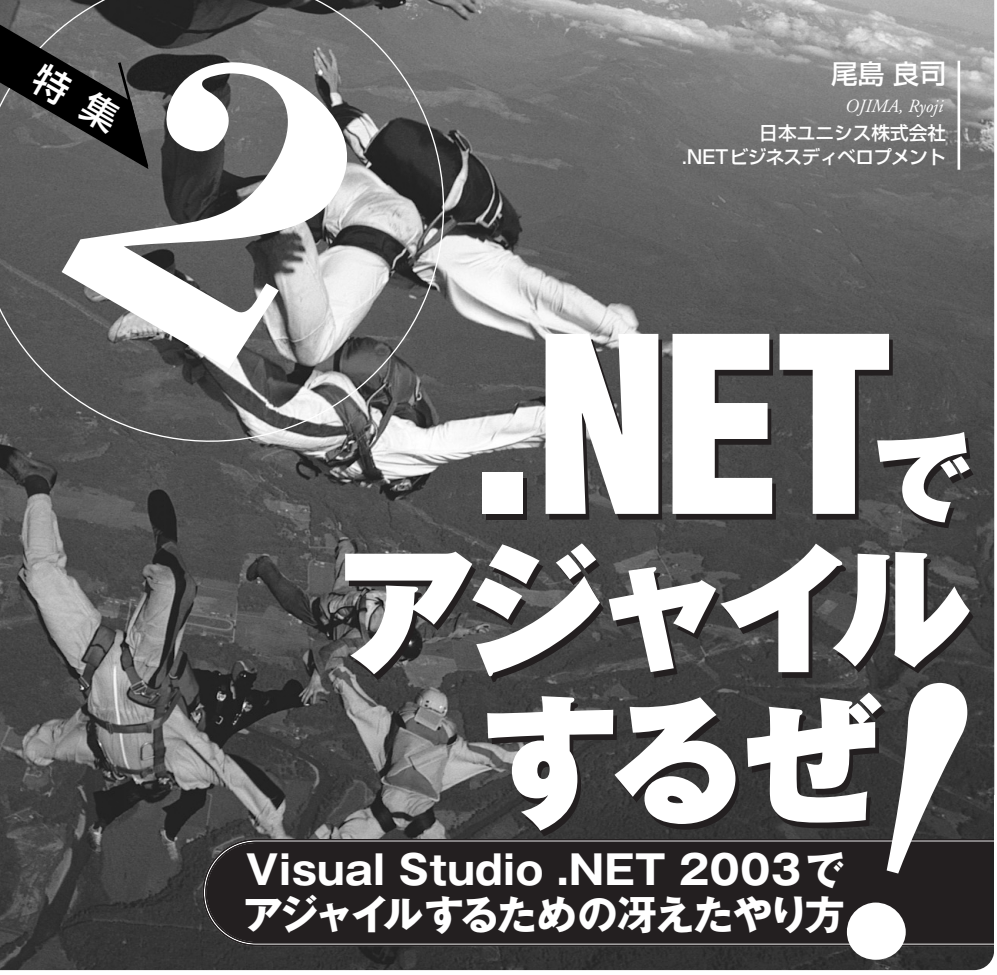


特集

2 .NETでアジャイルするぜ！

Visual Studio .NET 2003でアジャイルするための冴えたやり方

尾島 良司
OJIMA, Ryoji
日本ユニシス株式会社
.NETビジネスディベロプメント

Level

1 2 **3** 4 5

Technology Tools

- ☐ Visual Basic
- ☑ Visual C#
- ☐ Visual C++
- ☐ SQL Server
- ☐ Oracle
- ☐ Access
- ☐ ASP.NET
- ☑ Other:
  - Visual Studio .NET 2003
  - NUnit 2.2
  - TestDriven.NET 1.0.915d

Samples

はじめに

完璧だ！　パーフェクトだ！　俺ってすげぇ！

三十ウン年も生きてきたのですから、このように確信した瞬間が何回もありました。でも、その確信は、いつも必ず間違いでした。

思い出すのは中学の英語のテスト。動詞を名詞にしろという問題が出て、私は動詞の後ろにingをつけ、自信満々で提出したのでした。

そう、人間は必ず間違いをおかす生き物なのです。修学旅行で酒を飲んでコアダンプして「旅館に酔いました」と言いつくろったときも、徹夜明けで好奇心から靴下の臭いを嗅いで頭蓋骨にヒビが入るような衝撃をくらったときも、タイできれいなお姉さんだと思ったら少し特殊なお兄さんだったときも……。

ソフトウェア開発でも同様だと考えます。ソフトウェアは非常に複雑なモノなので、間違い（バグ）があって当たり前なのです。

でもそれなのに、バグまみれのソフトウェアは納品できないんです。しかも、昨今の開発では早く／安くなんて無茶な制約が課せられているんです。我々は、いったいどうすればよいのでしょうか？

ご安心ください。本稿でご紹介する「テスト駆動開発（Test Driven Development）」を取り入れれば、バグのないソフトウェアを早く安く作れちゃうんですよ、これが。

発想の転換

そのテスト駆動開発とは、テストファーストと呼ばれる“テスト”技法が進化／発展し、“開発”技法になったものです。

で、そのテストファーストとは、XP（eXtreme Programming）という開発方法論を構成するプラクティスのひとつでした。そのXPは、アジャイル（Agile）と呼ばれる軽量な開発方法論のひとつで、Agileとはウォーターフォールのような重厚長大な開発方法論の対極をなすものでした。

とまぁ、このようなテスト駆動開発の生い立ちを考えると、私のようなウォーターフォール教育を受けちゃった人間にはちょっと難しそうです[注1]。

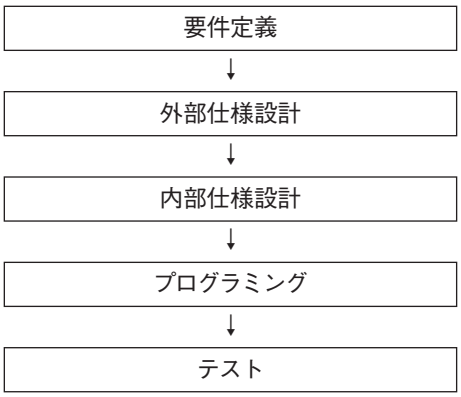
というわけで、テスト駆動開発をや

注1）ウォーターフォール教育なんか受けていないという恵まれた方は、私のような人を説得する材料として読んでみてください。

るために、まずはウォーターフォール脳を破壊し、発想を転換することにしましょう。

文書よりソースコード

ウォーターフォールは、開発を



などの工程に分割し、それぞれの工程を“文書”でつないでゆくという開発方法論でした。ウォーターフォールにおいては、この工程間をつなぐ文書が最も重要なものとされています。

でも、私の経験上、文書ってそんなに重要なものではないと思うんですよ。たとえば、プログラミングのときは内部仕様書（物理設計書）なんかほとんど見ません。たとえば、保守のときにはソースコードを直接見て、考えて、直接修正しちゃいます。プログラマの皆様、そうじゃないですか？　少なくとも私、内部設計書なんかほとんど読みません[注2]。

文書がイマイチ役に立たない理由は、文書とは曖昧な上に検証できないためだと考えます。

プログラマの皆様であれば、何をどう実装すればよいのかわからない仕様書を前にして途方にくれた経験があるでしょう。自然言語はプログラムのような厳密なものを表現するには曖昧すぎるのです。

まぁ、曖昧という問題だけなら頑張れば解決できます。設計者を質問漬けにしてもよいですし、UML2.0を使うなどの記法上の工夫をしてもよいわけです。

でも、頑張って文書の内容を明確にしても、はたしてその内容で本当に動作するのか不安になるときがありませんか？　ドキュメントのたった1ページを修正するのも怖くてこわくて、文書全体を何回も読み返したりすることがありませんか？　そう、文書には正しいことを検証するすべがないという更なる問題があるのです。

以上の問題は、プログラマなら誰でも知っているように、文書の替わりにソースコードを使用することで解決できます。ソースコードはコンパイラが厳密に解釈できるものですし、正しければ動き、間違っていれば動かないので容易に検証できるのですから。

その証拠に、文書の替わりにソースコードを使う方式のよくある（しかし不十分な）応用が、広く一般に見うけられます。そう、内部仕様書の作成とプログラミングの順序を入れ替えるという方式です。まずソースコードを作成し、コンパイル／実行して検証し、すべてが完成した後でソースコードを基にして内部仕様書を起こすわけです。

最後の内部仕様書を書く部分のコストはまったくの無駄なのですけど、従来みたいに苦労して間違いだらけの内部仕様書を書いて、解釈に悩んだり質問したりしながらプログラミングして、内部仕様書の間違いをひたすら修正するよりはかなり安くつくので、まぁ気にしないで一休み一休み。

テストコード≒外部仕様書
リファクタリング≒設計

でも、先ほどご紹介した方式、内部仕様書の作成とプログラミングの順序を入れ替えるという方式には問題があると考えます。

その問題は2つ。外部仕様書が曖昧な上に検証できないことと、内部設計に相当する作業をしていないので保守しやすい美しいコードである保証がないことです。

課題その1　外部仕様書が曖昧で検証できない

外部仕様書が曖昧で検証できないという問題は、先ほどと同じ解決方法、文書の替わりにソースコードを使うことで解決できそうです。問題の内容が一緒なのですからね。

でも、ここで新たな疑問がわいてきます。外部仕様書の替わりには、いったいどのようなソースコードを書けばよいのでしょうか？

答

その外部仕様書の替わりこそ、「テストコード」です。ア

注2）ER図とか、アーキテクチャ定義書とか、画面定義書とか、ビジネスルール集とかは、隅から隅まで読みますけど。