オブジェクト指向開発と開発プロセスの関連

Relation Between Object-oriented Development and a Development Process

桑原　高雄
Takao Kuwabara

概要

オブジェクト指向技術の普及と共に、新しい開発方法が数多く提唱され選択肢が増えている。これら多くの新しい開発方法は、従来の開発方法の欠陥を補うことを目的としているが、その動向は事前に定義された開発プロセスによる定型的な流れ作業による生産方式から、セル生産方式に似た柔軟なものへと広がりを見せている。これらは外見から開発プロセスをわかり難くしている。その原因の１つは、従来ソフトウェア開発で常識とされている前提条件を疑問視した仮説に基づいて構築されていることによる。この状況下で適切な開発方法を選択し効果的に実践してプロジェクトを成功に導くためには、外見からの形式的なアプローチではなく、その背景にある考え方を知っておくことが必要と考える。そのための共通テーマは**「オブジェクト指向の特長を活用して、開発期間中においてもオブジェクト指向のメリットを享受する方法」**と言える。その前提として存在するオブジェクト指向パラダイムの本質を理解していることが重要であり、その合理性や哲学を生かすための具体的な実現方法として開発方法を捉えなおすことが求められる。

1. はじめに

開発プロセスを歴史的に分類すると、3つに大別することができる（図1参照）。伝統的な開発プロセスとして「ウォーターフォール型開発プロセス」があり、その後ウォーターフォール型開発プロセスの欠陥を補うものとして「反復型開発プロセス」が出現した。ここ数年前から新しい潮流として反復型開発プロセスの一種である「アジャイル開発プロセス」と呼ばれているカテゴリーに属する様々な開発方法が注目されている。

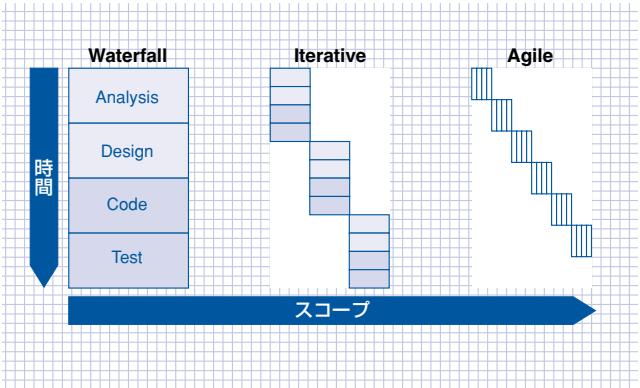

図１　スコープと開発フェーズ　出典：参考文献（６）

2. ウォーターフォール型開発プロセス

2.1 ウォーターフォール型開発プロセスの特徴

ウォーターフォール型開発プロセスは、伝統的な開発プロセスとして古くから多くのプロジェクトで利用されている。これは作業フェーズや作業フェーズ毎のアウトプットがドキュメントとして事前に定義されており、上流から下流までトップダウンで一方向に開発を進める。作業フェーズ毎に作業の中間成果物として作成されるドキュメントは、次の作業フェーズに情報を伝達するための手段として重視される。大規模開発に対応できる重厚長大なプロセスであり、開発を始めれば途中での要求の変化に対応する仕組みを持たない柔軟性に欠ける構造になっている。

2.2 ウォーターフォール型開発プロセスの問題点

ウォーターフォール型の仮説は、「要件は変化しない」ことを前提にしているが、現実には要件は変化する。したがって、開発が終わるまでに要件が変わり、要件に合致しないシステムを作るリスクを負うことになる。また開発途中において、要件の変化に対応しようとすれば、「手戻り作業」が発生し、コストや納期に影響を与える。作業フェーズが後になるほど、その程度は大きくなると言われている。これは、プロジェクトが失敗する大きな要因となりえる。

ウォーターフォール型の開発では、基本的に前の作業フェーズが完了しないと次の作業フェーズを開始することができない。したがって、結果としてトータルな開発期間が長くなる特徴がある。開発期間が長くなればなるほど、企業環境の変化・要求の変化・技術の変化などの外部環境の変化にさらされることになる。変化が穏やかな時代においては、長期間に及ぶ大規模開発も有効であったが、現在のように変化の激しい時代では、短期間に開発することが求められるようになってきている。

通常オブジェクト指向開発では、ウォーターフォール型開発プロセスは適合しない。なぜなら、オブジェクト指向の特長である変化に柔軟に対応するための機能を生かすことができない開発プロセスになっているからである。

3. オブジェクト指向技術を生かす開発プロセス

3.1 オブジェクト指向開発

オブジェクト指向言語を用いた開発やUMLを用いた開発が、必ずしもオブジェクト指向開発とは言えない。オブジェクト指向開発のメリットを出すには、オブジェクト指向パラダイムの目指しているところを明確に意識してソフトウェア設計を行うことが重要である。それらはオブジェクト指向設計原則[(1)][(2)][(3)]として表現されている。この設計原則に従うことによってオブジェクト指向開発のメリットはもたらされる。このメリットは開発中においても享受することができるが、それは反復型開発プロセスによるフィードバックによってもたらされる。

3.2 反復型開発プロセス

ウォーターフォール型開発プロセスの欠陥を改善するものとして、反復型の開発プロセスが提唱された。これはミニウォーターフォールを数回繰り返すことにより開発期間中の要求の変化を吸収する機会を設けるものである。ウォーターフォール型開発プロセスのメリットを継承しながら、デメリットを補う考慮がされていると言える。

その代表的なものがUP （Unified Process）である。UPは伝統的な開発方法と同様に作業フェーズや作業フェーズ毎のUML（Unified Modeling Language）によるドキュメントが事前に定義されているが、ミニウォーターフォールを反復することにより要求の変化に対応することも考慮されている。UPをフル装備で使用すると重厚長大な方法だと言えるが、それぞれの組織やプロジェクトに適合させるように、必要な部分を抽出しカスタマイズして柔軟に使用することを前提としている。

3.3 アジャイル開発プロセス

状況に合わせて「臨機応変」に効率よくソフトウェア開発を行なうアプローチとして、アジャイル（Agile：動きが俊敏な、頭の回転が早い）な方法に分類されるさまざまな開発プロセスが提唱されている。共通基盤として反復型開発プロセスが用いられている。その反復の1つのスコープを小さくし、反復のサイクルを短くし数多く用いることで、さらに変化に対応しやすく、短期間に開発することを狙っている。

アジャイルの代表的なものがXP（eXtreme Programming）である。XPは「ストーリーカード」や「受け入れテスト」と呼ばれる簡潔なドキュメントにより、ユーザーが要求を定義する。開発メンバーは生産効率を高めるため極力余分な中間成果物を作らず、最終成果物であるソースコードをベースにして作業を進める。UPのUMLドキュメントを用いた**計画的設計**に対して、XPはソースコードをベースにした「リファクタリング」による**漸進的設計**[(4)]を行う。漸進的設計とは事前に設計書を作らず、設計をしながらソースコードを作成し同時に実行して設計の妥当性を検証することを繰り返して設計を洗練させるものである。XPはUPのように個々の作業や中間成果物が事前に定義されている方法ではなく、UPの対極に位置付けられる。

（1）プロセスとツールよりも、個人との対話に価値をおく。
（2）包括的なドキュメントよりも、動くソフトウェアに価値をおく。
（3）契約交渉よりも、顧客との協調に価値をおく。
（4）計画に沿うことよりも、変化に対応することに価値をおく。

これは「アジャイル宣言[(5)]」と呼ばれているものであり、アジャイル開発プロセスの特徴を鮮明に表現している。

4. オブジェクト指向設計原則

4.1 OCP(Open-Closed Principle)

オブジェクト指向設計原則の1つであるOCP[1]は、Bertrand Meyerが提唱し「開放/閉鎖原則」と訳されており、設計原則で最も重要な設計目標とされている。OCPの意味するところは、**「拡張に対して開いており、修正に対しては閉じている」**ことである（図2参照）。言い換えれば**「既存のコードを修正することなしに拡張する」**ことを目標にして設計することである。ちなみにGoF（Gang of Four）デザインパターンは、この原則に従っている。このOCPによってオブジェクト指向のメリットはもたらされる。

（1）すべての要件が決まらなくても早期に開発に着手できる。
（2）後で要件が決まった段階で容易に追加が可能である（開発途中・開発後にかかわらず）。
（3）上記のとき既存のコードの修正がない（少ない）ため「手戻り」作業が発生しない。

OCPを完全に守れない場合は、変更箇所が分散しないように局所化させるように設計することもできる。OCPは設計目標を示すだけで、実現方法は以下の設計原則に従って行うことになる。

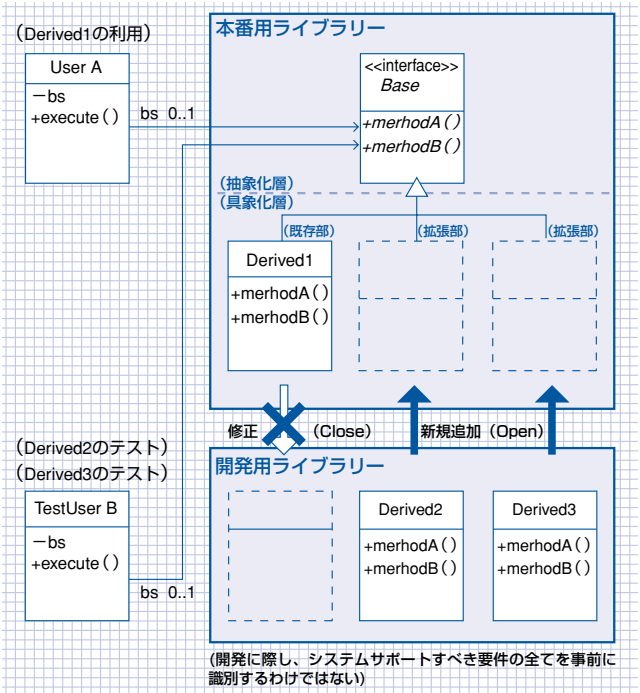

図2　OCPアーキテクチャー

4.2 LSP(Liskov Substitution Principle)

LSP[2]は、Barbara Liskovが提唱し「Liskovの置換原則」と訳されている。LSPはOCPを実現するための重要な設計原則といえる。LSPの意味は、**「サブクラス（実装クラス）は、基底クラス（インターフェース/抽象クラス）と置換可能でなければならない」**と言うことである。わかり易く言うと、**「抽象と具象を分離し、抽象を通して具象にアクセスするようにする」**と言うことである。利用する側は、どの具象クラスをアクセスしているか意識しなくても良いように（具象に依存しないように）、基底クラスの参照型変数で抽象メソッドにアクセスすることにより、具象クラスのメソッドにアクセスさせることである。

（1）抽象は、変化に対してその影響を受けにくい（変化に対して安定している）。
（2）具象は、変化に対してその影響を受けやすい（変化に対して不安定である）。
（3）利用者は、具象に依存しないで抽象に依存することにより、変化の影響を受け難い。

この性質を利用して、変化に対して不安定な具象クラスに依存しないで、安定した抽象クラスに依存させることにより、変化に対して安定したシステムを設計することができる。また具象クラスの変更の影響を、多くの利用者に拡散させないための設計を実現することも可能になる。

また利用する側は、基底クラス（インターフェース/抽象クラス）のみに依存し、具象クラスに依存しないため、インターフェースさえ明確に定義してユーザーの承認を得れば、その具体的な実装方法がすべて決まっていなくても開発に入れることになる。後から具体的な実装方法が決まった時点で順次具象クラスを追加していけばよく、既に開発した既存のプログラムに影響を与えることを回避することが可能になる。このようにしてOCPを実現することが可能になる。

この考え方は、Bertrand Meyerにより**「契約による設計」**と呼ばれ、またGoFにより**「インターフェースに対してプログラミングするのであって、実装に対してプログラミングするのではない」**とも言われており、オブジェクト指向の基本的な考え方になっている。インターフェースを決めることは、ユーザーと開発者の間での契約事項であり、インターフェースを変更することは契約をホゴにすることになる。

4.3 DIP(Dependency Inversion Principle)

DIP[2]は、Robert Martinが提唱し「依存関係反転原則」と呼ばれている。ここで言う依存関係とは、あるクラスが他のクラスを利用している場合、あるクラスは他のクラスに依存し

ていると言う。この関係がある時は他のクラスが変更されると、あるクラスはその影響を受け変更しなければならなくなるかもしれない。依存関係が連鎖し広がっている設計を行うと、依存先の変更に伴う影響は依存関係を遡って伝播していく可能性が高くなる（図3参照）。その結果として、変更箇所が分散化する最悪の事態に陥ることが予想される。このことから、依存関係による依存範囲は、小さいほうが良いことがわかる。最も望ましいのは、自分自身に依存することである。この場合は、自分自身の変更の影響を他に及ぼすことがなく、変更の局所化が可能になる。DIPは、それを実現するための設計原則である。その内容は、**「抽象（インターフェース/抽象クラス）に依存せよ、そうすれば依存関係がそこで遮断されて、更にその向こう側に及ばない」**と言う内容である（図4参照）。一般には抽象的結合と呼ばれているが、Robert Martinはもっとわかり易く**「依存のファイアウォール」**とも呼んでいる。

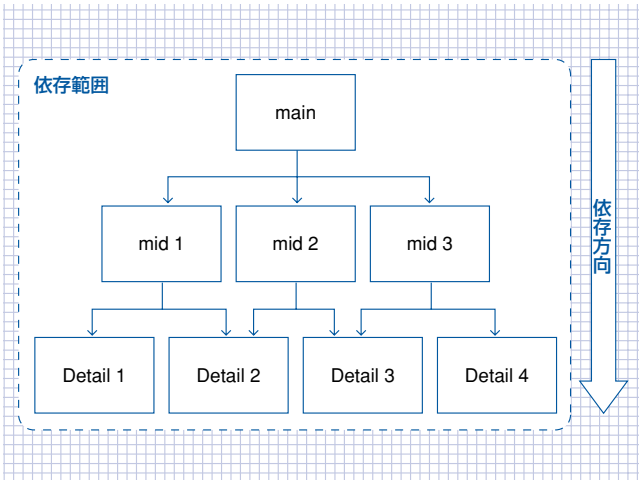

図3　非オブジェクト指向の依存関係

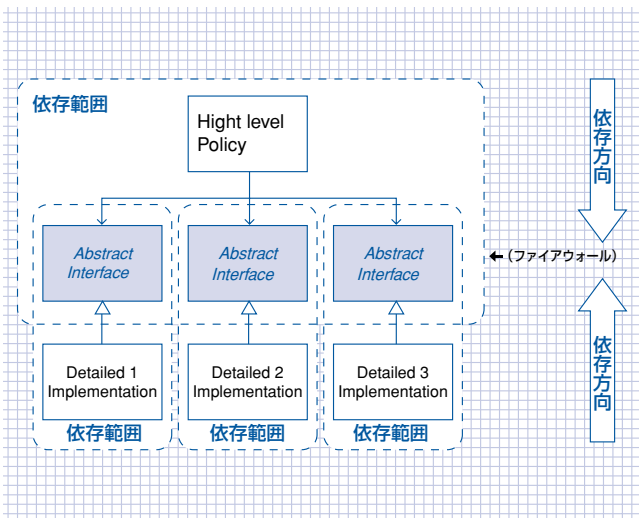

図4　オブジェクト指向の依存関係

5. 設計原則と反復型開発プロセスの効果

設計原則に従うなら、既存部分に極力影響を及ぼさないように、システムを拡張していくことができる。このオブジェクト指向の特性を開発中においても生かす方法は、インクリメンタルな開発が適している。OCPは既存部分に与える影響を少なくすることができるため**「手戻り作業を発生させない」**ように新しい機能を追加拡張していく反復型開発プロセスを効果的なものにする。これは開発期間中に要求の変更を受け入れたとしても「手戻り作業を発生させない」ように開発を進めることができるようになる。

依存関係のある異なるスコープであっても、若干の時間差を設けることで並行して開発することができる。これは、具象に依存しないで抽象に依存することによって実現される。すなわち、インターフェースさえ作成し存在すれば、個別の実装クラスがまだ存在していなくても開発を開始することを可能にする。これは並行作業を可能にして開発期間を短縮するために有効である。

反復型開発プロセスは、初期の反復においてリスクを発見することが可能であり、早期に対策をとることができる。また初期の反復における実績に基づき、以降の反復における計画をより正確なものにすることが可能になる。ウォーターフォール型開発プロセスに比べより安全と言える。

このような理由から反復型開発プロセスは、オブジェクト指向によるソフトウェア開発の基本的な開発プロセスと考えられている。アジャイル開発プロセスにおいても、オブジェクト指向開発の共通の要素として反復は欠かせないものとなっている。ここで重要なことは、**「オブジェクト指向設計原則に従うことが前提」**であることを忘れてはならないことである。

6. XPの開発プロセス

6.1 XPの漸進的設計

XPは迅速に開発を行うことを最大の目的としているため、極力無駄を排除する。リファクタリングによる漸進的設計を行うことにより、一連の開発プロセスにおいて中間の成果物を省略し、最終の成果物であるプログラムのソースコードを重視する。中間成果物を作成しないことにより反復の1サイクルの開発期間を短くし、「短期リリース」を実現しようとする。プロ

グラム中心主義といわれる所以である。これは一連の作業フェーズを同じ開発メンバーが担当することにより、作業フェーズ間の情報伝達を不要にする。設計フェーズからテストフェーズまでを同時並行的に繰り返し、設計の妥当性を検証しながら設計を洗練させていくアプローチをとる。

この背景には、「事前に完全な設計（計画的設計）を行うことは困難であり、設計を行ったとしてもその時点で設計の正当性を検証できない」という考え方がある。また事前に正確な設計を行うには、かなりの時間を必要とするだろう。正当であることを検証できない中間成果物の作成に、それほど時間を投入する価値があるのか。XPでは、このような考え方に基づいて漸進的設計を取り入れている。その結果、中間成果物はチーム内で情報伝達をおこなう手段としての重要な意味があるが、ソースコードをベースにした漸進的設計はその情報伝達手段を失うことになる。XPでは、この欠陥を補うため多くの対策がプラクティスとして考えられている。

6.2 XPのコミュニケーション手段

XPでは、ドキュメントによらないコミュニケーション手段が、プラクティスとして用意されている。

（1）要件分析からテストまでを同一人物が同時並行的に行い、工程間の情報伝達を不要にする。

（2）「ペアプログラミング」を行うことにより、他のチームメンバーと情報の共有化を図る。ペアを組む人を換えながらチーム内で情報の共有範囲を広げていく。

（3）「共同所有」により、プログラムソースそのものを共有化することを主張している。

（4）情報伝達媒体としてのソースプログラムを標準語にするため「コーディング標準」を設ける（コーディング標準の必要性を主張するが、設計標準に関しては必要性を主張していない）。

（5）チームメンバーが何時も目の届く範囲にいて、必要があれば何時でもコミュニケーションがとれる環境「オープンワークスペース」を作る。

（6）ユーザーとのコミュニケーションを図るためチームに参加する「オンサイトのユーザー」。

（7）わかり難いシステムについてのアーキテクチャーを、わかり易い「メタファ」（比喩）で表現し、会話の中で共有し洗練させる。

このようなコミュニケーション手段には、おのずと限界が生じる。XPは少人数の開発チームにおいては有効に機能するが、人数が多くなるとうまく機能しないと言われている理由である。

6.3 XPと設計原則

XPは最終成果物においても、無駄を省くことを主張している。すなわち、「現時点で必要とされる動作のための最小限の設計」（「シンプルな設計」）だけを行い「後で使うかもしれないと考えて柔軟な設計」を行ってはならないことになっている。それは、将来その機能が実際に使用される保証はどこにもなく、使われないかもしれない機能を予め追加して無駄にするよりも、必要になった時点で変更コストを払って機能を追加したほうがはるかに良いと主張している「YAGNI（You aren't Gonna Need It）：多分それは必要にならない」。これはXPとOCPが一見相反するように見える。筆者は、相反するものではないと考える。問題は、「現時点で必要とされる動作のための最小限の設計」の中にOCPが含まれるか否かというところにあるが、次の2つの理由により含まれると考える。

（1）XPでは変更コストは緩やかに上昇するという仮説に基づいて、新しい機能が必要になった時点で追加したほうが無駄がなく良いと主張している。この仮説が成り立つのはOCPに準拠している場合においてである。

（2）XPは「リファクタリング」を積極的に行うことを指導している。リファクタリングは、「内部構造の改善であり、インターフェースや振る舞いを変更しない」こととされている。

これら2つの理由により、この誤解を引き起こしやすい部分の整理ができる。OCPは、「現時点で必要とされる動作のための最小限の設計」に含まれ、「後で使うかもしれないと考えて柔軟な設計」にはOCPに基づいて将来拡張する機能（インターフェースに対する実装クラスの追加）が含まれると考えるのが自然である。

7. おわりに

第12回ソフトウェア開発環境専門セミナー（2003/07/10）の特別セッションにおいて、Peter Coad氏がアジャイルプロセス（企業独自の開発手法「秘伝のレシピをつくるには」）というテーマで、ボーランド社の事例を紹介した（表1参照）。ボーランド社では、UPとFDD（Feature-Driven Development）とXPの3つの開発方法を事前に用意し、組織やプロジェクト

毎に最適な開発プロセスを選択し適用している。現時点においてUPとXPはオブジェクト指向開発方法の対極に位置している。UPは要件が明確で安定している開発に適している。大規模で本格的な開発にも耐えうる方法である。要件が明確であれば、事前の開発計画立案や工数見積が可能である。反対にXPは要件が不明確であり変わり易い高度な開発に向いている。大規模な開発には適していない。この場合は事前の工数見積が困難であり、顧客との契約方式は実績精算方式が向いている。FDDは両者の中間に位置する。この3つの開発プロセスの組み合わせを変えることにより、多様なバリエーションが構成できる。どのような構成をおこなうかは、企業の方針や文化や技術的成熟度により変わると考える。アプリケーションの大量生産型企業と、ツールやパッケージ製品の研究開発型企業では、異なったものになるであろう。同じものであったとしても、それぞれの組織にあった最適化が行われた結果として運用方法や適用レベルが異なることが予想される。プロジェクトを成功に導くためには、自分たちの開発目標や組織に合った開発方法の選択と、絶えずチューニングして最適化を目指す努力が不可欠であると考える。ある開発方法を採用すれば利益を約束してくれる訳ではなく、意図的に利益を創出するような使用が必要である。

表1　Selecting mini-recipes and A spectrum of recipes

	UP	FDD	XP
Define	Use cases and Class diagrams	Features list and class diagrams (subteams then teams)	User stories
Design	Sequence diagram	Sequence diagram	Refactor
Build	Code	Feature teams (chief programmers and class owners)	Pair programming (team ownership)
Test	Code, test, inspect	Code, test, inspect	Write tests, code, test and Continuously inspect
Process	Give me control	Give me just enough process	Give me freedom
Developers	>20	10〜40	3〜12

出典：参考文献（7）

参考文献

（1）Robert C. Martin：“The Open-Closed Principle”,（1996）
（2）Robert C. Martin：“The Liskov Substitution Principle”,（1996）
（3）Robert C. Martin：“The Dependency Inversion Principle”,（1996）
http://members.core.com/8E/4B/dmumaugh/OOT/Design-Principles/
（4）Martin Fowler：“Is Design Dead?”,（2000）
http://www.martinfowler.com/articles/designDead.html
（5）Manifesto for Agile Software Development：
http://www.agilemanifesto.org/ ,（2001）
（6）長瀬嘉秀：@IT掲載の“長瀬嘉秀のソフトウェア開発最新事情（2）”,（2003）
http://www.atmarkit.co.jp/fjava/devs/column/nagase02/nagase02.html
オリジナルは「Object Mentor社　資料より」
（7）Peter Coad：“Agile Process: Developing Your Own Secret Recipes”,（2003）
第12回ソフトウェア開発環境専門セミナー資料

桑原　高雄
Takao Kuwabara
・技術本部
・昨年度までオブジェクト指向開発に携わっていたが、現在はEnterprise Project Management System導入プロジェクトに従事