PolyD - User's Manual

Antonio Cunei

April 3, 2005

Department of Computer Sciences
Purdue University

PolyD is a modular dispatching framework that enables the user to use a range of different dispatching mechanisms, and to create new ones when needed. PolyD advocates a modular approach to dispatching, and in particular a separation between the method selection, the method invocation, and all the internal implementation, including the dynamic generation of support classes and all the caching. This manual describes the features of the framework, and the way in which user-defined dispatching policies can be added to the system.

1 Introduction

PolyD can be used whenever a dispatching mechanism different from that offered natively by Java is required. For example, PolyD can be used to implement visitors, multidispatching, mixin-like dispatching, and similar techniques. In order to create a new dispatcher using PolyD, the user should create an interface that describes which messages will be dispatcher, and one or more implementing classes, that define which methods will implement the messages. The selection mechanism is described by means of "dispatching policies", and the method invocation can be customized using "invocation policies".

In order to express the different aspects of the dispatching, PolyD uses Java annotations, which allow for a natural and coincise representation. In order to guarantee backward compatibility, however, a pre-5.0 version of PolyD, which does not rely on annotations and does not use 5.0 features, is also available. An important source of information on PolyD is the public API documentation, available online at `www.ovmj.org/polyd`. This manual summarizes the main features of the framework, and shows some examples of its use.

2 Quickstart

Consider the following class:

```
class Impl {
 void dance(Dancer p, Stage q) {
  printComment("Dance is an expression of art!");
 }
 void dance(Person p, Stage q) {
  printComment("What is that guy doing on the stage?");
 }
 void dance(Person p, Place q) {
  printComment("That person is dancing. Strange.");
 }
}
```

The methods describe three possible responses to a combination of arguments. We would like to use real multimethods so that, even if statically we deal with generic `Person`s and `Place`s, dynamically the most appropriate method is chosen.

The way to create a dispatcher is the following: first of all a proper interface is defined, to specify that multimethods are to be used.

```
@ PolyD
@ DispatchingPolicy (MultiDisp.class)
interface Dance { void dance(Person p,Place q); }
```

The tag `@PolyD` is only used as a marker to denote a PolyD interface. The `@Dispatching-Policy` tag informs the toolkit that all specified methods will use that dispatching policy. We can now build and use the dispatcher:

```
Person joe     = new Person();
Place  office  = new Place();
Person nureyev = new Dancer();
Place  bolshoi = new Stage();

Dance d = PolyD.build(Dance.class,new Impl());
d.dance(joe,bolshoi);
d.dance(nureyev,bolshoi);
d.dance(nureyev,office);
```

Note that in all three cases the arguments to `dance()` are statically a `Person` and a `Place`, but nothing more specific. The output is:

```
What is that guy doing on the stage?
Dance is an expression of art!
That person is dancing. Strange.
```

Let us compare that against overloading:

```
@ PolyD
@ DispatchingPolicy (Overloading.class)
interface Dance { void dance(Person p,Place q); }
```

The result is now:

```
That person is dancing. Strange.
That person is dancing. Strange.
That person is dancing. Strange.
```

Further selection mechanisms are also available. A review of the various options of PolyD follows.

3 Building a Dispatcher

The construction of a dispatcher takes place using `PolyD.build()`. As previously mentioned, it is possible to use multiple bodies for every interface, as follows:

```
Interf d=PolyD.build(Interf.class,a,b,c);
```

where `a`, `b`, and `c` are instances of three different classes. The methods of all those classes are used together to build the dispatcher. Only the methods specified in the interface are used to build the dispatcher; other methods in the bodies are treated as support methods and ignored. The classes used for the bodies need not implement the interface.

A method specified in the interface, with a certain name and arity, causes all public methods with same name and arity to be included in the custom dispatcher. Even static methods are included. For example:

```
class Test {
 public void test(A a) {...}
 public static test(B b) {...}
 public void test(C c) {...}
}
```

is perfectly acceptable. Using a static method *might* lead to slightly faster dispatching, but that depends on what the JVM does with the code.

Multiple methods with same name and arity can be specified in the interface. For instance:

```
@ PolyD
@ DispatchingPolicy (MultiDisp.class)
interface Several {
 void test(A a, X x);
 void test(X x, B b);
}
```

The method `test()` will be usable on arguments that respect the shown combinations, but not others. The methods used in the interface can be of arbitrary arity, and use and return any values, including primitives. Only methods that are public in the implementing classes will be used, the others will be *silently* ignored.

4 @DispatchingPolicy

The tag `@DispatchingPolicy` is mandatory, and specifies the way in which a method is selected when a dispatcher is used. The tag can be used on the interface, or on individual prototypes. The selection on the methodprototypes overrides the specification for the interface. If all the method prototypes are individually tagged, the tag on the interface is optional. Methods with the same name can also be resolved in different ways. Example:

```
@ PolyD
interface Several {
 @ DispatchingPolicy(Overloading.class)
   void test(A a);
 @ DispatchingPolicy(MultiDisp.class)
   void test(C b);
}
```

If `C` and `A` are not related, the behaviour is obvious. If, just to make a convoluted example, `C` is a subclass of `A`, the policy in use depends on which of the two prototypes is chosen by Java, according to its own overloading mechanism. If the argument is statically known to be at least a `C`, multiple dispatching will be used, otherwise overloading is used instead.

There are a few standard dispatching policies available. In Section [later] we will see how to create new policies.

4.1 ovm.polyd.policy. MultiDisp

The policy implements a fully symmetric multiple dispatching. The policy considers inheritance through subclasses and subinterfaces as equivalent. Consequently, the policy also implements a form of multiple inheritance, just by specifying different methods for classes or interfaces.

If a dispatcher contains a method which uses this policy, and the call `PolyD.build()` is completed successfully, there is a guarantee that all subsequent method invocations will always find a matching method. In other words, there will never be a `MissingMethodException`. The policy is also able to detect several ambiguities at dispatcher building time, but not all of them because of Java's dynamic loading. All the remaining ambiguities will be detected later, at dispatching time.

What this boils down to is that only a single implementation of a method will ever be eligible for a certain combination of arguments, which makes multiple inheritance more manageable. If the list of combinations of arguments that are to be used with a certain method is known in advance, then all ambiguities can be detected at dispatcher building time. For more information, chech the option `@Preload`, later.

4.2 ovm.polyd.policy. Overloading

The policy mimics the usual static resolution adopted by Java on the list of arguments. All ambiguities are detected statically and if the call `PolyD.build()` is completed successfully no `MissingMethodException` will ever be thrown.

4.3 ovm.polyd.policy. NonSubsump

The resolution rule used by the `NonSubsump` policy searches at dispatching time for a method implementation whose list of parameter classes matches *exactly* the list of classes of supplied arguments. So, if we have a method defined on "`FieldAccess`", for example, the method will be called on instances of `FieldAccess` but not on instances of its subclasses. If a call is made and the policy cannot find an implementation that matches exactly, then the call is ignored by default, but the behaviour can be customized by the user.

4.4 ovm.polyd. MissingMethodException

Some dispatching policies might be unable to establish at dispatcher building time that all subsequent method calls will be successful. If a dispatching policy is unable to find a suitable method for a given combination of arguments, a default behavior is used. A typical behaviour is throwing a `MissingMethodException`, but every dispatching policy is allowed to specify a standard response. The user can override that default using the tag `@OnMissing`, explained in the following section.

5 @OnMissing

It is possible to specify, for a whole interface or for individual prototypes, what should be the handling for those messages that do not correspond, for the selected dispatching policy, to any applicable method. The tag `@OnMissing` specifies the desired behaviour, selected from the options listed in the following subsections. If a tag is specified for the whole interface, and a different one is speficied for a single prototype, the local one overrides the global selection.

5.1 ovm.polyd.Const.Missing. IGNORE

When a message does not match any method, the call is ignored. If the method is supposed to return a value of some sort, a dummy result is returned instead (zero, or null, or false).

5.2 ovm.polyd.Const.Missing. WARN

As above, but a warning message is additionally printed on the standard error stream.

5.3 ovm.polyd.Const.Missing. FAIL

The request for a message that does not match any available method (according to the selected dispatching policy) causes a `MissingMethodException` to be thrown.

5.4 ovm.polyd.Const.Missing. ABORT

If a matching method cannot be found, the execution aborts with a System.exit(10).

5.5 ovm.polyd.Const.Missing. STANDARD

The handling of the error situation is demanded to the `onMissing()` method of the selected dispatching policy. It is possible to customize this aspect by creating a subclass of the desired policy and overriding `onMissing()`.

6 @InvocationPolicy

In PolyD it is possible to specify additional actions that should be executed when a method is called. A special invocation policy can be attached to the whole interface or to individual prototypes. If no invocation policy is specified, the method is simply called; this is the default and also the faster mechanism.

6.1 ovm.polyd.policy. PlainInvocation

Only supplied as an example, the `PlainInvocation` policy performs a simple invocation, and returns the result supplied by the called method. This policy can be used as a template to create customized invocation policies.

6.2 ovm.polyd.policy. DebuggingInvocation

By adding this invocation policy to any interface or prototype, all method calls will be logged to the standard output stream, together with their arguments and their return values.

7 @IfNull

The value `null` has no class associated to it, and it it not possible to extract the list of classes of the arguments necessary to determine which method is the more appropriate one according to the given policy. When one or more arguments can be `null`, PolyD offers a way to specify the default behaviour. The tag @IfNull can be added to an argument of a prototype in order to specify the class that should be used when that argument is `null`. For example:

```
void test(long i,@IfNull(Place.class)Place p);
```

The class used in the @IfNull must be equal or an arbitrary subclass of the corresponding argument. The @IfNull tag is the faster way to specify the handlilng of null arguments, but a more general approach may be preferable in some cases. If no @IfNull tag is used for an argument, and a null value is encountered, the default handling is demanded to the specific dispatching policy. The handler can be customized by overriding the `remapNull()` method, which converts the list of encountered classes, including nulls, into the list of non-null classes that should be used for that particular message call.

8 @As

In PolyD it is possible to override the dynamic interpretarion of the class of arguments using the `@As` tag. For example:

```
void dance(@As(Person.class) Person a,Place b);
```

The class specified by the tag can be tag `@As` can identical or any superclass of the class of the argument, as long as compatible methods exist in the bodies supplied to build the dispatcher.

The main application of the `@As` tag is the implementation of a generalized form of “super”, that can be applied also in case of multiple inheritance or multiple dispatching. In that sense, specifying an @As class is similar to the qualified super form available in C++. In PolyD, however, the class can be any ancestor of the class of the parameter, and not just a direct superclass.

While the `@As` tag is sufficient to replicate the usual “super” form for individual prototypes, it might be useful to define a more general mechanism to determine a single “super” out of the various possibilities. That functionality can be achieved by defining a custom dispatching policy in which the most appropriate method corresponding to a list of classes of arguments is the method that would be selected by the desired “super” form.

9 @Name

In certain occasions, and we will shortly see examples, it is useful to bind together prototypes and methods even if their names differ. That result can be obtained using the @Name tag, as in the following example:

```
void dance(Person a,Place b);
...
@Name("dance")
@DispatchingPolicy(NonSubsump.class)
void nonSubsumpDance(Person a,Place b);
```

The two prototypes will both use the methods `dance()` defined in the bodies, but with different dispatching policies, or other different features. The tag `@Name` is particularly useful in conjunction with the `@As` tag in order to implement "super". For example:

```
void dance(Person a,Place b);
...
@Name("dance")
void danceSuper(@As(Person)Person a,Place b);
```

The implementations of `dance()` will also be accessible through the name danceSuper(), but in that case the first argument will always be interpreted as a Person. The tag `@Name` can be applied to prototypes in the interface, but it can also be used to rename methods in the bodies, if so desired.

10 @Self

The tag @Self can be applied to variables in the bodies in order to allow a method to call another method using the same dispatcher that was used to reach the current method in the first place. For example:

```
class Body {
  @Self Interf self;

  void m(B x) {
   self.m2(x);
  }
}
```

In the above example, the variable `self` will be automatically initialized when the dispatcher is created:

```
PolyD.build(Interf.class,new Body());
```

There can be multiple variables tagged with @Self, and they may refer to different interfaces. If a single body is shared among multiple dispatchers (which use distinct interfaces), each variable wil be initialized when the dispatcher corresponding to that interface is built. For instance:

```
class Body {
  @Self OverloadingInterface over;
  @Self MultidispInterface multi;

  void m(B x) {
   over.m2(x);
   multi.m2(x);
  }
}
Body b=new Body();
y=PolyD.build(OverloadingInterface.class,b);
z=PolyD.build(MultidispInterface.class,b);
y.m(...)
```

11 @Preload

Each dispatching policy has a choice of how much consistency checking to do statically (at dispatcher-building time) or rather dynamically. For example, the standard policy `MultiDisp` performs an extensive checking that guarantees that, if the dispatcher is built succesfully, no successive call will cause a `MissingMethodException`. Similarly, the policy also tries to determine as many potential ambiguities in the method definitions as possible.

Java, however, is founded on dynamic class loading, and that implies that new classes can potentially introduce new ambiguities or conflicts at any moment, according to the rules of a particular dispatching policy. Consequently, some of the checks could require a lazy approach, done after the main dispatcher construction. Such additional checks are only required for messages involving new classes, and the results are still cached in the built-in system, so the overhead is conceivably marginal.

If, however, the list of classes that will be used is known in advance, and it is preferable to force an early detection of potential error condition, the tag `@Preload` can be used to force the same checks in an eager fashion.

This is an example of `@Preload` in action:

```
@Preload({
  @Seq({Person.class,Office.class}),
  @Seq({Worker.class,Workplace.class}),
  @Seq({Dancer.class,Place.class}),
  @Seq({Dancer.class,Office.class})
})
void dance(Person p,Place q);
```

The specified combinations of classes will be checked and preloaded in the cache.

12 @Raw

The selection of the most appropriate method is most commonly done on the basis of the runtime class of the message arguments. Sometimes, however, it is necessary to pass further information from the call site to the method selector. In this case, raw arguments can be of use. If an argument, in a message prototype defined in the interface, is marked with the @Raw tag, then the argument will be passed "as-is" to the method selector, and it will not be used for the actual method invocation. Only objects or integers can be used as raw arguments.

Raw arguments can be used, for instance, to distinguish among different call sites, even if the concrete arguments are the same; that information can be used to implement general forms of "super". For instance, let us have a class C, subclass of B, subclass of A.

```
class X {
 void visit(C a) {
  ...
  next.visit(X.class,C.class,a);
 }
 void visit(A a) {
  ...
  next.visit(X.class,A.class,a);
 }
}
class Y {
 void visit(B a) {
  ...
  next.visit(Y.class,B.class,a);
 }
}
```

In this case, the first two arguments to the visit message are raw arguments that specify the call site. Those arguments are passed on to the method selection and are used to choose the best "next" visit method. After the next best method has been selected, the raw arguments are stripped and execution continues with the following visit method.

Another application of raw arguments is marking the remaining arguments in order to modify their interpretation. For instance, an enum class can define multiple states, and each raw specification can modify the following argument:

```
d.message(WHITE,a,RED,b,RED,c,WHITE,d);
```

the raw arguments are separated and passed to the method selection. Once the correct method is selected, taking into account the given argument modifiers, the proper `message(a,b,c,d)` will be called.

13 Custom Policies

It is possible to define personalized dispatching and invocation policies. The documentation in the PolyD API is a handy reference to the construction of user-defined dispatchers, and looking at the implementation of the standard policies can also be informative. This section can be used as a general reference about the main aspects involved.

13.1 Dispatching Policies

Each dispatching policy defines different aspects of the method selection and dispatching. The following are the main calls that can be defined.

13.1.1 compatibleSet

The routine `compatibleSet` performs a static preselection, finding in the supplied set of methods those that can are applicable for a given call site, for this dispatching policy. This routine can also be used to perform a consistency check on the set of supplied methods, discovering duplicate methods, violation in covariance rules, ambiguities, conflicts, and so on. If the selection performed by the dispatching policy is entirely dynamic, `compatibleSet` can just return the whole array of method given as argument, without performing any preselection. In this case it is not necessary to override, in the user-defined policy, the default implementation.

The list of classes corresponds to the classes that can be determined statically for a given call site. However, such list is not necessarily the actual list of specific static types of the arguments, but it depends on what Java can discriminate according to the list of prototypes in the interface used to build the dispatcher. An example is required to make this aspect clear. Let's assume that we have a class `A`, its subclass `B`, and a subclass of the latter `C`.

```
interface I {
 void m(A,B);
 void m(B,C);
}
...
 d.m(c,c);
 d.m(b,c);
 d.m(b,b);
 d.m(a,b);
```

Even if we know statically that `c` is of class `C`, `b` of `B`, and `a` of `A`, the four call sites will only be discriminated according to what Java knows according to the interface. The first two calls will be determined statically to be `(B,C)`, the last two `(A,B)`. It is important to keep this aspect present when implementing a policy that has a component of static resolution.

13.1.2 bestMatch

The dynamic counterpart of `consistentSet` is `bestMatch`, which determines in the preselected set the one and only method that is more appropriate for the list of classes supplied, representing the actual dynamic classes of the arguments supplied by the message. The function should return the index in the array of methods corresponding to the best match for the given classes. If no suitable method is found, `bestMatch` should return -1.

13.1.3 handleMissing

The default behavior in case a suitable method cannot be found is specified by the method OnMissing, defined in each dispatching policy. Such method can be overridden in order to implement the most appropriate handler for the specific case. The routine `handleMissing()` accepts as arguments the list of classes that caused the special handling and the set of applicable methods (which can have various names because of the @Name tag).

13.1.4 remapNull

The method `remapNull()` should transform an unexpected sequence of class of arguments, including nulls, into a sequence of non-null classes that can be used to perform the dispatching. If the resulting sequence still contains null values, `remapNull()` is not retried, but an exception is thrown instead.

13.1.5 disableCaching

If this method returns `true`, the standard caching mechanism offered by PolyD is disabled for this policy.

13.1.6 Other

Each dispatching policy should also define a few service methods. An important method is `theDispatcher()`, which should return an instance (any instance) of this dispatching policy. The policy is used as a singleton, so the same object is returned every time. The method `toString()` should return the name of the dispatcher.

13.2 Invocation Policy

The structure of an invocation policy is rather simple. A single method invoke needs to be defined:

```
public Object invoke(Object obj,Method m,Object[] args)
```

The method should perform all the additional operations required by this policy and call the supplied method, of the given object with the given arguments. If some of the arguments are primitives, they are wrapped and unwrapped following the conventions used by `Method.invoke()`.

14 Special dispatching

If the dispatcher makes use of a single body, and the class of the body implements the interface used to build the interface, then it is possible to build a "special" dispatcher that performs internally the dispatching in a slightly different manner, possibly achieveing a marginal performance improvement. The use of this special construction is not particularly recommended, and the feature might disappear altogether in future versions of the tool. The possible performance improvement is likely to be really minimal in any case.

In order to use the special dispatching feature, use this alternate form of the `build` constructor:

```
VisX ii2=buildSpecial(VisI.class,VisX.class);
```

An instance of X will be built and included as part of the dispatcher, which will be an actual subclass of the class of the body. If the construction of the body requires some arguments, the latter can be passed as an additional expression:

```
VisX ii2=buildSpecial(VisI.class,VisX.class,new Object{a,b,c});
```

If some of the arguments are primitives, then the corresponding boxed object can be passed as an argument, and the primitive class can be specified in an additional array of classes, of the same length of the array of arguments for the constructor:

```
VisX ii2=buildSpecial(VisI.class,VisX.class,new Ob-
ject[]{14},new Class[]{int.class});
```

Once again, special dispatching is prone to sudden and mysterious disappearance from the tool in future versions, without particular warnings.

15 The pre-5.0 API

All of the features described thus far are also available in the 1.1-compatible version of PolyD. They can be accessed by using a particular API that will now be described in detail. The API in question is necessarily rather verbose, but it is no more conceptually complicated than the remaining features that we have already seen so far. In the following examples some casts will be omitted for the sake of clarity.

15.1 Descriptors

The construction of dispatchers using the 1.1-compatible API relies on Descriptors, that are used to accumulate the kind of information that PolyD usually obtains exploring the annotations on interfaces and bodies. A new descriptor is created using:

```
Descriptor d1=new Descrip-
tor(Interf.class,new Class[] {BodyA.class,BodyB.class});
```

This descriptor will be used for a dispatcher built using the interface `Interf` and two bodies of class `BodyA` and `BodyB`. The dispatching policy can be added to the descriptor using:

```
d1.setDispatching(MultiDisp.class);
```

Similarly, the global handling for missing methods and the global invocations policy are accessed using the following calls:

```
d1.setInvocation(DebuggingInvocation.class);
d1.setMissingHandling(Missing.Ignore);
```

In order to set the properties of indiviual methods, they need to be extracted reflectively:

```
mt=Dance.class.getMethod("visit",new Class[]{Place.class});
d3.setMethodDispatching(mt,MultiDisp.class);
d3.setMethodName(mt,"lxn");
d3.setMissingHandling(mt,Missing.Ignore);
d3.setMethodPreload(mt2,new Class[][]
  {{String.class,Object.class},{Object.class,Place.class}});
d3.setMethodAsClasses(mt2,new Class[]{Place.class,null});
d3.setMethodRawClasses(mt3,new boolean[]{false,true,false,false});
d3.setMethodNullDefaults(mt4,new Class[]{null,Object.class,null});
```

The calls are auto-explicative. In the case of `setMethodAsClasses() and setmethod-NullDefaults()`, each position in the array is `null` for the position for which no default is specified. The last property that can be specified is the use of `@Self` variables:

```
f=Body.class.getField("self");
d3.setSelfField(f);
```

Once the descriptor is ready, it is possible to proceed with the creation of the actual dispatcher. It is possible to do so in various ways.

15.2 Factories

The easiest and by far fastest way to create a dispatcher is the use of a factory. The creation of a new factory, and the creation of new dispatchers, is illustrated by the following example:

```
Factory fact=d5.register();
Interf disp1=fact.getDispatcher1(bod1);
Interf disp2=fact.getDispatcher1(bod2);
```

This approach minimizes the time required to build a new dispatcher. If the descriptor was created specifying one body, the method `getDispatcher1()` should be used, if two bodies are used then use getDispatcher2() and so on up to `getDispatcher4()`. If more than four bodies are used, the method `getDispatcherN()` will accept an array of bodies. It is your responsibility to pass to `getDispatcher()` bodies that are compatible with the list of classes used to build the descriptor. If that requirement is not satisfied, the dispatcher creation will abort with an error.

15.3 Registered Dispatchers

If, due to the code configuration, it is not possible or practical to pass around a factory, it is still possible to create dispatchers after the `register()` operation as follows:

```
d5.register();
...
Interf disp1=buildFromDescriptor(Interf.class,bod1);
Interf disp2=buildFromDescriptor(Interf.class,bod2);
```

Essentially, after a descriptor is registered in the system, the creation of a new dispatcher using buildFromDescriptor() will look for the more recently registered descriptor associated with the supplied interface and classes of bodies, and will use the corresponding implementation to build new dispatchers. The mechanism is functionally equivalent to the use of factories, except for the speed penalty involved in looking up the maps to find the implementation.

15.4 Special Dispatchers

Even special dispatchers are accessible using the 1.1-compatible API, but only using registered descriptors:

```
d5.registerSpecial();
...
VisX ii2=buildSpecialFromDescriptor(VisI.class,VisX.class,new Ob-
ject[]{14},new Class[]{int.class});
```

The construction of special dispatchers, because of the extra lookup, is slightly slower than the construction of a regular dispatcher using a factory. Using a special dispatcher in order to save time during dispatching is a matter of personal preference, but overall there is a reasonable advantage only is the dispatcher is created once and used massively afterwards, and if it is reasonable to sacrify future compatibility and additional flexibility for such a small improvement.

16 Runabout Emulation

As an exercise to test the flexibility of PolyD, and in order to measure its performance on the field, a support layer for applications that use the Runabout was prepared, tested, and debugged. The result was included in the PolyD source tree, and its content is documented in this section.

16.1 ovm.polyd.legacy. RunaboutBis

`RunaboutBis` is a reasonably accurate replacement for the standard Runabout. It does support the standard `visitAppropriate(Object)` and `visitAppropriate(Object,Class)`, it supports `visitDefault()` and handles primitives. It does not support the special call `addExternalVisit()`, but it is an otherwise pretty complete and functional implementation.

16.2 ovm.polyd.legacy. RunaboutCore

It is also accessible as `ovm.polyd.legacy.Runabout`. `RunaboutCore` is similar to `RunaboutBis`, but slightly simpler, and it offers no support for primitives. The features are otherwise the same.

16.3 ovm.polyd.legacy. RunaboutStat

RunaboutStat is based on RunaboutCore and has the same basic features. While it runs, it keeps a complete count of all the dispatchers created and invoked. The final summary can be obtained by calling RunaboutStats.printStats().

16.4 ovm.polyd.legacy. RunaboutQuick

The RunaboutQuick is a fast implementation that uses factories to speed up the creation of new dispatchers. The speed increment can be rather substantial. It always ignores missing methods and does not handle primitives. Furthermore, it requires the source of original Runabout implementations to be slightly changed, adding a small bit of extra code as follows:

```
class Xyz extends Runabout() {...}
```

becomes

```
class Xyz extends RunaboutQuick {
   private static ovm.polyd.Factory fact=
     RunaboutQuick.prepare(Xyz.class);
   public Xyz() { super(); fact.getDispatcher1(this); }
...}
```

Apart from these small modifications, the rest of the pre-existing implementations remains unchanged. The resulting program can actually run faster than the original Runabout on certain machines.

ovm.polyd.legacy. RunaboutDisp

The file contains a port of the core method selection strategy used by the original Runabout. In order to obtain a faithful emulation, the code has been preserved mostly unchanged, with some minor adaptations. This policy only applies to unary methods, and in principle, being a perfectly standard PolyD dispatching policy, it can be used in other contexts as well. Basically, it was possible to recreate an alternative implementation using PolyD just by creating this new policy (less than 150 lines of code), while all the remaining infrastructure (code generation, caching, and so on) remained unchanged.

17 Layout of the package

The whole source of PolyD is contained in the `ovm.polyd` package, and its subpackages. In particular:

- `ovm.polyd.PolyD` contains the crucial "build" method, and related variations
- `ovm.polyd.Factory` and `ovm.polyd.Descriptor` contain public utils necessary to create dispatchers using the pre-5.0 API.
- `ovm.polyd.tag` contains all the annotations
- `ovm.polyd.policy` contains all of the standard policies
- `ovm.polyd.runemu` contains the Runabout emulation layer
- `ovm.polyd.test` and `ovm.polyd.test14` are simple tests, and are not contained in the distributed jar files.

There are various files corresponding to each build of PolyD:

polyd-YYYYMMDD-HHMM.jar is the jar file containing the Java-5.0 version of PolyD.

polyd-pre50-YYYYMMDD-HHMM.jar is the jar file containing the pre-5.0 version.

polyd-src-YYYYMMDD-HHMM.tgz is the main commented source tree.

polyd-runemu-YYYYMMDD-HHMM.jar is the Runabout Emulation Layer for PolyD.

polyd-runemu-pre50-YYYYMMDD-HHMM.jar is the pre-5.0 version of the Runabout Emulation Layer for PolyD.

polyd-runemu-src-YYYYMMDD-HHMM.tgz is the source of the Runabout Emulation Layer for PolyD.

polyd-javadoc-YYYYMMDD-HHMM.tgz contains the JavaDoc documentation for the public API of PolyD.

polyd-test-YYYYMMDD-HHMM.tgz is a file containing tests for both the 5.0 and the pre-5.0 version of PolyD.

The above files are available on `www.ovmj.org/polyd`. The pre-5.0 versions were tested using Sun's 1.3.1 VM, and also run under Sun's 1.2.2 JRE (although with minor, non-fatal warnings from the VM).

18 A word of caution

For misterious reasons, the JVM available on Mac OS X does not seem to compile efficiently, using JIT, the classes dynamically generated by PolyD, leading to some perfomance degradation. Sun's JVMs on x86 are not affected, and the performance is generally quite good.