2015.9.9

良い道具と良い職人

契約はアジャイル開発プロセスに追いついたのか Vol.2

高橋 雅宏

今まで、アジャイル開発プロセスで開発できる契約書とは、どのようなものが良いのかについて考えてきました。契約の当事者であるユーザとベンダのうち、「知的創造作業」を担うベンダにフォーカスして、話を進めていこうと思います。

また、自分自身がかつて、ソフトウェア開発者だった時の経験から、（良い道具を良く使いこなせる）優れたソフトウェア開発者についての思いを述べてゆきます。

それと同時に、自分自身が考え続けてきた「契約書の締結」という行為の当事者について、無意識のうちに前提にしてきたベンダの条件があったことに気付いたので、その事についても明らかにしたいと思います。

また、最近になって、契約書そのものも、「道具」として位置づけられる、と感じており、ある条件の下では、どのような取引があったのか（どのくらいの期間で、どのくらいの取引金額の、どのような種類の取引）を証明するだけの簡単なもので十分な気がしてきましたので、そのことについても触れたいと思います。

はじめに

Vol.1 では、「契約はアジャイル開発プロセスに追いついたのか」というテーマで書きました。ここ３０～４０年の間に、ソフトウェア開発のスピードや開発環境、開発プロセスが大きく変わってきた中で、契約だけが取り残されていると感じてきた思いを書きました。

今回 Vol.2 でも「契約」という立場から見てゆきたいと思います。

昔の契約書（今の契約書もそうですが）は、『かたっ！』（Vol.1 本橋正成さんの「ゆるっと行こう～ゆる思考のすすめ～」参照）という状態だったのではなかったかと思います。今のソフトウェア開発環境（状況により変化する開発要件や品質など）においては、契約が「かたっ！」のままでは、知的創造作業を行うベンダが、フットワーク良く開発できないことが殆どですので、開発のスピードを出せない、と思います。「ゆるっ」という考え方で、角を丸くしたほうが、開発のスピードを上げられるのではないかと思います。開発要件の角を丸くする事で、流線型に近くなり、開発スピードが上げられるようになると思います。この事は、詳しく後述します。

また、アジャイルプロセスは、「道具」だという話をしながら、新しくソフトウェア開発の世界に入ったばかりの、まだ開発経験の浅いソフトウェア開発技術者が、いきなり開発スピードを求められたときに、間違った開発スタイルや、考え方で成長してしまわないように、警鐘を鳴らそうと考え、この Vol.2 を書こうと思います。

ソフトウェア開発環境今昔

３０年以上前、私がソフトウェア開発者だった頃、開発環境が非常にしかも急激に進歩したと思います。そのときに、個人的には、少し疑問を持ちました。Vol.1 でも書きましたが、１９７０年代前半では、コンパイルを依頼してその結果を手にするまでに半日以上必要だった開発環境が、１９７０年代後半には、ホストコンピュータにつないだ端末から、ソースの修正、コンパイル、実行が、ほぼ待ち時間無しに依頼でき、しかもその結果もすぐに手に入れられるようになりました。この結果、コンパイルやソフトウェアの実行結果が、自分の手元に戻ってくるまでの待ち時間は大幅に短

縮されました。当時は、「夢のような開発環境になった」と思ったものです。

ここでは、開発環境の詳細な変遷については触れませんが、現在では、さらに開発環境が進化し、コンパイラに「シンタックスチェッカ」が装備され、ソースコードを入力中にスペルチェック等をしてくれるサポート機能（熟練プログラマには要らない機能かもしれません）が付いています。これにより、タイプミスのような開発者のケアレスミスを防止し、実装の段階から開発者をサポートしてくれる開発環境に進化してきています。

このような新しい開発環境に移りつつある中で、その当時私が感じていたのは、以下のような事でした。

- 「ソフトウェアが正しく動作する」という論理的な検証をキチンとしていないうちに、簡単にソース修正、コンパイル、実行確認するような、Trial & Error のような方法で開発していいのか？

成果物を改良する方法の一つに「Trial and error」がありますが、これは「試行錯誤」という手探りで答えを探し当てる方法ですので、実は、非常に効率の悪い方法です。ただ、誰も足を踏み入れたことの無い新しい研究分野等では、有効な方法だと思いますが、技術がかなり成熟したソフトウェア開発の世界では、開発効率の非常に悪い方法だと思います。

３０年以上前と現在の開発方法を比較すると、コンパイル結果を手にするのに半日以上必要だった開発環境では、1 回 1 回のコンパイルが「真剣勝負」でしたから、実装したソースで「間違いない事」の確信を持てる状態にしてから、コンパイルという作業をしていました。しかし、当時の新しい開発環境では、すぐ「ソース修正～コンパイル～実行」できる気楽さから、実装したソースが、本当に「正しい実行ロジックに基づいて動いている」という事を開発者自身が「間違いない」と確信しないまま、開発作業を進めてしまう事で開発のスピードが優先され始めたことに戸惑いを感じていました。

そして、最近では、もっと開発環境が整ってきており、テストツールでは、OK(グリーン)か NG(レッド)かを判定してくれるまでになっています。最近になって、以下のようにも感じてきました。

- OK(グリーン)、NG(レッド)の結果がすぐ出るが、「OK」が出ただけで、本当に正しいコードが実装できた事が証明できたわけではないのに、OK の結果だけで正しく実装されたと判断してしまっていいのか？

これらの進化した開発環境は、ソフトウェア技術者から、「なぜ OK になったのか」について考える事を奪っていると思います。このまま、なぜ、正しく動いたのか、を理解・検証しないまま、OK となった結果（事実）だけを見て、「テスト完了」にするような条件反射的な開発を続けていると、とんでもないことが起こると思います。

ソフトウェア開発の基本

ソフトウェアを開発する事の基本を改めて認識する事が必要なのでは無いかと思いました。ソフトウェア開発の基本は、マニフェスト風に言えば、

「動くソフトウェアを開発する技術」　よりも
「動く事の裏づけを証明しながらソフトウェアを開発できる技術」がより重要

と言うことだと思います。

言い換えれば、テストツールで『OK になった事だけ確認しながら開発する』のと、『なぜ OK になるのかをしっかり認識（証明）しながら開発する』のは、後者のほうが重要で、本来そうあるべきです。

余談ですが、昔（３０年以上前）は、わざわざ不要な命令を埋め込んで、ハードウェアとのタイミングを合わせるような実装をする事もありました。ソフトウェアの外部仕様としては、「意味がない命令」をあえて実装し、ハードウェアの動作とのタイミングをとる、このような実装は、なぜ OK になるのかをしっかり認識しないとできないことです。反対に、「意味がない命令」を挿入した理由は分からないけれど、「（なぜか）動いている」から良し、とするのとは、ぜんぜん違う話です。実装されているソースコード一つ一つが存在する理由や必要性をしっかり説明できる事がソフトウェア開発では重要だと思います。

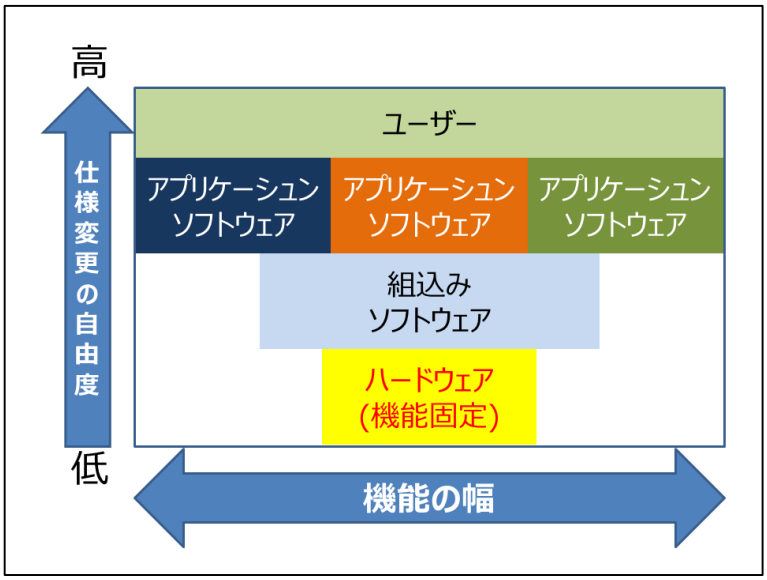
ソフトウェアを開発するという事

ソフトウェア開発は、城の石垣を築くようなもので、基礎の部分であればあるほど、キチンと築かないと後々、とんでもないこと、ひどいときには、石垣全体が崩れてしまうこともあります。

それは、ソフトウェアの仕様変更の自由度が、ハードウェアに近い基礎部分程小さい（組込みソフトウェア）からです。一般的に、ハードウェアには、仕様変更の自由度はありません。ハードウェアの仕様の自由度を広げる役割をしているのがドライバーに代表されるような「組込みソフトウェア」ですから、城の石垣で言えば、一番の基礎部分になります。

その基礎部分が不安定だと、その上に積む「アプリケーションソフトウェア」の動

作も不安定になってしまいます。ソフトウェアを「機能の幅」を横軸にして、縦軸に「仕様変更の自由度」を取って表現すると次の様な構造になります。



一番下のハードウェア（黄色い部分）は、仕様変更の自由度は無く（機能固定の為）、組込みソフトウェアが、ハードウェアの機能の幅を広げ、仕様変更の自由度も高めています。その上に位置するアプリケーションソフトウェアが、機能の幅をさらに広げ、仕様変更の自由度をユーザインターフェースレベルまで高めています。

石垣を築く際、弱い石を使えば、後で修理もできないことになります。ソフトウェア開発で例えれば、ネット上にある得体の知れないソースコードを不用意に使うことに似ていると思います。これは、ソフトウェア開発において命取りになります。

ソフトウェア開発では、「論理的に検証されたソースコードで開発する」と言う作業を積み上げてゆかないと、最初からやり直したり、振り出しに戻らなければならない事象になる可能性が非常に高くなると思います。

「キチンと開発する」という事

本来、ソフトウェア開発は、正しいとか、必ず動く、という裏づけ（確信）の上に積み上げられるべきで、そのベースになるのが、開発者の知識とか経験だと思います。論理的な裏づけなく「動いた」から問題は無いとか、理由は分からないけど「動いた」んだから、大丈夫と考えて、開発するのは、技術者の言葉ではないと思います。論理的な裏づけの無いロジックで実装したソフトウェアをプロの仕事として胸を張って納品するのは、間違っていると言うか、プロの仕事を馬鹿にした話です。

テストツールを使って、動作ロジック検証した際に、「グリーンになったからＯＫと判断していい」と言うのには、キチンとした前提があって成立しているのです。具

体的には「論理的な裏づけに基づいて動くコードを実装して、『グリーン』にしたからＯＫ」、なのであって、単に「グリーンになった事象だけを捉えてＯＫ」になるのではありません。

また、冗長なソースが無いかについては、常に意識して実装する事も重要です。そうしないと、後で、その冗長部分に「バグが潜んでいないか」を確認しなければならなくなります。無駄なソース部分を削ぎ落として、シンプルに実装する事も大事だと思います。前にも言いましたが、開発者は、実装したソースのひとつひとつの必要性をキチンと説明できるべきです。

動作ロジックの裏づけがないまま実装されたソースは、どこに埋めたか分からない地雷のようなものです。いつ、誰が踏んで爆発するか分からないのですから、始末が悪い代物です。しかも、除去しようにも、どうやって、除去すべきかが分からないという、とても危険な代物になっています。これは、とてもプロの仕事（お金をもらってやる事）ではありません。「動くからいい」と言うはプロの言葉ではありません。

どういう動作ロジックで動いているのか分からないけど、「テストでグリーンになったのだから大丈夫」という開発をしていたのでは、逆に動かなくなったときも、なぜ動かなくなったのかが分からない、という状態になる事を意味しています。動いているうちは、問題になりませんが、動かなくなったら、動くようにする方法が分からない状況になりますから、埋められた地雷を探すように、動かない原因を探しあてるか、そのソースコードをすべて捨てて作り直すか、の二者択一を迫られるかもしれません。論理的な裏づけの無いソースを実装するのは、とても危険な行為です。

ソフトウェアは鮮度がある人工物

ソフトウェアは、「生物（なまもの）」です。「生き物」あるいは「鮮度のある人工物」と言ってもいいと思います。先にも述べましたが、このような動的に捉えるべき物を契約書のような静的なワクにはめようとするからうまく契約できなかったのだと思います。

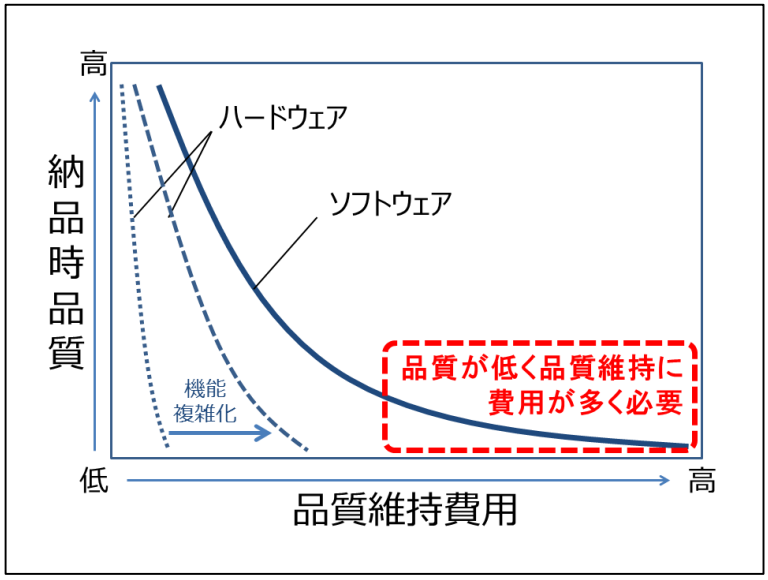
プロトタイプとして開発し、すぐ捨ててしまうソフトウェアを除けば、ソフトウェアというものは、開発完了したとたんに、どんどん陳腐化していきますので、鮮度を保つためには、手をかける必要があります。手をかけないと、たちまち、鮮度が落ち、使い物にならなくなります。

新機能の追加をしないまでも、それなりの手間をかけないと、ソフトウェアはそのうち使えなくなります。バグを作りこまないように開発するツールや開発技術も確立されてきていますが、リファクタリングをしないまでも、不具合の修正は必要です。開発時には不具合でなくても、使用環境の変化等の外的要因で不具合になることもあります。ソフトウェア開発の費用は、開発時の費用のみならず、鮮度を保つための維

持費用も考慮する必要があります。

ソフトウェアには生涯費用が必要

使い続けるソフトウェアは、必ずその鮮度を維持するための費用が必要になりますから、ソフトウェアを安く開発できたとしてもそのような納品時品質が悪くなる開発では、そのソフトウェアの寿命全体で見た場合の必要費用全体では、維持費が高くなるので、結果的には安くなっていないと思います。納品時品質を高くするためには、開発費用も高くなる傾向がありますが、「納品時品質」と「品質維持費用」には、以下のような関係があると思います。



ハードウェアとの比較で表示しましたが、ハードウェアも複雑さが増すと、品質維持費は、ソフトウェアのように高くなる傾向があります。不具合を起こす可能性のある部分が増えるのがその理由です。ソフトウェアは、複雑さの「カタマリ」なので、ハードウェアの維持費に比べて、かなり大きくなってしまいます。

安かろう、悪かろうでソフトウェア開発されたものは、納品時品質の高さは期待できませんから、納品時品質の高いソフトウェアよりも、維持費用が多く必要になると思います。キチンと開発されたソフトウェアは、納品時品質を高くできますから、バグの発生が少ないと言う観点で見ても、維持費が少なくて済みます。

一般的に、品質を高くするためには、開発費用も連動して高くなりますが、一時的な費用です。ソフトウェアが役目を果たし、使われなくなるまでの期間を「ソフトウェア寿命」と表現すると、このソフトウェア寿命に必要な「生涯費用」は、開発費用の他に、保守費用、品質維持費用が必要になりますから、「納品時品質を高める」ことは、ソフトウェア生涯費用を低減させる事につながると思います。

ソフトウェア開発をサポートする道具

アジャイルプロセスをはじめとするソフトウェア開発をサポートする道具は、プロジェクトの異状を早期に検知したり、開発作業を格段に効率化させるだけでなく、納品時品質を高めるような道具も充実してきてきました。ただ、これらのソフトウェア開発をサポートする道具は、使い方を間違えると効果を得られないばかりか逆効果になってしまいます。

また、優れたツールも、これを使いこなす職人がいてこそ活きます。家を建てるための道具であるノミやカンナで例えると、ノミでカンナのような仕事は出来ない事は無いですが、柱や、壁板の表面を滑らかに仕上げるにはカンナの方が、早く滑らかに仕上げられる事は、明らかです。しかし、カンナという道具をうまく使わないと、表面は滑らかに仕上がらないし、木の表面を滑らかに仕上げるために必要な時間も多く掛かってしまいます。道具は、用途に合ったものを正しく使わないと、かえって作業効率を悪くする可能性もあり「諸刃の剣」にもなりうる存在だと思います。カンナで、柱の表面を滑らかに仕上がる場合、カンナの刃の出し方やカンナを引く方向をキチンと意識して扱わないと、柱の表面がささくれ立ってしまい、表面を滑らかに出来ないばかりか、表面をザラザラにしてしまいます。（中学時代の木工工作の授業で、課題で作った「本立て」の表面がささくれ立ってザラザラになっているクラスメートが何人かいたことを思い出しました）

ソフトウェア開発と道具の話

全く同一のソフトウェアを開発する事は無いとしても、品質を維持し、効率的にソフトウェアを開発するには、道具の使い方を間違わないことが重要である事は先に述べました。そうしないと、開発時間と品質を維持できなくなり、プロとしての開発ができていない事になります。

道具を使いこなす、と言うのは、道具の用途や特性を知った上で、道具を活かすことが基本です。道具の用途や特性を知らなければ使えないし、使いこなせません。道具を使いこなして初めて、道具が本来持つ特性を活かすことができると思います。

アジャイルプロセスは、ソフトウェア開発に有効な「道具」である事は、周知の事実です。今では、様々なソフトウェア開発の有効な道具になっていると思います。

ただ、前にも書きましたが、良い道具も使い方を間違うと本来の効果を発揮できません。大工道具の「ノミ」で「カンナ」のような仕上がりを試みれば、腕のいい大工さんならできない事は無いでしょうが、作業に必要な時間の割りに、カンナで仕上げた様には仕上がらないでしょう。

ソフトウェア開発の道具はカスタマイズが必要

これは、何度も言いますが使い方や用途を間違ってもダメ、ということで、道具の効果を発揮するためには、道具の特性／効果／使用方法を知った上で使いこなすことも重要なことであり、「効果があるらしい」という噂だけで飛びついてしまうのは危険だと思います。

他のプロジェクトで成功したプラクティスであっても、カスタマイズが必ず必要で、オールマイティなやり方は存在しない、と考えたほうが良いと思います。

一般的に、道具は、使うために作られるのであって、使わない道具は作られません。ソフトウェアの場合、ユーザにとって、ソフトウェアそのものが、「ユーザのビジネスで富を得るため」の道具なのだと思います。

一方、ソフトウェア開発者にとっての道具は、開発言語コンパイラ、自動テストツール、開発プロセスなどが、技術者の開発力を強力にサポートする道具になります。

道具を使う人にとって何が道具になるかも変わってきます。契約書も道具と考えられ、ユーザは、ユーザを守る道具にしたい一方で、ベンダは、ユーザの理不尽な要求からベンダ自身を守りたいと考えていると思います。ユーザとベンダのどちらが有利とか不利という議論を始めたらその時点で、そのプロジェクトは、うまく行かないでしょう。

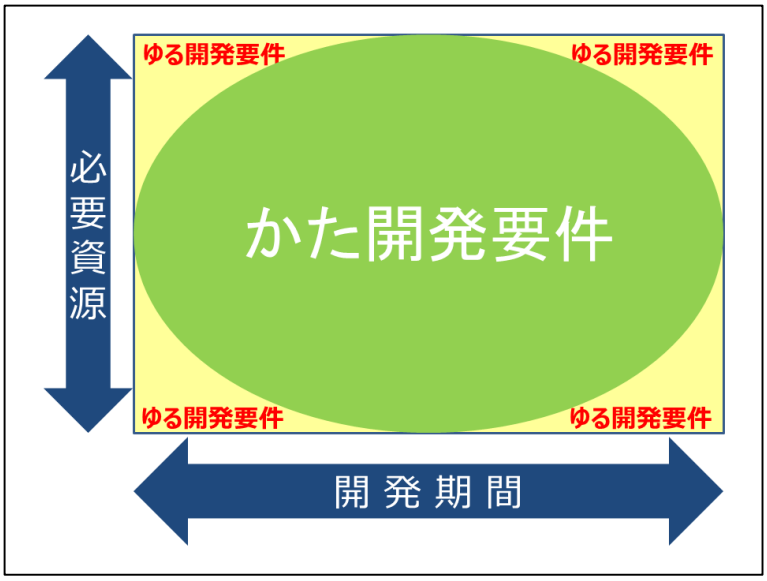
なぜなら、ユーザとベンダが、同じ方向を向いて開発しておらず、相手が自分にとって不利となるような行動をとらないかどうかを注視しているからに他なりません。

ソフトウェアの開発の道具をより機能させるために

ソフトウェアの開発の道具をより機能させるために必要な事。これは、開発要件を流線型にする事だと思います。これは、開発要件を「かた開発要件」と「ゆる開発要件」に明確に分けて取り扱うと言うことで実現可能です。流線型という形は、新幹線、リニアモーターカー、ジェット機など、とにかく、早く進める構造を持ったものですので、これらのものと同じように、「流線型」にする事が、有効だと思います。

さて、どうすれば、流線型になるのかと言うと、実は簡単で、「角ばった部分を丸くする事」で、流線型に近づいていくと思います。開発要件と言うのは、ユーザは兎角、「開発期間×必要資源」の長方形でできた器に、ギリギリ入りきる開発要件と、考えがちです。

長方形の面積	＝	開発期間×必要資源
楕円形の面積	＝	開発期間/2×必要資源/2×π
	＝	開発期間×必要資源×78.5%



ここで、２２％程度のゆるさを認めることで、かなり「角」が取れて、流線型（緑色の部分）になることが分かります。流線型にした要求仕様が実装完了したら、２２％の部分をどのように使うのかを両者合意の上で決めればいいと思います。次に開発予定の仕様を実装してもいいし、リファクタリングに使ってもいいし、ユーザ/ベンダ間で合意すれば、両者にとってバッファにする事もできると思います。想定外の事象が発生した際の保険としても使えますし、トラブルが無ければ、次のバックログを前倒して実施する事も可能になります。

契約について

ここで、契約書の事に触れておきたいと思います。

始めに言いたいのは、契約書だけでは何も解決できません。契約書は、契約当事者同士の揉め事の仲裁はできません。大事なのは、当事者同士の関係（つながり）Agile manifesto にもあるように「契約より関係」が大事だと思います。

なぜ大事かを考えるのには、契約当事者をシンプルな状態で見てみるのが近道です。自分ひとりの状況（一人だけの契約）が究極の良い関係になります。なぜなら、一人だけの契約ですから、自分自身の中で、自分自身の行動に合意しながら行動する事になるので、「自分の行動は自分で責任を取る」状態になります。この状態では、対立しようがありません。一人でやっても失敗する事がありますが、一人なら失敗しても自分の責任にできます、というより、自分が納得して行動した結果ですので自分の責任です。

しかし、契約当事者が二人だと、問題が発生したときに、責任の所在を特定する事が、一挙に複雑になります。どちらか一方の責任なのか、二人の共同責任なのか、の切り分けはもちろん、問題発生の「責任の重さの特定」が難しくなります。二人の責

任ならば、その責任比率はどうなるのかと言うことが、問題になってきます。この責任比率が、いつも「責任比率は両者とも５０％である」と言える両者の関係が大事になってきます。

そのためには、ユーザとベンダ間の普段のコミュニケーションが重要で、どう行動するかをいつも両者で納得して、開発プロジェクトを進行させる事が大事になってきます。両者が納得できない事は、その部分だけプロジェクトを進行させないのが一番です。ＴＰＳで言うところの「不良は止める」という事です。

あらゆる行動は、両者が合意した上で実施するようにすれば、責任は５０％ずつ負担する事になります。５０％のバランスを崩す比率は、トラブルの元になります。

表を見てください。「ベンダ：ユーザ」で、不具合発生時の責任比率を考えた時、仮に「60%：40%」とすると、ベンダはユーザの「1.5 倍」もの責任を負うことになります。もう少し僅差で、「55%：45%」であっても、「1.2 倍」の責任を負うことになりますので、「50%：50%」の責任分担でなければ、ユーザベンダ間で責任負担でもめることになると思います。また、妥当な比率（倍率）の意味・理由（裏づけ）が、難しくなります。

１００％での 責任比率 (ﾍﾞﾝﾀﾞ:ﾕｰｻﾞ)	ベンダ責任倍率 (ﾕｰｻﾞを「1.0」とする)
５０％：５０％	１．０：１．０
５５％：４５％	１．２：１．０
６０％：４０％	１．５：１．０
（略）	（略）
９０％：１０％	９．０：１．０
９５％：５％	１９．０：１．０

例えば、ベンダの納品時品質が悪かったケースでこの責任比率を考えてみると、その前提として、いつもの納品時品質を両者が知っていないと判断できないと思います。次に、そのプロジェクトで、いつもの納品時品質が出せなかったとしたら、その原因があるはずです。その原因が、両者が納得のいくもの（仕様確定時期の遅れなど両者で合意できるのも）ならば、責任比率は、「両者で折半」に合意できると思います。

本当は、責任比率の議論をしなくて済む方法を考えたほうが近道です。簡単な方法として、契約書の前文に「両者の契約に向けた心構えを書いて」おくのがよいと思います。具体的には、両者協力してことに当たることを宣言します。あとは、開発概要、契約期間、納期、納品物、対価（契約金額）、仕様の詳細は、プロジェクトの進捗に

合わせて固めていく方法でいいのではないかと思います。そもそも、ソフトウェア開発は、開発要件そのものが動的に変化してゆくのが常ですから、それを前提に開発要件を徐々に固めてゆく方法が良いと思います。

アジャイルプロセスで開発しやすい契約書

実務ではアジャイルプロセス開発が実現できる事を目指した「スポット契約用の開発委託契約書」を作りました。また、「アジャイルプロセスに適した契約書」というものをＩＰＡ（独立行政法人　情報処理推進機構）のプロジェクトにアジャイルプロセス協議会の立場で参加して、プロジェクトの各委員（弁護士含む）と協力して作りました。

実は、ＩＰＡのプロジェクトに参加していて、「何か違うような気」がしていました。プロジェクトが終わってみたときに気付いた事は、「アジャイルプロセスで開発ができる契約書」を作っていたのだ、ということでした。

アジャイルプロセスで開発する事が、目的（前提）の契約書を作っていました。本来、開発プロセスは、何でもいいはずで、極端に言えば、アジャイルプロセスでなくても良いと思います。ほとんどの開発委託契約には、「開発プロセス」を限定するような条文は無いと思います。

ユーザとベンダが、契約するのは、ユーザのやりたい事（一般的には、ユーザのビジネスで富を得ること）を実現するために、ベンダは、ＩＴ技術・ノウハウを駆使してそのやりたい事を手助けする事が最大の目的だと思います。

契約書の役割

最近私は、契約書の役割は、開発委託の取引があったという証拠書面としての役割を果たせば、それで十分なのではないか、と思うようになりました。ユーザ、ベンダ両者にとって、開発委託の取引をしたのであって、ライセンス契約ではない事の証拠書類となれば十分だと思います。このための前提は後述します。

「ゆる思考のすすめ」でも触れられていますが、ゆる思考は「時間的な制約」にとらわれない考え、と理解しています。一般的なソフトウェア開発では、始めは、開発すべきことが決められていても、終り（開発完了）の形は時間とともに変化してしまいます。契約書のような静的な約束事しか扱えないもので、動的なソフトウェア開発を契約（約束）する事はできないのだと思います。

それなのに、契約書を使ってソフトウェア開発の取引をしていたために、トラブルが発生していたのだと思います。もともと、動的に変化する「ソフトウェア開発プロジェクト」というものを静的にしか取り扱う事ができない「契約書」で扱おうとしていたところに、無理があったのだと思います。

優れた職人

ここで、人の話をしたいと思います。一般的に、優れた職人と言うのは、道具を使いこなせる事も大事ですが、自分が作ったものが、故障したり、壊れたりしたときに、それをまた元通りに、あるいは、もっと使いやすく修復できる人の事を言うのだと思います。

メーカのように大きな職人の集合体も一人の職人と置き換えることができます。自動車メーカーや航空機メーカーは、かなり大きい職人の集合体と見ることができます。自動車のユーザーは、どのような仕組みで、自動車が動いているのかを知らなくても、ガソリンを給油して、自動車の操作マニュアルどおりに運転すれば、自動車を動かすことができます。自動車を整備できるほど自動車の事を熟知する必要は、ユーザにはありません。動かす、スピードを出す、止める、後退する、位の操作方法を知っていれば、縦列駐車や、車庫入れの運転技術が無くても、自動車を運転するのに支障はありません。

でも、自動車メーカー側は、アクセルを踏み込むとなぜスピードが速くなるのか、それを実現しているいるのはどの部分でどのような部品が使われているのか、など詳細に熟知している必要があります。また、スピードが速くならなくなった場合には、その原因を特定できる程に、内部構造や仕組みを詳細なレベルで熟知している必要がありますし、熟知していることが求められます。同じことが、優れたソフトウェア開発者にも当てはまり、求められると思います。

また、ソフトウェア開発プロジェクトにおいて、そのような優れたソフトウェア開発者は、システム全体を知っている棟梁のような人として存在していると思います。

優れたソフトウェア開発者

自分の作ったソフトウェアを修理／改良／改造の要求が来たときに、あなたにはできますか？

この質問に「はい」と答えられる人は、自分の開発したソフトウェアが、どのような論理の元で、実行しているのかを理解している人だと思います。そうでないと、どの部分をどう修正していいか分からないはずです。３０年以上前では、ソフトウェア開発の世界では「動いたソフトは触るな」と言われていました。今ほど、テスト環境が整っていなかった頃ですので、下手にソースをいじって、動かなくなったら、その修正／動作確認作業が大変だった、と言う現実がありました。

その頃は、コンパイル＋テストプロ実行に半日必要だった開発環境でしたから、テストのやり直しは、勘弁して！という時代でした。今は、開発環境が整備され、早く開発する事が求められるが故に、技術者は、「Trial & error 式の実行テストで OK に

なった事だけで満足」してしまう傾向にあるような気がします。このため、キチンと開発できる技術者が育たない環境になっているのだと思います。

ソフトウェア開発技術者のレジェンド

最後に、今まで関わってきた契約書を作成する上で、無意識のうちに想定していた条件について、気付いた事がありました。それは、ベンダの「力量」です。

具体的なその「力量」とは、ソフトウェア開発技術者のレジェンドといわれるような以下のようなスキルを持った技術者（リーダー）が率いる開発者集団をイメージしていたことに気付きました。

〇考えられるあらゆる事象を想定して実装する
　→想定外の事象を発生させない（たとえ発生したとしても異状終了させない）
〇開発したソフトウェアをすべて記憶している
　→囲碁や将棋の世界での名人に匹敵する記憶力の持ち主
〇自分が実装したソースかどうかをすぐ判断できる
　→自分の開発スタイルが確立されているため自分が実装したコードが分かる

契約書で本当に約束すべきこと

契約において、本当は、「何を約束するのが大事なのか」は分かっていても、従来は「何を、何時までに、どの程度の金額で開発するか」を約束していることが殆どだったと思います。この約束の中で「何を」を約束するアプローチが間違っていたのだと思います。そもそも、ユーザは、「自分のビジネスをどうしたしたいか」は、理解していても、そのために、何を開発して欲しいのかを知らない事（発想できていない事）が殆どだと思います。

という事は、開発を始めるに当たって、ユーザの発想できていない代物を「開発します」という約束はできません。額面が白紙の小切手を渡すようなものです。ユーザ、ベンダの両者にとって、お互いに理解できている状況（ベンダの実績、技術者の経験など）と約束する事を関連付け、相手を信用して契約する事しかできません。

契約書の位置づけとして、ユーザとベンダの約束事が書かれた書類だけでなく、例えば、技術者〇人（工数を決める目的や個人を特定するのではなく、リーダー、プログラマなどの技術者スキルレベルを特定する）、開発期間（納期）、取引金額、が記載されていることで、少なくとも資産価値（ライセンス性：投入した工数に対する対価が高すぎない）がない事を証明する書類、程度の位置づけで十分だと思います。

これで、ユーザもベンダもフットワーク良くプロジェクトに取り組めると思います。先にも述べましたが、アプローチとしては、１００％の機能の 3/4 位（ただし、本当に使うもののみ）を「かた開発要件」として開発必須とし、1/4 は、作るかどうか分

からない柔らかい「ゆる開発要件」にしておくとスピードアップできます。

もっと簡単に言うと、ベンダの「知を働かせてプロジェクトに臨む事」を約束する事で、すべてのプロジェクトがうまく行くのかも知れません。

どんなに良い契約書も両者を平等に守ってくれないので、良いパートナーを見つけて、そのパートナーと仕事をするのが、プロジェクトを成功させるための一番の近道であると確信しています。

先に述べたように、契約をする資格のあるベンダであれば、「取引の記録」程度の位置づけの契約書で問題ないと思います。優れた道具を使いこなせる優れた技術者を擁するベンダと、本当に自分のビジネスの成功に取り組もうとしているユーザとが契約するのが理想の姿です。

（完）

コラムー１

（人は、見るもの聞くものを自分に都合の良いように無意識に解釈してしまう動物）一つの言語（偉人の言葉とか）も間違ったとらえ方をしたら、大変なことになる、と言う話。人という動物は、自分に都合のいいように考えたり、思い込んだりする習性があります。

アジャイルプロセス開発でのソースコードの共同所有→開発者が「ソースコードに責任を持たなくてよくなる」訳ではありません。

偉人の言葉で有名なのは、エジソンの「私は、失敗を恐れない」→そもそも対象となっている失敗が違う、誰もやったことが無いチャレンジングな事をやろうとして、結果的に失敗しても、そんな失敗は恐れる必要は無い、という事で、単純な計算ミスや手続きミスのような失敗ではないのだ。エジソンの言葉は、チャレンジングな行動を起こしてその結果がうまく行かなかったという失敗なのだ。そこを意識しないで、額面どおり受け取って理解しては、いけない。

さらに、エジソンの言葉は、本当は、

「失敗なんかしちゃいない。うまくいかない方法を七百通り見つけただけだ」や

「失敗すればするほど、我々は成功に近づいている。」

というもので、それらをもっと短縮して「失敗など恐れない」としても、この言葉だけを聞いただけでは、その言葉の本当の意味を理解する事はできません。

したがって、「ユーザからの言葉」の理解を「ベンダ自身の言葉」を使ってユーザに確認しながら、ユーザとの合意形成をしてゆくことが、トラブルを未然に防ぐためには、不可欠になってくると思います。

コラムー２

（ソフトウェアの持つ可能性）

ハードウェアの単純な機能だけを使って「できる事」を考えてみました。

ハードウェア:　赤い LED

機能:　点灯(or 消灯)

すると、単純な機能のハードウェア（LED）でも、供給する電気を ON/OFF させる事で、かなり多くの働きを持たせることができます。

たとえば、１秒ごとに電気をオン(供給)/オフ(切断)を繰り返せば、「点滅」させることができます。そのほかにも、さらに高速に電気のオン/オフを繰り返せば、供給する電圧を変化させること無く、明るさを調整可能になります。

ただし、明るさを調整するには、単純に、電気のオン/オフを繰り返すだけでは実現できません。

点滅させるには、１秒単位にオン/オフを繰り返せばいいですが、明るさを調節する場合は、人間の目にも留まらぬ速さで点滅させ、その「高速な点滅の中で、オフの時間の長さを調節する」、というコントロールが必要です。

赤の LED の明るさを変化させることができれば、さらに、青と緑の LED を追加して、色を表現する事ができるようになります。

単色の LED を３種類使って、様々な色を表示できるようになります。一つ一つのハードウェアの機能は単純でも、その制御方法によって、別の機能を持たせることができます。その制御をする役割を持っているのが「組込み系ソフトウェア」になります。

ハードウェアの機能は、仕様変更の自由度はゼロ（仕様変更不可）ですので、組込みソフトウェアは、仕様変更の自由度ゼロのハードウェアに、仕様変更の自由度を与えるものと捉えることができます。さらに、組込み系ソフトウェアの上には、アプリケーションソフトウェアが配置され、さらに、仕様変更の自由度を広いものにしています。

もともと、そんな機能が無い所から、様々な機能を生み出すことができる、それがソフトウェアであり、そんな力を秘めている「人工物」という事ができると思います。