第2章　プロセスモデル

ソフトウェア開発は反復型学習プロセスといえる。プロセスの実行により知識が抽出され、関係者に集積されていく。そして、体系化され具現化することで、Baetjerが「ソフトウェア資産」と呼ぶ[Bae98]成果物が生み出されていく。

技術的な見地からすると、ソフトウェアプロセスとは何を示すのか。本書ではソフトウェアプロセスを高品質のソフトウェアを開発するために必要なアクション、タスクから構成されるフレームワークであると定義する。それでは、プロセスとはソフトウェアエンジニアリングと同じ意味だろうか。答えは、YESでありNOでもある。ソフトウェアプロセスは、ソフトウェア開発の際に採用するアプローチを決める。一方、ソフトウェアエンジニアリングも技術的手法や自動化ツールといったプロセス内で使用する技術を提供する。

重要なのは、創造的で見識にあふれた人々により、プロダクトの特徴や市場の需要に適した成熟したプロセスに従い、ソフトウェエンジニアリングが行われるべきだという点である。

2.1 プロセスモデルの構造とプロセスフロー

プロセスは成果物が生み出される際に実施される一連のアクティビティ、アクション、タスクを含むと第1章にて記述した。これらアクティビティ、アクション、タスクはモデルやフレームワークを用いることでプロセスとの関連性が表現できる。

ソフトウェアプロセスを図2.1にて模式的に示す。図2.1のように、各フレームワークアクティビティはいくつかのソフトウェアエンジニアリングアクションを伴っている。各ソフトウェアエンジニアリングアクションは、1つのタスクセットによって定義される。タスクセットは、完遂されるべきタスク、生み出される成果物、求められる品質評価ポイント、進捗を示すためのマイルストンを特定する。

図2.1：ソフトウェアプロセスフレームワーク

Software Process　ソフトウェアプロセス
Process framework　プロセスフレームワーク
Umbrella activities　包括的なアクティビティ
Framework activity　フレームワークアクティビティ
Softwaer engineering action #1.1：ソフトウェアエンジニアリングアクション #1.1
Task sets　タスクセット
work tasks　タスク
work products　成果物
quality assurance points　品質保証ポイント
project milestones　プロジェクトマイルストン

第1章では、ソフトウェアエンジニアリングにおけるプロセスフレームワークとして5つの一般的なフレームワークアクティビティを定義した。コミュニケーション、計画策定、モデリング、構築、デプロイである。あわせて、ソフトウェアプロジェクトの追跡と管理、リスクマネジメント、品質保証、構成管理、テクニカルレビューといった包括的なアクティビティがプロセス全体に適用される。

ここで、ソフトウェアプロセスの重要な側面をもうひとつ紹介する。「プロセスフロー」は、プロセスフレームワーク内でフレームワークアクティビティ（アクションやタスクを含む）がどのような順序、時系列で発生するかの体系を示す。図2.2にプロセスフローを図示する。

```
図2.2：プロセスフロー

Linear process flow　リニアプロセスフロー
Iterative process flow　反復プロセスフロー
Evolutionary process flow　進化型プロセスフロー
Parallel process flow　パラレルプロセスフロー

Communication　コミュニケーション
Planning　計画策定
Modeling　モデリング
Construction　構築
Deployment　デプロイ
```

図2.2「（a）リニアプロセスフロー」では、5つのフレームワークアクティビティは順番に実行される。コミュニケーションから始まり、デプロイで締めくくられる。「（b）反復プロセスフロー」では、とあるアクティビティの前段階アクティビティが繰り返される。「（c）進化型プロセスフロー」では循環的にアクティビティが実行される。アクティビティの循環により、より進んだソフトウェアバージョンが生み出される。「（d）パラレルプロセスフロー」では、あるアクティビティと並行して他のアクティビティが実行される。ソフトウェアの一部分に対するモデリングアクティビティは、ソフトウェアの他部分の構築アクティビティと並行して実行される。

2.2 フレームワークアクティビティとソフトウェアエンジニアリングアクション[訳注1]

第1章にて5つの一般的なフレームワークアクティビティ（コミュニケーション、計画策定、モデリング、構築、デプロイ）を取り上げ基本的な定義をしたが、ソフトウェアプロセスとしてこれらアクティビティを正しく実行しようとすると、ソフトウェアチームはより多くの情報を必要とするだろう。つまり、「解決すべき問題の本質や、担当者やプロジェクトのスポンサーとなるステークホルダの特徴にあわせて、どのようなアクションを行うことが各フレームワークアクティビティに求められるのか」という重要な疑問に直面するのである。

[訳注1] ソフトウェアエンジニアリングアクションは、本書では単にアクションと記載する場合もある。

一人でも可能な複雑でもない要求を扱う小さなソフトウェアプロジェクトでは、コミュニケーションのアクティビティは特定のステークホルダとの電話やEメールでのやり取りにすぎない。開発者にとって唯一必要とされるアクションは「電話での会話」であり、以下のタスク（タスクセット）となる。

1. ステークホルダと電話にてコンタクトを取る。
2. 要求を議論して記録する。
3. 記録を整理して、端的な要件記述を作成する。
4. Eメールでステークホルダにレビューを依頼をし合意を取る。

プロジェクトに多くのステークホルダがいて、各ステークホルダが異なる要求を持ち、それぞれの要求に対立が発生しているような複雑な状況下では、コミュニケーションのアクティビティに6つのアクションが必要となる。それは、方向付け（inception）、要求獲得（elicitation）、推敲（elaboration）、交渉（negotiation）、仕様化（specification）、検証（validation）である。各ソフトウェアエンジニアリングアクションには、いくつものタスクや成果物が求められる。

2.3 タスクセットの特定

図2.1について再度確認しよう。各ソフトウェアエンジニアリングアクション（例：コミュニケーションアクティビティにて必要とされる要求獲得アクション）は、成果物、品質保証ポイント、プロジェクトマイルストン、実施するタスクの集合体であるタスクセットによって表現される。 異なるプロジェクトでは、異なるタスクセットが求められる。プロジェクトのニーズやチームの特徴に最も適したタスクセットを選び抜く必要がある。このことは、特定のプロジェクトニーズやチームの特徴にあわせてソフトウェアエンジニアリングアクションを柔軟に対応させるべきだという意味でもある。

> タスクセット
>
> タスクセットは、ソフトウェアエンジニアリングのアクションの目的達成に向けた実作業を定義する。たとえば、要求獲得（より一般的に「要件収集（requirements gathering）」と呼ぼう）はコミュニケーションのアクティビティにおける重要なソフトウェアエンジニアリングアクションである。要件収集のゴールは、開発するソフトウェアに対してのさまざまなステークホルダの期待を理解することである。
>
> 比較的単純で小さいプロジェクトでは、要件収集におけるタスクセットは以下となる。
>
> 1. プロジェクトにおけるステークホルダのリストを作る
> 2. ステークホルダを非公式的な会議に招待する
> 3. 各ステークホルダに質問をして、要件となる機能やフィーチャのリストを作る
> 4. 要件を議論し、最終的なリストを構築する
> 5. 要件の優先順位をつける
> 6. 不明確な部分を注記する
>
> 規模が大きく、複雑なソフトウェアプロジェクトでは、次のようなタスクセットが必要だろう。
>
> 1. プロジェクトにおけるステークホルダのリストを作る
> 2. 各ステークホルダに個別にインタビューを行い、ニーズの全体像を決める
> 3. ステークホルダの情報に基づいた機能やフィーチャの暫定的な要件リストを作る
> 4. アプリケーションの仕様調整を進める一連の会議を設定する
> 5. 会議を実施する
> 6. 会議の一部で非公式のユーザシナリオを作成する
> 7. ステークホルダのフィードバックによってユーザシナリオを洗練させる
> 8. ステークホルダの要件リストを改訂する
> 9. 品質機能展開（QFD）の技術を用いて、要件の優先順位をつける
> 10. 段階的にリリースできるように要件をパッケージ化する
> 11. システムにおける制約や制限事項を注記する
> 12. システムの妥当性を確認する方法を議論する
>
> 双方のタスクセットは共に要件収集を達成するものであるが、活動の深さと形式的手続きの度合いが大きく異なる。各アクションのゴールを達成し、品質と俊敏性を確保し得るタスクセットをソフトウェアチームは選択する。

2.4 プロセス評価と改善

ソフトウェアプロセスが存在するからといって、ソフトウェアが納期に納品され、顧客ニーズを満たし、長期的な品質を確保できるという保証はない（第15章参照）。プロセスパターン[訳注2]は確固たるソフトウェアエンジニアリング（第2部参照）の実践と共に存在しなければならない。さらに、ソフトウェアエンジニアリングを成功させるために必要不可欠な基本的なプロセス基準を確実に満たすよう、プロセス自体も評価される必要がある[*1]。

[*1] SEIによるCMMI-DEV[CMM07]では、ソフトウェアプロセスの特徴とプロセスの成功へ向けた多数の基準を詳細に記している。

[訳注2] プロセスパターンは2.5節や第3章（アジャイル開発プロセス）等で紹介される各プロセスモデルの総称。

多くのエンジニアにおける今どきの考え方は、ソフトウェアプロセスやアクティビティは数値的な測定や分析（メトリクス）を用いて評価されるべきだというものである。プロセスを評価するソフトウェアプロセスメトリクスについては第17章を見るとよい。プロセスアセスメントや改善手法の詳細については、第28章に示している。

2.5 規範的なプロセスモデル

規範的なプロセスモデル[*2]では、プロセス要素を規定し、プロセスワークフローを定義する。規範的なプロセスモデルは、ソフトウェア開発に構造と秩序をもたらす。プロセスに定義された進捗ガイドに従い、アクティビティやタスクは順次実施されていく。しかし、変化に晒されるソフトウェアの世界において規範的プロセスモデルを採用することはふさわしいのだろうか。伝統的なプロセスモデル（およびそのモデルによる秩序）が否定され、より構造化されていない何かにプロセスが置き換えられた場合、ソフトウェア業務における協調や一貫性の維持は不可能とはならないだろうか。

[*2] 規範的なプロセスモデルは、「伝統的な」プロセスモデルと呼ばれる時もある。

これらの問いに簡単な答えはないが、ソフトウェアエンジニアにはいくつかの選択肢が存在する。本章では、秩序とプロジェクトの一貫性の維持を第一に考える規範的なプロセスのアプローチについて概要を紹介する。これらを「規範的」と呼ぶ理由は、どのプロセスも一連のプロセス要素であるフレームワークアクティビティ、ソフトウェアエンジニアリングアクション、タスク、成果物、品質保証、および変更管理の仕組みをプロジェクトで規定しているためである。各プロセスモデルは、プロセスフロー（ワークフローとも呼ばれる）を規定している。各プロセス要素は他の要素と相互関係をもつ。

あらゆるソフトウェアプロセスは、第1章で述べた一般的なフレームワークアクティビティを含む。しかし、プロセスによってどのアクティビティを重視するかは異なり、各フレームワークアクティビティ（ソフトウェアエンジニアリングアクションとタスクを含む）におけるプロセスフロー定義も異なる。第3章および第4章では、多くのソフトウェアプロジェクトにおいて避けられない変更に対応するためのソフトウェアエンジニアリングプラクティスについて記す。

2.5.1 ウォーターフォールモデル

ある問題に対する要求が十分理解されていて、コミュニケーションからデプロイまで順に作業を実施できる場合がある。こうした状況は、明確に定義された既存システムの修正や拡張（例えば、政府の規制変更により必ず実施しなければならない会計ソフトウェアの修正等）で発生する場合がある。また、一部の新規開発でも、要求が明確に定義されており、変更があまり起こらないことは同様に起こり得る。

ウォーターフォールモデルは、線形逐次型モデルと呼ばれることもある。これは顧客の要件から始まり、計画策定、モデリング、構築、デプロイと順に進み、完成したソフトウェアの継続的なサポートの実施にいたるまで体系的かつ逐次的にソフトウェア開発が行われるアプローチ[*3]である（図2.3参照）。

[*3] Winston Royce[Roy70]によって提案された本来のウォーターフォールモデルは「フィードバックループ」を考慮しているのだが、このプロセスモデルを適用した組織の大部分は、厳密な逐次型プロセスとして扱ってしまった。

```
図2.3：ウォーターフォールモデル
Communication　コミュニケーション
Planning　計画策定
Modeling　モデリング
Construction　構築
Deployment　デプロイ

project initiation　プロジェクト開始
requirements gathering　要件収集

estimating　見積もり
scheduling　スケジュール作成
tracking　追跡管理

analysis　分析
design　設計

code　コーディング
test　テスト

delivery　リリース
support　サポート
feedback　フィードバック
```

ウォーターフォールモデルは、ソフトウェアエンジニアリングにおける最も古いパラダイムである。しかし、登場から50年以上が経過し、積極的に支持していた人々でさえ、今では有効性に疑問を抱くようになった。ウォーターフォールモデル適用時の問題には、次のようなものがある。

1. 現実のプロジェクトでは、モデルに示されているワークフローの順番どおり進行することはほとんどない。
2. プロジェクト開始時に顧客が要求事項をすべて明確に述べることは難しい。
3. 実際に動作するプログラムは、プロジェクトの後半になるまで入手できず、顧客は辛抱強く待たなければならない。
4. プログラムが動く段階になるまで大きな問題が発見されない場合がある。

今日、ソフトウェア開発はスピードが求められ、（機能やフィーチャ、情報コンテンツに対する）変更の奔流に晒されている。ウォーターフォールモデルは、こうした状況には適していない。

2.5.2 プロトタイピング

ソフトウェアの大まかな目的は定まっていても、機能やフィーチャの詳細まで決まっていないことは多い。開発においても、アルゴリズムの効率、OSの適合性、ユーザインタフェースの使いやすさに自信のない場合がある。このような場合には、プロトタイピングのパラダイムが最良のアプローチとなるだろう。

プロトタイピングは単独のプロセスモデルとしても適用できるが、本章で述べる他のプロセスモデル内で利用できる技術のひとつとして扱われる方が一般的である。どこに適用されるかは関係なく、要求があいまいな場合にプロトタイピングを用いることで、何を開発すべきかを他のステークホルダと理解を深めることができる。

フィットネスアプリを例とすると、次の1〜3のように段階的なプロトタイプが開発される。

1. スマートフォンとフィットネスデバイスが同期し、画面上に現在のデータだけ表示できる。
2. ゴールの設定ができる。クラウド上にフィットネスデバイスのデータを蓄積できる。顧客からのフィードバックを受け、ユーザインタフェースが追加、修正される。
3. SNSツールと連携し、フィットネスのゴールや進捗状況を友人と共有できる。

```
図2.4：プロトタイピングパラダイム

Communication　コミュニケーション
Quick plan　クイック計画
Modeling　モデリング
Quick design　クイック設計
Construction of prototype　プロトタイプの構築
Deplyment　デプロイ
Delivery & feedback　リリースとフィードバック
```

プロトタイピングパラダイム（図2.4参照）は、コミュニケーションから始まる。ソフトウェア技術者とステークホルダが顔を合わせ、ソフトウェアの全体的な目標を定める。その時点で思いつく要求事項をすべて明らかにし、さらに決定の必要がある部分は概要を作成する。迅速にプロトタイピングの計画策定をして、（「クイック設計」という形式で）モデリングが行われる。クイック設計では顧客が確認できる部分（ユーザインタフェースのレイアウトや出力形式等）に焦点を当てる。これによってプロトタイプができあがり、さらなる洗練へ向けたフィードバックのためにステークホルダの評価を受ける。プロトタイプは、さまざまなステークホルダのニーズを満たし、エンジニアが開発すべきことをより理解するために繰り返し使われる。

プロトタイプはソフトウェア要求を明らかにするためだけの仕組みとして使用することが理想的である。実際に動作するプロトタイプをつくる場合には、既存のプログラムの一部を利用したり、動くプログラムを手早く生成できるツールを用いることが多い。

ステークホルダとエンジニアは双方ともプロトタイプを作ることを好む。ユーザは実際のシステムの感触を得ることができ、エンジニアはただちに動くものを構築できるからである。しかしプロトタイピングは、次の理由により問題を引き起こす可能性がある。

1. ステークホルダは目に入ったものが実際のソフトウェアのバージョンだと勘違いしてしまう。プロトタイプのアーキテクチャ（プログラムの構造）もあわせて進化し続けるものとは考えもしない。プロジェクトメンバーが全体としてのソフトウェア品質や長期にわたる保守性を考慮してない場合に特に問題となる。
2. エンジニアもプロトタイプを短時間で動くようにするため、実装面で妥協してしまう。慎重な性格でない限り、システムの主要な部分を理想とはほど遠い選択結果が占めてしまう。

このようにいくつか問題はあるが、プロトタイピングはソフトウェアエンジニアリングにおける有効なパラダイムである。重要なことは、最初にゲームのルールを決めておくことである。つまり、要求を定義するための道具としてプロトタイプをつくることを、あらゆるステークホルダに理解してもらう必要がある。時折、プロトタイプを最終的なプロダクトに育て上げようという意見が発生する。しかし実際には、顧客の変わりゆくニーズによって、プロトタイプは（少なくとも部分的に）廃棄されるものだ。

2.5.3 進化型プロセスモデル

他の複雑なシステム同様、ソフトウェアは長い期間をかけて進化するものである。開発が進むにつれてビジネスやプロダクトに対する要求が変化するため、最終プロダクトへとまっすぐ突き進むことは現実的ではない。市場における厳しい納期制約において、複雑なソフトウェアプロダクトをすべて完成させることは不可能である。ただ、競合やビジネスにおける強い期待に応える限られたバージョンを作ることや、全体的なフィーチャが理解されたうえで改善を施したバージョンのリリースであれば可能である。このような状況では、成長し変化するプロダクトに適応するよう設計されたプロセスモデルが必要となる。

Barry Boehm[Boe88] が最初に提案したスパイラルモデルは、反復型のプロトタイピングモデルとウォーターフォールモデルにおける体系的で統制された側面を組合せた進化型のプロセスモデルである。このモデルは、徐々に価値が高まるソフトウェアを迅速に開発できる特徴がある。

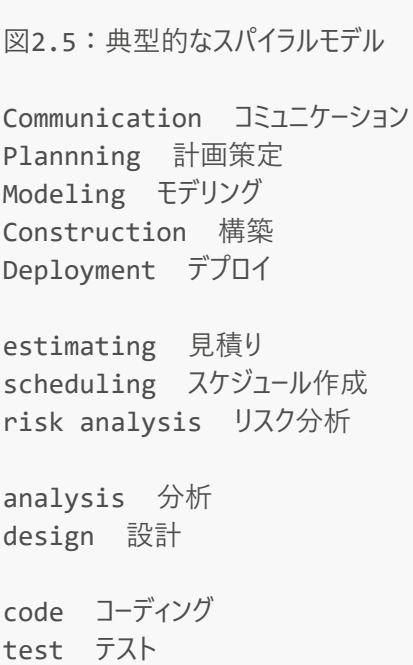
スパイラルモデルでは、進化型のリリースを繰り返しながらソフトウェア開発が行われる。この開発イテレーション（成果物を生み出す反復を意味する）における初期の成果物は、紙に描かれたモデルやプロトタイプの場合もある。後半の開発イテレーションでは、より価値の高いプロダクトが徐々にできあがっていく。

スパイラルモデルは、エンジニアリングチームが定めるいくつかのフレームワークアクティビティに分割される。説明のため、前述した一般的なフレームワークアクティビティを用いてみよう[*4]。各々のフレームワークアクティビティは、図2.5のらせん上の一区画で表される。進化型プロセスが始まるとソフトウェアチームは、らせんの中心から時計回りに進みながら、図の区画に示されたアクティビティを実施していく。サイクルが進むごとに、リスク（第26章）が検討される。らせん上にはアンカーポイントマイルストン（らせんに沿って実施される作業の成果物と達成条件を組合せたもの）が定義されている。

第1サイクル（図2.5の中央に近い内側を起点とする）ではプロダクト仕様が開発される。以降、らせんを1周するたびにプロトタイプ、さらに洗練されたプロダクトが順次開発される。計画策定の区画を通過するたびに、プロジェクト計画が再確認される。リリース後の顧客からのフィードバックにもとづいてコスト、スケジュールが調整され、プロジェクトマネージャはソフトウェアの開発に必要なイテレーションの残り回数を見直す。

[*4] 本章で解説するスパイラルモデルは、Boehmによって提唱されたモデルに手を加えたものである。オリジナルのスパイラルモデルについては、参考文献[Boe88]を参照のこと。また、Boehmのスパイラルモデルについてのより新しい論考は参考文献[Boe98]や[Boe01a]を参照のこと。

図2.5：典型的なスパイラルモデル



```
delivery  リリース
feedback  フィードバック
```

ソフトウェアがリリースされたら完了する他のプロセスモデルと異なり、スパイラルモデルはソフトウェアのライフサイクル全体に適用できる。スパイラルモデルは、大規模なシステムやソフトウェアの開発に対する現実的なアプローチである。スパイラルモデルでは、プロジェクトのすべての段階において技術的リスクを考慮することが求められるので、適切に運用することでリスクを回避できるだろう。

しかし他のパラダイムと同じように、スパイラルモデルも万能ではない。進化型アプローチで開発の統制をとることができることを顧客に納得させることは難しく、契約が絡む場合には特に困難であろう。スパイラルモデルを役立てるにはリスク評価についての深い知識や技術が必要であり、成功するかどうかはリスクの扱い方に依存する。主要なリスクを発見し管理できないならば、間違いなく問題が起こる。

現代のソフトウェアは、継続的な変更、非常に厳しいスケジュール、および顧客やユーザの満足が強く求められるという背景から特徴づけられていることはすでに述べた。多くの場合、市場に出るまでの時間は最重要マネジメント要素である。マーケット投入時期を逃すと、プロジェクトそのものが無意味なものとなりかねない[*5]。

[*5] 重要：市場において最初のプロダクトであることが、必ずしも成功を保証するものではない。実際、成功したソフトウェアプロダクトの多くは、2番手もしくは3番手で市場に登場している。先行したプロダクトの失敗から学んでいるということだ。

進化型モデルの目的は、高品質のソフトウェア[*6]を、反復もしくはインクリメンタルな方法で開発することである。進化型プロセスを用いて、柔軟性、拡張性、および開発速度を高めることも可能である。ソフトウェアチームとそのマネジャーは、プロジェクトとプロダクトにおける重大な調整項目や（ソフトウェア品質を決定する究極の要素である）顧客の満足度の間で、適切なバランスをとることに挑戦しているのである。

[*6] ここでは、ソフトウェア品質を広い意味で定義している。顧客満足だけではなく、技術的な判断基準（第2部参照）も含めた意味をもつ。

2.5.4 統一プロセス

ある意味、統一プロセス[Jac99]は伝統的なソフトウェアプロセスモデルにおける優れた特徴や性質を引き出そうとしているが、アジャイルソフトウェア開発（第3章参照）の優れた原則の多くも実装しているという特徴をもつ。統一プロセスは、顧客とのコミュニケーション、並びに顧客視点でシステムを記述する合理的な手法であるユースケース[*7]を重要視している。 また、ソフトウェアアーキテクチャの重要性を強調しており、「アーキテクトが理解容易性、将来の変更への対応、および再利用のような正しいゴールへ集中できるようにしている」[Jac99]。統一プロセスは反復的かつインクリメンタルなプロセスフローを提唱し、現代のソフトウェア開発には不可欠な進化型の考え方を取り入れている。

[*7] ユースケース（第7章参照）は、ユーザから見たシステムの機能やフィーチャを記述したテキスト（シナリオ）やテンプレートである。ユースケースはユーザによって書かれ、より包括的な分析モデルを作る基盤となる。

統一モデリング言語（UML：unified modeling language）は統一プロセスを支えるために開発された。UMLはオブジェクト指向システムのモデリングと設計に向けた強力な表記法を有し、あらゆるドメインのソフトウェア開発のモデリングにおける業界標準となった。UMLの基本的な記法やモデリングルールに慣れていない読者に対して、Appendix1では導入向けチュートリアルと推奨書籍リストを用意している。

図2.6は、統一プロセスの「フェーズ」を示したもので、本章2.1で述べた一般的なアクティビティと対応付けている。

図2.6：統一プロセス

Communication　コミュニケーション
Plannning　計画策定
Modeling　モデリング
Construction　構築
Deployment　デプロイ

Software increment　ソフトウェアのインクリメント

Inception　方向付け
Elaboration　推敲
Transition　移行
Production　運用
Release　リリース

統一プロセスの「方向づけ（inception）フェーズ」は、顧客とのコミュニケーションと計画策定のアクティビティに対応する。基本的なビジネス要求は、ソフトウェアが実現された際に主要なタイプのユーザがそれぞれどのようなフィーチャ、機能を求めているかを記述したユースケース（第7章参照）によって暫定的に表現される。計画策定では、リソースを識別し、主要なリスクを評価し、ソフトウェアのインクリメントに向けてスケジュールを作成する。

推敲（elaboration）フェーズは、一般的なプロセスモデルにおけるコミュニケーションとモデリングのアクティビティに対応する（図2.6参照）。

推敲では、方向づけフェーズで作成した暫定的なユースケースを洗練し、拡張する。また、5つのビュー（ユースケースモデル、分析モデル、設計モデル、実装モデル、デプロイモデル）を含むアーキテクチャベースライン[*8]を作る。この時点で計画の修正が行われる場合もある。

[*8] 重要：アーキテクチャベースラインは、それが使い捨てでないという点からプロトタイプとは呼ばない。統一プロセスの次フェーズでベースラインはより詳細化されていく。

統一プロセスの構築（construction）フェーズは、一般的なソフトウェアプロセスの構築アクティビティと完全に同一である。ソフトウェアのインクリメント（すなわちリリース）に必要なすべてのフィーチャおよび機能は、ソースコードに実装される。

コンポーネントが実装されるにつれて、ユニットテスト[*9]が設計され毎回実行される。その後、統合アクティビティ（コンポーネント結合とインテグレーションテスト）が実行される。受入テストがユースケースをもとに設計され、統一プロセスの次フェーズの開始前に受入テストが実施される。

[*9] （ユニットテストを含む）ソフトウェアテストの包括的な内容は第19章～第21章にて記す。

統一プロセスの移行（transition）フェーズは、構築アクティビティの後半と、デプロイアクティビティの最初の部分（リリースとフィードバック）から構成される。ソフトウェアと関連するドキュメントがエンドユーザによるベータテスト向けに提供され、ユーザは欠陥と修正が必要な点について報告する。移行フェーズの完了時には、ソフトウェアのインクリメントが利用可能なソフトウェアとしてリリースされる。

統一プロセスの運用（production）フェーズは、一般的なプロセスのデプロイアクティビティと一致する。このフェーズでは利用中のソフトウェアがモニタされ、稼動環境（インフラ）に対するサポートが行われ、欠陥レポートや変更要求が提示され、評価される。

構築、移行、運用の各フェーズが同時に実施されることや、次のインクリメントの開発が並行して始まっている場合もある。統一プロセスの5つのフェーズは順番に行われるのではなく、どちらかといえば、時間的に少しずらして並行実施されることを意味する。

あらゆるソフトウェアプロジェクトにおいて、統一プロセスにおけるワークフロー内のすべてのタスクが実行されるわけではない。チームは、必要に応じたプロセス（アクション、タスク、サブタスク、成果物）を採用すべきだ。

2.6 プロダクトとプロセス

すでに紹介したプロセスについて、メリットとデメリットを表2.1にまとめた。本書における過去の版では他のプロセスモデルについても議論している。現実としては、どのプロセスもあらゆるプロジェクトに適用できるような完璧なものではない。たいてい、ソフトウェアチームは自分たちのプロジェクトのニーズにあわせて、2.5節で記載したプロセスモデルもしくは第3章記載のアジャイルプロセスモデルのいずれかのうち、1つもしくは複数を採用している。

表2.1 プロセスモデルの比較

プロセス名	メリット	デメリット
ウォーターフォール	計画を理解しやすい 要件が明確な小さなプロジェクトでうまくいきやすい 分析やテストが順に実施される	変化への対応が考慮されていない テストがプロセスの後半になってしまう 顧客の承認判断が最終段階となってしまう
プロトタイピング	要件の変更の影響を小さくする 顧客と早い段階から頻繁に協業する 小さなプロジェクトに非常に適している プロダクトが却下される可能性を低くする	顧客との協業が遅れに繋がる場合がある プロトタイプのままリリースする誘惑にかられる プロトタイプを廃棄するロスが発生する 計画およびマネジメントが難しい
スパイラル	継続的に顧客と協業する 開発リスクがマネジメントされる 大きく複雑なプロジェクトに適している 拡張性があるプロダクトに非常に適している	リスク分析の成否がプロジェクトの運命を決める プロジェクトのマネジメントが難しい 開発のエキスパートチームが求められる
統一プロセス	品質ドキュメントが重視される 継続的に顧客と協業する 変化への対応が考慮されている 保守プロジェクトに非常に適している	ユースケースが正確ではない場合がある ソフトウェアインクリメントの統合時に工夫が必要 フェーズが重なる箇所で問題が発生しやすい 開発のエキスパートチームが求められる

プロセスが貧弱ならば、できあがったプロダクトが思わしくないのは当然である。しかし、プロセスに過剰な信頼を寄せることも同じように危険である。Margaret Davisは、プロダクトとプロセスの双対性について

次のように述べている[Dav95a]。

> およそ5年から10年ごとに、ソフトウェアの分野では、プロダクトの問題からプロセスの問題に論点を移すことによって「問題」を再定義している。...《省略》

> 振り子は、自然な性質として両極端の地点の中間に落ち着くが、ソフトウェア分野では、片方のアプローチがうまくいかない場合には、新しい力が作用し、振り子が反対方向へ振れてしまうことが絶えず発生する。こうした振れがよくないのは、作業が適切に実行されていないことはいうまでもなく、仕事の意味が極端に変化するために開発者が混乱するからである。振れによって「問題」が解決されることはなく、失敗する運命にある。なぜなら、プロダクトとプロセスは双対的に捉えるものであるにもかかわらず、対立するものとして捉えているためである。

> ...《省略》完全な人工物をプロセスだけ、またはプロダクトだけしか見ないのであれば、その背景、使い方、意味、価値について決してわかりはしない。

> 人間のすべての活動はプロセスとみなすことができるが、人間は誰でもそれらの活動から自分にとって価値のある意味を導き出している。そして、それを複数の人間によって使われたり価値を認められたり、繰り返し使われていくうちに、考慮されていなかった状況で使われる可能性がある表現や実例が生み出される。つまり人間は、自分自身や他の人間によるプロダクトの再利用から満足という感情を得るのである。

> したがって、ソフトウェア開発における再利用というゴールの迅速な浸透は、開発者が自分の仕事から得る満足を潜在的に増加させる一方で、プロダクトとプロセスの双対性の受け入れを急がせてもいる。...《省略》

人間は、プロダクトをつくり上げた達成感に劣らぬ満足感を創造的なプロセスを行うことによっても得ることができる。芸術家は、額縁に納まる作品の仕上がりを楽しむように筆の運びそのものも楽しむ。作家は、書き上げた本と同じようによい隠喩を探すことを楽しむ。創造的なソフトウェアの開発者も、できあがったプロダクトと同じようにプロセスからも満足を得るべきである。ソフトウェアエンジニアリングに従事する創造的な人々が成長し続けるためには、プロダクトとプロセスの双対性が重要である。

2.7 まとめ

ソフトウェアエンジニアリング向けのプロセスモデルは、プロセスフレームワーク、アクティビティ、アクション、タスクを包含したものである。プロセスフローによって、各プロセスモデル内のフレームワークアクティビティ、アクション、タスクがどのような順序、時系列で発生するかという体系の違いを表現できる。プロセスパターンは、ソフトウェアプロセスにおいてよく遭遇する問題を解決するために使われる。

規範的なソフトウェアプロセスモデルは、ソフトウェア開発に秩序と構造をもたらすことを目的として長年適用されてきた。これらのモデルが提案するプロセスフローは異なっているが、同一のフレームワークアクティビティ（コミュニケーション、計画策定、モデリング、構築、デプロイ）から構成される。

ウォーターフォールモデルのような逐次的なプロセスモデルは最も古いソフトウェアエンジニアリングのパラダイムである。リニアプロセスフローと呼ばれるこれらのフローは、継続的な変更要求、進化するシステム、厳しいスケジュール等、近年のソフトウェア開発における現実とは適合しない場合がある。それでも、要求がきちんと定義され変更の可能性が少ない状況においては、これらのプロセスフローは適用できる。

インクリメンタルプロセスモデルは、反復的な特性を持ち、動くソフトウェアの新しいバージョンを短時間で生み出していく。進化型プロセスモデルは、ソフトウェア開発プロジェクトの多くが反復的でインクリメンタルな性質をもつという認識から、変化に対応することを目的として設計された。プロトタイピングやス

パイラルモデルのような進化型モデルは、インクリメンタルに成果物（あるいは、実行可能なソフトウェア）を短期間で作成し続ける。これらのモデルは、コンセプト開発から長期にわたるシステム保守まであらゆるタイプのソフトウェアエンジニアリングアクティビティに適用可能である。

統一プロセスは、「ユースケース駆動、アーキテクチャ中心、反復型およびインクリメンタル」なソフトウェアプロセスであり、UML手法およびツールを活用する。

問題と考察のポイント

2.1 Baetjerが「プロセスのなかで、ユーザと設計者、ユーザと開発ツール、設計者と開発ツール（および技術）の間に相互関係が生まれる」と述べている[Bae98]。1〜4それぞれについて5つずつ具体的な質問を挙げよ。

1. 設計者がユーザに尋ねるべき質問
2. ユーザが設計者に尋ねるべき質問
3. ユーザが開発するソフトウェアプロダクトについて自問すること
4. 設計者が開発するプロダクトとそのプロセスについて自問すること

2.2 2.1節に示す各プロセスフローの違いを議論せよ。記述されているプロセスフローのいずれかを適用する際に起こりうる問題を特定せよ。

2.3 「コミュニケーション」のアクティビティにおける一連のアクションを示せ。提示したアクションの1つに対してタスクセットを作成せよ。

2.4 「コミュニケーション」におけるよくある問題は、ソフトウェアがすべきことに対して2名のステークホルダがそれぞれ対立したアイデアをもつ場合に発生する。つまり、相互に対立した要求を持っているのである。この問題を表現し効果的な解決アプローチとなるプロセスパターンを構築せよ。

2.5. ウォーターフォールモデルが適用可能だと考えられるソフトウェアプロジェクトの例を3つ挙げよ。具体的に示すこと。

2.6. プロトタイピングモデルが適用可能だと考えられるソフトウェアプロジェクトの例を3つ挙げよ。具体的に示すこと。

2.7. スパイラルプロセスのフローに従い、スパイラルの外側へと向かう際に、開発あるいは保守対象のソフトウェアはどのような状況であるか述べよ。

2.8. プロセスモデルを組合せることは可能か。モデル組合せが可能である場合には例を示せ。

2.9. ソフトウェアを「必要十分な」（good enough）品質で開発することの利点、欠点は何か。すなわち、開発スピードを品質よりも優先させる場合、どのようなことが起こるか。

2.10. ソフトウェアコンポーネント、あるいはプログラム全体でさえも、それが正しいことを証明できる。では、なぜ皆がそれを行わないのだろうか。

2.11. 統一プロセスとUMLは同じものか。その回答について説明せよ。