2. ビルドの多様性

|  |
| --- |
| Are you a master builder, or a master butcher?  おまえは石工の達人か、それとも非道の達人か？  (映画「十戒」より) |

# ビルドとは

私たちがソフトウェアを開発するときに中心となるのは「ビルド」です。ビルドには、2つの意味があります。

## 1. 実行可能なソフトウェアのこと (名詞)

ソフトウェア開発では、実行可能なソフトウェアを何度も繰り返してリリースしながら作業をすすめます。この、実行可能な開発中のソフトウェアのことをビルドといいます。



## 2. 実行可能なソフトウェアを構築すること (動詞)

実行可能なソフトウェアは、複数のプログラムファイルや設定ファイルをコンパイル[[1]](#footnote-1)して統合し、構築します。この構築作業のことも、ビルドといいます[[2]](#footnote-2)。



このように意味が2つあると、「次のビルドをビルドしてくれる？」などと言わねばいけないので、少々ややこしいですね。ただし、ソフトウェア開発において「ビルド」といえば、一般に名詞での意味を意識した方が誤解が少ないと思います。というのも、「build」という英単語は名詞としても使われる語ですが、本来「実行可能なソフトウェア」という意味はないからです。つまり、名詞としての「build」はソフトウェア開発における専門用語なのです。この専門用語としてのビルド (名詞) を併用すれば、動詞の「build」は日本語で「 構築する」あるいはもっと簡単に「 作る」などと言っても、問題なく意味が通じます。もっとも、「あ、もうビルドできてるよ！」などと言えば、これが名詞なのか動詞なのか区別できませんし、する必要もありませんね。

# ビルドの位置づけとリリースの順序

図2 - 1を見てください。これは3ヶ月でソフトウェアを開発するときに、どのような位置づけのビルドをどのような順番でリリースすることになるのかを示したものです。この例では、ビルドを毎週リリースしながら開発しています。この週次のビルドのことをウィークリービルドといいます。また、このようなアプローチ自体を指してウィークリービルドということもあります。

各ビルドには番号を付けて、それぞれを区別できるようにします。この番号のことをビルド番号といいます。ビルド番号については後述します。



**これまでとりあげたビルドは累積・進化するものなんですよね？**

**累積・進化するものであれば、「・・・側面」は変化しない。「・・・品質」も落ちたら問題。**

**使いすてのプロトとは区別する必要がある。**

**また、例えば日々のビルドとかのビルドと計画で特定されたビルドとは区別する必要がある。「求められる品質」が定義されるのは後者だけ。**

図2 - 1 ビルドの位置づけの変化

**へん**

各ビルドで、少しずつ機能を追加していきます。開発がすすむにつれ、各ビルドで各ビルドの位置づけによって、注目すべき側面や、求められる品質が変化します。図で示したように、同じビルドが複数の位置づけを持ちえることに注意してください。

各ビルドの位置づけ (図中のファーストビルドやウィークリービルドなど) については後で詳しく説明しますが、とりあえず語義通りの意味であると捉えれば十分です。つまり、ファーストビルドは最初のビルド、ウィークリービルドは週次のビルドです。RCビルドは、Release Candidate (リリース候補) の略です。RCビルドがテストにパスすれば、それはそのままリリースビルド (ユーザーにリリースするビルド) になります。

# ビルドをリリースする範囲と、反復の階層

ビルドをリリースする範囲を図2 - 2に示します。これは、そのビルドごとに異なります。一般に、ウィークリービルドはプロジェクトのQAチームだけにリリースしますが、マイルストーンビルドはユーザー (社外) にもリリースすることがあります。最終的には、リリースビルドをユーザーにリリースすることが、ソフトウェア開発プロジェクトのゴールとなります。



図2 - 2 ビルドをリリースする範囲

図2 - 1に示したビルドのリリースを表す丸印の大きさは、図2 - 2と同じく、リリースの範囲が反映されていると考えればわかりやすいでしょう。

このリリースの範囲は、いわゆる反復型開発の「反復」の大きさに対応します。つまり、リリースの範囲が小さいビルドのリリースは頻繁に繰り返しますが、リリースの範囲が大きいビルドの頻度は少なくなります。図2 - 3を見てください。



図2 - 3 反復の階層とリリースの範囲 (例)

例えば、開発者は、自分の開発作業のために、数分～数時間おきに自分の環境でビルドします。そして1週間に1度、ビルドマシンでビルドした統合ビルドをQAチームにリリースします。さらに、ユーザーにリリースするのは1ヶ月に1度です。このように、反復は階層構造を成しており、小さな反復の単位がより大きな反復を構成します。

ビルドのリリース以外にも、ソフトウェア開発プロセスにおけるいわゆる「反復」には、いろいろな単位で繰り返されるものがあります。XPにおけるテストから実装・リファクタリングという数時間単位での反復、スクラムにおけるデイリースタンドアップミーティングによる1日単位の反復、ウィークリービルドのリリースによる1週間単位の反復、スクラムのスプリントによる1カ月単位の反復、さらにUPのフェーズによる数カ月単位の反復、などです。これらも、ビルドのリリースと対応づけて捉えるとわかりやすいでしょう[[3]](#footnote-3)。

# コラム 開発中の催し② デイリースタンドアップミーティング

|  |
| --- |
| **コラム 開発中の催し② デイリースタンドアップミーティング**  スタンドアップミーティングとは、立ったまま行う会議のことで、一般には進捗の報告と問題点の共有のために催されます。立ったまま行うのは、会議を早く切り上げるための工夫です。スタンドアップミーティングは長くても15分程度で終えるのが一般的です。  デイリースタンドアップミーティングとは、毎日実施するスタンドアップミーティングのことです。朝会としても、夕会としても構いません。すぐに終えることができるように、昼食の前に実施する組織もあります。スクラムというソフトウェア開発プロセスでは、デイリースタンドアップミーティングの実施を推奨しており、これをデイリースクラムと呼んでいます。  スタンドアップミーティングに使える時間は短いですから、何を話すべきかを明確にしておかないと、有意義な時間を過ごせません。例えば、デイリースクラムでは、全員が以下の3点だけを話すべきとしています。  **ちなみに、この3つの話はエンジニアリング的にはいずれも推奨できないものです。**   * 昨日、何をしたか？ * 今日、何をするか？ * 困っていることはないか？   これ以外のトピックについて話したくなったら、デイリースタンドアップミーティングの後に、そのトピックの関係者だけを集めて、改めてミーティングをするようにします。  デイリースタンドアップミーティングは、要する時間に比して効果が非常に大きいものです。ぜひ、みなさんのチームでも実施してみてください。 |

# なぜビルドを定期的にリリースするのか

開発中のビルドを (不定期にでなく) 定期的にリリースすることにはメリットがあります。各ビルドをテストして、次のような数字を定期的に収集できるからです。

* このビルドで発見したバグの数
* このビルドで修正したバグの数
* これまでに修正したバグの数 (累計)
* まだ修正できていないバグの数 (累計)

これらの数字は、ビルドの品質が安定してきたかどうかを示す重要な指標です。バグの数が収束してくれば、ユーザーに対してそのソフトウェアをリリース可能な時期が近づいてきたと判断できます。バグ収束曲線やバーンダウンチャートなどを使ってグラフにすると、さらにその意味を掴みやすくなります。これらの数字やグラフについては5章で改めて取り上げます。

ただし、ビルドは絶対に定期的にリリースすべきである、というわけではありません。開発の初期は、製品を定期的にビルドできる状態にならないことがあります (なるべく早い時期から定期的にリリースできるようにしましょう)。また、開発の末期では、コードの修正が少なくなるために、ビルドを出す頻度を減らしても問題ないことがあります。ビルドのリリースにもコストがかかりますから、無駄な作業ならやらない方が好ましいのです。しかし、必要なとき、可能なときには、ビルドを定期的にリリースすべきです。

# ビルド曜日を決める

図2 - 1に示した通り、本書ではビルドをQAチームにリリースする頻度として1週間を勧めます。1日では短すぎることがあり、3日とか4日では使いにくく、2週間では長すぎるからです。ビルドとQAリリースをする曜日を決めておくと良いでしょう。



図2 - 4 毎週金曜をビルド日、毎週月曜をQAリリース日とした例

 

ビルド日にビルドしたビルドは、BVT (Build Verification Test) をして、これにパスすれば、QAにリリースします。このようにビルド曜日とQAリリース曜日を決めると、チーム内に心地よいリズムと緊張感がうまれます。現在開発しているビルドがいまどういう状態なのかを、チームの全員が把握し、共有しやすくなるのです。例えば、毎週金曜日をビルド日として設定した場合には、毎週木・金曜はちょっとした修羅場になるでしょう。このように、また、各メンバーが自分の作業のゴールやマイルストーンをビルドごとに設定できるので、見通しが良くなります。また、ビルドが安定してくれば、BVTに必要な時間も少なくて済むので、ビルド曜日とQAリリース曜日を同じ曜日に設定して構わないでしょう。

**ちなみに、こうなっているところで、問題・課題を抱えていない組織は見たことがありません。**

海上自衛隊は1週間の感覚を失わないように、金曜日をカレー曜日として、必ず週に１度カレーを食すそうですが、ビルド曜日はそれと似ているかもしれませんね。

# ビルドの世代交代

図2 - 1には、各ビルドのリリースのタイミングを示しましたが、もちろん各ビルドとお付き合いするのはリリース時だけではありません。リリース前にはこのビルドを開発し、リリース後にはこのビルドをテストする必要があります。図2 - 5を見てください。



図2 - 5 ビルドの世代交代

これは、ビルドを1週間に1度、QAにリリースしながらソフトウェアを開発するときに、各ビルドの状態がどのように遷移するか、また作業対象とするビルドがどのように入れ替わっていくかをUMLの状態遷移図で示したものです[[4]](#footnote-4)。図中の「開発中」「テスト中」というのは、各ビルドの状態を表しています。このとき、各ビルドの寿命は (1週間ではなく) 2週間となります。QAチームがN番目のビルドをテストするのと並行して、開発チームはN+1番目のビルドを開発します。例えばある週では、3番目のビルドをテストしながら、4番目のビルドを開発します。その次の週では、4番目のビルドをテストしながら、5番目のビルドを開発します。このように、開発とテストを並行して行っていきます[[5]](#footnote-5)。QAによるテストの結果は、バグ追跡システムを介して開発チームにフィードバックされます。

# ビルドに番号をつける

**テストで見つかった欠陥のFixはどこで行うの？**

各ビルドには、それぞれを区別できるように番号をつけて管理します。この、ビルドを一意に識別するための番号をビルド番号といいます。

当該のビルドを使っている (テストしている) 人が、この番号を確認するための手段を用意しなければなりません。QAやユーザーがバグを報告するときは、必ずビルド番号も一緒に報告してもらわねばならないからです。どのビルドでバグが再現するのかわからないと、そのバグの調査に支障が出てしまいます。

例えば、コンソールアプリケーションを開発するなら、起動時にビルド番号とバージョン番号を表示するようにすると良いでしょう。図2 - 6は、Windowsのcmd.exeが起動時に表示するバージョン情報の例です。



図2 - 6 cmd.exeが起動時に表示するバージョン情報

このほか、/version などのようなバージョン情報を表示するコマンドを用意するのも有効です。5章に、バージョン情報を表示するJavaプログラムの例をのせましたので、参照してください。

GUI[[6]](#footnote-6)をもつウィンドウアプリケーションなら、About box (バージョン情報) ダイアログを用意するのが一般的です。例えば、インターネットエクスプローラは、Alt+H Aとキー入力[[7]](#footnote-7)すると、バージョン情報ウィンドウを表示します。



図2 - 7 Internet Explorer 8 (英語版) のAbout box

ウェブアプリケーションを開発するなら、バージョン情報を表示するページを用意します。また、クライアントサーバ製品を開発するなら、クライアントのバージョン番号を確認する手段と合わせて、クライアント側からサーバ製品のバージョン情報を確認する手段も用意しておくべきです。

このように、みなさんが開発するソフトウェアにもバージョン情報をつけて、各ビルドを区別できるようにしておきましょう。

# バージョン番号とは

バージョン番号とは、ユーザーにリリースするときに増やしていくソフトウェアのバージョンの番号で、一般にメジャーバージョン番号とマイナーバージョン番号のふたつからできています。バージョン番号はビルド番号と同じくリリースの世代を表すものですが、ビルド番号よりも大きな反復の単位で少しずつ増やします。例えば、ユーザーへの最初のリリースでメジャー番号を1, マイナー番号を0としたとき、バージョン番号は1.0と表記します。その後、細かいバグの修正を加えたリリースでは、バージョン番号を1.1, 1.2, 1.3, …のようにマイナー番号を増やしていきます。そして、重要な新機能が追加されるのに伴い、バージョンを2.0とします。ただし、バージョン番号のつけ方に決まりがあるわけではありません。内部的には大きな変更が加えられたソフトウェアでも、見た目がそれほど変わっていなければ、リリース時にはマイナー番号を増やしてメジャー番号はそのままにすることがあります。反対に、内部的にはほとんど変わっていないのに、見た目が大きく変わっていれば、メジャー番号を増やしてリリースすることもあります。

以上のように、バージョン番号は、基本的にどのようにつけても構わないのですが、そのルールやポリシーは決めておく必要があります[[8]](#footnote-8)。ある製品のを複数のブランチからリリースするときには、それぞれを区別できるように、バージョン番号をブランチと対応させておくと良いでしょう。この方法については、★3-15節で改めてとりあげます。次節で、バージョン情報の書式をより詳しく見てみましょう。

**リビジョン番号の話しはしないのですか？**

**そうだけど、組織内ではルールを決めておく必要がある。**

# コラム TeXのバージョン番号

|  |
| --- |
| **コラム TeXのバージョン番号**  TeXは、ドナルド・クヌース先生が開発した書籍の組版ソフトウェアです。クヌース先生が「The Art of Computer Programming」という本をお書きになったとき、組版された数式がとても汚かったのが許せず、美しく組版できるソフトウェアを自分で作ってしまいました。TeXは公共に供されており、WYSIWYGワープロ[[9]](#footnote-9)が全盛となった現在でも、大学の論文のように数式を多く含む文章ではまだTeXが使われているようです。  TeXはバグが少ないことでも有名ですが、そのバージョン番号の付け方も美しさを求めるクヌース先生の属性が反映されていて興味深いものです。クヌース先生は、TeXのバージョン3をリリースしたとき、これ以上の新規機能の追加は行わないことを宣言しました。そして、バグ修正版のバージョン番号を3.1, 3.14, 3.141のように増やしていきました。クヌース先生は、ご自身が亡くなったとき、TeXのバージョンをπにするとしています。TeXが完全なソフトウェアに漸近していくイメージが想起されますね。現在のTeXのバージョンは3.1415926です。バージョン番号は、ソフトウェアをリリースする側が自由につけて良いことを示す好例でしょう。  もっとも、ソフトウェアの種類によっては、バージョン番号にユーザーの意向を反映しなければならないこともあります。  **知りませんでした。**  **おもしろい！** |

# バージョン情報の書式

バージョン情報とは、バージョン番号とビルド番号を組み合わせたものです。バージョン情報の書式は、どうあるべきという決まりがあるわけではありませんが、少なくともメジャーバージョン#、マイナーバージョン#、ビルド#の3つを含めることが多いです。もうひとつ数字を加えた次のような形が、バージョン情報として一般的に使われる書式です。

**~~① なぜマイナー番号があるのにビルド#が必要なの？~~**

**② マイルストーン＃も、私は使ったことない＆見たことないので、？？という感じがします。**

|  |
| --- |
| <メジャーバージョン#>.<マイナーバージョン#>.<マイルストーン#>.<ビルド#> |

「各ビルドを一意に特定できる番号をつけておく」だけなら、ビルド番号だけでも用は足ります。しかし、新しいバージョンをマーケティングして製品に新しい価格をつけたり、以前のバージョンとは異なるサポートのポリシーを適用したり、以前のバージョンとの互換性についてユーザーに注意を喚起したりするために、バージョン番号もバージョン情報に含めます。それぞれの番号を、いつ1ずつ増やすのか、ポリシーを決めておくと良いでしょう。例えば、デイリービルドや、後で紹介するCI (Continuous Integration; 継続的な統合) というアプローチを使って1日に何度もビルドをするなら、次のような書式も使いやすいと思います。

|  |
| --- |
| <メジャーバージョン#>.<マイナーバージョン#>.<ウィークリー#>.<ビルド#> |

一日に何度もビルドするときも、それぞれできちんとビルド番号を数えます。しかし、このビルド番号とは別に、週次で1つずつ増やす番号を用意しておくのです。ビルドのリリース計画は、この週次の番号で立てれば分かりやすくなります。CIについては、★4-16節で改めてとりあげます。

また、一日に一度だけビルド (デイリービルド) をするなら、ビルド日をそのままビルド番号にすることもできます。例えば、2010年8月4日のビルドの番号は100804、9月10日のビルド番号は100910のようにします。

このほかのバージョン情報に関して検討すべきポリシーとしては、ビルド番号を通しでつけていて、メジャーバージョン番号を増やしたときに、ビルド番号を1にリセットするか、そのまま使い続けるかということがあります。どちらでも問題はないのですが、私は1に戻さず、そのまま使うことを勧めます。大きいビルド番号は、そのソフトウェア製品が長く保守され、種として長く生き続けていることを示すからです。大きなビルド番号は、そのソフトウェアが良い性質を持っている証なのです。また、ビルド番号を1にリセットしなければ、ビルド番号だけでもその製品のビルドを一意に特定しやすくなります。

**別の視点もあります。**

バージョン情報の書式は、製品のリリース計画や開発計画と密接に関連しています。ぜひ丁寧に検討して、みなさんのソフトウェア開発に最適なバージョン情報の書式を探してみてください。

# ビルドのリリースを計画する

ウィークリービルドのアプローチを使うときは、ビルドのリリース計画には日付よりもビルド番号を使う方が便利です。例えば、表2 - 1のようなビルドのリリース計画書を記述します。

表2 - 1 およそ3か月で開発するときのビルド計画書の例

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **build#** | **Milestone** | **ビルド日** | **QAリリース日** | **ユーザーリリース日** |
| 1 | First build | 2011/02/04 | - | - |
| 2 |  | 2011/02/18 | 2011/02/20 | - |
| 3 | M1 (Alpha) | 2011/02/25 | 2011/02/27 | 2011/03/04 |
| 4 |  | 2011/03/04 | 2011/03/07 | - |
| 5 | **どっかで定義されるのでしょうが、各ビルドの（少なくともリリースする）スコープも定義されていないと、リリース計画はほとんど意味をなしません。　→　後ろにありましたね。** | 2011/03/11 | 2011/03/14 | - |
| 6 |  | 2011/03/18 | 2011/03/21 | - |
| 7 | M2 (Beta) | 2011/03/25 | 2011/03/28 | 2011/04/01 |
| 8 |  | 2011/04/01 | 2011/04/04 | - |
| 9 |  | 2011/04/08 | 2011/04/11 | - |
| 10 |  | 2011/04/15 | 2011/04/18 | - |
| 11 | RC1 (Final) | 2011/04/22 | 2011/04/25 | - |
| 11 | RTM | - | - | 2011/04/29 |

ここに例示した計画書は、ソフトウェアをおよそ3カ月で、ウィークリービルドをリリースしながら開発するものです。ここでは、金曜日をビルド日とし、月曜日をQAへのリリース日としています。また、2番目のビルドから毎週リリースすることを計画しています。

**でも、「ちょっとした修羅場」にはなるんですよね？**

ウォーターフォール型の開発プロセスでは、結合テストの時期にはひと月の間、毎日修羅場になることも珍しくありませんが、ウィークリービルドのアプローチは、このリスクを時間的に分散させてリズムを作ります。このとき、各ビルドで何を実現するのかを決めておかないと、開発が予定どおりに進んでいるのかどうかを検証することができません。どのユースケースをどのビルドで実装するのか決めておくとよいでしょう。

**個人的にはUCでスコープを定義するのはおすすめできません。**

表2 - 2 各ビルドで何を実現するのかを決めておく[[10]](#footnote-10)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **UseCase#** | **機能の概要** | **CC[[11]](#footnote-11)予定 ビルド#** | **CC実績 ビルド#** | **実装 担当** | **テスト 担当** |
| UC#1 | 会議室の予約 | build#2 | t.b.d. | 津田 | 中原 |
| UC#2 | 予約済み会議室の確認 | build#2 | t.b.d. | 津田 | 中原 |
| UC#3 | 会議室の登録 | build#3 | t.b.d. | 津田 | 中原 |
| UC#4 | 予約の15分前に予約者にリマインダメールを送信 | build#3 | t.b.d. | 津田 | 中原 |
| … |  |  |  |  |  |

ソフトウェアのドメインによっては、上のようなユースケースに基づく単純な計画書にはならないかもしれません。しかし、何らかの形で、どのビルドでどこまでの機能を実現するかを決めておくことが必要です。もちろん、反復を繰り返すごとにユーザーの要求が変化してしまうことがあるので、最初の段階から完全な計画を立てることは困難ですが、少なくとも次のマイルストーンビルドまでは、各ビルドで何を達成する予定なのかを決めておくべきです。各ビルドはQAチームにリリースされ、計画が達成できたかどうか検証されます。

**想定する計画が違うのでしょうが、そうはなりません。**

ウィークリービルドを単位として計画を立てることで、ソフトウェア開発に付随するほかの計画の多くもすべてビルド番号で行うことができるようになります。この機能制限を解除するのは「何月何日までに」ではなく「どのビルド#までに」、このバグを修正するのも「何月何日までに」ではなく「次のビルドで」、このサブシステムのテストを開始するのも「何月何日から」ではなく「どのビルド#から」、といった具合です。

また、このようなやり方を続けると、以前開発したのと同じようなドメインや規模のソフトウェアを開発するとき、ビルドを何回くらいリリースすれば品質が安定し、リリース可能になるかを予測できるようにもなります。例えば「この規模のソフトウェアならビルドを20回くらい出せば品質が安定してくるだろう」などと予測できるようになるのです。

ソフトウェアの開発計画を丁寧に月日で立ててみたけど全然予定どおりにいかなかった、という経験をお持ちの方は、ぜひ一度ビルドを単位として開発計画を立ててみてください。開発期間が決まっていれば、ウィークリービルドをリリースできる回数も決まってきますので、計画は立てやすいはずです。実現しなければならない機能を何番目のビルドで実装するのかを決め、一緒にテスト計画も立案しましょう。

# ビルドで作業の進捗を測る

ソフトウェア開発の進捗を測るのは難しいのですが、ビルドを使えば比較的簡単に正しい進捗率が得られます。

例えば、2週間 (10営業日) で完了する予定の作業があるとしましょう。これを担当した人は9日目の夕刻に「作業は順調で、進捗率は予定通り90%です！」と上司に報告しました。さて、この進捗率は信頼できるものでしょうか？ 果たして、彼はその1週間後に「難しい問題が発生したため、作業が遅れています。進捗率は95%になりました。」さらにその1週間後、「進捗率は99%です、あともう少しで終わります。」と報告しました。この次の週は、進捗率はどれくらいになっているでしょうか？ このように、進捗率が90%からなかなか進まない症状を90%シンドロームといいます。

このような曖昧な進捗率を避けるには、作業ではなく、結果に基づいて進捗を測ることです。例えば、コードコンプリートした機能の数や、テストに合格したシナリオの数などを使います。つまり、実際に動作するソフトウェア (ビルド) を使って、進捗を測るのです。実装しなければならないすべての機能に対して、いくつの機能を実装し終えたかを数えれば、より信頼できる進捗率が得られます。これを結果駆動のアプローチといいます。



図2 - 8 ビルド番号で作業のタイミングを図る

# コラム ソフトウェア開発のメタファ② 建築

|  |
| --- |
| **コラム ソフトウェア開発のメタファ② 建築**  ソフトウェア開発と建築の類似性は、これまでに多くの人が指摘しています。例えば、建築にソフトウェアが影響を受けた例のひとつにデザインパターンがあります。デザインパターンは、ありがちな問題とその解決方法をカタログ化したものです。これは、建築家のクリストファー・アレグザンダー氏が著書「パタン・ランゲージ」の中で提唱したのが最初です。アレグザンダー氏は、「小さな人だまり」「座れる階段」「街路を見下ろすバルコニー」など、過去の街づくりにおいて普遍的に使われてきたパターンを収集し、新しい街づくりをするときもこれらのパターンで構成すべきだとしました。  これに大きな影響を受け、これまでにソフトウェアエンジニアらが普遍的に使ってきたソフトウェアの構造を収集し、カタログ化したものがソフトウェアのデザインパターンです。線形リストやスタックといった古典的かつ実践的なデータ構造と同じように、デザインパターンはありがちなソフトウェアの構造に名前をつけ、暗黙知を形式知へと変換しました。  また、建築に使うさまざまな図面 (立面図、側面図、配管図など) は、同じ建物のいろんな側面に注目して異なる図面を使い分けるという意味でUMLと対比されます。UMLとは、ソフトウェアの構造を説明する図面のセットです。ソフトウェアのさまざまな側面に注目し、クラス図、状態遷移図、シーケンス図など、複数の図を使い分けてソフトウェアシステムをビジュアルに描写します。ただし、一般にUMLは仕様を厳密に記述する図面にはなりえません。もちろん、ラフ・スケッチにはなります。  **ちなみに、UMLは通常、図面になりえない。-- 図面が仕様という意味であれば。**  **勿論、（ラフ）スケッチにはなる。**  このほか、エリック・レイモンド氏の「伽藍とバザール」というソフトウェア開発に関する有名な論文も、大聖堂建築とソフトウェア開発を対比させたものです。  そもそも、ビルド (build) には、建物を建築するという意味がありますね。ビルディングといえば建物のこと、ビルダーとは石工のことです。もし、美しいものを建てたいという気持ちになりたいなら「大聖堂」という小説を薦めます。主人公のトム・ビルダーが、あなたを壮大な物語に誘うでしょう。  **現状はそうかもしれませんが、今、多くの人が探してますよね。アランケイも含めて。**  建築物とソフトウェアの大きな違いは、建築物は物理学の安定した数式に支配されるのに対して、ソフトウェアにはそうした数式がないことです。このため、ソフトウェアの品質をはかる客観的な尺度をつくるのはとても困難です。アラン・ケイ氏をはじめ、この数式を探し続けている人たちもいます。 |

# ビルドのライフサイクル

図2 - 9を見てください。これは、図2 - 5からひとつのビルドを抜き出して、そのライフサイクルがどのように始まり、どのように終わるのか、その生涯をUMLの状態遷移図で示したものです。



図2 - 9 ビルドの状態遷移

これは、ビルドが段階を踏みながら、開発チームからQAチームへリリースされ、テストされるプロセスを示しています。ビルドの各状態について説明しましょう。

## 開発中

担当者は開発者です。各開発者は、このビルドで達成しなければならないことや、このビルドをビルドする日 (納期) を意識しながら、ソフトウェアを開発します。この作業のインプットとしては、前回のビルドのソースコード、種々の仕様書 (要求仕様、設計仕様、画面仕様など)、ビルド計画で計画されているこのビルドで実装すべき機能ユースケースのリスト、前回のビルドまでで報告されているバグなどです。バグを修正したら、このバグ報告票の状態を解決済みに変更します。バグを追跡する方法は、5章で詳細にとりあげます。

**前のUCのこと？**

**これはどこで作るの？**

## ビルド中

担当者はビルドマスター (ビルド担当者) です。QAにビルドのリリースが予定されている日の1日くらい前には、ビルドマスターがビルドマシンで候補ビルドをビルドします。ビルドマスターと開発者全員とのビルド中のやりとりを図2 - 10に示します。ビルドマスターは、ビルドの前に開発者に対して「そろそろビルドするけど、大丈夫？」などと告知をして、開発者のお尻を叩きます。このビルドで完了する予定の作業項目を完了できていない開発者は、ビルドマスターを拝み倒してビルドの開始を少しだけ待ってもらうことができます。ビルドマスターを拝んだ開発者は、急いでソースコードの修正を完了し、これを構成管理ツールにコミットします[[12]](#footnote-12)。開発者全員の同意が得られたら、ビルドマスターはビルドを開始します。もしコンパイルエラーなどが出て、ビルドが通らなければ、開発者にこれを修正したコードのコミットを催促し、期日までにビルドがリリースできるようにします。候補ビルドができたら、その旨を開発者に告知します。具体的なビルドの手順については、4章で詳しく紹介します。

**ちらほらありますが、津田さんがどういった状態の開発を目指して、この本を書いているのか疑問になることがあります。**



図2 - 10 ビルド中のやりとり

## QAリリース候補 (開発者によるテスト中)

担当者は開発者です。ビルドマスターからの告知を受けて、QAリリース候補ビルドを自分の環境にインストールし、QAによるテストに耐える品質かどうかをテストします。自分が追加したはずの新しい機能が正しく実装されているか、修正したはずのバグが正しく修正されているかを確認するわけです。この、ビルド直後に行う軽いテストのことをスモークテストといいます。特に、QAチームでのテストが実施できる品質に達しているかどうかを判定するスモークテストのことをBVT (Build Verification Test) といいます。実装したはずの機能がうまく動作しなければ、制限事項としてリリースノートに追記します。また、治したはずのバグが修正できていなければ、［解決済み］としたバグ報告票の状態を［開始］などの状態に戻さなければいけません[[13]](#footnote-13)。

**統合/ビルド・スモークテストはビルドマスターのところでやるんじゃないの？　これもやらなきゃいる意味ないんじゃないの？**

BVTで問題が見つからなければ、この候補ビルドは「候補」ではなくなり、そのままQAにリリースするビルドとします。これをQAチームにリリースしてもらうようにビルドマスターにお願いします。これを受けて、ビルドマスターはQAチームにリリースする作業 (リリースノートの記述・とりまとめ、ビルドをQAチームに見える場所にコピー、QAチームにリリースの告知など。図中の「ビルドをQAチームにリリース」の部分) をします。

テストの結果、予定していたテストが実施できなくなるようなバグが見つかれば、ビルドの状態を［開発中］に戻します。もちろん、予定どおり QA チームにビルドをリリースできるように、必要な修正を施した上で、次の候補ビルドをビルドしなければなりません。

ここまでの作業が終わるまでは、開発中のビルドや候補ビルドが誤って QA チームに漏れてしまわないように注意します。QA チームに見える場所 (ファイルサーバなど) に、リリース前のビルドを置いてはいけないということです。リリース前のビルドとは、不正なビルド番号が付いた不正なビルドのことです。こういったビルドがQAに漏れてしまうと、不正なビルドと正しいビルドと区別できないため、バグを正しく追跡できなくなってしまいます。

ただし、製品の品質が安定してくれば、QAリリース候補という状態を設けず、BVTを省略することもあります。ビルドが完了したら、できあがったビルドをすぐにそのままQAチーム (もしくはプロジェクトチーム全体) にリリースします。このようにできれば、ビルドを自動化できる範囲が広がるので有利です[[14]](#footnote-14)。

## テスト中 (リリース済み; QAによるテスト中)

担当者はQAです。予定していたテストケースを実施して、バグを探します。バグを発見したら、バグ報告票を起票します。また、このバグを再現できる自動テストを作成できれば作成して、次のビルドのリリースに備えます。また、開発者がこのビルドで修正したと報告してきているバグが、本当に受け入れ可能な形で修正されたかどうかを確認します。このビルドを対象としたテスト作業は、次のビルドがリリースされるまで続きます[[15]](#footnote-15)。つまり、QAによるこのビルドのテストと、開発者による次のビルドの開発は並行して行われます。

**テスト完了のクライテリアはどこで決められる？**

**それとも時間がきたら、途中でも終わり？**

ここで示したのは、開発中のビルドをQAチームにリリースするときの手順です。顧客にリリースする候補となるビルドは、さらに候補ビルドという段階がいくつか増えます。候補ビルドという考え方はとても重要で、候補ビルドによるテストがパスすれば、そのビルドをもう少し広い範囲 (例えば開発チームからQAチーム、QAチームから全社、全社から顧客など)、つまりより上の階層での反復という位置付けでリリースすることができるようになります。

また、リリース候補ビルド(RCビルド)を、「これは正式版じゃないから、これを使って問題が生じても責任はもてません。でも、正式版を良くするために、試しに使ってみてね。何か問題を見つけたら教えてね」と伝えた上で、ユーザーにリリースすることもあります。いわば、RCビルドを使ってリリースのリハーサルをするわけです。RCビルドで問題が見つからなければ、これをそのままリリースビルドとします。

ただし、このようなRC版には一定の期間が過ぎたら使えなくなるなどの機能を埋め込んだり、バージョン情報にも「RC版」と書かれたりする場合があります。このようなときは、RC版で問題が見つからなくても、これをそのままリリース版 (正式な製品版) にすることはできません。若干の修正をして、リリース版をビルドし直します。場合によっては、次の候補ビルド (RC2版などと呼ばれます) も、予めユーザーにリリースする計画を立てておくことがあります。

## テスト済み

次のビルドがリリースされるに伴い、このビルドは使われなくなります。ただし、バグの調査のために、ある程度の期間はこの古いビルドも保存しておくと良いでしょう。★3-13節に、古いビルドを使ってバグが導入されたタイミングを調べる方法を説明しましたので参照してください。

# ビルドの種類

表2 - 3に、ビルドの種別を列挙します。

表2 - 3 ビルドの種別

|  |  |  |
| --- | --- | --- |
| 分類 | ビルドの名前 | 訳語 |
| 開発の段階ごとに  順にリリースするもの | Integration build | 統合ビルド |
| First build | 最初のビルド |
| Prototype build | プロトタイプビルド |
| Milestone build | マイルストーンビルド |
| Alpha build | アルファビルド |
| Beta build | ベータビルド |
| Freeze build | 凍結ビルド |
| Candidate build | 候補ビルド |
| Release build | リリースビルド |
| Gold build | ゴールドビルド |
| Patch build | パッチビルド |
| Maintenance build | 保守ビルド |
| 定期的にリリースするもの | Regular build | 定期ビルド |
| Monthly build | 月次ビルド |
| Biweekly build | 隔週ビルド |
| Weekly build | 週次ビルド |
| Daily build | 日次ビルド |
| Morningly build | 毎朝ビルド |
| Noonly build | 毎昼ビルド |
| Nightly build | 毎晩ビルド |
| ビルドの時期を表す もの | Latest build | 最新ビルド |
| Next build | 次のビルド |
| Previous build | 前のビルド |
| Old build | 古いビルド |
| Older build | より古いビルド |
| New build | 新しいビルド |
| Newer build | より新しいビルド |
| ビルドの性質や品質を表すもの | Debug build | デバッグビルド |
| Stable build | 安定したビルド |
| LKG build | 最新の安定したビルド |
| Reproducible build | 再現可能なビルド |
| Testable build | テスト可能なビルド |
| Dogfood build | ドッグフード用ビルド |
| Dogfoodable build | ドッグフード可能なビルド |
| Development build | 開発中のビルド |
| Experimental build | 実験ビルド |
| Broken build | 壊れたビルド |
| 統合ビルドではないもの | Private build | こっそりビルド |
| Buddy build | 相棒ビルド |
| Special build | 特別ビルド |
| Interim build | 仮ビルド |
| ビルドされた場所やリリースの範囲による分類 | Internal build | 内部ビルド |
| External build | 外部ビルド |
| Leaked build | 漏れたビルド |

数が多いので、よく使うものについて解説します。

## 開発の段階ごとに、順にリリースするもの

## Integration build

複数の開発者による成果物を統合し、動作するソフトウェアに構築したものをインテグレーションビルド (統合ビルド) といいます。統合ビルドはQAチームにリリースし、QAチームによるテストの対象とします。このため、統合ビルドは、ビルドマシン上でビルドし、きちんとビルド番号を管理する必要があります。ビルドマシンとは、統合ビルドをビルドするための専用のマシンのことです。ビルドマシンについては、4章でとりあげます。

## First build

ソフトウェア開発プロジェクトにおいて初めてリリースされる最初のビルド、ビルド番号が1の記念すべきビルドです。このビルドでは主に製品の実現可能性 (Feasibility) の調査と検証をゴールとして設定することが多くなります。見てくれが悪かろうが、多少不安定だろうが、あまり気にしません。重要なのは、曲がりなりにもきちんとビルドされていて、取りあえず動作することです。

もちろん、最初のビルドの位置付けが必ずそうなるというわけではありません。場合によっては、ユーザビリティに注目して最初のビルドを開発し、これを早い時期に顧客にリリースすることもあります。この手法をプロトタイピング (試作品=プロトタイプの提示) ということがあります。

私の経験では、開発期間がおよそ1年程度のプロジェクトでは、最初のビルドをリリースできるのは大抵、開発が始まってから2カ月後くらいになることが多いと思います。また、開発期間がおよそ2カ月程度のプロジェクトであれば、最初のビルドをリリースするのは開発が始まってから2週間後くらいになるでしょう。

このほか、最初のビルドには、構築した開発環境とビルド環境の妥当性を検証し、以後の反復をスムーズに行えるようにするという重要な役割もあります。このために、メニューやボタンはあるが、それを押しても何も起こらない、機能をまったくもたないビルドを作ることもあります。これをゼロ機能リリース (ZFR; Zero Feature Release) といいます。

## Milestone build

大きな反復の単位においてリリースされるビルドです。マイルストーンとは、長い道に1里ごと (mile) に置かれる石 (stone) のことです。あなたが旅をしているとき、マイルストーンを見れば、自分がその道程のどのあたりにいるのかわかるというわけです。一般に、マイルストーンという語は、マイルストーンビルドをリリースする日 (時点) を指しますが、前のマイルストーンから次のマイルストーンまでの期間を指してもマイルストーンということがあります。これは、いわゆる反復型開発の反復 (iteration) に対応するものです。これについては7章で扱います。つまり、マイルストーンの重要な成果物が、マイルストーンビルドというわけです。例えば、「ある重要な機能が実装されて利用可能な状態になる」マイルストーンビルドを、開発を通じて何度かリリースすることになります。

マイルストーンビルドを、ユーザーやプロジェクトの利害関係者にリリースすることもあります。QAチームの外に漏れるビルドということです。開発中のビルドを早くユーザーに評価してもらうことで、適切なフィードバックを得ます。また、ユーザーに評価してもらうための最初のマイルストーンビルドをアルファビルド、 その次のより完成度の高いマイルストーンビルドをベータビルドということもあります。

リリース前の最後のマイルストーンビルドで、すべての機能のコードコンプリート (実装完了) を宣言することを勧めます。つまり、次の最後のマイルストーンの期間中は新規機能の追加は行わず、バグの修正だけを行います。これは、安心して安定化 (Stabilization) の作業を行える重要な時期となります。

## Freeze build

製品がいよいよ完成に近づき、その製品のある側面を凍結したビルドをフリーズビルドといいます。ソフトウェア製品を開発するときは、次のような側面を順に凍結します。

* Feature freeze (機能の凍結)  
  仕様を凍結して、これ以上の機能追加はしない、と宣言します。開発の初期の段階でフィーチャーフリーズを行えるのが理想ですが、今日のソフトウェア開発では、初期にユーザーの要求を凍結してしまうと、品質の高いソフトウェアを開発するのは難しいといわれています。
* UI freeze (ユーザーインターフェイスの凍結)  
  もう、画面のレイアウトやメッセージの変更はしない、と宣言するビルドです。UIフリーズが宣言されれば、製品のマニュアルに添付する画面写真の撮影を開始できるようになります。
* Code freeze (ソースコードの凍結)  
  もう、コードを修正することはしない、と宣言するビルドです。  
  コードフリーズの後は、安心してフルリグレッションテストを行えます[[16]](#footnote-16)。このほか、ビルド作業のために一時的にコードの修正 (コミット) を許さないことを指してもコードフリーズ (ブランチフリーズ) ということがあります。この場合、ビルド作業が終わればコードフリーズは解除します。

**コードを直さないのに、なんでリグレッションテストをする必要があるの？**

特に、誰からの要請もなければ、フィーチャーフリーズやUIフリーズを意識する必要はありませんが、コードフリーズは意識した方が良いでしょう。コードフリーズは、いつフルリグレッションテストを開始できるかという重要なタイミングを意味するからです。

**こっちの方は分かるけど**

このほか、機能仕様書を書き終えてレビューを通した段階をスペックコンプリート、テスト計画書を書き終えてレビューを通した段階をテストプランコンプリート、凍結した仕様の実装を完了することをコードコンプリートといいます。ただし、組織によっては、これらの用語を違う意味で使うことがあるので注意が必要です。例えば、最初のビルドをつくるためにコードフリーズ (ブランチフリーズ) したタイミングを指してコードコンプリートとよぶ組織もあります。また、スペックコンプリートを指してフィーチャーフリーズということもあれば、コードコンプリートを指してフィーチャーフリーズということもあります。このような用語については、プロジェクトのキックオフ時にその意味をチームで共有しておきましょう。

## Candidate build

ビルドです。候補ビルドにもいろいろなものがあります。QAにリリースするウィークリービルドの候補、各マイルストーンでの成果物となるマイルストーンビルドの候補などです。また、最終的にユーザーにリリースするリリースビルドの候補をRC ビルド (Release Candidate build) といいます。

普通、「候補」といえば選挙を思い出しますね。候補者は、投票による試験を受け、合格した候補者が晴れて正当な何某になれるわけです。「候補ビルド」も同じです。候補ビルドは、本当にそれをリリースしても大丈夫かどうかを試験され、合格すれば晴れてそのままリリースビルドになれます。例えば、QAにリリースするビルドの候補ができたら、開発者はこのビルドを使って、自分が行った実装や修正が正しく行われているかどうかを確認します。正しく行われていなければ、それを制限事項としてリリースノートに記述したり、修正済みと報告したバグ報告票の状態を元に戻したりしてから、そのビルドをQAにリリースします。修正されていなければテストが実施できないような、致命的なバグが見つかれば、この候補ビルドは落選です。ソースコードを修正して新しい候補ビルドを作成し、もう一度最初からテストをやり直します。特に、修正しなければユーザーにリリースできないバグのことをショーストッパーバグ (Showstopper bug)、もしくは単にショーストッパー[[17]](#footnote-17)といいます。あるいは、ブロッカー (Blocker) ということもあります。

どのような位置付けのビルドの候補かによって、この確認のためのテストに求められる丁寧さも異なります。例えば、RCビルドは、フルリグレッションテストの対象となります。リグレッションテストとは、いままでに書かれたテストケースや、いままでに報告されて修正した (はずの) バグの再現手順をすべてテストし直すことです。もしフルリグレッションテストでショーストッパーを発見したら、これを修正した上で次のキャンディデートビルドをビルドし、フルリグレッションテストを最初からやり直さなければなりません。リグレッションテストについては、5章で詳細に取り上げます。



図2 - 13 ショーストッパーバグはすべて  
取り除かないとリリースはできない

## Release build

最終的にユーザーにリリースするビルドをリリースビルドといいます。このビルドを指して最終ビルド (Final build)、ゴールドビルド (Gold build)[[18]](#footnote-18)、GA (General AvailabilityもしくはGeneral Acceptance) などということもあります。

リリースビルドの候補となるビルドをRCビルドといいます。また、GC (Gold Candidate) ということもあります。GCがリグレッションテストや受け入れテストにパスしたら、RTM (Release To Market) を宣言し、晴れてゴールドビルドのリリースとなります。ソフトウェア開発プロジェクトでは、このビルドを手に入れるために作業をしてきたのであり、ソフトウェアを開発してきた企業にとって、このビルドはまさに財産 (gold) そのものです。

上記以外にも、デバッグビルドに対して、デバッグ情報を含まないビルドのことをリリースビルドということがあります。例えば、ユーザーやQAにリリースしないビルド (プライベートビルド) であっても、それがデバッグ情報を含んでいなければ、それはリリースビルドです。

## 定期的にリリースするもの

## Weekly build

週次ビルドです。ソフトウェアの規模にもよりますが、build#1をリリースしてから1～2週間後にbuild#2を、さらにその1～2週間後にbuild#3をリリースできるようになるでしょう。2～4番目のビルドから、ビルドを1週間おきにQAチームにリリースし、実装された機能からテストしていきます。QAによるテストは開発と並行して進めます。この、1週間おきに定期的にリリースするビルドをウィークリービルドといいます。また、このような反復のアプローチを指して「ウィークリービルド」ということもあります。このほか、隔週ごとにリリースするビルドはBiweekly buildといいます。

## Daily build

日次ビルドです[[19]](#footnote-19)。ビルドを頻繁にQAリリースし過ぎてしまうと、QAがテストをし切れなくなり、ビルドの消化不良を起こします。では、なぜデイリービルドを作るのかといえば、テストを自動化できた場合にはそれが非常に有用だからです。ビルドと自動化されたテストの実行には数時間かかることもよくあるため、自動化されたテストを実行するためのビルドは毎晩行うことがあります。この毎晩のビルドをナイトリービルド (Nightly build) といいます。誰もコードをコミットしないような深夜の時刻を狙って、ビルドスクリプトを定期的に自動実行するように設定します。Windowsならタスクスケジューラを、Unixの環境ならcronを使えば、簡単にナイトリービルドを実現できます。

ビルド時間がそれほど長くないとか、開発者が夜中にも作業をしてコミットするようなときは、QAにリリースするビルドを早朝に開始することもあります。このような毎朝のビルドをMorningly buildといいます。

## Latest build

最新のビルドです。やはり、文脈によって最新のビルドというのがどのビルドを指すのか違ってくるので、注意が必要です。単にレイテストのビルドといえば大概最新のウィークリービルドのことですが、デイリービルドを立てているプロジェクトでは昨日のビルドのことでしょうし、最新の安定したビルドはどれだっけ？ ということもあるでしょう。この最新の安定したビルドをLKGビルド (Last Known Good build) といいます。バグがあるかどうかをテストするには、最新のビルドで行うのが一番ですが、リリースが多すぎるとQAチームはビルドをテストし切れなくなってしまいます。このため、開発チーム内でのスモークテストはデイリービルドで、QAチームでのリグレッションテストはウィークリービルドで、といったこともよく行われます。

また、Next build (次のビルド)、Previous build (前のビルド) などといった語も使われます。このほかの定期的にリリースされるビルドにはBroken buildがあります[[20]](#footnote-20)。

## ビルドの性質や品質を表すもの

## Debug build

デバッグ情報やデバッグ用のコードを含んだビルドです。デバッグ情報とは、デバッグオプションをつけてコンパイル・ビルドしたときに生成される情報のことです。この情報を含んだデバッグビルドを使うと、デバッガ上で実行でき、プログラムを1行ずつ実行したり、プログラム中の変数の値を参照したりできます。コンパイラが付加する情報のほか、開発者がデバッグ用のコードをプログラムに埋め込むこともあります。デバッグビルドは、リリースビルドよりも実行速度が遅く、ビルドのファイルサイズや実行時に消費するメモリサイズも大きくなります。

開発の初期～中期には、デバッグビルドを統合ビルドとしてQAにリリースすることもよくあります。この場合、どのビルドからデバッグ情報を取り外すかは、慎重な検討が必要です。

# コラム C/C++言語のプリプロセッサとデバッグビルド

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **コラム C/C++言語のプリプロセッサとデバッグビルド**  本コラムでは、デバッグビルドとリリースビルドの違いについて、もう少し詳しく説明します。  C/C++ 言語には、プリプロセッサという機能があります。これは、ソースプログラムファイルをコンパイルする前に、(C/C++の文法に関係なく) 開発者が決めたルールによってこのファイルを変形・加工できてしまうという無茶苦茶なすごい (危険な) 機能です。これを利用すると、デバッグ用のコード (ログ出力コードなど) を物理的にソースファイルから除去できます。例えば、次のようにします。   |  | | --- | | // program.c  …  Some\_function1();  #if !defined(NDEBUG)  // ここにデバッグ用のコードを書く  Some\_debug\_function();  #endif    Some\_function2();  … |   C:\idd\doc\debug.png   |  |  |  | | --- | --- | --- | | // program.c  …  Some\_function1();  Some\_function2();  … |  | // program.c  …  Some\_function1();  // ここにデバッグ用のコードを書く  Some\_debug\_function();    Some\_function2();  … |   このようにして、デバッグビルド用とリリースビルド用に、異なるプログラムファイルをコンパイル直前に生成できます。このほか、プリプロセッサを使えば、ある条件に従って特定の文字列を別の文字列に一括置換することもできます。  プリプロセッサはなんでもありの強力な機能なので、とても危険です。プリプロセッサを使えば、プログラムを書き換え、その挙動を完全に変えてしまえるからです。なぜ、このようなとんでもない機能があるかというと、主に2つの用途が意図されています。   1. 複数のプラットフォームに対応する   C/C++言語は、各プラットフォーム (OS) に最適の性能を引き出すことを主眼にデザインされています。そのため、複数のプラットフォーム上で動作するプログラムを書くときは、プラットフォーム間の違いを意識してプログラムを書かねばならないときがあります。例えば、次のようにします。   |  | | --- | | …  #if defined(WINDOWS)  // ここにWindows用のコードを書く  #elif defined(UNIX)  // ここにUNIX用のコードを書く  #else  #error You must define WINDOWS or UNIX  #endif  … |   これにより、ひとつのソースファイルを複数のプラットフォームに対応させることができます[[21]](#footnote-21)。   1. デバッグ情報を埋め込む   先述した方法のほか、assert文を使ってデバッグ情報をプログラムに埋め込むことができます。assert文は、次のように引数に真偽値を渡して使います。   |  | | --- | | …  int some\_function(int\* pData)  {  assert(pData != 0);  …  }  … |   assert文も、NDEBUGを定義してコンパイルすると、ソースファイルから完全に除去される仕組みになっているので、リリースビルドには含まれません。  assertとは「表明する」という意味です。上記では、「変数pDataは、0であるはずがない」と表明しています。いわば、開発者が記述したコメントのようなものです。しかしこのコメントは、デバッグビルドの実行時には実際に評価・検査されます。つまり、assert文はデバッグビルドに直接埋め込まれるテストコードというわけです。もし、引数の値が真でなかった (上記ではpData変数の値がゼロだった) ら、表明に失敗したことになります。このとき、どのソースファイルのどの行で表明に失敗したのかを画面に表示して、プログラムは異常終了します。プログラムが、プログラマが表明 (意図) した通りに動いていないということは、何らかのバグが入り込んでいることを示すからです。バグの症状が現れた箇所と、実際にそのバグの原因となるプログラム上の記述は遠く離れることがあります。しかし、assert文はより早い段階でバグを検出できるため、非常に有効です。  ずいぶん前の話です。筆者がとあるプログラムを読んでいたとき、大事そうな部分に大事そうなコメントが英語で書いてありました。何だろうと思って読んでみると、「今日はおなかが空いたからもう帰る」と書いてあったのです。英語圏の人は、プログラミング言語と自然言語である英語が似ているために、こんなコメントを書く人がいるのかもしれませんが、それ以来、私はコメントって何だかなぁ、と思うようになりました。もちろん、コメントが全くないプログラムも信用できませんが、冗長で無駄なコメントだらけのプログラムもいただけません。無駄なコメントは重要なコメントを隠し、コメントを保守する手間を大きくします。  しかし、assert文だらけのプログラムはとても信頼感があります。リリースビルドでは、assert文は完全にソースファイルから除去されるため、assert文がリリースビルドの処理速度を遅くしたり、コードサイズや使用メモリサイズを無駄に大きくしたりすることはありません。  Java言語には、このようなプリプロセッサ機能はありません。Javaはもともとマルチプラットフォームに対応しているので、危険なプリプロセッサの機能を搭載する必要はありませんでした。Javaは、C/C++の文法をお手本にした部分が多いのですが、プリプロセッサの機能は始めから意図的に省かれたのです。しかし、assert文が使えないのは不便だということで、JDK1.4からは、Javaでもassert文が使えるようになりました。もちろん、Javaのassert文は、プリプロセッサ以外の方法で実現されています。assert文のオン/オフは、C/C++ではコンパイル時に指定しますが、Javaではプログラムの実行時に指定します。  assert文を書くときに注意したいのは、副作用をもつ式は評価してはいけないことです。もし副作用があると、デバッグビルドとリリースビルドでプログラムの挙動が変わってしまうからです。次の例を見てください。   |  | | --- | | …  int i = 0;  assert(++i == 1); // iに1を加えたら1である  printf("%d", i);  … |   このプログラムはデバッグ版では1を、リリース版では0を表示します。上記のassert文は、その表明自体は正しいのですが、iに1を加えるという副作用をもっているからです。副作用をもつデバッグコードにより、デバッグ版は問題なく動くのに、リリース版ではバグが出てうまく動かない、ということもあります。このため、C/C++言語で書かれたプログラムは、デバッグビルドとリリースビルドの両方を使ってテストすべきです。 |

## Stable build

安定したビルドです。安定とは、バグが少なく安心して使える状態のことです。不安定とは、ときどき動いたり動かなかったりするような状態をいいます。

ソフトウェアの品質が安定するまでには、時間がかかるものです。そこで、オープンソースの製品などでは、レイテストビルド (最新ビルド) のほかにステーブルビルド (安定ビルド) を公開していることがよくあります。早く新しい機能を試してみたい、多少不安定でも構わないと願うユーザーのためにレイテストビルドを公開します。これと同時に、新しい機能は取りあえず必要ないけど、安定したバージョンで安心して作業したいと望むユーザーのために、ちょっと古いが安定しているステーブルビルドをダウンロード可能な状態のまま公開しておくわけです。

すべてのビルドが安定して動作すれば幸せですが、初めからステーブルビルドを作ろうと思ってもそうはいきません。そのビルドをリリースしてからしばらく時間がたたなければテストした後でなければ、そのビルドが安定したビルドである、と確信を持つことはできないからです。 ただし、十分な数のテストが自動化されていれば、ビルドと自動テスト完了後に「比較的安定したビルドができたみたい」と表明することはできるでしょうできます。

## Reproducible build

再現可能なビルドです。2つの意味があります。どちらの意味で使われるのかは、文脈により異なります。

### ビルド自身が再現可能である

当該のビルドを失くしても、それをビルドしたときのソースコード一式を取り出してリビルドすることにより、全く同じビルドを再現できるとき、このビルドは再現可能であるといいます。この意味では、すべての統合ビルドは再現可能であるべきです。

### あるバグを再現できるビルドである

あるバグが報告されたとき、そのバグを再現できるビルドのことです。あるバグを、ビルド#Nで再現できないが、ビルド#N+1で再現できるとき、このバグはビルド#N+1から導入されたことになります。これは、バグの原因を調査する上で非常に重要な情報となります。

## Dogfoodable build

ドッグフーディングが可能な品質に達したビルドです。ドッグフードとは、プロジェクトチーム全員、もしくは社内全体にリリースして、日常的に使ってもらい、フィードバックを得ることを意図したビルドテスト手法のことです。また、そのためのビルドをドッグフードビルド、あるいは単にドッグフードといいます。バグが多く残っているビルドでは、さすがにドッグフードできません。ドッグフードが可能な種類のソフトウェアでは、ドッグフード可能な品質に到達することが大きなマイルストーンのひとつとなります。

# コラム ドッグフードとは

|  |  |
| --- | --- |
| **コラム ドッグフードとは**  ドッグフードとは、開発中のソフトウェアをテストするための手法です。ドッグフードを開発・販売する企業の社長が、株主総会のときに株主の前でドッグフードを食べてみせた、という故事に由来します。  みなさんは、ドッグフードを食べたことはありますか？ 実は私もありませんが、きっと不味いと思います。でも、きっと食べてみても死んだりはしないはずです。だから、良いドッグフードを開発したいなら、それを愛犬に食べさせる前に自分で食べてみるべきです。  同じように、現在開発中のソフトウェアは、きっと使いにくいと思います。でも、きっと使ってみても大事なデータがなくなったりはしないはずです。だから、良いソフトウェアを開発したいなら、それをユーザーに使ってもらう前に自分で使ってみるべきです。  このように、現在開発中のソフトウェアをドッグフードに見立て、これをソフトウェア開発以外の場面でも普段から使うようにします。これにより、日常の業務にテストの作業を兼ね、バグを早く発見し、ユーザビリティのテストも行います。このようなテスト手法を指してドッグフードとかドッグフーディングなどといいます。これは、エリック・レイモンド氏の論文「伽藍とバザール」にある目玉たくさん効果を狙ったものといえます。目玉がたくさんあれば、深刻なバグを発見できずに見逃してしまうリスクは減少します。  例えば、コンパイラのソースコードをそのコンパイラでビルドしたり、バグ追跡システムのバグをそのバグ追跡システムに登録して管理したり、構成管理ツールのソースコードをその構成管理ツールで管理したりなどは、すべてドッグフードといえます (ドッグフードに厳密な定義があるわけではありませんが、開発中のではなく、安定した前のバージョンのソフトウェアを使うなら、これはドッグフードとは言えないかもしれませんね)。  包括的なテストを漏れなく、また繰り返し実施し続けるというのは非常に難しいことです。そこで、ある程度の品質に達した (dogfoodable となった) 開発中のソフトウェアをチームの内外に公開し、期間を区切ってドッグフード・プログラムを実施します。この間に一番バグを発見した人に賞を贈呈するなどの工夫をすることもあります。  ドッグフードは、早くバグを見つけるというだけでなく、開発者の開発に対するモチベーションを高める効果があります。自分が日常的に使うソフトウェアは、より便利なものにしたくなりますね。また、開発中のソフトウェアにより多くの人 (ドッグフード・プログラムに参加している人) に迷惑をかけたくない、だから早くバグを修正しなくちゃ、という動機も働きます。ただし、残念ながらドッグフーディングの手法が困難なドメインのソフトウェアもあります。私は株のオンライン・トレーディング・システムの開発に携わったことがありますが、このときはさすがにドッグフードの手法を持ち込むことはできませんでした。  次のような話を聞いたことがあります。あるメールサーバを開発している企業は、以前は自社のインフラに自社製品を使っていました。ところが、時の社長が「メールを携帯電話でも読みたい」と言い出し、その機能を持たなかった自社製品の使用をやめて、他社のメールサーバの導入を決定しました。鶴の一声です。もちろん、開発チームは大反対しましたが、社内のメールサーバは他社製品のものにリプレースされてしまいました。その後、この企業は自社製品のシェアを大きく落としたということです。ソフトウェア開発をしている企業では、その経営者もソフトウェアの性質と開発手法をよく理解しておかねばならないという好例 (悪例?) ですね。  毎日の食卓にドッグフードをのせて、食べる習慣をつけましょう！それが、あなたのソフトウェアを美味しくする秘訣です。  C:\idd\doc\dogfood.png  図2 - 14 ドッグフードを食べてみよう   |  | | --- | | To: 全社員  From: XXプロジェクトテストリード  Date: 2010/09/25  Subject: もうすぐ、製品XXのドッグフード・プログラムが始まります！  みなさま  現在、XXプロジェクトで開発している製品が  ドッグフード可能な品質に達しました。  つきましては、ドッグフード・プログラムへの参加を募ります。  あなたの参加とフィードバックが、本製品の成功に不可欠です。  ぜひ、本製品の品質向上にご協力ください！  - 期間  2010/10/01 - 2010/10/31 まで  - 製品の場所  \\fileserver\shared\product\<build\_num> にあります。  適宜、新しいビルドがリリースされます。  なるべく最新のものを使用して下さい。  - インストール方法  古いバージョンをアンインストール後、ディレクトリ中にある  product.msi を実行して下さい。  - 使用方法  製品に添付されたヘルプファイルを参照して下さい。  - バグの報告方法  http://xxx で、バグを報告できます。これに際しては  http://xxx/help.html にあるガイドラインをよくお読みください。  - 賞品  バグを多く発見した上位3名までに、温泉ペア宿泊券を贈呈します！  - ご注意  本プログラムに関するすべての情報は社外秘となりますのでご注意下さい。  万一、本ソフトウェアにより重要なデータが失われても、責任は負いかねます。  顧客データを扱うような業務には、使用しないで下さい。  以上 | |

## 統合ビルドではないもの

## Private build

こっそりビルドです。開発者が自分のサンドボックス[[22]](#footnote-22)でビルドしたビルドをプライベートビルドといいます。統合ビルドが制限事項付き (ある機能がまだ使えないのでテストできないなど) でQAにリリースされたが、QAがその制限事項が解除されるビルドのリリースをどうしても待てないとき、開発者にお願いしてその問題が修正されたビルドを先に作ってもらうことがあります。逆に、開発者がテストを依頼するために自分のサンドボックスでビルドしたものを、こっそり特定のQAに渡すことがあります。このビルド番号が管理されないこっそりビルドのことをプライベートビルドといいます。

このビルドはビルド番号が正しく管理されたものではありません。また、一般に、プライベートビルドは再現可能ではありません (後で同じビルドを作り直すことはできません)。このため、意図した人以外の人の手に渡らないように気を付けなければなりません[[23]](#footnote-23)。特定のQAにだけリリースすべきビルドであって、QAチーム全体にはリリースすべきではないということです。そうしないと、ビルド番号が混乱してバグ報告票の品質に問題が出てしまいます。プライベートビルドをビルドするときは、必要に応じて、このバージョン情報にはビルド番号を表示しないようにし、代わりに「これはプライベートビルドです」と表示するようにしておけば安全ですが、ちょっとした手間がかかります。

このビルドで見つかったバグについては、バグ報告票を記述するべきではありません。ビルド番号が管理されておらず、ソースコードのタグ付けもされていないので、ほかの開発者やQAがこのバグを再現できないかもしれないからです。もしプライベートビルドでバグを見つけたら、その発見者がこれをビルドした開発者に口頭 (もしくはメール) で伝えるようにするか、きちんとした統合ビルドで同じバグを再現させたうえで、バグ報告票を記述するようにします。

## Buddy build

相棒ビルドです。リポジトリに変更したソースファイルをコミットする前に、相棒 (同僚) の開発者にそのファイルを渡して作ってもらったビルドをバディビルドといいます。バディビルドも、プライベートビルドの一種です。無事にバディビルドが通り、スモークテストでも問題が出なければ、このファイルをリポジトリにコミットします。★3-9節にバディビルドに関するより詳細な説明を載せましたので、参照してください。

このほか、Special build (特別なビルド) や、Interim build (仮ビルド) などといった言葉を使うこともありますが、これらの意味は文脈によって異なります。雪の世界に住むエスキモーは、雪を表現する言葉を数十種類も持っているそうですが、ビルド中心の生活をする私たちは、さまざまな性格をもつビルドを正しく呼び分けたいものですね。

# コラム 動詞としてのビルドの種別

|  |
| --- |
| **コラム 動詞としてのビルドの種別**  本文中では、名詞としてのビルドの種類を紹介しました。しかし、動詞としての「ビルド」にも、いくつか種類があります。   * **Full build (完全ビルド)** – すべてのファイルをコンパイルして、ビルドします。 * **Clean build (クリーンビルド)** - 前回のビルドによる中間生成ファイルをきれいに削除してからビルドします。フルビルドと同じ意味と考えて良いでしょう。 * **Rebuild (再ビルド)**  – もういちど、ビルドをやり直します。 * **Incremental build (インクリメンタルビルド)** - 修正したプログラムファイルだけをコンパイルし、前回のビルドから変わっていないプログラムファイルは前回のビルドで生成された中間ファイルを使ってビルドします。これにより、ビルド時間を劇的に短くします。 * **Partial build (部分ビルド)** - 部分的にビルドします。 * **Distributed build (分散ビルド)** - 複数のマシンを使って並行してビルドを走らせることで、見かけ上のビルド時間を短くします。   フルビルドとインクリメンタルビルドについては、4章で取り扱います。 |

# リリースノートの添付

**リリース手順については言わなくてよいの？**

リリースノートは、各ビルドのリリースに際して添付する文書です。次のような項目を記入します。

* 製品名
* ビルド番号 (バージョン情報)
* 構成管理ツールのブランチとタグ名
* リリース場所 (ファイルサーバ名やディレクトリの位置など)
* インストール手順
* 変更点、新機能
* 修正したバグの一覧
* 制限事項
* このビルドのソースにつけたタグ名
* このビルドのソースのチェンジセット番号 (コミット番号)
* その他、このビルドに含まれるモジュールやファイルの一覧など

リリースノートは、リリース先の人に読んでもらうことを意図した文書ですから、どこにリリースするかで記述すべき項目も異なります。上記はQAにビルドをリリースするときの例です。しかし、ユーザーにリリースするときには、構成管理ツールに関する情報 (タグ名やチェンジセット番号など) はリリースノートには記載すべきではありません。また、その場合はリリースノートをビルドに同梱するのが普通ですから、リリース場所を記載する必要もありません。

頻繁にビルドをリリースするときには、各ビルドにリリースノートを添付するのが煩雑なこともあります。そのようなときは、ビルドマスターからプロジェクトチームへのビルド完了通知メールを、リリースノートの代替とします。このような小さなリリースノートを、ビルドノートということもあります。

バグ追跡システムやテストケース管理ツールによっては、リリースノート (ビルドノート) を登録し、各ビルドの状態を追跡できるものがあります。

表2 - 4 ビルドノートの例

|  |
| --- |
| To: 開発プロジェクトチーム  From: ビルドマスター  Date: 2010/09/22  Subject: ビルド#8をリリースしました  みなさま  製品XXのビルド#8をリリースしました。  **ビルドの場所:** [\\fileserver\projectXX\_drop\build\008](file:///\\fileserver\projectXX_drop\build\008)  **インストール方法:**  …  **タグ名:** XX\_20100922\_build#008  **チェンジセット番号:** 538  **このビルドで修正したバグ:**  BTSをご参照ください。  **制限事項:**  自動化されたBVTにより、xxの機能が  リグレッションしたことがわかっています。  このため、yyの機能が使えなくなっています。  すでにBTSにファイル済みです。(BTS#0144)  よろしくお願いします。 |

現在は、一日に何度もビルドするアプローチが使われます[[24]](#footnote-24)。これらのビルド全てに対してビルドノートを添付するのは面倒ですし、添付したところで役には立たないでしょう現実的ではありません。この場合は、ビルドシステムからのビルド完了通知メールをビルドノートとすれば十分ですの代替とします。

# ビルドの配置 (インストール)

各ビルドには、インストーラ、もしくはインストール手順が添付されているべきです。というのは、ユーザーが正しくソフトウェアをインストール[[25]](#footnote-25)できるかどうかも、QAによるテストの対象とすべきだからです。また、正しい手順でインストールされたソフトウェアをテストするのでなければ、品質の悪いバグ報告票が記述されてしまうことにもなります。

なるべく早い時期のビルドから、インストーラもしくはインストール手順書を準備しましょう。また、ビルドのリリースを繰り返すごとに、各ビルドのインストール手順は変化していきますから、リリースノートには常に最新の正しいインストール手順書を添付します。ソフトウェアをインストールする手順を早期に確立することで、ウォーターフォール型開発における「統合テストの悪夢」から逃れることができます。

また、QAは繰り返しビルドをインストールしますから、アンインストールの手順も必要です。次のビルドをインストールする前に、前回のビルドを痕跡を残さずきれいにアンインストールする必要があるからです。

# まとめ

本章では、ビルドの意味を説明し、その種類を紹介しました。また、各ビルドにおいて、そのリリース範囲が決められているべきことや、ビルド番号が付加されているべきこと、リリースノートやインストール手順が添付されているべきことなど、ビルドの要件についても説明しました。

ひと言にビルドといっても、理解しておくべきこと、検討すべきことは非常に多くあります。しかし、ビルドのことを考えるだけで、ソフトウェア開発の計画や見積もりなどの大事な懸案事項を広くカバーできるのですから、便利なものです。ビルドの性質をよく理解することが、上手なソフトウェア開発の第一歩といえます。

# 参考文献

## Software release life cycle

<http://en.wikipedia.org/wiki/Development_stage>

## Feature Complete

<http://en.wikipedia.org/wiki/Feature_complete>

## Freeze

<http://en.wikipedia.org/wiki/Freeze_(software_engineering)>

## Golden master

<http://en.wikipedia.org/wiki/Golden_master>

## Eating your own dog food

<http://en.wikipedia.org/wiki/Eating_one%27s_own_dog_food>

## 伽藍とバザール

Eric S. Raymond (著), 山形浩生 (訳)

http://cruel.org/freeware/cathedral.html

## 反復型開発のエコノミクス

Walker Royce (著), 藤井 拓 (監訳)

<http://www.amazon.co.jp/dp/4894715449>

## 闘うプログラマー

Pascal G. Zachary (著), 山岡 洋一 (訳)

<http://www.amazon.co.jp/dp/4822247570>

## 大聖堂

Ken Follett (著), 矢野 浩三郎 (訳)

<http://www.amazon.co.jp/dp/4797332565>

## Eskimos and Snow

<http://tribes.tribe.net/triviarocks/thread/c3ae77dc-0691-4649-8348-4ec6b5d5c2eb>

## 18 Eskimo words for snow

<http://home.egge.net/~savory/snow.htm>

## "SNOW" WORDS OR NAMES

<http://www.lowchensaustralia.com/names/snow.htm>

1. コンパイルとは、開発者が読めるプログラミング言語で書かれたプログラムを、コンピュータが理解できる形式に翻訳することです。この作業は、「コンパイラ」というソフトウェアで自動化されます。 [↑](#footnote-ref-1)
2. 実際のビルド作業には、コンパイルのほかにも多くの作業が含まれます。これについては4章で詳しく扱います。 [↑](#footnote-ref-2)
3. XPとスクラムは、アジャイルなソフトウェア開発手法のひとつです。スプリントとは、スクラムにおける反復の単位で、1ヶ月間とするのが一般的です。UPとはUnified Process (統一プロセス) の略で、反復型プロセスのフレームワークです。UPのフェーズは、2週間から6週間程度の複数の反復から構成されます。これらについては7章で紹介します。 [↑](#footnote-ref-3)
4. 状態遷移図は、開始状態を黒丸記号で、終了状態をブルズアイ (牛の目玉) 記号で表記します。 [↑](#footnote-ref-4)
5. これをさらに進めたものとして、書籍「初めてのアジャイル開発」には、要求チームがN+1回目の反復のための要求を定義し、開発チームがN回目の反復のための開発をし、テストチームがN-1回目の反復の成果物をテストすることを同時並行して行うイテレーションの作業のパイプラインが紹介されています。 [↑](#footnote-ref-5)
6. GUIとは、Graphical User Interfaceの略で、グイと発音します。 [↑](#footnote-ref-6)
7. HとAは、それぞれHelpとAboutを表します。 [↑](#footnote-ref-7)
8. この一例を、★3-17節で取り上げます。 [↑](#footnote-ref-8)
9. WYSIWYGとはWhat You See Is What You Get (見たものを得る) の略で、ウィジウィグと発音します。画面表示と印刷結果が同じになるという意味です。 [↑](#footnote-ref-9)
10. #はナンバーと読み、番号という意味です。また、t.b.d.はto be describedもしくはto be determinedという意味で、今は書いていないが、そのうち書かれるべきという意味です。このほか、n/aという表記が使われることがあります。これはnot applicableもしくはnot availableの略で、書く必要がないとか、利用不可という意味です。 [↑](#footnote-ref-10)
11. CCはコードコンプリート (実装完了) の略です。コードコンプリートした後も、まだ安定化 (バグ出しと修正) の作業が残っています。 [↑](#footnote-ref-11)
12. コミットの手順は3章で説明します。 [↑](#footnote-ref-12)
13. バグ報告書の状態については、5章で扱います。 [↑](#footnote-ref-13)
14. ビルドの自動化については、4章で扱います。 [↑](#footnote-ref-14)
15. テスト完了のには、「次のビルドがリリースされる」のほか、「実施を予定したテストケースをすべて完了する」を設定することもあります。あなたが開発するソフトウェアに適したクライテリアを探しましょう。 [↑](#footnote-ref-15)
16. テストの結果、ショーストッパーバグを発見したら、もちろんコードフリーズを宣言した後でもコードを修正しなければなりません。リグレッションとリグレッションテストについては5章であつかいます。 [↑](#footnote-ref-16)
17. 邦訳「闘うプログラマー」の原著のタイトルが「Showstopper!」です。このノンフィクションには、Windows NTを毎週水曜日にビルドしながら、ドッグフードを使って開発した話が迫力のある筆致で描写されています。「ビルド」をどう扱うべきか、もっと深く知りたい方には、読んでみることを強く勧めます。 [↑](#footnote-ref-17)
18. Gold buildには、CDやDVDに焼くとか、Webで公開するためのイメージ、というニュアンスがあるようです。 [↑](#footnote-ref-18)
19. 広辞苑第5版によれば、日次には「にちじ」「ひつぎ」「ひなみ」などと複数の読み方があり、どれも「毎日の」という意味です。「ひついで」【日次】と読むと、日の都合とか日柄、日取りのこと、「ひなみき」【日次記】とは日記のことです。 [↑](#footnote-ref-19)
20. もちろん冗談です。Broken buildを定期的にリリースしないようにがんばりましょう。 [↑](#footnote-ref-20)
21. #errorは、意図的にコンパイルエラーを発生させるためのプリプロセッサ命令です。 [↑](#footnote-ref-21)
22. サンドボックスとは、開発者用の開発環境 (開発用マシン) のことです。サンドボックスについては、3章でとりあげます。 [↑](#footnote-ref-22)
23. このように、意図しない人にビルドをリリースしてしまうことを、「ビルドが漏れる」とか「リリースが漏れる」といいます。 [↑](#footnote-ref-23)
24. このアプローチを継続的な統合といいます。詳細は★4-12節を参照してください。 [↑](#footnote-ref-24)
25. ソフトウェアの種類によっては、ビルドが動作するように配置することを「インストール」とはいいません。例えば、WebアプリケーションをWebサーバ上にいれるのは「デプロイ (配置)」といいます。また、携帯電話に新しいビルドを入れるとき、このビルドイメージを携帯電話端末に「フラッシュ」する、などということがあります。それぞれ固有の手順が必要なのは「インストール」と同じです。 [↑](#footnote-ref-25)