



Test Pattern Catalog

xUnit を利用して、効率的にテストを作成するためのテストパターンカタログ

Rev.2003.12.5

目次

第1章	テストパターンとは	1
概要	1
良いテストの条件	1
ターゲットプログラムの要件	1
まとめ	1
第2章	テストパターン詳細	2
ヌルオブジェクト	テスト全般.....	2
インフォメーションセンター	テスト全般.....	4
ビューステート	UIテスト.....	5
オーダーシート	UIテスト.....	8
ロールバック	DBテスト.....	9
補章	参考文献等	12
参考文献	書籍.....	12
DB テストのソースプログラム	12

第1章 テストパターンとは

概要

eXtreme Programming (以下 XP) でテストファーストは重要なプラクティスである。当然、良いテストを作成するためには、テストファーストについての知識や経験が必要になる。実際、テストファーストでテストを作成するときに、どのようにテストを作成してよいか悩むことが少なくない。この様なときに、テストだけではなく、テストとターゲットプログラム（製品となる部分のプログラム）の組合せによって解決方法が見つかることが良くある。

そこで、本ドキュメントでは、テストしやすいプログラム構造とテストの組合せをテストパターンとして紹介する。テストファーストでテストの作成に困ったときのヒントとして利用してほしい。あるいは、ペアプロの休憩時間の話題として、利用させることを希望する。

良いテストの条件

良いテストは、以下のような条件を備えている。

テスト目的が明確である

何をテストするか明確である。また、その目的も明確で、更にひとつであること。

テストの判定が正しい

テストの成功、失敗が正しく判断されている。

テストを独立して実行することができる

テストが他のテストに依存することなく、独立している。

繰り返し実行することができる

何度でも繰り返し、テストを実行することができる。

テストを実行しても、状態が変化しない

テストを実行し、成功した場合でも失敗した場合でも、テストを実行する前と後で何も変わらない。

ターゲットプログラムの要件

良いテストを容易に作成するためには、ターゲットプログラムが以下のような条件のいくつかを満足していることが必要である。

メソッドが一つの機能を実現している

メソッドが明確な一つの機能だけを提供する。

メソッドの結果を提供する

外部に対して、メソッドの処理結果を判断できるような何らかの方法を提供する。

クラスやメソッドの独立度が高いこと

クラスやメソッドが、他のクラスやメソッドとの依存が低い。

特定の環境への依存度が低い

特定のファイルやデータベース構造などに対する依存度が低い。

まとめ

結局の重要なポイントは以下の2点です。

シンプル

テスト、クラス、メソッドなど、全てがシンプルであることが最善である。常にシンプルになっているか確認し、改善（リファクタリング）して、シンプルを追求する。

低い依存度

テスト、クラス、メソッドなど、全てが他のテストやクラスやメソッドに対して、依存している部分を低くすることが重要である。依存度を低くして、高い独立性を追求する。

第2章 テストパターン詳細

Overview

テストで問題といわれるのがユーザインターフェイスのテストなど効率的に行うためのテストパターンなどについて、以下に解説を行う。

ヌルオブジェクト

テスト全般

目的

テストを正しく失敗させる

テストファーストでテストを実行した場合で、まだオブジェクトが存在していないために、テストが正しく失敗せずに、例外が発生してしまうことがある。このようなときに、テストを正しく失敗させる。

テストコードをシンプルにする

テスト内のオブジェクトの存在確認を削除する。

解説

テストを正しく失敗させるために、本来のテストの前にオブジェクトが存在するか確認することがある。しかし、この確認は、本来テストに必要なものであるにも関わらず、数多く同じコードが存在することになる。

この様なときに、ヌルオブジェクトを導入し、余計なロジックを付加することなくテストファーストによる最初の実行でも、正しくテストを失敗させることができる。

利用場面

以下のような場合に、利用することができる。

オブジェクトを返すメソッドのテスト

クラスが提供するメソッドが処理結果としてインスタンスを返すようなメソッドをテストする場合に、正しくテストを失敗させることができる。

テストの最初の実行（レッドバー）

テストの最初の実行時に、レッドバーと理由を正しく表示させる。

制限事項

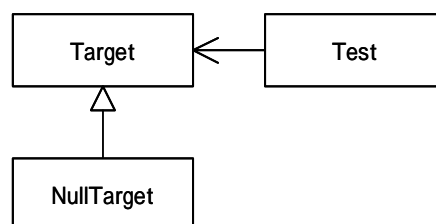
このテストパターンには、以下のような制約事項がある。

ある時点で必要とされないクラスが存在してしまう

このパターンで導入したヌルオブジェクトのクラスは、ある時点で必要がなくなる場合がある。

構造と構成要素

Target
テスト対象となるクラス
NullTarget
テスト対象のヌルクラス
Test
テストクラス



サンプルプログラム

Party クラスと Member クラスがある。今、Party クラスに ID を渡して該当する Member を取得する GetMember メソッドのテストを作成する。

ヌルオブジェクト導入前のテスト

```
1: using System;
2: using TP.Organization;
3: using NUnit.Framework;
4: namespace TP.Organization.Test
5: {
6:     [TestFixture]
7:     public class PartyTestCase
8:     {
9:         [Test]
10:        public void TestGetMember()
11:        {
12:            Party target;
13:            Member member;
14:
15:            target = new Party();
16:            member = target.GetMember("10");
17:            if (member == null )
18:                Assertion.Fail("Member オブジェクトがヌル");
19:            Assertion.AssertEquals( "Member が取得できない","10", member.ID );
20:        }
21:    }
22: }
```

ヌルオブジェクト導入による省略箇所

ヌルオブジェクト導入前の Party クラス

```
23: using System;
24: namespace TP.Organization
25: {
26:     public class Party
27:     {
28:         public Party()
29:         {
30:         }
31:         public Member GetMember(string id)
32:         {
33:             return null;
34:         }
35:     }
36: }
```

ヌルオブジェクト導入後の Party クラス

```
37: using System;
38: namespace TP.Organization
39: {
40:     public class Party
41:     {
42:         public Party()
43:         {
44:         }
45:         public Member GetMember(string id)
46:         {
47:             return Member.newNull();
48:         }
49:     }
50: }
```

目的

環境に依存するプログラムをテストする

プログラムの動作がファイルやデータベースなどの環境に依存する場合がある。このようなケースで、効率的にテストを行う。

解説

環境に依存するプログラムをテストする場合、テストを実行する前にテストのための環境を構築する必要がある。テストを実行する前に、テスト用の環境を動的に構築する方法では、テストの作成工数やテストの実行時間が掛かってしまう。

この様なときに、事前にテスト用に環境を構築し、実行時に環境を切り替えてテストを可能とする。ターゲットプログラムの環境に依存する設定部分をインフォメーション・センタークラスで一元管理する。また、一元管理した、環境情報を設定変更可能とし、テスト時には、様々なテスト環境を切り替えてテストを行う。

利用場面

以下のような場合に、利用することができる。

データベースの状態に依存するテストの実行

データベースの状態に依存するテストを行う場合に、状態を再現したデータベースに環境を変更してテストを実行する。

設定ファイルやデータファイルに依存するテストの実行

ファイルの内容に依存するテストを行う場合に、様々な内容のファイルを用意し、対象を変更してテストを実行する。

制限事項

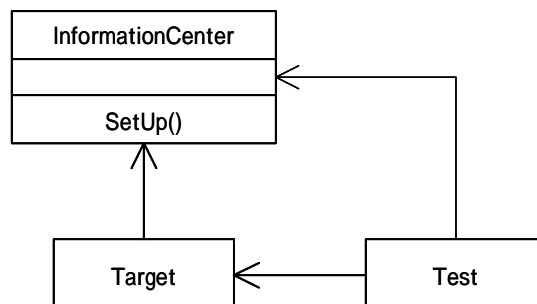
このテストパターンには、以下のような制約事項がある。

必ずしも必要とされない設定変更機能が残ってしまう

このパターンで導入したインフォメーション・センターのクラスは、要求に必要な機能が実装される。

構造と構成要素

Target
テスト対象となるクラス
InformationCenter
設定管理クラス
Test
テストクラス



目的

ユーザーインターフェ이스のロジックをテストする

ユーザーインターフェ이스のテストは、自動化することが難しい。ユーザーインターフェ이스の状態やロジックを別クラスに分離して、テストする。

ユーザーインターフェ이스の状態変化をテストする

ユーザーインターフェ이스は、状態変化に応じて描画を変更する場合がある。このような動作をテストすることは難しい。ユーザーインターフェ이스の状態やロジック分離したクラスにイベントを追加することで、テストを行う。

解説

ユーザーインターフェ이스のテストは、一般に手作業で行うことが多い。これは、ユーザーインターフェ이스の動作や変化を自動テストで確認することが難しいからである。

この様なときに、ユーザーインターフェ이스の状態やロジックを別クラスに分離し、自動テストを行う。また、分離したクラスにイベントを追加することで、描画の更新タイミングなどのロジックに対してテストを可能とする。

利用場面

以下のような場合に、利用することができる。

ユーザーインターフェ이스のテスト

ユーザーインターフェ이스の状態やロジックを自動テストする。

シナリオテスト

複数のユーザーインターフェイスに対応したビューステートクラスを作成し、様々なシナリオテストを行う。

制限事項

このテストパターンには、以下のような制約事項がある。

描画情報を二重持ちする場合がある

ユーザーインターフェイス自体ではなく、分離したクラスで状態を関するために、情報の二重持ちが発生する場合がある。

ユーザーインターフェイスの描画内容はテストできない

ユーザーインターフェイスで高度なコントロールを使用してグラフを表示する場合などでは、データなどはテストすることができるが、グラフ描画自体のテストを行うことはできない。

構造と構成要素

View

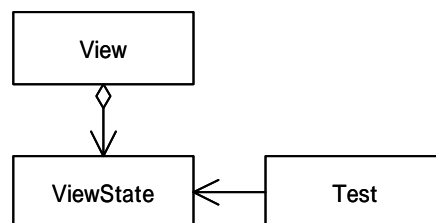
ユーザーインターフェイスのクラス

ViewState

ユーザーインターフェイスの状態管理クラス

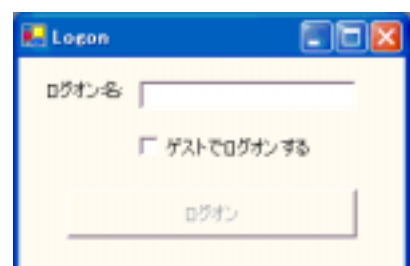
Test

テストクラス



サンプルプログラム

画面の状態を保持する `LogonFormState` クラスとテストクラス `LogonFormStateTestCase` クラスがある。画面の状態としては、ログオンボタンの状態およびゲストログオン指定された状態を管理する。また、ボタンの状態変化更新のタイミングも通知を行う。



ログオン画面の状態保持クラス

```
51: using System;
52: namespace Xpjug.NetBank.Core
53: {
54:     public delegate void ViewStateUpdateEventHandler();
55:     /// <summary>
56:     /// LogonFormState の概要の説明です。
57:     /// </summary>
58:     public class LogonFormState
59:     {
60:         public event ViewStateUpdateEventHandler UpdateEvent;
61:         private void RaiseUpdateEvent()
62:         {
63:             if ( UpdateEvent != null )
64:                 UpdateEvent();
65:         }
66:
67:         private string name;
68:         public string Name
69:         {
70:             get{
71:                 if( guestFlag == true )
72:                     return "guest";
73:                 else
74:                     return name;
75:             }
76:             set{
77:                 name = value;
78:                 RaiseUpdateEvent();
79:             }
80:         }
81:         private bool guestFlag;
82:         public bool GuestFlag
83:         {
84:             set{
85:                 guestFlag = value;
86:                 RaiseUpdateEvent();
87:             }
88:         }
89:         public bool EnableName
90:         {
91:             get{
92:                 if( guestFlag == false )
93:                     return true;
94:                 else
95:                     return false;
96:             }
97:         }
98:         public LogonFormState()
99:         {
100:             name = "";
101:             guestFlag = false;
102:         }
103:         public bool CanLogon()
104:         {
105:             if( guestFlag == true || ( name != null && name != "" ) )
106:                 return true;
107:             else
108:                 return false;
109:         }
110:         public bool IsGuest()
111:         {
112:             return guestFlag;
113:         }
114:     }
115: }
```

描画の更新イベント

ログオンの可能状態

ログオン画面の状態保持テストクラス

```
116: using System;
117: using NUnit.Framework;
118: using Xpjug.NetBank.Core;
119:
120: namespace Xpjug.NetBank.Test
121: {
122:     /// <summary>
123:     /// LogonFormStateTestCase の概要の説明です。
124:     /// </summary>
125:     [TestFixture]
126:     public class LogonFormStateTestCase
127:     {
128:         private LogonFormState target;
129:
130:         [SetUp]
131:         public void Setup()
132:         {
133:             target = new LogonFormState();
134:             target.UpdateEvent += new ViewStateUpdateEventHandler( UpdateState );
135:         }
136:
137:         [Test]
138:         public void TestInit()
139:         {
140:             Assertion.AssertEquals( false, target.CanLogon() );
141:             Assertion.AssertEquals( "", target.Name );
142:             Assertion.AssertEquals( true, target.EnableName );
143:         }
144:
145:         [Test]
146:         public void TestSetName()
147:         {
148:             UpdateEventFlag = false;
149:             target.Name = "xpjug";
150:             Assertion.AssertEquals( true, target.CanLogon() );
151:             Assertion.AssertEquals( "xpjug", target.Name );
152:             Assertion.AssertEquals( true, UpdateEventFlag );
153:             Assertion.AssertEquals( false, target.IsGuest() );
154:             Assertion.AssertEquals( true, target.EnableName );
155:         }
156:
157:         [Test]
158:         public void TestGuest()
159:         {
160:             UpdateEventFlag = false;
161:             target.GuestFlag = true;
162:             Assertion.AssertEquals( true, target.IsGuest() );
163:             Assertion.AssertEquals( true, target.CanLogon() );
164:             Assertion.AssertEquals( "guest", target.Name );
165:             Assertion.AssertEquals( true, UpdateEventFlag );
166:             Assertion.AssertEquals( false, target.EnableName );
167:
168:             UpdateEventFlag = false;
169:             target.GuestFlag = false;
170:             Assertion.AssertEquals( false, target.IsGuest() );
171:             Assertion.AssertEquals( false, target.CanLogon() );
172:             Assertion.AssertEquals( "", target.Name );
173:             Assertion.AssertEquals( true, UpdateEventFlag );
174:             Assertion.AssertEquals( true, target.EnableName );
175:         }
176:
177:         private bool UpdateEventFlag;
178:         private void UpdateState()
179:         {
180:             UpdateEventFlag = true;
181:         }
182:     }
183: }
184:
```

初期化画面の状態テスト

画面更新イベントのハンドラ

目的

- クラス間の依存関係を無くして、テストを実行する
- クラス間に直接的な依存関係を無くし、クラスの独立性を高くして、単体テストを実行する。
- ユーザーインターフェイス関連クラスと処理クラスを分離する
- ユーザーインターフェイス関連クラスと処理クラスを分離し、個別に単体テストの実行を可能にする。

解説

クラス間の依存関係が強い場合、単体テストを作成することが難しい。

この様なときに、処理実行に必要な情報をオーダーシートクラスに分離し、クラス間の関連を無くすことで、テストの作成を容易にする。また、複数のビューステートクラスに対して、オーダーシートクラスを渡して、処理実行情報を集めることで、処理を実行せずにシナリオテストを可能とする。また、ユーザーインターフェイスを通さずに、オーダーシートに様々な設定を行ったテストを可能にする。

利用場面

以下のような場合に、利用することができる。

処理のバリエーション実行テスト

オーダーシートクラスに様々な設定を行い、カバレッジの高いテストの実行を行う。

シナリオテスト

複数のビューステートクラスにオーダーシートクラスを渡す方法で、シナリオ型のユーザーインターフェイステストを行う。

制限事項

このテストパターンには、以下のような制約事項がある。

描画情報を二重持ちする場合がある

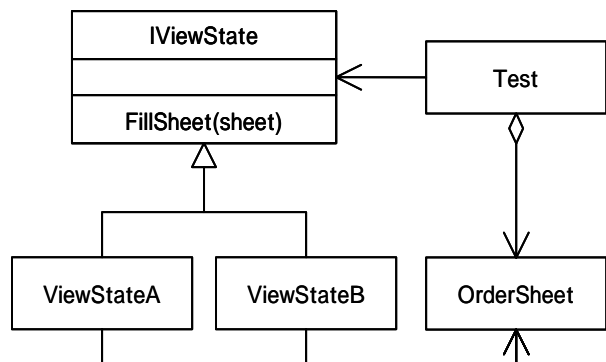
ユーザーインターフェイス自体ではなく、分離したクラスで状態を関するために、情報の二重持ちが発生する場合がある。

ユーザーインターフェイスの描画内容はテストできない

ユーザーインターフェイスで高度なコントロールを使用してグラフを表示する場合などでは、データなどはテストすることができるが、グラフ描画自体のテストを行うことはできない。

構造と構成要素

- OrderSheet
ユーザーインターフェイスのクラス
- IViewState
ユーザーインターフェイスのクラス
- ViewStateA , ViewStateB
ユーザーインターフェイスの状態管理クラス
- Test
テストクラス



目的

DB のテストを行う

DB のテストでは、書き換えや削除処理を行うと DB の状態が変わってしまいテストを繰り返し行うことが難しい。この様な制約なくし、手軽に DB のテストを実行する。

手軽に高速に DB のテストを行う

時間の掛かる DB のセットアップや後処理を用意することなく、てがるに DB のテストを実行する。

解説

DB のテストを行うことは以下のような要因によって、単体テストを作成することが難しい。

- ・ テストを行うために、特定の状態になった DB を用意することは大変である。また、その時間も掛かり、テストを短時間で実行することができない。
- ・ テストのために、モックオブジェクトを用意することで、ある程度 DB に関連するロジックをテストすることはできるが、テストできる範囲が限られてしまう。

テストとターゲットプログラムを同じコネクションに含めることで、テストを実行した後に、全てのトランザクションをロールバックして、DB の状態を変えずにテストを行う。

利用場面

以下のような場合に、利用することができる。

DB の更新処理テスト

DB の更新処理を自動テストする。

制限事項

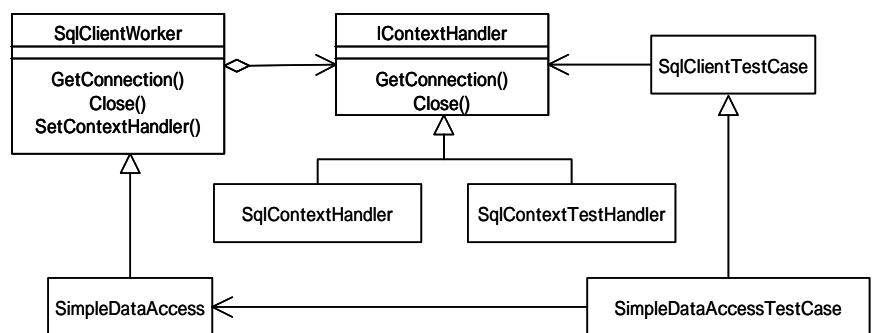
このテストパターンには、以下のような制約事項がある。

RDB の依存する

RDB のロールバックの機能に依存する。

構造と構成要素

- SqlClientWorker
DB 処理の基本クラス
- IContextHandler
DB のコンテキストを管理するインターフェイス
- SqlContextHandler
通常のコンテキスト管理クラス
- SqlContextTestHandler
テスト用のコンテキスト管理クラス
- SqlClientTestCase
テストの基本クラス
- SimpleDataAccess
テストのターゲットクラス
- SimpleDataAccessTestCase
テストクラス



サンプルプログラム

SimpleDataAccess クラスとテストを行う SimpleDataAccessTestCase クラスがある。

SimpleDataAccessTestCase クラス

```
185: using System;
186: using System.Data.SqlClient;
187: using Xpjug.DBTest.Core;
188:
189: using Xpjug.DBTest.TestBase;
190: using NUnit.Framework;
191:
192: namespace Xpjug.DBTest.Test
193: {
194:     /// <summary>
195:     /// SimpleDataAccessTestCase の概要の説明です。
196:     /// </summary>
197:     [TestFixture]
198:     public class SimpleDataAccessTestCase : SqlClientTestCase
199:     {
200:         private SimpleDataAccess target;
201:         private const string connectionString = "server=(local); uid=sa; pwd='Password'; database=TestDB";
202:
203:         [SetUp]
204:         public void SetUp()
205:         {
206:             target = new SimpleDataAccess( connectionString );
207:             SetupContextHandler( target, connectionString );
208:         }
209:
210:         [TearDown]
211:         public void TearDown()
212:         {
213:             TearDownContextHandler();
214:         }
215:
216:         [Test]
217:         public void AddTest()
218:         {
219:             int recordCount;
220:             string[] itemData;
221:
222:             recordCount = GetRecordCount("TestData");
223:             target.Add("XYZ", "東京都");
224:
225:             Assertion.AssertEquals("データが追加されなかった",
226:                                     recordCount + 1,
227:                                     GetRecordCount("TestData"));
228:
229:             itemData = ReadOneRecord("Select * From TestData Where name='XYZ' ");
230:
231:             Assertion.AssertEquals("データが正しく書き込まれなかった",
232:                                     "東京都", itemData[2]);
233:
234:             Console.WriteLine("Data[" + itemData[0] + "][" + itemData[1] + "][" + itemData[2] + "]);
235:
236:         }
237:     }
238: }
```

DB のロールバック

レコードの追加テスト

SimpleDataAccess クラス

```
239: using System;
240: using System.Data.SqlClient;
241:
242: using Xpjug.DBTest.Core;
243: using Xpjug.DBTest.TestBase;
244:
245: namespace Xpjug.DBTest.Test
246: {
247:     /// <summary>
248:     /// SimpleDataAccess の概要の説明です。
249:     /// </summary>
250:     public class SimpleDataAccess : SqlClientWorker
251:     {
252:         private string connectionString;
253:         public SimpleDataAccess( string connectionString )
254:         {
255:             this.connectionString = connectionString;
256:         }
257:
258:         private const string insertCommandText = "insert into [TestData] (Name,Address) values ( @Name, @Address )";
259:
260:         public void Add( string name, string address )
261:         {
262:             SqlConnection con;
263:             SqlTransaction tr;
264:
265:             con = GetConnection( connectionString );
266:             tr = con.BeginTransaction();
267:
268:             SqlCommand command = new SqlCommand( insertCommandText, con, tr );
269:             SqlParameter param;
270:
271:             param = command.Parameters.Add( "@Name", System.Data.SqlDbType.VarChar );
272:             param.Direction = System.Data.ParameterDirection.Input;
273:             param.Value = name;
274:
275:             param = command.Parameters.Add( "@Address", System.Data.SqlDbType.VarChar );
276:             param.Direction = System.Data.ParameterDirection.Input;
277:             param.Value = address;
278:
279:             command.ExecuteNonQuery();
280:
281:             tr.Commit();
282:
283:             Close();
284:         }
285:     }
286: }
```

補章 参考文献等

参考文献

書籍

eXtreme Programming テスト技法 xUnit ではじめる実践 XP プログラミング
日本 XP ユーザグループ 著、監修 長瀬嘉秀、翔泳社、2001
テスト駆動開発入門
ケント・ベッグ 著、ピュアソン・エデュケーション、2003

DB テストの簡易フレームワーク(全ソース)

SqlClientWorker クラス

```
287: using System;
288: using System.Data.SqlClient;
289:
290: namespace Xpjug.DBTest.Core
291: {
292:     /// <summary>
293:     /// SqlClientWorker の概要の説明です。
294:     /// </summary>
295:     public class SqlClientWorker
296:     {
297:         private IContextHandler contextHandler;
298:         public SqlClientWorker()
299:         {
300:             contextHandler = new SqlContextHandler();
301:         }
302:         public void SetContextHandler(IContextHandler newContextHandler )
303:         {
304:             contextHandler = newContextHandler;
305:         }
306:         public SqlConnection GetConnection( string connectionString )
307:         {
308:             return contextHandler.GetConnection( connectionString );
309:         }
310:         public void Close()
311:         {
312:             contextHandler.Close();
313:         }
314:     }
315: }
```

IContextHandler クラス

```
316: using System;
317: using System.Data.SqlClient;
318: namespace Xpjug.DBTest.Core
319: {
320:     /// <summary>
321:     /// IContextHandler の概要の説明です。
322:     /// </summary>
323:     public interface IContextHandler
324:     {
325:         SqlConnection GetConnection( string connectionString );
326:         void Close();
327:     }
328: }
```

SqlContextHandler クラス

```
329: using System;
330: using System.Data.SqlClient;
331:
332: namespace Xpjug.DBTest.Core
333: {
334:     /// <summary>
335:     /// SqlContextHandler の概要の説明です。
336:     /// </summary>
337:     public class SqlContextHandler : IContextHandler
338:     {
339:         private SqlConnection currentConnection;
340:
341:         public SqlConnection GetConnection( string connectionString )
342:         {
343:             currentConnection = new SqlConnection( connectionString );
344:             currentConnection.Open();
345:             return currentConnection;
346:         }
347:
348:         public void Close()
349:         {
350:             if( currentConnection != null )
351:             {
352:                 if( currentConnection.State == System.Data.ConnectionState.Open )
353:                 {
354:                     currentConnection.Close();
355:                 }
356:             }
357:         }
358:     }
359: }
```

SqlContextTestHandler クラス

```
360: using System;
361: using System.Data.SqlClient;
362: using Xpjug.DBTest.Core;
363:
364: namespace Xpjug.DBTest.TestBase
365: {
366:     /// <summary>
367:     /// SqlContextTestHandler の概要の説明です。
368:     /// </summary>
369:     public class SqlContextTestHandler : IContextHandler
370:     {
371:         private SqlConnection currentConnection;
372:
373:         public SqlConnection CurrentConnection
374:         {
375:             get { return currentConnection; }
376:         }
377:
378:         public SqlConnection GetConnection( string connectionString )
379:         {
380:             if( currentConnection == null )
381:             {
382:                 SqlCommand cmd;
383:
384:                 currentConnection = new SqlConnection( connectionString );
385:                 cmd = new SqlCommand( "BEGIN TRANSACTION", currentConnection );
386:                 currentConnection.Open();
387:                 cmd.ExecuteNonQuery();
388:             }
389:
390:             return currentConnection;
391:         }
392:
393:         public void Close()
394:         {
395:             //何もしない
396:         }
397: }
```

```

398:     public void RealRoolback()
399:     {
400:         SqlCommand cmd;
401:
402:         cmd = new SqlCommand( "ROLLBACK", currentConnection );
403:         cmd.ExecuteNonQuery();
404:         currentConnection.Close();
405:         currentConnection = null;
406:     }
407: }
408:
409: }

```

SqlClientTestCase クラス

```

410: using System;
411: using System.Data.SqlClient;
412:
413: using Xpjug.DBTest.Core;
414:
415: namespace Xpjug.DBTest.TestBase
416: {
417:     /// <summary>
418:     /// SqlClientTestCase の概要の説明です。
419:     /// </summary>
420:     public class SqlClientTestCase
421:     {
422:         public SqlClientTestCase()
423:         {
424:             //
425:             // TODO: コンストラクタ ロジックをここに追加してください。
426:             //
427:         }
428:         private SqlClientWorker target;
429:         private SqlContextTestHandler sqlContextHandler;
430:
431:         public void SetupContextHandler( SqlClientWorker newTarget,
432:                                         string connectionString )
433:         {
434:             target = newTarget;
435:             sqlContextHandler = new SqlContextTestHandler();
436:             target.SetContextHandler( sqlContextHandler );
437:             target.GetConnection( connectionString );
438:         }
439:
440:         public void TearDownContextHandler()
441:         {
442:             sqlContextHandler.RealRoolback();
443:         }
444:
445:         public int GetRecordCount( string tableName )
446:         {
447:             SqlCommand cmd;
448:
449:             cmd = new SqlCommand( "Select count(*) From " + tableName,
450:                                 sqlContextHandler.CurrentConnection );
451:
452:             return (int)cmd.ExecuteScalar();
453:         }
454:
455:         public string[] ReadOneRecord( string selectString )
456:         {
457:             string[] result;
458:
459:             SqlCommand cmd;
460:             SqlDataReader dataReader;
461:             int i;
462:
463:             result = null;
464:             dataReader = null;
465:
466:             cmd = new SqlCommand( selectString, sqlContextHandler.CurrentConnection );
467:
468:             try

```

```

469:         {
470:             dataReader = cmd.ExecuteReader();
471:             dataReader.Read();
472:
473:             result = new string[dataReader.FieldCount];
474:
475:             for( i = 0; i <= (dataReader.FieldCount - 1); ++i )
476:             {
477:                 result[i] = dataReader[i].ToString();
478:             }
479:         }
480:         catch( Exception )
481:         {
482:         }
483:
484:         finally
485:         {
486:             if( dataReader != null )
487:             {
488:                 dataReader.Close();
489:             }
490:         }
491:
492:         return result;
493:     }
494:
495:     public bool IsExistRecord( string selectString)
496:     {
497:         bool result;
498:         string temp;
499:         SqlCommand cmd;
500:         SqlDataReader dataReader;
501:
502:         result = true;
503:         dataReader = null;
504:
505:         cmd = new SqlCommand( selectString, sqlContextHandler.CurrentConnection );
506:
507:         try
508:         {
509:             dataReader = cmd.ExecuteReader();
510:             dataReader.Read();
511:
512:             temp = dataReader[0].ToString();
513:         }
514:         catch( Exception )
515:         {
516:             result = false;
517:         }
518:
519:         finally
520:         {
521:             if( dataReader != null )
522:             {
523:                 if( dataReader.IsClosed == false )
524:                 {
525:                     dataReader.Close();
526:                 }
527:             }
528:         }
529:
530:         return result;
531:     }
532: }
533: }

```
