Rapport d'alternance - Tests automatisés avec Ranorex Studio

Dans le cadre de mon alternance au sein de l'entreprise EBP Informatique, j'ai été amené à travailler sur l'automatisation des tests logiciels, un élément essentiel dans le processus de développement d'applications.

L'automatisation des tests consiste à exécuter des scénarios de test de manière automatique afin de vérifier le bon fonctionnement d'un logiciel. Contrairement aux tests manuels, cette approche permet de rejouer rapidement et de façon répétée des cas de test, sans intervention humaine. Elle est particulièrement utile dans les phases de validation et de maintenance.

Les principaux avantages de cette méthode sont multiples. Elle permet un gain de temps considérable, notamment pour les tests récurrents. Elle réduit également les risques d'erreurs humaines et améliore la fiabilité globale des validations. De plus, elle facilite la détection des anomalies lors des mises à jour du logiciel (tests de non-régression).

Pour réaliser ces tests automatisés, j'ai utilisé l'outil Ranorex Studio. Ce logiciel offre la possibilité d'enregistrer les actions d'un utilisateur sur une interface graphique, telles que les clics, les saisies clavier ou encore la navigation entre différentes fenêtres. Ces actions sont ensuite rejouées automatiquement pour simuler un comportement utilisateur.

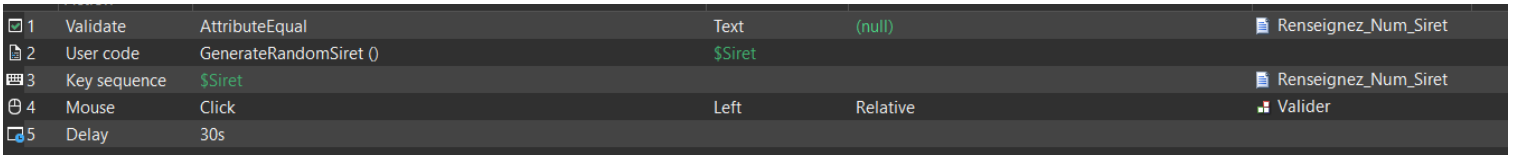
Le fonctionnement de Ranorex repose sur des enregistrements appelés *recordings*. Chaque *recording* correspond à un scénario de test complet, contenant l'ensemble des étapes nécessaires à son exécution. Ces enregistrements peuvent être modifiés et enrichis afin de s'adapter à différents contextes.

En complément, l'outil permet d'intégrer du code en langage C#, ce qui offre une plus grande flexibilité. Cela permet notamment de gérer des cas complexes, comme l'utilisation de conditions, de boucles, ou encore la vérification dynamique de données.

Les tests automatisés ont été réalisés sur le logiciel EBP Comptabilité, développé par l'entreprise. Ces tests portaient sur des fonctionnalités essentielles, telles que la saisie d'écritures comptables, la navigation dans l'interface ou encore la génération de documents.

Ce projet m'a permis de mieux appréhender les enjeux liés à la qualité logicielle, ainsi que le rôle central des tests dans le cycle de développement. Il m'a également permis de développer des compétences techniques en automatisation, en utilisation d'outils spécialisés et en programmation, tout en renforçant ma rigueur et ma capacité d'analyse.

Premier test - Création d’un dossier sur le logiciel EBP Comptabilité



1	Validate	AttributeEqual	Text	(null)	Renseignez_Num_Siret
2	User code	GenerateRandomSiret ()	$Siret		
3	Key sequence	$Siret			Renseignez_Num_Siret
4	Mouse	Click	Left	Relative	Valider
5	Delay	30s			

Le premier test est un test simple qui permet de créer un dossier dans le logiciel.

Ce test simule exactement les actions d’un utilisateur réel.

Le test commence par vérifier que le champ du numéro SIRET est vide. Cette étape est importante pour s’assurer que le test démarre dans de bonnes conditions. Ensuite, un numéro SIRET est généré automatiquement grâce à un UserCode écrit en C#.

Le numéro de SIRET généré est ensuite saisi dans le champ prévu. Cette étape permet de tester la saisie utilisateur. Le test clique ensuite sur le bouton de validation pour lancer la création du dossier.

Après le clic, un temps d’attente est ajouté. Ce délai permet au logiciel de traiter la demande. Sans ce délai, le test pourrait continuer trop vite et provoquer des erreurs.

Cette façon de créer des tests automatisés sur Ranorex est très rapide à mettre en place. Il est très utile pour tester des fonctionnalités basiques et ne demande pas de connaissances en programmation.

Cependant, ce type de test a des limites. Il dépend beaucoup de l’interface du logiciel. Si un bouton change de place ou si un nom change, le test peut ne plus fonctionner. Il est aussi difficile de vérifier des informations complexes.

Deuxième test - Vérifier les messages d’informations du logiciel EBP Comptabilité

Ce UserCode a été créé pour compléter un test Ranorex réalisé sur le logiciel EBP Comptabilité. Son rôle principal est de vérifier automatiquement les rapports générés après l'exécution de traitements. Dans EBP, certaines opérations peuvent se terminer de manière asynchrone, c'est-à-dire que le logiciel continue à travailler en arrière-plan puis affiche ensuite un rapport de fin de traitement. Un simple enregistrement Ranorex ne suffit pas toujours à contrôler correctement ces fenêtres, car elles peuvent apparaître avec un délai, changer de position ou contenir plusieurs messages à lire.

Le code apporte donc une logique plus fiable qu'une suite d'actions enregistrées. Il ouvre la zone des traitements terminés, repère les rapports disponibles, les ouvre un par un, lit leur contenu, les ferme, puis conserve tous les messages trouvés dans une variable commune. Cette variable est ensuite utilisée pour vérifier qu'un message attendu est bien présent. L'objectif est d'éviter une validation visuelle manuelle et de rendre le test plus stable et plus réutilisable.

Dans un test automatisé classique, Ranorex clique sur des éléments précis de l'interface à partir de chemins ou de positions. Le problème est que l'interface d'EBP peut se modifier après chaque fermeture de rapport. Les boutons peuvent bouger légèrement, le contenu peut se recharger, et une même information peut être détectée plusieurs fois par Ranorex à cause de la structure interne de l'application.

Le UserCode répond à cette difficulté en ne se basant pas uniquement sur une position fixe. Il recherche les éléments dans l'interface à chaque étape, vérifie leur texte ou leur nom accessible, supprime les doublons détectés et utilise un compteur pour savoir quel rapport traiter ensuite. Cette méthode est plus robuste, car elle s'adapte mieux aux petits changements d'affichage.

Le fonctionnement

Le traitement commence par l'ouverture de la fenêtre des traitements terminés depuis la barre de statut du logiciel. Le code attend ensuite que la fenêtre soit réellement chargée, en recherchant la présence d'un texte contenant l'idée de traitement terminé. Cette attente évite de cliquer trop tôt, ce qui est une cause fréquente d'échec dans les tests automatisés.

Une fois la fenêtre disponible, le code lance une boucle. À chaque passage, il cherche le prochain rapport à ouvrir. Quand un rapport est trouvé, il clique sur son bouton, attend l'apparition de la fenêtre de rapport, lit les informations visibles, fait défiler le contenu si nécessaire, puis ferme la fenêtre. La boucle continue tant qu'il reste des rapports, avec une limite de sécurité pour éviter une boucle infinie.

La lecture et la récupération des messages

La partie la plus importante du UserCode est la lecture des rapports. Le code parcourt les éléments affichés dans la fenêtre ErrorsDialog et récupère les textes disponibles dans plusieurs attributs, comme Text, Caption ou AccessibleName. Cette approche augmente les chances de récupérer les messages, car selon le type de composant graphique, l'information peut être stockée dans un attribut différent.

Pour éviter d'enregistrer plusieurs fois la même ligne, le code utilise une collection qui n'accepte pas les doublons. Il conserve aussi l'ordre de lecture dans une liste séparée, afin que le contenu final reste compréhensible. Les lignes trop courtes ou uniquement composées de chiffres sont ignorées, car elles ne correspondent généralement pas à des messages utiles pour la validation du test.

La gestion du défilement dans les rapports longs

Certains rapports peuvent contenir plus de lignes que ce qui est visible à l'écran. Le code prévoit donc un défilement automatique. Il commence par se placer en haut du rapport, lit les premières lignes, puis descend progressivement page par page. Après chaque défilement, il relit les éléments visibles et ajoute uniquement les nouvelles lignes trouvées.

La boucle de défilement s'arrête lorsque l'ascenseur vertical arrive en bas ou lorsque plusieurs passages ne trouvent plus de nouvelle information. Cette double sécurité permet d'éviter de rester bloqué si la barre de défilement n'est pas correctement détectée. En complément, le code effectue une dernière lecture après un raccourci Ctrl+End pour s'assurer que la fin du rapport a bien été analysée.

La vérification finale du message attendu

Après la lecture de tous les rapports, les textes sont regroupés dans la variable ContenuRapports. Cette variable joue le rôle de mémoire temporaire pendant l'exécution du test. Elle permet à un autre recording Ranorex d'utiliser le résultat déjà lu sans rouvrir les rapports.

La méthode de vérification reçoit un message attendu en paramètre. Si ce texte est présent dans ContenuRapports, le test affiche un succès dans le rapport Ranorex. Sinon, il affiche un échec. Cette étape transforme donc le contenu des rapports EBP en véritable critère de validation automatique.

La gestion des erreurs et la stabilité du test

Le code contient de nombreux blocs try/catch. Leur objectif n'est pas de cacher les erreurs, mais d'éviter qu'un petit problème d'interface bloque tout le scénario. Par exemple, si un bouton n'est pas trouvé, le code écrit un avertissement dans le rapport Ranorex et tente de continuer quand c'est possible. Si une fenêtre ne peut pas être fermée normalement, il utilise une fermeture de secours avec Alt+F4.

Le code ralentit aussi les actions de la souris et du clavier. Cette décision améliore la stabilité du test, surtout dans une application de gestion comme EBP où les fenêtres peuvent mettre un certain temps à s'afficher. Les délais volontairement ajoutés permettent à l'interface de se mettre à jour avant que Ranorex ne poursuive les actions.

L’intérêt du UserCode dans le projet

Ce UserCode apporte une vraie valeur au projet de test, car il automatise une vérification qui serait difficile à réaliser proprement avec un simple enregistrement. Il rend le scénario plus fiable, plus lisible dans les rapports Ranorex et plus adapté aux traitements asynchrones.

Pour résumer, ce UserCode permet de traiter automatiquement les rapports de fin de traitement dans EBP Comptabilité. Il ouvre la fenêtre des traitements terminés, identifie les rapports disponibles, lit leur contenu même lorsqu'il faut faire défiler la fenêtre, regroupe les messages trouvés et vérifie ensuite la présence d'un message attendu.

Le code améliore la fiabilité du test Ranorex car il tient compte des problèmes fréquents d'une interface dynamique : délais d'affichage, repositionnement des boutons, doublons, rapports longs et fenêtres qui ne se ferment pas toujours de la même manière. Il permet donc d'obtenir une validation automatique plus solide et plus professionnelle.

TraiterTousLesRapports	Lance le traitement complet et prépare la lecture des rapports.
TraiterProchainRapport	Repère le prochain rapport disponible et l’ouvre.
LireEtFermerRapport	Récupère les lignes du rapport, gère le scroll et ferme la fenêtre.
VerifierContenuRapports	Contrôle que le message attendu a bien été trouvé.
ContenuRapports	Stocke tous les textes récupérés pendant le test.

Captures d'écran du UserCode de la vérification des messages d'informations du logiciel EBP Comptabilité

```
/*
 * Created by Ranorex
 * User: samuel.tardy
 * Date: 16/01/2026
 *
 * To change this template use Tools > Options > Coding > Edit standard headers.
 */
using System;
using System.Linq;
using Ranorex;
using Ranorex.Core;
using Ranorex.Core.Testing;

namespace Ranorex_Compta_EtatsFi_Immo.UserCode_Collection
{
    /// <summary>
    /// Gère le traitement automatique de plusieurs rapports Ranorex asynchrones (ErrorsDialog).
    /// Parcourt les rapports affichés dans le ThreadForm, lit leur contenu et les ferme un par un.
    /// </summary>
    [UserCodeCollection]
    public class Verification_Messages_Asynchrones
    {
        /// <summary>
        /// Nombre de rapports déjà traités dans la session courante.
        /// On utilise un compteur plutôt que des positions car l'UI peut se redessiner
        /// (les boutons bougent légèrement) après chaque fermeture de rapport.
        /// </summary>
        public static int nombreRapportsTraites = 0;

        /// <summary>
        /// Contient la concaténation de tous les textes trouvés dans les rapports.
        /// Persiste entre les appels (accessible depuis un autre recording).
        /// </summary>
        public static string ContenuRapports = "";

        /// <summary>
        /// Point d'entrée principal : traite tous les rapports disponibles un par un.
        /// Réinitialise le compteur et le contenu accumulé avant de démarrer.
        /// </summary>
        [UserCodeMethod]
        public static void TraiterTousLesRapports()
        {
            try
            {
                Report.Info("Début du traitement des rapports");
                // Réinitialise le compteur des rapports déjà traités et le contenu accumulé
                nombreRapportsTraites = 0;
                Verification_Messages_Asynchrones.ContenuRapports = "";

```

```
// Ralentit la souris et le clavier pour plus de stabilité
Mouse.DefaultMoveTime = 300;
Keyboard.DefaultKeyPressTime = 100;

// Ouvre la fenêtre des traitements terminés via le bouton de la barre de statut
try
{
    Button boutonTraitements = Host.Local.FindSingle<Button>(
        "/form[@controlname='HostForm']/element[@controltypename='BarDockControl']/?/?/statusbar[@accessiblename='Status']/button[@accessiblename='Traitements terminés' and @text='Traitements terminés
        45000);
        boutonTraitements.Click();
    Delay.Milliseconds(1000);
}
catch (Exception exOpen)
{
    Report.Warn("Impossible de cliquer sur 'Traitements terminés' : " + exOpen.Message);
}

// Recherche le conteneur principal (ThreadForm > layoutManager1)
Container layoutManager = null;
try
{
    layoutManager = Host.Local.FindSingle<Container>(
        "/form[@controlname='ThreadForm']/container[@caption='layoutManager1']",
        10000);
}
catch { }

if (layoutManager == null)
{
    Report.Error("Layout Manager introuvable - vérifier que ThreadForm est ouvert");
    return;
}

// Attend que le contenu soit chargé (présence d'au moins un élément 'terminé')
for (int attente = 0; attente < 20; attente++)
{
    bool contenuCharge = false;
    foreach (var e in layoutManager.Find<Unknown>(".//"))
    {
        try
        {
            foreach (var attr in new[] { "Text", "Caption", "AccessibleName" })
            {
                try
                {
                    var v = e.Element.GetAttributeValueText(attr);
                    if (!string.IsNullOrWhiteSpace(v) &&
                        v.Trim().IndexOf("termin", StringComparison.OrdinalIgnoreCase) >= 0)
                    { contenuCharge = true; break; }
                }
                catch { }
            }
            if (contenuCharge) break;
        }
        catch { }
    }
    if (contenuCharge) break;
    Delay.Milliseconds(500);
}

int count = 0;
int maxIterations = 50; // Sécurité : maximum 50 rapports pour éviter une boucle infinie

// Traite les rapports tant qu'il en reste et qu'on n'a pas atteint la limite
while (count < maxIterations && TraiterProchainRapport(layoutManager))
{
    count++;
    Report.Info($"Rapport {count} traité");
    Delay.Milliseconds(500); // Petite pause entre chaque rapport
}

// Affiche un bilan final du traitement
Report.Success(count > 0 ? $"Total: {count} rapport(s) traité(s)" : "Aucun rapport terminé trouvé");
Report.Info($"ContenuRapports accumulé:{Environment.NewLine}{Verification_Messages_Asynchrones.ContenuRapports}");

// Clique sur "Cacher les traitements terminés" avant de fermer la fenêtre ThreadForm
try
{
    Button boutonCacher = Host.Local.FindSingle<Button>(
        "/form[@controlname='ThreadForm']/container[@caption='layoutManager1']/element[@controlname='closeAllButton']/button[@accessiblename>'Cacher les traitements te']",
        3000);
    Report.Info("Clic sur 'Cacher les traitements terminés'");
    boutonCacher.Click();
    Delay.Milliseconds(500);
}
```

```
            catch (Exception exCacher)
            {
                Report.Warn("Impossible de cliquer sur 'Cacher les traitements terminés' : " + exCacher.Message);
            }

            // Ferme la fenêtre des traitements terminés (ThreadForm) une fois tous les rapports lus
            try
            {
                Form threadForm = null;
                if (Host.Local.TryFindSingle("/form[@controlname='ThreadForm']", 3000, out threadForm))
                {
                    Report.Info("Fermeture de la fenêtre ThreadForm");
                    threadForm.Close();
                    Delay.Milliseconds(500);
                }
                else
                {
                    Report.Info("ThreadForm déjà fermée ou introuvable");
                }
            }
            catch (Exception exClose)
            {
                Report.Warn("Impossible de fermer ThreadForm : " + exClose.Message);
            }
        }
        catch (Exception ex)
        {
            Report.Error("Erreur: " + ex.Message);
        }
    }

    /// <summary>
    /// Recherche et traite le prochain rapport disponible.
    /// Retourne true si un rapport a été traité, false s'il n'y en a plus.
    /// Stratégie : pour chaque bouton "Rapport" (closeButton) non encore traité,
    /// vérifie qu'il y a bien un "terminé" en dessous, puis clique dessus.
    /// Les éléments "terminé" et les boutons sont déduplifiés par proximité de position
    /// (tolérance ±10 px) pour éviter les faux doublons UI.
    /// </summary>
    /// <param name="layoutManager">Conteneur principal ThreadForm > layoutManager1</param>
    /// <returns>True si un rapport a été traité, False sinon</returns>
    public static bool TraiterProchainRapport(Container layoutManager)
```

```
178         {
179             try
180             {
181                 // 1. Collecte et déduplification des positions "terminé" (tolérance ±10 px)
182                 var posTerminesUniques = new System.Collections.Generic.List<int>();
183                 foreach (var e in layoutManager.Find<Unknown>(".//"))
184                 {
185                     try
186                     {
187                         foreach (var attr in new[] { "Text", "Caption", "AccessibleName" })
188                         {
189                             try
190                             {
191                                 var v = e.Element.GetAttributeValueText(attr);
192                                 if (!string.IsNullOrWhiteSpace(v) &&
193                                         v.Trim().IndexOf("termin", StringComparison.OrdinalIgnoreCase) >= 0)
194                                 {
195                                     int top = e.Element.ScreenRectangle.Top;
196                                     // N'ajoute que si aucune position similaire n'existe déjà (±10 px)
197                                     if (!posTerminesUniques.Any(t => Math.Abs(t - top) <= 10))
198                                         posTerminesUniques.Add(top);
199                                     break;
200                                 }
201                             }
202                             catch { }
203                         }
204                     }
205                     catch { }
206                 }
207 
208                 Report.Info($"Éléments 'terminé' trouvés (déduplifiés): {posTerminesUniques.Count}");
209                 if (posTerminesUniques.Count == 0) return false;
210 
211                 // 2. Collecte et déduplification des boutons closeButton (tolérance ±10 px)
212                 var boutonRapports = new System.Collections.Generic.List<System.Tuple<int, int, Unknown>>();
213                 foreach (var b in layoutManager.Find<Unknown>(".//element[@controlname='closeButton']"))
214                 {
215                     try
216                     {
217                         var rect = b.Element.ScreenRectangle;
218                         // Déduplification : ignore si une position similaire est déjà présente (±10 px)
219                         bool dejaPresent = boutonRapports.Any(t =>
220                             Math.Abs(t.Item1 - rect.Left) <= 10 && Math.Abs(t.Item2 - rect.Top) <= 10);
221                         if (!dejaPresent)
222                             boutonRapports.Add(System.Tuple.Create(rect.Left, rect.Top, b));
```

```
                }
                catch { }
            }

            // 3. Ne conserve que les boutons ayant un "terminé" en dessous (boutons valides)
            var boutonsValides = boutonRapports
                .Where(b => posTerminesUniques.Any(t => t > b.Item2))
                .OrderBy(b => b.Item2)
                .ToList();

            Report.Info($"Boutons valides (déduplifiés, avec 'terminé' en dessous): {boutonsValides.Count}");

            // 4. Sélectionne le N-ième bouton valide selon le compteur
            // Cette approche est robuste aux reflows UI : peu importe que les positions
            // changent légèrement après chaque fermeture, on prend toujours le suivant dans l'ordre.
            if (nombreRapportsTraites >= boutonsValides.Count) return false;

            var bouton = boutonsValides[nombreRapportsTraites];
            nombreRapportsTraites++;
            Report.Info($"Clic sur le bouton Rapport n°{nombreRapportsTraites} à Left={bouton.Item1}, Top={bouton.Item2}");

            bouton.Item3.Click();
            Delay.Milliseconds(2000);

            Form rapportForm = null;
            if (Host.Local.TryFindSingle("/form[@controlname='ErrorsDialog']", 5000, out rapportForm))
            {
                LireEtFermerRapport(rapportForm);
            }
            else
            {
                Report.Info("ErrorsDialog non apparu après le clic - rapport considéré traité");
            }

            return true;
        }
        catch (Exception ex)
        {
            Report.Error("Erreur dans TraiterProchainRapport: " + ex.Message);
            return false;
        }
    }
```

```
265
266         /// <summary>
267         /// Lit le contenu d'un rapport (ErrorsDialog) et le ferme.
268         /// Si aucune fenêtre n'est passée en paramètre, tente de la retrouver automatiquement.
269         /// Accumule le texte lu dans <see cref="ContenuRapports"/>.
270         /// Gère les listes virtualisées : fait défiler l'ascenseur vers le bas page par page
271         /// et collecte les nouvelles lignes à chaque étape, jusqu'à ce que l'ascenseur
272         /// soit tout en bas et qu'aucune nouvelle ligne n'apparaisse.
273         /// </summary>
274         /// <param name="rapportForm">Fenêtre ErrorsDialog à traiter (optionnel)</param>
275         public static void LireEtFermerRapport(Form rapportForm = null)
276         {
277             if (rapportForm == null)
278             {
279                 // Tente de localiser la fenêtre de rapport si elle n'a pas été fournie
280                 if (!Host.Local.TryFindSingle("/form[@controlname='ErrorsDialog']", 2000, out rapportForm))
281                 {
282                     Report.Warn("ErrorsDialog introuvable - fermeture alternative");
283                     // Fermeture de secours via le raccourci clavier Alt+F4
284                     Keyboard.Press("{Alt down}{F4}{Alt up}");
285                     Delay.Milliseconds(500);
286                     return;
287                 }
288             }
289
290             Report.Info("=== Contenu du rapport ===");
291
292             // HashSet pour dédupliquer les lignes déjà lues (invariant à l'ordre de scroll)
293             var lignesUniques = new System.Collections.Generic.HashSet<string>();
294             // Liste ordonnée pour conserver l'ordre d'apparition
295             var lignesOrdonnees = new System.Collections.Generic.List<string>();
296
297             // Fonction locale : lit les éléments visibles et ajoute les nouveaux dans les collections
298             Action lireVisiblesToAction = () =>
299             {
300                 foreach (var e in rapportForm.Find<Unknown>(".//"))
301                 {
302                     try
303                     {
304                         foreach (var attr in new[] { "Text", "Caption", "AccessibleName" })
305                         {
306                             try
307                             {
308                                 var val = e.Element.GetAttributeValueText(attr);
309                                 if (!string.IsNullOrWhiteSpace(val)
```

```
                                && val.Trim().Length > 3
                                && !System.Text.RegularExpressions.Regex.IsMatch(val.Trim(), @"^\d+$")
                                && lignesUniques.Add(val.Trim())) // Add() retourne false si déjà présent
                            {
                                lignesOrdonnees.Add(val.Trim());
                                break; // Un seul attribut par élément suffit
                            }
                        }
                        catch { }
                    }
                }
                catch { }
            }
        };

        // Localise le conteneur scrollable principal (liste / grille / panel)
        // On prend l'élément non-Form ayant la plus grande hauteur dans la fenêtre.
        Unknown zoneDefilement = null;
        try
        {
            foreach (var c in rapportForm.Find<Unknown>(".//"))
            {
                try
                {
                    // Exclut les boutons, barres de titre, etc. (hauteur trop faible)
                    var rect = c.Element.ScreenRectangle;
                    if (rect.Height > 50 &&
                        (zoneDefilement == null || rect.Height > zoneDefilement.Element.ScreenRectangle.Height))
                        zoneDefilement = c;
                }
                catch { }
            }
        }
        catch { }

        // Remonte en haut du contenu pour partir du début
        try
        {
            if (zoneDefilement != null)
                zoneDefilement.Click();
            else
                rapportForm.Click();
```

```
352                 Keyboard.Press("{Ctrl down}{Home}{Ctrl up}");
353                 Delay.Milliseconds(300);
354             }
355             catch { }
356
357             // Première lecture (lignes initiales visibles)
358             lireVisiblesToAction();
359             Report.Info($"Lignes lues après ouverture : {lignesUniques.Count}");
360
361             // Boucle de défilement page par page
362             // Arrêt si :
363             //   - l'ascenseur vertical atteint sa valeur maximale, OU
364             //   - aucune nouvelle ligne n'est trouvée sur 3 passes consécutives (filet de sécurité)
365             int tentativesSansNouveaute = 0;
366             const int maxTentativesSansNouveaute = 3;
367             const int maxIterations = 300; // Sécurité absolue (300 pages max)
368             int iteration = 0;
369
370             // Tentative de localisation de la scrollbar verticale et de son indicateur (poignée)
371             Unknown scrollbarVerticale = null;
372             Unknown indicateurPosition = null;
373             try
374             {
375                 var scrollbars = rapportForm.Find<Unknown>(".//scrollbar[@accessiblename='Vertical']");
376                 if (scrollbars.Count > 0)
377                 {
378                     scrollbarVerticale = scrollbars[0];
379                     var indicateurs = scrollbarVerticale.Find<Unknown>(".//indicator[@accessiblename='Position']");
380                     if (indicateurs.Count > 0)
381                         indicateurPosition = indicateurs[0];
382                 }
383             }
384             catch { }
385
386             while (tentativesSansNouveaute < maxTentativesSansNouveaute && iteration < maxIterations)
387             {
388                 iteration++;
389
390                 // Vérifie si l'indicateur est tout en bas de la scrollbar (coordonnées écran)
391                 if (scrollbarVerticale != null && indicateurPosition != null)
392                 {
393                     try
```

```
            {
                var rectScrollbar   = scrollbarVerticale.Element.ScreenRectangle;
                var rectIndicateur  = indicateurPosition.Element.ScreenRectangle;
                const int margePx   = 8; // tolérance en pixels
                Report.Info($"Scrollbar bas={rectScrollbar.Bottom}, Indicateur bas={rectIndicateur.Bottom}");
                if (rectIndicateur.Bottom >= rectScrollbar.Bottom - margePx)
                {
                    Report.Info("Ascenseur en bas — lecture complète.");
                    break;
                }

                // Clique dans la zone de piste sous l'indicateur → équivaut à Page Down
                int clickX = rectScrollbar.Left + rectScrollbar.Width / 2;
                int clickY = rectIndicateur.Bottom + (rectScrollbar.Bottom - rectIndicateur.Bottom) / 2;
                // Garde une marge pour ne pas cliquer hors de la scrollbar
                if (clickY > rectScrollbar.Bottom - 3) clickY = rectScrollbar.Bottom - 3;
                Report.Info($"Clic piste scrollbar à X={clickX}, Y={clickY}");
                Mouse.Click(new System.Drawing.Point(clickX, clickY));
                Delay.Milliseconds(400);
            }
            catch
            {
                // Repli clavier si le clic échoue
                try
                {
                    if (zoneDefilement != null) zoneDefilement.Click(); else rapportForm.Click();
                    Keyboard.Press("{PageDown}");
                    Delay.Milliseconds(300);
                }
                catch { }
            }
        }
        else
        {
            // Aucune scrollbar trouvée : défilement clavier classique
            try
            {
                if (zoneDefilement != null)
                    zoneDefilement.Click();
                else
                    rapportForm.Click();
                Keyboard.Press("{PageDown}");
```

```
                    Delay.Milliseconds(300);
                }
                catch { }
            }

            // Lit les nouvelles lignes apparues après le défilement
            int compteAvant = lignesUniques.Count;
            lireVisiblesToAction();
            int nouvelles = lignesUniques.Count - compteAvant;

            Report.Info($"Scroll page {iteration} — {nouvelles} nouvelle(s) ligne(s) (total : {lignesUniques.Count})");

            if (nouvelles == 0)
                tentativesSansNouveaute++;
            else
                tentativesSansNouveaute = 0; // Remet le compteur à zéro dès qu'on trouve quelque chose
        }

        // Lecture finale après Ctrl+End pour s'assurer qu'on a tout capturé
        try
        {
            if (zoneDefilement != null)
                zoneDefilement.Click();
            else
                rapportForm.Click();
            Keyboard.Press("{Ctrl down}{End}{Ctrl up}");
            Delay.Milliseconds(300);
            lireVisiblesToAction();
        }
        catch { }

        // Rapport de synthèse et accumulation
        Report.Info($"Total lignes lues dans ce rapport : {lignesOrdonnees.Count}");
        var lignes = new System.Text.StringBuilder();
        foreach (var ligne in lignesOrdonnees)
        {
            Report.Info(ligne);
            lignes.AppendLine(ligne);
        }
        Report.Info("=== Fin du rapport ===");

        if (lignes.Length > 0)
```

```
        {
            if (Verification_Messages_Asynchrones.ContenuRapports.Length > 0)
                Verification_Messages_Asynchrones.ContenuRapports += Environment.NewLine;
            Verification_Messages_Asynchrones.ContenuRapports += lignes.ToString().Trim();
        }

        // Cherche un bouton de fermeture (Fermer, OK ou Close) et clique dessus
        Button bouton = null;
        if (Host.Local.TryFindSingle(".//button[@text='Fermer' or @text='OK' or @text='Close']", 1000, out bouton))
            bouton.Click();
        else
            // Si aucun bouton n'est trouvé, on ferme la fenêtre avec Alt+F4
            Keyboard.Press("{Alt down}{F4}{Alt up}");

        Delay.Milliseconds(500);
    }

    /// <summary>
    /// Vérifie que le message attendu est bien présent dans les rapports lus.
    /// À appeler depuis un recording à la place d'une action Validate standard.
    /// </summary>
    /// <param name="messageAttendu">Texte à rechercher dans <see cref="ContenuRapports"/></param>
    [UserCodeMethod]
    public static void VerifierContenuRapports(string messageAttendu)
    {
        if (Verification_Messages_Asynchrones.ContenuRapports.Contains(messageAttendu))
            Report.Success($"Message trouvé dans les rapports : \"{messageAttendu}\"");
        else
            Report.Failure($"Message NON trouvé dans les rapports : \"{messageAttendu}\"");
    }
}
}
```