



Dynamic languages, Ruby, and Rails (and why web developers like them)

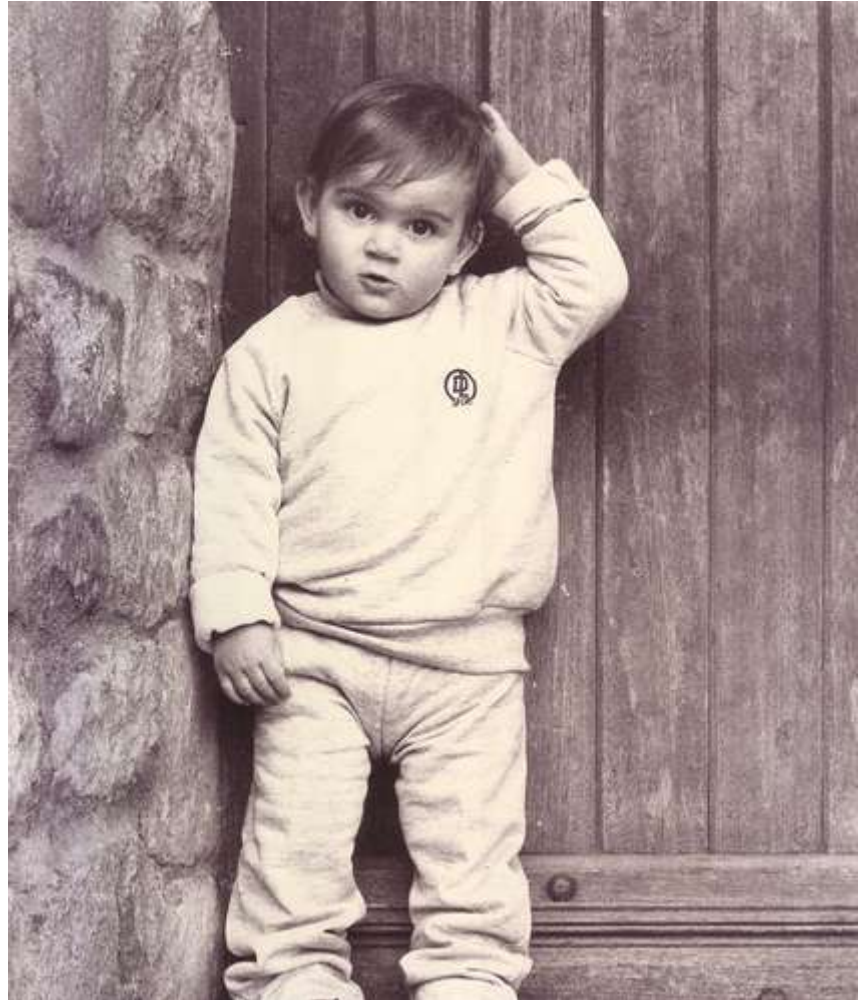
Dr. Elliot Smith
Software Engineer, Talis

September 2008



What is a dynamic language?

A dynamic language is
one which employs
dynamic typing





Static vs. dynamic typing

- **Static**
 - Variables know the type of their objects
 - "Unsafe" operations rejected at compile time (e.g. method which doesn't exist, passing object of wrong type to a method)
 - Objects have one immutable type and must be cast to change type
 - Typically have manifest types (see next slide)
- **Dynamic (tends to be associated with "scripting languages")**
 - Objects know their types; variables don't care
 - "Unsafe" operations rejected at runtime
 - Objects can change type
 - Typically neither manifest nor implicit types

Implicit vs. manifest typing: an aside

- Implicitly-typed
 - Types are implicit, e.g. ML
 - The system infers the types
- Manifestly-typed
 - Types are explicit, e.g. Java
 - Programmer defines the types
- Dynamically-typed languages tend to be neither implicitly- nor manifestly-typed
 - The type only matters at runtime
- Back to static vs. dynamic...





Examples (static vs. dynamic)

// Java (static, manifest)

```
public class EmployeeMangler {  
    public static void main(String[] args) {  
        String employee = "Elliot";  
        employee = 1;    // won't compile  
    }  
}
```

Ruby (dynamic)

```
employee = "Elliot"  
employee = 1
```



Strong vs. weak typing: another aside

- "In a weakly typed language, variables can be implicitly coerced to unrelated types [...] in a strongly typed language they cannot, and an explicit conversion is required."
- Weakly-typed
 - Allows operations between mismatched types, e.g. PHP, Perl
 - Implicit type coercion
- Strongly-typed
 - Prevents operations between mismatched types, e.g. Python, Ruby
 - Explicit type coercion
- A dynamic(ally-typed) language can be strongly- or weakly-typed



Examples (strong vs. weak)

// PHP (dynamic, weak)

```
<?php
echo 1 + "1";      // = 2
echo "1" + 1;     // = "11"
?>
```

Ruby (dynamic, strong, neither implicit nor manifest types)

```
>> puts 1 + "1"
TypeError: String can't be coerced into Fixnum
    from (irb):1:in `+'
    from (irb):1
(puts 1.to_s + "1" or puts 1 + "1".to_i do work)
```

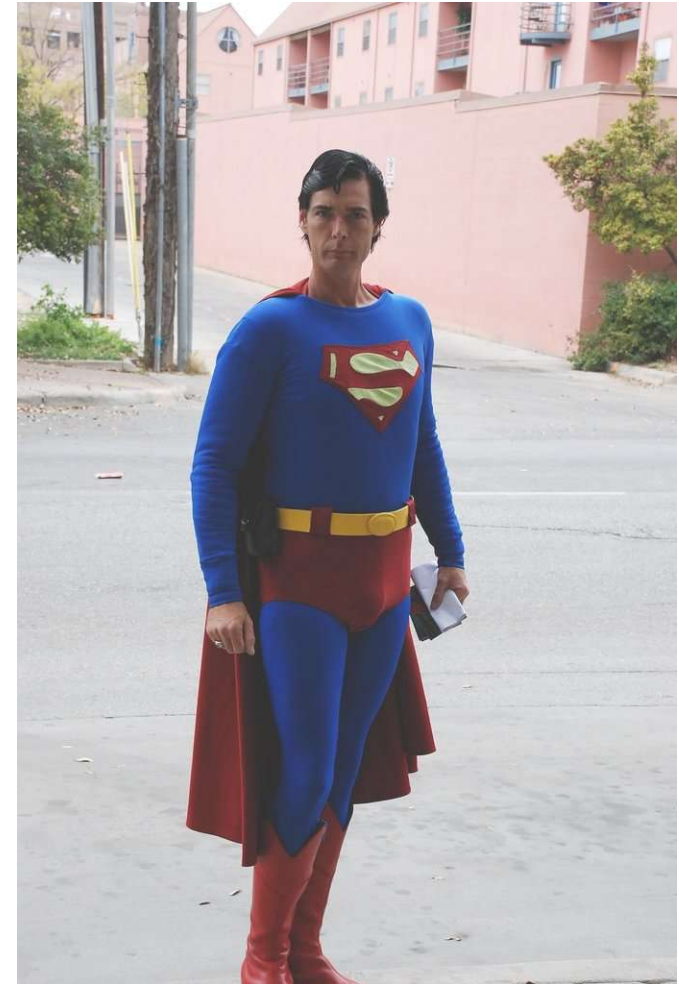


Pros and cons of static typing

- Pros
 - IDEs can be more helpful: code completion, refactoring
 - Helps avoid errors introduced by mismatched types
 - "Unsafe" operations rejected at compile time
 - BUT "If it's not tested, it's broken." (Bruce Eckel)
 - May run faster: because the compiler knows the types in use, it can produce optimised code
- Cons
 - Lots of machinery to put in place to get a program moving (one file per class in Java)
 - Code is verbose: you have to specify types for variables

Pros and cons of dynamic typing

- Pros
 - Code is less verbose
 - Typically a faster edit-compile-test-debug cycle
 - Enables dynamic meta-programming (though Java 7 is catching up)
- Cons
 - Code completion is harder
 - No type checking at compile time: you become reliant on run-time testing
 - May be lots of implicit code which isn't readily visible



Which is better: dynamic or static?

Hibernate vs. ActiveRecord

- Both are ORM layers
 - Provide a mapping from classes to database tables
- Hibernate = Java: static, manifest
- ActiveRecord = Ruby: dynamic
- Hibernate is more "enterprise":
 - Supports stored procedures, composite primary keys, legacy schemas etc.
- But they are functionally similar



```

public class Event {
    private Long id;
    private String title;
    private Date date;
    public Event() {}
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    public Date getDate() { return date; }
    public void setDate(Date date) { this.date = date; }
    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }
}

```

```

package org.openadvantage.eventmanager;
import org.hibernate.Session;
import java.util.Date;
import org.openadvantage.eventmanager.Util;
public class Manager {

```

```

    public static void main(String[] args) {
        Manager mgr = new Manager();
        mgr.createAndStoreEvent("BCS Meeting", new Date());
        Util.getSessionFactory().close();
    }
    private void createAndStoreEvent(String title, Date theDate) {
        Session session = Util.getSessionFactory().getCurrentSession();
        session.beginTransaction();
        Event theEvent = new Event();
        theEvent.setTitle(title);
        theEvent.setDate(theDate);
        session.save(theEvent);
        session.getTransaction().commit();
    }
}

```

```

package org.openadvantage.eventmanager;
import org.hibernate.*;
import org.hibernate.cfg.*;
public class Util {

```

```

    private static final SessionFactory sessionFactory;
    static {
        try { sessionFactory = new Configuration().configure().buildSessionFactory(); }
        catch (Throwable ex) {
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }
    public static SessionFactory getSessionFactory() { return sessionFactory; }
}

```

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="connection.url">jdbc:mysql://localhost/eventmanager</property>
    <property name="connection.username">eventmanager</property>
    <property name="connection.password">police73</property>
    <property name="connection.pool_size">5</property>
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="current_session_context_class">thread</property>
    <property name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>
    <property name="show_sql">>true</property>
    <property name="hbm2ddl.auto">create</property>
  </session-factory>
</hibernate-configuration>

```

```

<?xml version="1.0"?>
<project name="hibernate-testing" default="compile" basedir=".">
  <property name="src.dir" value="src"/>
  <property name="build.dir" value="build"/>
  <property name="build.classes" value="${build.dir}/classes"/>
  <property name="build.lib" value="${build.dir}/lib"/>
  <property name="lib.dir" value="lib"/>
  <path id="cp">
    <fileset dir="${lib.dir}"><include name="*.jar"/></fileset>
    <pathelement location="${build.classes}"/>
  </path>
  <target name="clean" description="Removes all generated files and directories">
    <delete dir="${build.dir}"/>
  </target>
  <target name="prepare" description="Generate directories for holding build">
    <mkdir dir="${build.dir}"/>
    <mkdir dir="${build.classes}"/>
    <mkdir dir="${build.lib}"/>
  </target>
  <target name="compile" depends="clean,prepare,copy-resources"
description="Compiles all source code">
    <javac srcdir="${src.dir}" destdir="${build.classes}" classpathref="cp" />
  </target>
  <target name="copy-resources">
    <copy todir="${build.classes}">
      <fileset dir="${src.dir}">
        <exclude name="**/*.java"/>
      </fileset>
    </copy>
  </target>
  <target name="run" depends="compile">
    <java fork="true" classname="org.openadvantage.eventmanager.Manager" classpathref="cp"/>
  </target>
</project>

```

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate
Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="org.openadvantage.eventmanager.Event" table="events">
    <id name="id">
      <generator class="native"/>
    </id>
    <property name="title"/>
    <property name="date" type="timestamp"/>
  </class>
</hibernate-mapping>

```

```
// connect.rb
```

```
require 'rubygems'  
require_gem 'activerecord'  
require 'mysql'
```

```
ActiveRecord::Base.establish_connection(  
  :adapter => "mysql",  
  :host    => "localhost",  
  :username => "eventmanager",  
  :password => "police73",  
  :database => "eventmanager"  
)
```

```
// manager.rb
```

```
require 'event'  
e = Event.new  
e.title = "BCS meeting"  
e.date = Time.now  
e.save
```

```
// create_events.rb
```

```
require 'connect'  
class CreateEvents < ActiveRecord::Migration  
  def self.up  
    create_table :events do |t|  
      t.column :title, :string  
      t.column :date, :datetime  
    end  
  end  
  def self.down; drop_table :events; end  
end  
CreateEvents.down  
CreateEvents.up
```

**Not required once database table has
been created**

```
// event.rb
```

```
require 'connect'  
class Event < ActiveRecord::Base  
end
```



Conclusions?

- **Hibernate:**
 - Better support for legacy database structures
 - Less "magic" - explicit mapping of instance methods to fields and classes to tables
 - Standardised deployment into application servers
 - More efficient - connection pooling, caching etc.
ActiveRecord catching up
- **ActiveRecord:**
 - Code is more readable and less verbose (partly because of use of dynamic meta-programming: more later...)
 - Everything is written in Ruby (no "XML sit-ups")
 - Obvious things are done for you
- **Obvious point: Appropriateness depends on the situation**



Why Ruby is a dynamic language par excellence

- Domain-specific languages (DSLs)
 - Take advantage of Ruby's syntactic sugar
- Runtime class extension (aka monkey patching)
 - Enables the butchery of Rails internals in plugins
- "Method missing"
 - Provide methods "on the fly"
- Mixins
 - Multiple inheritance without the complexity
- Blocks/Closures
 - Pass functions as arguments to other functions



Later...

How Rails takes advantage of Ruby's features



But first...

**Rapid application
development**



Rails takes advantage of Ruby



Domain-specific languages

```
class Post < ActiveRecord::Base
  # references the id field of the authors table
  belongs_to :author
  # adds methods at run-time
  validates_presence_of :title
end
```

```
class Author < ActiveRecord::Base
  has_many :posts
end
```



DSLs are performing run-time class (instance) extension

- `belongs_to` is a class method from `ActiveRecord::Base`
- When `belongs_to` is called while defining the `Post` class, it appends a series of methods to any instances of `Post`, e.g.

```
p = Post.new
p.author
p.author?
p.create_author
```
- These methods are not in the `Post` class: only in instances of that class
- `belongs_to` has altered the `Post` constructor



Method missing

- Dynamic finders, e.g. `find_by_first_name_and_last_name`
 - Not generated for every possible field combination
- They are actually handled by Ruby's `method_missing` method (in a roundabout way):
 - If a method is called on an object, but the method doesn't exist, you can capture the call
 - It can then be routed to a different method which **does** exist
- In this case, the `find_by_title` method is missing:
 - So `method_missing` manipulates the arguments passed to `method_missing`, and invokes the `find` method with the manipulated arguments



Mixins

- Useful if you are writing your own Rails classes, e.g.
 - You want all your models to share functionality
 - But you don't want to rewrite the inheritance hierarchy
 - All models subclass `ActiveRecord::Base`, so you could edit the source code for that class
 - You could do runtime class extension, but it's unnecessary
- Instead, you can write a **module**
 - Which can be "mixed in" to any class you like
- Example: `find_like`



Blocks/Closures

- **Blocks** are very useful for writing custom iterators, e.g. for array transformations:

```
posts = Post.find(:all)
posts.sort_by {|p| p.title}
posts.select {|p| p.author_id?}
```

- `{ ... }` = **block**; `|p|` = **block parameter**
- A **closure** is a block with local variables "frozen" into it
 - A closure can be passed as an argument to another function(!)
- Example: enables arbitrary functions as controller filters in Rails



Other good things about Rails

- MVC
- Standardised directory layout
- Full stack framework
 - Unit testing, templating, helpers, ORM, AJAX
- Generators and scaffold
 - Productivity tools to get up and running quickly
- Convention over configuration
 - Don't need configuration of the obvious things, e.g. mapping URLs to controllers and actions



Questions?



Dynamic languages, Ruby, and Rails (and why web developers like them)

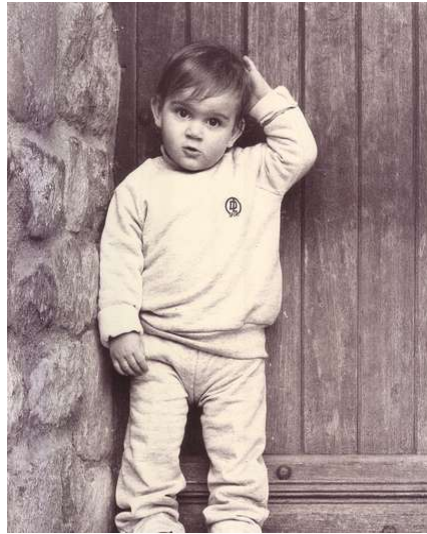
Dr. Elliot Smith
Software Engineer, Talis

September 2008



What is a dynamic language?

A dynamic language is
one which employs
dynamic typing





Static vs. dynamic typing

- Static
 - Variables know the type of their objects
 - "Unsafe" operations rejected at compile time (e.g. method which doesn't exist, passing object of wrong type to a method)
 - Objects have one immutable type and must be cast to change type
 - Typically have manifest types (see next slide)
- Dynamic (tends to be associated with "scripting languages")
 - Objects know their types; variables don't care
 - "Unsafe" operations rejected at runtime
 - Objects can change type
 - Typically neither manifest nor implicit types

Implicit vs. manifest typing: an aside

- Implicitly-typed
 - Types are implicit, e.g. ML
 - The system infers the types
- Manifestly-typed
 - Types are explicit, e.g. Java
 - Programmer defines the types
- Dynamically-typed languages tend to be neither implicitly- nor manifestly-typed
 - The type only matters at runtime
- Back to static vs. dynamic...



Image: "Happy Halloween" by Catskills Grrl

(<http://www.flickr.com/photos/catskillsgrrl/58073391/>)

License: <http://creativecommons.org/licenses/by-nc-nd/2.0/>



Examples (static vs. dynamic)

```
// Java (static, manifest)  
public class EmployeeMangler {  
    public static void main(String[] args) {  
        String employee = "Elliot";  
        employee = 1;    // won't compile  
    }  
}
```

```
# Ruby (dynamic)  
employee = "Elliot"  
employee = 1
```



Strong vs. weak typing: another aside

- "In a weakly typed language, variables can be implicitly coerced to unrelated types [...] in a strongly typed language they cannot, and an explicit conversion is required."
- Weakly-typed
 - Allows operations between mismatched types, e.g. PHP, Perl
 - Implicit type coercion
- Strongly-typed
 - Prevents operations between mismatched types, e.g. Python, Ruby
 - Explicit type coercion
- A dynamic(ally-typed) language can be strongly- or weakly-typed

Top quotation from:

http://www.ferg.org/projects/python_java_side-by-side.html#typing



Examples (strong vs. weak)

// PHP (dynamic, weak)

```
<?php
echo 1 + "1";    // = 2
echo "1" + 1;   // = "11"
?>
```

Ruby (dynamic, strong, neither implicit nor manifest types)

```
>> puts 1 + "1"
TypeError: String can't be coerced into Fixnum
    from (irb):1:in `+'
    from (irb):1
(puts 1.to_s + "1" or puts 1 + "1".to_i do work)
```



Pros and cons of static typing

- Pros
 - IDEs can be more helpful: code completion, refactoring
 - Helps avoid errors introduced by mismatched types
 - "Unsafe" operations rejected at compile time
 - BUT "If it's not tested, it's broken." (Bruce Eckel)
 - May run faster: because the compiler knows the types in use, it can produce optimised code
- Cons
 - Lots of machinery to put in place to get a program moving (one file per class in Java)
 - Code is verbose: you have to specify types for variables

Pros and cons of dynamic typing

- Pros
 - Code is less verbose
 - Typically a faster edit-compile-test-debug cycle
 - Enables dynamic meta-programming (though Java 7 is catching up)
- Cons
 - Code completion is harder
 - No type checking at compile time: you become reliant on run-time testing
 - May be lots of implicit code which isn't readily visible

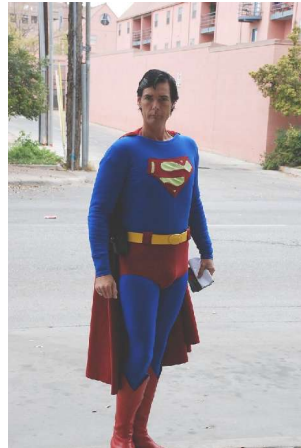


Image: "Super" by Freakishly Tall

(<http://www.flickr.com/photos/saucydelarosa/415696433/>)

License: <http://creativecommons.org/licenses/by-nc/2.0/>

Which is better: dynamic or static? Hibernate vs. ActiveRecord

- Both are ORM layers
 - Provide a mapping from classes to database tables
- Hibernate = Java: static, manifest
- ActiveRecord = Ruby: dynamic
- Hibernate is more "enterprise":
 - Supports stored procedures, composite primary keys, legacy schemas etc.
- But they are functionally similar



12

Image: "Postmodern Dialogues - Round One: Freudian psychoanalysis versus socialist/anarcha-feminism" by huxleyesque

(<http://www.flickr.com/photos/huxleyesque/277993413/>)

License: <http://creativecommons.org/licenses/by-sa/2.0/>

```

public class Event {
    private Long id;
    private String title;
    private Date date;
    public Event() {}
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    public Date getDate() { return date; }
    public void setDate(Date date) { this.date = date; }
    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }
}

package org.openadvantage.eventmanager;
import org.hibernate.Session;
import java.util.Date;
import org.openadvantage.eventmanager.Util;
public class Manager {
    public static void main(String[] args) {
        Manager mgr = new Manager();
        mgr.createAndStoreEvent("ICS Meeting", new Date());
        Util.getSessionFactory().close();
    }
    private void createAndStoreEvent(String title, Date theDate) {
        Session session = Util.getSessionFactory().getCurrentSession();
        session.beginTransaction();
        Event theEvent = new Event();
        theEvent.setTitle(title);
        theEvent.setDate(theDate);
        session.save(theEvent);
        session.getTransaction().commit();
    }
}

package org.openadvantage.eventmanager;
import org.hibernate.*;
import org.hibernate.cfg.*;
public class Util {
    private static final SessionFactory sessionFactory;
    static {
        try { sessionFactory = new Configuration().configure().buildSessionFactory(); }
        catch (Throwable ex) {
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }
    public static SessionFactory getSessionFactory() { return sessionFactory; }
}

```

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0/EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="connection.url">jdbc:mysql://localhost/eventmanager</property>
        <property name="connection.username">eventmanager</property>
        <property name="connection.password">password</property>
        <property name="connection.pool_size">5</property>
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="current_session_context_class">thread</property>
        <property name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>
        <property name="show_sql">true</property>
        <property name="hibern2.default_schema">create</property>
        <mapping resource="org/openadvantage/eventmanager/Event.hbm.xml"/>
    </session-factory>
</hibernate-configuration>

```

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate
Mapping DTD 3.0/EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="org.openadvantage.eventmanager.Event" table="events">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="title">
            <property name="date" type="timestamp"/>
        </class>
</hibernate-mapping>

```

```

<?xml version="1.0"?>
<project name="hibernate-testing" default="compile" basedir=".">
    <property name="src.dir" value="src"/>
    <property name="build.dir" value="build"/>
    <property name="build.classes" value="${build.dir}/classes">
    </property name="build.lib" value="${build.dir}/lib">
    </property name="lib.dir" value="lib">
    </path id="cp">
        <fileset dir="${lib.dir}"><include name="*.jar"></fileset>
        <pathelement location="${build.classes}">
        </path>
    </target name="clean" description="Removes all generated files and directories">
        <delete dir="${build.dir}">
        </target>
    </target name="prepare" description="Generate directories for holding build">
        <mkdir dir="${build.dir}">
        <mkdir dir="${build.classes}">
        <mkdir dir="${build.lib}">
        </target>
    </target name="compile" depends="clean,prepare,copy-resources"
description="Compiles all source code">
        <javac srcdir="${src.dir}" destdir="${build.classes}" classpathref="cp" />
    </target>
    </target name="copy-resources">
        <copy todir="${build.classes}">
        <fileset dir="${src.dir}">
            <exclude name="**/*.java">
        </fileset>
    </copy>
    </target>
    </target name="run" depends="compile">
        <java fork="true" classname="org.openadvantage.eventmanager.Manager" classpathref="cp"/>
    </target>
</project>

```

```
// connect.rb
require 'rubygems'
require_gem 'activerecord'
require 'mysql'

ActiveRecord::Base.establish_connection(
  :adapter => "mysql",
  :host => "localhost",
  :username => "eventmanager",
  :password => "police73",
  :database => "eventmanager"
)
```

```
// manager.rb
require 'event'
e = Event.new
e.title = "BCS meeting"
e.date = Time.now
e.save
```

```
// create_events.rb
require 'connect'
class CreateEvents < ActiveRecord::Migration
  def self.up
    create_table :events do |t|
      t.column :title, :string
      t.column :date, :datetime
    end
  end

  def self.down; drop_table :events; end
end
CreateEvents.down
CreateEvents.up
```

Not required once database table has been created

```
// event.rb
require 'connect'
class Event < ActiveRecord::Base
end
```



Conclusions?

- Hibernate:
 - Better support for legacy database structures
 - Less "magic" - explicit mapping of instance methods to fields and classes to tables
 - Standardised deployment into application servers
 - More efficient - connection pooling, caching etc.ActiveRecord catching up
- ActiveRecord:
 - Code is more readable and less verbose (partly because of use of dynamic meta-programming: more later...)
 - Everything is written in Ruby (no "XML sit-ups")
 - Obvious things are done for you
- Obvious point: Appropriateness depends on the situation



Why Ruby is a dynamic language par excellence

- Domain-specific languages (DSLs)
 - Take advantage of Ruby's syntactic sugar
- Runtime class extension (aka monkey patching)
 - Enables the butchery of Rails internals in plugins
- "Method missing"
 - Provide methods "on the fly"
- Mixins
 - Multiple inheritance without the complexity
- Blocks/Closures
 - Pass functions as arguments to other functions



Later...

How Rails takes advantage of Ruby's features



But first...

Rapid application development



Rails takes advantage of Ruby



Domain-specific languages

```
class Post < ActiveRecord::Base
  # references the id field of the authors table
  belongs_to :author
  # adds methods at run-time
  validates_presence_of :title
end
```

```
class Author < ActiveRecord::Base
  has_many :posts
end
```



DSLs are performing run-time class (instance) extension

- `belongs_to` is a class method from `ActiveRecord::Base`
- When `belongs_to` is called while defining the `Post` class, it appends a series of methods to any instances of `Post`, e.g.

```
p = Post.new
p.author
p.author?
p.create_author
```
- These methods are not in the `Post` class: only in instances of that class
- `belongs_to` has altered the `Post` constructor

* If two modules attempt to Monkey-Patch the same method, one of them (whichever one runs last) "wins" and the other patch has no effect.

* It creates a discrepancy between the original source code on disk and the observed behaviour that can be very confusing when troubleshooting, especially for anyone other than the Monkey-Patch author.

* A Monkey-Patch can lead to upgrade problems when the patch makes assumptions about the patched object that are no longer true.



Method missing

- Dynamic finders, e.g. `find_by_first_name_and_last_name`
 - Not generated for every possible field combination
- They are actually handled by Ruby's `method_missing` method (in a roundabout way):
 - If a method is called on an object, but the method doesn't exist, you can capture the call
 - It can then be routed to a different method which **does** exist
- In this case, the `find_by_title` method is missing:
 - So `method_missing` manipulates the arguments passed to `method_missing`, and invokes the `find` method with the manipulated arguments



Mixins

- Useful if you are writing your own Rails classes, e.g.
 - You want all your models to share functionality
 - But you don't want to rewrite the inheritance hierarchy
 - All models subclass `ActiveRecord::Base`, so you could edit the source code for that class
 - You could do runtime class extension, but it's unnecessary
- Instead, you can write a **module**
 - Which can be "mixed in" to any class you like
- Example: `find_like`



Blocks/Closures

- **Blocks** are very useful for writing custom iterators, e.g. for array transformations:

```
posts = Post.find(:all)
posts.sort_by {|p| p.title}
posts.select {|p| p.author_id?}
```
- `{ ... }` = **block**; `|p|` = **block parameter**
- A **closure** is a block with local variables "frozen" into it
 - A closure can be passed as an argument to another function(!)
- Example: enables arbitrary functions as controller filters in Rails



Other good things about Rails

- MVC
- Standardised directory layout
- Full stack framework
 - Unit testing, templating, helpers, ORM, AJAX
- Generators and scaffold
 - Productivity tools to get up and running quickly
- Convention over configuration
 - Don't need configuration of the obvious things, e.g. mapping URLs to controllers and actions



Questions?