Rapport

Capture And Replay

Travail d'Étude et de Recherche (M8)

Rémy GIRAUD
Tarek MODHAFAR

10 juin 2004

Laboratoire Informatique
Signaux et Systèmes de
Sophia Antipolis (I3S)

Enseignants reponsables : Philippe Lahire
Pierre Crescenzo

Table des matières

1 Introduction

1.1 Présentation des encadrants

Commençons par présenter les personnes à l'origine de ce projet de *Capture and Replay.*
Tout d'abord monsieur Philippe Lahire qui est notre encadrant principal. Il est Maître de conférences à l'Université de Nice - Sophia Antipolis. Il effectue ses activités de recherche au sein du Laboratoire I3S (Laboratoire d'informatique) où il appartient au projet "Objets et Composants Logiciels" et il est en délégation à l'INRIA. Ses activités d'enseignement sont principalement au département informatique de l'Institut Universitaire de Technologie de Nice Côte d'Azur.
Mr. Lahire s'était déjà penché sur la question durant l'été 2003 avec Mme. Karine Arnout. Elle est assistante de recherche dans la Chaire de Génie logiciel dirigée par le professeur Dr. Bertrand Meyer (concepteur du langage *Eiffel*) à l'École Polytechnique de Zürich (ETH Zürich).
Le second enseignant responsable de notre sujet de TER est monsieur Pierre Crescenzo. Il est enseignant au sein du Département d'Informatique de la Faculté des Sciences de Nice qui est une unité de l'Université de Nice-Sophia Antipolis. IL est chercheur dans le Projet "Objets et Composants Logiciels" au sein du Laboratoire Informatique, Signaux et Systèmes qui est une unité mixte de l'Université de Nice-Sophia Antipolis et du Centre National de la Recherche Scientifique.

1.2 Les étudiants

Nous sommes deux étudiants à avoir travaillé sur le TER *Capture and Replay.* Nous -Rémy Giraud, Tarek Modhafar- sommes étudiants en maîtrise d'informatique de lŠUniversité de Nice-Sophia Antipolis.

1.3 Résumé

Nous allons dans cette partie brièvement décrire notre sujet de TER. Il s'agit de concevoir et d'implémenter un ensemble de fonctionnalités qui permettent à une application graphique écrite en *Eiffel* (**EiffelStudio 5.4**) de mémoriser les événements générés par l'utilisateur lors d'une première utilisation et de les rejouer à sa prochaine exécution (mécanisme de "*Capture/Replay*"). Cette application sera dédiée uniquement à *Windows*. Les événements à capturer pourront être, par exemple, un clic de souris, un déplacement du curseur, la frappe d'un caractère, etc ...

La bibliothèque graphique d'*Eiffel* (**EiffelVision**) est implantée au dessus de l'*API* graphique de *Windows* et elle est vraiment calquée sur cette dernière. Elle est interfacée avec *C* pour atteindre l'*API* de *Windows* et les fonctionnalités à développer seront donc principalement écrites en *C* mais il est conseillé de programmer en *Eiffel* si c'est possible.

Le travail demandé inclut à la fois des problèmes de conception et d'implémentation. Il sera en particulier important et définir un moyen d'identifier les objets graphiques afin de pouvoir reconnaître l'objet sur lequel s'applique l'événement lorsqu'on exécute à nouveau la même application. Il sera aussi intéressant d'étudier comment il est possible de rejouer certains événements même lorsque l'application a été légèrement modifiée entre temps.

1.4 Cahier des charges synthétique

Nous avons repris le cahier des charges de la pré-soutenance que nous avons résumé succinctement.

1.4.1 Fournitures

- *Windows*
- *EiffelStudio 5.4*
- Un pack de fichiers Capture and Replay
- Un script d'installation

1.4.2 Processus

- Recherche d'informations sur les techniques de *Capture And Replay* déjà pré existantes pour s'en inspirer.
- Installation des outils de développement (*EiffelStudio*, *C*...).
- Prise en main des logiciels et apprentissage de *Eiffel* (le minimum de chose à savoir).
- Intégration des fichiers *Eiffel* fournis (intégration à notre environnement du travail déjà réalisé).
- Étude des événements qui passe à travers de *Eiffel* et à récupérer avec l'*API Windows*.
- Modélisation d'une structure permettant de représenter les *Widgets*, avec un minimum d'information.
- Implémentation du *Capture And Replay*.

1.4.3 Objectifs et Priorités

- Avancer dans l'approche du *Capture And Replay*.
- Identifier les événements non capturés et d'arriver à les reproduire.
- Bien documenter notre travail.

1.4.4 Dépendances, contraintes

- Interaction entre *Eiffel* et *Windows*.
- Utilisation d'*Eiffel* plutôt que de *C*.

1.4.5 Risque

Il sera très facile de partir sur une mauvaise piste, vu que nous ne connaissons pas bien comment sont gérés les événements. Il faudra donc bien s'intéresser à cette partie et faire beaucoup de petits tests avant de proposer une solution pour pouvoir représenter les *Widgets*.

1.4.6 Méthodes et outils

- Conception et implémentation des classes en *Eiffel* réalisées avec *EiffelStudio*.
- Liaison entre l'*API Windows* et la bibliothèque graphique. d'*EiffelVision* développée *C* avec le compilateur *Borland C++ 5.5*.
- La bibliothèque de *Windows*, la *msdn*.

1.4.7 Échéancier

Voici comment nous avons planifié notre projet, à partir du 10 mai 2004 :

10/05 → 10/06	Prise en main des logiciels et apprentissage de *Eiffel* (le minimum de chose à savoir).
10/05 → 12/05	Intégration des fichiers *Eiffel* fournis (intégration à notre environnement du travail déjà réalisé).
13/05 → 19/05	Étude des événements qui passe à travers de *Eiffel* et à récupérer avec l'*API Windows*.
19/05 → 30/05	Modélisation d'une structure permettant de représenter les *Widgets*, avec un minimum d'information.
31/05 → 10/06	Implémentation, documentation et test du *Capture And Replay*.

1.4.8 Fonction du produit

- *Capture* : opération transparente, fonctionnalité intégrée à l'environnement de développement *EiffelStudio*. Cette opération stockera dans un fichier texte les événements générés par l'utilisateur lors d'une première utilisation de son application.
- *Replay* : fonctionnalité appelée par l'utilisateur. Le fichier texte employé à la capture permettra de le rejoue.
- *Fichier de Sauvegarde* : l'utilisateur pourra choisir le nom du fichier texte de Capture. Ainsi, il pourra rejouer différentes versions de son application lors du *Replay*.

1.4.9 Plate forme matérielle

Ce projet est basé sous *Windows*, et ne fonctionnera pas sur les autres plates-formes car nous utiliserons des fonctions de l'*API* de *Windows*.

1.4.10 Temps de réponse / Empreinte mémoire

- *Capture* : stockage des informations sur les composants dans un fichier ne devant pas ralentir l'exécution.
- *Replay* : la lecture des informations stockées dans le fichier ne devant pas ralentir l'exécution.
- *Fichier de Sauvegarde* : contient l'information vitale pour le rejoue.

2 Planning

Maintenant nous allons décrire succinctement les différentes parties du sujet et les tâches que nous avons réalisé pour ce projet (certaines des taches seront détaillées dans la section 3). Nous indiquerons aussi le temps passé sur chaque tâche.

2.1 Découpage en tâches

Voici les différentes tâches qui composent notre projet :

- **Prise en main** : étude des éléments pertinents du langage de programmation *Eiffel*, en rapport avec le projet. Et prise en main de l'environnement de développement *EiffelStudio 5.4* (voir 3.2.1).

- **Intégration des C & R** : phase d'intégration des fichiers de *Capture and Replay* à l'environnement de développement (voir 3.2.3).

- **Étude C & R** : étude des classes de *Capture and Replay*. Cette tâche était initialement regroupée avec la précédente, mais elle a abouti avant (voir 3.1.4, 3.2.3, 3.1.3).

- **Étude Eiffel** : étude des classes de la bibliothèque *Eiffel* (voir 3.1.4).

- **Étude msdn** : étude de la bibliothèque de *Microsoft*, l'*API* de *Windows* (voir 3.1.4).

- **Étude Messages** : étude des messages qui passe à travers de *Eiffel* et à récupérer avec l'*API Windows*. Cette tâche était prévue dans le planning initial. Après réflexion nous avons transformé cette tâche en tâche message.

- **Modélisation Widgets** : Modélisation d'une structure permettant de représenter les *Widgets*, avec un minimum d'information. Cette tâche a été supprimée parce que la modélisation qui a été faite, était en parfaite adéquation avec ce que nous voulions faire.

- **Messages** : Tâche regroupe la recherche et l'implémentation des messages en vue de leur capture (voir 3.1.4, 3.1.4, 3.1.4).

- **Rejoue** : Tâche de débugage du *Replay*. Durant cette phase nous avons essayé de trouver pourquoi nous avions des levées d'exceptions pendant le rejoue.

- **process message** : Cette tâche peut s'intégrer à la tâche message, mais nous l'avons séparée parce qu'elle présente un tournant majeur dans le projet (voir 3.1.4).

- **Temporisation** : Introduction de la notion de temps durant le rejoue (voir 3.2.4).

- **Thread** : Tâche visant à permettre la gestion de certains messages pendant la phase de rejoue. Ces messages sont générés par l'utilisateur pendant la phase de rejoue (voir 3.2.5).

- **Script** : Script d'installation et de sauvegarde des fichiers entrant en jeu dans le *Capture and Replay* (voir 3.2.6).

- **Makefile** : Tentative d'introduction d'un fichier de Makefile. Nous n'avons pas trouvé la solution pour que le compilateur *C* inclut avec *Eiffel* prenne en compte notre Makefile.

- **Finalisation** : Rédaction du rapport, du manuel d'utilisateurs, du manuel de maintenance et nettoyage des sources pour le livrable.

2.2 Planning

Dans notre planning, la durée en jours indique le cumul total des jours des intervenants pour chaque tâche.

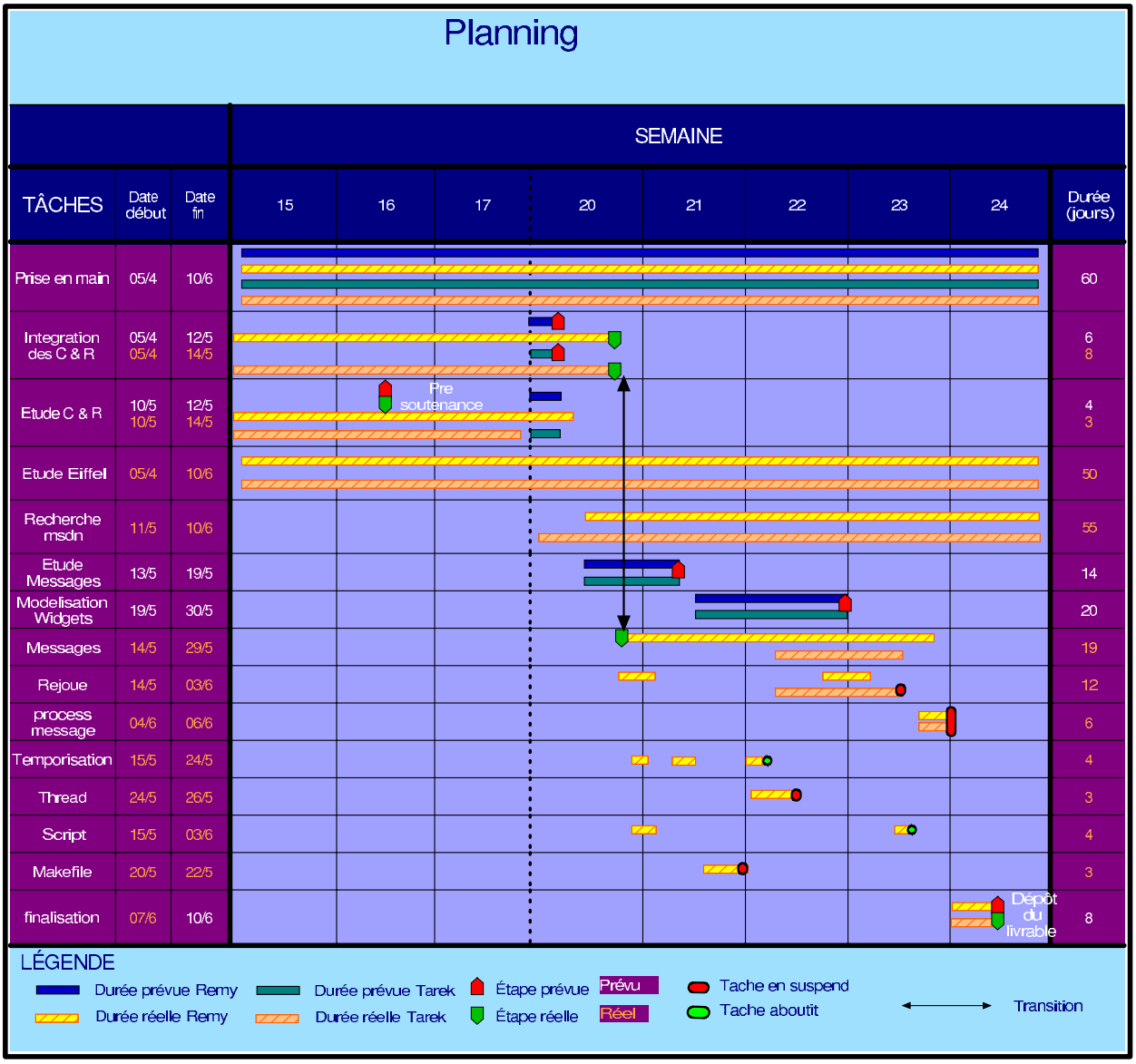

FIG. 1 – Planning

3 Description travail réalisé

3.1 Description générale

Nous allons, dans cette partie, vous expliquer le fonctionnement général d'une application écrite en *Eiffel* sous *Windows*, voir comment sont gérés les messages ou événements, ce qui nous a été fourni pour les capturer, afin de mieux comprendre notre travail. Cette phase de compréhension des applications écrites en *Eiffel* est primordiale pour réaliser le *Capture And Replay*. Elle s'est étalée sur toute la phase du projet, au fur et à mesure des problèmes rencontrés et encore aujourd'hui, nous n'avons pas tout compris. Mais avant tout, commençons par rappeler l'intérêt de disposer du *Capture And Replay*.

3.1.1 Intérêt du Capture And Replay

En réalisant le *Capture And Replay*, nous permettons à un utilisateur de reproduire toutes ces actions qui ont un effet visuel sur l'application. Ainsi il pourra faire des scénarios de ces applications pour montrer leurs intérêts. Si l'implémentation le permet (tout dépend comment nous l'implémentons), le *Capture And Replay* pourrait très bien être utilisé pour arriver dans un certain état de l'application par exemple juste avant de cliquer sur le bouton ouvrir qui fait buger l'application. En effet, nous nous sommes placés très bas dans les classes *Eiffel*, et nous nous contentons seulement de rejouer un événement (ou message). Si le message a pour but de notifier à la fenêtre que l'utilisateur a cliqué sur le bouton ouvrir, la fenêtre fera appel au traitement de l'événement (ou message) qui déclenchera l'appel à la fonction de l'utilisateur dès que nous cliquons sur le bouton. Nous ne nous séparons donc pas du contexte d'exécution du programme du *Capture And Replay*. Lorsque le client veut rejouer son application, il relance **son** application et pas une que nous aurions pu écrire. Tout est intégré dans les classes *Eiffel* que nous avons modifiées. Utiliser ces classes permet donc d'avoir le *Capture And Replay*, tout comme nous utilisons une classe qui produit un bouton clignotant. Si l'utilisateur se sert de cette classe, alors il disposera d'un bouton qui clignote sans avoir à rien faire. Pour le *Capture And Replay*, c'est pareil, si l'utilisateur se sert de nos classes, sans rien avoir à faire il disposera du *Capture And Replay*. Nous insistons bien sur ce point car il est important de bien le saisir.

3.1.2 Localisation des fichiers susceptible d'être mis en jeu

Nous allons, ici, localiser sommairement où mettre en œuvre le *Capture And Replay* et pourquoi s'intéresser à ces endroits. Le *Capture And Replay* s'effectue sur des composants graphiques, nous souhaitons jouer puis rejouer une succession d'événements qui ont un impact visuel sur l'écran. Il faut donc avoir une application graphique et à partir de celle-ci, capturer les actions effectuées par l'utilisateur pour pouvoir les rejouer ultérieurement.

Dans le cadre de ce projet, nous nous sommes intéressés qu'aux applications sous *Windows* et écrites en *Eiffel*, celui-ci étant un langage de programmation objet très complet. Il permet d'écrire des applications graphiques simplement, sous n'importe quelle plateforme. Vu que nous nous intéressons qu'aux applications écrites sous *Windows*, nous nous sommes penchés que sur certaines classes. Bien évidement, nous avons accès à tout le code des classes de *Eiffel*, et surtout à tout le code de la bibliothèque graphique, sans quoi nous n'aurions pas pu mettre en œuvre le *Capture And Replay*.

Dans *Eiffel*, nous retrouvons (pour *Windows*) 2 grandes catégories de classes :

- WEL_classes qui représentent les classes graphiques pour *Windows* de la bibliothèque de *Eiffel* (**W**indow **E**iffel **L**ibrary).

- EV_classes qui représentent un ensemble de classes graphiques *Eiffel* portable au niveau des plateformes (**E**iffel **V**ision).

Les *EV_classes* s'appuient par exemple sur les *WEL_classes* pour les applications sous *Windows* et elles s'appuient sur les *GTK_classes* (classes graphique pour *UNIX*) pour des applications tournant sous *UNIX*.

C'est donc au niveau des *EV_classes* et des *WEL_classes* que nous nous sommes intéressés et que nous avons étudié tout au long de notre projet.

3.1.3 Fourniture

Il y a quand même énormément de classes, même si on se limite aux *EV_classes* et aux *WEL_classes*. Pour nous aider, nos encadrants nous ont fourni une ébauche de classes *Eiffel* permettant de mettre en œuvre le *Capture And Replay*, c'est-à-dire un ensemble de fonction *Eiffel* permettant de capturer un événement et de pouvoir le sauvegarder sur le disque dur. Il y avait aussi un début d'amorce de Capture dans certains fichiers *WEL* ou *EV*, qui nous a permis de localiser à peu près les endroits où des événements sont créés.

Les fichiers *WEL* ou *EV*, modifiés par nos encadrants, étaient destinés pour *Eiffel Studio 5.3*. Entre temps, une nouvelle version est sortie, la *5.4*, et l'implémentation des *WEL_classes* et des *EV_classes* avait changée. Du

coup, le travail effectué ne fonctionnait plus du tout, ce qui n'était pas une mauvaise chose car il nous a obligé à vraiment nous plonger dans le code pour le remettre en état de marche.

La première étape de notre travail a donc été de comprendre les fichiers fournis pour repartir de ces fichiers. Nous reviendrons sur l'utilité des classes de *Capture And Replay* et leur fonctionnement dans les parties suivantes (voir section 3.1.4 Intégration et compréhension des fichiers de Capture And Replay, section 3.2.3 Compréhension du mécanisme des événements sous Eiffel).

3.1.4 Compréhension du mécanisme des événements sous Eiffel

Nous avons vu dans les parties antérieures les fournitures données mais pour pouvoir s'en servir à bon escient, il est indispensable de comprendre comment les événements sont gérés avec *Eiffel* sous *Windows*.

Corrélation Eiffel, API Windows, fonction C

Tout d'abord, il faut rappeler que *Eiffel* permet assez facilement de faire appel à des fonctions d'un autre langage de programmation grâce à une clause *External*. *Eiffel* s'appuie énormément sur ce principe pour mettre en œuvre ces applications graphiques. Ainsi, il n'est pas étonnant de retrouver des appels à des fonctions *C* de l'*API* de *Windows* pour, par exemple, afficher une boîte de dialogue ou afficher des menus etc. *Eiffel* a la possibilité soit d'appeler ces propres fonctions *C*, soit directement les fonctions de l'API de *Windows*. De ce fait, *Eiffel* n'utilise pas seulement des fonctions *C* pour afficher des éléments graphiques, mais il a reproduit entièrement le même système de notification de messages et d'envois de messages que *Windows*. Nous constatons donc en étudiant les classes *Eiffel*, que certaines sont des "*Wrappeur*" de fonctions et de constantes de l'*API* de *Windows*. Par exemple la constante **WM_PAINT** est déclarée dans le fichier *C* "*Winuser.h*" de l'API de *Windows*, et nous retrouvons cette même constante dans la classe *Eiffel* "*WEL_WM_CONSTANTS.e*" avec le nom **Wm_paint**. Nous avons pris cet exemple mais nous aurions pu le faire avec toutes les autres constantes déclarées dans le fichier " **WEL_WM_CONSTANTS.e** ".Ainsi, en étudiant la documentation de *Windows* (la MSDN surtout), nous obtenons beaucoup d'informations sur ce que représente un message ou sur ce que fait une fonction de l' *API* de *Windows*.

Les messages sous Eiffel Windows

Comme nous l'avons déjà dit, *Eiffel* a reproduit le même système que sous *Windows*. Un message pour *Windows* est une structure *C* contenant 4 champs essentiels :

- **hwnd** qui est un pointeur sur une fenêtre (un élément graphique, Widget).
- **message** qui est le numéro du message par exemple **Wm_paint**.
- **lparam** paramètre en plus du message.
- **wparam** idem à lparam.

Dans *Eiffel*, nous retrouvons une classe de message *WEL_MSG* avec justement ces 4 variables qui sont mises à jour grâce à des appels à des fonctions *C*.

Le **hwnd** représente en fait la fenêtre ou *Widget* sur lequel le message doit être traité, c'est-à-dire à qui est destiné le message **message**. Il se peut que le message est besoin de paramètres, c'est ce que représente les 2 entiers **lparam** et **wparam**. Par exemple, le message envoyé lorsque nous pressons une touche a besoin de préciser la touche pressée, elle sera donc contenue dans un des champs **lparam** ou **wparam**. Les messages ou événements (même signification) permettent à l'application de rester dans un état cohérent. Par exemple, lorsque nous masquons l'application par une fenêtre et que nous revenons à l'application, celle-ci doit être redessiner, donc le système poste un message de type **Wm_paint** (le champ **message** est égal à **Wm_paint**) qui devra être traité ultérieurement.

Dans la phase de capture, c'est donc ces instances de messages que nous devons mémoriser d'une façon ou d'une autre, mais avant tout, il faut savoir où sont générés ces événements et où faut-il se placer pour les intercepter.

Comment capturer / rejouer les messages ?

Nous avons vu juste avant comment étaient identifiés les messages maintenant il serait bon de savoir comment les capturer. Cette partie n'a pas été réalisée par nous même, puisque les fichiers donnés par nos encadrants réalisaient déjà cette action. Toutefois, il a été indispensable de savoir comment les classes fournies fonctionnaient, c'est ce que nous allons vous expliquer sommairement dans cette partie avant d'y revenir dans la partie : Intégration et compréhension des fichiers de *Capture And Replay* (sec 3.2.3).

Le principe est assez simple, les fonctions écrites dans les fichiers fournis permettent à partir d'un message (*WEL_MSG*) de le mémoriser dans un tableau lors de la capture, et une fois que l'application graphique a été quittée,

ce tableau est stocké sous forme de fichier sur le disque dur. Ensuite, lors de la prochaine exécution de l'application graphique, les fonctions dans les fichiers fournis nous permettent de récupérer les messages qui sont stockés dans le fichier et de pouvoir nous en servir lorsque nous le désirons.

Où capturer les messages ?

Nous disposons donc de tout le nécessaire pour capturer ou récupérer un message, il faut donc savoir où les messages sont générés et traités pour les capturer à ce niveau là.

Message_loop de EV_APPLICATION_IMP

Un des premiers endroits où nous avons voulu capturer les messages se situe dans la classe *EV_APPLICATION_IMP* de *Eiffel*, au niveau de la fonction **message_loop**. En effet, cette fonction représente la boucle de traitement de message de l'application. Lorsque cette boucle termine, l'application se termine.

Nous allons ici expliquer son fonctionnement, qui s'inspire largement d'une boucle d'événement que nous écririons pour une application en *C* pour *Windows*.

Commençons par introduire le mot Système que nous utiliserons dans ce rapport. Lorsque nous parlerons de système, nous parlerons de *Windows*. Par exemple, si nous disons que les messages sont générés automatiquement par le système cela signifiera que *Windows* aura généré des messages, *Eiffel* n'intervenant. L'API de *Windows* propose des fonctions comme **PostMessage** ou **PeekMessage** qui permettent d'enregistrer un message dans une file de messages et de lire le premier message de la file en le retirant de la file. *Eiffel* utilise ces fonctions de *Windows* pour enregistrer les messages et pour les lire. C'est au niveau de **message_loop** que justement ces messages (ou événements) sont lus et retirés de la file de message de *Windows*. Comme le traitement de la boucle de message est calqué sur le modèle de *Windows*, nous allons tout d'abord expliquer comment nous la ferions sous *Windows*, avant d'expliquer comment *Eiffel* l'a réalisé.

Sur le site de la MSDN, le mécanisme de traitement des messages est assez bien expliqué. Il y a une boucle qui consulte les messages dans la file de message, ce message à peut être besoin d'être modifié (appel à une fonction **TranslateMessage** qui, par exemple, pour une saisie d'une touche de clavier, va convertir un des champ du message en la valeur de la touche saisie) puis ce message devra être envoyé à la **window_procedure** concernée (appel à la fonction **Dispatch** qui dispatche le message à la bonne **win-**

dow_procedure). La **window_procedure**est une procédure qui inspecte le code du message et en fonction de cette valeur applique la bonne fonction pour traiter le message.

Au niveau de *Eiffel*, nous retrouvons ce système avec quelques légères différences (voir schéma récapitulatif fig 2). Dans le modèle de *Windows*, chaque élément graphique ou chaque *widget* a sa propre **window_procedure.** Dans le cas de *Eiffel*, il n'y a qu'une seule **window_procedure** et dans cette procédure, il va y avoir un appel à la fonction **process_message.** Cet appel s'effectue sur un objet qui représente l'élément graphique et par le biais du Polymorphisme de *Eiffel*, la bonne fonction **process_message** est appelée. Dans cette dernière fonction, il y a tous les traitements de messages en fonction du type de message, ce qui correspond au contenu d'une **window_procedure** dans le modèle de *Windows*. Pour résumer, sous *Windows*, chaque éléments graphiques a sa propre **window_procedure** pour traiter les messages, chaque élément graphique enregistrant l'adresse de leur fonction **window_procedure** au niveau de l'application et la boucle de message dispatche le message à la bonne **window_procedure.** Pour *Eiffel*, chaque élément graphique a sa propre fonction **process_message** (équivalent de la **window_procedure** pour *Windows*), la boucle de message dispatche le message à la seule **window_procédure** qui existe, et dans cette dernière et grâce au polymorphisme, fait appel à la bonne fonction **process_message.**

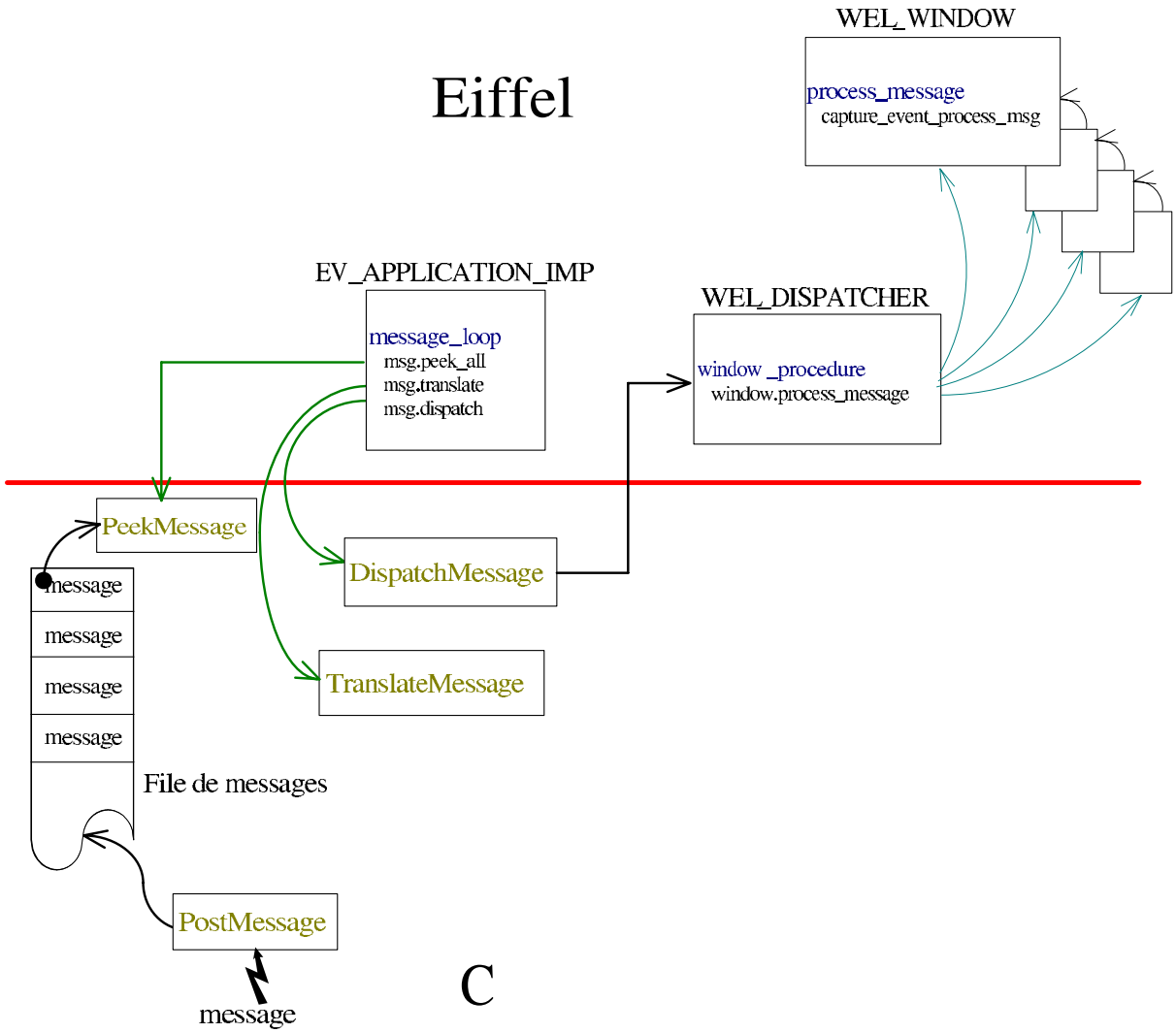

FIG. 2 – Description boucle de traitement de messages

Nous venons de voir le principe de la gestion de la boucle d'événements de *Eiffel* et nous avons expliqué ce dont nous avons compris et retenu à ce sujet. C'est donc à ce niveau là que, dans un premier temps, nous avons capturé nos premiers événements. A chaque fois que l'application lisait un message (**peek_all**) nous capturions le message. Ainsi nous avons pu récupérer tous les messages de déplacements de souris, de pression sur le clavier et autres.

En revanche nous avons vite constaté que tous les événements relatifs aux boîtes de dialogues ne sont pas capturés. C'est-à-dire que des messages ne passent pas par la boucle de traitement des messages, pour des messages générés par le système par exemple. En effet, certains messages, qui sont postés par le système, ne sont pas mis dans la file d'attente des messages

et le système fait appel directement à la **window_procedure** concernée. Pour *Eiffel* et les boîtes de dialogues, la **window_procedure** appelée et la **dialog_procedure** de *WEL_DISPATCHER* qui ressemble à la fonction **window_procedure** de la même classe.

dialog_procedure de WEL_DISPATCHER

Comme nous venons de le voir, en nous plaçant au niveau de la boucle de lecture des messages, certains n'y passent pas et c'est le système qui appelle directement la bonne "**window_procedure**". Pour une meilleure compréhension, nous avons expliqué que sur le modèle de *Eiffel* il n'y avait qu'une seule "**window_procedure**". En fait, il y en a deux, toutes les deux situées dans la classe *WEL_DISPATCHER*, l'une qui s'appelle **window_procedure** et l'autre **dialog_procedure.**
L'application (EV_APPLICATION_IMP ou WEL_APPLICATION), va donc enregistrer ces deux adresses au niveau de WEL_DISPATCHER et en fonction du message, l'une ou l'autre de ces fonctions sera appelée.

Les messages qui passent dans la fonction **dialog_procedure** ne semblent pas être dans la file d'attente des messages lue par la fonction **message_loop** de EV_APPLICATION_IMP, c'est donc à ce niveau là que nous pouvons capturer les messages relatifs aux boîtes de dialogues.

A ce niveau là, nous avons pu localiser les messages sur la fenêtre principale, les touches du clavier, certains messages des boîtes de dialogues mais aucun message relatif aux barres de menus n'est capturé. Il passerait par un autre endroit, mais où ?

Process_message de WEL_WINDOW et ces descendants

Comme tous les messages ne semblent pas être capturés, nous avons continué à chercher où nous pourrions trouver les messages relatifs aux barres de menus. En ce plaçant au niveau des fonctions **process_message** nous avons réussi à obtenir directement pleins d'événements, relatifs aux fenêtres aux touches du clavier, aux boîtes de dialogues et surtout aux barres de menus. Nous vous rappelons que **process_message** est la fonction qui est appelée par **window_procedure** dans le modèle de *Windows* et qui sert à traiter le message, c'est-à-dire appeler la bonne fonction de traitement du message. Il suffit donc de se placer au niveau cette fonction dans la classe *WEL_WINDOW* et de capturer tous les messages ici. Il faut bien évidement voir tous les fils de *WEL_WINDOW* qui redéfinissent cette fonction, ce qui est très facile avec l'environnement de programmation *EiffelStudio*. En ce plaçant ici, il semble que nous récupérons tous les messages intéressants.

3.2 Travail effectivement réalisé

Dans la partie précédente, nous avons expliqué le principe de fonctionnement des applications *Eiffel* au niveau de la gestion des messages et nous avons localisé les points importants pour capturer les messages. Maintenant que nous avons expliqué tout ce fonctionnement, nous allons pouvoir entrer plus en détail sur le travail réalisé en approfondissant certaines des tâches que nous avons réalisées.

3.2.1 Prise en main de EiffelStudio et du langage Eiffel

Pour mener à bien notre projet, nous avons utilisé le langage *Eiffel* et son environnement de développement *EiffelStudio 5.4*. Nous n'avions jamais programmé en *Eiffel*, ce projet nous l'a fait découvrir. C'est un langage de programmation orienté objet qui a comme caractéristique majeure d'accepter l'héritage multiple avec un mécanisme d'héritage de fonctions "*feature*" beaucoup plus complet que tous les autres langages de programmation objets que nous connaissions avant ce projet. Sa syntaxe est relativement simple est classique, elle ressemble assez à la syntaxe en *Ada* par exemple, avec les mêmes opérateurs d'affectation et de comparaison. Comme nous devions écrire le code le plus possible en *Eiffel*, nous avons essayé le plus possible d'apprendre ce langage pour obtenir un résultat le plus propre possible, d'autant plus qu'à notre sens, tout le projet pouvait être réalisé en *Eiffel*.

Pour nous aider dans l'apprentissage de ce langage, nous disposions de l'environnement de développement *EiffelStudio 5.4*. Ce dernier, après une prise en main un peu difficile du au manque de connaissance de son fonctionnement, nous a fait gagner énormément de temps et nous pensons que sans lui, nous n'aurions jamais pu faire tout ce que nous avons réalisé. Notre travail étant essentiellement un travail de recherche, il a fallu naviguer dans les différentes classes, savoir quelle fonction est appelée et par qui etc. *EiffelStudio* permet de faire toutes ces opérations très facilement.

3.2.2 Environnement de développement EiffelStudio 5.4

Nous allons, ici, faire un rapide tour d'horizon des fonctionnalités de *EiffelStudio* afin de montrer son utilité et ses points forts pour développer un projet.

La navigation dans EiffelStudio

Il est très facile de naviguer d'une classe à l'autre, de fonctions en fonctions, juste par des "*drag en drop*" dans la fenêtre. Voici la figure qui illustre ceci fig 3.

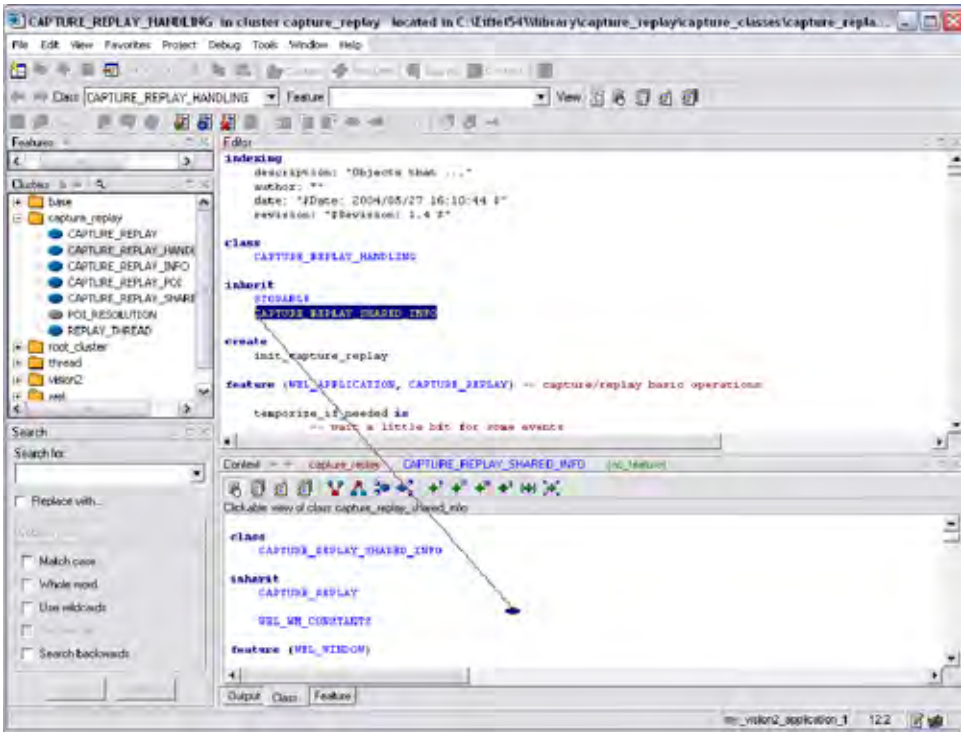

FIG. 3 – Le glisser déposer de EiffelStudio

Nous pouvons cliquer sur une classe pour la mettre sur l'une des deux parties qui servent à éditer le code, et la classe cliquée s'affichera à l'endroit voulu. Nous pouvons effectuer cette opération sur n'importe quelles classes, fonctions ou variables. La partie centrale basse permet d'avoir des informations supplémentaires sur une fonction ou une classe ou bien même une variable, c'est l'objet de la prochaine partie.

Informations supplémentaires données par EiffelStudio

La partie centrale basse permet d'avoir des informations supplémentaires sur une fonction, une classe ou une variable. Il suffit de placer l'élément dont nous souhaitons avoir des renseignements supplémentaires dans la partie basse de l'environnement pour obtenir par exemple le graphe d'héritage, savoir qui appelle une fonction, qui sont les clients d'une classe etc (voir fi-

gures fig 4 et fig 5).

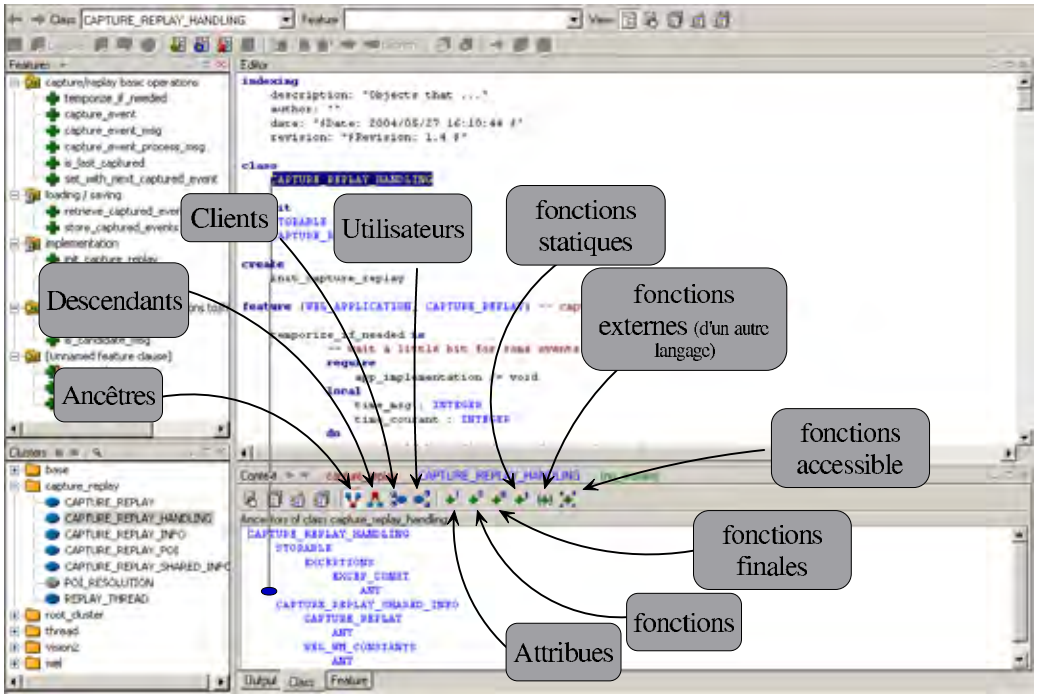

FIG. 4 – Les informations supplémentaires sur une classe

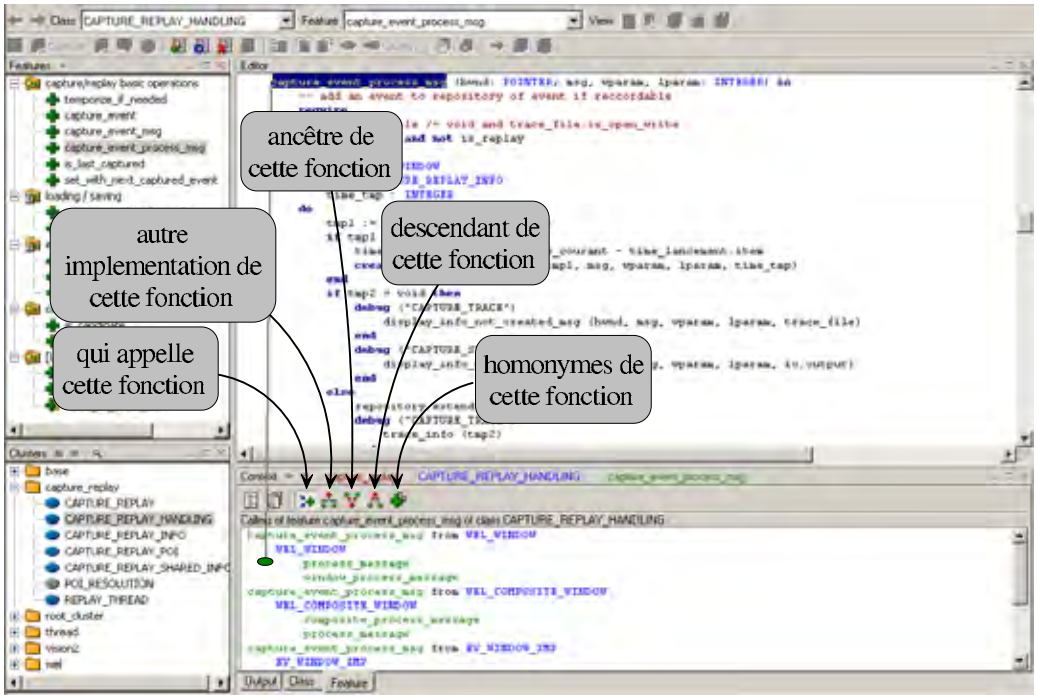

FIG. 5 – Les informations supplémentaires sur une fonction ou variable

Ces fonctionnalités ont donc permis de connaître la structure des classes *Eiffel*, savoir comment elles sont reliées entre-elles et de pouvoir retrouver par exemple les fonctions redéfinies.

Le débuguer de EiffelStudio

La dernière fonctionnalité de *EiffelStudio* que nous désirons souligner et le débuguer qui est intégré à l'environnement. Il permet à tout moment dans le code de mettre un point d'arrêt, et affiche toutes les variables qui peuvent être consultées à tout moment. Dans notre phase de compréhension des événements, avoir ce genre de débugguer facilite grandement l'étude.

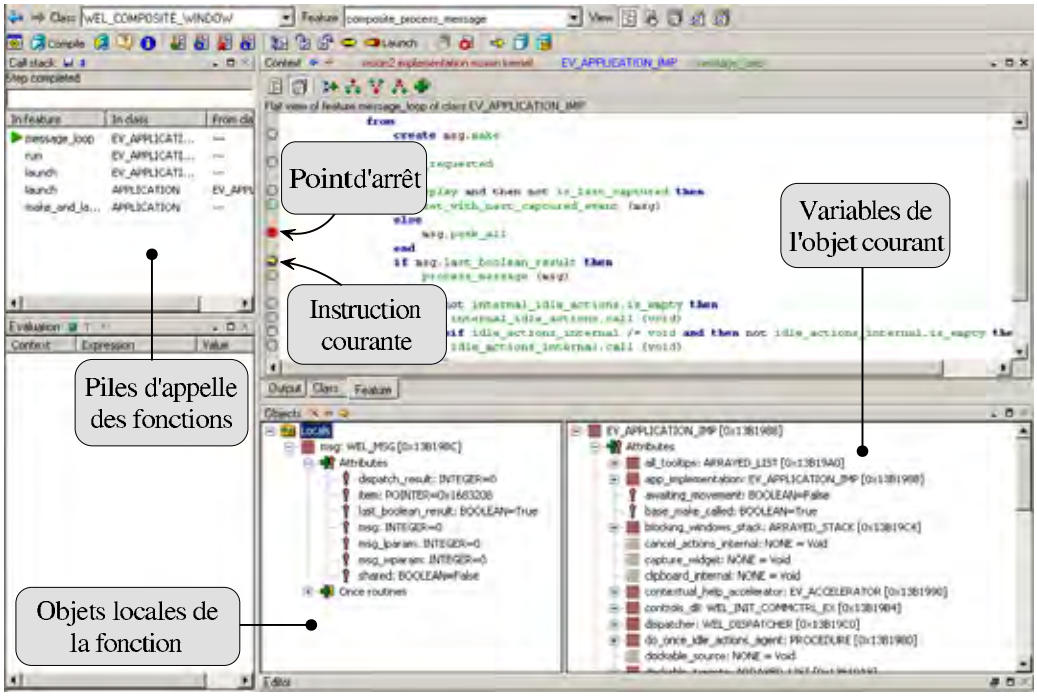

FIG. 6 – Le debugeur de EiffelStudio

Un petit regret tout de même sur le débuguer, c'est que parfois et notamment lorsque nous utilisons les *threads*, certaines variables sont dites non initialisées alors qu'elles contiennent effectivement des choses.

3.2.3 Intégration et compréhension des fichiers de Capture And Replay

Nous sommes repartis sur les fichiers *Eiffel* fourni par nos encadrants. Ces fichiers se décomposaient de la manière suivante :

- Un ensemble de classes *Capture And Replay.*
- Un ensemble de classes modifiées dans la bibliothèque graphique *Eiffel.*

Dans un premier temps, nous voulions compiler le projet pour savoir ce qui avait déjà été fait. Nous avons donc remplacé les classes modifiées de la bibliothèque graphique par les fichiers fournis par nos encadrants, et compilé. Ce que nous avions oublié, c'est que, entre le temps où nos encadrants ont réalisé ces classes et le temps où nous nous en servions, une nouvelle version de *EiffelStudio* était sortie. Le code des classes *Eiffel* modifiés avait donc changé entre temps et nous ne pouvions plus remplacer par simple copier coller. Nous avons perdu beaucoup de temps là-dessus, car nous n'avions aucune idée du code qui avait été rajouté dans les fichiers pour prendre en compte le *Capture And Replay* d'où un début de projet difficile. C'est Mr Lahire qui lors d'une visite à *Sophia Antipolis* nous a aidé dans cette première étape et grâce à lui nous avons pu compiler les classes.

Nous avons pu compiler et lancer l'application, mais à ce moment là, nous ne capturions encore aucuns événements, et ce n'est que lorsque nous avons compris un peu le principe de **message_loop** (que nous avons expliqué dans une partie précédente) que nous avons réussi à capturer les premiers événements.

Revenons tout de même sur les fichiers de *Capture And Replay* pour bien comprendre comment il a été mis en œuvre. Les classes permettent de capturer un message qui est stocké en mémoire et sauvegardé dans le fichier "*FileOfEvent*" à la fin de l'application. Si le fichier est présent, alors nous sommes dans le mode rejoue. Ce mode lit ce fichier, et simule les événements les uns après les autres. Si ce fichier est absent, alors nous sommes dans le mode de capture d'événements. Un événement est un message qui a pour cible un élément graphique. Cette cible est référencée par un pointeur dans le message. Une référence n'est valide que le temps de l'application, si nous fermons l'application et la relançons, il y a très peu de chance d'avoir la même référence. La solution mise en œuvre pas nos encadrants pour palier à ce problème est la suivante : Les éléments graphiques ont tous un père unique qui est *WEL_ WINDOW*. L'idée est donc, pour chaque création d'un élément graphique, de mettre dans le constructeur du *WEL_ WINDOW*, une fonction qui va mémoriser (dans une liste) les caractéristiques de l'élément graphique (sa taille, ses coordonnées etc). Après plusieurs créations d'éléments graphiques, nous avons donc accès à une liste de caractéristiques d'éléments

graphiques. Lorsque nous capturons un message, nous avons le pointeur sur l'élément graphique concerné par celui ci. Nous pouvons donc mémoriser les caractéristiques de l'élément graphique **et** le message concerné. Au moment du rejoue, nous recréons comme précédemment la liste de caractéristiques des éléments graphiques, et quand nous voulons rejouer un message, nous lisons les caractéristiques de l'élément graphique concerné par le message (mémorisé lors de la capture), et nous essayons de récupérer le pointeur qui correspond à ces caractéristiques dans la liste de caractéristiques des éléments graphiques. Ce petit mécanisme permet donc de retrouver le pointeur vers le bon élément graphique lors du rejoue.

Nous avons vu les points importants des fichiers qui nous ont été fournis, le reste ne demandant pas de grande explication pour être compris.

3.2.4 Temporisation

Une fois que nous avons capturé nos premiers événements et que nous avons voulu les rejouer, il n'y avait pas de temporisation entre deux messages rejoués. Ainsi une application, qui a été capturée pendant 1minute, pouvait être rejoué en quelques secondes. Si nous voulons faire un bon *Capture And Replay*, il fallait donc se pencher sur ce sujet pour rejouer les événements dans le même temps que lors de la capture.

Pour cela, nous avons ajouté aux informations du message le temps écoulé depuis le début du lancement de l'application. Ainsi, lors du rejoue, pour chaque message, nous avons l'indication du temps écoulé depuis le lancement, et nous pouvons en déduire à quel moment il faut rejouer le message.

Pour mettre en œuvre la temporisation, nous faisons appel à une fonction *C* qui récupère le temps système. Grâce à cette fonction, nous pouvons obtenir le temps au lancement de l'application, et le temps d'arrivée de chaque message pour en déduire le temps écoulé depuis le lancement de l'application.

Nous avons connu plusieurs difficultés à ce niveau là, puisque nous réalisions notre premier appel à une fonction *C* à partir de *Eiffel* et qu'il a engendré pas mal d'erreur, notamment au niveau des conversions de type entre *Eiffel* et *C*. Un autre problème aussi fut que la fonction *C* utilisée au départ avait un effet de bord (envoi d'un signal qui bloque l'application *Eiffel*). La fonction *C* a utilisé devait prendre en compte un temps très fin puisque la seconde n'était pas un bon indicateur, il a donc fallu trouver le bonne fonction *C* qui récupère le temps à la 100ème de seconde prés.

3.2.5 Thread

Lorsque nous avons capturé puis rejoué nos premiers messages, nous nous sommes aperçus que, pendant le rejoue, si l'application était masquée par une fenêtre ou un autre élément, l'application ne se rafraîchissait pas puisque lors de la capture il n'y a pas eu ce masquage.

Les messages sont rejoués dans la boucle de traitement des événements **loop_message** dont nous avons déjà parlé (voir **Message_loop** de *EV_APPLICATION_IMP*). Dans cette boucle, nous prenons le prochain message à rejouer, et nous le traitons. Pendant tout ce traitement, aucun autre message ne peut être rejoué et traité. Le fait de cliquer sur l'application pour la remettre en avant plan génère des messages mais qui ne seront jamais traités puisque dans la phase de rejoue, nous ne rejouons que les événements capturés.

Mais nous pourrions avoir envie d'avoir une boucle qui traite certains messages (dont le rafraîchissement de l'application par exemple) même pendant le rejoue et une autre boucle qui se contente de rejouer que les événements capturés. Nous aurions donc deux *thread* pour exécuter chacune de ces opérations, le premier **thread** vérifiant les événements produit par l'utilisateur pendant le rejoue, traitant si besoin le message, et le deuxième rejouant que les événements capturés. Ainsi, nous aurions peut être réglé des problèmes pendant le rejoue. (voir fig 7)

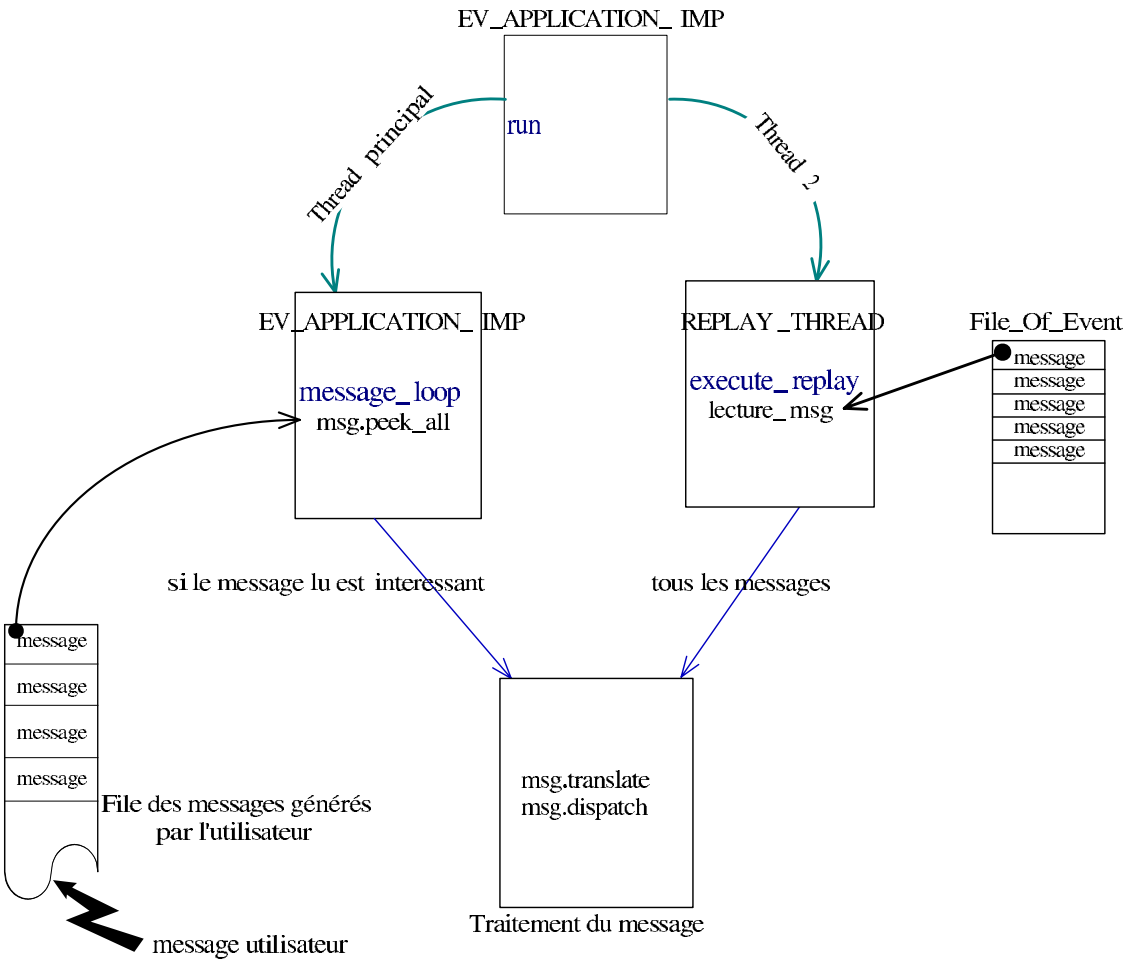

FIG. 7 – Utilité d'un deuxième thread

La mise en œuvre de cette fonctionnalité a été beaucoup plus difficile que prévu, nous avons besoin de partager des variables pour accéder aux informations nécessaires. Normalement, il n'y a pas de problème vu qu'un *thread* dispose du même espace mémoire qu'un autre *thread*, mais les variables à partager sont des variables spéciales (once routine). Ces variables, qui seraient comme des variables déclarées statiques en Java, sont par défaut indépendantes entre chaque *thread*. Nous, nous avions besoin d'avoir une valeur par processus et non pas une valeur par *thread*. En consultant la documentation, nous avons trouvé la solution théorique à ce problème, mais l'implémentation n'a pas semblé être aussi facile que la théorie. Nous avons laissé en suspend cette partie, car beaucoup d'autres choses devaient être faites pour peut être y revenir à la fin s'il nous restait du temps. Mais nous ne sommes pas revenu dessus à l'heure où nous écrivons ce rapport.

3.2.6 Script d'installation

Nous avons réalisé un petit script d'installation écrit en *C* qui va se charger d'installer les classes *Eiffel* modifiée pour prendre en compte le *Capture And Replay* ainsi que d'installer le module des classes de *Capture And Replay*. En effet, il n'est pas pratique de réaliser cette installation à la main, nous avons pu le constater par nous même la première fois et lorsque nous voulions montrer notre travail à nos encadrants.

Le script prend en paramètre un nom de fichier, celui-ci étant comme les fichiers de propriétés que nous pouvons trouver en *Java*. Il cherche à savoir où est le répertoire d'installation de *Eiffel* (**repInstallEiffel**), et dans quel répertoire (**repSrcEiffelBak**) il doit soit chercher (phase installation) ou soit copier (phase de sauvegarde) les fichiers à installer ou à sauvegarder. Ensuite, pour chaque fichier énuméré dans le fichier donné en entrée du script, il exécute la bonne opération de copie (soit de **repSrcEiffelBak** vers répertoire un sous répertoire de **repInstallEiffel** pour l'installation, soit de **repInstallEiffel** vers **repSrcEiffelBak** pour la sauvegarde).

Le script a parfaitement bien marché lors d'une démonstration avec notre encadrant qui a été satisfait de son utilisation. Pour avoir plus d'information sur ce Script consultez le guide d'installation qui explique plus en détail la syntaxe que doit avoir le fichier de propriétés et qui explique les différentes options du Script.

Il n'y a pas eu de problèmes majeurs à l'écriture de ce script, si ce n'est les traditionnels *segmentation fault* que nous pouvons obtenir à cause d'une mauvaise manipulation des chaînes en *C*. Jusqu'au jour d'aujourd'hui, le script a toujours bien fonctionné.

4 Bilan projet

Dans cette partie, nous allons revenir sur ce qui a été fait très rapidement, et surtout ce qui reste à faire pour que le projet marche mieux, et les idées que nous avons eu pour rendre le *Capture And Replay* plus attractif. Enfin, nous tirerons un bilan sur les choses que nous avons apprises ainsi que sur les choses qui nous ont manqué pour réaliser de fond en comble ce projet.

4.1 Avancement du projet

Dans cette partie nous allons exprimer en quoi nous avons fait avancer le projet et ce qu'il reste à faire pour avoir un *Capture And Replay* optimal.

Nous avons avancé dans le projet dans le sens où nous avons réussi à localiser tous les événements que nous pouvons retrouver sur un élément graphique, que nous avons pu comprendre comment ces événements sont produit et que cette partie ne sera plus à faire. Nous avons introduit la notion de temps pour que les événements soient rejoués dans le même temps que la capture et nous avons écrit un Script qui permet de pouvoir disposer du *Capture And Replay* de façon assez simple.

En revanche, la partie rejoue ne fonctionne pas tellement bien, nous arrivons à reproduire assez bien les touches au niveau du clavier. En fait, les messages relatifs aux touches de claviers ne comportent pas de pointeur dans leurs paramètres (**lparam** et **wparam**). Mais ce n'est pas le cas de tous les messages, il existe une grande diversité de messages (plus de 1024) et **lparam** ou **wparam** peuvent contenir des références à des structures ou objets très variés. Avec le temps imparti, nous n'avons pas eu le temps de nous pencher sur tous ces messages, pour trouver un système qui change les références en quelque chose de plus persistant. Du coup, la phase de rejoue a du mal à arriver jusqu'à la fin de la liste d'événements à rejouer.

Il faudra donc revenir sur ce point au niveau du rejoue et nous pensons que toutes les erreurs, pendant le rejoue, disparaîtront.

4.2 Idées pour améliorer le Capture And Replay

Tout au long du projet, nous avons eu des idées qui seraient intéressantes de réaliser, une fois que l'implémentation de rejoue fonctionnera correctement. Par exemple, nous arrivons très bien à capturer les événements de la souris lorsqu'elle se déplace. Mais lors du rejoue, la souris ne bouge pas, nous avons juste les messages qui disent que la souris a bougé à telle position. Notre idée est, donc, de réaliser un curseur virtuel qui à chaque fois que nous rejouons un message relatif au déplacement de la souris (facile à savoir) déplace le curseur virtuel. Nous pourrions donc visualiser à l'écran les déplacements de la souris, ce qui n'est pas encore le cas.

Une autre idée, serait de lancer une petite interface graphique en même temps que l'application de l'utilisateur pour la gestion du *Capture And Replay*. Ainsi, nous pourrions préciser à quel moment nous voulons capturer les événements, permettre de charger un fichier d'événements et de le rejouer à la volée etc.

Nous pensons que ces deux améliorations sont entièrement réalisables, les seules difficultés étant de connaître les classes *Eiffel* pour faire l'interface graphique ou pour dessiner un curseur à l'écran.

4.3 Enseignements retenus

Dans cette partie, nous allons tirer le bilan des choses que nous avons apprises pendant la réalisation de ce projet.

Tout d'abord, nous avons découvert un langage de programmation séduisant, qui permet de s'interfacer assez facilement avec d'autres langages de programmation et qui possède un mécanisme d'héritage très évolué. Ce langage couplé à *EiffelStudio* permet de tirer au maximum toute la puissance du langage.

Ensuite, l'étude des événements sous *Windows* nous a permis de comprendre comment une application graphique arrive à communiquer avec ses différents éléments graphiques pour garder l'application dans un état cohérent. Nous avons donc les idées plus claires sur une façon de mettre en œuvre les messages, comment les traiter etc ...Nous pensons qu'il est utile d'avoir vu ça, si par exemple un jour nous voulons créer nos propres classes qui gèrent une application graphique.

Au cours du projet, nous nous sommes penchés de très près sur les fonctions de l'API de *Windows*, il peut être valorisant pour notre carrière professionnelle d'avoir cette connaissance.

4.4 Les regrets

Nous allons maintenant exprimer nos regrets car nous regrettons un peu d'avoir été que deux pour réaliser ce projet. En effet, le travail d'étude et de recherche permet de travailler en équipe et de voir comment il est difficile de la gérer. Pour ce projet, après le départ de Jean-Charles Ledivelec, nous n'étions plus que deux et il est beaucoup plus facile de se gérer à deux, qu'à trois ou à quatre voir plus.

De plus, la présence d'une personne en plus aurait grandement aidée à aller plus loin dans ce projet. Ce dernier étant essentiellement un projet de recherche, où il faut trouver des endroits stratégiques et avoir la bonne inspiration, une personne de plus nous aurait apporté ces idées et son flair. Ainsi nous aurions sans doute pu aller plus loin dans la réalisation de ce projet.

Une des grandes difficultés du projet, c'est qu'il met en œuvre beaucoup de notions qui nous sont inconnus. Nous nous lançons dans un langage et une lecture des classes graphiques que nous ne connaissons pas. De plus, ces dernières sont très peu documentées. Il est donc très difficile de rentrer dans ce projet et la moindre manipulation au départ entraîne une multitude d'erreurs difficiles à corriger pour nous, qui sommes novices. Ce projet nous aura appris à nous débrouiller seul et à trouver l'information là où elle se trouve pour arriver à avancer.

4.5 Le mot de la fin

Le *Capture And Replay* est un sujet intéressant et qu'il aurait été intéressant de poursuivre car nous avons dû stopper nos efforts juste quand nous avions peut être trouvé la clé de la solution. Mais nous n'avions que peu de temps, un rapport à rédiger et des livrables, et nous avions pour objectifs de faire le plus possible mais bien quitte à laisser en suspend certaines choses. Nous pensons que tous les éléments sont dans les mains des futures personnes qui continueront ce projet pour le finaliser.

5 Annexes

Définitions

Capture : Récupérer les données affichées à l'écran et les sauvegarder (caractéristiques d'une fenêtre ou la position d'un clique de souris, ...).

Replay : Rejouer les actions qui se passent à l'écran et qui ont été mémorisées lors de la capture.

Evénement Message : Un événement ou un message représente tout ce qui peut être considéré comme une action de l'utilisateur (frappe au clavier, mouvements erratiques de la souris..). Ces événements ou messages peuvent être simulés par un programme .

Eiffel : Langage à objets, conçu en 1985-86 par Bertrand Meyer. C'est en fait tout un environnement de programmation à objets. Officiellement, ce n'est pas une surcouche (comme *C++*), mais bien au contraire un système idéal qui règle tous les problèmes et qui permet de tout programmer, depuis l'exemple idiot pour débutant jusqu'au méga-système critique de sécurité.

WEL : *Windows Eiffel Library.* La bibliothèque d'extension de *Windows Eiffel.* Elle a été conçue pour rendre la programmation de *Windows* plus facile, plus fiable, plus commode, et plus puissante. *WEL* est la couche *Windows*-spécifique d'*EiffelVision*, la bibliothèque portative de graphiques d'Eiffel.

API : *Application Programming Interface.* Interface de programmation d'applications, contenant un ensemble de fonctions courantes de bas niveau, bien documentées, permettant de programmer des applications de "Haut Niveau". On obtient ainsi des bibliothèques de routines, stockées par exemple dans des *DLL* ou des *NLM.*

msdn : *MicroSoft Developer Network.* Canal officiel de diffusion des "documentations techniques" - *SDK* et *DDK* - de *Microsoft.*
La *msdn* regroupe toute la documentation concernant *Windows*, notamment la description des fonctions *C* qui nous servirons pour capturer les événements au niveau de l'*API* de *Windows.*

Widget : Élément d'une interface graphique.
Exemples : bouton, menu, fenêtre, ascenseur. On fait de plus en plus la différence entre les simples "Contrôles", et les *widgets*, qui sont censés être un peu intelligent, et en tout cas avoir un code associé assez complexe.
En anglais, un "*widget*" est une sorte de "machin" et le terme est utilisé dans ce sens depuis le début du XXème siècle au moins. En informatique, on peut aussi considérer que le mot est une contraction de "*WInDow gadGET*".

Références

[1] Eiffel Software Inc : resources *Eiffel*. `http://archive.eiffel.com` , 2004

[2] Roland Trique : Jargon Français. `http://www.linux-france.org/prj/jargonf` , 2003

[3] Justin : Outils pour comprendre l'enregistrement et le rejeu d'applications sous X11. `http://www.tls.cena.fr/~jestin/Video/filmer.html` , 2000

[4] Microsoft Corporation : librairie *Windows*. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/winui/windowsuserinterface/userinput.asp` , 2004