【資料 4】情報システム開発/管理に見るレジリエンス・エンジニアリング

（文責：桑田成規）

はじめに

- レジリエンス・エンジニアリングが対象とするのは《システム》
  - システムとは、個々の要素が相互に影響を及ぼしあって機能するまとまりや仕組み
  - 広く、社会システム、インフラ、生態系、組織、などが対象となる
- この発表では、《システム》の中でも《情報システム》に着目し、情報システムの開発や管理におけるレジリエンス・エンジニアリングの具体例を中心にお話しします

レジリエンス・エンジニアリング、というのは非常に広い概念で、個々の要素が相互に影響を及ぼし合って機能するまとまりや仕組み、つまりシステムを対象としてレジリエンスがどう働いているかということを研究する分野です。

システムとは、具体的には社会システムとか、インフラとか、生態系とか、人間も含めていろいろなものが対象となるのですが、ここでは、私の専門に近い情報システムという分野でのレジリエンスというのを考えてみようと思います。

レジリエンスの4つの考え方

D. Woods; DOI:10.1016/j.ress.2015.03.018

Linear systems

1. Resilience as Rebound
2. ･･･as Robustness

Adaptive/Non-Linear systems

3. ･･･ as Graceful Extensibility
4. ･･･ as Sustained Adaptability

本講演に先立ちレジリエンスを研究されているDavid Woods先生という、アメリカの有名な工学系の先生のYouTubeや論文を拝見しました。私が今やっていることが、このレジリエンスにどう関係するかということを、このWoods先生のレジリエンスの4つのコンセプトに沿ってまとめていきたいと思います。

4つのコンセプトについて説明しますと、1つは、一番シンプルなところで、リバウンドとしてのレジリエンスというものです。次がrobustness。これは日本語でいうと、堅牢性という言葉になりますが、この２つがまずlinearなシステムの考え方として当てはまるものだ、とおっしゃっています。その先はadaptive/non-linearの世界で、柔軟性が必要とされるシステムにおいて、graceful extensibilityとsustained adaptabilityの2つの考え方を提示しておられます。これがいったいどういうものかということも簡単に含めながら、進めて行きます。

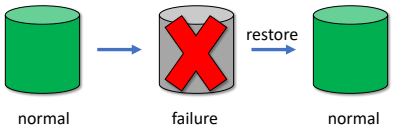
目 次

1. Rebound
2. Robustness
   - 単一障害点(SPOF)
   - 冗長化(Redundancy)
3. Graceful Extensibility
   - Graceful Degradation
   - Software Extensibility
   - 仮想化
4. Sustained Adaptability
   - Cloud/Elastic Computing
   - 周辺警備隊(SNAFU Catchers)
5. ソフトウェア開発におけるGraceful Extensibility

今日お話しする内容はこういった構成で、最初の４つが先ほどの４つの考え方に相当するものです。最後に、できあがったシステムのことではなく、システムを作る段階のソフトウェア開発においても、やはりこういったレジリエンスの考え方というのが重要でありまして、そこでどういうふうな具体的なツールであったり、進め方をしたりしているのか、ということを最後にお話しできればと思っております。

1. Resilience as Rebound

- システムが破壊的な状況に陥っても、システムを元の状態に戻すことができる
- 情報システムでは、復旧もしくは復元(recovery/restore)可能性



最初に、リバウンドとしてのレジリエンスです。リバウンドとは元に戻るという意味ですが、情報システムの場合でいいますと、システムが一旦壊れてしまい故障を起こしてしまって使用不能な状態になってから元の状態に戻る、そういった状態の遷移のことをいいます。しかし、情報システムの世界ではリバウンドという言葉は使いません。かわりに、皆さんもよく耳にすると思うのですが、リカバリーやリストアという言葉を使います。日本語では復旧とか復元とのことですね。

Rebound(Recovery)の手法

- バックアップの取得
  - 範囲:フルバックアップ、差分バックアップ
  - タイミング:リアルタイム、日次、など
  - 世代:残しておくバックアップデータの個数
  - 媒体:オンライン(ネットワークドライブ)、オフライン(テープなど)
- バックアップからの復元
  - 取得方法により、所要時間、確実性、復旧の程度が異なる
- システムの特性や予算に応じて手法を選択

これは非常にシンプルで、具体的に何をするかというと、このリカバリー手法としてはバックアップを取っておく。これはもう基本中の基本で、たぶんどなたでもされていると思うのですが、「バックアップにもいろいろございまして」、という話です。範囲をどうするかということで、全部いっぺんに取るフルバックアップというやり方とか、また、全部取ると大変なので、電子カルテシステムなどは日ごとに差分を残していって、たとえば本院の場合は7日分、曜日ごとに差分のバックアップを取って、また次の曜日が来たらフルバックアップを1回取って、また差分を残すというよ

うなことをやっています。

バックアップのタイミングをリアルタイムでとるのか、日次で取るのか、ということもあります。当然、リアルタイムで取ることが理想的であって、日次で取ると1日分のデータがロストしてしまう可能性もあります。

あと、世代管理、これに関しては、先ほど申し上げたとおり、本院では7世代ということになります。で、何の媒体で取るのかというところも重要でして、オンラインで取るのか、テープのようなオフラインで取るのかということです。

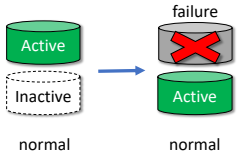
最近は、ランサムウェアがはやっていて、とある病院ではこれで甚大な被害を受けて、診療が何日もストップしたということがございました。あれは、結局、オンラインでバックアップを取っていたがゆえに、その攻撃もオンラインでつながっている先まで及んでしまった、つまり、バックアップ自体もランサムウェアの対象になってしまったということで、復元が非常に難しかったということがありました。ですので、今、厚労省の方から、バックアップをオフラインで取っているかという調査も来ているですが、このへんもやはりいろいろと見直していく必要があるだろうということになっています。

それで、バックアップは、結局、復元できなければ意味がないわけで、先ほど言いましたように、バックアップファイルが壊されてしまうとか、壊れてしまうということはもちろん避けなければいけませんし、バックアップの取り方によってどこの時点まで戻れるかということも変わってくる。これはシステムの特性、どれだけミッションクリティカルなシステムなのかとか、あるいはお金をどれだけかけていいのかということによって変わって

くるものです。以上が1番のリバウンドの説明になります。

2. Resilience as Robustness

- システムが、障害発生時においても、元の状態を維持することができる
- 情報システムでも、堅牢性(robustness)



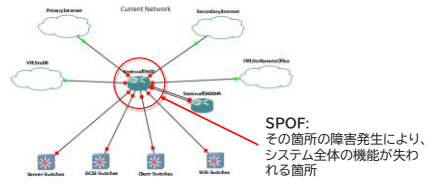
次にrobustness、堅牢性についてです。堅牢性というのは壊れない、というよりも、壊れても稼働をそのまま保証する、つまり、元の状態を維持したまま障害にどれだけ対応できるか、ということをrobustnessといいます。

これは情報システムでも同じ言葉を使っていまして、普段、たとえばこの図のような構成があったとします。これは1つのシステムなのですが、実は、その中に二つの機能が動いていて、１つはアクティブで、もう1つはスタンバイの状態、つまり動いていないという状態です。

普段はこのアクティブの方を使っている。そこで、かりに、このアクティブのものが壊れてしまったとしても、スタンバイしているものが、それを検知して、自分がアクティブに変わる。まあ、数秒ぐらいは止まるかもしれませんけれども、外見上、このシステムを使っている人は、実は中では壊れているのだけども、そのまま元のように使える。こういったものが、どの程度の品質まで保たれているか、ということが堅牢性ということになります。

Robustnessの手法

- 冗長性(多重化による性能維持;redundancy)
  - 重要な部品(機能)や装置を複数配置する
    →単一障害点(Single Point Of Failure)を排除

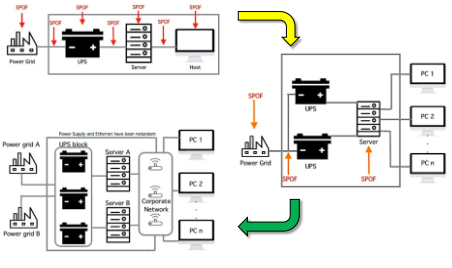


https://community.spiceworks.com/topic/2129466-physical-network-topology-for-no-single-point-of-failure

それで、具体的にどんな手法があるかということですが、本当に多種多様ではありますが、基本的には、冗長性を確保するということです。冗長性というのはredundancyという言葉の日本語訳なのですが、多重化、つまり同じものをいくつも用意しましょうという非常にシンプルな考え方です。

そのときに重要なのが、単一障害点です。SPOFとわれわれは言いますが、これをできるだけ少なくする、ゼロにすることが理想です。で、単一障害点というのは何かというと、たとえばこの図でいいますと、これはいろんな装置がハブ状につながっていて、周りにある雲のような形のものがネットワークのインターネットだと思っていただければいいかと思います。それで、このネットワークを使っている人がこの図の下の方や横の方にいて、通信をしているわけです。そのときに、たとえば、この赤い丸で囲った、真ん中にある機械が一つしかなくて、壊れてしまうと、どの人もどこにも通信できなくなってしまって、もう大変なカタストロフが起こってしまうわけなのです。こういった真ん中のところをSPOFといいまして、システムの構成を眺めてSPOFを排除していくというのが、基本的なrobustnessの具体的思考ということになります。

Redundancy: avoid the SPOF



https://www.rankred.com/single-point-of-failure/

SPOFを避けるという意味で、さらに具体的な例として、この図のようなシステムがあったとします。左上の図を見てください。左側にあるPower Gridは発電所です。そこから電気が来て、UPS（無停電電源装置）という安定的な電源を供給する装置につながります。さらに、それがサーバーにつながり、サーバーがホストにつながって、ホストからサーバーをいろいろ制御するという仕組みになっています

が、これは「SPOFだらけ」なわけです。発電所が壊れても当然ダメですし、送電線が壊れてもダメですし、UPSが壊れてもダメ、ということで、もうすべてがSPOFになっている。これは非常に脆弱なシステムで、堅牢性が低いという評価になります。

それで、少しマシにしようということで、右の図のように、いくつかを二重化して、redundancyを設けるわけです。けれども、やはり、いくつかSPOFは残っているので、さらに完璧にするなら左下の図のような構成になります。

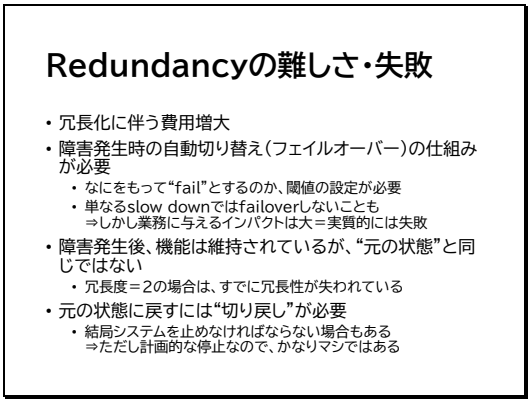
ということで、２カ所の発電所から電気を受けて、複数のUPSを持って、複数のサーバーを持って、複数のネットワークを持って、複数のホストを持つ、こういった構成が理想である、と、考え方としてはこういうことになります。堅牢性を上げるためには、多重度を上げる、つまり多重の数を大きくするのはもちろんですし、あらゆる装置に対して、多重性を持たせると見ることが基本となります。

さまざまなredundancy

- ディスクドライブ(HDD/SSD)
  - データを複数のディスクドライブに分散保管
- 電源
  - 非常用電源の確保
  - 装置内の電源回路の複数配備
- ネットワーク
  - スイッチ・ルータの複数配備
  - 複数の通信経路の確保
- サーバー
  - サーバーを複数配置(クラスタ構成)

それで、どのような冗長化があるのかというと、先ほど出ましたけれども、電源を複数にするというのもそうですし、サーバーの中でも電源装置が1個しかないと、それが壊れるとサーバーに電源が供給できないということで、サーバーの中に複数個の電源装置を持つというのも普通に行われています。

ネットワークの装置ももちろん複数にしますし、通信の経路も複数持ちます。サーバーも複数にします。複数のサーバーを束ねる機能をクラスターといいまして、これは後ほどご説明したいと思います。ハードディスクも複数持つというのがもう普通のことになっています。



では、「今はそういう時代でございます」ということで、この冗長性を上げればいいのかというと、単純に冗長度を２にすると費用が２倍になります。実際には２倍以上かかることが多いです。なぜかというと、障害が起こった時に切り替えをすることをフェイルオーバーと言いますが、さっきいいましたように、アクティブとスタンバイがあって、スタンバイをアクティブにする動きを自動的に行わないと、あまり意味がない。そういうフェイルオーバーの仕組みそのものにもお金がかかります。当然、単一のシステムであれば、そういうものはいらないが堅牢性が低い。堅牢性を上げるために冗長度を上げると、切り替えの作業が必要になって、費用が2倍以上かかる、ということです。

さらに実務的には、私も経験があるのですけれども、どこで切り替えるかという「閾値」をどう設定するかというのがさらに難しい問題です。

サーバーだけでなく、どんな機械もそうですが、0か1かで動いているわけでなくて、なぜか0.3位になってしまう、ということもあるのです。バチンと全部なくなってしまうような落ち方をすれば、それはすぐに切り替わるのですが、そうではなくて、なんだかよくわからないけれども、すごく遅くなっている、アップアップしている状態で、0.1ぐらいになっているというときに、切り替えるのか切り替えないのかというのは、この「閾値」を決めておかなければいけないということなのです。

私が実際に経験したのは、なぜかサーバーが動いたり動かなかったりしていて、フェイルオーバーが起こらなかったけど、電子カルテの画面展開がすごく遅くなっていたケースです。利用者の先生方は「何が起こったんだ」

みたいな感じで、画面をクリックしまくる、クリックしまくると、なおさらサーバーに負荷がかかってますます遅くなるという負のループが始まってしまい、結局、丸一日業務ができなかったというようなことがありました。

これは、障害発生時にはフェイルオーバーする、といいながら、実質的にはフェイルしてしまっていることになります。だから、単純に冗長化すればいいというものではない、ということです。

もう１つの問題は、切り替えた後です。スタンバイがアクティブになってメデタシメデタシかというとそうではなくて、その状態は、完全に元の状態とは同じではないのです。たとえば、冗長度２、つまりマシンが2台あったときに、１台が壊れ、切り替わってもう１台が動いている時点で残ったマシンがSPOFになっています。すでに冗長性がなくなってしまっているのです。ですから、切り戻しといいまして、いつかは元の冗長度2の状態に戻さないといけない。そのためにはシステムを止めないといけないことが結構あるので、「結局、止まるんじゃないか」という話になります。それでも、計画的に止めることになるので、マシではあるのですが、フェイルオーバー後の１台で動いているときに、残った１台に障害が起こったらどうしようと考えると、もう心配でたまらないわけです。

ですので、冗長性はもちろん大事です。けれども、冗長にしたからといって、すべてハッピーになるわけではない、ということです。

Linearモデルの限界

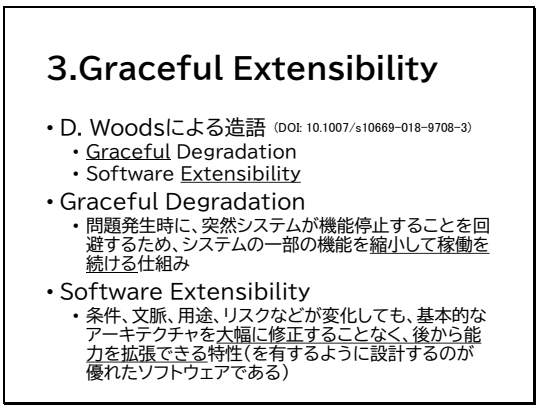
- 堅牢性の程度を高めることは可能である：
  - 冗長度を上げる
  - フェールオーバー(障害発生時の切り替え)の精度を上げる
- しかし、ハードウェアの事前準備が必要であり、ひとたび動き始めれば、堅牢性は一定
  - 所定の擾乱(disturbance)にのみ対処可能
  - 想定範囲外の障害や状況変化には追従不可
- これはresilientといえるのか？

これまでは、レジリエンスの1と2を見てきたわけですが、ここまでがリニアモデルといわれるものです。

費用はかかりますが、堅牢性の程度を高めることはもちろん可能ですし、フェイルオーバーの精度も緻密に計算すれば、閾値も適切に決められるのかもしれない。しかし、一番大きな問題は、ハードウェアというものは事前準備が必ず必要で、設計段階でお金のことも含めていろいろ考えて決めた、まではいいのですけども、いったんそれで決まってしまって動き始めると、その時点で、たとえば冗長度2であれば2と決まってしまうので、堅牢性は一定ということになり、あらかじめ想定された擾乱（障害）にしか対処できないということです。

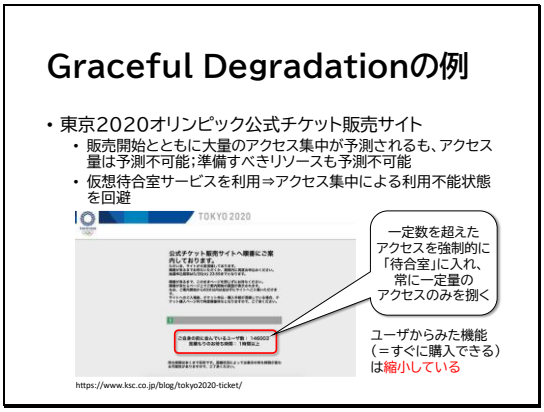
これがリニアのリニアたる所以になります。ですから、想定外のこと、たとえば二重障害、つまり一つが落ちて片肺運行になったときに、さらにもう一つも落ちるといったことに対しては、もう完全に追従は不可能ということになります。

これはすべてハードウェアの制約というふうにお考えいただければいいと思います。ただ、そこで、果たしてこれはレジリエントといえるでしょうか。



Woods 先生のいう 3 つ目の graceful extensibilityというところのノンリニアの世界からが、本当のレジリエンスということになるのだろうと思います。

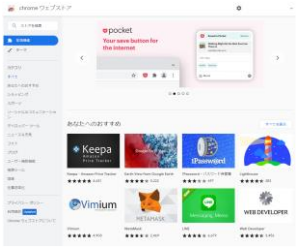
このgraceful extensibilityという言葉は、Woods先生が作られた造語でございまして、情報システムでいうgraceful degradationという言葉と、もう1つはsoftware extensibility、この二つを掛け合わせたものというふうにおっしゃっています。１つ目のgraceful degradationというのは、問題が発生した時に、突然、システム全体が止まってしまうことを避けるために、システムの一部を縮小、あるいは機能を縮小して稼働を続けることです。ですから、アップアップしだしたら、ちょっと規模を縮小してなんとかシステムを動かしましょうというのが、このgraceful degradationという考え方です。これは情報システムではよく使われる手法です。もう1つのsoftware extensibilityについてですが、ソフトウェアは、先ほど出てきたハードウェアとは違って、いろいろな周りの状況が変わったり、リスクが変化したりしたとしても、基本的な構成を大幅に変えなくても、あとから機能追加できる特性があります。これをsoftware extensibilityと呼んでいます。



Graceful degradationについて、具体例をお話ししたいと思います。たとえば、2021年に行われた東京2020オリンピックにおいて、公式チケットを販売するサイトがありました。これは、事前に、たくさんの人が買いに来るだろうってことは分かっていたわけですけれども、一体どれだけアクセスされるかは実際にはわからなかったわけです。ですので、どれだけ大きいサーバーがいるのだろう、どんなネットワークがいるのだろう、ということは実質的には予測不可能であったと思います。そこで、このオリンピックサイトでは、どういう手段をとったかというと、仮想待合室サービスというものを利用したのです。それで、何が起こるかというと、このような画面が出るわけです。実際にご覧になった先生方もいらっしゃるのではないかと思いますけれども、一定数のアクセスを超えたら、それ以降に来た人は強制的に待合室に入れられる、たとえばこの画面でいうと、あなたの前に146,003人いますよ、あなたの順番が来るまで1時間以上かかりますよ、というようなことを画面に出します。これが仮想待合室です。これは、典型的なgraceful degradationの例でして、要するに、もう一定数しかアクセスは同時に捌きません、それを超えるものについてはどうぞ外でお待ちください、というやり方です。これは、システムとしては、全然、機能縮小してないように見えるのですけれども、ユーザーから見た機能というのは、つまりネットで買えば、普通、「ちょっと待てばすぐ買える」程度に期待されていた機能が大幅に縮小しているのです。実際、何時間も何十時間も待ったという例があったと聞いています。

Software Extensibilityの例

- アドオン(Add-on)/アドイン(Add-in)/プラグイン(Plug-in)
  ⇒ソフトウェアに未装備の機能を追加する補助的なソフトウェア



ソフトウェア本体が、機能拡張を許容する設計になっている必要がある

Google Chromeの機能拡張(アドオン)

次にsoftware extensibilityの具体例です。この図はChromeというWebブラウザーの画面です。Chromeには、Chrome自体の機能を拡張するためのアドオンがあります。たとえばKeepaはAmazonの価格をトラックしてくれるアドオンで、自分がこの商品買いたいと思ったときに、Amazonの価格がどういうふうに変動しているかをお知らせしてくれるものであったり、他にもLINEのプラグインがあったり、といろいろなアドオンがあります。こういった機能は、本来Chromeには備わっていないのですけれども、後でこういうふうな拡張ができる、これがソフトウェアの大きく特徴、というわけです。もちろん、ソフトウェア自身がそういうふうな設計になっていないと、このような拡張はできませんが、ソフトウェアにはこのような特性があるということです。

改めてGraceful Extensibility

- 擾乱に対し、対応限界付近、あるいは、限界を越えて拡張する能力
- Graceful Degradationが機能縮小を意味するのに対し、Graceful Extensibilityは、限界付近での調整／適応によってポジティブな結果をもたらす可能性もある、という意味が込められている
- では、情報システムにおけるGraceful Extensibilityとは？
  - (事前準備された)ハードウェアに起因する制約を突破する方法とは

そこで改めて、この２つの言葉を組み合わせた造語であるgraceful extensibilityについて考えてみたいと思います。Woods先生の定義で言いますと、graceful extensibilityとは、擾乱、障害、危機に対して対応限界付近あるいは限界を超えて拡張する能力のことです。Graceful degradationという言葉が、単なる機能縮小を意味するということから、よりポジティブなextensibilityという言葉を組み合わせて、より良い結果をもたらすこともあるという意味が込められている、というふうに先生はおっしゃっておられました。

では実際に、情報システムでgraceful extensibilityにどういうものが当てはまるかということを考えてみようと思います。具体的に言うと、先ほどから出てくるハードウェアの制約をどういうふうに乗り越えていくか

ということがポイントとなります。

ハードウェアの制約

- 物理的に存在している
  - 電源に接続しなければならない
  - ネットワークケーブルを接続しなければならない
  - 電源ボタンでON/OFFしなければならない
  - 部品が故障してしまう
- ハードウェアが設置された場所に行かなければできないことがある
  - 機能拡張時：部品の追加、構成の追加
  - 障害発生時：部品の置き換え、構成の変更

そこで、具体的にこのハードウェアの制約が何かということを考えてみますと、当たり前ですけれども、物理的に存在してしまっているということであります。電源に接続しなければいけないとか、ネットワークケーブルを挿さないといけないとか、電源を入れるにはボタンでオンにしないといけないとか、部品が故障してしまうというようなこととか、ごくごく当たり前のことです。それで、問題は、その場所に行かないとアクセスできない、ということで、たとえば機能を追加したい、拡張したい、というときに、部品を追加したり、構成を変えたりするのに、そのハードウェアの場所に行かないと実現しないのです。もちろん、障害が発生した、何とかしなきゃ、というときに、部品を変えようと思ったとしてもやっぱりそこに行かなければならない。というふうに、この物理的な存在そのものが大きな制約になっているのがハードウェアの特徴です。

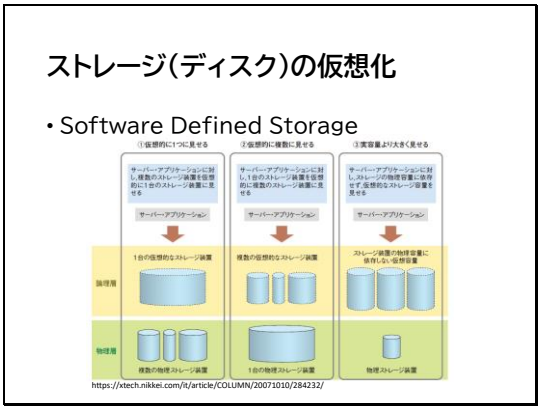
仮想化技術（Virtualization）

- ハードウェアの物理的な存在が隠蔽され、ハードウェアが提供する機能のみが利用できる状態
  - あたかもそこに“ある”かのように見える
  - しかし、物理的な実体の場所は問わない
- “ハードウェアのソフトウェア化”
  - ソフトウェア化により、ハードウェアにSoftware Extensibilityが生まれる
  - ネットワークケーブルの接続、電源のON/OFF、CPUの増強、ディスクの増量などの作業が、ソフトウェアの機能として提供される

そこで、情報科学技術として、もう20年以上前になりますが、仮想化という考え方が出てきました。で、これは一言でいうと、ハードウェアのソフトウェア化ということになります。ハードウェアの物理的な存在が隠ぺいされて、ハードウェアの提供する機能のみが存在しているように見える、つまりあたかもハードウェアがそこにあるかのように見える技術です。実際、どこにあるか全然わからない、日本かもしれないし、アメリカかもしれないし、シベリアかもしれない、そんな状況になります。結局、ハードウェアをソフトウェアにするということによって、ハードウェアに、

Woods先生が言うところのソフトウェア独自のextensibilityが生まれるということになります。

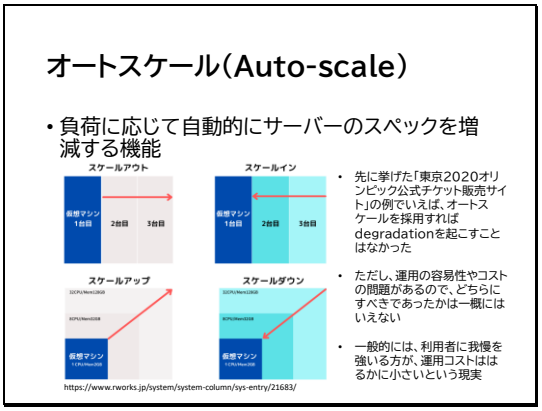
ここが非常に大きなポイントかな、というふうに私は論文を読んでいて思いました。具体的には、さっき言ったような、ネットワークケーブルを挿したり、電源を入れたり、いろいろな部品を交換したり、追加したりという作業が、ソフトウェアの機能として行える、場所によらない、ということになるのが大きなポイントです。



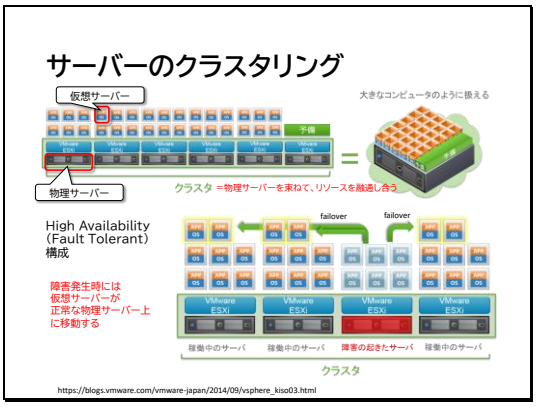
たとえば、ということで、ディスクを仮想化するという方法をご紹介したいと思います。これはsoftware defined storageといいます。ストレージというのは、ディスクだと思っていただければ結構です。図の下にある物理層が、実際の存在としてのディスクです。一番左の例でいうと、実はここに三本あるのだけども、これを大きな１つのディスクとして見せるという仮想化の方法です。たとえば、図①の左のディスクが1テラバイト、真ん中がレ0.5テラバイト、右が3.5テラバイトだとすると、これを足した容量の5テラバイトが１つのディスクであるかのように見えて、実際にそのように使えます。逆に、図②のように、１つの大きな物理ディスクがあって、それを分割して別々のサーバーに使ってもらうということもできますし、さらに進んで、図③のように、実はちょっとしか物理ディスク容量ないのだけども、たくさんあるように見せかける技術もあります。実際のシステムというのは、最初に使うディスク量はそれほど多くなく、使っているうちにどんどん増えてくるので、最初の段階では、要る分だけを与えて、足りなくなってきたら追加しよ

う、という考え方で、サーバー側から見ると、仮想化のおかげで、最初からフルボリュームあるように見えます。ということで、結局、ここの論理層レベルではもうソフトウェアになってしまっているということなのです。

実際のサーバーの仮想化の例をお見せします。これは本院で使っている仮想サーバーを管理する場面になります。画面左側にずらっと並んでいるのが仮想サーバーです。一つの行が一つのサーバーだと思っていただければよいです。ハードウェアだったら部品を足したり、蓋を開けたりとかしないといけないのですが、そういったものは全くいらなくて、画面上で「設定の編集」というところで、CPUは今２つだけど４つにしましょうとか、メモリは4ギガだけど、これを16ギガにしましょうとか、ハードディスクはちょっと少ないから増やしましょうとか、ネットワークは、今これにつながっているけど、別のものにつなぎましょうとかいったことが、全部この画面上でできます。ハードウェアがソフトウェアとしてしか見えないのですね。こういったことができるのがソフトウェアの非常に大きな利点であろうというふうに思います。



もう一つの利点が、オートスケールです。一般的に、サーバーに負荷がかかると処理しきれなくなってくるわけですが、その負荷量に応じて自動的にサーバーのスペックを上げたり下げたりする機能です。スケールアウトというのは最初1台しかないものを2台、3台と、自動的に増やしていく仕組みで、スケールインはその逆です。だから、負荷が高くなってきたなと思ったら、台数を増やしていき、もういらない、1台で十分だ、ということであれば減らす。別の方法として、マシンの性能を上げる場合に、１台のままでCPUを増やしたりメモリを増やしたりするのがスケールアップで、その逆がスケールダウンです。こういったことが、ハードウェアの追加をしなくても、もちろん最初にある程度大きい容量のハードウェアを用意する必要があるのですが、その容量の中でいろいろと調整ができるということです。だから先ほどの例であげました、東京オリンピックの例で言えば、このスケールアウト、あるいはスケールアップいう仕組みを採用していれば、degradationすることなく、皆さんがスムーズにお買い物ができたと思います。ただし、こういうことをするのには非常にお金がかかるので、それがよかったかどうかはよくわかりません。一般的には、ちょっとユーザーに我慢してもらいましょう、と考えるのが管理者の立場であって、その場合の運用コストはもう全然違います。ですので、東京オリンピックではそういう選択をしたということだったのだろうと理解しています。



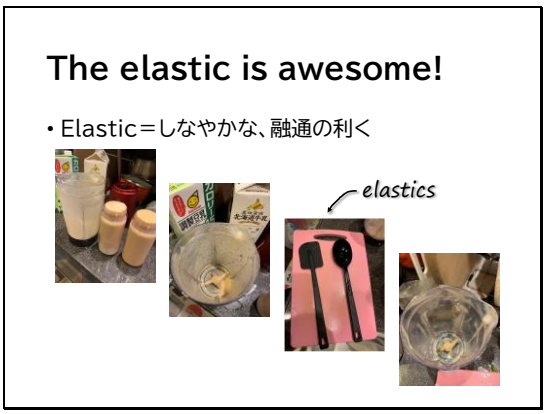
次に、サーバーのクラスタリングについてお話しします。先ほど言いました、サーバーを束ねるという技術です。図の左上にあるように、物理的なサーバーが６台あり、その上に仮想的なサーバーがたくさん載っている状況で、1台は空けておいて予備にしていたとしても、これはクラスターという技術によって、図の右上にあるように１台のサーバーに見えるのです。それで、この物理サーバーは「物理」ですから、いつかどこかが壊れるわけですけれども、たとえば、図の下の赤い物理サーバーが壊れたとしたら、この上に載っていた仮想サーバーたちはみんな空いているところによけていくのです。こういうフェイルオーバーをすることで、サーバー全体としては普通どおり動いていきます。それで、またこの壊れた物理サーバーを復旧させて元に戻せば、先ほど動いた仮想サーバーをまた元に戻すという作業をしてあげればいいということになります。こういったことが、レジリエンスの考え方にはフィットしている、つまり、gracefulなextensibilityなのだろうなというふうに考えました。

4. Sustained Adaptability

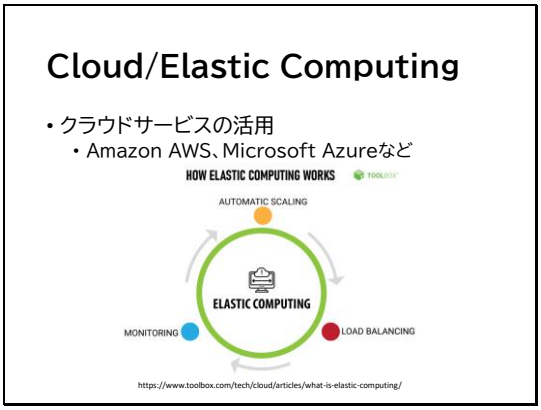
- Graceful extensibilityをどうやって長期にわたり持続するか
- 仮想環境でいえば、仮想インフラのスケールが大きければ大きいほど、擾乱に対する対応は容易
- しかし、ローカル環境(たとえば病院内)でそれほどのリソースを持てるのか？　維持可能か？

次に４番目なのですが、graceful extensibilityを持続するにはどうしたらいいかということです。情報システムの場合は、システムのライフサイクルが5年か6年、あるいは7年ぐらいなので、あまり長期のことを考えることがないのですが、あえて言えばどうなるかという仮定の話をします。たとえば、さっき言ったようなgracefulな仕組みを私のいる病院で持てるか、維持できるかというと、絶対にそれはできないです。ただ、お金をかけて、余力のある大きなサーバーを用意して、大きな仮想システムを作れば非常に

ハッピーであることには間違いがないし、sustainabilityも非常に上がってくるのは間違いありません。

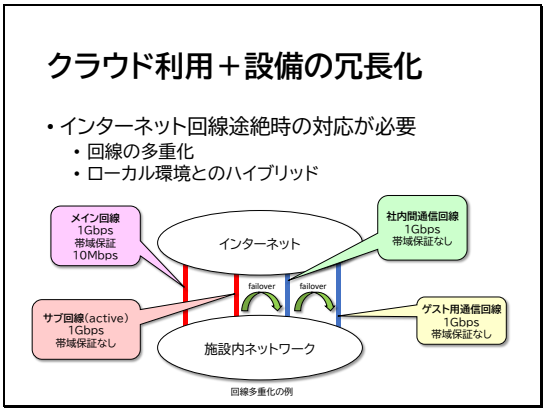


今朝、自宅のキッチンでこの写真を撮ったのですが、elasticという考え方があります。これは情報システムの世界ではよく使う言葉で、しなやかなとか、融通の利くという意味です。ここでは、このしなやかな、elasticというのはすごくいい、という話をしたいのですけれども、私は、毎朝、野菜ジュースを作っていまして、こういうふうに、ミキサーにかけるわけです。それで、ジュースをミキサーから容器に移し終えると、まだミキサーにジュースが残っている。これを何とかして取りたい、全部食べ尽くしたいのですけれども、普通の金属のスプーンでやっても全然ダメで、こういうゴムべらなどを使うと非常にきれいに、こんなにきれいに取れちゃうという話で、elasticってすごくいいなと、今朝、思ったのです。



なぜこの話をするのかというと、情報システムの世界では、まさにelastic computingという言葉がよく使われていまして、これがここ5年ぐらいで本当によく普及してきた技術になります。AmazonのAWSや、MicrosoftのAzureという仕組みは、先ほどご紹介したgraceful extensibilityを自動でやってくれるというところが大きな特徴です。この図にあるようなオートマチックスケーリングのことです。コンピューターの負荷が高くなったら自動的に性能を上げてくれたり、ロードバランス、要するに負荷が均等になるようにして、複数のサーバーに振り分けてくれたりしてくれます。常に負荷をモニタリング

していて、このサイクルを自動でぐるぐる回してくれるので、非常にハッピーなシステムですね。ここまでくるとAmazonや、Microsoftにお任せするということができで、非常にsustainableなシステムができるだろうというふうに考えます。



ただ、一つ問題は、クラウド環境というのは、どうしてもネットワークが繋がってないといけないので、実際に使おうとすると、回線を冗長化する必要があると言うことです。これは本院の例なのですけれども、本院では４つの回線を持っていまして、図の赤い線のメインとサブの回線をActiveに使っていて、ユーザーによってどちらから出て行くのかを決めて、うまく均等になるようにしています。この他に、「社内」というのはちょっと変な言い方ですけれども、他の拠点と通信するためや、ゲスト用として、それほど通信量は多くない回線も準備しています。もし、メインで使っている赤い線に障害起こった場合には、こちらにフェイルオーバーして、最終的には、このゲスト用の回線１本だけでもなんとか業務ができるという設計にしております。本院では、クラウドをまだ本格的には利用してないですけれども、今後、クラウドコンピューティングを使うようになってくれば、こういったところでも、やはりきめ細かな配慮というのが必要になってくるというのが現状です。

辺境警備隊（SNAFU Catchers）

- クラウドを利用したところで、障害ポイントは残る
- いずれにしても、情報システムのgraceful extensibilityを実現し持続するには、システムの周辺で、常に擾乱の有無を監視し、変化に気づき対応する人間（＝SNAFU Catchers）が必要
- クラウドの「中」にも、SNAFU Catchersがいるはず

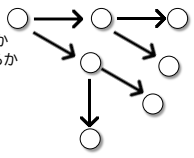
SNAFU Catchers: People admire the beauty of their plan in their world, while just at the edge is someone wrestling with dragons. (D. Woods)

https://www.youtube.com/watch?v=GnVXfgC-5Jw

これはWoods先生のYouTubeに出てきた図です。結局、いくらお任せするものがあったとしても、障害ポイントは絶対に残ってしまうので、それを一つ一つ拾い上げて、障害が起こったら、あるいは起こる前に、それに対処するという人間がどうしても必要になってきます。Woods先生は、SNAFU Catcherとおっしゃっていましたけれども、混乱を拾い集めて対処する人という意味です。私は、辺境警備隊というふうに名付けました。きっと、AmazonとかMicrosoftにお任せしたとしても、そっちの「中の人」にも、こういったSNAFU Catcherがいるはずです。この図が示していることは、自分の計画を見渡して、「何て素晴らしい世界なんだ」と思っている人がいる一方で、ふと世界の境界部分に目をやると、こういうふうにドラゴンと戦っている人がいる、こういう人をSNAFU Catcherというわけです。だから、こういう人たちがどうしても必要ですよね、という話になるのだろうと思います。これは私も実感としてそのように思います。

辺境警備隊の心構え

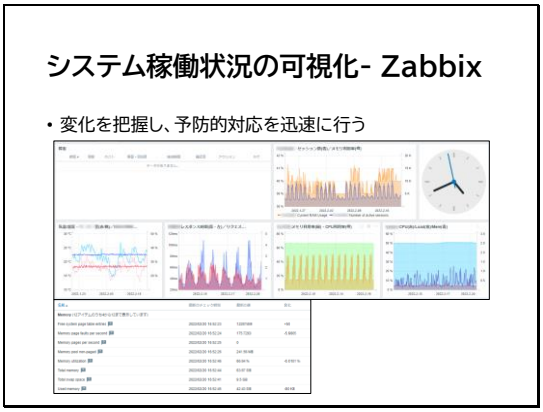
- 頻繁な介入による軌道修正
  - 早期発見・早期解決
  - 問題を放置すると依存先に影響⇒さらにその依存先に影響⇒・・・⇒指数関数的に影響先が増大⇒限界に近づく
- 努力のポイント
  - いち早く問題に気づくことができるか
  - いつでも・だれでも対応できるか
  - 対応者に十分な対応能力があるか



私自身も、自分のスタッフとともに、この辺境警備隊だというふうに思っているわけですが、その心構えとして、どんなことが必要かというと、頻繁に介入し軌道修正をすること、すなわち早期発見、早期解決ということが大事だと思います。問題を放置すると、もうご存知だと思いますけど、特に情報システムの世界では、一つのシステムが他のシステムに依存しているということがありまして、たとえば障害が起こったシステムが別のシステムに波及して、そのシステムからさらに別のシステムに影響が出て、と、指数関数的に影響が増えていくことが結構あるわけです。そうすると、

限界というのはあっという間にやってくる。最初に小さな問題だと思っていたものが、介入をしなかったために大きな問題になって、もう取り返しがつかなくなる、ということを避けたいということです。

ですので、われわれの使命は、いち早く問題に気づかなければならない、いつでも、誰でも対応できるようにならなければならない、対応するのに十分な能力がなければならない、というふうに思っております。



そこで、実際にわれわれが、SNAFU Catchersとして、使っているツールをいくつか紹介したいと思います。まず、状況変化を把握した上で、予防的対応を迅速に行うために、やはり監視が必要です。ここに示す、オープンソースのZabbixというツールを使って、たとえばサーバー室の温度、Webシステムのレスポンス、メモリの使用量、CPU、あるいは院内からインターネット回線で利用しているセッションの数（同時接続数）などをモニタリングしています。当然、基本的な、ディスクの使用量とか、メモリの使用量なども全部監視をしています。これらには閾値を設定して、閾値を超える問題が発生すれば、画面左上の部分に障害として上がってきますので、それを見てすぐに対応するということで、たとえばサーバーが停止するといった事態を防ぐ取り組みをしています。

課題/タスク管理 – Atlassian Jira

・課題トラッキングし、対応ナレッジを蓄積する

また、タスク管理には、Jiraというソフトウェアを使っています。課題をトラッキングして、対応した場合のknowledge、これは暗黙知になりがちなのですけども、そういったものを蓄積していくためのものです。ですので、何か問題が発生して、対応したということであれば、コンポーネントというカテゴリーに分けて入れていただくと、課題の状況がどうか、いつ解決したのか、担当は誰かということをデータベース化しています。それで、何かあったときに、過去に同じようなことがあったかということも、Jiraを見てわかるというような仕組みです。

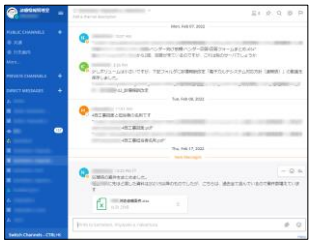
対応手順の一元管理 – Atlassian Confluence(Wiki)

・業務の属人化を解消し、標準化と効率化を図る

さらに、対応手順の一元管理ということで、Wikiの一種なのですが、Confluenceというソフトウェアも使っています。いわゆるマニュアルの管理です。われわれのところでは、だいたいの業務規範というのを決めていまして、どんな業務をするのか、その手順は何かというのを、基本的にはすべてここに入れていただくということにしています。Jiraの方は、どちらかというと、時的な対応で終わってしまいがちなのですけれども、定型的な業務を含めて、もう決まった対応があるのであれば、ここにマニュアルとして整理していきましょうということで、業務の属人化を解消して標準化と効率化を図るということを目的としてやっています。

報告の迅速化
- Mattermost(Chat)

- 気軽/手軽に報告し、迅速な情報共有を図る



さらに、コミュニケーションツールとして、チャットも使っています。これは院内だけで使えるものではあるのですが、メールで報告するというのはなかなか面倒くさくて、いちいち宛名を入れたり、タイトルを付けたりしなきゃいけないので、それが障壁となって報告が後回しになったり、できなかったりするのを避けたい。ですので、できるだけ手軽に、情報共有できるようにということで、スタッフを全員登録して、お互いにチャットでやりとりしたり、共通のチャンネルで全員に連絡したりすることができるようになっています。こういったツールを使いながら、われわれはSNAFU Catchersとしてがんばっています、というお話です。

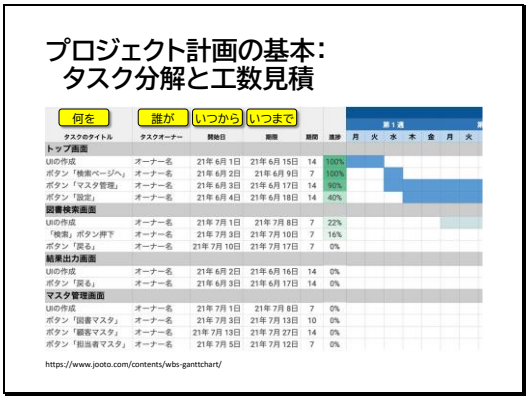
5. ソフトウェア開発における
graceful extensibility

- 大規模なソフトウェア開発はプロジェクト体制で進行する
- プロジェクト初期に立てた計画(人員配置、スケジュール、成果物の内容など)は、しばしば擾乱(仕様変更、見積時間の延長など)にさらされ、計画変更を余儀なくされる
- これらをなんとか融通をつけて、プロジェクトを完了させるのがプロジェクト管理の目的

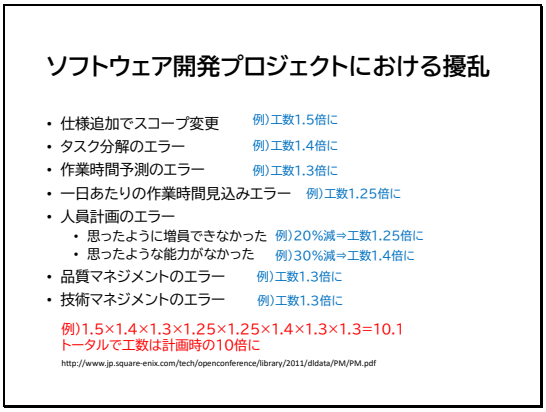
最後の話題ですけれども、ちょっと話が変わります。今まではできあがったシステムに対する管理というのがメインの話題でしたが、今度は、ソフトウェア開発、というシステムができるまでの話になります。これもいろいろと難しい問題があって、graceful extensibilityという考え方で対応しているというのが現状なので、そこをお話ししようと思います。

大規模のソフトウェアになると、開発は一人や二人でできるものでないので、何十人、何百人というプロジェクトでやることになります。このプロジェクトの初期に、計画を立てるわけです。何のタスクに何人を割り当てて、どんなスケジュールで、どんなソフトウェアができる、という計画を最初にするのですが、お客さんが何か急に変なこと言ってきたとか、しばしばいろいろな要因が発生して、見直しを迫られる、それを何度も何度もしなければいけない状況に陥ります。これをうまく処理

しないと、プロジェクトが破綻して目的の製品ができないということになります。ですので、これを融通する、つまりelasticな考え方を使って、プロジェクトを完了させるというのが、プロジェクト管理の目的になります。



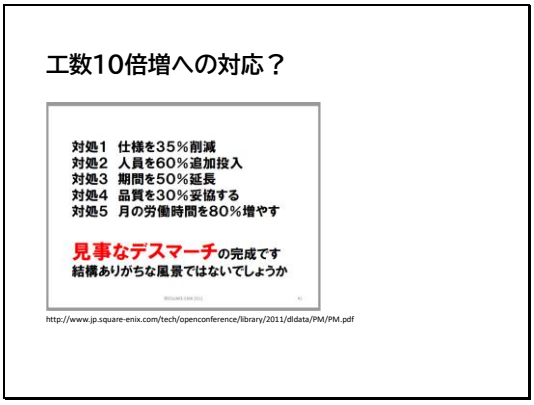
よくあるのは、このようなガントチャートを作ることです。一体どんな作業が工程として存在して、それを誰がやるのか。１人のこともあるし、チームの時もあります。それで、いつからやっていつまでで終わる見込みなのか、ということです。図の上の方の部分はすでに進捗100パーセントになって終わっている工程ですが、最初には、これらの工程をすべて見込みとして入れていって、各工程がスケジュールどおり終わるかどうかというようなことを確認していく作業をします。たとえば１年後に完成しますよ、というような計画となるわけですが、たいていそうはいかないです。



どんなことが起こるかというと、例えば、スライドにある仕様追加では、お客さんが「やっぱりこんな機能も欲しい」と途中で言ってくるようなことが起こります。タスク分解のエラーというのは、たとえば、想定してなかったタスクが出てきた、というような、先ほどのガントチャートにある一つ一つの工程を正確に把握できていなかったと言うことです。あと、ガントチャートの「期間」に書いてある、何日で完了する、という予測が間違っていたとか、１日あたりにこれぐらいできるだろうと思っていたのが、実はできませんでした、とか、途中で増員するつもりが、増員する人がいませんでした、とか、増員でやってきた人に十分な能力がありませんでした、とか、マネジメント

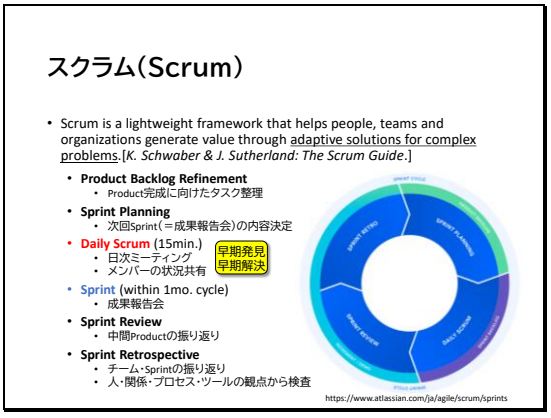
にエラーがありました、という話もあります。

　このスライドは、スクウェアエニックスというゲーム会社のシステム開発を担当されていた方のスライドを基に作ったのですけれども、たとえば、それぞれの工数が1.5倍だ、1.4倍だ、1.3倍だとなるとします。工数というのは、かかる日数というふうに思っていただければいいのですが、結局、これらを全部掛け合わせたものがトータルで効いてくるので、それぞれは小さいと思っていても、トータルで10倍になります。10倍の労力が必要になるものを、同じ納期、たとえば１年なら１年で完成させられるかというと、それは無理だろうと。



　では、プロジェクトマネージャーはどうするかというと、これは、冗談ですけれども、たとえば、仕様の35パーセントを諦めてお客様に泣いてもらいましょう、人をあと6割増やしましょう、１年って言っていた期間を１年半にしましょう、チェックの回数を減らして3割ぐらい品質を削減しちゃいましょう、クオリティを落としましょうと。あと、みんなにがんばって残業してもらって、残業時間を1.8倍にしましょう、とか。ここまでするとようやく10倍になるのですが、これはもう、いわゆるデスマーチです。ここに、「ありがちな」と書いていますから、実際ソフトウェア開発ではうまくいかないことの方が多くて、こんな光景はざらにあるということなのだろうと思います。

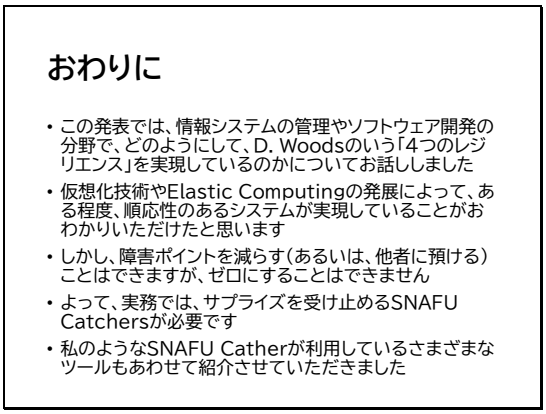
では、どうすればいいのかというと、一番の肝は、どの工程にどれだけ時間がかかるかということの誤差なのです。この写真にあるような、プランニングポーカーというカードがAmazonでも売られています。このタスクに何日かかりますという予測は、マネージャーが一人で決めたり、作業する人に何日かかるって言われたまま決めたりすることもあります。けれども、そうではなくて、関係する人集まって、全員で決めましょう、と、あたかもポーカーをするかのようにカードを出し合って、その平均値を取ったり、最大値と最小値を取ったりとか、こういう場で決めていくというやり方があります。非常に面白いやり方ですね。



また、われわれも活用しているのが、このスクラムという考え方です。これはフレームワークって書かれていますが、大げさに言うと確かにそうなのですけども、実はミーティングのやり方の話なのです。英語部分の下線部分にあるように、複雑な問題に対するadaptiveな解法を通じて人やチームを助けるフレームワークですよ、ということになっています。具体的には、青字部分のスプリントというのをやります。これは大体１ヶ月サイクルで成果発表会をする。ものを作ると１年、２年かかるプロジェクトをいきなり長いスパンで見るのではなく、細かく分けて１ヶ月ごとにまとめていくということです。どういうふうなプロセスがあるかというと、まずproduct backlog refinementという作業があり、これは要するに、製品を完成させるのにどんな作業が残っているのか、残作業整理をするということです。次に、スプリントプラニングで、1ヶ月先のスプリントでどんなものができて

いるはず、ということを決めます。デイリースクラムというのは毎日のミーティングのことで、メンバーが自分の状況、たとえば、今ちょっと進んでいますとか、遅れていますということを発表する、15分ぐらいの簡単なミーティングのことです。これはわれわれもやっています。ソフトウェア開発をやっているわけではないのですけれども、日々の進捗確認は非常に大事なので、朝礼のような形で各メンバーに発表してもらって、自分が持っているタスクの状況を確認して、問題があれば、そこで解決の道筋をつけるというようなことをやっています。

それで、スプリントでは、1ヶ月ごとにあらかじめスプリントプラニングで決められた成果を報告する。されに、それをレビューして（sprint review）、振り返って（sprint retrospective）、変更があればまた調整してというふうに戻っていきます。PDCAサイクルのソフトウェア開発版です。ただ、これに限らず、何にでも使えると思うのですが、われわれでいえば、たとえば、今はないですけども、システム更新するということになれば、具体的な成果を積み上げていかないと、リリースが間に合わないということになるので、たぶんこういったスプリントをすることになろうかと思います。われわれは、こういったものを活用して、あがいて、やっているわけですけれども、こういったものが、Woods先生のいうところのextensibility、gracefulなextensibilityの助けになるのではないか、というふうに思った次第です。



最後のまとめですが、この発表では、情報システムの分野では、Woods先生の４つのレジリエンスの考え方をどういうふうに実現しているのかについてお話をしました。そして、最近は、仮想化とかelastic computingという技術が出てきて、ある程度、順応性のある仕組みが実現しているのはないかと思います。

しかし、障害ポイントを減らすこと、あるいは他者に預けることはできるのですが、ゼロには絶対にならないということで、実務では、こういった障害とか擾乱を受け止めるSNAFU Catchersが要ります。また、私たち自身がそういう立場であるということです。そして、その私たちが、どんなツールを使っているのか、ということをご紹介させていただきました。