Data Model Based Test Case Design

Model-driven Information System Testing

Federico Toledo Rodríguez
Abstracta
Montevideo, Uruguay
e-mail: ftoledo@abstracta.com.uy

Beatriz Pérez Lamancha
Software Testing Center,
University of the Republic,
Montevideo, Uruguay
e-mail: bperez@fing.edu.uy

Macario Polo Usaoloa
Alarcos Research Group,
University of Castilla-La Mancha,
Ciudad Real, Spain
e-mail: macario.polo@uclm.es

***Abstract*—Software testing is a challenging task, but frequently the time is wasted in interactions between development team and testing team due to simple errors related with the data structure and neither with the complex business rules. That highlights that it is very important to verify that the application can handle correctly the data structure and the data types, and for this we consider to generate test cases based on the data model. We are developing a framework to generate executable test cases from a data model, to test information systems that use databases. In this article, we will present the test case design approach, based on the data model, in order to verify the correctness of the application layers that manage it.**

Keywords-test data; information system testing; model driven testing; automated test case generation

I. INTRODUCTION

The design of many applications starts with a conceptual modeling which is then used to define the database schema and the classes' structure of the domain tier of the application to be developed. Domain classes are then enriched with both methods to deal with the business goals, and with methods to deal with the persistence of their instances. Considering that the database structure is well designed, according to the requirements and the performance needs, then, it is necessary to verify that the application layer over it can manage correctly the particularities of the defined structure. Moreover, the same database structure could be accessed by different applications, such as a Web application for the customers, a desktop application as a backend, or a layer exposing Web Services in order to provide an integration mechanism with other systems. Thus, there is a correspondence between the logic components (e.g. classes, servlets and services) and the data structures (generally in a relational database). As the basic operations to manipulate data structures are the CRUD operations (*create*, *read*, *update*, *delete*) and almost any business method changing the state of a persistent instance will do a call to a CRUD operation, we will pay special attention on these methods on each entity.

Model-Driven Testing (MDT) [1] implies the automatic test case generation from models through model transformation. Our methodology follows a model-driven testing approach to automatically generate test cases from the data model, obtained from the database metadata. The generated test cases permit to verify the correctness of the CRUD operations of the entities defined in the system, according with certain coverage criteria. The methodology is supported with a framework that is based in the most important standards, mainly in the Unified Modeling Language (UML) [2].

In this article, we present how we design the test cases for information systems with databases. In Section II, the general framework is introduced. Then, in Section III, we present the main contribution of this article which is the test case design strategy. Section IV shows the state of the art regarding with test cases generation for database-driven applications. Finally, Section V draws some conclusions and future lines of work.

II. FRAMEWORK FOR INFORMATION SYSTEM TESTING

The methodology has three main phases (*Figure 1*). Each step we fits into different standards mainly from the *Object Management Group* (OMG), especially UML, in order to use general UML modeling tools. These three phases are:

- **Phase 1: Reverse Engineering.** Initially some reverse engineering techniques and tools are used in order to obtain the corresponding data model, from the physical schema of the database.
- **Phase 2: Model to Model Transformation.** The data model is processed looking for certain patterns and then generating automatically test cases for each pattern through model transformations. As a result, test cases for the data structures are generated, thus obtaining a test model.
- **Phase 3: Model to Text Transformation.** Last but not least, the test models are transformed into test code, obtaining executable test cases.

In order to represent the data model we use the UML Data Modeling Profile (**UDMP)** [3], that is an UML class diagram extension developed by IBM to design databases using UML, with the expressive power of an entity-relationship model. It defines concepts at a physical level and architecture (*Node, Tablespace, Database,* etc.), and the ones required for the database design (*Table, Column,* etc.). Several proposals use this profile to model the database structure [4-6].

 ISBN: 978-1-61208-233-2

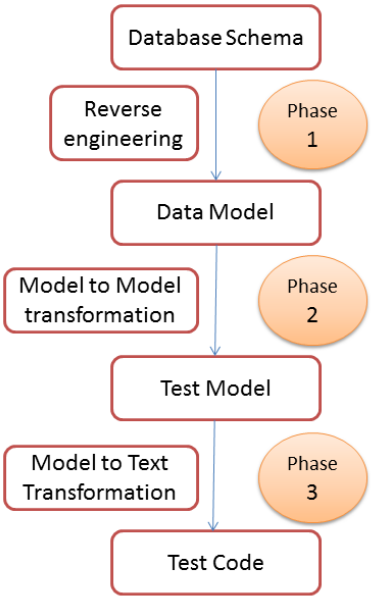

Figure 1 - Methodology and framework

For more detail on the framework, refer to [7]. In the following section, we focus on the test case design that is the most important part of the design of the second phase.

III. DATA MODEL CENTERED DESIGN

Given that in our case we generate test cases from a UDMP class diagram corresponding to the system data model, we will consider some coverage criteria as adequate to those artifacts, as for example some of the proposed by Andrews et al. [8] UML class diagrams:

- **Class Attribute (CA):** the test suite should make use representative values for each attribute in each class.
- **Association end Multiplicity (AEM)**: the test suite should make use every representative pair of multiplicities for the associations of the model.

These coverage criteria were designed for the context of testing a method or use case, where an object oriented model defines the behavior of the system. In our case we will apply the criteria for a data model instead of an object model, so we adjusted some aspects in order to make it applicable. The most important consideration is that the operations that we will be testing are *create*, *read*, *update* and *delete* of each entity. This is important to determine the oracle, because the expected results of these operations are well-known. Another consideration related with the multiplicity of the associations: according with the definitions given in the foreign keys we could have different kinds of association multiplicities, and for each one we have to considerate a special situation about the boundaries of the association end multiplicities.

To apply these criteria the framework will generate test cases to cover these situations for every substructure of the data model that matches any of the criterion, what means that for each class it will generate test cases according with *CA* criterion and for each association will generate test cases according with *AEM* criterion.

We designed the patterns, and the corresponding test cases to be generated, according with the characteristics of the relations and tables involved. In the rest of this section we present an initial design for patterns with one table, two tables and three, describing the different situations and the test cases that will be generated in order to reach the defined coverage criteria.

A. *One-table Patterns*

First, we designed test cases to test the most basic patterns: based on one table, which means to pay special attention to the attributes and the different combinations of their representative values, according to *CA* criterion.

For each attribute we can categorize in valid data and invalid data, according with the data type obtained from the column metadata, and from business rules (extracted for example from the *Check* constraints defined in the database). This way, we are defining representative test data for each attribute. In this step, we define categories and values for each one, even considering boundaries. For instance, according with the example of the *Figure 2* (one table to store the name, id and age of people), the table *Persons* has an integer attribute age, and imagine that it is defined a *check* that verifies that the value is greater than zero, then, a set of interesting values could be: {-100, -1, 0, 1, 100}. Another interesting example is related with *varchar* variables, as the *id* attribute of *Persons*, as it is defined with a length of 50, we could try with a string with 50 or fewer characters, and one with more.

Once we have interesting values for each column, we combine them with pair-wise algorithms, using our own tool called CTWeb [9]. By this way we obtain a reduced set of tuples with higher probability to find errors. If we take the Cartesian product of the different interesting values, as suggested by Andrews et al. for the CA criteria, we will have too many values, so, we decided to reduce the test set by this way.

If any of the different attributes' values used by the test case is invalid, the expected result is a fail. If we test the *create* operation then we have to check that the instance was not created, and if all the values were valid, the expected result is a pass, and we should check that the instance was created correctly with the values used in the parameters. The

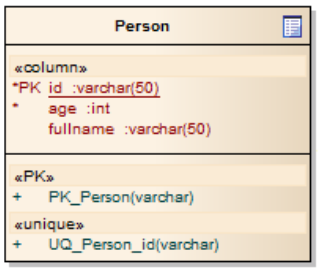

Figure 2 – Example with one table

same with the *update* operation, if all the input values are valid, we have to check that the values were updated, and if

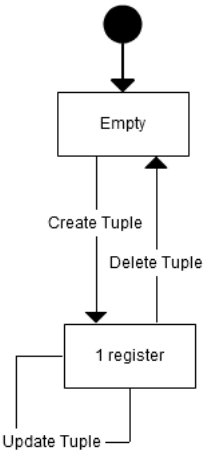

Figure 3 - State machine for 1 table

any of the input values were invalid, we have to check that the operation failed, and all the attributes in the database keep their original value.

The interesting operations are *create*, *read*, *update* and *delete* for each entity. The *read* operations are used to give support to the validation actions: if any assert fail, the error could be in the tested operation or in the *read* operation. Regarding *update* operation, there will be one for each attribute. Taking into account the previous considerations, we will apply the CRUD pattern [10] to considerate the whole life cycle of an instance, which implies to test the operation in the sequences that can be obtained expanding the regular expression: $C \cdot R \cdot (Ui \cdot R)^* \cdot D \cdot R$, where the *Ui* represents each operation that updates a different attribute. This is equivalent to generate test sequences according to the state machine presented in *Figure 3*, where there is an invocation to *read* operation in each state, in order to verify that the actual state is the expected.

Applying the CRUD pattern to the example of the entity *Person* we could generate the following test sequence:

1. Create Person
2. Read Person
3. Update Id Person
4. Read Person
5. Update Age Person
6. Read Person
7. Update Full Name Person
8. Delete Person
9. Read Person / should fail

Note that the final state of the database is the same one than the initial, what is convenient in order to have independent test cases: the execution order does not affect the expected result.

The same test sequence, which is the test behavior, can be executed with different test data, that it is going to be stored in a separated structure of the test model called *data pool*. This is known as *data-driven testing* approach [11], and the main advantage is that we can add easily new test cases just adding new rows to the data pool, indicating new interesting situations to cover with the data inputs. Therefore, the data pool will have the combination of the representing values, obtained from CTWeb.

B. *Two-table Patterns*

For this pattern, we will show as an example the one of *Figure 4*: the table *Journal* stores the different journals relating the editor responsible, whose information is stored in the table *Persons.*

Regarding the data inputs, we apply in each table the same process that for one table, except for those attributes included in the foreign key: first we define representative values and then we combine them with CTWeb in order to fill the data pools. For the foreign keys, we will have into account the *AEM* criterion, what means that we will try to associate instances in a way that covers the different representative multiplicities. The association ends of two tables (a referencing and a referenced table) could have multiplicity of 0..1 (in the referenced table side if the foreign key allows nulls, or in the referencing table side if the foreign key is unique), 1 (in the referenced table side if the foreign key does not allow nulls) or 0..* (in the side of the referencing table). Therefore, we can have the following combinations:

- 0..1 → 0..1
- 0..1 → 1
- 0..* → 0..1
- 0..* → 1

We are only considering the ones that can be implemented in a database schema with foreign keys, because for example the relation 1 → 1 it is not possible to implement with foreign keys between two tables.

The example of *Figure 4* corresponds with the last situation: 0..* → 1, from *Journal* to *Person*.

For each situation, we want to cover *AEM* criterion, and for this it is necessary to test associating entities with representative multiplicities, what is the boundaries of the defined ranges. For this, we consider to try each instance associated with 0, 1 and 2 instances of the other table. We consider that associating two instances is good enough to test the multiplicity "*".

According with this idea, different states of the database are defined, and considering the example of *Figure 4* some of these states are:

- One journal referencing one person (rel.: 1 – 1)
- Two journals with the same person (rel.: 2 – 1)
- One person that is not referenced (rel.: 1 – 0)

As we have 3 possibilities (0, 1 and 2) for each association end, we have 9 combinations. Some of these

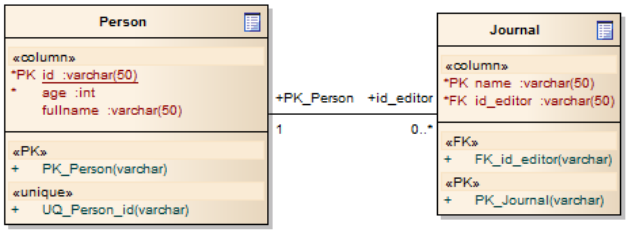

Figure 4 – Example with two tables

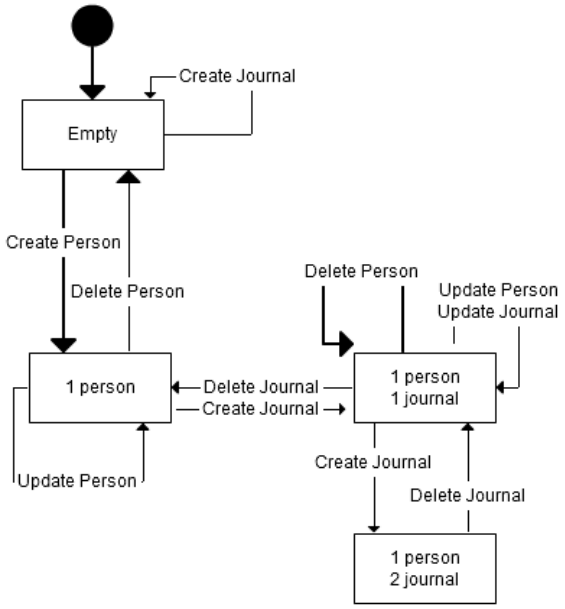

Figure 5 - Excerpt of the State Machine for Journal and Person

combinations are invalid according with the relation, as for example: in a relation 0..1 → 0..1, we cannot associate 2 registers with the same register of the other table. So, the expected result is defined by the validity of the data inputs and the validity of the number of instances to associate according with the foreign key.

The operations of *create*, *update* and *delete* force a change in the database state, only when we execute them with valid data. If we execute them with invalid data then the state should not change. The *update* operation also includes the update of the foreign key, considering that the valid data is the existing keys in the referenced table and invalid data when it does not, and similarly for *create* operation (it is interesting to test the creation of a *Journal* which references a *Person* that does not exist). If the foreign key has more than one attribute, it is necessary to considerate the update of them in the same operation.

With all these considerations, we defined a state machine, with the different states and transitions already described. The test cases that we design for this kind of patterns are based on the state machine coverage, for example trying to reach all paths, or all states and transitions. *Figure 5* shows an excerpt of the state machine for the example with *Journals* and *Persons*, and from this excerpt we present a possible test sequence generated from it (remember that after each operation there is a *Read* to verify the expected state):

1. Create Journal (without association) / should fail
2. Read Journal
3. Create Person
4. Read Person
5. Update Person (for each attribute)
6. Read Person
7. Create Journal with Person
8. Read Journal
9. Update Journal (for each attribute)
10. Read Journal
11. Update Person (for each attribute)
12. Read Person
13. Create Journal with Person (rel.: 2 – 1)
14. Read Journal
15. Delete Journal
16. Read Journal
17. Delete Person / should fail
18. Read Person
19. Delete Journal
20. Read Journal
21. Delete Person
22. Read Journal

Note that also, in this case we preserve at the end of the test case execution the original state of the database. On the other hand, this criterion subsumes the previous with one table, because the states of the table Person are part of the states of this pattern, and all the transitions of the first example are also included in this one. That means that if we find and generate test cases for a two tables' relation, it is not necessary to worry about generating test cases for each table apart.

C. *Three-table Patterns*

In the previous subsection, we are not including a type of binary relation at a conceptual level, which are the *many to many* relations, because at a database level it is implemented with three tables: two tables with the data of the entities, and another auxiliary table to store the relations, referencing the

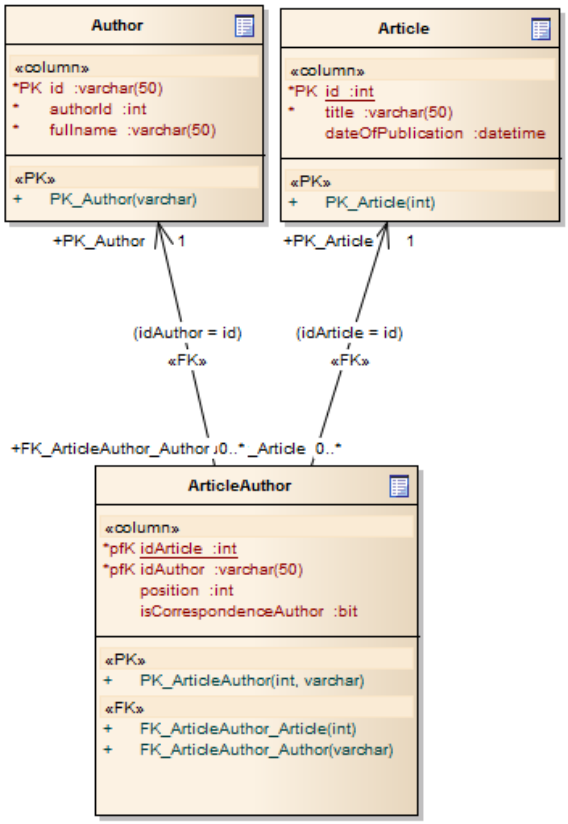

Figure 6 - Example with 3 tables

primary keys of the entities, defining its own primary key as the addition of the primary keys' attributes. Also, could be more attributes in the auxiliary table to store data related with the association. Paying attention to *Figure 6*, we can see the relation between *Authors* and *Articles*, where the same author can have many articles, and the same article can have many authors. The relation table *ArticleAuthor* has some attributes to store information about the relation between the *Article* and *Author*, for example indicating if this this author is corresponding author for this article.

This particular case imposes some considerations. We should add some valid states to the previous state machine, including association ends: 2 – 1 and 1 – 2, and 2 – 2. Also, the update of the relation table has two foreign keys, therefore there will be two special *update* operations that have to consider to reference valid (existent) and invalid (inexistent) tuples in the referenced tables.

In this example is interesting to mark something else that is that the tester has the possibility to add extra information to the data model, in order to validate some aspects of the logic that cannot be represented in the database schema. In the relation between *Article* and *Author*, perhaps it does not make sense to have an article without any authors, but this cannot be implemented in the schema, it must be managed in the logic, therefore, we want to check it. So, after the reverse engineering process we could modify the data model in order to generate test cases that can verify this kind of situations, just changing the association end multiplicity from "0 – *" to "1 – *".

IV. Related Work

Regarding test data generation for systems with databases, Tuya et al. [12] define a coverage criteria based on SQL queries, applying a criteria similar to *Modified Condition/Decision Coverage* [13] but considering the conditions of FROM, WHERE and JOIN sentences, generating test data to cover this criterion. There are some approaches (from Haller et al. [14] and Emmi et al. [15]) where the code coverage criteria are extended in order to consider the embedded SQL sentences, generating database instances to cover the different scenarios proposed as interesting. Arasu et al. [16] propose to specify in some way the expected results of each SQL included in the test, and then they can generate test data to satisfy this specification. The proposal from Chays et al. [17], called AGENDA, takes as input the database schema and categorized test data given by the user, whereby generates test cases and initial database states, and validating after the test case execution the outputs and the final database state. Neufeld et al. [18] generate database states according to the integrity restrictions of the relational schema, using a constraint solver. As far as we know, many proposals for test data generation exist, but none of them focuses on automated test model generation using model transformations.

There are various proposals to generate test cases automatically from UML models, as the ones described by Offut et al. [19] and Brucker et al. [20], but as far as we known, only Fujiwara et al. [21] proposed a special consideration for information systems with databases. In this work, they propose to generate test cases considering a UML class diagram to represent the data model, and another to represent the screens. The data restrictions (foreign keys, relations between data inputs and database fields, etc.) and pre and post conditions of the methods under test are represented with *Object Constraint Language* (the OMG's standard rules definition language). The whole test model must be specified manually, and therefore, maintained. The test cases generated are centered on the given restrictions, while in our proposal we pay attention on the data model automatically obtained, without maintenance costs.

V. Conclusion and Future Work

This paper has presented a method for test case design based on the data model, what is useful for our framework to test information systems with databases. From a well-designed database we can validate, with few extra effort, that the logic that manages the structures does it correctly.

This approach could be applied for any kind of system that uses a data base. We are developing the first group of patterns in order to put it into practice and validate our ideas, and to compare with other approaches. We believe that we can save time and effort detecting many errors before to deliberate a version to the testing team. Doing so, we can let a tester concentrate in the hard and more interesting task of testing the complex business rules of a system.

Another important point within the future work is related with complex objects types for the columns, as well as complex rules taken from checks or from the source code.

We also want to validate the scalability of the idea. For each entity it is necessary to implement some adaptation layer, but then the test cases executes completely automatically, independently of the amount of patterns defined.

Moreover, as a future work, we plan to experiment with different kind of model-driven development tools, as GeneXus [22] or OOH4RIA [23], because this kind of tools generate the system code from data models in a structured way, what could permit us to generate automatically the adaptation layer, in order to generate executable test cases with no extra cost.

Acknowledgment

This work has been partially funded by the *Agencia Nacional de Investigación e Innovación (ANII, Uruguay)*, by DIMITRI project *(Desarrollo e Implantación de Metodologías y Tecnologías de Testing, TRA2009_0131, Spain)* and by MAGO/Pegaso project *(Mejora Avanzada de Procesos Software Globales, TIN2009-13718-C0201, Spain).*

References

[1] P. Baker, Z.R. Dai, J. Grabowski, O. Haugen, I. Schieferdecker, and C. Williams, *Model-Driven Testing: Using the UML Testing Profile*. 2007: Springer-Verlag New York, Inc.

[2] OMG. *Unified Modeling Language*. 1997 [retrieved: october, 2012]; Available from: http://www.uml.org/.

[3] D. Gornik, *UML Data Modeling Profile*. 2002, IBM, Rational Software.

[4] S. Yin and I. Ray. *Relational database operations modeling with UML* in *AINA'05: Advanced Information Networking and Applications*. 2005. Vol. **1** pp. 927-932.

[5] G. Sparks, *Database modeling in UML*, in *Methods & Tools*. 2001. pp. 10-22.

[6] K. Zieliriski and T. Szmuc, *Data modeling with UML 2.0*. Software engineering: evolution and emerging technologies, 2006. Vol. **130**: pp. 63.

[7] F. Toledo, B.P. Lamancha, and M.P. Usaola. *Towards a Framework for Information System Testing - A model-driven testing approach* in *ICSOFT*. 2012. Rome, Italy.

[8] A. Andrews, R. France, S. Ghosh, and G. Craig, *Test adequacy criteria for UML design models.* Software Testing, Verification and Reliability, 2003. Vol. **13** (2): pp. 95-127.

[9] M.P. Usaola and B.P. Lamancha. *CTWeb*. [retrieved: october, 2012]; Available from: http://alarcosj.esi.uclm.es/CombTestWeb/.

[10] T. Koomen, L. van der Aalst, B. Broekman, and M. Vroon, *TMap Next, for result-driven testing*. 2006: UTN Publishers.

[11] M. Fewster and D. Graham, *Software test automation: effective use of test execution tools*. 1999: ACM Press/Addison-Wesley Publishing Co.

[12] J. Tuya, M.J. Suárez-Cabal, and C. De La Riva, *Full predicate coverage for testing SQL database queries.* Software Testing Verification and Reliability, 2010. Vol. **20** (3): pp. 237-288.

[13] J.J. Chilenski and S.P. Miller, *Applicability of modified condition/decision coverage to software testing.* Software Engineering Journal, 1994. Vol. **9** (5): pp. 193-200.

[14] K. Haller. *White-box testing for database-driven applications: A requirements analysis*. 2009: ACM, pp. 13.

[15] M. Emmi, R. Majumdar, and K. Sen. *Dynamic test input generation for database applications* in *ISSTA'07: Software Testing and Analysis*. 2007, pp. 151-162.

[16] A. Arasu, R. Kaushik, and J. Li. *Data generation using declarative constraints* in *International conference on Management of data*. 2011: ACM, pp. 685-696.

[17] D. Chays and Y. Deng. *Demonstration of AGENDA tool set for testing relational database applications*. 2003: IEEE Computer Society, pp. 802-803.

[18] A. Neufeld, G. Moerkotte, and P.C. Loekemann, *Generating consistent test data: Restricting the search space by a generator formula*. The VLDB Journal, 1993. Vol. **2** (2): pp. 173-213.

[19] J. Offutt and A. Abdurazik, *Generating tests from UML specifications.* «UML»'99—The Unified Modeling Language, 1999: pp. 76-76.

[20] A. Brucker, M. Krieger, D. Longuet, and B. Wolff, *A specification-based test case generation method for UML/OCL*. Models in Software Engineering, 2011: pp. 334-348.

[21] S. Fujiwara, K. Munakata, Y. Maeda, A. Katayama, and T. Uehara, *Test data generation for web application using a UML class diagram with OCL constraints.* Innovations in Systems and Software Engineering, 2011: pp. 1-8.

[22] Artech. *GeneXus*. 1988 [retrieved: october, 2012]; Available from: http://www.genexus.com

[23] S. Meliá, J. Gómez, S. Pérez, and O. Díaz. *A model-driven development for GWT-based Rich Internet Applications with OOH4RIA*. 2008: Ieee, pp. 13-23.

 ISBN: 978-1-61208-233-2