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

PicoContainer 1.2 documentation

For developers extending PicoContainer
For developers Using PicoContainer

Top

This page last changed on May 12, 2004 by tirsen.

A list of all pages.

This page last changed on Nov 10, 2005 by paul.

Overview

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

Why use PicoContainer you would use PicoContainer in an application, or reusable library

Want more? See One minute description, Two minute tutorial, and Five minute introduction.

Despite it being very compact in size (the core is 100K and it has no mandatory dependencies outside the JDK),
PicoContainer supports different dependency injection types (both CDI and SDI) and offers totally customisable lifecycles.

NanoContainer builds on top of PicoContainer the support for several scripting metalanguages (XML, Groovy,
Bsh, Javascript and Jython), AOP, Web frameworks (Struts and WebWork), SOAP, JMX, and much more.

PicoContainer has originally been implemented in Java but is also available for other platforms and
languages. These are detailed in Ports.

News

PicoContainer 1.2-RC-1 has been released. Downloads are available from Downloads.

Contructor Dependency Injection ... Book Announcement

This page last changed on Oct 06, 2004 by paul.

Quite a few of the committers are have banded together to write a book on **Constructor Dependency Injection** and its title will be just that. It's subtitle is is likely to be *With PicoContainer*.

The book is going to be 50:50 science to practical. That is half a commentary on the pattenns, anti-patterns and history of CDI, IoC and COP, and half a how-to for PicoContainer and NanoContainer.

The book is going to be written by as many people as we can get to help. We intend to share credit thoughtout. The committers (or a list of individuals for legal reasons) are likely to retain copyright. While the book may developed from CVS at Codehaus, it is not in itself Open Source. It will one day drift into print (publisher to be announced) at the direction of the committers (or that list of people again). No other group or individucal is alowed to breach our copyright by printing the book.

From time to time significant contributors may be invited to be join the list of people directing the destiny of the book, and share in its profit, but that is at the discretion of the committers.

Although Paul and Aslak are kicking off the book as an effort, they fully expect to be marginalised by the content produced by others. The book is likely to rank contributors in order of contribution.

We're using http://txt2tags.sourceforge.net/ for the source of the book. Why?

1. It works with CVS.
2. It is diffable/mergable,
3. It is editable by lowest common denominator tools like Textpad without hurting your eyes with exposed angle brackets,
4. It renders to HTML,
5. Has an escape route to publisher friendly forms like Abobe Pagemaker. Tis open to debate tho.

Our CVS for the book is

http://cvs.picocontainer.codehaus.org/viewrep/picocontainer/book (see the copyright.t2t file for the full rules).

The rendered product (always a work in progress) will be hanging here - http://www.picocontainer.org/cdi-book.html

Defining Lightweight

This page last changed on Nov 10, 2005 by paul.

place holder for blurb on lightweight versus heavy weight

marker interfaces
ejb 2.0 being heavy
extends/implements/throws
martin's DI article

client usage and implementors point of view

Developer documentation

This page last changed on May 16, 2004 by rinkrank.

These documents are related to development activities. There is not much explanative text on this page as you are expected not have a short attention span if you are a developer 🙂

- How To Contribute
- Relative Volunteering
- Release Process

How To Contribute

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

PicoContainer is nothing without contributions from the user community. There are many ways to contribute. We are constantly working on the software and documentation, so it's a good idea to contact the team on the user or development mailing lists to avoid duplicating effort.

Documentation

One of the traditional weak points of open source software is the documentation. Any help with this aspect of the project will be welcomed with open arms, or at the very least with open email clients!

Examples

It would be fantastic if someone wanted to contribute with some examples. Please use our JIRA for this.

Feature Requests

If you want to request a new feature you can either make a request through the issue tracker or by sending a message to the development mailing list. The benefit of any new features will be discussed on the mailing list, so its a good idea to sign up so that you participate in the discussion.

Bug Reports

You can report bugs through the issue tracker interface or the development mailing list. Additional pico points are awarded for bug reports that include a failing unit test.

Enhancements

If you think you have found a design flaw in PicoContainer that you'd like to improve, please present your ideas on the developer mailing list and explain. Wait for someone to reply and see the outcome before submitting a patch. This will save a lot of time for both you and the developers.

However, if the improvement is really obvious and requires no discussion, just stick it in JIRA right away.

Use UnifiedPatch for this.

Improvements and Bugs

If you want lots of pico points, submit a fix in JIRA along with a description and unit test(s) (For bugs, it is OK to submit a failing test that highlights the bug).

Fixes are best sent as unified patch that we can apply to the codebase. Remember to tell us which version the patch should be applied against, or we'll get very confused. To be accepted into the codebase, patches must be released under the same license as PicoContainer itself. Also, **please** respect the coding standards and formatting. Don't reformat the existing code, as this makes it impossible for us to determine what you have changed.

If you have modified some files to fix something yourself, don't attach the modified files (as this prevents us from seeing what your changes consist of). Instead, make a unified patch of your locally modified sources (standing in the root folder of your local CVS copy):

```
cvs -q diff -b -u -N > patch-1.diff
```

Then attach patch-1.diff to an existing JIRA issue, or make a new Bug/Improvement issue if none exists. You should only attach entire files if they are new files (not yet in CVS).

If you want to submit a new and improved patch at a later time, please name it patch-2.diff etc.

The developers will then be able to apply the patch by issuing the following command from the root folder:

```
patch -p 0 < patch-1.diff
```

New Code

If you have a new feature request, then we'll listen extra hard if you show us how it works. A new feature might be best implemented as a patch to an existing class or as a new class. The PicoContainer API contains many extension points that allow new functionality to be integrated into the framework. We are rather anal about testing, so if you send us some code without any tests we will probably ask you to write the tests tests as well before we add it to the codebase.

Write Your Own PicoContainer Extension

If you have a reusable extension to PicoContainer (perhaps a fancy ComponentAdapter), we will be happy to include it directly into the PicoContainer or into a NanoContainer subproject project.

Become a Committer

We follow The Codehaus Manifesto when it comes to expanding the core team. We'll invite you, rather than you apply 🙂

Relative Volunteering

This page last changed on Apr 22, 2004 by paul.

Authors: Paul Hammant

Overview

OK, so you think you might want to volunteer for something, but not PicoContainer itself ?

Context: The container landscape for Pico and Nano are fairly crowded with developers. We have some big hitters steering our development too. The .NET port is already underway. It should be shadowing the development for the Java version. Clearly, there are features of the Java version that do not have peer equivalence in the .NET version. There are also features that are just not needed.

So how about volunteering for something that is related to PicoContainer?

Components.

What we really need are components. Lots of them. In most case it will be adaptations of existing beans. We're not interested in in having components in the PicoContainer or NanoContainer projects. We are interested in the having synergies with related projects. If someone wants to make a component to the Pico design then we'll support it.

One we consider really important is some subtle componentization of the excellent Jetty server. As well as lacing together the components of Jetty via Pico, it would be nice to have servlets that were Pico components (breaking the Servlet spec slightly). Thus an extended Jetty could instantiate servlets that had dependencies specified in their constructors.

Watch this space for a component (ComponentHaus) registry site being set-up

Release Process

This page last changed on Oct 02, 2005 by mauro.

The following process should be followed when a release is made. Please follow each step carefully.

- Check that the PicoContainer Roadmap or NanoContainer Roadmap have resolved all scheduled issues. If not, resolve them, or schedule remaining issues to a future version.
- Send email to the development mailing list telling peopl to stay away from the SCM.
- Set the release version number in PicoContainer's project.xml. For example, 1.0-SNAPSHOT would become 1.0-beta-1.
- Update any other projects you control that depend on PicoContainer.
- Make sure **none** of the external jar dependencies are SNAPSHOT dependencies, but releases or timestamped versions (so the release can be built in the future).
- Update the Downloads page and add entries for the version you're about to release.
- Update the PICO-JavaDoc Confluence shortcut http://docs.codehaus.org/admin/browseshortcuts.action. You may have to remove the old one and add it again with an updated value.
- Make a new java/picocontainer/site/picosite.zip (using the Teleport tool - Paul has a copy). TODO should be replaced with non-proprietory tool
- Commit your changes to the SCM.
- Wait for Continuous Integration to complete a successful build.
    - Alternatively - do maven pico:deploy -Dmaven.remote.group=users
- Go to the deployed site's download page and verify that the downloads work:
    - Maven builds for the source distros should work
    - The binary distro's docs should be browsable and links to javadocs should be local.
- Tag the relevant SVN trunk via svn cp <trunk URL> <tag URL>
- Mark the version as released in JIRA
- Make sure JIRA has versions matching the version now declared in project.xml.
- Update the Codehaus Blog (if you have access to it) and announce the release.
- Send email to the development mailing list announcing the release, with a link to the blog on http://blogs.codehaus.org/
- Reset the development version number to a snapshot of the next major.minor release in project.xml (/picocontainer/trunk) and commit the change to the SCM. For example, 1.0-beta-1 would become 1.0-SNAPSHOT.
- Set the version of the Nano/Micro components to the new Pico release in project.properties (/nanocontainer-*/trunk) and commit the change to the SCM.
- If this is a major release, also send announcements to TheServerside.com, FreshMeat.net and JavaLobby.org

Miscellaneous

This page last changed on May 16, 2004 by rinkrank.

These pages contain additional background information.

- Blog Entries
- Container Comparison
- Contextualized Lookup
- Dependency Injection Nirvana
- Differentiators
- History of Inversion of Control
- Inversion of Control Types
- NanoContainer
- Propaganda
- Terminology

Blog Entries

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

(In chronological order)

- Nano and Pico and Maven - Stefan Schmidt
- PicoContainer with UserManager again - Stefan Schmidt
- PicoContainer and Radeox - Stefan Schmidt
- Pico Container - Jeppe - Aspects of Software Development
- PicoContainer - Rickard Öberg
- IoC type2 vs. type3 - Mathias Bogaert
- Why IoC? - Rickard Öberg
- I give you...IoC type 0! - Hani Suleiman (The bileblog)
- Pico T-Shirts : "I was expecting a paradigm shift , and all I got was a lousy constructor" - Zohar Melamed
- On misunderstanding of Class-Component and Interface-Service relationships - Krage/Terminus
- On the structure of IoC Container - Krage/Terminus
- On Component Model and its mapping to java - Krage/Terminus
- A moment of IoC zen - J2gEEk
- PicoContainer 1.0 beta 1 has been released - TheServerSide.com thread
- Logo contest for PicoContainer - Carlos Villela
- IoC Introduction - Funzel's Wiki-Blog thingy
- TSS Article: Introducing the eXo Platform - TheServerSide article
- Blogwatch: A comparison of IoC in Spring and WebWork2 - TheServerSide.com thread.
- What's New in the eXo Platform 1.0 Beta 4 - TheServerSide.com article
- PicoContainer redux - That's sooo '82
- PicoPuzzled - Otaku, Cédric's Weblog
- PicoChickens - That's sooo '82
- Dependency Injection Pattern - Martin Fowler
- Container for Chicks - De tout et de rien
  "... I have started to play with PicoContainer a while ago and here is why I am seduced by PicoContainer (If someone read this entry, he will probably argue that container XXX provides what I am describing, but I don't care). It can be reduced to three words: ..."
- PicoContainer BSH configuration - More to do with NanoContainer really, by Alex Winton
- I don't need no stinking container - Ian Kallens's What's the noise weblog.
- Caucho Tutorial: Dependency Injection for Resources
- Scott Crawford & the EJB 3 Experts Group FAQ http://www.scottcrawford.com/ejb30faq.html
  "...I personally would like to thank Paul Hammant and Aslak Hellesoy for

pushing dependency injection towards me at an early stage of this JSR. I am really impressed that they care more about the success of their ideas than in taking credit or worrying about any competition which may or may not exist between EJB and lightweight containers. They did have an impact on me and EJB 3..."

- Mike Aizatsky lead deverloper at JetBrains (Intellij IDEA people) Listeners Dependency Injection

Container Comparison

This page last changed on May 16, 2004 by rinkrank.

Authors: ?

Overview

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

Spring Framework

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

Avalon

TODO

HiveMind

TODO

Contextualized Lookup

This page last changed on May 16, 2004 by rinkrank.

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 support this type of dependency resultion. Spring Framework does however.

Dependency Injection Nirvana

This page last changed on Nov 10, 2005 by paul.

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
    }
    publicString 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:

```
publicinterface 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);
      returnnew 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 ?

Differentiators

This page last changed on Oct 23, 2005 by paul.

Authors: Paul Hammant

There are a number of specifications from the last few years that proport to be container/component designs...

Sun specified nearly-IoC Containers & Component designs

It could be said that Sun have created a few 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.

Non IoC container-like things by various

Mainable (Sun)

```
publicstatic 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.

Singleton (pattern)

See Singleton Antipattern.

Static Factory (pattern)

Used in a similar way the Singleton pattern (and similarly detailed in "Design Patterns"), this allows disparate classes to create/get other components. Very dirty applications result from overuse of static factories.

(TODO: move to antipatterns section?)

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 grapical 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.

Other IoC container / component designs

Avalon-Framework 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, with a port underway to C#. 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 .

Apache-Avalon 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 type-1 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.

Carbon

todo

History of Inversion of Control

This page last changed on May 16, 2004 by rinkrank.

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.

Inversion of Control Types

This page last changed on Nov 10, 2005 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 :

```
publicinterface 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 :

```
publicinterface 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 :

```
publicinterface 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.

NanoContainer

This page last changed on May 16, 2004 by rinkrank.

NanoContainer Documentation

Propaganda

This page last changed on May 16, 2004 by rinkrank.

Overview

Various declarations on PicoContainer.

Mike Hogan on nanocontainer-dev (17th June 2003)

Fellas, congrats on the best IoC container I've seen. I've played with Avalon, XWork and Pico/Nano and the latter is the best I have used. (link).

Mathias Bogaert (25th June 2003)

paul_hammant: While it is fresh. Could you write a couple of paragraphs on you intial reaction and why you warmed to Pico ?
m_bogaert: sure
m_bogaert: I'll put it on email
paul_hammant: I'll put on the propaganda page
paul_hammant: please also don't hold back on how much of a hard time you gave Aslak
m_bogaert: I do realize that I can moan about stupid naming stuff
m_bogaert: but that's me, the perfectionist
paul_hammant: no dude, from a couple of days ago...
paul_hammant: not the rename stuff
paul_hammant: when you were fonder of type2 Setter Injector
paul_hammant: the intial YIM chat story you had with aslak
m_bogaert: ah ok
m_bogaert: yep I still have it...rereading to see how I evolved
paul_hammant: we can direct "instant objecters" to it in future to save time
m_bogaert: so aslak stated that I was bitching? hehe
m_bogaert: or giving a hard time?

paul_hammant: I was sitting next to him dude
paul_hammant: he did not say you were bitching
paul_hammant: it was clear though that he had an uphill struggle convincing you
paul_hammant: if it was some one he did not like, he might have given up and we would have made an enemy
paul_hammant: best to get the "user story" now to save time in future
m_bogaert: hehe
m_bogaert: ok sent

Here is what you asked for, my initial reaction on Type 3 Constructor Injector IOC:

When I started out with XWork IOC (~~type 2~~ Setter Injector), I really liked the clean enabler interfaces, and the fact that you could switch implementations and scopes by defining it in an XML file. Also the integration with XWork, the fact that I had my own little Actions implement eg. ConfigurationManager was so neat and clean. I was in love.

Then came along ~~Type 3~~ Constructor Injector IOC. My initial reaction was 'aaarg where would my nice enabler interfaces fit?', because ForgotPasswordAction implements UserManagerAware looked so sexy. So I ignored it for a while and kept on converting my components into ~~type 2~~ Setter Injector IOC. Converting. Converting. Damn, soooo many blahAware interfaces. Then I started thinking again. A component should manage itself. A component should enforce their dependencies. A component should not live without context (other components set on it). ~~Type 2~~ Setter Injector IOC doesn't support this all, AND makes me write 2 classes for each component (the component and the enabler).

After a chat with Aslak, I saw the light, and the next day it struck me... ~~type 3~~ Constructor Injector was a better solution then ~~type 2~~ setter Injector.

Terminology

This page last changed on Nov 10, 2005 by paul.

Also known as Picosaurus

Terms

These can be arrived at from the definitions.

- Everything is an object
- Everything which is not a type-1 component nor a domain-specific object is a POJO
- A type-2 component is a JavaBean, and a JavaBean is a type-2 component
- A type-3 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.

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.

Jovial Terms

Picoification

The act of refactoring a codebase to turn existing components into Pico compatible ones.

Picosicate

The art of dithering with respect to a pending Picoification.

Picant

The often false declaration that a component cannot be turned into a pico component. Leaves an unsavoury taste in mouth.

Picography

A retrospective article by a developer on the subject of Picoification of a set of components to the PicoComponent design. Usually a tool that the developer is fond of. Usually to a blog.

Piconate

See Picoification.

Picoscate

Inept Picoification. Code being messy as a result.

Picofile

A developer who rapidly believes PicoContainer to be the greatest thing ever.

Picosistance

The movement against Picoification and PicoContainer. This divided into two camps (who hate each other). There is the StaticIsKing group (many of whom hate each other as well as themselves). There are also fellow IoC believers who hold a candle for alternate IoC designs. Usually Setter Injector or Contextualized Lookup of which there are many. We have not identified any other groups in the Contructor Injector IoC space - could be something to do with Picovelation.

Picovelation

The realisation that there is no more simple or elemental IoC design for component composition than PicoContainer.

Project Information

This page last changed on May 16, 2004 by rinkrank.

These pages contain useful information about the PicoContainer project itself.

- Downloads
- Mailing lists
- Open Issues
- Ports
    - Rico
- Related Projects
- Releases
- Sister Projects
- Slogan
- Source Repositories
- Statistics
- Team
- TShirts

Downloads

This page last changed on Oct 26, 2005 by joehni.

On this page you'll find links to binary and source distributions. Also see Source Repositories.

Releases

- Changelog
- Roadmap

PicoContainer 1.2-RC-1 (2005-09-11)

Core	Gems
• jar • bin distribution (tar.gz) • bin distribution (zip) • src distribution (tar.gz) • src distribution (zip) • Javadocs • All Reports • Source	• jar • bin distribution (tar.gz) • bin distribution (zip) • src distribution (tar.gz) • src distribution (zip) • Javadocs • All Reports

PicoContainer 1.1 (2004-11-04)

- jar
- bin distribution (tar.gz)
- bin distribution (zip)
- src distribution (tar.gz)
- src distribution (zip)
- Javadocs
- All Reports

PicoContainer 1.0 (2004-06-05)

- jar
- bin distribution (tar.gz)
- bin distribution (zip)
- src distribution (tar.gz)
- src distribution (zip)
- Javadocs
- All Reports

Other downloads

Download with Maven

The Codehaus repository http://dist.codehaus.org/ supports the Maven repository structure.
This repository is sychronized any four hours with the repository on http://ibiblio.org/maven, Maven's default remote repository.

If you're using Maven to build a project that depends on PicoContainer, and you want to use always the latest snapshot versions build by Continuous Integration, then you should tell maven to use Codehaus' repository in addition to the one at Ibiblio. Put the following in your project's project.properties file:

```
maven.repo.remote = http://www.ibiblio.org/maven/,http://dist.codehaus.org/
```

Download sources

See Source Repositories

Mailing lists

This page last changed on Sep 29, 2005 by joehni.

List Name	Subscribe	Unsubscribe	Archive	News
PicoContainer User	Subscribe	Unsubscribe	Archive	News
PicoContainer Dev	Subscribe	Unsubscribe	Archive	News
PicoContainer SCM	Subscribe	Unsubscribe	Archive	News
PicoContainer Announce	Subscribe	Unsubscribe	Archive	-
PicoContainer Despots	Subscribe	Unsubscribe	Archive	-

For an explanation of the purpose of each mailing list, see DESPOTS:Project Mailing Lists

Open Issues

This page last changed on May 16, 2004 by rinkrank.

Open issues:

jira.codehaus.org(22 issues)				
T	Key	Summary	Assignee	Res
●	PICO-160	Design flaw working with JavaBeans	Joerg Schaible	UNRESOLVED
●	PICO-246	Ensure Pico 1.1 compatibility	Joerg Schaible	UNRESOLVED
●	PICO-264	Cannot stop already started components if start of container fails	Unassigned	UNRESOLVED
↗	PICO-91	refactor ComponentAdapter+ComponentFactory relationship	Unassigned	UNRESOLVED
●	PICO-197	PicoContainer jar should include proper package version specification	Unassigned	UNRESOLVED
+	PICO-188	Support mutual dependencies for setter injection components	Aslak Hellesoy	UNRESOLVED
	PICO-248	Merge MutablePicoContainer and PicoContainer	Unassigned	UNRESOLVED

		interfaces.		
↗	PICO-105	Remove key from ComponentAdapter	Aslak Hellesoy	UNRESOLVED
✚	PICO-227	Patch to allow CAF to be specified for children	Mauro Talevi	UNRESOLVED
▣	PICO-9	Write a demo application for PicoContainer	Unassigned	UNRESOLVED
↗	PICO-220	Alternate ComponentAdapter Implementation which is more flexible	Unassigned	UNRESOLVED
▣	PICO-258	Demos should also be build by CC	Mauro Talevi	UNRESOLVED
↗	PICO-260	Explanation of CAs	Joerg Schaible	UNRESOLVED
▣	PICO-244	Update docs	Paul Hammant	UNRESOLVED
↗	PICO-253	Per-Instance ComponentMonitor implementation	Unassigned	UNRESOLVED
✚	PICO-256	Inheritance over composition for CachingPicoContainer etc (to debate)	Unassigned	UNRESOLVED
↗	PICO-243	Order of the elements of an array dependency is not preserved	Unassigned	UNRESOLVED
↗	PICO-221	Enable ComponentAdapterFactory	Joerg Schaible	UNRESOLVED

		for visitor		
●	PICO-183	SICA throws Exception when setter cannot be resolved	Unassigned	UNRESOLVED
↗	PICO-181	refactor creation of default parameters	Unassigned	UNRESOLVED
✚	PICO-232	Selective Dependency Injection	Joerg Schaible	UNRESOLVED
☒	PICO-257	Docs reports have no stylesheet	Mauro Talevi	UNRESOLVED

Ports

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

Platform	API	Download	Documentation	Description
Java	Docs, API	Downloads	User Documentation	The original implementation.
.NET		Source Repositories		Follows the Java design closely. Currently a work in progress by Michael Ward and Maarten Grootendorst .
Ruby	Rico	Source Repositories		Follows the ideas of the original implementation, but a quite different API and implementation, due to Ruby's dynamic nature.
PHP		Source Repositories		Follows the ideas of the original implementation. Currently a work in progress by Pawel Kozlowski

Rico

This page last changed on May 16, 2004 by rinkrank.

Author: Dan North

Overview

Rico is a container written in Ruby, demonstrating the same Inversion-of-Control principles as the Java Pico Container.

It is primarily a proof of concept at this stage - there is an embarrassingly small amount of actual code but it seems to cover the fundamentals.

The Constructor Injector IoC model depends on the signature of the constructor(s) of an object to identify its dependencies. Ruby does not have static types, so we use a list of keys to represent dependencies, passed in when we register a component.

Basic operation

require 'rico' # the main container

include Rico # so we don't need to use Rico:: everywhere

container = Rico.new

1. Register a component with key "one"
   container.register "one", Object

1. Register a component with dependencies
   container.register "two", Complex, ["one"]

1. Get a component
   component = container.component "two" # constructs Object, passes it to Complex

2. constructor and returns the result

Additional behaviour

Rico has the same multicasting behaviour as Pico - you can call methods on a multicaster object and any registered components that understand the method will have it called. This is useful for lifecycle management.

Next steps

Rico currently doesn't support the same ComponentAdapter model as Pico, which would bring it much more into line. Also we need a sample app or two to demonstrate how it works.

Related Projects

This page last changed on Nov 10, 2005 by paul.

Authors: Aslak Hellesoy, Paul Hammant

Overview

These are other projects we know of that are using PicoContainer. If you have an addition to this list, please drop us a mail.

Pico Powered Apps & Components

These Products are powered by PicoContainer -
(alphabetical order)

- Backpedal
- Generama Code gen infrastructure for Java
- (The) Exo Platform - integrated application suite
- Jesktop - The Java Desktop, at least before Sun started using the term (PH)
- JingDAO - DAO layer for Java
- Middlegen
- PhoenixJMS - JMS server
- Radeox - wiki application
- SnipSnap - wiki application
- Spice Components - Some of at least
- Thicky - Thick-client experience, page by page.
- XDoclet - Defacto standard code generator for Java.

PicoPowered logo -

In order to add that logo to your own project's documentation, just use this link:

```
<img src="http://www.picocontainer.org/pico-powered.png" alt="Pico Powered"/>
```

PicoCompatible Components and Containers

Reusable components that can be deployed in a PicoContainer
They can all use this logo:

As can alternative containers that pass the PicoContainer Technology Compatibility Kit (TCK)

In order to add that logo to your own project's documentation, just use this link:

```
<img src="http://www.picocontainer.org/pico-compatible.png" alt="Pico Compatible"/>
```

Apache projects compatible with PicoContainer. Apache do not co-brand

- Directory (formerly LDAPd)
- AltRMI

Releases

This page last changed on May 29, 2005 by mauro.

On this page you'll find the online documentation for our releases.

Java

PicoContainer 1.0 (2004-06-05)

- Javadoc Online
- Reports Online
- Downloads here

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

NanoContainer

NanoContainer is a container that allows a scripting front end to PicoContainer. Script languages supported: XML, Groovy, BeanShell and JavaScript (Rhino).

NanoContainer subprojects extend PicoContainer with the following features, things like:

- AOP (through integration with Nanning)
- Web inegration
- Remoting
- JMX
- ...and much more

See http://www.nanocontainer.org.

MicroContainer

A container that takes drop in applications coded according to NanoContainer and PicoContainer ideals. The archive is similar in concept to an EAR file, and automatic JMX and publication is possible. It is a work in progress. See http://www.microcontainer.org. Note this is not to be confused with JBoss Microcontainer which came later.

MegaContainer

Nothing to see yet, but domain registered.

This page last changed on May 16, 2004 by rinkrank.

PicoContainer slogan

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

Courtesy of Zohar Melamed

Why not buy a TShirt with that slogan?

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

Source Repository Structure

The Subversion source repository is structured as different modules, each with its independent main trunk, branches and tags, as it enables us to have independent release cycles for these modules.

The top-level modules are:

- java/
- dotnet/
- ruby/
- php/
- book/
- site/

The platform modules are further subdivided into separate modules for Pico/Nano/MicroContainer (where available). Additional sub-modules per platform can be present, eg for Java NanoContainer, where the module is sufficiently scoped to require its own release cycle.

Browse repository

The repository can be browsed at http://svn.picocontainer.codehaus.org/

Java modules

The Java module paths are

- java/picocontainer/
- java/nanocontainer/
- java/nanocontainer-ioc/
- java/nanocontainer-nanowar/

- java/nanocontainer-persistence/
- java/nanocontainer-remoting/
- java/nanocontainer-sandbox/
- java/nanocontainer-tools/
- java/microcontainer/
- java/demos/

.Net modules

The .Net module paths are

- dotnet/picocontainer/
- dotnet/nanocontainer/

Ruby modules

The Ruby module paths

- ruby/picocontainer/

Php modules

The Php module paths

- php/picocontainer/

Svnserve Access

Svnserve access is via a specific port (typically 3690 on Unix systems) and requires ssh access for developers. This is often not allowed by corporate firewalls.

In the following <module-path> is one of the paths listed above (eg java/picocontainer) and <module-name> is the name of the checked out module in your local working copy.

Anonymous Access

```
svn co svn://svn.picocontainer.codehaus.org/picocontainer/scm/<module-path>/trunk
<module-name>
```

Developer Access over SSH

```
svn co
svn+ssh://<username>@svn.picocontainer.codehaus.org/scm/picocontainer/<module-path>/trunk
<module-name>
```

where <username> is your unix Codehaus username.

Note: that path is different when accessing over SSH, ie scm/picocontainer in place of picocontainer/scm.

WebDav Access

WebDav is an alternative access protocol supported by SVN and being http-based is often allowed by corporate firewalls.

WebDAV over SSL:

```
svn co https://svn.codehaus.org/picocontainer/<module-path>/trunk
```

This access protocol is available to all, but only developers can commit changes. Developers require a unix password for Codehaus username (contact Bob at the Haus to get assigned a password if you don't have one).

To test https commit access, one may eg:

```
svn co https://svn.codehaus.org/picocontainer/java/picocontainer/trunk picocontainer
cd picocontainer
echo "test" > TEST
svn add TEST
svn commit --username username --password your-password --message "Testing https"
```

Note: you only need to specify username and password once per client.

svn cp/mv: currently not possible via WebDav, due to the configuration of httpd proxy. If you need to do any svn cp or mv you will have to use svn+ssh access, until the Haus httpd is upgraded (very soon).

Statistics

This page last changed on May 16, 2004 by rinkrank.

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

Overview

A project like PicoContainer is always made possible by the effort of a lot of people. Read about the developers and the contributers' names below. All of them have worked together to make PicoContainer the most valued IoC framework on the planet.

Committers

PicoContainer committers introduce themselves in their own words and are listed in alphabetical.

Paul Hammant

Paul is 37, has worked for ThoughtWorks UK since 2002 and is co-founder of the PicoContainer project. He used to freelance but is now very happy at TW. Paul practices XP on client sites and loves open source on which he is chief zealot for TW. He formerly worked at Apache on the Avalon project and remains there working on AltRMI. He loves the simplest thing as a design metaphor, hates too much XML. He loves the lowest common denominator (LCD) as a mechanism for facilitating divergent designs. To that end, he encourages multiple implementations of LCD ideas and APIs. Paul hopes to see Milli, Kilo and Mega implementations of the Pico idea. He hopes to see a myriad of interchangeable Pico Components. He hopes to see standard APIs for components emerge from those implementations, but never be forced when teams cannot agree. It would be fairly easy to guess that Paul does not like big up-front design. That not withstanding the fact that he used to do bucket-loads of it Paul with Aslak, wrote the first lines. Paired, under the influence of a couple of beers, and based on Joe's story of Rachel Davies' comments to his setter injector IoC work at the start of June 2003.

It is also worth noting that Paul got drunk with Aslak to start PicoContainer on the day a world class coder was effectively expelled from Apache. He voted against that action and won't go into details, but believes that it was wrong. PicoContainer was a way of handling the pain he felt. It deliberately embraces the LCD ideal, and multiple implementations. Paul will eternally resist the unification of all Pico and Nano containers, and also resist a big fat XML design for declarations of component needs and wants in Pico and Nano containers.

Paul is also well known for Enterprise Object Broker and Jesktop

Aslak Hellesøy

Aslak is very accomplished in the Open Source space and co-founder of PicoContainer. He leads XDoclet, and has taken over QDox from Joe. He wrote most of the impressive MiddleGen, and has refactored Generama out of both. From an observers point of view Aslak is prolific. Aslak was assimilated into ThoughtWorks in 2003. [Words by Paul]

Dan North

Dan is a passionate Agile Coach and Developer. He's adept at a huge range of languages, with Ruby being his current favorite. This is his first foray into Open Source. Dan was assimilated into ThoughtWorks in 2002. [Words by Paul]

Konstantin Pribluda

I'm 32 and live in Wiesbaden. I started programming at the age of 12 and had access to really cool systems ranging from WANG 3200 through PDP 8/11 to System-360 (actually sovjet clones of them, but who cares?). My first "paid for" work was in 1988, and from that point I financed my study freelance software development. I developed fiscal management systems, medical video archiving, internet applications for telcos and java applet games (later was for personal fun)...

After study I continued to freelance (this time for better paying, but also fast failing) companies and started to work on open source software (xdoclet). Now I'm permanently hired project leader, and I'am in position to use (and develop) whatever technologies I see fit - of course xdoclet, pico, nano and a lot of other stuff.

My biggest private interest (besides open source programming) is mountainbike racing.

Jörg Schaible

I was invited by Paul and Aslak early in 2004 to participate on the development of this project. I am currently 36 and write code for nearly 20 years in several languages and lately also Java. I have experience in AI software, did system programming and developed and maintained a plattform-independent GUI and database framework, and had to deal with CMS programming in JavaScript and J2EE. I was always involved in automated builds and tests for all these projects and one artifact left is JsUnit.

Pico got my interest after some first steps with IoC using the Avalon framework. I felt in love with Pico from the first moment, because of its simplicity and natural way of programming. Yet, there is a lot of space for further development on top of it and I am happy to be part of it.

My interests besides programming are my wife (who always gets too less time), reading books (a lot of fantasy and SF stuff), hearing celtic-rooted music (although I like others also lot), and - when in mood - cooking

Last, but not least, beeing a Christian is a part of my life, where I have my roots.

James Strachan

James used to dislike IoC, when the type that forced you to implement interfaces was the only choice. He championed the whole 'beans' effort at Apache, particularly in Jakarta-Commons. Most recently he's one of the CoreDevelopers and a leading figure on Apache's Geronimo project. Most exciting of all of James' activities has to be the new Groovy language (thank god James got off his XML horse - Ed). James refuses to join ThoughtWorks. [Words by Paul]

Mauro Talevi

I first came across Paul at an XTC (eXtreme Tuesdays Club) on Agile development at the Ol' Bank of England Pub in London.
Back then PicoContainer had yet to be born. We were both working with Avalon IoC and in particular the Phoenix microkernel.
Ever since discussing all the limitations of the Avalon framework - in particular the dependency on an API - I knew that
the way forward was the Pico way.

I've worked in Java and enterprise systems for several years in various business domains - ranging from space data
search portal for ESA to real-time messaging at the BBC - and I've come to appreciate the beauty, elegance and power of CDI.

Jon Tirsén

Jon is a passionate fellow who's invidually accomplished in the Open Source space. Nanning is one of his babies, and he's a committer to Prevayler. With Aslak, DamageControl is is latest project. Jon was assimilated into ThoughtWorks in 2003. [Words by Paul]

Joe Walnes

Joe Walnes is one of those high accomplished people, with a trail of "I must work with him" fans and quality Open Source projects behind him. He wrote SiteMesh, and large chunks of OpenSymphony. He started QDox, and XStream. One of the unsung heroes of Java/.Net development. Joe was assimilated into ThoughtWorks in 2002. [Words by Paul]

Michael Ward

I live in Chicago, or at least that is where all my bills are sent. As it seems to be the lingo, I was assimilated into ThoughtWorks back in 2003. I have been developing professionally since 1997 and have had the great opportunity to work and live in many great locations with some exceptionally talented individuals. I was immediately drawn to PicoContainer because of its simplicity. The API is small and does not force you to define all your configurations in XML. The projects I have worked on are easier to test, refactor, debug and extend because a natural side effect of utilizing CDI is a small and easy to comprehend code base.

Have worked/contributed to MicroContainer, the JMX module for NanoContainer, PicoContainer.NET and NanoContainer.NET.

Michael Rimov

Michael?

Emeritus Committers

Emeritus committers are those committers who have contributed significantly to the development of the project,
but have been inactive for a period amount of time.

- Thomas Heller
- Mike Hogan
- Stephen L. Molitor
- Leo Simons
- Chris Stevenson

Contributers

A lot of people have contributed ideas and code to the PicoContainer's code base. See the names in the list below in alphabetical order.

- Nicolas Averseng
- Mathias Bogaert
- Rao Chejarla
- Jeppe Cramon
- Stacy Curl
- Laurent Etiemble
- Scott Farquhar
- Obie Fernandez
- Jay Fields
- Steve Freeman
- Mario Gutierrez
- Matt Ho
- Jacob Kjome
- Rafal Krzewski
- Aapo Laakkonen
- Graham Lea
- Philipp Meier
- Zohar Melamed
- Kouhei Mori
- Mirco Novakovic
- Rickard Öberg
- Kevin O'Neill
- Miguel A. Paraz
- Gilles Philippart
- Michael Rettig
- Adam Rosien
- Pedro Santos
- Jens Schumann
- Nick Sieger
- Paulo Silvera
- Vincent Tence
- Calvin Yu

TShirts

This page last changed on May 16, 2004 by rinkrank.

Buy one from Future Freak today and spread the word! They come in black and white, and have short or long sleeves. They have PicoContainer's logo on the front, and Zohar Melamed's slogan on the back.

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

- License
- Lifecycle
- One minute description
- Patterns
  - Good Citizen
  - Interface Implementation Separation
  - Inversion of Control
    - Dependency Injection
      - Constructor Injection
      - Setter Injection
- Presentations
  - JavaPolis 2003 Slideshow
- Two minute tutorial
- Very Advanced Topics
  - Arrays, Collections and Maps
  - Implementation hiding and Hot swapping
- Writing Components
  - API for Components
  - Component Configuration
  - Mock Objects

Antipatterns

This page last changed on Feb 20, 2004 by tirsen.

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

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 {
    privatefinal 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.

```
publicinterface B {
}

public class BImpl implements B {
}
```

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:

```
publicinterface A {
}

public class AImpl implements A {
}

public class BImpl implements B {
    privatefinal 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 {
    privatefinal 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.

This page last changed on Apr 15, 2004 by rinkrank.

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;
    privateint 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;
    privateint someId;

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

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

    // ...
}
```

'AnotherControllerFactory' is an interface:

```
publicinterface 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) {
        returnnew 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 Antipattern

This page last changed on Oct 23, 2005 by paul.

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).

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.

Compatibility

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

These documents describe how to write components that are compatible across different containers.

- Avalon Framework

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

Authors: Paul Hammant

Overview

Apache's Avalon's component 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 (contextualized depenedency lookup) design.

This document details how to do this manually.

A simple example component

Example component APIs

```
publicinterface Engine {
  void runEngine();
}
publicinterface 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 = newObject(); // 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 = newObject(); // 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;
  privateString 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();
  }
}
```

This page last changed on May 16, 2004 by rinkrank.

Authors: Paul Hammant, Aslak Hellesoy, Jon Tirsen

Overview

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 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).

Container Concepts

This page last changed on Sep 10, 2005 by paul.

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

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

But advanced use does:

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

The Components of PicoContainer

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

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 Oct 22, 2005 by joehni.

Authors: Jörg Schaible

The code snippes 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

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.

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

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 Extending PicoContainer with ComponentAdapter 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 Extending PicoContainer with ComponentAdapter 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 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

FAQ

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

- 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.

```
publicinterface 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(newInteger(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 Mar 30, 2004 by rinkrank.

Author: Paul Hammant

After a long period of watching IoC 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-design component as more important that the container.

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.

Five minute introduction

This page last changed on Oct 23, 2005 by paul.

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:

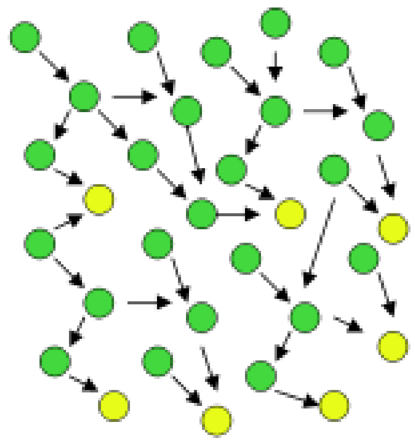
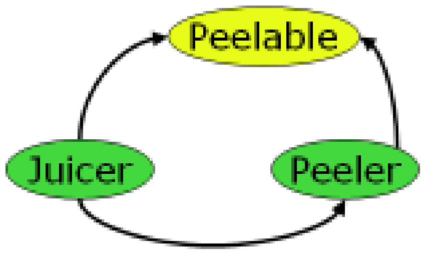
```
MutablePicoContainer pico = new DefaultPicoContainer();
pico.registerComponentImplementation(ArrayList.class);
List list = (List) pico.getComponentInstance(ArrayList.class);
```

(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:

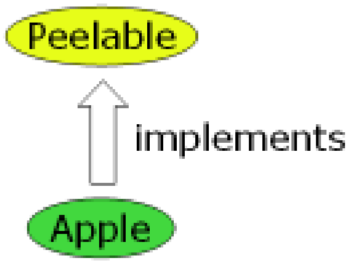
Complex Dependencies	Juicer Example
	

(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):



```
publicinterface Peelable {
    void peel();
}
```

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

```
public class Peeler implements Startable {
    privatefinal Peelable peelable;

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

    public void start() {
        peelable.peel();
    }

    public void stop() {

    }
}
```

```
public class Juicer {
    privatefinal Peelable peelable;
    privatefinal 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);
```

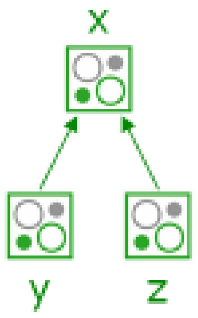
This will cause PicoContainer to do something similar to this behind the scenes (except that PicoContainer uses reflection):

```
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 Antipattern. 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).

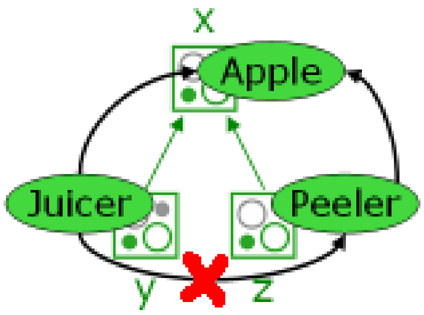


A container (and its registered components) can get access to components registered in a parent container, but not vice-versa. Consider this example, using the classes from above:

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 nullpeeler = (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).

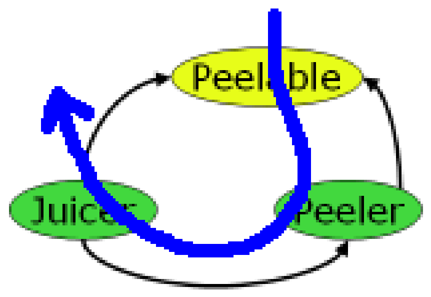
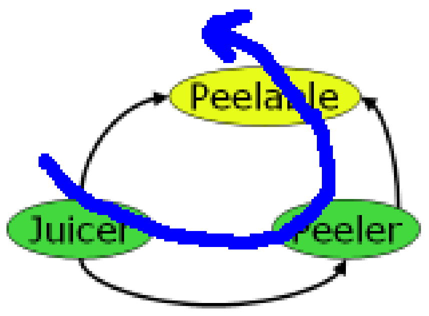
Since this obviously won't work, keep in mind that this was just an exercise to illustrate how container hierarchies work.

For a more concrete example of the usage of container hierarchies, see NanoContainer NanoWar.

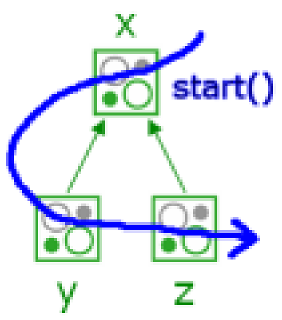
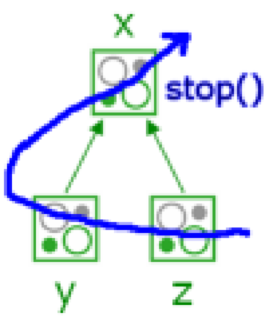
Lifecycle

PicoContainer has support for 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

License

This page last changed on Sep 29, 2005 by joehni.

This page last changed on Aug 24, 2005 by paul.

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:

```
publicinterface 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:

```
publicinterface 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();
```

One minute description

This page last changed on Sep 29, 2005 by joehni.

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

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

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
- Interface Implementation Separation
- Inversion of Control
  - Dependency Injection
    - Constructor Injection
    - Setter Injection

Patterns detailed elsewhere

Separate Interfaces From Implementation

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

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 Sep 05, 2005 by paul.

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 Inversion of Control Types document.

IoC Synonyms

One well-known synonym for IoC is DIP - described in Robert C. Martin's excellent Dependency Inversion Principle paper. A second nickname for IoC is The Hollywood Principle (Don't call us we'll call you).

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 :

```
publicinterface 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 {
  privatestatic 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 ..

```
publicinterface 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

- Dependency Injection
    - Constructor Injection
    - Setter Injection

Dependency Injection

This page last changed on Sep 13, 2004 by rinkrank.

Overview

See "Inversion of Control Containers and the Dependency Injection pattern" for a thorough description.

Subpages

- Constructor Injection
- Setter Injection

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 {
    privatefinal StockManager stockManager;
    privatefinalString 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.

Setter Injection

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

Presentations

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

- JavaPolis 2003 Slideshow

JavaPolis 2003 Slideshow

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

Aslak and Jon paired on a presentation at JavaPolis in 2003. ThoughtWorks paid for the excursion and even rented a booth for them to hang out at.

Contrary to the prevailing opinion in advance of the trip, they were sober, polished and popularly recieved in the presentation itself.

PicoContainer - Inversion of Control made easy (pdf)

Also available in Japanese: PicoContainer - Inversion of Control made easy (PowerPoint)

Two minute tutorial

This page last changed on Aug 24, 2005 by paul.

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.

```
publicinterface 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

Very Advanced Topics

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

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

TODO - Multiple Components of the same Type

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):

```
publicstaticinterface Fish {
}

publicstatic class Cod
        implements Fish {
}

publicstatic 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:

```
publicstatic class Bowl {
    privatefinal Fish[] fishes;
    privatefinal 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:

```
publicstatic class Bowl {
    privatefinal LinkedList fishes;
    privatefinal 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:

```
publicstatic class Bowl {
    privatefinal TreeMap fishes;
    privatefinal 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) {
        protectedboolean 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 {
    publicfinal Man man;

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

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

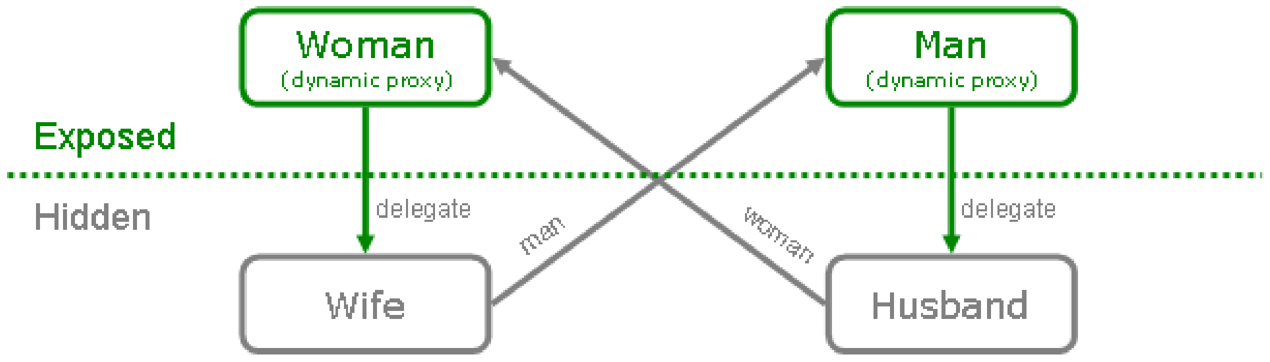
```
public class Husband implements Man {
    publicfinal Woman woman;

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

    publicint getEndurance() {
        return 10;
    }
}
```

```
publicinterface Woman {
    Man getMan();
}
```

```
publicinterface Man {
    int getEndurance();
}
```

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

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).

```
public class Superman implements Man {
    publicint 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?

Writing Components

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

- API for Components
- Component Configuration
- Mock Objects

This page last changed on Nov 10, 2005 by paul.

Sample Component

```
public class Football {
  // methods
}
```

//TODO describe

no real api, not even xml for Pico
apart from startable
cos that's not in the JDK

how you may want to interface/impl separate

advice against static public methods and singletone accessors, links to anti patterns

naming best practice

links to Nano for biggers uses of Pico.

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

Authors: Paul Hammant

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

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);
    pico.addParameterToComponent(Foo.class, String.class, "foo");
    pico.addParameterToComponent(Foo.class, Integer.class, newInteger(33);
    pico.start();
```

Obviously you'd have some soft coded implementation rather than that hard coded above. 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) {
      }
    }

    ....

    PicoContainer pico = new HierarchicalPicoContainer.Default();
    pico.registerComponent(Foo.class);
    pico.registerComponent(Wilma.class, WilmaImpl.class);

    pico.registerComponent(FredImpl.class);

    pico.addParameterToComponent(Foo.class, String.class, "foo");
    pico.addParameterToComponent(Foo.class, Integer.class, newInteger(33);
    pico.start();
```

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 );
    //pico.registerComponent( DefaultFooConfig.class );
//pico.registerComponent( PropertiesFileFooConfig.class );
//pico.registerComponent( XmlFileFooConfig.class );
//pico.registerComponent( ParallelUniverseFooConfig.class );
//pico.registerComponent( WebPageReadingFooConfig.class );
//pico.registerComponent( FromTibcoFooConfig.class );
//pico.registerComponentImplementation (
myAvalonConfigurableForLegacySupportFooConfig );
//pico.registerComponentImplementation ( someCarefullyDeserializedFooConfig );
    pico.start();
```

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.

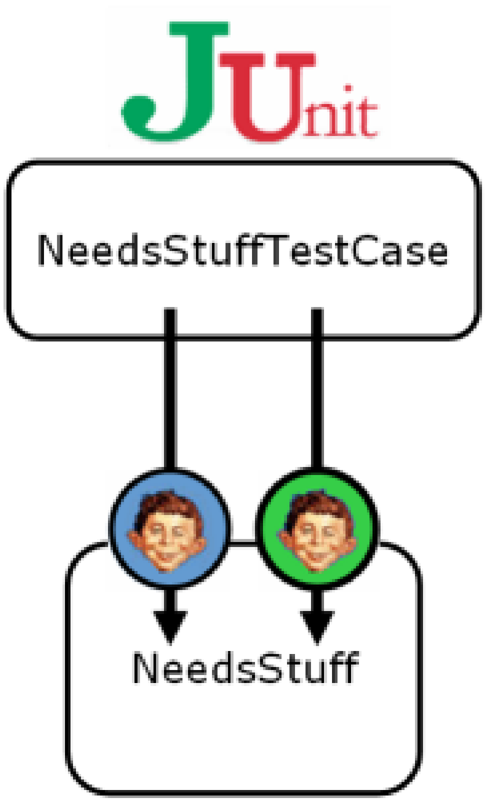
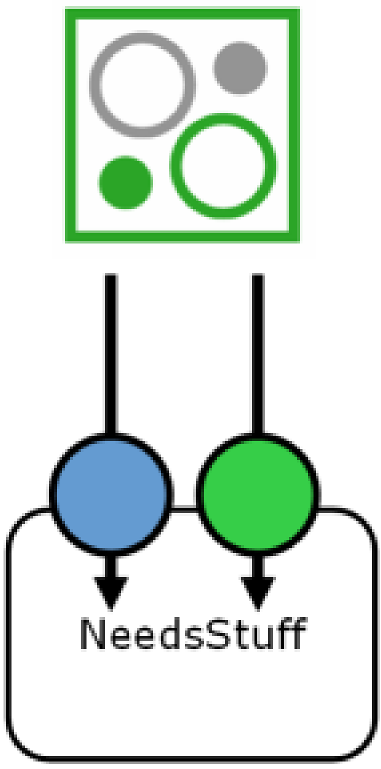
Mock Objects

This page last changed on Oct 23, 2005 by paul.

If you have had it with codebases that drag along everything and the kitchen sink, (possibly using the Singleton Antipattern) 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.
privatefinal BlueStuff bs;
    privatefinal GreenStuff gs;

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

    publicString 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

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

Direct Usage Scenarios

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 metadata choice

(for the plugin component design)

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, and whether you have to go to the tedious level of defining what dependencies are needed in that XML or not.

Extensions & participation in component infrastructure

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

Aspects, Monitoring, Instrmentation, 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.