User Study

Table of Contents

Goals................................................................................................................................. 2
Overview............................................................................................................................. 2
Pre-Questionnaire Results ............................................................................................... 2
Observations From Mishra Reader .................................................................................. 3
  Existing Code Contract Usage ...................................................................................... 3
  Enforcing Invariants .................................................................................................... 3
  Suggestions .................................................................................................................... 4
Discussion of Results ....................................................................................................... 5
**Goals**

The primary goals of the user study were:

1. Observe the use the Contract Inserter on a real software system
2. Observe how Code Contracts are used in practice
3. Better understand how developers reason about which invariants to enforce

**Overview**

We contacted the developers of several open source projects to gauge their interest in giving the Contract Inserter a trial run. The lead developer of the Mishra Reader project\(^1\) - a Google Reader client with over 27,000 downloads - agreed to participate.

The study consisted of three components:

1. A pre-questionnaire focusing primarily on the developer’s usage of Code Contracts and other analysis tools.
2. The developer using the Contract Inserter to annotate one of the projects in the Mishra Reader solution. All actions were logged to record exactly how the tool was used.
3. A post-questionnaire focusing primarily on how the developer reasoned about which contracts to add and why this was difficult or easy to do. General tool feedback was gathered at the end.

The study took place on Thursday February 28\(^{th}\) via Skype and lasted two hours.

**Pre-Questionnaire Results**

We received pre-questionnaire results from three developers.

Overall, they were dissatisfied with the current state of Code Contract technology. The major complaints were:

- Lack of Code Contracts on third party libraries, resulting in too many `Contract.Assume` statements.

---

\(^1\) [http://mishrareader.codeplex.com/](http://mishrareader.codeplex.com/)
• Limitations such as no debugging support, a lack of support for multithreading and large compiling overhead (the rewriter takes too long).

• Code Contracts are too verbose.

• One developer liked the static checker. The other two found it slow, noisy and complicated.

These results are not a surprise. The Code Contracts forums\(^2\) reveal numerous issues, especially with the static checker. The official Code Contracts documentation\(^3\) also points out a number of its limitations. However, the positives appear to outweigh the negatives for these developers, as Code Contracts continue to be used (though not as extensively as they may wish).

**Observations From Mishra Reader**

**Existing Code Contract Usage**

Mishra Reader is roughly 5000 lines of code spread across 150 classes (many classes are small subclasses) and 1100 functions (many functions are event handlers with few lines of code). The project we asked the developer to annotate was the largest project in the solution, containing 1300 lines of code, 31 classes, and 300 functions.

Existing contract usage in the Mishra Reader was extremely limited. In the entire solution, there were 77 Code Contracts: 59 Requires contracts, 14 Ensures contracts and 4 Invariant contracts. The majority of these contracts (69/77) were null checks on parameters and return values. No contracts used the Contract.ForAll or Contract.Exists statements. The developer commented afterward that he never considered contracts that verify collections until using the add-in. All contracts were specified on public functions (aside from the Invariant contracts in the private Invariants function).

In the project we asked the developer to annotate, there were only 11 Code Contracts: 10 Requires contracts, 0 Ensures contracts and 1 Invariant contract. All Requires contracts were null checks on parameters. This is a meager amount of contracts considering the ~300 functions in the project.

**Enforcing Invariants**

One of the goals of the study was to gain insight into how a developer reasons about which contracts to insert.


The developer repeatedly cited his intuition and knowledge of the project as a reason for why many contracts were correct. Some contracts were difficult to reason about due to a “lack of context”, particularly involving contracts containing member variables. By lack of context, the developer clarified he was unable to determine if a contract was correct by just viewing the function that a contract belongs to (he required the context of the entire class). This is partly due to the single function perspective offered by the add-in, however, the developer still had the option of browsing the full source code within Visual Studio.

The developer communicated several reasons in the post-questionnaire why he choose not to insert correct contracts:

- The developer believed contracts should only be stated at module boundaries (this includes constructors and public methods). For example, if some objects are verified as non-null when passed to the constructor, it is pointless/wasteful to recheck the null-ness of those objects inside every member function. This also means all contracts in private functions are automatically disregarded, despite correctness.

- There was an irrational concern of the runtime overhead of too many contracts (even when the contracts were null checks). This is ill-founded because Code Contracts can be turned off for Release mode builds and null checks are essentially free.

- Code bloat. The developer felt too many contracts pollute the code.

The developer inserted no contracts as documentation. His reasoning was that if an invariant is worthy of documentation, it should be stated via a Code Contract. No software bugs were discovered using the add-in. This is not a surprise because time was limited and attention was focused on the add-in.

Suggestions

- Automatic filtering of typeof invariants (invariants of the form `this.getType() == typeof(MyType)`). The developer complained they are trivial and far too numerous, yet never utilized a filter to remove them. In fact, no invariant filters were used at all. It is not totally clear if filtering was misunderstood or just not helpful, however, we assume the former because the developer marked a number of typeof invariants as false instead of filtering them out.

- Ability to toggle the visibility of public and private methods. The developer did not care about private methods and therefore did not want to view them in the namespace tree.

- The developer wanted to view the program traces that led to an invariant being true. This is a backwards view of the Daikon invariant generation process. Daikon discovers invariants by observing traces that cause an invariant to be false i.e. an invariant exists until a trace proves otherwise.
• Tighter integration with Visual Studio (see the discussion section below)

• Several smaller user interface suggestions.

Discussion of Results

In Mishra Reader, Code Contracts are a form of partial specification utilized to verify that the parameters and return values of public methods are not null. With only 77 contracts across 1100 functions, the majority of functions are left completely unspecified. Thus, Code Contracts in Mishra Reader are clearly not used to fully specify how the software should behave. Whether or not this behavior represents typical usage of Code Contracts is an open research question. An interesting consequence of this limited contract usage is that it greatly reduces the power of the static checker, as the checker relies on contracts to reason about possible contract violations (each additional contract provides additional information to the checker). For this reason, the static checker was significantly less powerful before .NET was supplemented with contract libraries.

The developer enforced contracts based on his intuition, yet this same intuition failed him when the context of contracts grew more complicated (we consider the context of a contract complicated if it contains one or more member variables). The developer told us this directly. The state of a member variable can be difficult to reason about because class functions can be invoked in any order, potentially altering the state of the member variables in unexpected ways. When the developer complained about a lack of context, it was synonymous with him being unable to conclude if certain contracts were actually correct, despite his intuitions.

The verbosity of Code Contracts is a concern. There should not necessarily exist a tradeoff between specification power and code bloat. Indeed, the Mishra Reader developer stated he choose not to specify parts of his software because he did not wish to pollute his code with too many contracts.

Aside from bug fixes and the suggestions listed in the observation section above, the major feature the developer would like to see is tighter integration with Visual Studio, such as direct integration with Resharper. Our impression was that it was a hindrance to use a separate dialog to browse and add Code Contracts. Using Resharper, we imagine a developer right clicking inside a function in Visual Studio and seeing a list of suggested Code Contracts. This is a more natural workflow that requires fewer steps to add a contract. However, there are number of interface problems that need to be addressed before such integration would be possible, such as how often to generate contracts and where to place certain features, such as filtering.

4 http://www.jetbrains.com/resharper/
Overall, while conclusions cannot be drawn based on a single developer, we made several useful observations that will help guide and shape future studies. We were pleased with the functionality of the add-in and the suggestions we received have already improved the tool.