ソフトウェア開発のこれまでとこれから

2013.04.13 Hybitz

アジェンダ

内容

- これまでの開発プロセス
- これからの開発プロセス
- デモアプリ紹介
- ちょこっと実演
  - テスト
  - デプロイ

これまでの開発プロセス

- 要件定義
- 工数見積
- 設計
- 実装
- テスト
- 納品

これまでの要件定義

- ユーザ自身が、何を必要としているかあやふや
  ・ソフトウェアの完成イメージを美化して想像
  ・カットオーバー直前で思っていたのと違うと判明

これまでの工数見積

- FP 数やステップ数などで見積＋経験による係数
  ・実際の生産性を反映できない
- 正確に見積もるためにじゃっかん設計に着手
  ・失注したら見積もる工数分が損失

これまでの設計

- 机上で論理を構築
  - 延々あーでもないこーでもないの繰り返し
  - 業務知識・スキルがないと誤解・論理破綻をまねく
  - 設計書の見栄えが気になってしょっちゅう微調整

これまでの実装

- 設計の完了が遅れて開発に入れない
  ・見切り発車で設計できている箇所から着手
  →設計変更による手戻り

これまでのテスト

- 設計・開発の遅れによる工数の削減
  ・テスト仕様書のレビューを怠る
  →やってるテストの網羅性・信頼性が欠ける
  ・マスタメンテやバッチなどのバックエンドを軽視
  →　本番稼働後にマスタ不整合が発覚しデータ復旧

これまでの納品

- ユーザが初めて目にしたソフトウェアが想像と全然違う
  - ・必死で改修
    →デスマーチ突入
  - ・気がついたら設計書が現状にそぐわない
    →直す暇がないので放置もしくはうすっぺらく修正
    →保守の段階で何が正しいのかわからず機能的デグレ
- とにかく検収が欲しい
  - ・ユーザが気がつきにくい部分はほりぼて感満載
    →　保守＆瑕疵対応で吸収

これまでの開発プロセスのおさらい

- 上流工程の課題が下流工程に皺寄せとなる
  - ・工数の食いつぶし
  - ・とりあえず○○とあいまいに設計
  - ・誤解をまねく表現で書かれた仕様で混乱
    （逆に瑕疵を回避する手段に利用している？）
- 上流工程がぐだぐだな時のよくある傾向
  - ・プログラム知らない人がいる
  - ・そもそも詰将棋みたいに設計を見通すなんて無理？
  - ・下流工程の頃にはおさらばする
    →無責任になる

これからどうしていくか？

- 工数を適切に消費する
- 誤解をまねかないように仕様を書く
- プログラムを知らない人が設計しない
- 最初からすべてを設計してしまわない
- 責任を持つ

これからの開発プロセス

- 要件定義
- テスト
- 実装
- 設計
- 納品
- 工数見積

これからの要件定義

- すべての要件を洗い出すのではなく最低限の要件のみ
  ・進む方向が正しいかの確認も含めて期間を決めて開発

これからのテスト

- とにかくテストケースを書く
- 自動テストをフル活用

これからの実装

- テストを作成しているので既に開発対象を理解している
- 限られた期間内で区切りよくできるところまで実装
  - その期間でできない部分を無理に着手しない
  - 時間と品質を重視

これからの設計

- テスト＆実装を既に終えている
  - あるべき姿を思い描くのが容易
- 信頼できるだけのテストカバレッジがある
  - あるべき姿になるようにリファクタリングする

これからの納品

- テストは済んでいる
  ・完成できている部分までならいつでも納品可能
- さっさとユーザに見せてフィードバックをもらう

これからの工数見積

- 一定の期間をイテレーションして開発
  - １つのイテレーションでの生産性が実績としてわかる
  - 次のイテレーションの生産性を実績から予測可能
  - 残っている要件と正確な生産性から計算

これからホントにいけそうか？

- 工数を適切に消費する
  実際に動く物を作りながら設計するので工数使って物は無しという状況にはならない
- 誤解をまねかないように仕様を書く
  自動テストはコンピュータが実行するので最初から０ or １の世界に近い
- プログラムを知らない人が設計しない
  自動テストから書くのでプログラムを書くことになる
- 最初からすべてを設計してしまわない
  明確なミニマム要件から開発するのであいまいさが無い
- 責任を持つ
  同じ人がテストも開発もこなすのでやり逃げできない

これから具体的にはどうやっていく？

- ソフトウェア開発により適した方法論を採用する
  可能な範囲でウォーターフォールからアジャイルへ
- 適切なツールを選ぶ

例えば、こんな感じ

- スクラム開発
  →開発を一定期間のサイクルで行う
- Redmine
  ・バックログプラグイン
  →スクラム開発を支援
- Git
  ・ブランチの扱いが用意
- Cucumber
  ・テストすることが仕様書を作成することに
- RubyOnRails
  ・コーディング量が少なく、自然言語（英語）に近い
  →仕様書なくてもコードを見ればわかる範囲が増える
  → DSL をうまく使えばコードなんだけど仕様書のよう
- Jenkins
  ・ソースに変更が入れば自動テストを実行
  →レグレッションをすばやく検知