効果的な XP の導入を目的としたプラクティス間の相互作用の分析

川端 光義
アジャイルウェア
kawabata@agileware.jp

阪井 誠
(株)SRA 先端技術研究所
sakai@sra.co.jp

小林 修
(株)SRA
o-kobaya@sra.co.jp

要旨

本論文では，XP(eXtreme Programming)のプラクティス間の相互作用について議論する．XP で定められた 13 のプラクティスのうち，選択的に XP のプラクティスを導入した 2 つのプロジェクトの事例を報告すると共に，開発者からプラクティス間の相互作用をヒアリングし，求められる効果に必要とされるプラクティスを分析した結果を報告する．実際のプロジェクトに XP のすべてのプラクティスを導入することは困難であるが，得られた知見を用いれば，より効果的な組合せを検討することが容易になると考えられる．

1. はじめに

変化に対して機敏に対応できるアジャイルソフトウェア開発のひとつである XP(eXtreme Programming)が注目されている[1]．ウォーターフォールモデルに基づく従来の開発方法論は，仕様を定義した後に凍結や変更の管理によって，大人数での開発を容易にしていた．しかし，仕様を凍結することで，開発の進捗とともに変化するユーザの要求や環境の変化に柔軟に対応することは困難になっている．そこで，仕様の凍結が特に困難と予想される部分にはプロトタイピングの手法を用いるなど，個々のプロジェクトで開発作業に工夫を加える必要があった．また，近年の高速で多機能なコンピュータの普及とそれに伴うユーザの多様化，ユーザインタフェースの多様化によって，ソフトウェアの仕様変更の可能性は高まっている[9]．このような状況の中，ソフトウェア仕様の変化を前提としたアジャイルソフトウェア開発が注目されている．

アジャイルソフトウェア開発の中でも，XP は 4 つの価値とプラクティスを定めており[2]，ソフトウェア開発の指針と方法の理解が容易であることから日本でも普及しつつある[13]．XP では4つの価値（コミュニケーション，シンプル，フィードバック，勇気）を重視している．コミュニケーションを高めることで開発チームが計画に向かって活動しやすくなり，シンプルな設計によって保守が容易で欠陥を少なくすることができる．また，各種テストによるフィードバックによって品質を高めることができる．最後の勇気によって大きな問題に対しても，小手先でない対処が可能になるのである，XP ではこのような価値を重視した開発を行うために，プラクティスを定義している．

このような XP をプロジェクトに導入する際には，適用するプラクティスを決定しなければならない．全てのプラクティスを適用できれば良いが，現実には，顧客などの外部要因やリソースの問題などにより採用できないプラクティスも存在するからである．また，プロジェクトの要件や特性により，一部のプラクティスは適用する必要がないケースや，あまり効果が期待できないケースもある．従来の研究においても，プラクティスを選択的に適用した多くの事例が報告されている[6] [8] [10] [12]．しかし，各プラクティスの効果に関して述べられているが，その組合せによってどのような相互作用が生じるかの議論は十分でなかった．プラクティス間の相互作用が明らかになれば，プロジェクトによって採用すべきプラクティス群や重点を置くべきプラクティス群が明らかになり，より効果的に XP の成果を収められると考えられる．

そこで，本論文では， 2 つのプロジェクトの事例と共に，開発者へのヒアリングにより得られたプラクティス間の相互作用を報告する．また，得られた結果の分析に基づき，選択的なプラクティスの導入方法について考察する．

2. XP (eXtreme Programming)

2.1. XP とプラクティス

XP とプラクティスは深い関係にある．プラクティスなくては XP ではなく，また XP の実践なくてはプラクティスが活かされない．XP は4つの価値を重視するが，これは具体的な実践方法ではなく，プロジェクトを成功に導くための本質的な理念である．この理念を実際のプロジェクトで実践すべき手法として具体化したものがプ

ラクティスであり，プロジェクトの問題点をあらゆる視点から解決するためにまとめられたのが，Kent Beck の「12 のプラクティス」[2]やさらに発展させた Ron Jeffries の「13 のプラクティス」[7]である．本論文では，以下に示す 13 のプラクティスを基に，プラクティス間の相互作用を用いた．

＜13 のプラクティス＞

全員同席
システム開発に携わる人全員が1つのチームとして，同じ場所で開発を進める．

計画ゲーム
開発者と顧客がそれぞれの役割を明確にし，リリース計画およびイテレーション計画を行う．具体的には，ユーザから見た機能(ストーリ)ごとにストーリカードを作成し，さらに開発者の作業(タスク)に分割してタスクカードを作成することで工数と期間を考慮した計画を作成する．

短期リリース
短期間（約1～2ヶ月）のリリースを繰り返し，ユーザのフィードバックを受ける．開発の開始あるいは直前のリリースから次のリリースまでをイテレーションと呼ぶ．

ユーザテスト
リリースするシステムのテストケースはユーザが定義する．

シンプルデザイン
要件を満たすために，必要不可欠な部分だけを実装し，そのコードもシンプルに保つ．YAGNI[†]の原則を常に心掛ける．

ペアプログラミング
2人のプログラマが1台のマシンでプログラミングを行う．

テスト駆動開発
機能を実装する前に，その機能を確認するためのテストから作成する．

リファクタリング
プログラムの動作を変えずに，内部の構造を改善する．

常時結合
計画ゲームで計画したタスク単位の小さな量で結合を行い，常に動作している状態にする．

コードの共同所有
コードは個人所有ではなく，チーム全員の所有とし，誰でもいつでも修正を行うことができる．

コーディング規約
チーム内でコーディングスタイルを統一するために共通のルールを作る．

メタファ
比喩（たとえ）によって，プログラムがどのように動くかの理解をチーム内で共有／共通化する．

適切なペース
無理なオーバーワークをせず，持続可能なペースで作業を行う．

これらのプラクティスのうち，実践できなかったプラクティスがあるとすれば，そのプラクティスによる効果を別の形でカバーしなければいけない[12]．例えば，ペアプログラミングというプラクティスはコードレビューの効果により品質を向上させるが，ペアプログラミングができない環境であれば，コードレビューすべきである．ただし，ひとつのプラクティス，例えば「ペアプログラミング」だけを適用するだけで，十分な品質が確保されるかというとそうではない．コード品質を確保する際に最大の効果を得るにはテスト駆動開発（以下，TDD）などのプラクティスも必要である．常にパスするテストプログラムを用意しなければ，品質を確保することは困難だからである．しかし TDD を適用した場合においても，経験の浅い開発者がテストケースを用意した場合，テストケース漏れが生じる可能性がある．テストケースの作成においてもレビューが必要であり，ペアプログラミングはそれをカバーするのである．このようにプラクティスは相互に作用することで，様々な効果をより確実なものとしている．

2.2. XP の導入事例

XP を導入する際にすべてのプラクティスを適用することは困難であり，どのようにプラクティスを選択し適用するかは重要な問題である． これまでにも XP の導入戦略や妥協点についての事例が報告されている．文献[6][12]では XP を導入する目的で，プラクティスの適用方法を工夫しているが，文献[10]では段階的にプラクティスを適用している．また，[8]では，プラクティスの効果と，適用が困難なプラクティスについて述べられている．

文献[6]では，ソフトウェア開発プロセスが厳密に定義された大きな組織における XP の導入について述べられている．この事例は，安全性が重視される大規模システムのうち，ひとつのサブシステムのみを XP で開発する試みであり，原則としてすべてのプラクティスの

[†]YAGNI = You aren't going to need it.（今必要のあることだけをやれ）

実践を目指しながらも，開発するシステムの性格や組織の方針を考慮して，計画ゲームへのユースケース図の利用や(開発標準に準拠するためでなく)必要最低限のドキュメント作成といった工夫を行っている．

文献[12]では少人数（2名）による科学研究プロジェクトでの XP の適用について述べられている．研究プロジェクトへの XP の適用を困難にする要因を述べ，その要因を克服するために，計画ゲームやシンプルデザインを顧客を役割と考えるなど可能な方法に読替えて実施したこと，ペアプログラミングとコードの共同所有が生産性や可読性を向上させたことについて述べられている．

文献[10]では，いったん混乱に陥った開発プロジェクトが，混乱から回復し安定した保守フェーズに移行する過程での，XP の適用について述べられている．この例では，プロジェクトの建て直しに途中から参加したメンバーが，システムの最初のリリース以後，段階的に XP のプラクティスを導入することで，保守体制を整えていく過程が述べられている．具体的には，環境整備を実施した後，以下のプラクティスを導入している．

・常時結合
・(穏やかな)リファクタリング
・シンプルデザイン
・コーディング規約
・スタンドアップミーティング

最後にプロジェクトが落ち着いた段階で，その他のプラクティスの導入を検討している．

文献[8]では，既存システムを Java ベースの新技術を利用して新たに書き直すプロジェクトにおいて，XP を適用した事例が述べられている．ここでは，ペアプログラミングが最も効果的であるとされ，さらに，ペアプログラミングが，シンプルデザイン・テスト駆動開発・リファクタリングの適用に効果があること，特に，テスト駆動開発の適用には，ペアプログラミングの実践が重要であると述べられている．また，すべてのプラクティスの実践状況や効果について述べられている．このほか，困難だった点として，外部チームとのコミュニケーション，顧客の同席，テスト担当者の位置づけの明確化があげられている．

このように XP を導入する際に，どのようにプラクティスを適用するかは，XP の導入効果に大きな影響を与えることから，事例の詳細な分析が必要である．特にプラクティスは独立しているわけではないので，プラクティス間の依存関係や相乗効果といった相互作用を考慮することが重要である．しかし，文献[6]と文献[12]では，各プラクティスは，それぞれ単独に扱われている．文献[10]では，プラクティスの段階的導入の一例を示しているが，プラクティス間の相互作用を明確にしたものではなく，異なる状況での導入の手がかりとはなりにくい．文献[8]では，ペアプログラミングが他のプラクティスの適用に効果があることが述べられているものの，述べられているプラクティス間の相互作用は一部である．このように，XP を導入する際に，どのようにプラクティスを選択すべきかを検討する際に役立つような分析は行われていなかった．

3. 事例に基づくプラクティス間の関係

3.1. 事例1：オンラインショッピングサイト構築

3.1.1. プロジェクト概要

今回，XP を導入したプロジェクトは Nissen On-line の稼動分析器システムである(表 1)．このシステムは，ニッセンのオンラインショッピングサイトに訪れた顧客が，いつ，どこのサイトから，どのような商品を，どのくらい購入したかを分析・評価する機能を提供する．

事例 1 で適用したプラクティスを表 2 に示す．XP の開発が始まった時には，要件の洗い出しがほぼ終了しており，それ以降の工程から XP を導入した開発に入った．

3.1.2. プロジェクトの経緯

第 1 イテレーション～第 2 イテレーション

第 1 イテレーションでは，対象のストーリが簡単なものだったので，明確な設計を全く行わず TDD で開発を進められた．しかし，TDD に慣れていないことや目先のスケジュールを優先したことから，リファクタリングが不十分であった．結果的にコードが複雑になり，第 2 イテレーションにそのまま組み込めないプログラムとなってしまった．第 2 イテレーションで大規模なリファクタリングを行うことになったので，第 1 イテレーションのコードは大部分が無駄になってしまった．このような経緯から，たとえ簡単なストーリであってもそのイテレーションで実装すべき機能についてのモデリングを行うことは必要であったと思われる．もし，CRC セッション[4]などのモデリングを行っていれば，大規模なリファクタリングにはならなかったはずである．また，リファクタリングをしっかり実践していれば，コードは再利用可能な部品となり，無駄にならなかったと考えられる．

ペアプログラミングは常に実践していたが，当初は適切なペースは守れなかった．スケジュールの遅延により，適切なペースではない日が生じた．それが1週間続いたところ，ペアの2名ともが遅刻気味となり，午前

表 1 事例 1:プロジェクトの概要

アイテム	データ
開発体制	マネージャ，開発者（3名），ユーザ(2名)
開発期間	5ヶ月（XP 開発期間3ヶ月）
言語	C#, ストアドプロシージャ
リリース回数	3
総イテレーション回数	4

表 2 事例 1:適用したプラクティス

プラクティス	適用度
全員同席	△
計画ゲーム	△
短期リリース	○
ユーザテスト	○
シンプルデザイン	○
ペアプログラミング	○
テスト駆動開発	○
リファクタリング	○
常時結合	○
コードの共同所有	○
コーディング規約	○
メタファ	×
適切なペース	○

○:全面的に適用，△:部分的に適用，×:適用せず

中は頭が働かないと訴えるようになった．残業してもスケジュールの改善が見られなかったため，その後は適切なペース（週40時間）をしっかり適用した．適用後，生産性の向上が見られたのである．

第 3 イテレーション～第 4 イテレーション

全員でアジャイルな設計を行った後は，比較的スムーズに開発が進んだ．開発者が 3 名だったため，2 名がペアプログラミングを行い，1 名はソロプログラミングであった．第 3 イテレーションに入って，バグや仕様の認識違いによる手戻りが，ソロプログラミングのモジュールで顕著になってきた．その後，ソロプログラミングのストーリであっても少し複雑な機能に関しては，トリプレットプログラミング[11]を行った．この結果，認識違いによる手戻りはほとんどなくなった．

ペアプログラミング，テスト駆動開発，そしてリファクタリングの3つが相乗効果として，シンプルなコーディングに大きく貢献した．コードのシンプルさにより第 3，第 4 イテレーションのストーリを見積りより早く実装し，スケジュールの遅れを取り戻すことができた．

リファクタリング技術に疎かった開発者 A は，リファクタリングの得意な開発者 B とペアプログラミングすることにより，開発者 A もリファクタリング技術を身に付けながら開発を進めることができた．反面，開発者 B は時にリファクタリングしすぎることがあった．これは YAGNI の原則を越えた，必要のないかもしれない要求への対応を行ったのである．この時，開発者 A がリファクタリングのバランスに対して，重要な役割を果たしていた．また，テスト駆動開発においてもペアプログラミングは有効であった．テスト駆動開発は，失敗するかもしれないテストだけを作成することで品質を高める[3]．しかし，ソロプログラミングの場合は，テストケースに漏れがあったり，無駄なテストケースを作成していたりすることが多くなる．ペアプログラミングによりテストケースの品質に対する検証が常に行われていた．さらにペアプログラミングは自然とコードの統一を図ることになり，ドキュメントを特に用意する必要なくコーディング規約が達成できた．

テスト駆動開発は常時結合を可能にした．テストケースが予め準備されていることによって，結合時に問題が即座に検出され，すぐに原因が特定できた．また，リファクタリングによって条件文をポリモーフィズムに置き換えることができたので，事前条件や例外などのテストケースを削減してテスト駆動開発のコストを抑えることができた．

メンバーの感想

- 大規模なリファクタリングが発生したことから，第1イテレーションの前に迅速なモデリングを行うべきであった(XP では，CRC セッションが紹介されているが，プラクティスほどの強制力はない)．
- 新たに追加すべきプラクティスとして「全員設計(Modeling Together)」を行うことが必要と感じた．
- ペアプログラミング，テスト駆動開発，リファクタリング，常時結合，コーディング規約の結びつきが強いことが判った．
- ペアプログラミングを常に実践する場合には，適切なペースは必須である．
- スタンドアップミーティング[2]は効果的であった．

(プロジェクトを通して毎日スタンドアップミーティング[2]を行い，壁に貼ったストーリカード，タスクカードで日々状況を確認したので，プロジェクト管理の面では特にドキュメントを作成せずに進捗や問題点が把握できた)．

3.2. 事例2：見積りパッケージソフトのカスタマイズ

3.2.1. プロジェクトの概要

事例 2 は見積りパッケージソフトのカスタマイズプロジェクトである(表 3). システム全体としては大規模なものであるが，カスタマイズ要件は全体の5分の1程度のものであった．適用したプラクティスを表4に示す．

3.2.2. プロジェクトの経緯

第 1 イテレーション

カスタマイズ要件の見積りから作業は始められた．カスタマイズの見積りは既存のプログラムによって，大きく異なるので，まず，プログラムを解析し，既存のプログラムがオブジェクト指向と大きく異なる作りであることが判明した．大きい機能のもので1メソッド2000行もあり，カスタマイズ機能を盛り込むことは，既存の動作に影響することが確実と思われた．このため，リファクタリングから作業を始めることになった．非常に多くのリファクタリングを行うことで，担当以外のソースコードへの影響が大きかった．ここでコードの共同所有が必須のプラクティスとなった．共同所有をしていなければ，リファクタリングする勇気も生まれて来なくなってしまうからである．この時，コードの共同所有のルールとして，1日1回以上同期を取ることを原則とした．3日以上同期を取らなければ，他の人と競合が多発してしまい，解決する時間の方が大きくなってしまうと考えられたからである．また，頻繁に共有するクラスに関しては，修正する直前に同期を取っておくことで競合は最小限に抑えられた．

第2イテレーション～第3イテレーション

第1イテレーションでリファクタリングを行った効果があった．第2，第3イテレーションでは，第1イテレーションで作成した機能に追加，変更仕様があり，リリースによるエンドユーザからのフィードバックもあった．第2，第3イテレーションのタスクで第1イテレーションに関係したタスク数と実績時間を表5に示す．ここで，第2・3イテレーションの総タスク数の半数以上が第1イテレーションに関係しているものの，その実績時間は，40%以下であった．リファクタリングされているソースコードに関係したタスクの方が実績時間が少なく，第1イテレーションでリファクタリングに要した時間は 22H であることから，リファクタリングにより生産性が向上したと考えられる．

表 3 事例 2：プロジェクトの概要

アイテム	データ
開発体制	マネージャ，仕様担当（2名），開発者（4名），ユーザ（1名）
開発期間	3ヶ月
言語	Java, Struts フレームワーク
リリース回数	2
総イテレーション回数	3

表 4 事例 2：適用したプラクティス

プラクティス	適用度
全員同席	△
計画ゲーム	○
短期リリース	○
ユーザテスト	○
シンプルデザイン	×
ペアプログラミング	×
テスト駆動開発	×
リファクタリング	△
常時結合	○
コードの共同所有	○
コーディング規約	×
メタファ	×
適切なペース	×

○：全面的に適用，△：部分的に適用，×：適用せず

表 5 事例 2：第 1 イテレーションと関係するタスク数及び実績時間

第1イテレーションに関係したタスク数／総タスク数	98／192
第1イテレーションに関係したタスクの実績時間／総実績時間	149H／374H

※ ドキュメント作成タスク含まず

メンバーの感想

●リファクタリングには，コードの共同所有が必須である．

(コードの共同所有ではバージョン管理ツールにCVS を利用した．CVS の特徴はソースファイルをロックすることなく，誰もが同じソースを同時に修正できる点である．XP プロジェクトでリファクタリングを頻繁に行っている場合は，CVS が適していた．)

●リファクタリングとして「メソッド抽出」[5]を多く行ったことが，生産性の向上に繋がった．

(第1イテレーションでリファクタリングを大規模に行ったが，それでも全体からすると5割も行っていない．リファクタリング対象のものが余りにも多く，場合によってはフレームワークまでを壊してしまうからである．リファクタリング実施時は，第2・3イテレーションの要件もある程度明確になっていたので，関係すると思われるものに絞ってリファクタリングを行った．)

3.3. プラクティスの相互作用とその効果

開発事例1，2の開発者に対して3．2節の内容をヒアリングした結果，XP の導入により以下に示す 5 つの効果があること，プラクティス間には相互作業があり，強い依存関係あるいは弱い依存関係のあることがわかった．そこで，相互作用の表記方法を定めて再度ヒアリングを実施して図1～5が得られた．これらはプラクティス間の相互作用を，それにより得られる効果ごとに分類したものである．各プラクティスを四角で表し，プラクティス間の関連を以下の表記で示している．

＜プラクティス間の関連＞

直線矢印 ・・・ 強い関連

点線矢印 ・・・ 弱いあるいは部分的な関連

矢印の向き ・ 矢印先のプラクティスは矢印元のプラクティスに依存する

(1) コミュニケーション効果によるドキュメント作成工数削減(図 1)

開発者の全員同席，計画ゲーム，ペアプログラミング，ユーザテスト，テスト駆動開発の実践により，事例 1 における開発者3名全員が，仕様書や設計書などのドキュメント類をほとんど作成せずに仕様，設計知識を共有できていた．

情報をメンバー間で共有するため，ペアプログラミングのペアは頻繁にローテーションを行う．そのため，ペアプログラミングを行うには，全員同席が必要となる．

ユーザテストは，少なからずドキュメントが作成されることになるが，具体的なテストケースがそのまま仕様の詳細を意図するため，仕様書としてのドキュメントは簡略化できるのである．

(2) 生産性向上（リファクタリングによる変更コスト削減）(図 2)

シンプルデザイン，ペアプログラミング，テスト駆動開発，リファクタリング，コードの共同所有の実践により，コーディングレベルのフィードバックを最大にすることで，手戻りを最小限にすることができ，部品の再利用が進み，結果，生産性が向上した．

ペアプログラミングのスピードを維持するためには，適切なペースが必須であり，計画をドライブできなければならない．また，シンプルデザインを維持するために，時には大きなリファクタリングが必要な場合もある．その時は，計画ゲームに影響する．

XP はシンプルデザインの方針で，システムを成長させていくが，テスト駆動開発により無駄な作り込みをせず，リファクタリングによりコードをシンプルに保つことができる．

リファクタリングは生産性を一時的に落とす場合もあるが，長い目で見れば部品の再利用が進み，生産性を向上させることになる．また，イテレーション開発では既存のコードに修正，追加をすることが多く，その際，

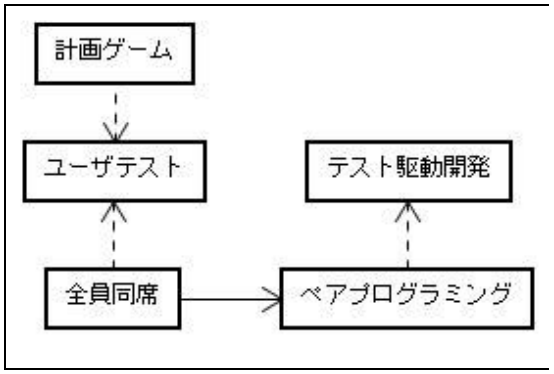

図 1 コミュニケーション効果のプラクティス群

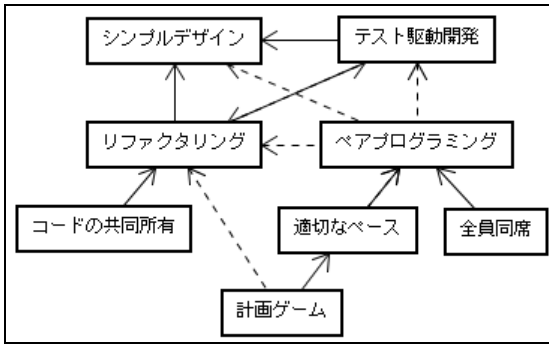

図 2 生産性向上のプラクティス群

複雑なコードでは，コードを理解するために多くの時間を費やす．リファクタリングされたシンプルなコードに対しての修正，追加は少ない時間で実装が進み，生産性を向上させる．

(3)　コード品質向上(図 3)

テスト駆動開発，ペアプログラミング，リファクタリング，常時結合，コードの共同所有の実践により，ユーザ要求を満たすコード品質が確保された．事例1において，第 1 リリース(プログラム本数110本，コード行数7，500行)後，検出された不具合はわずか3件であった．

テスト駆動開発でプログラマがテストケースを作成するが，そのテストケースに漏れや間違いが無いようにペアプログラミング中にレビューすることが必要である．じたがって，テスト駆動開発にはペアプログラミングが寄与している．

(4)　要求品質向上(図 4)

計画ゲーム，短期リリース，ユーザテストの実践により，ユーザの要求をより多く反映する機会が，短期間ごとに繰り返して得られることで，満足度が向上した．

短期リリースの繰り返しを実現するには，単体プログラムの結合テストを実施する時間的余裕はなく，常時結合されている必要がある．また，イテレーションの早い段階からユーザテストを実施することで，フィードバックが早くなり，要求品質に関わる欠陥も早く見つけられる．よって，スムーズにリリースが可能となるのである．

(5)　保守性向上(図 5)

ペアプログラミング，コードの共同所有，コーディング規約，テスト駆動開発，リファクタリングの実践により，保守のしやすい統一されたコードとなった．

リファクタリングによって，シンプルな読みやすいコードとなるが，作る人によって名前の付け方が違うと，誤解を招くため，コーディング規約が必要となる．また，コードの共同所有は，他の人のコードを修正する前提で作用するため，コーディング規約は必須である．

ペアプログラミングを行うと，自然とパートナーに合わせたコーディングをするようになる．そして，ペアがローテーションされることで，全員のコーディングは統一されるようになる．逆にコーディング規約が決まっていると，ドライバとナビゲータの間でコードの書き方について議論する必要がなくなり，スムーズにペアプログラミングが進められる．

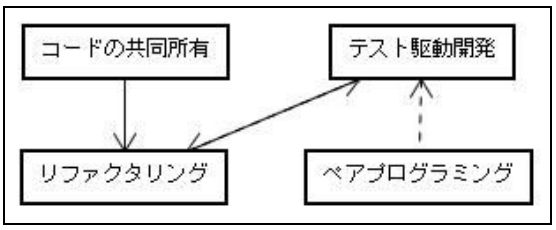

図 3 コード品質向上のプラクティス群

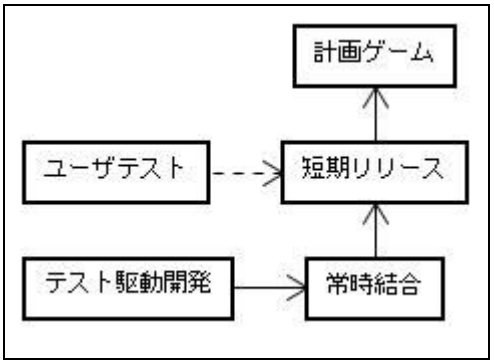

図 4 要求品質向上のプラクティス群

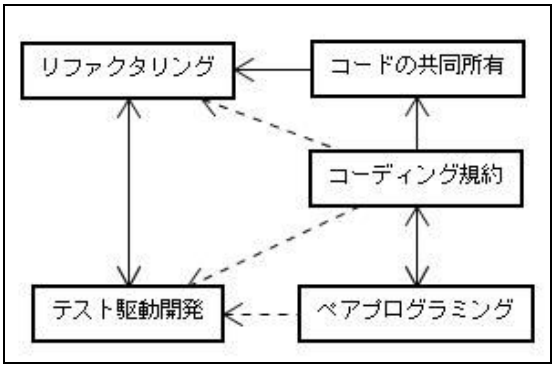

図 5 保守性向上のプラクティス群

4. 考察

3.3.で述べたプラクティス間の相互作用を効果と依存関係を表 6 に示す．”○”は各プラクティスが関係する効果を示し，効果欄にその数を示す．各矢印欄の記号は各プラクティスが依存するプラクティスを示しており，左向けの矢印欄にあるプラクティスは各プラクティスが依存するプラクティスであり，実線を1，点線を 0.5 とした合計値を依存値としている．同様に右向けの矢印欄にあるプラクティスは各プラクティスが依存されるプラクティスであり，合計値を寄与値としている (2 つの開発事例で導入できなかったメタファについては，相互作用に関する知見が得られなかったため，表には含んでいない) ．

この表を用いれば，期待する効果に必要なプラクティスがわかるだけでなく，各プラクティスの特性を知るこ

表 6 プラクティスの相互作用

	効果	コミュニケーション	生産性	コード品質	要求品質	保守性	<依存	>寄与	←→	←		→			<…			…>		
短期リリース(SR)	1				○		1.5	1		CI		PG			CT					
常時結合(CI)	1				○		1	1		TDD		SR								
ユーザテスト(CT)	2	○			○		1	0.5							ST		PG	SR		
計画ゲーム(PG)	3	○	○		○		1	1.5		SR		SP						CT		
シンプルデザイン(SD)	1		○				2.5	0		TDD	R				PP					
適切なペース(SP)	1		○				1	1		PG		PP								
コーディング規約(CS)	1					○	1	2	PP			CC						R		TDD
全員同席(ST)	2	○	○				0	1.5				PP						CT		
リファクタリング(R)	3		○	○		○	3.5	2	TDD	CC		SD			CS	PP	PG			
コードの共同所有(CC)	3		○	○		○	2	1		CS	TDD	R								
テスト駆動開発(TDD)	4	○	○	○		○	1.5	4	R			SD	CC	CI	PP					
ペアプログラミング(PP)	4	○	○	○		○	3	2.5	CS	SP	ST							SD	R	TDD

とができる. テスト駆動開発については, プラクティスの依存値が1.5と比較的少ないが, 効果は4と大きいので単体や少ないプラクティスの組合せでの導入が容易で効果の大きいプラクティスである. リファクタリング, ペアプログラミングは, 依存値が共に 3 と大きいので単体での導入は難しいと言える. しかし, リファクタリングは寄与値, 効果の数値も大きいので, 導入ができればより大きい効果を生み出すと考えられる. また, 効果が 3 で寄与値が 1 のコードの共同所有と効果が 4 で寄与値が 2. 5のペアプログラミングは, 効果が大きい割に寄与値が少ない. これらは, 他のプラクティスとは独立した様々なメリットを含んでいると考えられる.

反面, 表 6 に示す効果はヒアリングで得られた 5 つの効果の分類との関係を数値化したものであり注意が必要である. 効果の値の高いプラクティスを導入することは効率的ではあるものの, 必ずしも XP の目指す 4 つの価値のすべてを実現する際の効果を示したものではないからである. 特に表 6 の要求品質の向上と関係するプラクティスに注目すると, 他の効果に関係するプラクティスと性質が異なっているのがわかる. 表の結果では要求品質を向上させようとすると, 他の効果が得られないので, 効率はそれほど高くはない. しかし, ビジネスとして要求品質の効果が重要視されるプロジェクトであれば, 関連するプラクティス群は重要であるので, 優先して適用すべきである.

5. まとめ

2 つの XP のプラクティスを適用したプロジェクトの事例と共に, 開発者へのヒアリングに基づいてプラクティスの相互作用を整理した, また, 得られた結果を基に選択的なプラクティスの適用効果について考察した. これらの知見を用いることで, 効率的なプラクティスの選択が容易になると考えられる.

従来 XP を導入する際には, 各プラクティスのメリット・デメリットなどの知識だけでプラクティスを適用せざるを得なかった. その様な場合, 実践中に相互作用の問題が生じ, 効果があまり得られないことや失敗に終わることも考えられる. 経験によってモデル化されたプラクティス間の相互作用が明らかになったことで, 適用できないプラクティスがある場合に, その影響の予測や, 予め対策を打っておくことも容易になり, より安全に XP を導入できる可能性がある.

例えば, 生産性向上が最優先の目的のプロジェクトで XP の導入を試みる場合においても, 効果的に XP を導入できると考えられる. 表 6 の生産性の欄に示される 8 つのプラクティスを検討するほか, そのうち実践可能なプラクティスが依存するプラクティスも検討し, 実践不可能なプラクティスには代替案を検討する. このように表 6 を用いることで, 目的や制約を考慮した上で, 効果的なプラクティスの選択と代替案の検討が容易にな

る.

しかし，今回報告したプロジェクトで全てのプラクティスを様々な状況で実践できたわけではなく，必ずしも相互作用による効果を全て経験したとは言えない．またプロジェクトの特性はより様々なケースが存在するので，この知見が必ずしも一般的な解とは言えない．今後，様々なプロジェクトにおいてこの知見に基づいたXPを実践することにより，より正確な相互作用や，選択的に導入する際に必要なプラクティスが見出されると考えられる．今後はそれらの経験をケーススタディとして蓄積し，XP プラクティス導入パターンを構築することが今後の目標である．

参考文献

[1] Kent Beck, B. Boehm, “Agility through discipline: A debate,” IEEE Computer, pp.44-46, June, 2003.

[2] Kent Beck, “eXtreme Programming Explained: Embrace Change,” Addison-Wesley, 1999.

[3] Kent Beck, “テスト駆動開発入門”, ピアソン・エデュケーション, pp.190, 2003.

[4] David Bellin, Susan Suchman Simone, “実践 CRC カード ロールプレイとブレーンストーミングによる大規模システム開発手法”, ピアソン・エデュケーション, 2002.

[5] Martin Fowler, “リファクタリング プログラミングの体質改善テクニック”, ピアソン・エデュケーション, 2000.

[6] James Grenning, "Launching eXtreme Programming at a Process-Intensive Company", IEEE Software, Vol.18, No.6, pp.27-33, 2001.

[7] Ron Jeffries, “What is eXtreme Programming?”, http://www.xprogramming.com/xpmag/whatisxp.htm, 2001

[8] Jonathan Rasmusson, "Introducing XP into Greenfield Projects: Lessons Learnd", IEEE Software, Vol.20, No.3, pp.21-28, 2003.

[9] 阪井誠，松本健一，鳥居宏次, "ダウンサイジング時代のプロセス改善モデル," ソフトウェアシンポジウム'95 論文集, pp.131-140, June 1995.

[10] Peter Schuh, "Recovery, Redemption, and eXtreme Programming", IEEE Software, Vol.18, No.6, pp.34-41, 2001.

[11] Laurie Williams, Robert Kessler, “ペアプログラミング エンジニアとしての指南書”, ピアソン・エデュケーション, pp.209, 2003.

[12] William A. Wood, William L. Kleb, "Exploring XP for Scientific Research", IEEE Software, Vol.20, No.3, pp.30-36, 2003.

[13] 日本 XP ユーザグループ, “日本 XP ユーザグループ”, http://www.xpjug.org/, 2001.