VERSION 8.2

カスタマイズ・ガイド

MERCURY™

Mercury WinRunner

カスタマイズ・ガイド

Version 8.2

MERCURY™

Mercury WinRunner カスタマイズ • ガイド，Version 8.2

本マニュアル，付属するソフトウェアおよびその他の文書の著作権は，米国および国際著作権法によって保護されており，それらに付随する使用契約書の内容に則する範囲内で使用できます。Mercury Interactive Corporation のソフトウェア，その他の製品およびサービスの機能は次の 1 つまたはそれ以上の特許に記述があります。米国特許番号 5,511,185; 5,657,438; 5,701,139; 5,870,559; 5,958,008; 5,974,572; 6,137,782; 6,138,157; 6,144,962; 6,205,122; 6,237,006; 6,341,310; 6,360,332, 6,449,739; 6,470,383; 6,477,483; 6,549,944; 6,560,564; 6,564,342; 6,587,969; 6,631,408; 6,631,411; 6,633,912; 6,694,288; 6,738,813; 6,738,933; 6,754,701; 6,792,460 および 6,810,494。オーストラリア特許番号 763468 および 762554。その他の特許は米国およびその他の国で申請中です。権利はすべて弊社に帰属します。

Mercury，Mercury Interactive，Mercury ロゴ，Mercury Interactive ロゴ，LoadRunner，WinRunner，SiteScope および TestDirector は，Mercury Interactive Corporation の商標であり，特定の司法管轄内において登録されている場合があります。上記の一覧に含まれていない商標についても，Mercury が当該商標の知的所有権を放棄するものではありません。

その他の企業名，ブランド名，製品名の商標および登録商標は，各所有者に帰属します。Mercury は，どの商標がどの企業または組織の所有に属するかを明記する責任を負いません。

Mercury Interactive Corporation
379 North Whisman Road
Mountain View, CA 94043
Tel: (650) 603-5200
Toll Free: (800) TEST-911
Customer Support: (877) TEST-HLP
Fax: (650) 603-5300

© 1993-2005 Mercury Interactive Corporation, All rights reserved

本書に関するご意見，ご要望は documentation@mercury.com まで電子メールにてお送りください。

WRCG8.2JP/01

目次

WinRunner のカスタマイズへようこそ v
本書の使用方法 v
WinRunner の関連マニュアル vi
オンライン・リソース vi
表記規則 vii

第 1 部 : GUI 検査のカスタマイズ

第 2 章 : はじめに 3

第 3 章 : 標準オブジェクトを対象とするユーザ定義 GUI 検査の作成.....9
標準オブジェクトを対象とするユーザ定義 GUI 検査の作成について 9
キャプチャ関数の作成 12
比較関数の作成 15
新しいプロパティ検査の記録 18
新しいプロパティ検査と GUI オブジェクト・クラスの関連付け...........19
GUI オブジェクト・クラスの標準の検査の変更 20

第 4 章 : ユーザ定義オブジェクトを対象とする GUI 検査の作成..........23
ユーザ定義オブジェクトを対象とする GUI 検査の作成について...........23
検証のためのユーザ定義の GUI オブジェクト・クラスの追加 25
ユーザ定義の GUI オブジェクト・クラスを対象とするユーザ定義
検査の定義 28

第 5 章 : GUI 検査の作成 : 応用編 31
GUI 検査の作成の 応用手順について 31
検証対象の新しい GUI オブジェクト・クラスの追加............................32
キャプチャ関数と比較関数の作成 33
新しい検査の登録 34
標準の検査の設定 35
高度な GUI 検査の実装..35

第 6 部 : 記録方法のカスタマイズ

第 7 章 : 記録されるステートメントのカスタマイズ 43
記録されるステートメントのカスタマイズについて 43
ユーザ定義記録関数について 44
ユーザ定義記録関数の開発..47
ユーザ定義記録関数の GUI オブジェクト・クラスへの関連付け...........51
ユーザ定義実行関数の開発..51
ユーザ定義記録関数の例 52

第 8 章 : GUI オブジェクトへのユーザ定義プロパティの追加 59
GUI オブジェクトへのユーザ定義プロパティの追加について 59
ユーザ定義プロパティに対するクエリー関数の開発 60
ユーザ定義プロパティ用の検証関数の開発 61
ユーザ定義プロパティの登録 63
ユーザ定義プロパティの GUI オブジェクト・クラスへの割り当て 65
ユーザ定義プロパティ関数の例 67

第 9 章 : 割り当てられる論理名のカスタマイズ 71
割り当てられる論理名のカスタマイズについて 71
論理名関数について 73
論理名関数の開発 74
論理名関数のユーザ定義 GUI オブジェクト・クラスへの関連付け 74

第 10 部 : WINRUNNER API の使い方

第 11 章 :Mercury の API 関数 79
API 関数について 79

索引 87

WinRunner のカスタマイズへようこそ

WinRunner のカスタマイズへようこそ。WinRunner のさまざまな側面をカスタマイズして，WinRunner の機能を拡張し，テスト要件を満たします。

本書の使用方法

本書では，WinRunner のカスタマイズの背景にある主要概念を説明します。本書では，アプリケーションのテスト要件を確実に満たし，WinRunner を最大限に活用するための手順を段階的に提供します。

本書は次の部で構成されています。

第 1 部　GUI 検査のカスタマイズ

ユーザ定義の GUI オブジェクトを対象としたユーザ定義の検査の開発と実装方法を説明します。

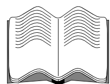
第 2 部　記録方法のカスタマイズ

テスト・スクリプトが読みやすくなるよう，ユーザ定義オブジェクトでの操作を WinRunner で記録する方法のカスタマイズの仕方について説明します。

第 3 部　WinRunner API の使い方

WinRunner の API 関数を使用して，ユーザ・インタフェースのレベルでは見えないアプリケーションの機能やユーザ・インタフェースを持たないアプリケーションをテストする方法を説明します。

WinRunner の関連マニュアル



本書のほかにも，WinRunner には次の印刷マニュアルが付属しています。

『**WinRunner インストール・ガイド**』: 単独のコンピュータまたはネットワークに接続されているコンピュータに WinRunner をインストールする方法を説明します。

『**WinRunner チュートリアル**』: WinRunner の基本的な使い方，およびアプリケーションのテストを開始する方法について説明します。

『**WinRunner 基本機能ユーザーズ・ガイド**』: アプリケーションを対象にテストを作成し実行するための WinRunner の最も一般的な機能について説明します。

『**WinRunner 上級機能ユーザーズ・ガイド**』: アプリケーションの特別なテスト要件を満たすための WinRunner の特別な機能の使用方法を説明します。

『**TSL リファレンス・ガイド**』: TSL（テスト・スクリプト言語）および TSL に含まれる関数を説明します。

オンライン・リソース

WinRunner には次のオンライン・リソースがあります。

最初にお読みください：WinRunner に関する最新ニュースと情報を提供します。

WinRunner の新機能：最新バージョンの WinRunner の新機能を説明します。

印刷用ドキュメント：PDF 形式の全マニュアルへのリンクが表示されます。印刷用ドキュメントは Adobe Acrobat Reader を使って表示および印刷できます。Adobe Acrobat Reader の最新バージョンは，**www.adobe.co.jp** からダウンロードできます。

WinRunner コンテキスト・センシティブ・ヘルプ：WinRunner の使用中に生じた疑問をすぐに解決できます。メニュー・コマンドとダイアログ・ボックスについて説明し，WinRunner のタスクを実行する方法を示します。Mercury カスタマー・サポート Web サイトで WinRunner ヘルプ・ファイルの最新版をご確認ください。

TSL リファレンス・ヘルプ：各関数に関する追加情報と使用例を示します。**TSL リファレンス・ヘルプ**は，［スタート］メニューの［WinRunner］グループ

または WinRunner の［ヘルプ］メニューから開くことができます。特定の関数のオンライン・リソースを開くには，コンテキスト・センシティブ・ヘルプ・ボタンをクリックして，テスト・スクリプト内の TSL ステートメントをクリックするか，テスト・スクリプト内の TSL ステートメント上にカーソルを置き，F1 キーを押します。

WinRunner サンプル・テスト：ユーティリティとサンプル・テストが説明付きで含まれています。

オンライン・カスタマー・サポート：普段お使いの Web ブラウザで，Mercury のカスタマー・サポート Web サイトを開きます。

Mercury Web サイト：普段お使いの Web ブラウザで，Mercury のホームページを開きます。このサイトでは，Mercury とその製品，およびサービスに関する最新情報を提供します。新しいソフトウェアのリリース，セミナー，展示会，カスタマー・サポート，トレーニングなどの情報も含まれています。

表記規則

本書では次の表記規則に従います。

1，2，3	太字の数字は，操作手順を示します。
＞	大なり記号はメニュー・レベルを区切ります（例：［**ファイル**］＞［**開く**］）。
［**太字**］	全角の大括弧に太字は，インターフェイスの要素の名前（例：［**実行**］ボタン）やその他の強調する項目を示します。
太字	**太字**のテキストは，メソッド名または関数名を示します。また，メソッドまたは関数の引数，構文の記述中のファイル名，書名を示します。新しい用語を紹介する場合にも使用します。
＜＞	山括弧は，ユーザの環境次第で変わる可能性のある，ファイル・パスや URL アドレスの一部分を囲みます（例：**＜製品のインストール・フォルダ＞¥bin**）。
Arial	Arial のフォントはそのまま入力する例やテキストに使用されます。
Arial bold	**Arial bold** のフォントはそのまま入力しなければならない構文記述のテキストに使用されます。

SMALL CAPS	SMALL CAPS のフォントは，キーボードのキーを示します。
...	構文内の 3 つの点は，同じ形式で項目をさらに含めることができることを意味します。プログラム例での 3 つの点は，プログラム行が意図的に削除されていることを示します。
[ ]	半角の大括弧は，省略可能な引数を囲みます。
|	2 つの値のうちの 1 つを選択しなければならない場合，これらの値を垂直バーで区切ります。

第 1 部

GUI 検査のカスタマイズ

第 1 章

はじめに

WinRunner は，ソフトウェアをテストするときに広範囲で使用できる一連の機能を提供します。WinRunner をカスタマイズしてこれらの機能を拡張し，アプリケーションの特定の条件を満たせます。本書では，WinRunner をカスタマイズして，テスト機能を拡張する方法について詳しい情報を提供します。

WinRunner では，以下の部分をカスタマイズできます。

➤ **GUI 検査**

WinRunner の標準の GUI 検査で，テスト環境の特殊な条件を完全に満たせない場合，ユーザ定義のプロパティ検査を作成して検証機能を拡張できます。

➤ **テスト・スクリプトの記録**

アプリケーションにユーザ定義のオブジェクトが含まれている場合，WinRunner がテスト・スクリプトに記録する関数をカスタマイズして，テスト・スクリプトを読みやすく分かりやすいものにできます。

➤ **Mercury API の使い方**

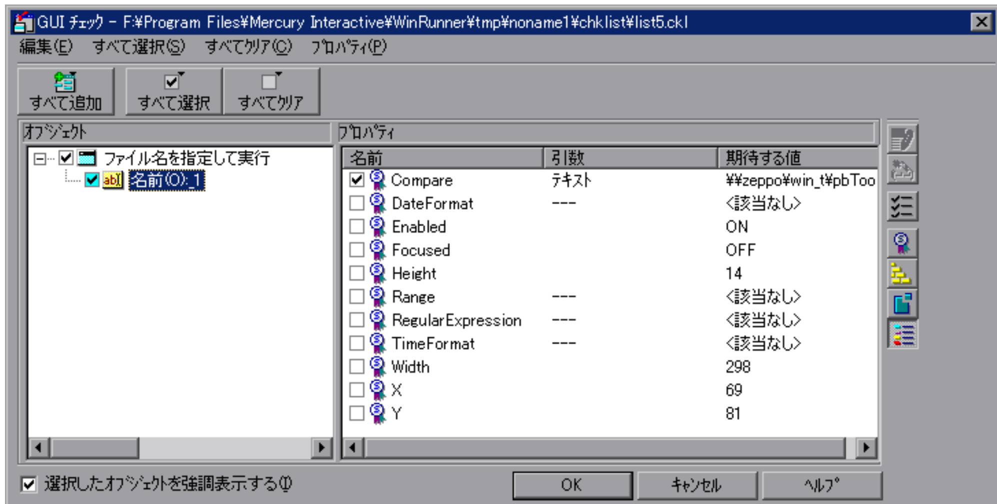
Mercury API（Application Programming Interface）を使用して，ユーザ・インタフェースと関連していないアプリケーションの関数を記録したり実行したりできます。

GUI プロパティ検査のカスタマイズ

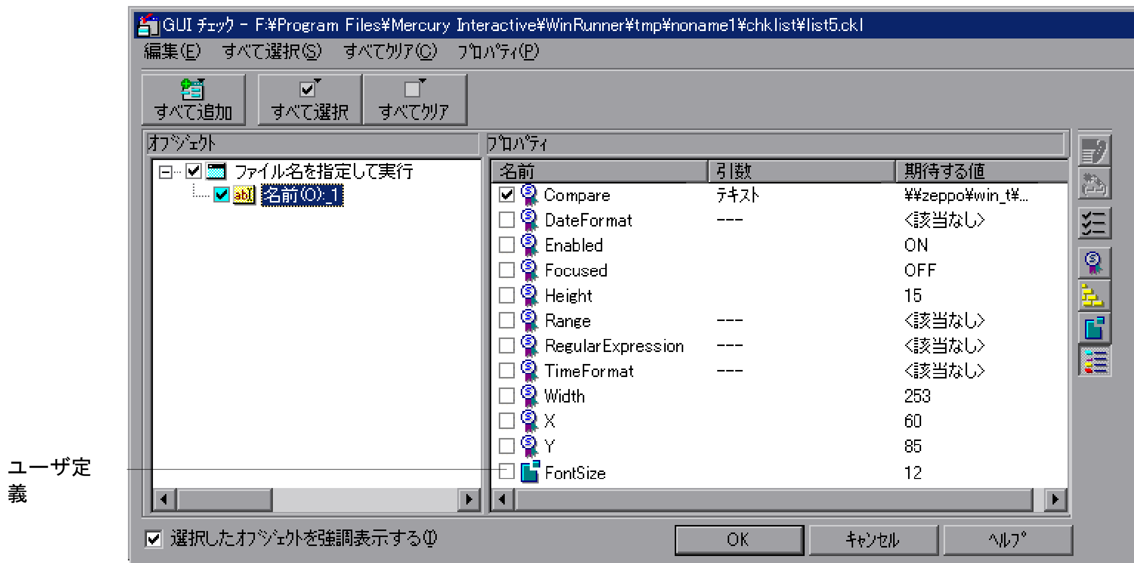
WinRunner は，アプリケーション内の GUI オブジェクトを検査するためのさまざまなプロパティ検査を提供します。WinRunner の標準のプロパティ検査で，テスト環境の特定の要件を完全に満たせない場合，ユーザ定義の GUI プロパティ検査を作成しテスト機能を拡張できます。WinRunner を使用すれば，GUI プロパティ検査をいくつかの方法でカスタマイズできます。

➤ 第 2 章「標準オブジェクトを対象とするユーザ定義 GUI 検査の作成」では，ユーザ定義プロパティ検査を開発し，標準の GUI オブジェクトで実行する方法を説明します。例えば，エディタで使われるフォントのサイズを対象とするプロパティ検査を開発できます。新しいプロパティ検査と標準の編集クラスを関連付けると，編集クラスのオブジェクトのチェックポイントを作成または編集するときに，必ずそのプロパティ検査は［GUI チェック］ダイアログ・ボックスに表示されます。

次の［GUI チェック］ダイアログ・ボックスでは，編集クラス・オブジェクトの標準のプロパティ検査を表示しています。



以下の［GUI チェック］ダイアログ・ボックスでは，編集クラス・オブジェクトの標準のプロパティ検査とユーザ定義のプロパティ検査を表示しています。



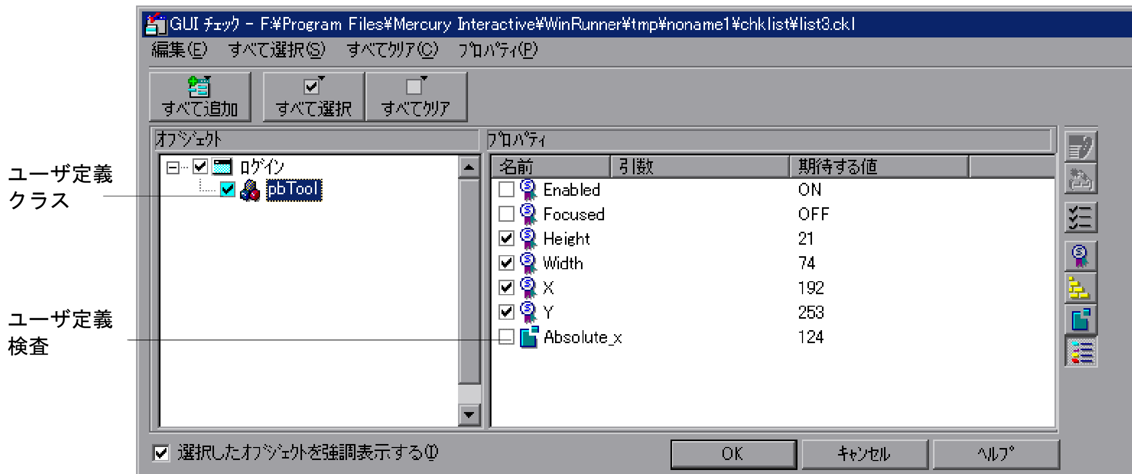
➤ 第 3 章「ユーザ定義オブジェクトを対象とする GUI 検査の作成」では，WinRunner のどの標準クラスにも属さない GUI オブジェクトがアプリケーションに含まれている場合にはどうすべきか説明します。これらのオブジェクトを対象とするユーザ定義検証クラスを作成できます。そして，これらのユーザ定義クラスのオブジェクトを検査するときに使用するプロパティ検査を指定できます。

例えば，検証用に「pbTool」というユーザ定義クラスを作成したとします。pbTool クラス・オブジェクトを対象とした GUI チェックポイントを作成すると，使用できるプロパティ検査が，［GUI チェック］ダイアログ・ボックスか［GUI チェックポイント作成］ダイアログ・ボックスのどちらか一方の［プロパティ］表示枠に表示されます。また，カスタマイズしたプロパティ検査をこの新しいユーザ定義クラスに追加できます。カスタマイズしたこれらのプロパティ検査は，そのクラスのオブジェクトを対象とする GUI チェックポイントを作成または編集するときは，必ず［GUI チェック］ダイアログ・ボックスに表示されます。

以下の［GUI チェック］ダイアログ・ボックスでは，ユーザ定義クラスに属するオブジェクトの標準のプロパティ検査を表示します。ユーザ定義オブジェクトとは，WinRunner が使用する標準のクラスに属さない任意のオブジェクトです。

これらのオブジェクトは，一般的な「オブジェクト」クラスに割り当てられます。このクラスには次の検査が含まれます。

ユーザ定義検査もユーザ定義クラスに追加できます。



- 第 4 章「GUI 検査の作成：応用編」では，［GUI チェック］ダイアログ・ボックスの独自のユーザ・インタフェースを作成する方法を説明します。［GUI チェックポイント］ダイアログ・ボックスは，ユーザ定義オブジェクト・クラスに属する GUI オブジェクトを対象とする検査を作成すると開きます。この章では，ユーザ定義ユーティリティを実装してユーザ定義検査の結果を表示する方法も説明します。

記録方法のカスタマイズ

標準の GUI オブジェクトで操作を記録すると，WinRunner は，読みやすく分かりやすいテスト・スクリプトを作成します。ただし，WinRunner の標準オブジェクトの振る舞いとは大きく違う，ユーザ定義の GUI オブジェクトで操作すると，結果としてテスト・スクリプトには汎用の **obj_** TSL ステートメントが記録されます。

アプリケーションにユーザ定義オブジェクトが含まれている場合，WinRunner によってスクリプトに記録される関数をカスタマイズして，テスト・スクリプトを読みやすく分かりやすいものにできます。WinRunner を使用して，記録したステートメントを以下の 3 つの方法でカスタマイズできます。

- 第 5 章「記録されるステートメントのカスタマイズ」では，ユーザ定義 GUI オブジェクトを対象に操作を行ったときに，WinRunner がテスト・スクリプトに記録する関数の呼び出しを指定する方法を説明します。
- 第 6 章「GUI オブジェクトへのユーザ定義プロパティの追加」では，任意の GUI オブジェクト・クラスに自身のプロパティを追加して，アプリケーション内で GUI オブジェクトを一意に特定する WinRunner のメカニズムを向上させる方法を説明します。
- 第 7 章「割り当てられる論理名のカスタマイズ」では，WinRunner がアプリケーションのユーザ定義オブジェクトに論理名を割り当てる手順をカスタマイズする方法を説明します。このカスタマイズを行えば，記録されたステートメントが参照している GUI オブジェクトがすぐに分かります。

WinRunner API の使い方

第 8 章「Mercury の API 関数」では，テスト内で必要な Mercury API 関数をすべて説明します。

第 2 章

標準オブジェクトを対象とするユーザ定義 GUI 検査の作成

標準 GUI オブジェクトを対象に実行するユーザ定義のプロパティ検査を開発することによって，アプリケーションの GUI オブジェクトを一意に識別する WinRunner のメカニズムを強化できます。

本章では，以下の項目について説明します。

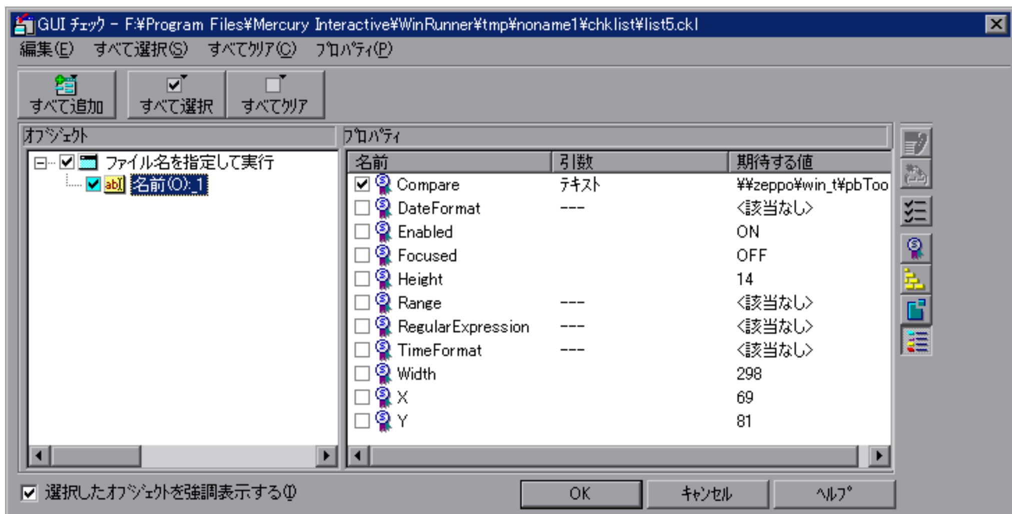
- キャプチャ関数の作成
- 比較関数の作成
- 新しいプロパティ検査の記録
- 新しいプロパティ検査と GUI オブジェクト・クラスの関連付け
- GUI オブジェクト・クラスの標準の検査の変更

標準オブジェクトを対象とするユーザ定義 GUI 検査の作成について

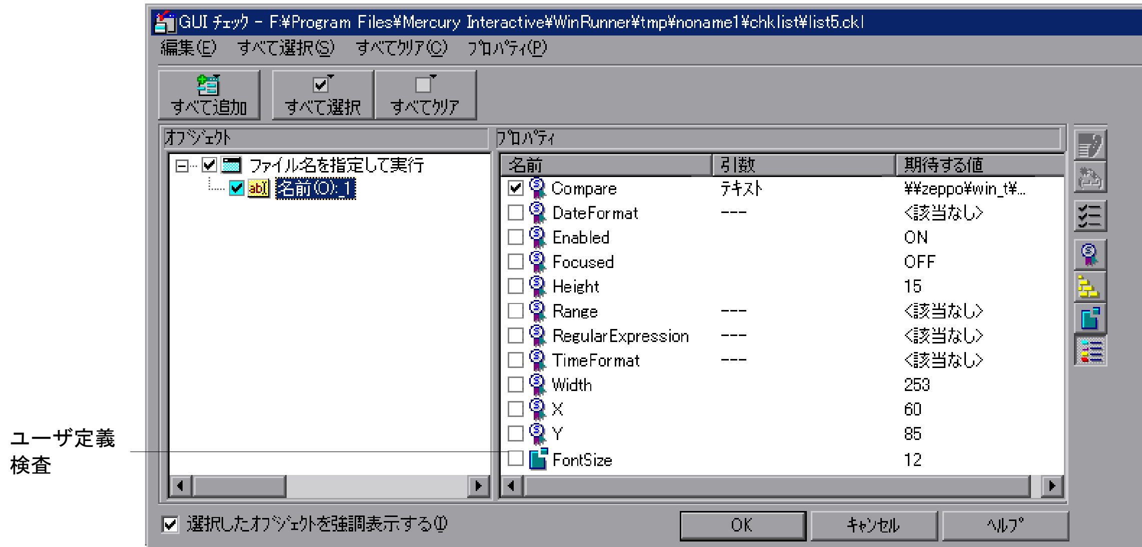
GUI チェックポイントをテスト・スクリプトに挿入することにより，WinRunner でアプリケーションの状態を検査する方法とタイミングを指定します。チェックポイントを挿入する操作の一環として，WinRunner が検査するオブジェクト・プロパティを定義します。WinRunner には，GUI オブジェクト・クラスごとに，選択可能な一連の標準のプロパティ検査があります。標準のプロパティ検査でテスト要件を満たせない場合，独自の「**ユーザ定義**」のプロパティ検査を開発できます。検査するオブジェクトのクラスの［GUI チェック］ダイアログ・ボックスにユーザ定義のプロパティ検査を追加します。GUI オブジェクトを対象とする標準の検査と，［GUI チェック］ダイアログ・ボックスの詳細については『**WinRunner ユーザーズ・ガイド**』の「GUI オブジェクトの検査」の章を参照してください。

例えば，編集ボックスで使われるフォントのサイズを検査するとします。プロパティ検査を開発して，このプロパティを検査できます。新しい検査と標準の編集クラスを関連付けると，「オブジェクト」表示枠で編集オブジェクトが強調表示され，「プロパティ」表示枠にはカスタマイズしたフォント・サイズ・プロパティとともに編集クラス・オブジェクトの標準プロパティが表示されます。

以下の［GUI チェック］ダイアログ・ボックスには，編集クラスの標準のプロパティ検査が表示されています。



以下の［GUI チェック］ダイアログ・ボックスには，ユーザ定義のフォント・サイズ・プロパティ検査と編集クラスの標準のプロパティ検査の両方が表示されています。



標準の WinRunner の GUI オブジェクト・クラスの［GUI チェック］ダイアログ・ボックスにユーザ定義プロパティを追加するには，次の手順を実行します。

1. ユーザ定義のプロパティの期待結果と実際の結果をキャプチャする関数を作成します。
2. 期待結果と実際の結果を比較する関数を作成します。
3. プロパティを登録します。
4. 登録したプロパティと標準の GUI オブジェクト・クラスを関連付けます。
5. GUI オブジェクト・クラスの標準のプロパティとして新しいプロパティを設定します（オプション）。

WinRunner の関数ジェネレータを使用して，必要なすべての関数呼び出しを生成し，それらを直接テスト・スクリプトに挿入できます。関数は，関数ジェネレータの「GUI verification」カテゴリにあります。関数の自動生成と自動挿入の詳細については，『**WinRunner ユーザーズ・ガイド**』の「関数の生成」の章を参照してください。

キャプチャ関数と比較関数は，使う前にコンパイルしなければなりません。テスト・スクリプトから関数を実行してもコンパイルできますが，コンパイル済みのモジュールにこれらの関数を入れておいて，起動テストからこのモジュールをロードすることをお勧めします。このようにすれば，WinRunner のすべてのセッションでこれらの関数を使用できます。コンパイル済みのモジュールの詳細については，『**WinRunner ユーザーズ・ガイド**』の「コンパイル済みモジュールの作成」を参照してください。

この章で説明したように, 標準の WinRunner GUI オブジェクト・クラスの［GUI チェック］ダイアログ・ボックスにプロパティを追加できます。また，アプリケーションに，WinRunner のどの標準クラスにも属さない GUI オブジェクトがある場合，それを検証する新しいユーザ定義の GUI オブジェクト・クラスを定義できます。次のようにします。

- 新しいクラスで選択できるプロパティを指定します。詳細については，第 3 章「ユーザ定義オブジェクトを対象とする GUI 検査の作成」を参照してください。
- 新しいクラスのユーザ定義のユーザ・インタフェースとユーザ定義の結果表示ユーティリティを使用して，［GUI チェック］ダイアログ・ボックスをカスタマイズします。詳細については，第 4 章「GUI 検査の作成：応用編」を参照してください。

標準の WinRunner クラスを対象とする新しいプロパティ検査を追加できるほかに，標準で検査されるクラスのプロパティ群に新しいプロパティを追加したり，検査対象プロパティを変更したりできます。例えば，プッシュ・ボタンを対象とする標準の検査は，標準でプッシュ・ボタンが有効になっているかどうかのみを検査します。push_button クラスのほかの標準 / ユーザ定義のプロパティも，標準で検査するように指定できます。例えば，ボタンのラベルを標準の検査として含めることができます。

キャプチャ関数の作成

キャプチャ関数を作成し，ユーザ定義検査の期待結果を生成し，実際の結果を保存できます。例えば，編集ボックスで使われるフォントのサイズを対象とした検査を開発すると，フォント・サイズを実際に判別して記録するのはキャプチャ関数です。

キャプチャ関数の構文は次のとおりです。

public **capture_function_name (** in *object,* inout *value* [, in *arg_list*] **)**

- *capture_function_name* は，キャプチャ関数の名前です。
- *object* は，検査対象 GUI オブジェクトの記述が代入される *in* パラメータです。
- *value* は *inout* パラメータです。
  - キャプチャ関数の結果が数値または文字列の場合，キャプチャ関数は，関数の結果を *value* パラメータに代入します。例えば，フォント・サイズ 10 を *value* パラメータに代入します。
  - キャプチャ関数の結果が長すぎる（2K バイト以上）またはテキスト形式ではない場合，キャプチャ関数は，結果をファイルに格納しなければなりません。WinRunner が *value* パラメータに代入するファイル名には一意の名前を使います。
- arg_list は，ui_function パラメータから渡されるオプションの in パラメータです。ui_function パラメータは，ユーザ定義のユーザ・インタフェースを持つ［GUI チェックポイント］ダイアログ・ボックスを作成するために **gui_ver_add_class** 関数を使う場合のみ使用します。詳細については，第 4 章「GUI 検査の作成：応用編」を参照してください。

すべてのテストでキャプチャ関数を使えるように，関数を *public* として宣言します。*capture_function_name* をキャプチャ関数の名前と置き換えます。ユーザ定義関数の詳細については，『**WinRunner ユーザーズ・ガイド**』の「ユーザ定義関数の作成」を参照してください。

使用例 1：オブジェクトの X 軸の絶対座標の検査

次の例に示すユーザ定義のキャプチャ関数 *get_abs_x* は，画面の原点からの GUI オブジェクト左上角の x 座標を返します。TSL 関数 **obj_get_info** を呼び出して，オブジェクトの画面座標 abs_x を特定します。

```
public function get_abs_x (in object, inout value)
{
    return (obj_get_info (object, "abs_x", value));
}
```

次の例では，キャプチャ関数の結果をファイルに格納する方法を紹介します。

```
public function get_abs_x(in object, inout file)
{
    auto x_coord, rc;
    rc=obj_get_info(object, "abs_x", x_coord);
```

```
    file_open(file,FO_MODE_WRITE);
    file_printf(file,"%s",x_coord);
    file_close(file);
    return(rc);
}
```

使用例 2：テキストの色の検査

次の例では，編集フィールドのテキストの色を検証します。edit_get_text_color キャプチャ関数は，*GetDC* という Windows API 関数を使用して，編集フィールドのデバイス・コンテキストを取得します。*GetPixel* 関数を使用して，まず編集ボックスの背景の色を見つけ，次に前景の色を見つけます。API 関数の使い方の詳細については，『**WinRunner ユーザーズ・ガイド**』の「外部ライブラリからの関数の呼び出し」を参照してください。

```
# Windows API 宣言をロードする。
load("c:¥¥wrun¥¥lib¥¥win_api",1,1);

# キャプチャ関数
public function edit_get_text_color(in obj_name, inout rgb_val)
{
    auto hWnd, hDc, ret, i, w, h, backcolor, rc;

    # 編集フィールドの幅と高さを取得する。
    rc=obj_get_info(obj_name, "handle", hWnd);
        if(rc != E_OK)
            return(rc);
    rc=obj_get_info(obj_name,"height",h);
        if(rc != E_OK)
            return(rc);
    rc=obj_get_info(obj_name,"width",w);
        if(rc != E_OK)
            return(rc);

    # 編集フィールドのデバイス・コンテキストを取得する。
    hDc=GetDC(hWnd);

    # 背景の色を見つける。
backcolor=GetPixel(hDc,2,2);

    # 編集フィールドをスキャンして前景の色を見つける。
    for (i=1; i<w;i++)
```

```
    {       ret=GetPixel(hDc,int(h/2),i);
        if((ret != backcolor)|| (ret==0)) break;
    }

    # デバイス・コンテキストを解放する。
    ReleaseDC(hWnd, hDc);
    rgb_val=ret;
    return(E_OK);
}
```

比較関数の作成

キャプチャ関数がユーザ定義検査の期待結果と実際の結果を特定し終えると，WinRunner は，結果を検証して検査が成功したかどうかを判定します。検査を検証するには，WinRunner の標準の比較関数 **default_compare_func** を使用するか，独自の比較関数を作成します。

標準の比較関数の使い方

default_compare_func 関数は，期待結果と実際の結果を比較します。**default_compare_func** 関数は，結果が一致する場合は 0，一致しない場合は 1 を返します。関数は，形式（数字または文字列）を基準に結果を比較します。

default_compare_func 関数は，期待結果と実際の結果の単純な比較の場合には問題なく動作します。例えば，オブジェクトの x 軸の絶対座標を検査する場合に **default_compare_func** 関数を使用できます。期待座標が 250，実際の座標が 200 の場合，**default_compare_func** 関数は 1 を返し，不一致があったことを示します。期待座標と実際の座標が両方とも 250 の場合，**default_compare_func** 関数は 0 を返し，検査が成功したことを示します。

独自の比較関数の作成

検査の成功の有無を判定するために複雑な比較が必要な場合，独自の比較関数を開発しなければなりません。例えば，8 March 1996 や 08/03/1996 などのように異なった書式の日付を比較するとします。この場合，**default_compare_func** 関数は使えないため，独自の比較関数を開発しなければなりません。

比較関数の 構文は次のとおりです。

```
public comparison_function_name ( in expected, in actual [, in arg_list] [, inout
display_information] )
{
    ...
    return(Mercury_error_code);
}
```

- comparison_function_name は，比較関数の名前です。
- *expected* は，*in* パラメータです。キャプチャ関数が期待結果を *value* パラメータに代入すると，*expected* パラメータには期待結果の値が代入されます。キャプチャ関数が結果をファイルに格納すると，*expected* パラメータには結果ファイルのパスとファイル名が代入されます。
- actual は，in パラメータです。キャプチャ関数が実際の結果を value パラメータに代入すると，actual パラメータには実際の結果の値が代入されます。キャプチャ関数が結果をファイルに格納すると，actual パラメータには結果ファイルのパスとファイル名が代入されます
- *arg_list* は, *ui_function* パラメータから渡されるオプションの *in* パラメータです。*ui_function* パラメータは，ユーザ定義のユーザ・インタフェースを持つ［GUI チェックポイント］ダイアログ・ボックスを作成するために **gui_ver_add _class** 関数を使う場合のみ使用します。詳細については詳細については，第 4 章「GUI 検査の作成：応用編」を参照してください。
- display_information は，オプションの inout パラメータです。検査結果に独自の表示関数を指定する場合にのみ使います。WinRunner は，display_information パラメータに一意のファイル名を代入します。このファイルには，表示関数を呼び出すときに使う情報を格納できます。詳細については，第 4 章「GUI 検査の作成：応用編」を参照してください。

すべてのテストで比較関数を使えるように，関数を *public* として宣言します。*comparison_function_name* を比較関数の名前と置き換えます。ユーザ定義関数についての詳細は，『**WinRunner ユーザーズ・ガイド**』で「ユーザ定義関数の作成」の章を参照してください。

比較関数は，Mercury エラー・コードを返します。期待結果と実際の結果が一致する場合は E_OK，結果が一致しない場合は E_DIFF を返します。E_DIFF は，「GUI 検証で不一致が見つかった」ことを示すエラー・コードです。

使用例 1：簡単な比較関数

次の例に示す compare_number というユーザ定義の比較関数は，期待結果と実際の結果が同じかどうかを検査します。結果が一致する場合には if/else ステートメントを使って E_OK を返し，一致しない場合は E_DIFF を返します。このシナリオでは **default_compare_func** 関数を使うこともできます。

```
public function compare_number(in expected, in actual)
{
    if (expected==actual)
        return(E_OK);
    else
        return(E_DIFF);
}
```

使用例 2：ファイルからの結果の取得

この例では，キャプチャ関数は期待結果と実際の結果を格納しています。ファイル名は，それぞれ exp_file と act_file として比較関数に渡されます。

```
public function compare_number(in exp_file, in act_file)
{
    auto expected, actual;

    file_open(exp_file,FO_MODE_READ);
    file_getline(exp_file, expected);
    file_close(exp_file);
    file_open(act_file,FO_MODE_READ);
    file_getline(act_file, actual);
    file_close(act_file);
    if (expected==actual)
        return(0);
    else
        return(E_DIFF);
}
```

新しいプロパティ検査の記録

キャプチャ関数と比較関数を作成してコンパイルしたら，新しい検査を登録しなければなりません。TSL 関数 **gui_ver_add_check** を使用して検査を登録します。

gui_ver_add_check 関数の構文は次のとおりです。

gui_ver_add_check (*check_name, capture_function, comparison_function* [,*display_function*] [,*type*] **);**

- *check_name* は，検査の名前を定義します。名前は，スペースを含まないものでなければなりません。*check_name* は，適当な検査ダイアログ・ボックスの下部に表示されます。19 ページ「新しいプロパティ検査と GUI オブジェクト・クラスの関連付け」を参照してください。
- *capture_function* は，検査で使うために開発したキャプチャ関数の名前です。
- *comparison_function* は，**default_compare_func** 関数または検査で使うために開発した比較関数の名前です。
- *display_function* は，検査結果を表示する独自の表示ユーティリティを使えるようにするオプションのパラメータです。*display_function* パラメータは，ユーザ定義のユーザ・インタフェースを持つ［GUI チェックポイント］ダイアログ・ボックスを作成するために **gui_ver_add _class** 関数を使う場合のみ使用します。詳細については，第 4 章「GUI 検査の作成：応用編」を参照してください。
- type はオプション・パラメータです。検査対象がウィンドウの場合は（1）を，その他の GUI オブジェクトの場合は（0）を指定します。type が宣言されない場合，標準値（0）を指定したとみなされます。

次の例では，**gui_ver_add_check** 関数は，オブジェクトの x 軸の絶対座標を対象とする検査を登録します。

```
gui_ver_add_check("Absolute_x","get_abs_x","compare_number", "",0);
```

次の例では，検査を登録するときの **default_compare_func** 関数の指定方法を示します。

```
gui_ver_add_check("Absolute_x","get_abs_x","default_compare_func", "",0);
```

gui_ver_add_check 関数の詳細情報については，「**TSL オンライン・リファレンス**」を参照してください。

新しいプロパティ検査と GUI オブジェクト・クラスの関連付け

新しいプロパティ検査は，登録した後に GUI オブジェクト・クラスと関連付けなければなりません。新しいプロパティ検査とクラスを関連付けることにより，［GUI チェック］ダイアログ・ボックスで対応するクラスに対して表示される，プロパティのリストの最後に新しいプロパティ検査が追加されます。

TSL 関数 **gui_ver_add_check_to_class** を使用して，プロパティ検査とクラスを関連付けます。この関数の構文は次のとおりです。

gui_ver_add_check_to_class (*class*, *property_check_name* **);**

- *class* には，MSW_class の名前か，検査が関連付けられている標準のクラスを指定します。
- *property_check_name* には，TSL 関数 **gui_ver_add_check** を使って定義したプロパティ検査の名前を指定します。新しいプロパティ検査は，［GUI チェック］ダイアログ・ボックス内でクラスに対して表示されるプロパティのリストの最後に現れます。

各 GUI オブジェクト・クラスごとに **gui_ver_add_check_to_class** 関数を繰り返すことにより，同じプロパティ検査を複数のクラスと関連付けられます。

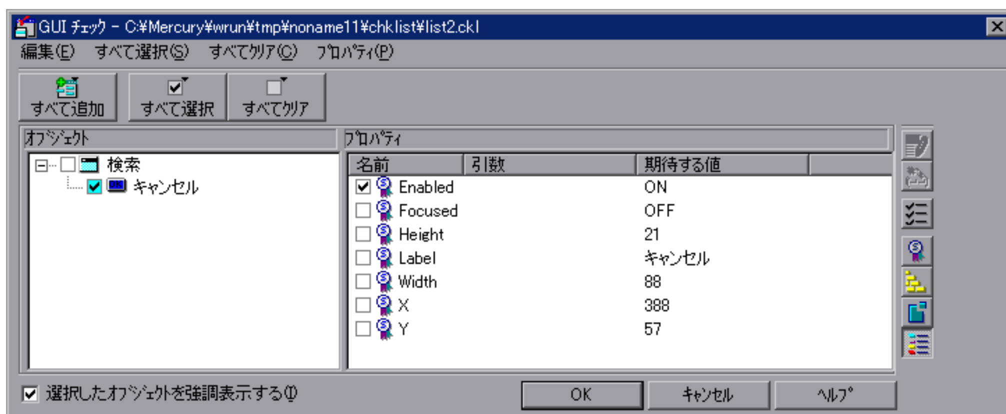
使用例

次の例では，Absolute_x 検査を push_button クラスに追加しています。

```
gui_ver_add_check_to_class ("push_button", "Absolute_x");
```

push_button パラメータは，GUI オブジェクト・クラスを示します。*Absolute_x* は，GUI オブジェクト・クラスと関連付けられているユーザ定義のプロパティ検査を示します。

以下の［GUI チェック］ダイアログ・ボックスには，push_button クラス・オブジェクトの標準の検査が表示されています。



以下の［GUI チェック］ダイアログ・ボックスには，push_button クラス・オブジェクトの標準の検査とユーザ定義検査が表示されています。

gui_ver_add_check 関数と **gui_ver_add_check_to_class** 関数の詳細については，「**TSL オンライン・リファレンス**」を参照してください。

GUI オブジェクト・クラスの標準の検査の変更

GUI オブジェクト・クラスの標準のプロパティ検査を変更できます。同様に，ユーザ定義のプロパティ検査を GUI オブジェクト・クラスの標準の検査として定義できます。例えば，WinRunner は，標準ではプッシュ・ボタンが有効かどうかだけ検査します。push_button クラスの標準の検査を変更して，ユーザ定義検査 Absolute_x を含められます。

ユーザ定義のプロパティ検査を標準の検査として定義するには，TSL 関数 **gui_ver_set_default_checks** を使います。この関数は，以前の標準プロパティ検査をすべて上書きします。この関数を使うときには，GUI オブジェクト・クラスの標準設定として設定したいプロパティ検査をすべて含めなければなりません。同じ関数を使用して，標準プロパティ検査として別の標準プロパティ検査を定義して追加できます。

gui_ver_set_default_checks 関数の構文は次のとおりです。

gui_ver_set_default_checks (*class*, *check_name*$_1$...*check_name*$_n$ **);**

➤ *class* には，MSW_class の名前か，標準の検査を設定する標準クラスの名前を指定します。

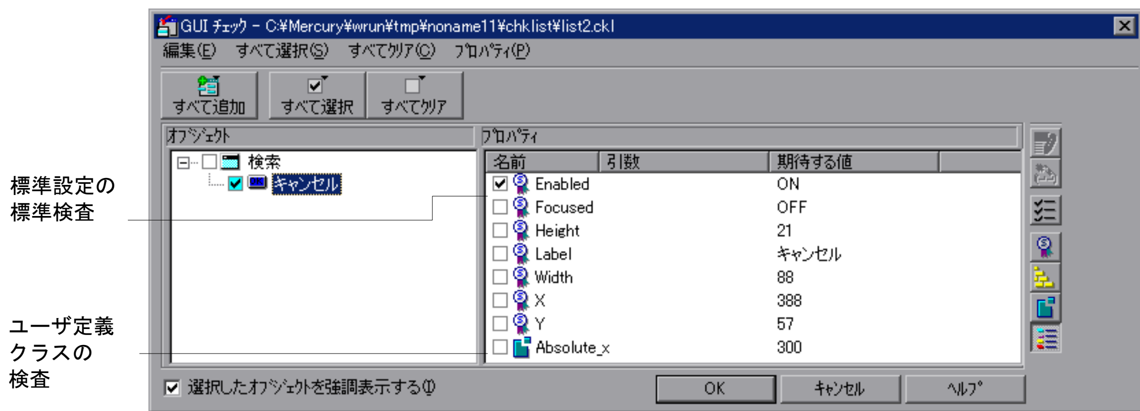
➤ *check_name*$_{1-n}$ には，標準設定として設定するプロパティ検査の名前を指定します。

使用例

次の例では，Enabled 検査と Absolute_x 検査を push_button クラスの標準の検査として設定しています。

```
gui_ver_set_default_checks ("push_button", "Enabled Absolute_x");
```

以下のダイアログ・ボックスでは，ユーザ定義の Absolute_x プロパティが追加されています。push_button クラスの標準設定の標準のプロパティ検査 Enabled が選択されています。



以下のダイアログ・ボックスでは，ユーザ定義の Absolute_x プロパティ検査と標準の Enabled 検査が標準のプロパティ検査になっています（つまり，標準で選択されるということです）。

標準プロパティ検査を複数定義する場合は，上の例のように，空白文字で標準のプロパティ検査を区切ります。

gui_ver_set_default_checks 関数の詳細については，「**TSL オンライン・リファレンス**」を参照してください。

第 3 章

ユーザ定義オブジェクトを対象とする GUI 検査の作成

検証のためのユーザ定義 GUI オブジェクト・クラスを作成し，各ユーザ定義クラスを対象に検査を開発できます。

本章では，以下の項目について説明します。

➤ 検証のためのユーザ定義の GUI オブジェクト・クラスの追加

➤ ユーザ定義の GUI オブジェクト・クラスを対象とするユーザ定義検査の定義

ユーザ定義オブジェクトを対象とする GUI 検査の作成について

ほとんどのアプリケーションには，WinRunner のどの標準 GUI オブジェクト・クラスにも属さない GUI オブジェクトが含まれています。標準では WinRunner は，これらのオブジェクトが汎用の「**オブジェクト**」クラスに属していると認識します。アプリケーションにこれらのオブジェクトが含まれている場合，これらのオブジェクトに対してユーザ定義の検証クラスを作成して，検査機能を強化できます。次に，プロパティ検査と，新しいユーザ定義クラスの［GUI チェック］ダイアログ・ボックスを開発します。

ユーザ定義オブジェクトを検査するときに，標準の WinRunner の［GUI チェック］ダイアログ・ボックスを使用します。必要に応じて，ユーザ定義のプロパティ検査を標準のダイアログ・ボックスに追加できます。例えば，アプリケーション内に WinRunner によって汎用の「**オブジェクト**」クラスに属していると分類されるオブジェクトがある場合，それらのオブジェクトに対して「pbTool」というユーザ定義クラスを作成できます。「pbTool」クラスは，独自の一連のプロパティ検査を持ち，これらのプロパティ検査は，標準の［GUI チェック］ダイアログ・ボックスに表示されます。

はじめ，これらのダイアログ・ボックスに表示されるプロパティ検査は，汎用の「オブジェクト」クラスのオブジェクトに対して表示されるプロパティ検査と同じです。一度ユーザ定義プロパティをユーザ定義クラスに関連付ければ，このユーザ定義クラスに属するオブジェクトを対象に GUI チェックポイントを作成 / 編集するたびに，これらのプロパティ検査が表示されるようになります。

WinRunner は，GUI オブジェクトのクラス（この場合「pbTool」クラスのオブジェクト）を認識できない場合，このクラスを一般的な「オブジェクト」クラスに割り当てます。［GUI チェック］ダイアログ・ボックスは，汎用のオブジェクト・クラスに関連付けられているプロパティ検査を表示します。汎用のオブジェクト・クラスに関連付けられているプロパティ検査のリストについては，『**WinRunner ユーザーズ・ガイド**』の「GUI オブジェクトの検査」の章を参照してください。

この章で説明するように，このユーザ定義クラスに対してユーザ定義検査を追加できます。ここでは，［GUI チェック］ダイアログ・ボックスを開発します。ユーザ・インタフェースはユーザ定義であり任意です。詳細については，第 4 章「GUI 検査の作成：応用編」を参照してください。

検証のための 新しい GUI オブジェクト・クラスを追加し，そのクラスを対象とする検査を開発して指定するには，次の手順を実行します。

1. 検証のための新しいユーザ定義 GUI オブジェクト・クラスを追加します。
2. 検査の期待結果と実際の結果を記録するキャプチャ関数を作成します。
3. 期待結果と実際の結果を比較する比較関数を作成します。
4. キャプチャ関数と比較関数で定義した検査を登録します。
5. 新しい検査と新しいユーザ定義クラスを関連付けます。
6. 新しいクラスの標準の検査を設定します。

開発するキャプチャ関数と比較関数は，使う前にコンパイルしなければなりません。テスト・スクリプトから関数を実行してもコンパイルできますが，コンパイル済みのモジュールにすべての関数を入れておいて，起動テストからこのモジュールをロードすることをお勧めします。このようにすれば，WinRunner のすべてのセッションでこの関数を使用できます。コンパイル済みのモジュールの詳細については，『**WinRunner ユーザーズ・ガイド**』の「コンパイル済みモジュールの作成」の章を参照してください

WinRunner 関数ジェネレータを使用して，必要な関数の呼び出しをすべて生成し，それらを直接テスト・スクリプトに挿入できます。関数は，関数ジェネレータの「GUI verification」カテゴリにあります。関数の自動生成と自動挿入についての詳細は，『**WinRunner ユーザーズ・ガイド**』の「関数の生成」の章を参照してください。

検証のためのユーザ定義の GUI オブジェクト・クラスの追加

標準の WinRunner の GUI オブジェクト・クラスに属さないオブジェクトを対象とするプロパティ検査を実装する場合，まずオブジェクトの新しい検証クラスを定義し，次にそのオブジェクト・クラスに対するプロパティ検査を指定しなければなりません。**gui_ver_add_class** 関数を使用して，新しいクラスを定義します。この関数の構文は，次のとおりです。

gui_ver_add_class (*class_name* **[**,*ui_function***] [**,*default_check_function***]);**

- *class_name* は，オブジェクトの MSW_class プロパティまたは標準のクラス・プロパティです。GUI スパイを使用して，MSW_class プロパティを見つけます。GUI スパイの使い方の詳細については，『**WinRunner ユーザーズ・ガイド**』の「GUI マップの構成設定」の章を参照してください。
- オプションの *ui_function* パラメータを使用して，カスタマイズしたユーザ・インタフェースを持つ［GUI チェック］ダイアログ・ボックスを開発し表示できます。あるチェックポイントの検査をより容易に選択できるように，独自のチェック用ダイアログ・ボックスを作成してもよいでしょう。詳細については，第 4 章「GUI 検査の作成：応用編」を参照してください。

［GUI チェック］ダイアログ・ボックス内の［**プロパティ リスト**］ボタンを使用して，*ui_function* パラメータを呼び出せます。このボタンは，ダイアログ・ボックスの［オブジェクト］表示枠にあるオブジェクトの内少なくとも 1 つが特定のクラスに属していなければ表示されません。特定のクラスとは，**gui_ver_add_class** 関数を使用して *ui_function* パラメータを定義したクラスです。

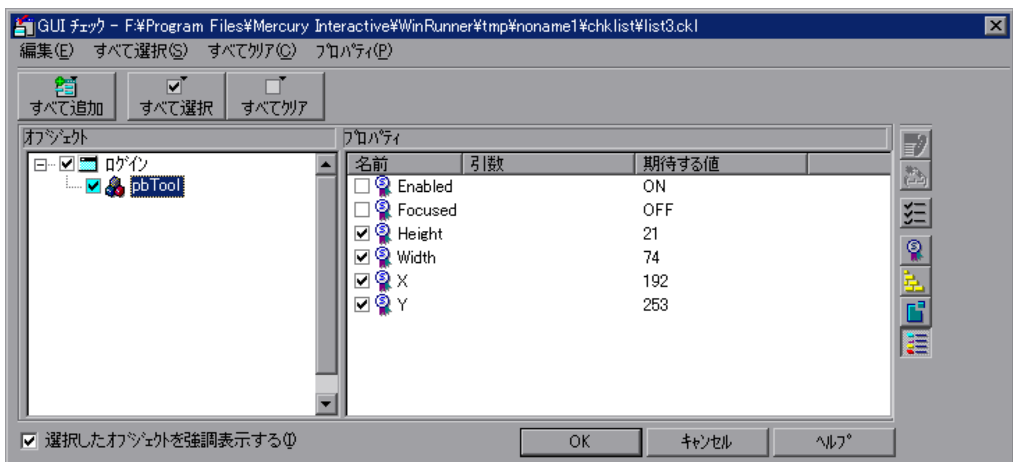
ui_function を指定しない場合，新しいクラスに対して表示される一連のプロパティ検査は，汎用の「オブジェクト」クラスに対して表示される検査と同じになります。プロパティ検査とクラスの関連付けの詳細については，19 ページ「新しいプロパティ検査と GUI オブジェクト・クラスの関連付け」を参照してください。

- オプションの *default_check_function* パラメータを使用して，新しいクラスに対して標準の起動時検査を指定できます。*default_check_function* パラメータは，

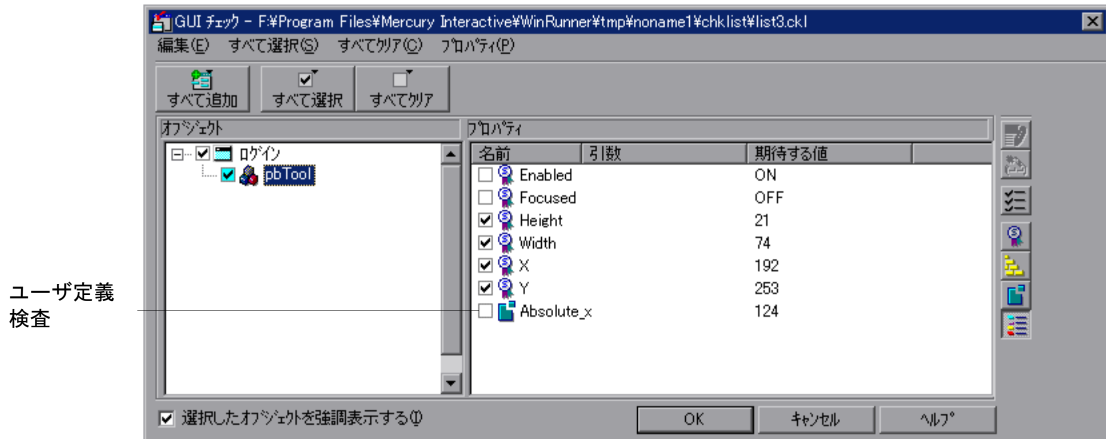
ui_function を指定して標準の検査を上書きする場合にのみ使います。詳細については，第 4 章「GUI 検査の作成：応用編」を参照してください。

標準では［GUI チェック］ダイアログ・ボックスに表示されるプロパティ検査は，汎用の「オブジェクト」クラスに対して表示されるプロパティ検査と同じです。ユーザ定義クラスにユーザ定義検査を追加すると，ユーザ定義クラスに追加した検査は［GUI チェック］ダイアログ・ボックスに表示されます。

以下の［GUI チェック］ダイアログ・ボックスには，すべてのユーザ定義オブジェクトの標準の検査が表示されています。これらのユーザ定義オブジェクトは，汎用の「オブジェクト」クラスと関連付けられます。



以下のダイアログ・ボックスで表示されているように，このユーザ定義クラスにユーザ定義検査を追加できます。

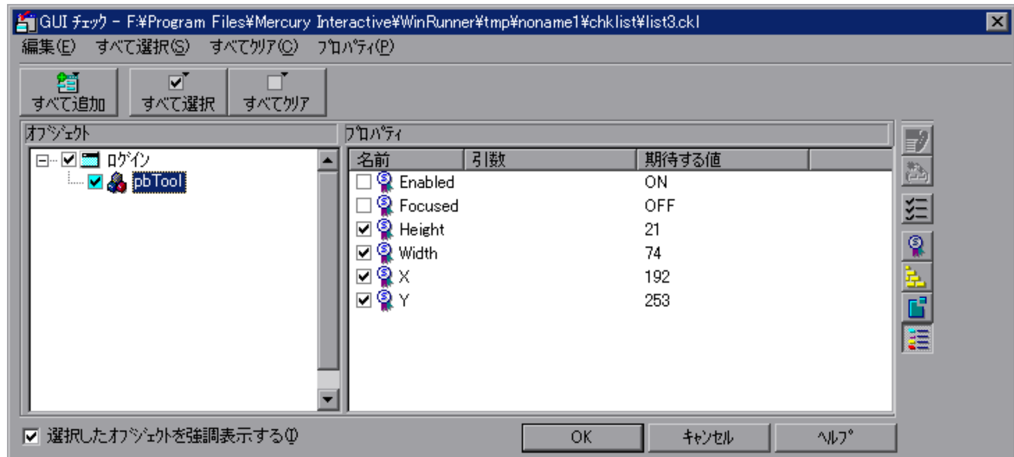


また，ユーザ定義オブジェクトを標準オブジェクト・クラスに割り当てることができます。詳細については，『**WinRunner ユーザーズ・ガイド**』の「GUI マップの構成設定」の章を参照してください。

使用例：検証するための新しい GUI オブジェクト・クラスの追加

WinRunner の GUI スパイを使用して，ペイント・アプリケーションのツールバーのプロパティを表示できます。GUI スパイの使い方の詳細については，『**WinRunner ユーザーズ・ガイド**』の「GUI マップの構成設定」の章を参照してください。WinRunner は，ツールバーが汎用の「オブジェクト」クラスに属していると認識します。ツールバーの MSW_class プロパティは「pbTool」です。以下のステートメントを使用して，ツールバーのユーザ定義クラスを作成できます。

```
gui_ver_add_class( "pbTool" );
```



標準では［GUI チェック］ダイアログ・ボックスに表示される「pbTool」クラスのプロパティ検査は，汎用の「オブジェクト」クラスに対して表示されるプロパティ検査と同じです。

ユーザ定義の GUI オブジェクト・クラスを対象とするユーザ定義検査の定義

検証するためにユーザ定義クラスを作成したあとで，このユーザ定義クラスにプロパティ検査を追加できます。

ユーザ定義クラスにプロパティ検査を追加するには，以下に示す作業を実行します。各作業の詳細については，第 2 章「標準オブジェクトを対象とするユーザ定義 GUI 検査の作成」を参照してください。

1 キャプチャ関数の作成

キャプチャ関数は，プロパティ検査の期待結果と実際の結果を定義し格納します。

2 比較関数の作成

キャプチャ関数がプロパティ検査の期待結果と実際の結果を特定したあとで，WinRunner は，結果を検証し検査が成功したかどうかを判定します。検査を検

証するには，独自の比較関数を作成するか WinRunner の標準の比較関数 **default_compare_func** を使用します。

3 新しい検査の登録

検証対象の新しい GUI オブジェクト・クラスを作成して開発します。キャプチャ関数と比較関数をコンパイルした後で，関数が定義する新しい検査を登録しなければなりません。TSL 関数 **gui_ver_add_check** を使用して，検査を登録します。

4 新しいプロパティ検査とクラスの関連付け

新しいプロパティ検査を登録した後で，検証対象の GUI オブジェクト・クラスと関連付けます。新しいプロパティ検査とクラスを関連付けることにより，［GUI チェック］ダイアログ・ボックス内で表示される対応するクラスのプロパティのリストに登録したプロパティ検査が追加されます。TSL 関数 **gui_ver_add_check_to_class** を使用して，プロパティ検査とクラスを関連付けます。

5 標準検査の設定

TSL 関数 **gui_ver_set_default_checks** を使用して，ユーザ定義の GUI オブジェクト・クラス の標準検査を設定します。

上記の TSL 関数の詳細については「**TSL オンライン・リファレンス**」を参照してください。

第 4 章

GUI 検査の作成：応用編

ユーザ定義の GUI オブジェクト・クラスに属するオブジェクト用の独自のユーザ・インタフェースを持つ［GUI チェック］ダイアログ・ボックスを作成できます。また，独自の検査結果の表示ユーティリティも実装できます。

本章では，以下の項目について説明します。

- 検証対象の新しい GUI オブジェクト・クラスの追加
- キャプチャ関数と比較関数の作成
- 新しい検査の登録
- 標準の検査の設定
- 高度な GUI 検査の実装

GUI 検査の作成の 応用手順について

アプリケーション内に WinRunner の標準の GUI オブジェクト・クラスに属さないオブジェクトが含まれている場合，それらのオブジェクトを検査するためにユーザ定義クラスを作成できます。各クラスに対して以下のことが行えます。

- WinRunner の標準の［GUI チェック］ダイアログ・ボックスの使用。このダイアログ・ボックスには，追加したすべてのユーザ定義検査と，汎用の「オブジェクト」クラスの標準の検査が含まれます。詳細については，第 3 章「ユーザ定義オブジェクトを対象とする GUI 検査の作成」を参照してください。
- この章で説明するように，カスタマイズしたユーザ・インタフェースを持ち，結果表示ユーティリティを関連付けた，独自の［GUI チェック］ダイアログ・ボックスを開発できます。

新しい GUI オブジェクト・クラスを追加して，次にユーザ定義のユーザ・インタフェースと結果表示ユーティリティを開発するには，次の手順を実行します。

1. 新しいユーザ定義の GUI オブジェクト・クラスを定義し，カスタマイズしたユーザ・インタフェースを持つ［GUI チェック］ダイアログ・ボックスを開発します。
2. 検査の期待結果と実際の結果を記録するキャプチャ関数を作成します。
3. 期待結果と実際の結果を比較する比較関数を作成します。
4. キャプチャ関数と比較関数で定義した検査を登録します。
5. 新しいクラスの標準の検査を設定します。

検証対象の新しい GUI オブジェクト・クラスの追加

gui_ver_add_class 関数を使用して，検証対象の新しい GUI オブジェクト・クラスを定義します。検証対象の新しいクラスの追加の詳細については，第 3 章「ユーザ定義オブジェクトを対象とする GUI 検査の作成」を参照してください。

gui_ver_add_class 関数を使用する場合，*ui_function* パラメータがユーザ定義関数の名前であることに注意してください。このユーザ定義関数を使ってカスタマイズしたユーザ・インタフェースを持つ［GUI チェック］ダイアログ・ボックスを開発して表示できます。［GUI チェック］ダイアログ・ボックスは，GUI チェックポイントの作成時に GUI オブジェクトをダブルクリックすると表示されます。カスタマイズしたユーザ・インタフェースを持つ［GUI チェック］ダイアログ・ボックスを作成すれば，特定のチェックポイントの検査をより効果的に選択できるようになります。例えば，「**テーブル**」を対象とするユーザ定義クラスがある場合，カスタマイズしたユーザ・インタフェースを持つ［GUI チェック］ダイアログ・ボックスを作成すれば，比較するテーブルのカラムやその比較基準を容易に選択できるようになります。

ui_function パラメータの構文は次のとおりです。

function **ui_function(** *in window, in object, inout check_list, inout arg_list* **);**

- *window* は，オブジェクトが属するウィンドウの記述です。
- *object* は，ユーザが選択したオブジェクトの記述です。

- *check_list* には 2 つの機能があります。関数を呼び出したときに，オブジェクトの標準チェックリストが関数に渡されます。関数は，新しいチェックリストを出力として渡されなければなりません。チェックリストは，空白文字またはコンマで区切られた複数の検査名で構成されます。
- *arg_list* は，出力として返す文字列です。*arg_list* は，パラメータとして各プロパティ検査のキャプチャ関数や比較関数に渡されます。*arg_list* パラメータが返したファイル名を使用して，*arg_list* パラメータをファイルに格納できます。

［GUI チェック］ダイアログ・ボックスの［**プロパティ リスト**］ボタンを使用して，*ui_function* パラメータを呼び出せます。このボタンは，ダイアログ・ボックスの［オブジェクト］表示枠にあるオブジェクトの内少なくとも 1 つが特定のクラスに属していなければ表示されません。特定のクラスとは，**gui_ver_add_class** 関数を使用して *ui_function* パラメータを定義したクラスです。

default_check_function パラメータの構文は次のとおりです。

function **default_check_function (** *in window, in object, inout check_list, inout arg_list* **);**

default_check_function のパラメータは，上の *ui_function* 関数のパラメータと同じです。

カスタマイズした［GUI チェック］ダイアログ・ボックスを開発するときの **gui_ver_add_class** 関数の使用例は，35 ページ「高度な GUI 検査の実装」を参照してください。

キャプチャ関数と比較関数の作成

ユーザ定義プロパティ検査の期待結果と実際の結果を作成して格納できるようにキャプチャ関数を作成します。キャプチャ関数が検査の期待結果と実際の結果を特定し終えると，WinRunner は，結果を検証して検査が成功したかどうかを判定します。検査を検証するには，独自の比較関数を作成して使うか，WinRunner の標準の比較関数 **default_compare_func** を使用します。

キャプチャ関数と比較関数の詳細については，第 2 章「標準オブジェクトを対象とするユーザ定義 GUI 検査の作成」を参照してください。

キャプチャ関数と比較関数の使用例の説明は 35 ページ「高度な GUI 検査の実装」を参照してください。

新しい検査の登録

キャプチャ関数と比較関数を作成した後で，関数が定義する新しいプロパティ検査を登録します。検査の登録については，第 2 章「標準オブジェクトを対象とするユーザ定義 GUI 検査の作成」を参照してください。

gui_ver_add_check 関数を使うと，*display_function* パラメータを通じて，検査結果を表示するユーザ定義の表示ユーティリティが WinRunner で使えるようになります。例えば，表示関数を持つ［GUI 検証結果］ダイアログ・ボックスでプロパティ検査を選択すると，以下のように［表示］ボタンが有効になります。［表示］ボタンをクリックすると，関連付けられている表示関数によって検査結果が表示されます。*display_function* を指定しないと，WinRunner は，標準の結果表示機能を使用します。

「check2」には表示機能が関連付けられているので，［表示］ボタンが有効になります。

display_function パラメータの構文は次のとおりです。

function **display_function (** *in expected, in actual, in result, in diff* **);**

- *expected* は，期待結果文字列かファイル名です。
- *actual* は，実際の結果文字列かファイル名です。
- *result* は，比較関数の結果です。比較が成功した場合は 0，不一致が見つかった場合は他の値になります。
- *diff* は，比較関数から受け取った文字列です。表示関数が必要とするファイル名やその他の情報を含めることができます。

標準の検査の設定

新しい検査の標準検査を設定するには，**gui_ver_set_default_checks** 関数を使用します。標準の検査の設定の詳細については，第 2 章「標準オブジェクトを対象とするユーザ定義 GUI 検査の作成」を参照してください。

高度な GUI 検査の実装

次の例では，GUI 検証の主要なカスタマイズ機能の使い方を紹介します。分かりやすくするために，すべての検査が単純に乱数を返すようにしてあります。使用例では，ユーザ定義クラス *AfxWnd* を追加し，次にそのクラスの検査を追加します。*ui_function* パラメータは，WinRunner の **pause_test** 関数を使用して，次のような単純な入 / 出力ダイアログ・ボックスを作成します。

WinRunner の pause_test 関数で開発したユーザ定義のユーザ・インタフェースを持つダイアログ・ボックス

この例では，GUI 検証のカスタマイズで使われるメカニズムを紹介します。外部 DLL を使えば，このプロトタイプを元にした高度な入 / 出力画面を実装できます。

このテストは，WinBurger で動作するように設計されています。WinBurger は，WinRunner に付属するサンプルのアプリケーションです。WinBurger の［Reset］ボタンは，ユーザ定義クラス「AfxWnd」に属します。このクラスは，この例でカスタマイズされます。WinBurger は，*＜インストール先フォルダ＞¥samples¥bin¥winbur* にあります。

```
# ユーザ定義関数をロードする。
reload("udf_gui");

# 新しい検証クラス「AfxWnd」を追加する。
rc=gui_ver_add_class("AfxWnd", "reset_ui_func"); # adds class “AfxWnd"

# check1 を登録する。
rc=gui_ver_add_check("check1","check1_capt","compare1");

# check1 をクラスに追加する。

""""# check2 を登録する。
rc=gui_ver_add_check("check2","check2_capt","compare2","display_func2");

# check2 をクラスに追加する。

""""# 新しいクラスに標準の検査を設定する。
rc=gui_ver_set_default_checks("AfxWnd","check1");
------------------------------------------------------------------------------------------

# udf_gui
function reset_ui_func(window, object, inout checklist, out arglist)
{
    auto res=pause_test("GUI 検証例 UI_function の表示 ¥n¥n チェック名を選択してください。¥n¥n 現在のチェック：" & checklist, "check1", "check2", "Both"
);
    if (res==0) {
        checklist = "check1";
        arglist = "User selected check1";
    }
    else
    if (res==1) {
        checklist = "check2";
        arglist = "User selected check2";
    }
    else
    if (res==2) {
```

```
        checklist = "check1 check2";
        arglist = "User selected check1 & check2";
    }
    else
        return -1;
    return 0;
}
```

キャプチャ関数 #1

```
function check1_capture(object, inout value)
{
    value = 100*rand();
    return 0;
}
```

キャプチャ関数 #2

```
function check2_capture(object, inout file)
{
    file_open(file,FO_MODE_WRITE);
    file_printf(file, “%"S”,””The result of check2 was sent to a file.¥n¥nThe result
of check2 is: " & 100*rand() )”);
    file_close(file);
    return 0;
}
```

比較関数 #1

```
function compare1(exp_val, act_val, arglist, inout diff_file)
{
    diff_file = "";
    if (exp_val != act_val) {
        return E_DIFF;
    }
    return E_OK;
}
```

比較関数 #2

```
function compare2(exp_file, act_file, arglist, inout diff_file)
{
    auto exp_buf, act_buf;
    read_file(exp_file, exp_buf);
    read_file(act_file, act_buf);
    if (exp_buf != act_buf || arglist != "") {
        file_open(diff_file,FO_MODE_WRITE);
        file_printf(diff_file,"Difference forced !");
        file_close(diff_file);
        return 1;
    }
    diff_file = "";
    return E_DIFF;
}
```

表示関数

```
function display_func2(exp_file, act_file, result, diff_file)
{
    auto exp_buf, act_buf, diff_buf;
    read_file(exp_file, exp_buf);
    read_file(act_file, act_buf);
    read_file(diff_file, diff_buf);
    pause_test("¥nExpected: " & exp_buf & "¥nActual: " & act_buf & "¥n¥nResult:
" & result & "¥nDiff: " & diff_buf, "OK", "Cancel", "Close");
    return 0;
}
function read_file(name, out buf)
{
    auto    tmp;
    buf = "";
    file_open(name,FO_MODE_READ);
    if (name != "") {
        while (file_getline(name,tmp)) {
            buf = buf & tmp;
        }
        file_close(name);
    }
}
```

第 2 部

記録方法のカスタマイズ

第 5 章

記録されるステートメントのカスタマイズ

ユーザ定義の GUI オブジェクトに対する操作を記録すると，結果として生成されるテスト・スクリプトには汎用の TSL ステートメント，**obj_** が含まれます。ユーザ定義記録関数を作成することで，テスト・スクリプトを読みやすくできます。

本章では，以下の項目について説明します。

- ユーザ定義記録関数について
- ユーザ定義記録関数の開発
- ユーザ定義記録関数の GUI オブジェクト・クラスへの関連付け
- ユーザ定義実行関数の開発
- ユーザ定義記録関数の例

記録されるステートメントのカスタマイズについて

多くのアプリケーションには，ユーザ定義の GUI オブジェクトが含まれています。こうしたオブジェクトは WinRunner のどの標準 GUI オブジェクト・クラスにも属しません。WinRunner はユーザ定義オブジェクトの動作を知らないので，こうしたオブジェクトを操作すると，汎用の **obj_mouse** ステートメントがテスト・スクリプトに記録されます。**obj_mouse** ステートメントは汎用なので，次の 2 つの問題が生じる可能性があります。

- 記録されたステートメントが記述的でないため，テスト・スクリプトは読みづらく，分析しづらいものになる。
- 記録されたステートメントに実行した操作が完全に記述されない。テストを実行したときに，WinRunner は記録された操作を正しく再現できない。

ユーザ定義の記録関数を実装すれば，これらの問題を 2 つとも解決できます。

ユーザ定義記録関数を使えば，汎用の **obj_** ステートメントの代わりに WinRunner に記録させるステートメントを指定できます。つまり，ユーザ定義オブジェクト・クラスに属する GUI オブジェクトに対して特定の操作を実行するときに，記録されるステートメントを指定できます。

ユーザ定義記録関数を実装するには，次の手順を実行します。

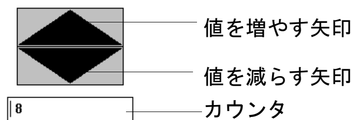
1. ユーザ定義記録関数を開発します。
2. ユーザ定義記録関数をユーザ定義 GUI オブジェクト・クラスと関連付けます。
3. 必要に応じて，ユーザ定義実行関数を実装します。

注：ユーザ定義記録関数と指定された GUI オブジェクト・クラスの論理名関数を組み合わせて，WinRunner によってテスト・スクリプトに記録されるステートメントをさらに読みやすくできます。論理名関数を使うと，WinRunner はユーザ定義 GUI オブジェクトに記述的な論理名を割り当てることができます。詳細については第 7 章「割り当てられる論理名のカスタマイズ」を参照してください。

ユーザ定義記録関数について

次のシナリオで，ユーザ定義記録関数の使い方を説明します。

アプリケーションにユーザ定義の「スピンボタン」が含まれているとします。スピンボタンの上向き矢印をクリックすると，関連付けられたカウンタの数が 1 つずつ増えていきます。下向きの矢印をクリックすると，カウンタの数が 1 つずつ減ります。どちらかの矢印をクリックして，マウス・ボタンを押したままにすると，その間カウンタの数は継続的に増加，あるいは減少します。



このスピンボタンは標準 GUI オブジェクト・クラスに属さないので，このスピンボタン用に「spinbutton」という名前のユーザ定義クラスを作成します。例えばこのスピンボタンに対する 3 つの異なるマウス・クリックを記録するとします。WinRunner は次のようなステートメントをテスト・スクリプトに記録します。

実行する操作	記録されるテスト・スクリプト・ステートメント	操作前のカウンタ	操作後のカウンタ
上矢印をクリック	obj_mouse_click ("SpinButton", 50, 22, LEFT);	1	2
下矢印をクリック	obj_mouse_click ("SpinButton", 50, 58, LEFT);	2	1
マウス・ボタンを押しながら，上矢印をクリック	obj_mouse_click ("SpinButton", 50, 22, LEFT);	1	6

注：シナリオは WinRunner が Visual Basic のサポート付きでインストールされていないと仮定します。Visual Basic のサポートをインストールしている場合には，シナリオに記述されたユーザ定義記録関数は，すでに spinbutton クラスにすでに実装されています。

記録されるステートメントの読みやすさの改善

上の記録されるステートメントはマウス・クリックの座標を除けばまったく同じです。したがって，各ステートメントでどのような操作が記録されたか（例えば，値を増やす矢印が押されたのか，値を減らす矢印が押されたのか，単な

るマウスのクリックなのか，あるいは一定時間マウス・ボタンが押され続けたのかなど）を判別することは困難です。

ユーザ定義記録関数を実装すれば，記録されるテスト・スクリプトを読みやすく，分析しやすいものにできます。その結果，WinRunner はより記述的なステートメントを記録できます。

記録済みのテスト・スクリプト・ステートメント	実行される操作
spin_up ("SpinButton", 50, 22, LEFT);	上向き矢印をクリック
spin_down ("SpinButton", 50, 58, LEFT);	下向き矢印をクリック

上記の「ユーザ定義」ステートメントからは，spin_up ステートメントは上（値を増やす）矢印のマウス・クリックを示し，spin_down ステートメントは下（値を減らす）矢印のマウス・クリックを示すことが一目で分かります。

実行精度の向上

これで WinRunner は，最初の 2 つの「単純」なマウス・クリックは要求どおりに実行します。ただし，マウス・ボタンを押して，押したままにすることを示す 3 番目のステートメントは正確に実行しません。3 番目のマウス・クリックは汎用の **obj_mouse_click** ステートメントとして記録され，マウス・ボタンが押されている間に矢印がアクティブになった回数についての情報が一切含まれていません。そのため，WinRunner は単純なマウス・クリックを実行して，カウンタが 1 から 2 に増加しますが，要求されたように 1 から 6 へは増加しません。

ユーザ定義記録関数を実装すれば，テスト実行時の問題を解決できます。ユーザ定義記録関数を使うことで，WinRunner が記録するステートメントには実行済みの操作が確実にすべて記述されます。これにより WinRunner で記録済みステートメントを思いどおりに実行できます。

次のテーブルは変更後の spin_up と spin_down ステートメントを示します。変更には，マウス・ボタンを押し続けている間に各矢印が何回アクティブになるかを定義したパラメータが含まれます。

実行された操作	記録されたテスト・スクリプト・ステートメント	操作前のカウンタ	操作後のカウンタ
上矢印をクリック	spin_up ("SpinButton", 1);	1	2
下矢印をクリック	spin_down ("SpinButton", 1);	2	1
上矢印をクリック，マウス・ボタンを押し続ける	spin_up ("SpinButton", 5);	1	6

ユーザ定義記録関数は WinRunner の非標準の関数を生成します。非標準関数を実行するには，「**ユーザ定義実行関数**」を用意する必要があります。詳細は 51 ページ「ユーザ定義実行関数の開発」を参照してください。

ユーザ定義記録関数と論理名関数の結合

ユーザ定義記録関数を論理名関数と組み合わせて実装すれば，テスト・スクリプトは一層読みやすくなります。WinRunner は論理名関数を使って，ユーザ定義 GUI オブジェクトに記述的な論理名を割り当てることができます。詳細については，第 7 章「割り当てられる論理名のカスタマイズ」を参照してください。

ユーザ定義記録関数の開発

ユーザ定義記録関数を作成して，テスト・スクリプトに記録されるステートメントをカスタマイズします。この関数は，ユーザ定義オブジェクト・クラスのオブジェクトに対して特定の操作を実行すると文字列を返します。WinRunner はこの文字列を，テスト・スクリプトに記録するステートメントの基礎として使用します。

例えば，ユーザ定義スピンボタンをクリックすると文字列を返す，ユーザ定義の記録関数を実装するとします。WinRunner はこの関数を実装する前に，次のようなステートメントを記録します。

```
obj_mouse_click("spinbutton",50,100);
```

記録関数を実装すると，WinRunner は次のようなステートメントを記録します。

```
spin_up("spinbutton", 1 );
```

ユーザ定義記録関数の記述

ユーザ定義記録関数は C プログラミング言語で記述してコンパイルし，DLL にリンクします。ユーザ定義記録関数を GUI オブジェクト・クラスに関連付ける際は，DLL の名前を指定します。51 ページ「ユーザ定義記録関数の GUI オブジェクト・クラスへの関連付け」を参照してください。

ユーザ定義記録関数のプロトタイプは次のとおりです。

int **function** (HWND FAR* *phWnd*, UINT *msg*, WPARAM *wParam*, LPARAM *lParam*, char* *str*, int *size*);

パラメータ

phWnd	操作対象 GUI オブジェクトのハンドルへのポインタ。ポインタの内容を変更して，他の GUI オブジェクトに対する操作を記録できます。
msg	受信した Windows のメッセージ。
wParam	最初のメッセージ・パラメータ。
lParam	2 番目のメッセージ・パラメータ。
str	WinRunner によって割り当て済みのバッファ。記録関数はテスト・スクリプトに記録する文字列を，割り当てられたバッファに代入します。
size	*str* バッファのサイズ（単位：バイト）。

記録関数は WinRunner がテスト・スクリプトに記録する文字列を *str* パラメータに代入します。「%%m」を GUI オブジェクトの論理名に置き換えるために，WinRunner の「%m」形式を使って文字列を作ります。例えば，ユーザ定義記録関数は以下の文字列を *str* パラメータに代入できます。

```
spin_up ("%%m", 1)
```

文字列の最後にセミコロン（;）を入れてはいけません。WinRunner はテスト・スクリプトにステートメントを記録するときにセミコロンを付け加えます。ステートメントがテスト・スクリプトに記録されるときに，WinRunner が「%m」を GUI オブジェクトの論理名で置き換えます。

値を返す

記録関数は関数の「送信モード」を示す値を返します。例えば，次のステートメントをユーザ定義記録関数に含めることができます。

```
return(SEND_LINE);
```

返すことのできる送信モードのオプションを以下に示します。

- SEND_LINE

  WinRunner に *str* パラメータで返される文字列をテスト・スクリプトに記録するように指示します。以前に格納された文字列がある場合，WinRunner は先にその文字列をテスト・スクリプトに記録します。

- KEEP_LINE

  WinRunner に *str* パラメータで返された文字列を格納して，タイマーを起動するように指示します。以前に格納された文字列がある場合，WinRunner は先にその文字列をテスト・スクリプトに記録します。タイムアウトが発生して，以降のマウス入力が検出されない場合は，WinRunner は指定された新しい文字列をテスト・スクリプトに記録します。マウスのシングル・クリックを検出した後は，KEEP_LINE を使用します。WinRunner はタイムアウトまでにシングル・クリックの後にもう 1 つクリックが続くことでダブル・クリックが成立するかどうか待機します。

- KEEP_LINE_NO_TIMEOUT

  WinRunner に *str* パラメータで返される文字列を格納するように指示します。以前に格納された文字列がある場合，WinRunner はテスト・スクリプトに文字列を記録します。

- REPLACE_AND_SEND_LINE

  WinRunner に *str* パラメータで返される文字列をテスト・スクリプトに記録するように指示します。以前に格納された文字列がある場合，WinRunner はその文字列を削除します。

- REPLACE_AND_KEEP_LINE

  WinRunner に *str* パラメータで返される文字列を格納して，タイマーを起動するように指示します。以前に格納された文字列がある場合，WinRunner はその文字列を削除します。タイムアウトが発生し，その後マウス入力がない場合，WinRunner は文字列をテスト・スクリプトに記録します。

- CLEAN_UP

  WinRunner に *str* パラメータに格納されている文字列がある場合には，テスト・スクリプトにそれらの文字列を記録するように指示します。

- NO_PROCESS

  str パラメータが空の場合，NO_PROCESS は WinRunner に GUI オブジェクトの標準関数を記録するように指示します。

  str パラメータが空でない場合，NO_PROCESS は WinRunner に関数を一切記録しないよう指示します。

Windows のメッセージの追加

WinRunner はすべての Windows メッセージを監視しますが，数多くの Windows メッセージの中から実際にはわずかな数のメッセージしか処理しません。すなわち，WinRunner は少数を除いて，その他のメッセージは無視します。実装する記録関数では，WinRunner にほかのメッセージも処理してもらう必要がある場合もあります。TSL 関数の **add_record_message** を使用して，処理してもらいたい追加のメッセージを指定できます。**add_record_message** 関数の構文は次のとおりです。

add_record_message (*message_number*);

- *message_number* は WinRunner に処理させたい，追加の Windows メッセージの番号，または識別子です。

例えば，次のステートメントは WinRunner に WM_MOUSEMOVE メッセージを処理対象メッセージのリストに追加するように指示します。

```
add_record_message(512);
```

ユーザ定義記録関数の例については，52 ページ「ユーザ定義記録関数の例」を参照してください。

ユーザ定義記録関数の GUI オブジェクト・クラスへの関連付け

add_cust_record_class 関数を使用してユーザ定義記録関数を GUI オブジェクト・クラスに関連付けます。**add_cust_record_class** 関数の構文は次のとおりです。

add_cust_record_class (*MSW_class*, *dll_name* **[,** *rec_func* **]**
[, *log_name_func* **]);**

- *MSW_class* は，ユーザ定義記録関数が関連付けられるユーザ定義オブジェクトの MSW_class です。
- *dll_name* はユーザ定義記録関数をコンパイルしてリンクした DLL の完全パスとファイル名です。GUI オブジェクト・クラスの論理名関数も存在する場合には，その関数もこの DLL に含まれます。詳細については，第 7 章「割り当てられる論理名のカスタマイズ」を参照してください。
- *rec_func* は DLL の記録関数名です。記録関数は WinRunner がテスト・スクリプトに記録する文字列を返します。
- *log_name_func* は DLL に含まれる論理名関数（存在する場合）の名前です。*log_name_func* 関数は *MSW_class* 内のユーザ定義 GUI オブジェクトにユーザ定義論理名を提供します。詳細については，第 7 章「割り当てられる論理名のカスタマイズ」を参照してください。

次の例では，**add_cust_record_class** 関数はユーザ定義記録関数，*SpinHighLevelRec* を SpinButton クラスに追加します。

```
add_cust_record_class("SpinButton", "c:¥¥arch¥¥vb_util.dll",
    "SpinHighLevelRec", " ");
```

ユーザ定義実行関数の開発

ユーザ定義ステートメントを生成するユーザ定義記録関数を実装する場合，ユーザ定義実行関数を開発して，WinRunner が記録済みのステートメントを実行できるようにする必要があります。例えば，次の関数をテスト・スクリプトに記録するユーザ定義記録関数を開発するとします。

```
custom_function(2,2);
```

custom_function はユーザ定義関数への呼び出しなので，WinRunner はこれを認識できず，そのため関数の実行もできません。そこで，custom_function ステートメントを実行するたびに WinRunner が何を実行するかを定義するユーザ定義関数を開発します。

ユーザ定義実行関数はユーザのアプリケーションからユーザ定義オブジェクトの状態や場所などの情報を取得して，マウス・カーソルを要求された場所に移動し，マウスあるいはキーボード入力を実行します。**obj_get_info** などいくつかの TSL 関数を使って，オブジェクトに関する情報を取得できます。また，**click**，**obj_mouse_drag** および **move_locator_abs** など，他の TSL 関数を使って機能を実行することもできます。

以下に実行関数，**spin_up** の TSL 実装例を示します。この例を元にして実行関数の実装を行うこともできます。

```
public function spin_up(win, times){
    auto hWnd;
    auto res;

    # GUI オブジェクト・ハンドルを取得して，DLL に送る。
    res = obj_get_info(win, "handle", hWnd);
    if(res != E_OK)
        return(res);

    # DLL, _spin_up を呼び出す。
    res = _spin_up(hWnd, times);

    # if _spin_up の呼び出しが失敗したら内部 TSL 関数を使用する。
    if(res != E_OK)
        process_return_value(res);
    return(res);
}
```

ユーザ定義記録関数の例

次の例では Visual Basic コントロール，ThunderlistBox のユーザ定義記録関数の実装を説明します。この例を元にして，ユーザ定義記録関数を実装することもできます。記録関数の戻り値を設定するために cust_rec.h をインクルードしています。

ファイル名：cust_rec.h

```
// 記録関数の戻り値。
#define SEND_LINE      0
#define KEEP_LINE      1
#define REPLACE_AND_SEND_LINE   2
#define REPLACE_AND_KEEP_LINE   3
#define CLEAN_UP       4
#define KEEP_LINE_NO_TIMEOUT    5
#define NO_PROCESS  6
```

ファイル名：cust_rec.c

```
#define EXPORTED _far _pascal __export
#include <windows.h>
#include "cust_rec.h"
#include <windowsx.h>   // Windows Messages Cracker

#define MAXKEYS  9
BOOL isMouseUponObject (HWND  hwin, LPARAM lParam);
WORD GetListKey (HWND  hwin, WPARAM wParam);

// ThunderListBox のユーザ定義記録
//----------------------------------------------------------------------
//                    ThunderListBox
//
// Visual Basic ListBox の実装。
// すべての ListBox コ マンドは Windows Messages Cracker の形式で書かれて
いる。

// 注：
// WM_KEY* メッセージを Windows のドキュメントに記載されているのとは違
う形で受け取る：
//    wParam = スキャン・コード ; 拡張キーの場合は，最上位のビットが設定さ
れている。
//    lParam は有益な情報は含まない。
//----------------------------------------------------------------------

int EXPORTED ThunderListBox(HWND FAR* pwin, UINT msg,
                     WPARAM wParam, LPARAM lParam,
                     LPSTR rec_str, int len)
```

```
{
   static bMouseDown   = FALSE;
   static bPushKeyDown = FALSE;
   WORD   wVirtKey;
   int    nReturn = SEND_LINE;
   int    nItemSel;
   char   szBuff[255];

   UINT ss;

   HWND win = *pwin;
   switch (msg) {
   case WM_LBUTTONDBLCLK:
      nItemSel = ListBox_GetCurSel(win);
      if (nItemSel != LB_ERR) {
         // 何かが選択された場合，選択された文字列を取得して
         ListBox_GetText(win, nItemSel, szBuff);
         // コマンドラインを送信する。
         wsprintf(rec_str,
            "ActivateThunderListItem (¥"%%m¥", ¥"%s¥")", (LPSTR)szBuff);
         // 先の「SelectThunderListItem」を置換する。
         nReturn = REPLACE_AND_SEND_LINE;
      }
      break;

   // bMouseDown フラグを設定する。
   case WM_LBUTTONDOWN:
      if (isMouseUponObject(win, lParam))
         bMouseDown = TRUE;
      break;

   case WM_LBUTTONUP:
      if (bMouseDown) {   // bMouseDown フラグが設定されており，
         bMouseDown = FALSE;
         nItemSel = ListBox_GetCurSel(win);
         if (nItemSel != LB_ERR) {
            // 何かが選択されていれば - 選択されている文字列を取得し，
            ListBox_GetText(win, nItemSel, szBuff);
            // コマンドラインを送信し，
```

```
            wsprintf(rec_str,
                "SelectThunderListItem (¥"%%m¥", ¥"%s¥")",
                (LPSTR)szBuff);
            // DBLCLICK に備えて保管する。
            nReturn = KEEP_LINE;
        }
    }
    break;

case WM_KEYDOWN:
    // 実際の VirtKey 値を取得し，
    // 有効ならば bPushKeyDown フラグを設定する。
    ss = HIWORD(lParam);
    ss = LOBYTE(ss);
    ss = MapVirtualKey(ss, 1);

    if (GetListKey (win, wParam) != 0xFFFF)
        bPushKeyDown = TRUE;
    break;

case WM_KEYUP:
    // 実際の VirtKey 値を取得する。
    wVirtKey = GetListKey (win, wParam);
    if (bPushKeyDown && (wVirtKey != 0xFFFF)) {
        // 値が有効で，bPushKeyDown フラグが設定されている場合，
        // フラグをリセットする。
        bPushKeyDown = FALSE;

        nItemSel = ListBox_GetCurSel(win);
        if (nItemSel != LB_ERR) {
            // 何かが選択されている場合は，選択された文字列を取得して，
            ListBox_GetText(win, nItemSel, szBuff);
            // 適切なコマンドライン（Activate または Select）を送信する。
            if (wVirtKey == VK_RETURN)
                wsprintf(rec_str,
                    "ActivateThunderListItem (¥"%%m¥", ¥"%s¥")",
                    (LPSTR)szBuff);
            else
                wsprintf(rec_str,
```

```
                        "SelectThunderListItem (¥"%%m¥", ¥"%s¥")",
                        (LPSTR)szBuff);
                }
            }
            break;

        default:
            break;
    }

    return (nReturn);
}

//------------------------------------------------------------------------
// isMouseUponObject
// hwin Window Rectangle 上にマウスが置かれているかどうか検査する。
//------------------------------------------------------------------------

BOOL isMouseUponObject(HWND hwin, LPARAM lParam)
{
    RECT    rect;
    POINT   ptMouse;
    BOOL    bResult;

    // ローカルのマウスの座標を取得する。
    ptMouse.x = LOWORD(lParam);
    ptMouse.y = HIWORD(lParam);

    // オブジェクト・ウィンドウの座標を取得する。
    GetWindowRect(hwin, (RECT FAR*)&rect);

    // ローカルのマウスの座標をウィンドウの座標に変換する。
    ClientToScreen(hwin, (POINT FAR*)&ptMouse);

    // オブジェクト・ウィンドウの矩形上にマウスがあるかどうかテストする。
    bResult = PtInRect((const RECT FAR*)&rect, ptMouse);

    return bResult;
}
```

```
//----------------------------------------------------------------------
// GetListKey
// Key メッセージが次のいずれかの Virtual Keys に対応する場合を検出する：
// VK_RETURN, VK_UP, VK_RIGHT, VK_LEFT, VK_DOWN, VK_HOME,
// VK_END, VK_PRIOR, VK_NEXT,
//
// wParam と lParam が変更されていて，何かの理由で，
// Keyboard のキー（ENTER を除く），および KeyPad の［ENTER］に対して，
// 最上位のビットが wParam 値に追加されるとみなす。
//----------------------------------------------------------------------

WORD GetListKey(HWND hwin, WPARAM wParam)
{
   int     i;
   BOOL    bKeyPad,
           bNumLock;

   // 通常の移動および実行キー。
   WORD wVirtKeys[MAXKEYS] =
      {VK_RETURN, VK_UP, VK_RIGHT, VK_LEFT,
       VK_DOWN, VK_HOME, VK_END, VK_PRIOR, VK_NEXT};

   // KeyPad のキー。値はデバッグによって検出した。
   bKeyPad = ((wParam >= 0x47) && (wParam <= 0x51));

   // NUMLOCK キーのトグルを検査する。最下位のビットを検査する。
   bNumLock = 0x01 & GetKeyState(VK_NUMLOCK);

   if (!bKeyPad || (bKeyPad && !bNumLock)) {
      // ［NUMLOCK］－［KEYPAD］以外のすべてのキーを処理する
      // （KeyPad の［ENTER］も除く）。
      for (i = 0; i < MAXKEYS; i++) {
         // wParam の最上位のビットをリセットし，Virtual Keys をテストする。
         if ((wParam & 0x7F) == MapVirtualKey(wVirtKeys[i], 0))
            // 現在の Virtual Key の値を返す。
            return wVirtKeys[i];
      }
   }
```

```
    // 「何も」返さない。
    return 0xFFFF;
}
```

第 6 章

GUI オブジェクトへのユーザ定義プロパティの追加

任意の GUI オブジェクト・クラスにユーザ独自のプロパティを追加して，アプリケーション内の GUI オブジェクトを一意に識別できるメカニズムを向上できます。

本章では，以下の項目について説明します。

- ユーザ定義プロパティに対するクエリー関数の開発
- ユーザ定義プロパティ用の検証関数の開発
- ユーザ定義プロパティの登録
- ユーザ定義プロパティの GUI オブジェクト・クラスへの割り当て
- ユーザ定義プロパティ関数の例

GUI オブジェクトへのユーザ定義プロパティの追加について

WinRunner は，オブジェクトの物理的記述を生成することで，各 GUI オブジェクトを一意に識別します。この物理的記述はオブジェクトの必須プロパティのリストで構成されます。必須プロパティを使ってオブジェクトを一意に識別できない場合，WinRunner はオプションのプロパティを物理的記述に含めます。それでもオブジェクトを一意に識別できないと，WinRunner はセレクタも含めます。WinRunner の GUI オブジェクトの識別方法については，『**WinRunner ユーザーズ・ガイド**』を参照してください。

オプションのプロパティとセレクタを含めると，物理的記述が長く，複雑になることがあります。WinRunner が GUI オブジェクトを一層効率的に識別できるようにするには，ユーザ独自の「**ユーザ定義**」プロパティを登録します。例えば，Visual Basic を使って作成された GUI オブジェクトには *vb_name* プロパ

ティが割り当てられます。任意のウィンドウで，*vb_name* プロパティは常に一意です。WinRunner を Visual Basic サポート付きでインストールすると，*vb_name* プロパティがユーザ定義プロパティとして自動的に追加されます。これにより，WinRunner は Visual Basic アプリケーション内で，より効率的に GUI オブジェクトを識別できます。

Visual Basic と同様，他の多くの開発環境でも GUI オブジェクトにプロパティを割り当てます。こうしたプロパティをユーザ定義の WinRunner プロパティとして定義することで，WinRunner がアプリケーションの GUI オブジェクトを一意に識別するメカニズムを高めることができます。

ユーザ定義プロパティを追加するには，次の手順を実行します。

1 ユーザ定義プロパティの値が必要な場合には，その値を取得するためのクエリー関数を開発します。
2 指定された GUI オブジェクトのユーザ定義プロパティの値が期待どおりであるどうか検証する関数を開発または指定します。
3 ユーザ定義プロパティを登録します。
4 ユーザ定義プロパティを GUI オブジェクト・クラスに割り当てます。

ユーザ定義プロパティに対するクエリー関数の開発

GUI オブジェクトのユーザ定義プロパティの値を評価して返すクエリー関数を作成します。例えば，*new_property* というユーザ定義プロパティを追加するとします。値が必要な時に，GUI オブジェクトの *new_property* の値を実際に評価して返すのは，クエリ関数です。

クエリ関数は C 言語で書き，コンパイルしてこれを DLL にリンクします。ユーザ定義プロパティを登録するときは，DLL の名前を指定します。63 ページ「ユーザ定義プロパティの登録」を参照してください。

クエリー関数の構文は，次のとおりです。

void **query_func** (HWND *hWnd*, LPSTR *str*, int *size*)

パラメータ

hWnd	ユーザ定義プロパティを評価する対象となる GUI オブジェクトのハンドル。

str	WinRunner によって割り当てられているバッファ。クエリー関数はこのバッファにユーザ定義プロパティの値を代入する。
size	*str* バッファのサイズ（単位：バイト）。

query_func 関数はライブラリで定義されるコールバック関数で，ユーザ定義プロパティを調べる必要があるときに WinRunner によって呼び出されます。*query_func* 関数はライブラリ定義関数名の場所を確保します。実際の名前はライブラリのモジュール定義（.DEF）ファイル内の EXPORTS ステートメントに指定して，エクスポートしなければなりません。

ユーザ定義プロパティのクエリ関数の例は，67 ページ「ユーザ定義プロパティ関数の例」を参照してください。

ユーザ定義プロパティ用の検証関数の開発

WinRunner では検証関数を使って，ユーザ定義関数が値を必要とするアプリケーション内で GUI オブジェクトを識別します。

検証関数について

検証関数の役割を理解するために，次のシナリオを考えて見てください。［GUI マップエディタ］を開き，ユーザ定義オブジェクトを選択したとします。オブジェクトは class プロパティの object とそのユーザ定義プロパティの *new_property* によって一意に識別されます。*new_property* の値が *Object1* だとします。このオブジェクトの物理的記述は次のようになります。

```
{class:"object", new_property:"Object1"}
```

［GUI マップエディタ］の［表示］ボタンをクリックすると，WinRunner は選択したオブジェクトを強調表示しようとします。しかし，まずその前にその場所を探さなければなりません。

必要なオブジェクトの場所を探すために，WinRunner はアプリケーション内の各オブジェクトを体系的に検査し，このオブジェクトが「object」クラスに属するかどうか問い合わせます。オブジェクトが「object」クラスに属さない場合は，WinRunner は次のオブジェクトに質問をします。そのオブジェクトが「object」クラスに属す場合は，WinRunner はクエリー関数を呼び出して，オブジェクトのユーザ定義プロパティである *new_property* の値を取得します。*new_property* の値を取得すると，WinRunner は検証関数を呼び出し，この値が

期待している値である Object1 と一致するかどうか判定します。検証関数が不一致を検出すると，WinRunner は必要なオブジェクトを探し続けます。クエリー関数が Object1 という値を返した場合，検証関数は適切なオブジェクトが見つかったことを示し，WinRunner はこのオブジェクトを強調表示します。

標準検証関数とユーザ定義検証関数の使用

ユーザ定義プロパティの値を検証する時に使用する関数は，WinRunner の標準プロパティ検証関数の 1 つであっても，ユーザ独自のユーザ定義検証関数であっても構いません。

標準プロパティ検証関数は次のとおりです。

- **string_verify**：ユーザ定義プロパティの値を文字列として比較します。
- **num_verify**：ユーザ定義プロパティの値を数値として比較します。
- **bool_verify**：ユーザ定義プロパティの値をブール式として比較します。

プロパティ値の簡単な比較でユーザ定義プロパティを検証できる場合は，標準関数を使用します。例えば，ユーザ定義プロパティの *new_property* を検査しており，この値が常に「Object1」または「Object2」といった文字列であるとします。このような場合は，標準の **string_verify** 関数を使って，*new_property* プロパティを検査できます。

ユーザ定義検証関数の開発

検証のためにユーザ定義プロパティが複雑な比較を必要とする場合には，標準検証関数では不十分なので，ユーザ独自のユーザ定義検証関数を開発する必要があります。

検証関数の構文は，次のとおりです。

BOOL **verify_func** (HWND *hWnd*, LPSTR *str*);

パラメータ

hWnd	プロパティを検証する対象となる，ウィンドウまたは GUI オブジェクトのハンドル。
str	ユーザ定義プロパティの値を含むバッファ。値はクエリー関数により取得されます。検証関数はこの値をユーザ定義プロパティの要求されている値と比較します。

検証関数は検査が成功すると TRUE を返し，失敗すると FALSE を返さなければなりません。

検証関数は，クエリ関数と同じく C 言語で記述し，コンパイルしてクエリー関数と同じ DLL にリンクします。

ユーザ定義プロパティ用の検証関数の例については，67 ページ「ユーザ定義プロパティ関数の例」を参照してください。

ユーザ定義プロパティの登録

WinRunner で使用できるユーザ定義プロパティを新たに作成するには，そのプロパティに名前を付けて，登録する必要があります。新しいユーザ定義関数を登録するには，**add_record_attr** 関数を使います。この関数の構文は次のとおりです。

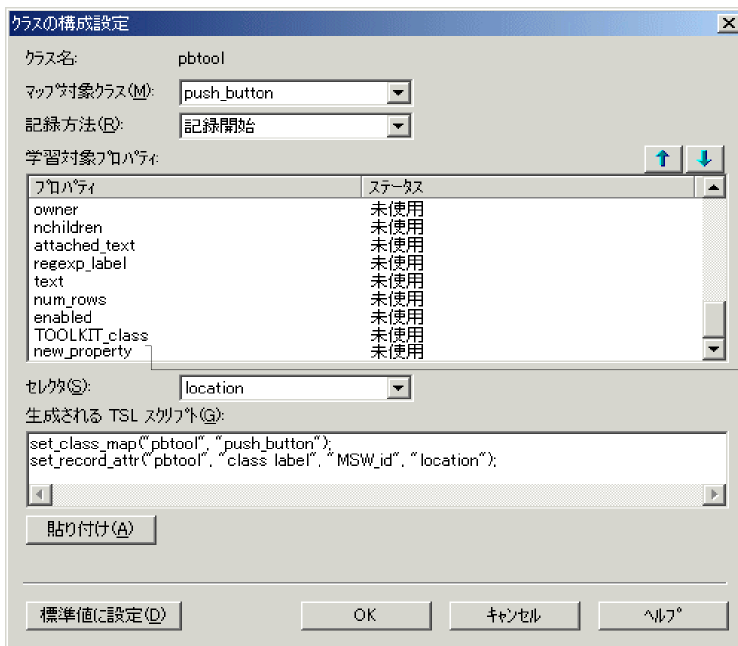
add_record_attr (*attr_name, dll_name, query_func_name, verify_func_name* **);**

- *attr_name* は登録するユーザ定義プロパティの名前です。
- *dll_name* はクエリ関数と検証関数が定義されている DLL の完全パスとファイル名です。
- *query_func_name* は DLL に含まれるクエリー関数の名前です。
- *verify_func_name* は WinRunner の標準プロパティ検証関数の 1 つか，DLL に含まれている検証関数の名前です。

次の例は，ユーザ定義プロパティ「*new_property*」を登録します。

add_record_attr("new_property", "c:¥¥arch¥¥vb_util.dll", "new_property_query", "string_verify");

add_record_attr 関数を実行した後に，新たなユーザ定義プロパティである *new_property* が［クラスの構成設定］ダイアログ・ボックス内の［指定可能なプロパティ］リストに表示されます。新規のプロパティはすべての標準およびユーザ定義 GUI オブジェクト・クラスで使用できます。これを適切なクラスに割り当てなければなりません。



新規ユーザ定義プロパティ，**new_property** が **add_record_attr** 関数を使って追加された。

ユーザ定義プロパティの GUI オブジェクト・クラスへの割り当て

ユーザ定義プロパティを登録したら，これを適切なユーザ定義クラスに割り当てる必要があります。通常は必須プロパティとして割り当てます。プロパティを割り当てるには，**set_record_attr** 関数を使います。［クラスの構成設定］ダイアログ・ボックスを使って，必要な **set_record_attr** 関数を生成し，これをテスト・スクリプトに貼り付けることをお勧めします。

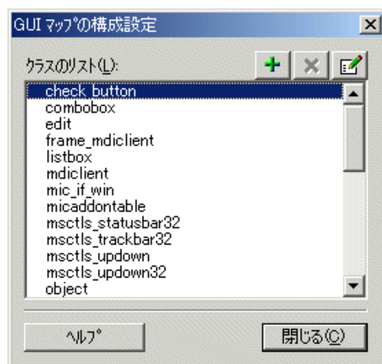
次の例では，ユーザ定義の *new_property* プロパティをユーザ定義の *cust_object* クラスに割り当てます。

```
set_record_attr("cust_object","class, new_property","MSW_id","location");
```

set_record_attr 関数の詳細については，「**TSL オンライン・リファレンス**」を参照してください。

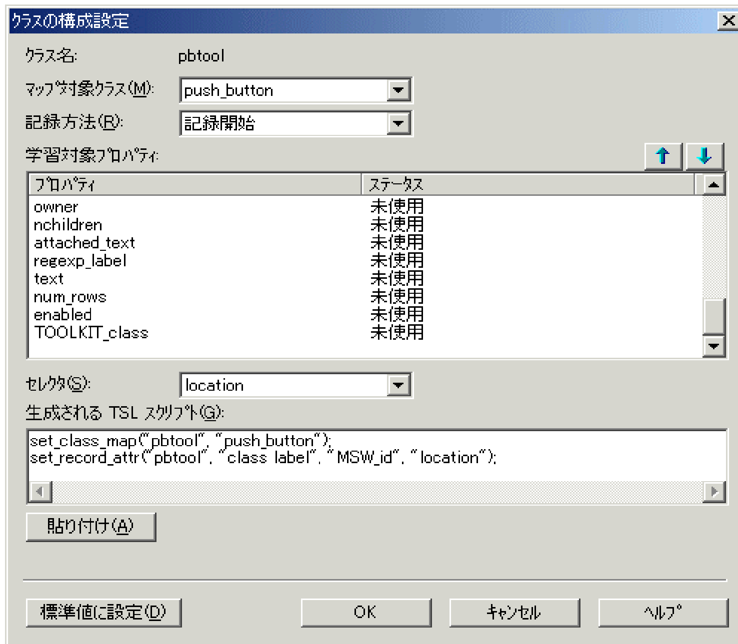
［クラスの構成設定］ダイアログ・ボックスを使って **GUI** オブジェクト・クラスにユーザ定義プロパティを割り当てるには，次のようにします。

1 ［**ツール**］＞［**GUI マップの構成**］をクリックします。［GUI マップの構成設定］ダイアログ・ボックスが開きます。



2 ［**クラスリスト**］で，新しいユーザ定義プロパティを関連付けるクラスを強調表示します。作成したクラスはユーザ定義クラスなので，リスト項目の左端に U の印（ユーザ定義を示す）付きで表示されます。

3 **［構成設定］ボタンをクリックします。**［クラスの構成設定］ダイアログ・ボックスが開きます。



4 **［学習対象プロパティ］リスト内で新規のユーザ定義プロパティを見つけて，選択します。**

5 選択したプロパティのステータスカラムをクリックし，プルダウンメニューから［**必須**］を選択します。［**生成される TSL スクリプト**］ボックスに表示される **set_record_attr** 関数を確かめてください。新規のユーザ定義プロパティが必須プロパティのリストに追加されていることに注意してください。

6 ステータスが［必須］となっているプロパティから，WinRunner がもはや一意の識別情報を必要としない任意のプロパティを見つけて，選択します。

7 必要に応じて，ステータスカラムのプルダウンメニューから，［任意］，［未使用］を選択します。

8 手順 6 と 7 を繰り返して，不要となった必須プロパティをすべて削除します。

注：class プロパティは一意の識別情報に必要がない場合もありますが，［必須］ステータスを変更しないでください。

---

9 ［**貼り付け**］をクリックして，生成した TSL ステートメントをテスト・スクリプトに貼り付けます。

10 ［**OK**］をクリックして，［クラスの構成設定］ダイアログ・ボックスを閉じます。

ユーザ定義プロパティ関数の例

以下にユーザ定義プロパティ関数の構造と実装について説明する例を示します。この例を元にユーザ定義プロパティ関数を実装することもできます。

この例では，プロパティ・クエリー関数は *vb_name_query* という名前です。*vb_name* プロパティは，Visual Basic アプリケーションのコントロールの「Name」プロパティと同じです。

このプロパティの値は VBAPI 関数の VBGetControlName を呼び出すことで取得します。VBGetControlName はウィンドウの「hWnd」を持つ，タスク・スタックを対象に呼び出します。この処理はウィンドウをサブクラス化することで実現します。「hWnd」のウィンドウ・プロシージャは，SetWindowLong を通じて「VbCtrlProc」に変更されます。そして，（特別な）メッセージが「hWnd」に送られます。このメッセージは「hWnd」を作成したタスクのコンテキストにおいて「VbCtrlProc」によって取り上げられます。VbCtrlProc は VBGetControlName を呼び出して，メッセージを処理します。*cust_att.h* ファイルはユーザ定義プロパティ関数の戻り値を設定します。

ファイル名：cust_att.h

```
#define WR_VB_SERVICE_STRING "MY_MESSAGE_IDENTIFIER_1"
enum {
    VB_GET_CTRL_NAME_IND,
    VB_GET_CTRL_INDEX_IND,
};

typedef struct {
    HWND hWnd;
```

```
    LPSTR value;
} VB_GET_CTRL_NAME_STRUCT;

typedef VB_GET_CTRL_NAME_STRUCT FAR* LPVBGCNS;
```

ファイル名：cust_att.c

```
#define EXPORTED _far _pascal __export
#include <windows.h>
#include <vbapi.h> // VBGetHwndControl と VBGetControlName 用。
#include "cust_att.h"

LRESULT EXPORTED VbCtrlProc(HWND hWnd,UINT message,WPARAM
wParam,LPARAM lParam);

HANDLE hmodDLL;

static UINT WR_VB_SERVICE_MSG = 0;

static FARPROC def_proc;

int FAR PASCAL LibMain
(
 HANDLE hModule,
 WORD wDataSeg,
 WORD cbHeapSize,
 LPSTR lpszCmdLine
)
{
 hmodDLL = hModule;
    WR_VB_SERVICE_MSG =
RegisterWindowMessage(WR_VB_SERVICE_STRING);
 return 1;
}

void EXPORTED vb_name_query(HWND hWnd, LPSTR value, int length)
{
    VB_GET_CTRL_NAME_STRUCT name_struct = {hWnd, value};
```

```
    def_proc = (FARPROC)SetWindowLong(hWnd, GWL_WNDPROC,
(LONG)VbCtrlProc);
    SendMessage(hWnd, WR_VB_SERVICE_MSG,
(WPARAM)VB_GET_CTRL_NAME_IND,
(LPARAM)(LPVBGCNS)&name_struct);
    SetWindowLong(hWnd, GWL_WNDPROC,(LONG)def_proc);
}

LRESULT EXPORTED VbCtrlProc(
    HWND hWnd,
    UINT message,
    WPARAM wParam,
    LPARAM lParam
)
{

    if(message == WR_VB_SERVICE_MSG) {
        switch(wParam) {
            case VB_GET_CTRL_NAME_IND:
            {
                LPVBGCNS p_name_struct = (LPVBGCNS)lParam;
                HCTL hctl = VBGetHwndControl(p_name_struct->hWnd);
                if(hctl)
                    VBGetControlName(hctl,p_name_struct->value);
                return((LRESULT)TRUE);
            }
            default:
                return((LRESULT)TRUE);
        }
    }
    return(CallWindowProc(def_proc, hWnd, message, wParam, lParam));
}

int EXPORTED vb_tag_query(HWND win, LPSTR value, int len)
{
    vb_name_query(win, value, len);
    if(value[0] == '¥0')
        return(FALSE);
```

```
    return(TRUE);
}
```

第 7 章

割り当てられる論理名のカスタマイズ

WinRunner がユーザ定義 GUI オブジェクトに論理名を割り当てる方法をカスタマイズできます。

本章では，以下の項目について説明します。

- 論理名関数について
- 論理名関数の開発
- 論理名関数のユーザ定義 GUI オブジェクト・クラスへの関連付け

割り当てられる論理名のカスタマイズについて

論理名関数を実装することで，WinRunner がアプリケーション内のユーザ定義 GUI オブジェクトに割り当てる論理名を制御できます。例えば，アプリケーションに WinRunner のどの標準 GUI オブジェクト・クラスにも属さないスピンボタンが含まれるとします。スピンボタンは劇場で売れたチケット数のカウンタを操作します。スピンボタンをクリックすると，WinRunner は次のようなステートメントを記録します。

```
obj_mouse_click ("SpinButton_2", 150, 300, LEFT);
```

SpinButton_2 という論理名は，特にアプリケーションにたくさんのスピンボタンが含まれる場合には，どの GUI オブジェクトが操作されているか，ほとんど分かりません。しかし論理名関数を実装することで，WinRunner がカウンタに割り当てる論理名の記述を改善できます。ユーザ定義の論理名関数を使って，「SpinButton_2」の代わりに「Tickets_Sold_(spin)」という論理名を返すようにすれば，次のような記録ステートメントが記録されます。

```
obj_mouse_click ("Tickets_Sold_(spin)", 150, 300, LEFT);
```

カスタマイズされた論理名からは，記録されたステートメントが参照している GUI オブジェクトがすぐに分かります。

論理名関数を実装するには，次の手順を実行します。

1 ユーザ定義オブジェクト・クラスの論理名を生成する論理名関数を開発します。

2 論理名関数をユーザ定義 GUI オブジェクト・クラスに関連付けます。

［GUI マップエディタ］を使って，任意の GUI オブジェクトの論理名を変更できますが，以下の点に注意してください。

➤ ［GUI マップエディタ］は，WinRunner によって論理名がすでに割り当てられている場合のみ使用可能です。

➤ ［GUI マップエディタ］を使う場合には，必要に応じて「**各**」論理名を「**手動**」で変更しなければなりません。

一方，論理名関数は実際に WinRunner が必要とするときに，はじめて論理名を生成し，1 つの論理名関数で特定のユーザ定義クラス内のすべての GUI オブジェクトに対して論理名を生成できます。

ユーザ定義記録関数をユーザ定義 GUI オブジェクト・クラスの論理名関数と組み合わせることができます。こうすることで，WinRunner がテスト・スクリプトに記録するステートメントが直感的なものになります。つまり，記録されたステートメントは，実行された操作と，その操作対象のオブジェクトの両方を表せます。詳細については，第 5 章「記録されるステートメントのカスタマイズ」を参照してください。

論理名関数について

GUI オブジェクトが GUI マップに追加されると，または［GUI スパイ］を使って GUI オブジェクトに関連付けられたプロパティを表示すると，WinRunner はオブジェクトに記述的な論理名を割り当てようとします。そのため，GUI オブジェクトに関連付けられたテキスト（プッシュボタンのラベルなど）がある場合は，WinRunner はそのテキストを元にして論理名を割り当てます。ほとんどの場合，これで GUI オブジェクトをよく表す論理名が割り当てられます。

GUI オブジェクトに付属のテキストがない場合，WinRunner は *MSWclass* プロパティを元に論理名を割り当てますが，割り当てられる論理名がオブジェクトを十分に表現しないものになる場合があります。その結果，記録されるテスト・スクリプト・ステートメントも，対応する GUI オブジェクトを明確に表さないものになります。論理名関数を実装すれば，WinRunner はユーザ定義 GUI オブジェクトに対して，より直感的な論理名を生成して，記録することができます。

実装する論理名関数は，様々な方法で必要な記述的論理名を生成できます。開発に使用しているアプリケーションと開発環境に応じて，方法を選択します。最も簡単な方法は，アプリケーションを開発するときに，そのアプリケーションで使用されるすべての GUI オブジェクトの記述的な詳細を含むデータベースを作成するプログラマによって提供されます。こうしたデータベースを発見し，アクセスすることができるなら，ユーザ定義論理名の理想的な情報源となるでしょう。

論理名関数の開発

論理名関数は C 言語形式で記述し，コンパイルして DLL にリンクします。論理名関数を GUI オブジェクト・クラスに関連付けるときに，DLL の名前と場所を指定します。詳細については，74 ページ「論理名関数のユーザ定義 GUI オブジェクト・クラスへの関連付け」を参照してください。

論理名関数の構文は，次のとおりです。

int **function** (HWND *hWnd*, char* *str*, int *size*);

hWnd	論理名を生成する対象となる GUI オブジェクトのハンドル。
str	論理名を格納するために WinRunner によって割り当てられているバッファ。論理名関数は生成された論理名を *str* パラメータに格納します。
size	*str* バッファのサイズ（単位：バイト）。

ユーザ定義オブジェクトの論理名関数の具体例については，第 6 章「GUI オブジェクトへのユーザ定義プロパティの追加」の最後の例を参照してください。

論理名関数のユーザ定義 GUI オブジェクト・クラスへの関連付け

add_cust_record_class 関数を使って，論理名関数を GUI オブジェクト・クラスに関連付けます。**add_cust_record_class** 関数の構文は次のとおりです。

add_cust_record_class (*MSW_class*, *dll_name* **[,** *rec_func* **]**
[, *log_name_func* **]);**

- *MSW_class* は論理名関数が関連付けられているユーザ定義クラスです。ユーザ定義クラスはすでに定義されていなくてはなりません。
- *dll_name* は論理名関数を含む DLL の完全パスとファイル名です。ユーザ定義記録関数が GUI オブジェクト・クラス用に存在する場合には，その関数は同じ DLL に含まれます。詳細については，第 5 章「記録されるステートメントのカスタマイズ」を参照してください。
- *rec_func* は DLL に含まれる記録関数（存在する場合）の名前です。記録関数はテスト・スクリプトに記録される文字列を返します。詳細については，第 5 章「記録されるステートメントのカスタマイズ」を参照してください。

➤ *log_name_func* は DLL に含まれる論理名関数の名前です。*log_name_func* 関数は *MSW_class* クラスのユーザ定義 GUI オブジェクトにユーザ定義論理名を指定します。

次の例では，**add_cust_record_class** 関数は論理名関数である vb_log_name をユーザ定義の SpinButton クラスに関連付けます。

```
add_cust_record_class ("SpinButton", "c:¥¥winrun¥¥arch¥¥vb_util.dll", " ",
    "vb_log_name" );
```

ユーザ定義記録関数の論理名関数との関連付け

ユーザ定義記録関数を論理名関数と一緒に実装することで，テスト・スクリプトを一層読みやすくできます。**add_cust_record_class** 関数を使って，ユーザ定義記録関数をユーザ定義 GUI オブジェクトの論理名関数と組み合わせます。この組み合わせによって，WinRunner がテスト・スクリプトに記録するステートメントは実行された操作とその操作対象のオブジェクトの両方を表せます。詳細については第 5 章「記録されるステートメントのカスタマイズ」を参照してください。

ユーザ定義プロパティ関数と組み合わされる論理名関数の例については，第 6 章「GUI オブジェクトへのユーザ定義プロパティの追加」を参照してください。

第 3 部

WinRunner API の使い方

第 8 章

Mercury の API 関数

本章では，必要な WinRunnerAPI（Application Programmer Interface）関数をすべて紹介します。

本章では，以下の項目について説明します。

- AUT 関数の型の定義
- マクロ

関数は上記の順番で示します。マクロのグループでは，関数はアルファベット順に並べられています。一部の関数は WinRunner エラー・コードを返すか，参照します。エラー・コードの詳細については「**TSL オンライン・リファレンス**」を参照してください。

API 関数について

API の関数にはいくつかの型があります。

「**型**」宣言はアプリケーションで実装した実行関数がどのような構造になっているのかを示します。内部テストで必要な型の定義は **MicFunctionProc** のみです。

マクロはアプリケーションで実装する関数で使えます。次のマクロがあります。

- **mic_cp_from_string**
- **mic_cp_to_string**
- **mic_destroy_buf**
- **mic_destroy_string**
- **mic_get_object**
- **mic_init_buf**
- **mic_init_string**
- **mic_set_object**

マクロでは以下の型を使います。

- MicString
- MicObjectBuf
- MicObject

MicFunctionProc

任意の再生関数のプロトタイプを記述します（関数の型）。

typedef int (MIC_PROTOTYPE ***MicFunctionProc**)(MicArgList *args*);

MicRegisterFunction により mic_if に登録されます。

引数

args	MicArgList はこの関数に渡されるすべてのパラメータを格納している構造体を指すポインタです。

説明

MicFunctionProc はツールキットによって実装される，TSL 再生コマンドに対応する任意の関数の型です（ツールキットは，テストしているアプリケーションの開発に使用する開発環境です）。関数は入 / 出力パラメータをいくつでも持つことができます。パラメータはデータ構造体として渡されます。パラメータを取得するには，ツールキットは **MicExtractArgs** を使わなくてはなりません。

戻り値

この関数は，成功すると MIC_E_OK を返し，失敗すると WinRunner のエラー・コードのいずれかを返します。エラー・コードの一覧については「**TSL オンライン・リファレンス**」を参照してください。

mic_cp_from_string

MicString を文字列にコピーします（マクロ）。

#include <mic_if.h>

mic_cp_from_string (*str*, *str_name*);

パラメータ

str	MicString のコピー先文字列。
str_name	MicString。

mic_cp_to_string

文字列を以前に初期化された MicString にコピーします（マクロ）。

#include <mic_if.h>

mic_cp_to_string (*str_name*, *str*);

パラメータ

str_name	MicString。
str	MicString にコピーする文字列。

mic_destroy_buf

MicObjectBuf 型のオブジェクト・バッファを解放します（マクロ）。

#include <mic_if.h>

mic_destroy_buf (*buf_name*);

パラメータ

buf_name	解放するバッファ。

mic_destroy_string

MicString 型の文字列を解放します（マクロ）。

#include <mic_if.h>

mic_destroy_string (*str_name*);

パラメータ

str_name	解放する文字列。

mic_get_object

MicObjectBuf 型のオブジェクト・バッファからオブジェクトを返します（マクロ）。

#include <mic_if.h>

mic_get_object (*buf_name*, *index*);

パラメータ

buf_name	GUI オブジェクトの取得元のバッファ。
index	バッファ内の GUI オブジェクトのインデックス。0 から始まります。

説明

このマクロはオブジェクト・バッファ内の 0（ゼロ）を基準にした位置 *index* にある GUI オブジェクトを返します。

mic_init_buf

MicObjectBuf 型のバッファを初期化します（マクロ）。

#include <mic_if.h>

mic_init_buf (*buf_name*);

パラメータ

buf_name	初期化するバッファ。

説明

このマクロは *buf_name* という名前のバッファを割り当てます。バッファを操作するために，次のマクロも使用できます。
mic_set_object, mic_get_object, mic_destroy_buf。

mic_init_string

MicString 型の文字列を初期化します（マクロ）。

#include <mic_if.h>

mic_init_string (*str_name*);

パラメータ

str_name	初期化する文字列。

説明

このマクロは文字列を割り当てます。文字列を操作するために，次のマクロも使用できます。
mic_cp_to_string, mic_cp_from_string, mic_destroy_string。

mic_set_object

MicObject 型のオブジェクトを，MicObjectBuf のオブジェクト・バッファに設定します。

#include <mic_if.h>

mic_set_object (*buf_name*, *index*, *object*);

パラメータ

buf_name	オブジェクトを入れるバッファ。
index	バッファ内のオブジェクトのインデックス。
object	設定するオブジェクト。

説明

このマクロはオブジェクト・バッファ内の 0（ゼロ）を基準にした位置 *index* にオブジェクトを入れます。バッファを操作するために，次のマクロも使用できます。

mic_set_object, mic_get_object, mic_destroy_buf

索引

A
Acrobat Reader vi
add_cust_record_class 関数 51, 74
add_record_attr 関数 63
add_record_mesage 関数 50
API 関数 79–85

D
default_check_function, 構文 33
display_function, 構文 35

G
gui_ver_add_check_to_class 関数 19
gui_ver_add_check 関数 18
gui_ver_add_class 関数 25
gui_ver_set_default_checks 関数 21
GUI 検査, 応用 31–39
  新しいオブジェクト・クラスの追加 32
  新しい検査の登録 34
  キャプチャ関数の作成 33
  サンプル・テスト・スクリプト 35
  比較関数の作成 33
  標準の検査の設定 35
GUI 検査, 標準のオブジェクト
  新しい検査の登録 18
  カテゴリへの関数の追加 19
  キャプチャ関数の作成 12
  比較関数の作成 15
  標準の検査の変更 20
GUI 検査, ユーザ定義オブジェクト 23–29
  新しいクラスの追加 25
  新しい検査の登録 29
  キャプチャ関数の作成 28
  検査とクラスの関連付け 29
  比較関数の作成 28
  標準検査の設定 29

M
Mercury API 79–85
mic_cp_from_string 関数 82
mic_cp_to_string 関数 82
mic_destroy_buf 関数 82
mic_destroy_string 関数 83
mic_get_object 関数 83
mic_init_buf 関数 84
mic_init_string 関数 84
mic_set_object 関数 85
MicFunctionProc 関数 81
MicObjectBuf 型 80, 82, 83, 84, 85
MicObject 型 80, 85
MicString 型 80, 82, 83, 84

S
set_record_attr 関数 65

T
TSL オンライン・リファレンス vi
TSL リファレンス・ガイド vi

U
ui_function, 構文 32

W
WinRunner
  オンライン・リソース vi
  コンテキスト・センシティブ・ヘルプ vi
  サンプル・テスト vii
WinRunner インストール・ガイド vi
WinRunner 基本機能ユーザーズ・ガイド

vi
WinRunner 上級機能ユーザーズ・ガイド vi
WinRunner チュートリアル vi
WinRunner の新機能 vi

お
オンライン・ヘルプ vi
オンライン・リソース vi

き
キャプチャ関数
構文 12
作成 12, 28, 33
記録関数，カスタマイズ 43–58
GUI オブジェクト・クラスと関連付ける 51
Windows メッセージの追加 50
開発 47
実行関数の開発 51
戻り値 49

さ
最初にお読みください vi
サポート・オンライン vii
サンプル・テスト vii

て
テスト，サンプル vii

ひ
比較関数
構文 15
作成 15, 28, 33
表記規則 vii
表示ボタン，GUI チェック・ダイアログ・ボックス 34
標準のオブジェクト，検査の作成 9–21

ふ
プロパティ，ユーザ定義 59–67
GUI オブジェクトクラスへの割り当て 65
クエリー関数の開発 60
検証関数の開発 61
登録 63
プロパティ・リスト・ボタン，GUI チェックポイント・ダイアログ・ボックス 25, 33

ゆ
ユーザ定義オブジェクト，検査の作成 23–29

ろ
論理名関数 71–75
GUI オブジェクトクラスとの関連付け 74
開発 74
記録関数との組み合わせ 75