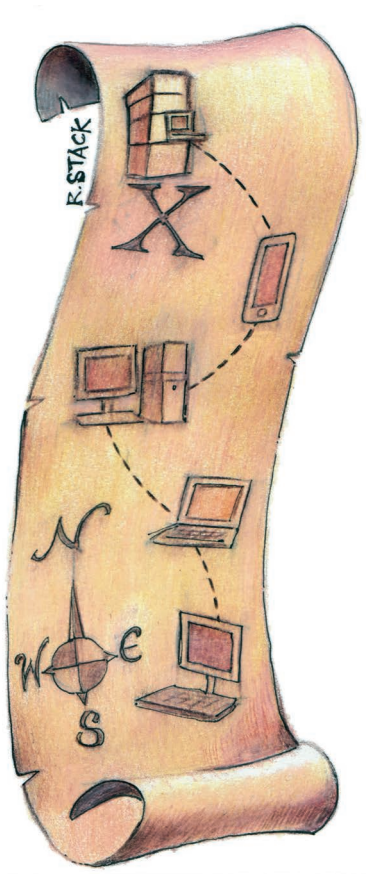
Designing a Framework with Test-Driven Development: A Journey



Eduardo Guerra, National Institute for Space Research, Brazil

Usually we read about agile development practices that seem like no more than hand waving. If that's how you feel, you'll enjoy traveling this detailed design journey and seeing up close how test-driven development (TDD) and refactoring are done in an agile environment. This article is especially insightful because of the collaborative shepherding by Rebecca Wirfs-Brock. Thank you both, Eduardo and Rebecca! –*Linda Rising, Associate Editor*

LAST YEAR, I used classic test-driven development (TDD) to develop an open source framework called Esfinge QueryBuilder (http://esfinge.sf.net). This framework interprets method signatures to dynamically generate database queries to different database technologies. Although TDD can be used for typical application development, the challenge for me was to use it to develop an extensible framework that uses reflection and interprets annotations. An additional challenging requirement was that this framework should provide the same functionality independent of the persistence paradigm used, providing a hotspot to plug in query generators for different kinds of databases.

TDD Approach

Esfinge QueryBuilder's development followed a traditional TDD approach, in which unit tests help define the developed classes' APIs and expected behavior. For those who aren't familiar with TDD, it's a development and design technique that creates tests before production code is created. I developed

0740-7459/14/$31.00 © 2014 IEEE

TABLE 1

Metrics from production code and test code in five versions of the framework.*

	Production code					Test code				
	1.0	1.1	1.2	1.3	1.4	1.0	1.1	1.2	1.3	1.4
NOC	35	46	51	58	69	40	52	55	63	104
NOM	104	128	138	215	274	183	213	222	257	358
LOC	597	777	826	1241	1614	984	1268	1337	1712	2484
CYCLO	179	214	222	320	445	184	215	224	259	348
CALL	301	330	339	426	680	554	661	719	943	1731
FOUT	46	47	49	64	128	17	11	16	18	75
CYCLO/ LOC	0.29	0.27	0.26	0.25	0.27	0.18	0.16	0.16	0.15	0.14
LOC/ NOM	5.74	6.07	5.98	5.77	5.89	5.37	5.95	6.02	6.66	6.93
NOM/ NOC	2.97	2.78	2.70	3.70	3.97	4.57	4.09	4.03	4.07	3.44
FOUT/ CALL	0.15	0.14	0.14	0.15	0.18	0.03	0.01	0.02	0.01	0.04
CALL/ NOM	2.89	2.57	2.45	1.98	2.48	3.02	3.10	3.23	3.66	4.83

** Number of classes (NOC); number of methods (NOM); lines of code (LOC); cyclomatic complexity (CYCLO); number of operation calls (CALLS); number of called classes (FOUT).*

the framework components analyzed here over a period of eight months. My development was guided by a list of requirements, including which behaviors should be able to be extensible.

No diagram or design representation was created before test creation, but I used flexibility requirements to first determine what responsibilities should be decoupled and consequently implemented in different classes. I only used tests to express my design decisions. Refactoring was performed frequently, mostly for local solutions, but I needed larger refactorings when the current solution wasn't suitable for the next requirement.

Framework Versions Overview

The experience documented in this article covers five versions of Esfinge QueryBuilder, from version 1.0 to 1.4, including the main component and the component for Java Persistence API (JPA) integration.

To understand how the framework code evolved over time, I measured some code metrics on both framework and test code. The metric suite I used contains measurements about complexity and coupling. Table 1 presents direct metrics and the computed proportions obtained.

With the exception of cyclomatic complexity, these metric values are considered low compared with existing statistical thresholds.[1,2] The computed proportions varied little through the versions, indicating that code standards were maintained as the framework evolved.

It's also interesting to note that lines of test code were always higher than lines of production code. The absolute cyclomatic number is very close between test and production code, which is what I expected. Well-formed tests rarely include conditional logic, even though they should cover all possible paths for production code. Although there were more lines of test code, the effort to develop the tests was considerably less than that for production code.

To give further insight into how these metrics are distributed among the classes, Figure 1 presents a polimetric view (a software representation based on multiple metrics), called CodeCity,[3] generated from each framework version for both production and test code.

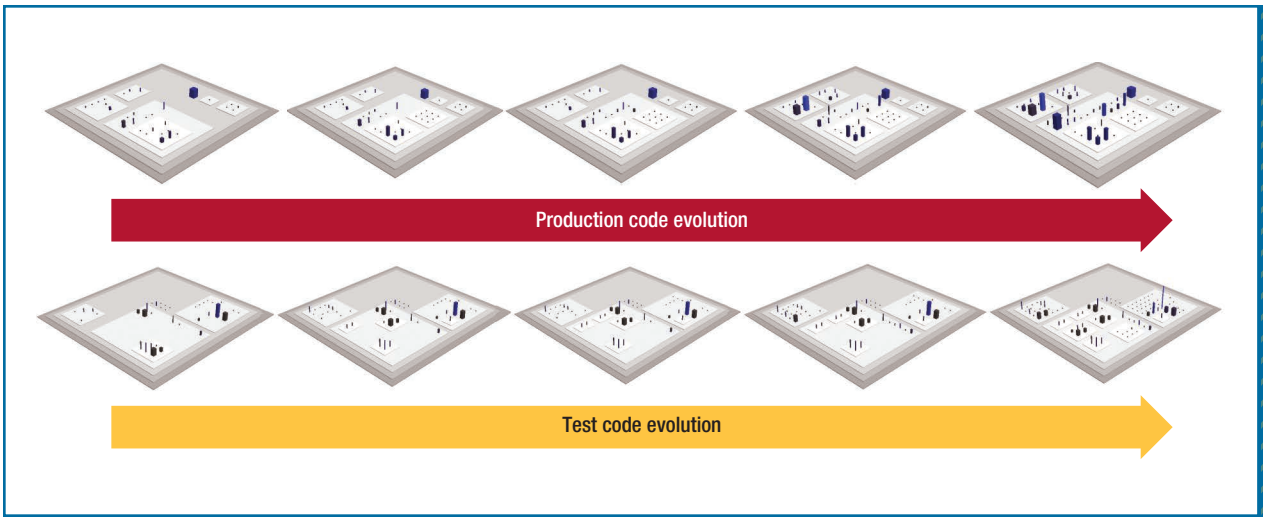


FIGURE 1. CodeCity evolution from framework version. Height represents the number of methods, length and width are the number of attributes, and color is the lines of code.

In this visualization, the height represents the number of methods, the length and width are the number of attributes, and the color is the lines of code.

By observing the CodeCity evolution, it's evident that the code is well distributed without anomalies, such as classes that keep growing in each version. My perception is that this code stability over time is one of the benefits of using TDD. This culture of keeping the code clean is reinforced by creating the simplest solutions that works and through consecutive refactorings, which are supported by automated tests.

Design Solution Evolution

As opposed to applications, in which new functionality is perceived externally, new features in frameworks can include the addition of a new extension point, called a hotspot. The introduction of flexibility that happens as part of the framework evolution process is usually more about refactoring the existing structure than introducing new code.

Figure 2 presents a class diagram with some important framework interfaces. The ones with a small flame represent framework hotspots; classes inside the gray rectangle are implemented in a specific database implementation. This diagram represents the structure from the most recent framework version.

I introduced several hotspots as the framework evolved, such as **QueryRepresentation**, which is responsible for storing the native database query. Existing interfaces also evolved to allow new features to be introduced that needed additional methods, such as **QueryVisitor**.

The **MethodParser** is a good example of how my design evolved through the framework versions following the TDD philosophy. Initially, **MethodParser** was a class that parsed the method information based on the framework's code conventions. When it was necessary to support the parsing of complex objects as parameters, it became clear that several classes should implement the parsing. So I extracted an interface from the existing class and introduced the method **fitParsingConvention()** to choose the right parser to handle a method invocation, along with a composite, **SelectorMethodParser**, to make the parser choice transparent to **QueryBuilder**.

As a result of these additions, **MethodParser** became a hotspot, and other implementations could easily be plugged into the framework. I introduced the abstract class **BasicMethodParser** because of the need to share code with the implementation. The process involved the extraction of methods, pulling them up in sequence. The framework's future needs are pointing to the creation of new components and additional hotspots in the parsing mechanism.

Designing Contracts through Tests

The first challenge in the beginning of the development was how to design the contracts between the framework classes. The first option that I considered was to use mock objects[4] in the tests for definition. A mock object is a fake object that

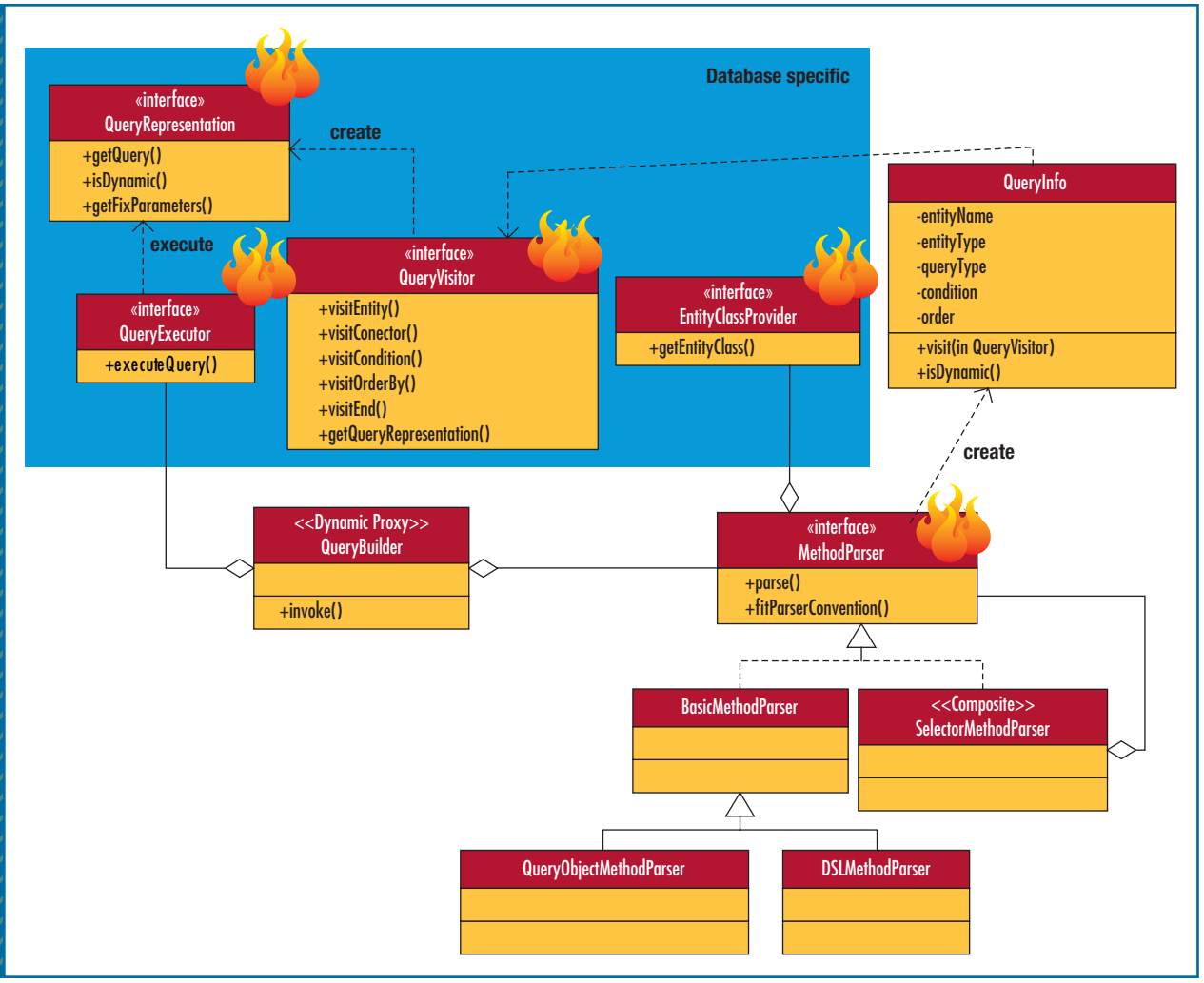


FIGURE 2. Class structure from the most recent version of Esfinge QueryBuilder. The interfaces with a small flame represent framework hotspots; classes inside the gray rectangle are implemented in a specific database implementation.

you can use to simulate a dependent object and verify the method invocation and values returned by the tested class.

The first problem I encountered using mocks happened when I needed to develop a class that depended on an external class. This class should use an API to process the Expression Language (EL; http://juel.sourceforge.net), but this API's interfaces are created by complex builders and have several methods that aren't necessary for me. Because of that, when I tried to mock this API class, the tests became hard to understand and develop.

To simplify my tests, I introduced a new class into the framework that encapsulated the use of the external API and only mocked this new element instead of the complicated API. The impact on my tests was huge: the new class fit perfectly with the tested class's needs, whereas the external API was designed to address a wide range of possibilities. By introducing this façade class, I improved my design by decoupling the rest of the framework from the details of the EL's external API.

From this experience, I learned to avoid mocking an external API or component. Instead, I found it much better to encapsulate external details in a class that provides services that offer exactly what's needed by the other classes.

The second problem I faced was

deciding which dependents to mock and which ones should be private implementation details that remain hidden. I first encountered this issue when I was creating an implementation of MethodParser. Initially, the dependence responsible for receiving and storing the parsed data was mocked in my tests. Because my design was evolving, the way that this information was stored changed several times, from a simple list to a tree implemented with the Composite pattern. My test class was coupled with this dependence in my mock definition, demanding changes on tests every time my solution evolved.

But the way that parsing information is stored should really be part of the parsing class's internal details and shouldn't be exposed to my tests or any other objects. Because I wanted to be able to freely refactor this part of my design, the wise choice was to encapsulate it inside the class. The decision to create a mock or not is in fact an important design decision that reflects whether a dependent should be exposed or not.

On the other hand, a mock object can be an important design aid when you need to define a stable contract between the developed class and a dependent. As an example of this, an important interface that I designed by using mock objects was QueryVisitor. Instead of accessing the internal structure of how the query elements were stored inside the QueryParser, my test verified if the correct methods were invoked in the QueryVisitor mock instance passed as an argument to the class.

An important kind of dependent that should be mocked in framework development is a hotspot, such as QueryVisitor. Because it represents a point where framework behavior can be extended, it's important to expose this interface and design it iteratively through the mock objects in your tests. If a test breaks as a consequence of a change in this dependent interface or from expected behavior, this can reveal where a framework client that extends this hotspot will have similar problems.

Big Refactorings: They Happen!

One of the practices of TDD is to continuously create the simplest solution that makes the current test suite pass. However, sometimes the solution being developed reaches a "dead end" when it's completely unsuitable for the next requirement. This usually happens when it becomes necessary to evolve a class whose implementation started in a previous iteration.

The first big refactoring I faced was in a class that implements the Visitor pattern and receives method calls for each query element the parser finds. In the initial version of the framework, this class generated the final query incrementally in each method call. However, the introduction of the feature to allow parameters to be removed when a null value is received made it necessary to intercept the next method call to know what should be introduced in the query. This requirement invalidated my previous solution.

My first impulse was to delete my code and start a new solution from scratch, but when I realized that if I did so I would be without automated tests for a long time, I felt uncomfortable. So I stepped back and planned how this big refactoring could be performed in small steps, making the test bar green at the end of each successive change. The two solutions needed to live together for some time, but by the end, it was possible to remove completely my prior solution. Fortunately, due to the decoupling I had already achieved, the scope of these greater refactorings was often restricted to a single class or to a small group of classes.

An important lesson that I took from these experiences is that big refactorings are inevitable when you follow a TDD philosophy in a complex set of classes. Don't think of rework as a consequence of a design mistake but as a natural part of design.

After my experience developing five releases of the Esfinge QueryBuilder, I concluded that it's possible to create software with complex design requirements by using TDD.

ABOUT THE AUTHOR

EDUARDO GUERRA is an associate researcher at National Institute for Space Research, Brazil. His research interests include software design, test automation, and agile methods. Guerra received a PhD in electronic and computer engineering from the Aeronautical Institute of Technology, Brazil. Contact him at eduardo.guerra@inpe.br.

However, using TDD alone isn't enough. Good software design doesn't "just happen" by creating tests first! A developer should choose how the tests should be created to drive the design in the desired direction.

By using TDD, the design is expressed through the tests, but the developer should know what he or she wants to create. In this context, the knowledge of design patterns and refactoring techniques is essential to drive the code to a good design solution. For instance, Esfinge QueryBuilder uses Visitor as a central pattern in its architecture, a solution that hardly would emerge naturally without prior pattern knowledge. If I didn't know about Visitor beforehand, it wouldn't have just emerged from writing new tests.

Despite TDD not being enough, it has some advantages that help the design process. Unit tests help define a single purpose for classes, enabling a good responsibility division. Additionally, because tests provide a vision from the client code's perspective, some design flaws are easy to detect. I strongly encourage you to give it a try!

Acknowledgments

I give a special credit to Rebecca Wirfs-Brock, who shepherded this article. Her contribution through several iterations was crucial to telling this story. Thanks a lot, Rebecca!

References

1. M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*, Springer, 2006.
2. E. Guerra et al., "A Reference Architecture for Organizing the Internal Structure of Metadata-based Frameworks," *J. Systems and Software*, vol. 86, no. 5, 2013, pp. 1239–1256.
3. R. Wettel, M. Lanza, and R. Robbes, "Software Systems as Cities: A Controlled Experiment," *Proc. 33rd Int'l Conf. Software Eng.*, ACM, 2011, p. 551.
4. S. Freeman et al., "Mock Roles, Not Objects," *Proc. 19th Annual ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages and Applications* (OOPSLA), ACM, 2004, pp. 236–246.