Communication Papers of the Federated Conference on Computer Science and Information Systems pp. 233–240
DOI: 10.15439/2018F214
ISSN 2300-5963 ACSIS, Vol. 17

Automated generator for complex and realistic test data—a case study

Richard Lipka
NTIS - New Technologies for Information Society
Faculty of Applied Sciences, University of West Bohemia
Plzen, Czech republic
Email: lipka@kiv.zcu.cz

Tomas Potuzak
Department of Computer Science and Engineering
Faculty of Applied Sciences, University of West Bohemia
Plzen, Czech republic
Email: tpotuzak@kiv.zcu.cz

***Abstract*—Some type of tests, especially stress tests and functional tests, require a large amount of realistic test data. In this paper, we propose a tool JOP (Java Object Populator) that uses a pseudorandom number generator in order to create test sets of complex Java objects, that can be automatically generated and directly used. Along with that, we also show usage of this tool in case study focused on performance evaluation of a real cashier system.**

The tool is designed to be able to set simple attributes of any Java object and in many cases also to create complex structures when objects are connected via references. Random values are created using the rules that are added to the class definition in form of annotation to each attribute. Using this tool simplifies creating of tests, as the tester does not need a detailed knowledge of data structures. The specification of expected values is delegated to the designer of the data model and becomes the part of the model. Furthermore, as the data objects are created at runtime, using reflection, the tests do not have to be changed when data carrying objects are modified.

I. Introduction

IN ORDER to ensure the reliability of each application, testing is a vital part of the software development process. As applications are becoming more complex — especially in terms of using many different existing libraries — and there is a pressure for fast development, one of the most important issues is fast automation of each part of software development process. Many kinds of testing, such as unit tests, are performed automatically, without the need for a human tester. However, the creation of the tests is still mostly a manual process, where a lot of code has to be created.

One of the issues of the testing is obtaining the test data. In some types of tests, for example, when the user interface is being tested, only simple values like numbers, dates, or strings are used. However, when unit tests are focused on the core of the application, it is often necessary to work with the creation of non-trivial testing objects, that are composed not only of simple attributes mentioned before but also contains references to other objects and creates arbitrary complex structures. This is even more significant in stress testing and benchmarking, when a large number of the test objects is required, to avoid biases caused by caching of too similar data. In such cases, testers have to manually prepare all instances, before the tests can be performed.

Furthermore, creating a large set of data is a tiring and repetitive task, so testers help themselves by using random generators that are provided by most programming languages. These generators can be easily used for creating test data, but usually, are designed only to generate numbers. In order to use them, test programmers are bound to have a detailed knowledge of data structure to choose appropriate parameters of the generators. Some attributes can be dependent on the others (for example weight and size of the object). Consequently, the tester has to understand these dependencies. Furthermore, the test created this way contains a lot of code strongly dependent on the structure of data carrying objects. When the implementation of these objects is changed, parts of the tests that set up data have to be revised and changed accordingly. This is reducing benefits of automated testing and forces the testers to return to the tests with each new version.

Another problem is that the characteristics of the data for the test are usually not written anywhere in an explicit form, only as the parameters of the random number generators. This makes updates of the tests more dificult, as the tester has all the time understand both strucutre of the domain objects and character of the test data.

In order to address described issues, in this paper, we propose a tool that enables to generate test sets of complex Java objects using pseudorandom numbers generators and annotations in source code of the applications under tests describing the possible values of the attributes.

II. Issues of Test Data Generating

A. Simple motivational example

Consider an application dealing with receipts. Each receipt is represented by an instance of class `Receipt`, with several attributes, such as `date`, `total price`, `salesman` and also a list of items that are on a receipt — each represented again by an instance of class (in this case the `item`). When we want to perform a stress test that adds, removes, or look up for receipts in the database, we need to generate several hundreds or thousands instances of these classes. In the same time, the application might perform other tasks, such as sending data over the network or calculating aggregated values from the receipts. In order to investigate whether the application

behaves correctly, the provided instances should be as realistic as possible.

The classical approach would be to create a generator of receipts and items that will ensure that all attributes of each receipt are set up properly. In order to do so, the creator of the test has to know all attributes that should be set, and also to have a knowledge of their characteristics. Often, the attributes are not a primitive data types but references on other objects, and if the generator should serve its purpose, it should handle this as well. Such generator can be used as long as the class `Receipt` is not changed. In the typical application, there can be dozens of domain classes like the `Receipt`, that serves mainly to carry data and that need to be generated during testing. Thus creating a generator to all of them usually is a time-consuming work.

We would like to have a tool that will be able to create instances of the `Receipt` class in one invocation, according to the characteristics provided in a human readable and understandable form. When random number generators are used in the code of the test, it is not obvious what the meaning of their parameters is without analysing the methods of the generator. It would be useful to have a declarative way of specifying these parameters in one place.

B. Test Data Sources

Data used for testing can be obtained from several sources, and their selection depends strongly on the purpose of testing. For example in unit tests, the most common way is to choose data in the way that the extreme values or decision points of the methods under testing are explored. On the other hand, for the purpose of integration tests or stress tests, it is important to use as realistic data as possible, in order to mimic the real usage of the tested software. We would like to have a tool that will allow generating a large number of instances of domain objects, with as little work of the tester as possible.

The realistic testing data can be obtained in three basic ways. They can be prepared in advance manually, they can be obtained from the production application, or they can be randomly generated.

Manual preparation of the test data is often necessary, but it is a tedious and error-prone work, so testers are usually looking for some way of automation. One option is to automatically capture the data from existing application and reuse them during the testing. This is relatively simple when the application under test and the application used for obtaining the data are the same, otherwise, a conversion is necessary. But either way, if the data are prepared manually or captured from the application, they are a static set that cannot be easily adapted to the changes in the application under test and that cannot be scaled according to the need of testers.

The randomly generated data have a great advantage in scaling, as when the generators are ready, it is easy to create an arbitrary amount of data. They can be also easily parametrized. Furthermore, they might be configured to create different data sets in each run or can create exactly the same sequence of data without the need to store a lot of testing objects.

III. Tool Design

Our main goal is to create a tool that will allow generating testing data from the definition of classes, enriched by the specific annotations. This way, the structure of data is easily visible from the application source text and is not hidden in the separate source code of tests. The same definitions are used in all tests in order to help the tester to avoid errors caused by code repetition and corrections only in some copies. It can also help to avoid the need to rewrite tests when domain classes are changed. As long as test requires valid instances of domain classes, the tool can provide them.

From the testing point of view, the new annotations serve as a detailed specification of the data type of each generated attribute. For example, instead of working only with information that an attribute is a `double` number, the annotations can specify that the value has for example a normal distribution with specified mean and standard deviation, and when necessary also with a minimal and maximal value that crops the highest and lowest values (in cases like human height). Furthermore, it might be specified that the value is a function of another attribute. The annotations can determine the desired structure of strings or, in more general cases, the characteristics of string language. In case that the attribute is a reference to another object, it is possible to specify the instance or the class that should be used to generate an instance that should be used in the reference, instead of filling the reference with `null` value as is common in contemporary mocking tools.

A. Structure of the Tool

The tool is composed of three main parts — the class analyser, the testing API and the random generators.

The class analyser is responsible for loading the classes that should be instantiated for the testing and searching for their published attributes (the attributes that have corresponding public getters and setters) and the annotations connected to them. The analyser also processes the dependencies between the classes, in order to be able to generate references to other objects. As the annotations are compiled into the bytecode, they can be directly accessed by reflection mechanism and there is no need for direct analysis of the source code of the classes.

The testing API allows testers to easily generate large collections of random data and access the annotations that specify the behaviour of the generators. This is achieved through objects populator. The populator provides a method `populate()` that takes a descriptor of a required class as a parameter and returns the desired number of automatically generated instances in a collection.

The last part, the random generators is responsible for generating primitive datatypes and instances of referred classes. Primitive data are created according to the rules given in annotations and instances of referred classes are created according to selected strategies. It is possible to generate new instances, search among already generated instances or just fill the reference with `null` value.

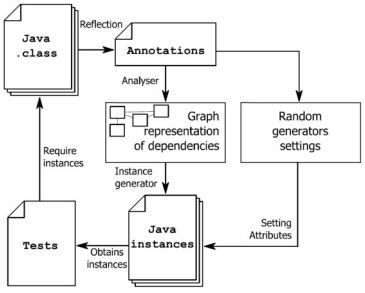

Fig. 1. Basic usage of the JOP.

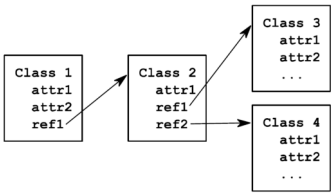

Fig. 2. Visibility of the attributes.

B. Using the JOP Tool

Our tool is intended to be used as a tool that simplifies writing of stress tests, benchmarks, and unit tests. So far it is designed to work with JavaBeans, i.e. classes that have setters and getters for each attribute and are intended for carrying data within the application. In order to work, the tool also requires an existence of public constructor without parameters. Because the generating of the instances is automated and depends on the reflection, it is necessary to set up attributes of each instance using the setters, as each setter deals only with one attribute. In multiple parameter constructors, it is not possible to find pairing between parameters and attributes and thus to choose the order of generated values that would be used in the constructor.

The process of JOP usage is shown in Fig 1. The tool can work with general JavaBeans, but without adding additional annotations (described in further sections), it cannot create data that will resemble the real ones – only to generate data from the whole space of each attribute type. The first thing the testers or the programmers have to do is to create appropriate annotation (see Section IV) of each attribute they want to randomly generate. Then, when the tests are created, testers can use generating methods and obtain the list of the instances that can be used for further testing. If no annotation is provided, the tool will behave differently for references and for primitive datatypes. References are by default set on `null`, primitive datatypes are generated with uniform distribution on the whole interval of the type.

IV. Class Analyzer

The class analyser is responsible for two main tasks —- reading annotations in order to set up the generators for the primitive attributes and analysing the structure of the generated classes in order to determine the instances generating order, setting up references, and generate dependent attributes. While parsing annotations is a simple task, working with dependencies brings several problems.

A. Types of Dependencies

There are two types of dependencies that may influence the process of generating data. The most straightforward is a dependency of one attribute on one or more other attributes from the same class. This can happen even when generating is not recursive and all references are only set to `null`. The more complicated situation is in the case of dependency on another class or on an attribute from another class. This can happen only when the generating of the data is recursive and the referenced classes are generated as well.

In our current implementation, the attribute can be dependent only on the attributes of directly referred classes, not on arbitrary method calls from the referred classes. We have decided to use this limitation in order to be able to create the order of generating only from the attribute definitions, without the need for analysis of the source text of all methods. An example is on the Fig. 2. In annotations within `Class 1` it is possible to use `attr1` and `attr2` from `Class 1` and `attr1` from `Class 2`. Attributes from classes 3 and 4 are not accessible.

B. Dependencies Within One Class

In the case of dependency within one class, the ordering of generating operations is straightforward. The analyser is generating values in the following order:

1) Create a set of all attributes A and an empty set N.
2) Find all primitive attributes without any dependencies and generate their value. Remove all such attributes from the set A and add them to the set N.
3) Search the set A for attributes dependent only on the elements from the set N. Move each such attribute from the set A to the set N and in the same time, generate its value (because it depends only on attributes from the set N, all required values have to already exist).
4) Repeat step 3 as long as the attributes from the set A are being moved.

If no attribute was moved and the set A is not empty, the remaining attributes contain circular dependency and cannot be resolved. In this case, the generating process throws an exception with the message which attributes and annotations caused the problem. It is important to note here that the mentioned dependency is caused by a tester, when setting up annotations — by claiming that attribute X should be generated as a function of attribute Y and, in the same time, that attribute Y should be generated as a function of attribute X. Removing this dependency does not require changes in domain classes, only in test data definition.

C. Annotations for the Class Dependency

It is possible to use the annotations in order to control generating of the whole graph of dependent objects. We are

using three strategies to do so.

At first, it is possible to forbid the generating of value for selected attribute, using `@Ignore` annotation. In such case, attribute will be skipped and no value will be assigned to it (if it contains a default value, it will not be changed).

It is also possible to set the value of the attribute to `null`, which is a common strategy of many data generating tools. If attribute contains different reference, it will be replaced by `null` reference. If this annotation is used for numerical type, value 0 will be used instead of `null`.

When the reference on new instance of the class should be created and assigned to the reference, annotation `@NewInstance` can be used. When the class contains a default constructor, it can be used directly. When the class contains multiple constructors or a factory method, it can be marked with `@Constructor` annotation, specifying which constructor will be used for instance generating. Parameters of such constructor or factory method can be annotated in the same way as attributes of the class, so the tool can generate their values. If no annotation is specified, the default values for each type will be used.

This annotation has to be paired with another one, that specifies the class provider. The provider is responsible for creating or obtaining desired object. There are several types of providers:

- `@TargetClass` specifies the name of the class which will be used to create new instance.
- `@RandomClass` specifies the list of classes and the probability of each one. This allows to select randomly one of several implementations.
- `@CustomClassProvider` specifies the class that is implementation of `@ClassProvider` interface and serves as a factory for creating of the instances. This annotation serves for using manually created data generators and adding them to the data generating process. The tester can create his own implementation of the generator in case that no approach provided by our tool is suitable for his or her needs.

```
@NewInstance
public Student student; // new instance of
    Student class using default constructor

@NewInstance
@RandomClass(value={Student.class,
    Teacher.class}, probabilities={0.75,
    0.25})
public Person person; // new instance of
    Student or Teacher class

@NewInstance
@CustomClassProvider(RandomStudnetProvider.class)
public Student student; // new instance using
    RandomStudentProvider factory
```

Instead of creating always new instances, it is also possible to assign one of the created instances. In such case, annotation `@SearchInstance` can be used.

During the data generating, all created objects (when annotation `@NewInstance` was used) are stored, so it is possible to search among them and use them repeatedly. `@SearchInstance` allows to specify criteria for searching among the existing objects and assign the reference to the annotated attribute. When this annotation is used alone, it will search first suitable instance (the instance of the appropriate class) and assign the reference to it. It can be also combined with tester specified annotation that will specify the rules for selecting required instance.

As there is no simple and general way how to create an annotation for instance search, we have decided to delegate this work to the matcher class that the tester will have to implement. In the tool, simple `InstanceMatcher` is prepared and the tester can use it to implement his own class that is able to decide, which searched instance is suitable. For each implementation, the corresponding annotation is automatically created and can be used immediately.

```
@SearchInstance
public Student student; // first existing
    instance of Student class will be used

@SearchInstance
@RandomStudentClass(age = 26)
public Student student; // user annotation,
    first instance of Student class with
    appropriate age will be used
```

In this example, tester implemented `RandomStudentClass` matcher that can compare the `age` attribute of provided `Student` instance and determine if the instance fulfills the criterion.

D. Generating the Dependent Instances

When the recursive generation is used, the whole process is divided into two steps. In the first step, all instances are generated and connected via references. In the second step, the dependencies between their primitive attributes are resolved and their values are generated. It is possible that the data objects contain a circular dependency, but, unlike the circular dependency between attributes, this can be solved by using strategy for searching instances.

The algorithm for generating instances works as follows:

1) Start from the generated class (the class that was required for the test).
2) Load the class annotation.
3) Check if the class is already in the dependency graph. If it is so, mark this dependency as `null`. If not so, add class as a node in the dependency graph. Store all annotations in the node.
4) Search for all attributes with the `@NewInstance` strategy. Process each such class recursively from the point 2.

This creates a tree of dependencies, with all classes that should be newly generated. Note, that the `@NewInstance` annotation means, that the new classes are always created.

Because of that, circular dependency is not allowed with this annotation, since it would lead to an infinite recursion. Instead, such references should be set to `null` or filled with instances that already exist.

When the dependency graph is finished, the creation of instances of all classes starts. The analyser keeps collections of references for each generated class that will be used for the `SearchInstance` annotation. The generating is performed using the dependency graph in following way:

1) Create an empty set A for generated attributes and empty set S for attributes with `@SearchInstance` annotation for each analyzed class, the queue of already created instances Q, set of created instances I and dependency graph G.
2) A shared copy of the collection of existing instances I_G is created.
3) Start from the class that should be generated, the instance of this class is added to the set I and to the queue Q.
4) First instance from Q is taken.
5) All attributes which should be generated are stored in the set A.
6) All attributes which should be searched (`@SearchInstance`) are stored in set S.
7) All attributes with `@NewInstance` annotation are checked if they are not causing circular dependency in graph G. If they cause circular dependency, their value is set to `null`. In the opposite case, the new instance is created and inserted into the queue Q and set I. The class is also added as a new node to the dependency graph G.
8) When the queue Q is empty, the algorithm ends. Otherwise it continues from step 4.

Because the searching of instances can be performed according to specified criteria, it is necessary to generate the values of the attributes, which are not dependent on the others. In order to find them, an empty set N is created. Then, all attributes from each version of A set for each class in I are analyzed. When the attribute has no dependency, the values are generated. These attributes are then moved from the set A to the set N.

Now it is possible to search instances in the sets I and I_G for each attribute from the set S. If no suitable instance for some of the attributes is found, its value is set to `null`. Otherwise, the reference on the instance is set to the attribute.

Finally, it is possible to generate the values for remaining attributes from the set A. To do so, it is necessary to go through the attributes in A_i sets for each instance in the I set and generate the value of the attributes using the algorithm described in section IV.B.

V. Primitive Value Generators and Annotations

We can divide generators into three main groups:

1) Number generators are responsible for generating any numeric value, integer or real.
2) Text generators are used for creating strings, according to the rules based on the length, language or structure of the string.
3) Object generators are responsible for generating of Java objects with the specific structure, such as `Color`, enumerate types or logic values.

Each attribute can be annotated with one specification of its values. When no annotation is used, the attribute is ignored (as if `@Ignore` annotation is used). The annotations for attribute generation can be combined with annotation of populators, which can be used to specify how the generated value will be used in the attribute. The populator annotations are used mainly for populating arrays and collections with primitive type values.

A. Number Generators

Java has a capability for generating random numbers, however it contains only a limited number of generators for different probabilistic distributions. We are using *Uncommons Maths* [1] library that is available under Apache Software Licence v. 2.0 and that provides among others a set of random number generators. As each distribution requires different parameters, special annotation (and corresponding generator) is created. Currently, the tool supports 8 different parametrized distributions.

For assigning a number generator to an attribute, appropriate generator annotation is used. Both continuous and discrete generators can be used for each data type.

As was mentioned before, the numerical attributes can be dependent on each other. In such case, annotation `@Expression` is used, in combination with other annotations for value generating. Annotation contains an expression that is evaluated when other attributes are generated.

```
@Expression (" rnd1 * atr1 + ref1.atr2 ")
protected int rnd1 ;
```

When evaluated, the value of `rnd1` and `atr1` will be searched in the instance where this expression is evaluated and the value of the `atr2` will be searched in the instance referred in the `ref1` attribute. The value of the attribute will be then determined as result of the provided expression. The expression can process basic arithmetic, as well as invocation of functions from `Math` library.

B. Text Generators

Generating realistic strings is a common problem, solved for example by `RandomStringUtils` class from Apache Commons project [2]. However, this class can only generate a random string of given length, with the discrete uniform distribution of probability of each character occurrence. It is possible to choose which characters will be used and which omitted, but there are no other setting possibilities. Such strings are not very realistic representation of words in any language and it is difficult to use them as a representation of other string entities (such as colour codes) as well. We are using two types of Markov chain generators:

1) Language based: Language based string generators use Markov chains with given corpus. The corpus serves to determine the probability of one letter following another letter or sequence of letters [3]. We have corpuses for English and Czech languages, but it is possible to use other corpuses provided as files.

2) Pattern based: Using arbitrary table itself allows to create Markov generators based on patterns, but it is not the most convenient way of doing so. Thus we have created a generator based on regular expressions. This generator is working in the similar way as Markov based, but instead of using probability table, it transforms the regular expression to Finite State Machine. The transitions of FSM represent the generating of next character to the string, but in this time, the probability of all characters in each state is equal.

C. Populators

Populators serves to simplify the generator logic and to allow to use each generator for any data type or data collection. Due to this, generator does not need to know anything about the type of attribute they are generating for and delegates this work on populator, which has the full knowledge of attribute declaration and can set data to the attribute. Each attribute can have one or multiple populators. When several populators are used, they are chained. There are four types of populator:

1) Numeric values populator: This populator, `@NumberValue`, serves to assign a numerical value to its attribute. Each generator is using the most generic data type it can (`double` for continuous distributions and `long` for discrete distributions) and the populator is responsible for transforming the value to the attribute data type.

2) Text value populator: This populator serves to change the provided value to `String`. It can be used on numeric values or on objects, when appppropriate `toString()` method is invoked. When optional parameter `length` is set, the string will be trimmed to the required length.

3) Array populator and collection populator: This populator serves to populate the provided array with values generated by a primitive value generator. Its parameters allows to specify the minimal and maximal size of the array (array of random length will be created) or exact length of the array. It is also possible to specify target type, to which the generated value will be casted, which is usefull when array is declared for an interface or an abstract class.

D. Populator chaining

Multiple populators can be specified for each attribute. This can be usefull for example when the value has to be transformed from number to string and then used to populate an array. Unfortunately, Java does not guarantee the order in which annotations will be processed, so we had to add `@PropertyPopulatorOrder` annotation, that will define an explicit order in which populators are used.

VI. Case Study

The tool was tested in a real stress testing and benchmarking of the system for receipt processing. The system is composed of central server collecting and distributing data for a large number of cash registers. It is used for distributing information about product price and also for mandatory electronic record of sales and receipt confirmation.

A. System Setup

The system consists of the central server, divided to application and database part that should serve multiple different companies, with a separate database for each company.

The goal of the measurement was to investigate how high load the server can handle, how fast it will be able to respond and also how it will behave under high load. The testing was performed on the production server and, with multiple instances of clients, modified to use automatically generated data instead of working with user input. The servers were running on Dell PowerEdge R820 cluster in a virtual environment, each server with 4 cores and 4 GB RAM, the database servers were equipped with Postgres database 9.5, each had 2 cores and 16 GB RAM. The clients were launched in bulks of 100 clients on one computer with Intel Xeon E3-1246, with 4 cores and 16 GB RAM. Clients were randomly divided into groups of 1 to 10 clients to simulate different size of companies with more cash registers — each group of clients shares one database. The environment of clients was observed during the whole duration of test to ensure that the clients will not become the bottleneck of the test. Eight instances of the test was run, with increasing number of clients.

B. Measured results

The experiments were running 24 hours, with a constant setting simulating high business load. The results are summarized in the table I. The table shows average and median times both for the processing of receipt and for processing of update, each time is measured from the start of the operation till its final confirmation. The measurement excluded the time required for data generating. The distribution of response times for updates and receipts was close to the exponential distribution, as can be demonstrated on histograms created for fourth experiment (other experiments showed similar behaviour). Stress tests helped to find several problems in server and client implementation, most notably with server side memory management and with reaction of clients to failed updates.

C. Experience with generator tool

The testing was performed on the application in several stages of the development. As the developers relied on agile methodology, source texts of the application went through several changes, not only regarding the added features, but also in a structure of the domain modal. Most notably, the `Receipt` class changed several times, from simple class with several attributes and two arrays with names and prices of items to an aggregated class containing collections of `Item` instances, customer and seller identification and other complex attributes. As the test data were generated automatically, these changes required only to add further annotations to the

TABLE I
EXPERIMENT SETUP AND MEASURED RESULTS

Setup		Updates				Receipts			
experiment	clients	avg. time [ms]	med. time [ms]	SD	avg. size [B]	avg. time [ms]	med. time [ms]	SD	avg. size [B]
1	200	12.0	5.7	22.0	178300	5.1	3.8	4.3	35833
2	400	11.3	5.9	19.9	177200	6.7	4.2	9.0	36920
3	600	57.2	12.5	74.4	171700	6.5	4.9	4.3	35750
4	800	77.7	30.6	82.4	166800	8.9	6.3	6.9	36160
5	1000	146.5	159.6	116.8	174600	10.2	6.8	8.8	37210
6	1200	98.9	70.6	91.3	172900	17.2	11.4	16.4	36330
7	1500	147.1	183.7	116.2	180250	33.6	23.7	30.6	36140
8	2000	140.2	140.9	84.3	184650	101.8	92.9	79.3	36780

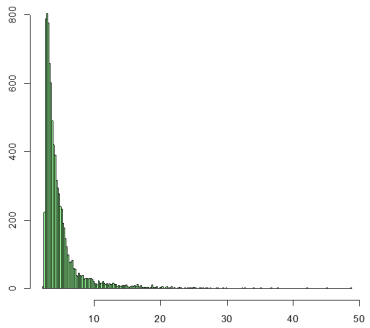

Fig. 3. Histogram of time (in miliseconds) required to performed update, normalized according to size.

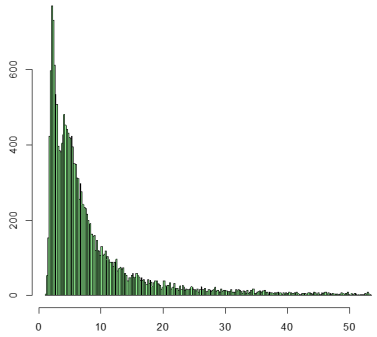

Fig. 4. Histogram of time (in miliseconds) required to accept receipt, normalized according to size.

domain objects, without need to make any changes in the tests themselves.

In order to see how demanding the running of the data generators during testing is, we measured times required to generate instances of data classes. In Table II you can see the times required for generating one instance of `Item` and `Receipt` classes. In last row the time required to generated one instance of the `Receipt` class with 10 `Items` is shown. The first column shows average duration of the first run of the generator, the second columns shows average time required when all JVM optimization took effect, after generating 100 of instances.

TABLE II
TIMES REQUIRED FOR DATA GENERATING

Class	first generating [ms]	after optimization [ms]
`Item`	9.25	1.12
`Receipt`	12.37	1.82
full `Receipt`	123.15	10.51

VII. RELATED WORK AND COMPARISON TO OTHER TOOLS

Testing based on random data is well established both in literature and practice. In some cases, the generators are very complex. For example in [4] a random generator is used to test the compiler by randomly generating Pascal programs fulfilling provided grammar and testing many possible paths of compilation. Several other examples of using random generators are summarized in [5]. The random generators are also used to achieve better code coverage in testing in different setups for a long time - [6], [7], [8], often in combination with some technique to limit the number of generated test data. In [6], genetic algorithms are used to searh for test data that provides the best code coverage and such approaches are still investigated [9], [**?**]. In [8], random test data are used in combination with guidance obtained from runtime analysis of the program under test. Similar approaches used not only for unit tests or load tests, but also for the tests of the user interfaces [10]. However, most literature focused on testing deals with methods for creating data in a deterministic way, in order to maximize code coverage of unit tests.

Several tools that allow generating of complex test data exists, but they are mostly intended for use in web applications or to test web services [11], [12], [13]. Typically, they allow to generate datasets in formats like JSON or XML, and use them as a result of web service or input for further processing. The definition of data structure is then separated from the rest of the program and tests, so it cannot dynamically react to changes in definition of data carrying objects. The main difference is in the ability to work with complex structures. The mentioned generators are able to generate test data according to the defined data structures, but cannot analyze the domain

objects that are connected with the transfered data and cannot accommodate to its changes.

The closest tool similar to JOP is PODAM [14], POjo DAta Mocker. This tool allows analysis of POJO objects and filleng their attributes with random data. It also supports both primitive data types and work with Java collections, user factories for supplying data that cannot be generated directly and additional exection of objects methods when data generation is finished. The mocker is designed to work in Spring framework environment, so when the application under test should use the PODAM generator, it becomes dependent on large part of the Spring framework. The main difference between PODAM and JOP is limited support of PODAM of working with references between objects - PODAM basically support only generation of additional objects in the object tree and no searching between already generated objects.

VIII. Conclusion and future work

We have presented a tool that should simplify testing, when a large amount of testing data object is required. Although such tools already exist, we believe that our approach helps to make tests simpler, by moving the definition of the data structure from the tests to the classes. This way, source texts of the tests are more independent on the implementation of data carrying classes. The data structure is kept on one clearly defined place. The tool is intended mainly for stress tests, measuring of quality of services and integration tests than for unit tests, as the generators are focused on creating of realistic looking data and not to achieve maximal code coverage. The generating is fully automated and does not require any effort from the tester, however, it requires the creator of data carrying classes to create specifications of the data structure.

Currently, we have the prototype implementation of the described tool, we are now working on creating a distributable version. The implementation is available publicly on GitHub, along with set of basic examples [1]. We have several issues that need to be addressed to make the tool more useful.

Our main focus is now to modify the JOP to allow the generating unit test data that would ensure the code coverage of the application under test. The current implementation is focused mainly on the stress testing and thus creating large number of data, but it seems useful to be able to direct the generating algorithm to the critical points of the application. As the code coverage achieved through the different data sets can lead to an enormous number of test cases, we also experiment with approaches to minimize the size of the test set using methods of combinatorial testing [15] and particle swarm optimization [16].

The other thing we would like to focus on is to adapt the description of the data generators to the form of the constraint of each attribute. Such constraints can then have a wider use, for example for validation of user input.

[1] https://github.com/mrfranta/jop/

Acknowledgment

The authors would like to thank Michal Dekany, who did a great job implementing the ideas from the paper to the working tool.

This work was supported by Ministry of Education, Youth and Sports of the Czech Republic, Institutional support for long-term strategic development of research organizations.

References

[1] D. W. Dyer. Uncommon math. Accessed: 2018-05-05. [Online]. Available: http://maths.uncommons.org/

[2] Random string utils. Accessed: 2018-05-05. [Online]. Available: https://commons.apache.org/proper/commons-lang/javadocs/api-2.6/org/apache/commons/lang/RandomStringUtils.html

[3] P. Brémaud, *Markov chains : Gibbs fields, Monte Carlo simulation and queues*, ser. Texts in applied mathematics. New York, Berlin, Heidelberg: Springer, 1999. ISBN 0-387-98509-3. [Online]. Available: http://opac.inria.fr/record=b1094914

[4] F. Bazzichi and I. Spadafora, "An automatic generator for compiler testing," *IEEE Trans. Softw. Eng.*, vol. 8, no. 4, pp. 343–353, Jul. 1982. doi: 10.1109/TSE.1982.235428. [Online]. Available: http://dx.doi.org/10.1109/TSE.1982.235428

[5] B. Wichmann. Some Remarks About Random Testing. Accessed: 2018-05-05. [Online]. Available: http://www.npl.co.uk/upload/pdf/random_testing.pdf

[6] C. C. Michael, G. E. McGraw, M. A. Schatz, and C. C. Walton, "Genetic algorithms for dynamic test data generation," in *Automated Software Engineering, 1997. Proceedings., 12th IEEE International Conference*, Nov 1997. doi: 10.1109/ASE.1997.632858 pp. 307–308.

[7] S. Poulding and J. A. Clark, "Efficient software verification: Statistical testing using automated search," *IEEE Transactions on Software Engineering*, vol. 36, no. 6, pp. 763–777, Nov 2010. doi: 10.1109/TSE.2010.24

[8] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler, "Grt: An automated test generator using orchestrated program analysis," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2015. doi: 10.1109/ASE.2015.102 pp. 842–847.

[9] C. Koleejan, B. Xue, and M. Zhang, "Code coverage optimisation in genetic algorithms and particle swarm optimisation for automatic software test data generation," in *2015 IEEE Congress on Evolutionary Computation (CEC)*, May 2015. doi: 10.1109/CEC.2015.7257026. ISSN 1089-778X pp. 1204–1211.

[10] K. Salvesen, J. P. Galeotti, F. Gross, G. Fraser, and A. Zeller, "Using dynamic symbolic execution to generate inputs in search-based gui testing," in *2015 IEEE/ACM 8th International Workshop on Search-Based Software Testing*, May 2015. doi: 10.1109/SBST.2015.15 pp. 32–35.

[11] Mockaroo. Accessed: 2018-05-05. [Online]. Available: https://www.mockaroo.com/

[12] Dtm test xml generator. Accessed: 2018-05-05. [Online]. Available: http://www.sqledit.com/xmlgenerator/

[13] Redgate. Accessed: 2018-05-05. [Online]. Available: http://www.red-gate.com/products/sql-development/sql-data-generator/

[14] Podam - pojo data mocker. Accessed: 2018-05-05. [Online]. Available: https://github.com/mtedone/podam

[15] M. Bures and B. S. Ahmed, "On the effectiveness of combinatorial interaction testing: A case study," in *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, July 2017. doi: 10.1109/QRS-C.2017.20 pp. 69–76.

[16] B. S. Ahmed, L. M. Gambardella, W. Afzal, and K. Z. Zamli, "Handling constraints in combinatorial interaction testing in the presence of multi objective particle swarm and multithreading," *Information and Software Technology*, vol. 86, pp. 20 – 36, 2017. doi: https://doi.org/10.1016/j.infsof.2017.02.004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584917301349