国立情報学研究所 市民講座資料（平成 24 年 2 月 15 日）

ソフトウェアの品質保証

－作るより検査が難しい－

中島　震
国立情報学研究所

１　はじめに

ソフトウェアが我々の身の回り、様々なところで使われている。携帯電話やスマホ、銀行 ATM、自動車、鉄道の切符自動販売機や自動改札、TV から洗濯機までの家庭電化製品などがある。しかし、ソフトウェアの存在に気が使いないことも多い。

これらの製品やシステムは「普通に」使っている限り、不具合に陥ることはない。製品出荷やシステム稼働の前に十分な検査を行って、不具合を予め取り除く。一般の工業製品と同様に、ソフトウェアについても品質保証に注意を払っている。

しかし、「普通でない」特殊な状況で、大きな障害を引き起こした事例が報告されていることも事実である。つまり、作ったソフトウェアに隠れていた不具合が、事前検査の網をすり抜けた。ソフトウェアを作ることと、検査が全く別のことになっている。工業製品としての未熟さを感じる。

ソフトウェアは見たり触ったりできない。エンジンの音がおかしい等、わかりやすい兆候が現れることは少ない。天変地異のように「突然」不具合が発現する。

事前検査では、「普通でない」特殊な状況まで含めて検査して品質保証しなければならない。ところが、「普通でない」からこそ検査が難しい。検査の状況を作れないことさえある。システム稼働中に「普通でない」状況に遭遇しないことを祈るしかないのだろうか。

ソフトウェアが中心的な役割を果たすシステムでは、「普通でない」状況が起こり得る理由が数多くある。それは、ソフトウェア自身あるいはソフトウェア開発の特徴と強く関係している。現在の「最善の実践（Best Practice）」では 100％品質保証することは困難である。

２　ソフトウェア不具合の事件

ソフトウェアの不具合が社会基盤や日常生活に大きな影響を与える。具体例として、TV ニュースや一般紙で報道された事故２件を紹介する。

２－１　大手３銀行の統合

2002 年 4 月、3 つの大手銀行（D、F、K と呼ぶ）が統合した銀行 M が発足初日に障害を起こす事件があった [11]。ATM 障害、

口座振替が不能、二重引き落とし、に加えて顧客の口座データ消失という二次被害が発生した。また、システム改修に携わっていた技術者が激務によって過労自殺し、2003 年に労災認定された。

直接の不具合は、システム統合のために改修したプログラムのバグ（特殊な条件で表面化）だった。大規模データによる負荷テストや不正データを用いた異常検出テストを実施しないままシステム稼働したことが社会的な問題の原因となった。

２－２　内部被曝検査機

Fukushima の原発事故に起因する被曝が大きな問題として関心を集めている。内部被曝検査機に関連する記事[3]にプログラム・バグ関係の話題が掲載されていた。バグに関連する話を要約する。

「バス移動式ホールボディーカウンタ（WBC)」という機器で測定した放射能量の値を調べた医師がおかしいことに気付く。原因は、放射能の種類ごとに体内に存在する量を推計するプログラムのバグだった。「ほとんど使われていないのでバグはわかりませんでした」とメーカー担当者が説明した。

２－３　共通する問題点

２つの例は、いずれも、「作ることは作ったが、正しく作動するかの検査が不十分だった」と説明されている。では、どのようにして検査すれば、不具合がないことを確認できるのであろうか。不具合がないことが本当に示せるのであろうか。

３　プログラム・テスティング

ソフトウェアの実用的な検査の方法はテストである。ソフトウェアの「実体」はプログラムであり、これを検査するので、プログラム・テスティングという。

３－１　テストする観点

プログラム・テスティングによるソフトウェア検査を考える前に、どのようにしてソフトウェアを開発しているかを整理しておく。図１に「Ｖ字型モデル」と呼ぶ開発過程の説明図を示した。

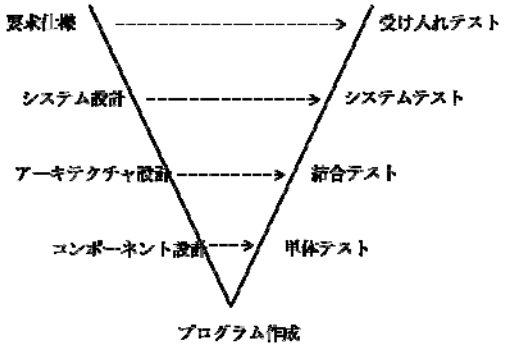


図１　開発過程のＶ字型モデル

左側のエッジ（バックスラッシュ）は開発上流工程と呼ばれ、ソフトウェアを具体化、詳細化していく過程である。その最終段階でプログラムを作成する。その後、右側エッジ（スラッシュ）のテスト工程を経て、品質保証がなされ最終的な成果物（不具合のないプログラム）が得られる。

まず、開発の進め方の概要（バックスラッシュ側）を説明する。最初に、利用者に提供する機能やサービスならびに期待する実行時性能などを検討する［要求仕様］。次に、要求仕様を実現するシステムの構成や仕組

みの概要を検討する［システム設計］。さらに、システム全体のデータの流れや制御の仕組みを具体化していく［アーキテクチャ設計］。最後に、プログラム作成の単位（図１ではコンポーネントと呼んだ）を決め、コンポーネントの詳細を決定する［コンポーネント設計］。ここまでに作成するものは「設計書」や「仕様書」であって、プログラムではない。最後に、「コンポーネント設計書」をもとにプログラムを作成する。

当然であるが、以上の作業は、基本的に、技術者が行うものである。各設計書に誤りがないかの検査は技術者がレビュー（文書の読み合わせ）する［仕様書レビュー］。「要求仕様」の項目が「システム設計書」に適切に盛り込まれているかといった工程間をまたぐ追跡性（トレーサビリティ）に留意しながら作業を進める。

現状、上記の仕様書レビューが、バックスラッシュ側での品質向上手段である。多くの担当技術者が参加しての長時間会議となることがレビュー作業のコストになっている。仕様書の書き方を標準化するなどの工夫をしているが、基本は技術者による読み合わせ検査である。この作業を効率化、軽減するような新しい技術の登場が期待されている。

プログラムが完成すると、スラッシュ側のテスト工程に移ることができる。テストはプログラムを対象とした検査であり、プログラムが完成して、はじめてソフトウェアのテストを行うことができる。

テストは小さい単位から少しずつ確認しながら対象範囲を広げていく。最初に、「コンポーネント設計書」通りにプログラムが作られているかの「単体テスト」を行う。個々のコンポーネント単独のテストに合格すると、コンポーネント組み合わせの「結合テスト」を実施する。組み合わせた場合の振る舞いは「アーキテクチャ設計書」に規定されている。同様にして、「システムテスト」を経て、最終的な利用者にみえる機能やサービスが規定の性能で実行できることを確認し、「受け入れテスト」が終わる。

なお、Ｖ字型開発モデルは説明の図であって、現実のソフトウェア開発が必ずしも、この通りに進むわけではない。他の開発法であっても、以下が共通する。

(1) 開発工程ごとに決めるテスト項目によって、様々な観点から検査する。
(2) 検査の対象はプログラムであって、設計書や仕様書ではない。

一般にソフトウェア開発は時間との戦いである。限られた期間内にシステムを完成させなければならない。そもそも十分な開発期間をとれない、図１の要求仕様やシステム設計で大きな見落としがあった、などの理由によって、その後の開発工程に時間的な余裕がとれなくなることがある。その場合、どうしても、プログラムを作り上げることが優先されて、テストにかける時間が圧縮される。

さらに、期間内にプログラムを完成させようとして過度な勤務体制に陥ることもある。疲労から初歩的な誤りばかりが増加し、ケ

アレスミスだらけのプログラムになるかもしれない。品質の悪いプログラムはテストの手間も大きいし、最悪、作り直しとなる。このような悪循環を「死の行進（デスマーチ)」と呼ぶ。

さて、第２－１節の事例では「システムテスト」や「受け入れテスト」が疎かになっていたのであろう。第２－２節の事例は、新聞記事からは、どこが問題なのか明らかでない。しかし、計算結果がおかしいというのは「単体テスト」か「結合テスト」で発見すべきことのように思える。

３－２　検査の網羅性

プログラム・テスティングでは、プログラムを実行して、その結果が期待通りであることを確認する。本節の以下では、最初に実施する「単体テスト」の場合を想定して説明する。

プログラムを実行するには、入力データを与える必要がある。「単体テスト」の場合、「コンポーネント設計書」記載の仕様をもとに、入力テストデータを作ればよい。

一般に、与えるデータ値によって、プログラム内の実行個所（実行経路）が変わる。もし、すべての実行経路を検査してバグがないことがわかれば、プログラムに誤りがないと結論つけることができる。

しかし、一般に全ての実行経路を調べることは不可能である。また、一組の入力テストデータに対しては、ひとつの実行経路が検査できるだけである。たとえば、プログラム本体中に繰り返し文が場合を考える。入力データに応じて、繰り返し回数が変わるような記述になっているとする。何回繰り返し実行する場合をテストすれば、完璧といえるのだろうか。

そこで、すべての実行経路をテストすることは諦めざるを得ない。プログラム中、１回でも実行して検査した場所は「検査済み」と見做す。近似的な検査の方法を採用するのが現実的である。たとえば、プログラムの実行文が一度でも検査できれば良い［文網羅］、条件分岐文の真と偽の各々が一度でも検査できれば良い［分岐網羅］、といった網羅性の基準が使われていた。最近では、もう少し詳細な網羅性基準を用いることが、安全性を重要視する航空機産業界の標準として推奨されている。

驚くほど大胆な近似を行っていると思われるかもしれない。いずれにしても、日常の言葉としての「検査の網羅性」とは異なる。それでも検査として有効なのは、プログラムに故意にバグを入れようなどとしないからである。また、単体テストによって不具合が沢山出てくるプログラムは「不良品」として棄ててしまう。すなわち、作り直すのが普通である。

３－３　正しさの基準

プログラム・テスティングで行うことは、予め決めたテスト項目に合格するかの検査である。テスト項目は「入力データに対して正しい結果」として整理される。一方、「正しい」と一口に云っても、「何が正しさの基準なのか」は明らかでないことも多い。

英語の学習を思い出してみよう。単語のスペルが正しいか、作った英文が文法に合っているか、といった基本的な正しさの基準があった。プログラムにも、このような機械的に決まる構文的な正しさの基準がある。プログラムを記述するプログラミング言語は人工言語であることから、構文的な正しさを検査する規則を整理し、自動的に検査することが可能である。

一方、構文的に正しくても、内容が正しくない文もある。「内容の正しさ」を判定するには、文を眺めていてもわからない。別途、正しいか誤っているかを決める情報が必要となる。たとえば、”His name is John.”が正しいかどうかはこの文からはわからない。

文の内容と同様に、プログラムが正しいか否かは、別途、正しさの基準（図１参照）を与える「テスト項目」が必要になる。「テスト項目」は技術者が作成する。機械的に決まるものではない。正しさの基準そのものが疑わしい場合、プログラムの正しさを判定することは無意味になる。品質保証の観点からもバックスラッシュで誤りを混入しないことの重要さがわかる。

３－４　ダイクストラの言葉

ＡＣＭ（アメリカ計算機学会）のチューリング賞がコンピュータ関係のノーベル賞と云われている。これを受賞した E.W. ダイクストラ氏が 1972 年の記念講演[4]で次のように述べた。

> 「プログラム・テスティングはバグがあることを示す効果的な方法になりえるが、バグがないことを示すのは絶望的である。」

現実に産業界で実践されているプログラム・テスティングによるソフトウェア検査は、正しさを保証する技術にならないのである。

４　規模の爆発

プログラム・テスティングが難しい理由のひとつとして、検査対象規模が大きいことがある。

４－１　規模の比較

ソフトウェアの実体はプログラムである。また、プログラムはプログラミング言語と呼ぶ人工言語で書かれた記述である。記述という点で、小説や新聞記事とった他の「書き物」と同様に、文字数や行数で規模を表現する。プログラムの場合、一般的には行数で規模を表す。

代表的なソフトウェアとして、スマートフォン（スマホ）のアンドロイド（Android）を考える。2008 年 11 月に公開された最初のバージョンのプログラム規模は約 800 万行で、そのうち約 160 万行がアンドロイド専用として新規に開発された。残りの 640 万行は 2008 年時点で既に開発済みで公開されていたプログラム（オープンソース）を使っている。また、スマホに組み込まれているプログラムはアンドロイドに加えて Linux オペレーティングシステムから構成されている。Linux も同規模（約 800 万行）という。なお、現時点ではアンドロイド 4.0

であり、さらに規模が大きくなっているだろう。おおざっぱに、2000 万行としておくことにする。

ジャンボジェット機（B747-400）の部品点数は約 600 万といわれている。部品 1 個をプログラム 1 行に対応させてみると、スマホの規模が大きいことがわかる。

マルセル・プルーストの「失われた時を求めて」は最も長い長編小説としてギネスブックに 2004 年まで掲載されていた。960 万 9000 文字（フランス語）という。英語の場合、4.5 文字を 1 語に換算する。同じ換算率と仮定すると、約 200 万語になる。1 語をプログラム 1 行に対応させると、「書き物」としての実用的なプログラム規模（スマホ）はギネスレベルの長編小説の 10 倍である。如何に長大であるかがわかる。

４－２　開発期間の長さ

規模が大きいと作成期間も長くなる。小説の場合でもソフトウェアの場合でも同じである。前者は、多くの場合、小説家本人が 1 人で作業するのに対して、後者では大人数の技術者が関わる。

「失われた時を求めて」はマルセル・プルーストが、1913 年から 1927 年の 14 年間、執筆に関わっていた。単純計算すると、1 ヵ月に 57000 文字（12000 語）になる。

途中でプロットの変更などを行った。たとえば、登場人物の変更があるとしよう。小説の大半が執筆済みの状況での書き直しになる。たとえ改訂内容は少しであっても、土台の記述が大規模であれば、辻褄の合った修正を適切に加えることは大変な仕事になる。このような変更は、ソフトウェア（図１）の場合で考えると、要求仕様の変更になるかもしれない。開発の途中で、突然、要求が追加変更される状況に対応する。

さて、このような変更に対しては、作成済みの小説 1000 万文字の至るところに追加、修正が必要だったと想像できる。この作業を進める方法として、タイプ原稿に余白がなくなると付箋（パプロル）を張り付けて加筆していた。加筆修正が「ごちゃごちゃ」という実体をよく表している。

図２　パプロル[15]

大手銀行勘定系システムの規模は 1 億行（10000 万行）に達する。1 人の技術者が 1 ヵ月で開発できるプログラムは 500 行～800 行と云われている。ここで「開発」という言葉には、「プログラム作成と（単体）テスト」を含めるのが普通である。単純計算すると 1 億行のプログラム開発には 1000 人の技術者が 10 年かかる。

プルーストの小説では、どのように話を展開するか、といったアイデアは、小説家 1 人の頭の中にあった。個人の才能と工夫に

よって、辻褄のとれた小説を完成させる。

ソフトウェアの場合、多数の技術者が関わる。ちょっとした変更であっても関係者全員に正しく伝えることが難しい。最初の決定通りに、プログラム作成後の要求変更などがないようしたい。しかし、長い開発期間であるから、細かな調整は必要であろう。バックスラッシュ（図１）が左上から下へ滑らかに進むというのは現実にはあり得ない。長期間にわたる大規模システム開発では、後戻りの管理や、パプロルのような「辻褄合わせ」の作業を工夫する方法が本質的な役割を果たす。

ソフトウェアは、開発期間が長いことから、当初予測しなかった変更が加えられることを想定したシステム開発の方法が不可欠である。変更にうまく対応できないと、既に規模の点から複雑な大規模システムが、スパゲッティが絡み合うように、さらに複雑化していく。エントロピーは増大する一方である。

５　複雑さが支配

５－１　統合システムの障害

第２－１節で紹介した統合システムの不具合原因を説明する。以下は入手可能な情報[6][11]をもとに再構成した。

図３にＭ銀行統合システムの構成概略を示した。もともと独立した３つの銀行（Ｄ、Ｆ、Ｋ）が経営統合した。図３では、各々の勘定系システムを旧Ｄ等と記した。

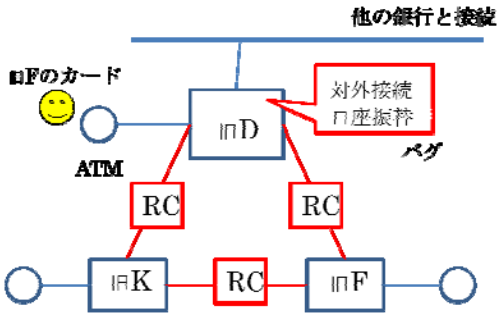


図３　統合システム構成

Ｍ銀行のシステムは旧Ｄ勘定系を中心に構成すると決定された。図３のように、旧Ｄシステムが他銀行と情報交換する外部ネットワークに接続する。統合前は独立した銀行だったＤ、Ｆ、Ｋの銀行口座は、統合後、他銀行からはＭの口座として見える。

銀行の預金者は、たとえば旧Ｆ銀行のキャッシュカードをＭ銀行で使いたい。図３では旧Ｄ銀行の支店ＡＴＭを預金者が使う場合である。旧Ｄ銀行システムには、この預金者の口座はないので旧Ｆ銀行システムに情報を送る。そこで、３つの旧システム間にＲＣ（リレーコンピュータ）を導入し、情報交換を可能にすることになった。

ＲＣの導入によって、３つの旧システムの大部分を流用することができる。その結果、Ｍ銀行勘定系システムを新たに開発する場合に、比べて大幅な開発コスト削減が可能と見込める。ＲＣに関わる処理だけを追加修正すればよい。

外部と接続する旧Ｄ銀行システムは他銀行からの振り込み口座振り分けといった機能も必要になる。したがって、３つの中では、

旧Ｄ銀行システムの修正が最も大きい。他に比べて、同じ期間での修正が大きく多忙を極めたことが想像できる。また、「対外接続系」を他銀行との情報交換に加えてＲＣとの情報交換処理も行うようにした。情報交換ということで共通の処理と考えたのであろう。

統合の日に合わせてプログラム開発を終えて、いざ、開業の時、ＡＴＭが使えない、他銀行からの口座振り込みができない、といったシステム障害が発生した。

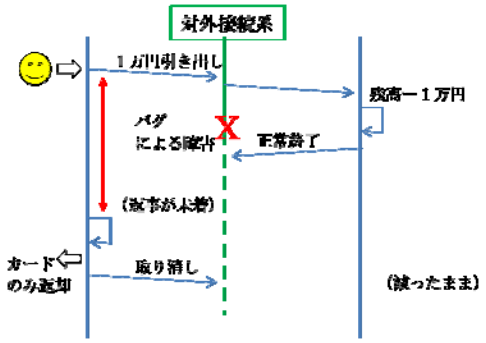


図４　バグによるシステム障害

図４はシステム障害のきっかけとなった処理列を例示したものである。直接的な原因は、旧Ｄシステムに追加した「対外接続系」にバグが残っていたことである。

顧客がＡＴＭから現金引き出し要求をした。その情報は対外接続系を経て旧システムが管理する口座情報を更新した。残高が減らされ、正常終了を示す戻りメッセージが送られた。このようなＡＴＭ処理や他銀行からの処理が同時に発生すると負荷が高くなる。おそらく高い負荷という「普通でない」状況で「対外接続系」のバグが表面化した。つまり、バグによって戻りメッセージを正しく処理できなかった。メッセージは未処理として蓄積される。

通常、メッセージ通信を行うプログラムは、相手があることから、何らかの理由で通信が失敗する可能性を考慮する。タイムアウト処理を行い、戻りメッセージを永遠に待つことがないようにする。図４の場合、戻りメッセージが到着しないことをタイムアウトで検知し、処理失敗が起こったと判断した。現金引き出し要求の取り消しメッセージを送信する。一方、ＡＴＭ端末では、現金は出さない。

「対外接続系」はバグでダウンしているので、この取り消しメッセージも伝えることができない。先と同様に未処理として蓄積される。その結果、預金者は現金を入手できない一方、口座の情報は減らされたという状況になってしまった。また、多くのメッセージが未処理のまま溜まった。

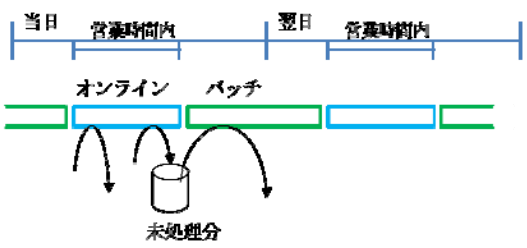


図５　口座振替

Ｍ銀行の問題はこれだけではなかった。図５は旧Ｄシステムの処理を示す。このシステムでは他銀行との口座振替に、2 つの方式を併用していた。営業時間内はオンライン処理で１件ずつ即座に処理する。一方、大量の依頼があった場合、あるいは午後遅

くの依頼など営業時間内の未処理分は一時蓄積しておき、夜間にバッチ処理（一括処理）する。次の日の営業時間開始までに、バッチ処理を完了すればよい。

しかし、現実には、夜間のバッチ処理が完了できないほど、大量の未処理依頼が溜まっていった。図３の口座振替プログラムが不正データを正しく判定できないというバグもあったという。これを解消しようとして、さらに、問題が連鎖的に起こり、１ヶ月間にわたって障害が続いた。

この統合システムの不具合の事件は、銀行が主役だったこともあり、社会的な問題としてビジネス書でも取り上げられた。ＲＣを導入したシステム統合の方針は誤りではなかったが十分なテストを行わなかったこと、見切り発車で経営統合の日を迎えたこと、が問題の根源であり、その原因は経営層の情報システム軽視である、という論調が主であった。

たしかに、経営層の考えの甘さが大きな原因であることは同意できるが、本当に、ＲＣの方法は妥当だったのだろうか。さらに、時間があれば、本当に十分なテストができたのであろうか。テストではシステムに不具合がないことを示せない（ダイクストラの言葉)。

５－２　ホ―アの言葉

第５－１節の事件の原因は、システムを必要以上に複雑にしてしまったから起きたというのが本当ではないだろうか。

ＲＣの導入や口座振替の修正は、プルーストの小説で登場人物が増えた場合に似ている。たとえ全体の修正規模が小さくても、至るところに修正が及んだかもしれない。ほんの少しの修正であっても、改修箇所のあったプログラムは全て再検査しなければならない。それも、従来は３つの銀行にあった銀行口座が全てＭ銀行口座となる。外部からの情報が全て正しいとは限らない。予期しないデータの到来を考慮したプログラムの作りと検査が必要だったはずである。

一般に、通信に関わるプログラムは、通信相手の状況によって適切な処理を行う必要があることから、多くの場合分けに対応する。その結果、テストしなければならない状況の組み合わせが増える。テストが難しいプログラムである。

旧Ｄシステムの改修は、必要に応じて、新たな建屋を作り、母屋と細い廊下でつなげて拡張していく温泉旅館のようである。たしかに、母屋の大部分はそのまま流用でき、修正は細い廊下を追加工事する個所だけかもしれない。その結果は、趣のある謎めいた迷路を持つ温泉旅館だろう。情報システムには、そのような趣は不要である。さらに、温泉旅館でも、複雑化することで緊急時の避難に支障をきたすかもしれない。

最近、新しい名所として、東京スカイツリーが話題を集めている。東京都内で高層ビルが増えたことから、「低い」東京タワーでは電波障害が増えてきた。そこで、さらに「高い」ＴＶ電波塔が必要になる。東京タワーが低いことが問題だとして、図６右の

ような「2 階建て継ぎ足し」を考える人はいない。十分な時間と開発コストをかけて、新しいタワーを建造する。

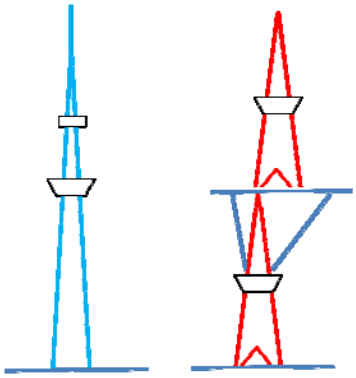


図６　東京スカイツリー

複雑さへの対応の失敗が不具合を招くことを、C.A.R. Hoare（当時オクスフォード大学、現在マイクロソフト研究所）がチューリング賞講演[7]で次のように述べている。

> 「ソフトウェアを設計する 2 つの方法がある。ひとつは単純にすることで明らかに欠陥をなくす方法であり、もう一方は複雑化させて明らかな欠陥がわからないようにする方法である。」

明らかな欠陥がわからない場合、わからないからこそ、その欠陥をみつけるテストも難しい。第５－１節のシステム統合の方法は、テストを困難にするようなシステム構成（図３）を採用したことが誤りだった。そもそもの方針が悪かったということであれば、図１の要求仕様の前から間違っていたことになる。だからこそ、ビジネス書で、「経営層の情報システム軽視」と議論されたのであろう。

６　経年変化

大規模システムの開発には莫大なコストがかかる。長い期間使うことでようやく開発コストを回収できる。

６－１　陳腐化への対応

家電製品や自動車のような装置（ハードウェア）は耐用年数がある。故障部品の交換など定期的な保守を行う。突発的な故障は異常処理としてソフトウェアで対応する。

プログラム（ソフトウェア）や書き物は故障しない。しかし、全ての書き物が一度完成（脱稿）すれば終わりというものでもない。長く利用されるためには、中身の改訂が必須なものがある。故障はしないものの、内容が陳腐化して、有用性が低下する。あるいは、無用の長物と化すからである。

長い年月、多くの人々が使っている広辞苑は、1935 年の辞苑からはじまり、1955 年広辞苑と名称を変えた。その後、1969 年第 2 版、1976 年第 2 版増、1983 年第 3 版、1991 年第 4 版、1998 年第 5 版、2008 年第 6 版と続く。最初の版から、70 年以上の歳月にわたって改訂されている。

ソフトウェアを長い期間にわたって利用するには、機能の改訂を継続することが必須である。業務システムの場合、関連する法律が変われば、それに伴ってプログラムで処理する伝票の情報が変化する。正しかった識別コードが無効になったりする。

また、システム実行を支える計算機ハードウェアやネットワークなどを高性能装置に

置き換えることで、システム稼働後の処理増加に対応する。置き換え時にはソフトウェア側の対応も必要なことが多い。すなわち、変化する動作環境に合わせたプログラム改修も行わなければならない。

第４－２節では、大規模システムの初期開発の際、たびたびの要求変更から、システムが複雑化するという問題を述べた。陳腐化への対応によって、さらに複雑化することは容易に想像できる。

従来、これらは、システム保守とか改修と呼ばれていた。しかし、ソフトウェアの難しさは、最初に決めた通り作れば良いものではないということにある。改修を前提とする技術といってよい。この点に関心を向けることを目的として、「ソフトウェア進化発展（Software Evolution）」と呼ぶ。

ソフトウェアは、本質的に進化発展する「生き物」であり、他の工業製品のような品質保証を難しくしている。進化発展の中に、品質保証を継続する仕組みを組み込んでおくことができれば良いのであろう。

６－２　ブラックボックス化の事件

規模の爆発、経年変化は、稼働中のシステム運用での事故につながることがある。ヒューマンエラーとされることが多いが、実は、システムの仕組みに対する知識が、ソフトウェアの進化発展の過程で散逸してしまったのかもしれない。一般紙でも報告された最近の事例[11]を紹介する。

第２－２節で紹介したＭ銀行が 2011 年 3 月に起こした振込システムの障害である。特定の口座に大量の振り込みが集中した。このシステムは旧Ｄ銀行のものであり、昼間のオンライン処理と夜間のバッチ処理で口座振替を行う（図５）。オンラインの上限に達したことからバッチ処理で対応することになった。ところが、バッチ処理で異常終了になるという新たな障害が発生した。

この障害原因は、バッチ処理にも上限値があったことをシステム担当者が知らずに作業を進めていたことにあった。上限値を設定しなおし、システム再開後、さらに別の障害が発生した。先の異常終了で振り込みデータが欠落し処理に必要なデータを失くしてしまっていた。消えたデータの修復作業に時間がかかり、次の日の営業時間に間に合わなかった。

この事件の直接の原因は、上限値の設定であり、プログラムのバグではない。そこで、システム運用の誤り、すなわち、ヒューマンエラーと云われている。しかし、長い運用の歳月で、システムの仕組みへの理解が欠けたというほうが正確だろう。

実際、M銀行のシステムは旧Ｄ銀行を引き継いだものであり、旧システムが稼働を開始したのは 1988 年という。その後、23 年間、使い続けてきた。2002 年の銀行統合の際の改修を含めて長い間の変更が加えられ、進化発展をとげてきたであろう。その結果、2011 年には、内部の仕組みが見えないブラックボックス化に陥っていた。

６－３　ドッグイヤー

1988 年から 2011 年という 23 年間が、どのくらい長いのかを、コンピュータやネットワークの技術変遷と比較して考える。

一般に、情報技術は変化が速いことから、１年をドッグイヤー（犬の 1 年）で考えることがある。人間の 23 年をドッグイヤーに換算すると、どのくらいになるだろう。同じソフトウェアをそのまま使い続けることは想像できない。

パソコンのオペレーティングシステムとして Windows が使われている。1988 年時点では Windows2.0、その後、Windows3.0、WindowsNT、Windows95、Windows98、Windows2000、WindowsMe、を経て、2002 年は WindowsXP になった。XP は今でも現役であるが、マイクロソフト社は、2006 年に VISTA、2009 年には Windows 7 を出している。今から見ると、Windows2.0 は本当に小さなシステムであった。

次に、インターネット関連の技術を見てみる。1988 年日本ではインターネットは学術ネットワークの段階であり、商用インターネットのはじまりは 1991 年である。1996 年がインターネット元年と呼ばれる。また、1994 年に最初のブラウザ Netscape が公開され、1999 年に携帯電話の i モードがはじまった。その後、スマホが登場したことで、携帯電話の特徴であるモバイルの世界とインターネットが統合する。2007 年アップルの iPhone が切り拓いたといってよい。

このように、ちょっと調べただけでも、大きな変化が起こっていることがわかる。特に、1990 年代中頃からのインターネット普及は、情報システムに多大な影響を与えた。

銀行関係では、24 時間運用のインターネット・バンキングが代表例である。インターネットで何処からでも気軽に使える。このサービスは一般の利用者にとって魅力が大きい。銀行としては競争力を失わないためにシステム化の必須要求機能である。

当然、M銀行もこのサービス提供を行っている。不思議なことは、そのベースとなっているのが、今から 23 年前の旧Ｄ銀行システムであるという点ではないだろうか。ソフトウェアの経年変化の大変さへの配慮が足りなかったのかもしれない。

システムの不具合が、23 年間、奥深くに隠れていた。それが、2011 年 3 月に、「普通でない」状況で突然噴火した。すべてのソフトウェアが休火山でないことを祈るだけである。

７　専門家の間違い

７－１　単純な間違い

人間は間違うことが多い。そうは云っても、専門家であるからには一般の人のような間違いは冒さないと期待したい。たしかに、間違う頻度は小さいだろう。しかし、

[絶対数]＝[記述総量]×[誤り頻度]

であるから、たとえ誤り頻度が小さくても記述の総量が多ければ、間違いの絶対数は

大きい。

ニューヨーク・タイムズ紙では、新聞記事の訂正一覧を掲載している[2]。通常の（週末でない）朝刊紙面の場合、約 13,800 文である。これに対して、訂正数は 1 日あたり 10～15 個なので、約 1000 行にひとつの間違いがある。

一般に、プログラムに含まれるバグは、1000 行に 1 件と云われている。新聞記事とプログラムで同じ頻度というのが面白い。新聞記者も、ソフトウェア技術者も、時間との戦いで作業している。このくらいの誤りは、如何なる専門家にも起こり得る、というかもしれない。

７－２　致命的な間違い

ソフトウェアを開発する場合、出発点、最初が大切である。要求仕様（図１）のところで、期待される機能をもれなく盛り込まなければならない。ここで抜けや誤りがあると、後の開発作業やプログラミングを正しく行っていても、役に立たないシステムが完成するだけである。

大規模システムの場合、受注開発が多い。特定の顧客向け専用システムの開発である。顧客が必要とする機能（要求）をもれなく洗い出す。しかし、発注者は業務に精通していても、情報システムの専門家ではない。システムとしての実現可能性を考慮した上で、自分が何を望んでいるかを整理して表現することは難しい。頭の片隅に隠れている要求の抽出に失敗すると、期待外れのシステムを開発することになってしまう。

さらに、システムがどのように使われるか、どのような危機にさらされる可能性があるか、を熟考しなければならない。特に、インターネットと接続するシステムでは、セキュリティに関わる外部からの攻撃や突然の処理依頼の増加など、オープンなシステムならではの脅威にさらされる。ここで考え落ちがあっては、システム運用時に不具合が生じる。

一方、現実には、開発コストが増大する、そんな状況は起きるはずがない、などの理屈をつけて、問題点に気づいていても、システムに盛り込まないで無視するかもしれない。2011 年に新聞などで有名になった言葉、「割り切った考え。すべてを考慮すると設計できなくなる」を思い出す。

しかし、この誤った「割り切り」はシステムの不具合の遠因である。システム開発時に、不具合の原因が作り込まれていることになる。最初から不良品と知って開発したのではないか、と云われても反論できない。

誤った想定のもとに開発されたシステムの品質保証とは何であろうか。テスト項目に抜けがあることになる。Ｖ字型開発のスラッシュ（図１）で如何にテストを強化しても虚しいだけである。「品質」とは何かという問いから考え直す必要がある。

８　利用者の振る舞い

８－１　誤発注事件

利用者の振る舞いがシステム障害を引き

起こすことも多い。2005 年 12 月に起きた誤発注の事件[6][14]を紹介する。

Ｊ社が株式市場に新規株式公開した。公開株数は 3000 である。これに対して、M証券から売り注文が入る。本来、「1 株 61 万円」のところを、担当者が入力を誤って「1 円 61 万株」の売り注文を出した。誤りに気付いて売り注文取り消しコマンドを発行したが、期待通りに作動しなかった。多数の買い注文が起こり、結果として、M証券は 400 億円の損失を出した。その後、この事件は、責任の所在と損害補償をめぐって訴訟になった。

事件のきっかけは、担当者の入力誤りである。システムは、入力誤りの警告を表示したが、担当者はこれを無視して作業を続けた。これによって、誤った売り注文が株式市場側のシステムに発行された。

取り消しコマンドが作動しなかったのは、株式市場側システムの不具合であることが後日半明した。事件が発生する 5 年前の 2000 年にテストを行った際にバグが見つかっていた。そのバグを修正したプログラムにバグが含まれていた。

通常、バグ修正した際には、変更が正しく行われているかのテストを実施する。たとえ変更が小さく 1 ヶ所であったとしても、関連するテスト項目をすべて再検査しなければならない。しかし、このシステムではテストが不十分であって、バグが隠れたままであった。実際、5 年の間、何も不都合なくシステムは作動しており、誰も不具合に気付かなかった。バグ修正した箇所であるからこそ、新たなバグが混入したとは思いもよらなかったのだろう。

一方、公開株数が 3000 であるにも関わらず、61 万株の売り注文を正しいものとして受け付けることも納得いかない。このような観点からの数量チェック機能は不要という判断でシステムを開発していた。つまり、要求機能の誤りともいえる。実際、2006 年には、チェック機能を組み込んだ。

８－２　思い違い

M証券のシステムは、作業者の入力値をチェックし警告を出している。システムの開発時に、端末操作者が警告を無視することまでは想定できない。

システム開発は要求仕様の整理からはじまる（図１）。ユースケース分析という技術を用いて、利用者の使い方からみたシステムの機能を洗い出す。しかし、利用者は「想定外」の行動をとる。第６－１節の事件を引き起こした警告の無視は、その例だった。M証券では、その後、システムが警告を出した場合の業務体制を整えた。担当者以外の人が再度確認する。すなわち、システムだけでは対応できない状況がある。

システム開発の要求仕様段階で行うことは、システムの使い方、提供機能やサービスなど、様々な観点からの予測を高い精度で行うことといえる。この段階での見過ごしはプログラム作成まで引き継がれる。テスト工程の最終段階である受け入れテストでも見過ごしは見つからない。ここでのテスト

項目は要求仕様に書かれた内容をもとにするからである。第７－２節で触れた致命的な間違いである。

さらに、利用者の振る舞いを完全に予測することは不可能に近い。警告無視まで考慮することはできない。利用者の不注意といえる。また、利用者はシステムの使い方について思い違いをするかもしれない。悪戯心で奇妙な使い方をすることもあるだろう。さらに、悪意を持ってシステムを攻撃する場合もあり得る。

攻撃への対処を考えればわかるように、このような状況への対策は、稼働システムの動作を監視し、「普通でない」状況に陥らないかを常に把握することである。開発工程の一部として実施する検査だけでは十分な確認ができないともいえる。

９　構築からの正しさ

９－１　事後確認の限界

本稿の副題は「作るより検査が難しい」であった。いや、作った積りになってはいけない。「検査に合格して初めて作ったといえる」であろう。

第３節で述べたように、ソフトウェアの検査はプログラム・テスティングである。プログラム開発を終えてから検査が始まる（図１）。開発の最終生成物であるプログラムを対象とすることから、事後検査と呼ぶ。

たしかに、テスト工程の後段階で行うシステムテストや受け入れテストは、テストの観点そのものが、システム全体の動作に関わることから、完成したプログラムを対象とする。本質的に「事後検査」である。

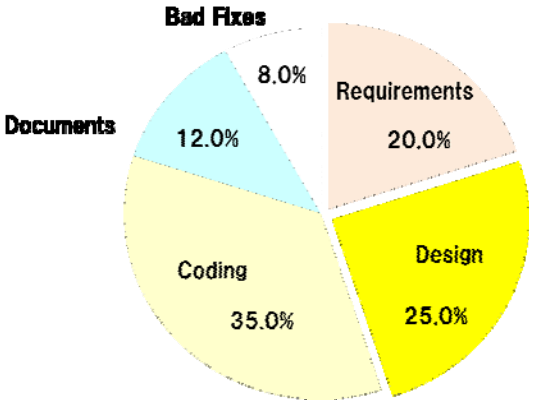


図７　不具合発生工程[12]

図７は米国の統計データである。開発の各工程で混入した不具合の割合を示す。検査終了後に検出した不具合の内容を吟味し、どの工程が原因となったかを整理したものである。

Requirements（図１の要求仕様）での誤りが 20%、Design（システム設計からコンポーネント設計）で 25%、Coding（プログラム作成）で 35%を占める。Documents は文書作成の誤りであり、Bad Fixes はバグ修正時に新たな誤りが入ったことを示す。たしかに、第８－１節の不具合は Bad Fixes の例だった。

図７から直ちにわかることは、プログラム作成よりも前の段階での不具合混入が約 50%であるという点である。ソフトウェア品質を向上させるには、開発上流工程で如何にして誤りを排除するかにかかっている。「作ってから検査する」（事後検査）のでは

なく、「開発（設計）と確認（検査）を同時に行う」方法が必要になる。

９－２　正しさの積み重ね

開発の上流工程で作成する設計書や仕様書は通常の文書である。「通常」といった理由は、自然言語（日本語など）や図表で書かれているということである。文書の意味内容を理解できるのは技術者であって、コンピュータが自動解析できる形式ではない。したがって、設計書や仕様書が正しく作成されているかを検査するのは技術者である。実際は、設計レビューと呼ぶ文書読み合わせによる確認作業による。

設計レビューで不具合を見つけ品質を高めるには、経験豊かな技術者が十分な時間をかけて作業する必要がある。開発経験や関連する知識を駆使することで、行間を読む、等の直感を働かすことで、設計書の問題点を洗い出す。

表現の曖昧さ、用語の不統一から生じる誤解、ひとつの機能に対する説明が複数個所で整合していない、など、通常の文書には多くの不具合が隠れている。第 7－1 節で新聞記事に残った間違いを話題にした。如何に注意しても通常の文書から誤りを完全に取り除くことはできない。そのような努力をしても図 7 のような状況である。

これに対して、開発上流工程の設計段階から意味定義の明確な言語（形式仕様言語）を用いる技術が研究されている[10][13]。これによって設計書の曖昧さをなくし、記述内容の不整合を系統的に検知可能にする。さらに、段階的に記述を詳細化、具体化する各ステップで、変更が正しいことを確認する。正しい初期記述から段階的に正しさを確認しながら最終的なプログラムを導き出す。これを「構築からの正しさ（Correct by Construction、CxC と略記）」と呼ぶ。

CxC の技術は長い研究の歴史があり、E.W. ダイクストラのチューリング賞講演[4]でも触れられている。この講演を契機に、CxC の研究が推進されたという云い方が正しいかもしれない。その後、学術的な基礎研究だけでなく実践的な技術開発も精力的に行われた。パリ地下鉄 14 号線の自動運行システム開発へ適用した話[1]が世界的に有名である。CxC の方法で正しいプログラムを導出できることから、単体テストを行わなかった。システムテスト等が必要なことは云うまでもない。

現在、CxC の方法は産業界での標準的な開発法にまで成熟していない。技術的な課題については本稿の範囲を超えるので詳述しない。国内でも重要技術として調査研究や実証実験を行っている[5]。産業界の標準的な開発法として定着すれば、ソフトウェア品質保証の技術が大きく変わる。

１０　「正しさ」再考

CxC によって全てが解決するだろうか。そんなことはない。残る課題の中から話題を 2 つ紹介する。

１０－１　妥協の産物

構築からの正しさ（CxC）は最初の出発

点となる初期記述が正しいことを前提とする。その後の詳細化、具体化の過程に誤りがないとしても、出発点が間違っていたら、何をしているのかわからない。図１の要求仕様が重要になる。

ところが、要求仕様の「正しさ」を論じることは極めて難しい。もともと「正しさ」は何らかの基準があって決まる相対的な事柄である。要求仕様の作成には、様々な立場の利害関係者（ステークホルダー）が絡む。システムに何を期待するか、システムのどの側面に責任を負うか、が異なる。立場によって正しさの観点が変わる。

別の言葉でいえば、要求仕様の作成は、正しさの基準を作っていくことである。主張が対立する場合、要求仕様作成は妥協の過程である、要求仕様は妥協の産物とも云える。正しさの基準も妥協の産物になる。現実には、開発費用の決定権を持つステークホルダの力が強いだろう。技術的に妥当な主張が通るとは限らない。

１０－２　実行時監視と自己適応

ソフトウェアの世界で長くコンサルタントをしている M. Jackson は次のように述べている[9]。

「情報システムが有用なのは外部の
世界と関わりを持つからである」

そして、開発の際に、実行時の動作環境を完全に知ることができない。このことがシステム開発の限界と説明した。

第８－２節で議論した利用者の振る舞いはシステムの外部（動作環境）と見做すこともできる。たしかに、システム開発時に利用者の振る舞いを予測することはできない。できることは、実行時にシステムが想定した振る舞い（期待）が壊れないかを監視することである。正しいことを確認するのではなく、不具合に陥らないことを監視する。システムの信頼性を向上させる技術として、実行時監視を併用する。

実行時監視の方法を用いることで、利用者の不注意、思い違い、悪戯、攻撃を検知する。監視する性質をうまく選択することで、要求仕様の破れを監視する方法も研究されている。さらに、監視性質の破れを検知した後、その原因を避ける方法をシステムに組み込むことができれば良い。そのようなシステム構成法として、「自己適応システム」や「自己改変システム」の研究が進められている。

１１　おわりに

「作るより検査が難しい」ことの直接的な理由は、検査対象ソフトウェア・システムの複雑さである。複雑さの第１の理由は、規模の爆発と陳腐化への対応の結果である。常に進化発展する「生き物」としてソフトウェアを理解しなければならない。複雑さの第２の理由は、システムが実世界の中で作動する実体であることに起因する。人間が活動する複雑な実世界との相互作用を的確に取り扱わなければならない。このような複雑さから、正しさの基準が何であるかが明確にならないことさえある。「作るより

も検査が難しい」。

今日、ソフトウェアはあらゆる工学領域に不可欠のシステム実現技術として浸透してきてきた。デジタル化、電子化、などの言葉で語られる技術革新は、本来、ソフトウェア化と呼ぶべきである。今後、この流れは、さらに加速される[16]。したがって、ソフトウェアの品質保証が益々重要になっている。

本稿では、ソフトウェアの品質保証の難しさを幾つかの観点から説明した。ソフトウェア固有の特徴も含まれるが、多くは、大規模システムに共通する課題ともいえる。ソフトウェアの場合、「作るのが簡単」な故に大規模化が加速し、問題が顕在化しやすい。「恐れず蓋をせず」、根拠のない信頼感をシステムに抱かず、原理的な限界を知って、ソフトウェア中心のシステムを利用しなければならない。「情報処理と情報システムの原理に対する理解」を欠いてはならないのである[8]。

参考資料

[1] J.-R. Abrial : Formal Methods in Industry -- Achievements, Problems, Future, Proc. ICSE 2006, pp.761-767, 2006.

[2] L. Amster and D. L. McClain (eds.) : Kill Duck Before Serving, St. Martin's Griffin 2002.

[3] 朝日新聞記事「プロメテウスの罠」、2011 年 12 月 25 日.

[4] E.W. Dijkstra : The Humble Programmer -- ACM チューリング賞講演,1972.

[5] DSF. http://www.nttdada.co.jp/dsf/

[6] 畑村創造工学研究所:失敗知識データベース、http://www.sozogaku.com/fkd/

[7] C.A.R Hoare : The Emperor's Old Clothes -- ACM チューリング賞講演, 1981.

[8] 情報処理学会 情報処理教育委員会 :「2005 年後半から 2006 年初頭にかけての事件と情報教育の関連に関するコメント」、2006 年 3 月 6 日.
http://www.ipsj.or.jp/12kyoiku/statement2006.html

[9] M. Jackson : Role of Formalism in Method, Proc. FM'99, pp. 56-56, 1999.

[10] 中島 震 : ソフトウェア工学の道具としての形式手法、NII-2007-007J, 2007.
http://www.nii.ac.jp/TechReports/07-007J-j.html

[11] 日経コンピュータ (編) : システム障害はなぜ二度起きたか、日経 BP 社 2011.

[12] Software Quality 2008.

[13] 玉井 哲雄 : ソフトウェア工学の基礎、岩波書店 2004.

[14] T. Tamai : Social Impact on Information System Failures, IEEE Computer, 42(6), pp.58-65, 2009.

[15] Wikipedia「失われた時を求めて」、http://ja.wikipedia.org/wiki/. 石木 隆治 : プルースト、清水書院〈人と思想〉1997.

[16] J. Wing : CPS Flower, 2008
http://varma.ece.cmu.edu/InfoCPS/Jeannette-Wing.pdf

ソフトウェアの品質保証
ー作るより検査が難しいー

平成24年2月15日
中島　震
国立情報学研究所

概要

- ソフトウェアの品質保証
  - 工業製品のソフトウェア化（電子化、デジタル化）が加速
  - 「動いて」当たり前？　⇒　障害が増加（新聞記事）
- 作るより検査が難しい
  - 検査しないで作った気になってはいけない
  - 不具合がないことの確認は既存の方法では困難である
- ソフトウェアは「複雑さ」に支配される
  - ソフトウェアが簡単に「複雑化」することが問題の根底にある

６つの話題

1. プログラムの検査は難しい
2. 複雑さが禍のもとになる
3. ソフトウェアは変化しなければならない
4. 間違えることを前提とする
5. 利用者の振る舞いは予測をこえる
6. 人工物構築に関わる複雑さを扱う

話題１

プログラムの検査は難しい

障害（1）

- 発生時期
  - 2002年4月：大手銀行統合初日
- 障害内容
  - ATM障害、口座振替が不能、二重引き落とし
  - 顧客の口座データ消失
  - 過労自殺、2003年労災認定
- 原因説明
  - 3つの銀行システム統合、プログラム残存バグ
  - 負荷テスト、不正データ検出テストが不十分

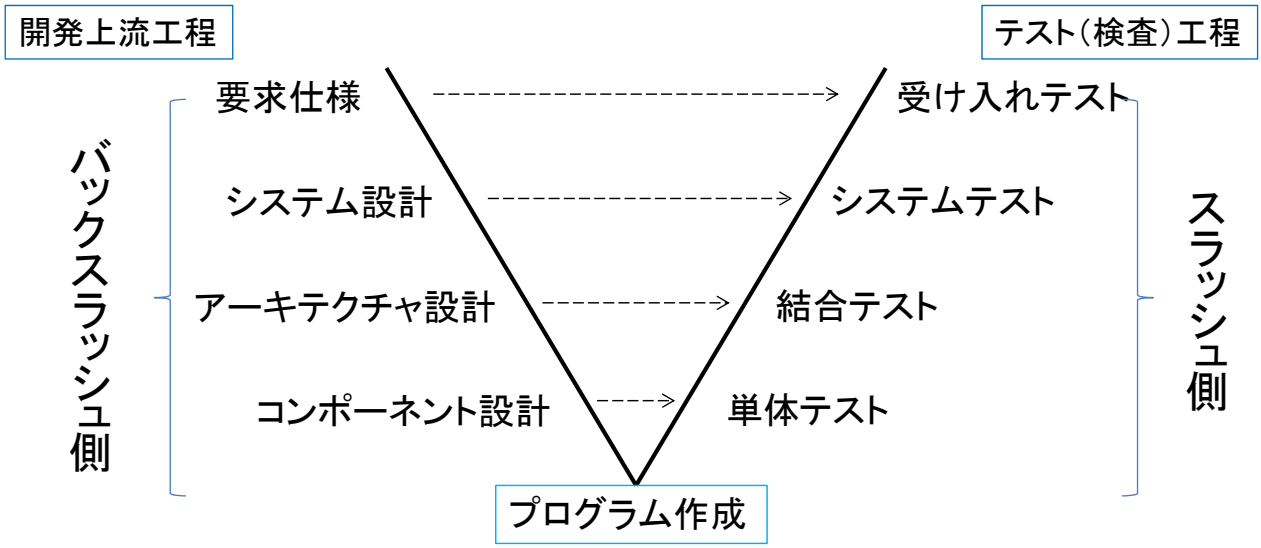
障害（2）

- 発生時期
  - 2011年夏：内部被曝検査機
- 障害内容
  - WBC（ホールボディカウンタ）の測定結果が不正
  - 放射能の種類ごとに体内に存在する量を推計
- 原因説明
  - 推計プログラムの計算バグ
  - 「ほとんど使われていないのでバグはわかりませんでした」（メーカ担当者の説明）

2つの障害に共通する点

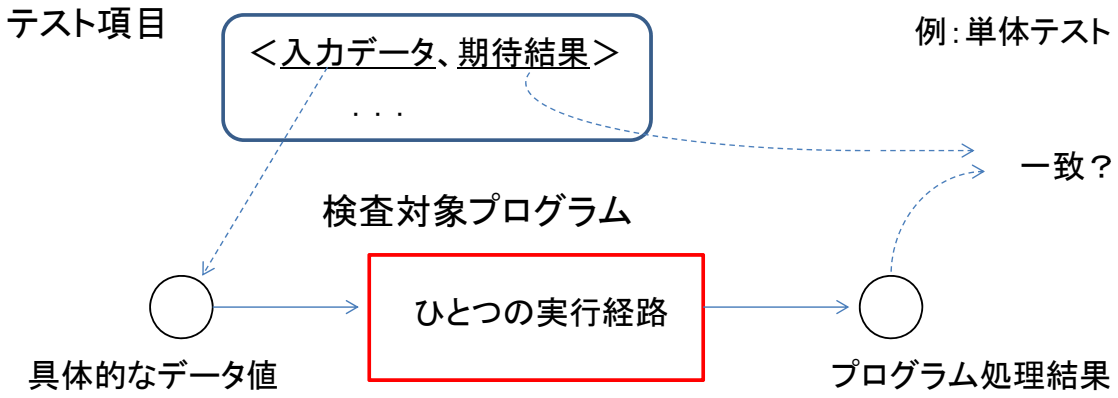
- 何となく納得？
  - 「プログラムを作ることは作ったが、正しく作動するかの検査が不十分だった」という説明
- 本講演での疑問
  - どのような検査で不具合がないことを確認できるのか？
  - 不具合がないことを本当に示せるのか？
  - そもそも、「不具合がない」、「正しい」とは？

開発過程の説明図



(1) プログラムが検査の対象（事後検査）
(2) 開発上流の工程ごとに検査の観点（テスト項目）

プログラム検査の難しさ



検査の網羅性に関する問題

すべての実行経路を調べることは、原理的に困難（膨大な組み合わせの数）

⇒ 現実には、「近似的な基準」を用いて検査終了とする

「正しさ」自身の難しさ

- 正しさの基準
  - 機械的に決まる「正しさ」は単純なものだけ
    - 文法エラー、など
  - 正しさの基準は相対的
    - His name is John. ⇒ 正しい？
  - 「テスト項目」を網羅的に作成できるか？
- プログラム検査との関係
  - 入力データ実行で検査可能な「テスト項目」で十分か？

プログラム検査の限界

program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.

バグがあることを示す効果的な方法

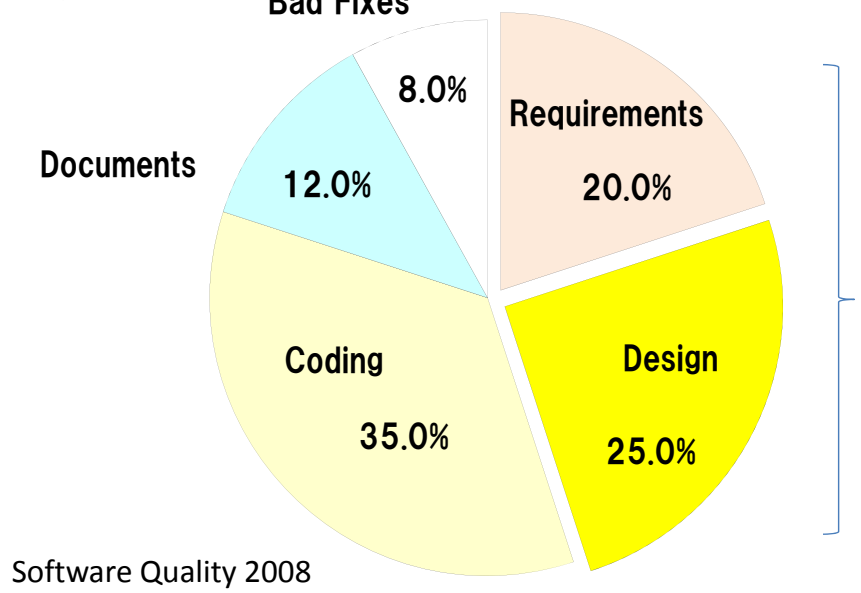
E.W. Dijkstra : The Humble Programmer (1972)

⇒ （テストデータが適切でないと）バグを見過ごす

不具合発生工程

テスト工程で発見した不具合が混入した工程

修正した筈が新たに
不具合を混入



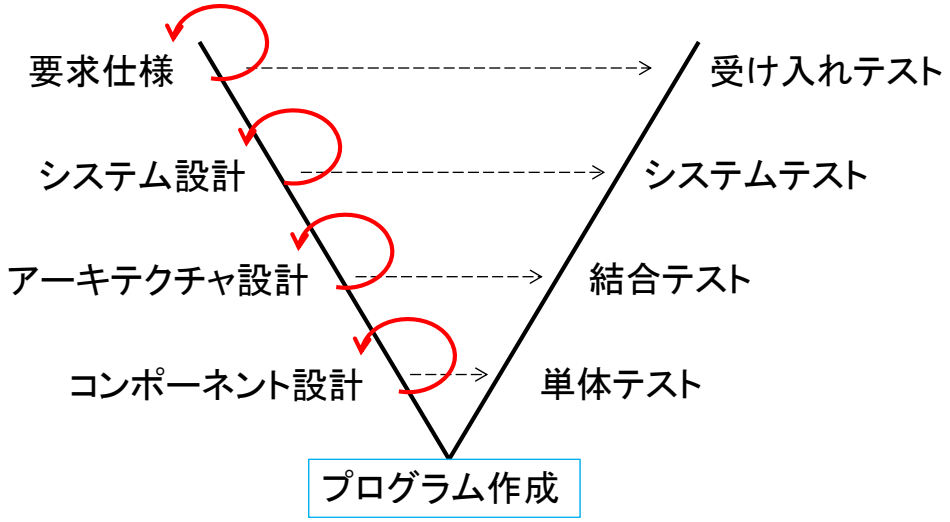
開発上流工程
で混入（45%）

検査対象は
プログラム

Software Quality 2008

開発上流工程での検査

現状、「仕様書レビュー」(読み合わせチェック)⇒長時間の会議



きちんと作ることの大切さ

- プログラム検査の問題点
  - プログラムを網羅的に検査することは原理的に困難
  - 「正しさ」の基準(テスト項目)に抜けがあると無力
- 「きちんと」したプログラム作り
  - 検査・確認しながらプログラム作り
  - 「複雑さ」を克服する技術が必要
  - ソフトウェアは簡単に複雑化してしまう

話題2

複雑さが禍のもとになる

複雑さと不具合

(I conclude that) there are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies

単純にして、あきらかに誤りがない

and the other way is to make it so complicated that there are no obvious deficiencies.

複雑にして、あきらかな誤りが(わから)ない

C.A.R. Hoare : The Emperor's Old Clothes (1981)

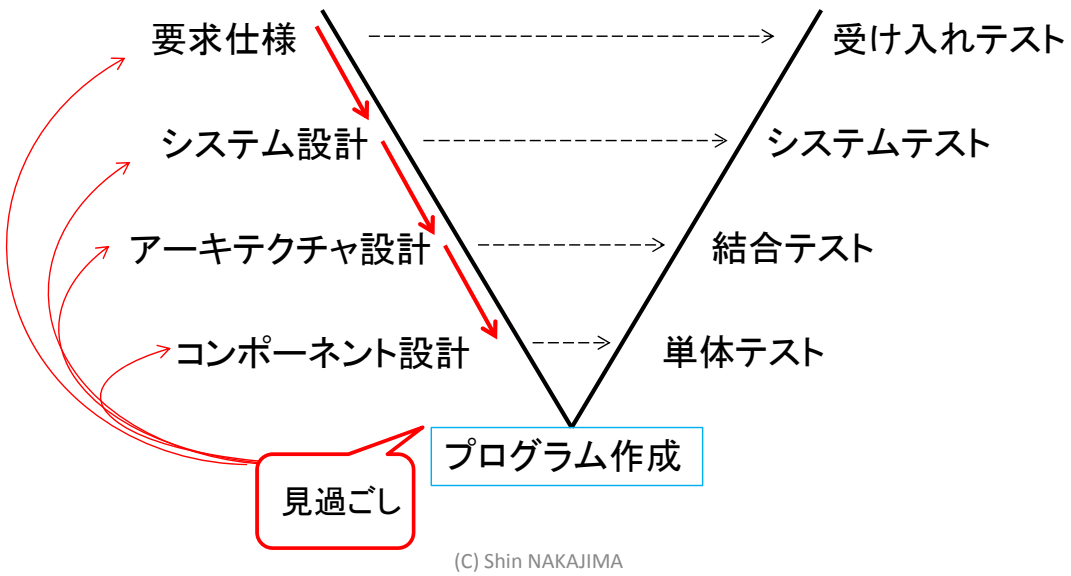
ソフトウェアの規模

- 規模の比較
  - 検査からみたソフトウェア規模＝プログラム行数
  - スマホ（Android） ⇒ 約2000万行
  - ジャンボジェット機 ⇒ 部品点数 約600万
  - 長編小説「失われた時を求めて」 ⇒ 約1000万字
- 大規模ソフトウェアの開発
  - 大手銀行勘定系システム ⇒ 1億行（10000万行）
  - 開発の生産性指標：500～600行／月・人
  - 1億行の開発 ⇒ 1000人が10年間という規模感

複雑化する理由（1）

後工程で「見過ごし」を発見 ⇒ 後戻り

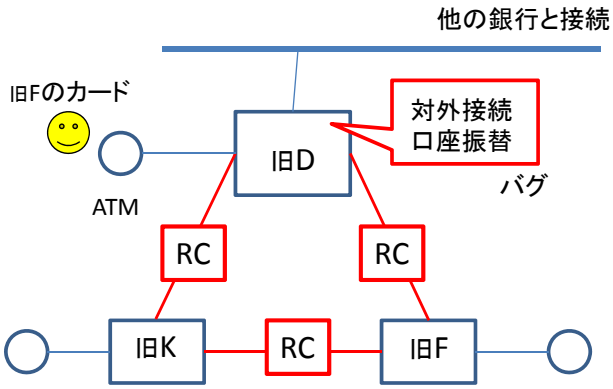
そもそも、要求仕様を明確にすることが難しい



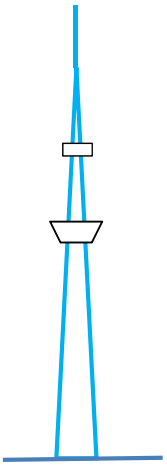
複雑化する理由(2)

大局感のないシステム構築

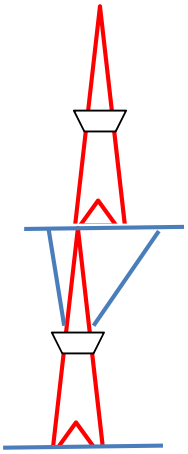
大手銀行の統合システム



東京スカイツリー



辻褄合わせ



話題3

ソフトウェアは変化しなければならない

経年変化への対応

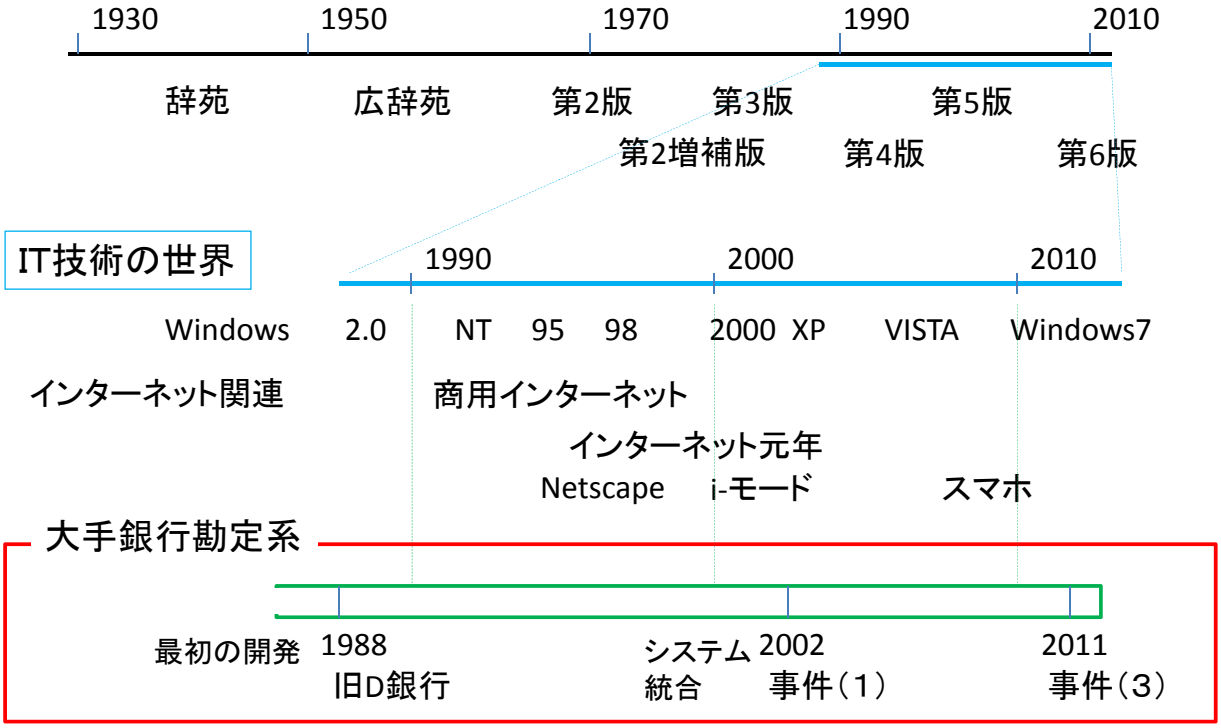
- 開発コストの観点
  - 長期間の運用で開発コストを回収
- システムの「改修」
  - ハードウェア（装置）は劣化故障
  - ソフトウェアは、不具合がなくても、機能の陳腐化

  ⇒ 進化発展（Software Evolution）

障害（3）

- 発生時期
  - 2011年3月大手銀行振り込み不能
- 障害内容
  - 特定口座への振り込みが集中
  - 異常終了によるデータ消失
- 原因説明
  - システム運用、処理上限値の設定ミス
  - 23年間（1988年サービス開始）のシステム
  - ブラックボックス化（システム情報の散逸）

長期間の改訂



ソフトウェア発展

- 大規模システムに共通する課題
  - 規模と開発期間
- ソフトウェアで顕在化する難しさ
  - 経年変化（機能の陳腐化）への対応
  - 進化発展する「生き物」

話題4

間違うことを前提にする

不具合の比較

- ハードウェア（装置）の不具合
  - 偶発的な不具合
  - 部品の耐用年数（故障確率）
  - 「部品交換」を中心とする保守作業
- ソフトウェアの不具合
  - 系統的な不具合
  - 開発過程で混入、人的要因
  - 利用環境からの乖離、機能の陳腐化

専門家が起こす間違い

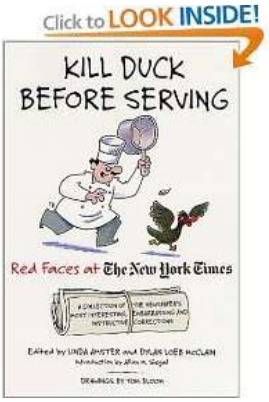
- ソフトウェアの不具合原因
  - ソフトウェア技術力の低さ(?)
  - 間違うことを前提
- 間違いの種類
  - 単純な間違い
  - 致命的な間違い

単純な間違い

プロフェッショナルは間違い(の量)が多い

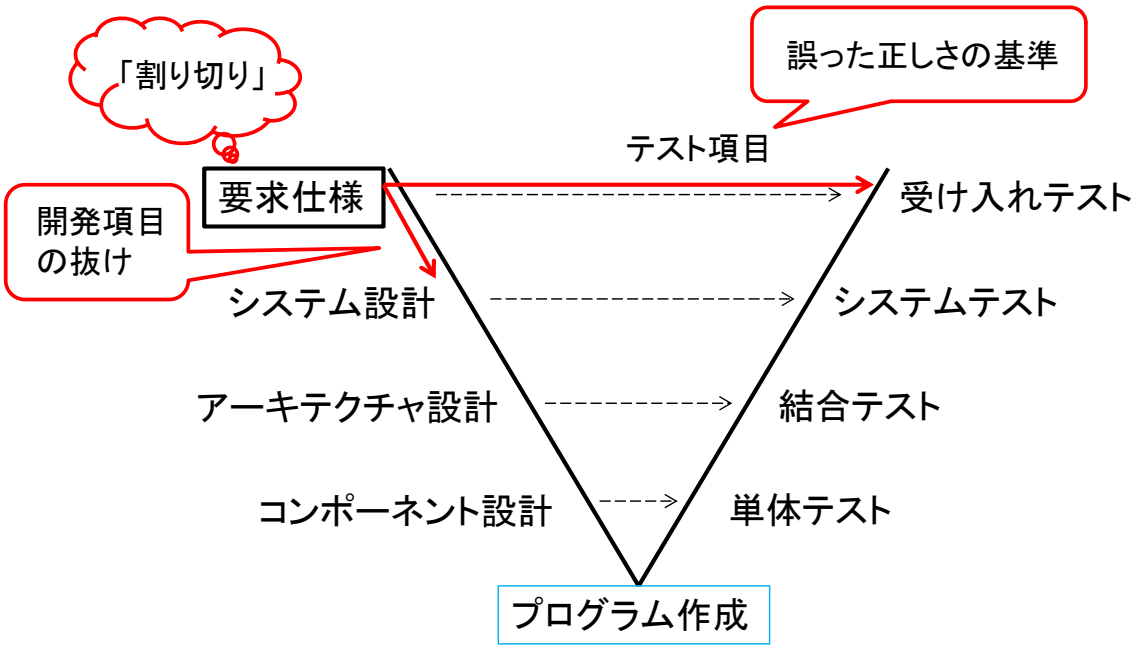
New York Times 紙
ウィークデイ:約13,800文
記事訂正 10〜15個/日
→ 1,000文に1つの間違い

「プログラム1000行に1つのバグ」
という統計データと同じ傾向



間違い集のペーパーバック

最初から欠陥品（？）



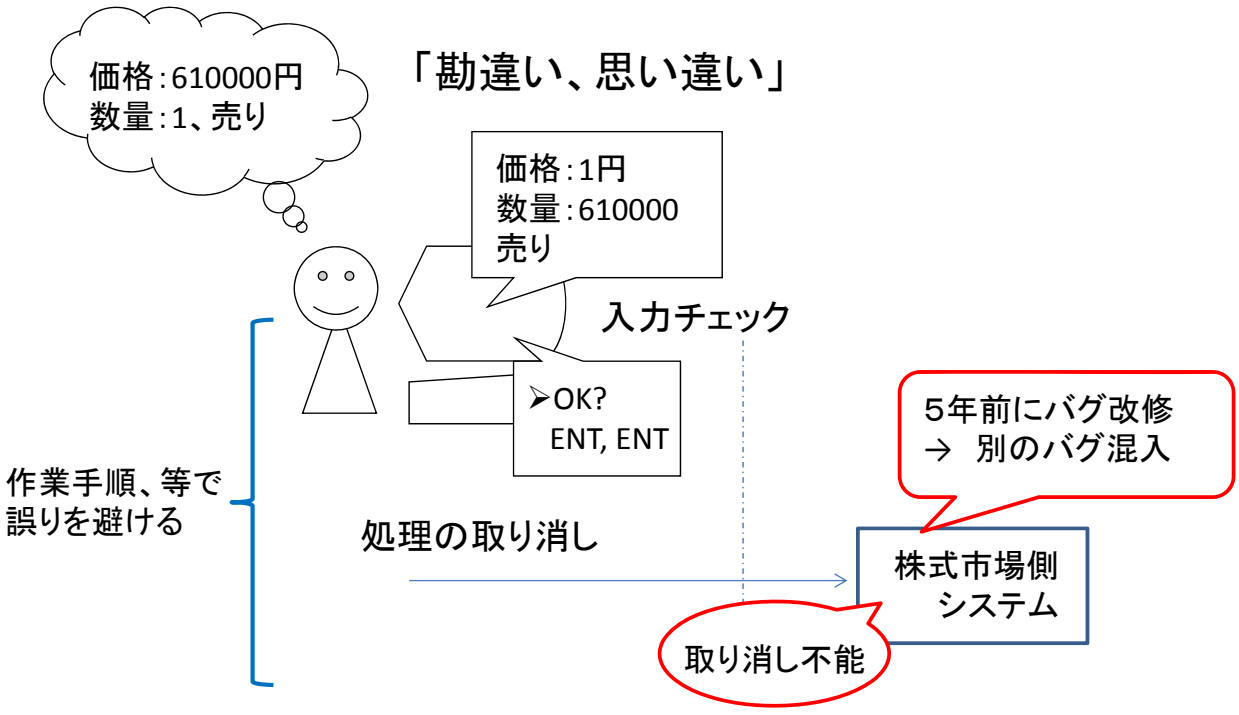
話題5

利用者の振る舞いは予測をこえる

障害(4)

- 発生時期
  - 2005年12月株式の誤発注(公開株数3000)
- 障害内容
  - 「1株61万円売り」を「1円61万株売り」と誤発注
  - 注文取り消しコマンドが作動せず
  - 誤発注の証券会社が400億円の損失
- 原因説明
  - 発注システムからの警告を担当者が無視
  - 株式市場側システムのバグで取り消し不能

利用者の振る舞い

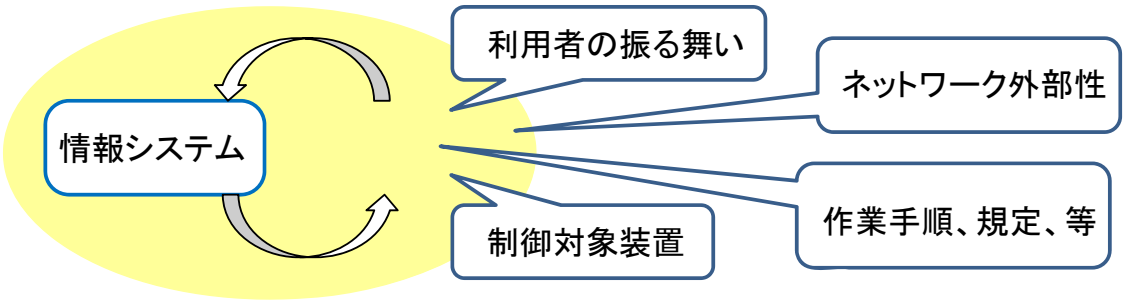


実世界との関係性

any useful computing system interacts with the world outside the computer

有用な情報システムは外界と相互に作用する

M. Jackson : The Role of Formalism in Method (1999)



話題6

人工物構築に関わる複雑さを扱う

「ソフトウェア工学」という研究分野

Software engineering is the branch of computer science that creates practical, cost-effective solutions to computation and information processing problems, preferentially applying **scientific knowledge**

Software is constrained not by Physical Laws, but by **Complexity**

複雑さが支配する

M. Shaw : Whither Software Engineering Education? (2011)

⇒ 進化発展する人工物（ソフトウェア）が示す**複雑さの科学**

最近の研究活動

- 構築からの正しさ
  - 開発上流工程での高い信頼性を達成する技術
  - 「検査しながら開発を進める」方法
- 自己適応システム
  - 「柔らかな不具合」（立場により異なる基準）という考え方
  - 利用者の振る舞いに起因する「柔らかな不具合」の検知
  - 「柔らかな不具合」を解消するシステム改変機構

http://researchmap.jp/nkjm/

構築からの正しさ

- 産業界との共同研究
  - 実用性の実証
  - 適用技法の研究
- 「形式手法とは？」
  - 技術移管（＝教育）
  - 書籍の公刊

DSF

2011

2011

2010

2008

最後に

まとめ

- 難しさの原因は「複雑さ」
  - 大規模化、経年変化、取り巻く環境、等
  - 長く使う大規模システムに共通

- ソフトウェア
  - 「作るのが簡単」→複雑さが加速、問題が広がりやすい
  - 根拠のない信頼感を抱かず、原理的な限界を知ること
  - 人工物構築に関係する「複雑さ」の科学