed on Jan 25, 2006 by mauro.

Authors: Aslak Hellesoy, Jon Tirsen

Basics

This is a quick introduction to PicoContainer's most important features. Read through it to get an idea of what PicoContainer is and isn't. If you just want to get some code up and running, try out the Two minute tutorial.

PicoContainer's most important feature is its ability to instantiate arbitrary objects. This is done through its API, which is similar to a hash table. You can put java.lang.Class objects in and get object instances back.

Example:

```
MutablePicoContainer pico = new DefaultPicoContainer();
pico.registerComponentImplementation(ArrayList.class);
List list = (List) pico.getComponentInstance(ArrayList.class);
```

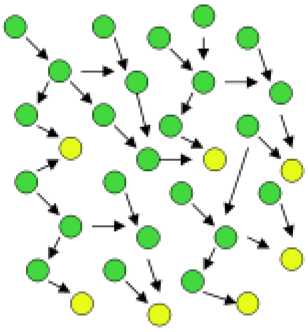
(i)MutablePicoContainer API

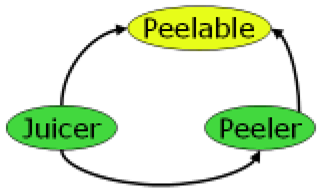
This code does the same as this:

```
List list = new ArrayList();
```

With a trivial example such as this there is no point in using PicoContainer. This was just to illustrate the basic API. PicoContainer becomes useful with larger number of classes and interfaces having complex dependencies between each other:

Complex Dependencies	Juicer Example

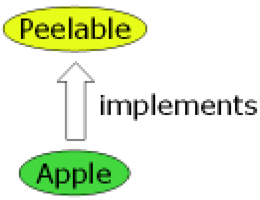




(Green means class, Yellow means interface).
PicoContainer identifies dependencies by looking at the constructors of registered classes (Constructor Injection). PicoContainer can also be though of as a generic factory that can be configured dynamically. PicoContainer is able to instantiate a complex graph of several interdependent objects.

Write some simple classes and interfaces with dependencies

The "Juicer Example" diagram above could translate to the following code (we added a concrete Peelable):




```
public interface Peelable {
    void peel();
}
```

```
public class Apple implements Peelable {
    public void peel() {
    }
}
```

```
public class Peeler implements Startable {
    private final Peelable peelable;

    public Peeler(Peelable peelable) {
        this.peelable = peelable;
    }

    public void start() {
```

```
            peelable.peel();
        }

        public void stop() {

        }
    }
```

```
    public class Juicer {
        private final Peelable peelable;
        private final Peeler peeler;

        public Juicer(Peelable peelable, Peeler peeler) {
            this.peelable = peelable;
            this.peeler = peeler;
        }
    }
```

(Note that this code suffers from the antipatterns Propagating Dependency and Concrete Class Dependency, but let's not worry about that for now)

Assemble components

You tell PicoContainer what classes to manage by registering them like this (the order of registration has no significance):

```
MutablePicoContainer pico = new DefaultPicoContainer();
pico.registerComponentImplementation(Apple.class);
pico.registerComponentImplementation(Juicer.class);
pico.registerComponentImplementation(Peeler.class);
```

(i)MutablePicoContainer API

Instantiate components

You can tell PicoContainer to give you an instance of a class like this (provided it has been registered previously):

```
Juicer juicer = (Juicer) pico.getComponentInstance(Juicer.class);
```

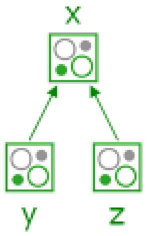
This will cause PicoContainer to do something similar to this behind the scenes (except that PicoContainer uses reflection):

```
Peelable peelable = new Apple();
Peeler peeler = new Peeler(peelable);
Juicer juicer = new Juicer(peelable, peeler);
return juicer;
```

Note how PicoContainer figures out that Apple is a Peelable, so that it can be passed to Peeler and Juicer's constructors.

Container hierarchies

PicoContainer provides a powerful alternative to the [Singleton](). With container hierarchies you can create singleton-like objects where you have fine grained control over the visibility scope of the instance. (The singleton pattern is static and global - it won't allow more than one instance, and it is visible from anywhere. Not nice).



A container (and its registered components) can get access to components registered in a parent container, but not vice-versa. Consider this example, using the classes from above:

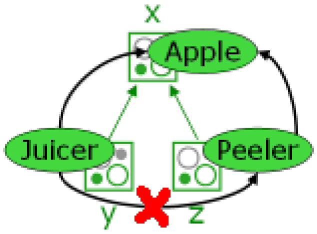
THIS WON'T WORK! It is for illustration purposes only!

```
// Create x hierarchy of containers
MutablePicoContainer x = new DefaultPicoContainer();
MutablePicoContainer y = new DefaultPicoContainer( x );
MutablePicoContainer z = new DefaultPicoContainer( x );

// Assemble components
x.registerComponentImplementation(Apple.class);
y.registerComponentImplementation(Juicer.class);
z.registerComponentImplementation(Peeler.class);

// Instantiate components
Peeler peeler = (Peeler) z.getComponentInstance(Peeler.class);
// WON'T WORK! peeler will be null
peeler = (Peeler) x.getComponentInstance(Peeler.class);
// WON'T WORK! This will throw an exception
Juicer juicer = (Juicer) y.getComponentInstance(Juicer.class);
```

This can be visualised as follows:



Let's analyse what will happen here:

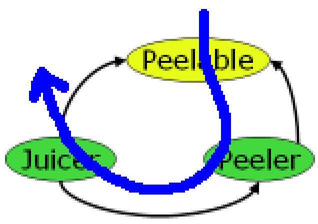
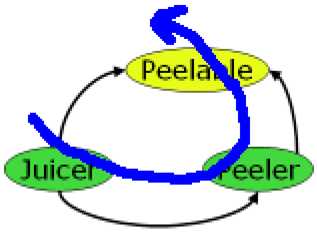
- Line 12 will work fine. c will be able to resolve the dependencies for Peeler (which is Fruit) from the parent container.
- Line 14 will return null, as a can't see Peeler.
- Line 16 will throw an exception, since Juicer's dependency to Peeler can't be satisfied (c can't be seen by b).

Since this obviously won't work, keep in mind that this was just an exercise to illustrate how container hierarchies work.
For a more concrete example of the usage of container hierarchies, see NanoContainer NanoWar.

Lifecycle

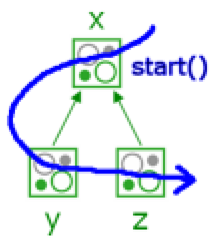
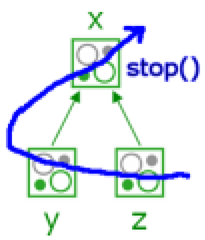
PicoContainer has support for 3.3 Lifecycle. If your classes implement Startable, you can control the lifecycle of all your objects with a simple method call on the container. The container will figure out the correct order of invocation of start()/stop() all the objects managed by the container.

Calling start() on the container will call start() on all container managed objects in the order of their instantiation. This means starting with the ones that have no dependencies, and ending with the ones that have dependencies on others:

PicoContainer.start()	PicoContainer.stop()
	

Lifecycle also works for hierarchies of containers. Calling start() on a container with child containers will start all the containers in a breadth-first order, starting with itself. Likewise, calling stop() will call stop() on all containers in the hierarchy in a depth-first order. The pictures below show what happens when start() and stop() are called on a container with children.

PicoContainer.start()	PicoContainer.stop()

⚠ In order for hierarchy-aware lifecycle to work, child containers must be registered as components in their parent container. Just creating a container with another one as a parent will **not** cause the parent container to know about the child container.

Direct approach

```
MutablePicoContainer parent = new DefaultPicoContainer();
MutablePicoContainer child = new DefaultPicoContainer(parent);
// We must let the parent container know about the child container.
parent.registerComponentInstance(child);
// This will start the parent, which will start the child.
parent.start();
```

Indirect approach

```
MutablePicoContainer parent = new DefaultPicoContainer();
parent.registerComponentImplementation("child", DefaultPicoContainer);
// This will instantiate the child container passing the parent (itself) as parent container.
// No need to register the child in the parent here.
MutablePicoContainer child = (MutablePicoContainer ) parent.getComponentInstance("child");
// This will start the parent, which will start the child.
parent.start();
```

⚠ Calling lifecycle methods on a container that has a parent container will **not** propagate the lifecycle to the parent container.

Next back to contents

One minute description

This page last changed on Jan 25, 2006 by mauro.

Authors: Paul Hammant, Aslak Hellesoy, Jon Tirsen

What is PicoContainer?

- PicoContainer is a lightweight container.
- It is **not** not a replacement for a J2EE container, as it doesn't offer any infrastructure services out of the box.
- It can help you write better code.

What does PicoContainer do?

- Dependency Injection. A way of instantiating components and lacing them together with other dependent components.
- PicoContainer is non-intrusive. Components don't have to implement any funny APIs. They can be POJOs.
- Lifecycle support is built-in. Components' lifecycle can be managed easily by PicoContainer (the default lifecycle is simple, but can be extended or totally customized).
- Very extensible design enabling virtually any form of extensions to the core.
- It is embeddable inside other applications. A 50k jar that has no external dependencies except JDK 1.3.

How do I use PicoContainer?

- Components are implemented as ordinary Java classes and do typically not have to rely on any PicoContainer APIs.
- The components are assembled in the container using a simple Java API that is similar to an intelligent hash map utilizing the type of its values.
- If lifecycle callbacks are required the simple lifecycle interfaces can be implemented. If you prefer to use your own lifecycle interfaces you can do that.
- Use the monitor support of PicoContainer to react on internal events e.g. by logging.

Why should I use PicoContainer?

- To modularize how dependencies between parts of your application are laced up. It is common having this scattered all over.
- To improve the testability of your code.
- To improve how components are configured in application.

Next: Two minute tutorial

Two minute tutorial

This page last changed on Jan 25, 2006 by mauro.

Authors: Jon Tirsen

This very short tutorial should get you up to speed with PicoContainer in 2 minutes. It does not go into why you should do it, read the Five minute introduction for that.

Download and install

Downloads the jar file and include it in your classpath.

Write two simple components

```
public class Boy {
    public void kiss(Object kisser) {
        System.out.println("I was kissed by " + kisser);
    }
}
```

```
public class Girl {
    Boy boy;

    public Girl(Boy boy) {
        this.boy = boy;
    }

    public void kissSomeone() {
        boy.kiss(this);
    }
}
```

Assemble components

```
MutablePicoContainer pico = new DefaultPicoContainer();
pico.registerComponentImplementation(Boy.class);
pico.registerComponentImplementation(Girl.class);
```

MutablePicoContainer API

Instantiate and use component

```
Girl girl = (Girl) pico.getComponentInstance(Girl.class);
girl.kissSomeone();
```

getComponentInstance will look at the Girl class and determine that it needs to create a Boy instance and pass that into the constructor to create a Girl. The Boy is created and then the Girl.

[PicoContainer API]

The Girl does not reach out to find herself a Boy but instead is provided one by the container. This is called the *Hollywood Principle* or "Don't call us we'll call you".

Introduce an interface for the dependency

Change the Boy class to implement a Kissable interface and change the Girl class to depend on Kissable instead.

```
public interface Kissable {
    void kiss(Object kisser);
}
```

```
public class Boy implements Kissable {
    public void kiss(Object kisser) {
        System.out.println("I was kissed by " + kisser);
    }
}
```

```
public class Girl {
    Kissable kissable;

    public Girl(Kissable kissable) {
        this.kissable = kissable;
    }

    public void kissSomeone() {
        kissable.kiss(this);
    }
}
```

Assemble and use components just as before:

```
MutablePicoContainer pico = new DefaultPicoContainer();
pico.registerComponentImplementation(Boy.class);
```

```
pico.registerComponentImplementation(Girl.class);
```

Or the preferred way:

```
MutablePicoContainer pico = new DefaultPicoContainer();
pico.registerComponentImplementation(Kissable.class, Boy.class);
pico.registerComponentImplementation(Girl.class);
```

Now run:

```
Girl girl = (Girl) pico.getComponentInstance(Girl.class);
girl.kissSomeone();
```

The Girl will be given a Boy, because PicoContainer understands that it is a Kissable

The Girl and the Boy no longer depend on each other, this is called the Dependency Inversion Principle since both components depend on the interface and no longer directly on each other.

Use simple lifecycle

Change the Girl class to implement the simple default lifecycle and do it's kissing when the container is started.

```
public class Girl implements Startable {
    Kissable kissable;

    public Girl(Kissable kissable) {
        this.kissable = kissable;
    }

    public void start() {
        kissable.kiss(this);
    }

    public void stop() {
    }
}
```

Assemble container as before but instead of calling the Girl directly just start the container like this:

```
pico.start();
```

This will instantiate *all* components that implement Startable and call the start method on each of them. To stop and dispose the container do as follows:

```
pico.stop();
pico.dispose();
```

(i)[Startable API]

(i)[Disposable API]

More Quick Facts

- PicoContainer can do Setter Depenedency Injection (SDI) via alternate ComponentAdapterFactories
- PicoContainer cannot do registration by class name. Make some (tiny) code to do that or use NanoContainer in conjunction with PicoContainer
- PicoContainer can has a pluggable design for Lifecycle - very flexible.
- PicoContainer really likes to see components registered by type (interface) rather directly as implementations.

Next: Five minute introduction

5.2 Terminology

This page last changed on Jan 26, 2006 by mauro.

Also known as Picosaurus

Overview

These can be arrived at from the definitions.

- Everything is an object
- Everything which is not a Contextualized Lookup component nor a domain-specific object is a POJO
- A Setter Dependency Injection (SDI) component is a JavaBean, and a JavaBean is an SDI component
- A Contructor Dependency Injection (CDI) Component is not a JavaBean, but is a POJO
- A JavaBean can be any of the three types of IoC and domain-specific all at the same time
- Spring, PicoContainer, XWork and other frameworks all support both Components and Data Objects to some extend. Avalon tries real hard to not support Data Objects, but that is nearly impossible.

Terms

Object or Instance

An "object" as per the Java Language Specification.

Class

A "class" as per the Java Language Specification.

POJO (Plain Old Java Object)

An object that does not use nor support any other kind of component/bean specification. It does not implement any lifecycle or marker interfaces, it does not provide any kind of metadata. Will usually follow the minimum requirements for classifying it as a JavaBean though.

Javabean / Bean

An object that is a "valid java bean" according to the JavaBean Specification. That is, an object that has a public default constructor, and that supports setting and/or retrieving all or a part of its state using so-called setter and getter methods, which are named getXXX(), setXXX() or isXXX(), where XXX identifies part of its state. The JavaBean specification defines a metadata framework, an event framework, and some other things which are less relevant in this context. A JavaBean may represent both data and functionality.

Data Object / Entity Bean / Data Bean / PODO

An object that only represents data, but contains no 'logic' or any kind of other functionality (except for perhaps some utility methods that transform the data into some other form, like getXXXasString(). Will often override equals() and hashCode(); will often be serializable. Does not create threads, does not keep references to non-data objects.

Component / Service

An object that only represents functionality. It is a "unit of logic", that can "do work". In a well-designed application, the work that a component can do is specified in a work interface. Will usually not override equals() or hashCode().

Passive Component

A component that does not use threads (not even indirectly).

Active Component

A component that uses threads (possibly indirectly through the use of a Thread Pool or Executor component).

Contextualized Dependancy Lookup Component

A component that has a public default constructor and that usually has its state set up and/or modified through the use of some kind of framework-specific mechanism.

Setter Injection Component

A component that has a public default constructor and that has its state set up through the use of setter methods.

Constructor Injection Component

A component that has one or more public constructors and that has its state set up through the arguments provided in one of these constructors.

Constructor/Setter Dependency Injection Component

A component that has a public constructor and that has its state set up through the arguments provided in that constructor and through the use of setter methods.

IoC Component

A component that does not implement any kind of logic to set up its own state, but completely depends on other parties to provide that state to it.

Non-IoC Component

A component that implements some kind of logic to set up its own state. Examples: a component that reads an XML file; a component that looks up dependencies in JNDI; a component that queries system properties.

Domain-Specific Object

An object that implements a specification that is less generic than Java itself in its applicability. Example: EJB, Servlets.

Spring-supported Object

An object that can be 'natively' used in the Spring Framework. Any JavaBean (with some extensions to that specification like "initialization completion" support), any object with a public default constructor that does not require any method calls to set up its state.

PicoContainer-supported Object

An object that can be 'natively' used in PicoContainer. Any Type-3 component.

XWork-supported Object

An object that can be 'natively' used in the XWork Framework. Any object with a public default constructor that has its state set up through XXXAware interfaces.

Fail Fast

An object that fails fast is an object that indicates as early as possible if it is in an inconsistent or unusable state (by throwing a fatal exception). The fastest possible way to FailFast is to do so during instantiation. This can be handled in an elegant way if the class is a GoodCitizen.

PicoComponent

A component designed to work in a PicoContainer.

PicoCompatible

A component compatible with PicoContainer. Is likely to be a PicoComponent.

Pico Powered

A product is said to be "PicoPowered" if it embeds or uses PicoContainer internally.

5.3 FAQ

This page last changed on Jan 26, 2006 by mauro.

- Can my component use multiple constructors
- Do I have to use PicoContainer to manage or use components designed for it
- How does PicoContainer compare to EJB containers
- How does PicoContainer decide what constructor to use
- How to use primitive types in constructors
- Is there an Uber or BorgContainer in the pipe
- When should I use PicoContainer
- Why Another IoC Framework
- Why Constructor Injection

Can my component use multiple constructors

This page last changed on Jul 24, 2004 by joehni.

Authors: Paul Hammant, Jörg Schaible

Yes.

You should perhaps code multiple constructors for a component:

```
class MyComp {

  private ThreadPool theThreadPool;

  public MyComp(ThreadPool threadpool) {
    theThreadPool = threadpool;
  }

  public MyComp() {
    theThreadPool = new DefaultThreadPool();
  }

  // other methods.

}
```

See additional comments at How does PicoContainer decide what constructor to use.

Do I have to use PicoContainer to manage or use components designed for it

This page last changed on Feb 24, 2004 by rinkrank.

No. Pico components are simple POJOs and are instantiable and configurable without using PicoContainer. This was one of the design goals.

How does PicoContainer compare to EJB containers

This page last changed on Mar 15, 2004 by paul.

Oh blimey, it is rather different. EJB has loads of things you must extend, implement, provide and throw. It is also not an IoC design. It is close in that components are managed by a container, but the cumbersome and static nature of the JNDI lookups ensure that it is not actually IoC. PicoContainer is not a superset of EJB though, as it provides no remoting capability (but NanoContainer will). At least the PicoContainer compatible components hosted in this project do not.

How does PicoContainer decide what constructor to use

This page last changed on Feb 24, 2004 by rinkrank.

PicoContainer will instantiate a given component using the "greediest" satisfiable constructor. By greedy, we mean the constructor that takes the most parameters. By satisfiable, we mean constructors where all arguments can be satisfied by other registered components.

If you register a component with no satisfiable constructors, or two or more ambiguous "largest" constructors, a RuntimeException will be thrown when you ask for the component instance.

We recommend, for the sake of predictablility, that PicoContainer compatible components use only one constructor (see Good Citizen), although this is by no means a requirement.

How to use primitive types in constructors

This page last changed on Jul 24, 2004 by joehni.

Authors: Jörg Schaible

PicoContainer will look for the greediest constructor of your component. But if your component's constructor depends on primitive types you may set the values explicitly.

```
public interface ThreadPool {
        void setSize(int);
}

public class MyComp {
        private ThreadPool threadPool;
        public MyComp(ThreadPool pool, int size) {
                threadPool = pool;
                threadPool.setSize(size);
        }
}
```

In this case you can set the parameters at registration time:

```
DefaultPicoContainer pico = new DefaultPicoContainer();
pico.registerComponentImplementation(ThreadPool.class, DefaultThreadPool.class);
pico.registerComponentImplementation(MyComp.class, MyComp.class, new Parameters[] {
        new ComponentParameter(),
        new ConstantParameter(new Integer(5));
})
MyComp myComp = (MyComp)pico.getInstance(MyComp.class);
```

Use ConstantParameter to set constant values and the ComponentParameter to let Pico resolve the dependency.

Is there an Uber or BorgContainer in the pipe

This page last changed on Mar 06, 2004 by rinkrank.

Q: Is a single Uber|Super|BorgCollectivePicoContainer in development? Something that can do all things for all people in all deployment situations?

A: No. We are delivering small interoperable containers and components of containers. There will never be a quest for the BorgContainer. We encourage diversity and compatibility with the essential idea here - components designed along the PicoContainer principles (dependencies and config in constructors).

When should I use PicoContainer

This page last changed on Feb 24, 2004 by rinkrank.

We recommend using PicoContainer when your application consists of many different classes (components) that are related to each other. Instantiating and lacing/connecting a lot of objects can be difficult task.

Why Another IoC Framework

This page last changed on Jan 17, 2006 by gjoseph.

After a long period of watching Inversion of Control frameworks, and for some of us avoiding them because of their cumbersome nature, we got together to write what we believe is the simplest possible IoC design. One, perhaps, that considers the so-designed component as more important that the container.

When we set out to do this in 2003, there was nothing equivalent to PicoContainer (xContainer orginally). XWork, that underpinned WebWork2 was a general/embeddable Setter Dependency Injection framework, but not well known. The Spring Framework started in the same timescale, and we were talking to its authors before Martin Fowlers Dependency Injection article. Later we discovered that HiveMind (now at Apache) also was in the same space.

Why Constructor Injection

This page last changed on Apr 07, 2004 by rinkrank.

Author: Paul Hammant

Constructor Injection is hard to swallow for people who have been living with Interface Lookup or Setter Injection for long. We think it's like TDD. Once you get the grasp of it, you don't go back. Here are some benefits with this type of IoC:

- Constructor Injection makes a stronger dependency contract
- Its more succinct in terms of lines of code
- Its more succinct in terms dependency-statement-mechanism i.e. no XML, attributes, enabler interfaces etc
- A component is characterized by InterfaceImplSeparation with the interface being the service offered to other comps, and the impl declaring whatever goddarned need it likes and that need being wholly up to the implementor of the component and nothing to do with the service contract.
- No indeterminate state. Not all the post instantiation setters may be called. Consider the non-container case for comp usage, if v1.2 of that comp introduces a new dependancy, with Constructor Injection the compiler will tell me.