善悪の彼岸
ーeclipse・JUnitによるテスト・ファーストー

岐阜経済大学　経営学部 経営情報学科　井戸　伸彦

来歴:

０．０版　２００３年７月４日

０．１版　２００６年１１月2日：JUNITの操作画面イメージの更新

スライドの構成

はじめに

- 本スライドでは、テストファーストと呼ぶ技法について初歩的な解説ならびに実習を行っています。
- 実習ではeclipseを使用しており、次のスライドで勉強済みであることを前提としています。
  - 「月に吠える　ーeclipseを用いたJavaアプリケーションの作成ー」
    - ◆eclipseの起動方法等は、このスライドを参照してください。
- 岐阜経済大学では、第2演習室のLinuxPC、および全WindowsPCに、eclipseはインストールされています。本スライドでは、これらでの環境での操作を記します。

（1）テストとは何か？

■ソフトウェアのテストは奥深い課題であり、簡単に全容を理解することは出来ません。

■ここでは、次のようにテストについて単純に考えておいてください。

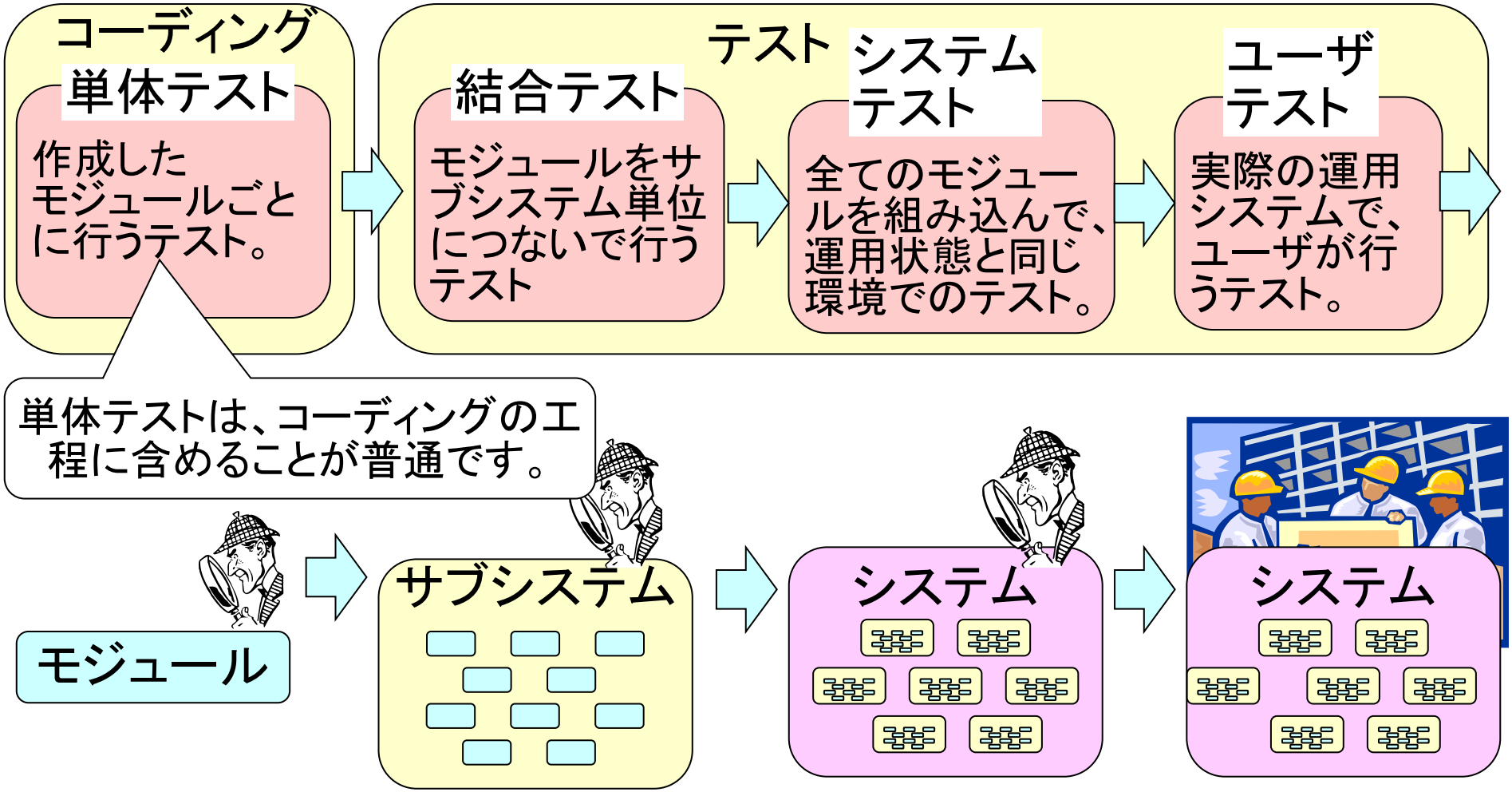
- プログラムが正常に動くかどうかの確認

■テストは、ソフトウェア開発の中で最大の工数を費やして行うものです。

- 「少しでも関連のある作業をテストに含めると、テストの総時間は、（ソフトウェア開発）全体の30%から90%をしめる」（Software Testing Techniques, Boris Beize, 1990）

（1．1）テスト工程

■テスト工程は、さらに、分割して考えることが普通です。

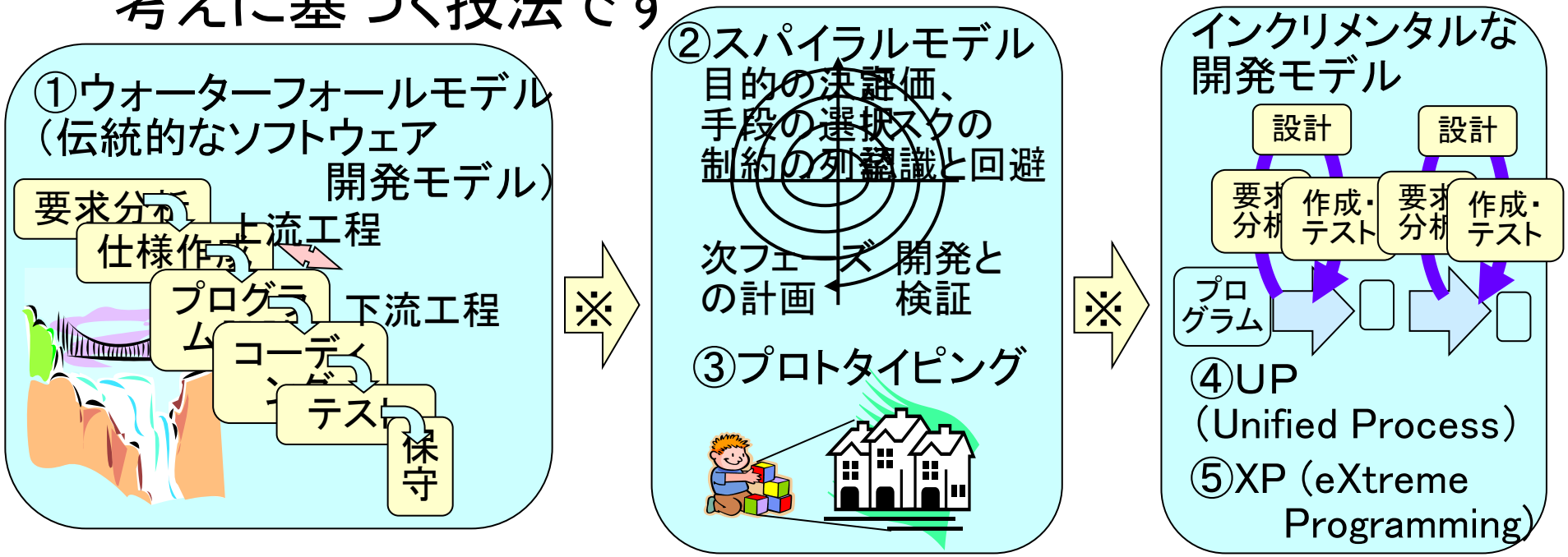


※モジュール：Javaのメソッドのような、プログラムの最小単位のことです。

（1．2）本スライドで学ぶこと

■本スライドでは、テストファーストと呼ぶ技法について初歩的な解説ならびに実習を行っていきます。

■大雑把に言えば、テストファーストの技法は、XP的な考えに基づく技法です



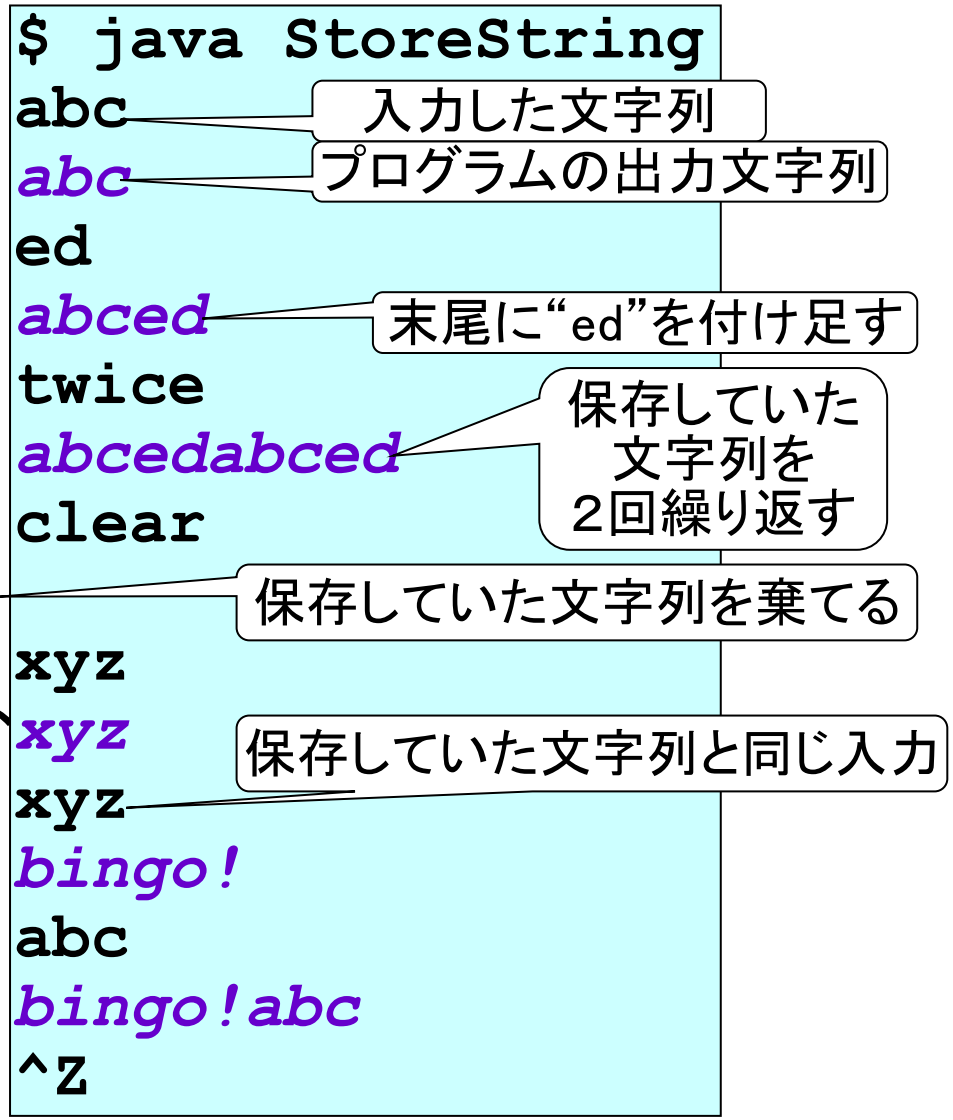
※　取って替わったのではなく、提案された順序を示す。

■本スライドでのテストという言葉は、単体テストを指しています。

（2．1）題材とするプログラム

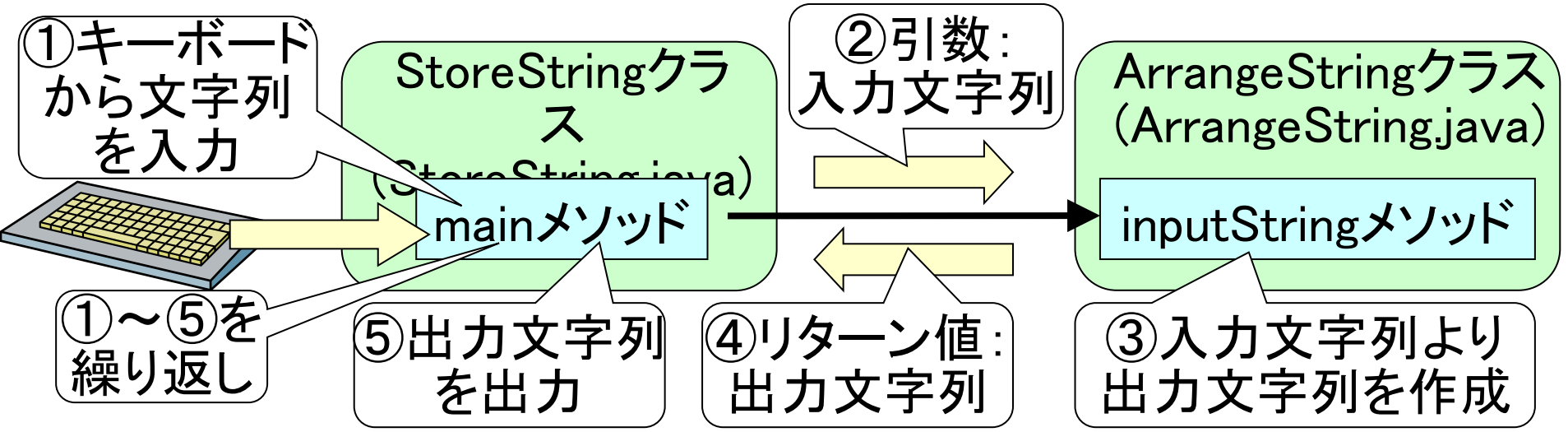
■下図のような入出力を行うプログラムを作成します。

- 文字列を入力するたびに、末尾に付け足す形で保存し、それを出力する。
- “twice”と入力された場合は、保存していた文字列を2回繰り返したものを保存し、出力する。
- “clear”と入力された場合は、保存していた文字列を棄て、空行を出力する。
- 保存していた文字と同じ文字が入力された場合は、“bingo!”という文字列を保存し、出力する。



（2．2）クラス構成

- ■今回は、下図に動作の概要を示した、2つのクラスでプログラムにて実現します。
- ■今回作成していくプログラムは、ArrangeStringクラスのInputStringメソッドです。
- ■StoreStringクラスのソースコードは、次のスライドに示すものをそのまま使います。



（２．３）StoreStirngクラス(StoreString.java)

■次のとおりとしてください（井戸のネットワークドライブからコピーしてインポートしてもOKです）。

```
import java.io.*;
public class StoreString {
  public static void main(String[] args)
                              throws IOException{
    BufferedReader kbd = new BufferedReader(
                          new InputStreamReader(System.in));
    ArrangeString arrangeString = new ArrangeString();
    String line;
    while((line=kbd.readLine())!=null
          && !line.equals("")){
      String output = arrangeString.inputString(line);
      System.out.println(output);
    }
  }
}
```

①キーボードから文字列を入力

①～⑤を繰返し

④リターン値：出力文字列

②引数：入力文字列

⑤出力文字列を出力

（メソッドinputStirngにて）
③入力文字列より出力文字列を作成

（２．４）最初のArrangeStringクラス

■最初のArrangeStirngクラスは、次のようにします（これが題意を満たしていないことは明らかです）。

```
public class ArrangeString {
  public String inputString(String inputString){
    return inputString;
  }
}
```

■このプログラムを順に作成していきます。

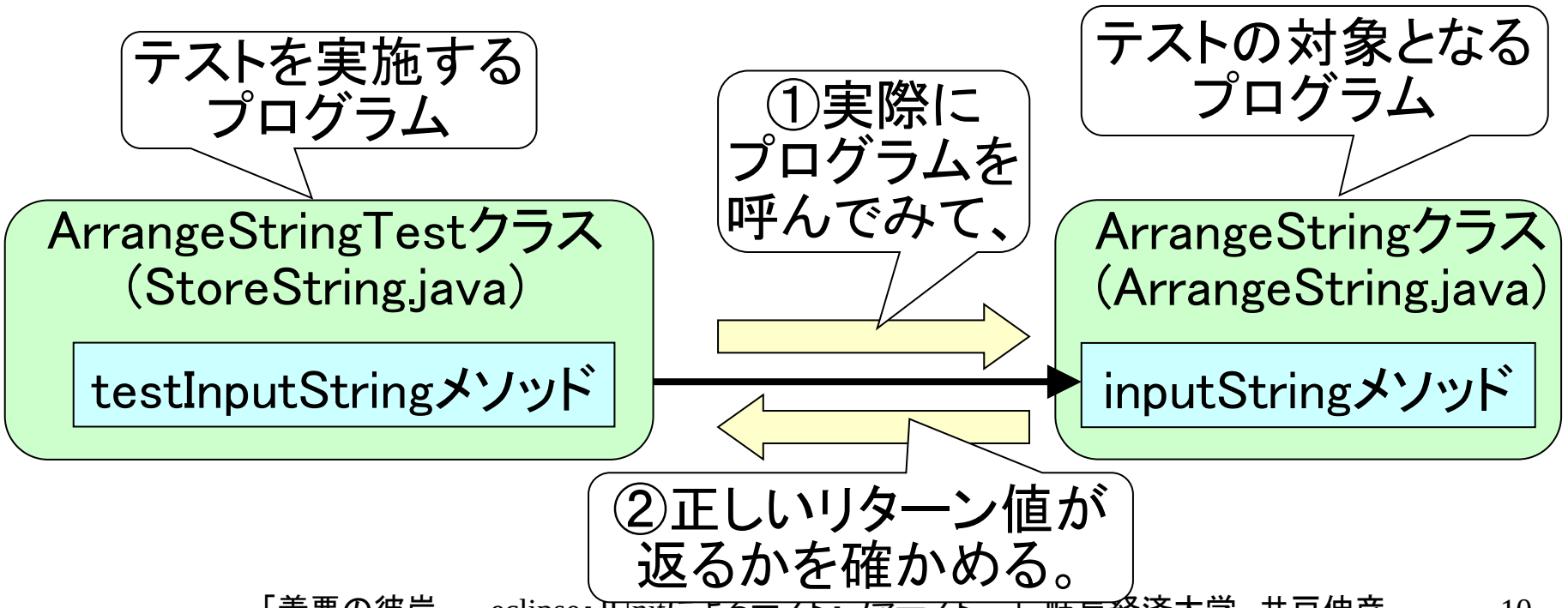
■StoreAtringクラスとArrangeStringクラスは、前スライド・本スライドにて記したとおりに、次のプロジェクトに作成したものとして、以下のスライドでは話を進めます。

- sampleProject

（３．１）最初に何を考えるか？

■スライド(2)項で示したプログラムを作成するに当たって、このスライドのタイトルにある（テスト・ファースト）とおり、最初にテストを考えます。

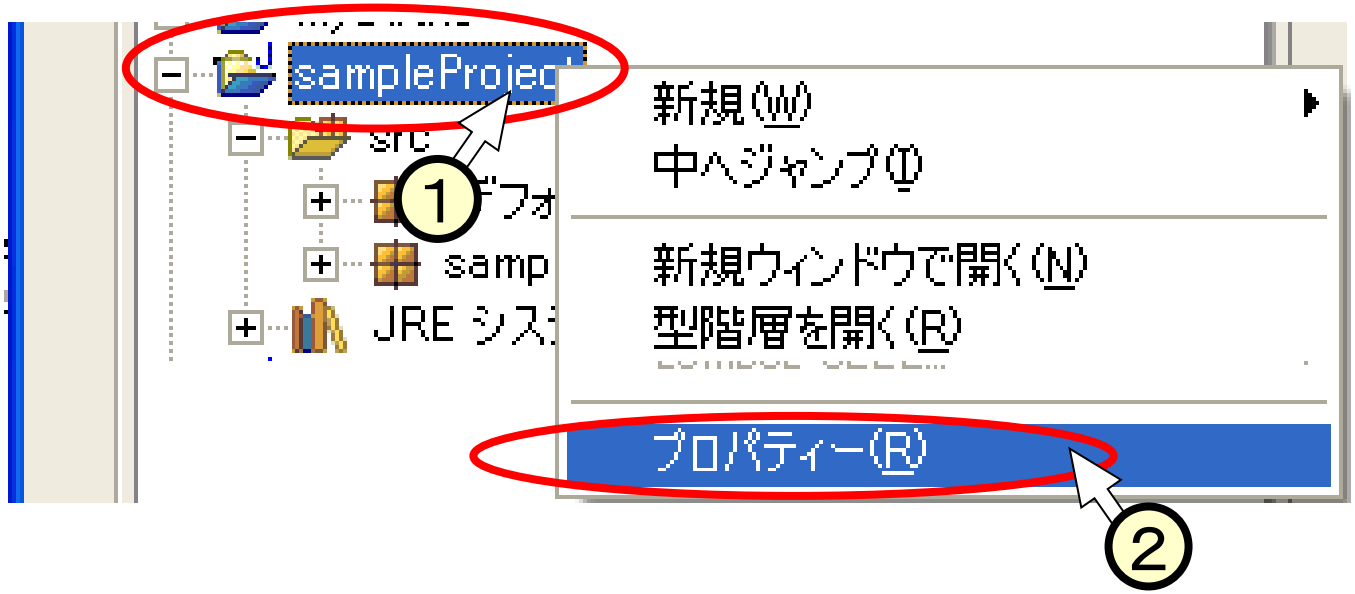
■これから作る下図左側のプログラムは、テストをするためのプログラムです。



（3．2．1）環境設定1　ープロパティー

■eclipseによる単体テストでは、JUnitというパッケージを用います。これのjarファイルをビルドパスに追加します。

■プログラムを作成するプロジェクト（ここではsampleProject）を右クリック（①）し、ポップアップメニューから、[プロパティー]をクリック（②）します。

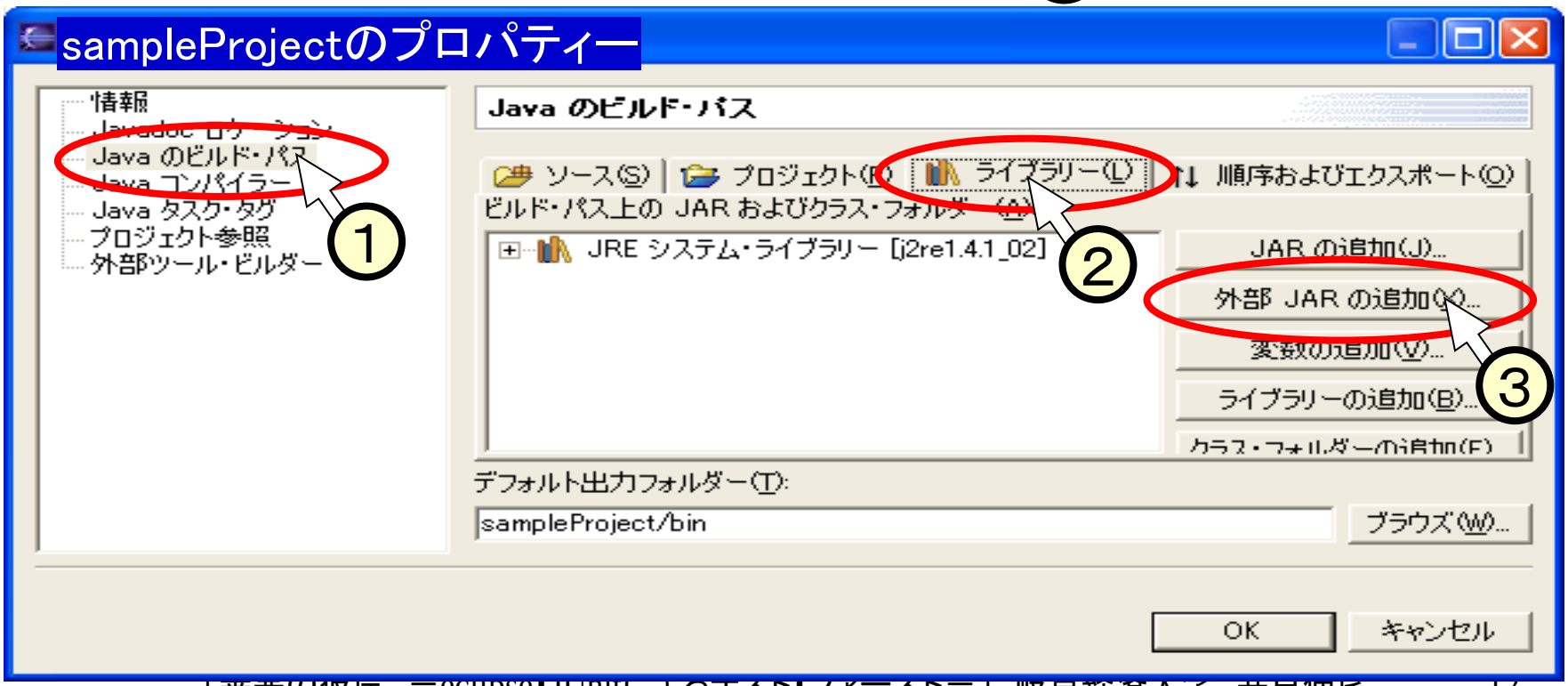


（3．2．2）環境設定2　－外部JARの追加－

■「プロジェクト(sampleProject)のプロパティー」ウインドウにて、左ペインにて[Javaのビルド・パス]をクリック（①）します。

■右ペインで[ライブラリー]のタグをクリック（②）します。

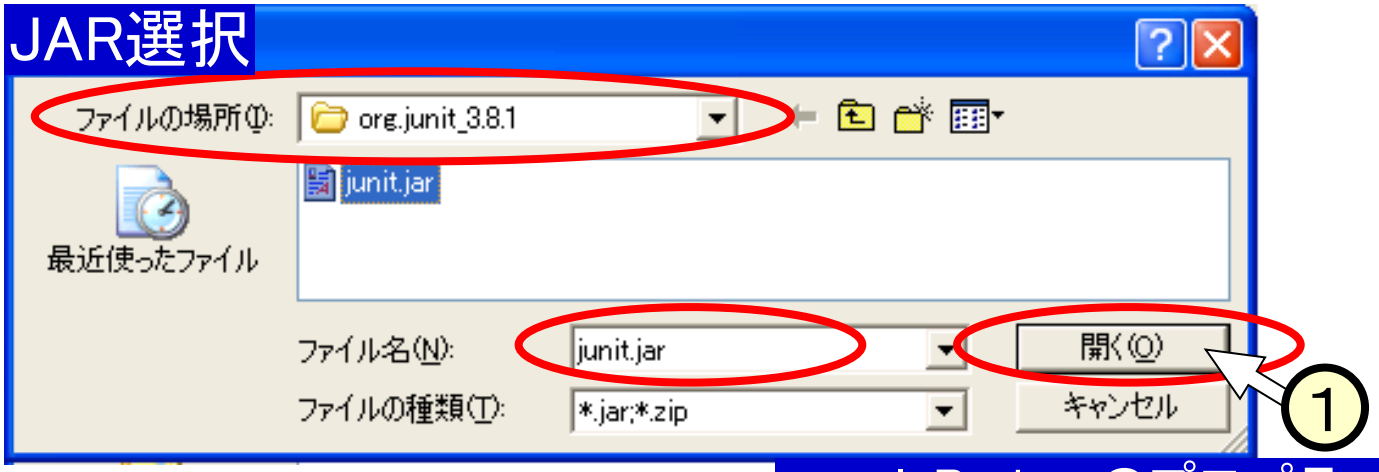
■[外部JARの追加]ボタンをクリック（③）します。



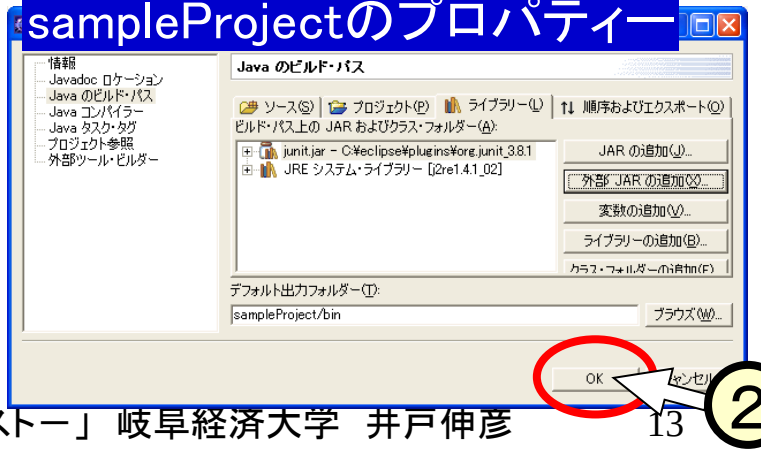
（3．2．3）環境設定3　－JAR選択－

■「JAR選択」ウインドウにて、次のファイルを選択します（①、いずれも大学での環境を前提としています）。

- Windowsの場合：[public]:¥ido¥systemDesign¥junit-3.8.1.jar
- Linuxの場合：/usr/local/eclipse/plugins/org.junit_3.8.1/junit.jar



■「プロジェクト(sampleProject)のプロパティー」ウインドウにて、JARファイルが追加されたことを確認（②）して、[OK]をクリックします。



（3．3）TestArangeStringクラス

■テストを実施するプログラムである、TestArangeStringクラスを次のとおり作成していきます。

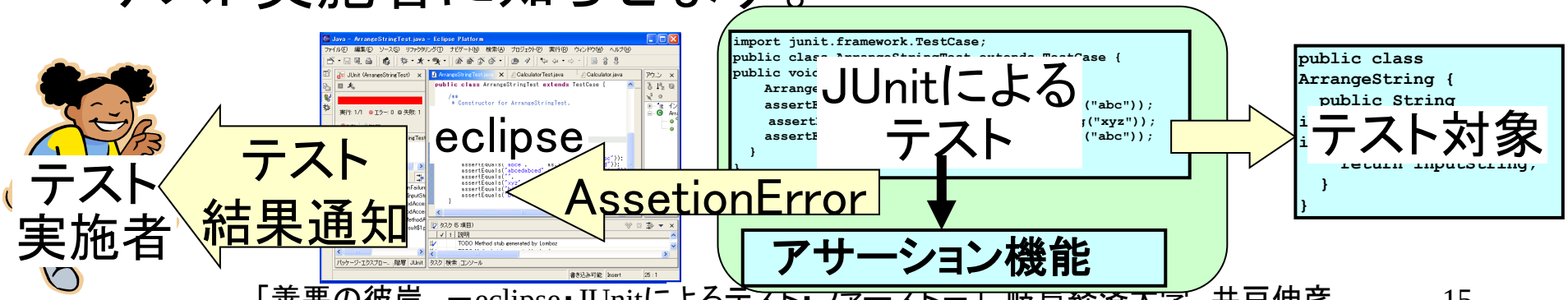
```
import junit.framework.TestCase;
public class ArrangeStringTest extends TestCase {
  public ArrangeStringTest(String arg0) {
    super(arg0);
  }
  public void testInputString() {
    ArrangeString as = new ArrangeString();
    assertEquals("abc",       as.inputString("abc"));
    assertEquals("abced",     as.inputString("ed"));
    assertEquals("abcedabced",as.inputString("twice"));
    assertEquals("",          as.inputString("clear"));
    assertEquals("xyz",       as.inputString("xyz"));
    assertEquals("bingo!",    as.inputString("xyz"));
    assertEquals("bingo!abc", as.inputString("abc"));
  }
}
```

（3．3．1）アサーション(assertion)機能

- Java言語の機能として、アサーション機能があります。これは、与えられた条件が成立しない場合に、エラー（java.lang.AssertionError）を発生させて、そのような事態が発生したことを知らせる機能です。
- 次のコードは、xが3以下のとき、AssertionErrorを発生させます。

```
assert x>3;
```

- JUnitではこの機能を用いて、テストにおいて失敗が生じた際にこれを報告する機能を実現しています。
- eclipseでは、JUnitの報告を、よりビジュアルな方法でテスト実施者に知らせます。



(3．3．2)テストの内容

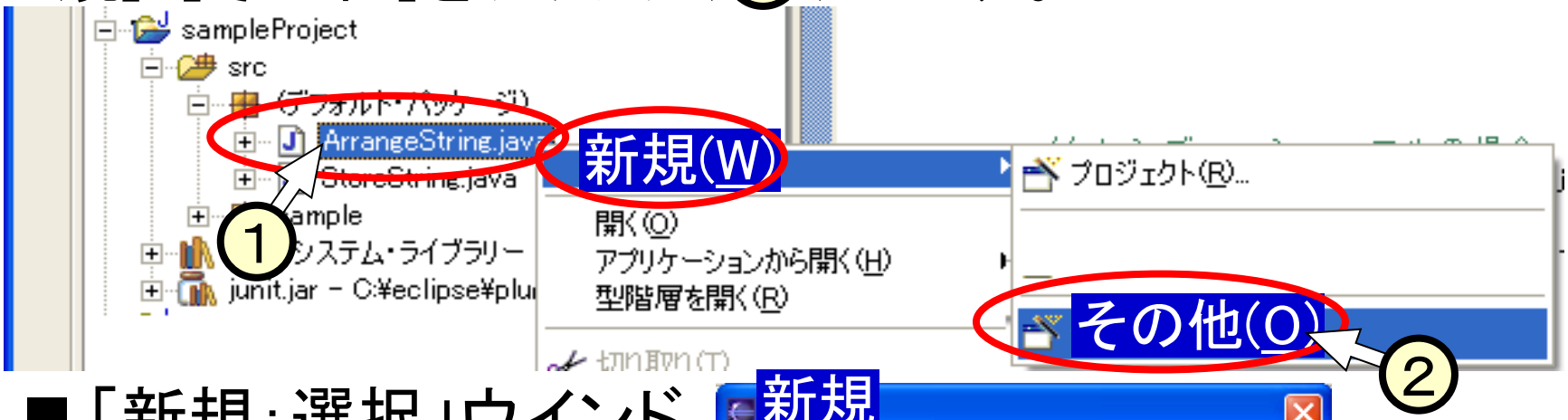
■スライド(3.3)のテストが、スライド(2.1)に示した動作例について、正しく動くかどうかを試していることは、明瞭であると思います。

```
$java .....
abc
abc
ed
abced
twice
abcedabced
  :
  :
```

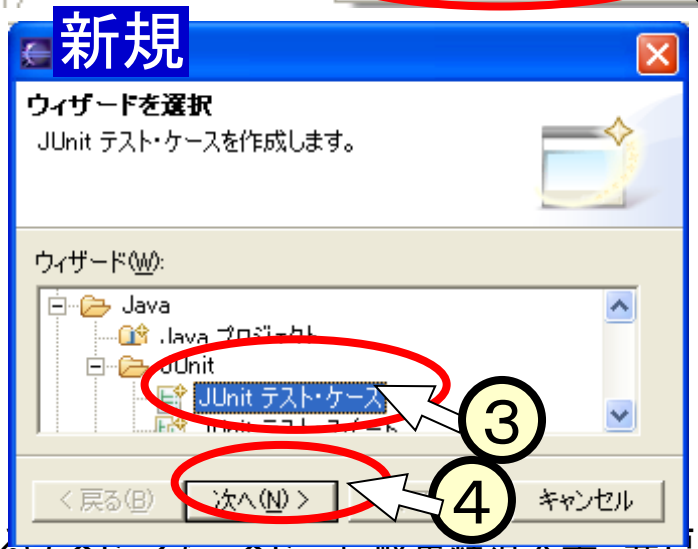
```
d testInputString() {
eString as = new ArrangeString();
Equals("abc",         as.inputString("abc"));
Equals("abced",       as.inputString("ed"));
Equals("abcedabced",as.inputString("twice"));
Equals("",            as.inputString("clear"));
Equals("xyz",         as.inputString("xyz"));
    assertEquals("bingo!",     as.inputString("xyz"));
    assertEquals("bingo!abc", as.inputString("abc"));
  }
}
```

（3．4．1）テスト・クラスの作成　－1－

■テスト対象であるArrageStringクラスのソースファイルを右クリック（①）し、ポップアップメニューから、[新規]–[その他]をクリック（②）します。



■「新規：選択」ウインドウにて、[Java]–[JUnit]–[JUnitテストケース]をクリック（③）し、[次へ]をクリック（④）します。



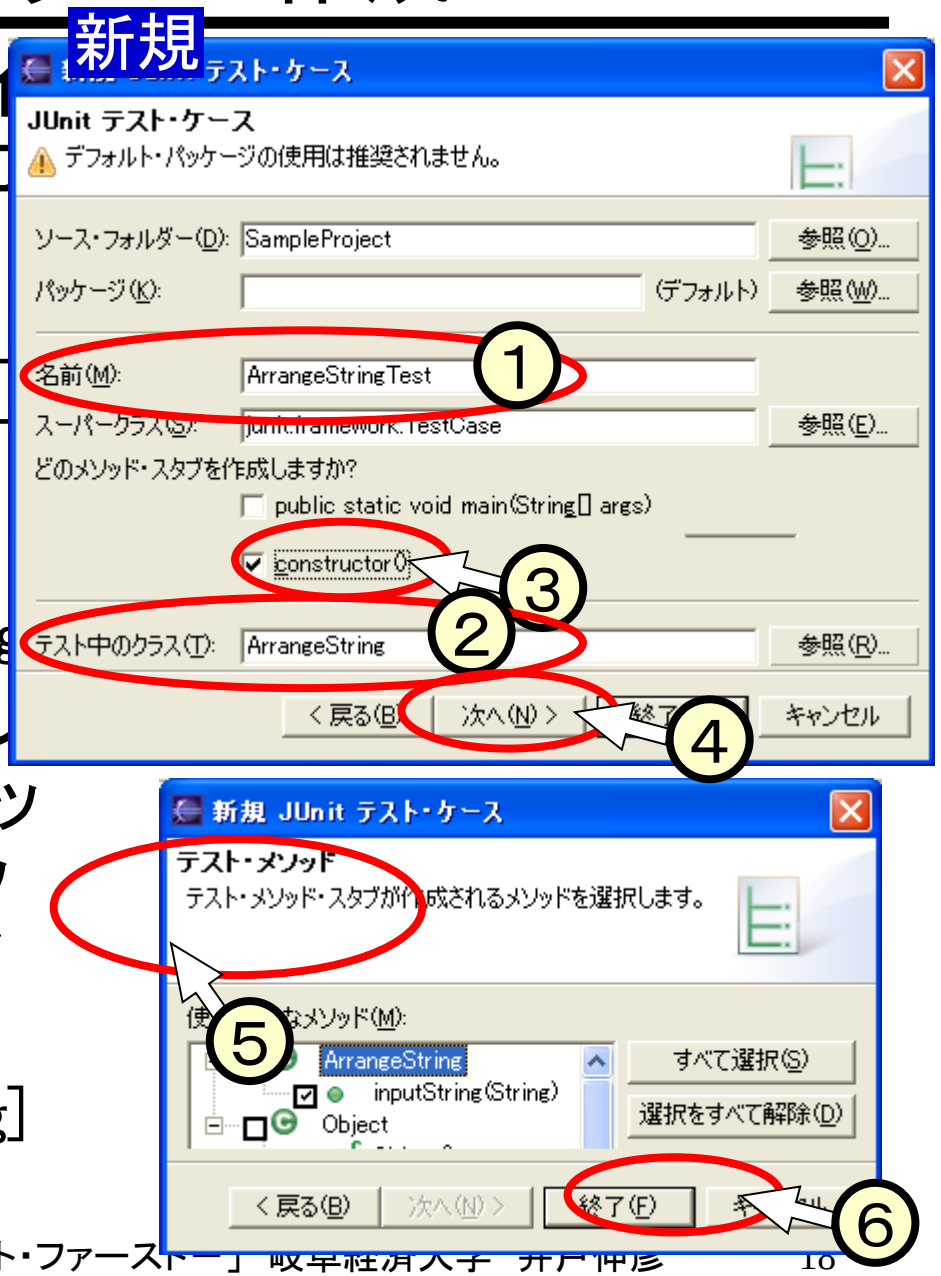
（3．4．2）テスト・クラスの作成　－2－

■「新規：Unit TestCase」ウィンドウにて、次のとおり入力されていることを確認（①、②し、“constructor”をチェック（③）して、[次へ]をクリック（④）ます。

- 名前：ArragneStringTest
- テスト中のクラス：ArrangeString

■「新規：テストケース」ウインドウにて、[使用可能なメソッド]欄の次のメソッドをチェック（⑤）し、[終了]をクリック（⑥）します。

- [ArrangeString]−[inputString]



（3．4．3）テスト・クラスの作成　－3－

■エディター・ビュー中にて、ArrangeStrigTestクラスのtestInputStringメソッドを編集していきます。

ArrangeStringTest.java

```
import junit.framework.TestCase;

/*
 * 作成日: 2004/07/03
 *
 * この生成されたコメントの挿入されるテンプレートを変更するため
 * ウィンドウ > 設定 > Java > コード生成 > コードとコメント
 */

/**
 * @author 既定
 *
 * この生成されたコメントの挿入されるテンプレートを変更するため
 * ウィンドウ > 設定 > Java > コード生成 > コードとコメント
 */
public class ArrangeStringTest extends TestCase {

    /**
     * Constructor for ArrangeStringTest.
     * @param arg0
     */
    public ArrangeStringTest(String arg0) {
        super(arg0);
    }

    public void testInputString() {
    }

}
```

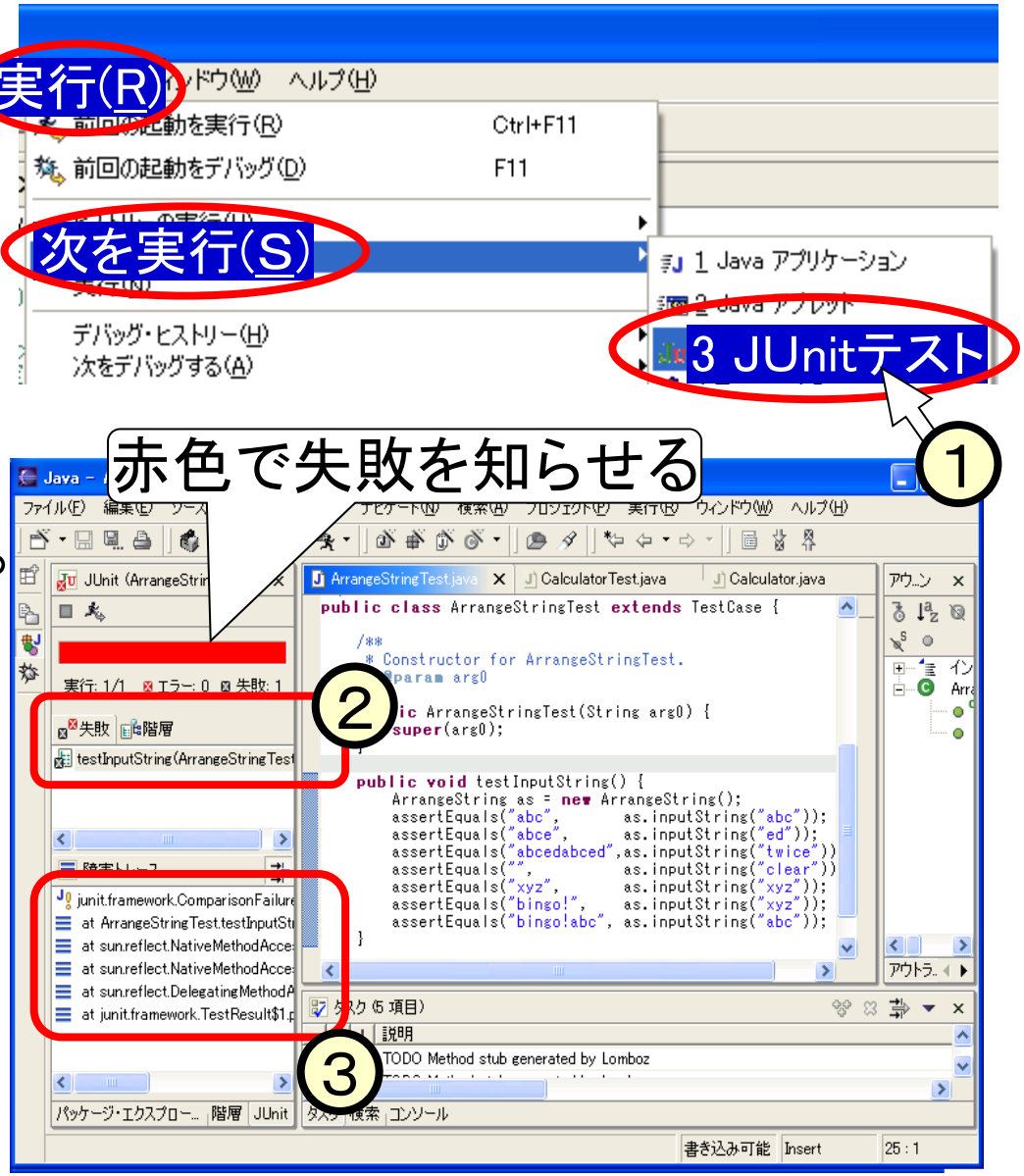
（3．5．1）テスト実行

■作成したテスト・クラス“ArrangeStringTest”を使って、テストを実行します。

■メニューから、[実行]−[次を実行]−[Junitテスト]をクリック(①)します。

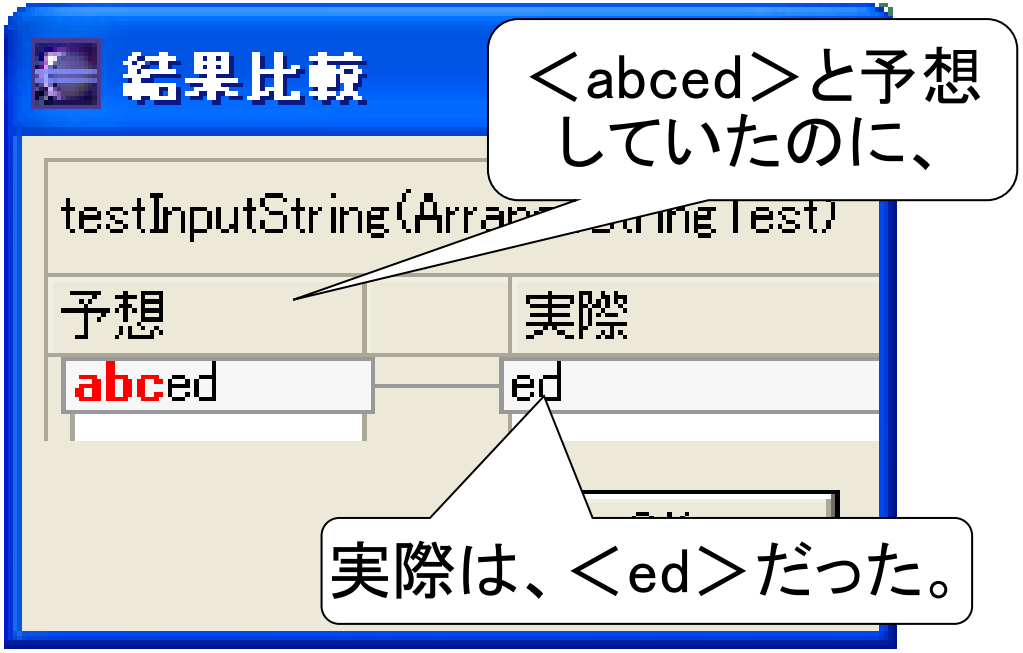
■テストが実行され、Junitビューには、次の結果が出力されます。

- 失敗したテスト(②)
  (testInputString)
- 失敗したテストのトレース(③)

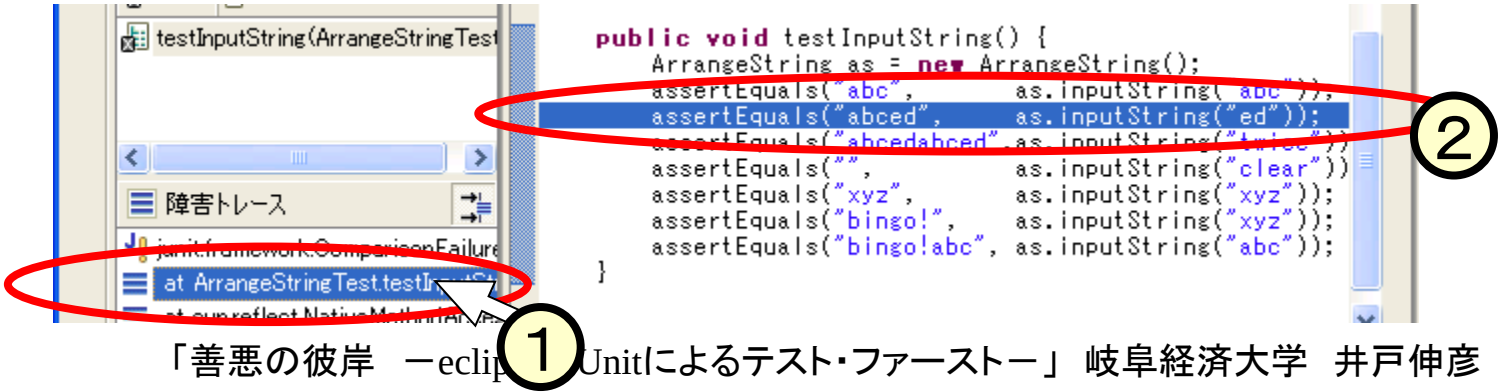


（3．5．2）失敗の内容を見る

■トレースの最初の行をダブルクリックすると、テストの結果が次のとおり表示されます。



■2行目をダブル・クリック（①）すると、エディター・ビューでは該当行がハイライト（②）されます。



（3．5．3）失敗から再編集へ

■失敗となったのは、次の行でした。

```
              :
assertEquals("abc",   as.inputString("abc"));
assertEquals("abced", as.inputString("ed"));
              :
```

■次の機能をまだ作っていないので、当然上記の行は失敗します。

- 文字列を入力するたびに、末尾に付け足す形で保存し、それを出力する。

■次のようにArrangeStringクラスを編集して、再度テストを実施します。

```
public class ArrangeString {
  private String storedString = "";
  public String inputString(String inputString){
    storedString = storedString + inputString;
    return storedString;
  }
}
```

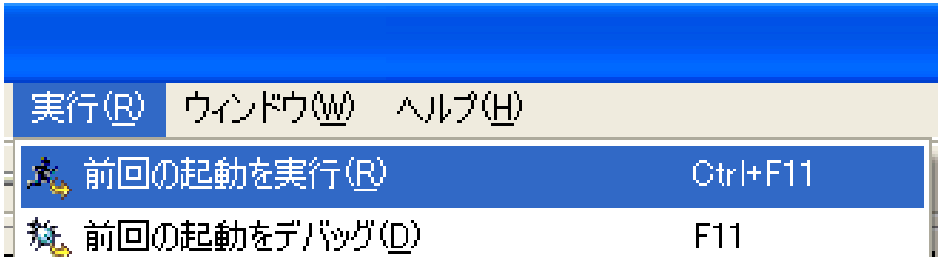
保存しておく文字列

保存しておく文字列の末尾に
入力文字列を付け足す

保存文字列を返す

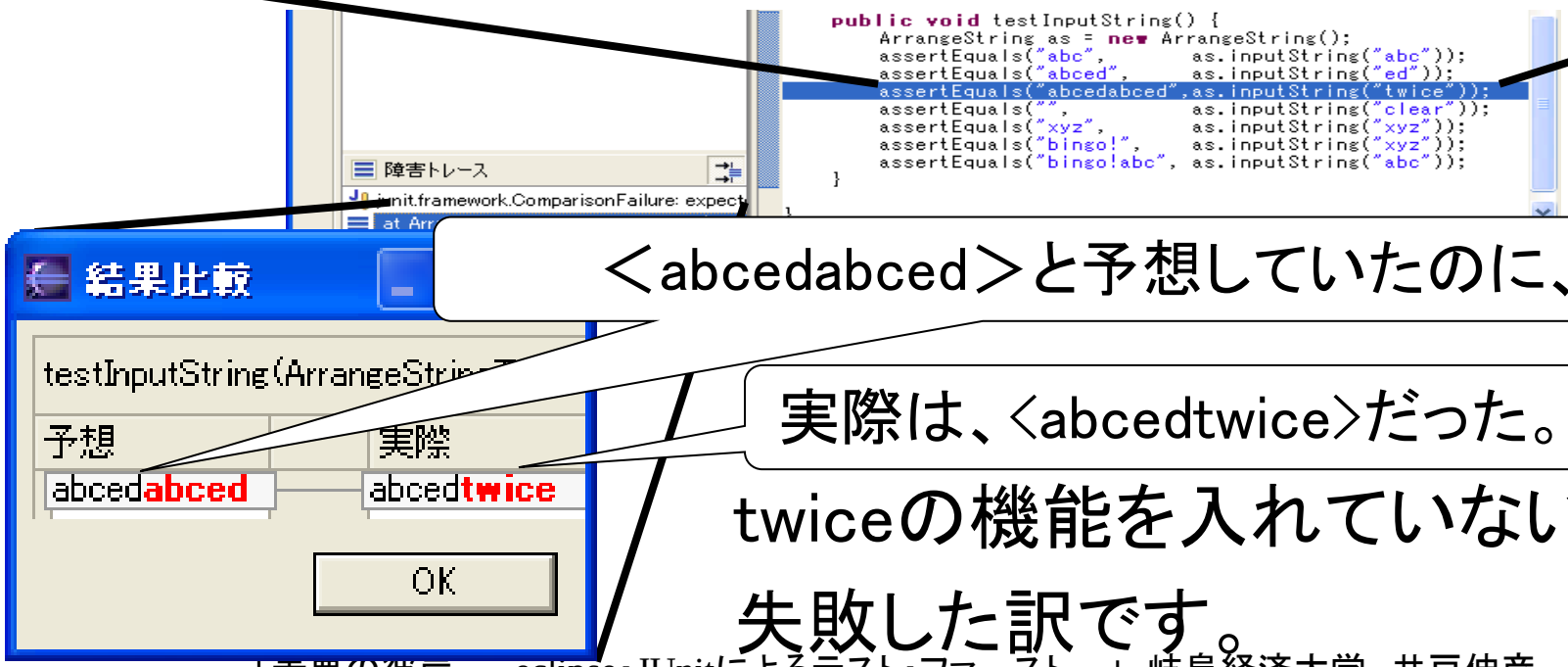
（3．5．4）2度目のテスト

■2度目のテストは、メニューから[]をクリックしても実行できます。



■今度は、次のようなところで失敗しました。

```
assertEquals("abcedabced",
             as.inputString("twice"));
```



＜abcedabced＞と予想していたのに、

実際は、<abcedtwice>だった。

twiceの機能を入れていないので、失敗した訳です。

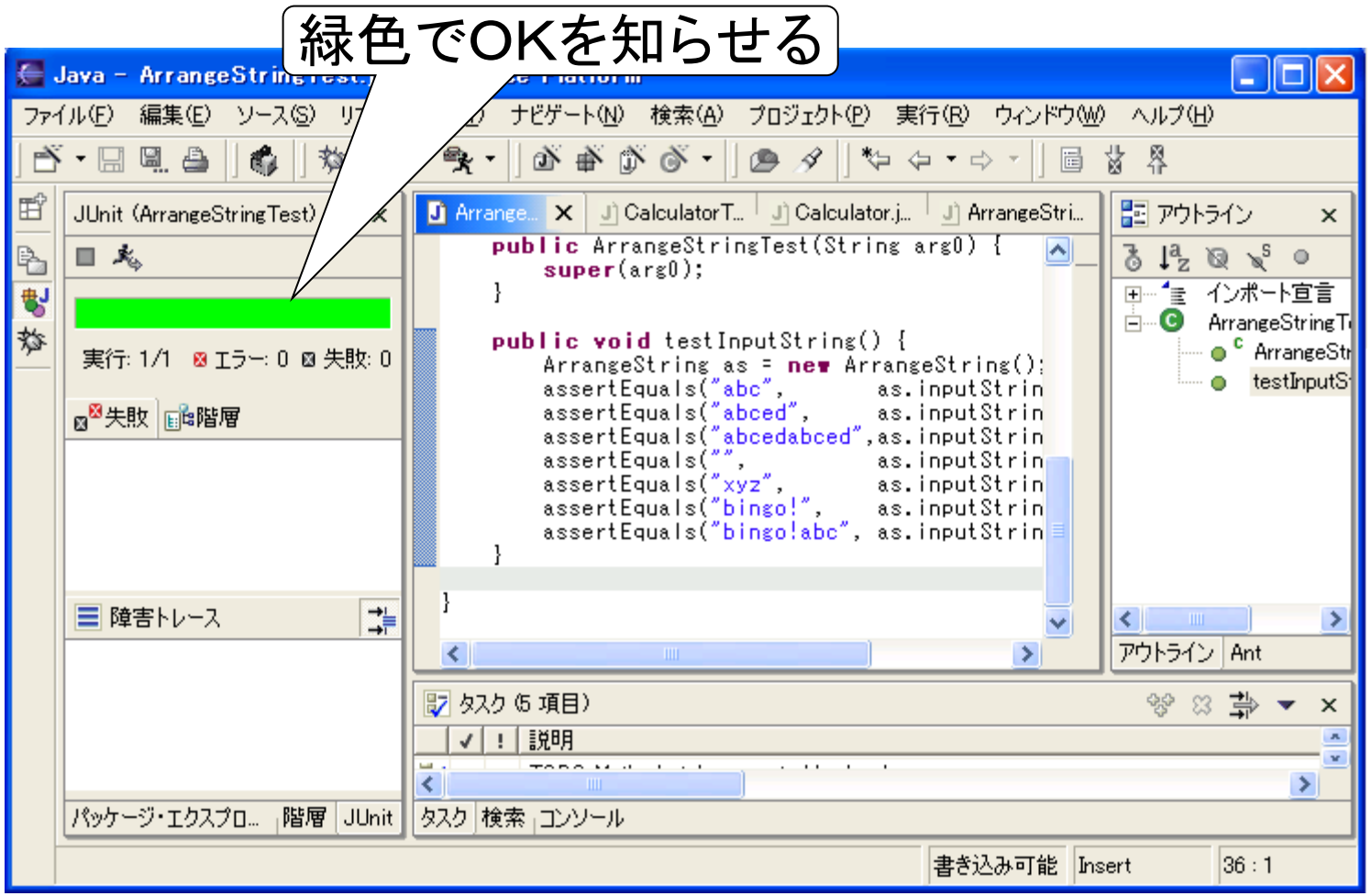
（3．5．5）2度目の編集

■次のようにArrangeStringクラスを編集して、twiceとclear、bingoに関わる機能を盛り込みます。

```
public class ArrangeString {
  private String storedString = "";
  public String inputString(String inputString){
    if(inputString.equals("twice")){
      storedString = storedString + storedString;
    }else if(inputString.equals("clear")){
      storedString = "";
    }else if(inputString.equals(storedString)){
      storedString = "bingo!";
    }else{
      storedString = storedString + inputString;
    }
    return storedString;
  }
}
```

（3．5．6）テスト実行OK！

■再度テストを実行すると、次のようにテストがOKとなったことが確かめられます。



（4）機能追加

■スライド(2)で示した仕様では、次の点が明らかではありませんでした。

①保存文字列が“twice”のときに、“twice”と入力されるとどうなるか？

②保存文字列が“clear”のときに、“clear”と入力されるとどうなるか？

```
$ java StoreString
twi
twi
ce
twice
twice
twicetwice
```

■ スライド(3.5.5)のプログラムでは、①”twicetwice”と出力、②””を出力となっていました。

■ これを次のように変更してみます。

①”bingo!”と保存して、出力する。

②保存していた文字列を棄て、空行を出力する（現状と同じ）。

（4．1）テストの追加

■新たな機能について、最初にテストを考えます。次のような追加しておきます。

```
import junit.framework.TestCase;
                    :
public void testInputString() {
    ArrangeString as = new ArrangeString();
    assertEquals("abc",         as.inputString("abc"));
                    :(中略)
    assertEquals("bingo!abc", as.inputString("abc"));
    as.inputString("clear");
    as.inputString("twi");as.inputString("ce");
    assertEquals("bingo!",     as.inputString("twice"));
  }
}
```

追加

（4．2）機能追加の実装（失敗）

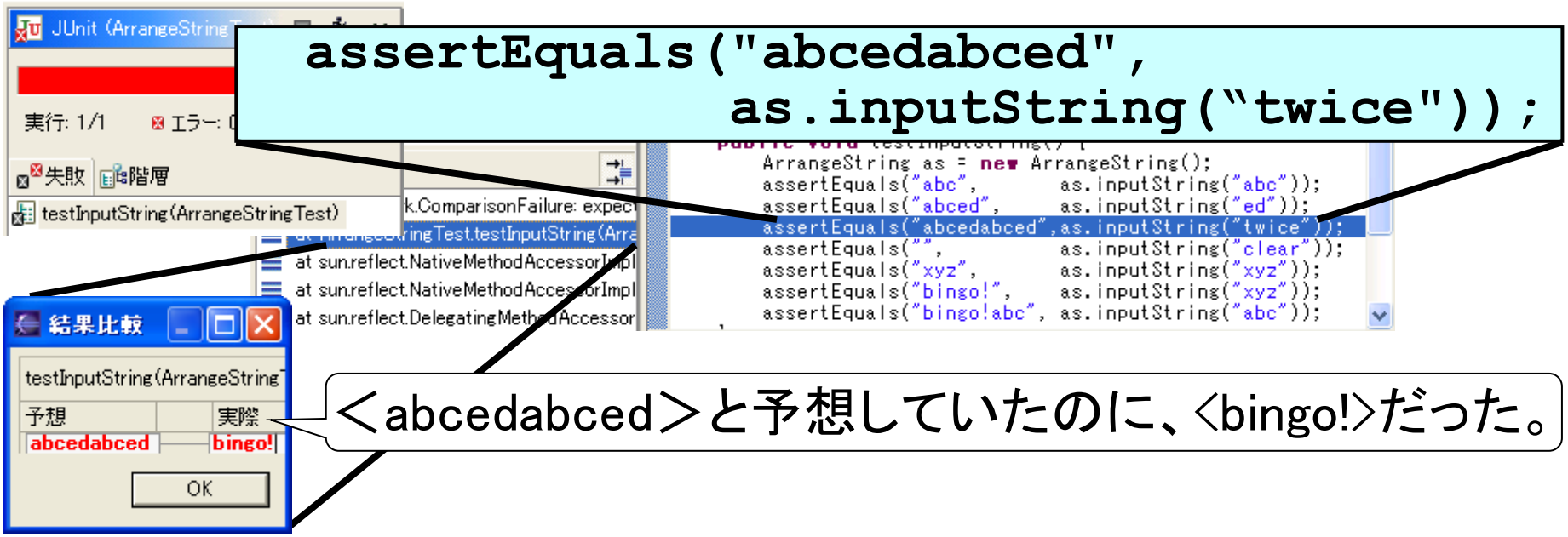
■保存文字列と入力文字列とが“twice”の場合に、保存文字列を“bingo!”にすれば良い訳ですから、次のように編集しました。

```
public class ArrangeString {
  private String storedString = "";
  public String inputString(String inputString){
    if(inputString.equals("twice") ||
       storedString.equals("twice")){
      storedString="bingo!";
    }else if(inputString.equals("twice")){
      storedString = storedString + storedString;
    }else if(inputString.equals("clear")){
      storedString = "";
    }else if(inputString.equals(storedString)){
      storedString = "bingo!";
    }else{
      storedString = storedString + inputString;
    }
    return storedString;
  }
}
```

追加

（4．3）デグレイド（degrade）

■前スライドのように編集して、再度テストを実行すると、次のように、失敗となります。

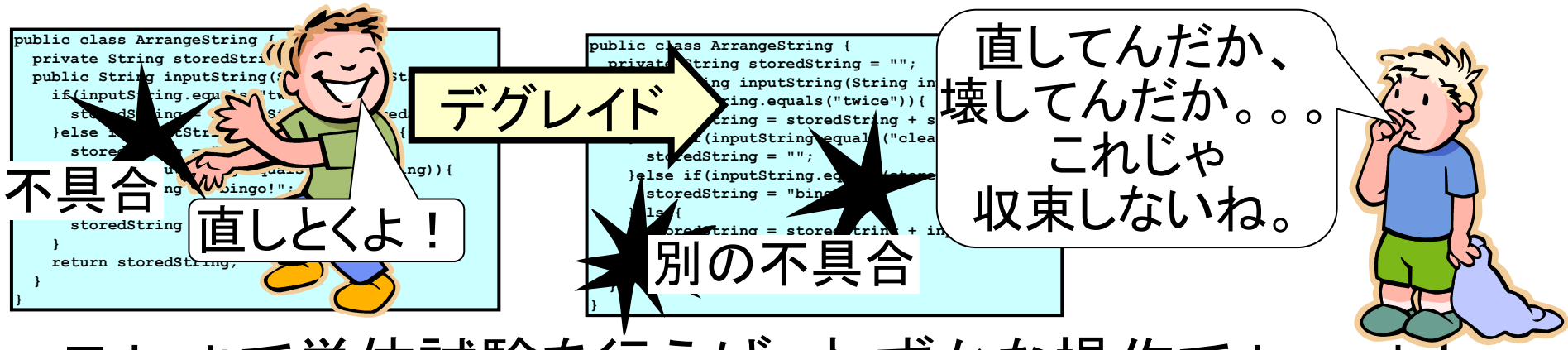


■機能を追加したつもりが、もともと正しく動いていた機能まで、動かなくなってしまったという訳です。

■このような事態を、“デグレイドした”（品質が落ちた）といいます。

（４．４）JUnitの有用性

- 当然ながら、デグレイドすることは望ましくありません。
- さらに、プログラムの間違いを発見して、これを直している時にデグレイドすると、もはや、直しているのか壊しているのかさえも怪しくなってきます。



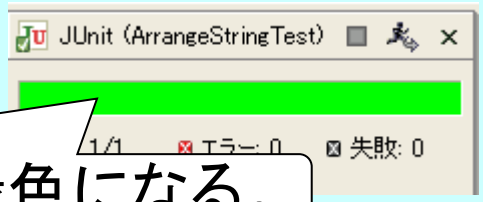
- Junitで単体試験を行えば、わずかな操作でdegradeしていないかどうかが、ほぼ確認できます（完璧に確認できる訳ではありませんが）。
- プログラムの品質を計画通りに向上させていく過程において、Junitは強力な武器となります。

(4．5)プログラムの修正

■スライド(4)の失敗を見れば、次のようにプログラムを修正すればよいことは、容易に判断がつきます。

```
public class ArrangeString {
  private String storedString = "";
  public String inputString(String inputString){
    if(inputString.equals("twice") &&
       storedString.equals("twice")){
       storedString="bingo!";
    }else if(inputString.equals("twice")){
      storedString = storedString + storedString;
    }else if(inputString.equals("clear")){
      storedString = "";
    }else if(inputString.equals(storedString)){
      storedString = "bingo!";
    }else{
      storedString = storedString + inputString;
    }
    return storedString;
  }
}
```

修正



この修正で、テスト結果は、OKを示す緑色になる。

（４．６）課題１

■次の機能追加を行い、テストを作成（ArrangeStringTestに追加）してください。

- “treetimes”と入力された場合は、保存していた文字列を3回繰り返したものを保存し、出力する。

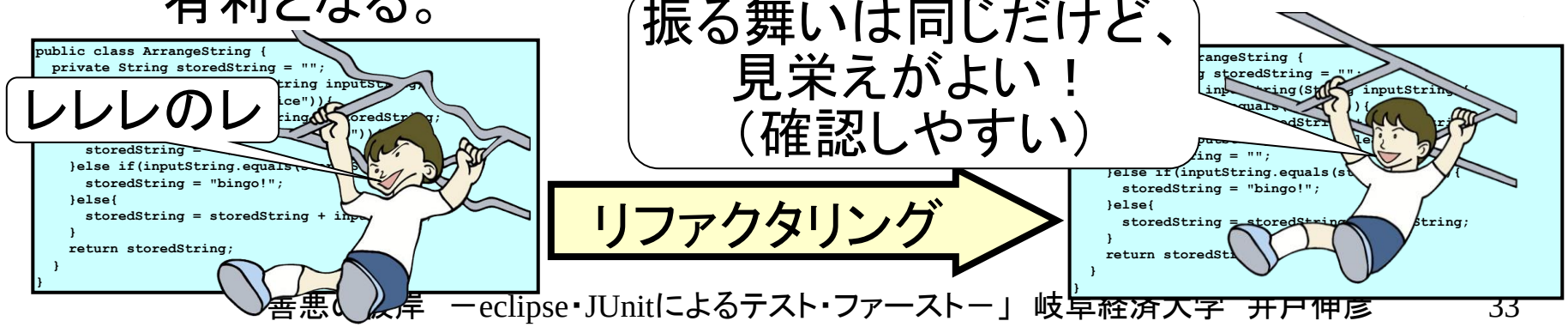
（5）リファクタリング（refactoring）

■良いプログラム

- 正しく動くだけでなく、正しく動くことが容易に確認できるプログラム。

■リファクタリング

- （意味）プログラムの振る舞いを変えることなくソースコードを変更すること。
- さまざまな理由（設計変更や間違いの修正）から、ソースコードの作成が進むにつれて、冗長で汚いものとなっていくことが多い。
- 冗長で汚くなったソースコードを、正しく動くことが容易に確認できるように変更しておけば、将来の仕様変更などの際にも有利となる。



（５．１）eclipseでのリファクタリング

■eclipseには、リファクタリングを自動で行う機能があります。これについて見ていきます。

（5．2）文字列の定数化

- スライド(4.4)のプログラムでは、“twice”や“bingo!”などの文字列を2回以上使っています。
- このような文字列は、定数として一箇所で定義しておけば、①文字列を変更する場合に便利であり、②文字列の意味づけをはっきりさせることも出来ます。

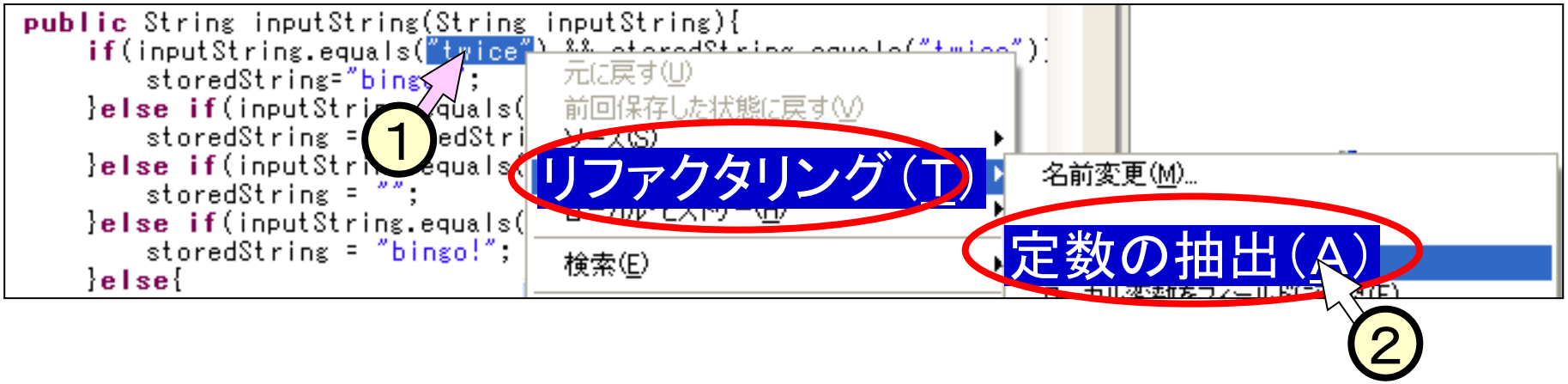
```
public class ArrangeString {
  private String storedString = "";
  public String inputString(String inputString){
    if(inputString.equals("twice") &&
       storedString.equals("twice")){
      storedString="bingo!";
    }else if(inputString.equals("twice")){
      storedString = storedString + storedString;
    }else if(inputString.equals("clear")){
      storedString = "";
    }else if(inputString.equals(storedString)){
      storedString = "bingo!";
    }else{
      storedString = storedString + input
    }
    return storedString;
  }
}
```

文字列の定数化

```
public class ArrangeString {
  private static final String COMMAND_CLEAR = "clear";
  private static final String MESSAGE_COINCIDENT = "bingo!";
  private static final String
      COMMAND_REPEAT2 = "twice";
  private String storedString = "";
  public String inputString(String inputString){
    if(inputString.equals(COMMAND_REPEAT2) &&
       storedString.equals(COMMAND_REPEAT2)){
      storedString=MESSAGE_COINCIDENT;
    }else if(inputString.equals(COMMAND_REPEAT2)){
      storedString = storedString + storedString;
    }else if(inputString.equals(COMMAND_CLEAR)){
      storedString = "";
    }else if(inputString.equals(storedString)){
      storedString = MESSAGE_COINCIDENT;
    }else{
      storedString = storedString + inputString;
    }
    return storedString;
  }
}
```

（5．2．1）手順（文字列の定数化） －1－

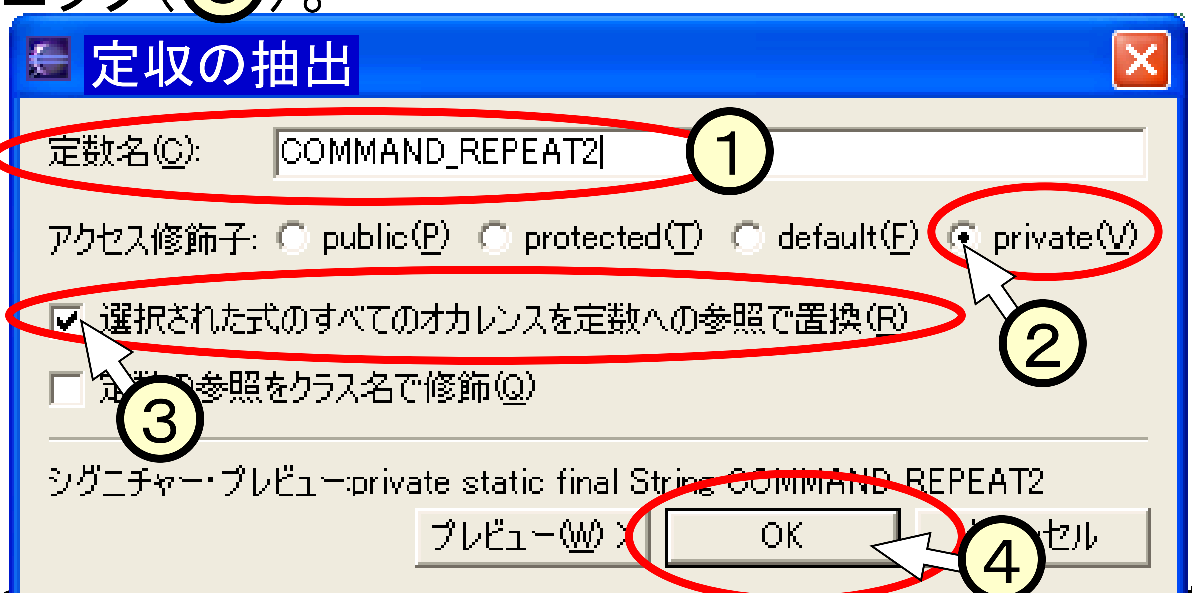
■定数化する文字列（下図では“twice”）をドラッグし、右クリック（①）して、ポップアップメニューから[リファクタリング]–[定数の抽出]をクリック（②）する。



（５．２．２）手順（文字列の定数化）　－２－

■「定数の抽出」ウインドウにて、次のように指定し、[OK]をクリック（④）する。

- 定数名：文字列の意味することが解るような命名を入力（①、図では、“COMMAND_REPEAT2”）。
- アクセス修飾子：[private]にチェック（②、クラス内からの参照のみであることを意味する）。
- [選択された式のすべてのオカレンスを定数への参照で置換]にチェック（③）。



(5. 2. 3)結果(文字列の定数化)

```
public class ArrangeString {
  private static final String COMMAND_REPEAT2 = "twice";
  private static final String COMMAND_CLEAR = "clear";
  private static final String MESSAGE_COINCIDENT= "bingo!";
  private String storedString = "";
  public String inputString(String inputString){
    if(inputString.equals(COMMAND_REPEAT2) &&
       storedString.equals(COMMAND_REPEAT2)){
      storedString=MESSAGE_COINCIDENT;
    }else if(inputString.equals(COMMAND_REPEAT2)){
      storedString = storedString + storedString;
    }else if(inputString.equals(COMMAND_CLEAR)){
      storedString = "";
    }else if(inputString.equals(storedString)){
      storedString = MESSAGE_COINCIDENT;
    }else{
      storedString = storedString + inputString;
    }
    return storedString;
  }
}
```

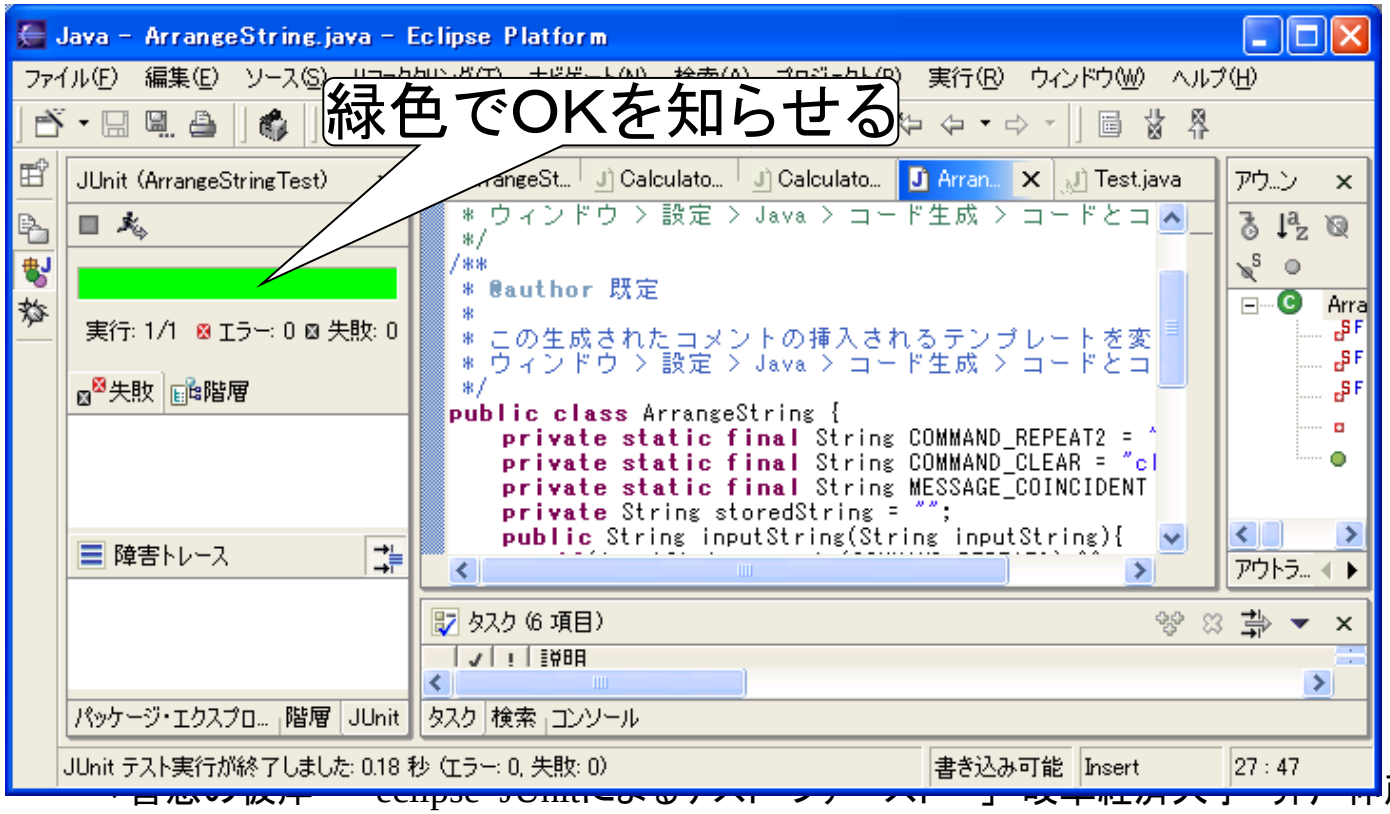
追加

xxxxx：修正部分

（5．2．4）テストの実行

■前スライドでは、文字列“bingo!”、“clear”についても定数化しています。

■定数化のリファクタリングが済んだら、テストを実施しておきます（変更するたび毎回です）。緑色が、OKを示しています。



(5．3)名前変更

■次のように名前を変更することにします。
- 変数名：
  - ◆inputString -> inputMsg
  - ◆storedString -> storedMsg
- メソッド名：inputString -> inputMsg
- クラス名：ArrangeString -> ArrangeMsg

■上記の名前の変更はあまり意味がありませんが、内容を表す解りやすい変数名とすることは、重要です。

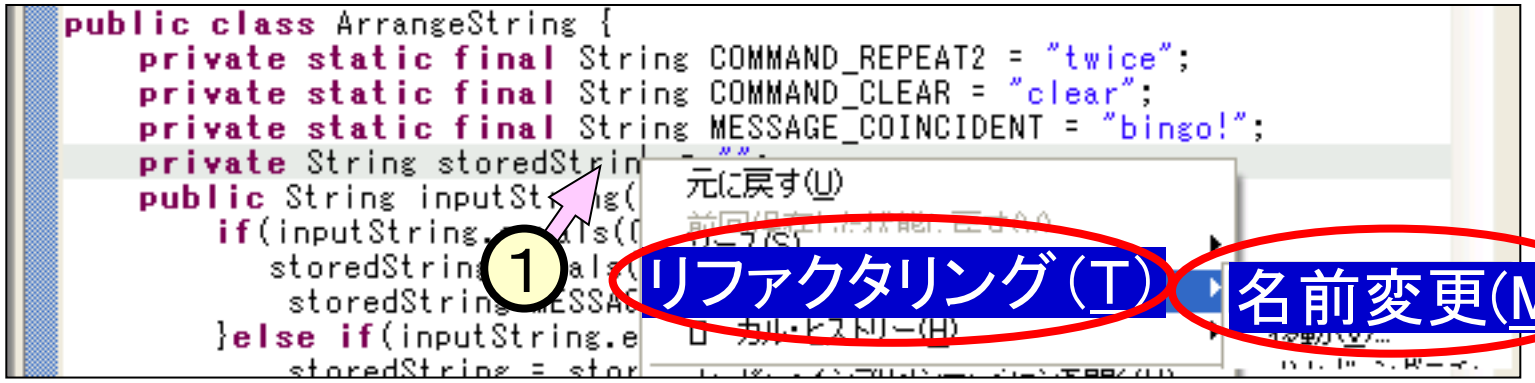
```
public class ArrangeString {
  private static final String COMMAND_REPEAT2 = "twice";
  private static final String COMMAND_CLEAR = "clear";
  private static final String MESSAGE_COINCIDENT= "bingo!";

  private String storedString = "";

  public String inputString(String
inputString){
    if(inputString.equals(COMMAND_REPEAT2) &&
       storedString.equals(COMMAND_REPEAT2)){
      storedString=MESSAGE_COINCIDENT;
    }else if(inputString.equals(COMMAND_REPE
      storedString = storedString + storedSt
    }else if(inputString.equals(COMMAND_CLEA
      storedString = "";
            :
    return storedString;
  }
}
```

名前変更

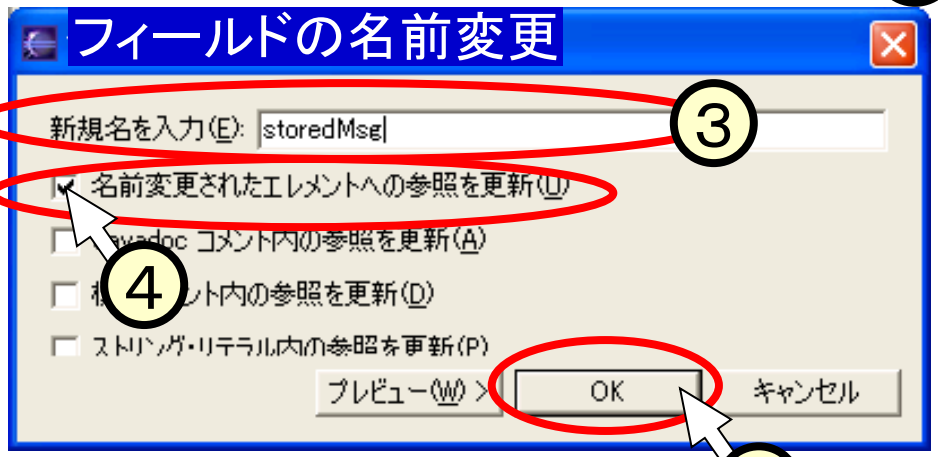
```
public class ArrangeMsg {
  private static final String COMMAND_REPEAT2 = "twice";
  private static final String COMMAND_CLEAR = "clear";
  private static final String MESSAGE_COINCIDENT = "bingo!";

  private String storedMsg = "";

  public String inputMsg(String inputMsg){
    if(inputMsg.equals(COMMAND_REPEAT2) &&
       storedMsg.equals(COMMAND_REPEAT2)){
      stor Msg=MESSAGE_COINCIDENT;
      utMsg.equals(COMMAND_REPEAT2)){
      toredMsg + storedMsg;
      putMsg.equals(COMMAND_CLEAR)){
      Msg = "";
            :
    return storedMsg;
  }
}
```

（5．3．1）手順（名前変更）－変数－

■名前を変更する変数（下図では“storedString”）を右クリック（①）して、ポップアップメニューから[リファクタリング]-[名前の変更]をクリック（②）します。

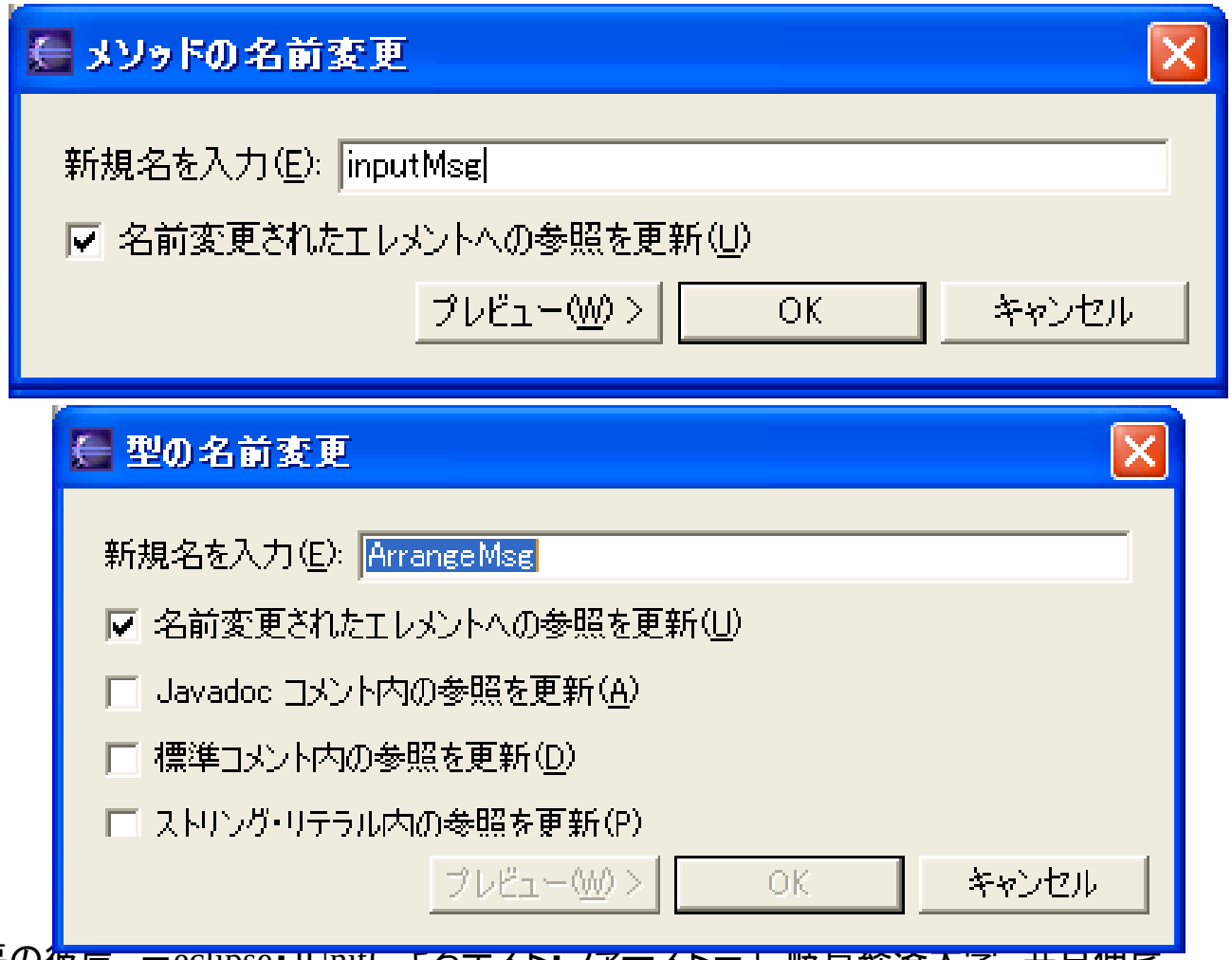


■「フィールドの名前変更」のウインドウにて、「新規名を入力」欄に新しい変数名を入力（③、ここでは“storedMsg”）し、[名前変更されたエレメントへの参照を変更]をチェック（④）し、[OK]をクリック（⑤）します。



（５．３．２）手順（名前変更）－メソッド名、クラス名－

■メソッド名、クラス名も、変数とほぼ同様の手順で名前変更が出来ます。



（5．3．3）結果（名前変更）ークラス内ー

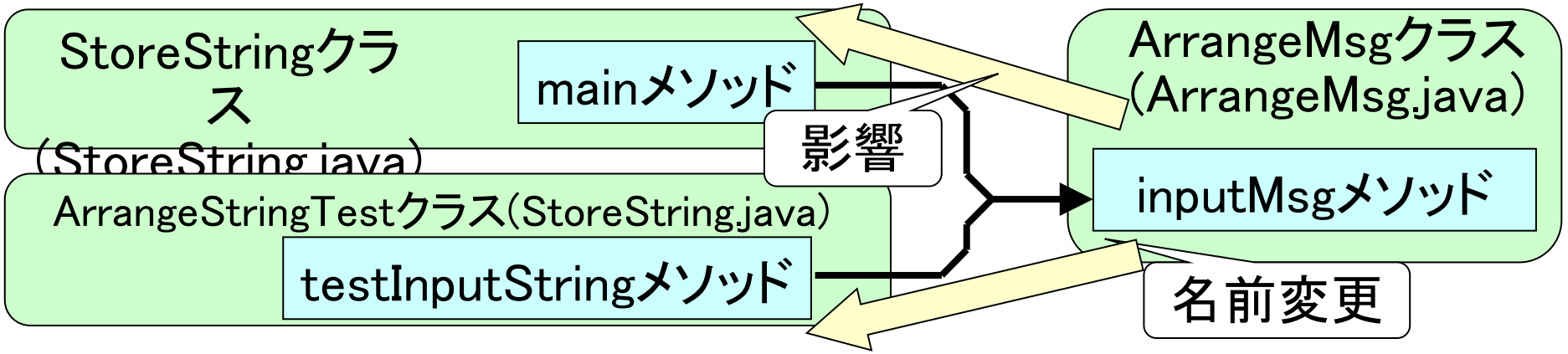
```
public class ArrangeMsg {
  private static final String COMMAND_REPEAT2 = "twice";
  private static final String COMMAND_CLEAR = "clear";
  private static final String MESSAGE_COINCIDENT = "bingo!";
  private String storedMsg = "";
  public String inputMsg(String inputMsg){
    if(inputMsg.equals(COMMAND_REPEAT2)  &&
       storedMsg.equals(COMMAND_REPEAT2)){
      storedMsg=MESSAGE_COINCIDENT;
    }else if(inputMsg.equals(COMMAND_REPEAT2)){
      storedMsg = storedMsg + storedMsg;
    }else if(inputMsg.equals(COMMAND_CLEAR)){
      storedMsg = "";
    }else if(inputMsg.equals(storedMsg)){
      storedMsg = MESSAGE_COINCIDENT;
    }else{
      storedMsg = storedMsg + inputMsg;
    }
    return storedMsg;
  }
}
```

xxxxx：修正部分

（５．３．４）結果（名前変更） －クラスの外－

■メソッドとクラスとの名前変更は、これを参照するクラスの外部にも影響します。

■eclispeでは、影響が出る外部の名前変更も自動で実行しています（ArrangeMsgクラスのファイル名も同様）。



StoreStringクラスの
mainメソッド
（StoreString.java）

```
public static void main(String[] args) throws IOException {
  BufferedReader kbd =new BufferedReader(new
                        InputStreamReader(System.in));
  ArrangeMsg arrangeString = new ArrangeMsg();
  String line;
  while((line=kbd.readLine())!=null && !line.equals("")){
    String output = arrangeString.inputMsg(line);
    System.out.println(output);
  }
}
```

（５．４．１）便宜的な機能追加

■次のような行を追加して、Java言語での算術演算子に用いられる記号（“+”、“−”、“*”、“/”、“%”）が先頭で入力された場合、入力文字列を無視して保存文字列を出力することにします。

```
public String inputMsg(String inputMsg){
  if(inputMsg.equals(COMMAND_REPEAT2)  &&
     storedMsg.equals(COMMAND_REPEAT2)){
    storedMsg=MESSAGE_COINCIDENT;
  }else if(inputMsg.equals(COMMAND_REPEAT2)){
                  :(中略)
  }else if(inputMsg.charAt(0)=='+'  ||
           inputMsg.charAt(0)=='-'  ||
           inputMsg.charAt(0)=='*'  ||
           inputMsg.charAt(0)=='/'  ||
           inputMsg.charAt(0)=='%'){
  }else{
    storedMsg = storedMsg + inputMsg;
  }
  return storedMsg;
}
```

追加

（5．4．2）メソッドの抽出

- 前スライドの追加部分は、やや長く、「Java言語での算術演算子に用いられる記号が先頭である」という条件が直ぐに解るわけではありません。意味不明な判定条件になっているといえます。
- これの条件判定をメソッドとして抽出し、どのような条件であるかが明瞭に解る名称を付けることにします。
- テストとしては、ArrangeStringTest.javaに次を追加しています（スライド(5.4.1)の機能追加よりも前にテストを作成しているとします、テスト・ファーストですから）。

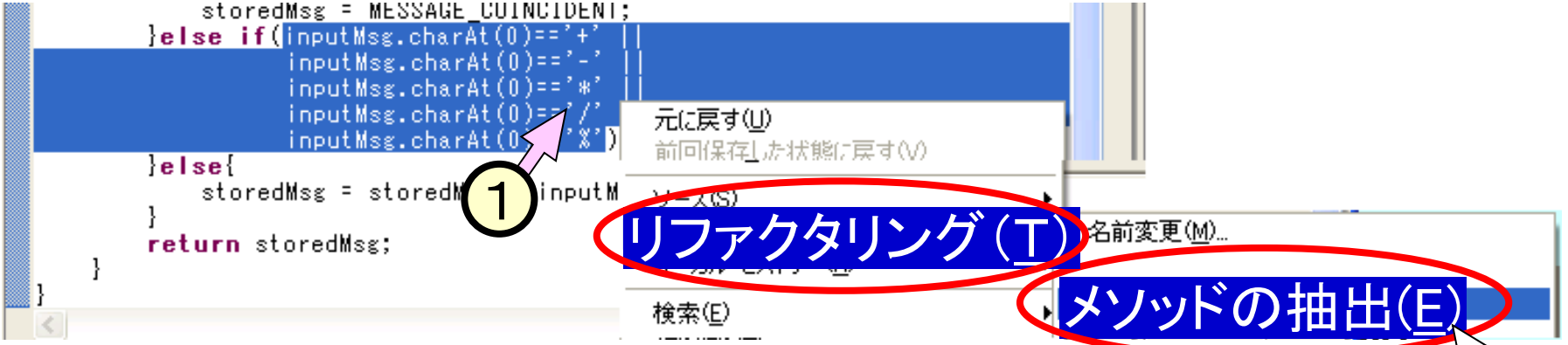
```
public void testInputString() {
  ArrangeMsg as = new ArrangeMsg();
  assertEquals("abc",        as.inputMsg("abc"));
                :
  assertEquals("bingo!",     as.inputMsg("twice"));
  assertEquals("bingo!",     as.inputMsg("+abc"));
}
```

追加

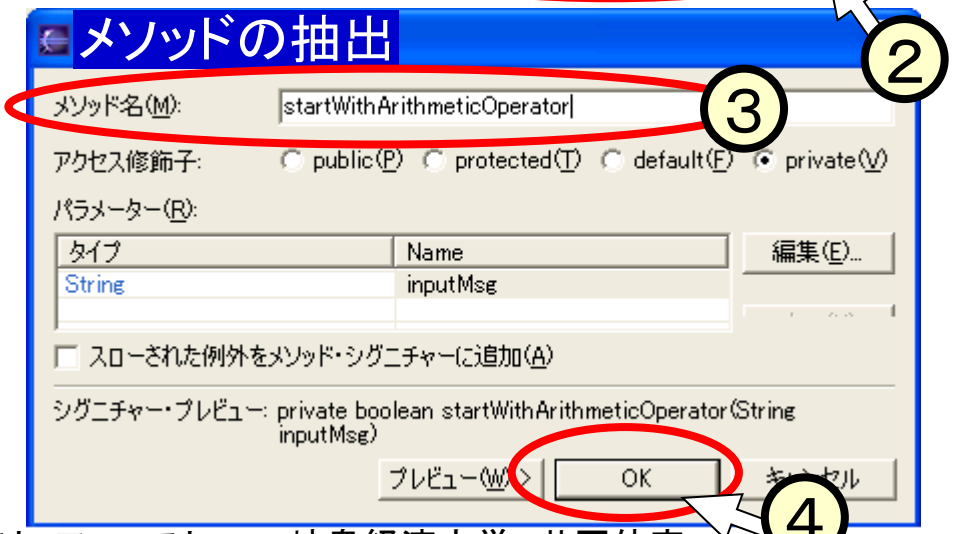
“＋”が先頭なので無視されている

（5．4．3）手順（メソッドの抽出）

■メソッドに置き換える部分（下図では“if(...)”の内部）をドラッグし、右クリック（①）して、ポップアップメニューから[リファクタリング]-[メソッドの抽出]をクリック（②）します。



■「メソッドの抽出」のウインドウにて、「メソッド名」欄に新しい変数名を入力（③、ここでは“startWithArithmeticOperator”）し、[OK]をクリック（④）します。



（5．4．4）結果（メソッドの抽出）

```
public String inputMsg(String inputMsg){
  if(inputMsg.equals(COMMAND_REPEAT2)  &&
     storedMsg.equals(COMMAND_REPEAT2)){
    storedMsg=MESSAGE_COINCIDENT;
  }else if(inputMsg.equals(COMMAND_REPEAT2)){
                  ・(中略)
  }else if(startWithArithmeticOperator(inputMsg)){
  }else{
    storedMsg = storedMsg + inputMsg;
  }
  return storedMsg;
}
private boolean startWithArithmeticOperator
                                    (String inputMsg) {
  return inputMsg.charAt(0)=='+' ||
         inputMsg.charAt(0)=='-' ||
         inputMsg.charAt(0)=='*' ||
         inputMsg.charAt(0)=='/' ||
         inputMsg.charAt(0)=='%';
}
```

変更

「inputMsgが算術オペレータで始まっている」と読める。

追加

（５．４．５）コメントについて

■昔は次のような考えが有力でした。
- プログラムには出来るだけコメントを入れて、何を行っているかを説明したほうが良い。

■現在は次のような考えが有力になっています。
- プログラムの了解性向上をコメントに頼っては駄目で、プログラム自体で理解できるようにすべき。

■上記のように「コメントによるプログラムの説明を排除したい」と考える理由は、次のようなものです。
- プログラムの修正・変更の中で、コメントは直ぐに陳腐化し、役に立つどころか、勘違いに導くケースもある。
- 陳腐化を防いでコメントを保守するには、大きな工数が掛かる。そもそもコメントはプログラムの内容とはかけ離れた記述も可能であり、保守対象として不適である。

■スライド(5.4)で示した「メソッドの抽出」の例は、上記のコメントに対する考え方に関わるものです。

（５．５）その他のリファクタリングの自動化機能

■eclipseがサポートするリファクタリングの自動化機能には、次のようなものがあります。

- 移動、メソッドシグニチャーの変更、匿名クラスをネストクラスに変換、ネストされた型をトップレベルに変換、プッシュ・ダウン、プル・アップ、インタフェースの抽出、インライン化、ローカル変数の抽出、ローカル変数をフィールドに変換、フィールドのカプセル化

■上記機能について本スライドでは触れませんが、機会があれば勉強してください。

（５．６．１）手動のリファクタリング

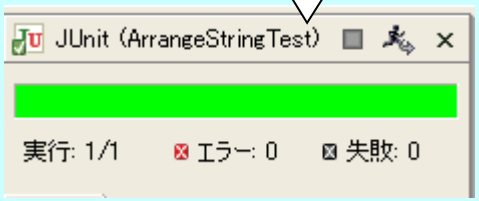
■さて、スライド(5.3.3)のソースコードは、次の点で気に入りません。
- inputMsgをCOMAND_REPEAT2と比較する箇所が2つあり、冗長で解りづらい。

■直そうか否か、迷うところです。
- もし、稼動中システムのプログラムであれば、修正しないでしょう。「正しく動いているプログラムは変更しない」というのは、ソフトウェア開発での黄金律です。
- 昔であれば、余程のことがない限り、正しく動いているプログラムは変更しなかったのですが、このスライドで紹介しているテスト・ファーストの手法では、自動化されたテストという強力な武器があります。
- プログラムの了解性を向上させる、将来の拡張を容易とするために、思い切って手動のリファクタリングを行うことにします。これは、テスト・ファーストの効用に拠るところです（ただし、JUnitで行うのは、あくまで単体テストまでです。「プログラムが振る舞いを変えない」ことを保証している訳ではありません）。

（5．6．2）プログラムの修正

```
public class ArrangeMsg {
  private static final String COMMAND_REPEAT2 = "twice";
  private static final String COMMAND_CLEAR = "clear";
  private static final String MESSAGE_COINCIDENT = "bingo!";
  private String storedMsg = "";
  public String inputMsg(String inputMsg){
    if(inputMsg.equals(COMMAND_REPEAT2)){
      if(storedMsg.equals(COMMAND_REPEAT2)){
        storedMsg = MESSAGE_COINCIDENT;
      }else{
        storedMsg = storedMsg + storedMsg;
      }
    }else if(inputMsg.equals(COMMAND_CLEAR)){
      storedMsg = "";
    }else if(inputMsg.equals(storedMsg)){
      storedMsg = MESSAGE_COINCIDENT;
    }else if(startWithArithmeticOperator(inputMsg)){
    }else{
      storedMsg = storedMsg + inputMsg;
    }
      return storedMsg;
  }
  private boolean startWithArithmeticOperator
                                        (String inputMsg) {
    return inputMsg.charAt(0)=='+' ||
           inputMsg.charAt(0)=='-' ||
           inputMsg.charAt(0)=='*' ||
           inputMsg.charAt(0)=='/' ||
           inputMsg.charAt(0)=='%';
  }
}
```

この修正で、
テスト結果は、
OKを示す緑色になる。



（５．７）課題２

■次の機能追加を行います。

- 母音小文字(‘a’,’i’,’u’,’e’,’o’)で始まる入力があった場合には、その文字を大文字に変換する。

■（１）テストを作成（ArrangeStringTestに追加）してください。

■（２）メソッドinputMsg内に機能追加を行ってください。

■（３）上記（２）で追加した機能について、メソッドの抽出によりリファクタリングを行ってください。

（5．8．1）課題3

■次のようなプログラムを、テストファーストにより作成してください。

■右図のような入出力を行うプログラムです（不明な仕様は適当に決めてください）。

- 数字列、“＋”、“＊”、“=”のみを入力では受け付ける。
- “=“の入力を契機に、入力された計算式の結果を出力する。
- “clear”と入力された場合は、保持していた数値を0とする。
- 計算結果出力直後に数字列の入力があった場合は、保持していた数値を廃棄し、入力された数字列による数値を保持する。
- 想定しない入力は無視する。

```
$ java Calculator
12
+
4
=
16
+
5
=
21
clear
0
2
*
18
=
36
^Z
```

入力した文字列

プログラムの出力文字列

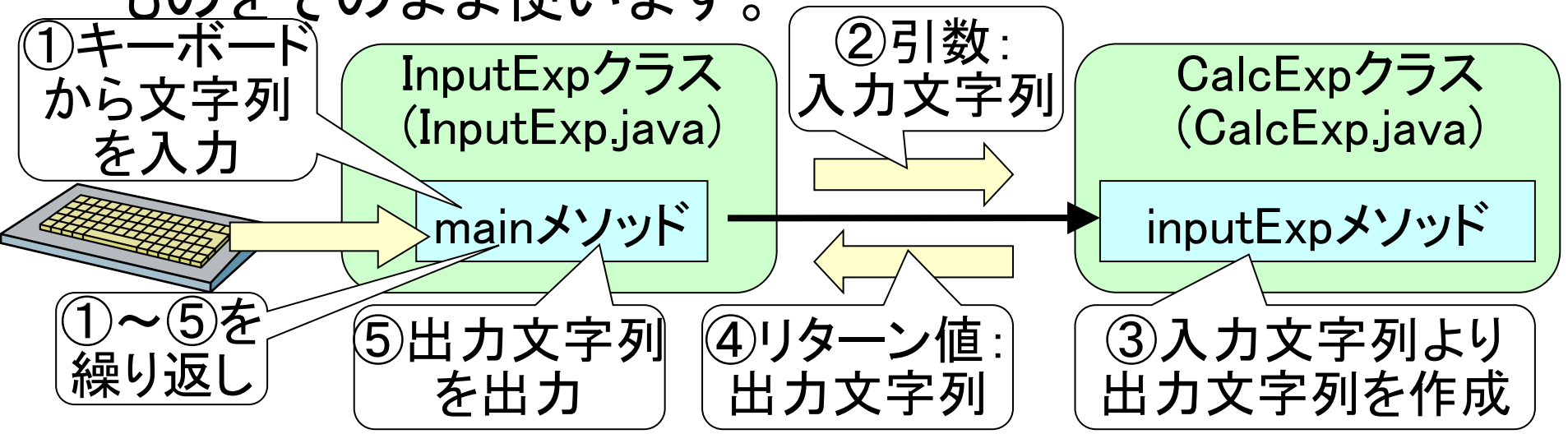
0にクリアする

プログラムの出力文字列

プログラムの出力文字列

（５．８．２）課題３　ヒント：クラス構成

- ArrangeStringクラスの際と同じように、下図に動作の概要を示した、２つのクラスでプログラムにて実現します。
- 今回作成していくプログラムは、CalcExpクラスのInputExpメソッドです。
- InputExpクラスのソースコードは、次のスライドに示すものをそのまま使います。



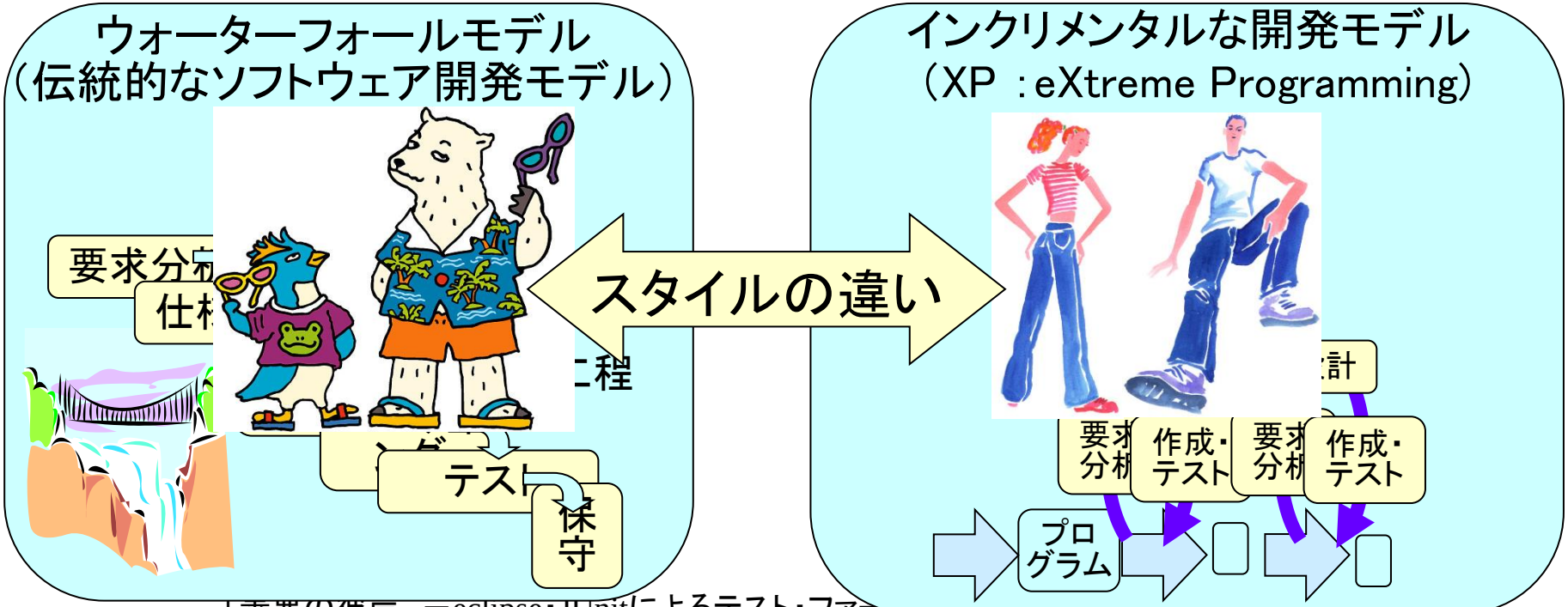
（5．8．3）課題3　InputExpクラス

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class InputExp {
  public static void main(String[] args)
  throws IOException{
    BufferedReader kbd = new BufferedReader(
        new InputStreamReader(System.in));
    CalcExp ce = new CalcExp();
    String line;
    while((line=kbd.readLine())!=null
        && !line.equals("")){
      String output = ce.inputExp(line);
      System.out.println(output);
    }
  }
}
```

（6．1）テスト・ファーストとソフトウェア開発

■テスト・ファーストは、単にテストをプログラムに先行して作成するというだけを意味するのではなく、ソフトウェア開発のスタイルに基づく意味を持っています。

■このスタイルは、XPの提唱者の一人、Kent Beckらの著作（”Test-Driven Development : By Example”）に見えるものです。



（6．2）開発サイクル

■テスト・ファーストの考えかたによれば、ソフトウェア開発は次のようなサイクルを繰り返して実施します。