WebOTX アプリケーション開発ガイド

WebOTX アプリケーション開発ガイド

バージョン： 7.1

版数： 第二版

リリース： 2008 年 1 月

目次

1.テスト・分析

1.1.TPTP

本章では、WebOTX 開発環境(Developer’s Studio)に含まれる Test and Performance Tools Platform(TPTP と呼びます)について説明します。

TPTP は、Eclipse 上で Java アプリケーションの動作に関連する統計情報を取得し、パフォーマンス計測や分析をプロファイリングするためのツールです。プロファイル・ツールの強力な視覚化機能を使用することで、アプリケーション内部のパフォーマンス及びメモリの使用に関する問題を特定することができますのでお役立てください。

Java コマンドで実行する Java アプリケーションと WebOTX EJB アプリケーションをもとに、アプリケーションのプロファイリング手順を説明します。

1.1.1.Java アプリケーションのプロファイリング準備

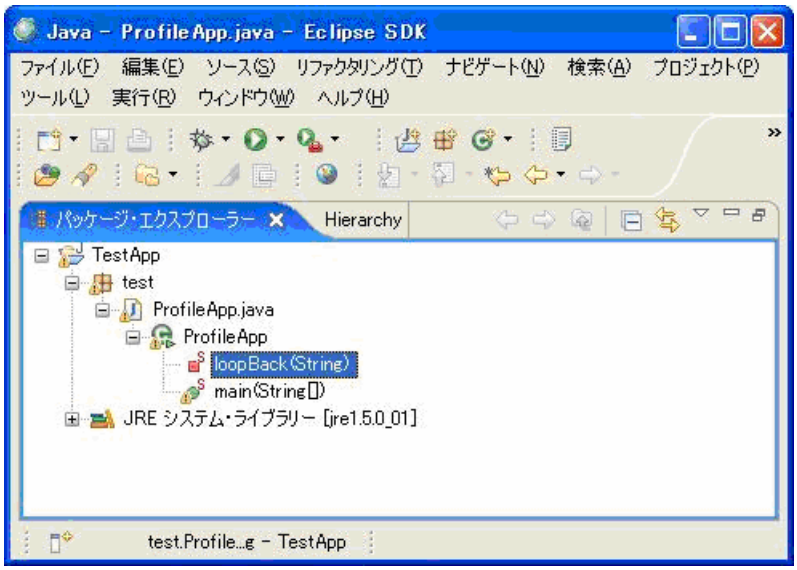
Java コマンドで実行する Java アプリケーションを使ったプロファイリング準備手順について記述します。

Developer’s Studio の起動

Developer’s Studio の起動は、「スタート」→「すべてのプログラム」→「WebOTX」→「Developer’s Studio」メニューを選択します。

Java アプリケーションの準備

今回プロファイリング用の簡単なアプリケーションとして、以下の構成の Java アプリケーション・プロジェクトを作成してください。以下の構成になるプロジェクトを例とします。



ProfileApp.java のコードは、以下の通りです。

ProfileApp.java

```
package test;

public class ProfileApp {

	/**
	 * @param args
	 */
	public static void main(String[] args) throws InterruptedException {
		// TODO 自動生成されたメソッド・スタブ
		String strInput = "ABCDE";
		String strOutput = "";
		System.out.println("Start");

		while(true){
			strOutput = loopBack(strInput);
			System.out.println("Executing loopback...");
			Thread.sleep(5000);
		}
	}

	private static String loopBack(String strInput){
		return strInput;
	}
}
```

プロファイル・ツールでプロファイリングプロジェクト作成

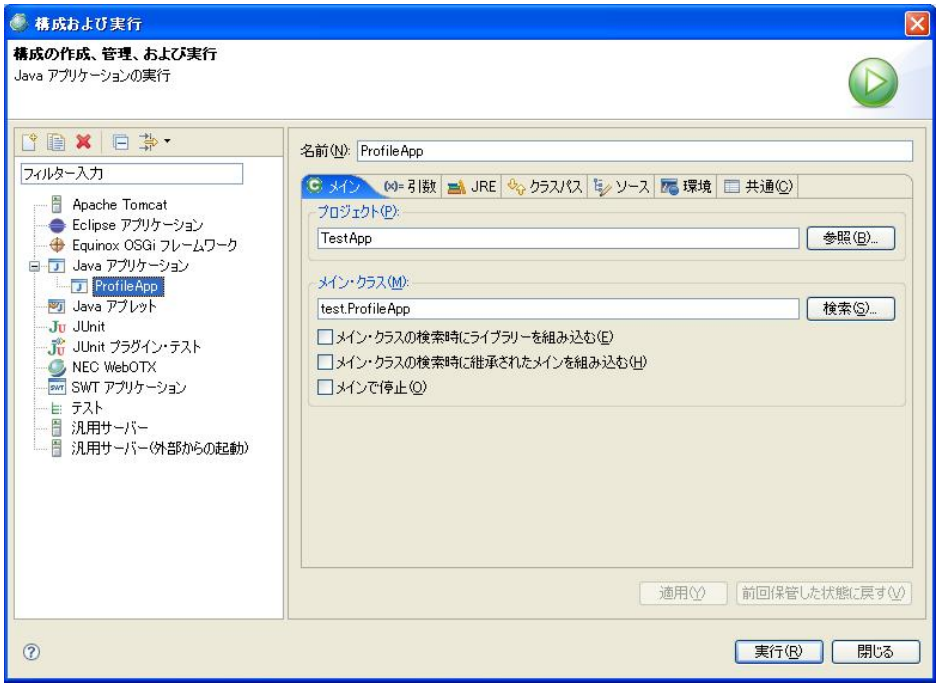
メニューから **ウィンドウ｜パースペクティブを開く｜プロファイルおよびロギング** を選択して、予めパースペクティブを切り替えておきます。

メニューから **実行｜構成および実行**を選択して[**構成および実行**]画面を表示します。

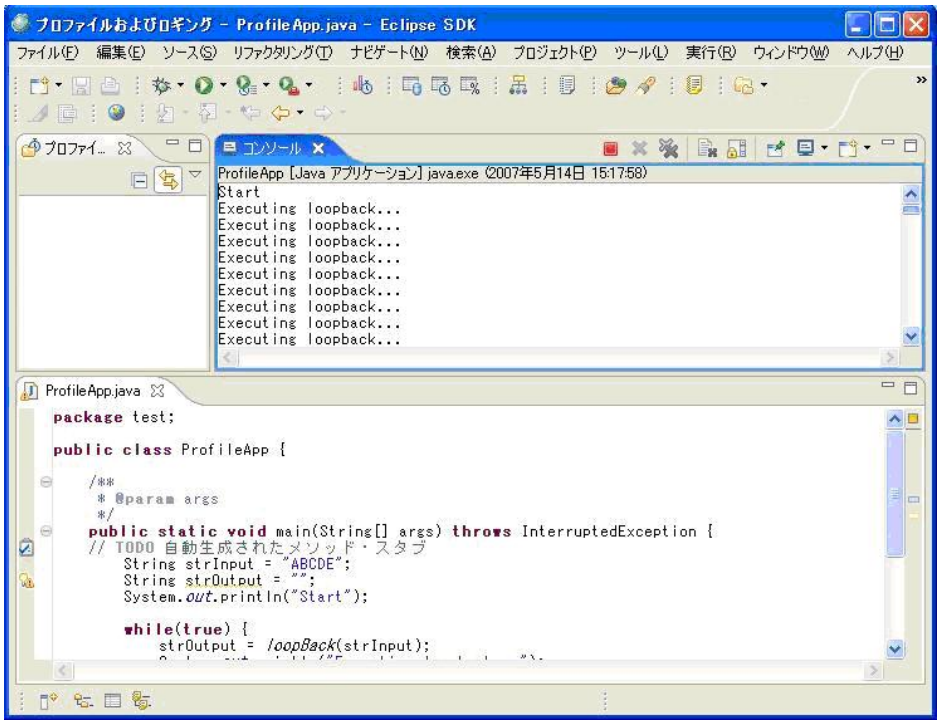
「構成」の「Java アプリケーション」を選択しダブル・クリックすることで新規実行構成を作成します。

[**メイン**]タブでは「名前」、「プロジェクト」と「メイン・クラス」に上記で作成したプログラムのプロジェクトとクラスを設定します。

設定を完了したら、[**実行**]ボタンを押してプログラムを実行します。

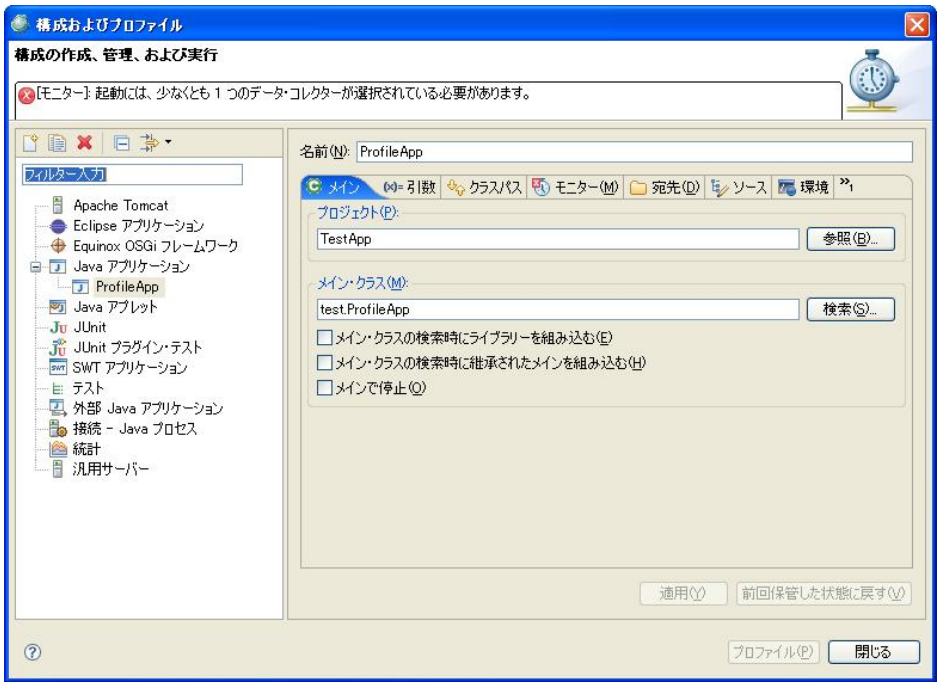


結果は以下のようになります。実行を停止させる場合、[**コンソール**]ビューの終了ボタンを押してください。

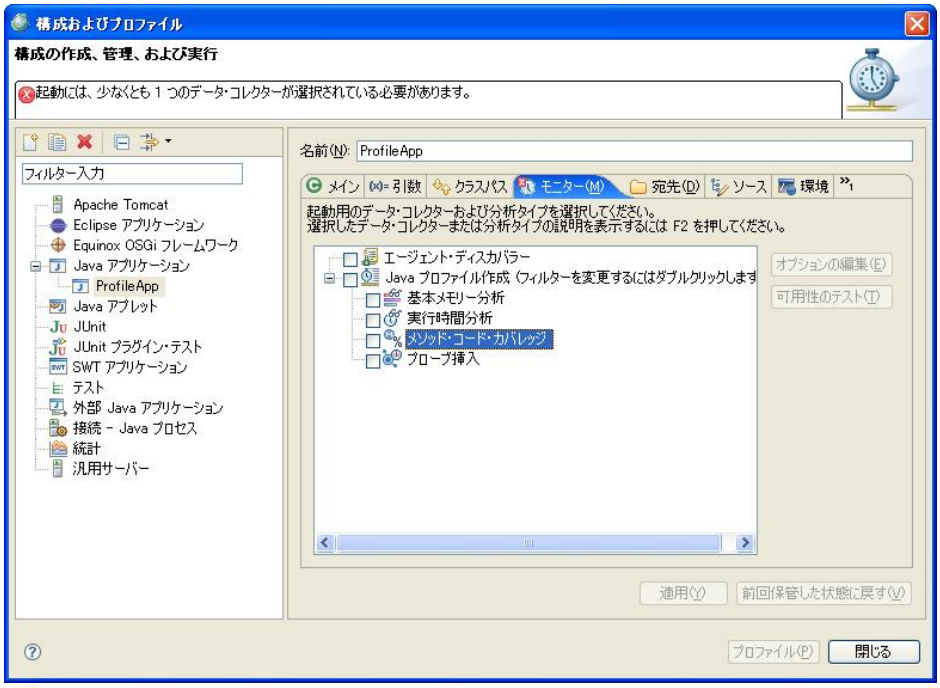


メニューから **実行｜構成およびプロファイル** を選択します。

[**構成およびプロファイル**]画面が表示されます。



[**モニター**]タブを選択します。Java プロファイル作成ノードを開く、以下の画面が表示されます



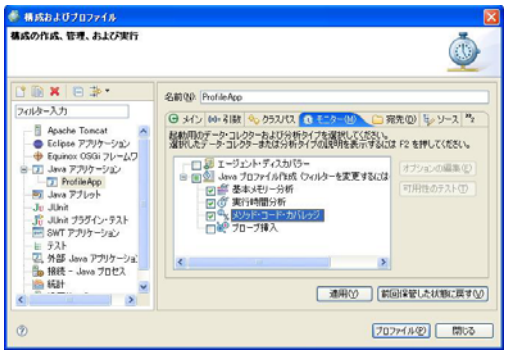
ここではどのようなプロファイルを行うかを設定します。

今回は以下の項目にチェックを入れます。

1、 基本メモリー分析

2、 実行時間分析

3、 メソッド・コード・カバレッジ

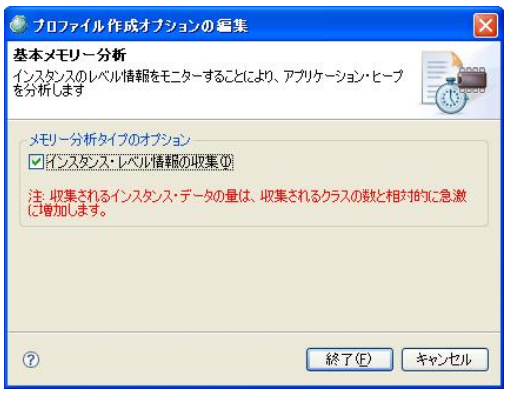


また「基本メモリー分析」と「実行時間分析」は以下のように設定します。

・基本メモリー分析

ー[**オプションの編集**]ボタンを押して、ダイアローグで、「インスタンス・レベル情報の収集」にチェックを入れます。

設定を完了したら、[**終了**]ボタンを押します。



・実行時間分析:「オプションの編集」ボタンを押して、ダイアログで、

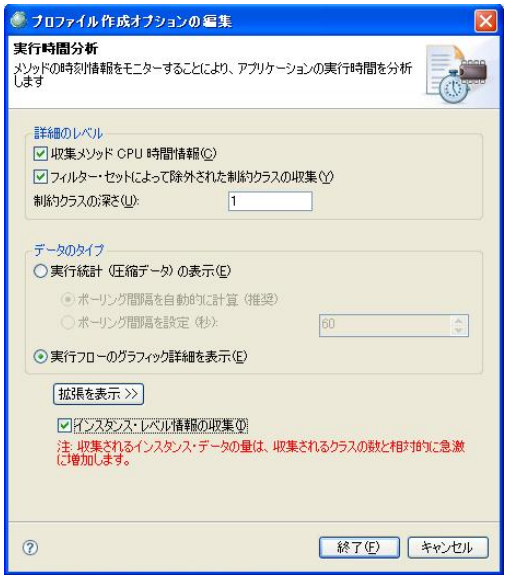
ー「収集メソッド CPU 時間情報」にチェックを入れます。

ー「フィルター・セットによって除外された制約クラスの収集」にチェックを入れて、「制約クラスの深さ」を 1 と設定します。

ー「実行フローのグラフィック詳細を表示」を選択します。

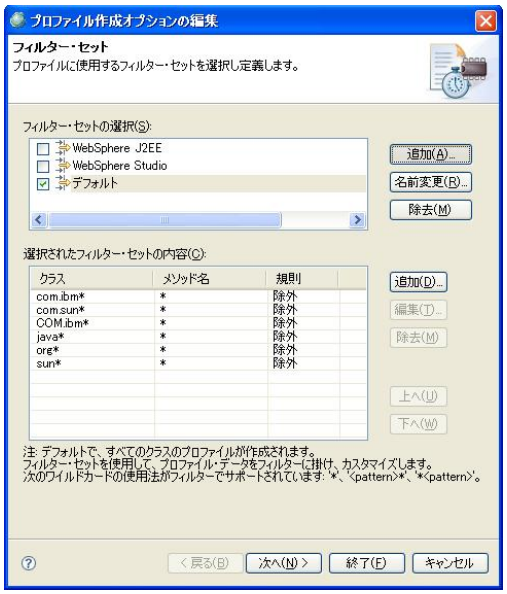
ー[**拡張を表示**]ボタンを押して、「インスタンス・レベル情報の収集」にチェックを入します。

設定を完了したら、[**終了**]ボタンを押します。



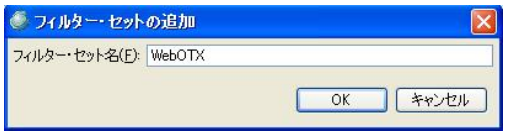
次にフィルター・セットの設定を行います。フィルター・セットはプロファイル対象を特定させるものです。

java プロファイル作成の選択で[**オプション**]ボタンを押します。「プロファイル作成オプションの編集」ダイアローグを表示します。

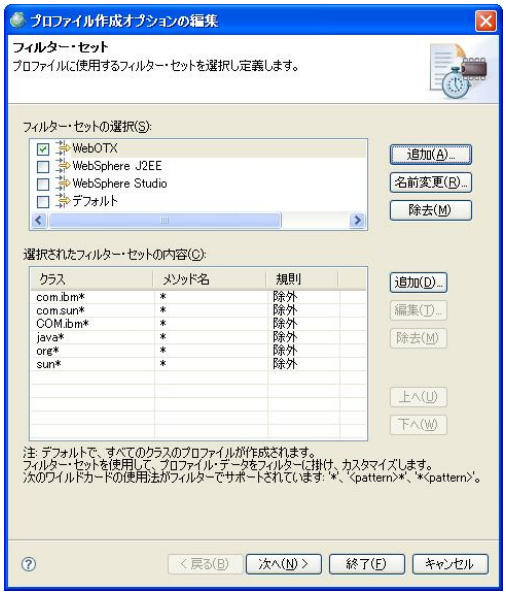


まずフィルター・セットの選択で[**追加**]ボタンを押します。

フィルターセット名を WebOTX とします。



追加した WebOTX を選択します。



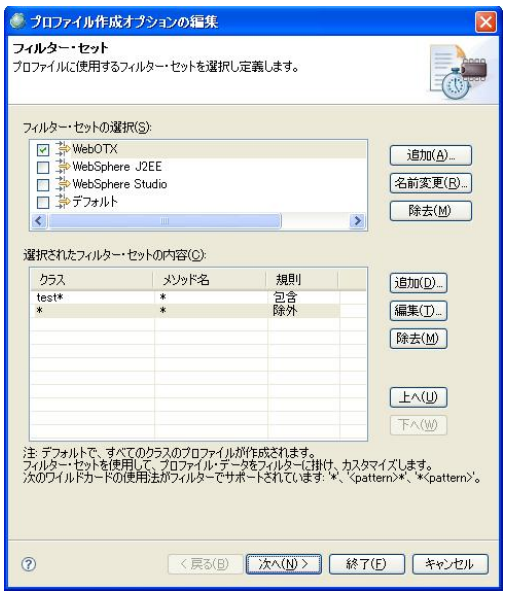
選択されたフィルター・セットの内容で、フィルター内容の設定を行います。

まず既存の項目をすべて削除します。

項目を選択して、選択されたフィルター・セットの内容の[**除去**]ボタンを押して削除します。

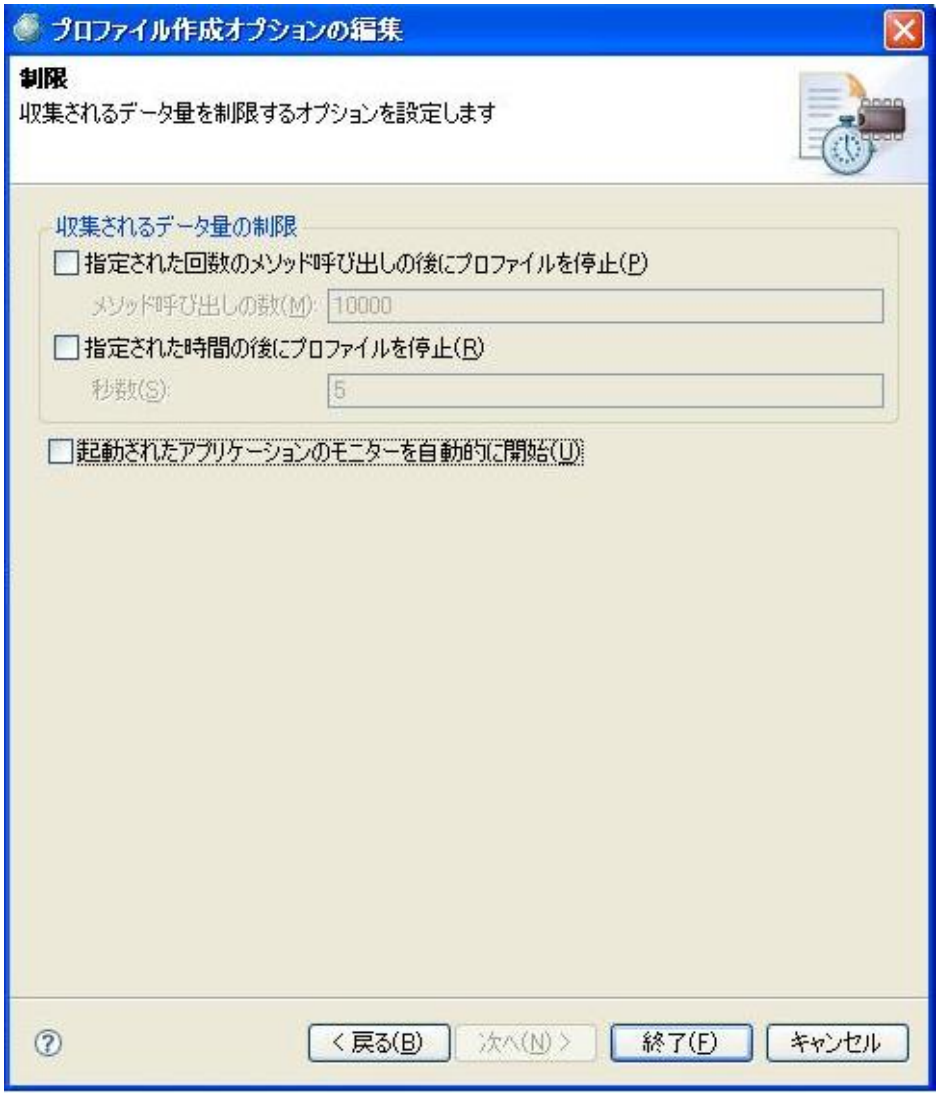
次に選択されたフィルター・セットの内容の[**追加**]ボタンを押して、[**フィルターの追加**]画面を表示して、フィルター内容を設定して[OK]ボタンを押します。今回は以下の2つのフィルターを追加します。

フィルター・セットの設定が完了したら、[**次へ**]ボタンを押します。



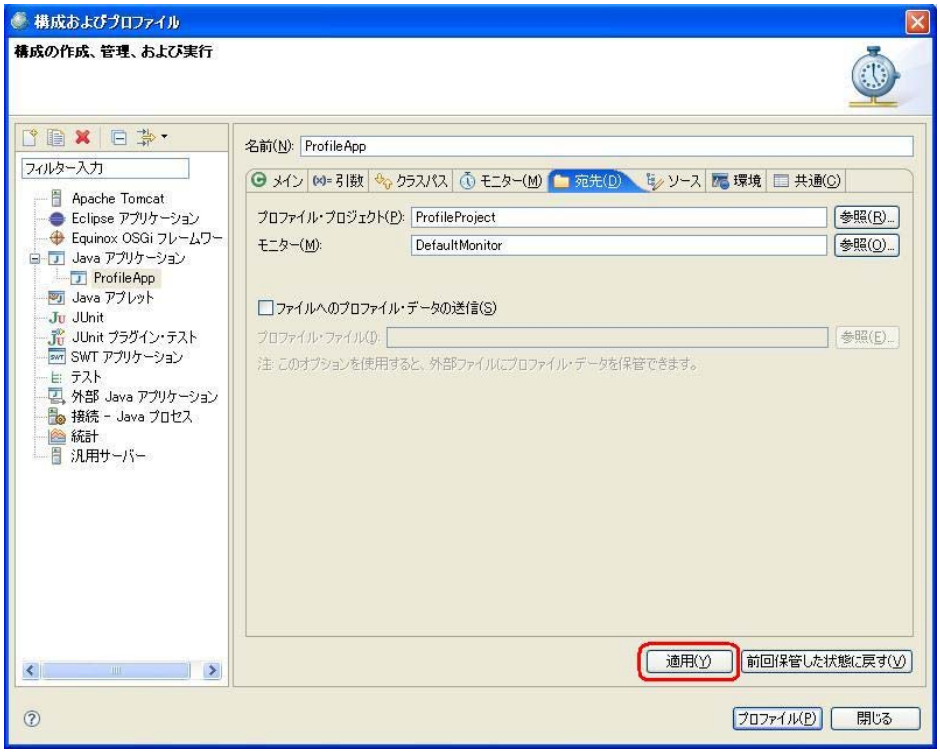
クラス	メソッド名	規則	説明
test*	*	インクルード	パッケージ profile_sample の全クラスと全メソッドを対象にする
*	*	除外	パッケージ profile_sample 以外の全クラスと全メソッドを対象外にする

[**制限**]ページでは、「指定された回数のメソッドを呼び出しの後にプロファイルの停止」や「指定された時間の後にプロファイルの停止」の設定を行えます。また「起動されたアプリケーションのモニターを自動的に開始」にチェックを入れることで、[**プロファイル**]ボタンを押してアプリケーションを起動すると同時にアプリケーションのモニターを自動的に開始します。「起動されたアプリケーションのモニターを自動的に開始」からチェックを外します。

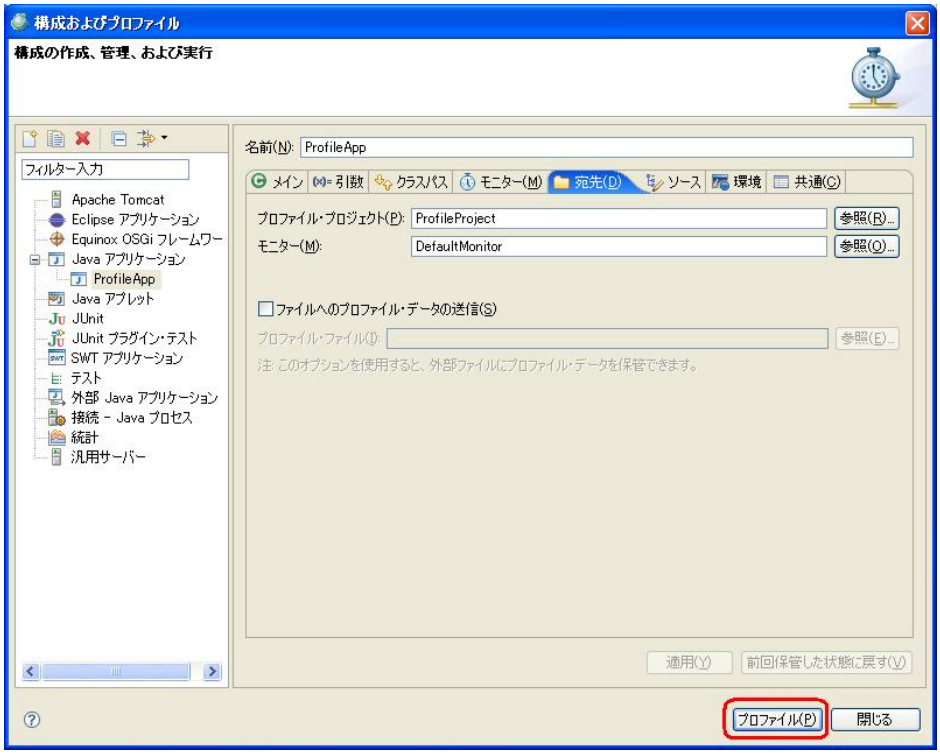


「終了」ボタンを押します

[**宛先**]タブでは、プロファイル結果の保管先プロジェクトや使用するモニターの設定などが行えます。今回は特に設定を変更しません。設定を完了すると、[**適用**]ボタンを押して実行構成を保管します。



[**構成およびプロファイル**]画面右下の[**プロファイル**]ボタンを押し、対象アプリケーションを起動させ接続します。



これで、プロファイルの準備は完了です。

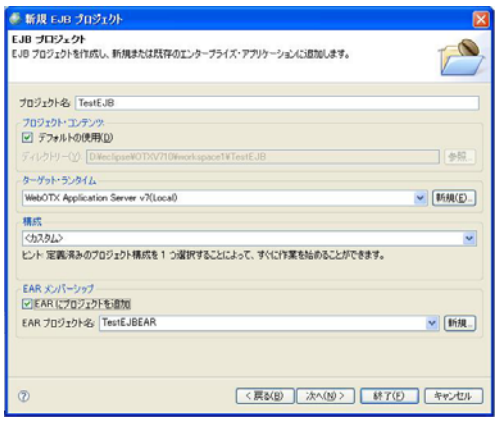
1.1.2.EJB アプリケーションのプロファイリング準備

WebOTX に配備された EJB アプリケーションを使ったプロファイリング準備手順について記述します。

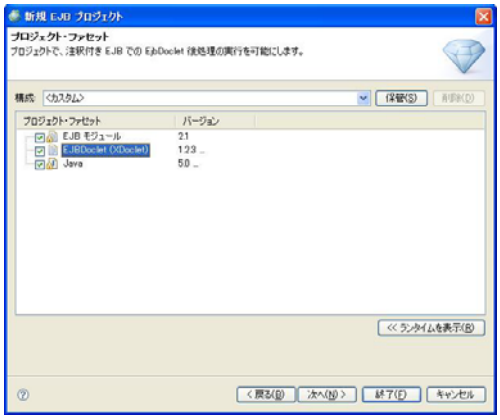
EJB アプリケーションの準備

EJB プロジェクトウィザードで作成します。

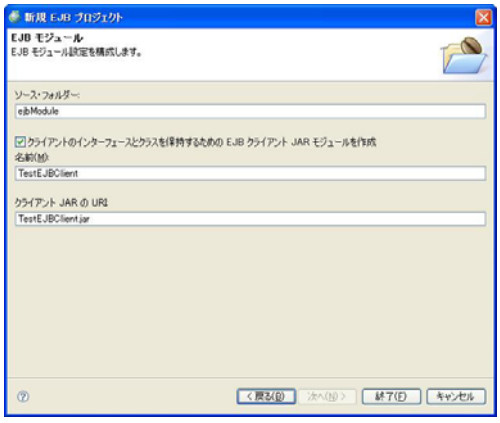
メニューから**ファイル|新規|プロジェクト**を選択し、[**EJB**]-[**EJB プロジェクト**]を選択します。プロジェクト名は、TestEJB とします。**Add project to an EAR のチェックボックスをオン**にします。EAR Project Name は、TestEJBJAR とします。



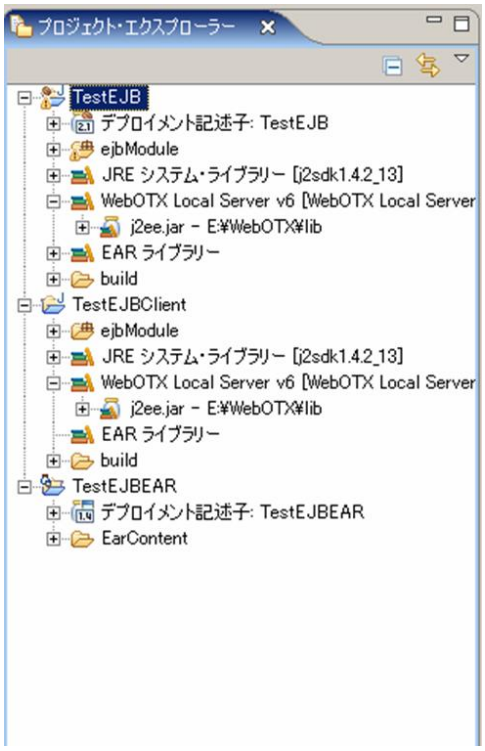
EJBDoclet (XDoclet)のチェックボックスをオンにして、[**次へ**]ボタンを押します。



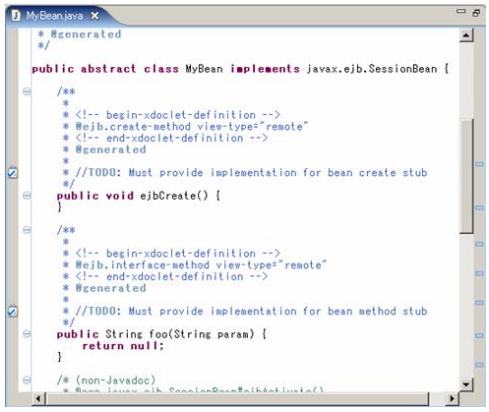
「クライアントのインタフェースとクラスを保持するための EJB クライアント JAR モジュールを作成」のチェックボックスをオンにして、[**終了**]ボタンを押します。



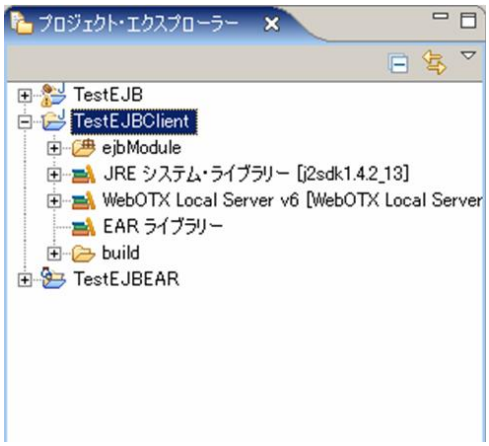
TestEJB、**TestEJBEAR**、**TestEJBClient** という名前のプロジェクトが以下の構成で生成されます。TestEJB プロジェクトでセッション Bean 作成しますが、省略します。ここでは既にステートレスセッション Bean を作成しているものとします。作成手順は**「EJB アプリケーション(WTP)」**を参照してください。



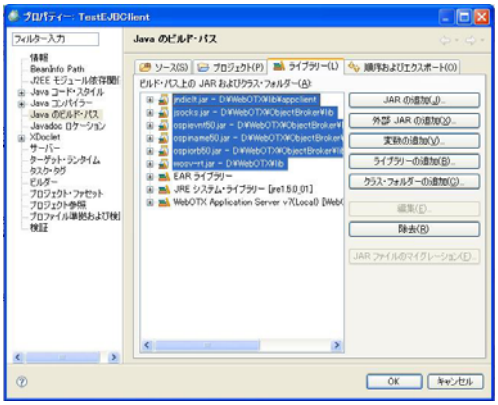
MyBean.java を開き、以下のように foo メソッドのコードの存在を確認します。プロジェクトをビルド後、TestEJBEAR プロジェクトににより EJB を配備します。手順は、**「プログラミング・開発(J2EE)」**の**「配備」**を参照してください。



TestEJBClient プロジェクトを開きます。TestEJBClient は、EJB クライアントアプリケーションです。

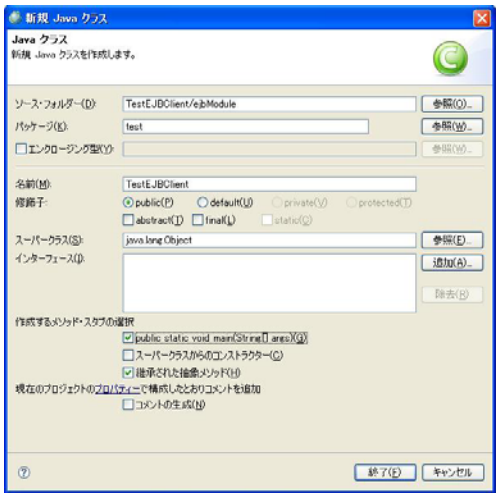


WebOTX クライアントランタイムをクラスパスに追加します。TestEJBClient のプロパティのビルドパスの設定で行います。



JAR ファイル名	機能
${INSTALL_ROOT}/lib/wosv-rt.jar	
${INSTALL_ROOT}/lib/appclient/jndiclt.jar	
${INSTALL_ROOT}/ObjectBroker/lib/ospievnt50.jar	
${INSTALL_ROOT}/ObjectBroker/lib/ospiname50.jar	
${INSTALL_ROOT}/ObjectBroker/lib/ospiorb50.jar	
${INSTALL_ROOT}/ObjectBroker/lib/jsocks.jar	

Public static void main(String[] args)のチェックボックスをオンにして、新規 Java クラスを作成します。

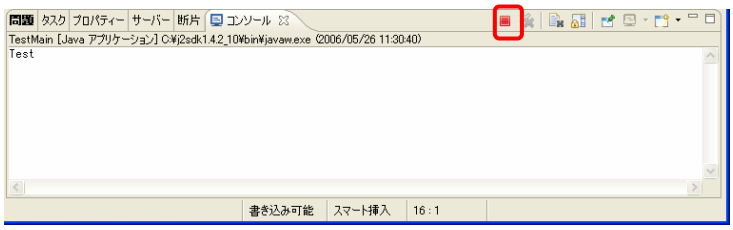


TestEJBClient.java をエディタで開きます。以下のように foo メソッドを呼び出すコードを実装します。

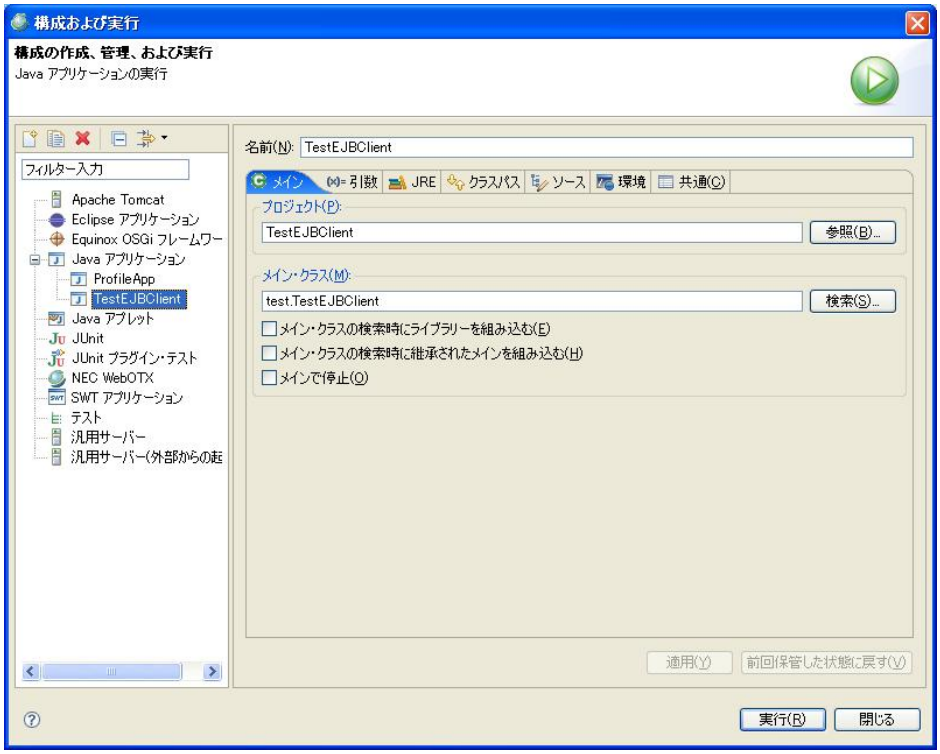
```
package test;

public class TestEJBClient {

	/**
	 * @param args
	 */
	public static void main(String[] args) throws Exception {
		// TODO 自動生成されたメソッド・スタブ
		String param = "ABCDE";
		test.MyHome home = (MyHome)MyUtil.getHome();
		My mytest = home.create();
		System.out.println("Start operation");
```

```
            while(true){
                mytest.foo(param);
                System.out.println("return opration");
                Thread.sleep(5000);
            }
        }
}
```

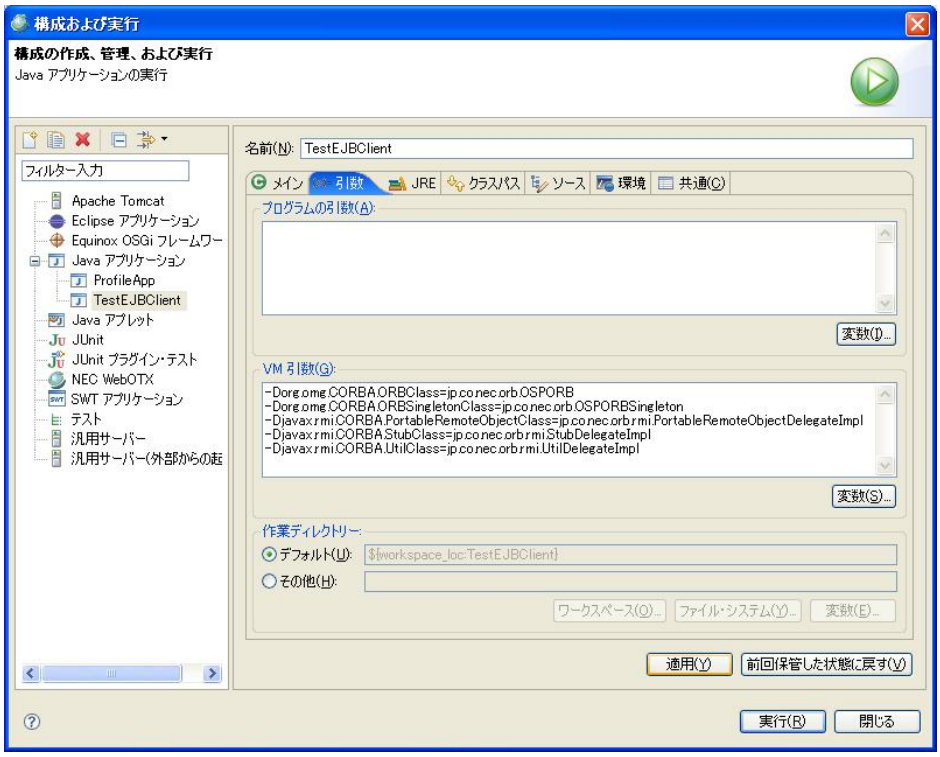
無限ループプログラムなので、停止するときはコンソールの停止ボタンを押してください。



メニューの **実行** | **構成および実行** を選択します。「Java アプリケーション」-「ダブル・クリック」で新規構成を作成し、プロジェクトを TestEJBClient とし、メイン・クラスを test.TestEJBClient と設定します。



引数タブを押します。VM 引数の設定を行います。



VM 引数の設定値は、以下のとおりです。(**-D[プロパティ名]=設定値**)

プロパティ名	設定値
org.omg.CORBA.ORBClass	jp.co.nec.orb.OSPORB
org.omg.CORBA.ORBSingletonClass	jp.co.nec.orb.OSPORBSingleton
javax.rmi.CORBA.PortableRemoteObjectClass	jp.co.nec.orb.rmi.PortableRemoteObjectDelegateImpl
javax.rmi.CORBA.StubClass	jp.co.nec.orb.rmi.StubDelegateImpl
javax.rmi.CORBA.UtilClass	jp.co.nec.orb.rmi.UtilDelegateImpl

「**適用**」ボタンを押してから、「閉じる」ボタンを押してください。

ビルドを実行してエラーが無ければ、EJB アプリケーションの準備は完了です。

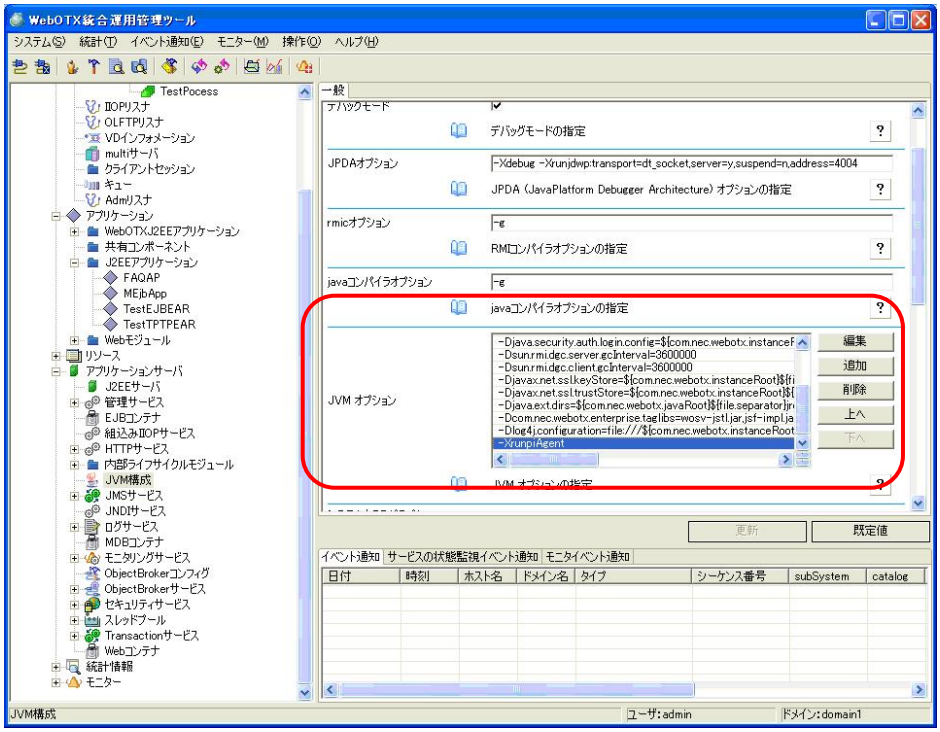
プロファイル・ツールでプロファイリングプロジェクト作成

本手順の前に**「初期設定」**の**「TPTP」**を行っていることを確認してください。

WebOTX の起動オプションに、**-XrunpiAgent** を追加します。WebOTX の各 Edition に分けて説明します。

WebOTX Standard-JEdition/Web Edition の場合:

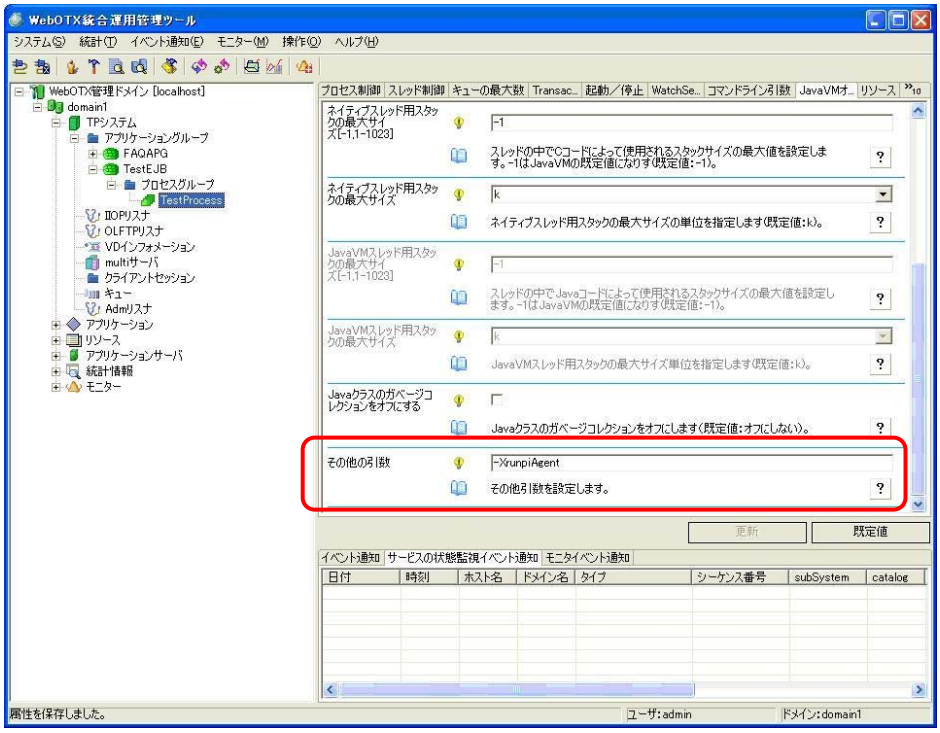
WebOTX 統合運用管理ツールを起動し、左のツリー画面から起動するドメイン(domain1 とします)|アプリケーションサーバ|JVM 構成を押し、「**JVM オプション**」に**-XrunpiAgent** を追加してください。設定後、WebOTX を再起動してください。



Web Edition は、Web アプリケーションのみが対象です。

WebOTX Starndard Edition/Enterprise Edition の場合:

WebOTX 統合運用管理ツールを起動し、配備した EJB アプリケーションのプロセスグループを押します。右画面の、「**JavaVM オプション**」タブを選択し、「**その他の引数**」に、**-XrunpiAgent** を追加してください。「設定後、WebOTX を再起動してください」



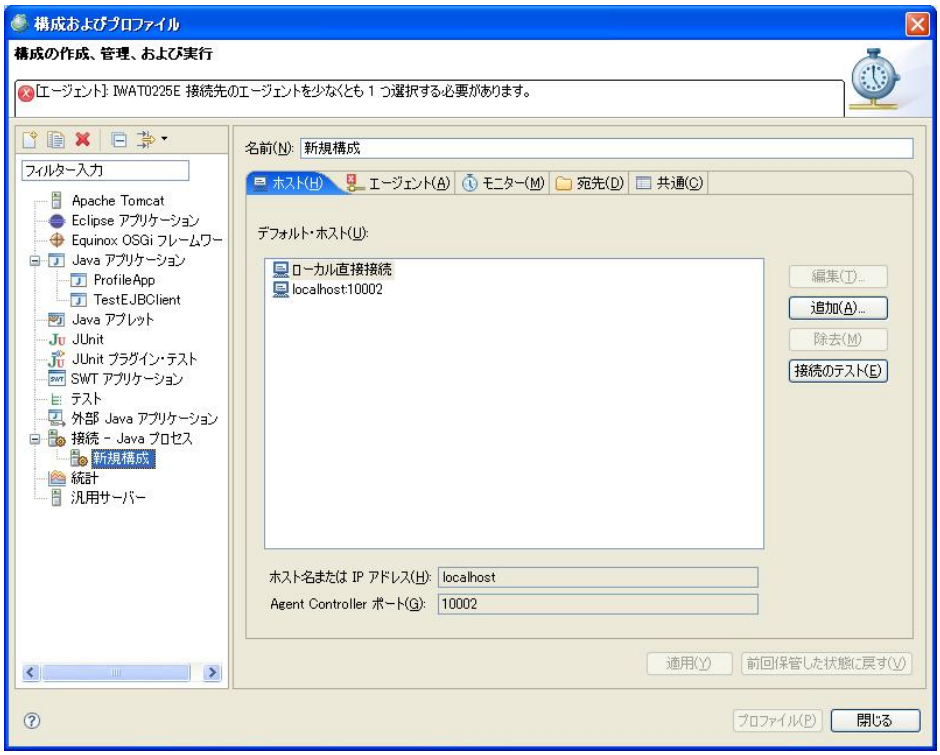
Eclipse のメニューから **実行｜構成およびプロファイル** を選択します。

[**構成およびプロファイル**]画面で、「**構成**」から**接続－Java プロセス**を選択して、ダブル・クリックして新規プロファイル構成を追加します。

「**名前**」は、新規構成とします。[**ホスト**]タブで選択されているホストの確認をします。

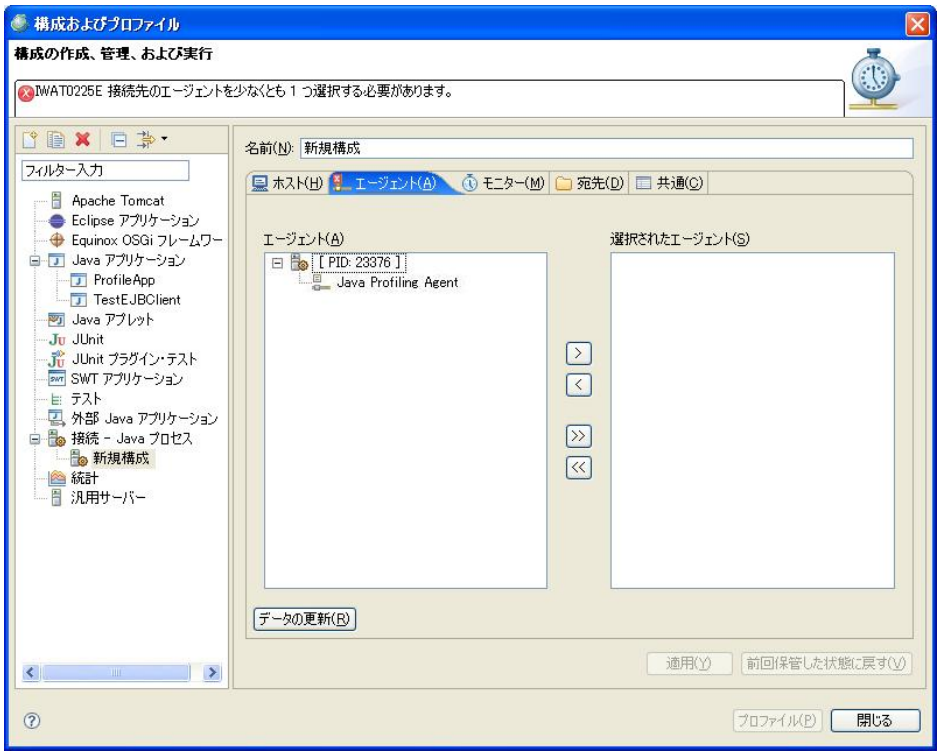
注意：TPTP Agent Controller サービスあれば及びこのサービス実施している時、

「localhost:10002」を選らんでください、でなければ、「ローカル直接接続」を選んでください。

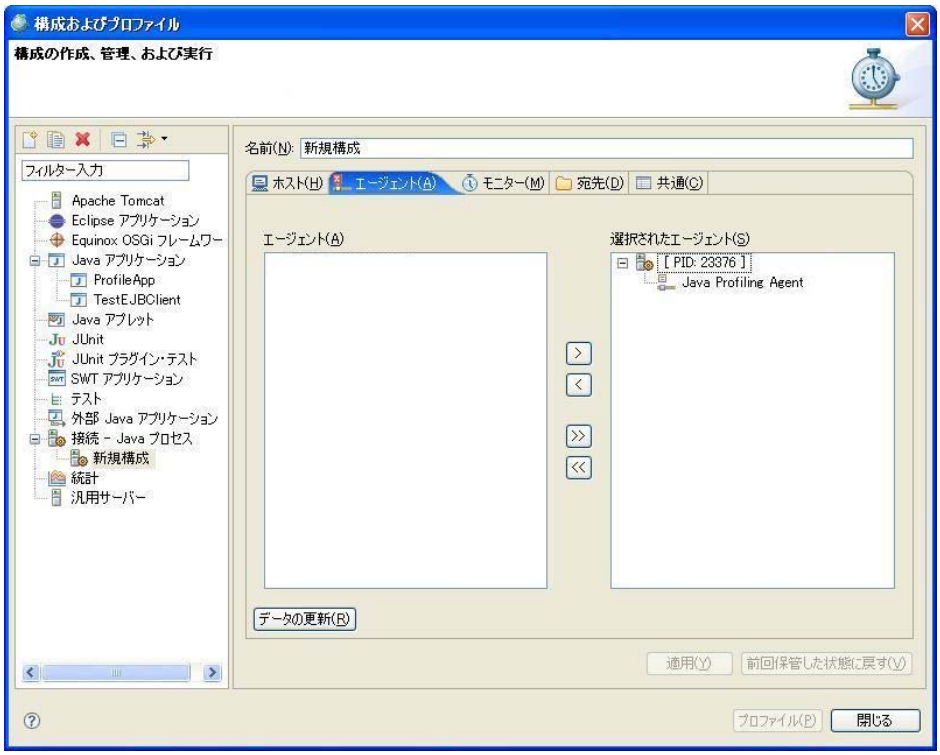


次に[**エージェント**]タブで起動中のエージェントを確認して、[＞]ボタンでプロファイルを行うエ

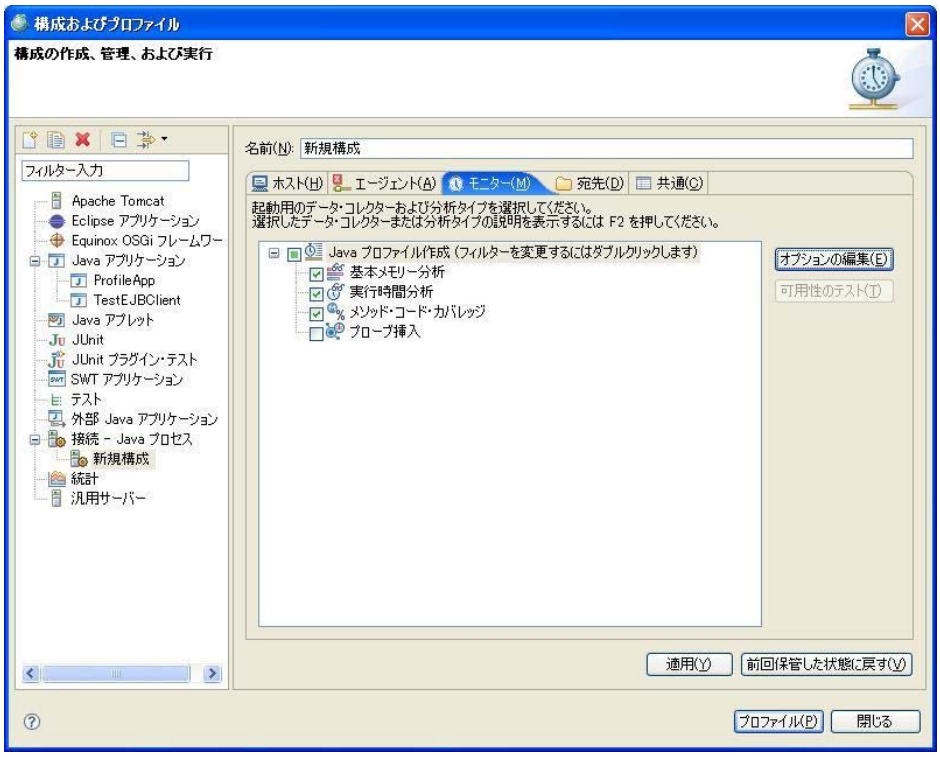
ージェントを選択されたエージェントへと移動させます。



エージェントに何も表示されない場合、**「初期設定」**の**「TPTP」**を行っていない可能性がありますので、必ず行ってください。

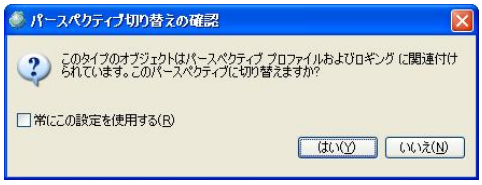


[モニター]タブにおいて、以上の「Java アプリケーションのプロファイリング準備」で作成したプロファイル・セットのように設定して[プロファイル]ボタンを押します。

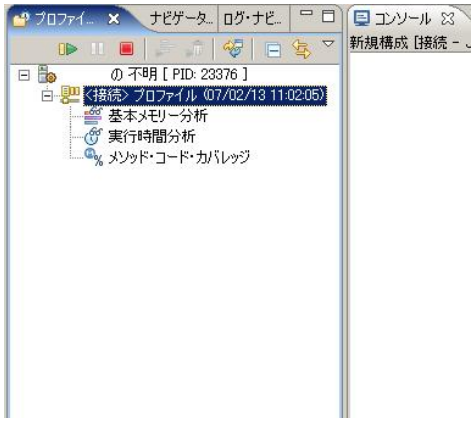


プロファイルのヒント画面が表示されます。

OK ボタンを押します。



プロファイル・モニターにプロファイル・リソースが表示されます。

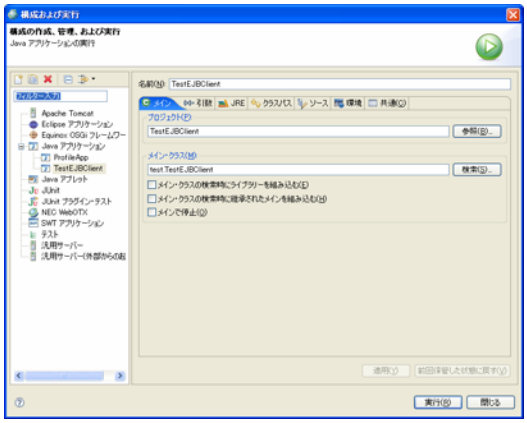


これで、プロファイルの準備は完了です。

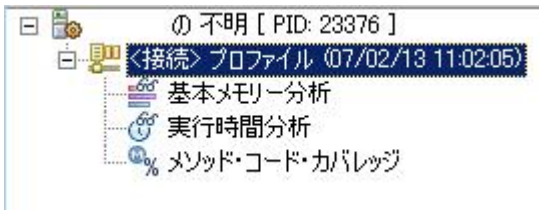
1.1.3.プロファイル操作

プロファイルの準備が整ったら、次はプロファイルを開始します。

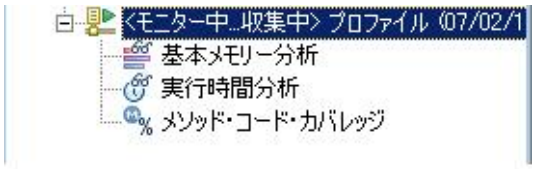
メニュー **実行** | **構成および実行**で、TestEJBClient を実行します。



[プロファイル]ボタンを押すと、[プロファイル・モニター]ビューにプロファイル・エージェントが追加され、[接続]状態になっています。

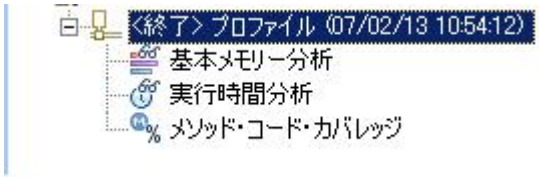


エージェントを選択して[プロファイル・モニター]ビューの 開始ボタン又はポップアップメニューからモニターの開始を押して、モニターを開始します。



[プロファイル・モニター]ビューの一時停止ボタン又はポップアップメニューからモニターの一時停止で、モニターを一時停止できます。

プロファイルの結果を確認するには、モニターを終了させる必要があります。
エージェントを選択して[[プロファイル・モニター]ビューの 停止ボタン又はポップアップメニューから終了を押して、モニターを終了させます。
このアプリケーションも終了させます



1.1.4.プロファイル結果の表示

プロファイル結果の表示を行います。
モニターの終了を確認後、以下の6つのビューを使用してプロファイル結果を表示します。
また各ビューの操作・機能については、「プログラミングと開発 TPTP 編」を参照してください。

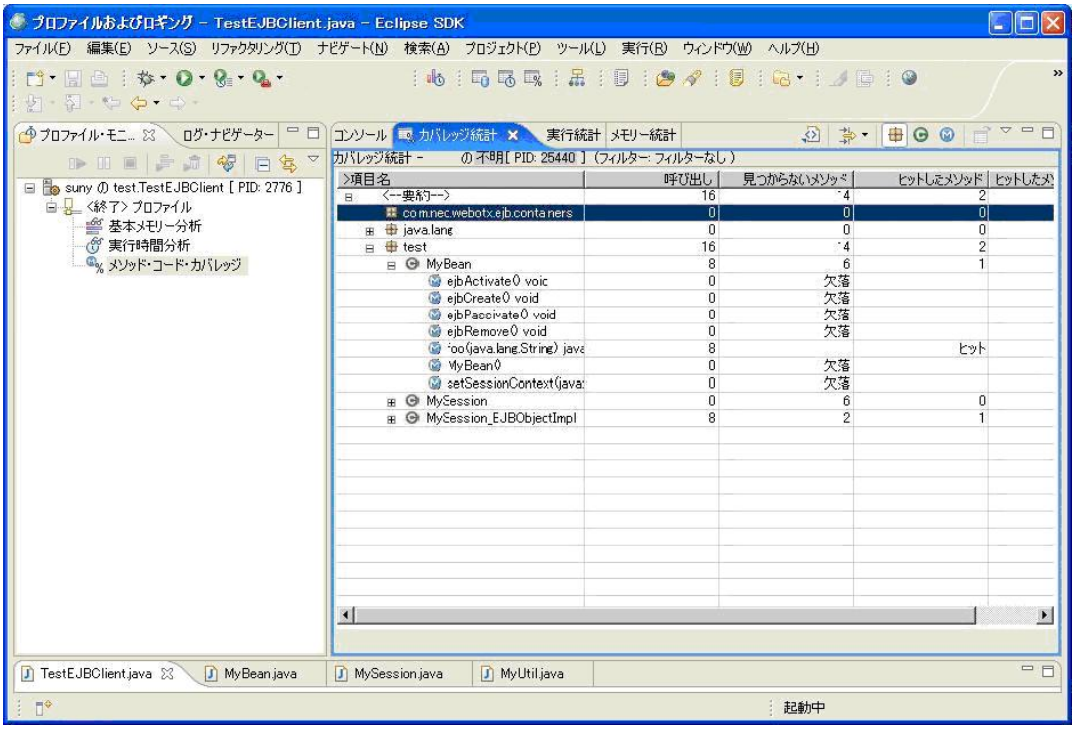
- カバレッジ統計
- メモリー統計
- オブジェクト参照
- 実行統計
- 実行フロー
- UML2 相互作用トレース

・[カバレッジ統計]ビューの表示

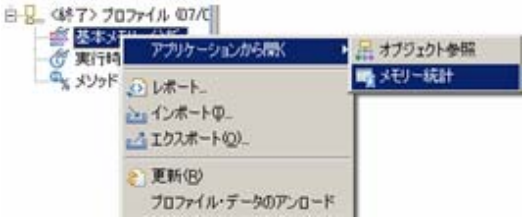
このビューでは、パッケージ及びクラスごとのメソッド使用に関するカバレッジ統計を表示します。

[プロファイル・モニター]ビューの対象エージェント配下にある[メソッド・コード・カバレッジ]を右クリックして、ポップアップメニューから アプリケーションから開く | カバレッジ統計 を選択します。

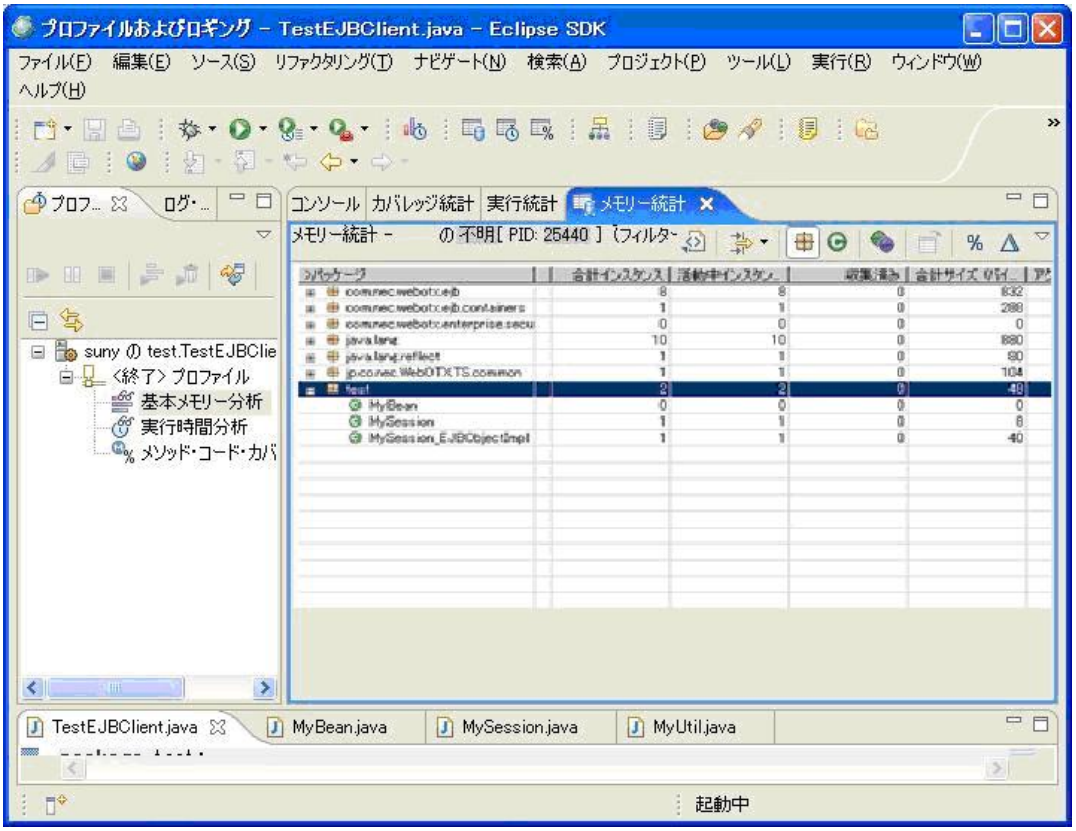
[カバレッジ統計]ビューが表示されます。EJB クライアントアプリケーションを実行すると EJB のクラスの統計情報が表示されます。



・[メモリー統計]ビューの表示



このビューでは、ロードされているクラスの数、活動しているインスタンスの数、およびそれぞれのクラスによって割り振られているメモリー・サイズなどの詳細情報を表示します。[プロファイル・モニター]ビューの対象エージェント配下にある[基本メモリー分析]を右クリックして、ポップアップメニューから アプリケーションから開く | メモリー統計 を選択します。[メモリー統計]ビューが表示されます。

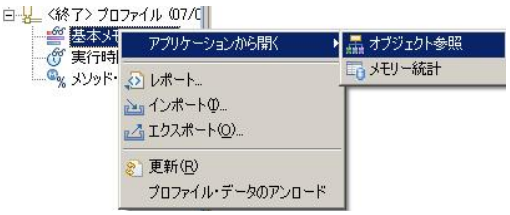


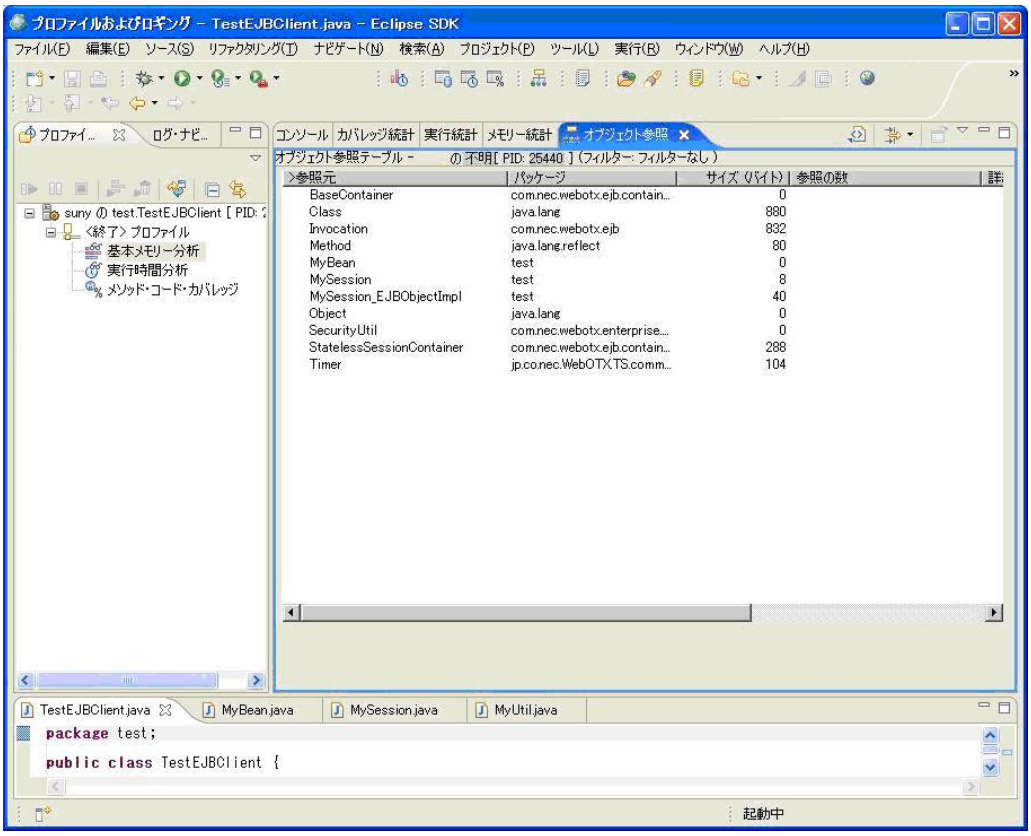
・[オブジェクト参照]ビューの表示

このビューでは、各オブジェクトがどのオブジェクトを参照しているかを表示します。

[プロファイル・モニター]ビューの対象エージェント配下にある[基本メモリー分析]を右クリックして、ポップアップメニューから アプリケーションから開く | オブジェクト参照 を選択します。

[オブジェクト参照]ビューが表示されます。

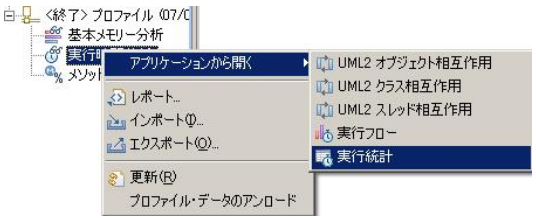


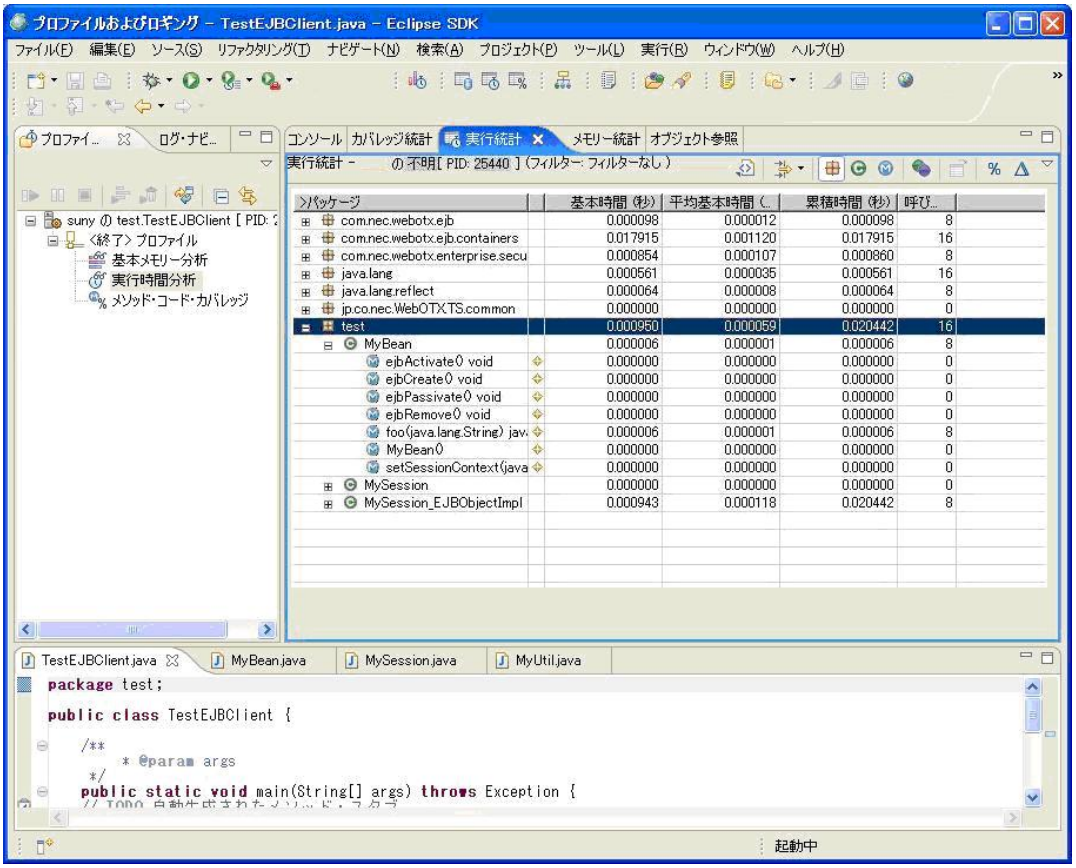


・[実行統計]ビューの表示

このビューでは、呼び出されたメソッドの数、およびすべてのメソッドのそれぞれの実行に要した時間などのデータを表形式で表示します。

[プロファイル・モニター]ビューの対象エージェント配下にある[実行時間分析]を右クリックして、ポップアップメニューから アプリケーションから開く | 実行統計 を選択します。[実行統計]ビューが表示されます。

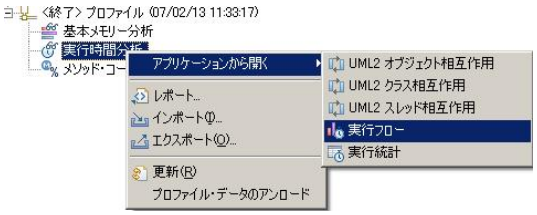


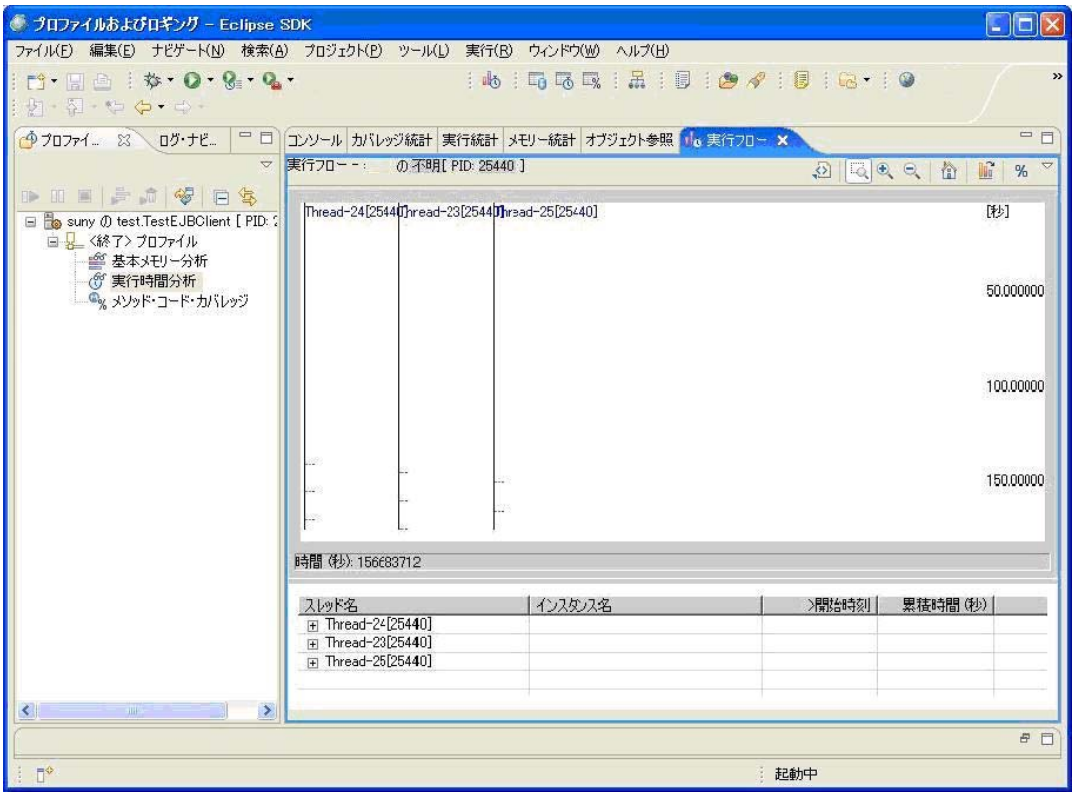


・[実行フロー]ビューの表示

このビューでは、呼び出されたメソッドの数、およびすべてのメソッドのそれぞれの実行に要した時間などのデータをグラフ形式で表示します。

[プロファイル・モニター]ビューの対象エージェント配下にある[実行時間分析]を右クリックして、ポップアップメニューから アプリケーションから開く | 実行フロー を選択します。[実行フロー]ビューが表示されます。





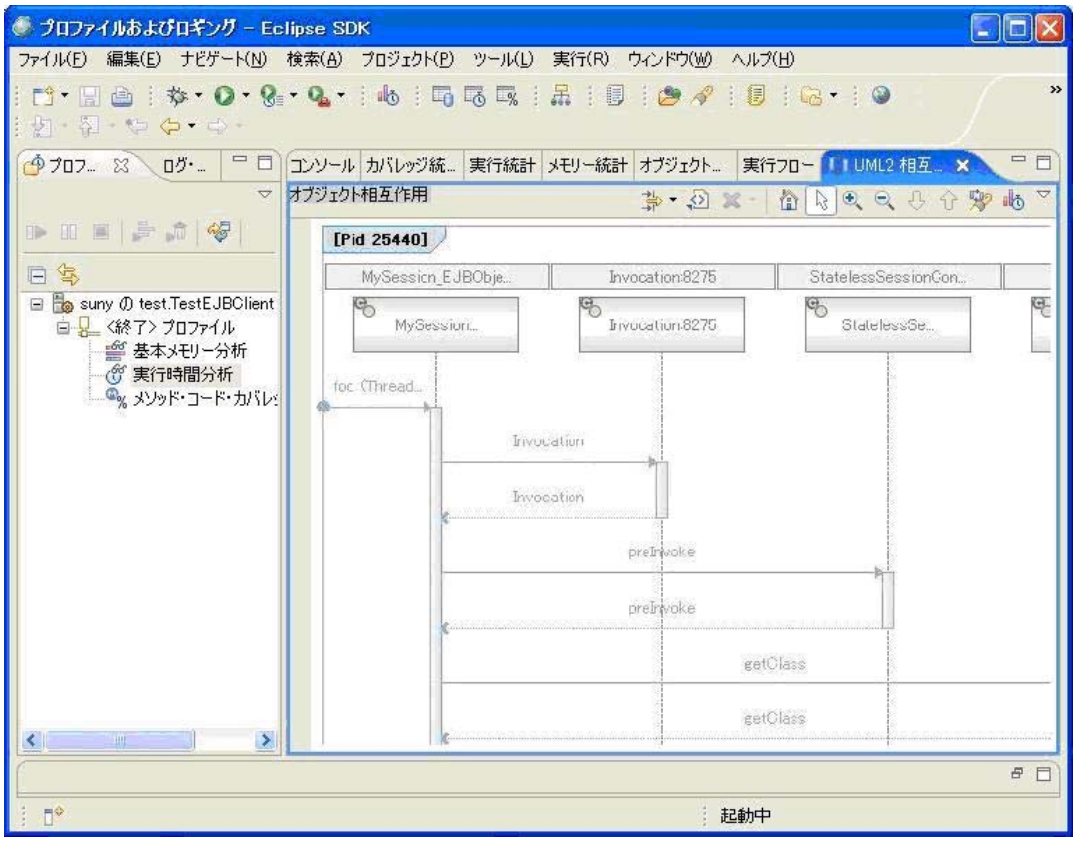
・[UML2 相互作用トレース]ビューの表示

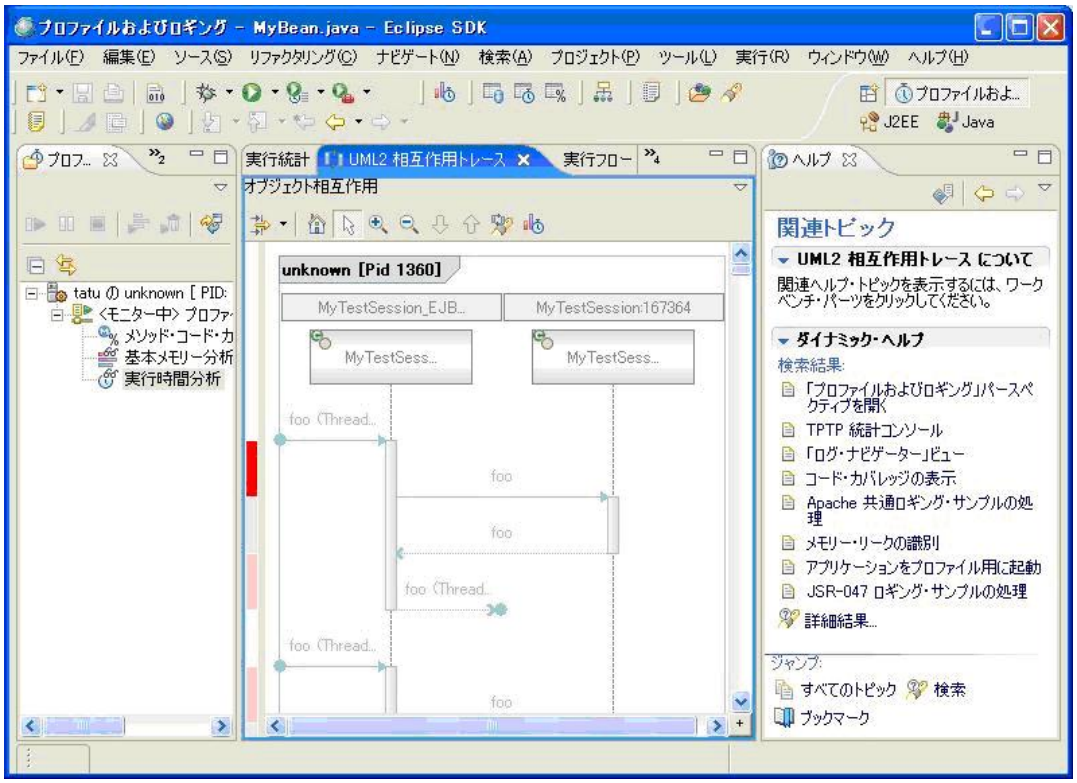
このビューでは、UML によって定義された表記法に従って、アプリケーションの実行フローを表示します。

[プロファイル・モニター]ビューの対象エージェント配下にある[実行時間分析]を右クリックして、ポップアップメニューから

アプリケーションから開く｜UML2･･･ を選択します。

[UML2 相互作用トレース]ビューが表示されます。


プロファイルおよびロギング – Eclipse SDK
ファイル(F) 編集(E) ナビゲート(N) 検索(A) プロジェクト(P) ツール(L) 実行(R) ウィンドウ(W) ヘルプ(H)
コンソール カバレッジ統... 実行統計 メモリー統計 オブジェクト... 実行フロー UML2 相互...
オブジェクト相互作用
[Pid 25440]
MySessicn_EJBObje... Invocation:8275 StatelessSessionCon...
suny の test.TestEJBClient
<終了> プロファイル
基本メモリー分析
実行時間分析
Invocation
Invocation
preInvoke
preInvoke
getClass
getClass
起動中

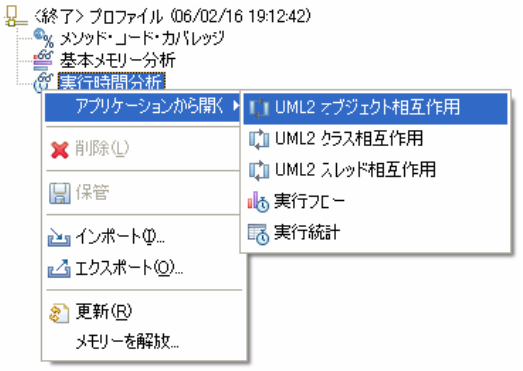

プロファイルおよびロギング – MyBean.java – Eclipse SDK
ファイル(F) 編集(E) ソース(S) リファクタリング(C) ナビゲート(N) 検索(A) プロジェクト(P) ツール(L) 実行(R) ウィンドウ(W) ヘルプ(H)
J2EE Java
実行統計 UML2 相互作用トレース 実行フロー
オブジェクト相互作用
unknown [Pid 1360]
MyTestSession_EJB... MyTestSession:167364
tatu の unknown [PID:
<モニター中> プロファ...
メソッド・コード・カ
基本メモリー分析
実行時間分析
foo
foo
foo
foo
ヘルプ
関連トピック
UML2 相互作用トレース について
関連ヘルプ・トピックを表示するには、ワークベンチ・パーツをクリックしてください。
ダイナミック・ヘルプ
検索結果:
「プロファイルおよびロギング」パースペクティブを開く
TPTP 統計コンソール
「ログ・ナビゲーター」ビュー
コード・カバレッジの表示
Apache 共通ロギング・サンプルの処理
メモリー・リークの識別
アプリケーションをプロファイル用に起動
JSR-047 ロギング・サンプルの処理
詳細結果...
ジャンプ:
すべてのトピック 検索
ブックマーク

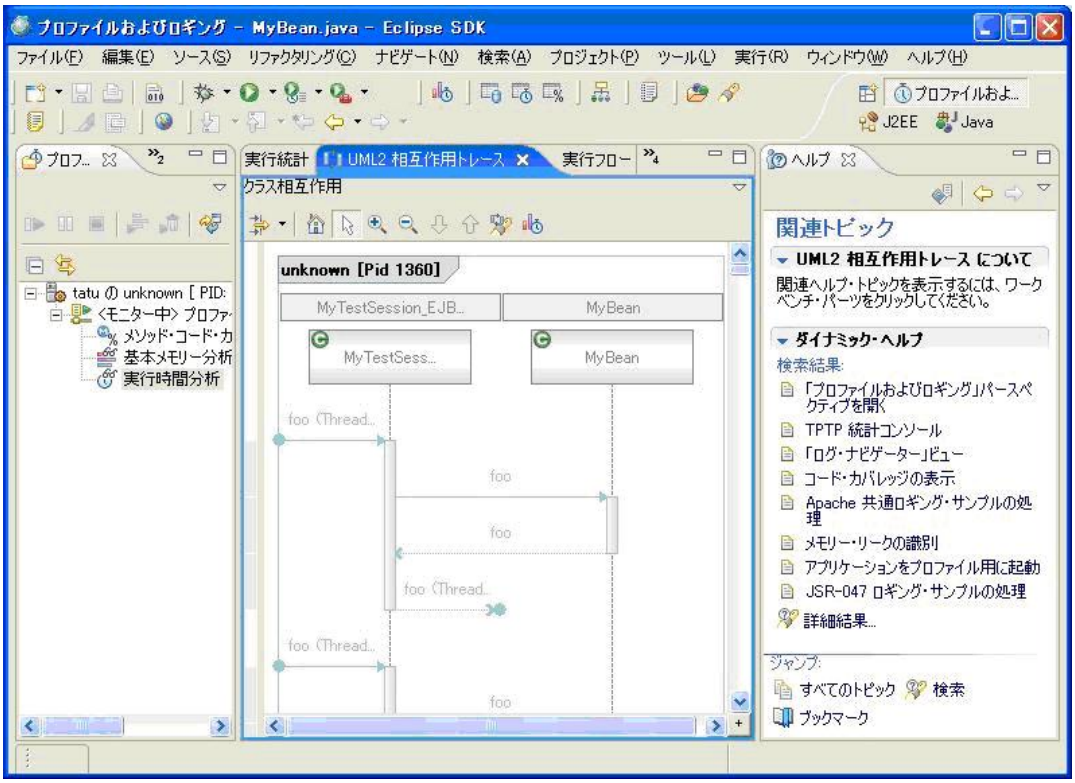
・[UML2 相互作用トレース]ビューの表示

このビューでは、UML によって定義された表記法に従って、アプリケーションの実行フローを表示します。

[プロファイル・モニター]ビューの対象エージェント配下にある[実行時間分析]を右クリックして、ポップアップメニューから アプリケーションから開く | UML2･･･ を選択します。

[UML2 相互作用トレース]ビューが表示されます。





1.1.5.アプリケーションプローブ

プローブは、プログラムのオブジェクト、インスタンス変数、引数、および例外に関する詳しいランタイム情報を収集するためにユーザーが作成する、再使用可能な Java コード・フラグメントです。

プローブを用いることにより、メソッドの開始時、終了時、例外発生時などのタイミングで、ソース・コードに記述していない処理を新たに追加して実行することができます。

新たに追加する処理はプローブと呼ばれます。

プローブはバイトコードに対して追加するため、ソース・コードがなくても処理を追加することができます。

このチュートリアルでは簡単な Java アプリケーションをもとに、プローブ機能使用の流れを以下のフローにしたがって説明します。

[3.6.1. プローブの作成]

↓

[3.6.2. プローブの編集]

↓

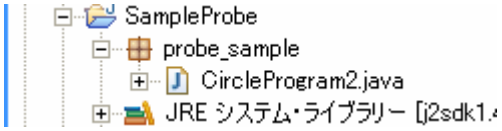
[3.6.3. プローブの装備]

↓

[3.6.4. プローブを装備したクラスの実行]

プローブ機能の確認用に以下の構成の Java アプリケーション・プロジェクトを作成してください。

（以下の図は[ナビゲーター]ビューで表示した構成です。）



<プログラム CircleProgram2.java>

```
package probe_sample;
import java.math.BigDecimal;
/**
* 円の面積や円周を求めるプログラム
*/
public class CircleProgram2 {
/* 円周率 */
private static BigDecimal PI = new BigDecimal(java.lang.Math.PI).setScale(2, BigDecimal.ROUND_HALF_UP);
    /**
    * 円の面積を求める
    * @param radius 半径
    * @return 面積
    */
    protected static BigDecimal CircleArea(BigDecimal radius){
        BigDecimal area =
        radius.multiply(radius).multiply(PI).setScale(2, BigDecimal.ROUND_HALF_UP);
          22
        System.out.println("円面積:" + radius + "*" + radius + "*" + PI + "=" + area);
        return area;
    }
    /**
    * 円周を求める
    * @param radius 半径
    * @return 円周
    */
    protected static BigDecimal Circumference(BigDecimal radius){
        BigDecimal length =
        radius.add(radius).multiply(PI).setScale(2, BigDecimal.ROUND_HALF_UP);
        System.out.println("円周 :" + radius + "*2*" + PI + "=" + length);
        return length;
    }
    public static void main(String[] args){
        System.out.println("サンプルプログラムを開始します");
        try{
            BigDecimal input = new BigDecimal(args[0]);
            System.out.println("¥n 半径 :" + input);
            System.out.println("円周 :" + Circumference(input));
            System.out.println("円面積:" + CircleArea(input));
        } catch(NumberFormatException e){
            System.out.println("¥n 数値を入力してください");
        } finally {
```

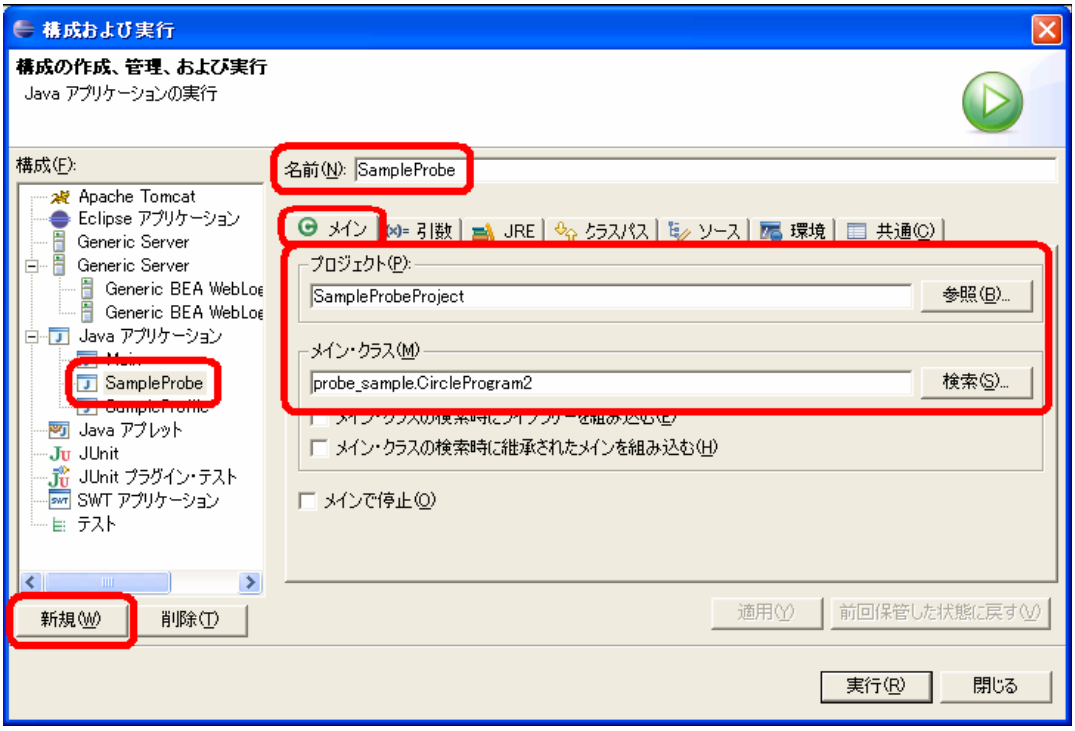
```
                System.out.println("¥n  サンプルプログラムを終了します");
            }
        }
}
```

このプログラムは以下のようにして実行してください。

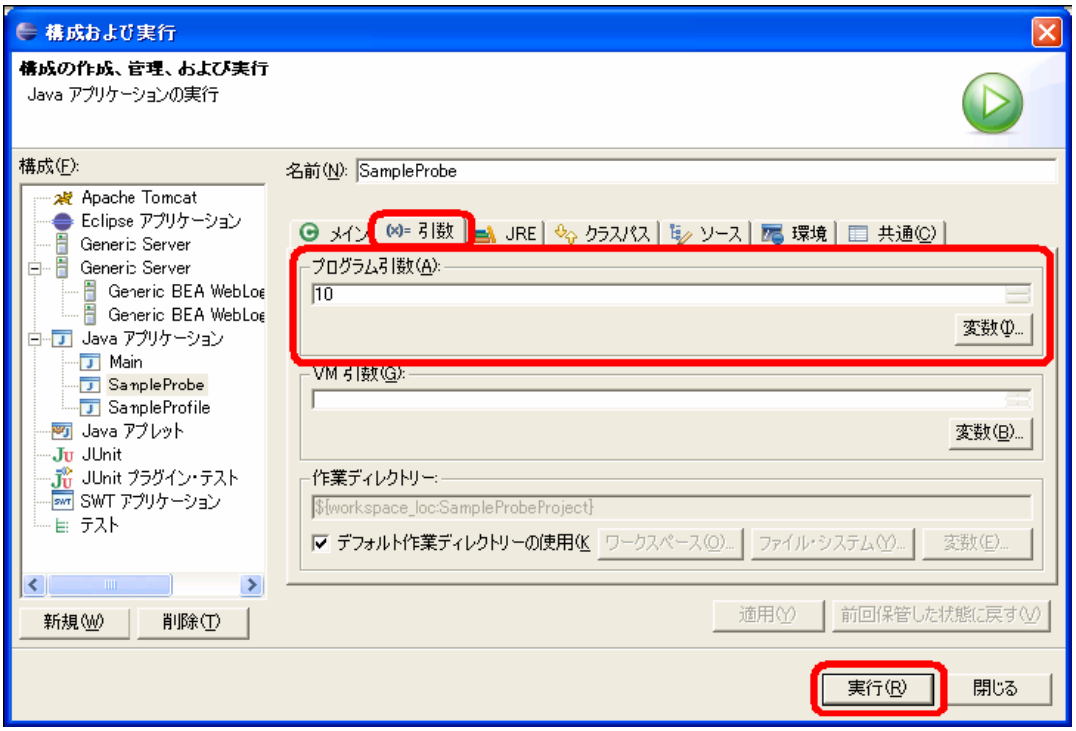
メニューから **実行｜構成および実行** を選択して[**構成および実行**]画面を表示します。

「**構成**」の「**Java アプリケーション**」を選択し[新規]ボタンを押して新規実行構成を作成します。

「名前」は任意の値を入力して、[**メイン**]タブでは「**プロジェクト**」と「**メイン・クラス**」に上記で作成したプログラムのプロジェクトとクラスを設定します。



また[引数]タブで「プログラム引数」に 10 と入力します。設定を完了したら、[実行]ボタンを押してプログラムを実行します。



実行結果は以下のようになります。

```
サンプルプログラムを開始します
半径　:10
円周　:10 * 2 * 3.14 = 62.80
円周　:62.80
円面積:10 * 10 * 3.14 = 314.00
円面積:314.00
サンプルプログラムを終了します
```

プローブの作成

始めにプローブ・ファイルを作成します。

プローブのフラグメント型には以下のものがあります。

プローブには、メソッド・プローブと Callsite プローブの 2 種類のプローブがあります。

メソッド・プローブは、メソッドそのものに対して処理を追加するタイプのプローブで、ターゲットとなるメソッド実行の直前、直後、例外発生時などに処理を追加することができます。

Callsite プローブは、メソッドの呼び出しに対して処理を追加するタイプのプローブで、ターゲットとなるメソッドの呼び出し前、呼び出し後に処理を追加することができます。

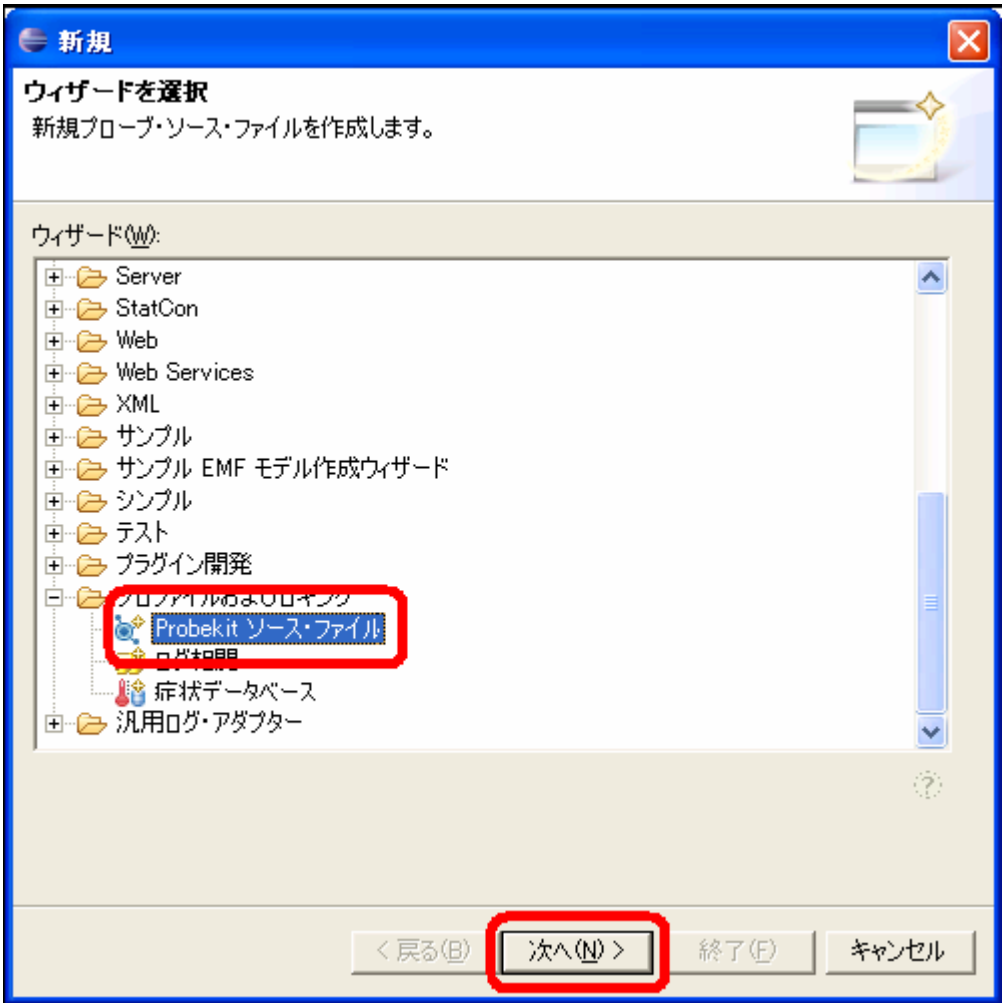
プローブ	フラグメント型	説明
メソッド・プローブ	entry	メソッド入り口で entry フラグメントが実行されます。
	executableUnit	プローブのターゲットとフィルター仕様に一致する、ソース・コードが使用可能なメソッドに含まれる実行可能な各コード・ユニットの直前に、executableUnit フラグメントが実行されます。

	catch	メソッドの catch 文節の先頭、または例外の結果で catch フラグメントが実行されます。
	exit	メソッドの出口で（通常のエグジット時、メソッドが例外をスローしたとき、またはスローされた例外がメソッド外に波及したときに）exit フラグメントが実行されます。
	staticInitializer	各プローブ・クラスのクラス・イニシャライザーの中で実行する staticInitializer フラグメント。
Callsite プローブ	afterCall	ターゲット・メソッドがエグジットした直後に（通常終了時、またはターゲット・メソッドが例外をスローしたときに）、呼び出し側のメソッドで afterCall フラグメントが実行されます。
	beforeCall	ターゲット・メソッドが呼び出される直前に、呼び出し側のメソッドで beforeCall フラグメントが実行されます。

ここでは、メソッドの先頭に処理を追加する entry を用いたプローブを作成します。

メニューから ファイル | 新規 | その他 を選択して、[新規]画面を開きます。

プロファイルおよびロギング | Probekit ソース・ファイル を選択し、[次へ]ボタンを押します。



「ファイル名」と「ソース・フォルダー」を以下のように入力して、[次へ]ボタンを押します。

ファイル名：ProbeTest.probe

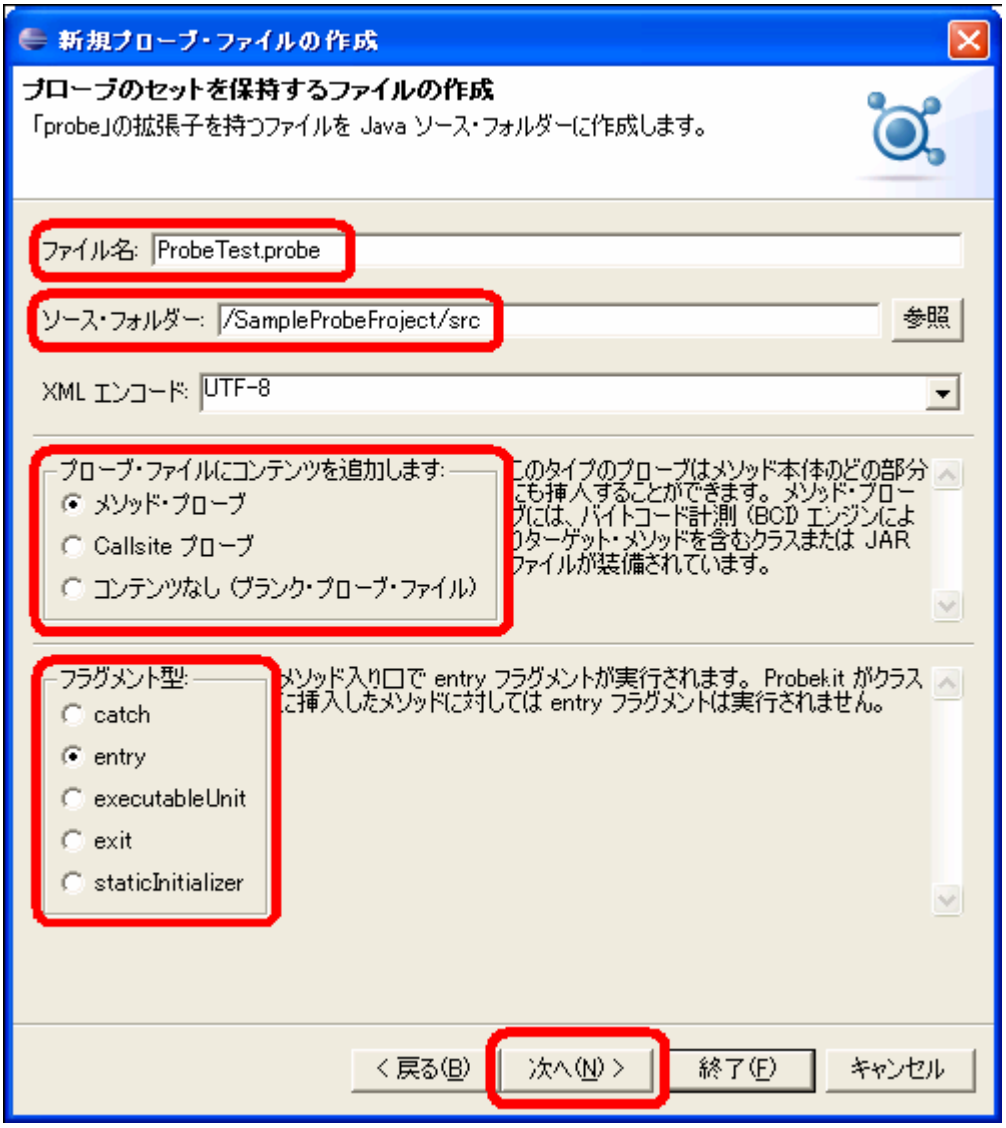
ソース・フォルダー：/SampleProbeProject/src

その他の項目は、デフォルト値のままにしておいてください。

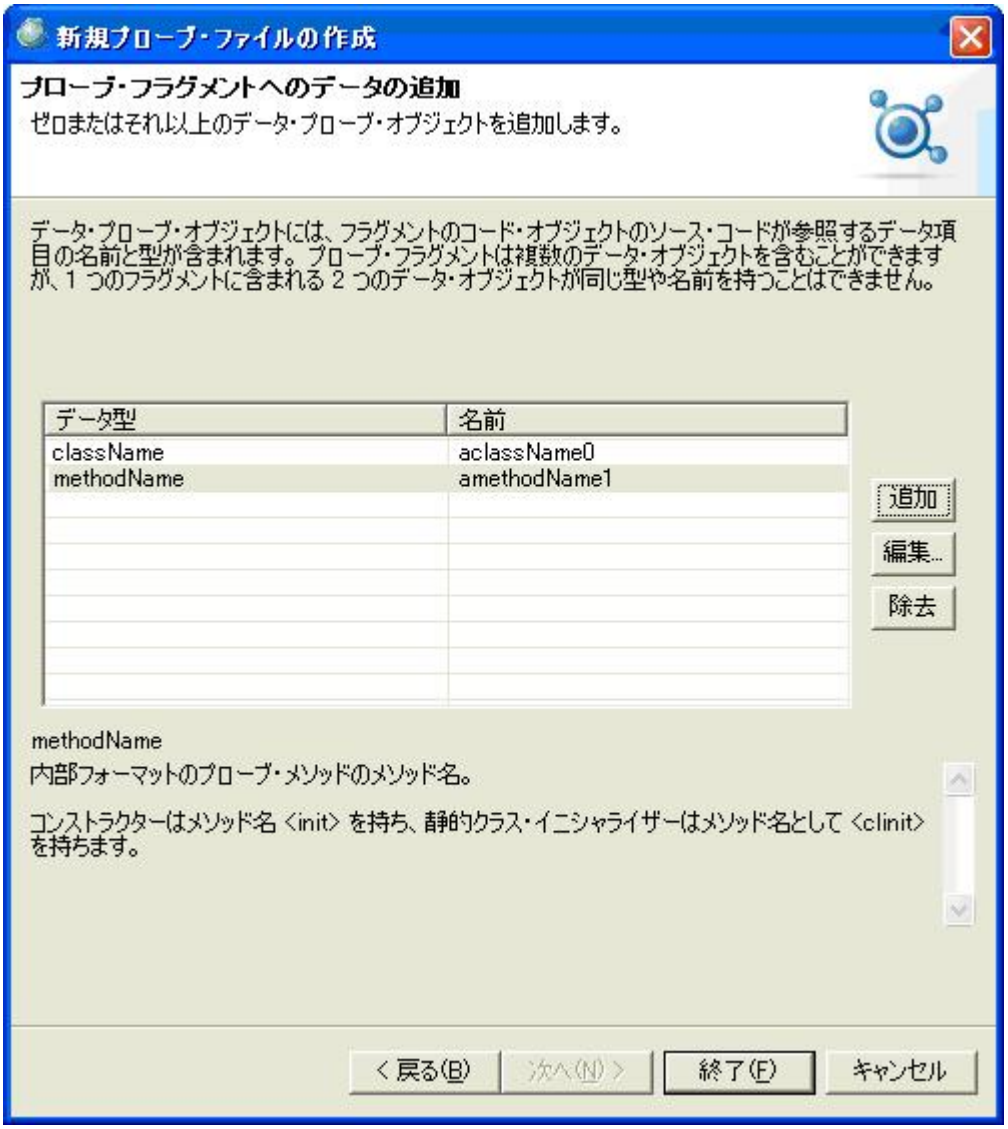
XML エンコード：UTF-8

プローブ・ファイルにコンテンツを追加します：メソッド・プローブ

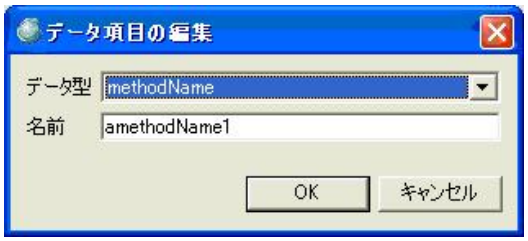
フラグメント型：entry



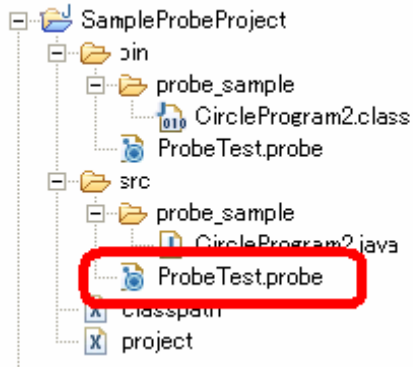
[追加]ボタンで使用するデータを追加して行きます。



またデータを選択して[編集]ボタンで、データ型や名前の編集を行えます。



データの追加が終わると、[終了]ボタンを押します。
新規プローブ･ファイルの作成が終了すると、設定したソース･フォルダー配下にプローブ・ファイル（拡張子 *.probe）が生成されています。



（クラス・フォルダー配下にプローブ･ファイルのコピーが生成されます）

プローブの編集

プローブは、プローブ・ファイル(*.probe)と呼ばれる XML 形式のファイルに記述します。

プローブ・ファイルには、処理を追加するクラスやメソッドの位置、追加する処理などを定義します。

TPTP では プローブ・ファイルを編集するためのエディターが提供されています。

生成されたプローブ・ファイルをダブル・クリックしてプローブ・エディターを表示することで、プローブの編集を行います。

今回はプローブ編集を簡単に説明しますが、プローブ編集の詳しい機能については「プログラミングと開発 TPTP 編」を参照してください。

[プローブ]タブの Probekit ソース・ファイル内のツリーで、プローブを選択します。

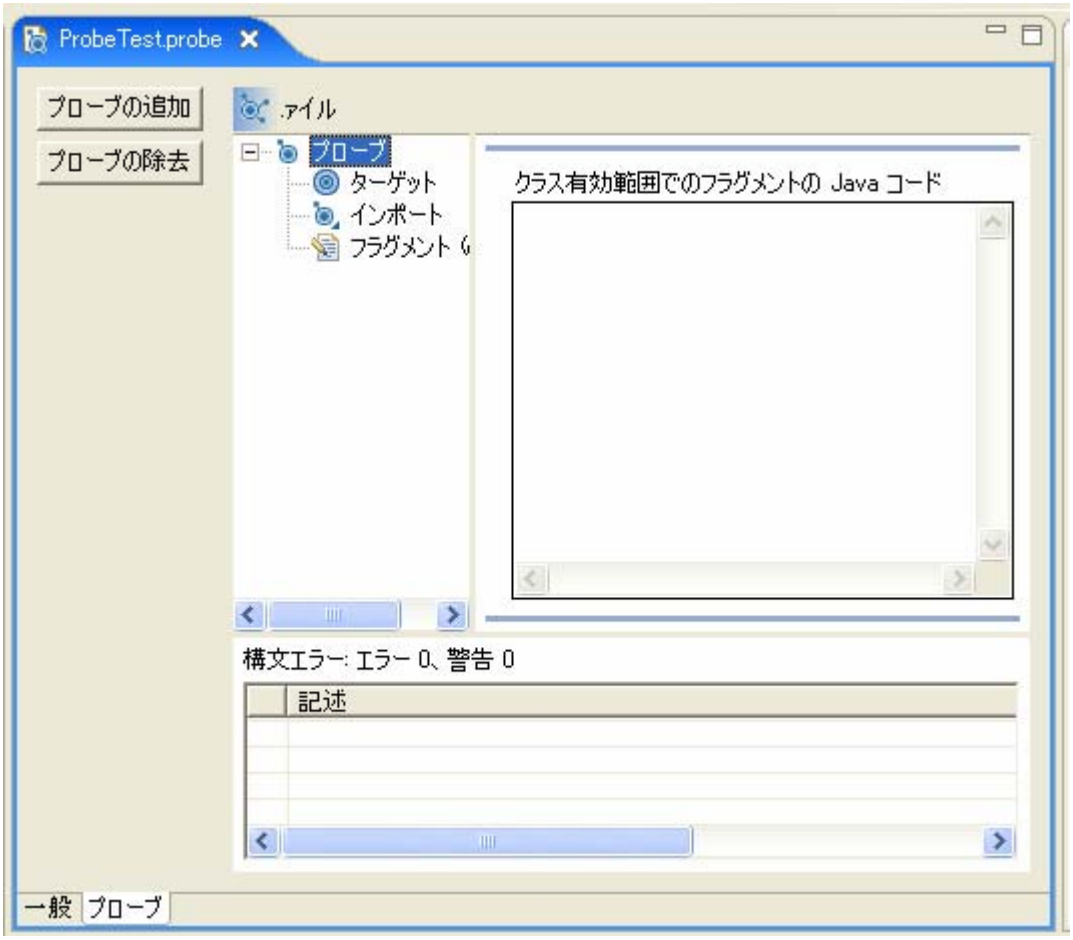
ここではプローブ・クラスにクラス変数などを作成できます。

作成したいクラス変数などをエディター右側のテキストエリアに Java コードで記述します。

メソッドの呼び出し回数を記憶させる変数 counter を定義するため、次のように入力します。

これによりプローブ・クラスにクラス変数 counter が初期値 0 で追加されます。

```
public static int counter = 0;
```



次に Probekit ソース・ファイル内のツリーで、ターゲットを選択します。

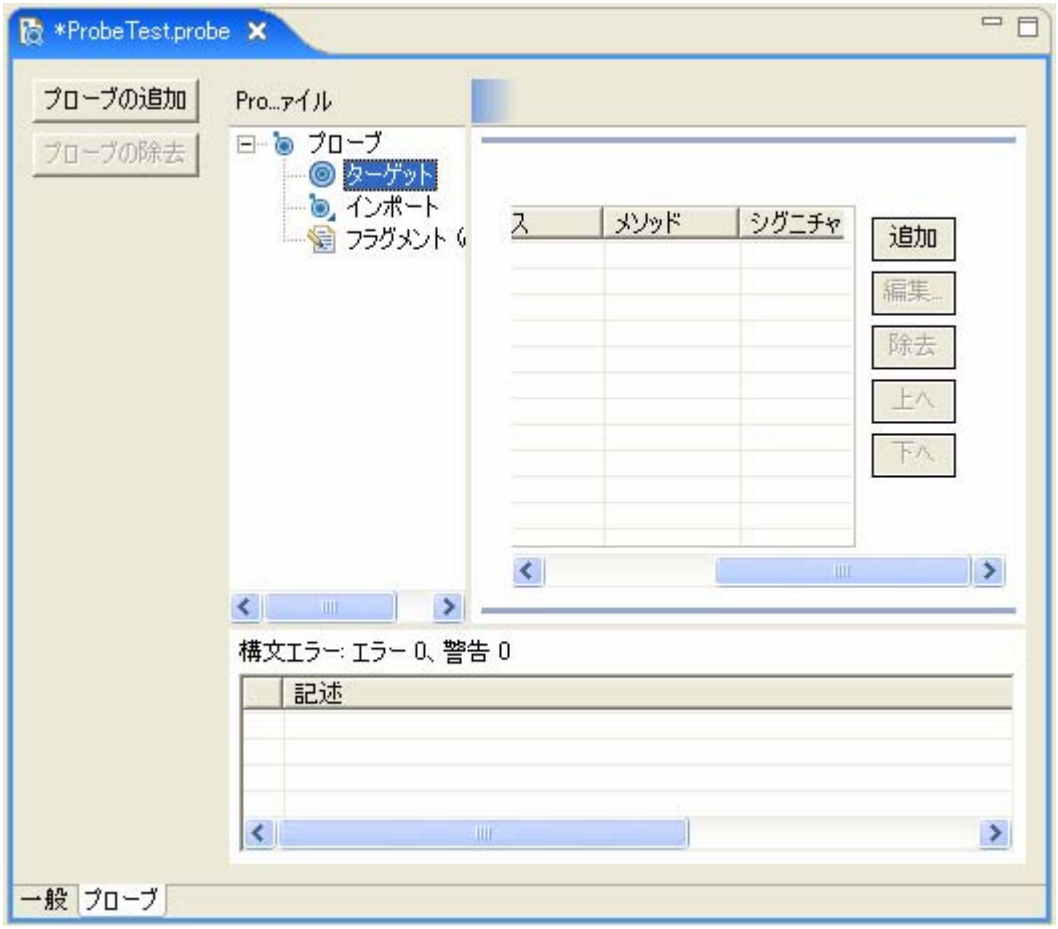
ここではプローブを埋め込むターゲットを設定します。

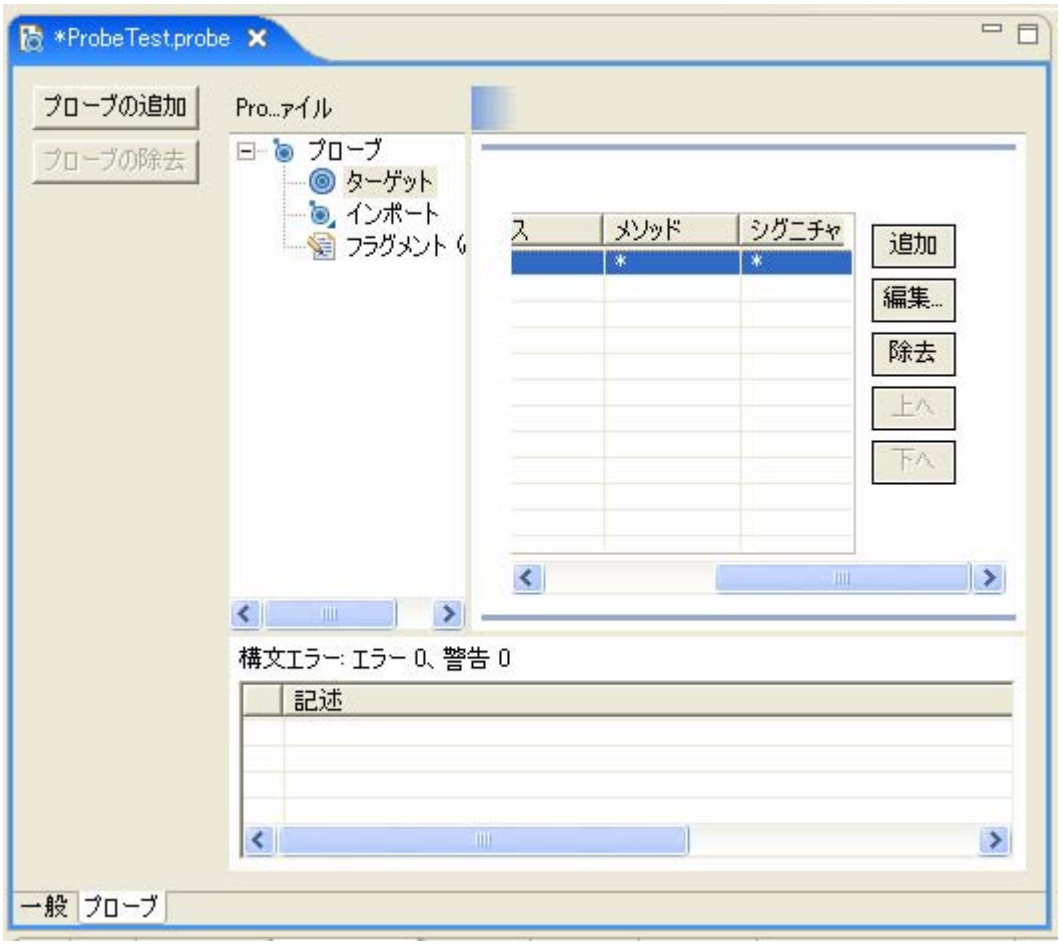
エディター右側にターゲットのテーブルが表示されるので、[追加][編集]ボタンを使用してターゲットの絞込みを行います。

クラス名が Circ で始まるクラスのすべてのメソッドにプローブが埋め込むため、次のように設定します。

タイプ	パッケージ	クラス	メソッド	シグニチャ
include	*	Circ*	*	*
exclude	*	*	*	*

include でターゲットを設定後、exclude でターゲット以外の排除の設定をする必要があります。





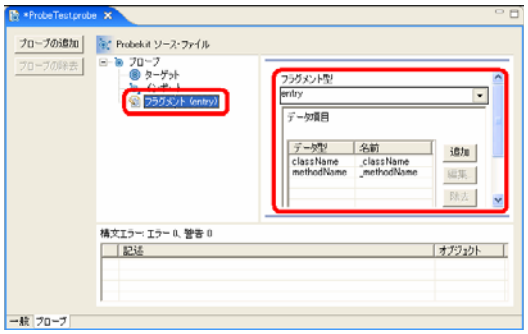
次に Probekit ソース・ファイル内のツリーで、フラグメントを選択します。

再度プローブのフラグメント型を指定します。

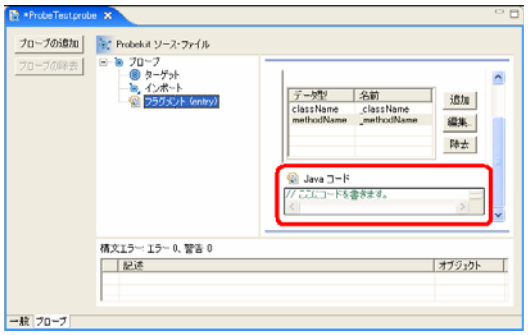
また、プローブで行う処理に必要な変数を[データ項目]に、必要なコードを[Java コード]に入力します。

ここでは、クラス名、メソッド名、メソッドの呼び出し回数を標準出力へ表示させるために[データ項目]、[Java コード]にはそれぞれ以下を記述します。

フラグメント型
Entry
データ型　名前
className _className
methodName _methodName
Java コード
counter++;
System.out.println("[Probe]: "
+ _className + "." + _methodName);
System.out.println("[Counter]: "
+ counter);



ファイル | 保管 を行い、編集を終わります。



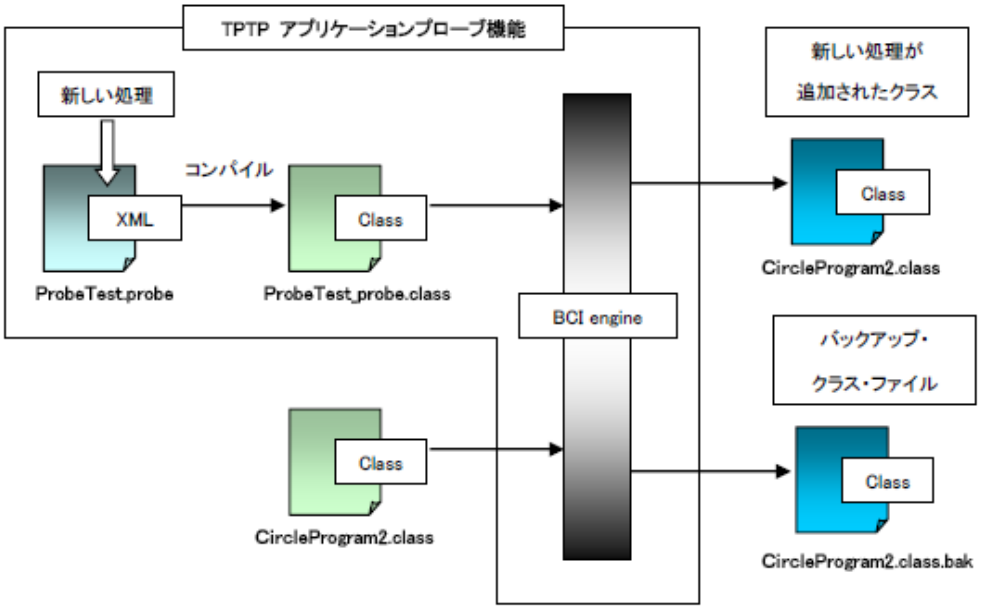
プローブの装備

編集したプローブ・ファイルをコンパイルして、対象プログラムのクラス・ファイルに装備させます。

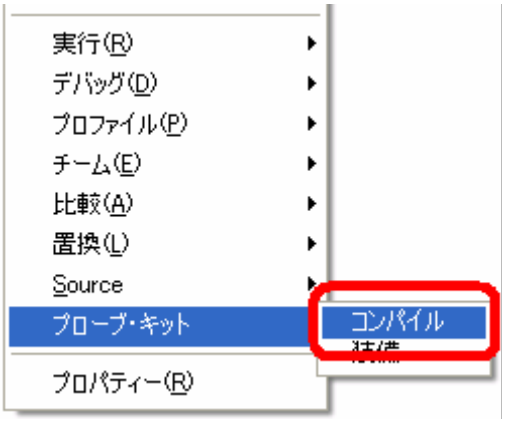
プローブの装備とは、プローブ・ファイルをコンパイルした結果生成されるクラス・ファイルを、対象プログラムのクラス・ファイルに埋め込み処理を追加させることです。

プローブ・ファイルは専用のコンパイラからプローブ・クラス・ファイルを生成して、プローブ・クラス・ファイルと対象のクラス・ファイルを Byte Code Instrumentation engine(以下 BCI engine)に与えて、新しい処理を追加したクラス・ファイルを作成します。

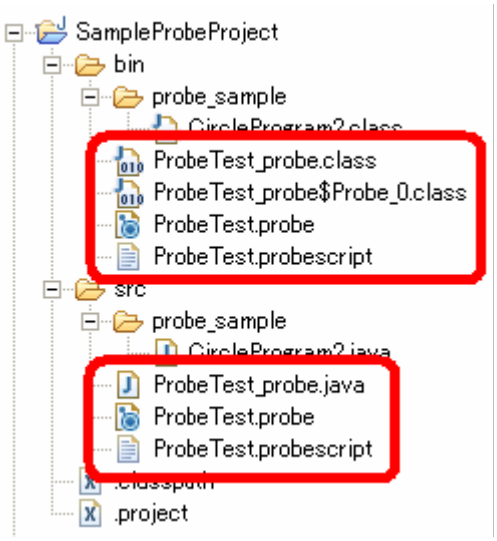
以下にプローブの装備の流れを示します。



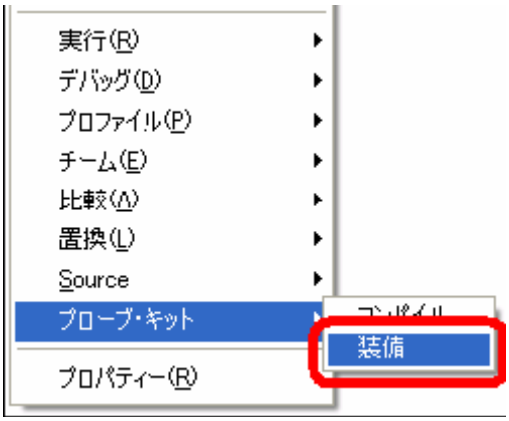
ソース・フォルダー配下にあるプローブ・ファイルを右クリックし、ポップアップメニューからプローブ・キット｜コンパイルを選択して、プローブ・ファイルのコンパイルを行います。



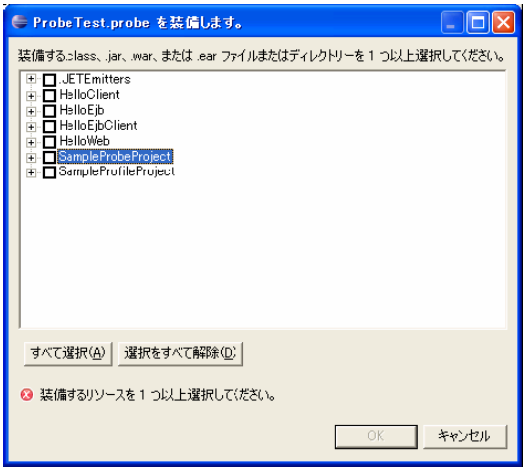
コンパイルに成功するとプローブ・ファイルと同じ場所にプローブ・スクリプト・ファイルとプローブ・ソース・ファイルが生成されます。
（プローブ・ソース・ファイルはパッケージ・エクスプローラ上では、自動的にデフォルト・パッケージが生成され、その直下に生成される）
またクラス・フォルダー配下にもプローブ・スクリプト・ファイルとプローブ・クラス・ファイルが生成されています。



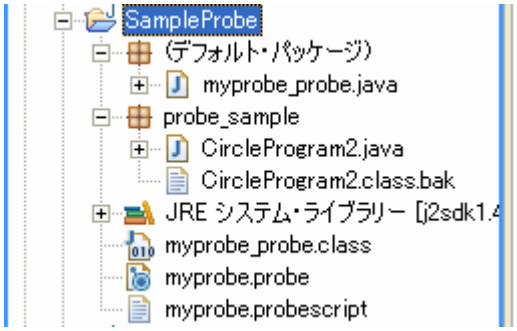
次にソース・フォルダー配下にあるプローブ・ファイルを右クリックし、ポップアップメニューからプローブ・キット｜装備 を選択して、プローブ・ファイルの装備を行います。



[***.probe を装備します]画面で、表示されているツリーを展開して、プローブを装備させたいクラス・ファイルにチェックを付けます。
今回は以下のクラス・ファイルにチェックを付けます。
SampleProbeProject/bin/probe_sample/CircleSample2.class
チェックを付け終わった後、[OK]ボタンを押します。



装備に成功すると、装備対象としたクラス・ファイルと同じ場所に［装備したクラス・ファイル名.bak］バックアップ・クラスファイルが生成されます。

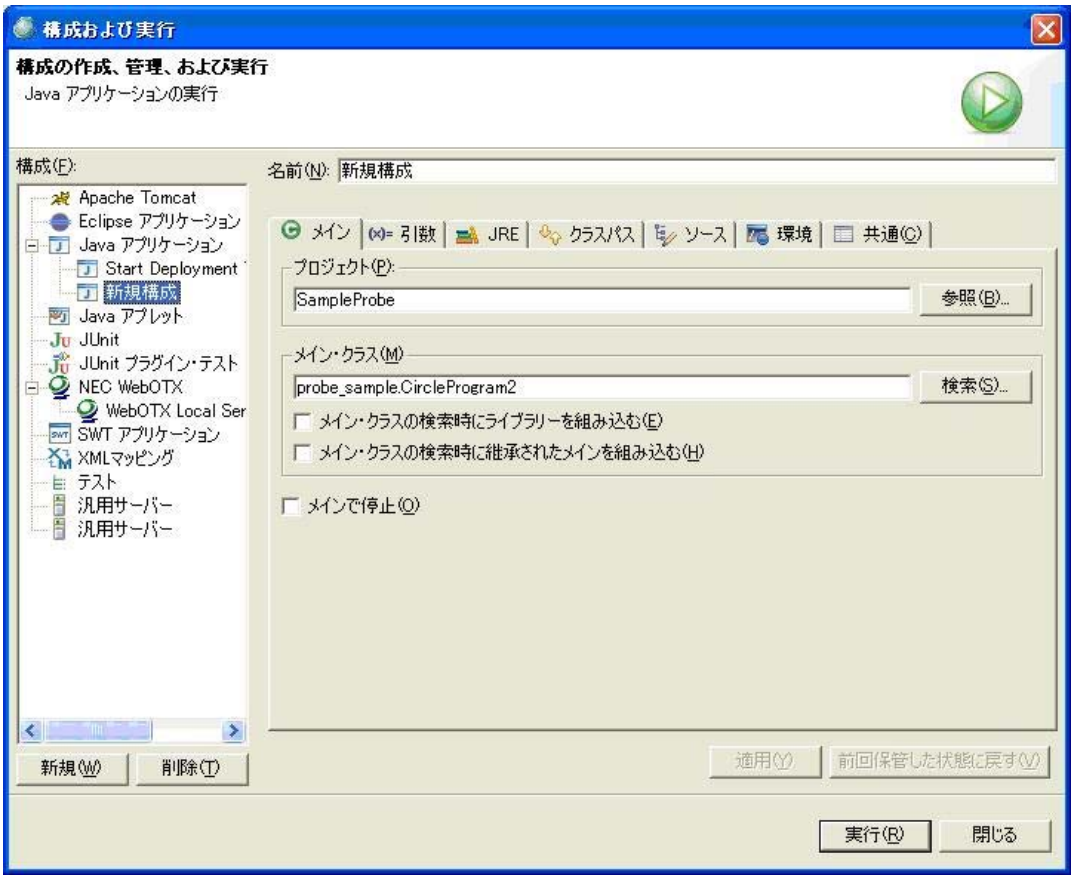


以上でプローブの装備が完了します。

プローブを装備したクラスの実行

プローブを装備したプログラムを実行させます。

プローブを装備させたプログラムの実行は、通常のプログラムと同様に行ってください。




```
[Probe]: probe_sample/CircleProgram2.<clinit>
[Counter]: 1
[Probe]: probe_sample/CircleProgram2.main
[Counter]: 2
サンプルプログラムを開始します

  半径  :10
[Probe]: probe_sample/CircleProgram2.Circumference
[Counter]: 3
円周  :10 * 2 * 3.14 = 62.80
円周  :62.80
[Probe]: probe_sample/CircleProgram2.CircleArea
[Counter]: 4
円面積:10 * 10 * 3.14 = 314.00
円面積:314.00

  サンプルプログラムを終了します
```

（この実行結果は、実行の構成で[引数]タブのプログラム引数に 10 を与えています。）

クラス・ファイルからプローブを除去したい場合、以下の方法を行ってください。

・ 再度ソース・ファイルをコンパイルする。

・ [.class]ファイルを削除して、[.bak]ファイルの名前を編集して「.bak」の部分を削除する。（この方法は一番新しく装備させたプローブを除去するもので、プローブを複数回装備させていると以前のプローブは除去できません。）

1.1.6.ログ総合・分析

TPTP の Log & Trace Analyzer は、複数のアプリケーションなどによる異なったフォーマットのログ・ファイルを Common Base Event (CBE) と呼ばれる XML 形式のログ・フォーマットで統合して、表示・関連付け・分析を行います。

また CBE 形式ではないログ・ファイルは、TPTP の Generic Log Adapter という定義された変換規則に従って対象のログ・フォーマットの項目を CBE 形式の項目に変換するプログラムで、変換してから読み込むことができます。

ここでは Log & Trace Analyzer における次の3つの機能について説明します。

ログの変換及び読み込み様々なログ・ファイルを Generic Log Adapter で CBE 形式に変換して読み込みます。

ログ同士の関連付けと相関表示読み込んだログ同士を関連付けて、シーケンス図で表示します。

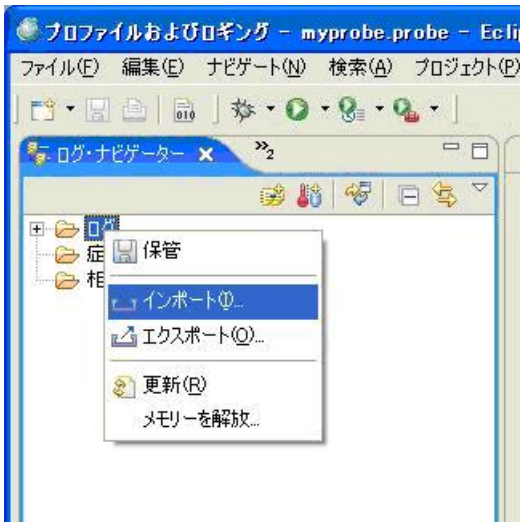
ログの分析エラーメッセージに対するアドバイスを記録した症状データベースを参照して、ログを分析します。

ログの変換および読み込み

Microsoft Windows アプリケーションのログの読み込みを例に、ログのインポート手順を説明します。

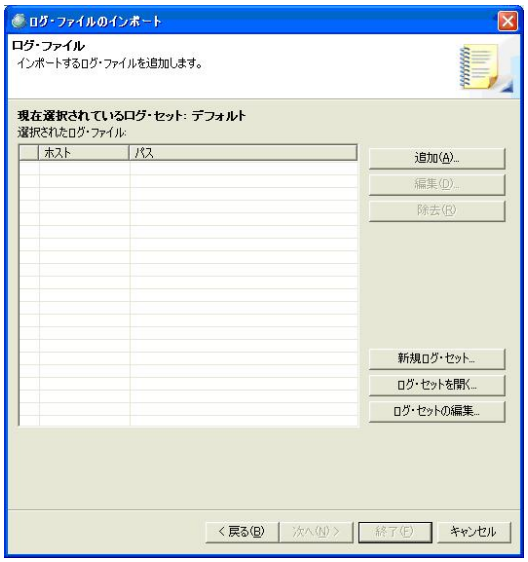
メニューから ウィンドウ | パースペクティブを開く | プロファイルおよびロギング を選択して、予めパースペクティブを切り替えておきます。

[ログ・ナビゲーター]ビューで、[ログ]フォルダを右クリックしてポップアップメニューから[インポート]を選択します。

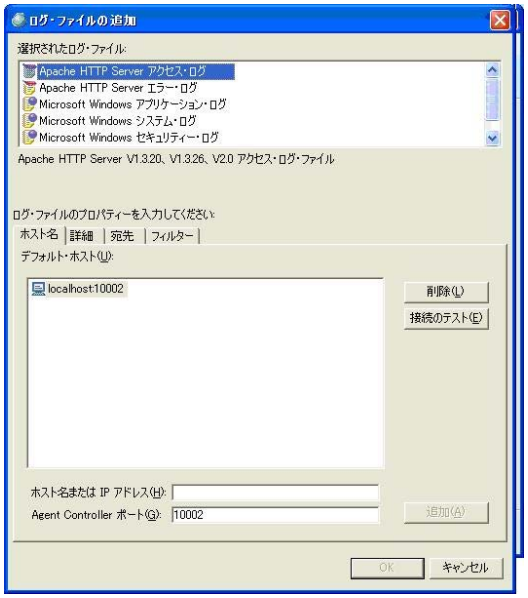


[インポート]画面で、インポート・ソースの選択から[ログ・ファイル]を選択して、[次へ]ボタンを押します。

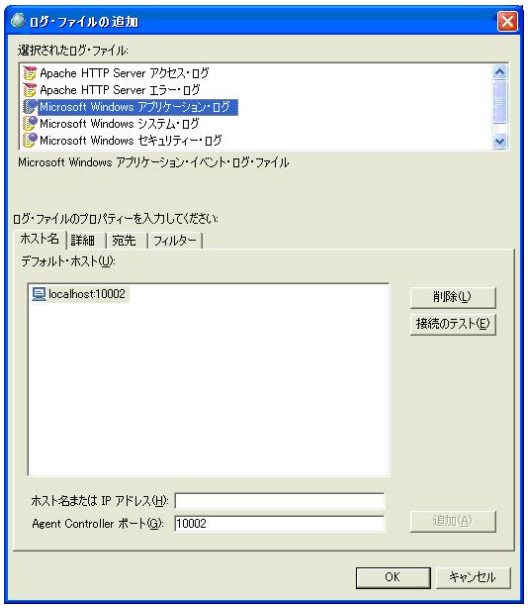
[ログ・ファイルのインポート]画面で、[追加]ボタンを押します。



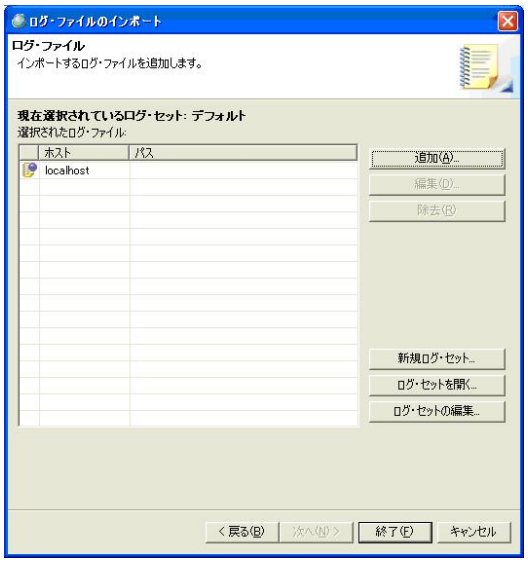
[ログ・ファイルの追加]画面で、以下のように設定します。
選択されたログ・ファイルで「Microsoft Windows アプリケーション・ログ」を選択します。
次に[ホスト]タブで Agent Controller を起動させているデフォルト･ホストを選択されていることを確認します。



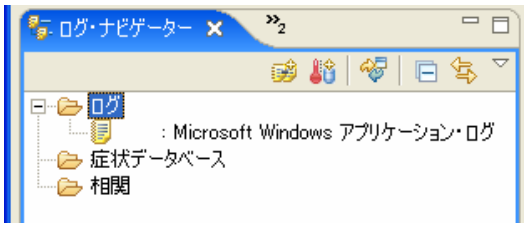
次に[宛先]タブで、保管先のプロジェクトや使用するモニターを設定します。
今回は特に変更しません。
設定を完了すると、[OK]ボタンを押します。



[ログ・ファイルのインポート]画面で、選択されたログ・ファイルに追加したログ・ファイルがあることを確認します。ここでログ・ファイルを選択して[編集]ボタンを押すことで、[ログ・ファイルの編集]画面が表示されログ・ファイルの編集を行えます。
([ログ・ファイルの編集]画面は[ログ・ファイルの追加]画面と同じ内容です)
またログ・ファイルを選択して[除去]ボタンを押すことで、選択したログ･ファイルをリストから削除できます。
インポートするログ･ファイルの設定が終わると[終了]ボタンを押します。

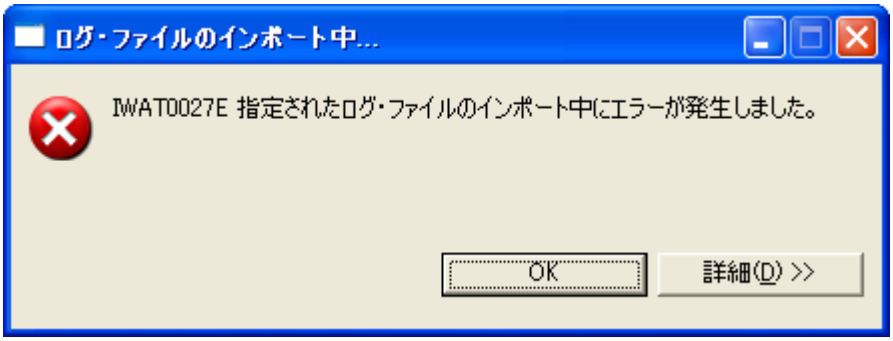


[ログ・ナビゲーター]ビューで、[ログ]フォルダの配下にインポートしたログが表示されます。



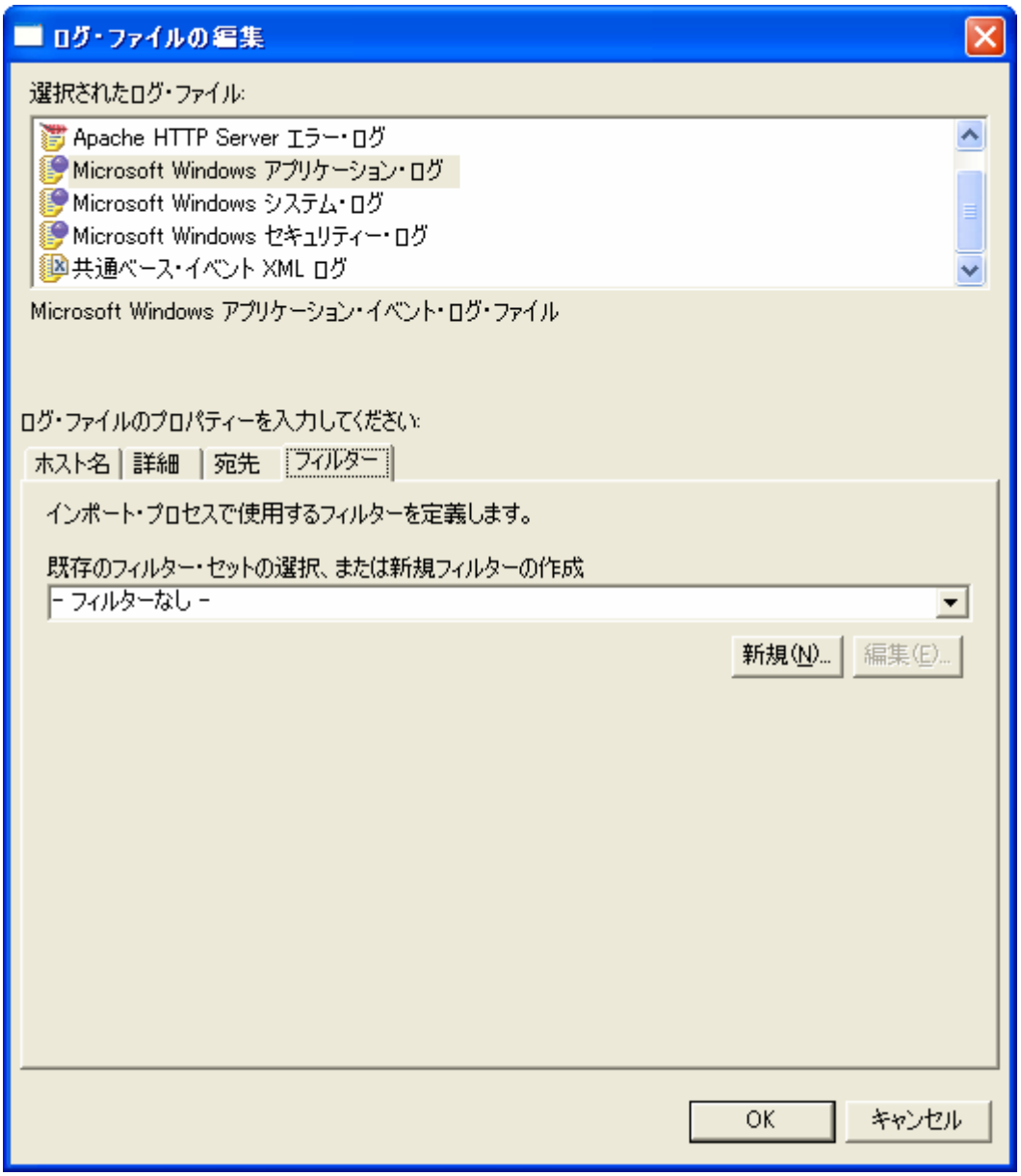
また[ログ・ファイルの追加]画面の選択されたログ・ファイルの種類によっては、[詳細]タブの設定が必要になるものがあります。
以下の例は Apache HTTP Server のアクセス・ログをインポートする場合のもので、[詳細]タブのアクセス・ログ・ファイル・パスに Apache HTTP Server のアクセス・ログ・ファイルのパスを設定する必要があります。

マシンによって Microsoft Windows のログをインポートする際にエラーが発生することがありますが、インポート自体はできています。



Microsoft Windows ログのインポート時にフィルターを設定しないでください。

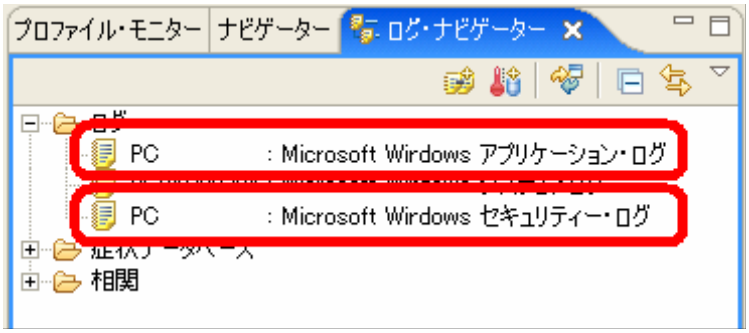
エラーが発生してインポートができません。

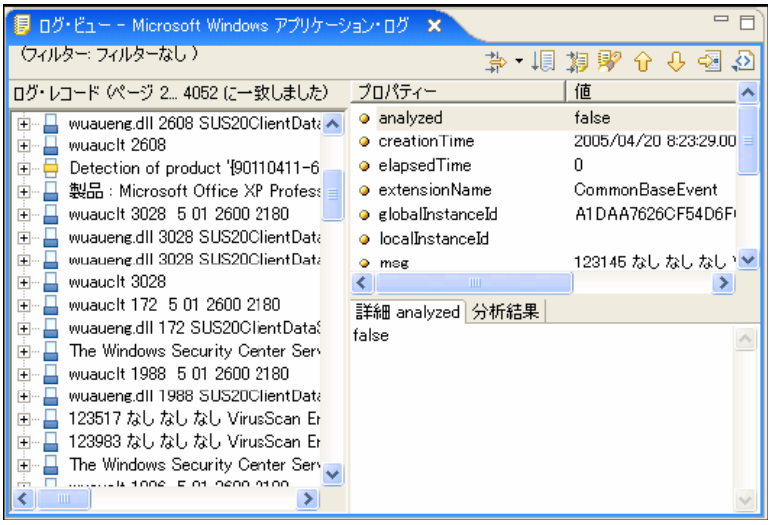


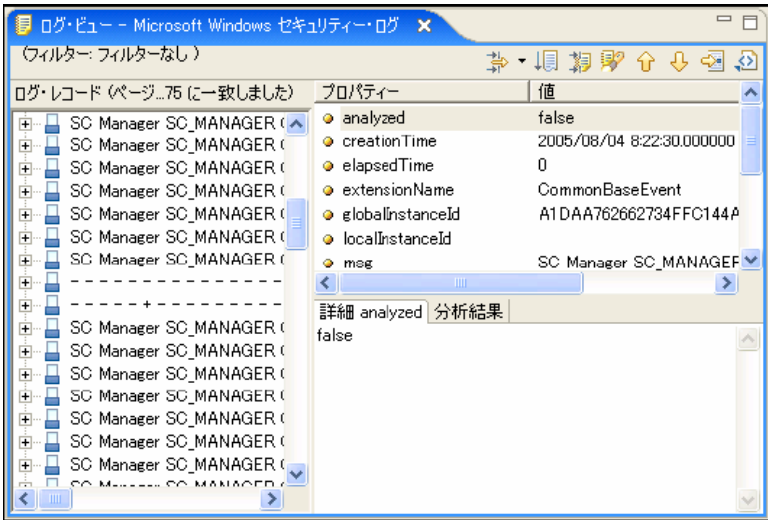
ログ同士の関連付けと相関表示

ここではログ同士の関連付けを行い、その関連を UML のシーケンス図で表示します。

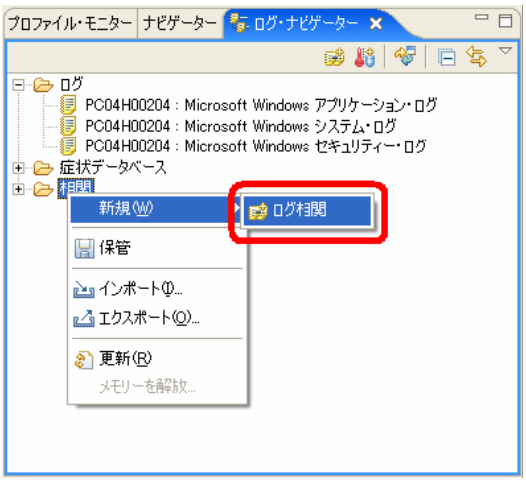
予め関連付け用に Microsoft Windows のアプリケーション・ログとセキュリティー・ログをインポートしておきます。







[ログ・ナビゲーター]ビューで、[相関]フォルダを右クリックしてポップアップメニューから 新規 | ログ相関を選択します。



[インポート]画面で、ログ相関の「名前」を設定します。

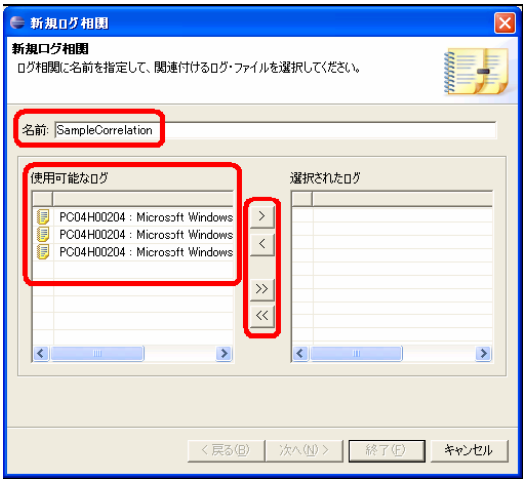
次に「使用可能なログ」から相関させるログを選択して、[>][<][>>][<<]ボタンでログの位置を操作して「選択されたログ」へと移動させます。

今回は以下のように設定します。

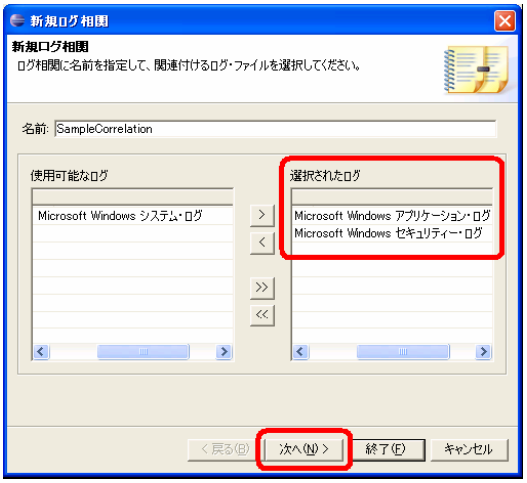
名前：SampleCorrelation

選択されたログ：

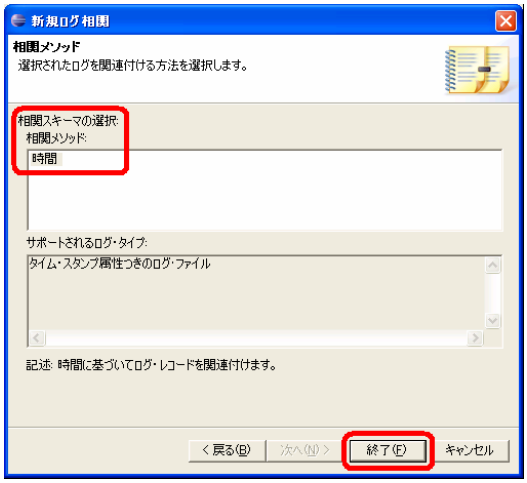
- Microsoft Windows アプリケーション・ログ
- Microsoft Windows のセキュリティー・ログ



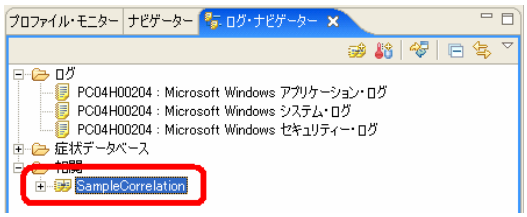
設定を完了すると、[次へ]ボタンを押します。



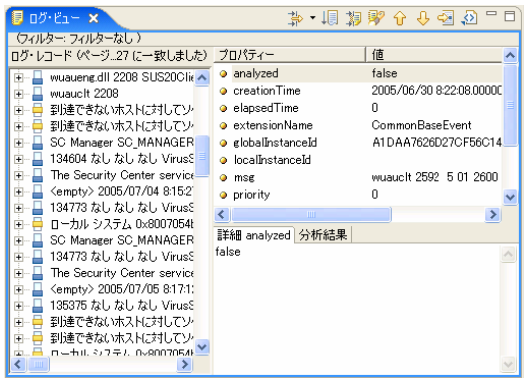
[相関メソッド]では、「相関メソッド」で「時間」が選択されていることを確認して、[終了]ボタンを押します。



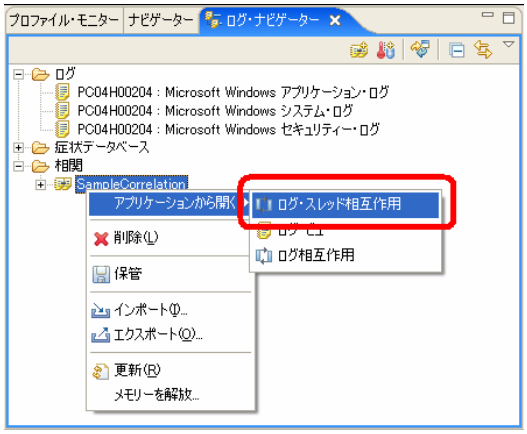
[ログ・ナビゲーター]ビューで、[相関]フォルダの配下に作成したログ相関「SampleCorrelation」が生成されます。



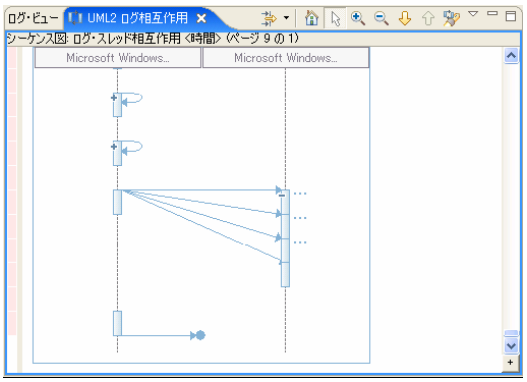
[ログ・ナビゲーター]ビューで、[相関]フォルダの配下に作成したログ相関「SampleCorrelation」が生成されます。



ログ相関「SampleCorrelation」を右クリックして、ポップアップメニューから アプリケーションから開く | ログ・スレッド相互作用 or ログ相互作用を選択します。

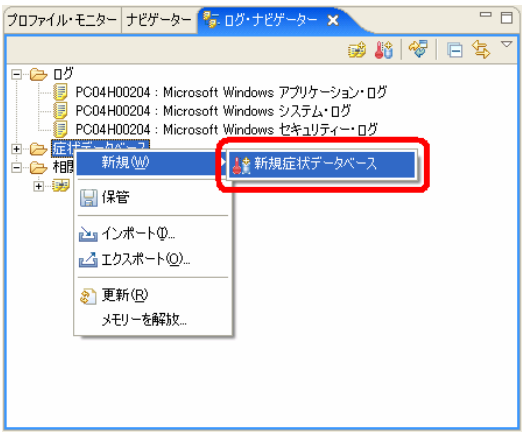


[UML2 ログ相互作用]ビューが表示され、ログ相関のシーケンス図を見ることができます。



まずは症状データベースの作成、編集を行います

[ログ・ナビゲーター]ビューで、[症状データベース]フォルダを右クリックしてポップアップメニューから 新規 | 新規症状データベース を選択します。



[新規症状データベース]画面で、ロケーションや名前や記述の設定を行います。

今回は以下のように設定します。

ロケーション：￥ProfileProject

名前：TestSymptomDB

記述：テスト症状データベース

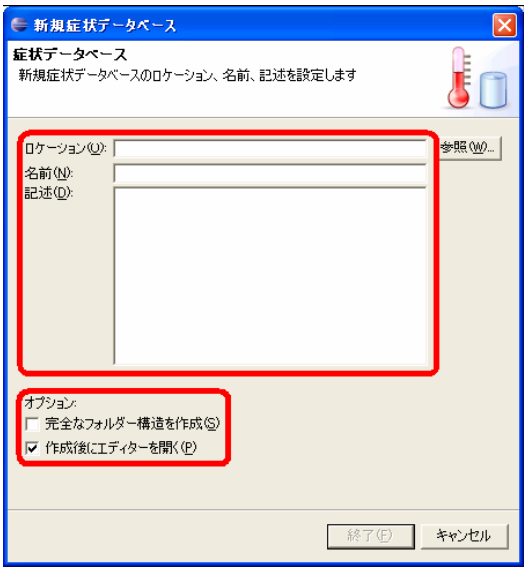
オプションの項目には以下の機能があります。

[完全なフォルダー構成を作成]

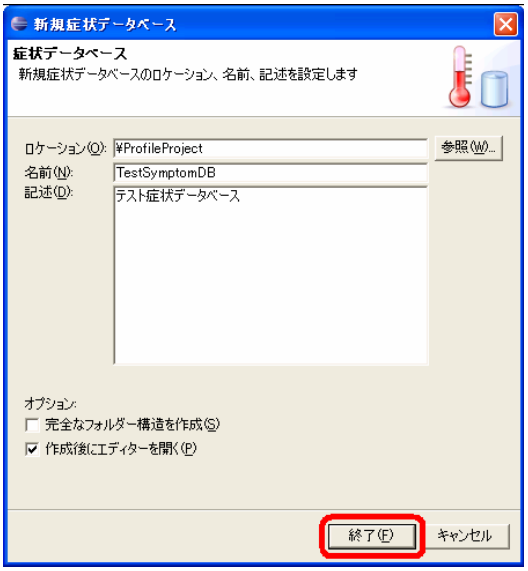
チェックを入れることで、ロケーションが存在しないものでもデータベース生成時に設定したロケーションを作成する。

[作成後にエディターを開く]

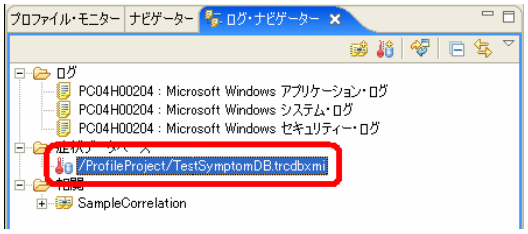
チェックを入れることでデータベース作成後、自動的に症状データベース・エディターを表示します。

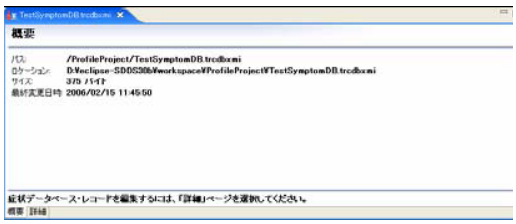


設定を完了すると、[終了]ボタンを押します。

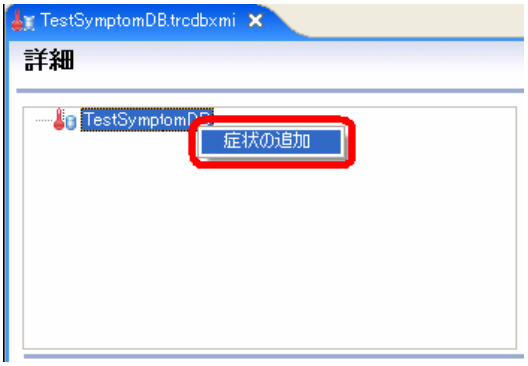


[ログ・ナビゲーター]ビューで、[症状データベース]フォルダ配下に作成した症状データベースが追加されます。
また、症状データベース・エディターが表示されます。





症状データベース・エディターで[詳細]タブを選択します。
表示されている症状データベース「TestSymptomDB」を右クリックして、ポップアップメニューから症状の追加を選択します。



症状データベース「TestSymptomDB」の配下に症状「症状１」が追加され、エディター右側が症状の編集画面に変わります。

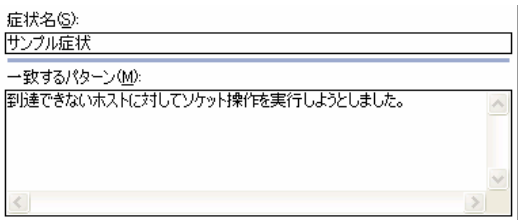
ここでは以下のように入力します。

症状名：サンプル症状

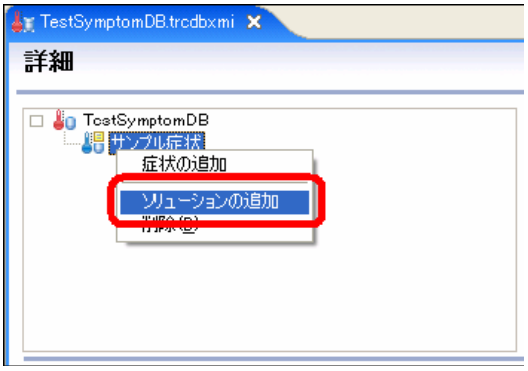
一致するパターン：到達できないホストに対してソケット操作を実行しようとしました。

ファイル｜保管　を行うことでエディター左側のツリーに反映されます。

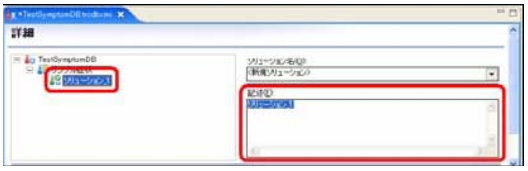
今回の使用する[一致するパターン]は、ログ・レコード中の msg に存在するものを入力してください。



症状「サンプル症状」を右クリックして、ポップアップメニューからソリューションの追加を選択します。

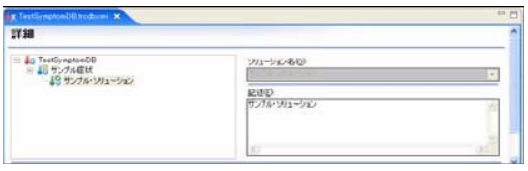


症状「サンプル症状」の配下にソリューション「ソリューション１」が追加され、エディター右側がソリューションの編集画面に変わります。

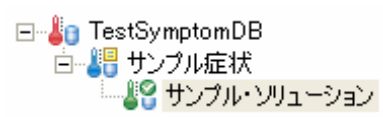


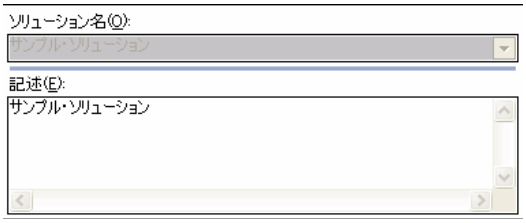
ここでは以下のように設定します。

記述：サンプル・ソリューション

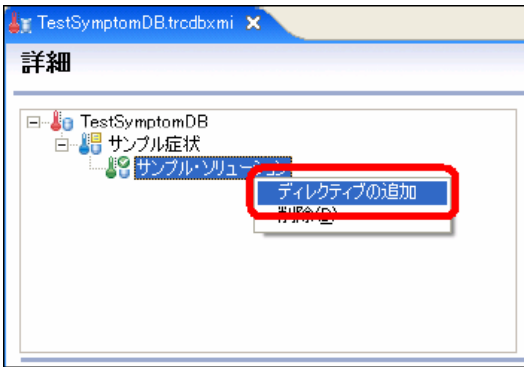


ファイル｜保管　を行うことでエディター左側のツリーに反映されます。

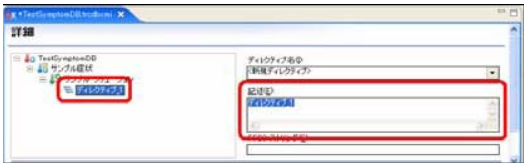




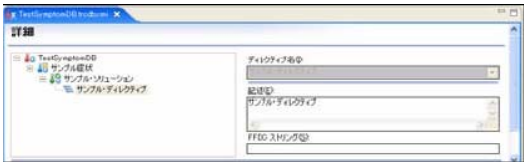
ソリューション「サンプル・ソリューション」を右クリックして、ポップアップメニューからディレクティブの追加を選択します。

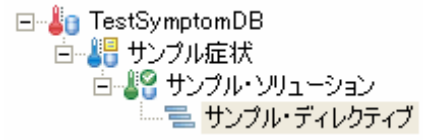


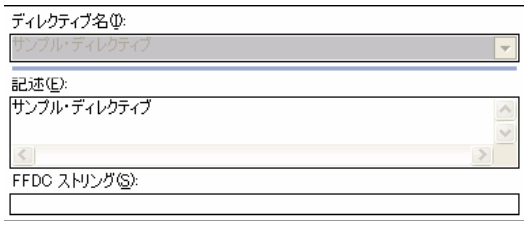
ソリューション「サンプル・ソリューション」の配下にディレクティブ「ディレクティブ1」が追加され、エディター右側がディレクティブの編集画面に変わります。



ここでは以下のように入力します。
記述：サンプル・ディレクティブ
ファイル｜保管 を行うことでエディター左側のツリーに反映されます。

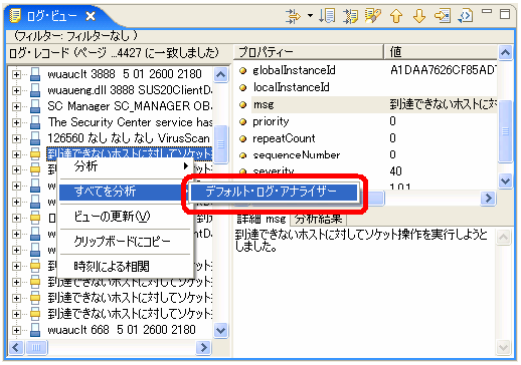




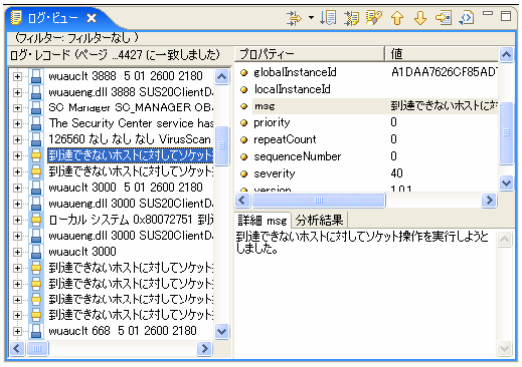


これで症状データベースの作成、編集を完了します。
次に作成した症状データベースを用いてログの分析を行います。

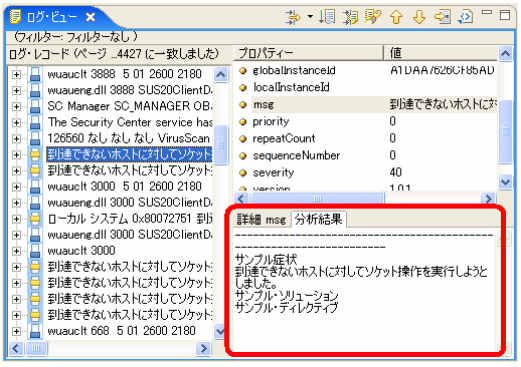
分析を行いたいログを[ログ]ビューで表示して、ログ・レコード上で右クリックし、ポップアップメニューから すべて分析 | デフォルト・ログ・アナライザーを選択します。



分析の結果、症状データベースの「一致するパターン」で定義した文字列と一致しなかったログ・レコードは青の枠で囲まれ、一致したログ・レコードは青で塗りつぶされます。



一致したログ・レコードの[分析結果]タブを見ると、症状データベースで定義した内容が表示されます。



Generic Log Adapter のアダプター・ファイルの作成・編集などによるログの関連付けの拡張機能については、サポート対象外です。

1.1.7.コンポーネントテスト

PTP Component Test はプログラムテストのフレームワークで、テストの定義、テストの実行、結果確認といったテストの一連の作業において、異なる種類のテストを共通した構成で表現して管理します。

TPTP Component Test では以下の 3 種類のテストが実行できます。

TPTP JUnit Test

TPTP から JUnit テストを行います。

TPTP URL Test

HTTP アプリケーションの自動化されたパフォーマンス・テストを行います。

TPTP Manual Test

手動によるテストを行います。

ここではそれぞれのテストを実行して説明します。

また TPTP Component Test では多くのリソース・ファイルを扱うため、 ファイルの種類ごとに保管・配置する場所を決めておくことで管理が容易になります。本ガイドではファイルの保管・配置を次のように行います。

src フォルダ :テストのソース・コードを格納します。

bin フォルダ :src フォルダをコンパイルした結果を格納します。

tests フォルダ :TestSuite を格納します。

(内部に junit、url、manual の 3 つのファルダを追加します)

TestSuite ・・・ テストの内容を定義するリソース

tests/datapool フォルダ : Datapool を格納します。

Datapool ・・・ テストデータの集合を定義するリソース

results フォルダ :テストの実行結果やレポートを格納します。

(内部に junit、url、manual の 3 つのファルダを追加します)

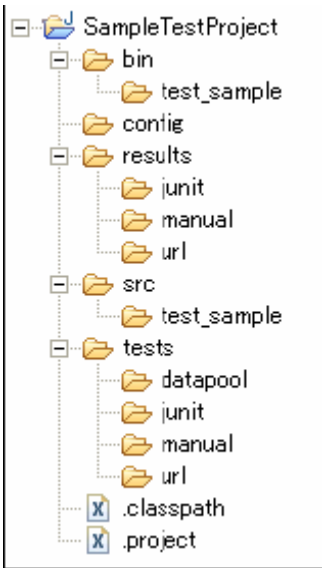
config フォルダ :Artifact、 Location、 Deployment を格納します。

Artifact ・・・ 実行するテストを指定するリソース

Location ・・・ テストを実行するマシン(ホスト)を指定するリソース

Deployment ・・・ テストの配備を表現、Artifact と Location の組み合わせを指定するリソース

[ナビゲーター]ビューで表示した構成です。

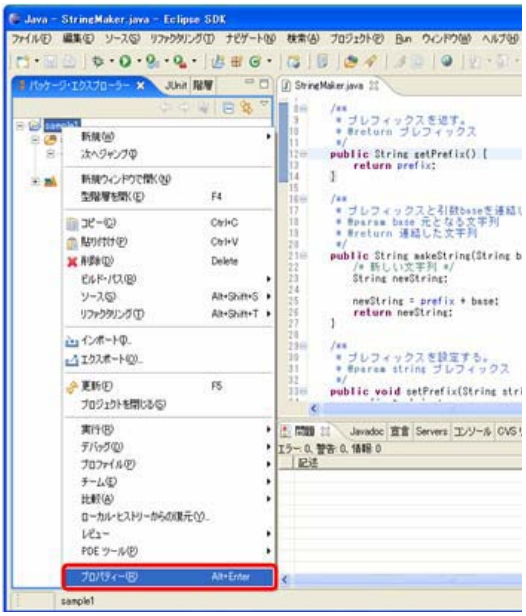


メニューから ウィンドウ | パースペクティブを開く | Java を選択して、予めパースペクティブを切り替えておきます。

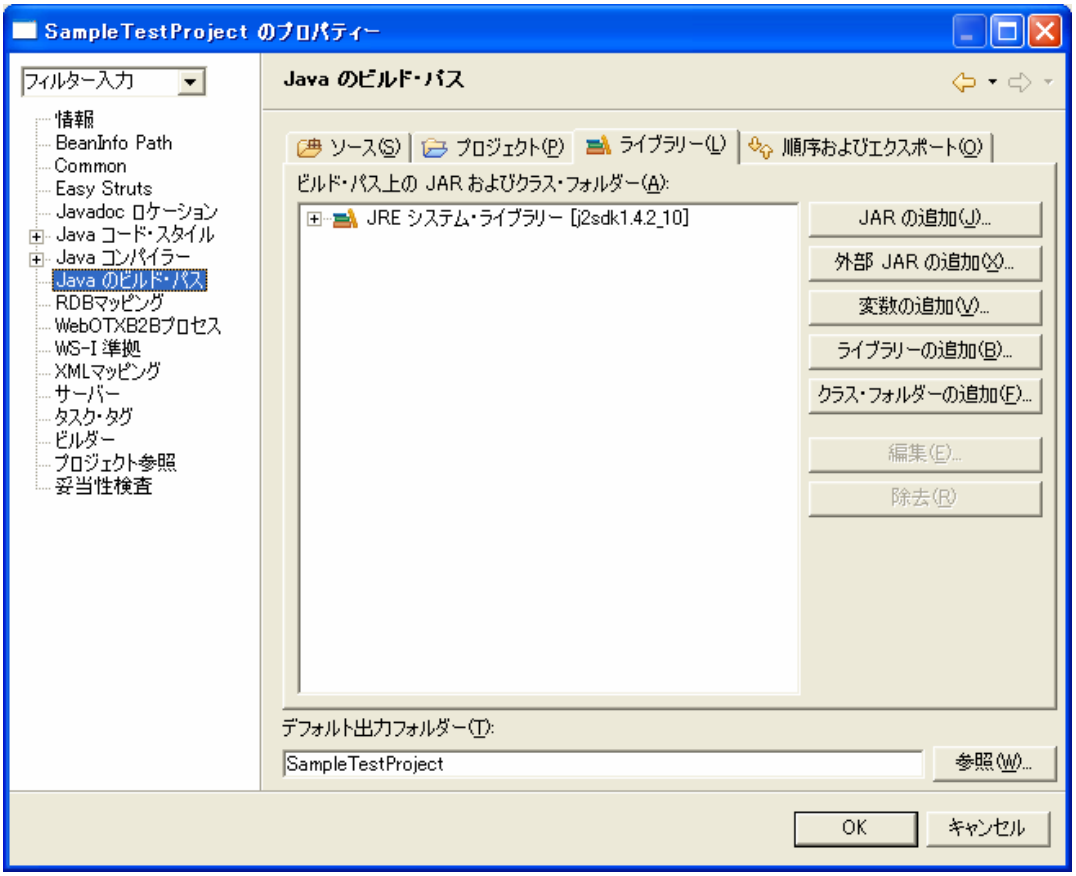
JUnit の準備

JUnit を使用してテストを行う場合、JUnit のクラスの JAR である「junit.jar」がクラスパスに存在する必要があります。Eclipse 上から JUnit を使用するためにプロジェクトに対してクラスパスを設定します。

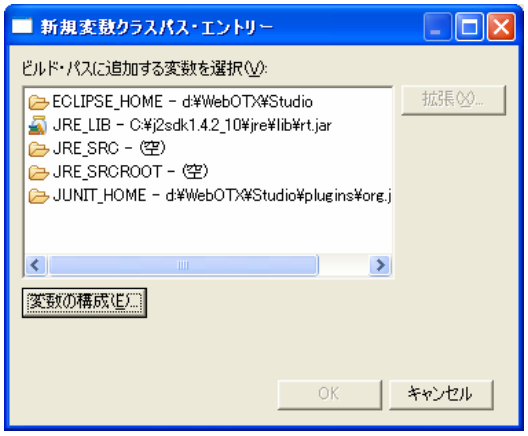
まず、パッケージ・エクスプローラーでプロジェクトを右クリックし、ポップアップメニューからプロパティーを選択します



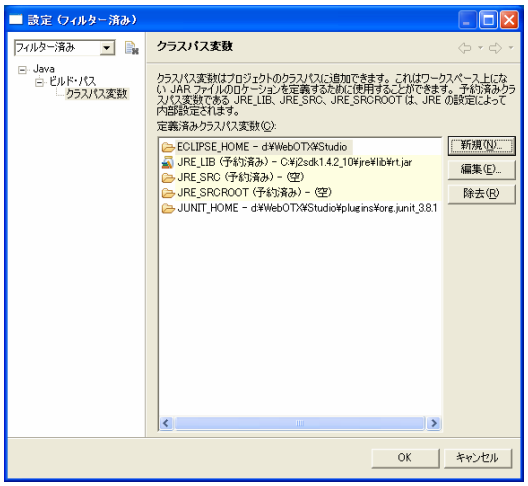
プロパティー画面の左ペインで「Java のビルド・パス」を選択して、右ペインの「ライブラリー」タブを開き、[変数の追加]ボタンを押します。



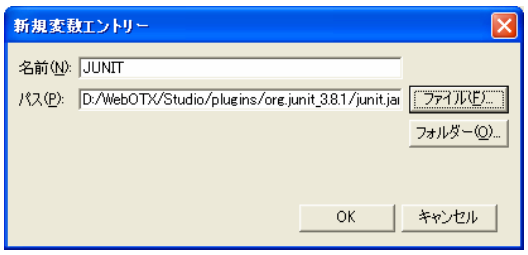
新規変数クラスパス・エントリー画面にて、「JUNIT」を選択し、[変数の構成]ボタンを押します。



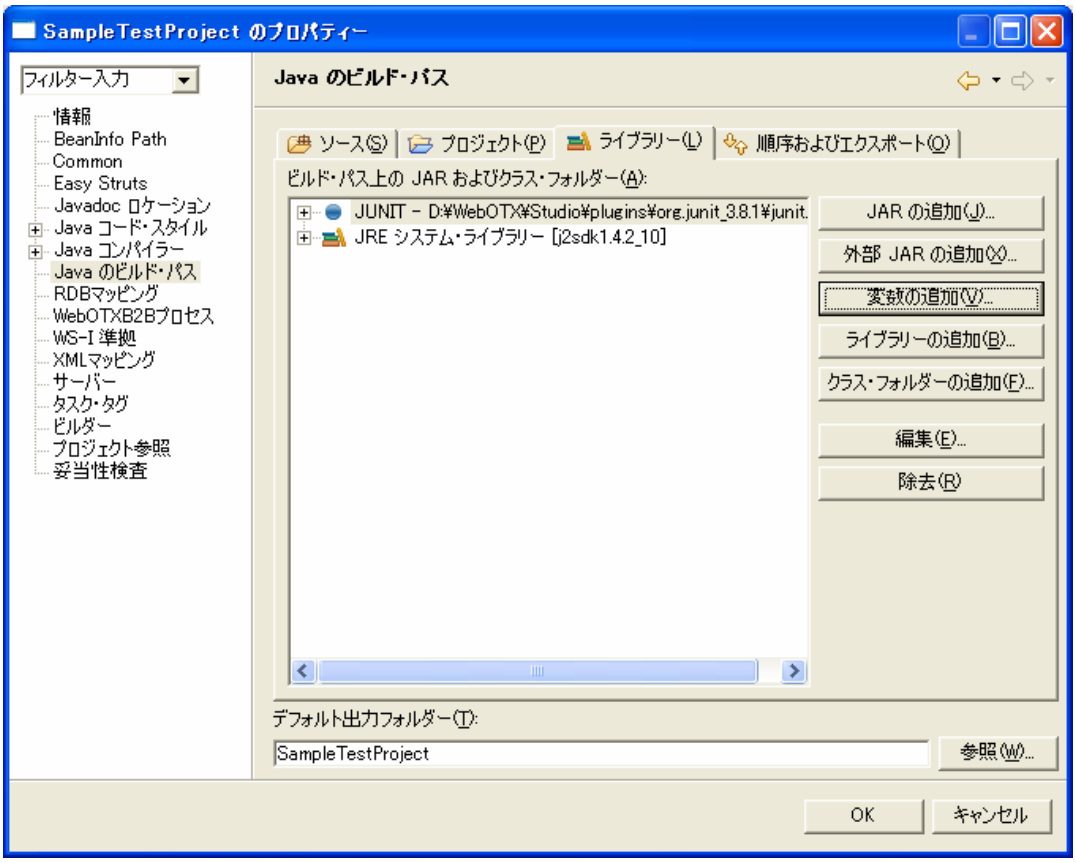
新規ボタンを押します。



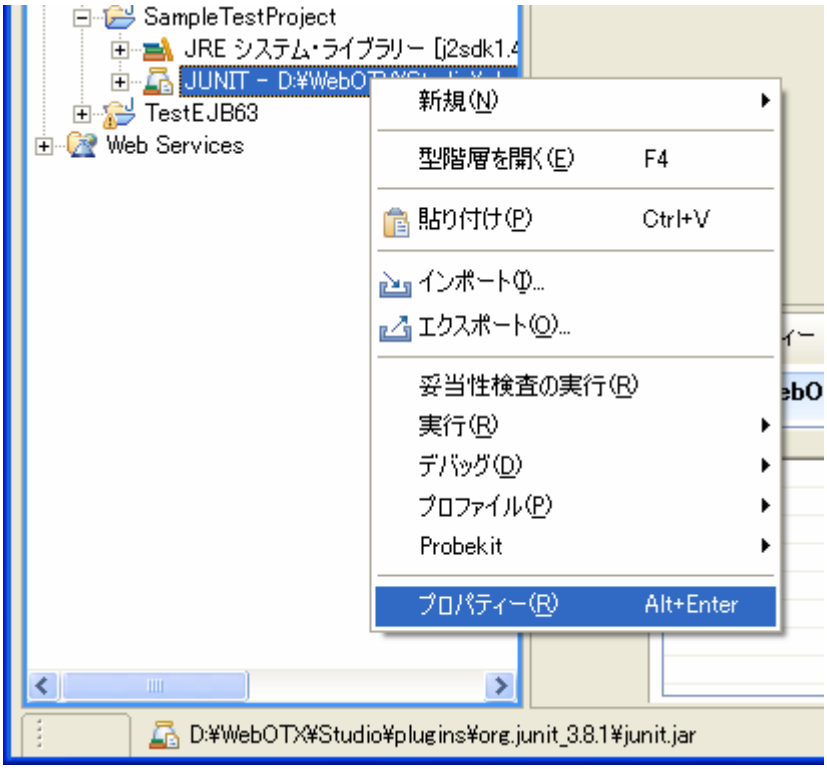
JUNIT の定義を行います。



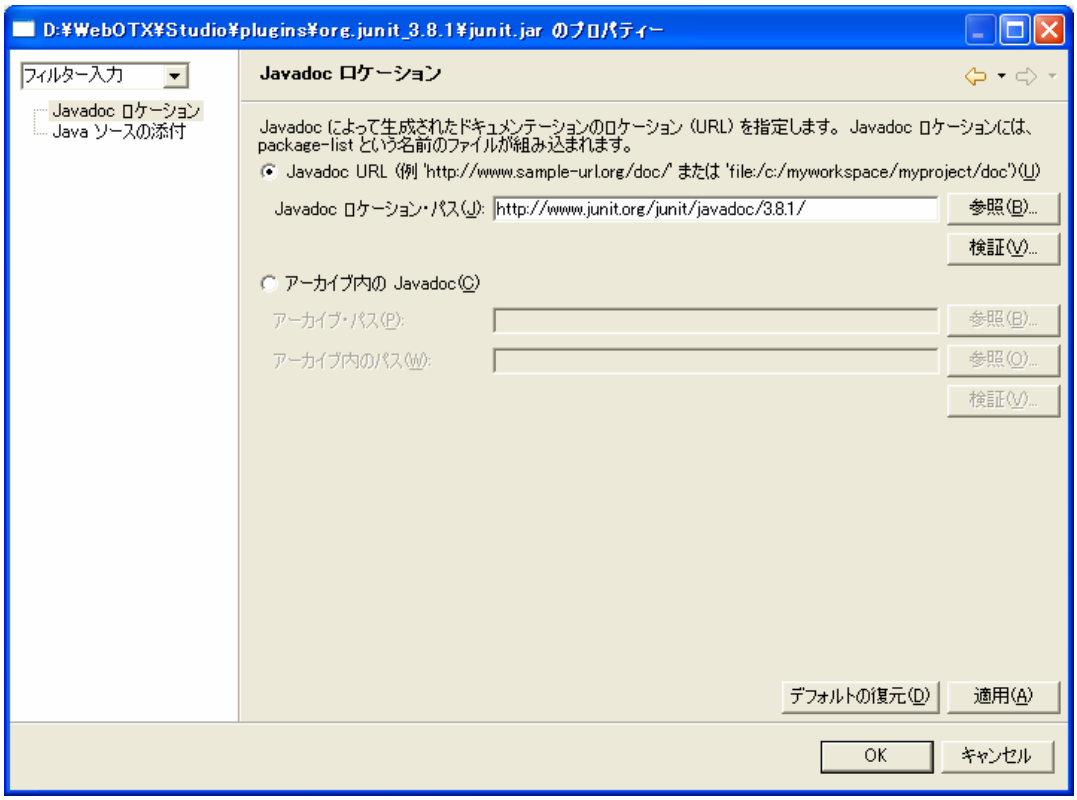
以下のようにjunit.jar へのクラスパスが追加されたことを確認してから、[OK]ボタンを押して、プロパティー画面を閉じます。



JUNIT のプロパティを開きます。

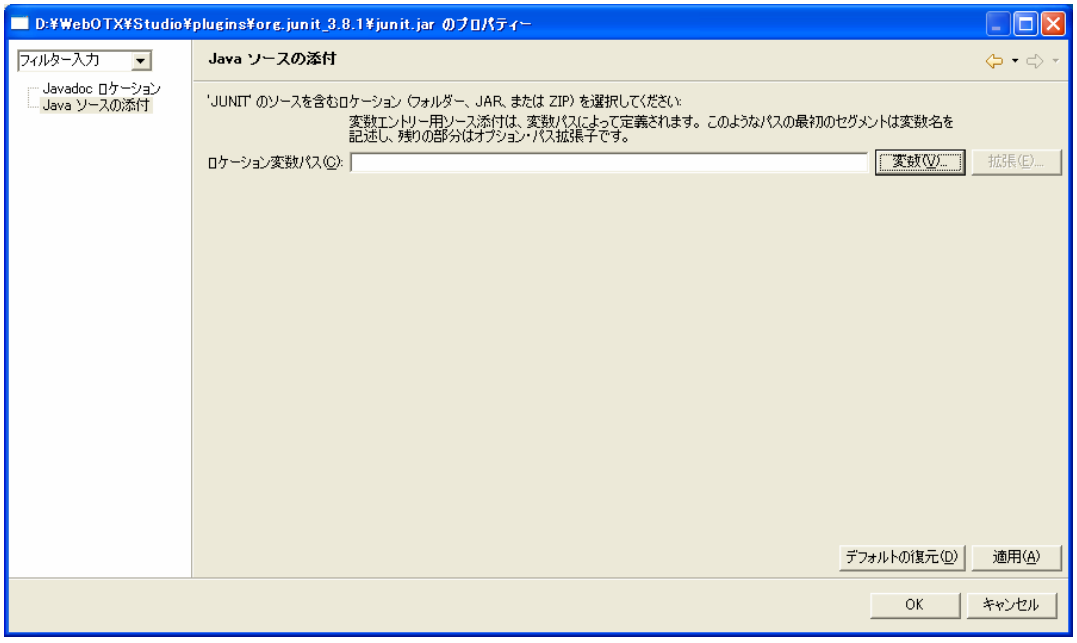


左ペインから Javadoc ロケーション を選択し、右ペインに javadoc へのパスを入力します。入力後、[適用] ボタンを押します。

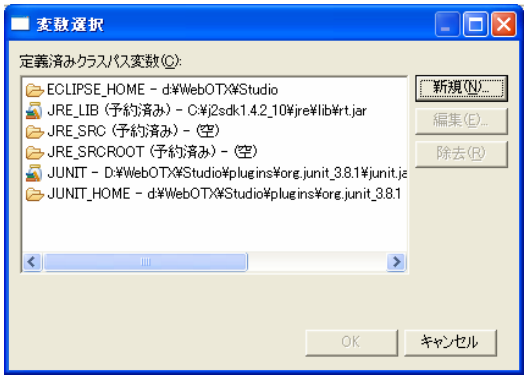


妥当性の検査は失敗する場合があります。ブラウザで「http://www.junit.org/junit/javadoc/3.8.1/index.html」が参照できれば、問題ありません。

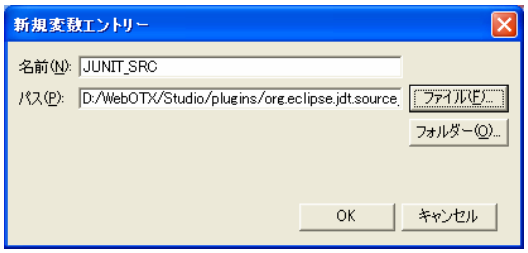
Java ソースの添付を選択し、変数ボタンを押します。



新規を押します。



JUNIT_SRC を設定します。

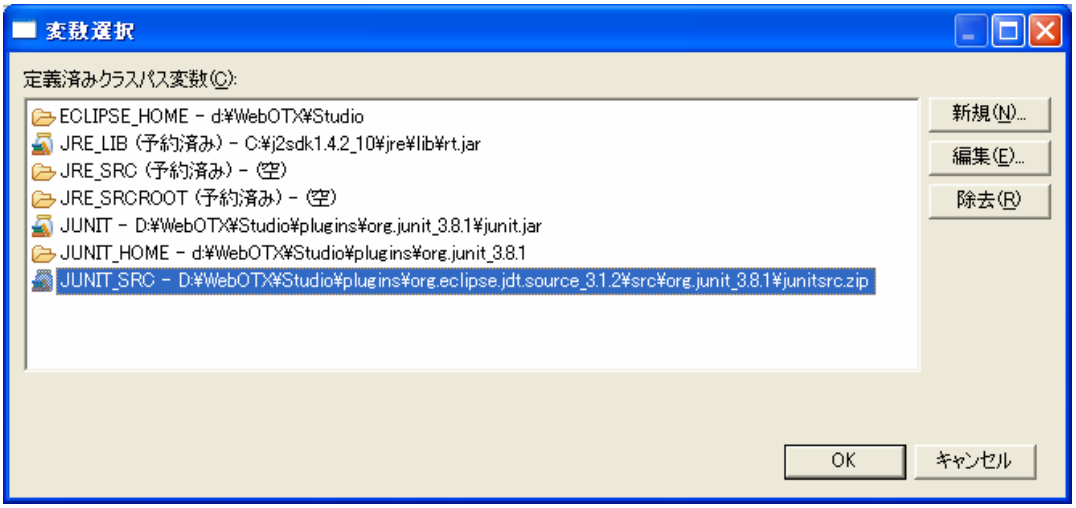


パスは、

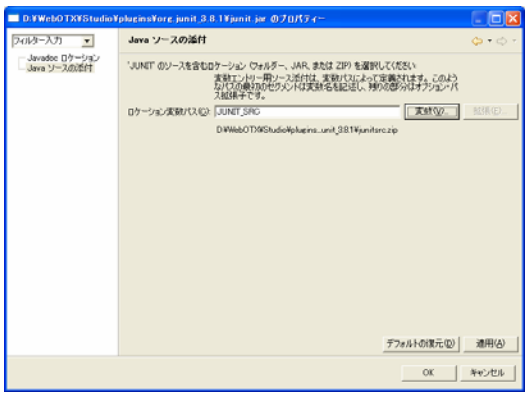
[WebOTX インストールディレクト]¥Studio¥plugins¥org.eclipse.jdt.source_3.1.2¥src¥org.junit_3.8.1¥junitsrc.zip

になります。

OK ボタンを押します。



OK ボタンを押します。



JUnit テスト

「8.1.1.Java ソースの新規作成」で作成したサンプルプログラム ProfileApp.java を用いて、JUnit テストを行います。サンプルプログラムの詳細はそちらを参照してください。

ProfileApp.java を以下の場所にコピーしてください。(プログラム中パッケージ名の修正も行ってください。)

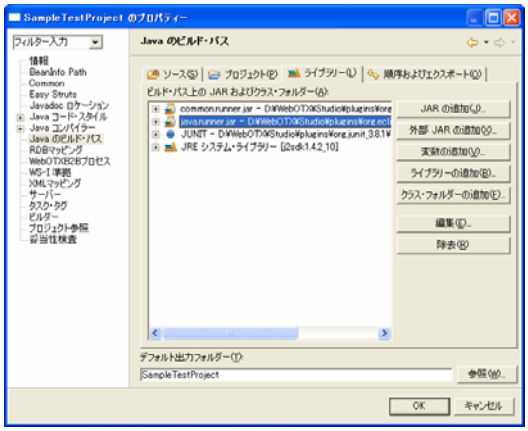
SampleTestProject/src/test_sample/ ProfileApp.java

SampleTestProject の Java ビルド・パスに以下のビルド・パスをプロジェクトに追加してください。

- ECLIPSE_HOME/plugins/org.eclipse.hyades.test.tools.core_4.0.1/common.runner.jar
- ECLIPSE_HOME/plugins/org.eclipse.hyades.test.tools.core_4.0.1/java.runner.jar

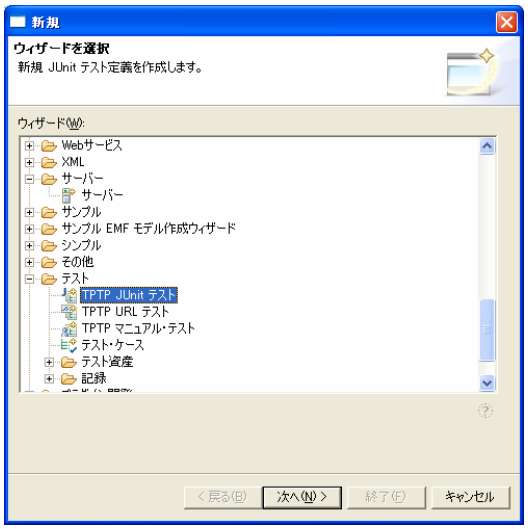
ビルド・パスの追加はプロジェクトを右クリックして、ポップアップメニューからプロパティーを選択、「***のプロパティー」画面で Java のビルド・パス | ライブラリー | [外部 JAR の追加]ボタン で行ってください。

ビルド・パスの追加が完了したら、[OK]ボタンを押します。



テスト・スイートではテストの内容を定義して、テスト・クラスを自動生成や、テスト・メソッドおよびテスト・メソッドの振る舞いの定義をします。テスト・スイートを作成します。

メニューから ファイル | 新規 | その他 を選択します。[新規]画面で、テスト | TPTPJUnit テスト を選択して、[次へ]ボタンを選択します。



[新規 JUnit テスト定義]画面が表示されます。

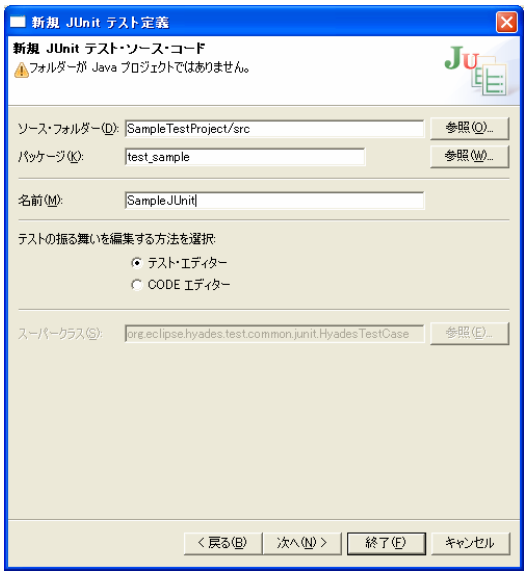
ここではテスト・ソース・ファイルの生成場所を設定します。

以下のように入力して[次へ]ボタンを押します。

ソース・フォルダー：SampleTestProject/src

パッケージ：test_sample

名前：SampleJUnit



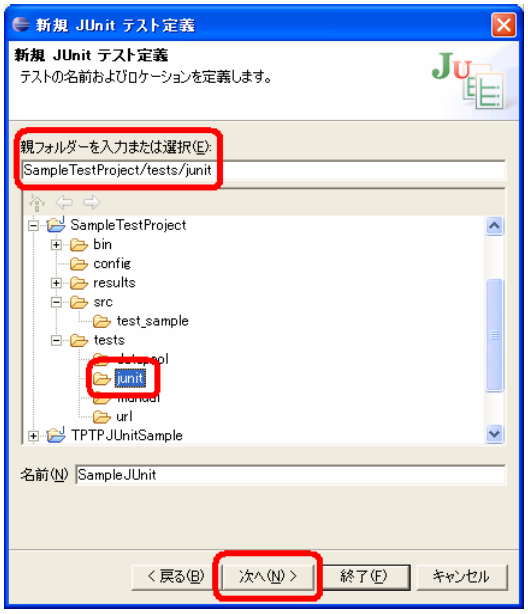
次にテスト・スイート･ファイルの生成場所を設定します。以下のように設定して[次へ]ボタンを押します。

親フォルダーを入力または選択：

SampleTestProject/tests/junit

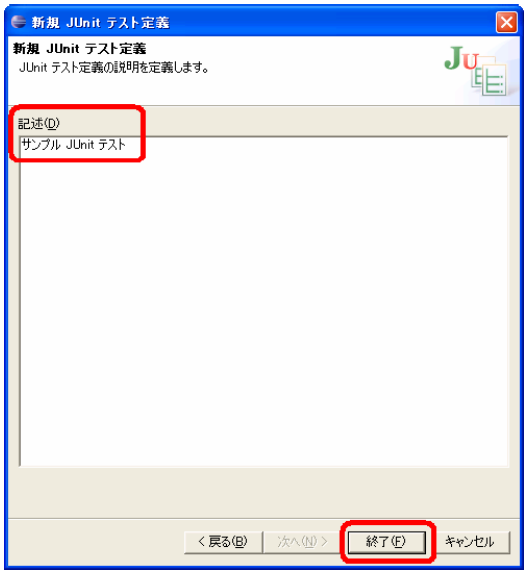
名前： SampleJUnit

(親フォルダーは直接入力しても、ツリーから選択してもかまいません)

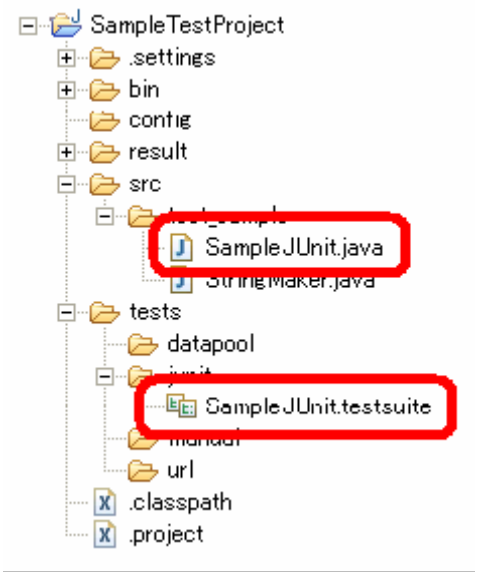


最後にテスト・スイートの説明を記述します。以下のように設定して[終了]ボタンを押します。

記述：サンプル JUnit テスト

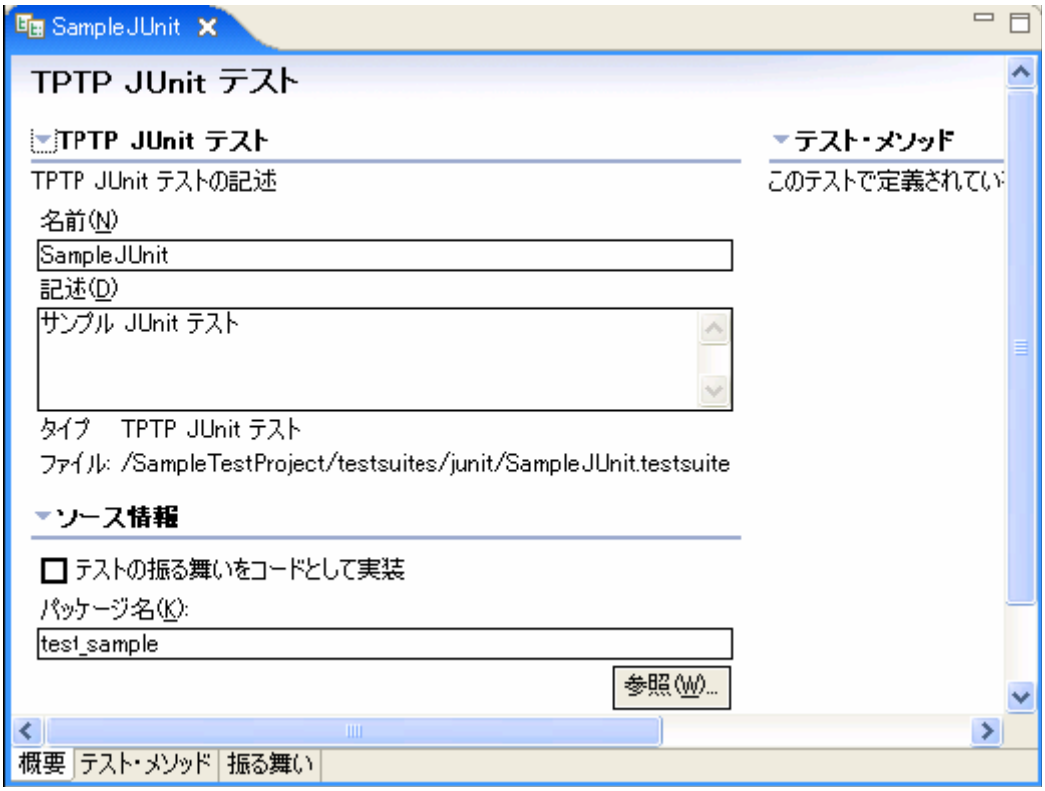


[新規 JUnit テスト定義]を終了すると以下のリソースが生成されます。



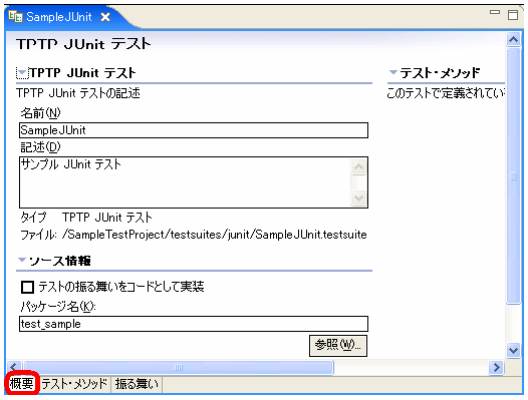
次に生成したテスト・スイート・ファイル SampleJUnit.testsuite の編集を行います。

SampleJUnit.testsuite をダブル・クリックしてテスト・スイート・エディターを表示させてください。



今回は 2 つのテスト・メソッドを定義して、その実行順序(振る舞い)を設定します。

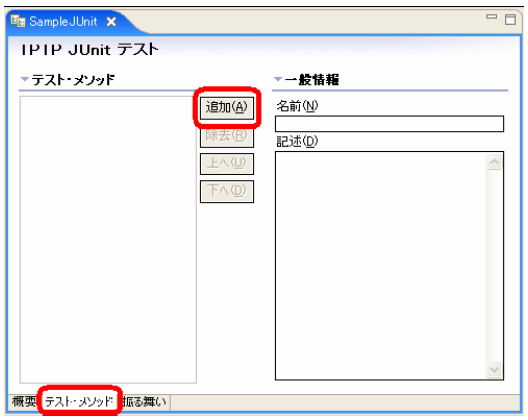
[概要]タブでは、そのテスト・スイート・ファイルの概要を確認することができます。ここでは特に編集は行いません。



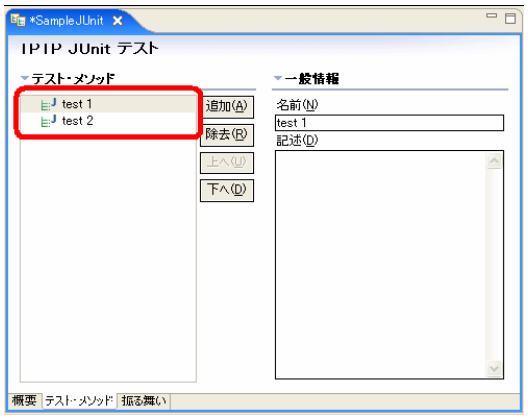
[テスト・メソッド]タブでは、テスト・メソッドの定義や編集を行うことができます。

今回は以下の 2 つのテスト・メソッドをテスト・クラスに追加します。

- testMakeString
- testSetGetPrefix



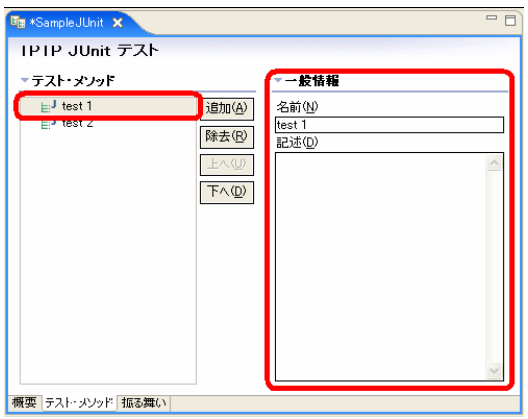
[追加]ボタンを押して、テスト・メソッドを追加します。



追加されたテスト・メソッドを選択して一般情報で以下のように編集してください。

・名前：testMakeString

記述： **MakeString()のテスト。事前にsetPrefix(arg1)で文字列を設定し、makeString(arg2)を呼び出すことにより、返却される文字列が arg1、arg2 の順に結合された文字列であることをテストする。**

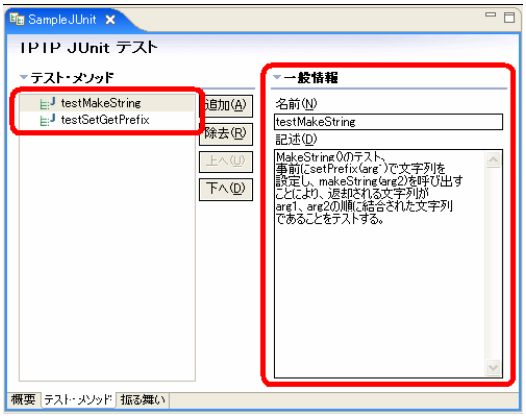


・名前：testSetGetPrefix

記述： **setPrefix()と getPrefix()のテスト。private メンバ prefix の setter,getter のテストを行う。**

名前はテスト・メソッド名、記述はテスト・メソッドへのコメントとなります。

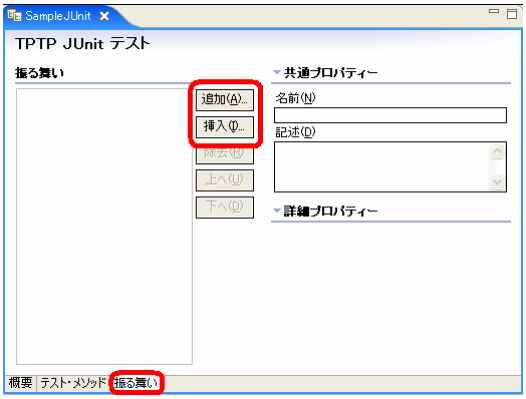
これでテスト・クラスに 2 つのテスト・メソッドを追加できました。



[振る舞い]タブでは、テスト・メソッドの実行順序や実行回数の設定を行うことができます。

先に追加した 2 つのテスト・メソッドの実行順序と実行回数を設定します。

[追加][挿入]ボタンでループや呼び出しを設定して行きます。



今回は以下の手順に従って設定してください。

始めに[追加]ボタンで、振る舞いにループを追加します。

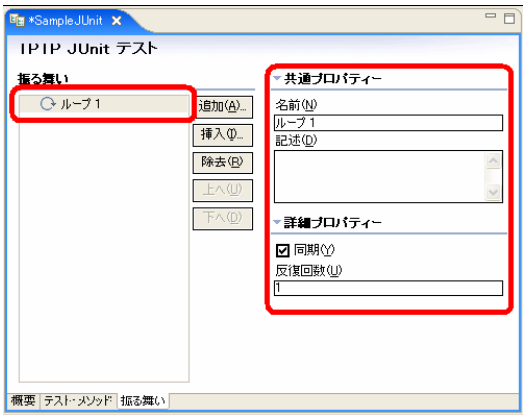
追加したループを以下のように入力します。

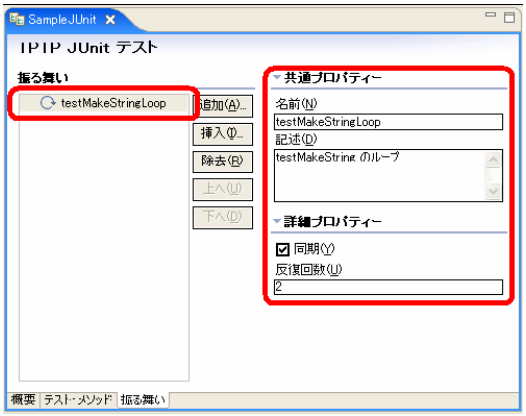
名前： testMakeStringLoop

記述： testMakeString のループ

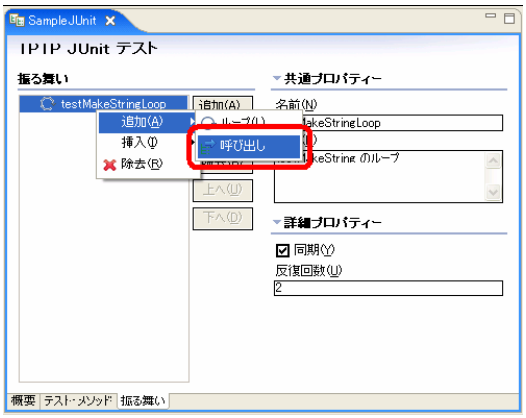
反復回数：2

振る舞い上のループを右クリックして、ポップアップメニューから 追加 | 呼び出し を選択します。





振る舞い上のループを右クリックして、ポップアップメニューから 追加 | 呼び出し を選択します。



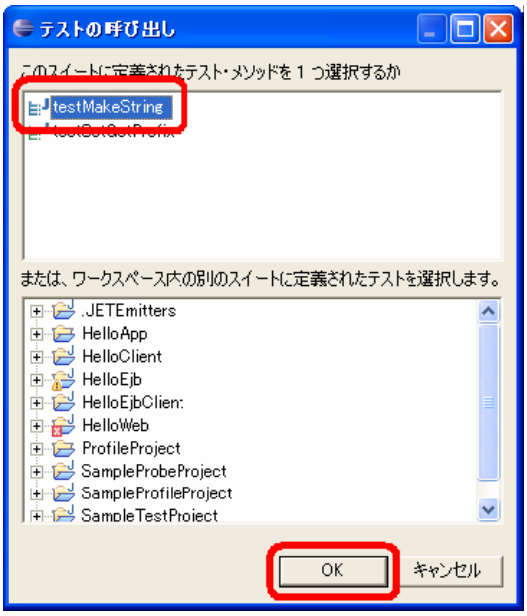
[テストの呼び出し]画面で、テスト・メソッドを選択して、[OK]ボタンを押します。

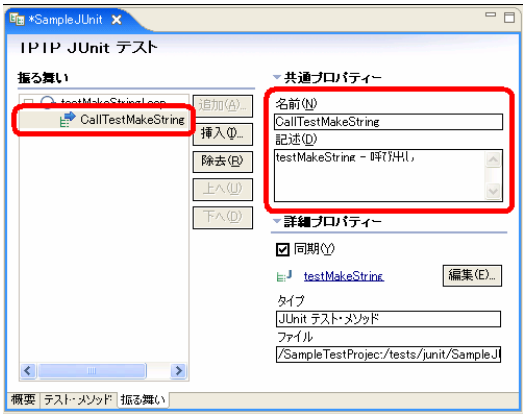
ここでは、テスト・メソッド testMakeString を呼び出します。

振る舞いのループの配下に呼び出しが追加されます。追加された呼び出しの共通プロパティーを以下のように設定します。

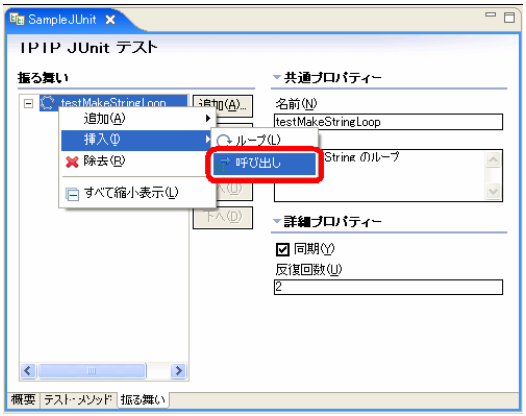
名前：CallTestMakeString

記述：testMakeString – 呼び出し





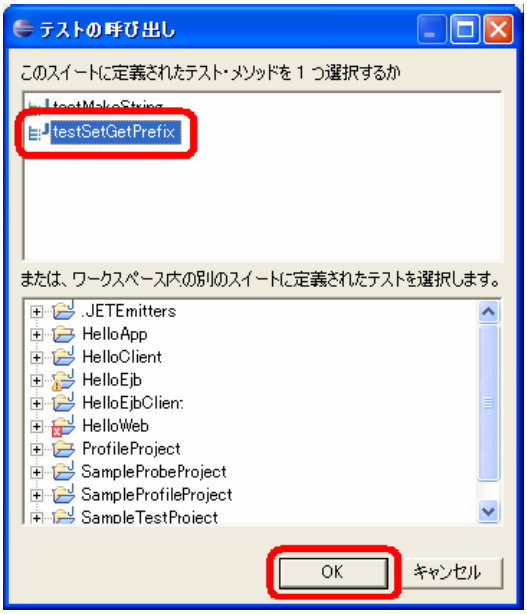
次に振る舞い上のループを右クリックして、ポップアップメニューから 挿入 | 呼び出し を選択します。

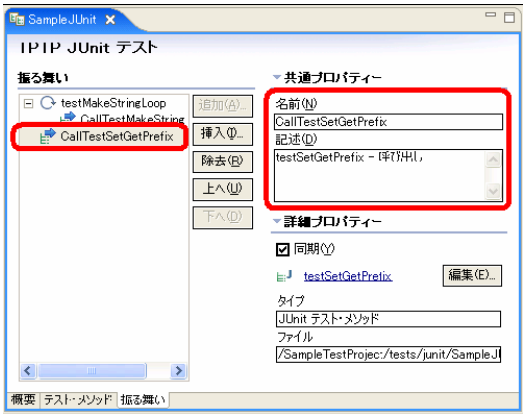


先ほどと同様に[テストの呼び出し]画面で、テスト・メソッドを選択して、[OK]ボタンを押します。ここでは、テスト・メソッド testSetGetPrefix を呼び出します。振る舞いのループと同じ階層に呼び出しが追加されるので、追加された呼び出しの共通プロパティーを以下のように設定します。

名前：CallTestSetGetPrefix

記述：testSetGetPrefix – 呼び出し





テスト・スイートの編集は以上で終わります。ファイル | 保管 を行います。

次にテスト・クラス・ファイル SampleJUnit.java の実装を行います。

テスト・スイート・ファイルの編集で以下のメソッドが追加されています。

```
/**
* testMakeString
*
* MakeString()のテスト。
```

```
* 事前に setPrefix(arg1)で文字列を
* 設定し、makeString(arg2)を呼び出す
* ことにより、返却される文字列が
* arg1、arg2 の順に結合された文字列
* であることをテストする。
*
* @throws Exception
*/
public void testMakeString() throws Exception{
	// Enter your code here
}
/**
* testSetGetPrefix
*
* setPrefix()と getPrefix()のテスト。
* private メンバ prefix の setter,getter のテストを行う。
*
* @throws Exception
*/
public void testSetGetPrefix() throws Exception{
	// Enter your code here
}
```

追加されたテスト・メソッドを実装します。(ここではそれぞれ 2 つのケースについてテストします。)

```
public void testMakeString() throws Exception{
	StringMaker obj = new StringMaker();
	obj.setPrefix("Mr.");
	System.out.println(obj.makeString("Smith"));
	assertEquals("Mr.Smith", obj.makeString("Smith"));
}
```

```
public void testSetGetPrefix() throws Exception{
	StringMaker obj = new StringMaker();
	obj.setPrefix("pre");
	System.out.println(obj.getPrefix());
	assertEquals("pre", obj.getPrefix());
}
```

以上でメソッドの実装が終了します。ファイル | 保管 を行います。

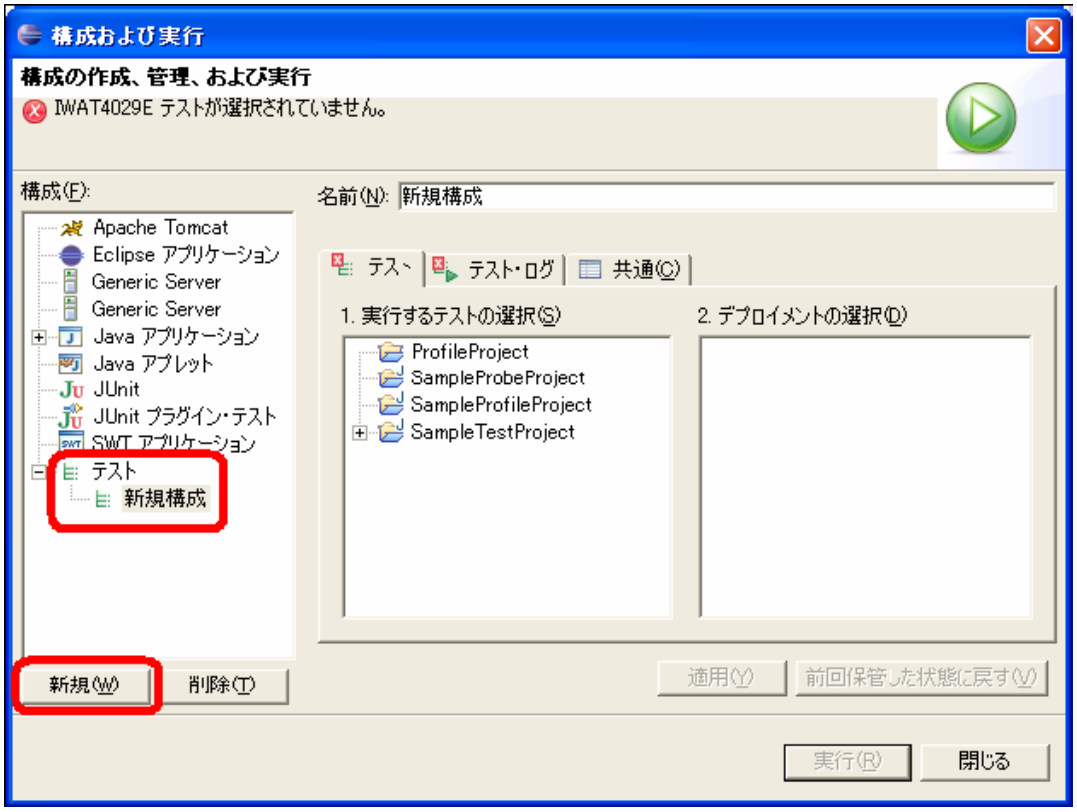
次にテストを実行します。

メニューの実行 | 構成および実行 を選択します。

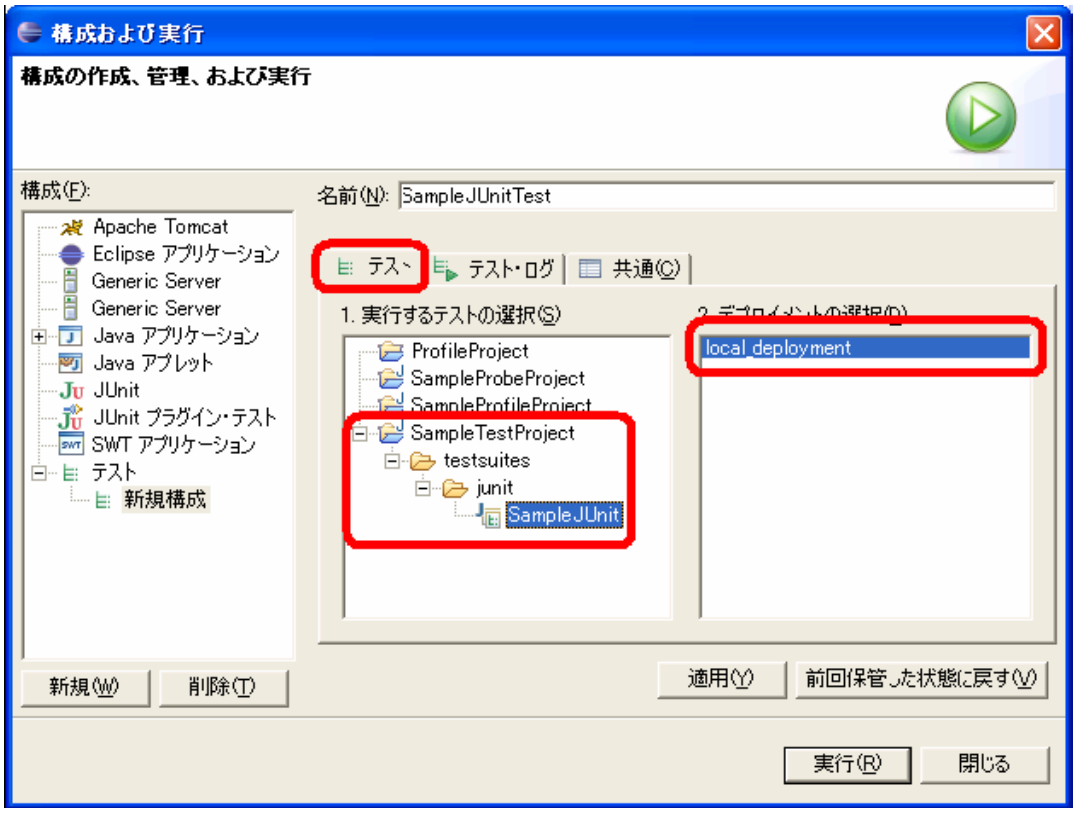
[構成および実行]画面が表示されます。

構成のテストを選択して[新規]ボタンを押して、新規構成を作成します。

名前を SampleJUnitTest と設定します。



[テスト]タブで、実行するテスト（SampleTestProject/TPTP JUnit テスト/tests/junit/SampleJUnit）を選択します。実行するテスト・ソースを選択すると、デプロイメントの選択に local_deployment が追加され、自動的に選択されます。



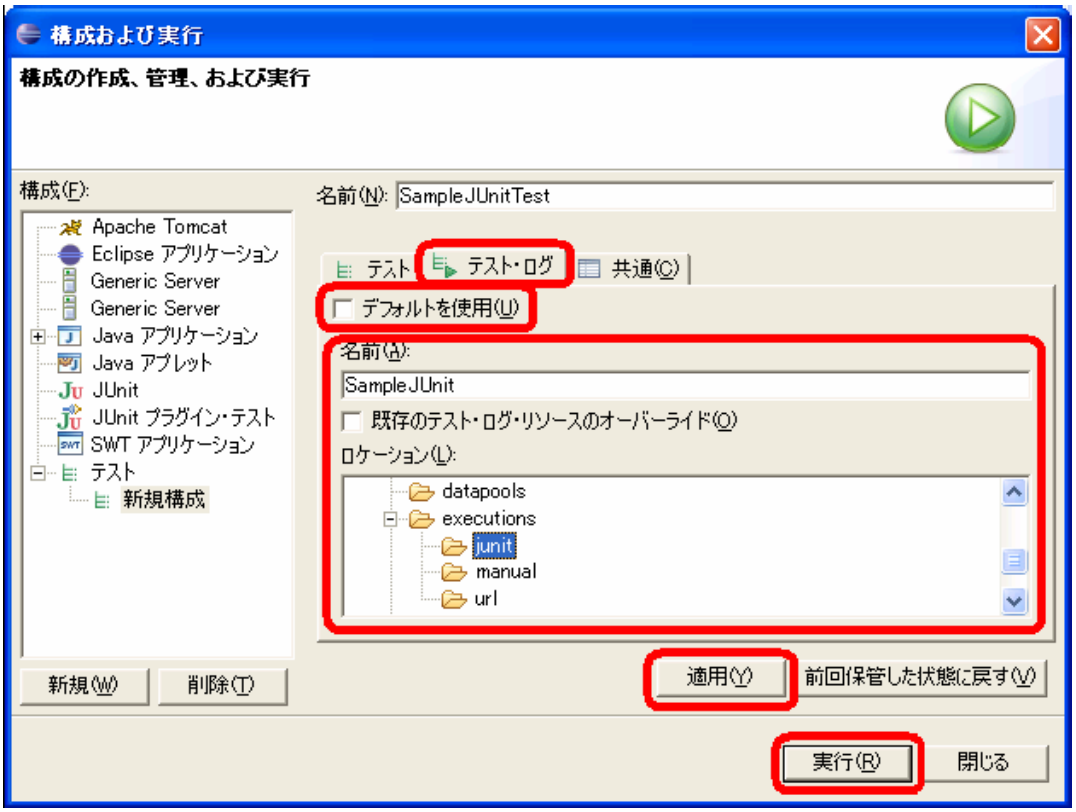
次に[テスト・ログ]タブで、テスト結果ファイルの保管場所を設定します。

デフォルトを使用からチェックを外して、以下のように設定します。

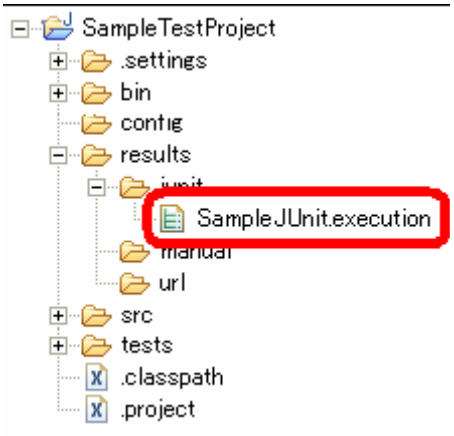
名前：SampleJUnit

ロケーション：SampleTestProject/results/junit

設定が完了しましたら、[適用]ボタンを押して実行構成を保管して、[実行]ボタンを押してテストを実行します。



テストが終了すると、以下の実行結果ファイルが作成されます。



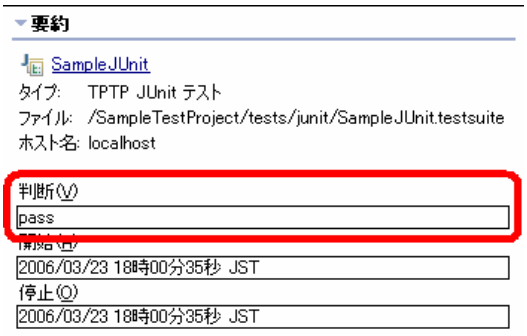
次に実行結果の確認を行います。

実行結果ファイル SampleJUnit.execution をダブル・クリックして、エディターに表示してください。

[概要]タブでは、そのテスト結果の概要を確認することができます。

[概要]タブの要約の判断で、テストの成否の判定を行えます。

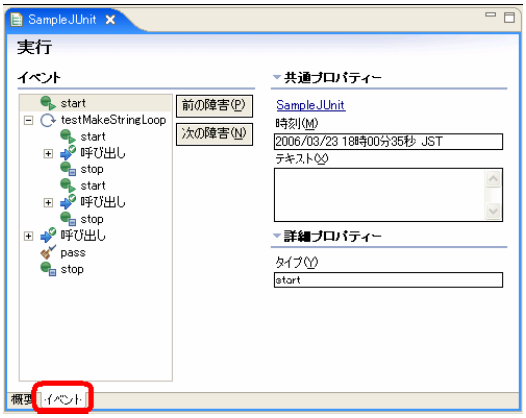
今回は pass と表示されているため、テストは成功したと判定できます。



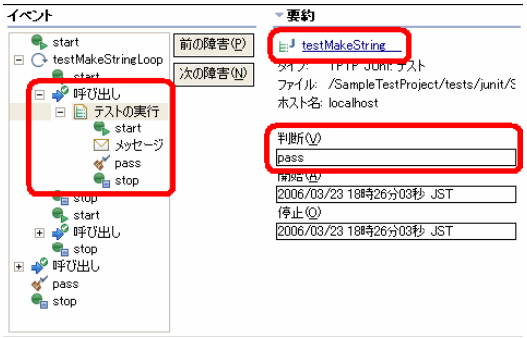
[イベント]タブでは、そのテストの各イベントについて確認することができます。

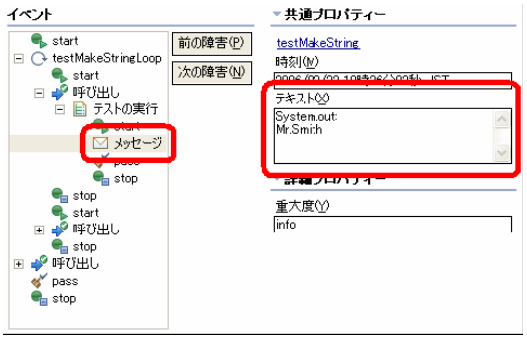
イベントのペインのツリーを展開すると、ループで呼び出しが 2 回行われていること、ループ後に呼び出しが1回行われていることが確認できます。

またイベントペイン最後の pass から全テストが成功しているとわかります。

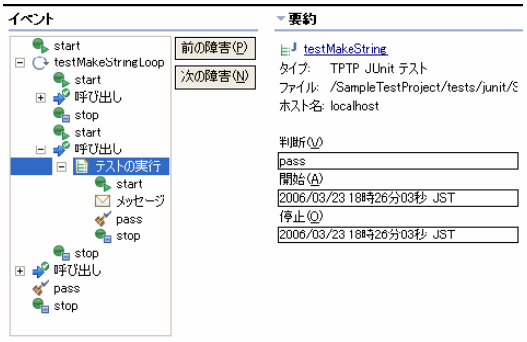


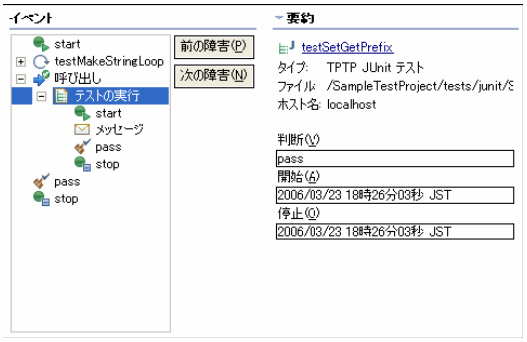
ループ中の呼び出しのテストの実行では testMakeString テスト・メソッドを行っていることがわかります。
また判断で pass とあることや、イベントペインのテスト実行のから testMakeString メソッドの実行が成功しているとわかります。またテスト・メソッド中の System.out.println()などの出力は、イベントペイン中のメッセージのテキストに表示されます。





同様にループ中の 2 つ目の呼び出しでも、testMakeString テスト・メソッドを行っていること、ループ後の呼び出しでは、 testSetGetPrefix テスト・メソッドを行っていることがわかります。





イベント[メッセージ]の[テキスト]にはプログラム中の出力を表示されますが、日本語は正しく表示されません。
テスト・メソッドを以下のように変更して実行すると次のようになります。

```
public void testMakeString() throws Exception{
        StringMaker obj = new StringMaker();
        obj.setPrefix("Mr.");
        System.out.println(obj.makeString("スミス"));
        assertEquals("Mr.スミス", obj.makeString("スミス"));
}
```

URL テスト

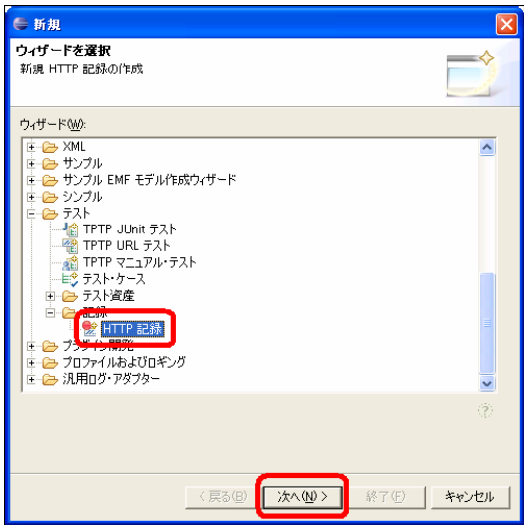
ここでは URL テストを行います。

テストを行いたい Web アプリケーションなどを予め起動させておきます。

またインターネットの設定で自動構成スクリプトを使用している場合は、設定を OFF にしておいてください。設定の方法は、ブラウザのメニューから ツール | インターネットオプション を選択して、[接続]タブの[LAN の設定]ボタンをクリックして、[ローカルエリアネットワーク(LAN)の設定]画面で、すべてのチェックを外してください。

始めにテストするブラウザ操作を記録して、その記録からテストを定義したテスト･スイートを作成します。

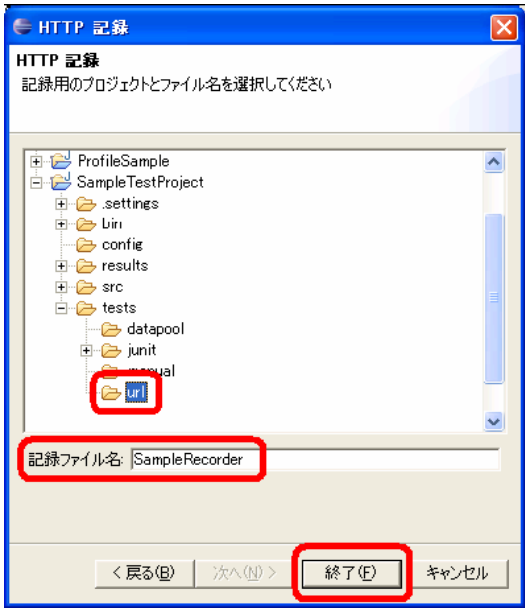
メニューから ファイル | 新規 | その他 を選択します。[新規]画面がで、テスト | 記録 | HTTP 記録 を選択して[次へ]ボタンを押します。



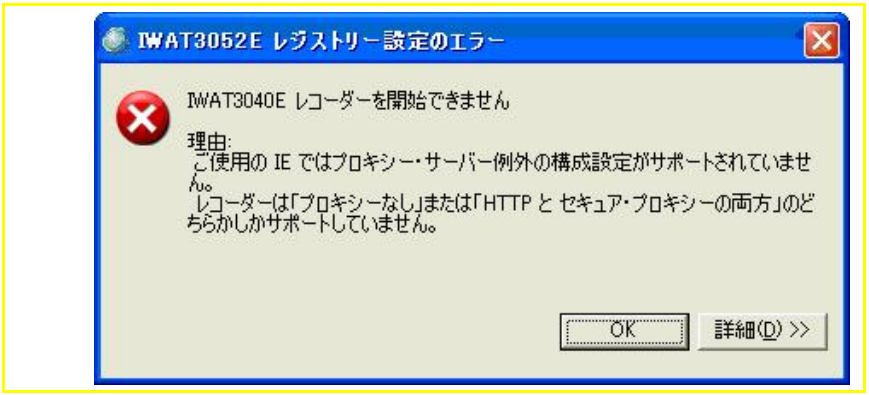
次に HTTP 記録ファイルの保管場所を設定します。以下のように設定して、[終了]ボタンを押します。

保管場所：SampleTestProject/tests/url

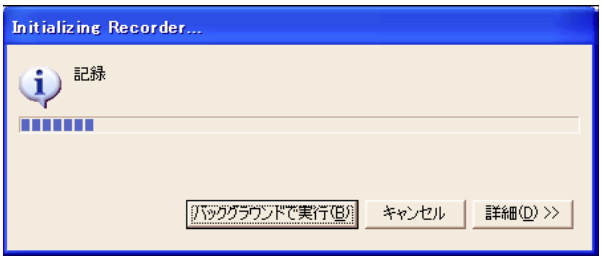
記録ファイル名：SampleRecorder

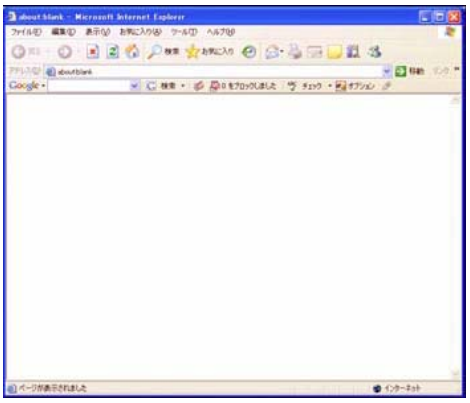


次のようなエラーが発生した場合は、Internet Explorer の設定で、プロキシサーバを使用する設定を、オフにしてください。



記録ファイルの作成が開始され、テスト用のブラウザが表示されます。

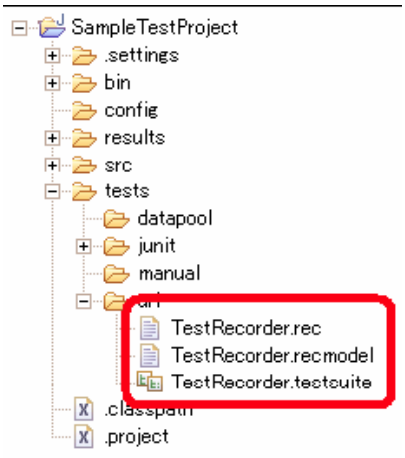




テスト用のブラウザでテストをしたい動作を操作してください。

操作が終了しましたら、ブラウザを閉じる か [レコーダー・コントロール]ビューの停止ボタンを押してください。

以下のファイルが作成されます。



次にテスト・スイート・ファイル SampleRecorder.testsuites の編集を行います。

SampleRecorder.testsuites をダブル・クリックして、テスト・スイート・エディターを表示します。

[概要]タブでは、テスト・ソース生成先のパッケージの指定と、実行でエミュレートするユーザー数の設定を行います。

以下のように設定を行ってください。

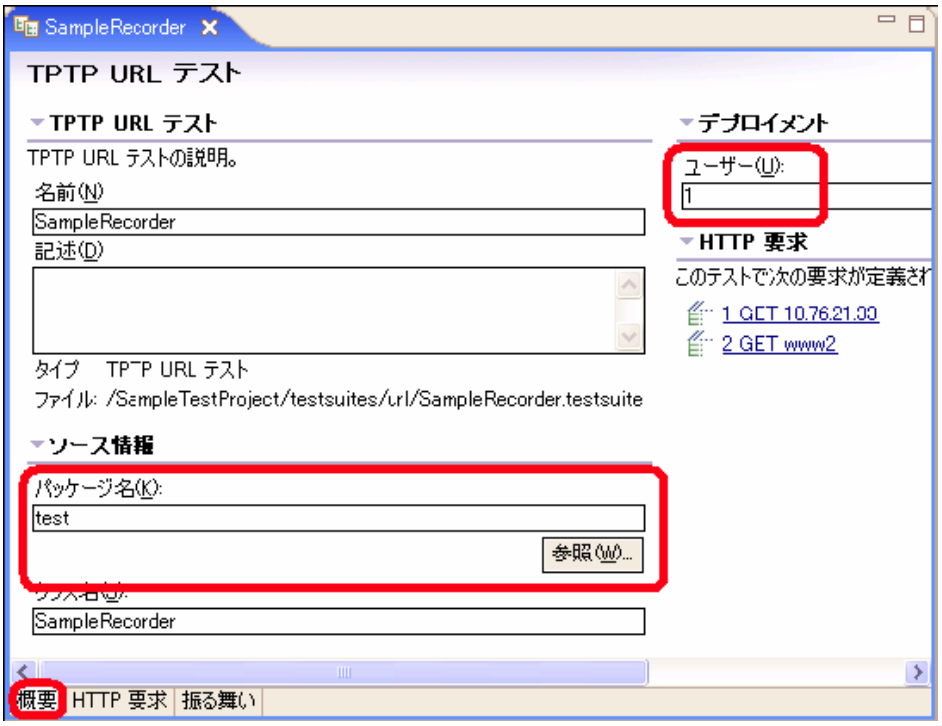
・ソース情報

パッケージ：test_sample(SampleTestProject/src)

・デプロイメント

ユーザー：3

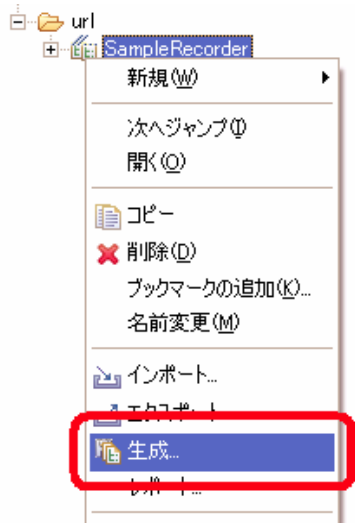
[HTTP 要求][振る舞い]タブは、今回は特に編集を行いません。



テスト・スイートの編集が終わりましたら、次はテスト・ソース・コードを生成します。

メニューから ウィンドウ | パースペクティブを開く | テストを選択して、パースペクティブを切り替えます。

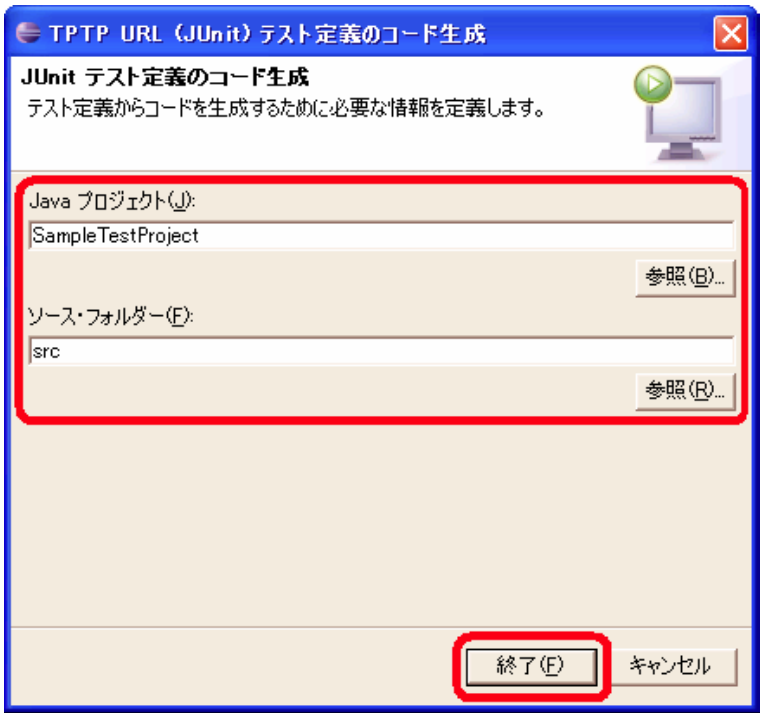
[テスト・ナビゲーター]ビューで対象テスト・スイートを右クリックして、ポップアップメニューから生成を選択します。



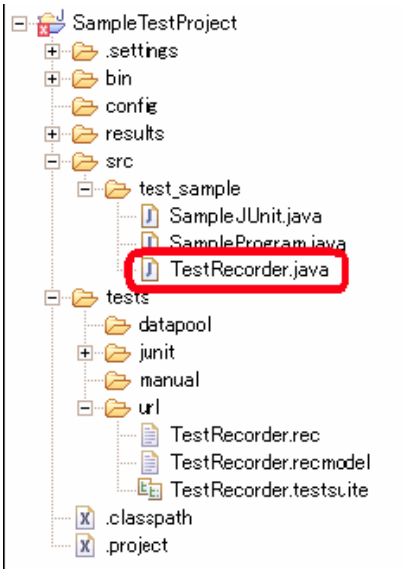
[TPTP URL(JUnit) テスト定義のコード生成]画面が表示で、以下のように設定されていることを確認して[終了]ボタンを押します。

Java プロジェクト：SampleTestProject

ソース・フォルダー：src



設定した場所にテスト・ソース・ファイル SampleRecorder.java が生成されます。



このときプロジェクトがエラーとなります。

これはテスト・ソース生成の際に自動的に追加されたビルド・パスに対象ファイルがないものがあるためです。

プロジェクトを右クリックして、ポップアップメニューからプロパティを選択して、Java のビルド・パス | ライブラリーで警告の出ているビルド・パスを削除してください。

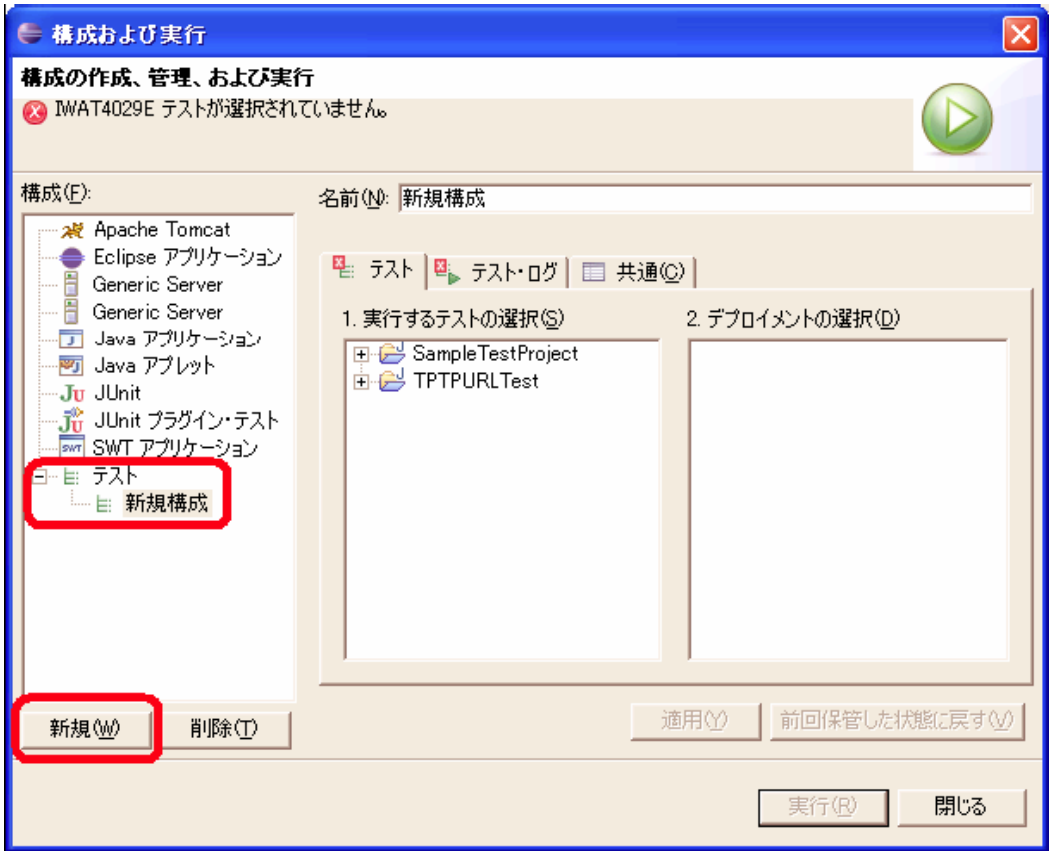
テスト・ソースが生成されビルドが完了しましたら、テストの実行をします。

メニューから実行 | 構成および実行 を選択します。

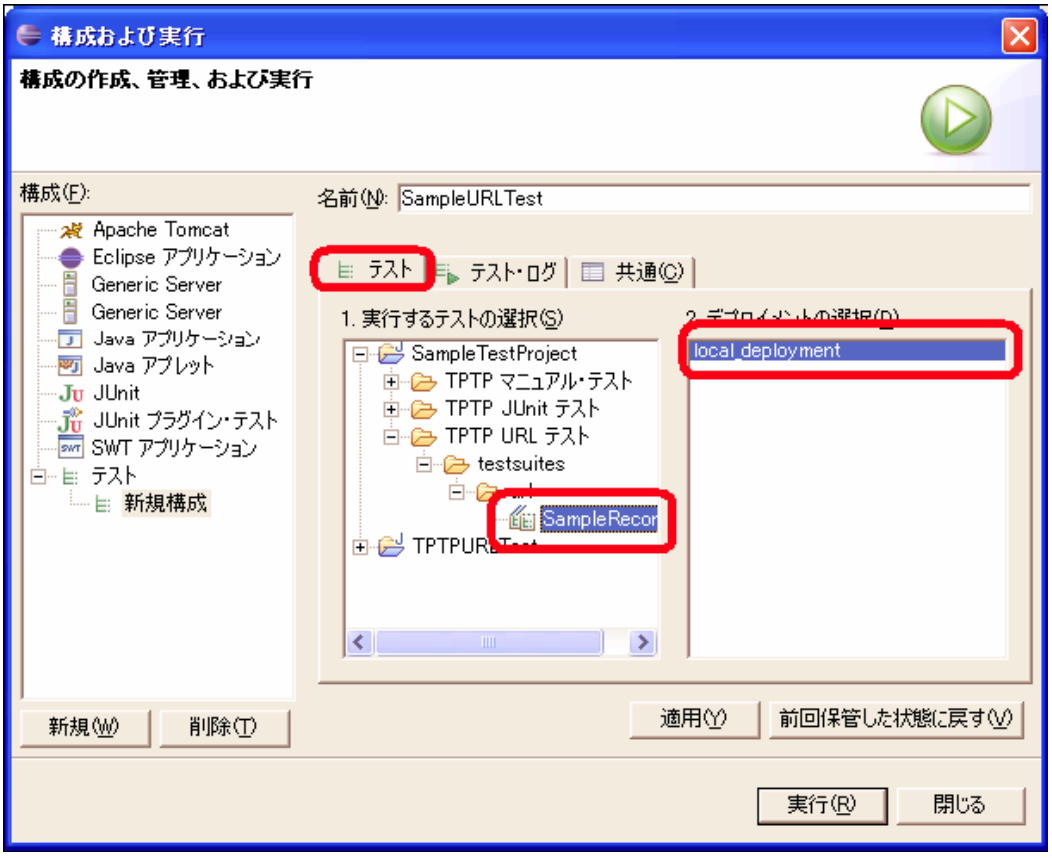
[構成および実行]画面が表示されます。

構成のテストを選択して[新規]ボタンを押して、新規構成を作成します。

名前を SampleURLTest と設定します。



[ホスト]タブで、実行するテストを選択します。実行するテスト・ソースを選択すると、デプロイメントの選択に local_deployment が追加され、自動的に選択されます。



次に[テスト・ログ]タブで、テスト結果ファイルの保管場所を設定します。

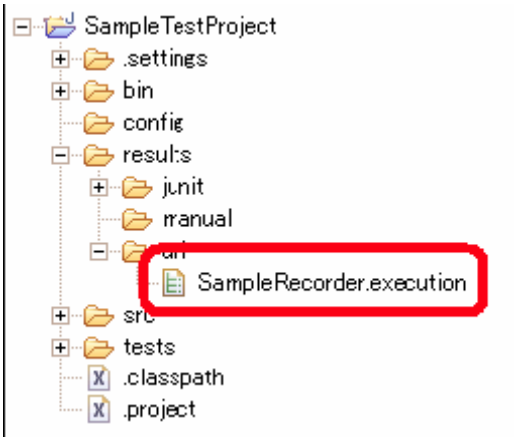
デフォルトを使用からチェックを外して、以下のように設定します。

名前：SampleRecorder

ロケーション：SampleTestProject/results/url

設定が完了しましたら、[実行]ボタンを押して、テストを実行します。

テストが終了すると、以下の実行結果ファイルが作成されます。



実行結果ファイル SampleRecorder.execution を表示して結果を確認してください。

SampleURL.execution をダブル・クリックして、エディターに表示してください。

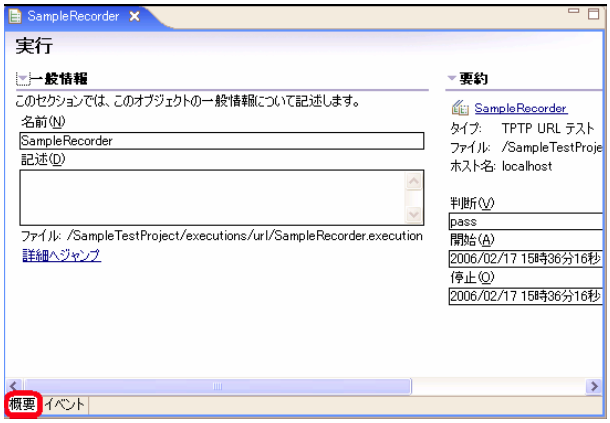
[概要]タブでは、そのテスト結果の概要を確認することができます。

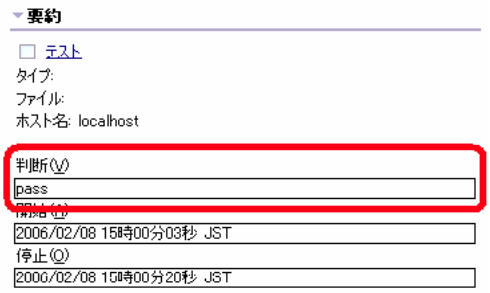
[概要]タブの要約の判断で、テストの成否の判

定を行えます。

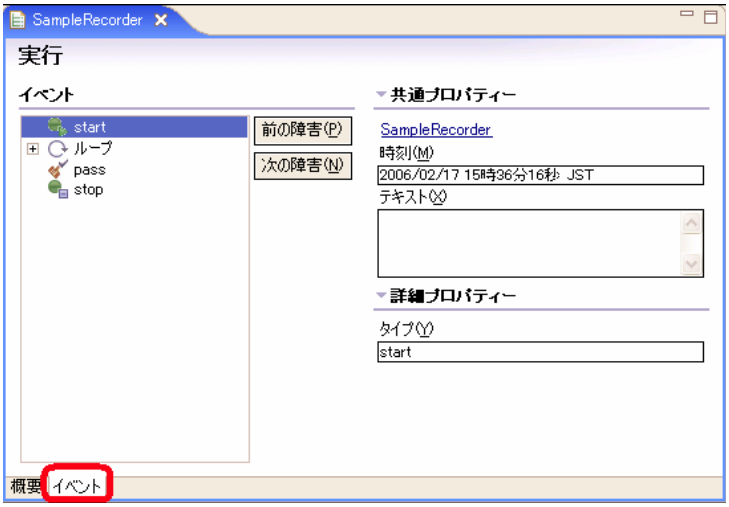
図の例では pass と表示されているため、テスト

は成功したと判定できます。

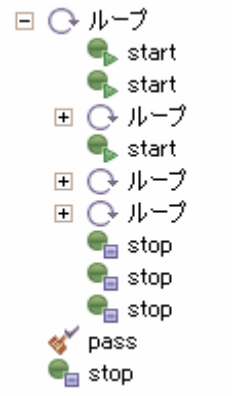




[イベント]タブでは、そのテストの各イベントについて確認することができます。



イベントのペインのツリーを展開すると、実行でエミュレートするユーザー数を 3 と設定したことから、同じループ処理が 3 回行われていることがわかります。



またイベントペイン最後の pass から全テストが成功しているとわかります。

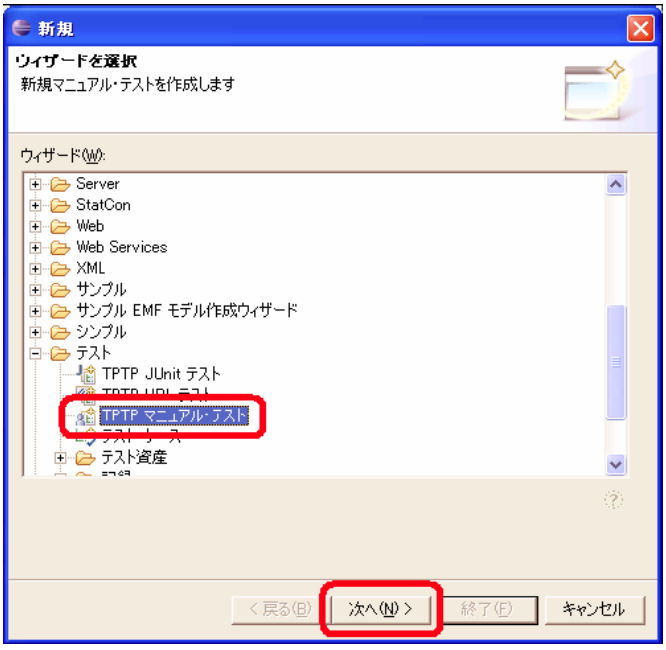
Manual テスト

ここでは Manual テストを行います。

まずテストの内容を定義する Manual テストのテスト・スイートを作成します。

メニューから ファイル | 新規 | その他 を選択します。

[新規]画面で、テスト | TPTP マニュアル・テストを選択して[次へ]ボタンを押します。

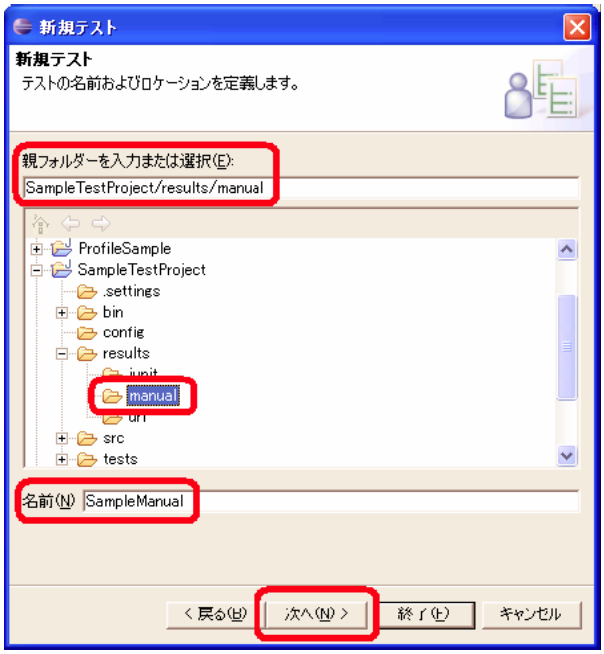


[新規テスト]画面で、テスト・スイートの保管場所を設定します。以下のように設定して[次へ]ボタンを押します。

親フォルダーを入力または選択:SampleTestProject/tests/manual

名前: SampleManual

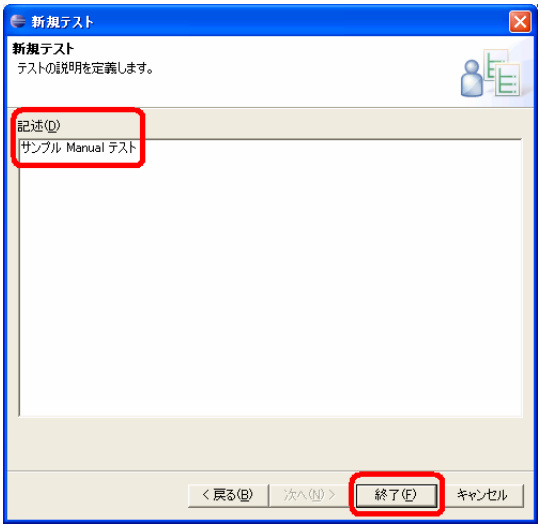
(親フォルダーは直接入力しても、ツリーから選択してもかまいません)



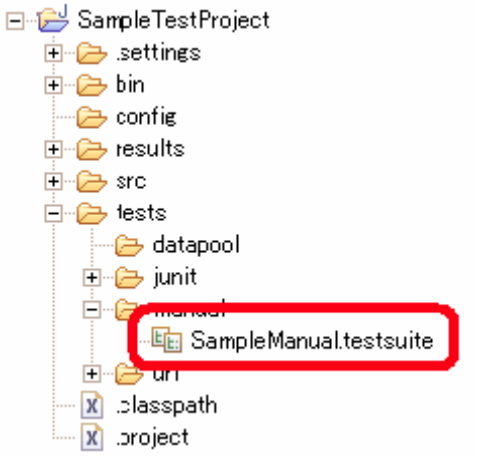
最後にテスト・スイートの記述を設定します。

以下のように入力して[**終了**]ボタンを押します。

記述:サンプル Manual テスト

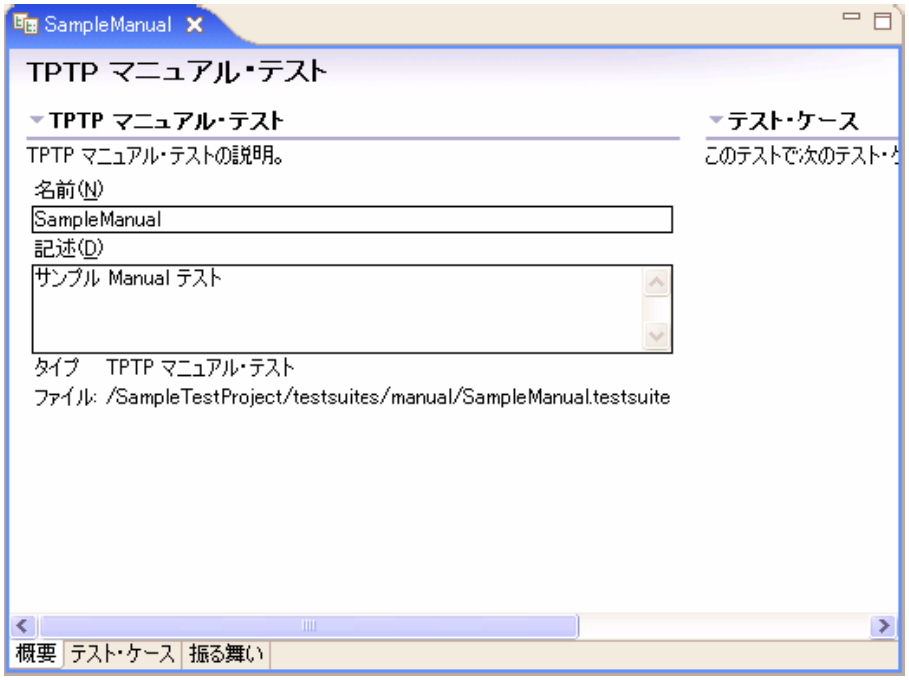


[新規テスト]を終了すると以下のリソースが生成されます。



次に生成したテスト・スイート・ファイル SampleManual.testsuite の編集を行います。

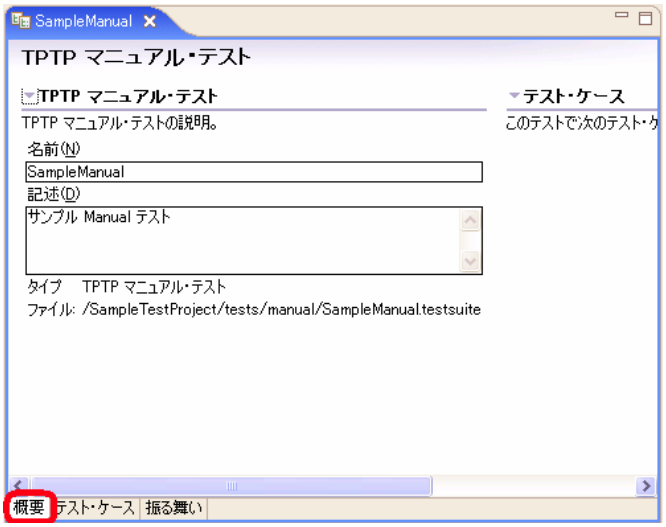
SampleManual.testsuite をダブル・クリックしてテスト・スイート・エディターを表示させてください。



今回は編集により 2 つのテスト・ケースを定義して、その実行順序(振る舞い)を設定します。

[概要]タブでは、そのテスト・スイート・ファイルの概要を確認することができます。

ここでは特に編集は行いません。



[テスト・ケース]タブでは、テスト・ケースの定義・

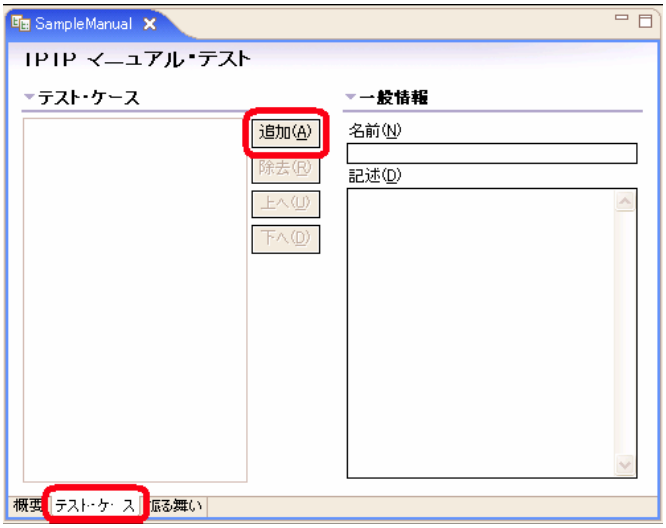
編集が行えます。

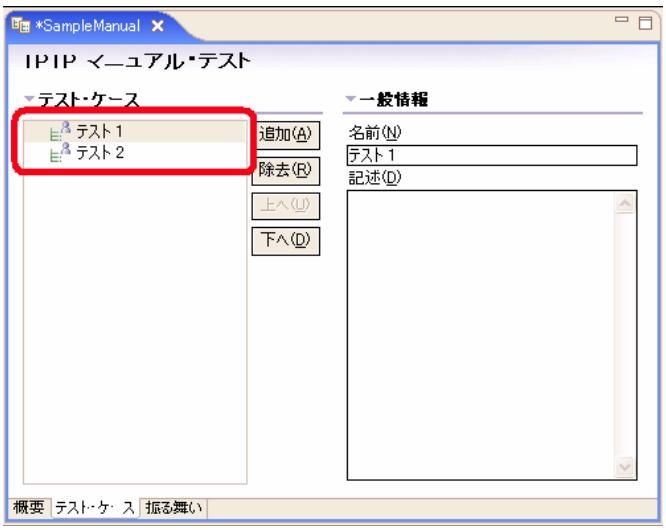
今回は以下の 2 つのテスト・ケースを追加します。

・test1

・test2

[追加]ボタンを押して、テスト・ケースを追加します。





追加されたテスト・ケースを選択して一般情報で以下のように編集してください。

・名前:test1

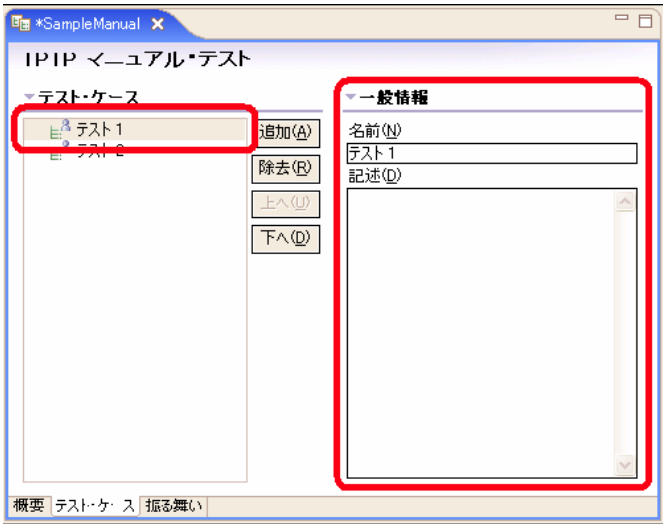
記述:テスト 1

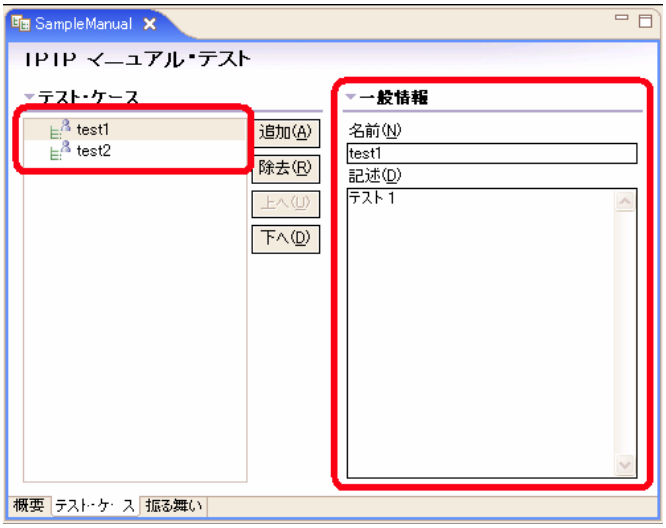
・名前:test2

記述:テスト 2

名前はテスト・ケース名、記述はテストの説明となります。

これで 2 つのテスト・ケースが定義できました



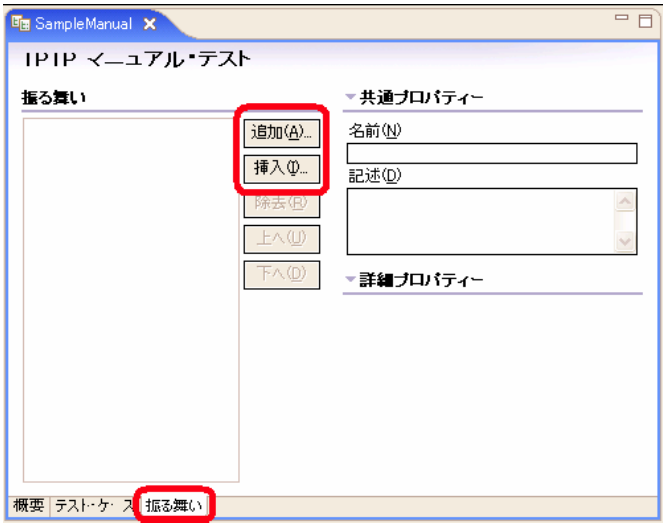


[振る舞い]タブでは、テスト・ケースの実行順序や実行回数の設定を行うことができます。

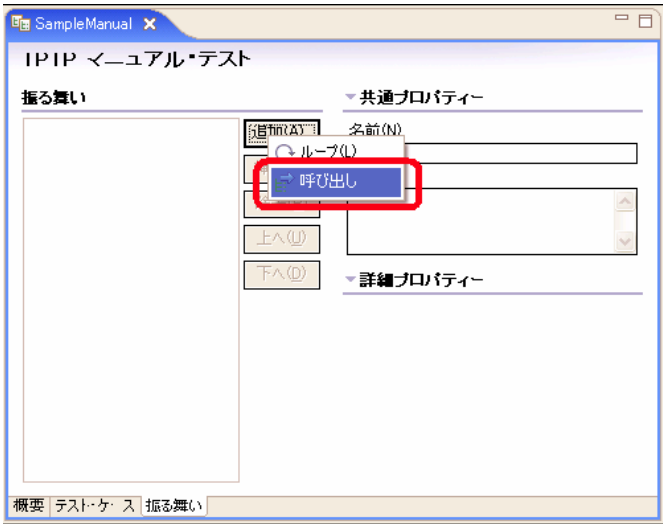
先に定義した 2 つのテスト・ケースの実行順序と実行回数を設定します。

[追加][挿入]ボタンでループや呼び出しを設定して行きます。

今回は以下の手順に従って設定してください。



まず[追加]ボタンを押して、呼び出しを選択します。



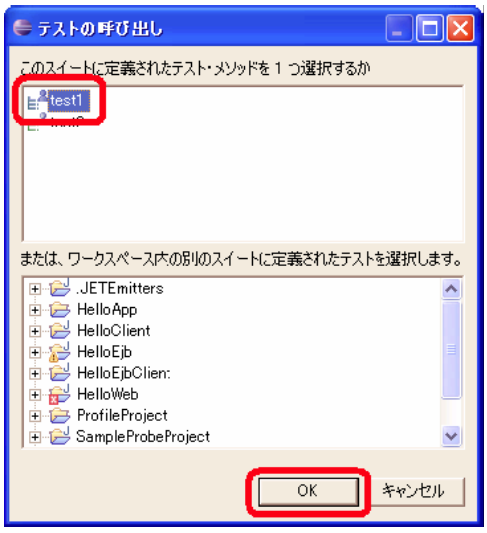
[テストの呼び出し]画面で、テスト・ケースを選択して、[OK]ボタンを押します。

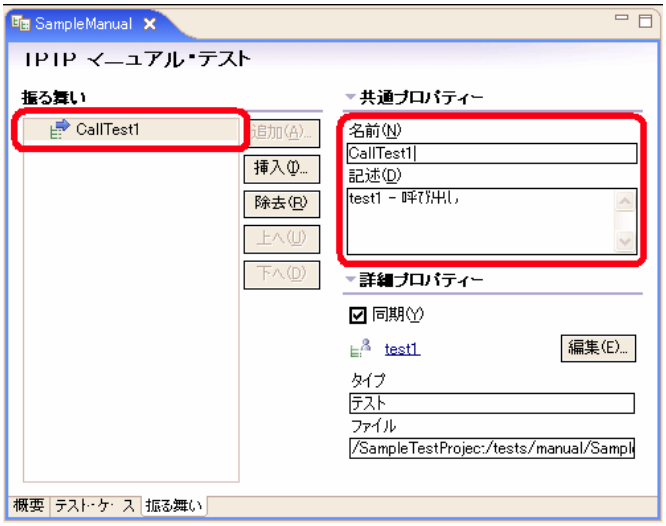
ここでは、テスト・ケース test1 を呼び出します。

振る舞いに呼び出しが追加されます。追加された呼び出しを以下のように設定します。

名前：CallTest1

記述：test1 － 呼び出し





次に[挿入]ボタンを押し、ループを選択します。

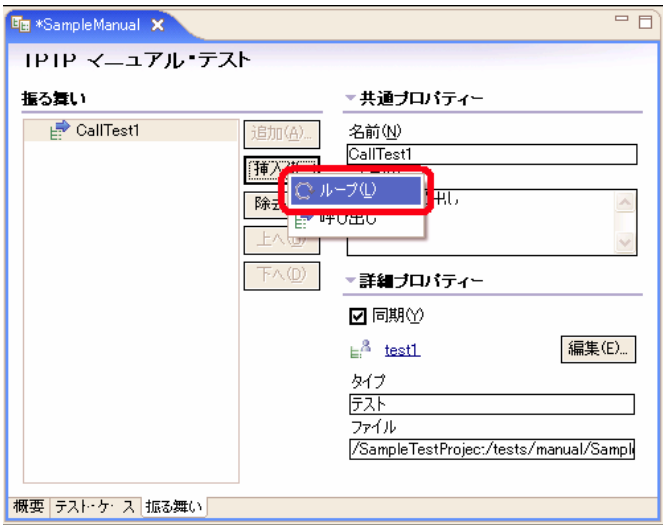
振る舞いペインの CallTest1 と同じ階層にループが追加されます。

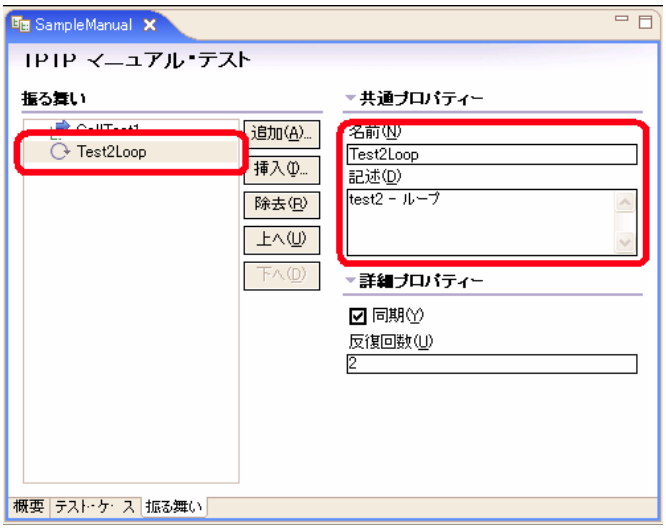
追加されたループを以下のように設定します。

名前: Test2Loop

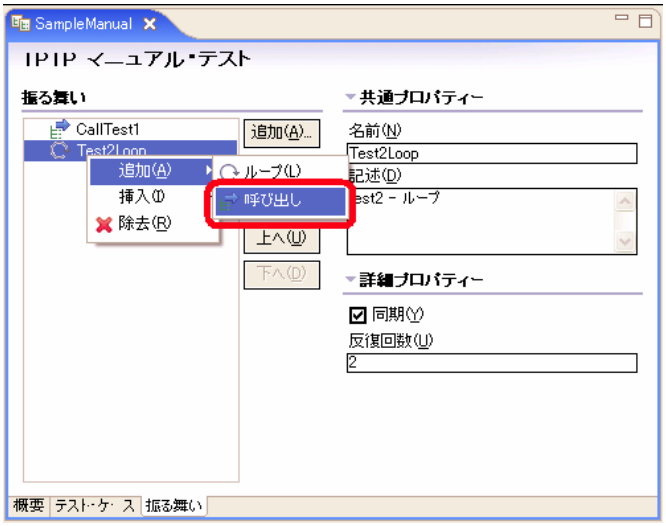
記述: test2 - ループ

反復回数:2





振る舞い上のループを右クリックして、ポップアップメニューから 追加 | 呼び出し を選択します。

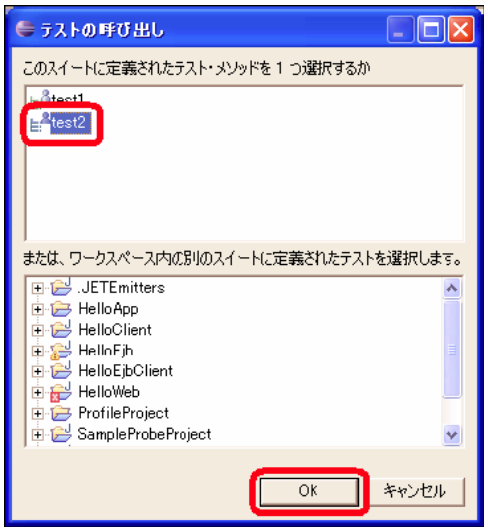


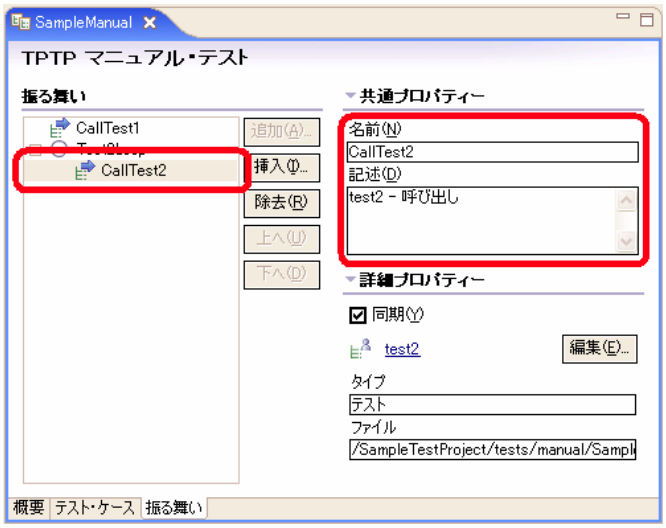
先ほどと同様に[テストの呼び出し]画面で、テスト・ケースを選択して、[OK]ボタンを押します。

ここでは、テスト・ケース test2 を呼び出します。振る舞いのループ下の階層に呼び出しが追加されるので、追加された呼び出しを以下のように設定します。

名前：CallTest2

記述：test2 - 呼び出し





テスト・スイートの編集は以上で終わります。ファイル｜保管 を行ってください。

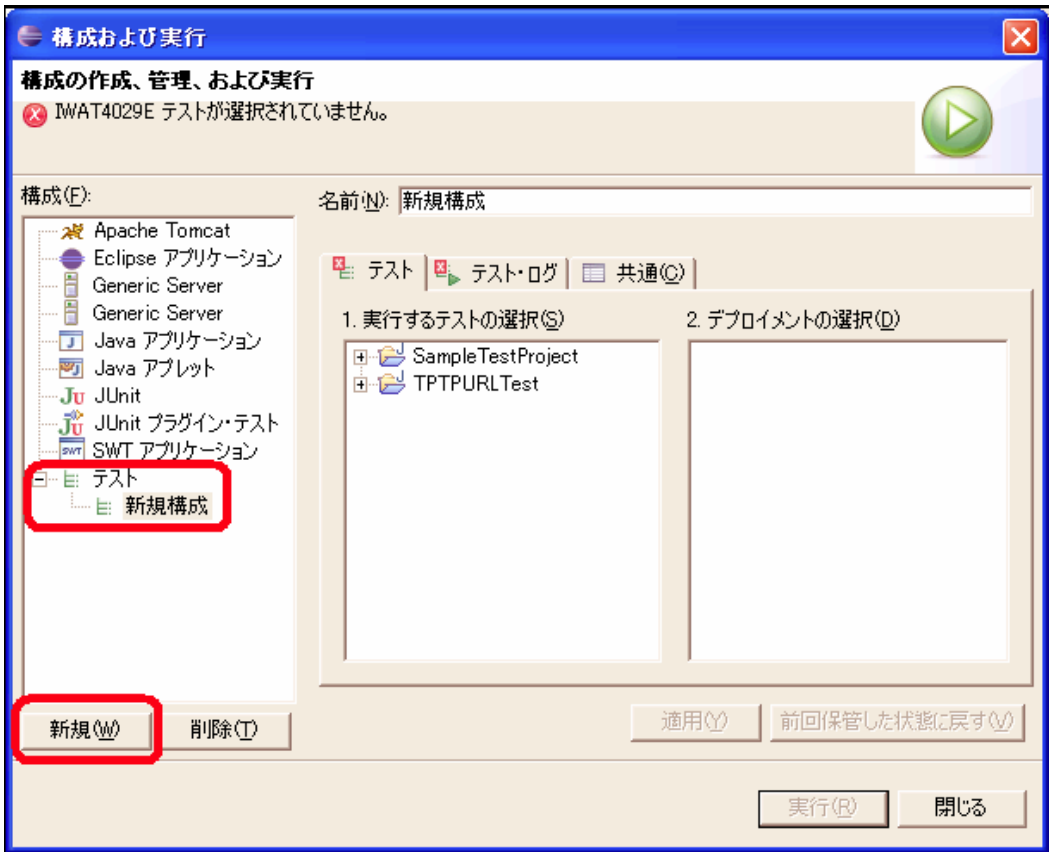
次にテストを実行します。

メニューから実行 | 構成および実行 を選択します。

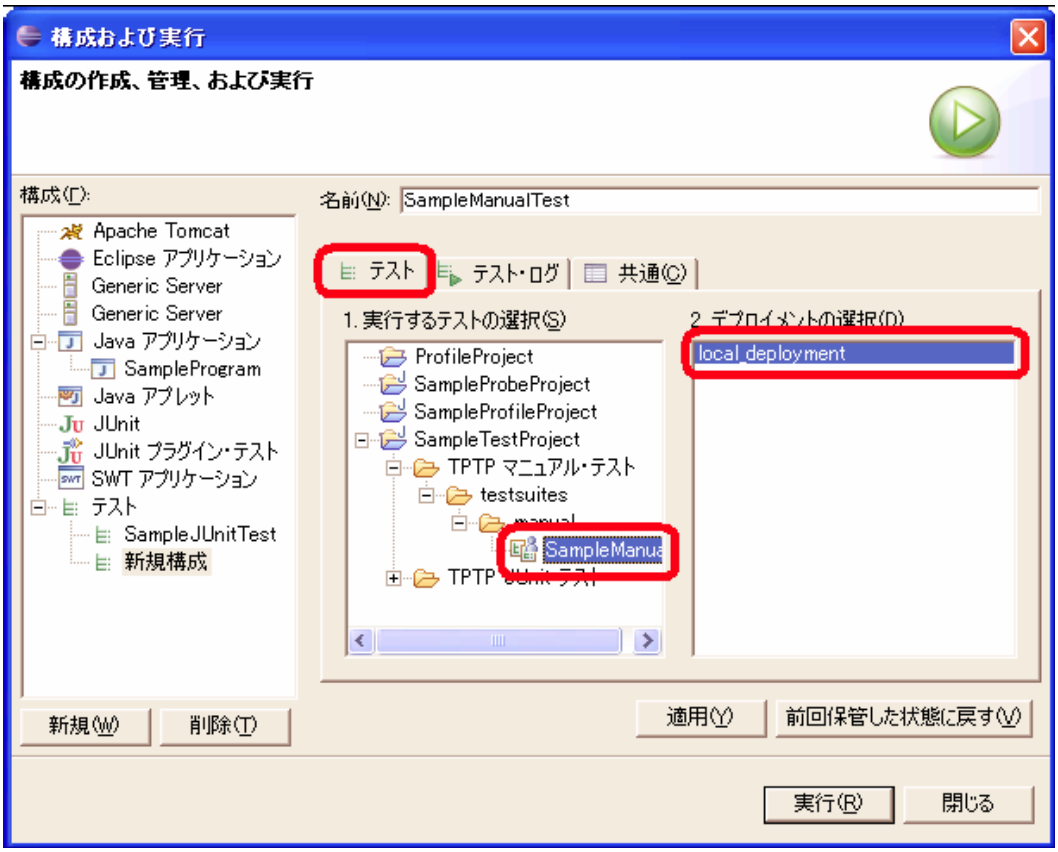
[構成および実行]画面が表示されます。

構成のテストを選択して[新規]ボタンを押して、新規構成を作成します。

名前を SampleManualTest と設定します。



[ホスト]タブで、実行するテストを選択します。実行するテスト・ソースを選択すると、デプロイメントの選択に local_deployment が追加され、自動的に選択されます。



次に[テスト・ログ]タブで、テスト結果ファイルの保管場所を設定します。

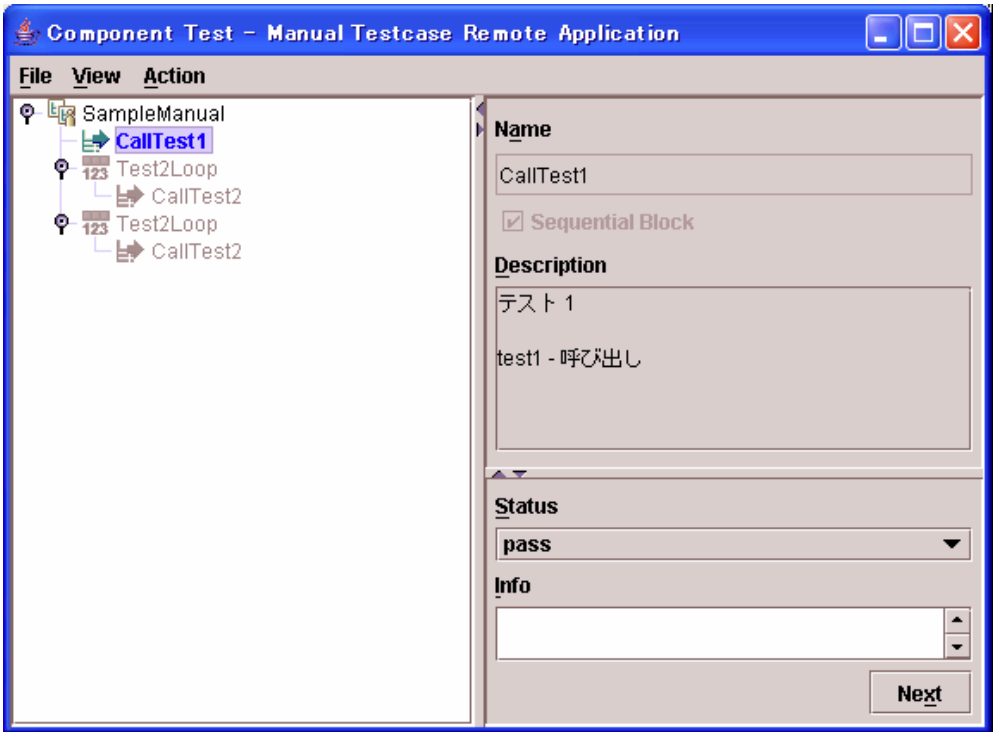
デフォルトを使用からチェックを外して、以下のように設定します。

名前：SampleManual

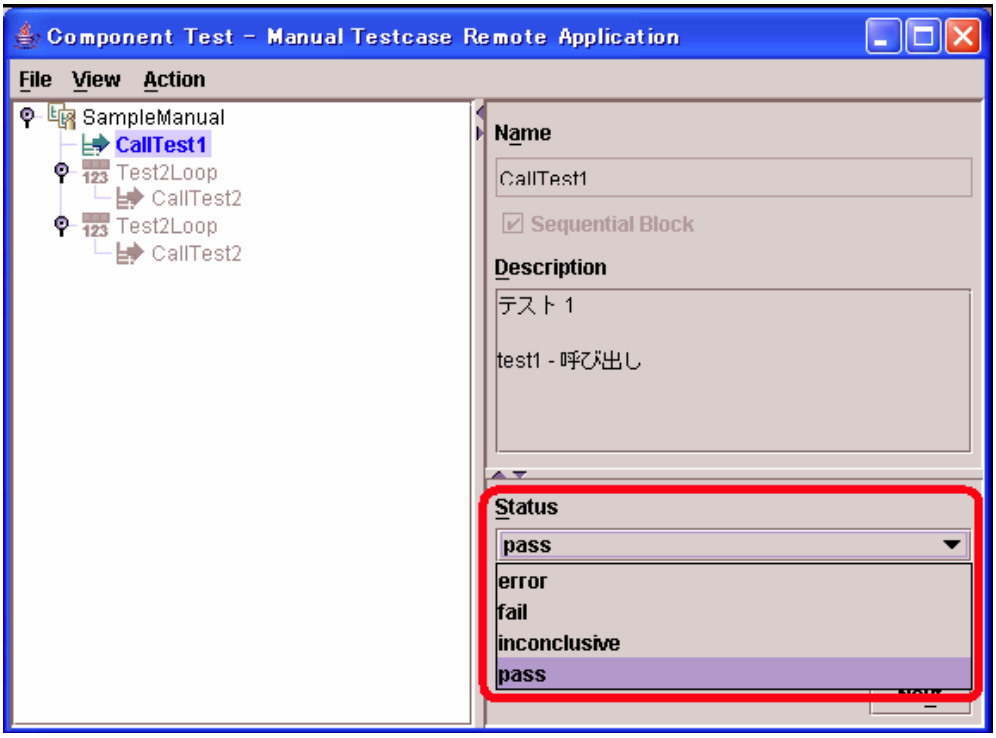
ロケーション：SampleTestProject/results/manual

設定が完了しましたら、[実行]ボタンを押して、テストを実行します。

テストを実行すると[Compornent Test]画面が表示されます。



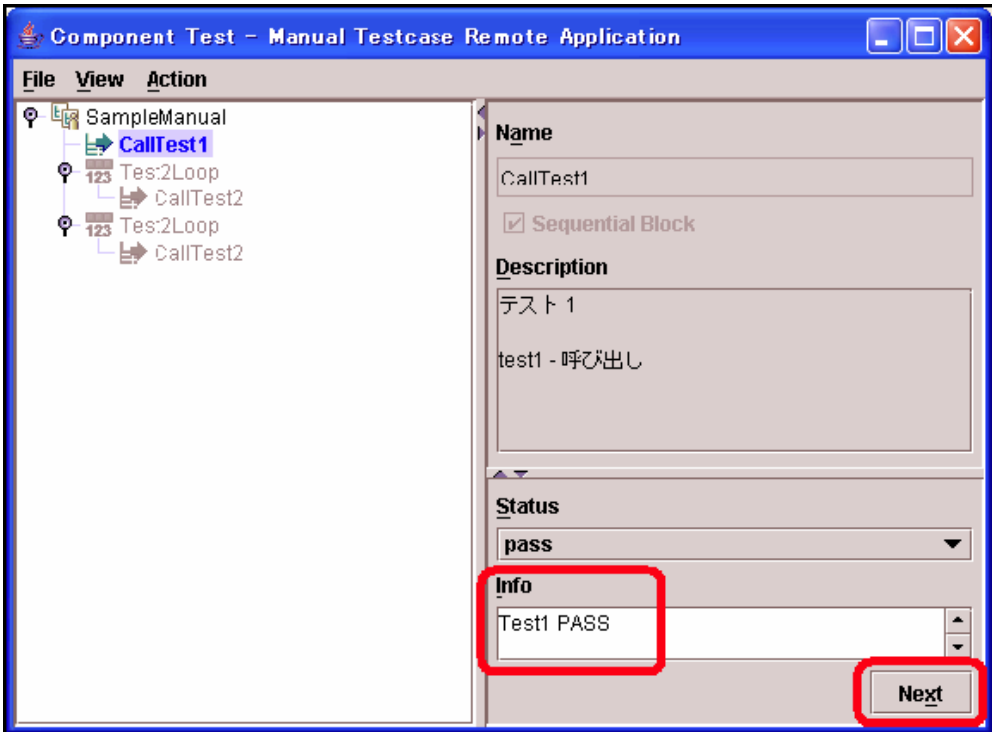
記述にあるテスト内容を実行して、結果を Status に設定していきます。



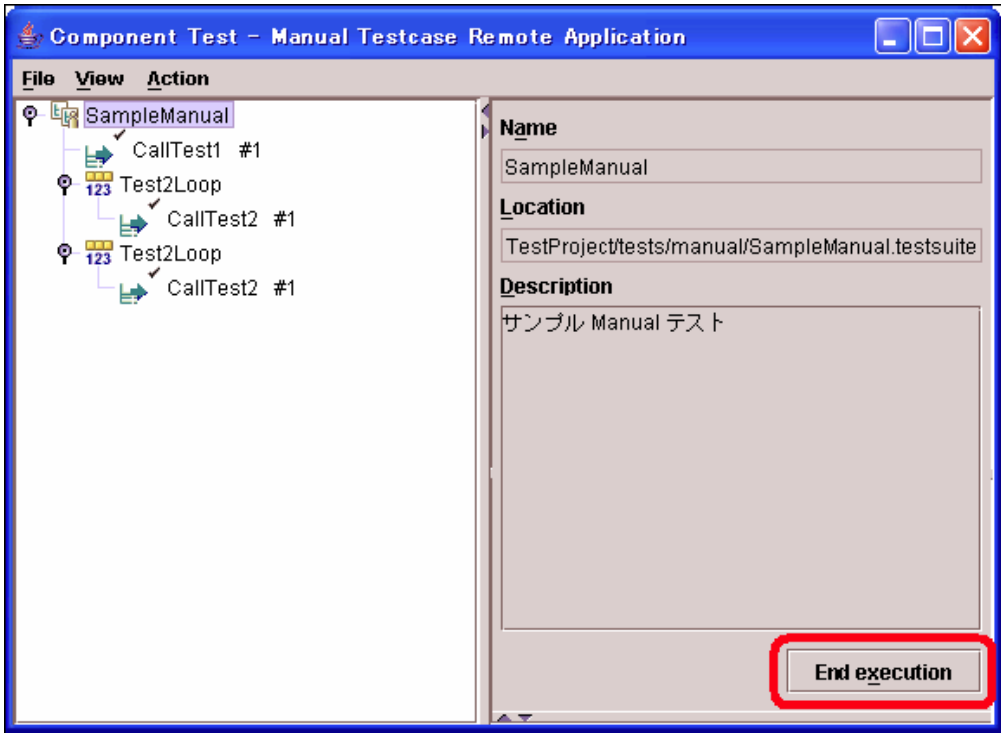
また Info にそのテストのメッセージを設定することができます。

テストが完了したら、[Next]ボタンを押して次のテストへ進みます。

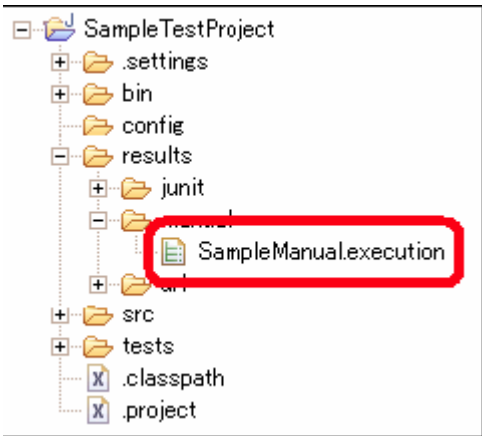
今回テストの一つで Status に fail を設定してください。



すべてのテストが完了したら、[End execution]ボタンを押してテストを終了します。



テストが終了すると、以下の実行結果ファイルが作成されます。



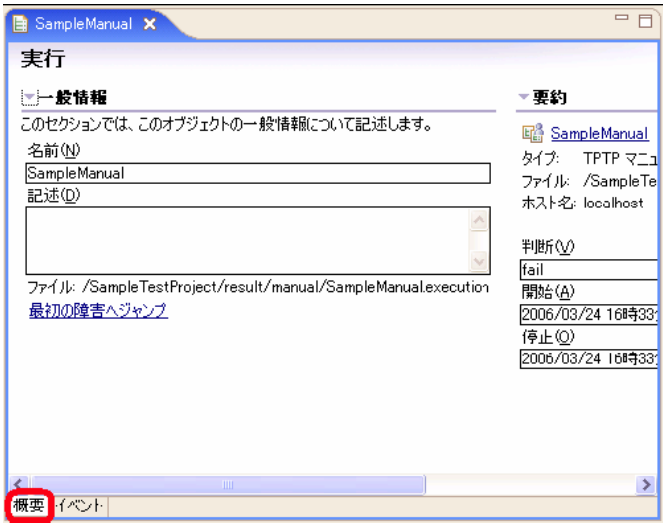
実行結果ファイル SampleRecorder.execution を表示して結果を確認してください。

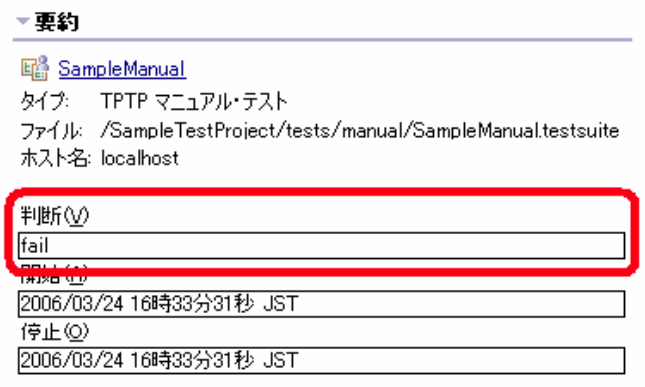
SampleRecorder.execution をダブル・クリックして、エディターに表示してください。

[概要]タブでは、そのテスト結果の概要を確認することができます。

[概要]タブの要約の判断で、テストの成否の判定を行えます。

今回は fail と表示されているため、テストは失敗したと判定できます。



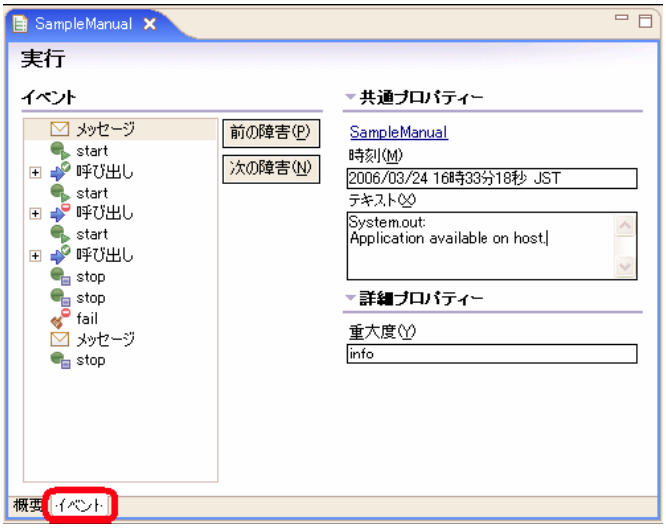


[イベント]タブでは、そのテストの各イベントについて確認することができます。

イベントのペインのツリーを展開すると、呼び出して行ったテストが 3 回あったと確認できます。

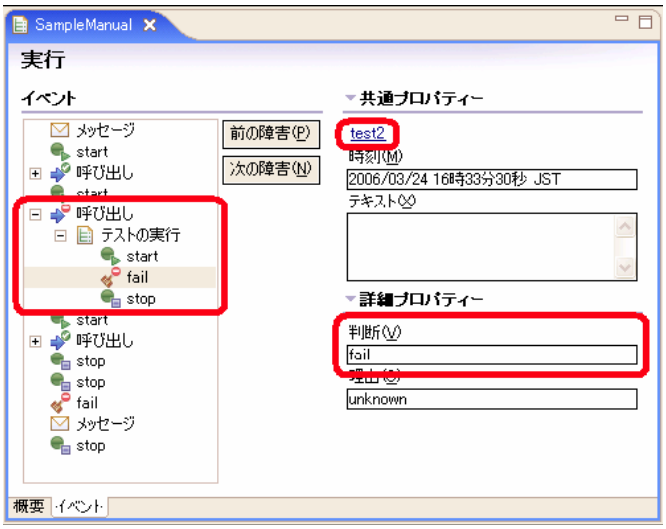
またイベントペイン最後のからテストが失敗しているとわかります。

（テストは、どれか一つのテスト結果が fail であった場合、そのテスト全体が失敗と判断されます。）



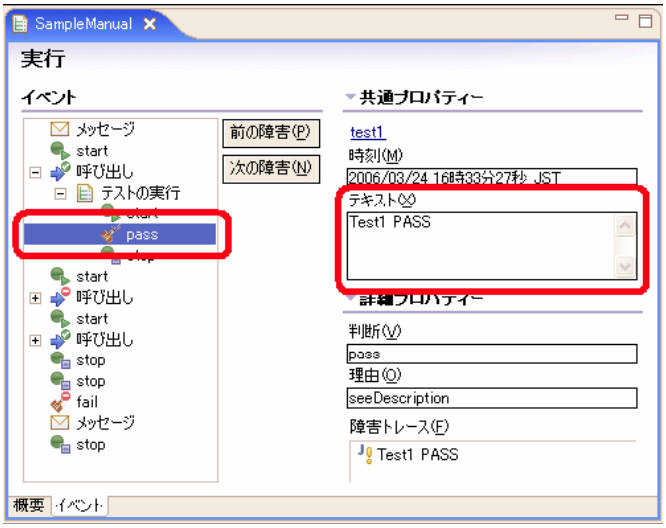
2 回目の呼び出しで test2 のテストを行っていることがわかります。

また判断で pass とあることや、イベントペインのテスト実行の fail からテスト test2 は失敗しているとわかります。



また[Compornent Test]画面の Info で記述した内容は、テスト結果のテキストに表示されます。

ここでは日本語の文字化けは起こりません。



データプールの使用

データプールはテストデータの集合を表現するもので、データを表形式で記述します。

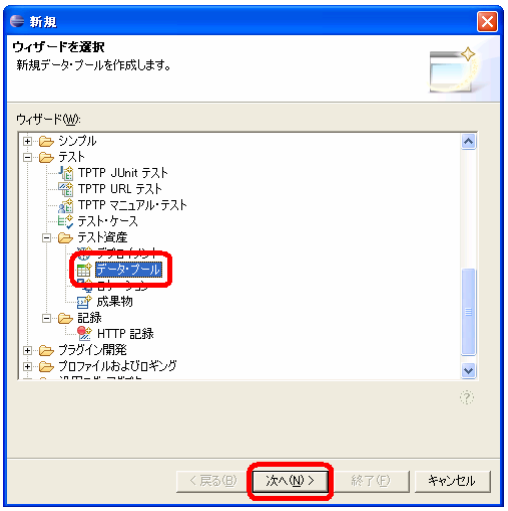
ここでは「3.8.1.JUnit テスト」で行った JUnit テストを参照してデータプールを使用したテストを行います。

先に「3.8.1.JUnit テスト」を行っておいてください。

データプールを作成します。

メニューから **ファイル | 新規 | その他** を選択します。

[**新規**]画面で、**テスト | テスト資産 | データ・プール** を選択して[**次へ**]ボタンを押します。



[**新規データ・プール**]画面で、データプールの保管場所を設定します。

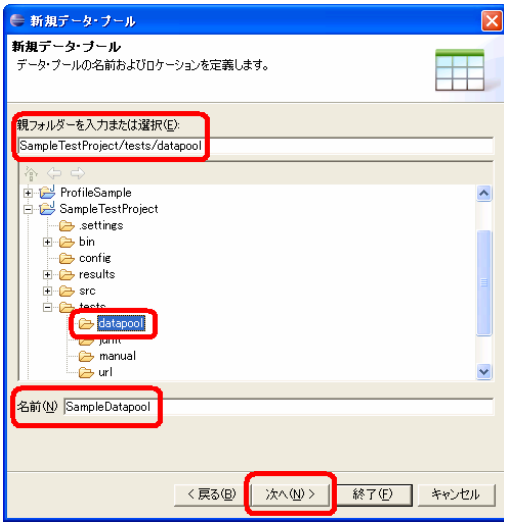
以下のように設定して[**次へ**]ボタンを押します。

親フォルダーを入力または選択:

SampleTestProject/tests/datapool

名前: **SampleDatapool**

(親フォルダは直接入力しても、ツリーから選択してもかまいません)



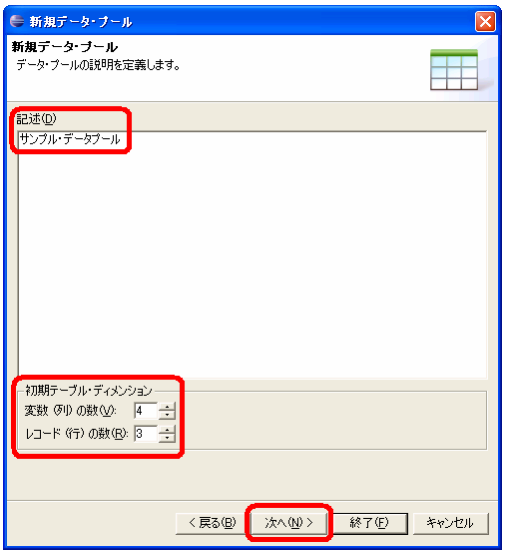
次にテスト・スイートの記述と初期のデータ・テーブルの設定をします。

以下のように入力して[**次へ**]ボタンを押します。

記述:**サンプル・データプール**

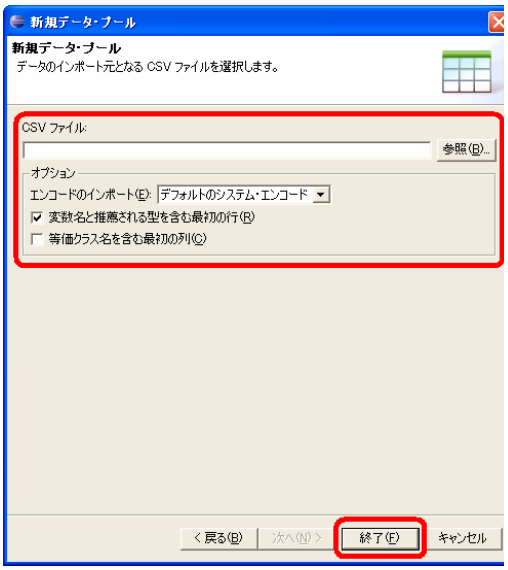
変数:**4**

レコード:**3**

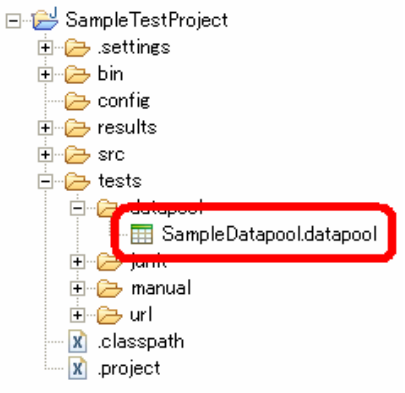


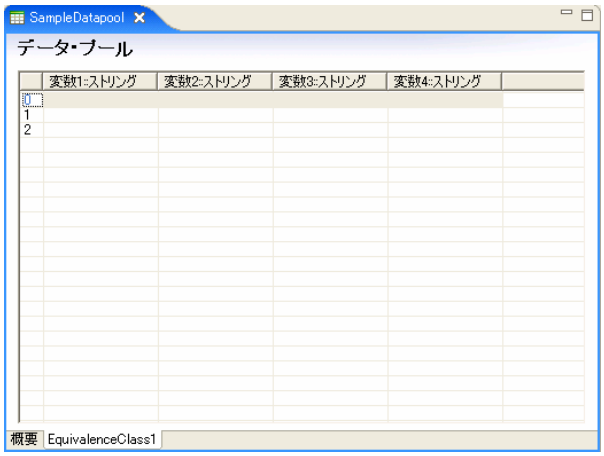
最後に CSV ファイルのインポートの設定をします。データプールは Excel で作成した CSV ファイルをインポートして作成することができます。

今回は特に設定をしませんので、[**終了**]ボタンを押して終了します。



以上でデータプール **SampleDatapool.datapool** が作成されます





次にデータプールの編集を行います。

テストデータの設定は、[**等価クラス**]タブ(右図では[EquivalenceClass1]タブ)で行います。

データプールで変数をクリックして、[**変数の編集**]画面で変数の編集を行います。

ここでは以下のように設定します。

設定を完了すると[OK]ボタンを押します。

・名前：**set**

タイプ：**String**

・名前：**make**

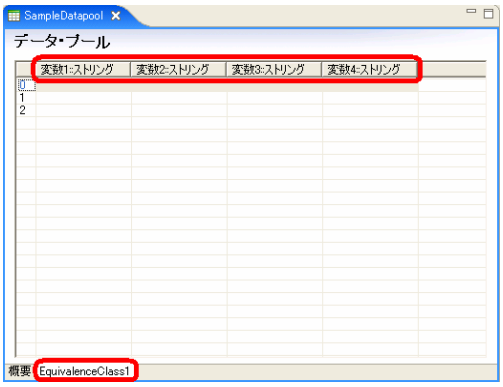
タイプ：**String**

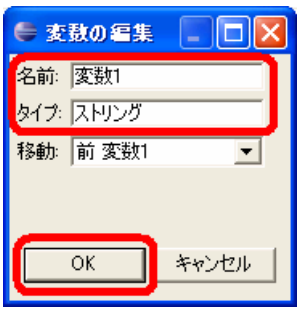
・名前：**result**

タイプ：**String**

・名前：**prefix**

タイプ：**String**

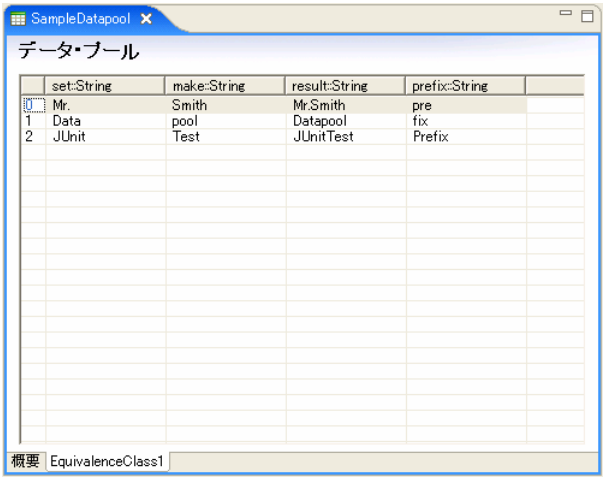




次に各セルに値を設定します。対象セルをクリックすることで入力が行えます。

以下のように設定してください。

	set::String	make::String	result::String	preix::String
0	Mr.	Smith	Mt.Smith	pre
1	Data	pool	Datapool	fix
2	JUnit	Test	JUnitTest	Prefix



ファイル | 保管 を行い、編集を完了します。

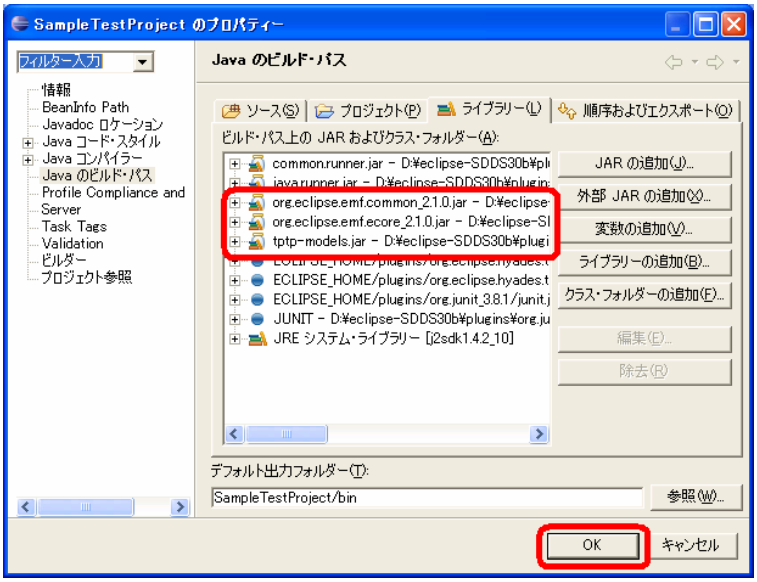
次にテスト・ソースの実装を行います。

テスト実装の前に以下のビルド・パスをプロジェクトに追加してください。

- **ECLIPSE_HOME/plugins/org.eclipse.tptp.platform.models_4.0.0/tptp-models.jar**
- **ECLIPSE_HOME/plugins/org.eclipse.emf.ecore_2.1.0.jar**
- **ECLIPSE_HOME/plugins/org.eclipse.emf.common_2.1.0.jar**

ビルド・パスの追加はプロジェクトを右クリックして、ポップアップメニューから**プロパティー**を選択、「***のプロパティー」画面で **Java のビルド・パス | ライブラリー | [外部 JAR の追加]ボタン** で行ってください。

ビルド・パスの追加が完了したら、[OK]ボタンを押します。



テスト・クラス・ファイル **SampleJUnit.java** を次のように実装します。

<プログラム SampleJUnit.java>

```
（省略）
import java.io.File;

（省略）
import org.eclipse.hyades.execution.runtime.datapool.IDatapool;
import org.eclipse.hyades.execution.runtime.datapool.IDatapoolFactory;
import org.eclipse.hyades.execution.runtime.datapool.IDatapoolIterator;
```

```
import org.eclipse.hyades.models.common.datapool.impl.Common_DatapoolFactoryImpl;

（省略）
public class SampleJUnit extends HyadesTestCase {
   public final static String DATAPOOL_PATH = "データ・プール・ファイルのパス";
   public final static String ITERATOR_CLASS_NAME =
      "org.eclipse.hyades.datapool.iterator.DatapoolIteratorSequentialPrivate";
   private IDatapoolFactory dpFactory = null;
   private IDatapoolIterator iter = null;

   （省略）
   protected void setUp() throws Exception {
      dpFactory = new Common_DatapoolFactoryImpl();
      IDatapool datapool = dpFactory.load(new File(DATAPOOL_PATH), false);
      iter = dpFactory.open(datapool, ITERATOR_CLASS_NAME);
      iter.dpInitialize(datapool, -1);
   }

   （省略）
   public void testMakeString() throws Exception{
      StringMaker obj = new StringMaker();
      while (!iter.dpDone()) {
         obj.setPrefix(iter.dpCurrent().getCell("set").getStringValue());
         String result = obj.makeString(iter.dpCurrent().getCell("make").getStringValue());
         String expectation = iter.dpCurrent().getCell("result").getStringValue();
         System.out.println(result);
         assertEquals(expectation, result);
         iter.dpNext();
      }
   }

   （省略）
   public void testSetGetPrefix() throws Exception{
      StringMaker obj = new StringMaker();
      while (!iter.dpDone()) {
         obj.setPrefix(iter.dpCurrent().getCell("prefix").getStringValue());
         String result = obj.getPrefix();
         String expectation = iter.dpCurrent().getCell("prefix").getStringValue();
         System.out.println(result);
         assertEquals(expectation, result);
         iter.dpNext();
      }
   }
}
```

プログラム中の**赤太字**はデータプール使用のための定型的な記述で、**黒太字**はテストのためにユーザーが設定する記述です。

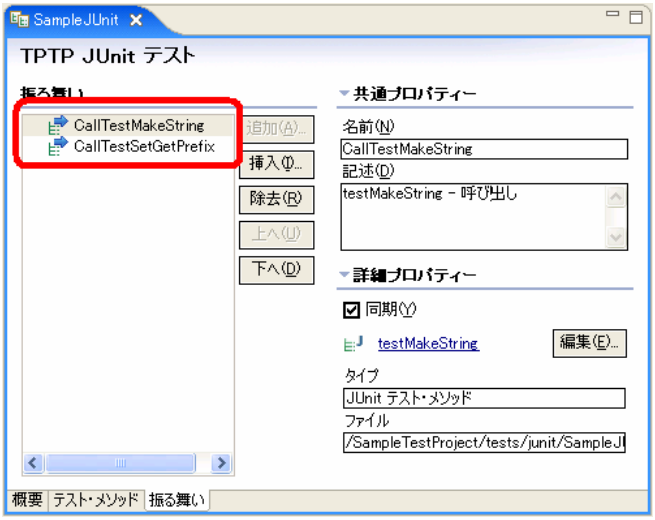
またデータ・プール・ファイルのパスは次のように**完全修飾パス**で設定します。

「C:/eclipse/workspace/SampleTestProject/tests/datapool/SampleDatapool.datapool」

またデータプール使用確認のために、テスト・スイート・ファイル SampleJUnit.testsuite の**振る舞い**を次のように設定してください。

・テスト・メソッド testMakeString()と testSetGetPrefix()を一度ずつ呼び出す。

テスト・スイート・ファイルの[振る舞い]タブの編集については、「3.8.1.JUnit テスト」を参照してください。



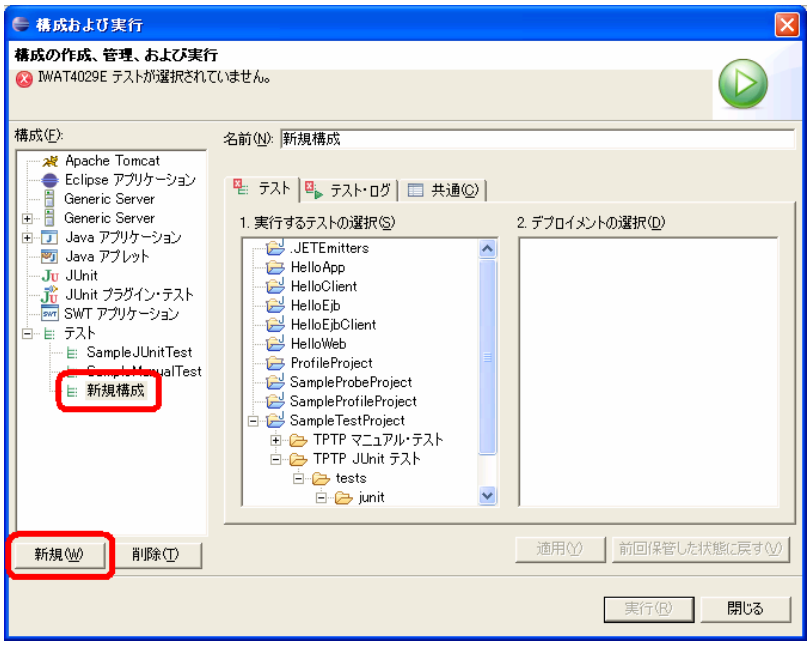
テストを実行します。

メニューの**実行｜構成および実行** を選択します。

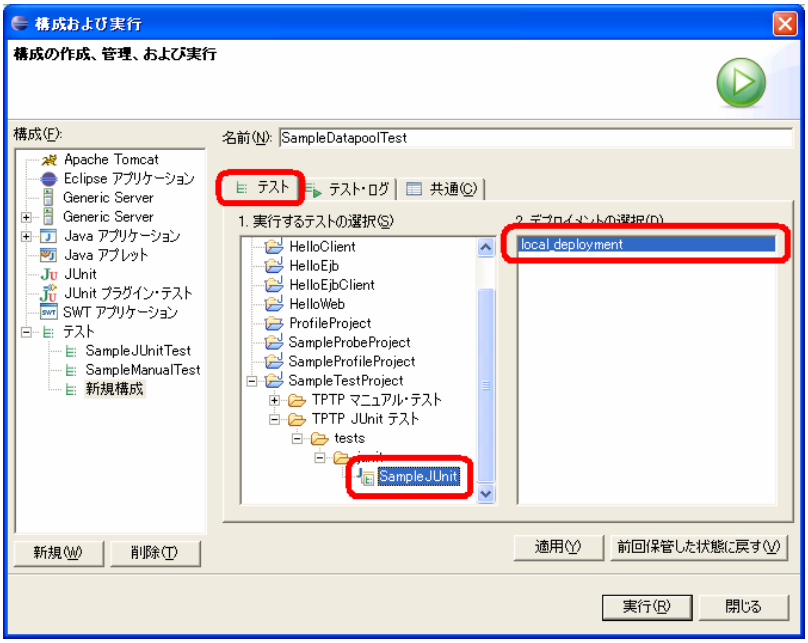
[**構成および実行**]画面が表示されます。

構成の**テスト**を選択して[**新規**]ボタンを押して、新規構成を作成します。

名前を **SampleDatapoolTest** と設定します。



[**テスト**]タブで、実行するテスト（SampleTestProject/TPTP JUnit テスト/tests/junit/SampleJUnit）とデプロイメント（local_deployment）を選択します。



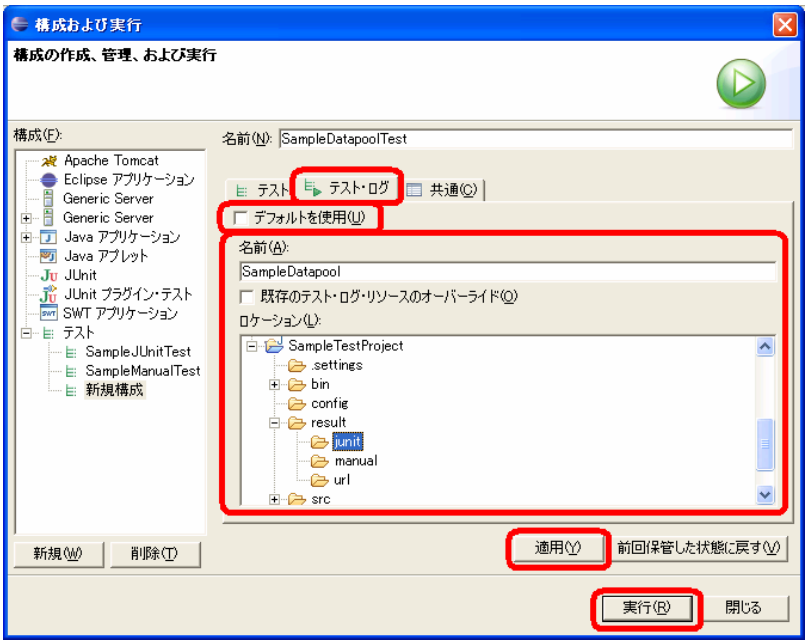
次に[**テスト・ログ**]タブで、テスト結果ファイルの保管場所を設定します。

デフォルトを使用からチェックを外して、以下のように設定します。

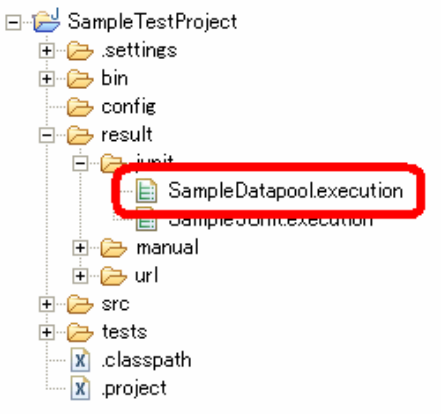
名前：**SampleDatapool**

ロケーション：**SampleTestProject/results/junit**

設定が完了しましたら、[**適用**]ボタンを押して実行構成を保管して、[**実行**]ボタンを押してテストを実行します。



テストが終了すると、以下の実行結果ファイルが作成されます。



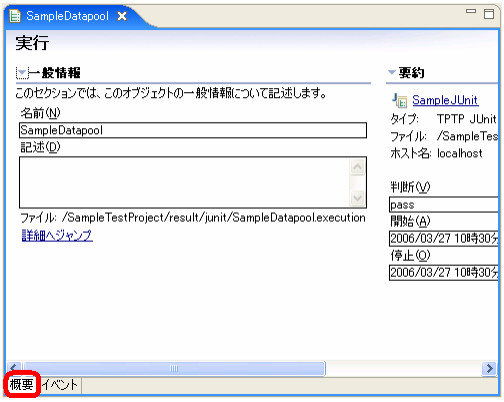
次に実行結果の確認を行います。

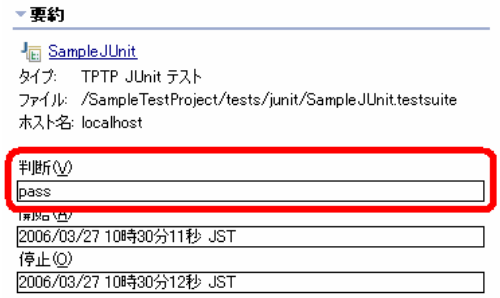
実行結果ファイル **SampleDatapool.execution** をダブル・クリックして、エディターに表示してください。

[**概要**]タブでは、そのテスト結果の概要を確認することができます。

[概要]タブの**要約**の**判断**で、テストの成否の判定を行えます。

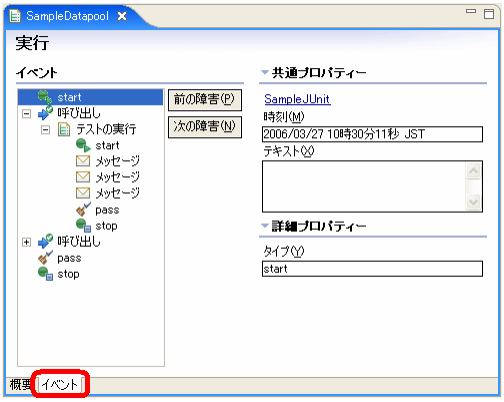
今回は **pass** と表示されているため、テストは成功したと判定できます。





[**イベント**]タブでは、そのテストの各イベントについて確認することができます。

イベントのペインのツリーを展開してイベントペイン最後の pass から全テストが成功しているとわかります。

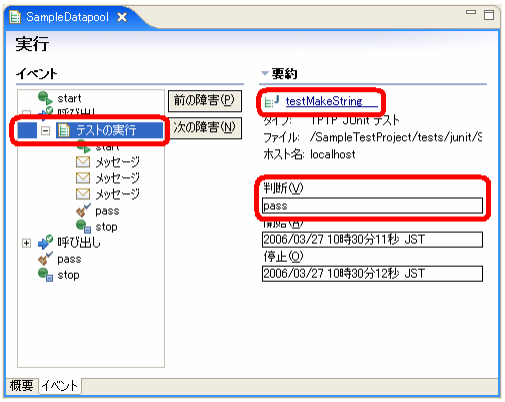


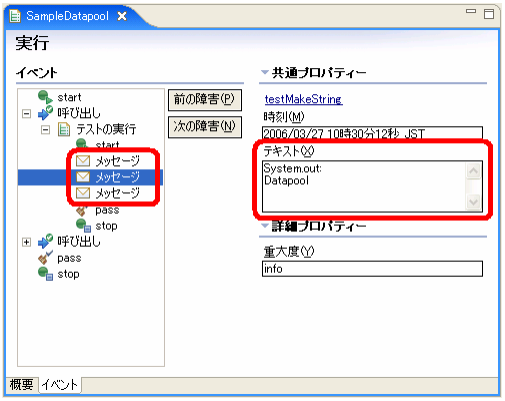
一つ目の呼び出しのテストの実行ではテスト・メソッド testMakeString を実行していることがわかります。

判断で pass とあることや、イベントペインの

テスト実行の pass から testMakeString メソッドの実行が成功しているとわかります。

またテストの実行における各メッセージのテキストから、テスト・クラスでデータプールのレコードごとに出力していた結果が確認できます。



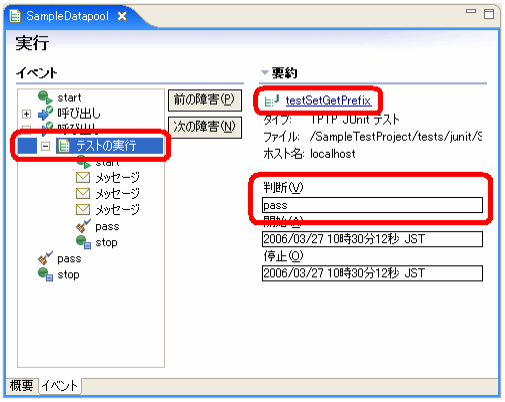


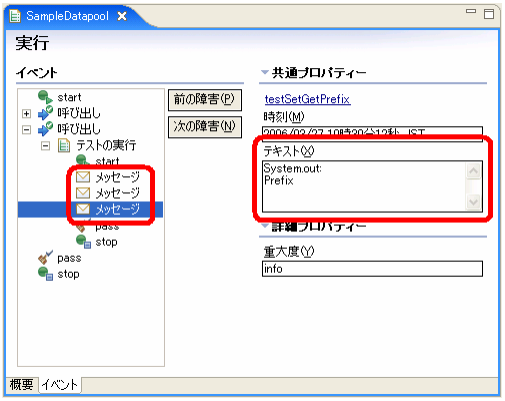
二つ目の呼び出しのテストの実行ではテスト・メソッド testSetGetPrefix を実行していることがわかります。

判断で pass とあることや、イベントペインの

テスト実行の pass から testSetGetPrefix メソッドの実行が成功しているとわかります。

またテストの実行における各メッセージのテキストから、テスト・クラスでデータプールのレコードごとに出力していた結果が確認できます。





デプロイメントの使用

デプロイメントはテストの配備を表現するもので、テストの実行に 1 つ以上必要となり、Artifact と Location の組み合わせを指定します。

ここでは「3.8.1.JUnit テスト」で行った JUnit テストで、作成したデプロイメント使用したテストを行います。

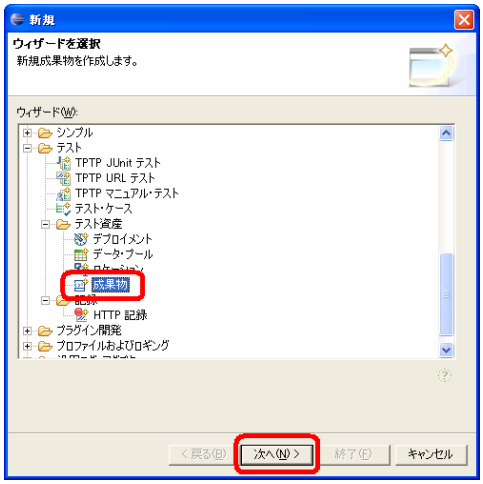
先に「3.8.1.JUnit テスト」を行ってください。

始めに Artifact を作成します。

Artifact はテストを配備する上で、実行するテストや実行マシンへコピーするライブラリを定義します。

（1）メニューから **ファイル | 新規 | その他** を選択します。

[**新規**]画面で、**テスト | テスト資産 | 成果物** を選択して[**次へ**]ボタンを押します。



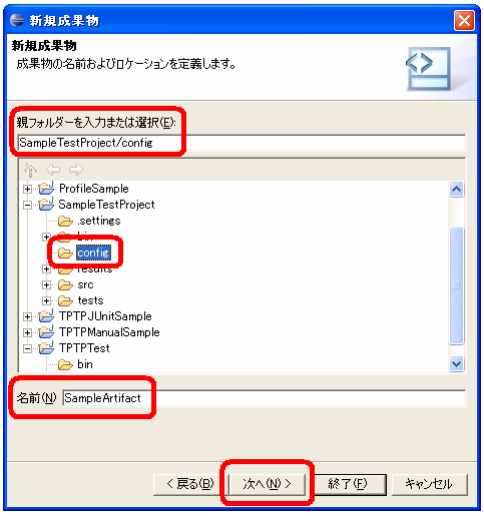
（2）[**新規成果物**]画面で、成果物の保管場所を設定します。以下のように設定して[**次へ**]ボタンを押します。

親フォルダーを入力または選択：

SampleTestProject/config

名前： **SampleArtifact**

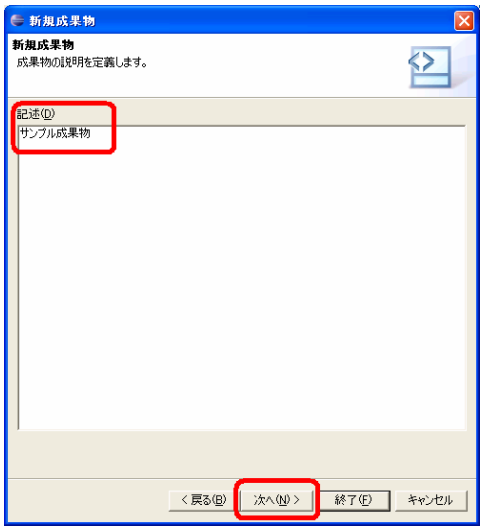
（親フォルダーは直接入力しても、ツリーから選択してもかまいません）



(3)成果物の記述を設定します。

以下のように入力して[**次へ**]ボタンを押します。

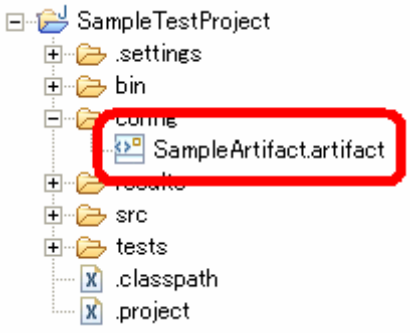
記述:**サンプル成果物**



(4)成果物に関連させるテスト・エレメントを選択します。

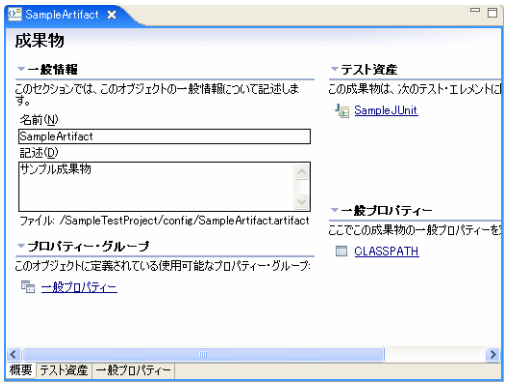
今回は「3.8.1.JUnit テスト」で作成した **SampleJUint.testsuite** を選択して、[**終了**]ボタンを押します。

(5)新規成果物の定義を完了すると、設定した保管場所に成果物 **SampleArtifact.artifact** が生成されます。



(6)成果物をダブル・クリックしてエディターを表示することで、成果物の編集を行えます。

今回は特に編集を行いません。



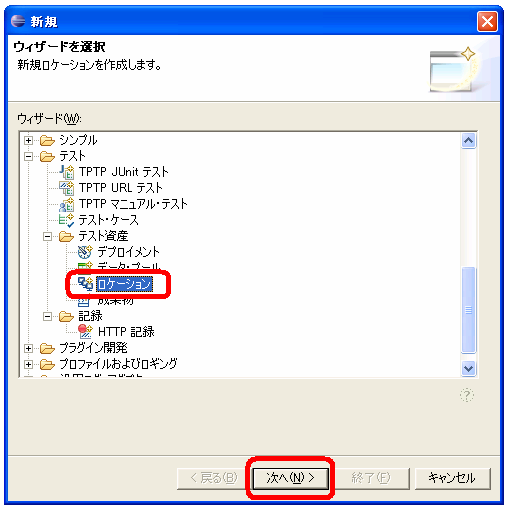
以上で、Artifact の作成を完了します。

次に Location を作成します。

Location はテストを配備する上で、テストを実行するマシンやテストを配置するファルダを指定します。

(1)メニューから **ファイル | 新規 | その他** を選択します。

[**新規**]画面で、**テスト | テスト資産 | ロケーション** を選択して[**次へ**]ボタンを押します。



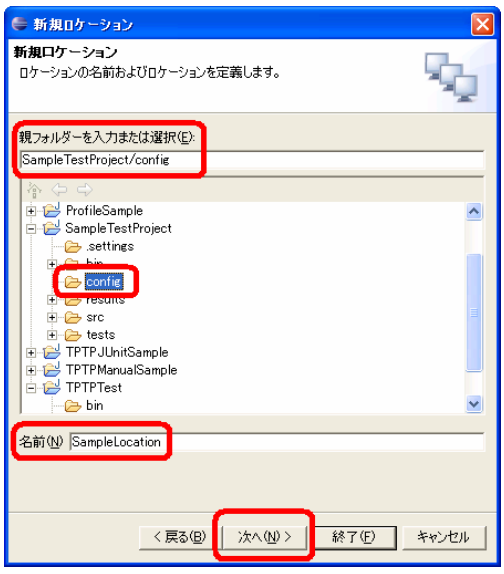
(2)[**新規ロケーション**]画面で、ロケーションの保管場所を設定します。以下のように設定して[**次へ**]ボタンを押します。

親フォルダーを入力または選択:

SampleTestProject/config

名前: **SampleLocation**

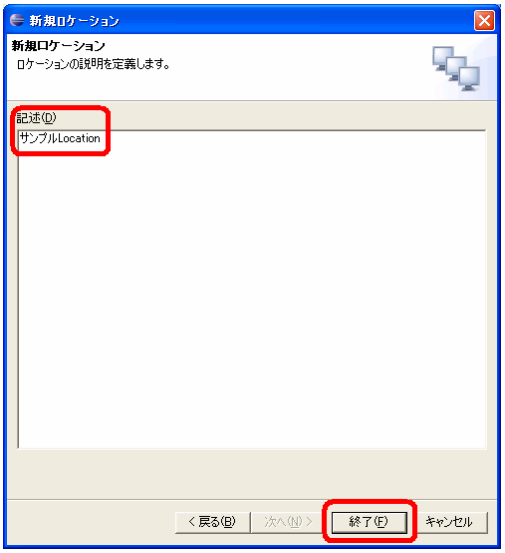
(親フォルダーは直接入力しても、ツリーから選択してもかまいません)



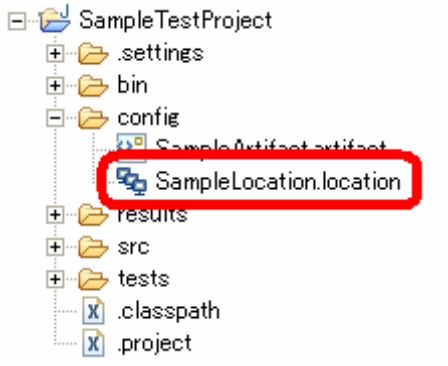
(3)ロケーションの記述を設定します。

以下のように入力して[**終了**]ボタンを押します。

記述：**サンプル Location**

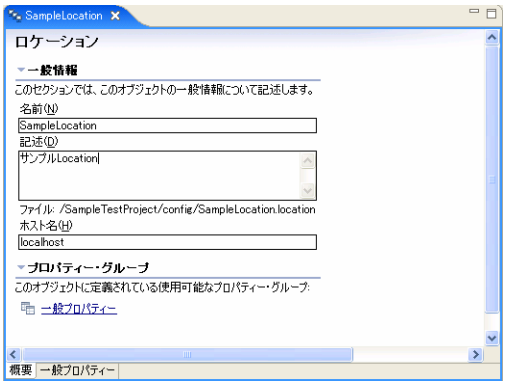


(4)新規ロケーションの定義を完了すると、設定した保管場所にロケーション **SampleLocation.location** が生成されます。



(5)ロケーションをダブル･クリックしてエディターを表示することで、ロケーションの編集を行えます。

今回は特に編集を行いません。



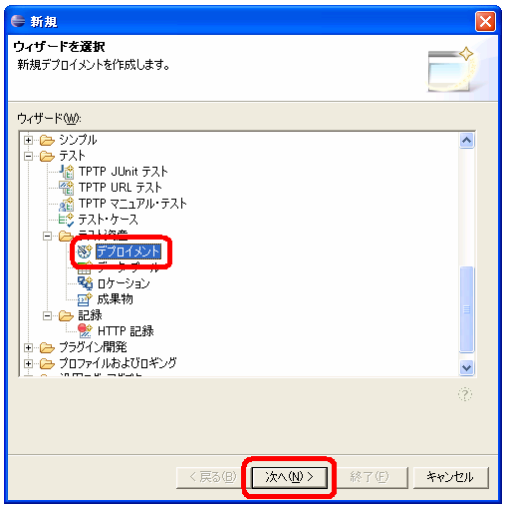
以上で Location の作成を完了します。

コンポーネントテスト機能のテスト環境のサポート対象は **localhost** だけです。
よってロケーションのホスト名には localhost 以外を設定しないでください。

最後に Deployment を作成します。

(1)メニューから **ファイル | 新規 | その他** を選択します。

[**新規**]画面で、**テスト | テスト資産 | デプロイメント** を選択して[**次へ**]ボタンを押します。



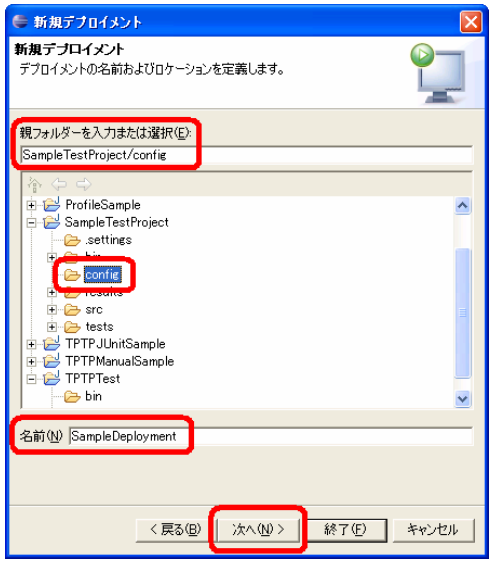
(2)[**新規デプロイメント**]画面で、デプロイメントの保管場所を設定します。以下のように設定して[**次へ**]ボタンを押します。

親フォルダーを入力または選択:

SampleTestProject/config

名前: **SampleDeployment**

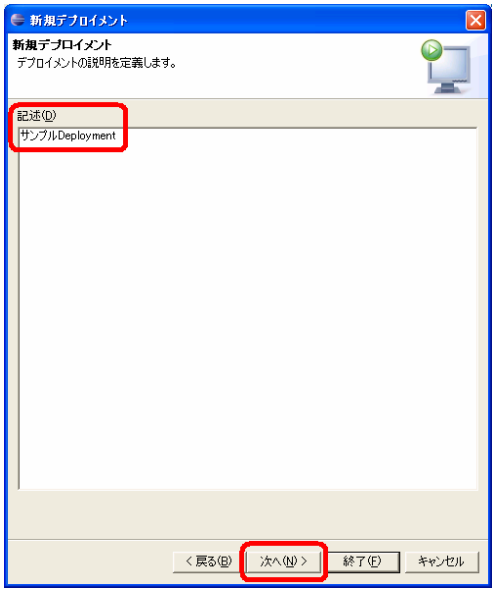
(親フォルダーは直接入力しても、ツリーから選択してもかまいません)



(3)デプロイメントの記述を設定します。

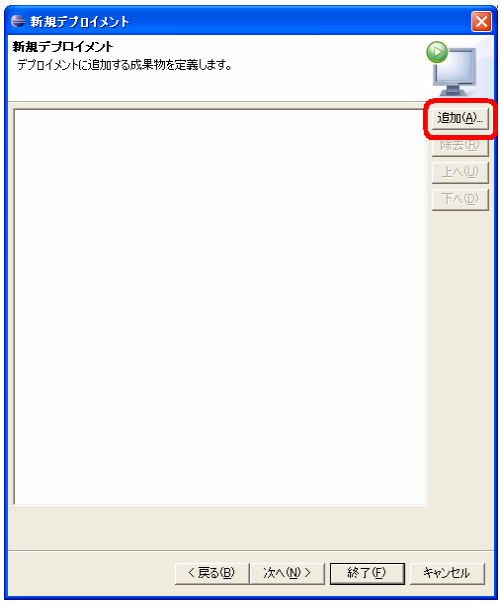
以下のように入力して[**次へ**]ボタンを押します。

記述:**サンプル Deployment**



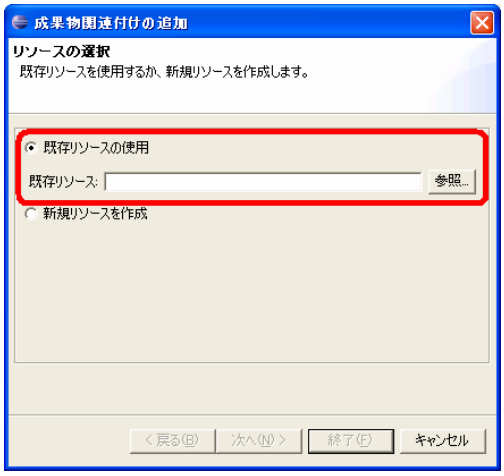
（4）デプロイメントに追加する成果物(Artifact)の定義を行います。

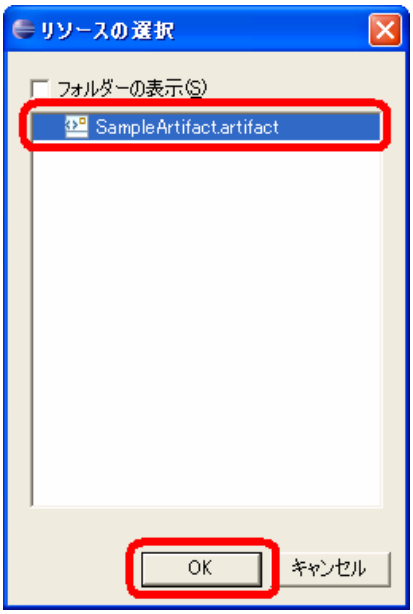
まず[**追加**]ボタンを押します。



（5）[**成果物関連付けの追加**]画面で、「**既存のリソースを使用**」を選択して、[**参照**]ボタンを押します。

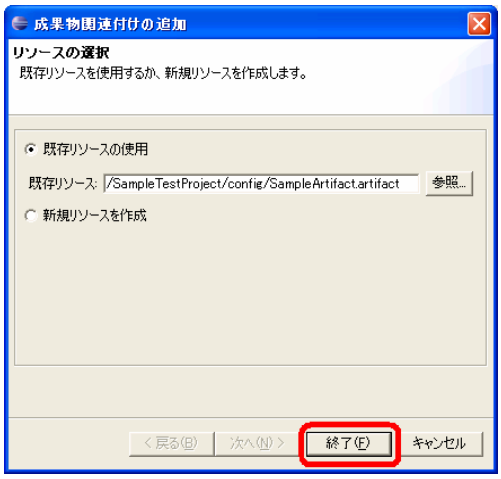
[**リソースの選択**]画面で、表示されている成果物(Artifact)から**SampleArtifact.artifact**を選択して、[**OK**]ボタンを押します。

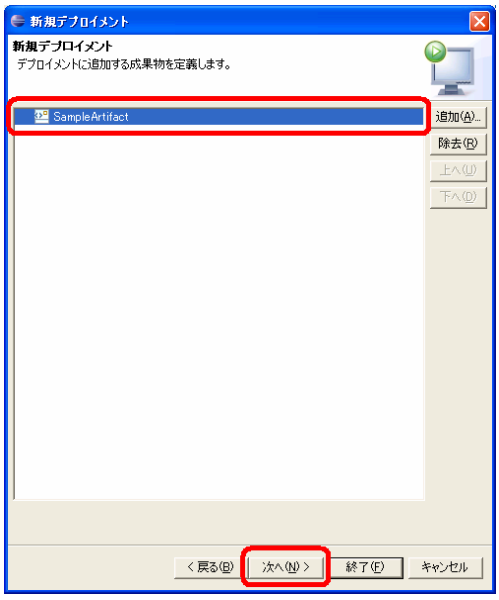




(6)[**成果物関連付けの追加**]画面に戻り、**既存のリソース**に選択して成果物(Artifact)のパスが表示されていることを確認して、[**終了**]ボタンを押します。

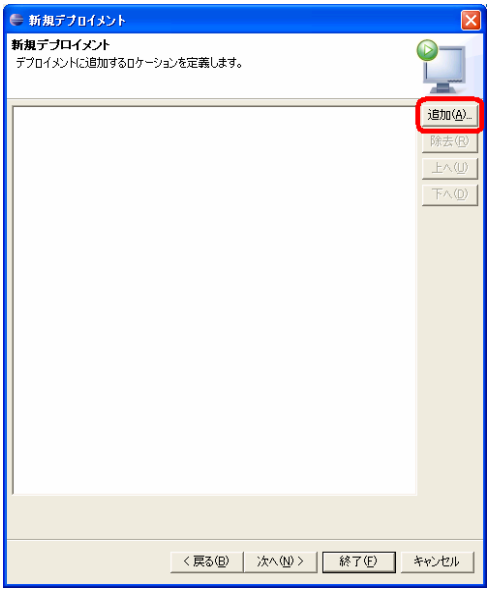
[**新規デプロイメント**]画面に戻り、成果物 **SampleArtifact.artifact** が追加・選択されていることを確認して、[**次へ**]ボタンを押します。





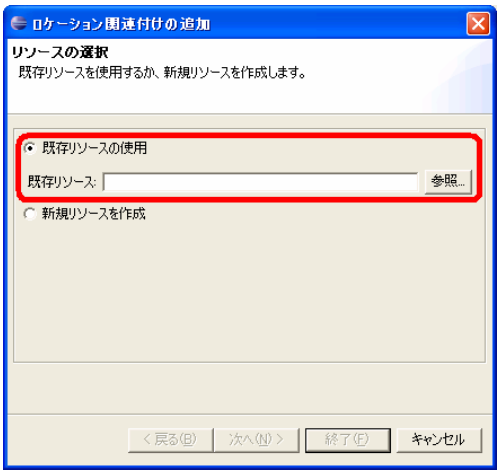
(7)デプロイメントに追加するロケーション(Location)の定義を行います。

まず[**追加**]ボタンを押します。



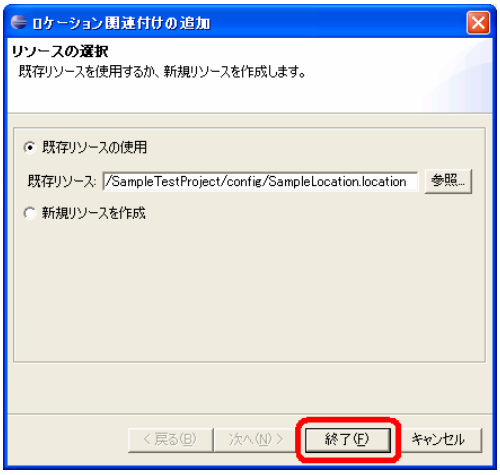
(8)[**ロケーション関連付けの追加**]画面で、「**既存のリソースを使用**」を選択して、[**参照**]ボタンを押します。

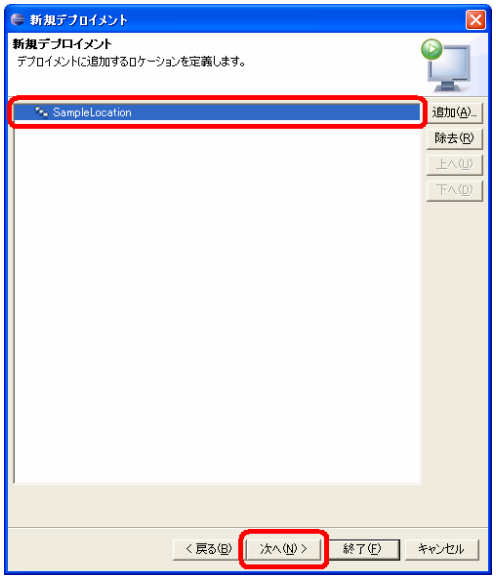
[**リソースの選択**]画面で、表示されているロケーション(Location)から **SampleLocation.location** を選択して、[**OK**]ボタンを押します。



(9)[**ロケーション関連付けの追加**]画面に戻り、**既存のリソース**に選択してロケーション(Location)のパスが表示されていることを確認して、[**終了**]ボタンを押します。

[**新規デプロイメント**]画面に戻り、ロケーション **SampleLocation.location** が追加・選択されていることを確認して、[**次へ**]ボタンを押します。



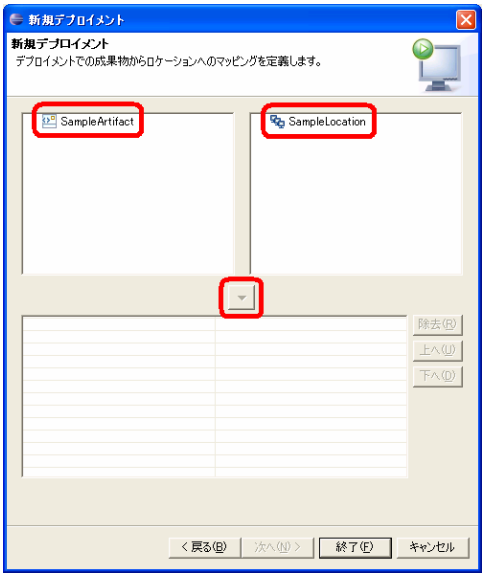


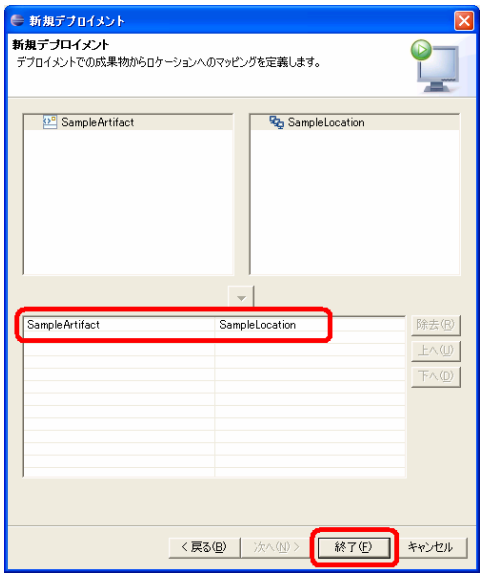
（10）デプロイメントでの成果物からロケーションへのマッピングを定義します。

画面上部のエリア内に表示されている **SampleArtifact** と **SampleLocation** を選択して[▼]ボタンを押します。

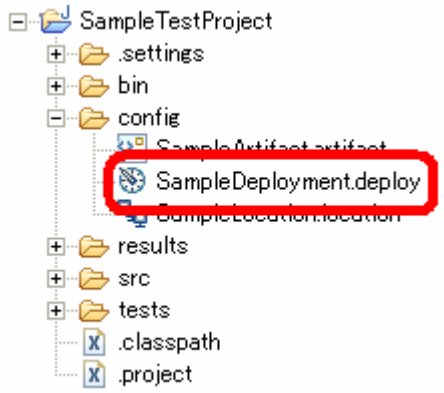
画面下部のエリア内に選択した成果物とロケーションのペアが表示されます。

[**終了**]ボタンを押します。



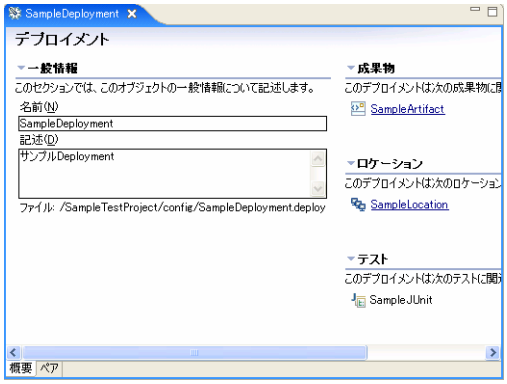


（11）新規デプロイメントの定義を完了すると、設定した保管場所にデプロイメント **SampleDeployment.deploy** が生成されます。



（12）デプロイメントをダブル・クリックしてエディターを表示することで、デプロイメントの編集を行えます。

今回は特に編集を行いません。



以上で Deployment の作成を完了します。

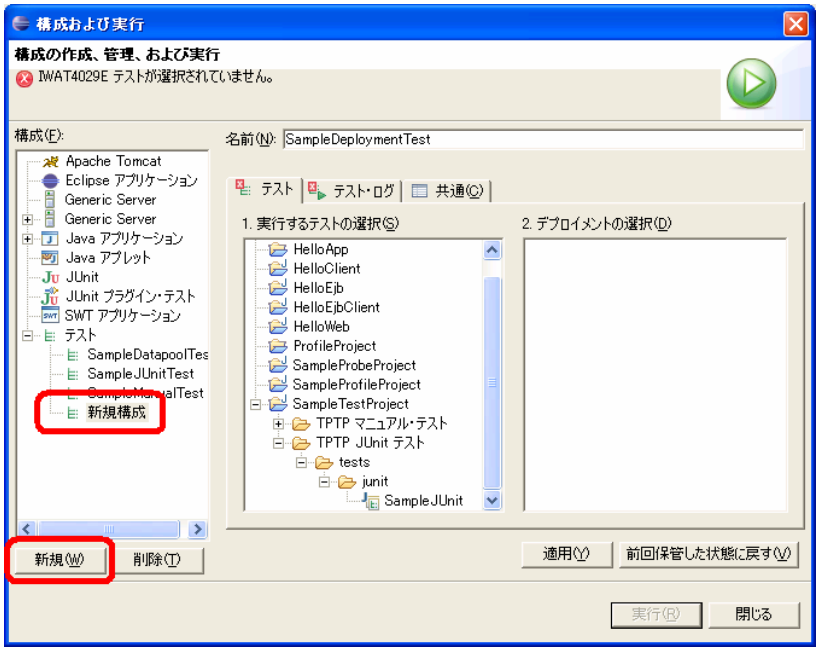
作成した Deployment を使用した実行は以下のようになります。

メニューの**実行｜構成および実行** を選択します。

[**構成および実行**]画面が表示されます。

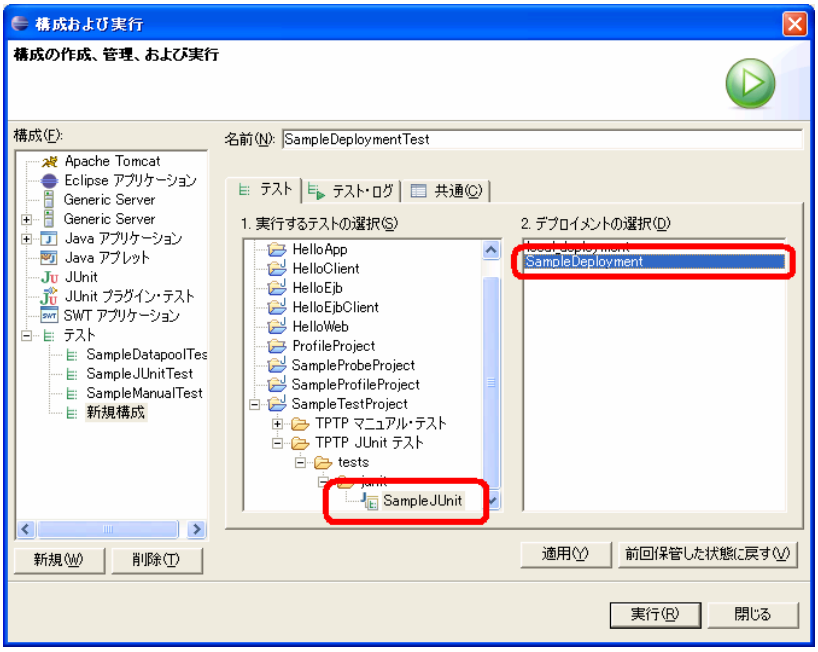
構成の**テスト**を選択して[**新規**]ボタンを押して、新規構成を作成します。

名前を **SampleDeploymentTest** と設定します。



[**テスト**]タブで、実行するテスト（SampleTestProject/TPTP JUnit テスト/tests/junit/SampleJUnit）を

選択します。次にデプロイメントの選択に **SampleDeployment** が追加されているので、選択します。



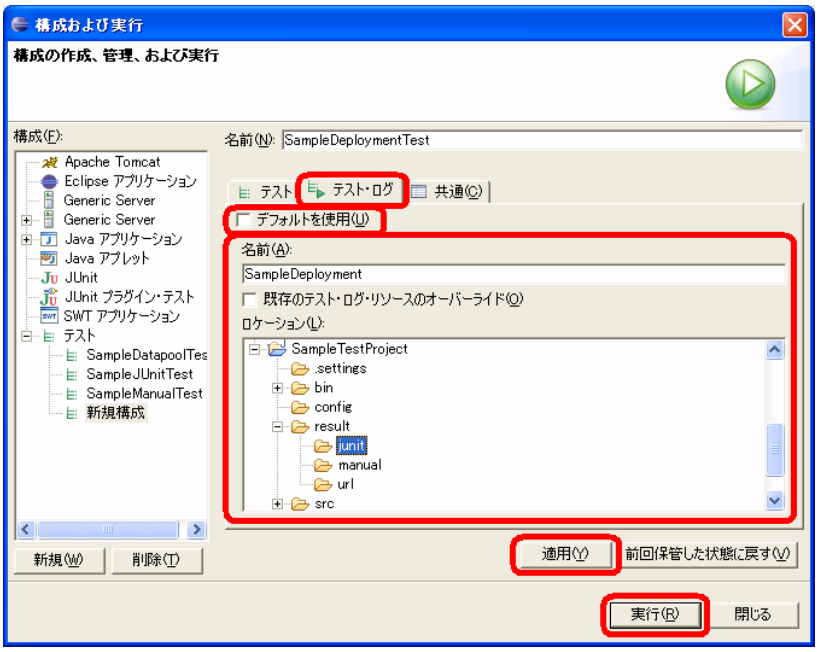
次に[**テスト・ログ**]タブで、テスト結果ファイルの保管場所を設定します。

デフォルトを使用からチェックを外して、以下のように設定します。

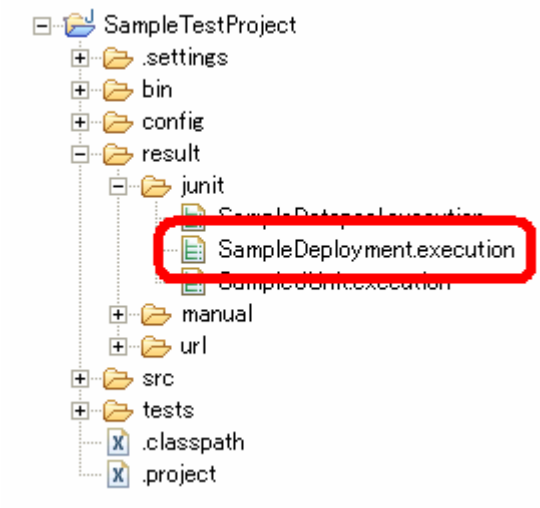
名前：**SampleJUnit**

ロケーション：**SampleTestProject/results/junit**

設定が完了しましたら、[**適用**]ボタンを押して実行構成を保管して、[**実行**]ボタンを押してテストを実行します。



テストが終了すると、以下の実行結果ファイルが作成されます。



次に実行結果の確認を行います。

実行結果ファイル **SampleDeployment.execution** をダブル・クリックして、エディターに表示してください。

このデプロイメントを使用したテストの結果は、テスト結果の判断やテスト・イベントの構成が「3.8.1.JUnit テスト」で行ったテストの結果と同様のものとなっていることを確認してください。

分析レポートの生成

TPTP ではテスト結果をもとに、HTML 形式のレポートを出力することができます。

テストから生成できるレポートには以下のものがあります。

・テスト・パス・レポート

・時間フレーム履歴

また URL テストからは上記の他に、以下のレポートも作成することができます。

・HTTP ページ・ヒット率

・HTTP ページ応答時間

ここでは各レポートの生成を行います。

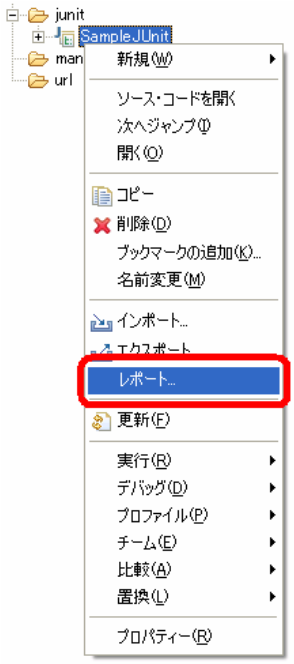
メニューから **ウィンドウ | パースペクティブを開く | テスト** を選択して、予めパースペクティブを切り替えておきます。

JUnit テストの結果をもとに、「**テスト・パス・レポート**」と「**時間フレーム履歴**」の作成を行います。

[**テスト・ナビゲーター**]ビューから、テストを完了した**テスト・スイート**を選択します。

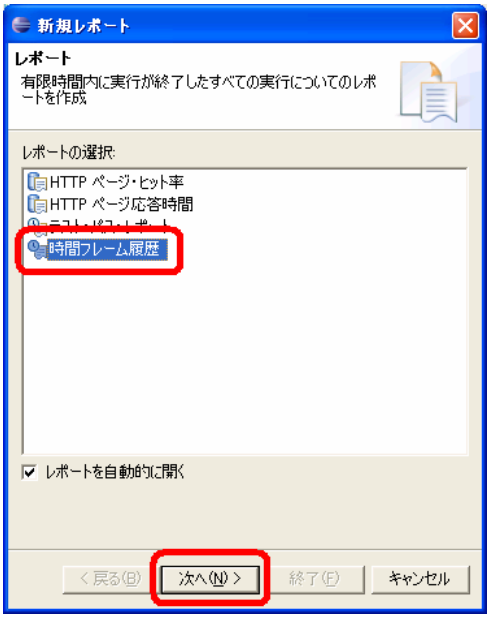
右クリックしてポップアップメニューから、**レポート**を選択します。

今回は **testsuites/junit/SampleJUnit.testsuite** を選択します。



[**新規レポート**]画面が表示されるので、**テスト・パス・レポート** または **時間フレーム履歴** を選択して、[**次へ**]ボタンを押します。

今回は **時間フレーム履歴** を選択してみます。



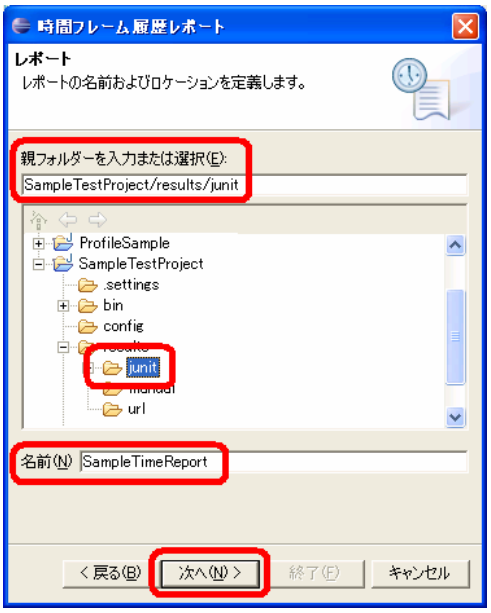
次に[**時間フレーム履歴レポート**]画面で、レポートの保管場所を設定します。以下のように設定して[**次へ**]ボタンを押します。

親フォルダー・・・:

SampleTestProject/results/junit

名前： **SampleTimeReport**

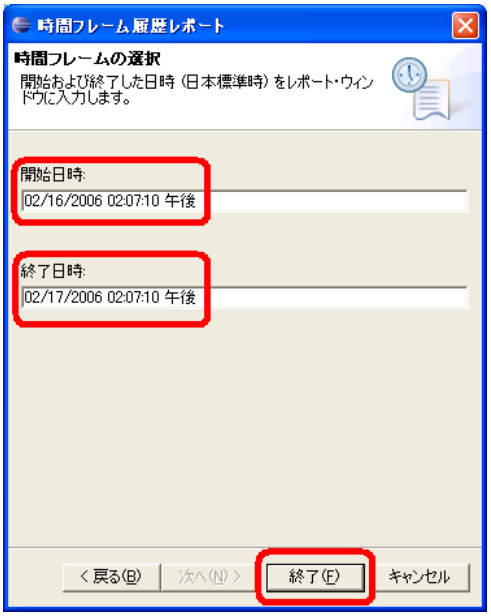
(親フォルダーは直接入力しても、ツリーから選択してもかまいません)



最後に、行ったテストの特定のために、テストを行った時間を設定します。

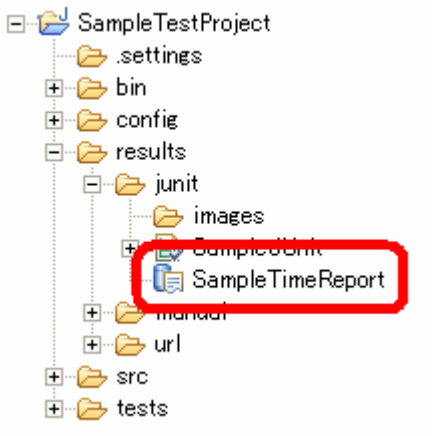
開始時間 と **終了時間** の間に行われたテストのテスト結果レポートを作成します。

開始時間と終了時間にそれぞれ値を入力して[**終了**]ボタンを押します。



レポートの生成が完了すると以下のエディターが表示され、レポートの内容が表示されます。

また設定した保管場所に以下のリソースが追加されます。



保管場所に追加されたリソースはすぐにビューには表示されないため、一度ビューの**更新**を行う必要があります。

レポートはエディターを閉じてしまうと再度 Eclipse 上では表示できないため、もう一度内容を確認したい場合はブラウザで表示してください。
その際にグラフを表示するために Scalable Vector Graphics (SVG) ブラウザー・プラグインが必要になります。
SVG ブラウザー・プラグインの詳細は、Eclipse ヘルプの **TPTP テスター・ガイド | TPTP でのパフォーマンス・テスト | テスト結果の分析** を参考にしてください。

次に URL テストのテスト結果をもとに「**HTTP ページ・ヒット率**」と「**HTTP ページ応答時間**」の作成を行います。

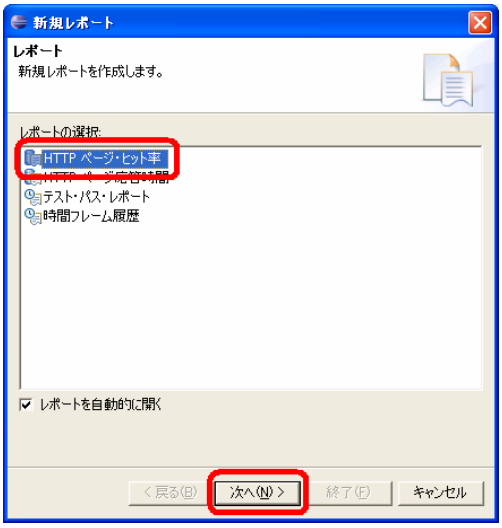
[**テスト・ナビゲーター**]ビューから、テストを完了した URL テストの**テスト・スイート**を選択します。

右クリックしてポップアップメニューから、**レポート**を選択します。

今回は **tests/url/SampleURL.testsuite** を選択します。

[**新規レポート**]画面が表示されるので、**HTTP ページ・ヒット率** または **HTTP ページ応答時間** を選択して、[**次へ**]ボタンを押します。

今回は **HTTP ページ・ヒット率** を選択します。



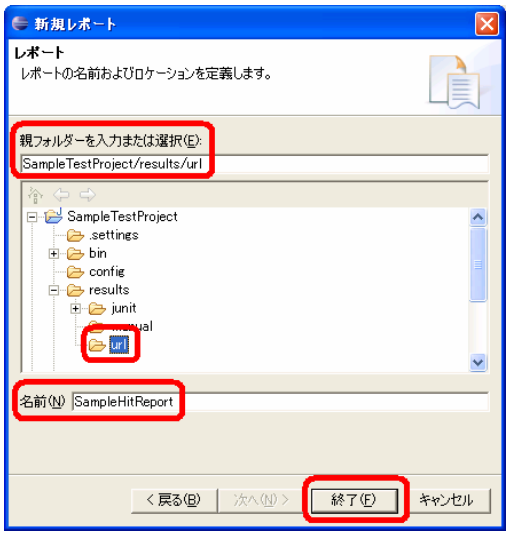
次に[**新規レポート**]画面で、レポートの保管場所を設定します。以下のように設定して[**終了**]ボタンを押します。

親フォルダー・・・:

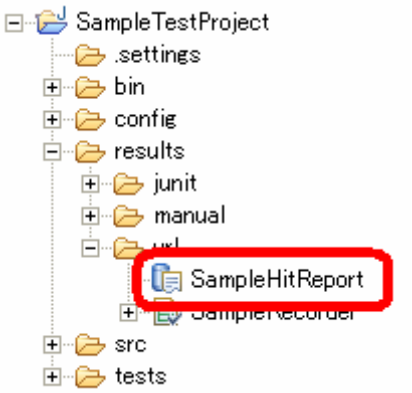
SampleTestProject/results/url

名前: **SampleHitReport**

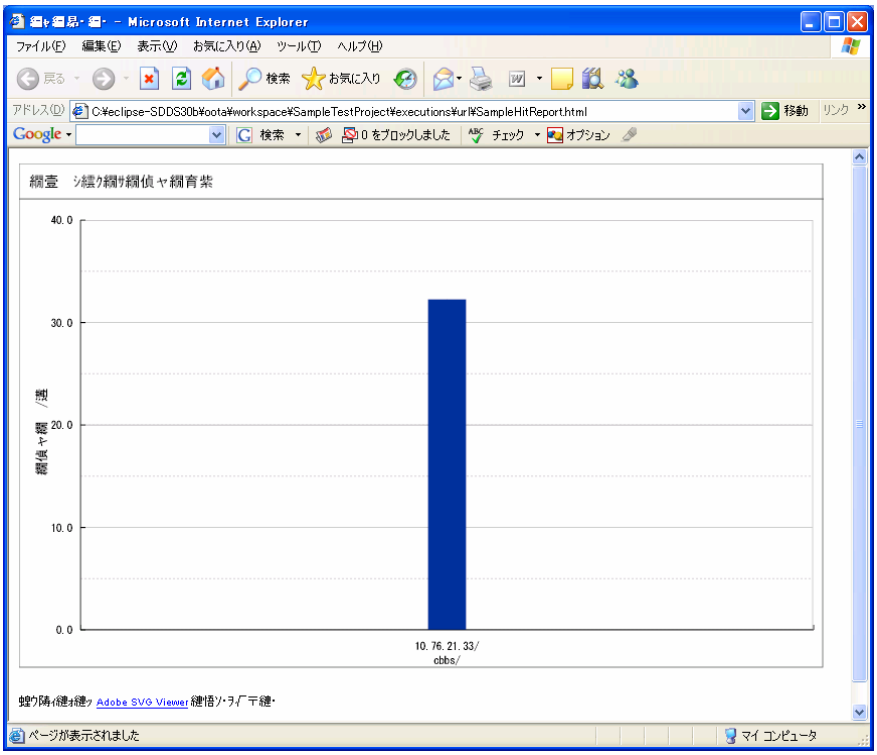
(親フォルダーは直接入力しても、ツリーから選択してもかまいません)



以上でレポートが作成され、設定した保管場所に以下のリソースが追加されます。



	保管場所に追加されたリソースはすぐにビューには表示されないため、一度ビューの**更新**を行う必要があります。
	「**HTTP ページ・ヒット率**」と「**HTTP ページ応答時間**」は Eclipse のエディターでは表示されません。内容を確認したい場合はブラウザで表示してください。



その際にグラフを表示するために Scalable Vector Graphics (SVG) ブラウザー・プラグインが必要になります。

SVG ブラウザー・プラグインの詳細は、Eclipse ヘルプの **TPTP テスター・ガイド | TPTP でのパフォーマンス・テスト | テスト結果の分析** を参考にしてください。

「**HTTP ページ・ヒット率**」と「**HTTP ページ応答時間**」レポートは生成時に日本語の部分が文字化けします。

1.1.8.モニタリング

TPTP はデータ収集エージェントからのデータの作成、デプロイメント、および収集を容易にしてくれる包括的なデータ収集フレームワークを提供しています。

従来、OS やサーバなどのアプリケーションを配置したプラットフォームの情報は、Windows のタスクマネージャなどを利用して別々取得してきましたが、データ収集フレームワークを使用したデータ収集エージェントを用いることで、それらの情報をまとめて取得して統計的に表示することができます。

ここでは TPTP のモニタリングツールを使用して、Windows のパフォーマンス情報をモニタリングする方法を説明します。

モニタリングの開始

メニューから **ウィンドウ｜パースペクティブを開く｜プロファイルおよびロギング** を選択して、予めパースペクティブを切り替えておきます。

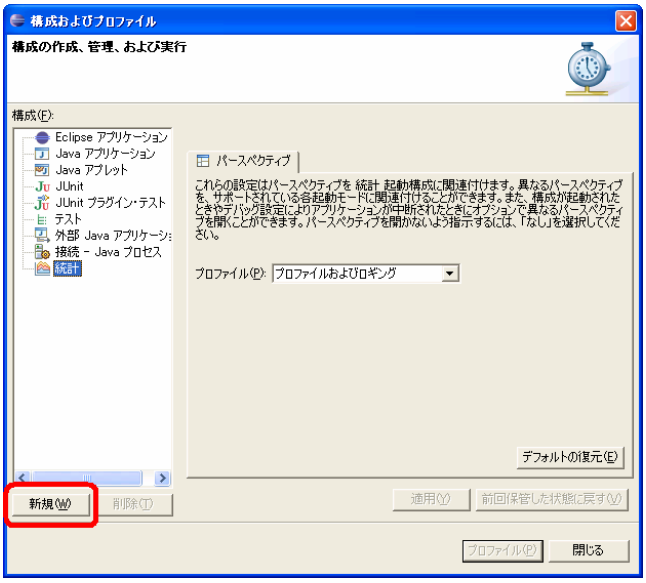
また **ウィンドウ｜ビューの表示｜その他** から以下のビューを予め表示させておきます。

・プロファイルおよびロギング｜エージェント・コントロール

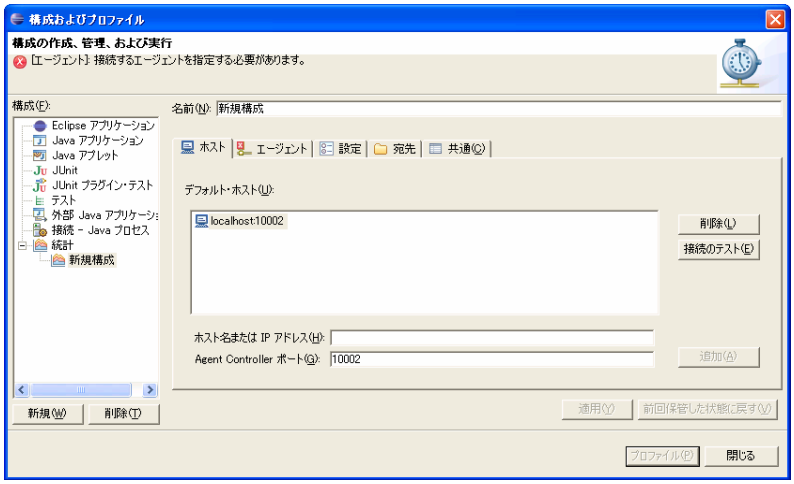
・プロファイルおよびロギング｜統計グラフ

・　プロファイルおよびロギング｜統計グラフの要約

メニューから **実行｜構成およびプロファイル** を選択して、**構成およびプロファイル**画面を表示します。構成内の **統計** を選択して[**新規**]ボタンを押します。



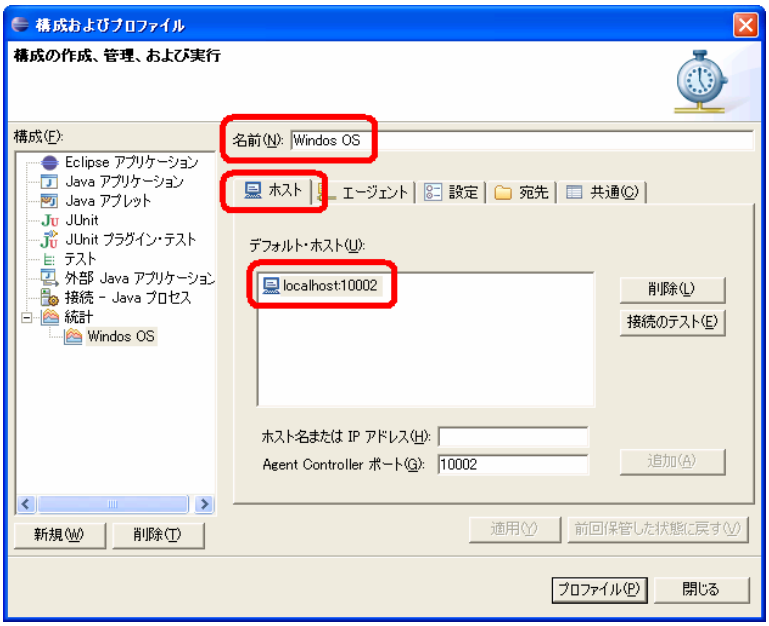
構成内の **統計** の下に **新規構成** が追加され、その内容が表示されます。



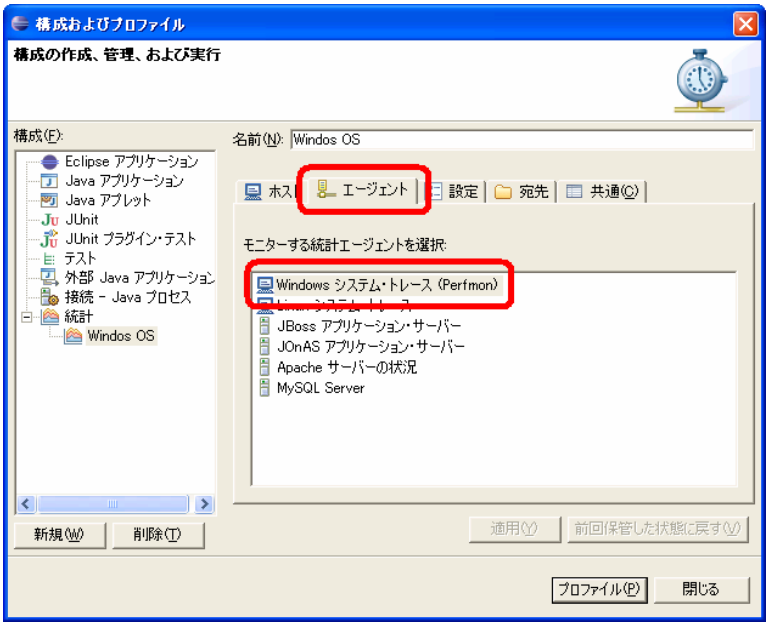
新規構成を次のように設定していきます。

名前を「**Windows OS**」とします。

[**ホスト**]タブで[デフォルト・ホスト]が「**localhost:10002**」が選択されていることを確認します。



[**エージェント**]タブで[モニターする統計エージェントを選択]で、「**Windows システム・トレース**」を選択します。

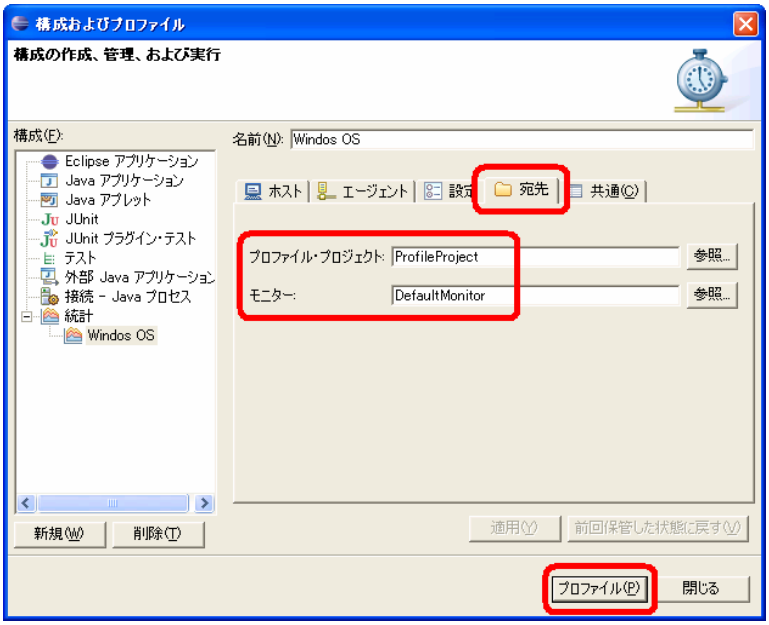


[**宛先**]タブで保管先の「**プロファイル・プロジェクト**」や使用する「**モニター**」を設定します。

今回はデフォルト値のままにしておきます。

設定を完了すると[**プロファイル**]ボタンを押します。

([**設定**]タブと[**共通**]タブはデフォルト値のままにしておきます)



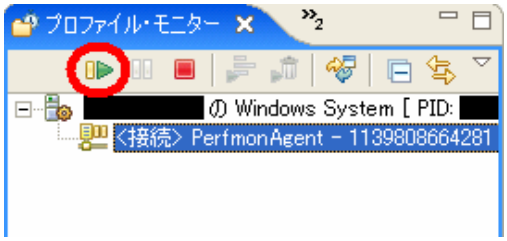
モニタリングのサポート対象は Windows だけです。

モニタリングの結果の表示

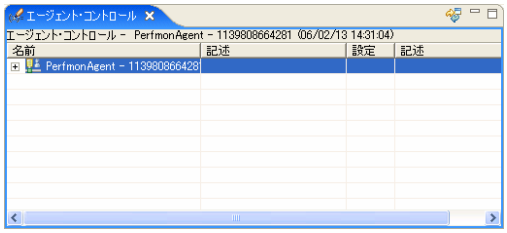
[**プロファイル・モニター**]ビューにエージェントが表示され、[**接続**]状態になっています。

モニタリングを開始することで、対象 OS のモニターを行います。

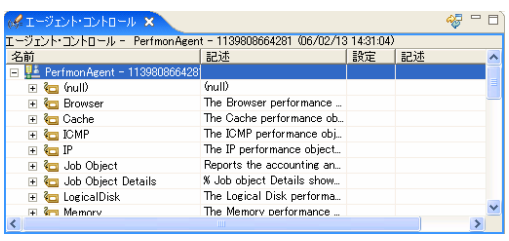
[**プロファイル・モニター**]ビューの 開始ボタンを押してモニタリングを開始します。

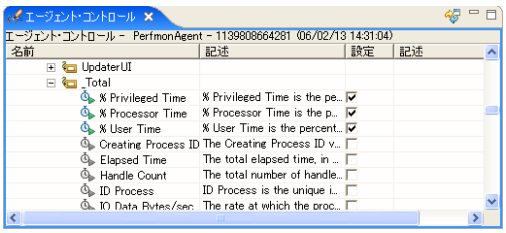


[**プロファイル・モニター**]ビューのエージェントが[**モニター**]状態になったことを確認すると、[**エージェント・コントロール**]ビューを表示してツリーを展開して表示させたいデータにチェックを入れます。

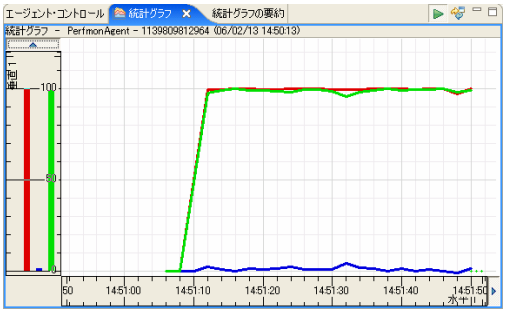


[**エージェント・コントロール**]ビューにモニター内容が表示されない場合、[**プロファイル・モニター**]ビューの （ビューアーにリンク）を押して、[**エージェント・コントロール**]ビューの内容を更新します。

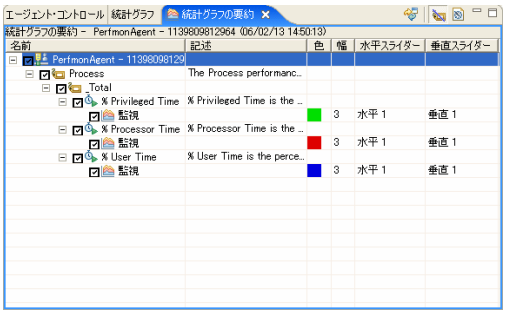




次に[**統計グラフ**]ビューを表示して、OS のデータのモニター経過を確認します。

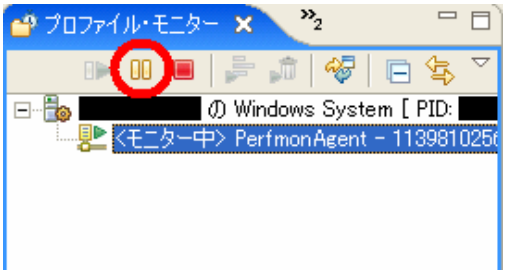


また[**統計グラフの要約**]ビューで、[**統計グラフ**]ビューに表示中のデータグラフにおける線の色や幅を設定することができます。

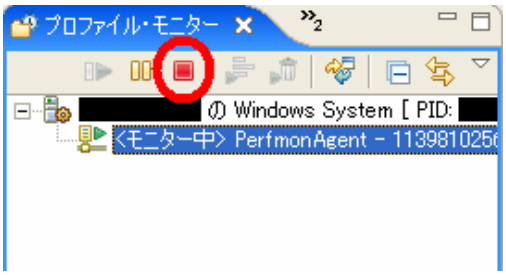


[**プロファイル・モニター**]ビューの 一時停止ボタンを押すことでモニタリングの一時停止を行えます。

再度 開始ボタンを押すことでモニタリングを再開できます。



[**プロファイル・モニター**]ビューの 停止ボタンを押すことでモニタリングを終了します。



以上でモニターを終了します。