Java Enterprise Edition Support in Search-Based JUnit Test Generation

Andrea Arcuri[1] and Gordon Fraser[2]

[1] Westerdals Oslo ACT, Faculty of Technology, Oslo, Norway, and University of Luxembourg, Luxembourg
[2] Department of Computer Science, The University of Sheffield, UK

Abstract. Many different techniques and tools for automated unit test generation target the Java programming languages due to its popularity. However, a lot of Java's popularity is due to its usage to develop enterprise applications with frameworks such as Java Enterprise Edition (JEE) or Spring. These frameworks pose challenges to the automatic generation of JUnit tests. In particular, code units ("beans") are handled by external web containers (e.g., WildFly and GlassFish). Without considering how web containers initialize these beans, automatically generated unit tests would not represent valid scenarios and would be of little use. For example, common issues of bean initialization are dependency injection, database connection, and JNDI bean lookup. In this paper, we extend the EvoSuite search-based JUnit test generation tool to provide initial support for JEE applications. Experiments on 247 classes (the JBoss EAP tutorial examples) reveal an increase in code coverage, and demonstrate that our techniques prevent the generation of useless tests (e.g., tests where dependencies are not injected).

Keywords: Java enterprise edition, JEE, search-based testing, automated unit test generation, database

1 Introduction

As the Java programming language remains one of the most popular programming languages, it is one of the dominant languages for research on software engineering and automated unit test generation. However, there are two main versions of Java: the Standard Edition (SE), and the one tailored for enterprise needs, i.e., the so called Java Enterprise Edition (JEE) [8]. JEE extends SE in various ways, for example by providing APIs for databases, distributed and multi-tier architectures, web applications (e.g., using servlets) and services (e.g., REST and SOAP). The popularity of the Java programming language is in part due to the use of the latter version of Java. However, there are large differences between SE and JEE programs.

In a typical Java SE application, there is an entry point class that has a `main` method with an array of strings as parameters, which represent the command line arguments. This main method then typically calls methods from other classes in

the application, and new object instances are created with the `new` keyword. Once the application is started, it then interacts with its *environment*, for example using a GUI, accessing the network, file system, console, etc. Writing a unit test for a class in this context usually means to instantiate it, call its methods with some input parameters, and to mock or simulate its interactions with the environment.

In JEE, in contrast to SE, the developed applications are not standalone: they need to be run in a container, like for example WildFly[3] or GlassFish[4]. These containers scan deployed applications for XML configurations or annotations directly in the Java classes. Object instances of the applications are created via reflection, and possibly augmented/extended (e.g., using proxy classes) based on the container's configurations. A typical case is access to databases: a Java class that needs to access the application's database will not need to have code to deal directly with all the low level details of accessing databases (e.g., handling of transactions), or configure it explicitly. In fact, a reference to a handler for the database can be automatically *injected* in a class by the container, and each of its method would be automatically marked for transaction delimiters (e.g., create a new transaction when a method is called, commit it once the method is finished, or rollback if any exceptions are thrown).

All these JEE functionalities make the development of enterprise applications much easier: engineers just need to focus on the business logic, where many complex tasks like handling databases and web connections are transparently delegated to the containers. However, these features make unit testing JEE classes more complicated. Given a class `X`, one cannot simply create an instance using `new X()` in a unit test, as that way all the dependencies injected by the container would be missing. This is a challenge for automated unit test generation: There has been a lot of research on how to automatically generate unit tests for Java software, and practitioners can freely download research prototypes like for example T3 [11], JTExpert [12], Randoop [9], or EvoSuite [7]. These tools, however, all target Java SE, and not JEE software.

To illustrate the effects of this, consider the example class `JeeExample` in Figure 1a, which contains a number of JEE features. `JeeExample` is an Enterprise Java Bean, as it is annotated with `@javax.ejb.Stateless`. It has a reference to an `EntityManager`, which is used to access the application's database. This reference is expected to be injected by the container, because the field `em` is annotated with `@PersistenceContext`. The class has two methods: `persist()` to save data, and a boolean `checkForMatch()` which just does some checking on the existing state of the database. `KeyValuePair` is an auxiliary class shown in Figure 1b.

Unit test generation tools intended for Java SE cannot cover any of the branches in this class. The reason is that the field `em` is not injected, and so all calls on it result in a null pointer exception. For example, Figure 2 shows a test generated by EvoSuite and Figure 3 shows one generated by Randoop. Without

[3]http://wildfly.org, accessed April 2016
[4]https://glassfish.java.net, accessed April 2016

```java
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateless
public class JeeExample {

  @PersistenceContext
  private EntityManager em;

  public void persist(String key, String value) {
    KeyValuePair pair = new KeyValuePair(key, value);
    em.persist(pair);
  }

  public boolean checkForMatch(String key,
                               String value) {

    KeyValuePair pair = em.find(KeyValuePair.class,
                                key);

    if(pair == null)
      return false;

    if(pair.getValue().equals(value))
      return true;
    else
      return false;
  }
}
```

(a) Class under test.

```java
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class KeyValuePair {
  @Id
  private String key;
  private String value;

  public KeyValuePair(){}

  public KeyValuePair(String key,
                      String value) {
    this.key = key;
    this.value = value;
  }

  public String getKey() { return key; }

  public void setKey(String key) {
    this.key = key;
  }

  public String getValue() { return value; }

  public void setValue(String value) {
    this.value = value;
  }
}
```

(b) Dependency entity class.

Fig. 1: Code example showing a stateless enterprise Java bean accessing a database.

```java
@Test(timeout = 4000)
public void test0() throws Throwable {
  JeeExample jeeExample0 = new JeeExample();
  try {
    jeeExample0.checkForMatch("z", "]#");
    fail("Expecting exception: NullPointerException");
    } catch(NullPointerException e) {
    verifyException("JeeExample", e);
    }
}
```

Fig. 2: Example test generated by the standard version of EvoSuite on the example class from Figure 1a.

handling dependency injection and database initialization, all tests result in null pointer exceptions. These tests are not particularly useful, as they test the class under test (CUT) only when it is not in an initialized, meaningful state.

In this paper, we describe and evaluate an approach to include JEE features in the search space of the search-based test data generation tool EvoSuite [7]. In particular, in this paper we provide the following contributions:

– Handling of dependency injection, which requires special care on how the tests are mutated and evolved. By construction, appropriate handling of dependency injection avoids that useless tests, like the one in Figure 2, are generated.

```
@Test
public void test1() throws Throwable {
  if (debug) { System.out.format("%n%s%n","ErrorTest0.test1"); }

  JeeExample jeeExample0 = new JeeExample();
  // during test generation this statement threw an exception of
  // type java.lang.NullPointerException in error
  jeeExample0.persist("hi!", "");
}
```

Fig. 3: Example test generated by Randoop on the example class from Figure 1a.

- Automated initialization of in memory, embedded databases.
- Handling of some JEE functionalities through environment mocks [3,4], like for example bean lookup resolution.
- An empirical study on 247 JEE classes, which shows that code coverage increases.

Using the JEE extension presented in this paper, EvoSuite generates the tests shown in Figure 4 when applied on the class `JeeExample` from Figure 1a (note, there are further initializations done in `@Before` and `@After` methods, but those are not shown due to space limitations). Seven tests are generated, which achieve full code coverage. Furthermore, those tests even point to bugs in the class `JeeExample`, for example by throwing exceptions like `PersistenceException`, `IllegalArgumentException`, `NullPointerException` and `EntityExistsException`. In particular, `test0` leads to a `PersistenceException` because it tries to persist to the database an entity with null id. `test1` leads to an `IllegalArgumentException` because the method `EntityManager#find` cannot be called with a null key. `test5` shows a null pointer exception due to the statement `if(pair.getValue().equals(...))` in the method `checkForMatch`, where `getValue()` returns null. Finally, `test6` tries to insert a new entity that already exists (same id) into the database, leading to a `EntityExistsException`. Note that no test was generated in which the field *em* was not injected (i.e., left null).

2 Background

2.1 Java Enterprise Edition

JEE aims at fulfilling enterprise needs by making it easier to develop distributed, multi-tier applications, such as web applications and web services. In this section, we briefly describe the main features of Java Enterprise Edition (JEE), in particular version 7. As this is a very large topic, here we only provide a high level overview to make the rest of the paper more accessible for readers not familiar with JEE. For further JEE details and links, we refer to [8].

```
@Test(timeout = 4000) public void test0() throws Throwable {
    JeeExample jeeExample0 = new JeeExample();
    Injector.injectEntityManager(jeeExample0, (Class<?>) JeeExample.class);
    Injector.validateBean(jeeExample0, (Class<?>) JeeExample.class);
    try {
      jeeExample0.persist((String) null, (String) null);
      fail("Expecting exception: PersistenceException");
    } catch(PersistenceException e) {}
}
@Test(timeout = 4000) public void test1() throws Throwable {
    JeeExample jeeExample0 = new JeeExample();
    Injector.injectEntityManager(jeeExample0, (Class<?>) JeeExample.class);
    Injector.validateBean(jeeExample0, (Class<?>) JeeExample.class);
    try {
      jeeExample0.checkForMatch((String) null, (String) null);
      fail("Expecting exception: IllegalArgumentException");
    } catch(IllegalArgumentException e) {}
}
@Test(timeout = 4000) public void test2() throws Throwable {
    JeeExample jeeExample0 = new JeeExample();
    Injector.injectEntityManager(jeeExample0, (Class<?>) JeeExample.class);
    Injector.validateBean(jeeExample0, (Class<?>) JeeExample.class);
    jeeExample0.persist("#", "#");
    boolean boolean0 = jeeExample0.checkForMatch("#", "#");
    assertTrue(boolean0);
}
@Test(timeout = 4000) public void test3() throws Throwable {
    JeeExample jeeExample0 = new JeeExample();
    Injector.injectEntityManager(jeeExample0, (Class<?>) JeeExample.class);
    Injector.validateBean(jeeExample0, (Class<?>) JeeExample.class);
    jeeExample0.persist("#", "#");
    boolean boolean0 = jeeExample0.checkForMatch("#", "\"");
    assertFalse(boolean0);
}
@Test(timeout = 4000) public void test4() throws Throwable {
    JeeExample jeeExample0 = new JeeExample();
    Injector.injectEntityManager(jeeExample0, (Class<?>) JeeExample.class);
    Injector.validateBean(jeeExample0, (Class<?>) JeeExample.class);
    boolean boolean0 = jeeExample0.checkForMatch("\"", "#");
    assertFalse(boolean0);
}
@Test(timeout = 4000) public void test5() throws Throwable {
    JeeExample jeeExample0 = new JeeExample();
    Injector.injectEntityManager(jeeExample0, (Class<?>) JeeExample.class);
    Injector.validateBean(jeeExample0, (Class<?>) JeeExample.class);
    jeeExample0.persist("", (String) null);
    try {
      jeeExample0.checkForMatch("", "");
      fail("Expecting exception: NullPointerException");
    } catch(NullPointerException e) {}
}
@Test(timeout = 4000) public void test6() throws Throwable {
    JeeExample jeeExample0 = new JeeExample();
    Injector.injectEntityManager(jeeExample0, (Class<?>) JeeExample.class);
    Injector.validateBean(jeeExample0, (Class<?>) JeeExample.class);
    jeeExample0.persist("", "");
    try {
      jeeExample0.persist("", "ZuWxZ_Ohnf[");
      fail("Expecting exception: EntityExistsException");
    } catch(EntityExistsException e) {}
}
```

Fig. 4: Example test suite generated by EvoSuite on the example class from Figure 1a, when using the JEE improvements presented in this paper.

JEE functionalities. JEE can be seen as a series of packages providing different functionalities, from database access to web communication handling. Among the main functionalities of JEE, some important examples are the following:

- Java Persistence API (JPA): This is used to automatically map Java classes to tables in databases, and to read/write those objects. To achieve this, these classes need to be annotated with the `@Entity` annotation (see the example in Figure 1b). Read/write operations are done through an `EntityManager` provided by the container.
- Enterprise Java Bean (EJB): These are objects responsible for the business logic of the application. *Beans* are instantiated and managed by the container. A Java object is made into an EJB by using annotations like `@Stateless`, `@Stateful` and `@Singleton` (see example in Figure 1a).
- Java Transaction API (JTA): This is used to handle the transactions with the databases. By default, each call to an EJB method will be in a transaction, which will be rolled back if any exceptions are thrown in the EJB code.
- Java Naming and Directory Interface (JNDI): This is used to find objects that were bound by name in the current application or remote servers.
- JavaServer Faces (JSF): This is used to create component-based user interfaces for web applications. A web page would be developed in the `xhtml` format, mixing together Html/CSS/JS elements with calls to the backend Java beans.
- Java Message Service (JMS): This is used to make asynchronous point-to-point and publish-subscribe communications between distributed components.
- Web Services: These are used to develop and query web services like REST and SOAP.

Convention over configuration. To simplify development, JEE follows the *convention over configuration* approach: A typical JEE application is not a standalone process packaged as a `jar` (Java Archive) file, but rather as a `war` (Web Application Archive) file that has to be deployed on a server container (e.g., WildFly or GlassFish). When such a `war` file is deployed on a server, the server will do a series of operations and initialization based on the `war`'s content. The developers do not need to configure them, unless they want to do something different from the standard convention.

For example, an entity Java class will be mapped to a database table with the same name as the Java class. The developers just need to use the annotation `@Entity`, and the server will take care of rest. However, if, for example, a given entity class needs to be mapped to a table with a different name (e.g, when using JPA on a legacy database that cannot be changed), further annotations/settings can be added to change that default naming convention. Similarly, all methods in an EJB are automatically marked for required transactions: the container will create a new transaction if the method is called from a non-transactional client. If this default behavior is not the desired one, JTA annotations (e.g., `@TransactionAttribute`) can be added to the EJB methods to achieve a different behavior.

On one hand, the use of convention over configuration makes development easier and quicker, as the engineers need to specify only the non-conventional cases. On the other hand, debugging and code understanding might become more difficult, as the container might do a lot of hidden operations behind the scenes that are not obvious for a non-expert in JEE.

Dependency Injection. One of the main characteristics that distinguish JEE from SE is *Dependency Injection*. If an object X needs to use Y, instead of instantiating Y directly (or calling an external method to get an instance of it), it will delegate the container to provide an instance of Y. This is particularly useful to decouple components, as an enterprise application could be deployed on different containers (e.g., WildFly and GlassFish) that have different implementations for resources like database management. Furthermore, dependency injection entails different wiring of the application based on different contexts without the need of recompilation. For example, in a testing scenario a container could rather inject a mocked bean instead of a production one. In JEE, there are different ways to do dependency injection. A typical case is to use annotations on private fields. See for example the `em` field in Figure 1a, which is automatically injected by the container because it is annotated with `@PersistenceContext`.

2.2 JUnit Test Generation with EvoSuite

The EvoSuite tool [7] automatically generates JUnit test suites optimized to achieve high code coverage. Test generation uses a search-based approach, where a genetic algorithm evolves a population of candidate solutions (test suites), guided by a fitness function that captures the target coverage criteria. A test suite in EvoSuite is a variable size set of test cases, and a test case, in turn, is a sequence of statements that instantiate or manipulate objects. The initial population consists of randomly generated tests, and then search operators are applied throughout the search. First, the fitness value for each candidate test suite is calculated. Then, individuals are selected for reproduction based on their fitness value; fitter individuals are more likely to be selected. With a certain probability, crossover is applied to the selected individuals, and then, with a certain probability, mutation is applied. Mutation consists of adding new (random) test cases to a test suite, and modifying existing tests (e.g., by adding, removing, or changing some of the statements). The search operators are applied until a new generation of individuals has been produced, and this then becomes the next generation. At the end of the search (e.g., when the allocated time has been used up), the resulting test suite goes through several post-processing steps such as minimization (i.e., removal of redundant statements) or assertion generation (i.e., addition of JUnit assert statements to check the observed behavior).

The search algorithm and the post-processing steps are both applicable regardless of whether the underlying Java class under test is Java SE or JEE code. Nevertheless, EvoSuite up to now was not able to generate tests for JEE specific code; the main reason for this lies in restrictions to EvoSuite's search space that result from the design of the mutation operators for the test cases. In

particular, consider the insertion of statements (which is also used to generate random test cases): EvoSuite either inserts a randomly selected method call of the class under test, or inserts a random method call to a randomly chosen object generated in the current sequence of calls. If the method takes parameters, these are either satisfied with existing objects in the test, or EvoSuite will recursively insert statements that generate the required objects.

Consider class `JeeExample` from Figure 1a: The candidate methods of the class under test are `persist`, `checkForMatch`, and the implicitly defined default constructor. All parameters are of type `String`, and EvoSuite will generate random or seeded strings. Although EvoSuite can also read from and write to public fields, the standard operators will not access the private field `em`, and thus EvoSuite has no means of initializing the `EntityManager`. Note that, if `JeeExample` would do dependency injection by providing a constructor with an `EntityManager`, then EvoSuite would attempt to explicitly instantiate one. However, this does not guarantee that EvoSuite would be able to create and configure a valid instance.

3 JEE Support in EvoSuite

In this section, we describe an approach to enable search-based tools like EvoSuite to generate unit tests for JEE software. We do not cover the whole JEE specification, as that is simply too large to cover in a single study, but rather focus on some of the most common features, in particular dependency injection, JPA and JNDI.

Support for these features is added via two techniques: First, the search space of call sequences is modified to include relevant JEE calls, for example to handle injection. The challenge lies in constraining these calls to result in valid JEE scenarios. Second, the code under test is instrumented to ensure that the JEE environment set up by EvoSuite is used, for example by directing all database accesses to an in-memory database automatically initialized by EvoSuite.

3.1 Dependency Injection

If dependency injection is not handled, then the tests generated by automated tools will just throw useless null pointer exceptions (recall Figure 2 and Figure 3).

One possibility would be to use an embedded container running on the same JVM of the tests, and to delegate all the dependency injections to it. However, such an embedded container would still need to be configured, e.g., to specify which beans should be used for injection when there is more than one alternative, and it would be difficult to customize for unit testing purposes (e.g., replace with mocks some beans using external resources). A simpler alternative, which we implemented, is to do the injection directly in the unit tests using support methods we developed. For example, in every test in Figure 4 the instantiation of the class `JeeExample` is followed by calls to `Injector#injectEntityManager`, which is a helper method that sets the entity manager.

Every time a new object is created as part of the search, EvoSuite checks if it, or any of its superclasses, has fields that should be injected. To check for injection, we look at JEE annotations like `@Inject`, `@PersistenceContext`, `@PersistenceUnit`, `@Resource`, `@EJB`, `@WebServiceRef`, `@ManagedProperty`, and `@Context`. We also look at annotations used in popular frameworks like `@Autowired` in Spring[5]. For each of these types of injections, the helper class `Injector` provides helper methods. For each injectable field, the corresponding helper methods are inserted in the test.

We distinguish between two kinds of injected objects: *pre-defined* and *new*. For some types of objects, EvoSuite defines specific, pre-defined instances that can be injected inject. These are, for example, customized entity managers, as done with `Injector#injectEntityManager`. The use of these pre-defined instances allows EvoSuite to more easily explore complex scenarios, where the random object initialization is unlikely to lead to interesting scenarios.

If for an injectable field f, of type F in a class X, we have no pre-defined instance in EvoSuite, we add a call to a more generic `Injector#inject`. This method takes not only X as input, but also an object i of type F, i.e., this new object i will be injected in the field f of object X. The new object i will be created as part of the search, just like standard method parameters, and will evolved like any other objects (e.g., mutation operators might add new calls on it). Note that this new object i might itself need injection for some of its fields, and this will be handled recursively.

After a class X has all of its injectable fields injected, EvoSuite checks for methods marked with `@PostConstruct` in the class hierarchy of X, and also add calls to `Injector#executePostConstruct`.

Adding `Injector` methods in the tests has major consequences their evolution during the search. Recall from Section 2.2 that EvoSuite performs several kinds of operations to evolve tests, such as deleting statements, changing function calls, creating new statements at any position in the test, etc. These can change the tests to an inconsistent state, e.g., EvoSuite could delete all calls to `Injector` methods. To avoid this issue, we defined *constraints* on the test statements, and modified all EvoSuite search operators to enforce these constraints. Given a class X with injectable fields, these constraints are for example:

- No call to `Injector` can be deleted until X is in the test.
- If X is deleted, then delete all its calls to `Injector`.
- Fields cannot be injected more than once.
- Calls to `Injector` should be automatically added when a new object is instantiated. Search operators should not add new unnecessary calls to `Injector`, or modify existing ones.
- Calls to `Injector` methods cannot take null as input. This might prevent the deletion of objects that are used as input.
- Between the statement where X is instantiated and its last call on `Injector`, X cannot be used as input in any method that is not an injector, and no call can be added on X (as X is not fully initialized yet).

[5]https://spring.io, accessed April 2016

One drawback of injecting fields directly in the unit tests is test maintenance. Assume tests are generated for a class X with some injected fields. Assume also that, in a future release of X, a new injected field is added, although the external behavior (i.e., its semantics) of X has not been changed. Now, it might well be that the previously generated tests for X will now fail due to null pointer exceptions on this new field, although no regression bugs were introduced. To avoid this kind of *false positives*, after each bean initialization we add a call to `Injector#validateBean` (recall Figure 4). This method checks if all injectable fields have indeed been injected. If not, an `AssumptionViolatedException` is thrown, which prevents the tests from failing (JUnit will consider a test throwing this exception as ignored, as if it was marked with `@Ignore`).

3.2 Database Handling

JEE applications tend to depend on databases. To test applications that access a database, a database needs to be configured and running. As this is typically not within the scope of capabilities of a unit test generation tool, this configuration would typically need to be done manually. To avoid this issue, we extended EvoSuite to be able to perform the initialization automatically. In particular, we use the Spring framework to scan the classpath for `@Entity` classes, and automatically start an embedded database (HyperSQL[6]) for those entities, using Hibernate[7] as JPA implementation.

When beans need entity managers, we inject custom ones that are configured for this embedded database. Furthermore, we mock `javax.persistence.Persistence`, which consists of only static methods to access entity managers, to return our custom entity manager.

The embedded database is cleaned up after each test execution, in order to avoid dependencies among tests. Starting/resetting the database is done in the `@Before` and `@After` methods in the tests. However, initializing a database is time consuming, and may potentially take several seconds to complete. Therefore, it is not initialized by default, but only if the CUT really uses the database.

3.3 JNDI Mocking

When unit testing a class, the CUT might use JNDI to access objects that have been initialized in other classes since the application was started, or remote ones outside the running JVM. This is a problem for unit testing, as JNDI lookups might fail. To avoid this issue, we mock JNDI, similarly to how EvoSuite already mocks environment interactions with the file system [3] and the network [4]. In particular, we provide a mock version of the class `javax.naming.InitialContext`. During EvoSuite's bytecode instrumentation phase, all calls to the original class are automatically replaced with calls to the mocked version. Furthermore, EvoSuite maintains information about the

[6]http://hsqldb.org, accessed April 2016.
[7]http://hibernate.org, accessed April 2016

known classes, their methods, and how to generate them (referred to as *test cluster*). This information is derived statically during initialization, and all references to the original class are replaced with references to the mock class.

By default, the mock class for `InitialContext` will fail to resolve any object lookups, i.e., it will return null. However, it also keeps track of all objects that have been requested during the search. If any objects were requested, then EvoSuite's test cluster is extended with additional methods to instantiate these objects and to make them accessible through JNDI. The mocked JNDI resolution is re-initialized at each new test execution, in order to avoid dependencies among tests.

4 Empirical Study

The techniques presented in this paper enable tools like EvoSuite to be applied on JEE software. By construction, tests with non-initialized beans (which are not useful; recall Figure 2 and Figure 3) are no longer generated. However, in order to understand the effects of this change, it is also important to see what is the impact on code coverage. In particular, in this paper we address the following research question:

RQ: What is the effect of the JEE extensions on branch coverage?

Note that looking at fault detection (e.g., the throwing of undeclared exceptions) is not trivial to do automatically, as the lack of dependency injection might lead to many failing tests that are just false positives (recall Figure 2 and Figure 3), because they would throw exceptions when non-injected fields are accessed. However, even when injection is handled, the CUT could lead to null pointer exceptions that show actual bugs (e.g., recall `test5` in Figure 4).

4.1 Experimental Setup

Open-source repositories like GitHub[8] and SourceForge[9] host a large amount of Java SE software, like libraries and applications. However, as JEE is targeted at enterprises, the amount of JEE software on open-source repositories is obviously lower. Furthermore, a JEE project might be simply marked as "Java", and so a systematic search for JEE projects is not necessarily trivial.

As JEE specifications are very large, and we are only interested in classes that use JEE features, we chose the set of JEE examples used to demonstrate JBoss EAP / WildFly application servers as case study. These consist of a total of 247 Java classes, hosted on GitHub[10].

On each of these 247 classes, we ran EvoSuite with and without our JEE extension, 30 times per CUT, for a total of $247 \times 2 \times 30 = 14{,}820$ runs. For

[8]https://github.com, accessed April 2016.
[9]https://sourceforge.net, accessed April 2016.
[10]https://github.com/jboss-developer/jboss-eap-quickstarts, accessed April 2016

the experiments, we used the default configuration of EvoSuite, which is assumed to show good results on average [2]. In each experiment, the search phase for EvoSuite was executed until either 100% branch coverage was achieved, or a timeout of two minutes was reached. For each run we collected data on the achieved branch coverage as reported by EvoSuite. Results were analyzed based on standard guidelines [1]. In particular, to assess statistical difference we used the non-parametric Mann-Whitney-Wilcoxon U-test, whereas we used the Vargha-Delaney $\hat{A}_{12}$ as effect size.

4.2 Results

Without JEE support, the default version of EvoSuite achieves an average of 77% branch coverage on these 247 classes. When using the techniques presented in this paper, branch coverage increases to 80%.

This modest +3% increase warrants closer inspection: Only 102 out of the 247 classes have some kind of JEE annotation for dependency injection. In contrast, many classes are trivial (e.g., skeletons with empty bodies representing some business logic), and might only be needed to compile other classes in which the JEE features are really used. In particular, `@Entity` classes (e.g, recall Figure 1b) usually have just basic setters and getters, and pose no challenge for unit test generation. This explains the already high coverage of 77% that EvoSuite can achieve even without any JEE support.

If we assume that a class, on which EvoSuite can achieve 90% or more branch coverage even without JEE support, does not depend on JEE, then the average coverage on the remaining 88 classes increases from 37.7% to 46.0%, i.e., a 8.3% improvement.

To get a better picture of the importance of handling JEE features, Table 1 shows detailed data on the 25 challenging classes where JEE handling had most effect: On these classes, the average branch coverage nearly doubles from 43.8% to 74.6%. All comparisons are statistically valid (p-values very close to zero), and the average effect size for $\hat{A}_{12}$ is nearly maximal, i.e., 0.98.

RQ: *JEE support significantly increases branch coverage (average +3%), with substantial increases in JEE relevant classes.*

5 Threats to Validity

Threats to *internal* validity result from how the experiments were carried out. The techniques presented in this paper have all been implemented as part of the EvoSuite tool. Although EvoSuite is a mature tool used by practitioners, no system is guaranteed to be error free. Furthermore, because EvoSuite is based on randomized algorithms, each experiment has been repeated several times, and the results have been evaluated with rigorous statistical methods.

To avoid disseminating flawed results, repeatability and reproducibility are cornerstones of the scientific process [5]. To address this issue, we released the

Table 1: Branch coverage comparison of EvoSuite with (JEE) and without (Base) support for JEE, on the 25 classes with the largest increase. Note, some classes have the same name, but they are from different packages.

Class	Base	JEE	$\hat{A}_{12}$	p-value
ManagedComponent	14.3%	41.2%	**0.96**	≤ 0.001
UnManagedComponent	47.0%	51.6%	**0.80**	≤ 0.001
ItemBean	89.7%	100.0%	**1.00**	≤ 0.001
HATimerService	57.1%	93.3%	**1.00**	≤ 0.001
SchedulerBean	60.0%	97.3%	**0.98**	≤ 0.001
IntermediateEJB	33.3%	66.7%	**1.00**	≤ 0.001
SecuredEJB	80.0%	98.7%	**0.97**	≤ 0.001
AsynchronousClient	20.0%	29.3%	**0.97**	≤ 0.001
RemoteEJBClient	25.0%	58.3%	**1.00**	≤ 0.001
TimeoutExample	60.0%	99.3%	**1.00**	≤ 0.001
GreetController	66.7%	100.0%	**1.00**	≤ 0.001
HelloWorldJMSClient	4.9%	23.1%	**1.00**	≤ 0.001
MemberResourceRESTService	19.1%	69.2%	**1.00**	≤ 0.001
MemberResourceRESTService	19.3%	67.7%	**0.96**	≤ 0.001
MemberRegistrationServlet	18.2%	87.1%	**1.00**	≤ 0.001
HelloWorldMDBServletClient	26.0%	61.3%	**0.99**	≤ 0.001
TaskDaoImpl	55.6%	77.8%	**1.00**	≤ 0.001
AuthController	30.0%	100.0%	**1.00**	≤ 0.001
TaskController	42.9%	100.0%	**1.00**	≤ 0.001
TaskDaoImpl	55.6%	75.0%	**0.96**	≤ 0.001
TaskListBean	75.0%	96.7%	**0.93**	≤ 0.001
TaskDaoImpl	55.6%	77.8%	**1.00**	≤ 0.001
TaskResource	55.4%	84.3%	**1.00**	≤ 0.001
Servlet	40.0%	60.9%	**0.92**	≤ 0.001
XAService	44.7%	49.3%	**0.96**	≤ 0.001
Average	43.8%	74.6%	0.98	

implementation of all the techniques presented in this paper as open-source (LGPL license), and we made it available on a public repository[11].

Threats to *construct* validity come from what measure we chose to evaluate the success of our techniques. We used branch coverage, which is a common coverage criterion in the software testing literature. However, it is hard to automatically quantify the negative effects of tests that do not handle dependency injection, as the presence of false positive tests on software maintenance is a little investigated topic in the literature.

Threats to *external* validity come from how well the results generalize to other case studies. To have a variegated set of classes showing different features of JEE, we chose the JEE examples used to demonstrate the JBoss EAP / WildFly application servers, which consist of 247 Java classes. Larger case studies on industrial systems will be needed to further generalize our results.

[11] www.github.com/EvoSuite/evosuite

6 Related Work

While there are numerous tools and techniques to generate unit tests for Java classes and programs, we are not aware of any work targeting unit tests for JEE classes directly.

Some of the problems caused by JEE are related to its use of databases. Emmi et al. [6] use dynamic symbolic execution to collect constraints on database queries and populate a database with data to satisfy these queries. The MODA framework [13] instruments database-driven programs to interact with a mock database instead of a real database. A test generator based on dynamic symbolic execution is then applied to insert entries into the database. A refined version of this approach [10] correlates various constraints within a database application. The code coverage increase reported by these approaches is comparable to the increases we observed in our experiments in this paper.

Besides database applications, other external dependencies such as filesystem [3], networking [4], or cloud services [14] have been integrated into test generation, typically by making test generators configure mock objects. For some of the JEE features, the approach presented in this paper also follows this strategy.

7 Conclusions

Jave Enterprise Edition (JEE) applications pose challenges that have not previously been handled by Java unit test generation tools. In order to address this problem, we have extended the EvoSuite unit test generation tool in order to support the core JEE features of (1) dependency injection, (2) database access, and (3) JNDI object lookups. This posed several technical challenges in order to ensure that several constraints on the validity of tests are maintained at all time during the search-based test generation. These techniques are fully automated, and require no human intervention (not even to initialize/run the databases). We are aware of no other tool that handles JEE specific functionalities.

An empirical study on 247 Java classes shows that, with high statistical confidence, our techniques improve branch coverage (+3% on average), especially on challenging classes heavily dependent on JEE functionalities (increase from 43.8% to 74.6%). Importantly, this approach prevents, by construction, the generation of misleading tests that throw null pointer exceptions just because dependency injections are not handled.

JEE has a very large set of specifications, and what has been addressed in this paper is just a first step. Future work will focus on handling other JEE components, like for example JMS and REST/SOAP web services. Furthermore, there is large space for improving the handling of databases, like for example extending the search to directly create objects in the database based on the class under test's queries.

All techniques discussed in this paper have been implemented as part of the EvoSuite test data generation tool. EvoSuite is open-source (LGPL license) and freely available to download. To learn more about EvoSuite, please visit our website at: `http://www.evosuite.org`.

Acknowledgments. This work is supported by the EPSRC project (EP/N023978/1) and by the National Research Fund, Luxembourg (FNR/P10/03).

References

1. Arcuri, A., Briand, L.: A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Software Testing, Verification and Reliability 24(3), 219–250 (2014)
2. Arcuri, A., Fraser, G.: Parameter tuning or default values? an empirical investigation in search-based software engineering. Empirical Software Engineering 18(3), 594–623 (2013)
3. Arcuri, A., Fraser, G., Galeotti, J.P.: Automated unit test generation for classes with environment dependencies. In: IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 79–90 (2014)
4. Arcuri, A., Fraser, G., Galeotti, J.P.: Generating TCP/UDP network data for automated unit test generation. In: ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE). pp. 155–165. ACM (2015)
5. Collberg, C., Proebsting, T.A.: Repeatability in computer systems research. Communications of the ACM 59(3), 62–69 (2016)
6. Emmi, M., Majumdar, R., Sen, K.: Dynamic test input generation for database applications. In: Proceedings of the 2007 international symposium on Software testing and analysis. pp. 151–162. ACM (2007)
7. Fraser, G., Arcuri, A.: Evosuite: automatic test suite generation for object-oriented software. In: ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE). pp. 416–419. ACM (2011)
8. Goncalves, A.: Beginning Java EE 7. Apress (2013)
9. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: ACM/IEEE Int. Conference on Software Engineering (ICSE). pp. 75–84 (2007)
10. Pan, K., Wu, X., Xie, T.: Guided test generation for database applications via synthesized database interactions. ACM Transactions on Software Engineering and Methodology (TOSEM) 23(2), 12 (2014)
11. Prasetya, I.S.W.B.: T3i: A Tool for Generating and Querying Test Suites for Java. In: ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE) (2015)
12. Sakti, A., Pesant, G., Gueheneuc, Y.G.: Instance generator and problem representation to improve object oriented code coverage. IEEE Transactions on Software Engineering (TSE) (2015)
13. Taneja, K., Zhang, Y., Xie, T.: Moda: Automated test generation for database applications via mock objects. In: IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 289–292. ACM (2010)
14. Zhang, L., Ma, X., Lu, J., Xie, T., Tillmann, N., De Halleux, P.: Environmental modeling for automated cloud application testing. IEEE Software 29(2), 30–35 (2012)