派生開発の現場で秩序あるリファクタリングを実施する方法

主査　　：飯泉　紀子　（株式会社日立ハイテクノロジーズ）
副主査　　：足立　久美　（株式会社デンソー）
アドバイザー：清水　吉男　（株式会社システムクリエイツ）
リーダー　　：菅原　誠一　（株式会社堀場エステック）
研究員　　：加川　俊哉　（株式会社アドバンテスト）
　　弦巻　智也　（テクニカルジャパン株式会社）
　　畑山　誠司　（アンリツエンジニアリング株式会社）
　　森江　里美　（アンリツエンジニアリング株式会社）

研究概要

派生開発では，同じソースコードを何度も変更することにより，ソースコードが劣化していく．劣化が進行すると，手戻りやデグレードなどの温床となり，開発の遅れが引き起こされる．このような問題の対策方法の一つとしてリファクタリングが挙げられる．リファクタリングの実施状況についてアンケート調査をおこなった結果，リファクタリングの実施の判断が個人に任されていたり，機能変更と同時に実施されていたりするなどの原因により，不具合が発生していることがわかった．そこで派生開発のプロセスとして知られる XDDP（eXtreme Derivative Development Process）をリファクタリングに用い，リファクタリング専用の変更プロセス R-XDDP（Refactoring-XDDP）を考案した．この R-XDDP を適用することで，リファクタリングの実施の判断方法やソースコード変更の方法・プロセスをプロジェクトチームとして統制できるようになるので，不具合を未然に防止する効果があると期待できる．

1. はじめに

既存システムへの機能追加や部分的な変更をおこなう派生開発においては，様々な内容の変更要求が頻繁に発生し，迅速な対応が求められる．経験が浅い担当者が変更要求の対応をしたり，納期などの都合により最善ではない方法で変更をおこなったり，不適切なプロセスによる変更要求の対応をすることにより，既存システムにおけるソースコードの構造が劣化し，手戻りやデグレードなどが発生し開発の遅れが引き起こされている．

このような問題に対する解決策として，派生開発に特化した XDDP[1][2]がある．XDDP は，変更要求の成果物として「変更 3 点セット（変更要求仕様書，TM（Traceability Matrix），変更設計書）」を作成する．この変更 3 点セットには全ての変更点が記載されるので，担当者が作成した成果物に対して適切なレビューをすることにより，ソースコードの劣化を最小限に抑えることも可能になる．しかし，一旦劣化させてしまったソースコードを機能変更と同時に改善することは難しい．

すでに劣化してしまった既存システムのソースコードを改善する一つの方法としてはリファクタリングの実施が考えられる．リファクタリングとは，ソフトウェアの既存の機能を保ったまま構造の変更をおこなう作業である．派生開発の現場でのリファクタリングの実施状況をアンケートで調査したところ，リファクタリングの作業の多くが担当者のスキル任せになっており，リファクタリングを実施するためのプロセスが存在しないことなどがわかった．

そこで，XDDP の変更プロセスを利用してリファクタリングを実施する新しいリファクタリング用変更プロセス R-XDDP を提案する．R-XDDP により，個々の担当者の勝手な判断によらないリファクタリングをおこなえることや，段階的に無理なくリファクタリングを適用していくことができる効果がある．リファクタリングに起因する過去の不具合事例を，

R-XDDP を適用することによって解決できるかの考察をおこなった．その結果，不具合を未然に防止する効果が期待できることを確認できた．

以降，第 2 章ではアンケートによる派生開発の現場におけるリファクタリングの実施状況の分析と課題の整理，および先行研究の調査結果について述べる．第 3 章では，解決策であるリファクタリング専用の変更プロセス R-XDDP の定義と，R-XDDP の現場への導入方法を説明する．第 4 章では，R-XDDP の効果について述べる．第 5 章では，本研究のまとめと今後の課題を示す．

2. 現状分析

2.1 リファクタリングに対する意識調査

派生開発の現場におけるリファクタリングに対する意識および実施状況を調査するため，研究員の現場にてプロジェクトマネージャーを含む開発関係者 92 名を対象にアンケートを実施した．

まず，ソースコードの劣化に関する質問では，78%がソースコードの劣化を意識しており，それによって開発が遅れた経験を半数以上が持っていることがわかった．ソースコードの劣化の要因は，元のソースコードの作りであると考える回答が 36%と最も多かった（図 1）．

元ソースコードの作りの悪さを改善する方法には，リファクタリングがある.リファクタリングについての質問では，開発中にソースコードを改善したいと感じる人が 90%とリファクタリングの意欲は高く，実際にリファクタリングをおこなったことがあると回答した人は 57 人で,全体の 62%であった．しかし，その内の 80%が，リファクタリングによって不具合が発生した経験があると回答しており，十分にリファクタリングの効果が出ていないことがわかった．

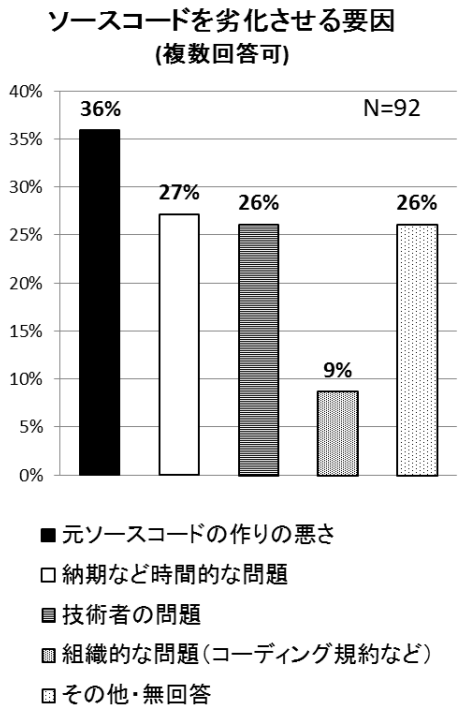

図 1 ソースコードを劣化させる要因

また，アンケートで報告された不具合事例 18 件（付録 A）を 4 つのパターンに分類した．これを表 1 に示す．

表 1 アンケートで報告された不具合の分類

No.	不具合のパターン	件数
1	関数内の機能を不要だと判断し，削除した結果，他プロジェクトでデグレードを起こした	3 件
2	関数内の機能を不要だと判断し，削除した結果，自プロジェクトでデグレードを起こした	4 件
3	機能を変えずに構造を変えたつもりが機能も変わった	6 件
4	リファクタリングではない変更によって，機能が変わった 例：機能の改善や性能の改善，不具合修正など	5 件

リファクタリングは既存機能を保ったまま構造の変更をおこなうが，No.1，2 のパターンでは機能の削除をおこなっている．また，No.4 はリファクタリングではなく，既存機能の改善や不具合修正などをおこなっている．以上のことから，リファクタリングとして実施する行為の内容に，不具合の原因があるのではないかと考えた．

2.2 リファクタリングの実態調査

そこで我々は，派生開発の現場でどのようにリファクタリングをおこなっているかの実態調査のため，先のアンケートと同じ 92 名を対象にアンケートを実施した．アンケートの質問と，回答からわかったことを表 2 に示す．

表 2 リファクタリングの実態調査アンケートの結果（一部抜粋）

No.	質問	回答からわかったこと
1	開発現場にソースコード改善のためのプロセスが存在するか？	74%が，存在しないと回答
2	ソースコードの改善の実施をどのように判断しているか？	改善にかかる工数や効果，影響範囲，現在の工程など，個人によって様々な要因を判断材料に用いている．（付録 B）
3	ソースコードの改善対象は何処だったか？（質問対象：実際に改善した事例があると回答した 57 人）	機能変更の変更箇所が 63%と最も多かった．しかし 46%は変更箇所の調査中に発見した箇所を対象としており，32%の人は機能変更とは無関係に以前から気になっていた箇所を対象とするなど，思い思いの箇所をリファクタリングしていた．（付録 C「付図 C-1」）
4	ソースコードの改善は，どのような形態で実施したか？（質問対象：実際に改善した事例があると回答した 57 人）	改善の実施事例の内 65%が機能変更と一緒に改善をおこなっていた．これは機能追加とリファクタリングの作業を区別し，別々に実施すべきとするリファクタリングの原則に反している[3]．機能変更と改善を分けて実施している事例は 33%に留まっており，改善のみをおこなった事例は 2%に満たない．（付録 C「付図 C-2」）

以上のことから，派生開発の現場ではリファクタリング用プロセスが存在しないことが多く，リファクタリングの実施判断や対象の決定，実施の形態の選択などが担当者任せになっていることがわかった．我々は，このような個人任せのリファクタリングの実施が不具合の原因であると判断し，派生開発の現場で運用可能なリファクタリング用プロセスによって，この問題を解決することができると考えた．

2.3 先行研究

リファクタリング用のプロセスを調査するにあたって，まず『リファクタリング』[3]を参照した．本書は様々なリファクタリングパターンの記載が中心となっている．リファクタリングの実施方法については，事前条件として JUnit テストフレームワークを用いたテストの構築や，リファクタリングを実施している最中の手順や注意に留まっており，一連のプロセスとしての実施方法は記載されていなかった．

次にリファクタリングを実施するためのプロセスを調査したところ，派生開発に特化したプロセスである XDDP で定義されている「小さなリファクタリング」[1]を発見した．

設計もなしにソースコードの変更をおこなうような機能変更に対して，XDDP では最低限のマナーとして，変更 3 点セットを中心に変更内容の定義とレビューをおこなう．変更 3 点セットは，機能を実現するための変更仕様，変更箇所，変更方法を定義した３種類のドキュメントで構成されており,各ドキュメントの内容は常にリンクしている必要がある．変更 3 点セットの作成およびレビューを適切におこなうことで，機能変更にともなうソースコードの劣化を最小限に抑えることや，機能変更と一緒にリファクタリングが実施されることを防ぐことが可能になる．

小さなリファクタリングは，XDDP を用いた機能変更と同時に実施することができる限定的なリファクタリングである．機能変更と同時にリファクタリングをおこなう最低限のマナーとして，リファクタリングパターンが当該関数内の凝集度[4]の改善のみに限定され

ている.図2は，既存関数funcX()内の処理P1，P2，P3を新規の処理関数funcA()，funcB()，funcC()に分割する場合，小さなリファクタリングで実施できるパターンと，実施できないパターンを図示したものである．処理P1～P3がfuncX()の目的とする機能に必要な処理であれば，funcX()を処理関数 funcA()～funcC()を呼び出す管理関数に変更するだけで良く，小さなリファクタリングを実施することができる．しかし,funcX()が手順的凝集度[4]の関数であり，処理P1～P3がそれぞれ異なる機能の処理であった場合，funcX()の呼び元である funcY()に遡って，funcX()ではなく funcA()～funcC()を呼び出すように変更する必要がある．当該関数以外の変更が発生するため，このパターンを小さなリファクタリングで実施することは不可能である．funcY()の変更目的はリファクタリングであり,機能変更を目的としている変更３点セットでは表現することができない．仮にfuncY()の変更を許容すると，変更３点セットのリンクの範囲を越えるため，funcY()の変更やそれに伴う影響などをレビューで発見することは非常に困難であり，担当者任せのリファクタリングが実施される危険性が高い．

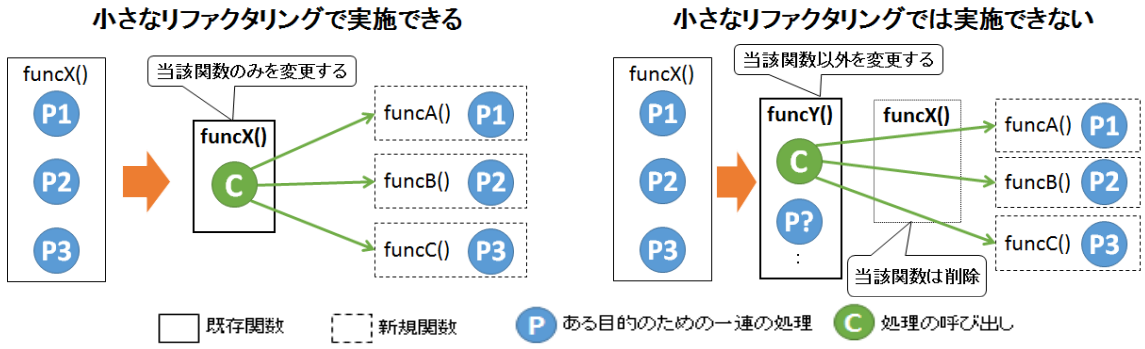

図２ 小さなリファクタリングで実施できるパターンと実施できないパターン

以上のように，小さなリファクタリングは，機能の変更が生じている当該関数の凝集度の悪化を食い止めることに限定されており，既存構造の変更が制限されている．そのため，それ以外の既存構造の変更を伴うリファクタリングを実施できないという課題が残る．

3. 解決策

第２章で現状の分析をした結果，派生開発の現場の多くはリファクタリング用のプロセスを持っておらず，リファクタリングの実施判断などが，担当者任せになっていることが分かった．そこで，それらの問題を解決するために，XDDPの変更プロセスを利用できるのではないかと考えた．

XDDP の変更プロセスでは，変更要求仕様書，TM，変更設計書という視点の違う３つの成果物を作成する．最初に，変更要求仕様書は，変更要求に対する漏れやミスを防ぐために，USDM（Universal Specification Describing Manner）という表記法を使用する．変更の仕様を記述する際には，現在の機能（Before）と変更後の機能（After）を“機能のBefore/After”として表現する．この変更による差分をレビューすることで，仕様の漏れやミスを防ぐことができる．

次に，TM は，すべてのシステム構成要素間の関係を表現する．これにより，変更仕様に対する変更箇所をレビューで確認できるようになる．

変更設計書には，TM 上に現れる変更箇所の具体的な変更方法を記述することで，変更方法の妥当性をレビューで確認できるようになる．

XDDP では視点の違う変更３点セットを作成し，レビューをおこなうことで，担当者の思い込みや勘違いを防止することができる．この仕組みを使用すれば，担当者の個人の判断によらないリファクタリングを実施できるのではないかと考えた．

3.1 リファクタリング時の変更プロセス R-XDDP の定義

アンケートで報告されたリファクタリング時の不具合（表 1）の半数は，リファクタリングの作業中に関数内の機能を不要だと削除した結果デグレードが発生していた．つまり，構造の変更であるリファクタリング作業と，機能の変更である削除の作業を同時におこなうことにより発生していることがわかった．同じくアンケート結果より，現状のリファクタリング作業は，変更と同時に実施されている場合が多いため，XDDP でリファクタリングを実施すると，機能の変更と構造の変更を混同して実施される可能性が高い．

そこで，構造の変更だけを取り扱う XDDP を R-XDDP として新たに定義し，機能の変更を取り扱う従来の XDDP と区別する．これにより，構造の変更と機能の変更を混同せずに実施することができると考える．この R-XDDP による構造の変更プロセスと XDDP による機能の変更プロセスを図 3 に示す．

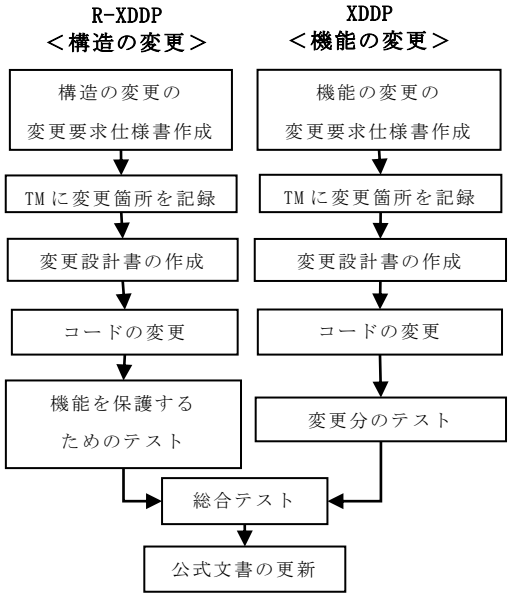

図 3 構造の変更と機能の変更プロセス

3.2 R-XDDP の変更 3 点セットの内容

R-XDDP では，変更要求仕様書，TM，変更設計書の３つの成果物を作成する．図 4 に，小さなリファクタリングでは実施することのできないパターン（図 2）の変更要求仕様書と TM の例を示す．

図 4 において，変更要求仕様書の①要求欄には，リファクタリング対象の構造の特徴をリファクタリングパターンとして記載する．リファクタリングパターンのサンプルを表 3 に示す．②理由欄には，リファクタリングを実施する理由を示す．③変更仕様欄には，リファクタリングパターンに一致する箇所の構造の変更を“構造の Before/After”として記載する．構造の Before には現在の構造を，After には変更後の構造を記載する．図 2 の小さなリファクタリングでは実施できないパターンは，構造の変更を関数の分割 QUA01-01 と呼び元の変更 QUA01-02 に分けて表現することで，実施することが可能になる．

④TM 欄および変更設計書は通常の XDDP と同様に作成する．

④TM 欄

		変更要求・変更仕様		file1.c	file2.c	file3.c
要求	QUA01	手順的凝集度になっている関数を，内包する機能ごとに独立させる		← ①要求欄		
	理由	不必要に処理が詰め込まれているため，今後，再利用される可能性のある関数を独立させておく		← ②理由欄		
	補足	機能ごとの分割をおこなう関数(QUA02-01 の対象)と，その呼び元の関数(QUA02-02 の対象)には同じ記号を付記し，繋がりを明記すること(#1,#2・・・で表現する)				
③変更仕様欄	□□□	QUA01-01	手順的凝集度の関数を内包する機能ごとに独立させる	(#1)funcX	(#2)funcZ	
	□□□	QUA01-02	当該関数を呼び出している上位関数では，独立させた関数の中で必要な関数を呼び出すように変更する	(#1)funcY (#2)funcB		(#2)funcE

図 4 R-XDDP の変更要求仕様書と TM

3.3 リファクタリングパターンによる効果

リファクタリング時には，③変更仕様欄に記載した“構造の Before/After”の Before の構造をプロジェクト内から探し出し,関数名を TM 欄④に記載する．この時，変更方法を考える必要がなく，リファクタリング対象の検索のみに作業が集中できるため効率の良いリファクタリング対象の検索が可能になる．

全てのリファクタリング対象の検索が終了した後で，③変更仕様欄に表現された“構造の Before/After”と，④TM 欄に書き出されたリファクタリング対象の関数についてレビューをおこない，実施の可否を判断する．レビューが完了した後に，変更設計書を作成し，構造の変更を実施するための具体的な方法の検討をおこなう．

最終的に，コードの変更をおこなう際には，リファクタリングパターンを限定することで同じようなパターンの変更を実施することになり，変更方法のパターンを覚え，効率的なコードの変更を期待できる．

表 3 リファクタリングパターンの例

No.	リファクタリングパターン	リファクタリングする理由
1	重複している処理を「共通関数」としてまとめる	変更発生時に修正漏れによる不具合を防ぐため
2	再利用性の高い処理を「共通関数」として抽出する	今後の効率化のために，使えそうな処理は抽出する
3	一時的凝集度の初期化処理の関数に対して，関数の分割をおこなう	機能追加になった際に，エラー処理などにある同じ処理の変更を忘れる可能性が高い為「共通関数」にしておく
4	論理的凝集度の悪い特徴を持つ switch 構造をなくす（呼び出し側の switch 引数はなくなる）	悪い switch 構造により関数が肥大化する傾向がある場合には，switch 構造をなくす必要があるため
5	論理的凝集を持つ関数に対して，関数の分割をおこない，処理を整理する（呼び出し側への影響はない）	内部の構造が複雑になり不具合を起こす可能性が高い場合は，適切に関数の分割をおこなう必要があるため

3.4 派生開発の現場への R-XDDP の導入方法

アンケート結果によると，多くの派生開発の現場ではリファクタリングの実施の際に専用のプロジェクトを立ち上げていない．そのため,何の準備もなく R-XDDP のプロジェクトを立ち上げることは難しいと考えられる．そこで，派生開発の現場への R-XDDP 導入方法として，XDDP の小さなリファクタリングからの導入方法を提案する．

小さなリファクタリングを実施することにより，小さなリファクタリングで実施できるパターンであるか実施できないパターンであるか（図 2）の判断を経験することができる．出来ないパターンの場合には，本来どのようなリファクタリングを実施すべきであるかの検討をおこない，小さなリファクタリングで実施できる範囲でのリファクタリングを実施する．関数の分割を実施する際には，「リアルタイム・システムの構造化分析」の「P-Spec」「C-Spec」の考え方[5]に基づいた関数分割（付録 D）をおこなうことで，規則性のある関数分割が実施できる．

小さなリファクタリングから導入することで，リファクタリング実施時に必要となるリファクタリングパターンの判断や，関数分割方法などのリファクタリングの基礎技術を習得することができる．そのため，R-XDDP の導入を容易にすることが可能だと考える．

R-XDDP の導入の際には，重複している処理を「共通関数」としてまとめる（表 3　No.1）といった，比較的単純なリファクタリングパターンを選択し，短期間で成果を出すことで無理のない R-XDDP のプロジェクトの導入を期待できる．

3.5 R-XDDP による担当者任せでないリファクタリングの実施

R-XDDP では，変更要求仕様書にリファクタリングパターンおよび構造の Before/After

を記載することによって，担当者がどのような構造の変更を実施しようとしているかを確認できる．そして，リファクタリングを実施しようとしている全ての変更箇所と変更方法が TM と変更設計書に現れるため，担当者が想定しているリファクタリングの具体的な変更内容を把握することができる．これらの情報をレビューすることで，構造は多少複雑でも，当面，そこに変更要求が発生しないと予想される箇所のリファクタリングをやめたり，担当者の技量によって，その箇所のリファクタリングを保留したり，どうしても必要な場合には担当者を代えるなどを判断し，対応することが可能になる．

変更設計書に具体的なリファクタリングの方法を記載することで，担当者が検討している変更のイメージを，ソースコードを変更する前に確認でき，不必要な劣化を防ぐ効果につながる．さらに，ソースコードを変更した後で，コードレビューの形で意図したリファクタリングが実施されていることを確認できる．最後に，変更前と変更後のソースコードの差分をとって，変更設計書と変更箇所を突き合わせることで，変更 3 点セット上に現れない，意図しないソースコードの変更や，良かれと思って担当者の判断で変更する行為を防ぐこともできる．

4. 解決策の検証および考察

4.1 R-XDDP のリファクタリング時の不具合に対する効果について

検証期間不足のため，実際の派生開発プロジェクトに R-XDDP を適用できなかった．このため,表 1 で挙げた不具合に対して R-XDDP の適用によって期待できる効果を以下に記述する．

表 1-No.1，2 の不具合に対しては，以下の効果が期待できる．

- 「構造の変更」プロセスと「機能の変更」プロセスを分けることで，「構造の変更」時の“ついで”に機能を削除するケースを防ぐことができる
- 変更前と変更後のソースコードの差分をとって，変更設計書と変更箇所を突き合わせることで，変更 3 点セット上には現れない，意図しないソースコードの変更や，良かれと思って担当者の判断で変更する行為を防ぐことができる

表 1-No.3 の不具合に対しては，“複雑な処理を簡略化”，“仕様を把握しきれていない”など，担当者の理解不足と思われる回答が多かった．この場合，R-XDDP プロセスにおける変更 3 点セットをレビューすることにより，担当者の理解不足を補うことが可能なため，無謀なリファクタリングによる不具合防止に効果がある．R-XDDP プロセスの変更 3 点セットレビューにより，以下の効果を期待できる．

- 変更後の構造が仕様と一致していることを確認できる
- 変更における影響範囲が適切，かつ漏れがないことが確認できる
- 変更内容（プログラミング，コーディング内容）が適切であることが確認できる
- 担当者の技術力不足などのリスクがある場合，リファクタリングを実施する／しないを判定することができる

表 1-No.4 の不具合は，既存の XDDP で変更内容を管理することにより，防止できる．

4.2 効果的なリファクタリングパターンの選択についての考察

表 3 にて今回の研究で対象とするリファクタリングパターンを例示したが，R-XDDP を実際のプロジェクトに適用する際は，プロジェクトに応じて，対象とするリファクタリングパターンを選択する必要がある．『Code Complete』では，「80 対 20 の法則」に従って，利益の 80%をもたらす 20%のリファクタリングに時間を使うべきとしており[6]，この 20%を目的にリファクタリングパターンの選択をおこなうことが望ましい．

今回の研究では，構造化設計の手法で確立している結合度の尺度[4]を利用し，不具合が発生しやすいとされる“論理的凝集度”などに的を絞って，リファクタリングパターンの選択をおこなった．このようにメトリクスに基づいて選択をおこなう際は，メトリクスの

みを基準とするのではなく，重複した処理の共通化や不具合の発生しやすい関数の改善など，今後の開発を視野にいれたリファクタリングの目的を設定し，その目的に適したリファクタリングパターンを選択することが肝要である．

4.3 単体テスト環境のない現場への R-XDDP 適用

マーチンは，単体テストをリファクタリングの実施に欠かせない事前条件と書いており，テストの完全な自動化を推奨している[3]．表 1 の No.3 の不具合事例「機能を変えずに構造を変えたつもりが機能も変わった」は，リファクタリング対象の単体テストがあれば防げる不具合であり，派生開発の現場では単体テストによるソースコードの保護が十分ではないことが推測される．しかし“レガシーコードのジレンマ”[7]として知られているように，単体テストがないソースコードは，単体テストの整備のために変更が必要である場合が多い．単体テストを整備するための構造の変更を，R-XDDP を使って実施することで，単体テストによる保護を実現することが可能であると考えられる．

5. 本研究のまとめ

5.1 結論

派生開発の現場にはリファクタリング用プロセスがないことが多く，担当者任せのリファクタリングによる不具合が発生していた．担当者任せのリファクタリングの対策として，リファクタリング用の変更プロセスである R-XDDP を考案した．R-XDDP では，構造の Before/After を表現した変更 3 点セットを作成し，レビューをおこなう．変更 3 点セットには担当者の想定するリファクタリングの具体的な変更内容が表現されるため，レビューをおこなうことで，担当者任せのリファクタリングを防ぐことが可能になる．

それによって，リファクタリングによる過去の不具合事例を，未然に防止する効果が期待できる．

5.2 今後の課題

今後の課題は，以下の 2 点と考える．

1 点目は，R-XDDP の実プロジェクトへの適用である．今回の研究では，過去の不具合事例に対して期待される効果の考察に留まったため，実際の派生開発のプロジェクトに R-XDDP を適用し，効果を検証していく必要がある．

2 点目は，R-XDDP の実施時に選択できるリファクタリングパターンの種類の拡充である．表 3 で例示したように，今回は凝集度の尺度からリファクタリングパターンを選択した．結合度の尺度に基づくリファクタリングパターンや,現場で R-XDDP を実施することにより新規に創出されるリファクタリングパターンを含め，リファクタリングパターンの種類の拡充を継続する．

最終的には，派生開発の現場で劣化したソースコードを改善し，開発スピードの向上を実現していきたい．

6. 参考文献

[1] 清水 吉男，“「派生開発」を成功させるプロセス改善の技術と極意”，2007
[2] 清水 吉男，“SQiP 第 6 分科会特別講義資料　派生開発プロセス[XDDP]のポイント”，2015
[3] マーチン・ファウラー，“リファクタリング:プログラムの体質改善テクニック”，2000
[4] 清水 吉男，“構造化設計のためのテキスト・ブック（凝集度/結合度)”，1991
[5] 清水 吉男，“構造化仕様書作成マニュアル-構造化設計ガイドブック”，1995
[6] スティーブ･マコネル,“Code Complete 第2版下-完全なプログラミングを目指して”，2005
[7] マイケル・C・フェザーズ，“レガシーコード改善ガイド”，2009