なんちゃって

XPの薦め

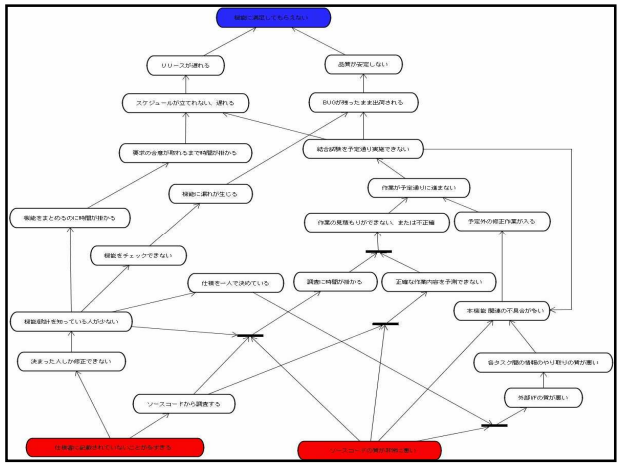
東京エレクトロン ソフトウェア・テクノロジーズ
出羽 弘明

ある日上司に言われました

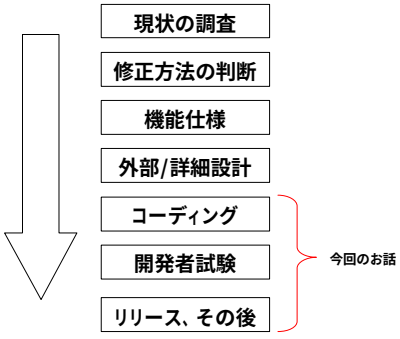
着手したときの状況

- 工数の1/3以上が不具合対処
- ソースをいじれば高確率でデグレ発生
- リーダー不在のチーム構成
  (一人っきりチーム！)

働けど、働けど我が暮らし楽にならず

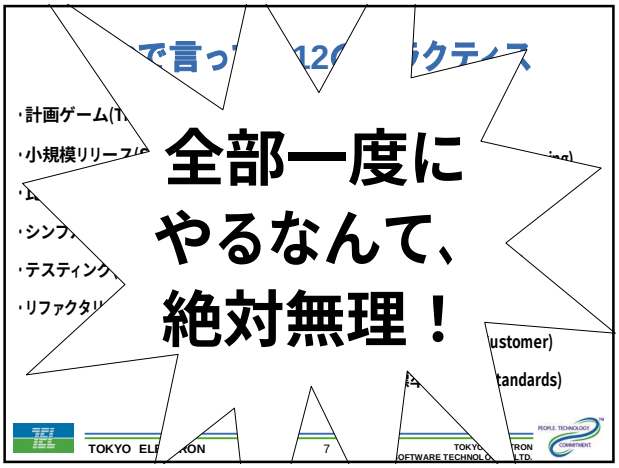


修正までの道のり



XPで言ってる
チームが共有すべき4つの価値

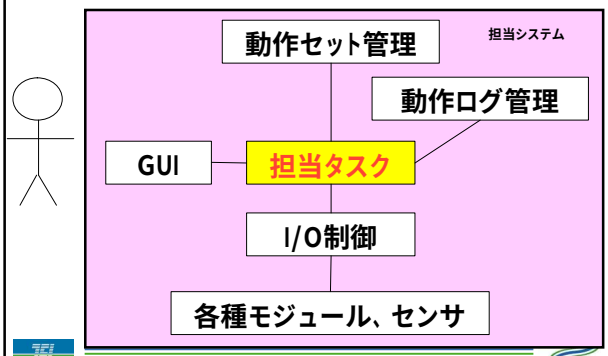
- 顧客と開発者の円滑なコミュニケーション
- 必要最小限の設計しか行なわない単純さ
- 頻繁なテストによるフィードバック
- 大胆な設計変更に立ち向かう勇気



開発環境

言語	: C
OS	: Unix系
ハードウェア	: PC/AT互換機
Unit Test	: Cunit for Mr.Ando
ソースコード管理	: CVS
Bug Tracking System	: 影舞

タスク紹介



まずはコーディング

コーディング前の壁

- まだコーダーがやってきていない
  新人をどうやって教育しよう…
- 前と同じく、コーダーの質でソフトの
  品質が決まるようだと困るぞ
- 納期が決まっていて、コーディングとテストで6ヶ月しかないぞ

そうは言っても、まずは教育からはじめなくては……

全部教えるなんて最初から諦めちゃえ

内容を限定した教育

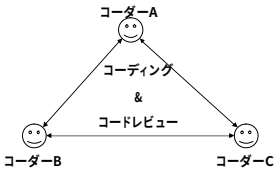
- 当社のコーディング規約を読ませる
- 周辺Toolの一つを作らせて、
  各人のコーディングの質を把握する
- 詳細設計書からコードに落とす練習をする

以上を3weekほどで実施した。

品質の高いコード作成

・Unit Testを行いながらのコーディング

・ペア(三すくみ)によるチェック体制



・測定ソフトによる、ソースコードの状態の把握

Unit Testとは？

関数の入/出力を利用して関数レベルでの正当性を検証するテスト方法。

例)

「Function1は入力値が0だった場合エラーを返す」という仕様であった場合に

retCode = Function1(0);

でretCodeがエラーであることを確かめるテスト方法

C/C++

gcovとは？

カバレッジを測定してくれるツール

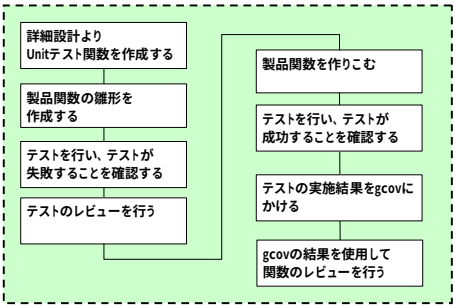
C/C++

・各関数が実行される頻度

・ファイルレベルの頻度情報

・関数レベルの頻度情報

コーディングFlow



Unit Testの効果

機能試験中の不具合修正 Phaseにおける結果

Unit Test動作回数：586回
発見したコードミス：20件

例)「==」と「!=」間違えた　「||」と「&&」間違えた etc…

これらをコーディングの段階で発見しなかった場合、

・BTSへの不具合の投稿

・デバッガを立ち上げての原因箇所の特定

・類似箇所検索

の余計な作業が発生していた

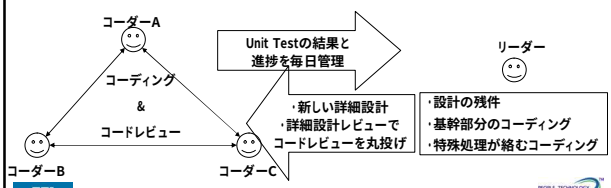
最悪の場合不具合を見逃していたかも

色々やってみてよかった点

・必ず、関数の詳細設計が出来上がっている

・Testを意識してソースを書くため、戻り値のチェック
やエラー処理などが揃い、ソースの品質が安定している

・三すくみでコードチェックを行わせていたので、
Testの実施と合わせて作業を安心して任せられる



Unit Testの導入

ただし既存のシステムに、導入するにはハードルが高い

・既存の関数が、テストを意識して作られていないかも
・戻り値を正しく評価していない関数がたくさんあるかも
・既存の関数にたくさん手を入れることになるかも
・「テストを作る工数が馬鹿にならない」と上司に言われるかも

なので

大規模な改修や作り直しや
新規タスク作成のときに導入する

Toolによる製品状態のモニタリング

CCCCにかけて、
McCabe's 複雑度30以上
LOC 100以上
の関数にリファクタ検討を行う

静的コード解析Toolにかけて、ランクA(危険度高)の指摘を修正する

結果をコーディング作業に
フィードバック

Toolによる製品状態のモニタリング

・CCCCにより、BIG Functionや"出来"の悪い関数を機械的に見つけることが出来るので、修正を検討できる
  一般的な目標値
    LOC:100以内
    MVG:30以内

・PG-Reliefにより、不具合の種となりうる箇所を潰せる
  主に発見された重大な指摘
    未初期化変数となってしまうPathの発見
    無限Loopに陥りかねないPath
    copy size間違え…etc

これらの評価に工数がほとんど掛からない

さあ、テストだ

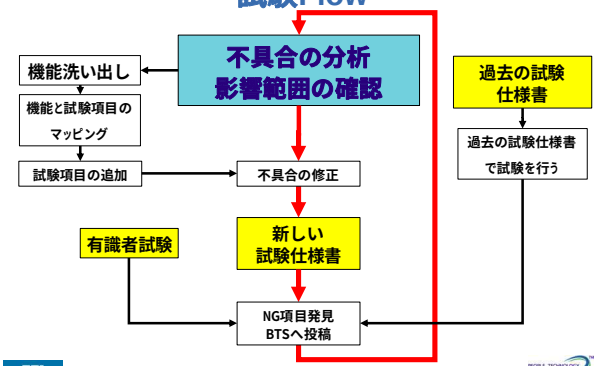
機能試験

・新規作成試験仕様書(166項目)
    新/旧比較試験
    項目に従った試験

・過去の試験仕様書 (383項目)

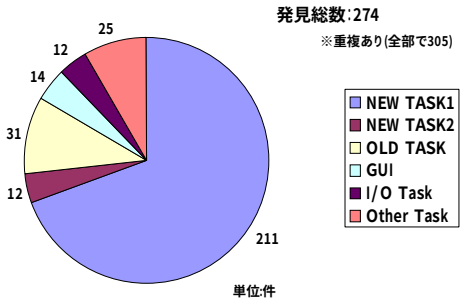
・有識者によるチェック

試験Flow



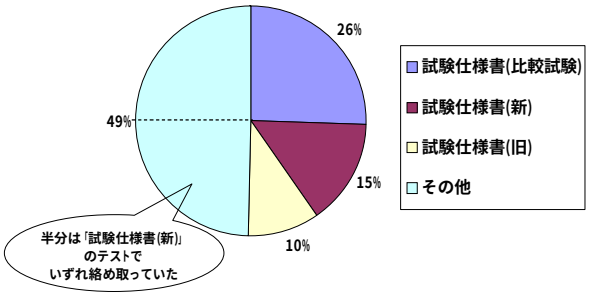
不具合検証の一例

発見不具合(Task別割合)



不具合検証の一例

不具合発見試験



不具合検証の一例

発見試験について

○今回のような作り直しの開発では、旧TASKとの比較試験は非常に効率的である

○有識者による、試験は不具合発見に有効である
- ・弱い部分を知っている
- ・動作が変であると、直ぐに検知できる

○過去の試験仕様書による試験は不具合発見に有効である
- ・マイニングできなかった機能を見つけることが出来る

BUGのデータを取っておこう

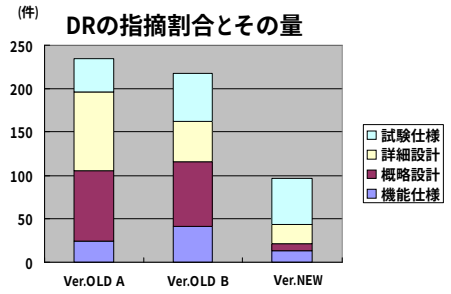
今回このような検証ができているのは、
BUG情報を取りためていたから

○バグはそのソフトの強い部分も
弱い部分も教えてくれる

○開発中に対策を打っていくことができる

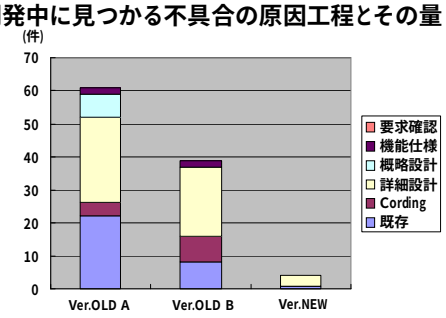
○みんなで情報を共有できる

「整備されたドキュメント」と「質の高いソースコード」の効果



「整備されたドキュメント」と「質の高いソースコード」の効果

開発中に見つかる不具合の原因工程とその量



XPで言ってる12のプラクティス

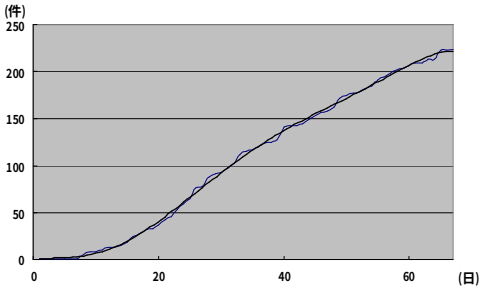
- 計画ゲーム(The Planning Game)
- 小規模リリース(Small Releases)
- 比喩(Metaphor)
- シンプルデザイン(Simple Design)
- テスティング(Testing)
- リファクタリング(Refactoring)
- ペアプログラミング(Pair Programming)
- 同所有権(Collective Ownership)
- 継続的インテグレーション (Continuous Integration)
- 週40時間(40-Hour Week)
- オンサイト顧客(On-Site Customer)
- コーディング標準(Coding Standards)

今日はこれまで！

Appendix

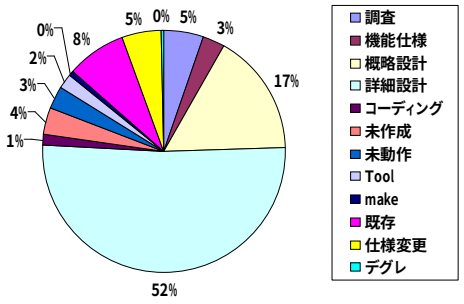
発見総数:223

NEW TASKでの発見不具合推移



※67稼働日での結果

混入工程



混入工程について

○詳細設計に起因する不具合が非常に多い

・人員を確保して教育する時間を十分に取れなかったため、
詳細設計の段階でレビューが不十分であった

・しかし、機能マッピングした試験計画で試験をおこなっている
ので想定している機能ないでの保障は行えているために
特に問題はない。(つまり予定通り試験で絡め取っている)

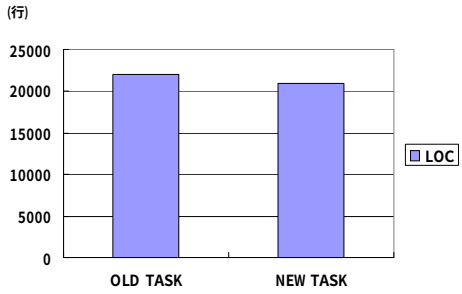
○コーディングによるデグレがほとんど発生しない
不具合におけるコーディングのデグレが占める割合：0.4%
(しかもUnit Test適応外関数)

機能マッピング例

機能に対して試験項目をマッピングして、内容の網羅性を検証

機能	試験仕様書1	試験仕様書2	試験仕様書3	試験仕様書4	試験仕様書5	試験仕様書6
指定コマンドの実行を行う					○	
外部に現在の実行状態を知らせる	○					
⋮						

コード量は?



性能評価

LOC分布比較(対数表示)