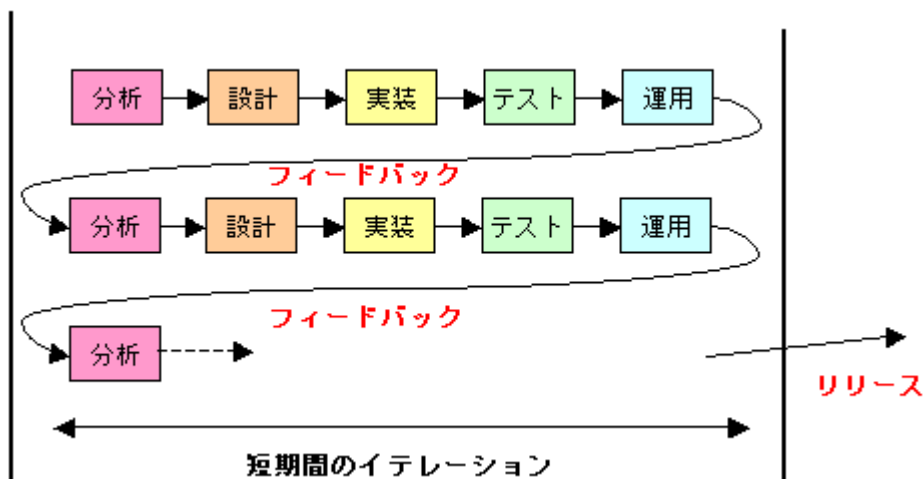


# これは？

「アジャイル開発」を「ゲーム製作」に適用する方法をまとめたものです。

## アジャイル開発とは



「反復型開発」の手法です。

独立したプロジェクト（イテレーション）を連続して繰り返すことにより、ソフトウェアのライフサイクル全体を構築するものです。

ウォーターフォールのように、

「一度決めた」ガチガチの仕様で最後まで突っ走るのではなく、

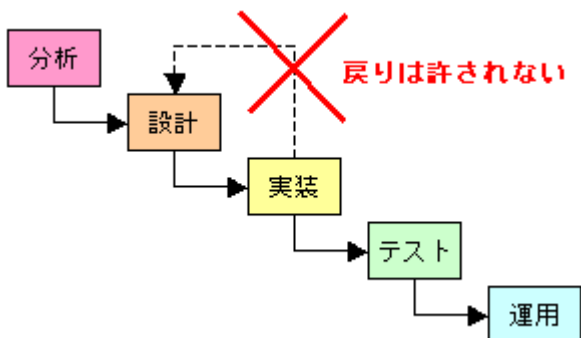
1. 仕様を決める
2. 開発
3. 評価

という過程を「短期間に繰り返す」開発手法です。

（仕様は作りながらかんがえる）

---

参考 「ウォーターフォール・モデル」



システム全体を一括して管理し、

1. 分析
2. 設計
3. 実装
4. テスト
5. 運用

の順に開発を行う手法。  
前の工程への戻りが起こらないように、綿密にチェックを行い、  
水が滝を流れ落ちるように開発を進めていくことからこの名前がついた。

まあ、この通りに実行できれば、かなり理想的なのですが、  
特にゲーム開発では、「面白くなければ仕様をひっくり返す」ことは当たり前なので、  
ウォーターフォール・モデルはあまりなじまないです、。

---

まあ、個人でのゲーム製作も、試行錯誤を繰り返して、仕様が二転三転するので、  
ある意味「反復型開発」といえるかもしれません。

ただ、個人で闇雲に開発を進めると、

- 計画性
- 方針
- 設計
- 優先順位

などが曖昧になってしまい、方向性を見失ってしまいがちです。

それを解決する1つの方法がアジャイル開発、といえます。

## タスクリストの作成

タスク（やるべきこと）を表に書き出します。  
そして「優先順位」を決めて、タスクを消化していきます。

例えば、こんな感じです

### **ID 状態 優先度 概要**

- 1 Fixed Middle Luaの組み込み
- 2 Started Critical ボスが死ぬと止まる
- 3 New Low 背景をうねうね動かす
- 4 New Middle チュートリアルレベルの作成

こういった一覧を作成しておく、やるべき作業が明確になり、  
また、作業が終わった後も、  
「うーん、こんだけのタスクを終わらせたのか、。オレって凄い！！」  
とか、自己満足に浸ることができます（w

まあ、方法は色々あります。

- 紙に書き出す
- Excel
- Trac

など、好きな方法、やりやすい方法がいいと思います。

kenmo的にオススメなのは、Tracなどのトラッキングシステムです。  
これは、タスクを登録するだけで、タスクの一覧を自動で作成してくれるので、  
タスクの管理を自動化できて、超便利です。

## タスクの属性

タスクには以下の属性を持たせると良いかもしれません。

### 1.見積もり期間

実装に必要な期間。

### 2.ステータス

よくあるトラッキングのシステムでは、問題発覚、作業仕掛中の事項を「Open」のステータス、終了事項を「Closed」のステータスとといいます。

- Openステータス
  - New : 問題について、初回レビューをまだしていない
  - Accepted : 問題が再現した / 問題対応への必要性が認知された
  - Started : 問題への対応が開始した
- Closedステータス
  - Fixed : 問題に対する対処が完了した
  - Verified : 検証が完了した
  - Invalid : 報告に誤りがあった
  - Duplicate : 報告が他のものと重複する
  - WontFix : この問題には対処しない
  - WorksForMe : 再現性なし

### 3.優先順位

- Low : 優先度低。次期リリース対応など。
- Middle : 優先度中。普通のタスク
- High : 優先度高。他のタスクに影響があったりするので早めにやっておいたほうがいいもの
- Critical : 最優先事項。アプリ強制終了など致命的なもの。

など

### 4.コンポーネント

- タイトル画面など、シーンによる分類。
- スクリプト
- レベル作成
- システム

などで、機能を分割します。

## 狩野分析法による優先度付け

「狩野分析法による優先度付け」という面白い手法があるので紹介しておきます。

- <http://d.hatena.ne.jp/masayang/20071213/1197534511>

# リスク駆動とクライアント駆動の反復型開発

タスクの優先度の付け方です。

無計画に開発を進めると、  
やりたいこと基準で開発を進めてしまい、  
全体のボリュームがおかしくなってしまいます。

## リスク駆動

リスクが高いものを優先度「高」とする方法です。

例えば、  
「トークン同士の相互作用（キャラ同士の衝突判定、それによるゲームシステムへの影響）」  
は、  
早い段階で決めておかないと、  
ゲームとして致命的な問題となりかねません。  
逆に、「演出関連」は後回しでも、問題となる可能性は低いです。

この場合、「トークン同士の相互作用」が、より優先度が高い、といえます。

## クライアント駆動

作る価値が高いものを優先度「高」とする方法です。

例えば、  
「パーティクルがモリモリ出るのがこのゲームの肝だ！」  
というのであれば、  
パーティクル実装の優先度が高いと言えます。

## リリース計画

ある程度開発がすすんだら、  
積極的に友達やネット上に公開して、  
ゲームの「評価」をしてもらいます。

アピールポイントを明確にすると  
「評価」しやすくなります。

そして、その結果をもとに、再分析を行い、  
次のリリースに向けて開発を進めます。

## 初回リリースに必要なもの

初回リリースには、以下の項目の実装があるのといいかな、と思います。

- 主要なトークンの実装
- チュートリアルレベルの作成

トークンの相互作用がハッキリしていないと、ゲームの意図を伝えるのが困難です。

また、「レベル」が存在しないと、  
どういう遊び方をしてもらいたいのか、  
ということが不明瞭で、評価がしづらくなります。

## エクストリーム・プログラミング

ケント・ベックらによって定式化され、提唱されているソフトウェア開発手法です。

### 計画ゲーム

次期リリースで実装する機能を決める方法です。

以下のどちらかの方法を用います。

1. 「リリース日」を基準に決める
  1. リリース日の決定
  2. 優先順位の高い順に並べ替え
  3. 見積もり期間がリリース日に達するまで機能を追加
2. 実装する「機能」から決める
  1. 次のリリースまでに必要な機能を選択
  2. 見積もり時間の合計を算出
  3. リリース日を割り出す

「リリース日」か「実装すべき機能」のどちらかを明確にします。

### シンプルな設計

なんでもできる汎用的なコンポーネント、  
というのは、開発者として非常に魅力的ですが、  
それよりも機能を必要最小限に押さえたほうが、  
機能の目的が明確になり、  
分かりやすい設計になります。

方針としては、

- 将来の変更の可能性を推測して設計しない
- 今すぐ必要でない汎用的なコンポーネントを作成しない
- コードの重複をなくす
- クラスやメソッドを比較的少なく抑える
- 簡単に理解できる設計

という感じです。

汎用的な仕組みを作るのは悪いことではありませんが、  
すぐに必要！ というのでなければ、  
後回しにしたほうが、モチベーションの低下を防ぐことができます。

## 頻繁なリファクタリング

機能の働きはそのままに、コードを簡潔な記述に置き換えることです。

面倒ですが、定期的にコード全体が無駄な処理がないかをチェックし、きれいなコードを保つことにより、

- ゲームの仕様を把握しやすくなる
- バグが見つかる
- 機能の追加がやりやすくなる

などのメリットがあります。

具体的な、リファクタリングの手法としては、

- 重複したコードを、サブルーチン化する
- 長すぎるメソッドを、サブルーチン化する
- 巨大なクラスを分割する
- 多すぎる引数をオブジェクト化する

などがあります。

## コーディング規約

首尾一貫したコーディングスタイルにすると、昔々に書いたコードでも分かりやすいものとなります。

- クラス / 変数 / 関数などの命名規約
- スペース / インデントの入れ方
- コメント / コメントヘッダの記述方法

などなど。

見やすいコードだと、精神衛生上、非常によろしく、開発のモチベーションを上げる効果があります。

(汚いコードはそれだけでやる気を削ぐことになってしまいます、(・`´) ;

## 参考リンク

2007年のGDCでも話題になってたらしいのでリンクしておきます。

- [\[ GDC07 # 38 \] Agile型開発でのゲームデザイン](#)

イテレーションとかスクラムについて簡単な説明があります。

- [\[ GDC07 # 21 \] 「Gears of War」はいかにして生まれたのか。Cliff Bleszinski氏が語る、有効なゲーム開発プロセス](#)

アジャイル開発を「Gears of War」へ適用した事例です。

手前ミソですが、kenmoがプログラムした「菜月さん ブースト2」では、アジャイル開発をほんのり導入してみました(・`´) ;

