Mean Bean 2.0.3

User Guide

1. Contents

2. Introduction

2.1. Welcome

Welcome to the Mean Bean User Guide. Mean Bean is an Open Source Java test library whose purpose is to make testing JavaBeans and "plain old java objects" (POJOs) as easy as possible.

2.2. Features

At present Mean Bean provides 3 main features:

- It tests that the getter and setter method pairs of a class function correctly.
- It verifies that the equals and hashCode methods of a class comply with the Equals Contract and HashCode Contract respectively.
- It verifies that the properties you believe are considered in an equals method are actually considered, and vice versa. We call this "property significance verification".

2.3. Why Test This Stuff?

You may be wondering: "why test this stuff?"

We think: "why not?" After all – it is code that might contain a bug.

Nowadays JavaBeans and POJOs form the bulk of an enterprise application's domain model. Even EJB, the once heavyweight behemoth, is POJO-centric as of version 3. If JavaBeans and POJOs form the bespoke core of your application, the domain model, then testing them is crucial. Mean Bean helps you do this easily.

Once you know the fundamental elements of your objects are correct, you can test business logic confident that the base on which it is built is rock solid.

2.4. Why Use Mean Bean?

Mean Bean helps you rapidly and reliably test fundamental objects within your project, namely your domain and data objects.

With just a single line of code, you can be confident that your beans are well behaved…

```
new BeanTester().testBean(MyDomainObject.class);
```

3. Using a Test Framework

3.1. Framework Agnostic

Mean Bean is not tied into any specific testing or unit testing framework. When Mean Bean detects a test failure, it ultimately just throws a standard java.lang.AssertionError with an appropriate message.

3.2. Using Mean Bean With JUnit

To use Mean Bean in a JUnit test, simply call it. For example, this JUnit 4 example test uses a BeanTester to test the public getter and setter methods of the class MyJavaBeanDomainObjectfunction correctly.

```
public class DomainTest {
  ...
  @Test
  public void gettersAndSettersShouldFunctionCorrectly() {
    BeanTester tester = new BeanTester();
    tester.testBean(MyJavaBeanDomainObject.class);
  }
}
```

You can place any Mean Bean test logic with a JUnit test in a similar manner.

```
public class MyTest {
  ...
  @Test
  public void myTestName() {
    // Mean Bean test logic ...
  }
}
```

4. Testing Setter/Getter Methods

4.1. Introduction

You can easily test the getter and setter methods of a class using Mean Bean's BeanTester. Pass the class in question to the testBean method and Mean Bean will throw an AssertionError if any getter or setter methods behave unexpectedly.

Currently Mean Bean only tests properties that have both a public getter and setter method. We will extend test coverage to include reduced levels of visibility in the future.

4.2. Algorithm

The BeanTester testBean method implements the following algorithm:

```
for i in 1 .. n do
  for each property in public getter/setter method pairs do
    generate suitable test data for property
    invoke setter with test data
    invoke getter
    test that getter returned same value as passed to setter
  end for
end for
```

Where n is the number of test iterations to perform per class. The default is 100, but this can be altered. Test configuration is covered later in this guide.

If the getter does not return the same value as the test data passed to the setter, the test has failed and an AssertionError is thrown with an appropriate message.

4.3. Supported Property Types

Mean Bean ships with Factories that generate test data for all standard Java types.

4.3.1. Primitive Types

- boolean
- byte
- short
- int
- long
- float
- double
- char
- Object types
- Boolean
- Byte
- Short
- Integer
- Long
- Float
- Double

- Character
- String
- Date

4.3.2. Lists

- List
- ArrayList
- LinkedList

4.3.3. Maps

- Map
- HashMap
- IdentityHashMap
- LinkedHashMap
- TreeMap
- WeakHashMap

4.3.4. Sets

- Set
- HashSet
- LinkedHashSet
- TreeSet

4.3.5. Collections

- Collection
- Queue
- Deque

If you think we've missed a "standard" type, please let us know and we'll add it. We'll cover how to create and register your own Factories later in this guide.

4.4. Unsupported Property Types

When Mean Bean encounters a property type it does not recognise, it cannot use any of the Factories it ships with. For example, when testing the Person class, the Address would not be recognised:

```
class Person {

  private Address address;

  public void setAddress(Address address) {
    this.address = address;
  }

  public Address getAddress() {
    return address;
  }
}
```

In this situation Mean Bean will try to create a special Factory on-the-fly based on the unrecognised type. This special Factory will create a new instance of the type (e.g.

Address), set all of its public properties (as of 2.0.0), and use it as test data. This will only be possible if the type has a no-arg constructor. If Mean Bean succeeds in instantiating an instance of the unrecognised type, it logs a warning. If you find that Mean Bean is failing to recognise what you believe is a standard type, please let us know and we will add it.

If the unrecognised type does not have a no-arg constructor, an exception will be thrown instructing you to register a custom Factory. Creation and registration of custom Factories is covered later in this guide.

4.5. Default Behaviour

The following example uses the BeanTester default behaviour. It tests that the public getter/setter method pairs of the class MyJavaBeanDomainObject function correctly. The default behaviour of the BeanTester testBean method is to test each and every public getter and setter method pair with randomly generated values n times. The default value of n is 100.

```
BeanTester tester = new BeanTester();
tester.testBean(MyJavaBeanDomainObject.class);
```

4.6. Configuring The Number Of Tests-Per-Property Globally

The number of times each getter/setter pair is tested can be altered for all classes by setting the iterations on the BeanTester instance before invoking the testBean method on any class you want to test. The following example changes the number of iterations to 75, meaning that each getter/setter pair of MyJavaBeanDomainObject and AnotherJavaBeanDomainObject is tested 75 times:

```
BeanTester tester = new BeanTester();
tester.setIterations(75);
tester.testBean(MyJavaBeanDomainObject.class);
tester.testBean(AnotherJavaBeanDomainObject.class);
```

4.7. Configuring The Number Of Tests-Per-Property For A Single Invocation

The number of times each getter/setter pair is tested can be altered for a single class by creating a Configuration object, setting the iterations on it, and passing this to the BeanTestertestBean method. The following example changes the number of times the getter/setter pairs of MyJavaBeanDomainObject are tested to 55:

```
BeanTester tester = new BeanTester();
Configuration configuration = new ConfigurationBuilder()
  .iterations(55).build();
tester.testBean(MyJavaBeanDomainObject.class, configuration);
```

Unless this Configuration object is provided, other invocations of testBean use the global iterations setting.

In the following example MyJavaBeanDomainObject‘s methods are tested 55 times whereas AnotherJavaBeanDomainObject‘s methods are tested 100 times.

```
BeanTester tester = new BeanTester();
Configuration configuration = new ConfigurationBuilder()
```

```
      .iterations(55).build();
    tester.testBean(MyJavaBeanDomainObject.class, configuration);
    tester.testBean(AnotherJavaBeanDomainObject.class);
```

4.8. Ignoring A Property

You can instruct Mean Bean to ignore (do not test) a named getter/setter pair ("property"). To do this, create a Configuration, ignore properties on it, and pass it to the BeanTester testBean method.

Given the following class MyJavaBeanDomainObject:

```
public class MyJavaBeanDomainObject {

  private String firstName;
  private String lastName;
  private int favouriteNumber;
  private Date dateOfBirth;

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  public int getFavouriteNumber() {
    return favouriteNumber;
  }

  public void setFavouriteNumber(int favouriteNumber) {
    this.favouriteNumber = favouriteNumber;
  }

  public Date getDateOfBirth() {
    return dateOfBirth;
  }

  public void setDateOfBirth(Date dateOfBirth) {
    this.dateOfBirth = dateOfBirth;
  }
}
```

The following example tests properties: firstName and favouriteNumber, and ignores properties: lastName and dateOfBirth.

```
BeanTester tester = new BeanTester();
Configuration configuration = new ConfigurationBuilder()
  .ignoreProperty("lastName")
```

```
      .ignoreProperty(“dateOfBirth”)
      .build();
    tester.testBean(MyJavaBeanDomainObject.class, configuration);
```

Unless this Configuration object is provided, other invocations of testBean will test all public getter/setter pairs as usual.

In the following example only MyJavaBeanDomainObject‘s firstName and favouriteNumberproperties are tested, whereas all of AnotherJavaBeanDomainObject‘s getter/setters pairs are tested.

```
    BeanTester tester = new BeanTester();
    Configuration configuration = new ConfigurationBuilder()
      .ignoreProperty(“lastName”)
      .ignoreProperty(“dateOfBirth”)
      .build();
    tester.testBean(MyJavaBeanDomainObject.class, configuration);
    tester.testBean(AnotherJavaBeanDomainObject.class);
```

5. Testing Equals Methods

5.1. Introduction

If you have implemented your own equals method in a class, overriding the equals method inherited from Object, you can easily test it using Mean Bean's EqualsMethodTester. TheEqualsMethodTester testEqualsMethod method does the following:

- Verifies that the equals method complies with the Equals Contract.
- Verifies that the properties you believe are considered in an equals method are in fact considered, and vice versa. We call this "property significance verification".

If the equals method does not comply with the Equals Contract, or exhibits unexpected behaviour with respect to property significance, the test has failed and an AssertionError is thrown with an appropriate message.

Currently Mean Bean (2.0.3) only tests the significance of properties that have public getter and setter methods. This is because the object manipulation logic used in the EqualsMethodTester is built on top of existing getter/setter manipulation logic used in the BeanTester. Using this logic allowed us to rapidly deliver the majority of functionality – but it is limited. As a priority we plan to alter EqualsMethodTester to manipulate objects directly via their fields rather than their getter/setter methods.

5.2. Algorithm

The EqualsMethodTester testEqualsMethod method implements the following algorithm:

```
test Reflexive Equals Contract item
test Symmetric Equals Contract item
test Transitive Equals Contract item
test Consistent Equals Contract item
test Nullity Equals Contract item
test equals logic is correct for different types
for i in 1 .. n do
  test property significance in equals
end for
```

Where n is the number of test iterations to perform per class. The default is 100, but this can be altered. Test configuration is covered later in this guide.

5.2.1. Reflexive Test Algorithm

The Reflexive Equals Contract item test algorithm is as follows:

```
create instance of class under test, object x
assert x.equals(x)
```

5.2.2. Symmetric Test Algorithm

The Symmetric Equals Contract item test algorithm is as follows:

```
create instance of class under test, object x
create instance of class under test, object y
assert x.equals(y)
assert y.equals(x)
```

5.2.3. Transitive Test Algorithm

The Transitive Equals Contract item test algorithm is as follows:

```
create instance of class under test, object x
create instance of class under test, object y
create instance of class under test, object z
assert x.equals(y)
assert y.equals(z)
assert x.equals(z)
```

5.2.4. Consistent Test Algorithm

The Consistent Equals Contract item test algorithm is as follows:

```
create instance of class under test, object x
create instance of class under test, object y
for j in 1..100 do
  assert x.equals(y)
  assert result is consistent
end for
```

5.2.5. Nullity Test Algorithm

The Nullity Equals Contract item test algorithm is as follows:

```
create instance of class under test, object x
assert x.equals(null) is false
```

5.2.6. Different Types Test Algorithm

The different types test algorithm is as follows:

```
create instance of class under test, object x
create object y of a different type to x
assert x.equals(y) is false
```

5.2.7. Property Significance Test Algorithm

The property significance test algorithm is as follows:

```
for each property in public getter/setter method pairs do
  create instance of class under test, object x
  create instance of class under test, object y
  change property of y to contain a different value
  if property is insignificant then
    assert x.equals(y)
  else
    assert x.equals(y) is false
  end if
end for
```

5.3. Default Behaviour

The default behaviour of EqualsMethodTester is to test the equals method n times using different random generated test data for each test. It tests that the equals method of a specified class is correct by verifying property significance and that it complies with the Equals Contract. The default value of n is 100.

Unless specified, all properties tested by property significance verification are assumed to be significant. That is, the testEqualsMethod method assumes that all properties are considered by the equals method.

The following example uses the EqualsMethodTester default behaviour.

```
EqualsMethodTester tester = new EqualsMethodTester();
tester.testEqualsMethod(factory);
```

The testEqualsMethod method takes an EquivalentFactory that creates independent but logically equivalent instances of the class you want to test the equals method of. For example, if you wanted to test the following class, Person, that overrides the equals method:

```
class Person {

  private String firstName;
  private String lastName;
  private int age;

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getFirstName() {
    return firstName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  public String getLastName() {
    return lastName;
  }

  public int getAge() {
    return age;
  }

  public void setAge(int age) {
    this.age = age;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
```

```
      return false;
    Person other = (Person) obj;
    if (age != other.age)
      return false;
    if (firstName == null) {
      if (other.firstName != null)
        return false;
    } else if (!firstName.equals(other.firstName))
      return false;
    if (lastName == null) {
      if (other.lastName != null)
        return false;
    } else if (!lastName.equals(other.lastName))
      return false;
    return true;
  }
}
```

You would create a Person EquivalentFactory that returns independent but logically equivalent Personobjects as follows:

```
class PersonFactory implements EquivalentFactory<Person> {
  @Override
  public Person create() {
    Person person = new Person();
    person.setFirstName("TEST_FIRST_NAME");
    person.setLastName("TEST_LAST_NAME");
    person.setAge(35);
    return person;
  }
}
```

And invoke the testEqualsMethod method by passing the Person EquivalentFactory as follows:

```
EqualsMethodTester tester = new EqualsMethodTester();
tester.testEqualsMethod(new PersonFactory());
```

EqualsMethodTester infers the class whose equals method should be tested from the type of object created by the specified EquivalentFactory.

You are required to provide an EquivalentFactory that creates logically equivalent objects because both the Equals Contract verification and the property significance verification require multiple logically equivalent objects to perform their tests. Simply cloning the object within the testEqualsMethod method would require the tester to implement the clone on each object. Additionally, you cannot provide a mock object to the testEqualsMethod method because it is not possible to mock the actual object you want to test.

The EquivalentFactory you provide must initialise all fields of the object for the test to function correctly.

5.4. On-The-Fly Factories

Rather than defining an EquivalentFactory for each class you want to test, Mean Bean (as of version 2.x) can now create an EquivalentFactory on-the-fly based solely on the type of the class under test. To use this feature, invoke the EqualsMethodTester testEqualsMethod method as follows:

```
EqualsMethodTester tester = new EqualsMethodTester();
tester.testEqualsMethod(Person.class);
```

In the above example, Mean Bean will attempt to create an EquivalentFactory that returns fully populated, independent and logically equivalent instances of Person when requested, and use this Factory when testing equals method logic. If it cannot, an exception is thrown.

This is now the preferred way to use Mean Bean to test equals method logic, however it will only work when the class under test has a no-arg constructor. As such, you may need to resort to the previously defined EquivalentFactory-based testEqualsMethod approach.

5.5. Specifying Property Significance For Tests

A “significant” property is one that is considered by an equals method and consequently one that should affect the logical equivalence of two objects. Conversely, an “insignificant” property is one that is not considered by an equals method and consequently one that should not affect the logical equivalence of two objects.

The testEqualsMethod method will verify that properties you believe are “significant” do affect logical equivalence, and vice versa. Unless specified, all properties tested by property significance verification are assumed to be significant.

To specify a property as “insignificant” to the testEqualsMethod method, provide its name as a String to the “insignificantProperties” vararg parameter. You can specify as many properties as you like. Mean Bean will assume you believe that the properties you do not specify are significant.

Returning to the Person class defined above, in the following example the firstName and age properties are specified as insignificant and the lastName is significant.

```
EqualsMethodTester tester = new EqualsMethodTester();
tester.testEqualsMethod(Person.class, “firstName”, “age”);
// or...
tester.testEqualsMethod(new PersonFactory(), “firstName”,
  “age”);
```

We know that this test will fail because the equals method of the Person class considers all three properties. Executing the above code with result in an AssertionError being thrown, failing the test.

If however the equals method were defined as follows, so that age is no longer considered:

```
public boolean equals(Object obj) {
  if (this == obj)
    return true;
  if (obj == null)
    return false;
  if (getClass() != obj.getClass())
    return false;
  Person other = (Person) obj;
  if (firstName == null) {
    if (other.firstName != null)
      return false;
  } else if (!firstName.equals(other.firstName))
    return false;
  if (lastName == null) {
    if (other.lastName != null)
      return false;
  } else if (!lastName.equals(other.lastName))
    return false;
  return true;
}
```

Then the following test would pass:

```
EqualsMethodTester tester = new EqualsMethodTester();
tester.testEqualsMethod(Person.class, “age”);
// or...
tester.testEqualsMethod(new PersonFactory(), “age”);
```

Notice that properties are specified by name, not getter or setter methods – e.g. “firstName”.

Prior to version 0.0.6, Mean Bean did not validate the names of the properties provided in the insignificantProperties vararg. Rather, it simply ignored unrecognised properties. As of version 0.0.6 however Mean Bean now throws an IllegalArgumentException when a unrecognised property is provided. This should ensure your tests are kept in-line with your code as well as prevent mistakes.

5.6. Configuring The Number Of Tests

Unlike BeanTester, there is currently (Mean Bean 2.0.3) no way to set the number of times equals is tested for all classes via an instance of EqualsMethodTester. This feature will be added in the near future. For now, configure the number of tests via a Configuration object, by setting iterations on it, and pass the Configuration to all invocations of testEqualsMethod.

Configuration objects can be used to configure testing on a per-class basis. In the following example, the fewTests Configuration tells EqualsMethodTester to test equals 20 times (rather than 100). The lotsOfTests Configuration tells EqualsMethodTester to test equals 200 times. As such,Person, Employee and Manager are all tested 20 times. Animal is tested 100 times (the default).Dog and Cat are tested 200 times.

```
Configuration fewTests = new ConfigurationBuilder()
  .iterations(20)
  .build();
Configuration lotsOfTests = new ConfigurationBuilder()
  .iterations(200)
  .build();
EqualsMethodTester tester = new EqualsMethodTester();
tester.testEqualsMethod(Person.class, fewTests);
tester.testEqualsMethod(Employee.class, fewTests);
tester.testEqualsMethod(Manager.class, fewTests);
tester.testEqualsMethod(Animal.class);
tester.testEqualsMethod(Dog.class, lotsOfTests);
tester.testEqualsMethod(Cat.class, lotsOfTests);
```

Unless a Configuration object with overridden iterations is provided, other invocations of testEqualsMethod use the global iterations setting.

5.7. Ignoring A Property

You can instruct Mean Bean to ignore (do not test) a named property. To do this, create a Configuration, ignore properties on it, and pass it to the EqualsMethodTestertestEqualsMethod method.

Using the previously defined Person class, the following example tests the Person equals method's compliance with the Equals Contract as well as the significance of all properties except firstName. That is, when property significance verification is performed, only lastName and age are verified.

```
EqualsMethodTester tester = new EqualsMethodTester();
Configuration configuration = new ConfigurationBuilder()
  .ignoreProperty(“firstName”)
  .build();
tester.testEqualsMethod(Person.class, configuration, “age”);
```

Age is still insignificant, so age is specified as such in the method call. However, since firstName is now ignored, there is no need to specify it as insignificant – it won't be tested.

Unless this Configuration object is provided, other invocations of testEqualsMethod will verify all properties. In the following example Person's lastName and age properties are verified, whereas all of Animal's properties are verified.

```
EqualsMethodTester tester = new EqualsMethodTester();
Configuration configuration = new ConfigurationBuilder()
  .ignoreProperty(“firstName”)
  .build();
tester.testEqualsMethod(Person.class, configuration, “age”);
tester.testEqualsMethod(Animal.class);
```

6. Testing HashCode Methods

6.1. Introduction

If you have implemented your own hashCode method in a class (overriding the hashCode method inherited from Object) you can easily test it using Mean Bean's HashCodeMethodTester. The HashCodeMethodTester testHashCodeMethod method verifies that the hashCode method complies with the HashCode Contract.

The HashCode Contract states that:

- Logically equivalent objects should have the same hashCode.
- The hashCode of an object should remain consistent across multiple invocations, so long as the object does not change.

If the hashCode method does not comply with the HashCode Contract, the test will fail and the HashCodeMethodTester will throw an AssertionError with an appropriate message.

Currently (Mean Bean 2.0.3) the HashCodeMethodTester does not perform Property Significance Verification in a manner like the EqualsMethodTester. This feature is on our Road Map and will be added in the near future.

6.2. Algorithm

The HashCodeMethodTester testHashCodeMethod method implements the following algorithm:

```
test hashCodes equal
test hashCodes consistent
```

6.2.1. HashCode Equals Test Algorithm

The HashCodes Equal test algorithm is as follows:

```
create instance of class under test, object x
create instance of class under test, object y
assert x.equals(x)
assert x.hashCode() == y.hashCode()
```

6.2.2. HashCode Consistent Test Algorithm

The HashCodes Consistent test algorithm is as follows:

```
create instance of class under test, object x
for i in 1..100 do
    assert hashCode is consistent
end for
```

6.3. Example Test

The following shows how to use the HashCodeMethodTester:

```
HashCodeMethodTester tester = new HashCodeMethodTester();
tester.testHashCodeMethod(factory);
```

The testHashCodeMethod method takes an EquivalentFactory that creates independent but logically equivalent instances of the class whose hashCode method you want to test.

For example, if you wanted to test the following class, Person, that overrides the hashCode method:

```
class Person {

  private String firstName;
  private String lastName;
  private int age;

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getFirstName() {
    return firstName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  public String getLastName() {
    return lastName;
  }

  public int getAge() {
    return age;
  }

  public void setAge(int age) {
    this.age = age;
  }

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + age;
    result = prime * result
      + ((firstName == null) ? 0 : firstName.hashCode());
    result = prime * result
      + ((lastName == null) ? 0 : lastName.hashCode());
    return result;
  }
}
```

You would create a Person EquivalentFactory that returns independent but logically equivalent Personobjects as follows:

```
class PersonFactory implements EquivalentFactory<Person> {
  @Override
  public Person create() {
    Person person = new Person();
    person.setFirstName("TEST_FIRST_NAME");
    person.setLastName("TEST_LAST_NAME");
    person.setAge(35);
    return person;
  }
}
```

And invoke the testHashCodeMethod method by passing the Person EquivalentFactory as follows:

```
HashCodeMethodTester tester = new HashCodeMethodTester();
tester.testHashCodeMethod(new PersonFactory());
```

HashCodeMethodTester infers the class whose hashCode method should be tested from the type of object created by the specified EquivalentFactory.

You are required to provide an EquivalentFactory that creates logically equivalent objects because the HashCode Contract verification requires multiple logically equivalent objects to perform its tests. Simply cloning the object within the testHashCodeMethod method would require the tester to implement clone on each object. Additionally, you cannot provide a mock object to the testHashCodeMethodmethod because it is not possible to mock the actual object you want to test.

The EquivalentFactory you provide must initialise all fields of the object for the test to function correctly.

6.4. On-The-Fly Factories

Rather than defining an EquivalentFactory for each class you want to test, Mean Bean (as of version 2.x) can now create an EquivalentFactory on-the-fly based solely on the type of the class under test. To use this feature, invoke the HashCodeMethodTester testHashCodeMethod method as follows:

```
HashCodeMethodTester tester = new HashCodeMethodTester();
tester.testHashCodeMethod(Person.class);
```

In the above example, Mean Bean will attempt to create an EquivalentFactory that returns fully populated, independent and logically equivalent instances of Person when requested, and use this Factory when testing hashCode method logic. If it cannot, an exception is thrown.

This is now the preferred way to use Mean Bean to test hashCode method logic, however it will only work when the class under test has a no-arg constructor. As such, you may need to resort to the previously defined EquivalentFactory-based testHashCodeMethod approach.

7. Logging Configuration

Mean Bean uses Apache Commons Logging. When testing Mean Bean we configure its logging with the following log4j.properties file placed in the src/test/resources. You can see it in the test jar available for download from version 0.0.5 onwards.

```
# Set root logger level to DEBUG and its only appender to A1.
log4j.rootLogger=INFO, A1  # A1 is set to be a ConsoleAppender.
log4j.appender.A1=org.apache.log4j.ConsoleAppender  # A1 uses
PatternLayout.
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p %c %x -
%m%n
```