Available
in
English

Détecter et éviter les fuites de mémoire et de ressources dans les applications .NET

En dépit de ce que beaucoup de personnes pensent, il est facile d'introduire des fuites de mémoire et de ressources dans les applications .NET. Le Garbage Collector, également appelé ramasse-miettes ou GC pour les intimes, n'est pas un magicien qui vous affranchirait totalement de vous soucier de la consommation de ressources et de mémoire.
J'expliquerai dans cet article pourquoi les fuites de mémoire existent en .NET et comment les éviter. Ne vous en faites pas, je ne vais pas me focaliser ici sur le fonctionnement interne du garbage collector et autres caractéristiques avancées de la gestion de la mémoire et de ressources en .NET
Il est important de comprendre les fuites et comment les éviter, particulièrement car elles n'entrent pas dans la catégorie des choses facilement détectables de manière automatisée. Les test unitaires ne vous aideront pas ici. Et quand votre application se plantera en production, vous serez à la recherche de solutions dans l'urgence. Alors, détendez-vous et prenez le temps dès maintenant d'en apprendre plus sur ce sujet avant qu'il ne soit trop tard.

Table des Matières

Introduction

Récemment, j'ai travaillé sur un gros projet .NET (appelons le projet X) pour lequel une de mes responsabilités était de chasser les fuites de mémoire et de ressources. Je traquais principalement les fuites liées à l'IHM, plus particulièrement dans une application Windows Forms basée sur le Composite UI Application Block (CAB).
Alors que certaines des informations que vais exposer ici s'appliquent directement à Windows Forms, la plupart des points s'appliqueront également à tout type d'application .NET (WPF, Silverlight, ASP.NET, service Windows, application console, etc.)

Je n'étais pas un expert en traque de fuites avant d'avoir à plonger dans les entrailles de l'application pour y faire du ménage. Le but du présent article est de partager avec vous ce que j'ai appris au cours de ma mission. J'espère que cela pourra être utile à toute personne qui devra détecter et corriger des fuites de mémoire et de ressources.
Nous commencerons par un aperçu de ce que sont les fuites, puis nous verrons en suite comment détecter les fuites et trouver les ressources qui fuient, comment résoudre et éviter les fuites, et nous finirons avec une liste d'outils et de... ressources utiles.

Fuites ? Ressources ? Qu'entendez-vous par là ?

Fuites de mémoire

Avant d'aller plus loin, commençons par définir ce que j'appelle une "fuite de mémoire". Reprenons simplement la définition trouvée dans Wikipedia (traduction de la version anglaise). Elle correspond parfaitement à ce que j'ai l'intention de vous aider à résoudre avec cet article :

> *En informatique, une fuite de mémoire est un type particulier de consommation non intentionnelle de mémoire par un programme qui ne libére pas comme il le devrait la mémoire dont il n'a plus besoin. La cause la plus classique d'une telle situation est habituellement un bug dans le programme qui l'empêche de libérer la mémoire qui n'est plus utile.*

Toujours dans Wikipedia en anglais : "Les langages qui présentent une gestion automatisée de la mémoire, tels Java, C#, VB.NET ou LISP, ne sont pas immunisés contre les fuites de mémoire."

Le garbage collector récupère uniquement la mémoire qui n'est plus accessible. Il ne libère pas la mémoire tant qu'elle reste accessible. En .NET, cela signifie que des objets accessibles par au moins une référence ne seront par relâchés par le garbage collector.

Handles et ressources

La mémoire n'est pas la seule ressource sur laquelle garder un œil. Quand votre application .NET tourne sous Windows, elle consomme tout un ensemble de ressources du système. Microsoft définit trois catégories d'**objets système**: user, graphics device interface (GDI), et kernel. Je ne fournirai pas ici la liste complète des objets. Nommons juste les plus importants :

- Le système utilise les **User objects** pour la gestion des fenêtres. Ils incluent : Accelerator tables, Carets, Cursors, Hooks, Icons, Menus et Windows.
- Les **GDI objects** servent pour le graphisme : Bitmaps, Brushes, Device Contexts (DC), Fonts, Memory DCs, Metafiles, Palettes, Pens, Regions, etc.
- Les **Kernel objects** servent pour la gestion de la mémoire, l'exécution des processus, et les communications inter-processus (IPC) : Files, Processes, Threads, Semaphores, Timers, Access tokens, Sockets, etc.

Vous trouverez tous les détails sur les objets système sur MSDN.

En plus des objets système, vous rencontrez des **handles**. Comme indiqué sur MSDN, les applications ne peuvent pas accéder directement aux données des objets ou aux ressources système qu'un objet représente. Au lieu de cela, une application doit obtenir un handle d'objet, qu'elle peut utiliser pour examiner ou modifier la ressource système.
En .NET toutefois, cela sera transparent la plupart du temps car les objets système et les handles sont représentés directement ou indirectement par des classes .NET.

Ressources non managées

Les ressources telles que les objets système ne sont pas un problème en elles-mêmes, toutefois je les mentionne dans cet article car les systèmes d'exploitation comme Windows ont des limites sur le nombre de sockets, fichiers, etc. qui peuvent être ouvertes simultanément. C'est pourquoi il est important que vous soyez attentif à la quantité d'objets système que votre application utilise.
Windows a des quotas pour le nombre d'objets User et GDI qu'un processus peut utiliser à un moment donné. Les valeurs par défaut sont 10 000 pour les objets GDI et 10 000 pour les objets User. Si vous avez besoin de lire ces valeurs sur votre machine, vous pouvez utiliser les clefs de registre suivantes, qui se trouvent dans HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows : GDIProcessHandleQuota et USERProcessHandleQuota.

Devinez quoi ? Ce n'est même pas aussi simple. Il y a d'autres limites que vous pouvez atteindre rapidement. Consultez cette entrée sur mon blog à propos du Desktop Heap, par exemple.

Etant donné que ces valeurs peuvent être personnalisées, vous pourriez penser qu'une solution pour s'affranchir des limites est d'augmenter les quotas. Je pense que c'est une mauvaise idée pour plusieurs raisons :

1. Les quotas existent pour une raison : votre application n'est pas seule sur le système et elle doit partager les ressources du système avec les autres processus qui tournent sur la machine.
2. Si vous les modifiez sur votre machine, elles risquent d'être différentes sur une autre. Vous devez penser à faire la modification sur toutes les machines où tournera votre application, ce qui n'est pas sans poser de problème du point vue de l'administration système.
3. Les quotas par défaut sont plus que suffisants la plupart du temps. Si vous trouvez que les quotas ne sont pas suffisants pour votre application, alors vous avez probablement un peu de ménage à faire.

Comment détecter des fuites et repérer les ressources qui fuient

Le véritable problème avec les fuites est bien formulé dans un article à propos des fuites avec du code GDI :

> *Même une petite fuite peut mettre le système à genoux si elle se produit plusieurs fois.*

Ceci est similaire à ce qui se produit dans le cas des fuites d'eau. Une goutte d'eau n'est pas un gros problème. Mais goutte par goutte, une fuite peut devenir un problème majeur.
Comme je l'expliquerai plus tard, un simple objet insignifiant peut maintenir en vie toute une grappe d'objets pesant lourd en mémoire.

Toujours dans le même article, vous pouvez apprendre que :

> *Il y a généralement trois étapes dans l'éradication d'une fuite :*
>
> 1. *Détecter une fuite*
> 2. *Trouver la ressource qui fuit*

3. *Décider où et quand la ressource devrait être libérée dans le code source*

La façon la plus directe de "détecter" des fuites est d'en souffrir.
Vous ne verrez probablement pas votre ordinateur tomber à court de mémoire. Les messages "Out of memory" sont très rares. Ceci vient du fait que les systèmes d'exploitation utilisent l'espace des disques durs pour étendre l'espace de mémoire disponible quand ils arrivent à court de RAM (ceci est appelé de la mémoire virtuelle).
Ce que vous avez plus de chance de voir, ce sont les exceptions "out of handles" dans vos applications graphiques Windows. L'exception exacte est soit une *System.ComponentModel.Win32Exception* soit une *System.OutOfMemoryException* avec le message suivant : "Erreur lors de la création d'un handle de fenêtre". Ceci arrive quand trop de ressources sont consommées en même temps, très probablement à cause d'objets qui ne sont pas libérés comme ils le devraient.
Une autre chose que vous pouvez rencontrer plus souvent est un ralentissement croissant de votre application ou de l'ordinateur tout entier. Ceci peut arriver car votre machine est simplement en train d'arriver à court de ressources.

Je me permettrais de faire l'affirmation brutale suivante : la plupart des applications fuient. La plupart du temps, ce n'est pas gênant car les problèmes résultant de fuites apparaissent uniquement si les applications sont utilisées intensément pendant une longue période de temps.

Si vous suspectez que des objets trainent en mémoire alors qu'ils devraient être libérés, la première chose à faire est de trouver quels sont ces objets.
Cela peut paraître évident, mais ce qui n'est pas aussi évident est comment trouver ces objets.

Ce que je vous suggère est de chercher avec votre profileur de mémoire préféré des objets de haut niveau ou des conteneurs racines inattendus ou restés à la traîne. Dans le projet X, il peut s'agir d'objets tels que des instances de LayoutView (nous utilisons le modèle MVP avec CAB/SCSF). Dans votre cas, tout dépend de quels sont les objets racines.

L'étape suivante est de trouver pourquoi ces objets sont maintenus en mémoire alors qu'ils ne le devraient pas. C'est là que les débuggeurs et les profileurs aident vraiment. Ils peuvent vous montrer comment les objets sont liés les uns aux autres.
C'est en regardant quelles sont les références entrantes d'un objet zombie que vous avez identifié que vous pourrez trouver la racine du problème.

Vous pouvez choisir de la jouer façon ninja (cf. SOS.dll et WinDbg dans la section à propos des outils ci-dessous).
J'ai utilisé l'outil JetBrains dotTrace pour le projet X et c'est ce que j'utiliserai également dans cet article. Je vous en dirai plus sur cet outil plus loin dans la même section Outils.

Votre but doit être de trouver la référence racine. Ne vous arrêtez pas au premier objet que vous trouverez, mais demandez-vous pourquoi cet objet est maintenu en mémoire

Les causes habituelles de fuites de mémoire

J'ai écrit plus haut que les fuites sont fréquentes en .NET. La bonne nouvelle est que la liste des causes possibles est limitée. Cela signifie que vous n'aurez pas à chercher parmi beaucoup de cas quand vous essaierez de résoudre une fuite.

Passons en revue les coupables habituels que j'ai identifiés :

- Références statiques
- Désabonnement manquant à un événement

- Désabonnement manquant à un événement statique
- Méthode Dispose pas appelée
- Traitement réalisé dans Dispose incomplet

En plus de ces pièges classiques, voici d'autres sources de problèmes plus spécifiques :

- Windows Forms : BindingSource mal utilisé
- CAB : appel manquant à Remove sur un WorkItem

Les coupables que je viens juste de lister concernent vos applications, mais vous ne devez pas oublier que des fuites peuvent aussi se produire dans d'autres bouts de code .NET sur lesquels vos applications s'appuient. Il peut en effet y avoir des bugs dans les bibliothèques que vous utilisez.

Prenons un exemple. Dans le projet X, une suite de contrôles graphiques du marché est utilisée pour construire l'IHM. Un de ces contrôles est utilisé pour afficher des barres d'outils. Cela fonctionne via un composant qui gère une liste de barres d'outils. Cela fonctionne correctement, sauf que bien que la classe barre d'outils implémente IDisposable, la classe qui gère les barres d'outils n'appelle jamais la méthode Dispose des barres qu'elle gère. C'est un bug. Heureusement, une solution de contournement est facile à trouver : appeler nous-mêmes Dispose sur chaque barre d'outils. Malheureusement, ce n'est pas suffisant car la classe barre d'outils est elle-même buggée : elle ne dispose pas les contrôles (boutons, labels, etc.) qu'elle contient. A nouveau, la solution est de disposer chaque contrôle que la barre d'outils contient, mais cela n'est pas si simple cette fois-ci car chaque sous-contrôle est différent.
Quoi qu'il en soit, ce n'est là qu'un exemple spécifique. Là où je veux en venir c'est que toute bibliothèque ou composant que vous pouvez utiliser peut causer des fuites dans vos applications.

Finalement, j'aimerais ajouter que dire "C'est le framework .NET qui cause des fuites !" est adopter une très mauvaise attitude qui consiste à s'en laver les mains. Même s'il peut bien sûr arriver que .NET crée des fuites lui-même, c'est quelque chose qui reste exceptionnel et que vous rencontrerez très rarement.
Il est facile d'accuser .NET, mais vous devriez quand même commencer par remettre en question votre propre code avant de vous défausser...

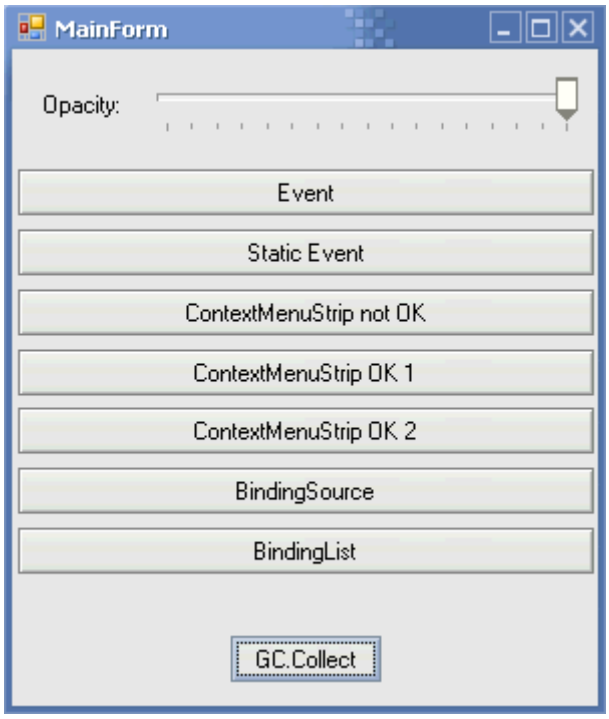
Les causes habituelles de fuites en pratique

J'ai listé les principales sources de fuites, mais je ne veux pas m'arrêter là. Je pense que cet article sera beaucoup plus utile si je peux illustrer chaque point avec un exemple rapide. Alors, prenons Visual Studio et dotTrace et parcourons des exemples de code. Je montrerai par la même occasion comment résoudre ou éviter chaque fuite.

Le projet X est construit avec le CAB et le modèle MVP (Model-View-Presenter), cela signifie que l'IHM est constituée de workspaces, de vues et de présentateurs. Pour garder les choses simples, J'ai décidé d'utiliser une application Windows Forms basique avec un ensemble de fenêtres. C'est la même approche que celle utilisée par Jossef Goldberg dans son billet à propos des fuites de mémoire dans les applications basées sur WPF. Je réutiliserai d'ailleurs les mêmes exemples pour les gestionnaires d'événements.

Quand une fenêtre est fermée et disposée, nous nous attendons à ce qu'elle soit libérée de la mémoire, n'est-ce pas ? Ce que je vais montrer ci-dessous est comment les causes que j'ai listées ci-dessous empêchent les fenêtres d'être libérées.

Voici la fenêtre principale de l'application d'exemple que j'ai créée :



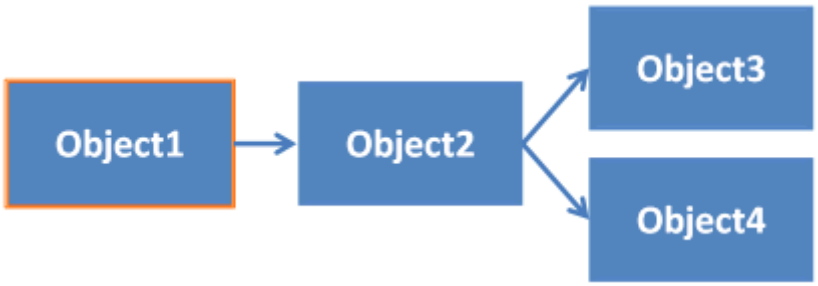
Cette fenêtre principale peut ouvrir plusieurs fenêtres filles; chacune peut causer une fuite de mémoire distincte.

Vous trouverez le code source de cette application exemple à la fin de cet article, dans le chapitre Ressources.

Les références statiques

Débarrassons nous tout d'abord du plus évident. Si un objet est référencé par un champ statique, alors il ne sera jamais libéré.
Ceci est également vrai avec des constructions telles que des singletons. Les singletons sont souvent des objets statiques, et même quand ce n'est pas le cas, ce sont quand même des objets ayant une longue durée de vie.

Cela peut sembler évident, mais gardez en tête que ce ne sont pas les seules références directes qui sont dangereuses. Le vrai danger vient des références indirectes. En fait, vous devez faire attention aux chaînes de références. Ce qui compte, c'est la racine de chaque chaîne. Si la racine est statique, alors tous les objets suivants de la chaîne resteront en vie pour toujours.



Si Object1 sur le diagramme ci-dessus est statique, et très probablement d'une longue durée de vie, alors tous les autres objets le long de la chaîne de référence seront maintenus en vie longtemps. Le danger est que la chaîne peut devenir trop longue pour réaliser que la racine de la chaîne est statique. Si vous ne vous préoccupez que d'un niveau de profondeur, vous considérerez qu'Object3 et Object4

s'en iront quand Object2 s'en ira. C'est correct, bien-sûr, mais vous devez prendre en compte le fait qu'ils pourraient ne jamais s'en aller parce qu'Object1 maintient toute la grappe d'objets en vie.

Soyez très prudent avec toutes sortes de statiques. Evitez les autant que possible. Si ça ne l'est pas, soyez très attentif aux objets que vos objets statiques et singletons maintiennent en vie.

Un type spécifique de statiques dangereux est les événements statiques. Je les aborderai juste après avoir abordé les événements en général.

Les événements, ou le problème du "lapsed listener"

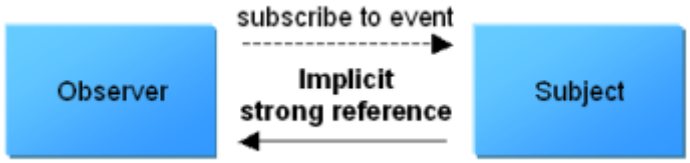
Une fenêtre fille s'abonne à un événement de la fenêtre principale pour être notifiée quand l'opacité change (EventForm.cs) :

```
mainForm.OpacityChanged += mainForm_OpacityChanged;
```

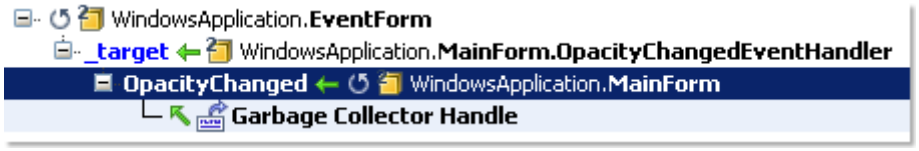
Le problème est que cet abonnement à l'événement OpacityChanged crée une référence de la fenêtre principale vers la fenêtre fille.

voici comment sont connectés les objets après l'abonnement :

Consultez ce billet sur mon blog pour en apprendre d'avantage learn sur les événements et les références. Voici un schéma de ce billet qui montre la référence "inversée" qu'un sujet vers ses observateurs :



Voici ce que vous verrez avec dotTrace si vous cherchez EventForm et cliquez sur "Shortest" dans "Root Paths" :



Comme vous pouvez le voir, MainForm maintient une référence vers EventForm. C'est le cas pour chaque instance d'EventForm que vous ouvrirez dans l'application. Cela signifie que toutes les fenêtres filles que vous ouvrirez resteront en mémoire tant que l'application est en vie, même si vous ne les utilisez plus.

En plus d'avoir les fenêtres filles maintenues en mémoire, vous aurez aussi potentiellement droit à des exceptions si vous changez l'opacité après qu'une fenêtre fille ait été fermée, parce que la fenêtre principale essaiera de notifier une fenêtre disposée.

La solution la plus simple est de supprimer les références en s'arrangeant pour que les fenêtres filles se désabonnent de l'événement de la fenêtre principale quand elles sont disposées :

```
Disposed += delegate { mainForm.OpacityChanged -= mainForm_OpacityChanged;
};
```

Nota Bene: Nous avons un problème ici car l'objet MainForm reste en vie jusqu'à ce que l'application soit terminée. Des objets interconnectés avec des durées de vie plus courtes peuvent ne pas poser de soucis de mémoire. Tout graphe isolé d'objets est libéré de la mémoire automatiquement par le garbage collector. Un graphe isolé d'objets est formé par deux objets qui se référencent juste l'un l'autre, ou par un groupe d'objets connectés avec aucune référence externe.

Une autre solution serait d'utiliser des "délégués faibles" (weak delegates), qui sont basés sur des références faibles. J'effleure le sujet dans mon message à propos des événements et des références. Plusieurs articles sur le web montrent comment mettre ceci en action. En voici un bon, par exemple. La plupart des solutions que vous trouverez s'appuient sur la classe WeakReference. Vous pouvez en apprendre plus sur les références faibles en NET sur MSDN.

Notez qu'une solution pour cela existe en WPF, sous la forme du modèle WeakEvent.

Il y a d'autres solutions si vous travaillez avec des frameworks comme le CAB (Composite UI Application Block) ou Prism (Composite Application Library), respectivement EventBroker et EventAggregator. Si vous voulez, vous pouvez aussi utiliser votre propre implémentation du modèle courtier/agrégateur/médiateur d'événements.

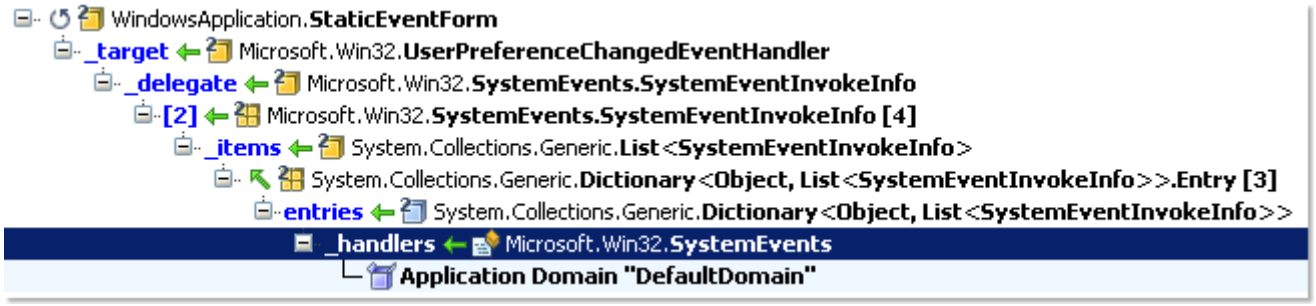
Les gestionnaires d'événements d'objets statiques ou à longue durée de vie avec des désabonnements manquants sont un problème. Un autre est les événements statiques.

Les événements statiques

Prenons directement un exemple (StaticEventForm.cs) :

```
SystemEvents.UserPreferenceChanged += SystemEvents_UserPreferenceChanged;
```

Ceci est similaire au cas précédent, sauf que cette fois-ci nous nous abonnons à un événement statique. L'événement étant statique, l'objet fenêtre à l'écoute ne sera jamais relâché.



A nouveau, la solution est de se désabonner quand nous en avons fini :

```
SystemEvents.UserPreferenceChanged -= SystemEvents_UserPreferenceChanged;
```

La méthode Dispose non appelée

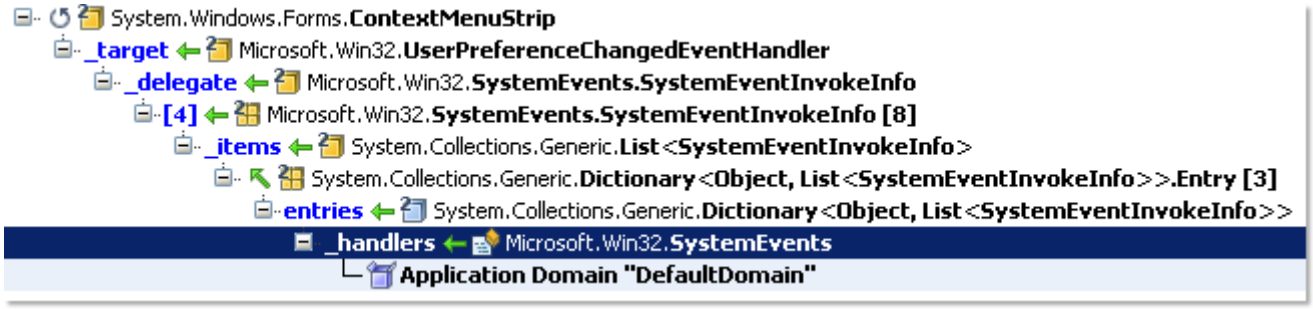
Vous avez été attentif aux événements, statiques ou non ? Parfait, mais ce n'est pas suffisant. Vous pouvez encore obtenir des références persistantes même avec du code de nettoyage. Ceci arrive parfois simplement parce que le code de nettoyage n'est pas appelé...

Utiliser la méthode Dispose ou l'événement Disposed pour se désabonner d'un événement et relâcher les ressources est une bonne pratique, mais cela ne sert à rien si Dispose n'est pas appelée.

Prenons un exemple intéressant. Voici du code qui crée un menu contextuel pour une fenêtre (ContextMenuStripNotOKForm.cs) :

```
ContextMenuStrip menu = new ContextMenuStrip();
menu.Items.Add("Item 1");
menu.Items.Add("Item 2");
this.ContextMenuStrip = menu;
```

Voici ce que vous verrez avec dotTrace après que la fenêtre soit fermée et disposée :



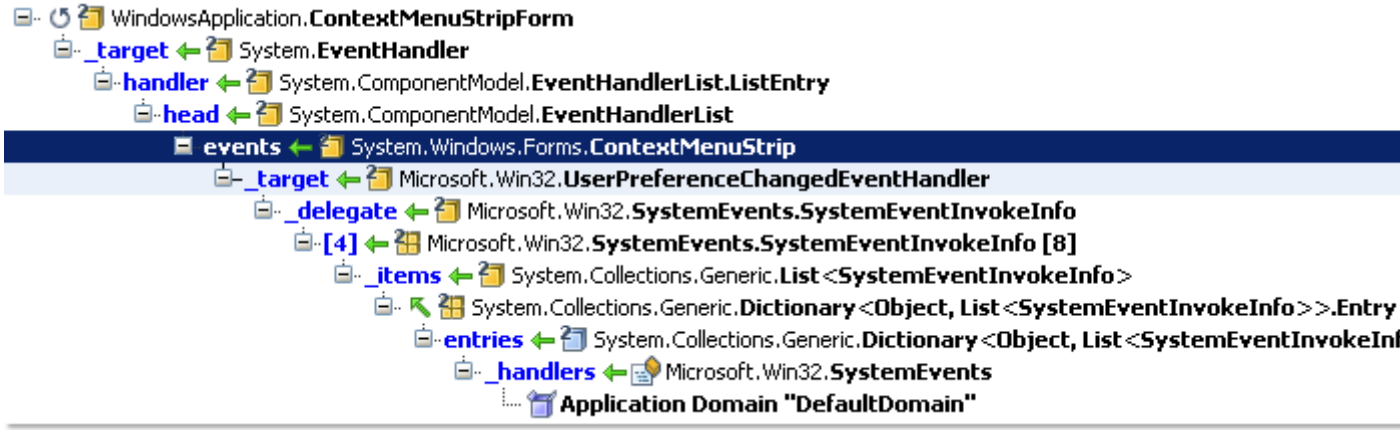
Le ContextMenuStrip est toujours en vie en mémoire ! Note : Pour observer le problème se produire, montrez le menu contextuel avec un clic-droit avant de fermer la fenêtre.

Une nouvelle fois, c'est un problème lié aux événements statiques. La solution est la même que d'habitude :

```
Disposed += delegate { ContextMenuStrip.Dispose(); };
```

Je pense que vous commencez à comprendre comme les événements peuvent être dangereux en .NET si vous ne faites pas très attention à eux et aux références qu'ils impliquent.
Ce que je veux mettre en évidence ici c'est qu'il est facile d'introduire une fuite juste avec quelques lignes de code. Auriez-vous pensé à de potentielles fuites de mémoire en créant un menu contextuel ?

C'est même pire que ce que vous pouvez l'imaginer. Non seulement le ContextMenuStrip est maintenu en vie, mais il maintient la fenêtre complète en vie avec lui !
Sur la photo d'écran suivante, vous pouvez voir que le ContextMenuStrip référence la fenêtre :



Le résultat c'est que la fenêtre sera en vie tant que le ContextMenuStrip l'est. Oh, bien-sûr vous ne devez pas oublier que tant que la fenêtre est en vie, elle maintient elle-même tout un ensemble d'objets en vie (à commencer par les contrôles et les composants qu'elle contient) :

C'est quelque chose que je trouve suffisamment important pour mériter un gros avertissement. Un petit objet peut potentiellement maintenir de grosses grappes d'objets en mémoire. J'ai vu cela arriver tout le temps sur le projet X. C'est la même chose avec de l'eau : une petite fuite peut causer de grands dégâts.
Parce qu'un simple contrôle pointe vers son parent ou vers des événements de son parent, il peut maintenir toute une chaîne de contrôles conteneurs en vie s'il n'a pas été disposé. Et bien-sûr, tous les autres contrôles contenus dans ces conteneurs sont également gardés en mémoire. Cela peut par exemple conduire à avoir une fenêtre complète et tout son contenu qui restent en mémoire pour toujours (du moins jusqu'à l'arrêt de l'application).

A ce stade, vous pourriez vous demander si le problème existe toujours avec ContextMenuStrip. Ce n'est pas le cas. La plupart du temps, vous créez des ContextMenuStrips avec le designer directement sur leurs fenêtres, et dans ce cas Visual Studio génère du code qui assure que les composants ContextMenuStrip sont bien disposés.

Si vous êtes intéressés de savoir comment cela est géré, vous pouvez jeter un œil à la classe *ContextMenuStripOKForm* class et son champ *components* dans le fichier ContextMenuStripOKForm.Designer.cs.

J'aimerais attirer l'attention sur une autre situation que j'ai vue dans le projet X. Pour une raison quelconque, il n'y avait pas de fichiers .Designer.cs associés avec les fichiers sources de quelques contrôles. Le code du designer était directement dans les fichiers .cs. Ne me demandez pas pourquoi. En dehors de l'aspect inhabituel (et non recommandé) de la structure de code, le problème était que le code du designer n'avait pas été copié dans sa totalité : soit la méthode Dispose était manquante, soit l'appel à *components.Dispose* était manquant. Je pense que vous comprenez les mauvaises choses qui peuvent arriver dans ces cas.

Traitement réalisé dans Dispose incomplet

J'imagine que vous avez maintenant compris l'importance d'appeler Dispose sur tous les objets qui ont cette méthode. Toutefois, il y a une chose que j'aimerais souligner à propos de Dispose. C'est très bien de faire implémenter l'interface IDisposable par vos classes et d'inclure des appels à Dispose et des blocs *using* partout dans votre code, mais cela n'est vraiment utile que si les méthodes Dispose sont implémentées correctement.

Cette remarque peut sembler un peu bête, mais si je la fait c'est parce que j'ai vu beaucoup de cas dans lesquels le code des méthodes Dispose n'était pas complet.
Vous savez comment cela arrive. Vous créez vos classes; vous leur faites implémenter IDisposable; vous ajoutez les désabonnements aux événements et relâchez les ressources dans Dispose; et vous appelez Dispose partout. C'est excellent, jusqu'à ce que plus tard une de vos classes s'abonne à nouvel événement ou consomme une nouvelle ressource. C'est facile à coder, vous êtes pressé de finir de coder et de tester votre code. Ca fonctionne correctement et vous en êtes content. Vous checkinez (c'est beau comme verbe, non ?). Parfait! Mais... oups, vous avez oublié de mettre à jour Dispose et de tout relâcher. Ca arrive tout le temps.

Je ne fournis pas de code pour cet exemple. Cela devrait être relativement évident.

Note : En plus de la méthode Dispose, vous devez également connaître la méthode Finalize. Dispose permet une libération des ressources explicite, alors que Finalize permet une libération implicite. Les deux méthodes doivent être utilisées de concert. Tout ceci est bien expliqué dans cet article par Joydip Kanjilal.

Windows Forms : BindingSource mal utilisé

Penchons-nous sur un problème spécifique à Windows Forms. Si vous utilisez le composant BindingSource, assurez-vous de l'utiliser de la façon pour laquelle il a été conçu.
J'ai vu du code qui exposait des objets BindingSource via des références statiques. Cela conduit à des fuites de mémoire à cause de la façon dont BindingSource se comporte. Une BindingSource garde une référence vers les contrôles qui l'utilisent comme leur DataSource, même après que ces contrôles aient été disposés.

Voici ce que vous verrez avec dotTrace après qu'une ComboBox soit disposée si sa DataSource est une BindingSource statique (ou d'une longue durée de vie) (BindingSourceForm.cs) :

Une solution à ce problème est d'exposer une BindingList à la place d'une BindingSource. Vous pouvez, par exemple, déposer une BindingSource sur votre fenêtre, affecter la BindingList en tant que DataSource pour la BindingSource, et affecter la BindingSource en tant que DataSource pour la ComboBox. De cette façon, vous utilisez toujours une BindingSource.
Regardez BindingListForm.cs dans le code source d'exemple pour voir cela en action.

Cela ne vous empêche pas d'utiliser une BindingSource, mais elle doit être créée dans la vue (la fenêtre ici) qui l'utilise. C'est du bon sens de toute façon : une BindingSource est un composant de présentation défini dans le namespace System.Windows.Forms. BindingList, en comparaison, est une collection qui n'est pas dépendante de composants visuels.

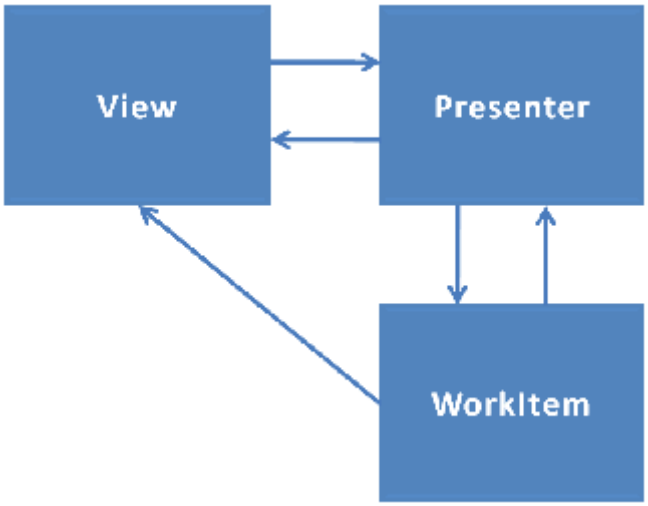
Note : Si vous n'avez pas vraiment besoin d'une BindingSource, vous pouvez très bien vous contenter d'utiliser juste une BindingList tout du long.

CAB : Suppression du WorkItem manquante

Voici maintenant un conseil pour les applications CAB, mais que vous pouvez appliquer à d'autres types d'applications.

Les WorkItems sont centraux lorsqu'on construit des applications CAB. Un WorkItem est un conteneur qui gère les objets vivants dans un contexte, et qui effectue l'injection des dépendances. Habituellement, une vue est ajoutée à un WorkItem après qu'elle soit créée. Quand la vue est fermée et doit être relâchée, elle doit être retirée de son WorkItem, sinon le WorkItem gardera la vue en vie parce qu'il maintient une référence vers elle.
Des fuites peuvent se produire si vous oubliez de retirer les vues de leur WorkItem.

Dans le projet X, nous utilisons le modèle de conception MVP (Model-View-Presenter). Voici comment les divers éléments sont connectés quand une vue est affichée :



Notez que le présentateur est également ajouté au WorkItem, de manière à ce qu'il bénéficie aussi de l'injection de dépendances. La plupart du temps, le présentateur est injecté à la vue par le WorkItem, au fait. Pour s'assurer que tout se libère correctement dans le projet X, nous utilisons une chaîne de libération comme suit :

Quand une vue est disposée (le plus probablement parce qu'elle a été fermée), sa méthode Dispose est appelée. Cette méthode invoque à son tour la méthode Dispose du présentateur. Le présentateur, qui connaît le WorkItem, retire la vue et lui-même du WorkItem. De cette façon, tout est disposé et libéré correctement.

Nous avons des **classes de base qui implémentent cette chaîne de responsabilité** dans notre framework applicatif, de manière à ce que les développeurs des vues n'aient pas à réimplémenter ceci et s'en soucier à chaque fois. Je vous encourage à implémenter ce type de modèle dans vos applications, même s'il ne s'agit pas d'applications CAB. **Automatiser les modèles de libération directement au sein de vos objets vous aidera à éviter les fuites par omission.** Cela assurera également que ce traitement est implémenté d'une seule façon, et non différemment par chaque développeur parce qu'il ou elle ne connait peut-être pas une façon correcte de le faire (ce qui pourrait conduire à des fuites par manque de savoir-faire).

Comment éviter les fuites

Maintenant que vous en savez plus sur les fuites et comment elles peuvent arriver, j'aimerais attirer l'attention sur quelques points et vous donner quelques conseils.

Discutons tout d'abord d'une règle générale. Habituellement, un objet qui crée un autre objet est en charge de le disposer. Bien-sûr, ce n'est pas le cas si le créateur est une fabrique d'objets.
A l'inverse, un objet qui reçoit une référence vers un autre objet n'a pas la charge de disposer celui-ci. En fait, cela dépend vraiment de la situation. En tout cas, ce qu'il est important de garder à l'esprit est qui possède un objet.

Seconde règle : **Chaque += est un ennemi en puissance !**

D'après ma propre expérience, je dirais que les événements sont la première source de fuites en .NET. Ils demandent à ce qu'on les vérifie deux fois plutôt qu'une. Chaque fois que vous ajoutez un abonnement à un événement dans votre code, vous devriez vous inquiéter des conséquences et vous demander si vous avez besoin d'ajouter un -= pour le désabonnement de l'événement. Si vous avez à le faire, faites le immédiatement avant que vous ne l'oubliiez. Souvent, vous ferez cela dans une méthode Dispose.

Faire en sorte que les objets à l'écoute se désabonnent eux-mêmes des événements auxquels ils sont abonnés est habituellement l'approche recommandée pour s'assurer qu'ils soient bien libérés par le garbage collector. Néanmoins, quand vous êtes sûr qu'un objet observé ne va plus publier de notifications et que vous souhaitez que ses abonnés puissent être libérés, vous pouvez forcer la suppression de tous les abonnements à l'objet observé. J'ai un exemple de code sur mon blog qui montre comment faire cela.

Un conseil rapide maintenant. Souvent, les problèmes commencent à apparaître quand des références vers des objets sont partagées entre plusieurs objets. Cela arrive parce qu'il peut devenir difficile de suivre qui référence quoi. Parfois il est préférable de cloner les objets en mémoire et d'avoir les vues qui travaillent avec les clones plutôt que d'avoir les objets vues et modèles entremêlés.

Enfin, même si c'est une bonne pratique bien connue en .NET, j'aimerais souligner à nouveau l'importance d'appeler Dispose. A chaque fois que vous allouez une ressource, assurez vous d'appeler Dispose ou d'encapsuler l'utilisation de la ressource dans un block *using*. Si vous ne faites pas cela systématiquement, vous pouvez rapidement vous retrouver avec des fuites de ressources (la plupart du temps des ressources non managées).

Outils

Plusieurs outils sont disponibles pour vous aider à suivre les instances d'objets, ainsi que les objets système et les handles. Citons en quelques uns.

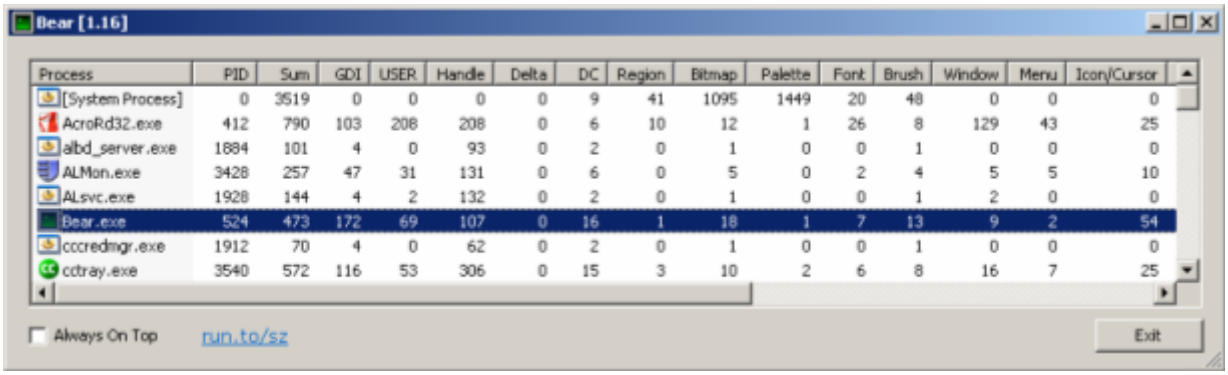
Windows Performance Monitor

Sur des machines de production, il est habituellement préférable de ne rien avoir à installer de nouveau pour détecter des fuites. Heureusement, vous pouvez commencer avec des outils intégrés à Windows.
Cet article vous montrera comment utiliser le Windows Performance Monitor (PerfMon), par exemple.

Bear

Bear est un programme gratuit qui affiche pour tous les processus tournant sous Windows :

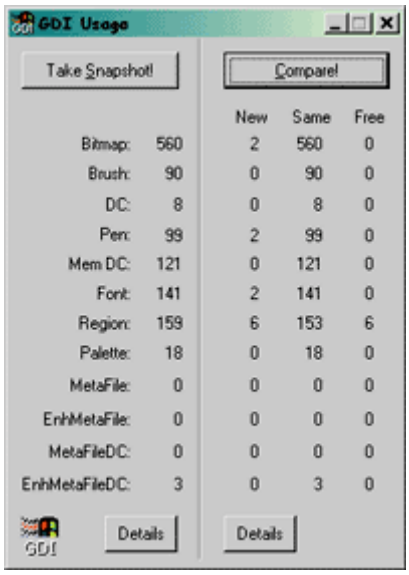
- le niveau d'utilisation de tous les objets GDI (hDC, hRegion, hBitmap, hPalette, hFont, hBrush)
- le niveau d'utilisation de tous les objets User (hWnd, hMenu, hCursor, SetWindowsHookEx, SetTimer et quelques autres)
- le nombre de handles



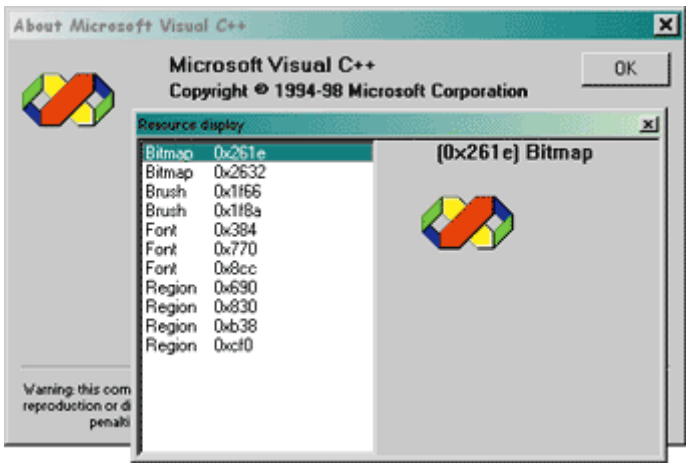
GDIUsage

Un autre outil utile est GDIUsage. Cet outil est gratuit également et son code source est disponible.

GDIUsage se concentre sur les objets GDI. Avec lui, vous pouvez prendre une photographie de la consommation GDI actuelle, exécuter une action susceptible de provoquer une fuite, et effectuer une comparaison entre l'état actuel des ressources avec l'état précédent. Cela aide beaucoup car cela permet de voir quels objets GDI ont été ajoutés (ou libérés) durant l'action.



En plus, GDIUsage ne fournit pas uniquement des nombres, mais peut aussi fournir un affichage graphique des objets GDI. Visualiser quel bitmap fuit simplifie la recherche de pourquoi il a fui.

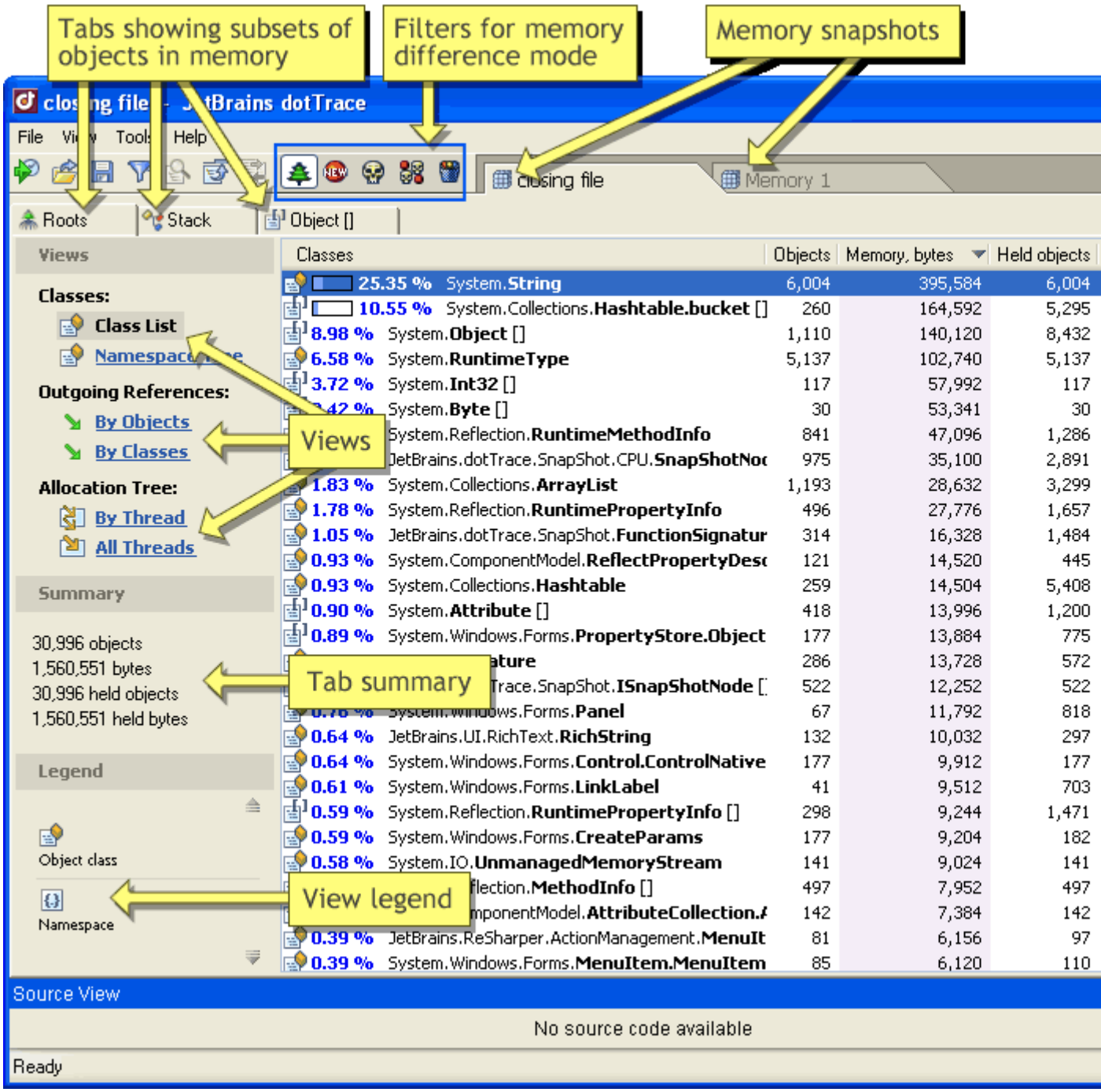


dotTrace

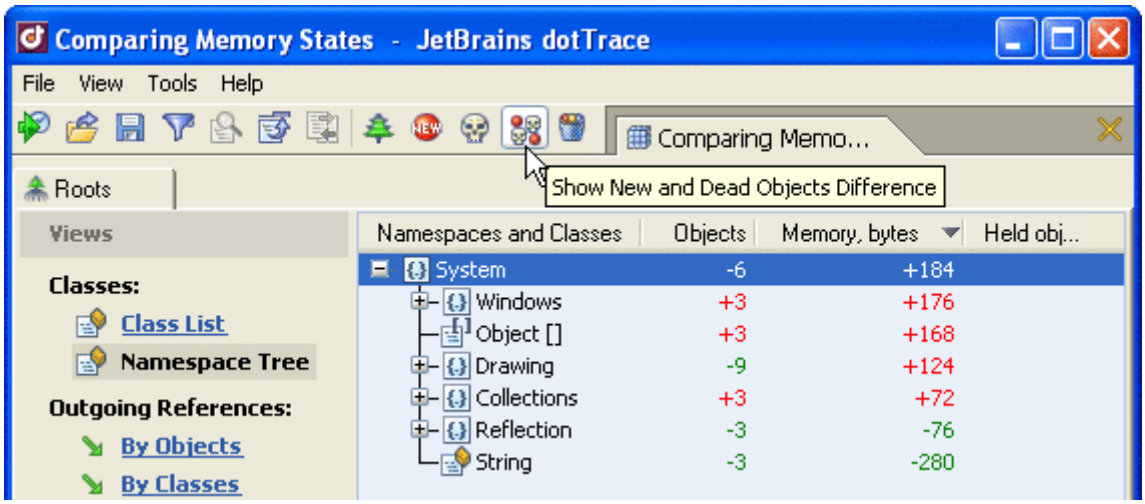
JetBrains dotTrace est un profileur de mémoire et de performance pour .NET.

Les photos d'écran que j'utilise dans cet article ont été prises avec dotTrace. C'est aussi l'outil que j'ai utilisé le plus pour le projet X. Je ne connais pas vraiment les autres profileurs .NET, mais dotTrace m'a fourni les informations dont j'avais besoin pour résoudre les fuites détectées dans le projet X (plus de 20 fuites à ce jour... est-ce que je vous ai dit qu'il s'agit d'un gros projet ?).

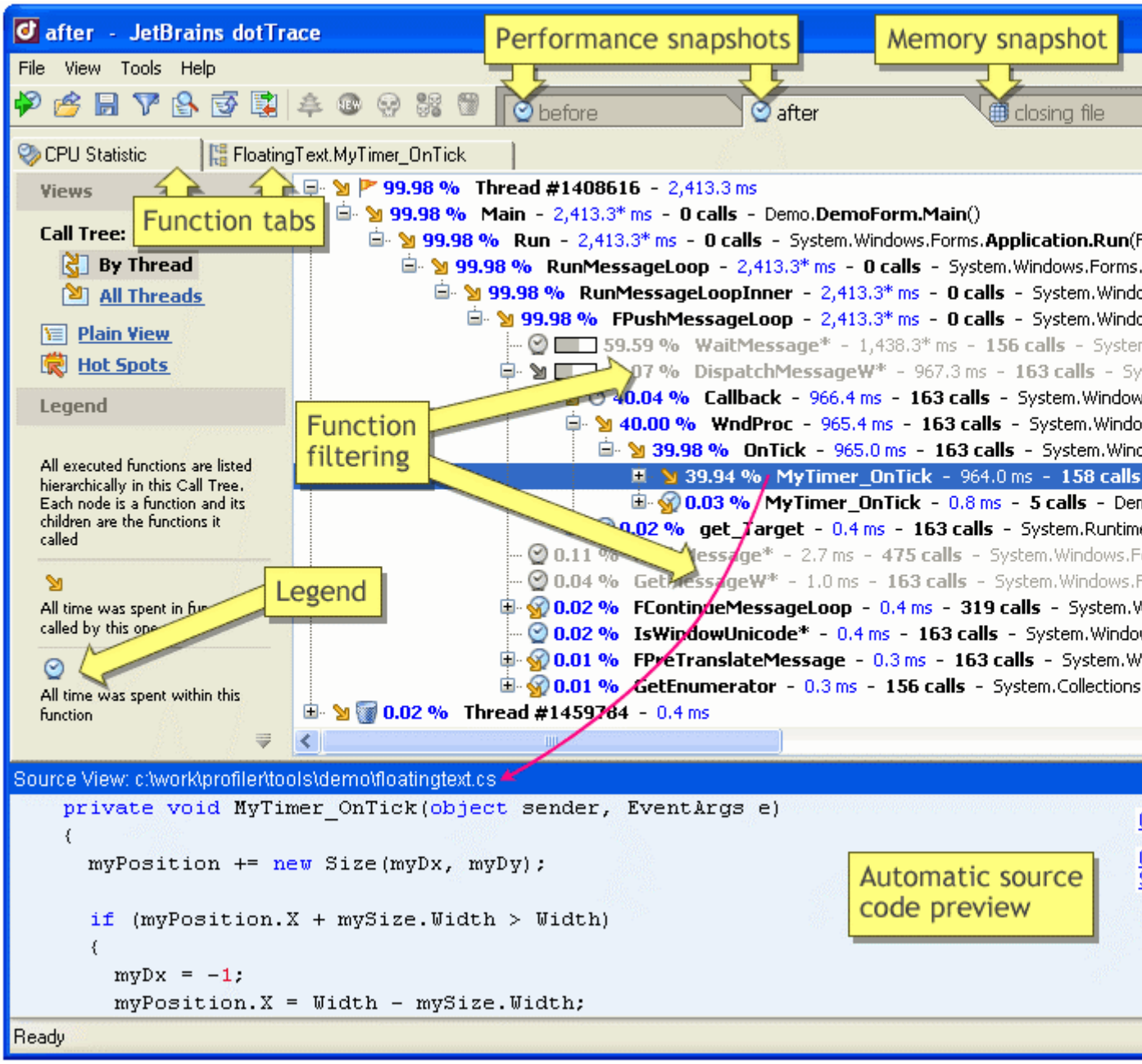
dotTrace vous permet d'identifier quels objets sont en mémoire à un instant donné, comment ils sont maintenus en vie (incoming references), et quels objets chaque objet maintient en vie (outgoing references). Vous pouvez aussi bénéficier d'un débuggage avancé avec les piles d'appel des allocations, la liste des objets morts, etc.



Voici une vue qui permet de voir les différences entre deux états de la mémoire :



dotTrace est aussi un profileur de performance :



dotTrace s'utilise en le lançant tout d'abord, puis en lui demandant de profiler une application en lui fournissant le chemin d'un fichier .EXE.
Si vous voulez inspecter la mémoire utilisée par votre application, vous pouvez prendre des instantanés pendant qu'elle tourne et demander à dotTrace de vous montrer les informations. La première chose que vous ferez probablement est demander à dotTrace de vous montrer combiens d'instances d'une classe donnée existent en mémoire, et comment elles sont maintenues en vie.

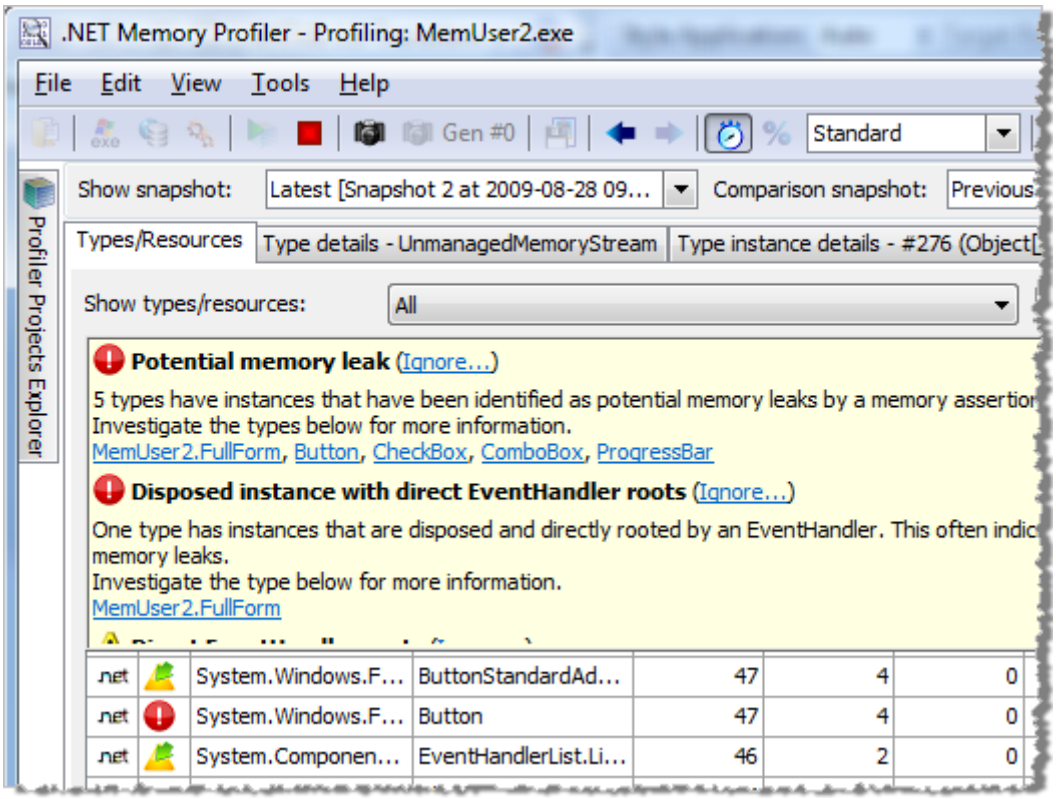
En plus de rechercher des instances managées, vous pouvez aussi rechercher des ressources non managées. dotTrace n'offre pas de support direct pour le suivi des ressources non managées, mais vous pouvez rechercher des objets wrappers .NET. Par exemple, vous pouvez regarder si vous trouvez des instances des classes Bitmap, Font ou Brush. Si vous trouvez une instance d'une telle classe qui n'a pas été disposée, alors la ressource système sous-jacente est toujours allouée à votre application.
Le prochain outil que je vais présenter offre un support intégré pour le suivi des ressources non

managées. Cela signifie qu'avec celui-ci, vous serez en mesure de rechercher directement des handles HBITMAP, HFONT ou HBRUSH.

.NET Memory Profiler

.NET Memory Profiler est un autre outil intéressant. Les fonctions utiles qu'il propose et que vous n'avez pas avec dotTrace comprennent :

- Visualiser les objets qui ont été disposés mais qui sont toujours en vie
- Visualiser les objets qui ont été libérés mais sans avoir été disposés au préalable
- Suivre des ressources non managées
- S'attacher à un processus en cours d'exécution
- S'attacher à un processus en même temps que le débuggeur de Visual Studio
- Analyse automatique de la mémoire (conseils et avertissements sur les problèmes fréquents avec la gestion de la mémoire)



Plusieurs autres profileurs .NET existent

Les outils cités jusqu'ici sont juste des exemples de ce qui existe pour vous aider. dotTrace et .NET Memory Profiler sont deux des nombreux profileurs de mémoire (et de performance) disponibles pour .NET. D'autres grands noms incluent ANTS Profiler, YourKit Profiler, PurifyPlus, AQtime et CLR Profiler. La plupart de ces outils offrent le même genre de services que dotTrace. Vous trouverez tout un ensemble d'outils dédiés au profilage en .NET sur SharpToolbox.com.

SOS.dll et WinDbg

Un autre outil que vous pouvez utiliser est **SOS.dll**. SOS.dll est une extension de débuggage qui vous aide à débugger des programmes managés dans le débuggeur WinDbg.exe et dans Visual Studio en fournissant des informations sur l'environnement interne du CLR (common language runtime). SOS peut être utilisée pour obtenir de l'information sur le garbage collector, les objets en mémoire, les threads et les verrous, les piles d'appel et plus.

Bien qu'il n'offre pas une interface graphique conviviale, SOS vous permettra de faire les choses que j'ai faites ci-dessus avec dotTrace.

WinDbg est l'outil que vous utiliserez le plus souvent quand vous vous attacherez à un processus en production. Vous pouvez en apprendre plus sur SOS.dll et WinDBg sur le blog de Rico Mariani, sur le blog de Mike Taulty (SOS.dll avec WinDbg and SOS.dll avec Visual Studio), et sur Wikipedia.

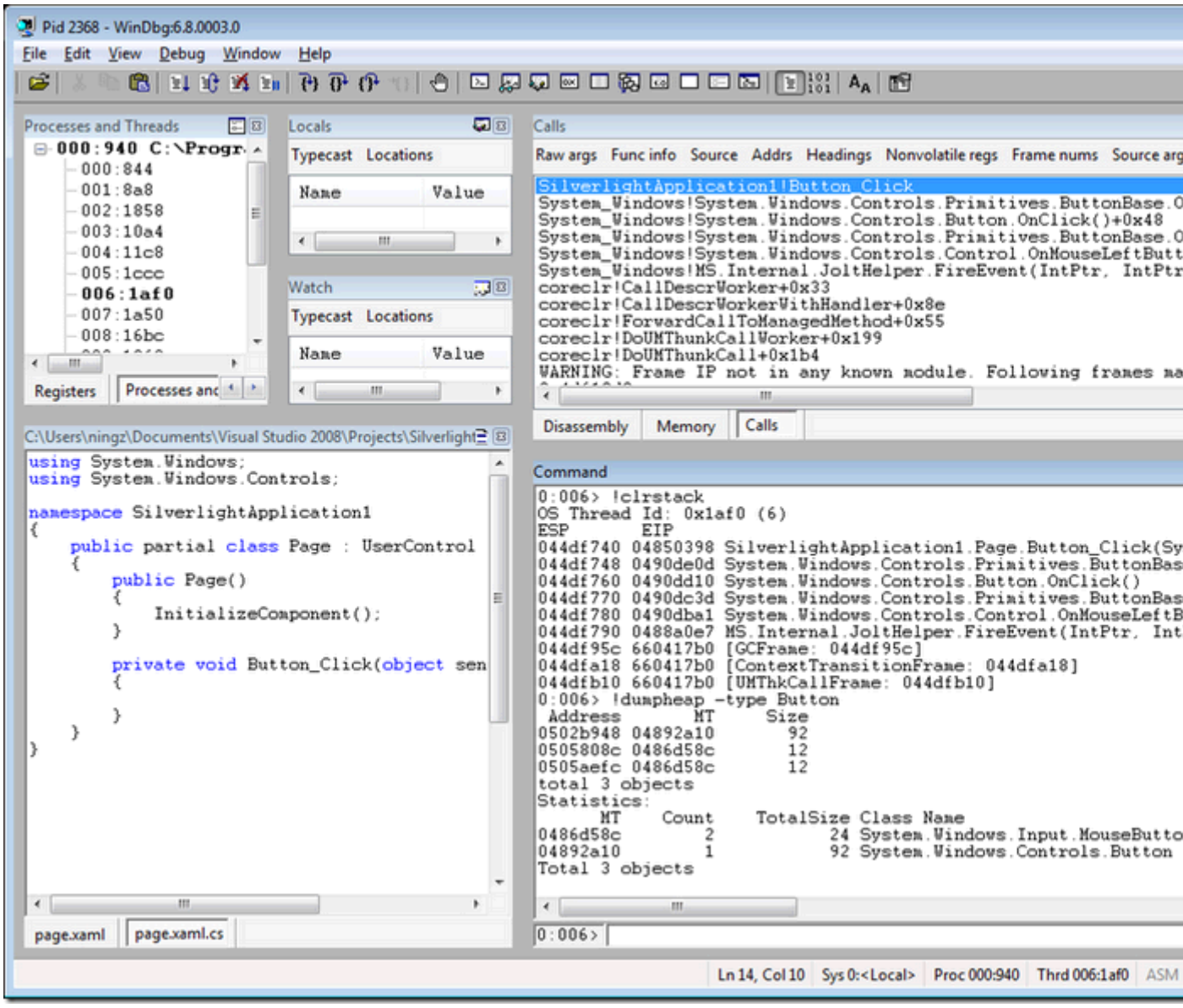
SOS.dll et WinDbg sont fournis gratuitement par Microsoft comme membres de l'offre Debugging Tools for Windows. Un avantage de SOS.dll et WinDbg en comparaison avec les autres outils que j'ai listés est leur faible consommation en ressources, tout en restant puissants.

Exemple de résultat obtenu avec sos.dll et la commande *gcroot* :

```
!gcroot 015b0f08
Note: Roots found on stacks may be false positives. Run "!help gcroot" for
more info.
Error during command: Warning. Extension is using a callback which Visual Studio does not implement.

Scan Thread 3492 OSTHread da4
ESP:7df450:Root:015dd70c(ASP.global_asax)->
0161fb48(System.Web.HttpContext)->
01624la0(System.Web.CachedPathData)->
016268b4(System.Web.Configuration.HandlerMappingMemo)->
01600fcc(System.Web.Configuration.HttpHandlerAction)->
055c9c1c(System.Configuration.RuntimeConfigurationRecord)->
055cb368(System.Collections.Hashtable)->
056lb788(System.Collections.Hashtable+bucket[])->
055ce3f8(System.Configuration.SectionRecord)->
055ceal4(System.Web.Configuration.TrustSection)->
055bbc58(System.Configuration.RuntimeConfigurationRecord)->
055acdcc(System.Configuration.RuntimeConfigurationRecord)->
055ae210(System.Collections.Hashtable)->
055b6cb4(System.Collections.Hashtable+bucket[])->
055af324(System.Configuration.FactoryRecord)->
015b0ed0(System.Configuration.RuntimeConfigurationRecord+RuntimeConfigurationFactory)->
015b0f08(System.CodeDom.Compiler.CodeDomConfigurationHandler)
```

Aperçu de WinDbg :

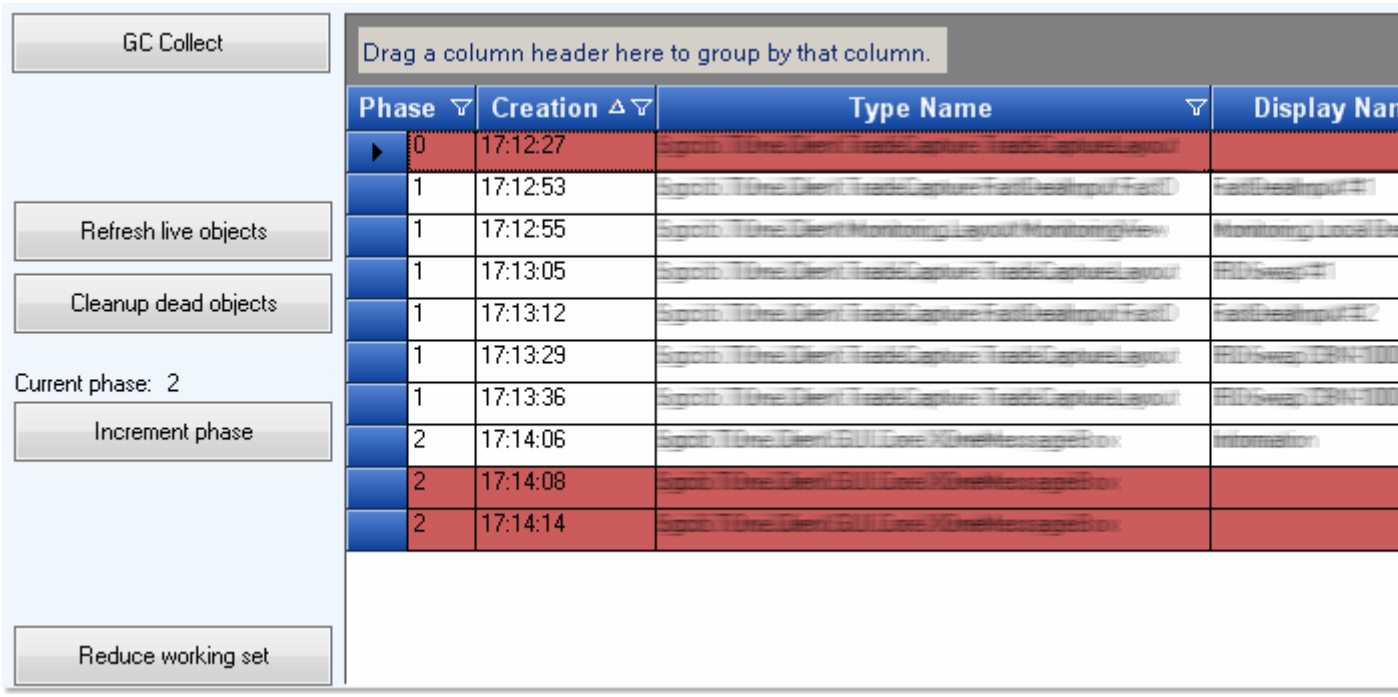


Outillage maison

En plus des outils proposés par le marché, n'oubliez pas que vous pouvez créer vos propres outils. Il peut s'agir d'outils standalones que vous réutiliserez pour plusieurs de vos applications, mais ceux-ci peuvent être un peu difficiles à développer.

Ce que nous avons fait pour le projet X est de développer des outils intégrés qui nous aident à suivre en temps-réel l'utilisation des ressources et les fuites potentielles.

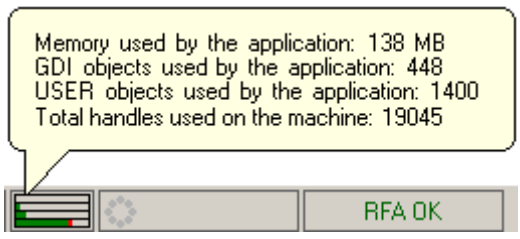
Un de ces outils affiche une liste des objets vivants et morts, directement depuis l'application principale. Il est constitué d'un service CAB et d'une vue CAB qui peut être utilisée pour vérifier si des objets que nous nous attendons à voir libérés l'ont bien été en réalité.
Voici une photo d'écran de cet outillage :



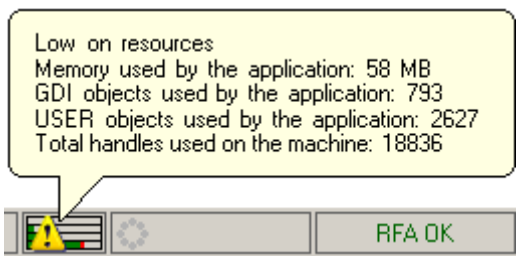
Suivre à la trace chaque objet de l'application serait trop coûteux et contre-productif étant donné le nombre d'objets impliqués dans une grosse application. En fait, nous ne gardons pas un œil sur tous les objets, mais uniquement sur les objets de haut niveau et les conteneurs racines de l'application. Ce sont les objets que j'ai conseillé de suivre quand j'ai expliqué comment détecter des fuites.

La technique que nous avons utilisée pour créer cet outil est très simple. Elle utilise les références faibles. La classe WeakReference vous permet de référencer un objet tout en permettant que cet objet puisse être recyclé par le garbage collector. En plus, elle vous permet de tester si l'objet référencé est mort ou vif, via sa propriété IsAlive.

Dans le projet X, nous avons aussi une widget qui fournit un aperçu de la consommation d'objets GDI et User :



Quand les ressources sont presque épuisées, la widget le signale avec un panneau d'avertissement :



En complément, l'application peut demander à l'utilisateur de fermer quelques uns des documents/fenêtres/onglets actuellement ouverts, et l'empêcher d'en ouvrir d'autres, tant que le niveau d'utilisation des ressources usage n'est pas repassé en dessous du niveau critique.

Pour lire le niveau d'utilisation actuel en ressources graphiques, nous utilisons la fonction GetGuiResources de User32.dll. Voici comment l'importer en C# :

```
// uiFlags: 0 - Count of GDI objects
// uiFlags: 1 - Count of USER objects
// GDI objects: pens, brushes, fonts, palettes, regions, device contexts,
bitmaps, etc.
// USER objects: accelerator tables, cursors, icons, menus, windows, etc.
[DllImport("User32")]
extern public static int GetGuiResources(IntPtr hProcess, int uiFlags);

public static int GetGuiResourcesGDICount(Process process)
{
  return GetGuiResources(process.Handle, 0);
}

public static int GetGuiResourcesUserCount(Process process)
{
  return GetGuiResources(process.Handle, 1);
}
```

Nous interrogeons la consommation de mémoire via la propriété Process.GetCurrentProcess().WorkingSet64.

Conclusion

J'espère que cet article vous a fourni une bonne base pour améliorer vos applications et pour vous aider à résoudre des fuites. Pourchasser les fuites peut être amusant... si vous n'avez rien de mieux à faire de votre temps :-) Parfois, cependant, vous n'avez pas le choix car résoudre les fuites est vital pour votre application.

Une fois que vous avez résolu les fuites, vous avez encore du travail. Je vous encourage vivement à améliorer votre application de manière à ce qu'elle consomme le moins de ressources possible. Sans perdre en fonctionnalité, bien entendu. Je vous invite à lire mes recommandations à la fin de cet entrée de blog.

Ressources

Le code source code de l'application de démonstration est disponible au téléchargement.

Voici quelques ressources intéressantes additionnelles si vous voulez creuser plus avant :

- Jossef Goldberg : *Finding memory leaks in WPF applications*
- Tess Ferrandez a une série d'articles sur les problèmes de mémoire (ASP.NET, WinDbg, et autre)
- Article MSDN par Christophe Nasarre : *Resource leaks: Detecting, locating, and repairing your leaky GDI code*
- Article en français par Sami Jaber : *Audit et analyse de fuites mémoire*
- Article par Joydip Kanjilal: *When and how to use Dispose and Finalize in C#*
- Article CodeProject par Shivprasad koirala : *Detecting .NET application memory leaks* (avec PerfMon)
- Mon entrée de blog à propos du Desktop Heap
- Mon entrée de blog à propos des lapsed listeners
- Mon entrée de blog qui montre comment forcer le désabonnement à un événement

English version of this article

A propos de l'auteur

Fabrice Marguerie

Je suis un expert .NET indépendant, reconnu MVP par Microsoft depuis 2004.
J'interviens sur des missions d'expertise technique et d'architecture.
N'hésitez pas à me contacter si vous avez besoin d'aide sur vos projets.

Je suis également auteur du livre LINQ in Action et d'autres articles, ainsi que responsable ou coresponsable des sites Proagora.com, SharpToolbox.com, JavaToolbox.com et AxToolbox.com

Mon profil Proagora

Vos remarques, suggestions, et questions sont les bienvenues.

Septembre 2009