プロセス改善の自己観測による設計手法有効性体験コースの実施

鈴森寿之[†] 久保田卓秀[†] 長田晃[†]
海谷治彦[†] 海尻賢二[†]

設計法の有効性を系統的、定量的に示すコースを考案、実施した。手順は以下の通りである。1. 設計手法を段階的に導入し、課題を解いて行く。2. 個々人のプロセス・プロダクトを計測する。3. 測度を用い手法の有効性を確認する。設計法としてプログラム言語のデータ定義・関数定義を基礎として、モデル化言語で良く使われる陰関数定義と不変条件を形式的に書く事に拘らずに利用した。

Applying an education cource where each student confirms validity of design methods by mesuring his or her software process

Hisayuki Suzumori,[†] Takahide Kubota,[†] Akira Osada,[†]
Haruhiko Kaiya[†] and Kaijiri Kenji[†]

We have proposed and applied an education course where each student confirms validity of design methods systematically and quantitatively. During our course, students confirm validity of design methods as follows; First, they practice several number of exercises, while design techniqes are guradually introduced. Second, process and product data of software development are recorded in each execise. Third, by evaluating process and product data using metrics, they can confirm validity of design methods. In this course, we have selected design methods that consist of implicit function definitions, invariants of data type, function definitions and data definitions. We didn't dwell on design that are written formally.

1. はじめに

ソフトウエア教育において、設計は重要な要素である。設計の重要性は様々な文献、講義の内容として知る事が出来る。例えば、設計でプログラムの構造を決める事で早い段階に過不足のある部分を発見し修正する事が出来る。また、設計で関数を決める事でどのような機能が必要なのか確認し、早期にプログラムが何をしなければいけないか、また何をしてはいけないか議論する事ができる。しかし、その重要性が各所でいわれるにもかかわらず特に初学者においては設計せずにプログラムに取り掛かってしまう傾向がある。そして、設計をするにしても、いつ、どのように、どれだけ書けば良いかはわかりずらい。設計の指針・手順を示し、設計の有効性を自分自身の開発の中で系統的、定量的に示す事が出来れば初学者は設計の有効性を自覚し、実際のソフトウエア開発においても早い段階で設計をする動機が得られるだろう。

本稿では設計手法の指針を示し、実際の自分自身の開発においてどのような有効性があるか系統的、定量的に示す学習コースを実施しその結果を報告することを目的とする。このコースのユーザは課題をこなし、設計手法を段階導入して行く。設計手法として今回は、開発に使用するプログラム言語[☆]のデータ定義、関数定義を基として、形式的に書く事に拘らずにモデル化言語で広く使われる不変条件、陰関数定義を加えたものを使う。又、本稿ではソフトウエア開発を定量化する必要がある。その手法として Personal Software Process^{SM}[3),4)] を利用する。本稿のコースを利用することで、コースのユーザは明確なプロセスや設計が無かった状態から、明確な手順による開発、定義された設計による開発を経験する事が出来るであろう。さらには幾つかの測度によって段階導入した設計法がはたして良いか否か確認出来るであろう。

† 信州大学
Shinshu University

☆ 今回は C 言語を利用した。

2. コース概要

今回のコースは VDM over PSP[6] として提案したものを基に作成した。この節ではコースの概要を述べる。

2.1 コースの構造

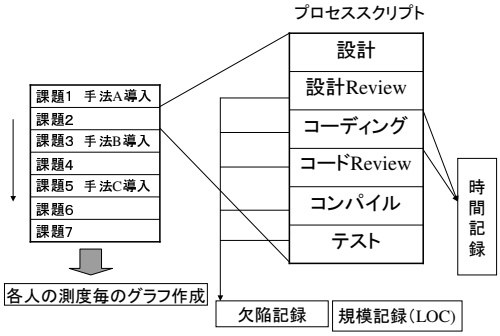

図 1 コース概要

図 1 はコースの概略を示す。このコースのユーザは、複数の課題をこなす。その上で、初めのユーザ自身の従来の開発方法から、最終的なコースの目標とする開発方法へ無理無く移行するために段階的に手法を導入して行く。

各課題での開発は、プロセススクリプトと呼ばれる物で規定される。ユーザはそれに従う事で明確に開発手順が定義され、ソフトウエア計測活動に役立てる事が出来る。初めは簡素な記述しか無いが、手法が導入されるに従い導入される手法の記述が加わりより具体的になる。そして、各課題でプロセスとプロダクトについて計測する。プロセスを基準に計る物は二つある。時間記録と欠陥記録である。時間記録はプロセスの各工程で費した時間である。欠陥記録はその欠陥が作り込まれた工程、除去された工程、欠陥の分類と欠陥の除去にかかった時間を記録する。プロダクトについては出来たプログラムの規模を LOC(Line Of Code) で計る。

全ての課題が終ったら、課題の進行を X 軸に取った各評価測度のグラフを作成する。各グラフ上での変化をみて総合的に手法の有効性を確認する。

2.2 設計品質を如何に計るか？

このコースで行われる課題は要求が明確に定義されており、開発は個人で行う規模である。従って、一般に良いソフトウエアは要求に添ったソフトウエアであるが、ここでは欠陥の少ないソフトウエアが良いソフトウエアと定義する。第 2.1 節で説明したように欠陥を記録しているので、それを主に用い品質を計る。

2.3 設計品質測定のための欠陥分類

設計品質を計るために欠陥分類を以下のように大きく 2 つに分ける。

- 設計欠陥
  - 使用するアルゴリズムを間違えた
  - 関数を余計に作ってしまった
  - メモリ開放を行ってなかった
  - 関数の引数を間違えた
- 構文欠陥
  - 文末のセミコロンを間違えた
  - 関数名を間違えた
  - を閉じてなかった
  - インクルードヘッダが違っていた

設計欠陥は意味的な欠陥であり、構文欠陥は単純な Syntax 上の間違いである。欠陥を記録する時この分類のどちらに入るかを判断し記録する。

2.4 設計品質評価測度

設計品質評価測度の定義とその目的を以下に示す。

- Raito of phase where Design Defects are injected(DDI)
  $\mathrm{DDI}(phase_i) = \frac{\mathrm{defects_injected_in_}phase_i}{\mathrm{all_design_defects}} \times 100$
  目的:ユーザは早い工程で悪い設計判断を見つける事が出来る。
- Raito of phase where Design Defects are Removed(DDR)
  $\mathrm{DDR}(phase_i) = \frac{\mathrm{design_defects_removed_in_}phase_i}{\mathrm{all_design_defects}} \times 100$
  目的:ユーザは設計段階でほとんどの設計欠陥を削除している。コーディング工程以降にはほとんど設計欠陥を残していない。
- Design Defect Removal Leverage(DDRL)
  $\mathrm{DDRL}(phase_i) = \frac{\mathrm{Defects/Hour}(phase_i)}{\mathrm{Defects/Hour(UnitTest)}}$
  目的:ユーザは早い段階で欠陥を削除出来る。
- Productivity
  $\mathrm{Productivity}(exercise_i) = \frac{\mathrm{LOC}}{\mathrm{total_development_hour}}$
  目的:生産性の向上を見る。
- Number of Design Defects par KLOC(NDDK)
  $\mathrm{NDDK}(exercise_i) = \frac{\mathrm{all_design_defects}}{\mathrm{KLOC}}$
  目的:設計欠陥数の下降を見る。

$phase_i$ はその課題のプロセススクリプトでの i 番目の工程を意味する。この場合、その課題のプロセススクリプトに記述のある全ての工程を計算しなければいけない。$exercise_i$ はコースにおける i 番目の課題を意味する。

それぞれ課題毎に全ての測度を計算する。そして、

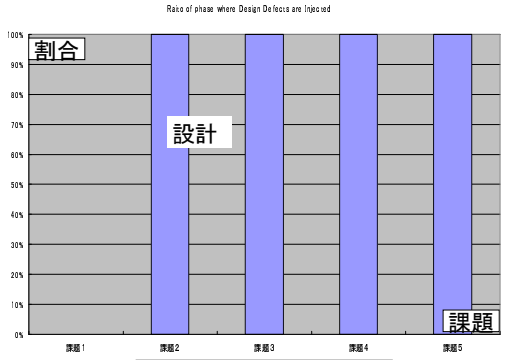

図 2　DDI(Raito of phase where Design Defects are Injected)

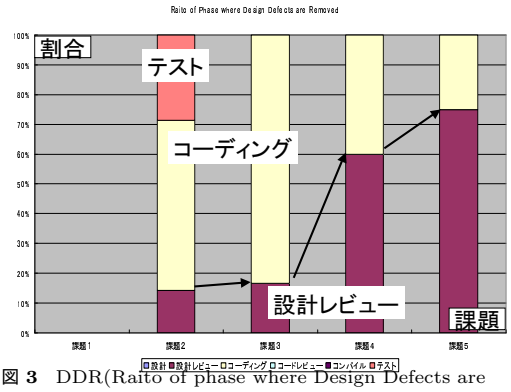

図 3　DDR(Raito of phase where Design Defects are Removed)

全ての課題が終了した時に、測度毎に、課題の進行を X 軸に取ったグラフを作成する。

2.5　測度毎のグラフの読み方の例

以下に測度毎のグラフの読み方について例示する。なおこのデータは今回の実験で取られた実データである。この場合は課題毎に手法を間断なく段階導入している。

図 2 は第 2.4 節で提案した、コーディング工程での設計欠陥の作り込みは設計判断が早い段階でよく検討されていないことを間接的に示すグラフである。このグラフの場合、課題 1 は設計欠陥が無かったので空である☆。この場合、課題 2 以降全ての設計欠陥は設計工程で作り込まれているので、設計をしっかり設計工程で行っている事を暗に示す。

図 3 は第 2.4 節で提案した、何処の工程が良く設計欠陥を除去するのに貢献しているかを示すグラフである。先と同様に設計欠陥が課題 1 には存在しなかったため空である。しかし、課題 2 以降は設計レビューの

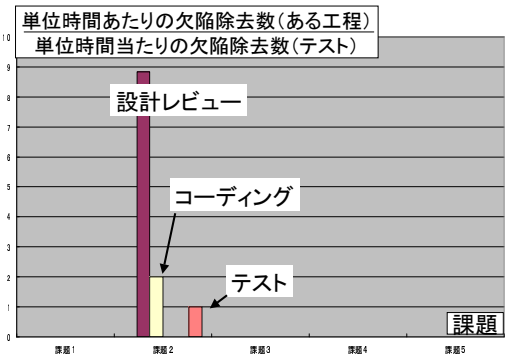

図 4　DDRL(Design Defects Removal Leverage)

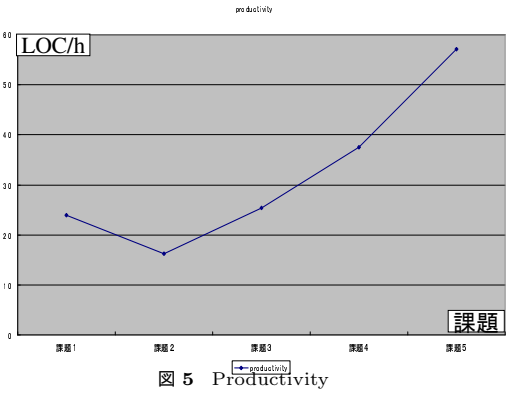

図 5　Productivity

貢献が段々上昇しているのが分かる。設計欠陥は早い段階で取れている事を示す。設計レビューで取ると利益があるのかは次のグラフで示される。

図 4 は第 2.4 節で提案した、単位時間にテストで除去出来る設計欠陥の数を 1 とし他の工程が設計欠陥を除去するのにその何倍の効率を持つかを示すグラフである。このグラフをみると設計レビュー工程で除去するのが一番効率が良い事が分かる。

図 5 は第 2.4 節で提案した、生産性の変化を見るグラフである。課題 2 で少し落ち込んでいるが全体的に段階導入に従い生産性が上がっていることが分かる。

図 6 は第 2.4 節で提案した、KLOC あたり、どのぐらい設計欠陥を作り込む可能性があるかを示すグラフである。課題 1 では設計欠陥が記録されなかったので空だが、課題 2 からは段階導入に従い減少している事が分かる。

以上のように、全体的に改善傾向がある事が分かる。課題を始める以前のこのデータの人の開発より段階導入された開発の方が行われた課題に関していえば良い事が分かる。

☆ テスト不足だと思われる。

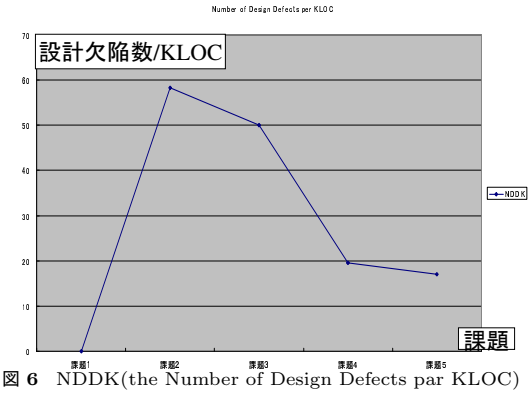

図 6　NDDK(the Number of Design Defects par KLOC)

3. 今回使用する設計法

本稿でのコースで利用する開発言語はオブジェクト指向言語ではない。基本的には開発に使用するプログラミング言語の関数定義とデータ定義を基礎として、それにモデル化言語[2),5),7)] で良く使われる要素を、形式的に書く事に拘らずに付け加えたものを設計法とする。

3.1 モデル化言語で良く使われる要素

モデル化言語で良く使われる要素として以下の物を利用する。

集合、系列、写像型　:自分で定義した型および、用意されている基本データ型に対しその集合、系列、写像を簡単に定義でき、利用する事が出来る。これを利用する事で、早期にデータ構造の過不足、データの振るまい上の欠陥を議論できる。

不変条件　:データ型が定義されても実際はデータ型が表す値の範囲全てを使う分けではない。現実世界などをあるデータ型に投影した時に絶対に起きてはならない、または起こらない条件をデータ型が常に満たす条件としてデータ型とともに記述する。そこれ記述する事で、データ型にどのような特性があるかよりはっきり自覚する事が出来、データの変更時における欠陥をより厳密に議論できる。

陰関数定義　:事前条件 (Pre-condition) と事後条件 (Post-condition) によりその関数の中身 (アルゴリズム) を書かずに、入力 (関数の引数と事前条件) と出力 (事後条件) のみでその関数が何をするかを書く。これを記述する事で、実装時より単純に関数の仕様を記述出来、早期に機能の過不足、欠陥を議論できる。

以上の要素を形式的に書く事に拘らずに利用する[1)]。これを便宜上、半モデル化言語と呼ぶ事とする。モデル化言語のように形式的に書く場合と違い、半モデル化言語ではコンピュータサポートによる半自動チェックの恩恵が得られない。そこでレビューでそれを代用し、講師側がレビュー時に使用する確認項目を提示し被験者がレビュー中に使うように指示した。

3.2 半モデル化言語の例

今回利用するプログラミング言語は C 言語である。C 言語のデータ定義、関数定義を基に半モデル化言語は表記される。そして、C 言語の文法や、記号も採り入れる。例として、以下を取り上げる。

> ATM(自動現金預け払い機) がある。ある人物がお金を引き落とす手順を設計せよ。手順は以下の通り。
> (1)　カードを入れる (ユーザ)
> (2)　カードから ID を読み込む
> (3)　暗証番号打ち込み (ユーザ)
> (4)　暗証番号読み込み
> (5)　引き出す金額を指定 (ユーザ)
> (6)　要求金額読み込み
> (7)　認証
> (8)　お金を出す。

3.2.1 例:データ構造定義

入力される顧客のデータを一時的に格納するデータ構造が必要であろう。それを以下のように定義する。

```
typedef int bool;
#define TRUE  1
#define FALSE 0

typedef struct one_of_client {
    int   id;//カードから読んだ識別子
    char *passwd;  //顧客が入力したパスワード
    bool auth_ok;  //認証の成否
    long int need_amount_of_money; //要求量
}one_of_client; //入力される顧客一人の情報
```

又、認証の時に参照されるであろう顧客情報の中の顧客一人のデータ構造も必要であろう。それを以下のように定義する。

```
typedef struct client {
     int  id;      //顧客の id
     char *name;  //顧客の名前
     char *passwd;//顧客のパスワード
     long int deposit_money;//顧客の預金量
}client;
```

3.2.2 例:抽象的なデータ型の利用

前節の顧客情報の中の顧客一人のデータ構造は顧客情報の中の一要素である。ここで顧客情報全体を簡潔に表すために集合を使う。

```
typedef set of client client_database;
```

set of と書き集合であると言う事をここでは宣言しているが、“client の集合” と日本語で書こうが関係ない。集合であると言う事が分かればよい。

3.2.3 例:不変条件

前節で書いたように C 言語の書き方と抽象的なデータ型 (集合) でデータ構造を定義した。それの不変条件を定義する。

one_of_client 型に関して見ると、以下の特性がある。

- 認証の成否は真、偽しかあり得ない
- 要求金額は 0 以下にはならない

形式に拘らず書き下す事が目的なので、上記のままでも良い。だが、データ定義を用いて書き下すと以下のようになる。

- inv1:auth_ok = TRUE or FALSE
- inv2:0 <= need_amount_of_money <= x.deposit_money

client 型を見るとその集合には、違う二つの要素の id は被らないという特性がある。それをデータ定義を用い以下のように書き下す。

`inv 3:client` の集合に含まれるすべての `x,y` について,`x != y` ならば `x.id != y.id`

3.2.4 例:関数定義

次に関数を定義する。関数定義は C 言語のそれと同様である。例の全ての関数を例示するのは冗長であるので (8) の部分の “お金を出す” 操作の関数のみ示す。その関数を以下のように定義する。

```
int put_money(one_of_client *c,
          client_database *data);
```

3.2.5 例 : 陰関数定義

お金を出すためには以下の条件が揃ってなければいけない。

- 認証済
- 顧客情報の中に要求している顧客がいる
- 顧客のお金が要求量を下回っていない

又、関数が適用された後は以下の状態になってなければいけない。

- お金を出した分だけ、顧客情報の中に入っている
  今回の顧客の貯金が出した分だけ減っている

以上をデータ定義、関数定義を用いて書き下すと以下のようになる。

- pre:
  - c.auth_ok == TRUE
  - client_database の中のある x において x.id == c.id となる x が 存在する
  - client_database に含まれるある x について x.id == c.id ならば c.amount_of_money_need <= x.deposit_money
- post:
  - client_database に含まれるある x について,x.id == c.id ならば,x.deposit_money = x.deposit_money - c.amount_of_money_need

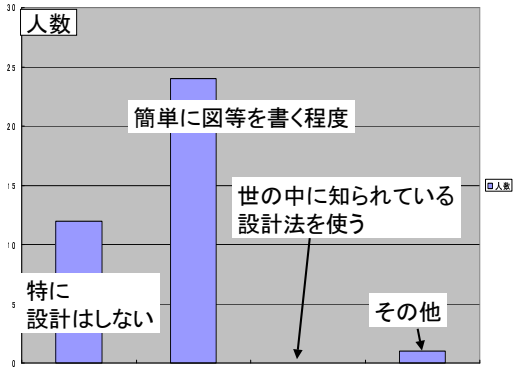

図 7 実験前のあなたの設計方法は？

4. 実 験 計 画

第 2 節で解説したコースを第 3 節で解説した設計法の有効性確認コースとして本コースの有効性を確認するために実際の教育の中で実施した。対象者は学部生 37 人である。なお授業のマテリアルは http://kaiya.cs.shinshu-u.ac.jp/2003/infexp/ から参照出来る。

4.1 被験者のスキル

被験者の実験前の開発手段は図 7、8 に示す通りである。

図 7 を見ると大半のものが簡単な絵や図を書く程度の設計しかしていない。特に設計していない被験者も大勢いる。その他の一人はコメントから、本稿の設計法に近い設計をしている明確に設計を行っている人は一人しかいない。他は明確に設計をすると言う習慣がほとんど無い。せいぜい簡単な絵や図を書く程度である。

図 8 に示されるように大半の被験者は作業手順を意識しない。プログラムを開発するときに自分がどのように開発しているか認識していない。他の 10 人近くの被験者は手順は認識しているが、アンケートのコメントから少なくともその中の一部の人間は try-and-error な開発方法である。その他の一人はコメントからウォーターフォール (以下 WF) モデル開発をしている。このことから、大半の被験者はプロセスを普段意識しない。そして、その他の被験者は手順は認識しているが少なくともその中の一部はそれが改善に結び付いていない。認識しているだけである。

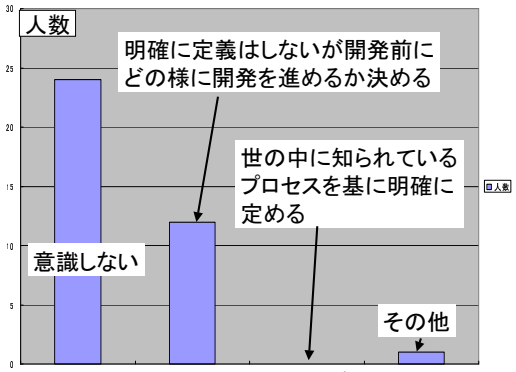

図 8　実験前のあなたの開発手順 (プロセス) は？

表 1　実験スケジュール

	導入手法	課題
第 1 回	時間測定、欠陥記録,WF 型開発	標準偏差、分散の計算
第 2 回	データ型定義、関数分割による設計、設計・コードレビュー	学生データの整合性チェック
第 3 回	設計におけるデータ型の不変条件の記述	生命の家系管理と検索
第 4 回	設計における陰関数定義の記述	ガソリンスタンドのシュミレーション
第 5 回	設計書を使った設計チェック	自動販売機のシュミレーション

被験者は学部 3 年生であるので、プログラミング経験は大体 3 年である。本稿のようなソフトウエア計測活動の経験は無い。

授業のスケジュールは 1 週間 1 回ペースで 6 回である。但し、3 回と 4 回の間に年末年始をはさんだ。

以上のような状況で適用実験を行った。

4.2　実験スケジュール

第 2 節で解説したように、課題の進行とともに手法を段階導入して従来の開発方法から目的の開発方法へと移行する。今回の目的は第 3 節で解説した設計法の有効性を確認する事である。従って、表 1 のように計画を組んだ。

第 6 回は収集したデータを測度毎のグラフにし、各々に配る。それぞれの課題のプロセススクリプトに記述された通りに開発し、手法を従来の自分の開発へ段階導入していきながら課題をこなす。

4.3　提供した測定ツール

計測活動は非常に面倒な作業である。従って、計測ツールを提供する事とした。第 2 節で、課題毎に測定するのは工程毎の時間記録、欠陥記録、規模記録であると解説した。被験者の労力を省くため、ツールは単純で分かりやすさを重視した。規模記録は講師側で

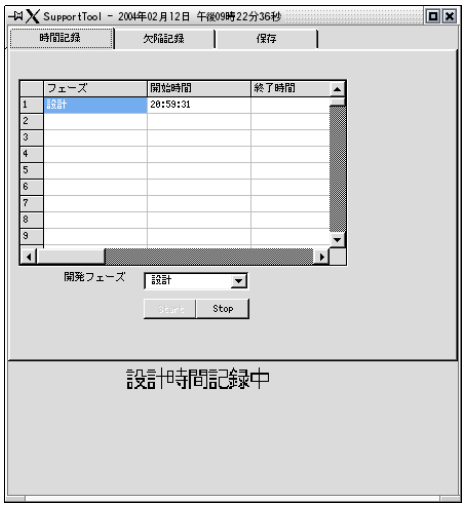

図 9　ツール画面 (時間計測)

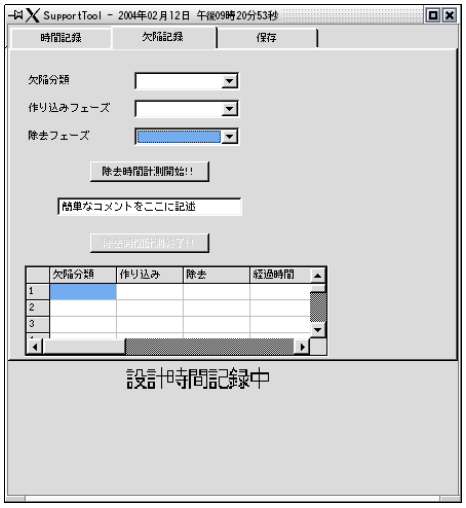

図 10　ツール画面 (欠陥記録)

行えるので、被験者は欠陥記録と時間記録を取れば良い。機能として、その二つを記録する GUI ツール☆を提供した。

図 9 の時間記録において使用者は、ツールの開発フェーズのリストから記録する工程を選択し、すぐ下の start ボタンを押す。すると計測が始まる。終了する場合は stop ボタンを押せば良い。

図 10 は欠陥記録のためのタブ画面である。使用者は欠陥を記録する時、上から順に選択し計測ボタンを押す。すると時間記録が始まりその下の停止ボタンを押せば時間記録が終了する。

ツールは結果を CSV 形式で保存する。使用者はそ

☆ 動作環境は linux である

表 2　正しく記録、または間違って記録した人数

	データ欠落無し	データ欠落あり
人数	21	17

れを提出するだけで良い。

5. 実 験 結 果

5.1 デ ー タ

被験者のデータ・プロダクト (プログラム・設計書) を収集したが、以下の問題点が現われた。

- プログラムのテスト不足
- 測定活動で記録されるべきデータの欠落
- 欠陥分類の理解不足
- 欠陥記録の煩わしさ

提出されたプログラムのほとんどが中身を見れば簡単に異常終了を起こすシナリオを見つけ出す事が可能なプログラムばかりであった。ひどい物になると引数なしで実行しただけで異常終了を起こす。課題毎に最低限行うべきテストケースをつけたが、大半の人がその状況しか想定していないようだ。

計測活動で記録されるべきデータ、時間記録と欠陥記録であるが、WF 開発を理解していない人や、一部の工程が無いもの、欠陥記録を放棄している者が見受けられた。人数の内訳は表 2 に示す。課題全てを通して正しく記録した人数は半数を少し越えるぐらいである。半数近い被験者が正しく記録していない。

最後のは、欠陥記録にはコメント覧があり、どのような欠陥であったか書き、記録する事が出来る。それを見る限り、欠陥分類を正しく理解していない人が見受けられた。

しかし、このような状況により分析に使えるデータはほとんど取れなかったが、改善がデータから見られた被験者が一部であるが、確認できた。第 2.5 節のグラフがその一つである。大半の被験者は生産性のグラフしか現われなかった。

5.2 アンケート

最後にアンケートを取った。手順はまず、先に実験の感想を聞くためのアンケートを行い。次に各被験者に各測度のグラフを渡し、典型的な改善例を全体に示した後、データを見てどう感じたかを聞いた。表 2 に示される、正しく記録した被験者のみを集計した結果を示す。正しく記録できなかった被験者のアンケートは省いた。

5.2.1 段階導入の速度

図 11 は手法の段階導入速度が適切かどうか聞いた結果を示す。多少難しかったようだ。大変だったと答えた被験者は主に練習不足を挙げた。

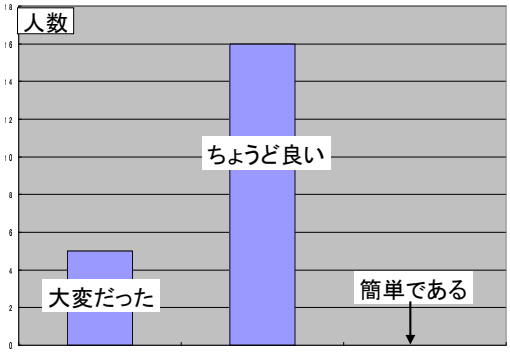

図 11　毎回新しい手法を紹介され、それを課題で使うのは？(データ提示前)

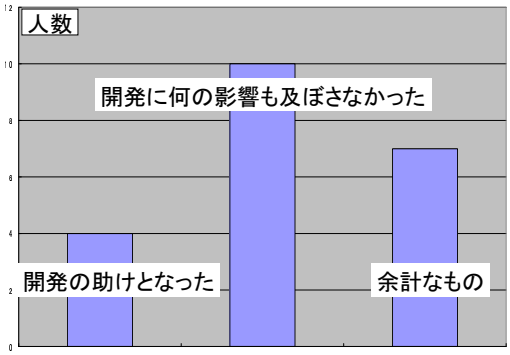

図 12　測定活動は開発の役にたったか？(データ提示前)

5.2.2 測 定 活 動

図 12 はデータを見せる前に取ったアンケートでの被験者の測定活動に関する意見を示す。4 人手助けになったと回答している。この被験者達は計測を自分を律するために使用している。計る事で、開発しているという自覚をよりいっそう持ち、集中できたと答えている。また、時間を計る事で効率の良い開発をしようと考えていた者もいる。何の影響も及ぼさなかったと答えている人は 10 人おり大半を占める。唯漫然と計測していただけの被験者もいれば、課題毎に測定データのフィードバックが無いために計測の意味を見出せなく選んだ被験者もいた。余計なものと答えた人はおもに欠陥記録の煩わしさとツールの使いにくさを挙げている。

図 13 はデータを提示した後にアンケートを取ったものである。

図 13 を見ると、有効性を知るためのソフトウエア計測活動について否定的な見解を持っている人は 1 人いかいない。ほとんど好意的な目で見てくれたようだ。半分に近い人が必要であると回答した。コメントとし

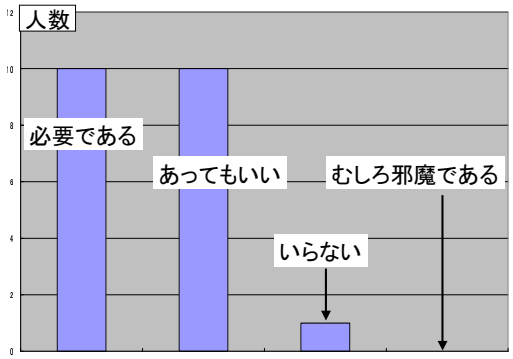

図 13　今回実験で行った計測活動について手法の有効性を知るためには？(データ提示後)

ては、“問題を見て設計をせず、コーディングに入る人がいるので、開発手段を早い時期に行った方が良い”というプロセスに言及するものから、“他の授業で設計を書くべきだと聞いたが、実際には書かなかった。無理矢理というか、必ず書かせる事で、その話しが本当だとわかった。” という段階導入と有効性に言及する物があった。いらないと答えた人はツールの不整備、計測の煩わしさを指摘した。

6. おわりに

本稿では提案した設計手法有効性確認コースが実際に働くかどうかを確認し結果を報告した。論理的なコースが前提とする Personal Software Process^{SM} の経験もなく、はっきりとした開発プロセスも無い人に対して行った。結果から、改善しているデータを確認する事が出来た。しかし、主に欠陥記録の不足、時間記録すべき工程の欠落などの理由から大半の人が全てのグラフを表せなかった。それにも関わらず、最終的には今回の有効性を調べる計測活動は大きく否定される事もなく、逆に半数近い人が必要であると回答した。さらには、良いと言われている事を確認し強く動機を持った人もいた。

問題点として以下があげられる。まず、ソフトウエア計測活動の難しさである。第 2 回と第 3 回の間に正しく計測できているかどうか全員に面接を行ったが、計測をするよりも、まず開発の方へ手がどうしても動いてしまうという意見が多かった。特に粒度の低い欠陥記録に対して顕著であった。やはり、記録、特に欠陥記録は訓練か、さらなるサポートツールが必要だろう。全く別の方法で欠陥よりも簡単に品質を定義することが必要かも知れない。

次に、欠陥分類の解釈である。一応、欠陥分類を例を出して説明したが、人によって解釈はまちまちであり、粒度も揃ってない。何を欠陥とし、何を一つと数えるか明確な基準が必要だろう。

有効性を確認するためのソフトウエア計測活動に否定的な意見を持っていない人でも、計測活動は邪魔だと思ってしまう人もいる。そのような人へのさらなるサポートツールが必要だろう。

謝　辞

今回のコースの実施に参加して下さった学部生に感謝致します。また、研究にデータを利用する事の許可を下さった受講者に感謝致します。

参 考 文 献

1) Derek Andrews. Formal methods in software engineering education. *proceedings of the 1996 International Conference on Software Engineering: Education and Practice*, pp. 514–515, 1996.
2) John Fitzgerald and Peter Gorm Larsen. *Modelling Systems, Practical Tools and Techniques in Software Development*. Cambridge University Press, 1998. VDM-SL, Toolbox Lite.
3) W. S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, 1995. The Complete PSP^{SM} Book.
4) W. S. Humphrey. *Introduction to the Personal Software Process*. Addison-Wesley, 1997.
5) Jonathan Jacky. *The way of Z – practical programming with formal methods*. Cambridge University Press, 1997.
6) Hisayuki Suzumori, Haruhiko Kaiya, and Kenji Kaijiri. VDM over PSP: A Pilot Course for VDM Beginners to Confirm its Suitability for Their Development. In *Proceedings of COMPSAC2003*, pp. 327–334, Dallas, Texas, Nov. 2003. IEEE.
7) 荒木啓二郎 張漢明. プログラム仕様記述論. オーム社, 2002.