Abstract—Ontologies have more than one level of the abstraction view. This ensures to create a precise model of the subject under study. Different level architecture requires using a specific model techniques such as power-type objects to represent object that exists on more than one level. Complex object structure makes hard to implement unit testing. This paper proposes a new approach to implement unit testing by using object metamodel. That makes unit testing fully automated and increase efficiency. The paper also proposes a required extension to JUnit framework and verifies the approach on the real model.

Keywords—Automated testing; reflection; metamodel; power-type; REA ontology

I. INTRODUCTION

Latest researches in modelling are focused on metamodel and working with meta-models. Resource-Event-Agent business ontology is defined as a multi-level ontology to define economic business processes. Main advantage of the REA model is ability to make real-time overlook of the resources in model. That means the management can have relevant information about supply items, cash flow, and orders almost immediately and do not have to wait until financial statements. The economic model contrary to model in any other domain has specific needs. We can define a simple business process for example – car production. Customer buys a new car. He signs the contract and defines parameters. Every car has specific serial number. At the time of signing the contract there is no serial number because the car is not in the product line yet. The model (information system) must carry the information about car type and must be able to define the specific attributes later. Car has many accessories and accessories can be bound to the car serial number (for example the GPS system). Every accessory is bound to contract and specific car; therefore it must be bound to specific car in model with no serial number. When the car is finished the serial number is created and all components in model must be informed about changing the value. These complex model structures can be created with power-types. However automated testing of these special object types is not trivial, consume developer time, and increase complexity of the whole implementation.

II. REA ONTOLOGY

We can find a several concepts in almost every information system. If concepts are determined and defined correctly a whole design of the information system is much easier and do not violate domain rules. An adaptation of the new requirements is easier as well. These concepts are known as REA (Resources, Events, Agents). Fundamentals of the REA concepts for economic software are economic resource, economic event, economic agent, commitment, and contract. REA also specifies a domain rules to ensure consistency of the whole information system from the business perspective. There are, of course, other approaches to define the modeling entities (archetypes, pleomorphs). These concepts are focused on certain subdomain in specific business domain therefore REA ontology can generally describe all these concepts. REA was originally proposed as a generalized accounting model and published by W.E. McCarthy in 1982. Since then, McCarthy and G. Geers have extended original REA model to a framework for enterprise information architecture and ontology for business systems in 2002. REA became the foundation for several standards such as ebXML, and Open-edi. Fundamentals of the REA concepts are presented in fig. 1.

Figure 1: REA concept fundamentals

Economic Resource is a thing that is valuable and unique to other and has utility for economic agents. Usually users want to plan monitor and control that resource which must be also a part of economic information system. Examples of the economic resources are money, material, labor, tools, services and products.

Economy Agent can be an individual or organization capable of having control over economic resources and capable of interaction between other economic agents. Economic agents are usually customers, vendor or other companies. The company itself is a specific economic agent,
because the REA model is created from the company perspective.

Economic event is an act that represents an increment or a decrement specific value of the economic resource. The economic event can occur instantaneously (purchase of merchandise) or over time (rental).

Commitment expresses the obligation created in a long-time economic event. For example, commitment represents a one item on the order.

Contract is a set of increments and decrements commitments. Contract also specifies what happens if the commitments are not fulfilled. Order document is a typical example of contract concept. Order consists of items to order and can specify the penalties if the items have not been received. Additional penalties are specified as another commitments.

REA has two abstraction conceptual levels – operational level and policy level. Operational level defines concepts that describes past events and policy level defines planning by future events.

III. TYPIFICATION IN REA ONTOLOGY

Basic typification is shown in fig. 2. StudentType instance represents a different sort of students. StudentType has three attributes – yearOfStudy, groupName and course. Student instance is always directly bound to specific StudentType and specifies the instance by three attributes – studentName, studentAddress and grades. Student instance can access to StudentType attributes. Relationship between StudentType and Student is 1:M. That means if one instance of Student class changes inner attribute of the StudentType, all instance of Student class bound to that StudentType instance reflects that change automatically.

Another important fact is that StudentType is instantiated first and represents a parent of the complex Student model structure. A child objects Students are instantiated afterwards. This structure is convenient if the user must change the values of the parent object periodically and needs a specific grouping, for example year of study in object StudentType.

Fig. 3 shows resource model in REA ontology with two abstraction levels. Policy level expresses the future event by ResourceType. Object Resource on operational level represents existing resources as Goods, Services, and Rights. Modelling problem between Resource and ResourceType must be solved by typification. When user creates an order a type of resources is created. At that time there is no Resource specification such as serial number, colour etc. Specific resource is specified only as a type of Resources not the actual instance of object Resource. ResourceType is specific instance that will be specified later during order process by objects Schedule and Contract. Another words ResourceType can be and will be changed anytime and every resource connected to ResourceType must be aware of that change.

From a developer point of view there are two possibilities to implement model with changing parent objects:

- Object reference as an attribute in parent
- Inner classes

ResourceType class has specific attribute – Resource. This attribute expresses an array of resources instantiated and related to ResourceType. All instances of ResourceType are stored in hash map and all Resource instances can access ResourceTypes attributes by searching with provided ID during Resource instantiation. This construction is not adequate because of complex data structure and complicated implementation. A hash map requirement can be resolved by using reflection in every child class to get reference to parent instance.

Inner classes are much more suitable to model typification. This construction is also called a power-type [6]. Power-type is construction between two or more meta-layers to model complex objects such as policy and operational level in REA ontology. Following testing process can be applied to both constructions however inner classes are more widespread and therefore the paper aims this construction.
IV. TESTING POWER-TYPES

Implementation of the power-type is class with inner classes. From a testing point of view a power-type should be tested on the lowest level. Software testing at the lowest level (programming language) is referred to as unit testing. At this level, individual inner classes and specified methods are tested. More emphasis is placed on verification and black box testing at this level. The main concern is that algorithms works correctly and specified values are verified. Basically a unit test is a test designed to evaluate a piece of code. A suite of test can be used to evaluate an entire solution. Unit testing should be divided into four areas:

- Class scope testing
- Inheritance-level class scope testing
- Implementation-level class scope testing
- Method scope testing

Class scope testing defines the interfaces for power-types and then establishes and maintains their effectiveness though the development. Power-type is usually not interface itself, but should contain some complex methods using interfaces from inner classes.

Inheritance-level testing aims to check all methods inherited from super classes. Power-type is a specific construction of inner classes therefore there is no inherited method from super class. In fact a power-type construction is a different approach to ensure inheritance.

Implementation-level class scope testing is exactly the opposite to inheritance-level testing. The main concern is to test all implemented methods in actual classes and none of the inherited methods. Overridden methods are regarded as implemented methods.

Method scope testing is an alternative to class-level testing and is focused to specific methods or to the most important method of the specific class. This approach is suitable for specific design classes with few core methods that do a lot of work in static context. Method-level testing is usually used in evaluating robustness.

Automated tools are used widely in the test driven development approach to minimize human work time (developer/tester work), speed up test process and reduce human error. One of the most used frameworks is JUnit. JUnit is a testing framework that was originally developed by Kent Beck [9] and Erich Gamma for testing at class-level scope. Since then, a lot of implementation specifications and testing patterns has been discovered [1]. JUnit has special package to compare object values – Assert (org.junit.Assert). This package provides libraries to check values defined by object types (Integer, Double, String, Long...).

However this package is prepared only to object values and primitive data types and cannot state of equation on complex structures such as Power-type. An example of typification is presented in fig. 4. Subject under study is university consisting of several faculties. Every faculty has at least one department.

At this example the typification is much more appropriate then inheritance. Top level of the model defines a university.

University has specific attributes, such as name, address and town. These attributes have been defined to faculty and department as well. Main difference between generalization and typification is in inner attributes. In generalization the department is defined by inner attributes, attributes inherited from the Faculty and attributes inherited form the University. All attributes are on the Department instance level. That means if university changes address, all instances of classes Faculty and Department must be informed about that change and therefore one change of the attribute generates many new method calls.

A model example of University-Faculty-Department shows fig. 5. This class diagram is based in fig. 4 by applying typification approach. University class represents top of the hierarchy with attributes universityName, universityType, rectorName, chancellor, address, and town.

Figure 4: Typification example

Figure 5: Class diagram of University-Faculty-Department model example
V. DESIGNING A POWER-TYPE TESTING LIBRARY

Power-type construction could be a significant part of the REA model implementations. A lot of models have been analyzed and determined basic requirements for testing library:

- Library must be able to test across the level of typification
- Test power-types automated
- Reduce the time to create test case
- No additional implementation to tested class
- Easily implemented
- No additional dependences to other libraries

The library must be able to reveal a number of inner classes in implemented power-type. Theoretically the number of inner classes is infinite as seen in fig. 6. Practically the number depends on programming language implementation (for Java programming language is maximum Integer.MAX_VALUE). For example model from fig. 5 has three meta-levels: University, Faculty and Department and library must equals all attributes in the right meta-level. Another example has only two levels. The library must determine all outer classes to get their references automatically.

![Figure 6: Inner class level](image)

Library must be able to work with automated testing frameworks such as JUnit. Automated testing framework works with assert library and expects Boolean value. A new method, which decides equality of a two power-types, will return always a Boolean value. Method will be marked as static as well according to design pattern Library. These conditions support integration to existing testing frameworks.

Significant lines of code must be implemented to test all inner attributes of the main class from fig. 5. Additional lines must be created in inner classes to access all attributes. Access methods are often implemented just for testing purpose and makes whole code confusing. 44 lines of code must be written to state that two instances of Department class are equal.

All testing methods should be separated from tested class [1]. The new library must be able to get all needed parameters from metamodel of the instantiated class (power-type).

The key to the easy implementation and usability is well-arranged code [10] and simplicity of design. New method must be created according to design pattern Library and provide documented interface to seamless integration to current project.

New library must also use only API of the current programming language (in this case Java API).

VI. POWER-TYPE METAMODEL

Every instance of the power-types class has the own metamodel carries important information about inner structure, number of attributes and method. This metamodel must be used to discover inner attributes and get reference to the concrete value. We used a reflection to allow work with metamodel in our testing library.

A. Reflection and metamodel

Reflection as a term in information science means ability to read and change program structure and behavior during the program running. Considering object-oriented programming approach, reflection means ability to read and change object attributes, read and execute the object methods, passing calling results and instantiate new objects. Generally the reflection is able to read object metamodel during program running without changing any object attributes. Reflection is widely used with Smalltalk programming language and scripting languages. Reflection can be used as a universal tool to make object persistent [2] or to generate project documentation.

Reflection enables creating a new object instance entered by name during program running. Following source codes are in Java programming language, but same function can be done with .NET platform and languages defined under Common Language Specification. Basically there are two requirements to programming languages:

1. Ability to read object metadata and work with them as a metamodel (object self-identification).
2. Some tool to enable object metamodel extension.

Before the source code of metadata model instantiation the metamodel must be discovered.

By using reflection technique most of the requirements defined in section 5 are met. Reflection can obtain a reference to instance of inner and outer class. Reflection can obtain all declared fields in instance. If instance consists of more then one object (typically power-types) last declared field is a reference to upper object and in Java language is marked as „this$0“. Number after dollar symbol expresses the level nested class. Therefore we must use a technique to discover number of nested classes in power-type as show in fig. 7.

```java
private static int getLevel(Object obj) {
    Field[] fields = obj.getClass().getDeclaredFields();
    String[] stringArray = (fields[fields.length -1].toString()).split(".this\$S");
    return new Integer(stringArray[1]);
}
```

![Figure 7: Method to determine number of nested classes](image)

Method has only one parameter – obj. First line gets a reference to declared fields of the prototype instance. Declared fields are part of the power-type metamodel.
Second line parses the String value obtained from the array of declared fields.

The whole metamodel of instantiated class can be discovered by method getClass(). This universal method is a part of Object metamodel defined by Java programming language and therefore an additional library import is not required. Method retrieves a runtime class of an object. Retrieved object is locked by static synchronized methods of represented class. To reveal a number of levels in power-type instance we do not have to browse all inner classes. Method getDeclaredFields() is a part of Object metamodel as well. This method retrieves an array of objects reflecting all the fields declared by the class or interface represented in Class object returns by getClass() method. This includes public, protected, and private fields. The method does not include inherited fields or fields from inner classes. Retrieved elements are not sorted and are not in any particular order. If the method returns array with length 0 there is no declared field in specific instance of class or interface. The retrieved array contains specific classes – Field. A Field class provides information about, and dynamic access to, a single field of a class or interface using their metamodel. The reflected field may be a class field or an instance field. Every returned field has method toString() returning String object to describe the field. Last string is a reference to outer class. The format is the access modifiers for the field, if any, followed by the field type, followed by a space, followed by the fully-qualified name of the class declaring the field, followed by a period, followed by the name of the field. The modifiers are placed in canonical order as specified by Java language specification. The power-type in fig. 5 includes this string: “final cz.osu.example.University$Faculty cz.osu.example.University$Faculty$Department.this$1”. In this context the final means that the reference has been created in instantiation process and cannot be change except calling class destructor. Modifier is followed by definition of full outer class name including packages. Inner classes are divided by dollar symbol.

Algorithm can be speed up by comparing the number of inner classes. The whole process shows flowchart in fig. 9.

On of the fundamental features of the object-oriented programming is encapsulation. Metamodel of the power-type respects this attribute with key word private. Reflection uses special method to retrieve value of the current field and this method returns specific data type wrapped to Object data type. From an algorithm point of view the specific data type is not relevant and we do not have to check or cast retrieved value. All objects in programming language (specifically Java) have method equals implemented by default Object specification and we use this method to compare values. However if the value is marked as private the method of the reflection package throws IllegalArgumentException as a response to obtain the value. Therefore we must set accessibility mark to every field that will be obtained and compared. This approach has one big advantage. The comparative process runs over all fields of the power-type metamodel on all levels not interfered by encapsulation and therefore we can assert that two power-types are equal or not.

B. Implementation and consequences

We choose a Java programming language because we have a lot of projects implemented on Java technologies and significant count of power-types for automated testing.

First step of the algorithm flowchart diagram instructs to obtain level of classes in power-types. Full implementation is relatively long and because of page limitation the paper shows only significant passages.

![Reflection field references](image)

In this example are inner classes represented as University$Faculty. Next string is definition of current power-type class composed from class University, Faculty, and Department. Last entry of the whole string value is the reference to outer class in notation “.this$1”. Diagram in fig. 8 shows comprehensive structure of the references for example from fig. 5.

The basic idea of the testing library is to obtain metamodel at the top level of the power-type and iterate through all values. If the values are equal (including complex object data types) the algorithm finds a reference to the next inner class and repeats the comparative process. When the comparative process ends on all level of the two compared power-types the process ends and returns a result. The
 automated and fully integrated into JUnit testing framework. The project PowerTest is an open-source and is available at https://sourceforge.net/projects/powertyptest/.

VIII. Conclusion

The paper is focused to problem of automated testing power-type objects. First section introduces the reader to the problem of power-types testing and summarizes the main challenges in testing complex object structures. Next two sections define key principles and terms of REA ontology and use of power-types in REA ontology designing model. These two sections determines of using power-types as necessary technique to support REA models creation. Section 4 discuss ordinary techniques to automated testing of power-type objects and defines basic approaches to test complex object models. A main requirements for creating a universal testing library has been defined. Based on the requirements a new testing library has been designed and implemented. A new testing library has been successfully tested in many REA ontology models. The results are significant reduction of time to realize unit testing and fully integration to JUnit framework. The new code is well arranged and number of necessary lines has been reduced three-times. Moreover the testing library is not bound only to REA ontology power-types and can be used on any other complex object with inner classes. The implementation is available as open-source at Sourceforge.

Acknowledgments: This paper is supported by IGA no. 6141, Faculty of Science - University of Ostrava

REFERENCES