Reference Documentation

Version 1.1.1

June 13, 2008

Table of Contents

Preface .....
1. Introduction .....
1.1. Getting Started 1
1.1.1. Morph.convert 1
1.1.2. Morph.copy 1
1.2. Customizing Morph 2
1.3. Concepts 2
1.3.1. Transformers 2
1.3.2. Reflectors 2
1.3.3. Wrappers 3
1.3.4. Languages 3
1.3.5. Contexts 3
2. Transformers .....
2.1. Introduction 4
2.2. The Converter Interface 4
2.3. The Copier Interface 4
2.4. Internationalization 5
2.5. The Transformer Interface 5
2.6. Combining Transformers 5
2.7. Transformer Implementations 6
2.7.1. Pre-Built Converters 6
2.7.2. Pre-Built Copiers 7
2.7.3. Writing Custom Transformers 7
2.8. Transforming Arbitrary Object Graphs 7
2.8.1. Introduction 7
2.8.2. Example 8
3. Wrappers .....
3.1. Introduction 13
4. Contexts .....
4.1. Introduction 14

Preface

Morph is a Java framework that eases the internal interoperability of an application. As information flows through an application, it undergoes multiple transformations. Morph provides a standard way to implement these transformations. For example, when a user submits data using an HTML form in a J2EE application, the data typically goes through three transformations. First, HTTP request parameters are converted to presentation-tier command objects. Second, the command objects are converted into business objects. Finally, the business objects are persisted to a database.

In addition to providing a framework for performing transformations like those described above, Morph provides implementations of many common transformations, including all three of the transformations in the example above (1 is partly done now, 2 and 3 are done). As you can see, Morph is surprisingly powerful out-of-the-box, but it can't solve every problem. Instead, it provides a simple API you can use to harness its power for your particular situation. It has been built from the ground up for flexibility and extensibility, and integrates seamlessly with dependency injection frameworks such as Spring [http://www.springframework.org], PicoContainer [http://www.picocontainer.org/] and Hivemind [http://jakarta.apache.org/hivemind/].

Many of the ideas in Morph were inspired by the Apache Jakarta Commons BeanUtils [http://jakarta.apache.org/commons/beanutils/] project, the Apache Jakarta Commons sandbox component called Convert [http://jakarta.apache.org/commons/sandbox/convert/] and the Context [http://jakarta.apache.org/commons/chain/apidocs/org/apache/commons/chain/Context.html] notion of the Apache Jakarta Commons Chain [http://jakarta.apache.org/commons/chain/] project. Morph synthesizes ideas from these various areas into one consistent API. Implementations are be provided that solve many common problems such as mapping HTTP request parameters to POJOs and converting a SQL statement into Java objects.

Special thanks to the developers of Hibernate and the Spring framework. This methods for generating this documentation were taken from Spring, which was in turn adapted from Hibernate.

Chapter 1. Introduction

1.1. Getting Started

The easiest way to use Morph is to use the `Morph` static class. The main operations it supports are `convert`, `copy`, `get` and `set`. These methods allow you to: `convert` an object from one type to another, `copy` information from one object to another (already existing) object, retrieve (`get`) information from anywhere in an object graph and `set` information anywhere in an object graph, respectively.

The benefit of using the `Morph` static class directly is that it's simple, and you don't have to do any special setup. No matter what project you're working on, all you have to do is drop the morph JAR into your project's library directory and reference the Morph class. In addition, you are assured that no matter how the application you are working on is configured, Morph will work the way you're used to.

The drawback of using Morph in this way is that you cannot do any customization. The `Morph` class is a static facade that makes it easy to get started using Morph. To accomplish this goal, it hides all of the powerful customizations Morph provides. Fortunately, once you outgrow the capabilities that come with Morph out of the box, your own custom Morph configuration will live peacefully side by side with the Morph static class. Your existing code that utilizes the Morph static class will continue to work as it did before, and the parts of your application that require special configuration can have that special configuration limited to only those parts of the system that need the added complexity.

1.1.1. Morph.convert

Morph.convert allows you to convert an object from one type to another. Here are some examples:

```
Integer three = new Integer(3);
// code without Morph
String string = new Integer(three);
// or (without using Morph)
string = "" + three;
// code using Morph
String string = Morph.convertToString(three);

String three = "3";
// code without Morph
Integer integer = new Integer(three);
// code using Morph
Integer integer = Morph.convertToIntegerObject(three);

String three = "3";
// code without Morph
int i= new Integer(three).intValue();
// or (without using Morph)
int i = Integer.parseInt(3);
// code using Morph
int i = Morph.convertToInt(three);
```

1.1.2. Morph.copy

Morph.copy allows information from one object to be copied to another object. The object to which information is copied may even be of a different type than the source object. A great example of when you need to do this type of thing is when you need the data in an HttpServletRequest to be available to lower tiers in your application but you don't want to tie your entire application to the servlet API. For example, let's say you are trying to get your data prepared for a method with a signature `IServiceInterface.service(Map data)`:

```
// without Morph
Map data = new HashMap();
for (Enumeration e=request.getParameterNames(); e.hasNext(); ) {
        String param = (String) e.next();
        data.put(param, request.getParameter(param));
}
// with Morph
Map data = new HashMap();
Morph.copy(data, request);
// actually with this particular example could also do
Map data = (Map) Morph.convert(Map.class, request);
```

1.2. Customizing Morph

A common convention in Java programming is to write objects as JavaBeans and expose their configuration parameters as JavaBean properties. The Morph framework exposes all configuration options in this way. This allows Morph to be configured programmatically using simple syntax. An even more powerful way to configure Morph is to use a dependency injection framework such as Spring. (Dependency injection frameworks are also sometimes called Inversion of Control containers, or IoC containers). This will allow you to configure Morph using the same format you use to configure the rest of your application.

If you're completely lost at this point and wondering what the heck a dependency injection framework is, take a look here [http://www.picocontainer.org/Dependency+Injection]. Note that Morph uses Setter Injection rather than Constructor Injection. I (Matt Sgarlata) personally think dependency injection containers are the most significant innovation in computer programming since object orientation. If you don't know what dependency injection is, stop now and take a look at Spring, PicoContainer, or Hivemind! My favorite is Spring.

1.3. Concepts

There is a whole lot of code in the Morph framework, but it all boils down to a few basic types of things: Transformers, Reflectors, Wrappers, Languages and Contexts. Each of these types of things is given its own package. We'll use the remainder of this section to briefly cover these types. For more information on a type, see the reference guide chapter about the type if one has been written. If you need more information or there is no reference chapter, see the JavaDoc [http://morph.sourceforge.net/apidocs/index.html] documentation. Documentation can always be improved, but the documentation for each of the main interfaces is fairly complete.

1.3.1. Transformers

Transformers transform data from one type to another. Transformers were essentially the inspiration for the entire framework, and they are targeted pretty squarely at the original goal for the framework, which was "to be able to convert anything to anything".

1.3.2. Reflectors

Reflectors were originally created to help implement Copiers, which are a type of Transformers. They provide a stateless model for accessing data from two main types of data structures: bean-like structures and container-like structures. It turns out reflectors are so useful, they can implement all sorts of neat functionality. If you have some type of special data type that you need Morph to understand, you probably want to write a Reflector. A good example of this is the `DynaBeanReflector`

1.3.3. Wrappers

Wrappers are very similar to reflectors, and in fact their APIs are nearly identical. The difference is that reflectors are stateless so that transformers can be implemented efficiently. Wrappers are more useful when you want to allow a method to take any type of bean-like or container-like data, but you don't want to have to overload the method for every conceivable bean (e.g. Object, Map) or container (e.g. Array, List)

1.3.4. Languages

Languages define a way to retrieve and modify arbitrary information in an object graph.

1.3.5. Contexts

Contexts are similar to bean wrappers in that they provide stateful access to information stored in a bean-like object. Unlike beans, contexts are backed by the full power of a language, so they can be used to modify and change any information in an object graph. Also, the default context implementations provided with Morph implement the Map interface. This allows you to easily pass Contexts between tiers of an application, even if different tiers of the application are dependent on different APIs (e.g. the Servlet API in the presentation tier and the JDBC API in the resource tier).

Chapter 2. Transformers

2.1. Introduction

A transformer transforms information taken from a source and makes it available at a destination. There are two main types of Transformers: Converters and Copiers. Converters convert an object of one type to a new object of a different type. Copiers copy information from an existing object to an existing object of a different type. Before we get into the reason for having two types of Transformers, let's take a closer look at Converters.

2.2. The Converter Interface

As previously mentioned, Converters allow an object of one type to be converted to an object of a different type. Here is the Converter interface:

```
public interface Converter extends Transformer {

        public Object convert(Class destinationClass, Object source, Locale locale)
                throws TransformationException;

}
```

As you can see, the Converter interface is very simple. By calling the convert method you are saying, "convert `source` into a new instance of `destinationClass`". This is the interface to use when you're doing a simple conversion from one basic type to another. For example, Morph includes converters that will convert a String to an int (`TextToNumberConverter`), a String to a StringBuffer (`TextConverter`) and many other converters.

2.3. The Copier Interface

Now let's take a look at the Copier interface:

```
public interface Copier extends Transformer {

        public void copy(Object destination, Object source, Locale locale)
                throws TransformationException;

}
```

The Copier interface is just as simple as the Converter interface. A call to the copy method basically means, "copy the information from the `source` to the existing `destination`. Copiers are used when you want to avoid or cannot create a new instance of the destination object. For example, if you want to copy the information in a `Map` to a `HttpServletRequest`'s attributes, you can't create a new `HttpServletRequest` request object, because the servlet container already creates the request object, and you can't create your own. An example of when you could but wouldn't want to create a new instance of the destination object is if you have multiple source objects that you want to be combined into one destination object. For example, if you had information in three different `Maps` that you would like copied to a single destination business object, you could call the copy operation multiple times with your existing business object as the destination object for all three copy operations.

Now that we've gone over why there are two different types of Transformers, let's make a simple rule of thumb

you can use to determine if you should implement a Copier or a Converter: *always prefer the Copier interface*. In other words, if the transformation you're writing can be expressed as a Copier, you should implement the Copier interface. This is because any copier can easily implement the convert operation: just create a new instance of the destination class, and then call the copy operation. In fact, if you subclass the `BaseCopier`, you will just have to implement the contract for the copy operation and the `Converter` interface will be automatically exposed for you.

2.4. Internationalization

You may have noticed that both the convert and copy operations have a `locale` parameter. This parameter is useful when you need to internationalize your application. For example, to convert a Double to a String, you can use the `Morph.convertToString(Object, Locale)` method which will delegate to the `NumberToTextConverter`. Now let's say you want the format of the textual representation of the number to be customized according to the locale of your application's users: English speakers use a period as the decimal separator and Spanish speakers use a comma. By passing in the correct `locale`, English users will see the Double 3564.12 as 3564.12 and Spanish users will see that same Double as 3564,12. You can customize the `NumberToTextConverter` by subclassing it and overriding its `getNumberFormat` method. For example, you could customize the converter to include a thousands separator or round decimals to a certain number of digits.

If you don't know the locale of your user or the locale isn't important, you can simply pass `null` in as the Locale.

2.5. The Transformer Interface

So far we've skipped over the base interface for Converters and Copiers to highlight the differences between the two interfaces. Now let's look at the similarities by examining the Transformer interface:

```
public interface Transformer extends Component {

        public Class[] getSourceClasses();

        public Class[] getDestinationClasses();

}
```

These methods allow a transformer to specify the types of transformations it is capable of performing.

This is a different than the one taken by other frameworks. In other frameworks, a transformer is responsible for performing a transformation and a separate registry is used to indicate which transformers can do which transformations. This is like having a restaurant where each person is allowed to eat, but isn't allowed to say what they like to eat. The restaurant's host examines each person and decides what that person will be served without consulting that person. As you can imagine, this gets pretty ugly pretty quick. Logically, each person in the restaurant knows what he or she wants to eat, so why not let them decide?

2.6. Combining Transformers

Transformers are easy to use directly with Morph, but we don't always know exactly what we're converting ahead-of-time. For example, if I have a bunch of objects I want to convert to Strings at once, I don't want to have to write a lengthy if/then statement that picks the right converter. I'd rather just write

`convert(String.class, source, locale)` and have the correct Converter chosen for me. To solve this problem, other frameworks introduced a registry where you state which transformers can be used for which transformations. This solves the problem of choosing how to pick a converter, but as we saw in our restaurant example, it introduces problems of its own.

Morph's solution to this problem is the `DelegatingTransformer`. It's a Transformer just like the other Transformers we've looked at, but instead of doing transformations itself, it delegates to other Transformers. Continuing with our restaurant example, the DelegatingTransformer is like a buffet. Each person that enters the restaurant gets in line for the buffet and each person gets to choose what they would like to eat. Now to really stretch this metaphor: the trick is to arrange the line in such a way that everyone's happy. Put the picky eaters in the front of the line so they can get what they like to eat, and put your puppy that will eat anything at the back of the line so that everything gets eaten.

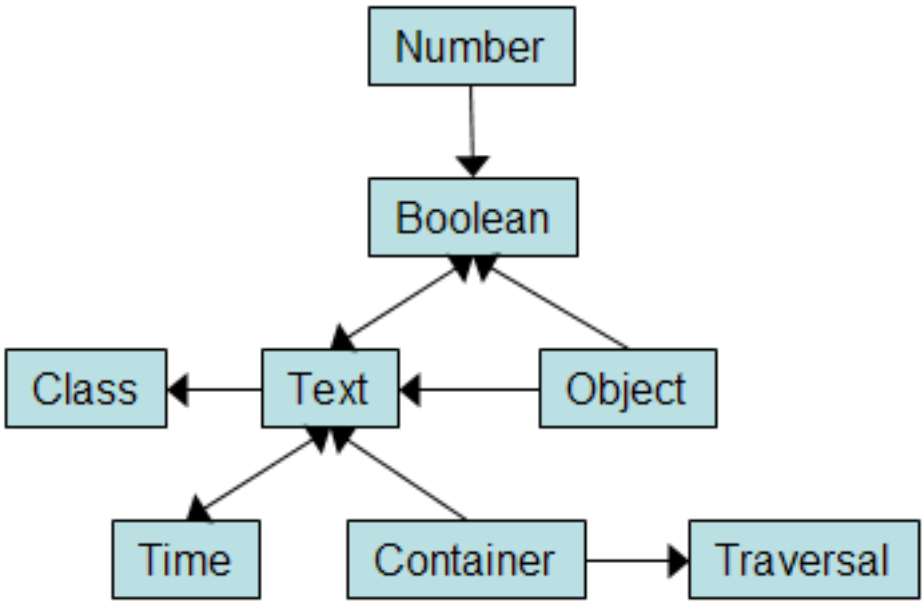
Now let's flee from this crazy restaurant and talk about transformations again. Morph includes both a `NumberToTextConverter` and a `ObjectToTextConverter`. The `ObjectToTextConverter` just calls an Object's `toString` method, whereas the `NumberToTextConverter` nicely formats a number based on a user's locale. Clearly, if we're converting a bunch of objects to Strings, we want the `NumberToTextConverter` to get chosen if the object to be converted is a number. If the object is not a number, we can fall back to the `ObjectToTextConverter`. We specify all this behavior simply by setting the `delegates` property of the `DelegatingTransformer`. The delegates are arranged in order of precedence. When the `DelegatingTransformer` does a transformation, it goes to each transformer in turn and asks if it can perform the requested transformation. If the transformer reaches the end of the list but couldn't find any transformers to do the requested transformation, a `TransformationException` is thrown.

2.7. Transformer Implementations

Morph comes with many Transformers pre-built so that hopefully you won't have to implement any yourself. In this section we'll briefly examine the transformers that are bundled with Morph, and see how to write our own.

2.7.1. Pre-Built Converters

The Converters included with Morph work with all the basic Java types: primitives, Characters, Strings, StringBuffers, Dates, Calendars, Numbers, Iterators, and Enumerators. For a complete list, see the JavaDoc of the `net.sf.morph.transform.converters` package. To get an idea at a glance of what you can convert to what, see the chart below. An arrow from one type to another indicates that a conversion in that direction is possible. For example, Numbers can be converted to Booleans, but not the other way around.



Converters included with Morph

2.7.2. Pre-Built Copiers

The Copiers included with Morph are focused on transferring information between *bean-like* objects and *container-like* objects. Bean-like objects can be copied using the `PropertyNameMatchingCopier`, which copies information from one object to another based on matching up property names in the source and destination objects. For example, if you had a PersonDAO data access object and a Person domain object that each had the properties firstName, middleName and lastName, the `PropertyNameMatchingCopier` would take care of copying the information to and from those two objects automatically.

If the properties don't match, you can use the `PropertyNameMappingCopier`. For example, if PersonDAO used firstName, middleName and lastName as property names and Person used firstName middleName and familyName, the PropertyNameMappingCopier can be customized to do this conversion by setting its `mapping` property.

2.7.3. Writing Custom Transformers

If you need to write a custom transformer, it's easy since Copiers and Converters have such simple interfaces. We recommend you try to extend an existing, pre-built transformer, but if you can't find one that does what you need you can also directly subclass `BaseConverter` or `BaseCopier`. See the JavaDoc for `BaseTransformer` for more information.

2.8. Transforming Arbitrary Object Graphs

2.8.1. Introduction

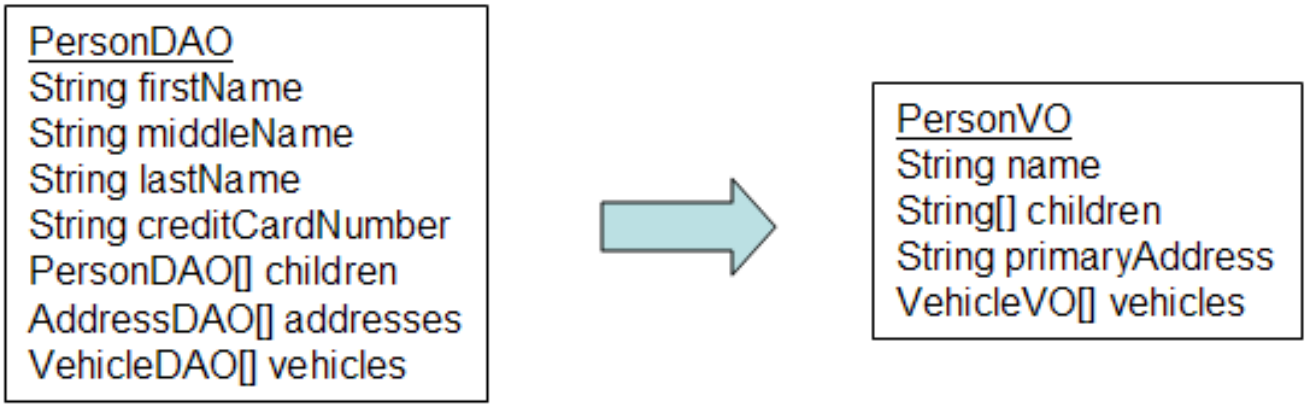
When information is passed between different tiers of an application, it often needs to be transformed into a different format. Essentially, what you need to do is transform one graph of objects into a different graph of objects with similar information. Without Morph, this type of code can quickly become a big mess that is difficult to modify when the structure of either object graph is changed. Morph helps isolate each of the

different types of transformations that are happening using a divide-and-conquer approach. Instead of writing one massive method that does the transformation, you write several Transformer classes, each of which is concerned only with transforming one node in the object graph from one type to another. You then combine all these Transformers using the `DelegatingTransformer`.

2.8.2. Example

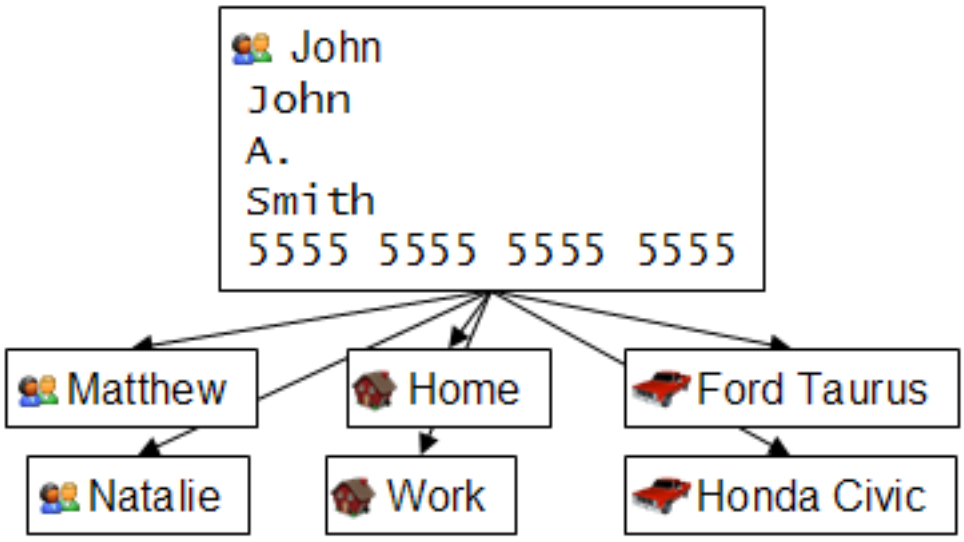
In this section we'll look at an example of transforming a data access object that holds information from a database into a value object to be exposed as part of a web service. Note that this example has been made intentionally as difficult as possible. Most use cases will require far fewer custom transformers to be written. You can see this example in action by examining the `net.sf.morph.examples.person.PersonExampleTestCase`.

Now let's get started. Below are our example objects, a PersonDAO (Person data access object) and a PersonVO (Person value object):



The PersonDAO and PersonVO classes

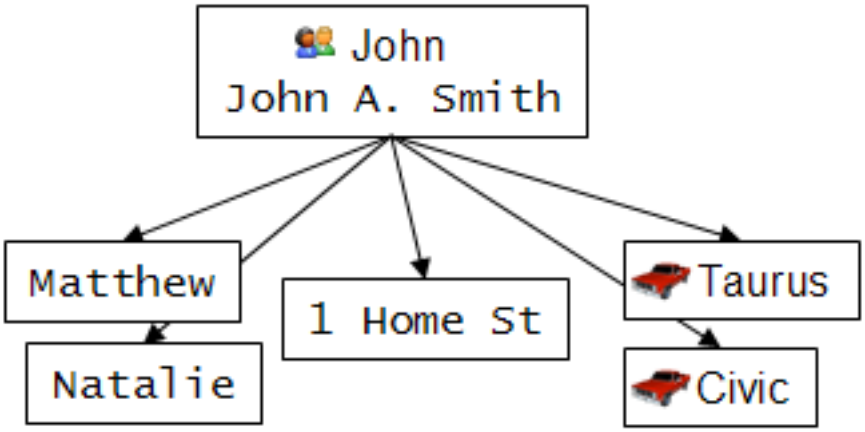
Below is an example PersonDAO object that represents John A. Smith. As we can see, his firstName is John, his middleName is A. and his lastName is Smith. His credit card number is 5555 5555 5555 5555. He has two children, Matthew and Natalie, a home and work address, a Ford Taurus, and a Honda Civic.



John A. Smith represented as a PersonDAO

We would like to convert John's PersonDAO into a PersonVO, like the one shown below. Notice the

creditCardNumber information is removed and his firstName, middleName and lastName have been combined to provide a single name. Similarly, all his address information was squished into a single String by listing only his primaryAddress, and converting it to a String representation. Finally, his Ford Taurus and Honda Civic are now just a Taurus and a Civic, because in our VehicleVO we decided we didn't need to include information about the vehicle's manufacturer.



John A. Smith represented as a PersonVO

2.8.2.1. Transforming VehicleDAO[] to VehicleVO[]

First we'll focus on converting the vehicles property of the PersonDAO to the vehicles property of the PersonVO. We'll assume for this example that the `VehicleDAO` can be converted to a `VehicleVO` by simply using the `PropertyNameMatchingCopier`. If this is the case, the `ContainerCopier` will be able to use the `PropertyNameMatchingCopier` to convert the `VehicleDAO[]` to a `VehicleVO[]` without any further effort on our part.

2.8.2.2. Transforming PersonDAO[] to String[]

For the children property of the `PersonDAO`, we will need to convert a `PersonDAO[]` array to `String[]`. If we assume that a `PersonDAO` can be converted to a `String` by simply calling the object's `toString` method, the `ContainerCopier` can do this conversion. It will automatically delegate to the `ObjectToTextConverter` to handle the `PersonDAO` to String conversion. If we want to write a different converter to handle the `PersonDAO` to `String` conversion, we can configure a `ContainerCopier` to use it by setting the `ContainerCopier`'s `graphTransformer` property.

2.8.2.3. Transforming AddressDAO[] to String

Now we'll focus on copying the PersonDAO.addresses property to the VehicleVO.primaryAddress property. We will also assume that a PersonDAO can be converted to a String by calling the object's toString method. We will have to write our own converter that takes a `PersonDAO[]` and transforms it to a String:

```
public class AddressDAOArrayToStringConverter extends BaseConverter {

        protected Object convertImpl(Class destinationClass, Object source,
                Locale locale) throws Exception {

                // the BaseConverter will make sure the source is of the correct type
                // for us, so we can just do a cast here with no error checking
                AddressDAO[] addresses = (AddressDAO[]) source;
                // we can also assume the source is not null, because we didn't
                // explicitly state that null was a valid source class
                AddressDAO address = addresses[0];
                // now we convert the first address to a String
```

```
                return address.toString();

        }

        protected Class[] getSourceClassesImpl() throws Exception {
                // if we wanted this converter to also handle converting null values
                // to Strings, we could write this line as:
                //
                //      return new Class[] { AddressDAO[].class, null };
                return new Class[] { AddressDAO[].class };
        }

        protected Class[] getDestinationClassesImpl() throws Exception {
                return new Class[] { String.class };
        }

}
```

2.8.2.4. Transforming PersonDAO[] to PersonVO[]

Now that we know which converters we need to transform the properties of a `PersonDAO` to the properties of a `PersonVO`, we are ready to transform our top-level `PersonDAO` object into a top-level `PersonVO` object. We will be able to use the `PropertyNameMappingCopier` to do most of the work, but we will need to subclass it to handle the conversion of the PersonDAO.firstName, personDAO.middleName and personDAO.lastName properties into a single PersonVO.name property. Here is our top-level converter:

```
public class PersonDAOToPersonVOCopier extends PropertyNameMappingCopier {

        protected void copyImpl(Object destination, Object source, Locale locale)
                throws TransformationException {

                super.copyImpl(destination, source, locale);

                // this cast is safe because our superclass makes sure the source is of
                // the correct type and not null
                PersonDAO personDAO = (PersonDAO) source;
                // construct the name
                String name = personDAO.getFirstName() + " "
                        + personDAO.getMiddleName() + " " + personDAO.getLastName();

                // this cast is safe because our superclass makes sure the destination
                // is of the correct type and not null
                PersonVO personVO = (PersonVO) destination;
                // save the name
                personVO.setName(name);

        }

        protected Class[] getDestinationClassesImpl() throws Exception {
                return new Class[] { PersonVO.class };
        }

        protected Class[] getSourceClassesImpl() throws Exception {
                return new Class[] { PersonDAO.class };
        }

}
```

Now that we have all our transformers written, we can go about performing our graph transformation. We can do everything programmatically, or we can use a dependency injection framework. Here is the code we'll need to do things programmatically:

```
// this is the overall transformer we'll use to do the graph copy
DelegatingTransformer graphTransformer = new DelegatingTransformer();
```

```
// AddressDAO[] to String
AddressDAOArrayToStringConverter addressConverter =
        new AddressDAOArrayToStringConverter();
// PersonDAO[] to PersonVO[]
PropertyNameMappingCopier personCopier = new PersonDAOToPersonVOCopier();
Map personMapping = new HashMap();
personMapping.put("children", "children");
personMapping.put("addresses", "primaryAddress");
personMapping.put("vehicles", "vehicles");
personCopier.setMapping(personMapping);
personCopier.setGraphTransformer(graphTransformer);

// the list of transformers that are involved in our overall graph
// transformation
List transformers = new ArrayList();
// always put your custom transformers first
transformers.add(personCopier);
transformers.add(addressConverter);

// then put in the default set of transformers as listed in the
// DelegatingTransformer.  this makes sure all the normal conversions
// you would expect from Morph are available (e.g. Integer 1 -> Long 1)
transformers.add(new DefaultToBooleanConverter());
transformers.add(new NullConverter());
transformers.add(new IdentityConverter());
transformers.add(new DefaultToTextConverter());
transformers.add(new TextToNumberConverter());
transformers.add(new NumberConverter());
transformers.add(new TraverserConverter());
transformers.add(new TextConverter());
// will automatically take care of PersonDAO[] to String[]
transformers.add(new ContainerCopier());
// will automatically take care of VehicleDAO[] to VehicleVO[]
transformers.add(new PropertyNameMatchingCopier());

// convert our list of transformers into an array
Transformer[] transformerArray = (Transformer[]) transformers.toArray(
        new Transformer[transformers.size()]);
graphTransformer.setComponents(transformerArray);

// copy the information from personDAO to personVO
graphTransformer.copy(personVO, personDAO);
```

Below is essentially the same code using Spring. The code may not be much shorter, but I feel it's clearer

```
<beans>

        <!-- VehicleDAO[] to VehicleVO[] -->
        <bean
                id="vehicleCopier"
                class="net.sf.morph.transform.copiers.PropertyNameMatchingCopier"/>
        <!-- PersonDAO[] to String[] -->
        <bean
                id="childrenCopier"
                class="net.sf.morph.transform.copiers.ContainerCopier"/>
        <!-- AddressDAO[] to String -->
        <bean
                id="addressCopier"
                class="net.sf.morph.examples.person.AddressDAOArrayToStringConverter"/>
        <!-- PersonDAO[] to PersonVO[] -->
        <bean
                id="personCopier"
                class="net.sf.morph.examples.person.PersonDAOToPersonVOCopier">
                <property name="mapping">
                        <map>
                                <entry key="children" value="children"/>
                                <entry key="address" value="primaryAddress"/>
                                <entry key="vehicles" value="vehicles"/>
                        </list>
                </property>
                <property name="graphTransformer">
                        <ref bean="graphTransformer"/>
                </property>
        </bean>
```

```
        <!-- the overall transformer we'll use to do the graph copy -->
        <bean
                id="graphTransformer"
                class="net.sf.morph.transform.DelegatingCopier">
                <property name="components">
                        <list>
                                <ref bean="personCopier"/>
                                <ref bean="vehicleCopier"/>
                                <ref bean="childrenCopier"/>
                                <ref bean="addressConverter"/>
                        </list>
                </property>
        </bean>

</beans>
```

Chapter 3. Wrappers

3.1. Introduction

A wrapper allows data in an object to be manipulated and provides a consistent API for different types. For example, instead of overloading a method so that it can accept both a Collection and an Object array, a single method signature utilizing the Container interface may be specified. *This chapter has yet to be written*

Chapter 4. Contexts

4.1. Introduction

A context is an object that can be passed between tiers of an application without exposing the APIs that are particular to any given tier. For example, in a web application the information submitted by a user in an HttpRequest object could be exposed to a business object through the Context interface. This leaves objects in the business tier independent of the servlet API and thus testable outside a Servlet container. *This chapter has yet to be written*