エクストリーム・プログラミング(XP)

❍ 歴史

XP は，1990 年代後半，Kent Beck を中心とするグループから発展してきた。Kent Beck は有名な Smalltalk プログラマで，XP は Smalltalk コミュニティが従来から育んできたプログラミング・プラクティスを発展させたものと考えることもできる。Kent Beck はまた，デザイン・パターンなどのオブジェクト指向の先進的な概念を発展させてきた Hillside グループの一員でもあり，XP のコミュニティとの間には強い関係がある。

❍ 特徴

- エンジニアリングを中心としている。
- 何をすべきかが非常に明確に定義されており，それを遵守することが強く求められる。

❍ 適用可能性

- プロジェクト規模が比較的小さい(初期段階で 10 人以下)。
- 要求が最初にあまり明確でなく，プロジェクト期間中に変更される可能性が高い。
- 明確な顧客が存在しており，プロジェクトに積極的に関与させることができる。
- 変更コストを下げることのできる開発環境/開発技術を持っている。

❍ 効果

- 要求/技術/状況などの変化への対応
- 高品質で創造的な製品
- 開発メンバの育成
- 高い生産効率

❍ 原則

■4 つの価値

XP では，コミュニケーション，単純さ，フィードバック，勇気に最大の価値を置く。

■5 つの基本原則

XP では，4 つの価値を 5 つの基本原則として具体化している。XP のプラクティスやプロセスは，この基本原則をもとに考えられている。XP ユーザが何かに迷ったり，複数の選択肢があった場合には，この基本原則に照らして選択すればよい。

- 素早いフィードバック
- 単純さの採用
- インクリメンタルな変更
- 変化を取り込む
- 質の高い作業

■その他の原則

基本原則より重要性は低いが，その他にもいくつかの原則がある。

- (結果だけではなく)学び方を教える
- (いきなり全部をやっつけるのではなく)最初は少しずつ始める
- (負けないための保守的な戦いではなく)勝ちに行く
- 具体的な実験を積み重ねる
- オープンで誠実にコミュニケーションをする
- 責任を引き受ける
- 自分たちに合わせてカスタマイズする
- 余計なものに時間を割かない
- 正直な測定

❍ 役割

XP では本来専門家を作らない。基本的には，開発チームのメンバ全員がプログラマである。その上でいくつかの役割を兼務する。また XP では，顧客も開発チームの一員と考えられ，大きな役割を担っている。

■プログラマ

プログラマはソフトウェアを開発する。そういう意味では他のどんな手法とも変わりはない。ただし，いわゆるコーディング，与えられた詳細な設計書をソース・コードに変換するのが仕事ではない。XP では，独立した分析者/設計者/ドキュメント作成者のような役割はない。あるユーザ機能を実現する上で必要なすべての責任をプログラマは引き受ける。

プログラマは，実現するべきユーザ機能(XP ではストーリと呼ぶ)を他のチーム・メンバと一緒に見積もる。次に，見積もったストーリを，より詳細な内部機能として実装するのに必要な作業(タスク)を考える。内部機能の実装は他のプログラマとペアを組んで行う。ペアは半日か 1 日ごとに適当に交代する。

実装作業においては，プログラマは，まず実装する内部機能に対するテスト・コードを少しずつ書くことから始める。テスト・コードを少し書いたら，今度はそのテスト・コードをパスするような実装コードを書き，実際にテストをパスすることを確認する。実装コードを書くうちに，コードに重複が出てきたり，特定の関数やメソッドが長くなりすぎたり，より適当な名前を思い付いたりしたら，それを修正し(リファクタリング)，再度テストをパスすることを確認する。このテスト－実装－リファクタリングのサイクルを十数分から数十分ごとに繰り返す。

実装したコードは，適当なタイミング(例えば毎日帰宅する前とか 2 時間ごととか)に，今までにチームが作成した他のコードと統合し，機能テストを実行する。他のコードとの整合性に問題があれば，そのコードを書いた人と相談して問題を解決する。統合に成功した実装コードは，テスト・コードとともに構成管理ツールにチェックインする。

実現したユーザ機能は，数週間ごとに顧客にデモンストレーションする。

■顧客

顧客は開発チームの重要な一員である。顧客の代表者，特に意思決定の能力と権限を持つ顧客の 1 人が開発チームと同じ場所にいることが求められる。これは，顧客に常に開発の状況や進捗を把握してもらうこと，実現するべきユーザ機能について不明瞭な点や複数の選択肢が生じた場合にできるだけ早い決定をもらうことを目的としている。

顧客に開発チームの開発現場に来てもらうこともあるし，逆に開発チームが顧客の所に行って開発を行う場合もある。いずれにしろ，開発チームと一緒にいる顧客は必ずしも 100%の時間を開発作業に割く必要はない。必要ない限りは，開発と関係のない自分の仕事をしていても構わない。

顧客には，上に挙げた，

- 常に開発の状況や進捗を把握する
- 不明点や選択肢に関する意思決定を即座に行う

以外にも多くの作業がある。

まず，顧客は開発チームと一緒に実現するべきユーザ機能をストーリの形にまとめる。そして予算，納期，品質などの制約と開発チームの提示した見積もりをもとに，実現するべきストーリに優先順位を付ける(計画ゲーム)。

顧客はまた，実現されたストーリの検収基準となるべき機能テストを作成する。顧客にテスト作成能力が不足している場合には，開発チームが機能テスト作成を手伝っても構わない。機能テストを通るかどうかが開発の完了基準となる。

顧客の要求の優先順位に変更が生じたり，新たな要求が生じたり，開発チームの見積もりに誤差が生じたりした場合には，開発の途中でも開発チームと一緒になって実現するべきストーリの洗い直しを行う。

■テスタ

テスタは，顧客が機能テストを作成するのを手伝ったり，テスト環境の整備を行ったり，テストが正しく適切に行われていることを確認したりする。テスタは，開発チームのメンバが適当に交代しながら兼務することが多い。

■トラッカ

トラッカは，プロジェクトの進捗を測定する役割である。ただし XP での測定は，開発の進捗に影響を及ぼさないように，できるだけ控えめに行うのが望ましい。測定は最低限必要な項目だけについて行い(通常 2〜3 項目程度)，測定の結果は開発チームの全員に公開する。

測定の指標は主にプロジェクト速度(見積もりと実績の比。28 ページ参照)と機能テストの達成率だが，その他にもプロジェクトが必要とする指標があれば，それについても測定を行って構わない。XP では数字至上主義は取らないが，憶測よりは現実的で客観的な指標を重視する。

■コーチ

コーチは開発チーム全体をまとめる。従来のプロジェクト管理でいえば，プロジェクト管理者に当たる役割である。コーチが従来のプロジェクト管理者と大きく異なるのは，管理者は，定義されたプロセスにメンバが従うように強制する役割であるのに対して，コーチは，メンバがよく仕事ができる環境を整え，メンバのやる気を引き出す役割である点である。

コーチはミーティングの場を作り，ミーティングをリードし，開発チームの作業場所の環境をできるだけ良くし，開発の適切なリズムを作り出し，開発の区切りにはメンバをねぎらい，メンバ間や顧客との人間関係を良好に保つようにする。

コーチは必ずしも技術的な側面に精通している必要はないし，メンバの技術上の問題に介入しすぎない方がよい。しかし，何らかの技術的なバックグランドを持っていたほうがコーチとしてうまく機能することが多いのも確かである。

■その他の役割

XP では，特に役割を固定して考えない。XP で重要な役割分担は，ビジネス上の価値を決定する顧客と技術上の見積もりを行うプログラマの 2 つである。その他の役割は補助的なものと考えてもよい。上記以外にも，プロジェクトにとってどうしても必要な役割があれば導入して構わない。

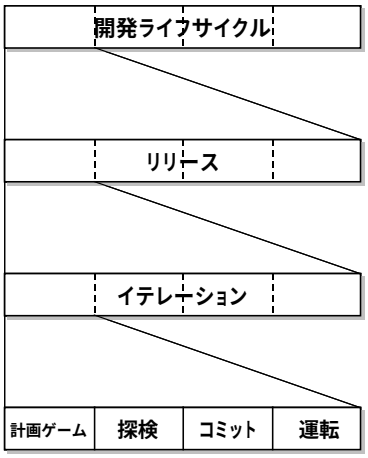
例えば，技術的な詳細に関する知識や経験が不足しているならばコンサルタントを導入する。特にオブジェクト技術や XP のような開発プロセスに関して開発チームに

経験者が 1 人もいない場合には，開発チームと一緒に働いてくれるメンタと呼ばれる経験者を加えると大きな効果を生み出すことができるだろう。

❍　プロセス

XP には，マクロ・プロセスとマイクロ・プロセスという 2 つのレベルのプロセスが存在する。マクロプロセスは数週間から数カ月にわたり，多くの利害関係者を巻き込むプロセスである。一方マイクロ・プロセスは，数時間から数日にわたり，個人あるいは数人程度がかかわるプロセスである。マクロ・プロセスはマイクロ・プロセスの繰り返しによって構成される。

■マクロ・プロセス



XP におけるマクロ・プロセスは，1 回数週間のイテレーションと，数回のイテレーションから成る 1 回数カ月のリリースが基本となる。

1 回のイテレーションは複数のユーザ機能を実現し，その成果を顧客にデモンストレーションしてフィードバックを得る単位である。1 回のイテレーションには，数週間(例えば 1〜4 週間程度)を割り当てることが多い。リリースは，数回のイテレーションで実現されたユーザ機能をユーザに引き渡す単位である。1 回のリリースに数カ月(例えば 1〜3 カ月程度)を割り当てることが多い。イテレーションの期間もリリース

の期間も，短すぎると十分なユーザ機能を実現することができないし，長すぎるとリスクが大きくなる。実際の期間はプロジェクトの状態や製品の種類によって変わってくる。

1 回のイテレーションあるいはリリースに先立って，顧客と開発者のチームは計画ゲームを行う(「プラクティス」の項の「計画ゲーム」を参照)。計画ゲームの結果，このイテレーションあるいはリリースで実現すべきユーザ機能(ストーリ)が選択される。ストーリの見積もり上の期間の合計は，イテレーションあるいはリリースの期間以下とし，顧客にとって価値の高いものから優先的に選択する。

選択したストーリは探検/コミット/運転の 3 つのフェーズを経て実現される。探検フェーズでは，開発チームはストーリをどのようにすれば実現できるかを調査し，タスク(内部機能を実現するに当たって必要な作業)に分解する。1 つのタスクを 1 枚のカード(8cm×11cm 程度の B6 の情報カードを使うことが多い)に書き出す。このカードをタスク・カードという。

場合によっては，ストーリと直接関係のないタスク(例えば構成管理ツールのインストールなど)もあり得る。探検に当たっては，利用できるリソースはないかをインターネット上で調べたり，プロトタイプを作ったり，マニュアルや論文を調査したり，適当な人を捜してインタビューすることもある。探検フェーズは，開発チーム全体で適当に分担しながら作業を行うのが普通である。

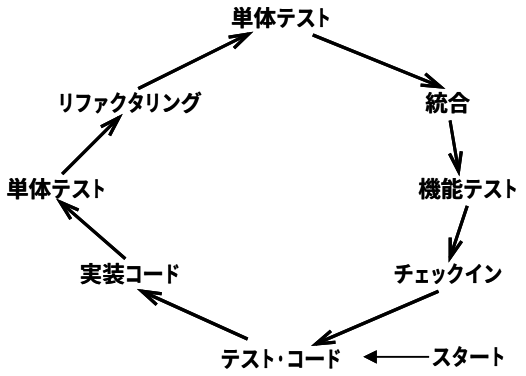
コミット・フェーズでは，タスクを個々のプログラマに割り当てる。タスクを割り当てられたプログラマは，タスクを実行するのにかかる時間を見積もる。最初は理想日(1 日を 100%作業し続けたとしてできる作業量)で見積もり，次に，理想日の作業を行うのに現実には何日かかるかを設定する。不明の場合には 2〜3 日とするのがいいだろう。プログラマ間でタスクのバランスができるだけとれるように，また，1 つのストーリに関連するタスクはできるだけ同じプログラマが担当できるように配分する。1 つのタスクは 1〜数日程度で実現できる規模が望ましく，大きすぎる場合には分割する。

運転フェーズでは，コミット・フェーズでのコミットメントに基づいて，個々のプログラマがタスクを実行していく。進捗の度合いは，トラッカが 2〜3 日ごとに調べる。コミットメントが守られそうにない場合には，タスクやストーリを変更したり，減らしたりする。ストーリを変更したり，追加/削除したりする必要がある場合には，再度顧客との計画ゲームを行う。

進捗の度合いの測定の基本になるのは，タスクやストーリの見積もり時間(理想日)と実際にかかった時間との比率である。この比率のことをプロジェクト速度と呼ぶことがある。プロジェクト速度は，一理想日の仕事をこなすのに現実にどれだけの日数がかかったかを表している。最初はプロジェクト速度を 2〜3 とすることが多いが，イテレーションやリリースを重ねるにつれてプロジェクト速度が一定の値に近づいてくれば，それを見積もりに反映させることができる。

進捗は，機能テストのうちどれだけをパスし，どれだけ残っているかによって測定することもできる。その他に，プロジェクト独自の測定を行っても構わない。重要なのは測定しすぎないことである。たいていの場合，十分に吟味した 2〜3 の指標について測定すればよく，それ以上の指標を測定しようとすれば，あまり意味のある測定が行われないばかりか，プロジェクトの進捗に影響を及ぼす場合もある。

■マイクロ・プロセス



一方マイクロ・プロセスは，運転フェーズにおいてプログラマが行うタスク実行の作業である。従来のソフトウェア開発においては「文書化→設計→実装→単体テスト→統合→機能テスト」の順に作業を行うが，XP では，逆に「単体テスト→実装→設計(リファクタリング)→統合→機能テスト→(必要ならば)文書化」の順に作業を行う。しかも，従来の開発ではこのサイクルに数週間から数カ月かけるのに対して，XP ではこのサイクルを十数分から数十分程度で繰り返す。

まず，プログラマはタスクとして分割された内部機能を取り上げて，その内部機能を実行したときに期待される振る舞いとその結果をテスト・コードとして記述する。テストは，できるだけ自動的に行われるようにする。テストを書いて実行することがプログラマの負担になるようであれば，そのうちにテストが実行されなくなったり，テスト内容がいい加減なものになりがちだからである。

単体テストの自動化ツールとして最も広く使われているのが，xUnit と呼ばれる一連のツール群である。そのうち JUnit は Java 用の単体テスト自動化ツールで，XP の提唱者である Kent Beck らが作成し，オープンソースソフトウェアとして配付している(http://www.junit.org/)。

テスト・コードの書き方はテスト自動化ツールにも依存するが，何をテストするかについては特別な点はない。引数のチェック，正常ケース，例外値/異常値/境界値のケース，スモーク・テストなどを順に記述する。一度にすべてのテストを書く必要はない。テスト・コードは常にテスト対象コードと一緒に保存して，いつでもすぐに実行できるようにしておく。

ひとまとまりのテスト・コード(実際には数行〜十数行程度)を書いたら，実行してみる。当然，この時点ではテスト対象コードが存在しないためにテスト・コードを正常に実行できないので，そのテスト・コードが実行できるようにテスト対象コードを書く。順次，様々な側面からのテスト・コードを追加していき，それに対応したテスト対象コードを追加していく。テスト対象コードは，常にテストを通るものでなければならない。また，この時点でのテスト対象コード記述の基準は，「テストをパスする最も単純なコードを書く」ことである。

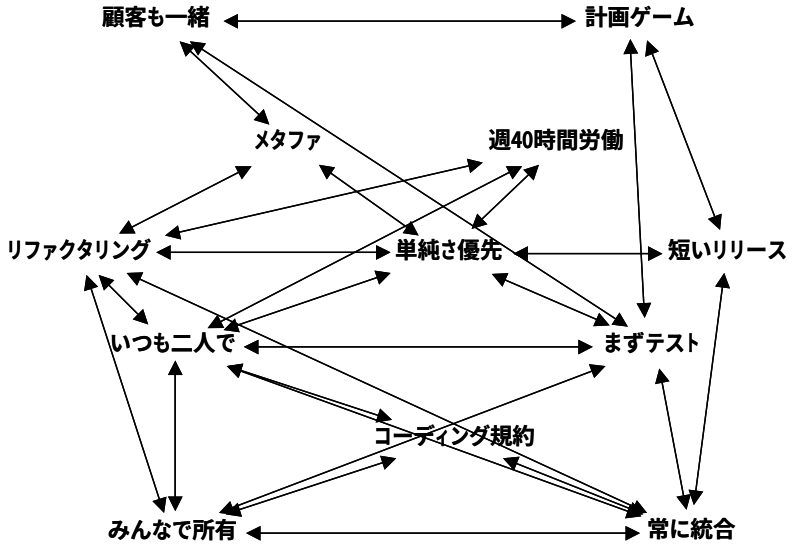
ある程度実装コードがたまってくると，コードの重複や肥大化，不適切な命名などコード品質の問題が見えてくるはずだ。そうしたらそれを修正する。この作業をリファクタリングと呼ぶが，従来のソフトウェア開発での詳細設計に相当する作業を実装コード上で行っているものと考えられる。リファクタリングを行ったら，即座にテストを実行して，リファクタリングによって間違いを入れ込んでいないことを確認する。

ある単位(例えばクラス)で実装コードが揃ったら，今までにプロジェクトで作成された他の実装コードと統合する。統合は，ant や make などのビルド・ツールを用いて自動化しておくのが普通である。統合と同時に機能テストを実行する。統合の完了したテスト・コードと実装コードは，構成管理ツールにチェックインする。

❍ プラクティス

XP では，以下の 12 のプラクティスを定義している。ただし，これらのプラクティスは固定的なものでも強制的なものでもない。XP を開始するのに最適と考えられる一群のプラクティスが，この 12 のプラクティスだと考えたほうがいいだろう。実際，提唱者の 1 人である Kent Beck も，年々いくつかのプラクティスを追加している。

プラクティスの間には，以下の図のような相関関係がある。つまり，あるプラクティスの実行には他のプラクティスの存在を前提している場合がある(例えば「リファクタリング」は「まずテスト」を前提としている)。したがって，プラクティスを加減する場合には注意が必要である。



(「XP エクストリーム・プログラミング入門 ― ソフトウェア開発の究極の手法」，ケント・ベック 著，長瀬嘉秀 監修，永田渉，飯塚麻理香 訳，ピアソン・エデュケーション，2000。日本語訳より許可を得て転載)

■計画ゲーム

計画ゲームとは，顧客と開発者の間で要求事項を確立するために行われる交渉のことである。計画ゲームでは，リリースやイテレーションで実現するストーリをストーリ・カードの形で決定する。

顧客には，自分にとって何が価値が高いストーリであるかを決める権利がある。一方，開発者には，そのストーリをどのように実現するか，そのストーリを実現するのにどれだけの時間がかかるかを決める権利がある。計画ゲームとは，両者がこれらの権利を使ってそれぞれ自分の取り分を最大にする(顧客は一定の時間内にできるだけ高い価値のストーリを実現してもらう，開発者はできるだけ自分たちのやりやすいやり方でストーリを実現する)ゲームである。

計画ゲームはたいていの場合，一定の日時に顧客のチームと開発者のチームが揃って行う。半日程度で終わる場合もあるし，開発者が問題領域の知識を得るために複数回にわたって行う場合もある。

顧客側は，実現したい機能を 1 枚のカード(8cm×11cm 程度の B6 の情報カードを使うことが多い)に 1 つのストーリの形で書く。これをストーリ・カードと呼ぶ。顧客側がストーリを書くのに慣れていない場合には，開発者がそれを手伝う。ストーリとは，顧客が実現したい機能を自然言語で平易に説明したものである。ストーリの名前の通り，その機能をユーザから見た場合の流れを表現する。

開発者側は，顧客の書いたストーリの 1 つ 1 つに対して，その実現にかかる時間を見積もってストーリ・カードに書く。見積もりは暦日(日，週，月など)で行う。見積もりは正確であるに越したことはないが，完全に正しいものであることを前提とはしていない。見積もりの誤差が大きくなってきたときには，計画ゲームをやり直す。

顧客側は，見積もりの書かれたストーリ・カードを優先度の順に並べる。そして，見積もり時間の和が，今回のリリースあるいはイテレーションの時間に達するまでのストーリを今回の実現対象ストーリとする。顧客の都合や状況の変化で優先順が変わったり，ストーリを追加/削除したくなったときには，計画ゲームをやり直す。

■短いリリース

XP では，顧客からのフィードバックをできるだけ早く，できるだけたくさん受け取るために製品を短期間でリリースする。ここで「リリース」とは，すべてではなくてもいくつかのストーリが正しく動作するソフトウェアを顧客に引き渡すことである。1 回のリリースまでの期間は，数カ月程度(例えば 1 カ月から 3 カ月程度)が適切である。1 つの製品を開発するために，必要なら複数回のリリースを繰り返す。

■メタファ

XP では，システムのアーキテクチャをメタファとして表す。通常のソフトウェア開発で使われるような厳密なアーキテクチャ定義は行わない。

XP で使われるメタファとは，文字通り「比喩」である。例えば，クライアント・サーバというアーキテクチャも，もともとはお客と給仕というメタファである。このような既存のメタファを使ってもいいし，開発メンバがディスカッションをして新しいメタファを考えてもよい。これから作ろうとしているソフトウェアをどういう形でプログラミング言語で表すかについて，開発メンバ全員が共通のイメージや理解を持てるようにすることが，このプラクティスの目的である。

多くの場合，メタファは図や絵などに表し，そのコピーを開発メンバ全員に配ったり，全員が見えるところに貼っておく。メタファは開発が進むにつれて変わったり，詳細化されていく場合もある。

■単純さ優先

XP では, 実装上可能な複数の選択肢がある場合には, 最も単純な実装を優先する。必要以上に複雑な実装は，品質低下のもととなるからである。単純なものを必要に応じて複雑にしていくことは簡単だが，いったん複雑に作ってしまったものを単純な形に戻すのは非常に難しい。

従来の開発では，将来の変更や機能追加を見越して，現在は必要のない複雑なコードを組み込んでいた。これは，コードの変更は非常にコストが高くかかるという前提のもとに，できるだけコードを変更せずに済ませるためのプラクティスであった。しかし，コードの変更にはさほどコストがかからないという前提があれば，現在必要のない複雑なコードの組み込みはむしろリスクとなり得る。また本来，単純さ優先は，

すべてのエンジニアリングに共通する原則のはずである。XP では，変更コストはさほど高くない(あるいはそうできる)という前提に基づいている。そして実際，それが可能な技術は揃ってきている。

■まずテスト

XP では，テストは最後のやっつけ仕事ではなく，すべての作業の前提となるものである。XP では大きく分けて 2 種類のテストがある。単体テストと機能テストである。

機能テストは，ストーリごとに作成する。ストーリ・カードを受け入れたら，必ずそれに対応する機能テストをできるだけ早く作成し，開発中は常に実行する。機能テストを作成するのは原則的に顧客側であるが，実際には顧客にテスト・プログラムを作成する能力がないことが多いので，必要に応じて開発者側が協力する。XP では，これが検収テストとしても用いられる。

機能テストを実行するのは納品直前だけではない。開発中「常に」(新しいコードを統合するたびに)実行する。もちろん,最初の頃はほとんどの機能テストはパスしない。リリースあるいはイテレーションの目的は，機能テストのパス率を 100%に近づけることである。そのために，機能テストのパス率を自動的に追跡できるようにするとよい。

一方，単体テストは，原則的にすべてのコードに対してそのコードを作成する前に個々の開発者が作成する。具体的には，開発者は次のようなステップを踏む。

① 作業すべきタスク(ここではプログラミング)を 1 つ選択する。
② そのプログラムに必要なクラスや関数をメタファに従って考える。
③ そのクラスや関数を呼び出したときに期待される動作を単体テストとして記述する。
④ 単体テストを実行する(テスト対象がまだ存在しないのだから当然失敗する)。
⑤ 単体テストをパスするようにテスト対象コードを書く。
⑥ 必要に応じて単体テストを追加して，同じことを繰り返す。

単体テストの単位は，クラス，メソッド，関数などである。単体テストの種類は，正常ケースのテスト，例外や境界値のテスト，スモーク・テスト(限界を確かめる)な

ど通常の単体テストと同じである。XP に特徴的なのは，テスト対象コードを書く前に単体テストを書くことと，単体テストは常にテスト対象コードと一緒に保存して，テスト対象コードを変更するたびに(できるだけ自動的に)実行することである。これによってコードに対する変更が他に影響を及ぼしていないことを常に確かめることができ，最終的には変更コストの低減，高品質な製品につながる。

別の面から見ると，単体テストは，テスト対象コードの仕様や使い方をすぐに実行可能な形で表しているものと考えることができる。XP では，特に必要のない限り詳細設計文書を作成しないが，単体テストがその代わりの役を果たしている。意味が明確に決まること，常にコードと同期していることなど，通常の自然言語で書かれた詳細設計文書よりも優れている点が多い。

XP では，「テストしていないコードは存在していないのと同じ」と言われる。バグのあるコードを利用しようとして悩むプログラマがいることを考えると，存在していないよりも質が悪い。

■リファクタリング

リファクタリングとは，ソフトウェアの機能を変えずにコードの品質を上げることである。

XP では，最初に単体テストにパスすることを目的とした単純さ優先の実装コードを書く。しかしこれだけに頼っていると，コードは次第に乱雑になっていき，見通しも悪くなってくる。また，XP ではストーリごとにタスクに分割していくが，このようにトップダウンの実装を行っていると，重複の多い悪いアーキテクチャになってしまいやすい。これを避けるために実装コードの上で常に設計し続けることをリファクタリング[1]という(デザイニングとも言われる)。

このようなリファクタリング対象コードを見つけたら, その場でコードを修正する。直接のリファクタリング対象でなくても，より良い実装方法を見つけたら，それをすぐにコードに反映させる。リファクタリングが可能なのは，「まずテスト」のプラクティスに基づいて常にテストが行われているので，もし間違った修正をしたとしてもそれをすぐに発見できるからである。

---

[1] リファクタリングの詳細については，“ リファクタリング―プログラムの体質改善のテクニック ”，マーチン・ファウラー，ピアソンエデュケーション，2000 をご覧いただきたい

実際には，リファクタリングは，

① 単体テストを書く。
② テスト対象コードを書き，テストする。
③ リファクタリングし，テストする。

というサイクルに則(のっと)って，数十分から数時間単位で行う。つまり，四六時中リファクタリングしている感じに近い。また「みんなで所有(後述)」のプラクティスに基づいて，他人の書いたコードであってもリファクタリングして構わない。

また時には，複数のメンバの担当部分やプロジェクト全体にまたがる大規模なリファクタリング(アーキテクチャの変更など)が必要になることもある。このような場合には,個々のメンバの作業を一定期間中止して,大規模なリファクタリングを行う。

■いつも 2 人で

XP では，コーディングは常に 2 人 1 組で 1 台のマシンに向かって行う。これをペア・プログラミングという。

ペア･プログラミングには多くの効用がある。1 つは開発メンバが常に刺激し合い，アイデアをぶつけ合いながら,多様な視点をもって作業できることである(これには体力を消耗しやすいというマイナス面もある)。2 つ目は，作業の進捗，問題点，他のメンバの能力，他のメンバの担当部分などについてメンバ全員が共通の認識を持てることである。3 つ目に，ペア・プログラミングが絶好の教育，技術移転の場となり得ることである。最後に，すべてのコードが常に複数の開発メンバによってレビューされていることである。

多くの場合，ペアは 1 日の最初に適当に組む。相手を指名してもいいし，慣れないうちは誰かが決めても構わない。ペアは半日から 1 日程度一緒に作業する。同じペアで何日も作業を続けず，できるだけ多くのメンバとペアを組むことが望ましい。通常は，ペアのうち一方がコードを書き，もう一方が口を挟んだり調べものをしたりすることになるが，この役割は適当なタイミングで交代する。ペアを組みたがらないメンバには無理強いをせず，少しずつ慣れていってもらうのがよい。

ペアの 2 人ともがあまり能力が高くない場合にも，ペア・プログラミングはそれなりの効果を発揮する。むしろペア・プログラミングで重要なのは，2 人が積極的にコーディングに参加することである。一方が受け身になったり，傍観者になってしまうと効果はあまりない。また逆に，本質ではないところで議論にはまってしまっても効果はない。そのような場合には，このようなペアを解消してペア・プログラミングに慣れたメンバと組ませるなどする。

ペア・プログラミングを行うと開発効率が半減するように思われるが，実際にはそんなことはない。ある調査では，

- 開発時間が 15%増加
- 設計品質の向上(ソース・コード行で 10〜30%減少)
- 欠陥が 15%減少
- 再作業が 1/15 に減少

という結果が出ている(“The Cost and Benefits of Pair Programming” by A. Cockburn and L. Williams，2000)。

■みんなで所有

XP では，実装コードの所有者を特に決めない。つまり，誰が書いたコードであろうとも，誰が変更(デバッグ，リファクタリング，機能追加など)しても構わない。所有者を決めてしまうと，コードにとって必要な変更が行われにくくなり，変更した場合にも変更に要するコストや時間が増加してしまうからである。

従来のソフトウェア開発では，コードに対する責任を持たせ，コードを無秩序に変更されないようにするために所有者を決め，所有者以外はコードを変更できないようにすることが多い。しかし XP では，「いつも 2 人で」や「まずテスト」などのプラクティスがあるのでそのような心配をする必要はないのである。もちろん，ちゃんとした構成/変更管理は必要になるが，現在の技術ならばそれらはツールに任せることができる。

■常に統合

XP では，統合はリリース前に 1 回だけ行うのではなく，常に(ある程度まとまってコードを書くたびに)行う。ここで統合とは，製品全体のコード群に新しいコードを追加することである。新しく追加するコードは，統合以前にそのコードの単体テストを完全にパスしていなければならない。また，統合と同時に製品全体のビルド，機能テストの実行を行い，出荷可能な形にしておく。

通常，統合のプロセスは ant/make/シェルなどのスクリプトとして記述し，それに基づいて自動的に行われるようにしておく。継続して統合されていれば，統合にかかるコストを全体として下げることができ，どの時点でも必要ならば顧客にリリースすることが可能になる。

■週 40 時間労働

XP では，基本的に残業を認めない。特に連続した残業は厳禁である。残業を続ければ，プログラマの能力は低下するからである。残業を続けても品質は向上しないし，実際には納期も短縮できないことが明らかになっている。残業せざるを得ないような状況に陥ったら，早めに顧客と交渉して計画ゲームを再度行う。

■顧客も一緒

XP では，開発チームは常に顧客と一緒に作業をする。特に，決定権のある顧客の代表にできるだけいつも開発チームの作業場所の近くにいてもらう(決定権さえあれば，いつも同じ人でなくても構わない)。これを顧客同室(on-site customer)という。

顧客の作業はいろいろある。最初はストーリ・カードを書くこと，計画ゲームをすることである。その後も，機能テストを作ること，開発者の質問に答え不明確な仕様を決定すること，開発チームの作業を見守る(監視する)ことが必要になる。特に後者の作業をするために，決定権のある顧客の代表が開発チームと同じ場所に常駐する。ただし，これらの作業は常にあるわけではないので，その他の時間は顧客が自分の仕事をしていても構わない。

XP では，ストーリごとの作業時間を開発者が見積もり，見積もりに誤差があった場合には計画ゲームをやり直す。したがって，顧客にとっては見積もりの誤差が開発者の怠慢によって生じたものでないことを何らかの形で担保する必要がある。顧客と

開発者の信頼関係がいったん成立するまでは，顧客同室のプラクティスがそれを担っている。

実際には，顧客が自分の作業場所を離れて開発チームの近くに常駐するのは難しい場合も多い。このような場合には，開発チーム全体が顧客の作業場所に移動して開発することもある。また，毎日ではなくて週に 2, 3 日だけ顧客同室を実施するのでも構わない。自社製品の開発のように顧客が明確でなかったり，どうしても顧客同室が無理な場合には，開発チームの中あるいはその周辺から 1 人を顧客代理として扱うこともある。

■コーディング規約

XP では，コーディング規約を決めて遵守する。もっとも，このプラクティスは XP 特有のものではなく, 多くのプロジェクトで行われている(べき)プラクティスである。XP では特に「いつも 2 人で」や「みんなで所有」「リファクタリング」のプラクティスのために，コーディング規約の遵守が必須となる。場合によっては，エディタなど開発ツールをメンバ間で統一することもある。

❍　参考サイト・参考文献

http://c2.com/cgi/wiki?ExtremeProgrammingRoadmap(Ward Cunningham)
http://www.xprogramming.com/(Ron Jeffries)
http://www.extremeprogramming.org/(Don Wells)
“XP エクストリーム・プログラミング入門 ― ソフトウェア開発の究極の手法”, ケント・ベック 著, 長瀬嘉秀 監修, 永田渉, 飯塚麻理香 訳, ピアソン・エデュケーション, 2000
その他邦訳書多数