

# 第六回課題 (1/3)

## ● 課題の概要

- 次の3つを結合して実行し、ファイルから読み込んだ図形や文字列が画面上に期待した形で表示できる (or できない) ことを確認する
  - 設計者として自分が作成したクラス
  - プログラマが作成したファイルからの読み込み部
  - プログラマが作成した画面への描画部
- 表示できない場合には原因を追究する

(注) ファイルからの読み込み部や画面への描画部を設計者も作成したケースがあるが、これらについては、プログラマが作成したプログラムを利用すること

# 第六回課題 (2/3)

## ● 提出物

- 無修正で問題なく動作した場合
  - Java のソースコード一式を jar などを用いて一つにまとめたもの
- 軽微な修正で問題なく動作した場合
  - 修正後の Java のソースコード一式を jar などを用いて一つにまとめたもの
  - 修正箇所の一覧
- それ以外の場合
  - うまく動作しない理由を分析したレポート

# 第六回課題 (3/3)

- 無修正または軽微な修正で動作した場合、本日の演習中(4時半頃まで)に動作を見せてください
- プログラマが作成した描画部のプログラムは以下のページにある
  - <http://www.is.titech.ac.jp/~zin4/TA/wiki/>
- 提出締切
  - 7月30日(月)
- 提出方法
  - <http://www.is.titech.ac.jp/~zin4/TA/kadai5/> にアップロードしてください

# プログラムの「正しさ」の確認方法

- いろいろな方法がある
  - 主にソースコードを対象とした検査
    - スタイルチェッカ, 構文チェッカなどの利用
    - プログラム検証
    - ソースコードレビュー
  - 実行をともなう検査
    - テスト
    - Design by Contract 用ツールの利用
    - assert の利用

# Design by Contract (1/2)

- Eiffel 言語には、事前条件、事後条件、不変条件を指定する構文がある
  - 事前条件: メソッドの実行前に成り立つべき条件
  - 事後条件: メソッドの実行後の成り立つべき条件
  - 不変条件: オブジェクトが(メソッドの実行途中を除き)常に満たすべき条件
- 単なるコメントではなく、実行時チェックの対象になる
- これらの条件を明示的に意識することも重要

# Design by Contract (2/2)

- いろいろ問題もあるが Java 用ツールもある

```
public class Account {
    private long balance;
    //@ invariant 0 <= balance;
    public Account() { balance = 0; }
    //@ requires 0 <= amount;
    //@ ensures balance = ¥old(balance + amount);
    public void deposit(long amount) {
        balance += amount;
    }
    //@ requires 0 <= amount && amount <= balance;
    public void withdraw(long amount) {
        balance -= amount;
    }
}
```

# Assert の利用 (1/3)

- よりお手軽なのは `assert`
  - `-ea` (または `-enableassertions`) オプションの指定が必要
  - Eclipse では, `Run` → `Run ...` とメニューをたどり, `Arguments` タブを開いて, `VM arguments` で `-ea` を指定

```
Shape parse(String line) {  
    assert line != null;  
    ...  
}
```

# Assert の利用 (2/3)

- 事後条件のチェックにも使える
  - Effects 条件をすべてチェックするのは大変だが、一部のチェックなら実用的
  - Return 文がたくさんあると面倒くさい

```
public String getShapeType() {  
    assert type != null && !type.equals("");  
    return type;  
}
```

# Assert の利用 (3/3)

- 基本的に開発時に利用する機能
  - リリース後も残したいチェックには使わない
  - Public メソッドの入力検査は, 別の方法で行うべき
- 副作用のある式を書かない
  - Assert により実行結果が変わらない範囲で使う
- コメントに近い役割も期待できる
  - 実際に実行されるので, 単なるコメントより信用できる面もある

# Hoare System

- 事前条件, 事後条件, プログラムの間の関係を表現する論理体系
  - $P \{ S \} Q$  という形の論理式を使う
    - $P$  が事前条件,  $Q$  が事後条件,  $S$  が文
- 以下のような公理や推論規則を使う

$$P[x \leftarrow e] \{ x = e \} P$$

$$\frac{P \wedge e \{ S_1 \} Q \quad P \wedge \neg e \{ S_2 \} Q}{P \{ \text{if } e \text{ } S_1 \text{ else } S_2 \} Q}$$

$$\frac{P \supset I \quad I \wedge \neg e \supset Q \quad I \wedge e \{ S \} I}{P \{ \text{while } e \text{ } S \} Q}$$

# 証明の例 (1/5)

## ● 例題: GCD

- 証明したいのは,  $x, y > 0$  なら  $x$  と  $y$  の GCD が返されること

```
int gcd(int x, int y) {  
    // requires  $x, y > 0$   
    // returns GCD( $x, y$ )  
    while (x != y) {  
        if (x > y)  
            x = x - y;  
        else  
            y = y - x;  
    }  
    return x;  
}
```

# 証明の例 (2/5)

- $x, y$  の初期値を  $x_0, y_0$  とする
- 基本的に証明したいのは下図の式

```
 $x = x_0 \wedge y = y_0 \wedge x, y > 0$   
{ while (x != y)  
  if (x > y)  
    x = x - y;  
  else  
    y = y - x; }  
 $x = \text{GCD}(x_0, y_0)$ 
```

# 証明の例 (3/5)

- While 文のループ不変条件を定める
  - $I \equiv x, y > 0 \wedge \text{GCD}(x, y) = \text{GCD}(x_0, y_0)$
- 証明すべきは以下の3条件になる
  - このうち1番目と3番目は通常の論理式

$$x = x_0 \wedge y = y_0 \wedge x, y > 0 \supset I$$

$$I \wedge x \neq y \{ \text{if } (x > y) \ x = x - y; \text{ else } \ y = y - x; \} I$$

$$I \wedge x = y \supset x = \text{GCD}(x_0, y_0)$$

# 証明の例 (4/5)

- 前頁の2番目の式をif文の規則で分解すると、証明すべきは以下の論理式になる

$$\begin{aligned} & x, y > 0 \wedge \text{GCD}(x, y) = \text{GCD}(x_0, y_0) \wedge x \neq y \wedge x > y \\ & \quad \{ x = x - y; \} \\ & x, y > 0 \wedge \text{GCD}(x, y) = \text{GCD}(x_0, y_0) \end{aligned}$$

$$\begin{aligned} & x, y > 0 \wedge \text{GCD}(x, y) = \text{GCD}(x_0, y_0) \wedge x \neq y \wedge x \leq y \\ & \quad \{ y = y - x; \} \\ & x, y > 0 \wedge \text{GCD}(x, y) = \text{GCD}(x_0, y_0) \end{aligned}$$

# 証明の例 (5/5)

- さらに代入文の規則で分解すると, 証明すべきは以下の式

$$\begin{aligned} x, y > 0 \wedge \text{GCD}(x, y) = \text{GCD}(x_0, y_0) \wedge x \neq y \wedge x > y \\ \supset x - y, y > 0 \wedge \text{GCD}(x - y, y) = \text{GCD}(x_0, y_0) \end{aligned}$$

$$\begin{aligned} x, y > 0 \wedge \text{GCD}(x, y) = \text{GCD}(x_0, y_0) \wedge x \neq y \wedge x \leq y \\ \supset x, y - x > 0 \wedge \text{GCD}(x, y - x) = \text{GCD}(x_0, y_0) \end{aligned}$$

# おわりに (1/2)

## ● 何が問題だったか？

- 近代的ソフトウェアの構築には、長期にわたって機能する大きな組織が必要
  - もちろん一般の会社等でも同様
- 大きな組織を柔軟に動かすのは非常に難しい
  - e.g. 意識の共有に時間がかかる
  - e.g. 内部規則が多数必要になる
  - e.g. 一度決めたことを簡単には変更できない

# おわりに (2/2)

- IT(or ICT) で近年何が変わったか？
  - 決まりきった手続きの自動化
    - e.g. IDE, Continuous Integration
  - コミュニケーションの円滑化
    - e.g. オープンソースの開発, オフショア開発
- 変化への対応
  - 問題を早く発見し, 迅速に対応する
    - 時々問題が発生するのは不可避
  - 顧客や開発チームメンバーとの意識共有と必要に応じた修正