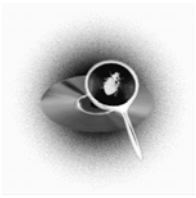
ソフトウェアテスト技術



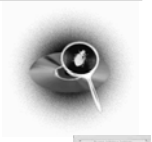
経営情報学会　第三回 情報システム工学研究部会
2007/2/17（土）
電気通信大学 電気通信学部 システム工学科
西　康晴

自己紹介

- 身分
  - ソフトウェア工学の研究者
    - » 電気通信大学 電気通信学部 システム工学科
    - » ちょっと「生臭い」研究／ソフトウェアテストやプロセス改善など
  - 元・ソフトウェアのよろず品質コンサルタント
- 専門分野
  - ソフトウェアテスト／プロジェクトマネジメント／QA／ソフトウェア品質学／TQM全般／教育
- 共訳書
  - 実践ソフトウェア・エンジニアリング／日科技連出版
  - 基本から学ぶソフトウェアテスト／日経BP
  - ソフトウェアテスト293の鉄則／日経BP
  - 基本から学ぶテストプロセス管理／日経BP
- もろもろ
  - TEF: テスト技術者交流会 / NPO ASTER: テスト技術振興協会
  - SESSAME: 組込みソフトウェア管理者技術者育成研究会
  - 情報処理学会 ソフトウェアエンジニアリング教育委員会
  - 経済産業省 組込みソフトウェア開発力強化推進委員会

TEF: Testing Engineer's Forum

- ソフトウェアテスト技術者交流会
  - 1998年9月に活動開始
    - » 現在1200名強の登録
    - » MLベースの議論と、たまの会合
  - http://www.swtest.jp/forum.html
  - お金は無いけど熱意はあるテスト技術者を無償で応援する集まり
  - "JaSST：ソフトウェアテストシンポジウム"も開催している
    - » 実行委員は手弁当／参加費は実費＋α
    - » 毎年4Qに東京で開催／今年はのべ約1500名の参加者
    - » 昨年は大阪・札幌でも開催／会場は満席
  - 「基本から学ぶソフトウェアテスト」や「ソフトウェアテスト293の鉄則」の翻訳も手がける
    - » ほぼMLとWebをインフラとした珍しいオンライン翻訳チーム



SESSAME: 組込みソフトの育成研究会

- 組込みソフトウェア技術者管理者育成研究会
  - Society for Embedded Software Skill Acquisition for Managers and Engineers
  - 2000年12月に活動開始
    - » 200名強の会員／MLベースの議論と、月イチの会合
  - http://www.sessame.jp/
- 中級の技術者を10万人育てる
  - PCソフトウェアのような「そこそこ品質」ではダメ
    - » 創造性型産業において米国に劣り、コスト競争型産業でアジアに負ける
    - » ハードウェアとの協調という点で日本に勝機があるはず
  - 育成に必要なすべてを開発する
  - オープンプロダクト／ベストエフォート
    - » 文献ポインタ集、知識体系（用語集）、初級者向けテキスト、スキル標準など
    - » 7つのワークグループ：組込みMOT・演習・MISRA-C・ETSS・子供・高信頼性
  - セミナーだけでなく、講師用セミナーも実施

講演の流れ

- 開発側の組織能力の向上に対するユーザ企業の責任
- テストによる組織能力の改善
- クラシックなソフトウェアテスト
- ソフトウェアテストの基本の「キ」
- 組み合わせのテスト
- テストプロセスの基本
- テストプロセス改善
- まとめ：ソフトウェアテストとは
- テスト「道」

ユーザ企業のミッション

- 適切「な」システムを作ること
  - 戦略的価値の高いシステムを作ること
  - 業務効率を高くできるシステムを作ること
- 適切「に」システムを作ること
  - 要求に合ったシステムを作ること
  - 信頼性の高いシステムを作ること

両者を別々に考えることは
果たして得策なのか．．．？

ユーザ企業の問題

- ユーザ企業の病理
  - コスト削減・納期短縮症候群
    - »「とにかくコストを下げてくれ」「とにかく納期を短くしてくれ」
  - 要求不確定症候群
    - »「関係部署との調整が難しくてね。でもカットオーバーは変わらないよ」
  - 受け入れテスト不能症候群
    - »「受け入れテストだから、とにかく業務の人間に触ってもらおう」
  - 無責任症候群
    - »「私使う人、あなた作る人」「プロなんだから任せておけばよい」

ユーザ企業の責任

- ITに無責任で経営は立ちゆくのか？
  - システム障害は、ユーザ企業が招いているような気がしてならない
    - » もちろん開発側の責任が重大なことは言うまでもない
  - ソフトウェア産業全体としてシステム障害が起きないような方策を ユーザ企業は一生懸命考えなくてはならない
    - » 監査には限界があり、市場原理では良貨は悪貨に駆逐されるかもしれない
- ユーザ企業はITに責任を持ち、 開発側と一緒に進めていく姿勢こそが重要である
  - どの要求がどのように変わりうる可能性があり、 どの要求は変わらないのか、を見極める必要がある
  - どうコストを削減したか、どう納期を短縮したかを評価する必要がある
  - 業務の根幹を為すシステムの評価は、自社主導で行う必要がある
  - 開発側を消耗させることは、自らの首を絞めることと同義である

ソフトウェアの品質は目を覆う状況である

- 「動かないコンピュータ」は無くならない
  - 成功するプロジェクトはわずか26.7%（日経コンピュータ）
    - » 止まってばかりの証券取引所
    - » 統合に成功するとニュースになる金融系システム
    - » トレンドマイクロのパターンファイルで30万件が問い合わせ
    - » ダイナミック・アップデートに失敗するデジタル家電
- 多発するソフトウェアのトラブル
  - エンタープライズ系システム：2005/6/20～7/19で報道されたトラブル
    - » 東急電鉄、松坂市、彦根市立病院、神奈川県教委、 寝屋川市、栗原市、浜田市、浜松市、郵政公社、南砺市、 彦根市、イートレード証券、スカイマーク・・・
  - 組込み系システム
    - » 国内の携帯電話：65万台が回収・無償交換、数十億円にのぼる損害
    - » 高級車のブレーキシステム：68万台がリコール
    - » 鉄道：ATCの誤作動により1ヶ月に50回以上の速度超過
    - » ダム：ゲートが開き貯水が流出

開発側のよくある負けパターン

- コストダウン優先によるデスマーチ化
  - 安物技術の導入→品質の低下→手戻り作業の発生 →残り予算減少による、さらなる安物技術の導入プレッシャーの増大
    - » オフショア、アウトソーシング単価低減、安物コンポーネント購入など
  - 結局、赤字プロジェクトになってしまう
- 納期達成優先によるデスマーチ化
  - 工数短縮による必要作業の省略→品質の低下→手戻り作業の発生 →残り工期減少による、さらなる納期プレッシャーの増大
    - » 設計軽視、レビュー削減、単体テスト削減、ドキュメント削減など
  - 結局、納期遅延になってしまう

ユーザ企業は
自らの首を絞めているのではないか？

目先の成功がシステム開発の目的？

- システムを安定的に供給できる産業を育てるという 視点が重要である
  - 今回のプロジェクトのQCDを達成すればいい、と思っている ユーザ企業がほとんどであり、開発側も同様に考えてしまう
  - この誤解によって個別最適に陥り、持続的な発想を阻害してしまう
    - » 日本人は個別最適が得意だと思われている？
    - » 個の総和が全体になるという要素還元的思想に基づいている
    - » 例）プロセス改善とプロジェクトマネジメントは相容れない
  - だからといって甘やかしたり言いなりになってよいわけではない
- 開発側は、持続的な活動を行う存在である
  - 持続的に開発現場を幸せにしなくてはならない
  - 持続的に利益を向上（≠維持）させなくてはならない
  - そのためには、持続的に「組織能力」を向上させなくてはならない
  - 決して開発側を“消耗”させてはいけない

「組織能力」の向上による開発側の強化

- 組織能力とは？
  - 開発力：技術力、管理力、経験力
  - 自律力：徹底力、目的理解力、カイゼン力
  - 融合力：共有力、全員参加力、俯瞰力
- 組織能力は簡単には向上しない
  - 組織能力の向上そのものも、組織能力の高さに依存する
  - 標準プロセスを整備し準拠するという思想よりも、身近なところから、 現場が自ら組織能力を向上していきたくなるようなプロセスにする
  - パートナーの組織能力も共に向上していく必要がある
  - 顧客も共に成長していく必要がある

開発側へのミッションに
「持続的な組織能力向上」を
取り入れる必要がある

開発側の組織能力を高めよう

- 組織能力は正のフィードバックがかかる
  - 開発力の高い技術者は、イキイキと仕事している(と思う)
  - イキイキと仕事をしている技術者は、指示待ちにならず自律力が向上する
  - イキイキと仕事をしている技術者は、その感覚を周りと分かち合おうとし、融合力が向上する
  - 高い自律力の技術者は、プロジェクトを通じて開発力を向上できる
  - 高い融合力の技術者は、他の技術者の経験を通じて開発力を向上できる
- 組織能力を上げる一つの方法は、品質にこだわること
  - 銀の弾丸は無い
  - 教育が必要なのは言うまでもない
  - コストや納期は安堵感、品質が高いと達成感？
  - 技術者は技術に触れていると楽しい
  - 仕事のカイゼンを通じて、小さな達成感を常に感じてもらう

品質

品質にこだわって開発側の組織能力を上げる

- 品質を向上しようと踏ん張らないのは、高い品質が自分たちの利益になると感じられないため
  - 品質の向上をコストダウンにつなげるコンセプト・方策を整理し共有する
    - »「品質を上げようとするとコストが掛かる」という考えは誤謬である
- 品質にこだわることを次のプロジェクトの成功につなげ、それを目に見えるようにする
  - ただの納期遅れに「次につながる」という言い訳をするのではなく、技術的に次につながっていくようなマネジメントを行う
  - 同じようなプロジェクトを次回はもっとよく開発できるようになってこそ、真のエンジニアリングと言える
    - » 同じプロジェクトを同じように開発するのは真のエンジニアリングではない

品質で開発側が幸せになる

- 品質で守る
  - 手戻りを減らすことで納期遅延や赤字プロジェクトを減らす
  - 信頼性を上げることで品質事故やクレームを防ぐ
- 品質で攻める
  - 自分たちの開発の弱みを把握しカイゼンし続けることで、コストダウンと納期短縮により競争力を強化し続ける
  - 信頼性を上げることで差別化し、ブランド力をつける
- 品質で幸せになる
  - デスマーチから抜けだし達成感のある仕事を行い、モチベーションが高く笑顔にあふれる現場を作る
  - 現場がやる気を出すことで、プロセスの改善や新技術の導入、チーム内の教育が活発になり、さらに競争力が高まる

恩恵を受けるのはユーザ企業

講演の流れ

- 開発側の組織能力の向上に対するユーザ企業の責任
- テストによる組織能力の改善
- クラシックなソフトウェアテスト
- ソフトウェアテストの基本の「キ」
- 組み合わせのテスト
- テストプロセスの基本
- テストプロセス改善
- まとめ:ソフトウェアテストとは
- テスト「道」

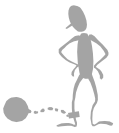
組織能力の改善によって手戻りを減らす

- 赤字や納期遅延の大きな原因の一つは手戻りである
  - 手戻りによって、前に進む作業が滞ってしまう
  - 手戻りが多いと、品質が低下するだけでなく、コストや納期もオーバーする
  - 手戻りが減れば、赤字削減や納期遅延防止に大きく近づく
- 手戻りを減らすには組織能力を改善する必要がある

  例）OOの組織的導入？

  プロジェクトマネジメント？

  プロセス改善？
- どの方策も確かに有効ではある. . .

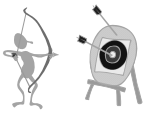


弱点を把握し、改善する能力が重要

- QCD向上のために手戻りを減らすには、「弱点を把握し、改善する能力」こそ必要である
  - 手戻りはバラバラに起こるわけではなく、弱点に多く発生する
  - 製造業の品質管理が強いのは、個々の技法やフレームワークではなく、この能力を組織的・継続的に発揮できる遺伝子があるからである
- 弱点とは何か
  - 想定通りに物事が進まない原因
    - » 不具合やリスク
  - プロジェクト的な弱点とプロダクト的な弱点がある
    - » プロジェクト的な弱点は、リスクが発生しやすいタスクである
    - » プロダクト的な弱点は、バグを作り込みやすい要件や設計、コードである
  - この2つの弱点は、複合して悪影響を及ぼすこともある

弱点を把握し、開発力を改善する

- どうやって弱点を把握し、改善するか？
  - プロジェクト的な弱点は、リスクマネジメントで把握し、プロセスやPMにフィードバックして改善する
  - プロダクト的な弱点は、テストで把握し、分析や設計、コーディングにフィードバックして改善する
- 弱点の把握には、リスクマネジメントやテストを組織的にきちんと行う必要がある
  - まず、しっかりテストやプロジェクト評価を行い、リスクや不具合を徹底的に洗い出す
  - 次に、きちんとした原因分析を行い、改善ポイントを特定する
    - 「作りっぱなし」ではいけない



テストによる組織能力の改善

- テストを強化する（質を高める）ことで、プロダクト的な弱点を抑えることができる
  - 短期的には、作り込んでしまったバグを可能な限り多く叩き出すことができる
    - テストを最後の砦として、お客様先にご迷惑をおかけしないようにできる
    - 少ない手間で早くたくさん危険なバグを検出することで、テストの生産性を向上し、より高い品質を達成できるようになる
  - 中期的には、プロダクト出荷時の品質指標を定めコストや納期とトレードオフをする素地を作ることができる
    - テスト消化率などの間接指標ではなく、「未消化テスト項目ごとの残品質リスク」などより直接的な品質指標を用いることができるようになる

テストによる組織能力の改善

- テストを強化する（質を高める）ことで、プロダクト的な弱点を抑えることができる
  - 長期的には、どういうバグが多いか、なぜそのバグを作り込んでしまったか、という「悪さの知識」をフィードバックし、開発力全体を向上することができる
    - 長期的な体質改善を進めることができ、開発全体の質や生産性が向上する
    - 「やらされ感」の伴うプロセス改善に、熱意を持って取り組むようになる
  - 「悪さの知識」のマネジメントが上手になると、運用時の障害管理の改善につながる
  - 上流で作り込んだバグが少ないとテストがスムーズに進み、テストの生産性が向上するため、より広い領域の（より多くの機能の）品質を保証できる

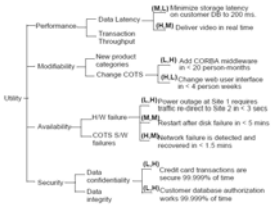
品質アーキテクトという役割

- 品質向上を上手に進めるためには、品質にまつわる様々なアーキテクチャを総合的に考えなくてはならない
  - 品質管理アーキテクチャ
    - 組織達成目標・組織成長目標と改善ターゲットの関係の明確化
    - プロセス品質とプロダクト品質のバランス
    - 品質保証と品質改善のバランス
    - 改善活動と教育のバランス
    - 改善を進めやすい組織構造と人事評価体系
    - 品質文化醸成の戦略
  - プロセス改善アーキテクチャ
    - プロジェクト達成と改善のバランス
    - トップダウンとボトムアップのバランス
    - 改善テーマの選定と優先順位付け
  - V&Vアーキテクチャ
    - レビュー・インスペクション・テストで検出する不具合のバランス
    - 開発とV&Vの工数バランス
    - レビュー・インスペクション・テストの工数のバランス
    - 品質保証と品質リスクのバランス

とても
クリエイティブな
ミッション

アーキテクチャの評価とテスト

- ATAMは誰の役目？
  - ATAM: Architecture Tradeoff Analysis Method
    - CMU/SEI-2000-TR-004
    - アーキテクチャ評価のためのフレームワーク
  - Utility Treesを用いて評価する
    - 品質特性を展開し、重要性とリスクとでトレードオフを行う
  - アーキテクトの役目？
    - しかしやっていることはエキスパートによるテスト設計とよく似ている

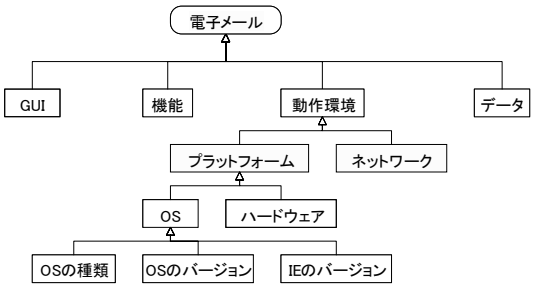


テストのアーキテクチャ

- テストももちろんアーキテクチャを考えなくてはならない
  - テスト観点の列挙とモデリング
  - テスト工数と品質リスクのバランス
  - テストの粒度とテスト設計工数のバランス
  - 網羅型テストと検出型テストのバランス
  - テスト実施プロセスのモデリングとムダ取り
  - 不具合知識の構造化と再利用
  - 上流でのテスト設計と下流でのテスト設計のバランス
  - （テスト容易性設計と開発速度のバランス）

テストをきちんと「開発」することで
上流から品質を作り込めるようにする

テスト観点の例



講演の流れ

- 開発側の組織能力の向上に対するユーザ企業の責任
- テストによる組織能力の改善
- クラシックなソフトウェアテスト
- ソフトウェアテストの基本の「キ」
- 組み合わせのテスト
- テストプロセスの基本
- テストプロセス改善
- まとめ：ソフトウェアテストとは
- テスト「道」

システム開発におけるテストの位置づけ

- クラシックなソフトウェアテストの位置づけ
  - テストは検査だから、最下流の工程である
  - テストは不具合を検出するだけの作業である
  - テストは確認するだけだから技術はいらない
  - テストはソフトウェアの動作を確認するだけであり、単なる仕様書との突き合わせ作業である
- モダンなソフトウェアテストの位置づけ
  - テストを含めて最上流から俯瞰的に品質を確保しないといけない
  - テストをきちんと設計しようとすることで、そもそも上流から不具合を作り込まないようにする
  - よいテストを設計するには高い技術力が必要であり、開発力を向上するためにはテスト力の向上が必須である
  - テストはお客様の（不）満足を出荷前に評価する作業なので、誰よりもお客様の業務、ひいては戦略を知らなくてはならない

よくあるクラシックなソフトウェアテスト

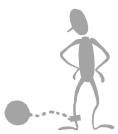
- 目を疑うようなことが頻発している
  - 実装が終わってからテストの準備を始めるので無駄な待ち時間が多い
  - いい加減な線表を引いて納期が来るまで徹夜する
  - テスト設計をせずにテスト項目だけを捻り出す
  - テスト対象の品質の状況が把握できないのでバグがちっとも見つからないテスト項目ばかり実施する
  - テスト項目の粒度が荒いので開発で再現せず修正されない
  - 開発の品質が低いのでテストが全然消化できない
  - テスト環境が把握できてないのでテスト対象の配布に時間がかかる
  - どのリリースに対するテスト項目か分からなくなる
  - 書いたはずのバグレポートが紛失して修正されない
  - などなど．．．

ソフトウェアテストの現状

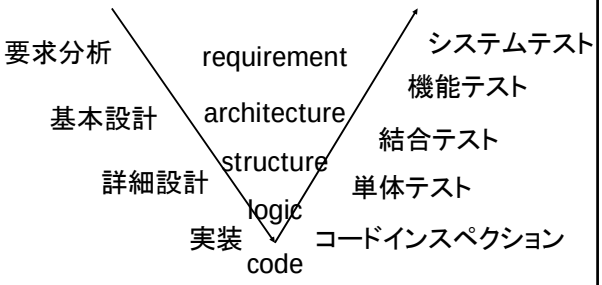
- 適切なテストプロセスやテスト技術を導入しないと失敗する
  - テストに技術はいらない、と思うと失敗する
    - » 適当に行き当たりばったりでテストすると、漏れやムラが大量に発生する
    - » 多くのテスト技術があることを認識し、自組織に合ったテスト技術を導入する
    - » テストの設計とレビューの設計は同義である
  - 最後にまとめてテストをすればいい、と思うと失敗する
    - » テストも開発と同じように分析・設計・実装が必要であり、管理も必要である
    - » フェーズ分けをせずまとめてテストをすると、テスト漏れが発生したり、原因分析に時間がかかる
- ソフトウェアテストは開発全体のボトルネックになっている
  - 全工程の3割～9割をテストに費やしている
  - 人海戦術で納期まで徹夜を繰り返すだけ、というのが現状だろう
  - テスト技術が低いため信頼性の保証にはほど遠い、という組織が多い
    - » 技術陣だけの問題ではなく、管理層や経営層の問題でもある

クラシックなテストのパラダイム

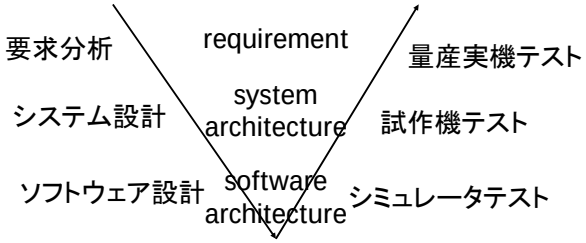
- Vモデル
  - 単体テスト／結合テスト／機能テスト／システムテスト／受け入れテスト
  - 回帰テスト
- ホワイトボックステスト／ブラックボックステスト
  - ホワイトボックステスト＝ソースコードテスト≒制御パステスト（パスカバレッジ）
  - ブラックボックステスト＝外部仕様テスト≒機能網羅テスト＋境界値テスト＋デシジョンテーブルテスト
- システムテストカテゴリ
  - ボリュームテスト／ストレージテスト／高頻度テスト／ロングランテスト
  - 構成テスト／両立性テスト／データ互換性テスト
  - 操作性テスト／セキュリティテスト・・・
- プロセス無しのテスト
  - 線表引きとテスト項目作成とテスト実施と不具合報告
  - 信頼度成長曲線



ソフトウェア開発のVモデル



組込みソフト開発の上位Vモデル



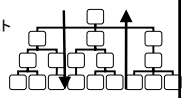
モジュールをテストしよう：単体テスト

- どんなテスト？
  - 開発者が行うテスト
  - モジュールレベルのテスト
  - 詳細設計やプログラミングに着目したテスト
- どんな手法でテストするの？
  - 境界値テスト
    - » モジュールの引数などに与えるテストデータを はじっこの値（境界値）にするテスト
  - 制御パステスト
    - » モジュール内のロジックを全て通すように テストデータを与えるテスト

両方とも必要！

設計をテストしよう：結合テスト

- どんなテスト？
  - 開発者が行うテスト
  - モジュールレベルのテスト
  - 概要設計や構造設計に着目したテスト
    - » 状態遷移設計やモジュール間構造設計に着目したテスト
- どんな手法でテストするの？
  - トップダウンテスト／ビッグバンテスト
    - » モジュール間の結合のミスを探すテスト
  - 状態遷移パステスト／状態遷移マトリクステスト
    - » 状態遷移図や状態遷移マトリクスのミスを探すテスト
  - など



コンパイル後に改めてテストしよう：機能テスト

- どんなテスト？
  - 開発者だけでなく、テスト担当者や品質保証部門も 行うテスト
  - 組み上がったソフトウェアレベルのテスト
  - 機能仕様に着目したテスト
- どんな手法でテストするの？
  - 機能網羅テスト
    - » 機能一覧を作って網羅するテスト
    - » 簡単そうだが意外にやらない
  - 境界値テスト
    - » 機能のパラメータに着目した境界値テスト
    - » 仕様のバグが見つかる

色々いじめてみよう：システムテスト

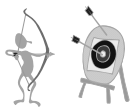
- どんなテスト？
  - 開発者だけでなく、テスト担当者や品質保証部門も行うテスト
  - ソフトウェアを使うユーザレベルのテスト
  - 要求仕様に着目したテスト
- どんな手法でテストするの？
  - ストレス系のシステムテスト
    - » ソフトウェアに負荷をかけるテスト
    - » よくバグが見つかるので手抜きしないでテストする
  - 環境系のシステムテスト
    - » 相性や互換性の問題を見つけるテスト
    - » データだけでなく、電気や熱の流れなどにも気を付ける
  - 評価系のシステムテスト
    - » どれくらい堅牢か、どれくらいユーザが満足するかの評価
    - » ユーザに使わせるだけでなく、きちんと考えて設計する

回帰テスト（リグレッションテスト）

- 変更・修正・保守の際のデグレードを防止するテスト
  - デグレードを検出して直すのは非常に難しい
    - » 変更個所に新たに作り込んでしまうバグ
    - » 変更個所以外に思わぬ影響を及ぼしてしまったバグ
    - » 抑えられていた“タガ”が外れてしまうバグ
  - 変更などのたびに、今まで行ったテストを全て行うのが原則
    - » 実際には工数が足りないので、どこを間引くかがノウハウとなる
    - » 変更の影響をあらかじめ十分検討できるプロセスとアーキテクチャが必要
  - 自動テストツール（ASQツール）をうまく活用することが重要
    - » 自動操作ツールや単体テスト自動化ツール（xUnit）を使う
    - » テストの自動化には独特のノウハウがある
    - » 最近はツールも安定してきた

回帰テスト（リグレッションテスト）

- 回帰テスト設計に銀の弾丸はない
  - 重要な機能をテストする
  - 操作フローで関連する機能をテストする
  - 似たような機能名の機能をテストする
  - 構造的に関連する機能をテストする
    - » モジュール呼び出し
    - » データ共有
    - » 外部エンティティ共有
    - » タイミング共有



回帰テストを
合理的に減らせるのは
良い設計の証拠である

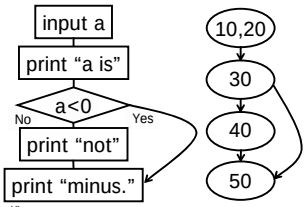
テストの狙い：ホワイトボックスとブラックボックス

- ホワイトボックステスト
  - プログラムの内部構造をもとにしてテストを設計する
    - » 主に初期に行う（単体テスト／統合テスト）
- ブラックボックステスト
  - プログラムの外部仕様をもとにしてテストを設計する
    - » 主に後期に行う（機能テスト／システムテスト）

互いに補いあう方法である

制御パステスト

- パスを網羅することで、
  ロジックを漏れなくテストする
  - パス：フローグラフ上の経路
    - » プログラムであればロジックの一つ
- パスを網羅するには
  - フローグラフを描く
  - パスの一覧表を作る
    - » ①：10→20→30→40→50
    - » ②：10→20→30→50
  - パスを通るデータを作成する
    - » ①：a = -1
    - » ② ：a = 0
  - 期待結果を導出する



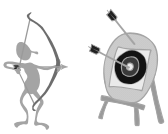
どの基準で制御パステストを行うか？

- モジュールの信頼性要求とテスト工数との
  バランスで「カバレッジ基準」を決める
  - カバレッジ基準が厳しいほど見逃す不具合の種類が減るので
    信頼性が向上するが、テスト工数は増加する
  - どのモジュールも一律の基準を用いようとするとバランスが崩れる
  - 各モジュールで定めたカバレッジ基準について、カバレッジ率を100%にする
    - » 適用除外はレビューで確認する
- 複合条件はレビューで確認し、C1基準でテストするのが現実的
  - 命令網羅（C0基準）
    - » 飛び越しやループのバグの
      見逃しが起こる
  - 分岐網羅（C1基準）
    - » 最低基準
  - 条件網羅（C/D基準）
    - » 複合条件のテストには不足
  - MC/DC
    - » if文の中の個々の条件のうち、
      単独で分岐に影響があるものを網羅する
    - » FAAのDO-178Bのレベル A基準
  - 複合条件網羅（C2基準）
    - » 条件が少数しか複合しない場合は必要だが、
      多数複合している場合は現実的でなくなる
  - 全パス網羅（C∞基準）
    - » 多すぎて現実的ではない

注）基準の表現は教科書によって異なることがある

境界値テスト

- 境界値を狙ってテストデータを設計するテスト
  - まず同値クラスと呼ばれる範囲に分割し、
    次に境界値を考えてテストデータを作成する
    - » モジュールの引数、返値の境界値
    - » if文の条件式など内部同値クラスの境界値
    - » テンポラリファイルの容量などのI/O周りの境界値
    - » 連続した範囲の境界値、順番の境界値、種類の境界値、時間の境界値
    - » 内容の境界値、メタデータの境界値
  - 同値クラスの抜けは、もっとも防ぎにくい
  - 中間値よりも境界値の方がバグを検出しやすい
  - 正常境界値だけでなく異常境界値もテストする



境界値テスト

- 境界値周りの不具合の代表例
  - if文などの条件文やwhile文などのループには、境界ずれや一つ違いのバグが多い
  - 大きさを持ったデータ構造には、バッファオーバーフローなど容量の限界値ギリギリの処理を忘れているバグが多い
  - 異なる個所で定義された同じ項目に関する仕様はあいまいだったり矛盾していることが多い
    - » 境界値テストを設計することできちんと期待結果を導出すると、あいまいな仕様に起因するバグを防止しやすい
- ブラックボックステストでの同値クラスの列挙は意外に難しい
  - 要求分析をしているのと同じ
  - お客様の使い方を徹底的に考え抜く
  - 規格や法規制、商慣習なども関係する
    - » 2000年問題に関する境界値テスト
  - 同様の原因でバグがありそうな同値クラスも挙げておく
    - » time_t構造体

リスト網羅テスト

- 一覧表にチェック欄を作り、チェックしていく
  - 基本中の基本だが面倒がって行わないことが多い
    - » ソフトウェアの持つ機能を漏れなくテストする
    - » 表を作って機能を全て書き、OKになったらチェックする

機能	*チェック欄*
機能1	レ
機能2	
機能3	レ
機能4	
機能5	レ
機能6	

マトリクス網羅テスト

- 2次元の組み合わせを網羅する方法
  - 機能でもデータでも何でも使える
    - » 行と列の組み合わせが網羅されるかをチェック

	条件A	条件B	条件C	条件D
条件1	レ			
条件2			レ	
条件3	レ			
条件4				レ
条件5		レ		
条件6				

    - »「ありうる組み合わせ」が網羅されるかをチェック

	条件A	条件B	条件C	条件D
条件1	レ	レ	レ	レ
条件2	レ	レ	レ	レ
条件3	レ	レ	レ	レ
条件4	レ	レ	レ	レ
条件5	レ	レ	レ	レ
条件6	レ	レ	レ	レ

デシジョンテーブル法

- 入力の組み合わせを網羅するためのテスト手法
  - 入力の組み合わせと、対応する動作の一覧表がデシジョンテーブル

		ルール1	ルール2	ルール3	ルール4
	スイッチX	○	○	×	×
条件部	スイッチY	○	−	×	−
	信号A	×	○	×	−
	信号B	×	○	×	○
動作部	ランプ点灯	○	○	×	×
	OK出力	×	×	○	×
	エラー出力	×	×	×	○

ルール → エントリ

○：Yes
×：No
−：どちらでもよい

カテゴリ型システムテスト

- システムテストとは
  - 機能の動作の確認後のテスト
  - 意地悪テスト／総合試験
  - 品質特性を確認するテスト
- ストレス系のシステムテスト
  - ボリュームテスト
    - » 大きなデータ、たくさんのデータを与えるテスト
  - ストレージテスト
    - » ディスクやメモリなどを残り少ない状態で使うテスト
  - 高頻度テスト
    - » 短時間にたくさん処理させるテスト
  - ロングランテスト
    - » 長時間実行させるテスト
- 環境系のシステムテスト
  - 構成テスト
    - » 他のものから悪影響が与えられないか、のテスト
  - 両立性テスト
    - » 他のものに悪影響を与えないか、のテスト
  - 互換性テスト
    - » データなどの交換をさせるテスト
- 評価系のシステムテスト
  - 障害対応性テスト
    - » 電源を引っこ抜くなどの障害を起こしてみるテスト
  - セキュリティテスト
    - » 機密保護などのセキュリティ機能の穴を突くテスト
  - ユーザビリティテスト
    - » 操作性や視認性などを評価するためのテスト

伝統的テスト完了基準の例

- 消化済みのテスト項目数
  - 「キロステップあたり200件のテスト項目を設計すること」
  - 「設計したテスト項目の98%以上を実施したこと」
- 規模あたりの予測残存不具合数・累積検出不具合数
  - 「予測残存不具合数がキロステップあたり0.05件以下であること」
  - 「累積検出不具合数がFPあたり0.45件を越えること」
- 未修整不具合の数と重要度
  - 「未修整の重バグが0、中バグが10以内、軽バグが100以内であること」
- 消化テスト項目あたりの検出不具合数
  - 「直近2日でバグが見つからないこと」
  - 「信頼度成長曲線（SRGM）の傾きが十分小さいこと」
- 会議結果
  - 「PMが了解すること」

講演の流れ

- 開発側の組織能力の向上に対するユーザ企業の責任
- テストによる組織能力の改善
- クラシックなソフトウェアテスト
- ソフトウェアテストの基本の「キ」
- 組み合わせのテスト
- テストプロセスの基本
- テストプロセス改善
- まとめ：ソフトウェアテストとは
- テスト「道」

システム開発におけるテストの位置づけ

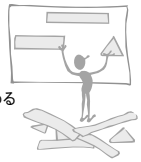
- クラシックなソフトウェアテストの位置づけ
  - テストは検査だから、最下流の工程である
  - テストは不具合を検出するだけの作業である
  - テストは確認するだけだから技術はいらない
  - テストはソフトウェアの動作を確認するだけであり、単なる仕様書との突き合わせ作業である
- モダンなソフトウェアテストの位置づけ
  - テストを含めて最上流から俯瞰的に品質を確保しないといけない
  - テストをきちんと設計しようとすることで、そもそも上流から不具合を作り込まないようにする
  - よいテストを設計するには高い技術力が必要であり、開発力を向上するためにはテスト力の向上が必須である
  - テストはお客様の（不）満足を出荷前に評価する作業なので、誰よりもお客様の業務、ひいては戦略を知らなくてはならない

ソフトウェアテストの基本の「キ」

- ソフトウェアテストはどんな技術？
  - 少ない手間で早くたくさん危険なバグを検出する技術
  - 製品出荷後のリスクを見積もる技術
  - 開発全体を改善するきっかけとなる技術
- ソフトウェアテストをきっかけにして開発全体を改善する
  - まずは基本的な手法を体系的に導入し、自分たちがどんなテストをしているのかを把握する
  - テスト漏れを少しでも減らすべくテストを分析し改善する
  - テストで検出したバグを分析し、バグを作り込んだ原因を明らかにし、要件定義や設計、マネジメント、プロセスを改善する

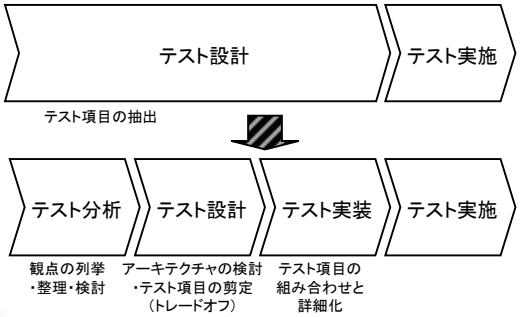
テスト設計プロセス：段階的詳細化

- 分析
  - テスト目標を定める
  - ユーザの要求や要求仕様、システム構造などからテストに必要な観点を列挙し整理する
- 設計
  - 整理されたテスト観点を基に、テストのアーキテクチャを決める
  - 段階的に詳細化しながらテスト項目を作成する
    - » フローグラフからパスを抽出するなど
    - » 当該製品の具体的なパラメータは当てはめない
- 実装
  - 実施可能なテストケースを作成する
    - » 具体的なパラメータを当てはめる
    - » テスト項目を組み合わせる
    - » 期待結果を導出するなど
- それぞれレビューが必要



段階的詳細化により適切に粒度を管理でき再利用性も向上する

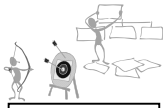
テスト設計プロセスの確立：分析・設計・実装



基本的なテストの方針

- 「テストしなければバグは見つからない」
  - 漏れなくテストを行う
  - 漏れなくテストを行うためには、思いつきでテストをあげてはならない

網羅



- 「大事なところをテストする」
  - 最もバグを検出しやすいテストを行う
  - バグが起きてはいけない順にテストを行う

ピンポイント

少ない手間で早くたくさん危険なバグを検出する

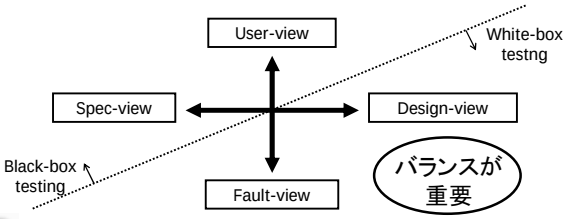
テスト設計プロセスの確立：ホワイト・ブラック・グレー

- ホワイトボックステスト
  - プログラムの内部構造や設計情報をもとにしてテストを設計する
    - » 主に初期に行う（単体テスト／統合テスト）
    - » テスト対象が小さい場合は効果が高い
    - » テスト対象が大きい場合は情報が多すぎて扱いきれない
- ブラックボックステスト
  - 要求仕様や外部仕様をもとにしてテストを設計する
    - » 主に後期に行う（機能テスト／システムテスト）
    - » テスト設計が比較的容易である
    - » 組み合わせのテストは発散してしまう
- グレーボックステスト
  - ブラックボックステストの精度を向上させるために、ホワイトボックスの情報を（少し）用いるテスト
    - » 組み合わせテストを合理的に減らすために用いる
    - » 開発が成熟しており、コミュニケーションが円滑でないと実現できない

適材適所が重要

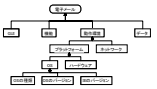
テスト設計プロセスの確立：代表的な4つの観点

- User-view
  - ユーザが何をするかを考える
- Fault-view
  - 起こしたいバグを考える
- Spec-view
  - 仕様を考える
- Design-view
  - 設計やソースコードを考える

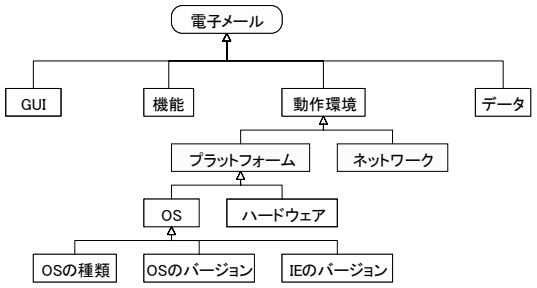


テスト設計プロセスの確立：テスト観点の分析

- テストの観点などをモデリングするための記法：NGT
  - 名称：NGT（Notation for Generic Testing）
    - » 一般的なテストのための記法
    - » テスト分析モデルとテスト設計モデルを記述する
    - » 個々のテスト技法で扱うものは記述しない
  - テスト分析モデル
    - » ビュー、フォーカス・クラス、関連による、テスト対象のモデリング
  - テスト設計モデル
    - » フォーカス・クラス、ズームイン/ズームアウト、テスト項目数の概算によるトレードオフ
      確定/暫定フォーカス・クラスによるリスクの明示
  - テスト設計の主な考慮事項
    - » テスト観点と同値クラス
    - » カバレッジ基準と境界値・閾値
    - » 組み合わせ



NGTによるテスト観点の分析の例



テスト設計の手法

- User-oriented testing
  - – 統計的操作テスト法
  - – ユースケース網羅法
- Spec-oriented testing
  - – 機能網羅法
  - – 同値分割法
  - – 境界値分析法
  - – GUIパステスト法
  - – 原因結果グラフ
  - – CFD法
  - – トランザクションフローテスト法
  - – 直交配列法：HAYST法
- Design-oriented testing
  - – 制御パステスト法
  - – 状態遷移テスト法
  - – データフローパステスト法
- Fault-oriented testing
  - – リソースパステスト法

基本的な手法は
最低限マスターしておく
必要がある

講演の流れ

- 開発側の組織能力の向上に対するユーザ企業の責任
- テストによる組織能力の改善
- クラシックなソフトウェアテスト
- ソフトウェアテストの基本の「キ」
- 組み合わせのテスト
- テストプロセスの基本
- テストプロセス改善
- まとめ：ソフトウェアテストとは
- テスト「道」

テストが爆発する原因は組み合わせである

- テストは無限であると主張する組織の多くは、テスト設計の際に何を列挙すればいいか分からない
  - 機能だけでいいか？
    - » データ構造／パラメータ範囲／機能の起動オプション／機能の起動方法／設定／動作環境／タイミング・・・
- テスト設計の際には、考慮すべき項目を始めに明示してから組み合わせなくてはならない
  - 項目を明示したうえで、どの項目を組み合わせるべきかを考える
    - » 組み合わせれば掛け算、組み合わせる必要がなければmaxでよい
    - » 禁則が多い場合は、意外に組み合わせ数は増加しない
  - 工数と相談して、組み合わせレベルを設計する

組み合わせそのものよりも
理解不足が原因の場合が多々ある

組み合わせのテスト

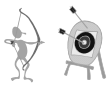
- 組み合わせを網羅してテストしなくてはいけない場合にはいくつかのパターンがある
  - 連続：A→Bでも B→Aでも不具合が起こるが、A→X→Bでは起こらない
    - » AとBが競合を起こしているがXで排他がかかる場合
  - 順序：A→Bで正常なのに、B→Aでは不具合が起こる
    - » Aが初期代入でBが読み出しの場合
  - 履歴：A→BでもA→X→BでもA→X→Y→Bでも不具合が起こる
    - » Aが共有メモリを壊してBが読み出す場合
  - 蓄積：AまたはA→Bがn回以上実行されると不具合が起こる
    - » Aがメモリリークを起こす場合
- 組み合わせの「段数」が深いほどみつけにくいバグになる
  - 組み合わせでテストが爆発すると嘆く前に、何段の組み合わせによるバグを検出したいか、とテスト工数をバランスさせる

ブラックボックスによる組み合わせテスト

- 全ての組み合わせを網羅すると、組み合わせバグは必ず検出できるが、テスト項目数が爆発する
  - 単一で網羅すると、テスト項目数は少なく済むが、組み合わせバグを検出できるとは限らない
- 直交配列表を使うと、2段に関する組み合わせバグを少ないテスト項目数で必ず検出できる
  - 3段以上の組み合わせバグについては検出できるとは限らない
    - » テストしたい組み合わせが分かっていれば割り付けられる可能性が高い
  - 複数の水準は、複数の因子を割り付けるか、多水準直交表を用いる
  - 禁則については補完を行うか、All pair法などを用いる
    - » きちんと対応するにはHAYST法を用いる
- あくまでブラックボックスの保証型テストに有効
  - 組み合わせバグの原因である内部の依存関係には着目していない
  - グレーボックスで内部構造を考慮した方が効率的になる場合もある

グレーボックスによるテスト設計

- アーキテクチャに着目してテストを設計する
  - 設計モデルに着目してテストを設計する
    - » 設計上存在する依存関係に着目してテストを行う
    - » malloc/freeを繰り返すように動作させるなど
- ウィークポイントに着目してテストを設計する
  - 設計時に考慮モレを起こしやすい部分に着目してテストを設計する
    - » 負荷がかかりやすい部分
    - » 構成変更が起こりやすい部分
- バグパターンに着目してテストを設計する
  - 設計者がやってしまいがちなバグを推測してテストを設計する
    - » 古くはバッファオーバーフロー
    - » バグパターンをどれだけ蓄積できるかがキー
    - » バグの見逃しのパターンにも着目する



設計の時点で組み合わせテスト工数を抑える

- 組み合わせが発生するボトルネックを分析してテスト工数を抑える
  - テスト工数の爆発やTestabilityの低下につながる設計を防止する
    - » 製品設計時にモジュール性を考慮した方がよいことは皆知っている
    - » でもモジュール性を減らした時にどうなるかを定量的に把握したことはない
- 製品設計時にテストを考慮してトレードオフを行う
  - テスト工程も考慮しながらトータルでトレードオフを行う製品設計
    - » 通常の製品設計時のトレードオフでは、テストのことなど頭にない
    - » 製品設計時には「楽だけどちょっと複雑」、テスト時には「工数爆発」
    - » 製品設計の工数がほんの少し増えても、テストで激減すればトータルで得

Testability（テスト容易性）を考慮して設計を行う

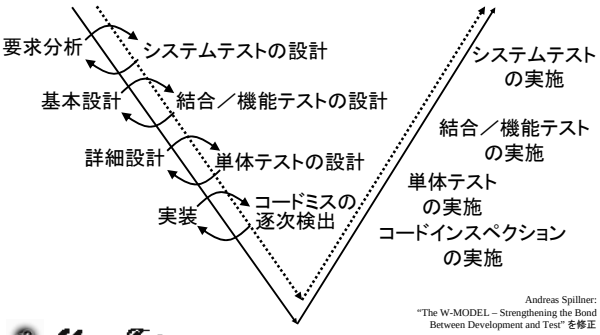
- Testabilityを考慮して設計を行う／レビューをする
  - Testability: テスト容易性（DFT: Design For Test）
  - James BachのTestabilitiy要素
    - » 操作性： ソフトウェアがうまく動けば動くほど、テストはどんどん効率的になる
    - » 観測性： 見えるものしかテストできない
    - » 制御性： ソフトウェアをうまくコントロールすれば、テストを自動化し最適化できる
    - » 分解性： テストの適用範囲を制限することにより、より速やかに問題を切り分け、手際よく再テストを行える
    - » 単純性： テストする項目が少なければ少ないほど、テストを速やかに行える
    - » 安定性： 変更が少なければ少ないほど、テストへの障害が少なくなる
    - » 理解容易性： より多くの情報があれば、それだけ手際よくテストができる

開発とテストが協調することで
品質の高い製品が早く安く出荷できるようになる

アーリーテスティング

- テストファースト（TDD）は基本中の基本
  - プログラムを組む前にテストを設計する
  - 開発時に統合環境上で単体テストを実装する
  - テスト結果をOKにするという発想で開発を進めていく
  - 開発前にテストを設計すると、それだけで品質が上がることもある
    - » きちんとテストを設計しないと、ただの作業手順の入れ替えになる
- 開発の段階的詳細化に並行してテストも詳細化する
  - 曖昧な要求事項を定量化できる
    - » 「きちんと」「素早く」の防止
    - » 境界値に関わる矛盾の予防
  - 要求事項とのトレーサビリティを確保しやすくする
    - » 仕様網羅率やテスト優先順位をきちんと定量的に扱う
    - » 仕様変更によるテスト設計変更を容易にする
  - テストを段階的に詳細化してテスト設計根拠を明らかにする
    - » テスト設計根拠に対してレビューを行う

ダブル・Vモデル



Andreas Spillner:
"The W-MODEL – Strengthening the Bond Between Development and Test" を修正

講演の流れ

- 開発側の組織能力の向上に対するユーザ企業の責任
- テストによる組織能力の改善
- クラシックなソフトウェアテスト
- ソフトウェアテストの基本の「キ」
- 組み合わせのテスト
- テストプロセスの基本
- テストプロセス改善
- まとめ：ソフトウェアテストとは
- テスト「道」

テストプロセスの基本

- テスト設計とテスト計画を混同してはいけない
  - テストにもプロジェクト管理が必要
- テスト設計・実装では、段階的詳細化を行う
  - まずはウォーターフォール的にでフェーズ分けを行い、サイクル型テストに移行する
- テスト実施では、短いサイクルで管理し改善する
  - 状況に応じて素早くアクションを取る「きっかけ」とする
  - プロセスに改善を埋め込む
- テストのバランスを管理する
  - 始めはバグ検出型テストや新規のテストを多めに
  - だんだん保証型のテストや回帰テストを多めにしていく
- テストプロセスを上流に移行し開発と並列に実施し、上流で品質を造り込む

リスクベースによるテスト完了基準

- 「リスク」を基にしてテストの優先度を決める
  - リスク指数の高いテスト項目を優先してテストする
  - 残りリスク指数が許容範囲内になったらテストを止める
    - » リスクを金額に換算できるとEVMに統合できる
  - テスト進捗・修正進捗をリスク指数で報告する
  - テストできなかった項目のリスクを明らかにし、別の手段を講じて潰す
- 「リスク」とは？
  - 動いた場合の価値／動かない割合／動かない場合のシステムに対するダメージ／動かない場合のプロジェクトに対するダメージ／不具合が見つかる割合
- リスク指数算出の考え方
  - リスクファクターを掛け合わせて考える
    - » リスク評価＝実現確率×損害（Rick Craig流）
    - » リスク優先度指数＝重要度×修正優先度×発生確率（Rex Black流）
  - 単純にかけ算で考えると失敗する
    - » たまにしか発生しないけど致命的な不具合は、リスク指数が低く計算されてしまう場合がある

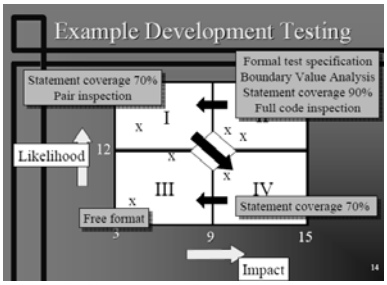
リスクベースドテストの例

	Likelihood						Impact			
	Complexity	New development	Interfacing	Technology	Experience level		User importance	Usage intensity	Safety	
Item 1	5	3	2	1	5	16	5	4	1	10
Item 2	2	1	2	1	2	8	3	3	1	7
Item n										

リスクを明らかにした上で
後回しにできる品質リスクを
顧客と相談し合意する

Erik van Veenendaal:
"Risk Based Testing in Practice"
http://www.bits-chips.nl
/portal/documents/Risk%20based%20Testing
%20Erik%20van%20Veenendaal.pdf

リスクベースドテストの例



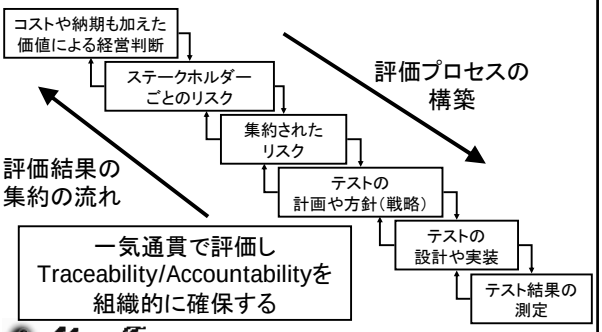
リスクに応じて
テストの方法や
濃さを変える

Erik van Veenendaal:
"Risk Based Testing in Practice"
http://www.bits-chips.nl
/portal/documents/Risk%20based%20Testing
%20Erik%20van%20Veenendaal.pdf

テスト結果の経営指標への集約

- 現状では. . .
  - 納期が足りない足りないと騒いでいるが、本当に工数が足りないのか
    それともオーバークォリティなのかどうか分からない
  - 手をこまねいているうちにプロジェクトが失敗に陥ってしまう
  - 自組織や受注組織の実力が分からないので、
    利益率の低い"安全な"開発しかできない
- 進行中のテスト結果をきちんと分析し、
  リアルタイムに経営指標に集約する仕組みを
  組織内に整えていくことが重要である
  - 必要な納期超過なのかオーバークォリティなのかが分かる
  - プロジェクトが失敗に陥る前にリスクマネジメントを実施できる
  - 自組織や受注組織の実力が分かり、野心的な開発が積極的にできる

テスト結果の経営指標への集約



合理的な品質評価プロセスの構築

- 品質、コスト、納期などのバランスを取って
  合理的な出荷判断を行う
  - 技術視点での出荷判断ではなく、経営視点での出荷判断にシフトする
  - 品質をリスクおよびコストで表し、単一の指標でトレードオフを行う
  - 時間がないから出荷、という事態は絶対に避ける
- 出荷に関わるステークホルダーごとに
  重要視する品質特性を明らかにする
  - それぞれ重要視する品質特性が異なる
  - 広義の品質としての品質特性も十分検討しなくてはならない
- 品質特性ごとにテストの計画や方針を定める
- 各品質特性を網羅的に評価する
  - テストできないところがリスクとして明確化される

講演の流れ

- 開発側の組織能力の向上に対するユーザ企業の責任
- テストによる組織能力の改善
- クラシックなソフトウェアテスト
- ソフトウェアテストの基本の「キ」
- 組み合わせのテスト
- テストプロセスの基本
- テストプロセス改善
- まとめ：ソフトウェアテストとは
- テスト「道」

テストプロセス改善

- 欧米ではテストプロセス改善モデルが
  いくつか発表されている
  - SW-TMM（イリノイ大学）
  - TPI（SOGETI）
  - 他にもいくつかある
- 基本的思想は開発プロセス改善と同じ
  - テストに関する要素作業を定め達成度をレベル分けする
  - 要素作業に多少の差がある
  - 開発プロセス改善との親和性に気を付ける必要がある
- プロセス改善は日本のお家芸・・・のはず
  - 全員参加とポストモーテムとナレッジセンターが改善のキモ
  - いかに上流フェーズに貢献できるか、を考える

テストプロセス改善：TPIのKey area

1. テスト戦略
   - A: 単一高位レベルテスト
   - B: 高位レベルテストの組み合わせ
   - C: 高位レベルテストに低位レベルテストor評価の組み合わせ
   - D: 全てのテストレベルと評価レベルの組み合わせ
2. ライフサイクルモデル
   - A: 計画、仕様化、実行
   - B: 計画、準備、仕様化、実行、完了
3. 関与の時点
   - A: テストベースの完成時点
   - B: テストベースの開始時点
   - C: 要求定義の開始時点
   - D: プロジェクトの開始時点
4. 見積もりと計画
   - A: 実証された見積もりと計画
   - B: 統計的に実証された見積もりと計画
5. テスト仕様化技法
   - A: 非公式の技法
   - B: 公式の技法
6. 静的テスト技法
   - A: テストベースのインスペクション
   - B: チェックリスト
7. 尺度
   - A: プロジェクト尺度（成果物）
   - B: プロジェクト尺度（プロセス）
   - C: システム尺度
   - D: 組織尺度（複数システム）
8. テストツール
   - A: 計画ツールと制御ツール
   - B: 実行ツールと分析ツール
   - C: テストプロセスの自動化強化

テストプロセス改善：TPIのKey area

9. テスト環境
   - A: 管理され制御された環境
   - B: 最適環境におけるテスト
   - C: 要求に即応できる環境
10. オフィス環境
    - A: 適切かつタイムリーなオフィス環境
11. コミットメントと意欲
    - A: 予算と時間の割り当て
    - B: プロジェクト組織に統合されたテスト
    - C: テストエンジニアリング
12. テスト役割と訓練
    - A: テスト管理者とタスク
    - B: （公式の）手法的、技法的、機能的支援、管理
    - C: 公式の内部品質保証
13. 方法論の展開
    - A: プロジェクトで固有
    - B: 組織で共通
    - C: 組織で最適化、研究開発
14. コミュニケーション
    - A: 内部コミュニケーション
    - B: プロジェクト内コミュニケーション（欠陥、変更管理）
    - C: テストプロセスの品質についての組織内コミュニケーション
15. 報告
    - A: 欠陥
    - B: 進捗（テストと成果物のステータス）、アクティビティ（コスト、時間、マイルストーン）、欠陥と優先度
    - C: リスクと提言、尺度による実証
    - D: ソフトウェアプロセス改善特性への提言

テストプロセス改善：TPIのKey area

16. 欠陥管理
    - A: 内部欠陥管理
    - B: 柔軟な報告機能による広範な欠陥管理
    - C: プロジェクト欠陥管理
17. テストウェア管理
    - A: 内部テストウェア管理
    - B: テストベースとテスト対象物の外部管理
    - C: 再利用可能なテストウェア
    - D: テストケースに対する追跡性システム要求
18. テストプロセス管理
    - A: 計画と実行
    - B: 計画、実行、監視、調整
    - C: 組織における監視と調整
19. 評価
    - A: 評価技法
    - B: 評価戦略
20. 低位レベルテスト
    - A: 低位レベルテストライフサイクルモデル（計画、仕様化、実行）
    - B: ホワイトボックス技法
    - C: 低位レベルテスト戦略

注意点
- 鵜呑みにしない
- 現実・現場の問題点を見る
- 改善を回すプロセスが重要

テストプロセス改善の実施例

- 泥臭いが効果的である
  - テストチームへの受け入れテストを行う
    - » バグが多すぎるとレポート作成に時間を取られてテストが進まない
  - テスト環境の構築を十二分に考慮してプロジェクトを進めておく
    - » 開発環境で動いた機能をテスト環境で動かすには想像以上に時間がかかる
    - » テスト環境と実機の差異をできるだけ減らすようにする
  - 仕様書をきちんと作成し保守しておく
    - » 仕様書が無いとリバースエンジニアリングになる
    - » 期待結果を調べるだけで時間がかかる
  - 不具合の切り分けでテストがストップしないようにする
    - » あらかじめ切り分け担当者を配置しテストと並列で実施する
    - » 切り分けが楽になるよう実機テストにソフト単体のバグを持ち込まない
  - ドキュメント作成の手間を減らす
    - » フォーマット／テンプレート／例文を準備しておく
  - 再現性を確保し手戻りを減らすように記録をきちんと取っておく

忘れられがちな改善ポイント

- テスト項目の粒度
  - 粒度とはテスト項目の記述の詳細度
    - » ログインする／ユーザ名にNsh、次にパスワードにESECと入力し[OK]を押す
  - テスト実施担当者のスキルによって粒度を変える必要がある
  - 期待結果やチェックポイントの粒度も問題になる
    - »「きちんと動作すること」は、どこの何を見れば分かるのか？
    - » 粒度を細かくすると設計工数がかかるが実施時に悩む時間は減る
      - 粒度を粗くしてテスト実施者の自由度を組み込むこともある
- モチベーションも重要
  - 集中力を落とさないようにテスト実施順序を工夫する必要がある
  - 単純作業に感じさせないように目的や責任、権限を与える必要がある
- テスト技術者に必要な才能
  - 誠実さ／忍耐／細部まで観察できる能力／経験をパターン化する能力／組み合わせの概算力／他の人の気持ちが分かること など

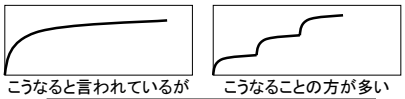
不具合が検出された際に増える作業

- 期待結果の作成に関する作業
  - 仕様書の調査や開発側へのヒアリング
  - 仕様書との突き合わせ
- 検出された不具合に関する作業
  - 検出された不具合の再現性を確認するための再現テスト
  - 検出された不具合の発生範囲を切り分けるための探索テスト
  - 検出された不具合に関するデータの収集と整理、および不具合報告書の作成
  - 口頭やミーティングでの不具合の内容に関する報告
- その後のテスト作業の調整に関する作業
  - そのテスト項目が動作しないことによって実施不能になるテスト項目の検討
  - 同様の不具合を検出するためのテストの再設計および再実装に関する検討

不具合率が
高いのは
大きなリスクである

信頼度成長曲線によるテストプロセス改善

- バグの分布と均一なテストを仮定して、残存バグ数を統計的に推定する方法
  - ゴンペルツ曲線やロジスティック曲線を当てはめる
    - 「ソフトウェア信頼性モデル－基礎と応用」, 山田茂著, 日科技連出版
    - フリーや商用のツールがある
  - 統計的に信頼できるほどプロセスが定まっておらず過去のデータもないのであれば、残存バグ数推定などは難しい
  - そもそも適切にテストが設計できてないので大きな抜けなどがある
  - 中身をきちんと考慮しないと統計値で出荷判断してはいけない



こうなると言われているが　　こうなることの方が多い

バグの分布のムラとテストのムラをきちんと考慮しないといけない

信頼度成長曲線によるテストプロセス改善

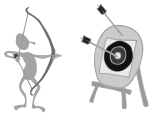
- 最低限何を考えるべきか：統計値が全てではない
  - どんなテストをどの程度設計したか
    - 要求カバレッジ、仕様カバレッジ、設計カバレッジ、実装カバレッジ
    - テスト設計技法の持つ特徴など
  - どんなテストをどの程度実施したか
    - 実施カバレッジ、テスト密度、テスト粒度
    - テストサイクル数、テスト件数など
  - どんなバグがどの程度見つかったか
    - バグの致命度、バグの数、発生頻度、発生条件
    - 発生場所の偏り、類似のバグの数、原因特定時間など
  - どんな修正をどの程度直したか
    - 修正にかかる時間、修正にかかる時間のばらつき
    - 類似のバグをどの程度修正したか、デグレードの程度
    - 未修正バグの数、未修正バグの重要度など
- 信頼度成長曲線を運用する前に管理図でプロセスを定量化してみるとよいだろう
  - ただし出荷基準として用いるのではなく、パターン認識で「悪さ」の兆候を見抜くツールとするのがよいだろう

テストプロセス改善の技法の例

- ディレイ分析
  - 見つかった（見逃した）バグが、本来はどのレビューやテストのフェーズで見つかるべきだったか、を分析することで問題点を把握する
  - 本来見つかるはずのレビューやテスト工程を改善する
  - 問題点のあるレビューやテスト工程で見逃しただろうバグを狙ってテストを設計する
- カバレッジ分析
  - カバレッジ基準を決めて、カバレッジを測定し、評価する
    - 決めたカバレッジは100%達成するようにする
    - 適用除外をきちんと管理し、分析するプロセスを決める
  - 見逃したバグが、どのカバレッジを網羅しなかったことによるものか、を分析することで問題点を把握し、改善する
    - カバレッジは制御パステストだけの概念ではない
    - 実施カバレッジの問題か、設計カバレッジの問題か
    - 同値クラスの抜けか、カバレッジ基準の甘さか、カバレッジ率の低さか、テスト設計の抜けか
    - 工数がかかりすぎるための間引きの失敗か
    - 組み合わせバグなのでカバレッジを上げても見つからないか

テスト設計の改善

- 不具合モード分析
  - バグは偏在する
    - 似たようなバグが何度も検出される
  - 過去の不具合のパターンを蓄積しておき、そこをピンポイントで狙ってテストする
    - ヒット率の改善
    - 組み合わせテストの補完
    - 回帰テストの改善
    - 開発上流の改善
  - パターンの蓄積の観点がキモとなる
    - キーワードで整理する
    - 機能ツリーで整理する
    - 構造パターンで整理する
    - バグを作り込みやすい特徴で整理する
  - 開発とのコミュニケーションを密にするツールにもなる
    - テスト側から「弱点」（不具合の起きやすい構造）を開発にフィードバックできる



テスト設計の改善

- テスト漏れは必ず発生する
  - 組み合わせによる膨大なテスト項目は消化できない
  - 組み合わせによってテスト項目数が増加するため全てのテスト項目は実施できない
    - テストの「間引き」が重要になる
  - そのためピンポイントでバグを検出する必要がある
- 不具合の起きそうなところをピンポイントで狙ってテストを設計する
  - 過去のテスト漏れを蓄積・分析し、テストの「間引き」の方法を改善する
  - 過去のバグを蓄積・整理し、バグが起きそうなところを推測する
    - すなわちプロダクトの「弱点」である

フィードバックテスト

フィードバックテストによるテスト設計の改善

- 不具合が起きそうな「弱点」を狙ってテスト項目やレビュー時の指摘事項を設計する
  - 過去に検出された不具合を精査し、同じような原因で発生していると推測できる不具合を列挙する
    - 始めは特定の製品の特定のバージョンに絞った方がよい
  - 不具合の原因となるメカニズムをモデル化する
    - モデル化されたメカニズムが、再利用のためのテスト資産となる
  - モデルに当てはまる構造を持つ機能のテストを設計する
    - きちんとレビューやインスペクションを行っている組織では、その結果を同じように資産化しておき、フィードバックテストに用いるべきである
- 開発とのコミュニケーションを密にするツールにもなる
  - テスト側から「弱点」（不具合の起きやすい構造）を開発にフィードバックできる
  - 開発側が「弱点」を把握しテスト側に伝える“フィードフォワードテスト”につながっていく

フィードバックテストによる本当のメリット

- フィードバックテストのメリット
  - テスト漏れを防ぐことができる
    - » 2度と同じテスト漏れは起こさないようにできる
  - 上手な「間引き」をすることができる
    - » バグの見つからないテストをうまくサボることができる
- 本当のメリット
  - 自分たちの開発の「弱点」のリストを手に入れることができる
    - » ツール屋さんやコンサルタントからは入手できない
  - バグを分析し自分たちの「弱点」を的確に把握ことで、効果的かつ効率的なプロセス改善を行うことが出来る
    - » プロダクトのバグの分析とフィードバックだけでなく、プロジェクトのリスクの分析とフィードバックも行うべきである

講演の流れ

- 開発側の組織能力の向上に対するユーザ企業の責任
- テストによる組織能力の改善
- クラシックなソフトウェアテスト
- ソフトウェアテストの基本の「キ」
- 組み合わせのテスト
- テストプロセスの基本
- テストプロセス改善
- まとめ:ソフトウェアテストとは
- テスト「道」

テストが上手になると

- 製品出荷時の信頼性や品質が向上する
  - ユーザ先で発生する不具合が減少する
- 納期遅れや残業・休出が減少する
  - 全体の3割～9割を占めるテスト工程が効率化される
  - リスクベースや経営視点によって合理的な出荷判断ができる
- 開発プロセスの改善が的確に進む
  - バグを作り込みやすい開発プロセスの「弱点」が露わになり重点的に改善できる
- ソフトウェア全体の開発力が向上する
  - 質の高いテストやレビューが開発とコンカレントに進むので、上流から信頼性の造り込みが可能になる
  - 質の高いテストをするためには、よい分析やよい設計、よい実装が必要だということに気付き努力するようになる
  - 受注側だけでなく、発注側の品質も向上するようになる

まとめ:ソフトウェアテストとは

- ソフトウェアテストはどんな技術か
  - 少ない手間で早くたくさん危険なバグを検出する技術
  - 製品出荷後のリスクを見積もる技術
  - 開発全体を改善するきっかけとなる技術
- ソフトウェアテストをきっかけにして開発全体を改善する
  - まずは基本的な手法を体系的に導入し、自分たちがどんなテストをしているのかを把握する
  - テスト漏れを少しでも減らすべくテストを分析し改善する
  - テストで検出したバグを分析し、バグを作り込んだ原因を明らかにし、要件定義や設計、マネジメント、プロセスを改善する

講演の流れ

- 開発側の組織能力の向上に対するユーザ企業の責任
- テストによる組織能力の改善
- クラシックなソフトウェアテスト
- ソフトウェアテストの基本の「キ」
- 組み合わせのテスト
- テストプロセスの基本
- テストプロセス改善
- まとめ:ソフトウェアテストとは
- テスト「道」

ソフトウェアテストの専門書

- アメリカだけで50冊以上、日本では主に10冊程度
  - » 体系的ソフトウェアテスト入門
    - Rick Craig等,日経BP, ISBN4-8222-8207-4
  - » 基本から学ぶソフトウェアテスト
    - Cem kaner等,日経BP, ISBN4-8222-8113-2
  - » ソフトウェアテスト技法
    - Boris Beizer,日経BP,ISBN4-8227-1001-7
  - » 知識ゼロから学ぶソフトウェアテスト
    - 高橋寿一, 翔泳社, ISBN4-7981-0709-3
  - » ソフトウェア・テストの技法
    - Glenford J.Myers,近代科学社,ISBN4-7649-0059-9
  - » ソフトウェアテスト293の鉄則
    - Cem kaner等,日経BP, ISBN4-8222-8154-X
  - » インターネットアプリケーションのためのソフトウェアテスト
    - Hung Q. Nguyen等,ソフトバンクパブリッシング, ISBN4-7973-2206-3
  - » 基本から学ぶテストプロセス管理
    - Rex Black, 日経BP, ISBN4-8222-8199-X
  - » 自動ソフトウェアテスト
    - Elfriede Dustin等,ピアソンエデュケーション, ISBN4-8947-1488-4
  - » ステップアップのためのソフトウェアテスト実践ガイド
    - 大西建児, 日経BP, ISBN4-8222-2968-8
  - » テストプロセス改善
    - Tim Kooman等,共立出版, ISBN4-3200-9734-3
  - » ソフトウェアテスト12の必勝プロセス
    - Rex Black, 日経BP, ISBN4-8222-8212-0
  - » アメリカのテストの専門書のリスト:
    - http://www.cigitallabs.com/resources/hotlist/publications-books-sqt.html

Alan Davis の鉄則:テスティングの原理

原理107 テスト項目と要求項目と関係づけよ
原理108 テストを行う時期よりずっと前にテストを計画せよ
原理109 自分のソフトウェアを自分でテストするな
原理111 テスティングは欠陥の存在をあらわにするだけだ
原理113 エラーを見つけてこそテストがうまくいったといえる
原理114 半分のエラーは15%のモジュールで発見される
原理116 テストケースには期待される結果を含めよ
原理117 無効な入力をテストせよ
原理118 常に過負荷状態のテストをせよ
原理125 エラーの原因を分析せよ
原理126 エラーを個人のものにするな

Kanerの「よい」テスト

- よいテストは、高い確率で不具合を検出できる
- よいテストは、冗長性がない
- よいテストは、同様のテストのうち最良のものでなければならない
- よいテストは、単純すぎたり、複雑すぎたりしてはならない

SPMNのベストプラクティス

- テストをチーム全員のミッションとせよ
- はじまるよりずっと前にテスト計画を立てよ
- きちんとテストを設計し、行き当たりばったりにするな
- 仕様策定や基本設計の段階からテストせよ
- テストを進捗条件に使え
- テスト項目を仕様と結びつけ、リスクを記述せよ
- テストは早くから頻繁に行え
- テストに必要な部品やツールも一緒に開発せよ
- 統合化テストツールを使え
- コスト、カバレッジ、結果、効率、見逃しを測定せよ
- テストを特別扱いするな

おすすめの鉄則:目的・技法・報告

003 テストは幅広い顧客へのサービス業と心得よ
004 顧客の価値と向き合うこと
008 プログラムの失敗を成功の母とせよ
010 テスト「完了」と聞いたら注意すべし
011 テストで品質が保証できるなどと思うな
021 技術的、創造的、批判的、実践的思考が優れたテストのカギ
023 漫然とテストすることはテストとは呼ばない
025 モデリングはテストの決め手だ
031 本当の「要求仕様」は何かを把握せよ
042 混乱をテストに活かすべし
043 慣れは見落としの素、新鮮な眼をいつも絶やさずに
058 障害レポートはテスト実施者の名刺である
065 障害管理システムをプログラマの評価に使うな
076 再現しないエラーも全て報告せよ。時限爆弾になる恐れがある。
085 問題点をはっきり報告し、解決策は書かない

おすすめの鉄則:自動化・管理・戦略

106 ゴミクズを自動化しても意味はない
116 GUIテストツールは、互換性と習得性とサポートで選べ
118 テストの自動化はソフトウェア開発そのものである
119 テストの自動化は決して安くない投資である
136 自動化よりテスト容易性を上げることにまず投資せよ
154 バグを憎んで人を憎まず
158「管理」文化にするな
162 要求は常に変更されるものだと心得よ
163 機能、信頼性、時間、コストの間のトレードオフ関係に注意しろ
185 充分なテストとは、正しい状況判断を下すのに充分な情報を得られること
198 数字しか見ない経営陣ほど危険なものはない
199 バグ総数によるプロジェクト進捗測定はするな
226 スタッフの士気こそ貴重な財産
282 テストの質は戦略に左右されると肝に銘じろ
288 テストをレベルに分けて作戦を立てろ
289 グレーボックステストを実施せよ

テストの達人は開発の達人である

- 達人は、間合いを掌握する
  - 網羅することで、抜けやモレ、思いこみによるバグを無くす
  - 品質とは何か、を広く考えてユーザの満足度を向上する
- 達人は、一刀両断にする
  - バグの出やすい弱点を把握し、改善する
- 達人は、道具の手入れを欠かさない
  - 開発の最初からツールを考慮し、骨の髄まで使いこなす
- 達人は、型に始まり型を崩し型に戻る
  - 技法の原理原則を理解し、最大限の効果を発揮させる
  - きれいな設計・実装のソフトウェアは、テストも楽である:テスト容易性設計
- 達人は、気配を感じ取る
  - バグやリスクが発生する前に予兆を検知し未然防止する
- 達人は、鍛錬を欠かさない
  - 現状に満足せず、常に問題点を探し改善するとともに、勉強を欠かさない
- 達人は、達人を知る
  - 自分からレビューをしてもらい、良いコードを読み、コミュニティに参加する
- 達人は、刀を抜かずに相手の心を斬る

Boris Beizer のテスト「道」

フェーズ0 ー テストはデバッグの一部である

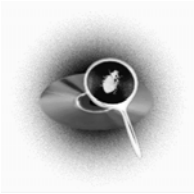
フェーズ1 ー テストの目的は、
ソフトウェアが動くことを示すことである

フェーズ2 ー テストの目的は、ソフトウェアが動かないことを
示すことである

フェーズ3 ー テストの目的は、何かを証明することではなく、
プログラムが動かないことによって発生する
危険性をある許容範囲までに減らすことである

フェーズ4 ー テストは行動ではなく、テストをしないで品質の
高いソフトウェアを作るための精神的訓練である

ご清聴ありがとうございました



電気通信大学 西 康晴
http://blues.se.uec.ac.jp/
nishi@se.uec.ac.jp