派生開発における
母体に由来するバグとその対応

JaSST'09 講演資料

株式会社 システムクリエイツ
代表取締役 清 水 吉 男
URL=http://homepage3.nifty.com/koha_hp
shimz@nifty.com

agenda

- 派生開発時に発生するバグには、新規開発時とは違った性質のバグが混じっている。
- 派生開発という制約の多い開発状況に由来するものもあるが、新規開発時の設計、実装方法に由来するものも少なくない。
- そして、原因をさかのぼったところには両者共に「ソフトウェアエンジニアリング」に対する知識・技術の不足がある。

1. 派生開発の特徴
2. 派生開発に於けるバグの特徴と対応
3. 派生開発でのバグの発症を減らす根本的方法
4. ソフトウェアエンジニアリングの不在

1. 派生開発の特徴

- 実際のソフトウェア開発の案件のほとんどは派生開発である
  - 90%は派生開発と思われる
- それにもかかわらず、派生開発と新規開発の違いを適切に認識していないために、新規開発とは違った原因でバグが埋め込まれる

保守開発との違い

- 「ソフトウェア保守」の最新の定義（JIS X 0161:2008）

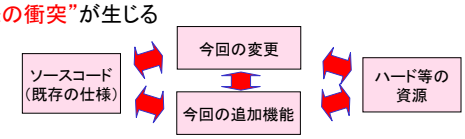
参照 http://www.jisc.go.jp/

保守のタイプ		作業内容
訂正	是正保守	ソフトウェア製品の引き渡し後に発見された問題を訂正するために行う受身の修正.
	予防保守	引き渡し後のソフトウェア製品の潜在的な障害が運用障害になる前に発見し、是正するための修正.
	緊急保守	是正保守実施までシステム運用を確保するための、計画外で一時的な修正.
改良	適応保守	引き渡し後、変化した又は変化している環境において、ソフトウェア製品を使用できるように保ち続けるために実施するソフトウェア製品の修正.
	完全化保守	引き渡し後のソフトウェア製品の潜在的な障害が、故障として現れる前に、検出し訂正するための修正.
	改良保守	新しい要求を満たすための既存のソフトウェア製品への修正.

- 現実の開発は「保守」では説明がつかない状況が生じている
  - 「携帯電話」から「ケータイ」への過程は「保守」か？
  - 「プロダクトライン」の開発は「保守」か？
  - 「インクリメンタル開発」の2回目以降は「新規開発」か？

複雑な仕様の衝突が生じる

- 派生開発では、新規開発とは比較にならないほど"複雑な仕様の衝突"が生じる



  - この"仕様の衝突"に対応する方法を持っていなければ、担当者の思い込みと勘違いの状態を解消できず不適切な変更を許してしまうことになる
- 複数の担当者の間でも、担当領域間で作業の衝突が生じる
  - ソースコードを変更する前に、ソースコードのレベルで衝突状態を知る方法がないと混乱する

"部分理解"の制約を受ける

- 現実には「全体を理解」できる状況ではなく、思い込みと勘違いが混入する
  - 理解するための資料やソースコードを読み解く技術が不足
- 「全体を理解すれば問題は解消する」という強い思いが、理解できなかったときの対応を備えていない

このような中で安易な「一人プロジェクト」が行われている

- 最初から派生開発では「部分理解」の制約を受けるという前提で、思い込みと勘違いへの対応策を講じるべき

変更方法も1つとは限らない

- 見掛け上は同じ結果を得る変更方法が複数ある
  - 依頼：ある条件のときは表示の位置をB欄からC欄に変更する
    - 対応1：表示処理のところで条件を判断して表示の場所を変更する
    - 対応2：早い段階でデータの区分を変更してC欄に出るようにする
    - 対応3：データ区分とは別に表示区分のコードを導入して対応する

テストでは、どの対応でも「OK」となる

- ただし、どの方法を選ぶかによって、後に「潜在バグ」となって顕在化する
  - 担当者の経験の深さや分担範囲によっては選択肢が見えない可能性がある

影響範囲にも2種類ある

- 今回の変更依頼に関連する影響範囲には2つの異なったタイプがある
  - ① 今回変更される仕様に直接関係している仕様
    - 必須関係や排他関係など何らかの依存関係のある機能の仕様
    - 類似の操作を持つ機能の仕様
    - 同じデータを扱う機能の仕様
  - ② 母体の作り方によって関係する箇所
    - 抽象化されたモジュールに関係している箇所や機能の仕様
    - 逆に抽象化されていないことによる波及もある
    - これまでの派生開発で不適切な変更を加えたことによる波及

①・・・仕様書や設計書などの文書から発見できる可能性がある
②・・・ソースコードを見ないと発見できない

2. 派生開発に於けるバグの特徴と対応

- そこで行われているプロセスが、派生開発特有の状況に適切に対応できていないために、新規開発とは違ったバグが入り込むことになる
- 変更ミス、変更モレ：
  - 変更した箇所は他の機能でも使っていた
  - 他に関連して変更すべき箇所があったことに気付かなかった
  - 追加機能を受け入れる際の変更で既存機能に障害が発生
- 不適切な変更：
  - 見かけは依頼通りに変更されたように見えるが、他にもっと適切な箇所、適切な方法があったことに気付かなかったため、その後の派生開発のなかで問題となって発症する

「原因プロセス分析」のモレ

- 派生開発におけるバグの原因には2種類ある
- 今回実施した派生開発プロセスに起因するもの
  - 変更要求を捉えないまま該当箇所を変更したことで変更モレが生じた
  - 思い込みと勘違いを防ぐ手だてを講じないままいきなりソースコードを変更したことでデグレを起こした

⇒ XDDPで対応可能

- 母体のソースコードに本当の原因があるもの
  - アーキテクチャが障害となって既存の機能の応答が悪くなった
  - 一部機能の処理が増えたため通常の操作が時々エラーとなる
  - 状態遷移の関数が5000行を越えるため間違って変更した

プロセス

新規開発崩しのプロセスが気付きを阻害している

- 変更の依頼は一般に「仕様」のレベルで届く。そのまま対応したのでは変更モレなどが生じる

機能仕様書を変更し、 設計書の該当箇所を変更し、 該当する関数仕様を変更し、 そうしてソースコードを変更する、 という「新規開発崩し」のプロセスを実施した	⇒	依頼された変更箇所に直接関連している箇所だけしか変更されない 母体のソースコードの作り方による影響範囲に気付きにくい

- 対応策：
  - 変更に対応したプロセスを導入し、変更仕様を一手に扱う要求仕様書が必要（XDDPで対応可能）
  - 影響範囲に気付くための資料を準備することも有効

プロセス

見つけ次第にソースコードを変更

- 不適切に作られたソースコードを理解するには相当な技術と経験が必要

思い込みと勘違いのままの状態で、”ここ”と思った箇所のソースコードをいきなり変更してしまう	⇒	変更すべき箇所がモレた 他の機能が使えなくなった 変更の意味を取り違えた

- 対応策：
  - 変更要求を捉え、変更仕様とセットで把握する（XDDPで可能）
  - ソースコードを理解し表現する技術（設計エンジニアリング技術）

プロセス

潜在バグの顕在化

- 変更の要求を満たす実現方法は1つとは限らない

適切な変更かどうかチェックされずにソースコードが変更された	⇒	後の派生開発での変更でバグになって顕在化する
見付次第にソースコードを変更したことで、より良い変更方法への切り替えに支障を来した		
処理の一部をカット＆ペーストで複製し、変数名を変更してカモラージュした	⇒	後の変更で関連箇所から見落とした （今のEditorでは発見できないケースがある）

- 対応策：
  - 変更要求や理由を把握したり、担当者の思い込みと勘違いを防ぐプロセスを導入する（XDDPで対応可能）
  - 保守性の品質要求を課すことで不正な作業を規制する

プロセス

機能追加に伴うバグ

- 機能追加の場合、追加機能の要求仕様しか書かれていない

追加機能の要求仕様は作られるが、これを現状のソースコードに組み込むための変更が追加機能の担当者任せになっていて、不適切な方法で機能追加が行われていることがチェックされていない	⇒	既存機能が壊される
		後の潜在バグを仕込む

- 対応策：
  - 「新しい機能を追加する」という変更要求を立てて、この機能追加を受け入れるための変更仕様を記述する（XDDPで対応可能）
    - その結果、機能追加の場合は「追加機能要求仕様書」と「変更要求仕様書」の2種類の要求仕様書で対応することになる

母体

不適切なアーキテクチャに起因するケース(1)

- タスクのアーキテクチャ設計が不適切なため、今回の変更によって新たな不具合が発生する

新規設計の段階でタスクの負荷分散が図られていない	⇒	機能追加に伴ってタスクの応答性が低下する
いくつかのタスクが同じ優先度に設計されている	⇒	動作が保証されないし、今回の派生開発で変更が加わったことで今までと動作が変わる

- 対応策：
  - 変化に強いアーキテクチャ設計のパターンを学習する
  - 「AFD」「タスク間シーケンス」などを活用してシミュレーションを行う

母体

不適切なアーキテクチャに起因するケース(2)

- 一般に変更は「データ」部分に発生するが、不適切なアーキテクチャによって変更箇所が散らばる

機能中心のタスク設計が行われている	⇒	変更箇所があちこちに散らばるため、変更モレが起きやすい
タスク間のグローバルデータが多用されている	⇒	アクセスされるタイミングが読めないために、今回の変更によって内容が壊される

- 対応策：
  - 「データ中心アーキテクチャ設計」や「オブジェクト指向アーキテクチャ設計」などのアーキテクチャパターンを研究する
  - 「往復メッセージ」とタスクの負荷分散でグローバルデータを回避

母体

不適切なアーキテクチャに起因するケース(3)

- メイン関数の周期構造の中で、フラグを判定してイベントを拾う方法では、新たな変更に対応できないケースがある

メインモジュールの周期処理のなかでフラグの判定によってイベントの発生を認識する構造になっている	⇒	今回の変更によって周期の時間が変わったことでイベントを拾う順序が変わってしまい，エラーとなる

- 対応策：
  - イベントをコード化して「リングバッファ」で対応する
  - 「RTOS」を導入する

母体

データ構造の選択を間違えた

- 新規開発で持ち込まれた不適切なデータ構造によって、派生開発時に障害となることもある
- データ構造を、派生開発のなかで作り替えることは困難

機能の使われ方を考えずに単純な配列構造だけで作られた	⇒	データの追加や削除が頻繁に行われるようになったところで応答性の悪化が表面化
データの受信処理でキューの制御を持ち込むべきところを、リングバッファで対応していた	⇒	通信速度の変更によってリングバッファがオーバーフローし、データの取り損ないが表面化

- 対応策：
  - データ構造とアルゴリズムの学習

母体

モジュールの凝集度や複雑度が悪い

- 母体のソースコードの凝集度や複雑度が悪いことによって、モジュールの再利用性に支障を来たしたり、変更に伴って劣化が進む

一つのモジュールの中に複数の機能が組み込まれていて、この中の1つだけを利用できない	⇒	一部の機能をコピーして、新しく作るモジュールに埋め込むたことで、変更モレが生じる
母体のモジュールが既に論理的凝集になっていて論理的凝集の選択肢を増やして対応	⇒	理解性が悪くなり、変更ミスが起きる

- 対応策:
  - モジュールの分割基準や尺度の学習
  - リファクタリング

母体

巨大な状態遷移の処理関数

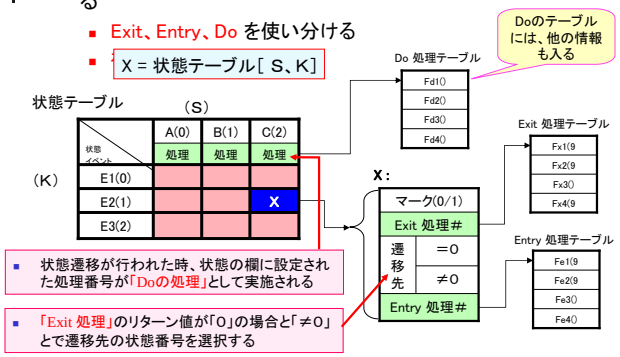
- 組み込みシステムでは状態遷移がコアであり、派生開発で変更の対象となる可能性が高い

「SWITCH文」の2段構成と、そのなかで「IF文」が使われるため、複雑度は800を越すことも少なくない	⇒	理解が不十分な状態で間違った変更が行われる
それぞれの「CASE」の処理をコピー＆ペーストで作られると、マトリクスが正しく選択されることの検証に、すべての組み合わせのテストケースが必要になる	⇒	テスト工数の不足を招き、テストモレを誘発する

- 対応策:
  - マトリクスを二次元のデータで作成することで、状態遷移のメインの関数は数10行で実現できるし、マトリクスが正しく選択されるかどうかの確認はマトリクスの大きさに関係なく4箇所で済む

（参考）状態遷移をデータテーブルで実現する方法

- 状態とイベント#から該当する状態テーブルのセルを特定する
  - Exit、Entry、Do を使い分ける
  - X = 状態テーブル[S、K]



- 状態遷移が行われた時、状態の欄に設定された処理番号が「Doの処理」として実施される
- 「Exit 処理」のリターン値が「0」の場合と「≠0」とで遷移先の状態番号を選択する

3. 派生開発でのバグを減らす根本的方法

- バグはプロセスに起因している
- そこで行うプロセスが要求に対して適切であれば、バグが入る余地はない
- 1種類のプロセスでは、変更と機能追加という全く性質の異なった要求を満たすことはできない。
- 派生開発でのバグの多くは、この要求とプロセスのミスマッチから起きている

XDDPによる発生開発プロセスを導入する

- XDDPのプロセスを導入することで、派生開発のプロセスに起因するバグの多くが姿を消す

変更要求と機能追加の二本立ての要求仕様書 要求と仕様の階層表現	⇒	変更モレ、変更ミスが大幅に減る
ソースコードレベルの変更仕様 変更プロセスに於ける3種類の中間成果物	⇒	ソースコードを変更する前に、変更仕様〔変更内容〕について3つの視点からレビューが可能
プロセスと成果物が合理的な連鎖を構成している	⇒	成果物間での検証が容易

変更要求仕様書を用意する

- 変更を「要求」と「仕様」の階層で捉える
  - ソースコードの該当箇所を変更仕様として変更要求の下に記述しておくことで、あとでそこに戻ってくることができる
  - 影響範囲に含まれる変更仕様を1つの変更要求の下にまとめることで、さらに連想を誘導する
  - その後に発見したより良い変更箇所や変更方法へ切り替えることに抵抗がなくなる
- 機能追加を、同時に「変更要求」としてとらえ、追加機能を受け入れるための変更方法をそこにまとめることができる
  - 既存機能への副作用を防止する効果が得られる

保守性の要求仕様の設定と設計技術の確立

- 新規開発でも派生開発でも「保守性」の品質要求を仕様化し、それが守られていることをテスト（検証）する仕組みが必要
  - 新規開発と派生開発では保守性の要求仕様は異なる
- ソースコードの静的検査
  - ソフトウェアのテストの一部にスタティカル･テストを取り入れて、作り方を検証する
- 一方で、エンジニアには保守性を実現するための技術の習得を推進する
  - 要求に対応させたアーキテクチャ設計技術
  - タスクの外にグローバルデータを出さない設計技術
  - モジュールの分割と尺度に基づいた設計技術　など

機能を実現するための設計技術とは別の技術

派生開発向けの資料の整備

- 「影響範囲に気付く成果物」（補助成果物）は、新規設計では必ずしも必要としないが、派生開発時には欲しい資料である
  - 機能間の依存関係を表すマトリクス
  - 各種のパラメータとそれに関係する機能のマトリクス
- 要求仕様とのトレーサビリティ
  - トレーサビリティ･マトリクス＝要求仕様と設計書やソースコードの関連がわかる
  - ツールや番号の付加による追跡性の確保

ビジネス系の保守開発の世界では用意されている可能性がある

4. ソフトウェアエンジニアリングの不在

- 派生開発では、変更行数の割に多くのバグが検出されていることが1000行当たりのバグの発生率でわかっている
- 原因の過半は新規開発時の設計･実装の仕方にあるが、派生開発時にもソースコードの読解技術が不足しており、機能追加においては新規開発と同じ問題が生じている
- 背景に、ソフトウェアエンジニアリングの技術の不足という問題が横たわっている

ソフトウェア･エンジニアリングが無視されている

- 新規開発時に、要求エンジニアリングや設計エンジニアリングの知識や技術が欠落した状態で設計され実装されている
  - 設計プロセスでは、分割･階層化を繰り返すことで、単純なモジュールの集合に変換する
  - 分割･階層化の中で、プロセスと成果物が合理的に繋がっていることで、問題の発生が抑制され発見も容易になる
  - それぞれの設計の階層には意味があるにも関わらず、それらを無視したプロセスが実施されている
- 設計エンジニアリングの技術が不十分なために品質を織り込む機会がない
  - こうして作られたソースコードを読み解くのは苦痛

ソースコードから理解したことを表現できない

- 適切な新規設計の技術の不足は、派生開発でソースコードを読んで理解したことを表現する手段がないことを意味する
  - 「アーキテクチャ設計」は一般に派生開発では出てこないが、その他の設計エンジニアリングの技術は派生開発でも必要
  - ソースコードからわかることは「仕様」であって、直接的には「要求」や「理由」は簡単にはわからない
  - 理解したことを表現しれたものがなければ、本人も周りの人も、彼が理解しているかどうかを知る方法はない
  - そこに存在した 「要求」や「理由」がわからなければ、そこを変更するのが怖くなり、同じような処理を複製する方法を選んでしまう

派生開発の方が難しい

- 設計エンジニアリングを十分に習得していなくても、要求仕様が適切であれば、個々の機能を実現するという範囲であればソースコードが書けることもある
  - 機能追加においては、表面的には「仕様通り」の動きをするソースコードは書けるだろうが、望ましい設計作法は満たさないだろう
  - 派生開発では「アーキテクチャ設計」の機会はほとんどないために、この技術の不足はほとんど障害にならない
- ただし、このようなレベルの人が、派生開発で他人の書いたソースコードを読んで設計の意図を理解することは難しい
  - 好き勝手に弄られたソースコードの場合は、さらに読み解くのは難しくなる

ソフトウェアエンジニアリングを習得すべき

- 多くの人は、ソフトウェアエンジニアリングを“ドキュメントを作るための方法”と誤解している

「ソフトウェアエンジニアリングは、ドキュメントを作成するためのものではない。品質を作り込むためのものであり、品質が向上すれば手戻りが減少する。手戻りが減少すれば納期が早まる」

（「実践ソフトウェアエンジニアリング」ロジャー S. プレスマン著）

- プロセスを固定してしまおうとするのは、ソフトウェアエンジニアリングを習得していないために、要求に対して適切なプロセスを選択し、自在に組み立てることができないところから起きている

ソフトウェアエンジニアリングの欠落は、新規開発だけでなく、派生開発の場面でも障害となっている

参考文献

- 参考文献

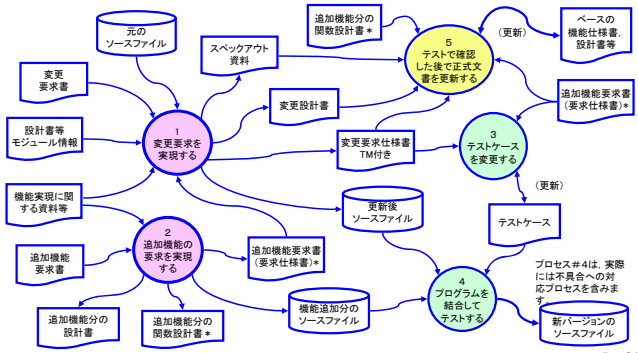
①「実践ソフトウェアエンジニアリング」（ロジャー・プロスマン、日科技連）

②「要求を仕様化する技術・表現する技術」（清水吉男、技術評論社）

③「『派生開発』を成功させるプロセス改善の技術と極意」（清水吉男、技術評論社）

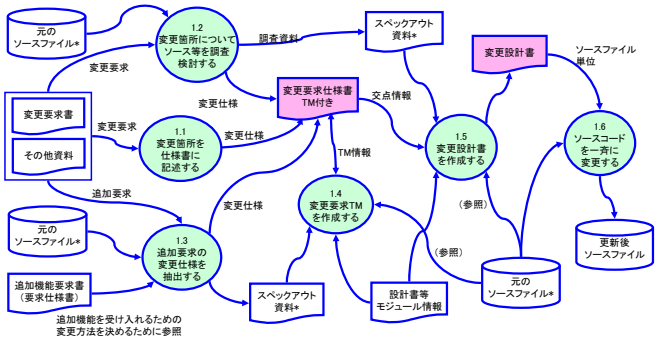
派生開発には2種類のプロセスを使う

- 「追加」と「変更」では「要求」が違う → プロセスも異なるべき
- それぞれ「①変更用PFD」「②追加用PFD」として下位のPFDで展開



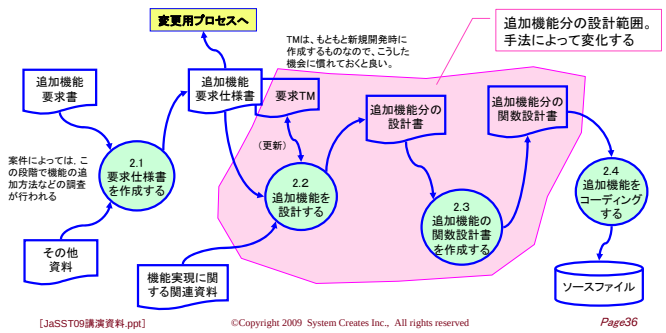
変更を扱うプロセス

- 「設計する」というプロセスは存在しない



機能追加を扱うプロセス

- 設計手法の違いによっていくつかのプロセスは変化する
  - 追加がないときは、このプロセスは存在しない
  - 「1.3」のプロセスが終了すれば「2.2」以降に取り掛かる



機能追加における保守性の品質要求

- 今回追加される機能に対する品質要求は、新規開発時の品質要求と基本的には同じ
  - パフォーマンスなどの機能を補足する品質要求
  - 交換性や保守性などの作り方に関する品質要求

要求	QUA20	[保守性]追加機能の保守性を既存の保守性のレベルを継承して欲しい	
	理由	システム全体の保守性のレベルを落としたくない	
	□□□	QUA.02-1	モジュールの複雑度は20以下を原則とする
	□□□	QUA.02-2	"手順的凝集度"以下になる場合は事前に承認をとる
	□□□	QUA.02-3	処理と管理は明確に分離し、関数の呼び出しの深さは5以内とする
	□□□	QUA.02-4	タスク間でアクセスし合うグルーバル・データを作らない．グローバル・データとなることが避けられない時は、事前に承認をとる． 【説明】割り込み処理との間で共有するケースはこの制限外となる．

変更にも保守性の品質要求を課す

- 機能追加分の保守性の要求とは別に用意する
- 変更における保守性の要求は「劣化」を防止するのが目的

要求	QUA20	[保守性]変更の際に現状の保守性のレベルを悪化させない	
	理由	もともと「保守性」を考慮して設計されているから	
	□□□	QUA20-01	処理内容の一部を不用意に複製せず共通機能分割で対応する 【説明】他の呼び出し箇所を変更することになる
	□□□	QUA20-02	変更によって凝集度を悪化させる場合は、モジュールを適切に分割して独立させる（制限付きリファクタリング） 【説明】この場合、新しいモジュールの仕様／設計書が必要
	□□□	QUA20-03	新たな処理を組み込むことで複雑度が20を超えるときはモジュールを適切に分割する（制限付きリファクタリング） 他に方法がない時は、事前にPLの許可を得ること
	□□□	QUA20-04	新たにタスクの外にグローバルデータを出さない 他に方法がない時は、事前にPLの許可を得ること

かんたんな自己紹介

① 1968～1971 失敗の3年間（失敗の記憶しかない）・・・撤退

（1年間のブランク）

② 1972～1977 汎用機（オフコンを含む）の期間
- 1973年・・・「USDM」を考案

③ 1977～1987 組み込みシステムの期間
- 1978年・・・「XDDP」の原型考案

現役時代

④ 1987～1995 システムコンサルティングの期間

⑤ 1995～今日 プロセス改善のコンサルティングへ

- 現役時代の「22年間」・・・仕様トラブル、納期遅れ なし

プロセス改善のコンサルティングの根拠

- 「営業しない」という基本方針は72年以降今でも達成中