Emergent Optimization in Test Driven Design

Michael Feathers
Object Mentor
565 Lakeview Parkway, Suite 135
Vernon Hills, IL 60061 USA
+1 305 773 9698
mfeathers@objectmentor.com

Abstract

Programmers are often scared that they won't be able to optimize later. For that reason, they tend to optimize early leading to brittle design. Test driven design can lead to emergent optimization and code that is readily optimizable. If programmers develop test first, many of their upfront concerns about performance can be deferred.

Keywords

Test driven design, test first design, code optimization, extreme programming, emergent design

INTRODUCTION

Optimization is a favorite hobby of programmers. I haven't yet met a good programmer who hasn't spent a good deal of time thinking about optimization. As much as many programmers love to consider the relative speeds of different solutions, long time members of the programming community point out that premature optimization is one way to make a complete muddle of a design. The best advice is often attributed to Michael Jackson via his two laws of optimization [1]:

[The First Rule of Program Optimization]
Don't do it.
[The Second Rule of Program Optimization
---For experts only] Don't do it yet.

Despite Michael Jackson's admonishment, programmers often spend a large amount of time considering the costs of various language constructs and implementation strategies early in development. Some languages invite this sort of analysis more than others by presenting a large number of options of varying cost. However, attention to low-level performance is only worth the time if it produces gains that cannot be realized later. Whether those gains can be realized depends on the way that the program is modularized.

Ken Auer and Kent Beck point out that better factoring allows you optimization alternatives that you wouldn't have otherwise [2]. In this paper, I'll use an example to show how incremental test driven design can lead to optimizations that may not be considered in an upfront analysis of performance. I'll also show that in some cases where code isn't transparently optimized, it is readily optimizable because of the modularization that results from a test driven approach.

THE PROBLEM

I was designing a small parser for program-generated documents in a subset of HTML. At the time that I started it wasn't clear whether many of the features of HTML would needed, so I decided that it would be quicker to gravitate towards a solution via a small set of tests rather than use a parser generator.

The code I was developing was in C++. I used CppUnit [3] as the test harness.

My first task was to read a tag from an input stream and pass it along to another part of the software. In the first set of tests, I built up a little parser class that had the ability to skip past text that was not a part of a tag. Here is a test method on a test class:

```
void HTMLParserTest::TestSkipUntil()
{
  parser.SetInput("  <br>");

  parser.SkipCharsUntil('<');
  assert(parser.NextChar() == '<');
}
```

Later, I discovered that I was ready to start parsing tags, so I decided to create a ReadTag method that returned a tag from the input stream. But how could I determine whether I really had a tag? One option was to start asking for the various parts of a tag. I knew that in one of my tasks I would need to be able to see the name of each tag, so I decided I needed a test like this:

```
void HTMLParserTest::TestReadTag()
{
  parser.SetInput("   <br>");

  HTMLTag tag = parser.ReadTag();
  assert(tag.GetName() == "BR");
}
```

This test shows that the parser should be able to read past an HTML line-break tag and that we should be able to determine the name of the tag read. But, does it really? At this point it just shows that I should be able to get the name of the tag. What is the simplest way of satisfying the test? First of all, the HTMLTag class needs to know the text that it is working from. I could have the parser read the text into a string and pass it along to the constructor of the tag. If I am doing this intermediate step, I

should change the test. Maybe I should first verify that the parser has read past the tag.

```
void HTMLParser::TestReadTag()
{
  parser.SetInput("   <br>");

  HTMLTag tag = parser.ReadTag();
  assert(parser.HasNextChar()        ==
false);
  // assert(tag.GetName() == "BR");
}
```

To satisfy that test, I wrote ReadTag like this:

```
HTMLTag HTMLParser::ReadTag()
{
  SkipCharsUntil('<');
  while(HasNextChar()
        && NextChar() != '>')
    GetNextChar();
  return HTMLTag("");
}
```

Then I enabled the other assert in the test and changed ReadTag to this to make it pass.

```
HTMLTag HTMLParser::ReadTag()
{
  SkipCharsUntil('<');

  std::string tagText;
  while(HasNextChar()
        && NextChar() != '>')
    tagText += GetNextChar();

  return HTMLTag(tagText);
}
```

Notice that at each step, I've written the simplest code that I could to make each test pass, but I haven't let efficiency factor into my decision.

By conventional C++ standards, the code I've written so far is clearly inefficient. I create an object and return it by value, and then I create a fresh string to hold the tag text. Is there any value in this plainly inefficient code? Let's hold off for a little while longer and see.

How should we get the name of the tag? The simplest thing would be to have the GetName method parse the name out of the text:

```
std::string HTMLTag::GetName() const
{
  std::string name;
  for (int n = 1; n < tagText.size()
      && IsNameChar(tagText[n]); n++)
    name += ::toupper(tagText [n]);
  return name;
}
```

With this little method, the test passes, but look at the downside. The name of each HTMLTag is going to be calculated on demand; reparsed every time GetName is called. That will be remarkably inefficient. But is it bad? At this point, we don't know. If efficiency became a concern, we could use a profiler to determine whether this string creation is really a bottleneck in the system's performance. If we discovered that it was, there are a couple of optimizations we could perform. We could move the parsing of name into the constructor of tag:

```
HTMLTag::HTMLTag(std::string tagText)
  : tagText(tagText)
{
  CreateNameFromText();
}

std::string HTMLTag::GetName() const
{
  return name;
}
```

Or we could implement a lazy cache by having GetName save its result to an instance variable and checking it so that the parsing only happens on the first call:

```
std::string HTMLTag::GetName() const
{
  if (name.size () != 0)
    return name;

  for (int n = 1; n < tagText.size();
      && IsNameChar(tagText [n]); n++)
    name += ::toupper(tagText [n]);
  return name;
}
```

The choice depends upon whether there will be some clients of tag that don't call GetName. If there are, then a lazy cache might be the better choice.

As I moved forward, I used the same strategy for parsing the tag attributes as well, attributes like "clear" in the string "<br clear=all>" and then I was struck by a realization:

On some tags, the attributes will never be accessed. By parsing them on demand, the code was becoming unintentionally optimized.

I thought back to my decision to do this parser by hand. When I started, I thought that the grammar was far too simple to use a parser generator, so I decided to proceed test first. Interestingly, optimization was emerging as an effect of my test decisions.

Surprisingly, the code looks rather robust in the face of other possible optimizations as well. We can note the fact that the input string is never modified and pass references and offsets through the object structure. The factoring that we've arrived at supports all of these optimizations. With luck, we'll never have to do them.

APPROACHES TO DEVELOPMENT

There are many different ways to approach problem solving, but one of the most natural is *Divide and Conquer*. 'Divide and conquer' was the approach that I was using

when I first considered the parser generator. The breakdown can be seen as follows:

1. Parse tags and tag internals
2. Use tag and tag internals

By decomposing a problem into small sequential chunks, we can often get a good straw-man modularization for a system. But, what are the qualities of that modularization? Often it a decent view of the gross steps that can be taken to solve a problem. The implicit sequencing (do 1, then do 2) makes the solution easy to understand, and communicate, but the work done in step 1 is anticipatory; it may not be needed in all execution paths.

Let's consider another example. In a transaction-based payroll system, validation of the input data in a transaction can occur as it is constructed from a transaction source, or it can be deferred until the transaction is executed. In a 'divide and conquer' design, there is no force that leads to one solution over the other. However, deferring validation can be more efficient if there are cases where the validation does not have to occur. One example would be a business rule that states that employees cannot be changed from hourly to salaried status between pay periods. If a 'change employee' transaction is executed between pay periods, the system can discard it without validating the data needed to perform the transaction. However, if the validation is done prior to execution, it is wasted effort.

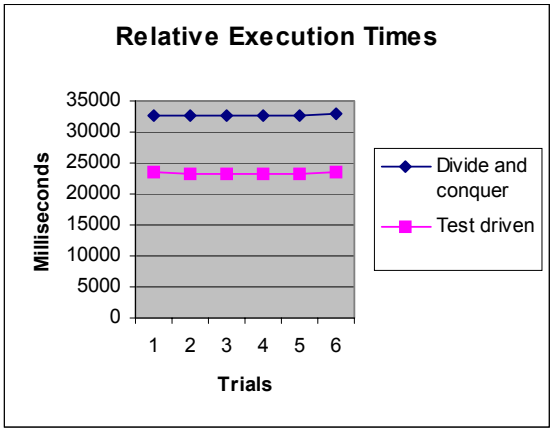

Figure 1

To demonstrate this quantitatively, I wrote two example payroll applications. One was designed with the 'divide and conquer' strategy. The other was designed test first. In the 'divide and conquer' design case, it was reasonable to validate transactions immediately. When I drove the design of the system using tests, I was only confronted with the need for validated input when I had to execute the transaction. At execution time, the 'change employee' transaction checked the execution date first before performing its validation. As a result, the system's execution was faster.

Figure 1 shows a graph of timing information for the 'divide and conquer' solution and the test first solution.

Each trial consisted of sets of eight 'change employee' transactions applied one million times to the system from an in-memory source. Half of the constructed transactions had execution dates that were valid and half didn't. Postponing the implementation of validation led to the opportunity to make it conditional during transaction execution. In a real payroll system, the amount of time saved would vary depending upon the proportion of 'change employee' transactions and the amount of validation that can be deferred.

CONCLUSION

In test first design, capability is only added to a system on demand. Programmers work from the "outside in", asking themselves "what test do I need to show that I do not have the result yet" and "what method do I need to get this result?" This can lead to designs that have a strong "calculate on demand" flavor. As we saw in an earlier section, calculations can easily be cached or moved to earlier points in the execution if the system requires optimization.

While it is possible for developers to anticipate cases where work is unnecessary during 'divide and conquer design,' it appears that the implicit deference that happens in test first design can lead to more efficient solutions without conscious optimization.

ACKNOWLEDGEMENTS

I'd like to thank Bob Martin, all of my fellow mentors and especially Mike Hill, Bob Koss, and Erik Meade for their support and valuable feedback.

REFERENCES

1. Jackson, Michael A. *Principles of Program Design*. Academic Press, London and New York, 1975
2. Auer, K., Beck K. Lazy Optimization: Patterns for Efficient Smalltalk. *Pattern Languages of Program Design 2,* Addison Wesley, 19-42
3. CppUnit C++ Unit testing framework http://cppunit.sourceforge.net