Università degli Studi di Padova

YAST
Architettura .NET di Sportello per Transazioni Bancarie

Laurea Triennale

Laureando: Matteo Monego

Relatore: Prof. Giorgio Clemente

Dipartimento di Ingegneria dell’Informazione
Anno Accademico 2012-2013

Indice

1 Tecnologia COM (Component Object Model) ... 7
1.1 Terminologia COM ... 7
1.2 Class Factory ... 9
1.3 Globally Unique IDentifiers (GUIDs) ... 9
1.4 Interfacce e Classi COM ... 10
1.5 L'interfaccia IUnknown ... 11
1.6 Interface Definition Language (IDL) ... 12
1.7 Librerie dei Tipi (Type Libraries) ... 12
1.8 Interfaccia IDispatch e Binding ... 13
1.9 Ereditarietà di interfaccia ... 15
1.10 Aggregazione e Contenimento COM ... 15
1.11 Marshaling COM ... 17
1.12 COM e il registro di sistema ... 18
1.13 Compatibilità dei componenti ... 20
2 Il Framework .NET ... 21
2.1 Cos'è il Framework .NET ... 21
2.2 Codice managed (Gestito) ... 23
2.3 La Common Language Specification (CLS) ... 24
2.4 Il Common Type System (CTS) ... 24
2.5 Tipi di valore e tipi di riferimento ... 25
2.6 Stacking e Heaping nel Framework .NET ... 25
2.6.1 Lo Stack delle chiamate ... 26
2.6.2 Il Managed Heap ... 29
2.6.3 La frammentazione esterna ... 30
2.6.4 Allocazione di memoria nel Managed Heap ... 31
2.6.5 Deallocazione di memoria nel Managed Heap ... 32
2.6.6 Generazioni ... 34
2.6.7 Finalizzatori: pulizia implicita delle risorse ... 35
2.6.8 Il metodo Dispose(): pulizia esplicita delle risorse ... 39
2.6.9 La keyword Using ... 40
2.6.10 Tipi nello Stack e nell'Heap ... 40
2.7 Windows Communication Foundation (WCF) ... 42
2.7.1 Definizione del sottosistema WCF ... 43
2.7.2 Esempio di creazione di un servizio con WCF ... 44

3 Overview di YAST ... 47
3.1 Introduzione ... 47
3.2 Schema dell'architettura ... 48
4 Struttura di una transazione .NET ... 49
4.1 Il pattern Model-View-Controller ... 49
4.2 Gestione del flusso di navigazione ... 50
4.3 YNavigation ... 53
4.4 YManager ... 54
4.5 Nodi di navigazione NavMap ... 63
4.6 Nodi di navigazione NavService ... 64
4.7 Nodi di navigazione NavExcel ... 64
4.8 Nodi di navigazione NavNested ... 65
4.8.1 NavNested COM senza ritorno dati ... 66
4.8.2 NavNested COM con ritorno dati ... 67
4.8.3 NavNested .NET senza ritorno dati ... 69
4.8.4 NavNested .NET con ritorno dati ... 70
4.8.5 Chiamate Nested .NET da transazioni COM ... 72
5 Inter Process Communication (Pipes) ... 75
5.1 Named Pipes ... 75
5.2 PipeServer ... 76
5.3 PipeClient ... 80
5.4 Stack Tracing ... 81
6 WCF in YAST ... 85
6.1 Utilizzo di WCF in YAST ... 85
7 CoreClient ... 91
7.1 Lancio di transazioni .NET ... 91
7.2 Modalità di lancio ACTIVEX ... 95
7.3 Modalità di lancio DIRETTA ... 96
7.4 Modalità di lancio NESTED ... 96
7.4.1 Nested .NET → .NET ... 96
7.4.2 Nested VB6 → .NET ... 98
7.5 Terminazione di una transazione ... 103
8 CoreClientPlugin ... 107
8.1 Componenti ActiveX ... 107
9 Conclusioni ... 115

Introduzione

Il periodo di tirocinio svolto presso la Lynx S.p.A. mi ha visto coinvolto nella progettazione di una nuova infrastruttura applicativa denominata YAST (Yet Another Sportello Target), modellata per lo sviluppo di transazioni bancarie, commissionata da un noto gruppo bancario italiano. L'ambiente di Sportello ha utilizzato, fino ad oggi, un'architettura basata sulla tecnologia COM, le cui transazioni sono state sviluppate nell'ormai obsoleto Visual Basic 6.
Lo scopo è stato quindi quello di sostituire l'architettura precedente (e, con il tempo, anche tutte le transazioni), portandola in ambiente managed .NET. La scelta del Framework .NET come nuova piattaforma è stata una scelta quasi obbligatoria (e molto vantaggiosa), dal momento che può essere considerata, attualmente, la tecnologia più importante per lo sviluppo in ambiente Windows.
La progettazione dell'architettura ha compreso un insieme di attività volte allo scopo di individuare la soluzione implementativa migliore, quella che consente di soddisfare gli obiettivi funzionali (e quelli non funzionali) attesi dal committente e dall'utilizzatore finale. Il risultato è stata la definizione dell'organizzazione strutturale del sistema stesso, che comprende i suoi componenti software, la loro interfaccia e le relazioni fra le parti. Pensare all'architettura di un sistema software semplicemente come alla sua struttura è peraltro estremamente riduttivo. Un'architettura è molto di più. Essa comprende infatti anche le modalità con cui le singole parti che la compongono si integrano e interagiscono a formare un tutt'uno, considera gli aspetti legati all'interoperabilità con i sistemi circostanti (questo, come si vedrà, sarà fondamentale nel caso di YAST), comprende caratteristiche non legate ai casi di utilizzo, ma orientate a favorire l'evoluzione nel tempo del sistema a fronte dei suoi cambiamenti strutturali e in relazione all'ambiente in cui esso è inserito. L'architettura è quindi una rappresentazione che permette all'architetto di analizzare l'efficacia del progetto per rispondere ai requisiti stabiliti e di considerare e valutare le alternative strutturali in una fase in cui i cambiamenti abbiano ancora un impatto relativo al risultato finale.
Per quanto riguarda il linguaggio di programmazione con cui si è deciso di operare la scelta è ricaduta naturalmente su C#, progettato specificamente per lavorare con il Framework .NET. Di fondamentale importanza è il fatto che C# sia un linguaggio nativamente orientato agli oggetti, con tutti i vantaggi che tale paradigma comporta, a differenza di Visual Basic 6, il quale non può essere considerato un linguaggio orientato agli oggetti a tutti gli effetti, non potendo, ad esempio, implementare l'ereditarietà ed altri concetti chiave di tale paradigma.

Il Framework .NET mette inoltre a disposizione una nuova tecnologia di comunicazione, il sottosistema WCF, e la libreria per la creazione di interfacce grafiche WPF.
Il processo di rinnovamento dell'ambiente di Sportello non avrebbe mai potuto essere istantaneo, vista la quantità di componenti che ne fanno parte. La gradualità richiesta per il passaggio ha introdotto il bisogno di retrocompatibilità nei confronti delle transazioni COM già esistenti, con le conseguenti difficoltà tecniche che l'interoperabilità tra processi così diversi comporta.
Altre problematiche che è stato necessario affrontare riguardano la gestione di transazioni con comportamenti molto diversi tra loro, che hanno portato in certi casi a dei problemi di ritenzione di memoria.
Al fine di chiarire meglio il contesto in cui YAST è stato sviluppato (ed è tutt'ora in evoluzione), i primi capitoli mostrano una panoramica delle principali tecnologie utilizzate, compresa una breve introduzione al Framework .NET.

1 Tecnologia COM (Component Object Model)

Questo capitolo presenta un'introduzione alla tecnologia COM, della quale è stato fatto largo utilizzo per lo sviluppo della precedente architettura di Sportello (in particolare di controlli ActiveX). COM permette la comunicazione tra processi e la creazione dinamica di oggetti con qualunque linguaggio di programmazione che lo supporti e, benché si preveda una sua progressiva sostituzione (se non altro parziale) da parte del Framework .NET, ad oggi la sua importanza è ancora fondamentale.
Prima di discorrere di COM nello specifico, vale la pena di considerare l'ampio utilizzo che viene fatto dei componenti software in maniera più generale.
Ci sono molti fattori che portano uno sviluppatore a far uso di un componente, ma il motivo principale è che lo sviluppo software è spesso un'attività costosa sia in termini di tempo che di denaro. In un mondo ideale, sarebbe possibile scrivere del codice una volta e successivamente riutilizzarlo all'infinito, attraverso vari tool di sviluppo, anche in circostanze che lo sviluppatore non aveva previsto.
I primi tentativi di produrre codice riutilizzabile si basavano sulla creazione delle Class Libraries, generalmente sviluppate in C++. Questi primi tentativi soffrivano di gravi limitazioni, come difficoltà di condivisione di parti del sistema, problemi di persistenza e di aggiornamento dei componenti senza la ricompilazione, la mancanza di modelli di linguaggio adatti allo scopo. Un componente software è un'unità binaria di codice riutilizzabile. Negli anni sono emersi una grande quantità di standard per lo sviluppo e la condivisione di componenti, ma, per le applicazioni desktop, la tecnologia COM è quella più matura e con le migliori prestazioni. Per capire il funzionamento di COM, introdotta nelle sue prime versioni da Microsoft nel 1993, è importante sapere che si tratta di un protocollo (o standard), non di un linguaggio orientato agli oggetti. A tutti gli effetti, COM non è solamente una tecnologia, ma una metodologia di sviluppo software: COM definisce infatti il modello di programmazione interface-based, dove gli oggetti incapsulano i metodi e i dati che caratterizzano ogni oggetto istanziato attraverso un'interfaccia ben definita. I componenti COM possono inoltre essere scritti in linguaggi diversi: si parla quindi di standard binario, cioè uno standard che si applica dopo che il programma è stato tradotto in codice macchina.

1.1 Terminologia COM

Molti testi utilizzano i termini componente, oggetto, client e server in maniera diversa, altri ancora fanno riferimento allo stesso concetto quando utilizzano una di queste parole. E' pertanto necessario fare chiarezza sulla terminologia adottata.
COM è un'architettura client/server: il server mette a disposizione, attraverso un oggetto, le funzionalità che il client utilizzerà. Un oggetto può, nello stesso tempo, essere parte sia di un server che di un'applicazione client.

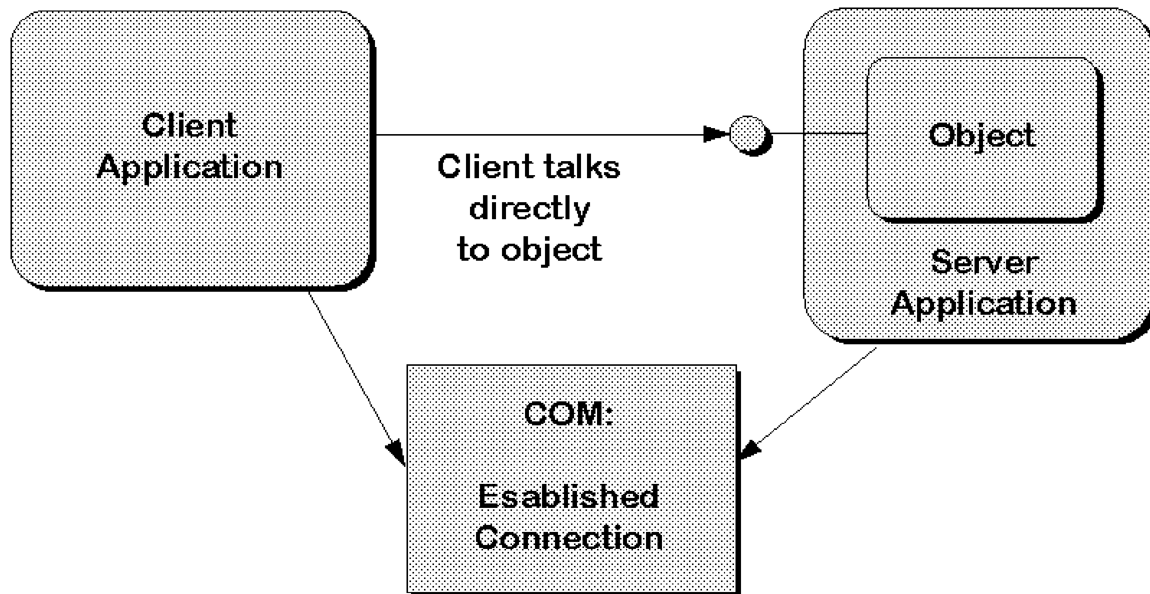

Fig. 1.1 – Architettura Client/Server di COM

Gli oggetti sono istanze di classi COM che rendono il servizio disponibile al client. E' lecito quindi parlare di comunicazione tra un client ed un oggetto, oltre che tra client e server. Questo oggetti sono detti *oggetti COM* o *oggetti componente*. In questo capitolo, faremo riferimento a queste entità con il termine oggetti. Il client e i suoi server possono far parte dello stesso processo oppure essere parte di processi diversi. In base allo spazio di indirizzi utilizzato, si possono distinguere due modalità di utilizzo degli oggetti:

- COM In-process server: questi server sono distribuiti come Dynamic Link Libraries (DLL), e, come tutte le DLL, sono caricati nello spazio degli indirizzi del client la prima volta che il client accede al server;
- COM Out-of-process server: sono server distribuiti come file eseguibili (EXE) e vengono eseguiti nel loro proprio spazio degli indirizzi.

Quando vengono creati oggetti COM, lo sviluppatore deve essere a conoscenza del tipo di server all'interno del quale l'oggetto risiederà, ma se l'oggetto è stato implementato correttamente, la modalità di rilascio non ha controindicazioni sull'uso di quest'ultimo da parte del client.
Chiaramente, ci sono vantaggi e svantaggi nell'utilizzare una modalità di rilascio piuttosto che l'altra: il processo di caricamento di una DLL in memoria è molto veloce, come anche chiamarne un metodo. Gli eseguibili, d'altra parte, sono una soluzione molto più robusta (a seguito di errori imprevisti del server, il client riesce ad evitare il crash) e più sicura, dal momento che in questo caso il server ha un proprio contesto di sicurezza.
In un sistema COM, il client è completamente isolato dall'oggetto che fornisce le funzionalità. Tutto quello che il client necessita di sapere è che la funzionalità sia disponibile: si dice pertanto che *COM agisce da contratto tra client e oggetto*. Se l'oggetto rompe il contratto, il comportamento del sistema non sarà determinabile a priori.

1.2 Class Factory

Ogni server contiene un oggetto chiamato *Class Factory*. Lo scopo del class factory, quando richiamato dal COM runtime, è quello di istanziare oggetti di una particolare classe. Per ogni classe COM esiste un class factory: quando un client richiede un oggetto da un server, il class factory della classe che definisce l'oggetto richiesto crea un nuovo oggetto di tale classe e ne dispone l'utilizzo per il client.

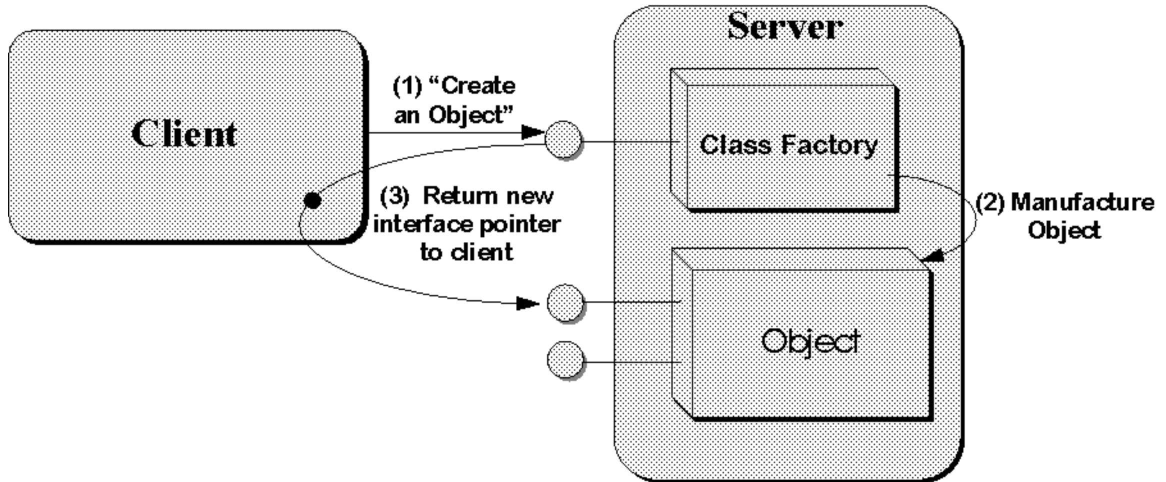

Fig. 1.2 – Class Factory

Un server non è altro che un file binario contenente il codice richiesto da una o più classi COM. Questo includerà naturalmente sia il codice necessario a COM per istanziare gli oggetti in memoria, sia il codice necessario all'esecuzione dei metodi degli oggetti contenuti nel server. Nonostante questa sia la modalità di funzionamento standard dei server, ne esistono altre implementazioni: in una di queste i class factory possono creare un'istanza dell'oggetto una sola volta, quindi passare ai vari client il riferimento all'oggetto appena creato. Questo tipo di implementazione è detta singleton, dal momento che esiste un'unica istanza dell'oggetto per ogni processo.

1.3 Globally Unique IDentifiers (GUIDs)

Un sistema che potenzialmente espone migliaia di interfacce, classi e server, ognuno dei quali deve avere la possibilità di essere referenziato in fase di runtime necessita che ogni elemento che lo compone possa essere individuato in maniera univoca. Chiaramente, l'utilizzo di identificatori espressi in una forma leggibile ad un essere umano porterebbe a potenziali conflitti: per questo motivo COM utilizza dei Globally Unique Identifiers (GUIDs), numeri a 128 bit dei quali è garantita (su una base probabilistica) l'univocità in tutto il mondo. E' possibile generare 10 milioni di GUIDs al secondo fino all'anno 5770, ed è lecito aspettarsi che ognuno di questi sia diverso dagli altri. Un esempio di GUID è il seguente:

{E6BDAA76-4D35-11D0-98BE-00805F7CED21}

1.4 Interfacce e Classi COM

Sviluppare con COM significa sviluppare utilizzando delle interfacce. Tutte le comunicazioni tra gli oggetti sono effettuate attraverso le loro interfacce che, nel mondo COM, sono *astratte*: non prevedono quindi implementazione. Il codice associato ad un'interfaccia deriva infatti dall'implementazione di una classe. La modalità con cui un'interfaccia è implementata differisce da oggetto ad oggetto. Un oggetto *eredita il tipo dell'interfaccia*, non la sua implementazione: le sue funzionalità sono modellate astrattamente con le interfacce e implementate attraverso l'implementazione di una classe. E' quindi lecito affermare che le interfacce definiscono *cosa* un oggetto può fare, mentre le classi definiscono *come* può essere fatto. Le classi COM (coclassi) forniscono il codice associato ad una o più interfacce, incapsulando così tutte le funzionalità nella classe. Due classi possono avere la stessa interfaccia, ma possono implementarla in maniera diversa; si tratta del classico polimorfismo del paradigma object oriented. COM non supporta il concetto di ereditarietà multipla, ma si tratta di una limitazione trascurabile, dal momento che ogni singola classe può implementare più interfacce.

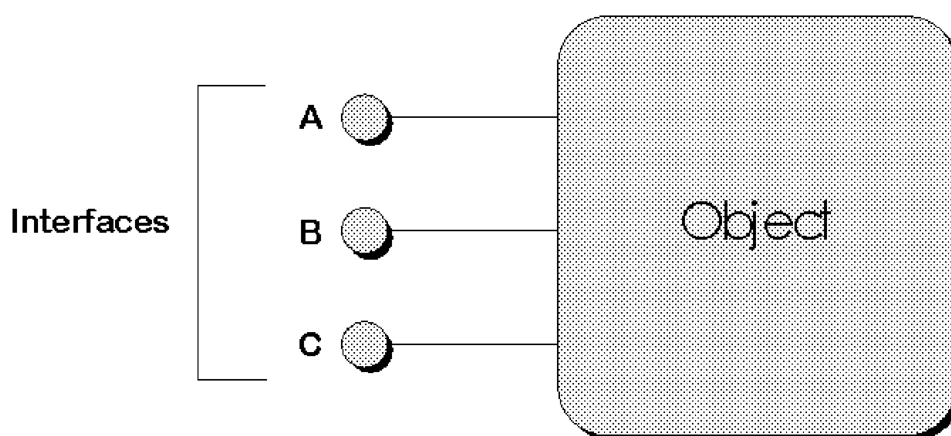

Fig. 1.3 – Interfacce COM

Una coclasse è quindi una classe della quale COM può creare un'istanza e successivamente passare il risultante oggetto al client affinché ne possa utilizzare i servizi, definiti dalle interfacce di tale classe. Si tratta quindi di un'implementazione concreta di una o più interfacce, ed è strutturata in maniera molto simile alle classi in altri linguaggi di programmazione Object Oriented. A differenza di questi, infatti, COM prevede una netta separazione dell'interfaccia dall'implementazione: non è possibile accedere agli oggetti in maniera diretta, ma solo attraverso le interfacce.
Le interfacce sono a tutti gli effetti il mezzo attraverso il quale gli oggetti COM comunicano tra loro. Quando si lavora con oggetti COM, lo sviluppatore non utilizza mai l'oggetto COM in maniera diretta, ma ottiene prima l'accesso alle funzionalità dell'oggetto attraverso una delle sue interfacce: le funzioni virtuali sono chiamate dal client e implementate dal server. Il client di un oggetto mantiene un puntatore all'interfaccia di quest'ultimo; spesso ci si riferisce a tali puntatori come a *puntatori opachi*, dal momento che il client non può venire a conoscenza dei dettagli implementativi di un oggetto attraverso di essi. Il concetto di interfaccia è fondamentale in COM. Le specifiche COM (Microsoft, 1995) ne enfatizzano i seguenti quattro punti:

- *Un'interfaccia non è una classe*. Un'interfaccia non può essere istanziata, dal momento che non presenta implementazione;

- *Un'interfaccia non è un oggetto*. Un'interfaccia è un gruppo di funzioni relazionate tra loro ed è lo standard binario attraverso il quale client e oggetti comunicano tra loro;
- *Le interfacce sono fortemente tipizzate*. Ogni interfaccia possiede un identificatore di interfaccia, eliminando così la possibilità di collisioni tra loro;
- *Le interfacce sono immutabili*. Le interfacce non sono mai versionate. Una volta definita e pubblicata, un'interfaccia non può essere cambiata.

L'ultimo dei quattro punti sopra elencati è di importanza fondamentale: una volta che un'interfaccia è stata pubblicata non è possibile cambiarne la firma. E' possibile cambiare in ogni momento l'implementazione dei metodi, sia questo un intervento riguardante un piccolo bug o la completa revisione del codice del metodo. I client dell'interfaccia non si accorgono del cambiamento nell'implementazione delle funzionalità, perché l'interfaccia è rimasta immutata: questo significa che quando delle modifiche vengono rilasciate sotto forma di DLL o di eseguibili, i client non necessitano di ricompilazione per far uso delle nuove funzionalità. Se, d'altra parte, la firma dell'interfaccia necessita di una modifica, *deve essere creata una nuova interfaccia che esponga la firma delle nuove funzionalità*. Le interfacce più vecchie o deprecate non sono rimosse dalla classe, per assicurarsi che tutte le applicazioni client possano continuare a comunicare con il server appena modificato. I nuovo client avranno la possibilità di scegliere se utilizzare le vecchie o le nuove interfacce.

1.5 L'interfaccia IUnknown

Tutte le interfacce COM derivano dall'interfaccia *IUnknown*, e tutti gli oggetti COM devono implementarla. Questa interfaccia svolge due funzioni importanti:

- controlla il ciclo di vita dell'oggetto;
- fornisce, a runtime, supporto sui tipi.

Il ciclo di vita dell'oggetto è controllato da due metodi, *AddRef* e *Release*, più un contatore di riferimenti interno. Ogni volta che un oggetto è creato o duplicato, viene invocato il metodo *AddRef* e incrementato il contatore di riferimenti; successivamente, quando il client non necessita più un riferimento all'oggetto, viene invocato il metodo *Release*, che decrementa il contatore dei riferimenti. Se questo contatore raggiunge lo zero, l'oggetto distrugge sé stesso. I client utilizzano l'interfaccia *IUnknown* anche per ottenere altre interfacce su un oggetto: quando il client vuole richiedere un'altra interfaccia sull'oggetto, chiama semplicemente il metodo *QueryInterface*, il quale fornisce l'interfaccia e chiama a sua volta *AddRef*. Di fatto, è responsabilità del metodo COM che ritorna un'interfaccia incrementare il contatore dei riferimenti con *AddRef*: il client chiama il metodo *AddRef* esplicitamente solo quando un'interfaccia è da duplicare.
E' possibile richiedere all'oggetto, attraverso una chiamata a *QueryInterface*, ogni altra interfaccia sull'oggetto. Quando viene richiesta una particolare interfaccia, il metodo *QueryInterface* può restituire un blocco di memoria già assegnato, oppure può allocare un nuovo blocco di memoria e ritornare quello: l'unico caso in cui viene ritornato lo stesso blocco di memoria è quando è la stessa interfaccia *IUnknown* ad essere richiesta.
Inoltre, quando si comparano due puntatori ad interfacce per controllare se puntano allo stesso oggetto, per entrambi deve essere richiesta l'interfaccia *IUnknown* e il confronto

deve essere effettuato sui puntatori a quest'ultima interfaccia. Per questo motivo, si dice che *l'interfaccia IUnknown definisce l'identità di un oggetto COM*.
Quando si sviluppa in Visual Basic, è buona pratica chiamare il metodo *Release* esplicitamente, assegnando ad un'interfaccia il valore *Nothing* per forzare il rilascio delle risorse che sta utilizzando. In ogni caso, il metodo *Release* verrebbe chiamato automaticamente quando l'oggetto non è più utilizzato, cioè nel momento in cui quest'ultimo esce dal proprio ambito (*scope*). Se si utilizzano variabili globali, è invece necessario richiamare il metodo *Release*. Inoltre, in Visual Basic, il sistema si occupa di eseguire tutte le operazioni riguardanti il conteggio dei riferimenti, agevolando così lo sviluppatore e rendendo l'uso degli oggetti COM molto semplice.
Al contrario, sviluppando in C++, è necessario incrementare e decrementare il contatore dei riferimenti per controllare correttamente il ciclo di vita dell'oggetto.

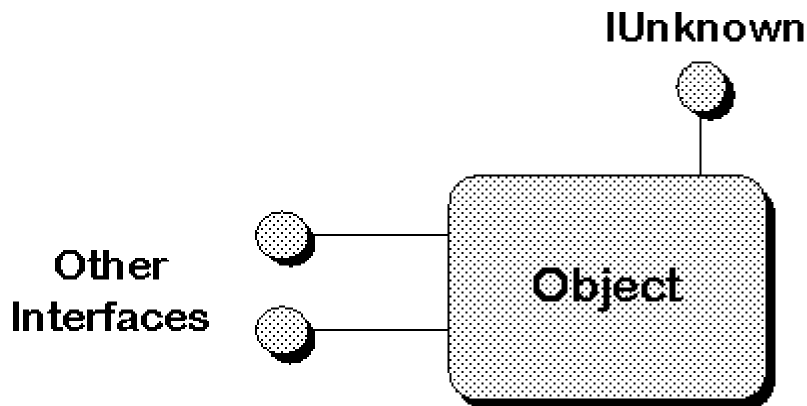

Fig. 1.4 – Interfaccia IUnknown

1.6 Interface Definition Language (IDL)

Il Microsoft Interface Definition Language (MIDL) è utilizzato per descrivere gli oggetti COM, incluse le loro interfacce. Il MIDL è un'estensione dell'Interface Definition Language (IDL) definito dal DCE (Distributed Computer Environment), dove era utilizzato per la definizione di chiamate a procedure remote tra client e server. Le estensioni MIDL riguardano l'ODL (Object Definition Language). Attualmente, quando si parla di IDL, si fa a tutti gli effetti riferimento al linguaggio MIDL.
Il linguaggio IDL definisce le interfacce pubbliche degli oggetti: quando compilato, l'IDL crea una libreria dei tipi (*Type Library*).

1.7 Librerie dei Tipi (Type Libraries)

Si può pensare ad una *Type Library* come ad una versione binaria di un file IDL. Si tratta di un elemento fondamentale di ogni componente COM, che contiene una descrizione di tutte le coclassi, interfacce, metodi e dei tipi contenuti in un server COM. Le librerie dei tipi hanno estensione .tlb, ma possono anche essere incorporate in altri file, come ad esempio nelle *Object Libraries* (.olb) o anche nelle DLL. In genere, quando si vogliono dichiarare variabili o utilizzare tipi definiti in una libreria dei tipi, è necessario un riferimento a quest'ultima: in fase di compilazione verrà utilizzata per recuperare informazioni sui tipi, allo scopo di permettere l'*early binding*.

Il compilatore Visual Basic automatizza il processo di creazione e di registrazione delle *Type Libraries* e, a meno che non sia specificato diversamente, incorpora la type library nell'eseguibile o nella DLL; in C++, invece, queste sono compilate su un file separato (.tlb) durante la compilazione. L'utility Microsoft OLEView può essere utilizzata per visualizzare ed ispezionare il codice IDL di una *Type Library*, anche se incorporata in una DLL, in un eseguibile, in un file .olb o in un controllo ActiveX.
Il codice IDL di una libreria dei tipi è composto fondamentalmente di due parti: una sezione compresa tra parentesi quadre contenente degli attributi, dove è anche definito il GUID, la versione e l'helpstring:

```
// Generated .IDL file (by the OLE/COM Object Viewer)
//
// typelib filename: 1

[
  uuid(53CED51D-432B-45B2-A3E0-0CE2C24235D4),
  version(1.0),
  helpstring("Messenger Private Type Library")
]
```

Questa sezione è direttamente seguita dall'entità alla quale gli attributi definiti devono essere applicati: inizia con il nome dell'entità seguita da una sezione compresa tra parentesi graffe che definisce l'entità in dettaglio:

```
library MessengerPrivate
{
  // TLib : OLE Automation : {00020430-0000-0000-C000-000000000046}
  importlib("stdole2.tlb");
  // Forward declare all types defined in this typelib
  interface IMsgrSessionManager;
  interface IMsgrSession;
  interface IMsgrLock;
  interface IMessengerPrivate;
  dispinterface DMessengerPrivateEvents;
  dispinterface DMsgrSessionManagerEvents;
  dispinterface DMsgrSessionEvents;
  [
    odl,
    uuid(305D86C6-6896-4099-91F5-CB7BA7733563),
    helpstring("Messenger Session Manager Interface"),
    dual,
    oleautomation
  ]

  interface IMsgrSessionManager : IDispatch {
...
```

1.8 Interfaccia IDispatch e Binding

IDispatch è una delle interfacce standard esposte dagli oggetti COM, con lo scopo di esporre il protocollo Automation. Le applicazioni che mettono a disposizione oggetti che utilizzano Automation sono dette ActiveX o Automation Server: Microsoft Excel è un Automation Server che espone degli oggetti che possono essere utilizzati per la creazione di nuove applicazioni. Dal momento che tutti gli oggetti Automation sono oggetti COM, devono implementare l'interfaccia *IUnknown*; di fatto, l'interfaccia *IDispatch* eredita da quest'ultima i metodi *AddRef*, *Release* e *QueryInterface*. La lista completa dei metodi esposti da IDispatch comprende inoltre:

- GetTypeInfoCount;
- GetTypeInfo;
- GetIDsOfNames;
- Invoke.

IDispatch fornisce il mezzo con il quale un client può capire quali proprietà e quali metodi sono supportati da un oggetto a runtime ed eventualmente invocarli.
Il protocollo Automation funziona in questo modo: dopo che un client ha ricevuto un riferimento all'interfaccia *IDispatch*, può chiedere ad un oggetto se supporta un determinato metodo chiamando *GetIDsOfNames* e passando come parametro alla funzione una stringa contenente il nome del metodo. Se l'oggetto non supporta il metodo richiesto, la chiamata fallisce. Se, al contrario, il metodo è supportato, *GetIDsOfNames* restituisce un identificatore per il metodo richiesto, chiamato *DISPID*. Un DISPID non è altro che un intero che identifica un determinato metodo dell'oggetto. Un client che ha ottenuto un DISPID valido può successivamente eseguire la chiamata ad un metodo chiamando il metodo *Invoke*.
Binding è il termine con cui si fa riferimento al processo di ricerca di una funzione dato un puntatore ad un oggetto. COM supporta tre tipologie di binding:

- Late Binding: il metodo chiamato su un oggetto viene ricercato attraverso il suo DISPID solo a runtime, è pertanto necessario chiamare prima *GetIDsOfNames* e successivamente *Invoke*. Le chiamate da parte del client a metodi non implementati dall'oggetto falliranno durante l'esecuzione, non avendo il compilatore alcun modo di sapere in anticipo se il metodo esiste effettivamente. Non è necessaria una Type Library. L'esempio che segue mostra del codice Visual Basic che implementa il late binding:

```
Dim obj As Object
Set obj = New Class1
Call obj.someMethod
```

Il tipo dell'oggetto `obj` non è conosciuto fino a quando il comando `Set` non viene eseguito a runtime.

- Early Binding: il binding viene effettuato durante la compilazione. Il client può così effettuare le chiamate ai metodi direttamente dall'oggetto. L'early binding è possibile grazie alla presenza nel programma compilato di una *vTable* (tabella dei metodi virtuali) contenente gli indirizzi dei metodi dell'oggetto. E' necessario l'utilizzo di una Type Library. Nell'esempio che segue viene mostrato come implementare l'early binding in Visual Basic:

```
Dim obj As Class1
Set obj = New Class1
Call obj.someMethod
```

- ID Binding: il DISPID di ogni proprietà o metodo dell'oggetto è fissato ed è parte della descrizione del tipo dell'oggetto. Se l'oggetto è descritto da una libreria dei tipi, l'Automation controller può leggere i DISPID da questa durante la compilazione, evitando così la chiamata al metodo *GetIDsOfNames*. Necessitando della sola chiamata al metodo *Invoke*, il tempo di esecuzione del metodo richiesto è circa la metà di quello richiesto dal late binding.

L'interfaccia *IDispatch* supporta il late binding e l'ID binding. L'early binding (a cui ci si riferisce spesso con il termine *vTable binding*) invece non utilizza l'interfaccia IDispatch, ma una type library fornisce tutte le informazioni richieste in fase di compilazione, così da permettere al client di conoscere il layout dell'oggetto. Questo è senza dubbio il metodo più efficiente per la chiamata dei metodi di un oggetto COM, senza contare che permette il controllo dei tipi durante la compilazione.

1.9 Ereditarietà di interfaccia

Un'interfaccia consiste in un gruppo di metodi astratti e di proprietà. Se un'interfaccia eredita da un'altra interfaccia, allora tutti i metodi e le proprietà dell'interfaccia base sono ereditati dall'oggetto che eredita.
E' necessario distinguere l'ereditarietà di implementazione, utilizzata molto spesso in linguaggi di programmazione ad oggetti come C++ o SmallTalk, dall'ereditarietà di interfaccia: nell'ereditarietà di implementazione un oggetto eredita *del codice*, mentre nell'ereditarietà di interfaccia è *la definizione dei metodi* che viene ereditata.
La coclasse che implementa le interfacce deve provvedere anche all'implementazione di tutte le interfacce ereditate. Lo standard COM prevede un meccanismo di ereditarietà singola: ogni interfaccia può ereditare i metodi di una ed una sola interfaccia preesistente.

1.10 Aggregazione e Contenimento COM

E' prassi comune, per oggetti COM complessi, di far uso di altri oggetti COM al loro interno: si parla in questo caso di *contenimento* di un oggetto COM, in cui l'oggetto contenente si avvale delle interfacce dell'oggetto contenuto senza renderle disponibili al di fuori di esso. Il contenimento è la forma più semplice di riutilizzo di codice binario. L'oggetto contenuto (interno) non è a conoscenza di essere contenuto in un altro oggetto (esterno). L'oggetto esterno deve implementare tutte le interfacce supportate dall'oggetto contenuto: quando sono invocati dei metodi dell'oggetto esterno, questo passa semplicemente la richiesta al metodo corrispondente dell'oggetto interno. Per l'aggiunta di nuove funzionalità all'oggetto esterno, è possibile procedere in due modi:

- aggiungere la funzionalità direttamente all'oggetto esterno, senza passare per l'oggetto interno;
- implementare una nuova interfaccia e passare ancora le chiamate all'oggetto interno.

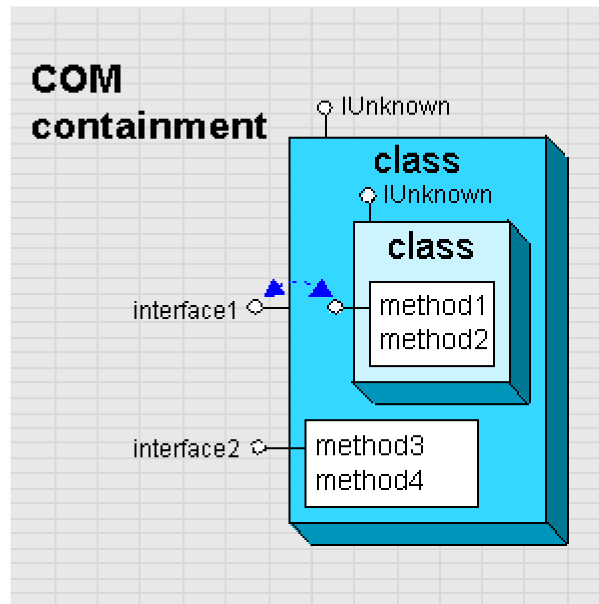

Fig. 1.5 - Contenimento COM

Si parla invece di *aggregazione* COM quando un oggetto esterno controlla quali interfacce di un oggetto interno esporre. L'oggetto interno è a conoscenza di essere stato aggregato in un altro oggetto e passa ogni chiamata a *QueryInterface* all'oggetto esterno. Per i client di oggetti che usano l'aggregazione, non c'è modo di sapere quali interfacce siano implementate dall'oggetto esterno e quali siano implementate dall'oggetto interno.

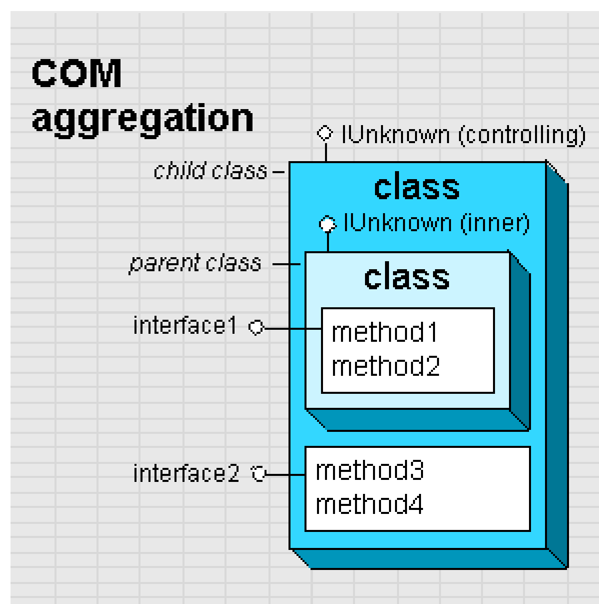

Fig. 1.6 – Aggregazione COM

1.11 Marshaling COM

COM è stato progettato per permettere ai client di comunicare in modo trasparente con gli oggetti, a prescindere dal luogo in cui questi ultimi sono eseguiti, nello stesso processo, sulla stessa macchina o su macchine diverse.
Dal punto di vista di un client, si accede agli oggetti attraverso dei puntatori ad interfacce. Naturalmente, un puntatore è contestualizzato in un singolo processo. Questo significa che, se l'oggetto è in-process, la chiamata lo raggiunge direttamente, senza alcun intervento da parte dell'infrastruttura del sistema. Se, al contrario, l'oggetto è out-of-process, la chiamata raggiunge prima l'oggetto *proxy* (messo in genere a disposizione dall'infrastruttura COM). Il proxy non fa altro che creare un pacchetto contenente i parametri della chiamata (inclusi i puntatori alle interfacce) e genera successivamente un'appropriata chiamata di procedura remota (RPC – *Remote Procedure Call*) verso l'altro processo (o l'altra macchina) dove è situata l'implementazione dell'oggetto. Questo processo di "impacchettamento" dei dati e dei puntatori ad interfacce è detto *Marshaling*. Dal punto di vista del server, tutte le chiamate alle funzioni di interfaccia di un oggetto sono effettuate attraverso un puntatore a quella interfaccia. Anche qui, un puntatore è valido solo se interno al contesto di un singolo processo.
Se l'oggetto è in-process, il chiamante è il client di sé stesso. Nel caso in cui l'oggetto sia out-of-process, il chiamante è a tutti gli effetti un'oggetto detto *stub*, gestito generalmente dall'infrastruttura COM. Lo stub riceve la chiamata di procedura remota dal proxy ed estrae i parametri passati (*unmarshaling*); successivamente chiama l'interfaccia appropriata sull'oggetto server.
COM mette a disposizione un'implementazione del marshaling chiamata *standard marshaling*. Questa implementazione è ottima per la maggior parte degli oggetti e rende il processo del tutto trasparente a client e server. Parametri differenti sono passati via marshaling in modi diversi: il passaggio di un intero è relativamente semplice, poiché viene semplicemente copiato nel buffer del messaggio; il passaggio di un array è leggermente più complicato, dal momento che gli elementi devono essere ordinati. Esistono funzioni per gestire il marshaling di qualsiasi tipo di dato.
Nello standard marshaling, i proxy e gli stub sono risorse disponibili a livello di sistema operativo, e comunicano tra loro attraverso un protocollo standard.
Il diagramma in figura 1.7 mostra il flusso della comunicazione tra un client ed un server appartenenti a processi diversi. La chiamata al metodo del client passa attraverso il proxy per continuare nel canale, che è parte della libreria COM.
Il canale invia il buffer contenente i dati sottoposti a marshaling alla libreria di runtime RPC, che lo trasmette all'altro processo.

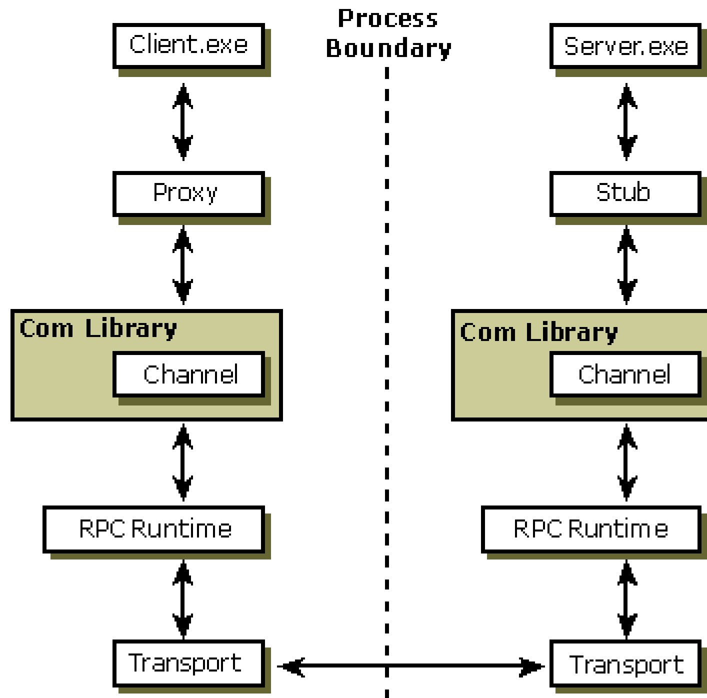

Fig. 1.7 – Standard Marshaling

1.12 COM e il registro di sistema

COM utilizza il registro di sistema di Windows per memorizzare informazioni riguardanti le varie parti che compongono un sistema COM. Durante lo sviluppo, si assegnano alle classi dei componenti dei nomi leggibili dall'essere umano, ognuna delle quali può essere quindi identificata a livello di sistema attraverso un nome completo, detto ProgID, che assume la forma *nomecomponente.nomeclasse*.
Questa modalità di denominazione può essere però soggetta a conflitti; per questo motivo COM assegna alle classi, alle interfacce, alle DLL, agli eseguibili e alle type libraries dei GUID (vedi paragrafo 1.4), che saranno utilizzati successivamente a runtime per riferirsi a tali componenti. Affinché sia possibile usare un oggetto COM, questo deve essere registrato nel registro di sistema, viene quindi creata una voce nel registro che associa il CLSID del componente al file su disco.
La funzione più importante e più facile da capire del registro di sistema in ambito COM riguarda l'istanziamento di oggetti COM in memoria. Nel caso di un server in-process, i passi seguiti sono i seguenti:

1. Il client richiede i servizi di un oggetto COM;

2. l'SCM (Service Control Manager COM) cerca l'oggetto richiesto attraverso il class ID (un GUID);

3. la DLL è trovata e caricata in memoria. L'SCM chiama la funzione della DLL *DllGetClassObject*, passando il nome della classe desiderata come primo parametro. La funzione *DllGetClassObject* è la funzione che rende una DLL una DLL COM;

4. la classe implementa normalmente l'interfaccia *IClassFactory*, attraverso la quale l'SCM crea un'istanza dell'oggetto chiamando il metodo *CreateInstance* su questa interfaccia;

5. infine, l'SCM richiede all'oggetto appena creato l'interfaccia che il client ha richiesto e passa un puntatore ad essa a quest'ultimo. Successivamente, l'SCM esce di scena e client e server comunicano tra loro direttamente.

La chiave di registro HKEY_CLASSES_ROOT (HKCR) contiene tutte le informazioni (ProgIDs, CLSIDs e IIDs) sugli oggetti COM presenti nel sistema.

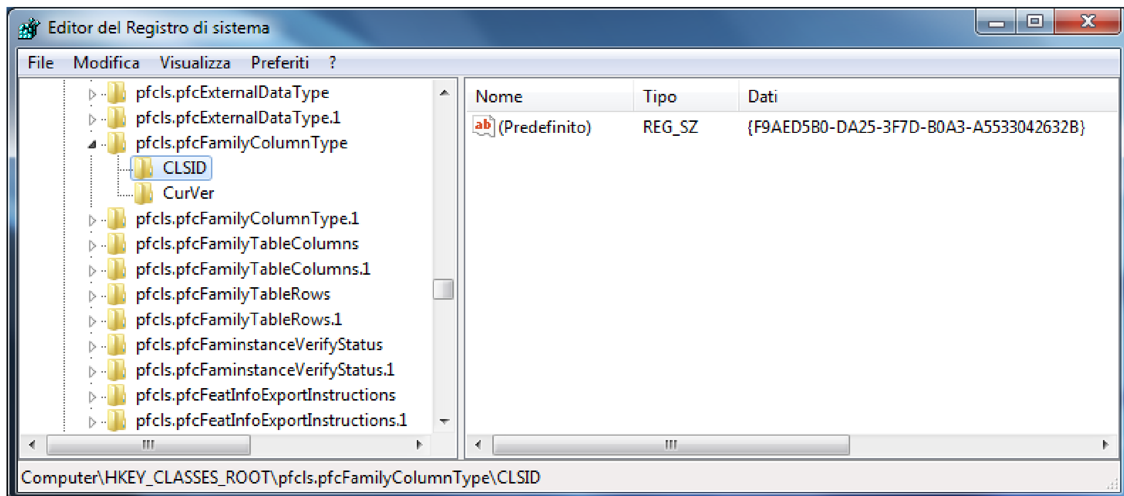

Fig. 1.8 – CLSID nel registro di sistema

Quando viene compilato un server ActiveX in Visual Basic 6, vengono generati ID distinti per ognuna delle classi e delle interfacce da queste esposte: un identificatore di classe è detto CLSID, mentre un identificatore di interfaccia è detto IID (a tutti gli effetti si tratta comunque di GUIDs). Questi identificatori sono salvati nella type library che Visual Basic genera per il server, che a sua volta sarà registrata nel registro di sistema (e alla quale si farà riferimento sempre attraverso un GUID).
Quando un componente deve essere eseguito, come prima cosa viene chiamata una funzione della libreria COM che, ricercando il ProgID della classe nel sottoalbero HKEY_CLASSES_ROOT del registro di sistema, permette di recuperare il corrispondente CLSID. Questa funzione è richiamata in fase di esecuzione, quando il programma istanzia il componente, oppure in fase di compilazione, quando il componente viene creato con l'operatore New.
Successivamente, COM ricerca il CLSID appena recuperato nel sottoalbero HKEY_CLASSES_ROOT\CLSID del registro di sistema, contenente tutte le informazioni necessarie all'esecuzione del componente, come il path all'eseguibile o il GUID della type library (se questa è stata creata in un file separato).

1.13 Compatibilità dei componenti

Spesso risulta necessario modificare un componente, per correggere degli errori o per aggiungere delle nuove funzionalità: queste modifiche possono causare problemi con le classi già esistenti. Ad esempio, se viene modificato l'ordine con il quale i metodi sono elencati in una vTable possono verificarsi problemi dovuti a chiamate ad un offset sbagliato; un altro problema potrebbe riguardare un numero di parametri oppure tipi diversi da quelli che il metodo si attende. Visual Basic 6 definisce tre livelli di compatibilità:

- **Versione identica**: una versione identica non apporta modifiche all'interfaccia della versione precedente. Questo significa che la modifica non riguarda le firme dei metodi o la definizione delle proprietà, ma solo la loro implementazione. In questo caso Visual Basic non crea nuovi GUID per le classi e le interfacce, ma utilizza gli stessi ID della versione precedente. Non sarà dunque necessario adeguare il codice dei client, i quali potranno utilizzare la nuova versione del componente senza alcun rischio di incompatibilità.

Un comportamento che in un primo momento può apparire ambiguo, è che durante la creazione di un componente compatibile Visual Basic 6 genera nuovi CLSID ed IID per ognuna delle classi e delle interfacce presenti. Il nuovo componente conterrà anche tutti i GUID delle versioni precedenti, così da mantenerne la compatibilità. Questa procedura porta ad un sovraffollamento di chiavi nel registro di sistema, soprattutto se il componente è stato compilato in compatibilità molte volte. Questo comportamento ha però una spiegazione: Visual Basic 6 si adegua alle regole COM, una delle quali specifica che ogni interfaccia pubblicata non dovrebbe mai essere cambiata;

- **Versione compatibile**: una versione è compatibile con la precedente se l'interfaccia prevede l'aggiunta di nuovi metodi e proprietà ma non la modifica dei membri esistenti. La versione compatibile manterrà inalterati gli offset delle funzioni già esistenti nella vTable, limitandosi ad estenderla rendendo così disponibili i nuovi elementi; solo i client compilati con la nuova versione del componente potranno utilizzarli. L'installazione del nuovo componente su una macchina nella quale è presente la versione precedente comporterà la sovrascrittura di quest'ultima;
- **Versione incompatibile**: la modifica dell'interfaccia dei metodi e delle proprietà già esistenti (come l'aggiunta o la rimozione di parametri, *anche se facoltativi*) porta ad una versione del componente incompatibile con la precedente. In questo caso è necessario cambiare il nome del nuovo componente, per evitare che l'installazione sovrascriva la versione precedente già utilizzata dai client.

2 Il Framework .NET

Il Framework .NET (nella sua versione 4.0) costituisce la piattaforma sulla quale la nuova architettura YAST e le nuove transazioni sono sviluppate. Sarebbe troppo ambizioso voler descrivere il Framework .NET nei dettagli in queste poche pagine, (sempre se possibile, dal momento che sull'argomento sono stati scritti libri interi), ma allo scopo di spiegare il funzionamento della nuova architettura è necessario assimilare alcuni concetti fondamentali sul suo funzionamento.

2.1 Cos'è il Framework .NET

Il .NET è una collezione di strumenti, tecnologie e linguaggi che cooperano in un framework per fornire le soluzioni richieste per sviluppare e distribuire robuste applicazioni aziendali. Le applicazioni .NET sono capaci di comunicare l'una con l'altra, a prescindere dalle piattaforme o dai linguaggi utilizzati. Qualcuno potrebbe lecitamente chiedersi a cosa sia dovuta l'etichetta "NET" nel nome: si tratta di un termine con il quale Microsoft ha voluto sottolineare l'importanza che assumeranno nel futuro le *applicazioni distribuite*, nelle quali l'elaborazione è distribuita tra client e server.

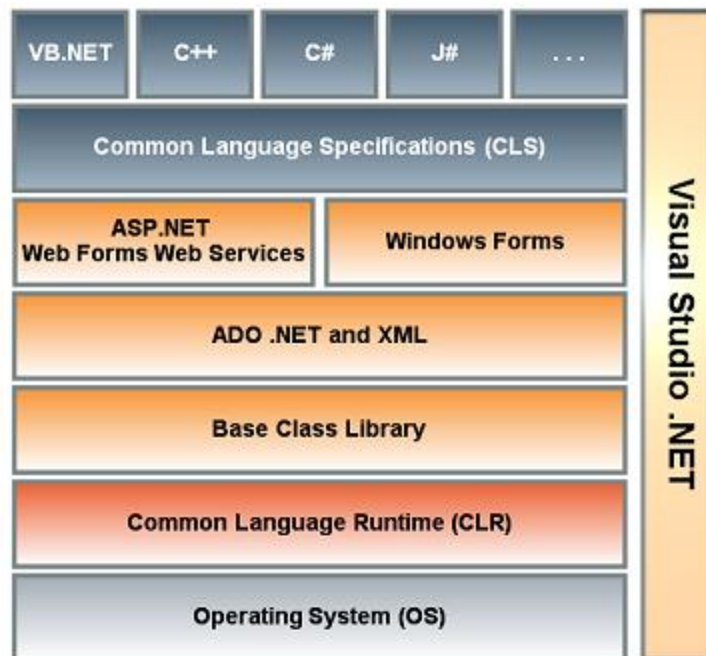

Fig. 2.1 – Struttura del Framework .NET

Guardando il diagramma in Fig. 2.1, si può notare come il Framework .NET poggi sopra al sistema operativo. Attualmente, i sistemi operativi che lo supportano in modo completo sono Windows XP, Windows 2000, Windows Vista, Windows 7 ed il neonato

Windows 8. Alla base del Framework si trova il *Common Language Runtime* (CLR), il cuore del Framework .NET , cioè la componente incaricata dell'esecuzione del codice e delle conseguenti allocazione di memoria, sicurezza, verifica del codice, verifica dei tipi, gestione delle eccezioni, accesso ai metadati e, non da ultimo, la gestione del Garbage Collector. Si tratta quindi dell'ambiente di runtime del Framework, il cui scopo è quello di eseguire tutte le applicazioni .NET scritte in uno dei linguaggi supportati dalla piattaforma (stiamo parlando di *linguaggi managed*, o *gestiti*). Per poter eseguire un pezzo di codice scritto ad esempio in C#, occorre utilizzare un compilatore che non produca il classico codice macchina direttamente interpretabile, ma una forma ibrida e parziale, chiamata *Microsoft Intermediate Language* (MSIL) o *Common Intermediate Language* (CIL), più comunemente detta *Intermediate Language* (IL). Nel momento dell'esecuzione, le istruzioni dell'IL sono gestite da un JIT-ter, un compilatore *Just-In-Time* (Fig. 2.2), che traduce l'Intermediate Language in codice macchina; è fondamentale capire che il processo di traduzione del codice viene eseguito on-demand, e non riguarda l'IL nella sua interezza, ma solo la porzione di codice che deve essere eseguita.

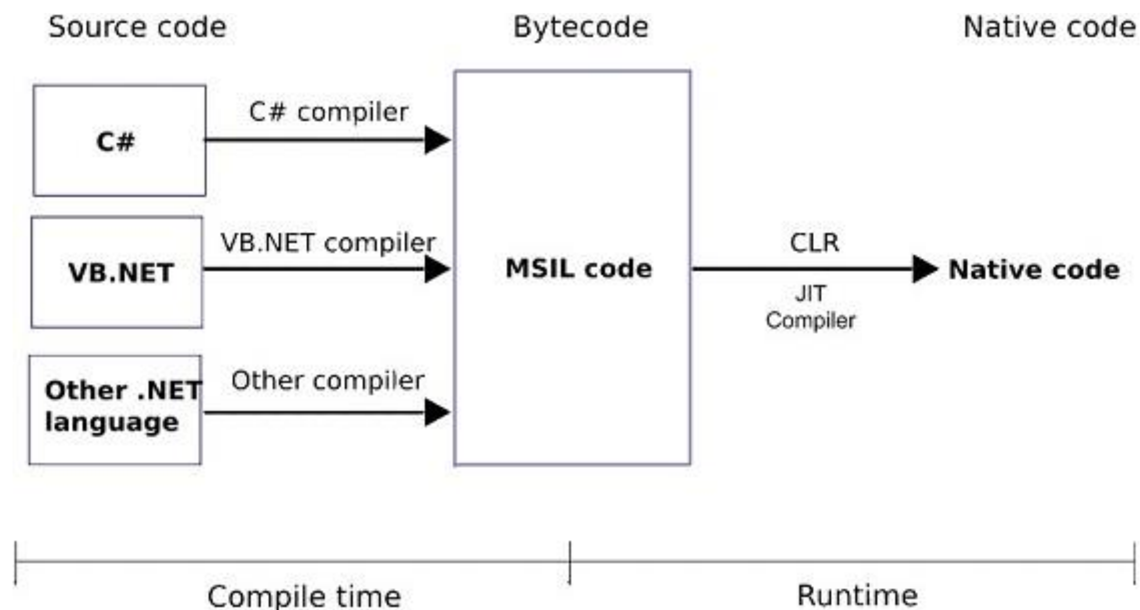

Fig. 2.2 – Conversione in MSIL e compilazione Just-In-Time

Durante il processo di traduzione da codice sorgente a MSIL, il compilatore aggiunge dei *metadati*, atti a descrivere il contenuto del risultato della compilazione; a tutti gli effetti sono informazioni aggiuntive che sono allegate al codice MSIL per descriverne meglio il contenuto. Al suo interno, il CLR è composto di due elementi fondamentali: MSCOREE.DLL, responsabile della compilazione del codice IL in codice macchina e dell'allocazione/deallocazione degli oggetti in memoria, e MSCORELIB.DLL, che contiene buona parte della Base Class Library, che vedremo a breve. Lo strato del Framework immediatamente superiore al CLR è la *Class Library*.
Come si può dedurre dal nome, si tratta di un insieme di classi standard, organizzata in una gerarchia di Namespace (ad esempio, la maggior parte delle chiamate a funzioni di sistema sono raccolte nel Namespace denominato “System”). Queste librerie implementano un gran numero di funzioni comuni, come la lettura-scrittura da file, interazione con database (ADO .NET) e la manipolazione di documenti XML.

La Class Library del Framework .NET è suddivisa in due parti:

- la Base Class Library (BCL): si tratta di un piccolo sottoinsieme dell'intera Class Library, che costituisce il nocciolo delle classi che vengono utilizzate come API di base del CLR. Le classi presenti in mscorlib.dll e alcune delle classi in System.dll e System.core.dll sono considerate parte del BCL.

Namespace presenti nella BCL
System
System.CodeDom
System.Collections
System.Diagnostics
System.Globalization
System.IO
System.Resources
System.Text
System.Text.RegularExpressions

Fig. 2.3 – Namespace presenti nella Base Class Library

- la Framework Class Library (FCL): questo insieme di classi comprende la BCL, e si riferisce all'intera Class Library del .NET Framework. Include un esteso insieme di librerie, come Windows Forms, ADO .NET, ASP .NET, Language Integrated Query (LINQ), Windows Presentation Foundation (WPF) e Windows Communication Foundation (WCF).

Prima di continuare, introducendo il livello più alto del Framework .NET, è opportuno chiarire il concetto di *codice gestito*.

2.2 Codice managed (Gestito)

Con il termine *codice managed* si indicano tutte quelle applicazioni gestite/eseguite attraverso il Common Language Runtime. Viceversa, del codice che non viene eseguito dal CLR è detto *unmanaged*, pertanto un programma scritto in C++ (che non sia C++ .NET) è un'applicazione unmanaged.
Durante la prima fase della compilazione si viene a creare, come visto in precedenza, il codice IL. Successivamente, nel momento dell'esecuzione, il CLR compila l'IL in codice macchina usando il JIT-ter, per poi eseguire un controllo sulla congruenza dei tipi e applica le politiche di sicurezza. Infine, il CLR crea un *Application Domain*, che non è altro che il contesto di esecuzione vero e proprio.
In sostanza, il percorso seguito dal CLR per l'esecuzione di codice managed è il seguente:

- cerca i metadati associati al membro richiesto;
- scorre lo stack di esecuzione;
- gestisce le eventuali eccezioni sollevate;
- gestisce le informazioni relative alla sicurezza.

E' importante notare che la compilazione Just-In-Time avviene solo per la prima richiesta; esiste infatti un processo il quale fa in modo che la fase di compilazione da codice sorgente in IL sia effettuata solamente una volta, riutilizzando la versione già compilata a fronte di successive richieste di esecuzione dello stesso codice.

2.3 La Common Language Specification (CLS)

Un'altra componente fondamentale del Framework .NET è la Common Language Specification. Abbiamo detto in precedenza che una delle particolarità che rendono l'ambiente di sviluppo .NET particolarmente vantaggioso è la capacità delle applicazioni, anche create con diversi linguaggi che supportino l'infrastruttura (come ad esempio VB. NET, C# e J#), di poter comunicare tra loro.
Il merito di questa integrazione è da attribuire alla Common Language Specification.
La CLS rappresenta una serie di specifiche che il compilatore ed il relativo linguaggio devono rispettare per fare in modo che un componente sia in grado di integrarsi con componenti scritti in linguaggi diversi. Perché questo avvenga, è quindi necessario che i tipi e i metodi *pubblici* utilizzati siano CLS-compliant, cioè compatibili con tali specifiche. I tipi e i metodi *privati*, invece, possono anche non esserlo, dal momento che solo quelli pubblici sono esposti direttamente al CLR. Un semplice esempio di quanto la CLS sia importante è rappresentato dal fatto che C# è *case sensitive* (e quindi distingue tra lettere maiuscole e minuscole), mentre VB .NET non lo è. Questo comporta che in C#, la variabile `Dummy` ha un significato differente rispetto a `dummy`, mentre in VB. NET i due nomi identificano lo stesso oggetto. In casi come questo entra in gioco la CLS, con lo scopo di eliminare ogni possibile ambiguità: nel caso appena visto, le specifiche vietano di esporre membri pubblici con lo stesso nome e case differente.
La CLS regola pertanto la modalità con cui i tipi devono essere esposti, strutturati ed organizzati.

2.4 Il Common Type System (CTS)

Il CTS rappresenta un'altra importante parte dell'architettura del Framework .NET. Il Common Type System, come si può dedurre dal nome, stabilisce come i tipi debbano essere dichiarati e gestiti dal CLR. Il suo scopo è, come appena visto per la CLS, garantire il supporto e l'integrazione multi-linguaggio. In effetti, ogni linguaggio di programmazione ha una propria sintassi e delle proprie regole, a volte molto differenti da linguaggio a linguaggio. Lo scopo del CTS è quello di fare in modo che un componente scritto, ad esempio, in C#, possa comunicare con un altro componente scritto in VB. NET condividendo lo stesso concetto di numero intero. Questo è possibile poiché entrambi i tipi, `int` per C# e `Integer` per VB. NET, sono convertiti in fase di compilazione nel tipo `System.Int32`, che è la rappresentazione del numero all'interno del Framework .NET. Per denotare i tipi, lo sviluppatore può usare le parole chiave tipiche del linguaggio in uso, oppure utilizzare direttamente i tipi definiti dal CTS, essendo i primi degli alias dei secondi.
Nella tabella in Fig. 2.4 sono messe a confronto le definizioni dei tipi primitivi nel CTS, in VB .NET, in C# e nel C++ standard.

CTS Data Type	VB .NET Keyword	C# Keyword	C++/CLI Keyword
System.Byte	Byte	byte	unsigned char
System.SByte	SByte	sbyte	signed char
System.Int16	Short	short	short
System.Int32	Integer	int	int or long
System.Int64	Long	long	__int64
System.UInt16	UShort	ushort	unsigned short
System.UInt32	UInteger	uint	unsigned int or unsigned long
System.UInt64	ULong	ulong	unsigned __int64
System.Single	Single	float	Float
System.Double	Double	double	Double
System.Object	Object	object	Object^
System.Char	Char	char	wchar_t
System.String	String	string	String^
System.Decimal	Decimal	decimal	Decimal
System.Boolean	Boolean	bool	Bool

Fig. 2.4 – Tipi di dati primitivi del CTS

2.5 Tipi di valore e tipi di riferimento

All'interno del CLR abbiamo il supporto per due categorie fondamentali di tipi:

- tipi di valore: rappresentati dalla maggior parte dei tipi primitivi, dalle enumerazioni e dalle strutture; contengono direttamente il valore dei dati;
- tipi di riferimento: sono rappresentati da classi, il loro scopo è quello di fornire una struttura del codice di tipo Object Oriented; contengono solo un riferimento ad una locazione di memoria.

Nella pratica, la differenza tra queste due categorie è rappresentata dalla zona di memoria dove gli oggetti di uno e dell'altro gruppo sono allocati: i tipi di valore sono allocati nello s*tack*, mentre i tipi di valore sono gestiti nel *managed heap* del Common Language Runtime. Quest'ultimo è gestito dal Garbage Collector, un componente in grado di deallocare in automatico la memoria occupata da oggetti che non sono più riferiti. I due prossimi capitoli saranno dedicati ad un approfondimento sulla gestione dello stack, dell'heap e del Garbage Collector da parte del Framework .NET.

2.6 Stacking e Heaping nel Framework .NET

L'idea di aggiungere questo capitolo sulla gestione della memoria, ed il successivo sul Garbage Collector, risponde ad un'esigenza che, a mio parere, è di importanza fondamentale per quanto riguarda l'utilizzo di codice managed da parte di uno sviluppatore. Chiunque abbia avuto a che fare con del codice non gestito, sa cosa vuol dire farsi carico della gestione della memoria dell'applicazione, la quale, se non effettuata in maniera accurata, può generare pericolosi ***memory leaks***, cioè un consumo involontario di memoria dovuto, appunto, alla mancata deallocazione della memoria da parte dei processi che la utilizzano. La programmazione sul Framework .NET, che adotta una strategia di deallocazione automatica della memoria, viene quindi giustamente usato con sollievo da molti, ma, a mio avviso, necessita che lo sviluppatore

conosca nei minimi dettagli il funzionamento della Garbage Collection, se non altro per tenere sotto controllo l'efficacia delle applicazioni dal punto di vista delle prestazioni.
Iniziamo dicendo che la memoria che un programma utilizza è divisa in quattro sezioni:

- l'area del codice, dove il programma compilato risiede in memoria;
- l'area delle variabili globali, dove sono salvate le variabili globali;
- lo stack, dove sono allocati i parametri passati alle funzioni e le variabili locali di queste ultime;
- l'heap, dove risiedono le variabili allocate dinamicamente.

Sulle prime due aree non vi è molto da dire. Le sezioni più importanti, dal punto di vista dell'esecuzione del codice, sono lo stack e l'heap.

2.6.1 Lo Stack delle chiamate

Lo stack delle chiamate è una porzione di memoria incaricata di tenere traccia, in ogni istante dall'inizio dell'esecuzione, delle procedure o funzioni chiamate nel codice. Il suo funzionamento si basa sulla struttura dati omonima: la struttura dati stack. Come suggerisce il nome, si tratta di una pila di oggetti posti l'uno sull'altro: nella fase di inserimento (detta *push*) si aggiunge il nuovo elemento sopra a quello che è stato precedentemente inserito, nella fase di estrazione (detta *pop*) si estrae l'elemento in cima (detta *top*) alla pila (Fig. 2.5). Questo comportamento categorizza lo stack tra quelle strutture dati che utilizzano una politica di accesso detta *LIFO* (*Last-In First-Out*).

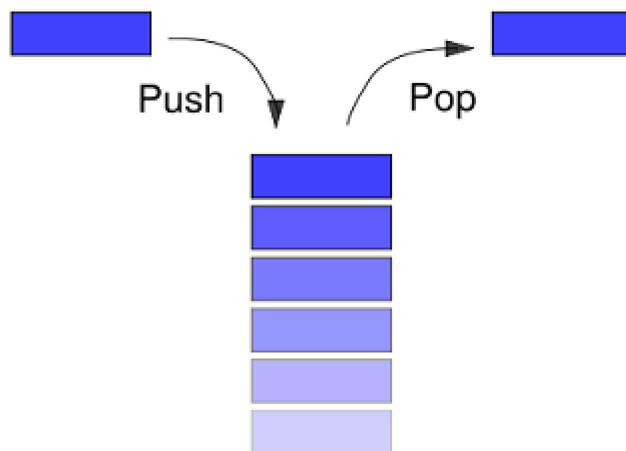

Fig. 2.5 – La struttura dati Stack

Gli elementi logici che durante l'esecuzione del codice sono allocati nello stack sono detti *frame*. Ogni funzione richiamata dal programma è associata, nel momento del richiamo, ad un proprio frame, il quale contiene i parametri passati dalla procedura chiamante, le variabili locali della funzione e un indirizzo di ritorno.
Lo stack è effettivamente gestito dall'elaboratore attraverso due puntatori, lo *Stack Pointer* (SP) e il *Frame Pointer* (FP) (quest'ultimo è anche detto *Base Pointer* o BP). Lo Stack Pointer punta sempre al top dello stack, mentre il Frame Pointer svolge una duplice funzione: nel corso della esecuzione della procedura, esso può essere utilizzato come registro base per accedere, con opportuni valori di offset, ai dati (parametri o locali) che si trovano nel frame; terminata l'esecuzione della procedura, l'informazione contenuta nel Frame Pointer viene usata per rilasciare l'area di memoria allocata e

riportare lo stack nello stato in cui si trovava prima che la procedura fosse attivata. A seconda dell'implementazione, si dice che lo stack "cresce verso l'alto", quindi verso indirizzi di memoria sempre più alti, oppure "cresce verso il basso", cioè verso indirizzi di memoria via via minori. Sui processori *Intel*, *Motorola*, *SPARC* e *MIPS* lo stack cresce verso il basso. L'operazione di inserimento di un frame nello stack si ottiene con le seguenti operazioni (Fig. 2.6 e Fig. 2.7):

- salvataggio dell'indirizzo del frame precedente nello stack (push di FP);
- attivazione del nuovo frame (SP → FP);
- allocazione di un'area di memoria di estensione *ext* per il nuovo frame ((SP – ext) → SP).

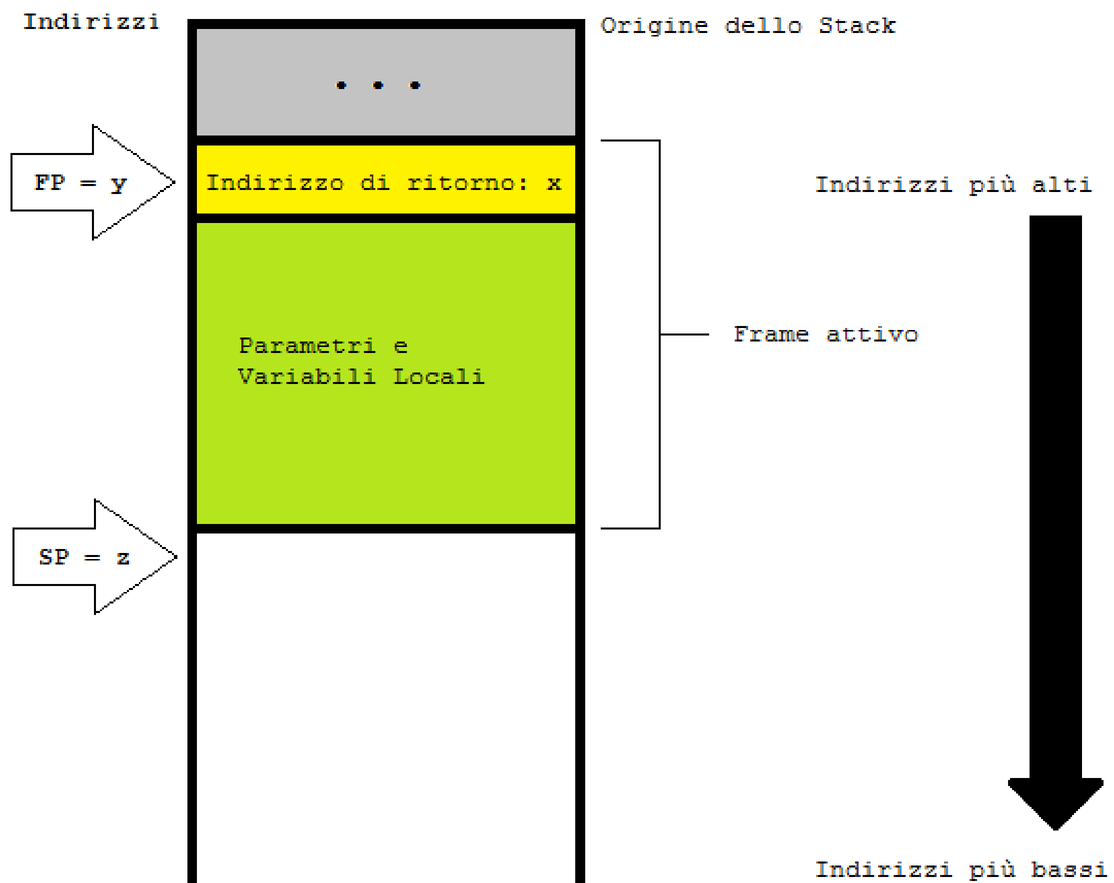

Fig. 2.6 – Situazione iniziale dello stack

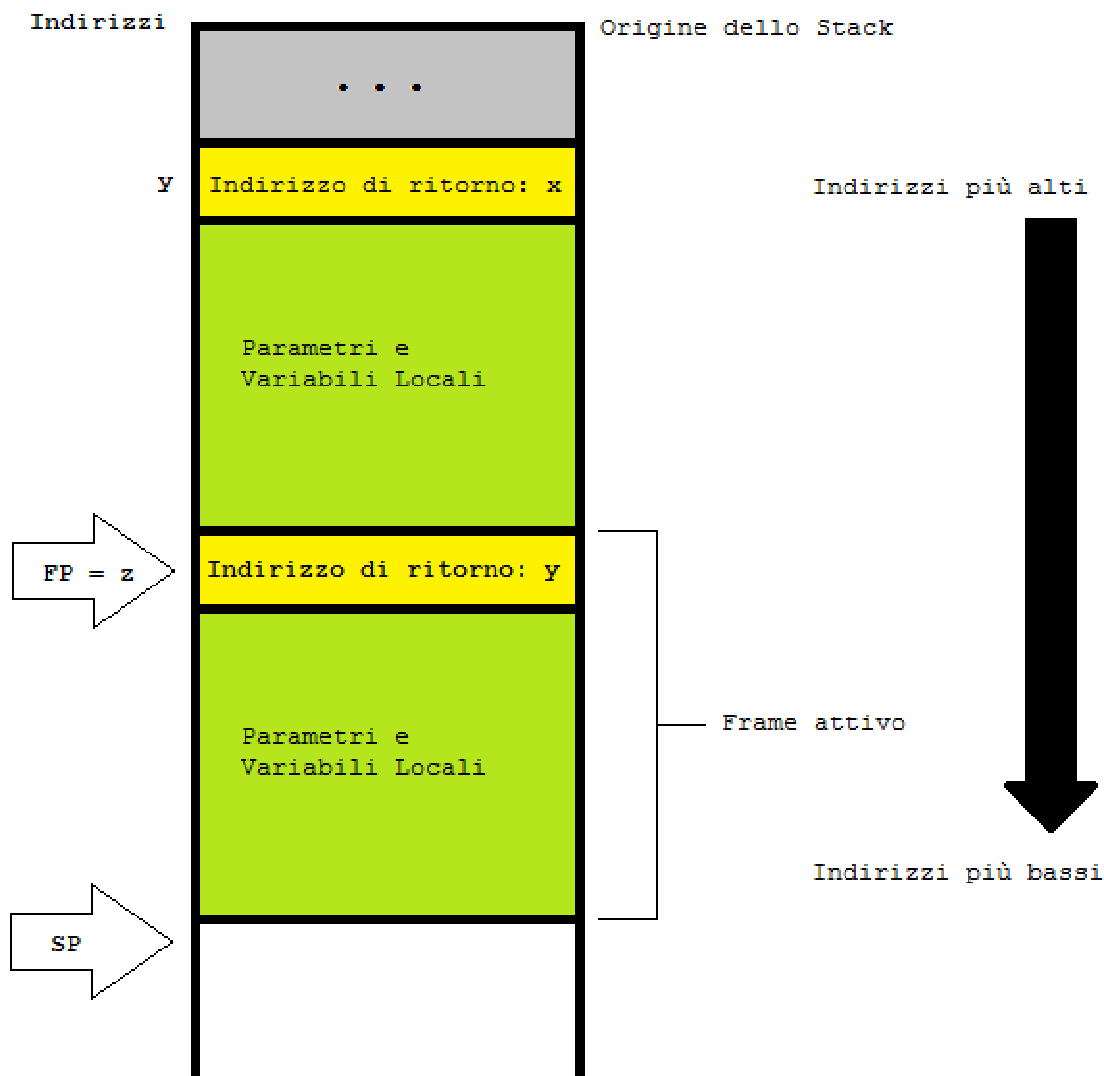

Fig. 2.7 – Lo stack dopo l'inserimento di un nuovo frame

L'operazione di rilascio dell'area di memoria corrispondente al frame di una funzione avviene riportando lo stack nello stato in cui si trovava prima che la procedura fosse attivata: ciò si ottiene, in modo abbastanza semplice ed immediato, con le seguenti operazioni:

- rilascio del frame attivo (FP → SP);
- riattivazione del frame precedente, ripristinando nel registro FP il valore precedentemente salvato (pop FP).

Nel framework .NET, ogni qual volta dal codice viene lanciata un'eccezione, il CLR scorre lo stack all'indietro in cerca di un blocco catch da utilizzare per gestirla. Se questo blocco non viene trovato, il CLR terminerà l'applicazione.
In Visual Studio 2010, durante l'esecuzione in modalità Debug di un'applicazione, è possibile visionare lo stack delle chiamate in qualsiasi momento. In questo modo lo stack diventa un'utile fonte di informazioni sul flusso di esecuzione del programma, fornendo addirittura una valida alternativa all'utilizzo di log.

Come esempio, prendiamo il codice mostrato in figura 2.8. La classe `Program` presenta un metodo `Main` che innesca una sequenza di chiamate annidate:

```
1  namespace ConsoleApplication1
2  {
3      using System;
4
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              ChiamamiPerPrimo("A");
10             Console.ReadLine();
11         }
12
13         private static void ChiamamiPerPrimo(string arg)
14         {
15             Console.WriteLine(arg);
16             ChiamamiPerSecondo("B");
17         }
18
19         private static void ChiamamiPerSecondo(string arg)
20         {
21             Console.WriteLine(arg);
22             ChiamamiPerTerzo("C");
23         }
24
25         private static void ChiamamiPerTerzo(string arg)
26         {
27             Console.WriteLine(arg);
28         }
29     }
30 }
31
```

Fig. 2.8 – Esempio di codice con chiamate annidate

Eseguendo questa applicazione in Debug e bloccando l'esecuzione in corrispondenza della riga 27, la finestra dello stack delle chiamate mostrerà il seguente contenuto:

```
ConsoleApplication1.exe!ConsoleApplication1.Program.ChiamamiPerTerzo(string arg) Riga 27
ConsoleApplication1.exe!ConsoleApplication1.Program.ChiamamiPerSecondo(string arg) Riga 22 + 0xb byte
ConsoleApplication1.exe!ConsoleApplication1.Program.ChiamamiPerPrimo(string arg) Riga 16 + 0xb byte
ConsoleApplication1.exe!ConsoleApplication1.Program.Main(string[] args) Riga 9 + 0xb byte
```

Fig. 2.9 – Aspetto dello stack delle chiamate per il codice in figura 2.7

2.6.2 Il Managed Heap

L'heap è un'area di memoria utilizzata per l'allocazione dinamica della memoria e, al contrario dello stack, non ha nulla a che vedere con l'omonima struttura dati. A differenza dello stack, i blocchi di memoria sono allocati e liberati senza seguire alcun ordine particolare, perché la modalità di allocazione e la dimensione di questi blocchi non sono conosciute fino al momento dell'esecuzione effettiva del codice. Inoltre,

mentre lo stack gestisce autonomamente l'allocazione e la deallocazione dei frame, l'heap deve preoccuparsi della pulizia dei blocchi di memoria non più utilizzati.
Per soddisfare una richiesta di allocazione a runtime, è necessario cercare e trovare un blocco di memoria non utilizzata (chiaramente delle dimensioni adeguate) nell'heap, e questo comporta una serie di problematiche di non semplice soluzione: il problema principale per la maggior parte degli algoritmi di allocazione dinamica della memoria è quello di evitare la *frammentazione* esterna, cercando di mantenere efficiente l'allocazione e la deallocazione.

2.6.3 La frammentazione esterna

Essendo l'heap soggetto a continue allocazioni e deallocazioni di oggetti di dimensioni eterogenee, il suo spazio libero non tarda ad essere frammentato tra blocchi di memoria utilizzati. In questo caso, la ricerca nell'heap dello spazio necessario per allocare nuovi oggetti diventa inefficiente, perché risulta necessario trovare un blocco di memoria libera di dimensioni sufficienti tra i vari frammenti. Consideriamo come esempio un heap che ha origine alla locazione di memoria `0x4000`, con una estensione di 32 byte (Fig. 2.10 a), e supponiamo di allocare 3 oggetti in sequenza, due da 8 byte (in rosso e in arancione in Fig. 2.10 b) e l'ultimo da 16 byte (in grigio in Fig. 2.10 b).

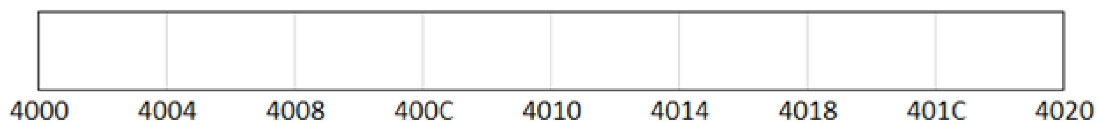

Fig. 2.10 a – Heap di 32 byte vuoto

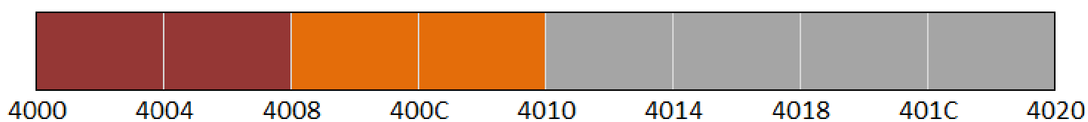

Fig. 2.10 b – Heap di 32 byte dopo il riempimento

Supponiamo ora di rimuovere il primo oggetto da 8 byte e quello da 16 byte, l'heap risultante avrebbe in tal caso 24 byte di memoria libera (Fig. 2.10 c), ma un oggetto delle dimensioni maggiori di 16 byte non troverebbe posto, a causa della frammentazione delle due aree di memoria libera.

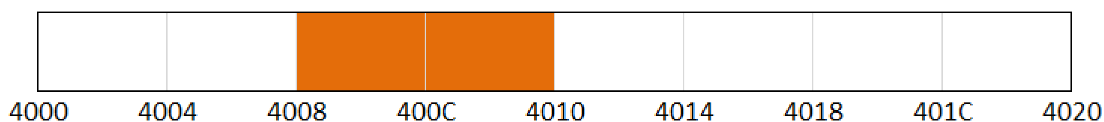

Fig. 2.10 c – Heap frammentato

Nei prossimi capitoli vedremo come il Framework .NET gestisce l'allocazione e la deallocazione della memoria nel managed heap, e come, attraverso la garbage collection, venga ridotto il fenomeno della frammentazione.

2.6.4 Allocazione di memoria nel Managed Heap

Durante l'esecuzione di codice gestito, il Managed Heap mantiene un puntatore che indica la posizione in memoria dove il prossimo oggetto dovrà essere allocato. Inizialmente, questo puntatore punterà al base address dell'area di memoria dedicata all'heap. Quando viene eseguita un'istruzione di allocazione, l'operatore `new` si assicura che la regione di memoria puntata possa effettivamente contenere i bytes richiesti dal nuovo oggetto, facendo spazio se necessario. Se l'oggetto può essere allocato in quella regione, allora verrà generato un riferimento ad esso, sarà invocato il suo costruttore ed infine l'operatore `new` restituirà il suo indirizzo in memoria.
Al termine delle operazioni che coinvolgono un oggetto precedentemente allocato, è generalmente sufficiente lasciare che il Garbage Collector (*GC*) si prenda carico della rimozione della porzione di memoria non più riferita. Bisogna però fare attenzione: stiamo parlando di memoria, non di risorse. Se viene creato un nuovo oggetto nel Managed Heap e quest'ultimo alloca risorse come handle o connessioni, allora bisogna prima assicurarsi che quell'oggetto abbia prima rilasciato le sue risorse prima di delegare il lavoro di pulizia al Garbage Collector.
Quando il CLR è caricato, generalmente, sono inizializzati due Managed Heaps:

- `SOH` (*Small Objects Heap*), dedicato a oggetti di dimensione relativamente piccola, generalmente di dimensioni inferiori agli 83KB (84992 Byte);

- `LOH` (*Large Objects Heap*), per l'allocazione di oggetti di grandi dimensioni, maggiori di 83KB.

Quando un processo richiede una data quantità di memoria, il CLR esamina la richiesta e assegna al processo la memoria di una di queste due aree in base alla quantità richiesta. Un processo può lanciare anche altri thread, i quali avranno necessariamente un loro stack, ma non un loro heap: ogni thread usa infatti lo stesso Managed Heap, e questo significa che il `SOH` e il `LOH` sono condivisi tra tutti i thread in esecuzione. Abbiamo visto che l'algoritmo utilizzato dal CLR per l'allocazione di memoria nel Managed Heap è piuttosto semplice: dopo una richiesta di memoria, questa viene allocata e il puntatore alla prossima regione di memoria allocabile viene aggiornato.
Questo modo di procedere è completamente diverso rispetto a quello utilizzato in altri linguaggi di programmazione, ad esempio C++, dove la memoria è allocata nell'heap attraverso delle *linked lists* (*liste concatenate*) direttamente gestite dal Sistema Operativo.
La versione 4 del Framework .NET ha una limitazione di un massimo di 2GB di allocazione di memoria per ogni singolo processo a 32 bit. Questa limitazione è dovuta al fatto che su macchine a 32 bit sono indirizzabili al massimo 2^32 = 4.294.967.296 byte, corrispondenti a 4GB memoria indirizzabile. Questa è però distribuita equamente tra Kernel (2GB) e memoria per le applicazioni (2GB), da cui il limite di cui prima. Su macchine a 64 bit, su cui sono indirizzabili 2^64 = ~1,8x10^19 byte, corrispondenti a 16EB (ExaByte), il limite di memoria dedicata alle applicazioni resta al momento sempre di 2GB.

2.6.5 Deallocazione di memoria nel Managed Heap

La deallocazione della memoria nel Managed Heap funziona in maniera diversa rispetto a quella utilizzata nelle applicazioni Windows normali. Il Framework .NET possiede un sofisticato meccanismo di deallocazione della memoria denominato Garbage Collector. L'algoritmo utilizzato dal Garbage Collector è abbastanza semplice:

- Etichetta tutti gli oggetti nell'heap come “garbage”;
- Cerca i blocchi di memoria ancora riferiti e li etichetta come validi;
- Dealloca tutti gli oggetti segnalati come “garbage”;
- Ricompatta l'heap.

Ma come fa il Garbage Collector a sapere quando un oggetto non è più utilizzato?
Ogni applicazione possiede un insieme di ***radici***. Ogni radice riferisce ad un oggetto allocato nel Managed Heap o è impostata ad un valore nullo. Per esempio, tutti i puntatori ad oggetti statici e globali di un'applicazione sono considerati radici di questa, come anche i registri della CPU contenenti puntatori ad oggetti che risiedono nel Managed Heap.
La lista delle radici attive è mantenuta dal JIT e dal CLR, ed è accessibile all'algoritmo del Garbage Collector, il quale, all'inizio del processo di garbage collection assume che tutti gli oggetti contenuti nell'heap siano da eliminare: in altre parole, assume che nessuna delle radici dell'applicazione riferisca ad un oggetto nell'heap. Quindi, il GC comincia ad attraversare le radici e costruisce un grafo contenente tutti gli oggetti raggiungibili da queste (ad esempio, potrebbe individuare una variabile locale che punta ad un oggetto nell'heap).
In figura 2.11 è rappresentato un heap con alcuni oggetti allocati e il puntatore `NextObjPtr` alla prossima area di memoria disponibile per la prossima allocazione. Le radici dell'applicazione riferiscono direttamente agli oggetti A, C, D e F, che diventeranno parte del grafo creato dal GC. Come si vede dalla figura, l'oggetto D riferisce all'oggetto H, pertanto, quando il Garbage Collector inserirà D nel grafo, noterà questo riferimento “indiretto” ed anche H sarà aggiunto al grafo. Il procedimento termina solo dopo che tutti gli oggetti raggiungibili sono stati aggiunti al grafo, in maniera ricorsiva.

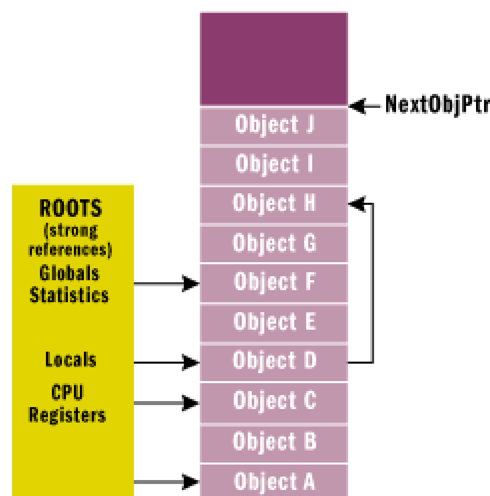

Fig. 2.11 – Heap e puntatore NextObjPtr

Quando questa parte del grafo è completa, il Garbage Collector controlla la prossima radice e scorre gli oggetti riferiti nuovamente. Se incontra un oggetto che era già stato

aggiunto in precedenza, semplicemente non lo aggiunge al grafo e termina il controllo su quel ramo. Questo comportamento è estremamente importante, per due motivi:

- aumenta le performances in maniera significativa, dal momento che non attraversa più di una volta lo stesso oggetto;
- previene la possibilità di loop infiniti dovuti, ad esempio, alla presenza in memoria di strutture dati come *liste concatenate circolari.*

Alla fine del processo di controllo dei riferimenti, quando tutte le radici sono state esaminate, il grafo del Garbage Collector conterrà l'insieme di tutti gli oggetti che sono, direttamente o indirettamente, accessibili dall'applicazione; i restanti oggetti sono invece considerati “garbage”.
A questo punto, il GC attraversa l'heap in maniera lineare, cercando i blocchi di memoria da eliminare (considerati ora spazio libero), quindi compatta tutti gli oggetti ancora riferiti verso gli indirizzi più bassi (ricordiamo che l'heap cresce con gli indirizzi di memoria), rimuovendo così la frammentazione che si era venuta a creare con la rimozione di porzioni di memoria non contigue.
Ovviamente, questo passaggio invalida tutti i riferimenti agli oggetti nell'heap, pertanto la successiva azione compiuta dal GC è quella di modificare le radici dell'applicazioni per fare in modo che i puntatori puntino alle nuove locazioni, correggendo anche eventuali riferimenti tra oggetti (vedi il caso degli oggetti D e H in Fig. 2.11). E' opportuno notare che, per evitare un degeneramento delle prestazioni, il compattamento viene effettuato solo tra gli oggetti dello *Small Object Heap* e non tra quelli del *Large Object Heap,* evitando così lo spostamento di oggetti di grandi dimensioni.
Alla fine del processo di compattamento, l'heap rappresentato in Fig. 2.11 assume la configurazione visibile in Fig. 2.12, e al GC non resta che far puntare il puntatore `NextObjPtr` all'area di memoria immediatamente successiva all'ultimo oggetto presente nell'area appena compattata.

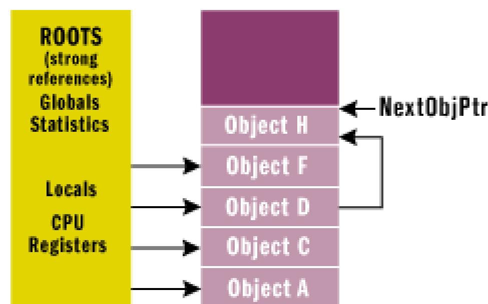

Fig. 2.12 – Heap della figura 2.11 dopo il processo di compattamento

L'esecuzione della Garbage Collection incide in maniera significativa sulle performances, e questo è l'effetto collaterale più importante dell'utilizzo di un Managed Heap. In ogni caso, il GC viene richiamato solo quando necessario (a meno che non sia richiamato esplicitamente dal codice), quindi solo quando l'heap è pieno. Fino a quel momento, un Managed Heap è senza dubbio più veloce di un heap in ambiente runtime di C. Allo scopo di ottimizzare il funzionamento del Garbage Collector è stato introdotto il concetto di ***generazioni***, che verrà esaminato nel prossimo paragrafo.

2.6.6 Generazioni

Per ottimizzare le performances del Garbage Collector, il SOH è suddiviso in tre *generazioni*: 0, 1 e 2 (Fig. 2.13). L'algoritmo che il GC utilizza è a tutti gli effetti il prodotto di alcune assunzioni che l'industria del software ha scoperto essere sempre valide, sperimentando diversi algoritmi di garbage collection. La prima di queste afferma che è più veloce compattare la memoria di una sola porzione piuttosto che di tutto il Managed Heap. La seconda asserisce che più gli oggetti sono recenti è più la loro vita sarà breve, mentre quelli da più a lungo in memoria avranno tempi di vita più lunghi. Infine, i nuovi oggetti tendono ad essere relazionati gli uni agli altri e sono utilizzati dall'applicazione più o meno nello stesso arco di tempo.
A runtime, il GC alloca i nuovi oggetti nella generazione 0. Gli oggetti creati precedentemente e che sono sopravvissuti alle collections sono promossi e allocati nelle generazioni 1 e 2. Dal momento che, come accennato sopra, è più veloce compattare una regione dell'heap piuttosto che tutta la memoria, al garbage Collector è concesso di rilasciare la memoria in una generazione specifica ogni volta che viene eseguita una collezione, che in genere viene lanciata quando la generazione 0 è satura. Se un'applicazione tenta di istanziare un nuovo oggetto quando la generazione 0 è piena, il Garbage Collector scopre che non vi è spazio per allocare l'oggetto ed esegue una collezione esclusivamente sulla generazione 0, poiché, in base alla seconda assunzione, è proprio lì che risiedono gli oggetti con vita più breve. In genere, una collezione effettuata nella generazione 0 libera abbastanza spazio per consentire all'applicazione di procedere con la creazione di nuovi oggetti. Dopo aver eseguito la collezione e il compattamento, il GC promuove gli oggetti ancora riferiti e li passa alla generazione 1 del Managed Heap, dal momento che sembrano promettere un tempo di vita più lungo. Come risultato, il Garbage Collector non dovrà riesaminare nuovamente gli oggetti nelle generazioni 1 e 2 ogni volta che esegue una collezione nella generazione 0, ottimizzando in questa maniera le performances.

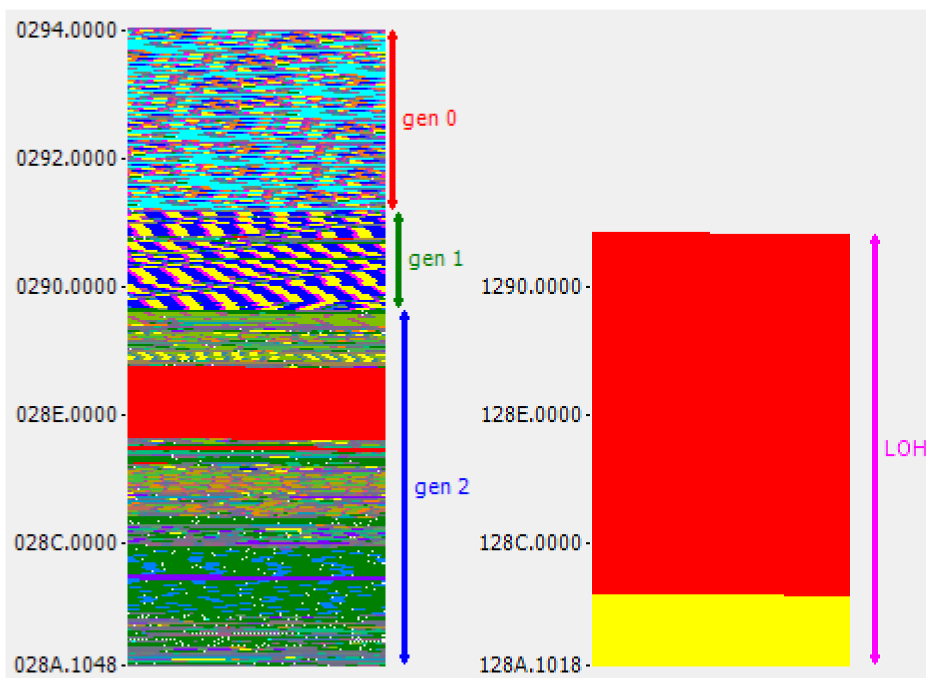

Fig. 2.13 – Generazioni 0, 1, 2 e il Large Object Heap

Dopo che il Garbage Collector ha effettuato la sua prima collezione sulla generazione 0 e ha promosso gli oggetti raggiungibili alla generazione 1, continua ad allocare memoria per i nuovi oggetti nella generazione 0 fino a quando non sarà necessaria una nuova collezione. Questo processo è interrotto nel momento in cui il Garbage Collector ritiene necessario esaminare gli oggetti nelle generazioni contenenti oggetti più "vecchi". Per esempio, se una collezione della generazione 0 non libera spazio sufficiente per l'allocazione di un nuovo oggetto, il GC può effettuare una collezione sulla generazione 1 e poi sulla generazione 2. In ogni caso, dopo ogni collezione, il GC compatta gli oggetti raggiungibili nella generazione 0 e li promuove alla generazione 1. Gli oggetti che sopravvivono alle collezioni sulla generazione 1 sono promossi alla generazione 2. In quest'ultima, supportando il GC solo tre generazioni, gli oggetti che non vengono eliminati dalla collezione non vengono spostati, ma rimangono nella generazione 2 fino a quando non saranno considerati irraggiungibili. Finalizzazione e risorse non gestite
Abbiamo visto che quando il Framework .NET istanzia un oggetto, lo alloca nel Managed Heap. L'oggetto rimane in memoria fino a quando non è più referenziato da codice attivo, e a quel punto viene considerato "garbage", quindi è pronto per essere dellocato dal Garbage Collector.
Prima che il GC deallochi la memoria, il Framework .NET richiama il metodo `Finalize()` dell'oggetto, ma gli sviluppatori sono responsabili della chiamata al metodo `Dispose()`. I due metodi non sono equivalenti, nonostante eseguano entrambi la pulizia dell'oggetto, e hanno delle differenze significanti tra loro. Il Garbage Collector, agendo sul Managed Heap, si occupa della rimozione di oggetti nativi .NET, quindi di risorse gestite: il rilascio di eventuali risorse non gestite (ad esempio file handles e connessioni) è qualcosa che va oltre il suo scopo, e queste devono essere rilasciate esplicitamente da codice. Esistono infatti delle situazioni particolari dove è necessario allocare memoria per delle risorse non gestite da codice gestito. Questo può accadere quando viene instaurata una connessione ad un database attraverso una classe. L'istanza di connessione al database è una risorsa non gestita incapsulata nella classe e dovrà essere rimossa non appena non sarà più necessaria. In casi come questi, è necessario liberare la memoria occupata dalle risorse non gestite in modo esplicito, poiché il Garbage Collector non la deallocherà implicitamente.

2.6.7 Finalizzatori: pulizia implicita delle risorse

La finalizzazione è il processo con cui il Garbage Collector permette agli oggetti di sbarazzarsi di ogni loro risorsa non gestita, prima di distruggere a tutti gli effetti l'istanza dell'oggetto. La distruzione effettiva di un oggetto è segnata dalla chiamata al suo metodo `Finalize()`, cui segue l'effettivo rilascio della memoria. Un'implementazione del metodo `Finalize()` è detta "finalizzatore" (*finalizer*), il cui scopo è quello di liberare qualsiasi risorsa esterna che un oggetto stia trattenendo. La finalizzazione è un processo di importanza fondamentale, poiché una classe che contenga riferimenti a risorse non gestite e senza l'implementazione di un finalizzatore può portare alla presenza di *memory leaks* dovuti a risorse non gestite orfane se l'istanza della classe viene distrutta prima del loro rilascio. Inoltre, l'implementazione dei finalizzatori è un'operazione che deve essere eseguita con la massima attenzione, potendo portare ad un deterioramento delle prestazioni dell'applicazione se non eseguita correttamente.
Il problema delle prestazioni deriva dal fatto che gli oggetti finalizzabili sono inseriti e rimossi nelle cosiddette *code di finalizzazione* (*finalization queues*), che sono strutture dati interne contenenti puntatori agli oggetti che implementano un finalizzatore,

controllate dal Garbage Collector. Quando i puntatori a questi oggetti sono inseriti nella coda, si dice che l'oggetto è stato *inserito nella coda di finalizzazione*. Ad ogni collezione, il GC controlla se ogni oggetto considerato “garbage” è puntato da un elemento della lista e, se viene trovata una corrispondenza, il puntatore interessato nella coda di finalizzazione è spostato in un'altra coda, detta *Freachable* (pronunciata *“F-Reachable”*). In questa coda (controllata sempre dal GC) sono aggiunti i puntatori agli oggetti il cui metodi `Finalize()` è pronto per essere chiamato. In figura 2.14 è possibile vedere la rappresentazione di un Managed Heap contenente 10 oggetti, A, B, C, D, E, F, G, H, I e J. In particolare, gli oggetti C, E, F, I e J possiedono un finalizzatore e di conseguenza dei puntatori a tali oggetti sono posti all'interno della coda di finalizzazione.

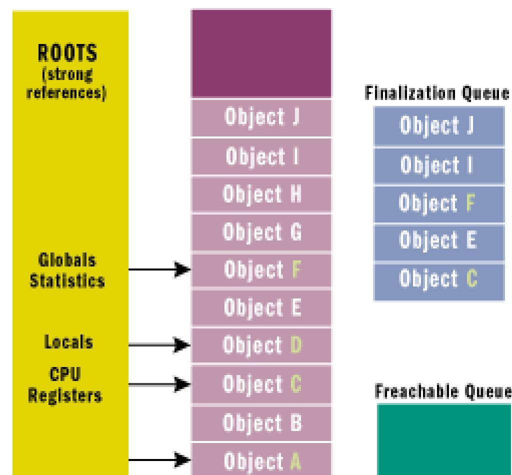

Fig. 2.14 – Coda di finalizzazione e coda Freachable prima della collezione

Nel momento in cui viene effettuata una collezione, il Garbage Collector individua gli oggetti A, C, D ed F come raggiungibili (come si può vedere in figura sono puntati da delle radici), quindi controlla nella coda di finalizzazione se sono presenti riferimenti ai restanti oggetti, i quali, non avendo riferimenti, sono destinati ad essere eliminati. Dopo la collezione, la situazione del Managed Heap e delle due code è quella rappresentata in Fig. 2.15. Gli oggetti B e G, non avendo finalizzatore, sono semplicemente stati rilasciati, mentre i puntatori agli oggetti E, I, J, presenti nella coda di finalizzazione prima della collezione, sono stati spostati nella coda Freachable (F e C invece non sono stati spostati perché gli oggetti corrispondenti sono ancora riferiti da delle radici).
Gli oggetti E, I e J non sono ancora stati rilasciati, poiché il loro metodo `Finalize()` deve ancora essere richiamato.

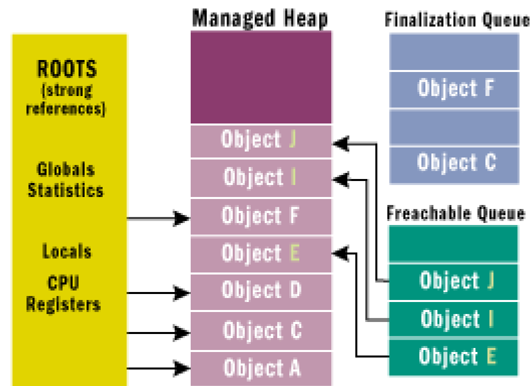

Fig. 2.15 – Coda di finalizzazione e coda Freachable dopo la collezione

La chiamata ai metodi `Finalize()` degli oggetti puntati degli elementi nella coda Freachable è effettuata da un thread particolare che, se la coda è vuota, è dormiente. Non appena la coda si popola di qualche puntatore, il thread viene risvegliato e ne rimuove tutte le entry, chiamando prima, per ognuna di queste, il metodo `Finalize()` dell'oggetto puntato.
E' bene evidenziare che l'esecuzione del metodo di finalizzazione è eseguito su un thread separato gestito dal GC, per questo motivo non bisognerebbe mai eseguire, in un metodo `Finalize()`, del codice che interagisca con il thread che lo sta eseguendo.
Dopo che è stato invocato il metodo `Finalize()`, la memoria allocata per gli oggetti E, I e J non è liberata immediatamente: il rilascio vero e proprio avverrà con la successiva collezione, durante la quale il Garbage Collector tratterà gli oggetti E, I e J come "spazzatura", non avendo questi più alcun puntatore nella coda di finalizzazione. Nella realtà potrebbe essere necessario attendere più di due collezioni prima del rilascio effettivo, dal momento che gli oggetti potrebbero essere spostati in un'altra generazione. La situazione finale del managed Heap e delle due code è rappresentata in Fig. 2.16.

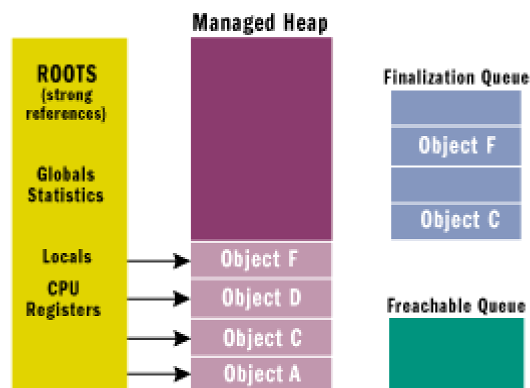

Fig. 2.16 – Situazione delle code dopo l'esecuzione del metodo Finalize()

Il tempo e l'ordine di esecuzione dei finalizzatori non può essere predeterminato o previsto. Per questo motivo si dice che la natura della finalizzazione è non-deterministica. Proprio a causa di questo comportamento, il Framework .NET non può garantire che il metodo `Finalize()` sia chiamato su una data istanza, di conseguenza non è possibile appoggiarsi a questo metodo per rilasciare delle risorse non gestite.
In C#, il metodo `Finalize()` non può essere chiamato da codice, né può essere soggetto a override: è generato implicitamente se è presente un distruttore per la classe. Ad esempio, nel codice:

```
public class Test
{
    // Codice

    ~Test()
    {
        // Codice eseguito in Finalize()
    }
}
```

la sintassi `~Test()` dichiara un distruttore esplicito, permettendo così di scrivere del codice di pulitura che verrà eseguito durante la finalizzazione di ogni istanza della classe `Test`. Il Framework .NET traduce implicitamente il distruttore e crea una chiamata al metodo `Finalize()`, strutturando la procedura come segue:

```
protected override void Finalize()
{
    try
    {
        // Codice all'interno del distruttore di classe
    }
    finally
    {
        base.Finalize();
    }
}
```

In sostanza, viene come prima cosa effettuato un tentativo di eseguire il codice atto al rilascio delle risorse esterne contenuto nel distruttore, quindi, attraverso una clausola `finally` (il cui codice viene eseguito solo dopo aver tentato di eseguire il codice nel blocco `try`) viene chiamato il metodo `Finalize()` a segnalare che la memoria dedicata all'oggetto è spazzatura.
La finalizzazione è quindi una tecnica non esplicita per il rilascio delle risorse non gestite. Non potendo controllare quando il Garbage Collector richiama il metodo `Finalize()`, è necessario implementare un distruttore per effettuare un previo rilascio delle risorse esterne.
Concludiamo questa parte sulla finalizzazione con un breve riassunto dei concetti più importanti da tenere presenti quando è necessario implementare un finalizzatore:

- l'override del metodo `Finalize()` non è permesso in C#, il codice per il rilascio delle risorse esterne deve essere implementato nel costrutto del distruttore di classe (al contrario, in VB .NET è possibile eseguire l'override di tale metodo, ed è quindi sufficiente porre il codice per il rilascio delle risorse non gestite al suo interno);
- i finalizzatori dovrebbero rilasciare solamente risorse non gestite;
- mai allocare memoria nei finalizzatori, né chiamare metodi virtuali nel loro codice;
- evitare sincronizzazioni e il sollevamento di eccezioni non gestite nei finalizzatori;
- l'ordine di esecuzione dei finalizzatori è non deterministico, in altre parole non si può contare sull'utilizzo di un altro oggetto all'interno di un finalizzatore;
- non creare distruttori di classe senza codice da eseguire al loro interno;
- il codice di rilascio all'interno dei distruttori dovrebbe essere il più veloce possibile, per evitare un degrado delle prestazioni dell'applicazione.

2.6.8 Il metodo Dispose(): pulizia esplicita delle risorse

A differenza di quanto visto finora per il metodo `Finalize()`, gli sviluppatori dovrebbero effettuare il rilascio delle risorse esterne in maniera esplicita attraverso il metodo `Dispose()`. E' possibile infatti richiamare questo metodo su ogni oggetto che implementi l'interfaccia `IDisposable` per il rilascio di risorse esterne alle quali l'oggetto potrebbe trattenere dei riferimenti, rendendo così il processo di rilascio deterministico. Generalmente, il metodo `Dispose()` non rilascia memoria gestita, e, per essere precisi, non rimuove nemmeno l'oggetto dalla memoria. L'effettiva rimozione è in effetti lasciata (come al solito) al Garbage Collector, che eliminerà l'oggetto solo quando lo riterrà conveniente.
Inoltre, è di importanza fondamentale che lo sviluppatore che implementa un metodo `Dispose()` ricordi di richiamare `GC.SuppressFinalize(this)` per evitare l'esecuzione del finalizzatore.
Un oggetto dovrebbe implementare l'interfaccia `IDisposable` e il suo metodo `Dispose()` non solo quando è necessario rilasciare esplicitamente delle risorse non gestite, ma anche quando lo stesso oggetto crea istanze di altre classi che a loro volta utilizzano queste risorse. L'interfaccia `IDisposable` è pertanto la scelta migliore quando si vuole gestire la pulizia delle risorse da codice anziché delegarla al Garbage Collector. La sua struttura è molto semplice, poiché consiste di un solo metodo senza argomenti, `Dispose()`:

```
public interface IDisposable
{
    void Dispose();
}
```

Il codice che segue mostra come implementare il metodo `Dispose()`::

```
class Test : IDisposable
{
    ~Test()
    {
        Dispose(false);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (disposing)
        {
            // Rilascio delle risorse gestite
        }

       // Rilascio delle risorse non gestite
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
}
```

Nel codice precedente le risorse possono essere liberate in qualsiasi momento richiamando il metodo dispose con la variabile booleana disposing settata a `true`. In questo caso viene eseguito un rilascio di entrambi i tipi di risorse, gestite e non, quindi viene evitato il processo di finalizzazione (non necessario, poiché non è stato richiamato il distruttore della classe) attraverso il comando `GC.SuppressFinalize(this)`. Nel caso in cui,

al termine dell'utilizzo dell'oggetto, ci si dimentichi di liberare le risorse non gestite, entrerebbe invece in gioco il distruttore, richiamando il metodo dispose con il parametro a false, liberando in questo modo le sole risorse non gestite.

2.6.9 La keyword Using

L'istruzione `using` consente al programmatore di specificare quando devono essere rilasciate le risorse utilizzate dagli oggetti, limitandone l'ambito (*scope*). Affinché possa essere utilizzata, l'oggetto fornito come parametro deve implementare l'interfaccia `IDisposable` poiché, come vedremo, il CLR traduce il costrutto in un blocco try-catch nella cui clausola finally è richiamato il metodo `Dispose()` di cui si è parlato nel capitolo precedente. Il tipico utilizzo di `using` è il seguente:

```
using (StreamReader sr = new StreamReader("File.txt"))
{
    String line = sr.ReadToEnd();
}
```

Il codice nell'esempio fa in modo che l'oggetto `sr`, una volta raggiunta la fine dell'istruzione `using`, rilasci la risorsa non gestita utilizzata, il file “Prova.txt”. A tutti gli effetti, il codice all'interno di un'istruzione `using` viene tradotto dal CLR nel modo seguente:

```
StreamReader sr = new StreamReader("File.txt");

try
{
    String line = sr.ReadToEnd();
}
finally
{
    // Controlla se l'oggetto è null
    if (sr != null)
    {
        // Chiama il metodo Dispose() dell'oggetto
        ((IDisposable)sr).Dispose();
    }
}
```

2.6.10 Tipi nello Stack e nell'Heap

Concludiamo questa panoramica sulla gestione della memoria da parte del Framework .NET definendo quali dati sono allocati nello stack e quali nel Managed Heap. Le classi di elementi che prenderemo in considerazione sono tre: i tipi di valore, i tipi di riferimento e i puntatori. I primi due sono già stati introdotti nel paragrafo 2.5, ma senza entrare nei dettagli. In C#, i tipi di riferimento sono i seguenti:

```
class
interface
delegate
object
string
```

Un tipo di riferimento viene sempre allocato nel Managed Heap.
I tipi di valore, invece, sono identificati dalle seguenti keyword:

```
bool
byte
char
decimal
double
enum
float
int
long
sbyte
short
struct
uint
ulong
ushort
```

Il terzo tipo di elementi che possono essere allocati in memoria sono i *riferimenti ad un tipo*, meglio conosciuti come puntatori. Nel codice managed i puntatori non sono utilizzati in maniera esplicita, ma sono gestiti dal Common Language Runtime. Generalmente, quando accediamo ad un tipo di riferimento in memoria, lo facciamo attraverso un puntatore, il quale non è altro che una variabile contenente un indirizzo di memoria o il valore `null`.
Per quanto riguarda il luogo di allocazione di tipi di valore e puntatori, le cose non sono così dirette come abbiamo visto per i tipi di riferimento. Come esempio, consideriamo il seguente metodo:

```
public int Incrementa(int pValue)
{
    int result = pValue + 1;
    return result;
}
```

Quando questo metodo viene richiamato dal nostro codice, il suo parametro (`pValue`) viene salvato nello stack. Successivamente, il controllo viene passato alle istruzioni del metodo e, se è la prima volta che questo viene eseguito, verrà preventivamente effettuata una compilazione JIT. Durante l'esecuzione del metodo, viene istanziata la variabile intera `result`, che sarà allocata nello stack: *i tipi di valore dichiarati nel corpo di un metodo sono infatti sempre allocati nello stack*.
Se, invece, un tipo di valore non è dichiarato all'interno di un metodo, allora deve essere dichiarato all'interno di un tipo di riferimento, e *sarà in questo caso allocato nel Managed Heap insieme al tipo di riferimento che lo contiene*.
Vediamo come esempio la classe (e quindi tipo di riferimento) `MyInt` e il metodo `Incrementa` che, al suo interno, ne crea un'istanza:

```
public class MyInt
{
    public int MyValue;
}

public MyInt Incrementa(int pValue)
{
    MyInt result = new MyInt();
    result.MyValue = pValue + 1;
    return result;
}
```

Anche in questo esempio, quando il metodo `Incrementa` viene richiamato, il parametro `pValue` viene salvato nello stack. Successivamente, essendo `MyInt` un tipo di riferimento,

l'oggetto `result` viene allocato nel Managed Heap (e con lui la variabile intera `MyValue`) e viene referenziato da un puntatore allocato nello stack. La situazione descritta è rappresentata in Fig. 2.17.

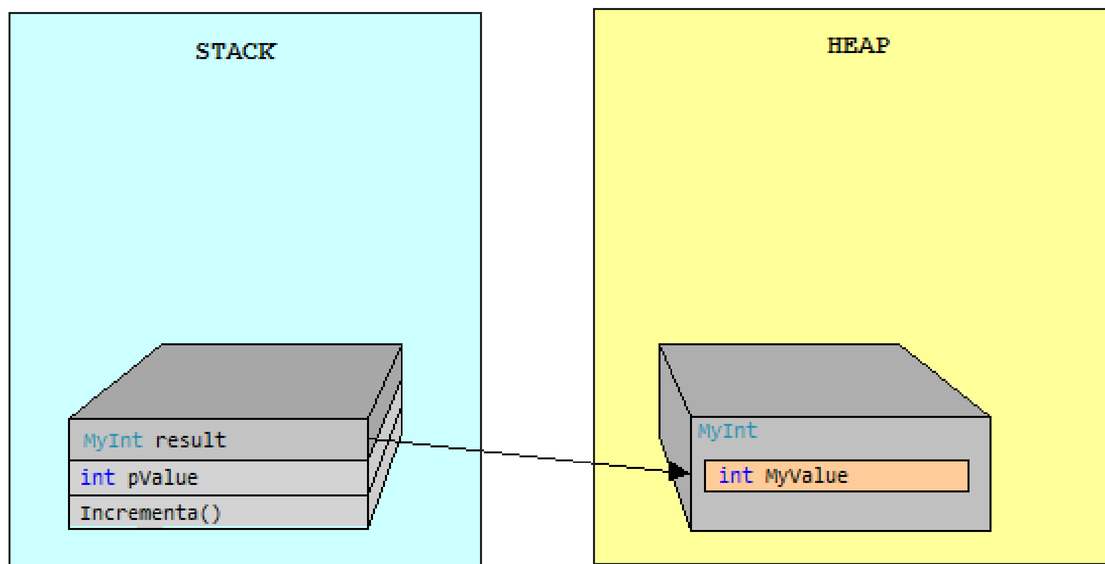

Fig. 2.17 – Referenziamento di un oggetto MyInt attraverso un puntatore

2.7 Windows Communication Foundation (WCF)

Questo paragrafo è dedicato ad un'introduzione alla tecnologia WCF, disponibile dalla versione 3.0 del Framework .NET. WCF è utilizzata dall'architettura YAST solo per alcune transazioni particolari, con lo scopo di ottimizzarne le prestazioni in fase di caricamento (vedremo più avanti come questo sia possibile).
Windows Communication Foundation (WCF) è un sottosistema applicativo proprietario della Microsoft per la compilazione di applicazioni orientate ai servizi. Grazie a WCF è possibile inviare dati come messaggi asincroni da un endpoint del servizio ad un altro. Un endpoint del servizio può appartenere a un servizio disponibile in modo continuo ospitato da IIS (Internet Information Services) *oppure essere un servizio ospitato in un'applicazione*. Un endpoint può essere un client di un servizio che richiede dati da un endpoint del servizio. Il messaggio può essere semplice come una parola o come un singolo carattere inviato in formato XML o complesso come un flusso di dati binari.
Di seguito vengono indicati alcuni scenari di esempio:

- Servizio protetto per elaborare le transazioni aziendali;
- Servizio che fornisce dati correnti ad altri, ad esempio un rapporto sul traffico o un altro servizio di monitoraggio;
- Servizio di chat che consente a due persone di comunicare o di scambiare dati in tempo reale.

Sebbene sia possibile creare questo genere di applicazioni anche con programmi precedenti a WCF, grazie a quest'ultimo lo sviluppo degli endpoint è stato semplificato; infatti, se da un lato ogni protocollo di rete (HTTP, FTP, SMTP, etc..) ha un suo modello di programmazione, e necessita quindi di una conoscenza specifica da parte degli sviluppatori per poter essere utilizzata, WCF è stato realizzato con l'intento di ricondurre ad un unico modello diverse tecnologie, rendendo più semplice ed uniforme

la programmazione in ambiente Windows (attualmente WCF è disponibile per Windows Vista, Windows XP SP2, Windows Server 2003, Windows 7 e Windows 8).

2.7.1 Definizione del sottosistema WCF

Un servizio WCF si basa sugli *endpoint*, che sono le porte attraverso le quali le applicazioni comunicano con il mondo esterno; si può quindi affermare che un servizio WCF sia una collezione di endpoint. A sua volta, un endpoint è costituito da quelli che sono i pilastri di WCF: Address, Binding e Contract.

Address:
L'Address è l'indirizzo al quale il servizio risponde. L'indirizzo è composto da un *URI*, una *Identity* ed una lista di *Header*. Nella fase di definizione di un Address, l'informazione principale è l'URI, che corrisponde all'indirizzo fisico del servizio. Header e Identity sono informazioni necessarie solo in casi particolari. Ad esempio quando vi sono più endpoint, può essere utile avere diversi Header a seconda dell'endpoint che il client utilizza. In parole semplici si può definire l'address come il *DOVE*.

Binding:
Gran parte della soluzione proposta da WCF sta nel concetto di Binding. Infatti, se ci si può occupare del codice senza preoccuparsi dell'infrastruttura di trasporto, lo si deve soprattutto a questa feature. I Binding si occupano di quello che avviene tra il momento in cui il servizio spedisce logicamente il messaggio ed il momento in cui viene fisicamente trasmesso in rete. In questo lasso di tempo vengono eseguiti numerosi passi che seguono una precisa *pipeline* di cui i Binding sono responsabili. Durante l'esecuzione della pipeline il messaggio deve attraversare due livelli:

- il primo si occupa del *Behaviour* (comportamento), ovvero delle trasformazioni che deve subire un messaggio;
- il secondo si occupa del *Channel* (canale), ovvero dell'instradamento verso il canale fisico di trasporto.

Nel primo livello ci si occupa della conversione dei dati dal formato “codice” al formato messaggio; ad esempio, vengono trasformate le informazioni da una classe al formato XML di un messaggio SOAP. Inoltre, i Behaviour si occupano anche della sicurezza, della criptazione delle informazioni e di tutte le funzioni di gestione del dato. Durante la seconda fase, il messaggio viene introdotto nel canale di trasporto secondo quanto specificato in fase di configurazione. E' in questa fase che si istanzia il canale del protocollo originale su cui viaggeranno le informazioni. Dal momento che a questo livello può essere necessario operare su dettagli specifici del protocollo utilizzato, è proprio qui che possono essere aggiunte informazioni sulla modalità di trasmissione. Come detto in precedenza, questo processo avviene per mezzo di una pipeline: possiamo quindi vedere tutte le opzioni finora illustrate (protocollo, formattazione, etc..) come moduli da inserire nel flusso di elaborazione del messaggio. La gestione dei Binding può essere interamente gestita in fase di configurazione e, semplicemente cambiando poche voci, si può passare dalla pubblicazione in modalità Web su HTTPS criptato con un certificato digitale ad un trasporto diverso del messaggio come SMTPS (SMTP *Secured*) senza dover modificare codice. Se si volesse dare una parola chiave ai binding questa sarebbe *COME*.

Contract:
I *Contract* (o contratti) rappresentano l'interfaccia software vera e propria, ovvero le API che il servizio pubblica. I Contract rappresentano il *COSA*.

2.7.2 Esempio di creazione di un servizio con WCF

L'esempio che segue mostra come sia possibile creare un semplice servizio con WCF, strutturando la fase di creazione in quattro blocchi, definiti nei punti seguenti:

- 1. Definizione del servizio;
- 2. Implementazione del servizio;
- 3. Hosting del servizio, implementazione del server;
- 4. Avvio del server, implementazione del client.

Il servizio consiste in un singolo metodo, `Ping(string name)`, che accetta come parametro un nome e risponde con una stringa di saluto. Ad esempio, passando in input il nome “Matteo”, verrà restituito “Ciao, Matteo”.
Definire in servizio WCF significa definire le sue funzionalità in un *contratto di servizio,* rappresentato nel nostro caso dall'interfaccia `IService`:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ServiceModel;

namespace WCFSimple.Contract
{
    [ServiceContract]
    public interface IService
    {
        [OperationContract]
        string Ping(string name);
    }
}
```

L'attributo `OperationContract` definisce che il metodo `Ping` è parte di un contratto di servizio.
Una volta definito il servizio, questo può essere implementato. A questo scopo è necessario definire una classe che implementi l'interfaccia del servizio. L'attributo `ServiceBehavior`, in questo caso, chiede al framework WCF di creare una nuova istanza della classe del servizio per ogni chiamata effettuata ai metodi del servizio.

```
namespace WCFSimple.Server
{
    [ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
    class ServiceImplementation : WCFSimple.Contract.IService
    {
        #region IService Members

        public string Ping(string name)
        {
            Console.WriteLine("SERVER - Processing Ping('{0}')", name);
            return "Ciao, " + name;
        }

        #endregion
```

```
    }
}
```

Per effettuare l'hosting del servizio, il metodo `Main()` crea un oggetto di tipo `ServiceHost` per gestire ogni istanza dell'implementazione del servizio e la sua pubblicazione su qualsiasi numero di endpoint (nel nostro esempio, il servizio sarà disponibile su un solo endpoint). Il metodo `AddServiceEndpoint` accetta come primo argomento il contratto del servizio, dal momento che la classe passata in precedenza all'oggetto `ServiceHost` può potenzialmente implementare più di un servizio. Il secondo parametro specifica il protocollo da utilizzare per la comunicazione: nel nostro caso viene utilizzato TCP. Il terzo identifica infine l'IP e la porta dell'endpoint (nell'esempio viene utilizzata l'interfaccia di loopback e la porta 8000). Successivamente, attraverso il metodo `Open()`, il servizio viene avviato. Essendo il programma pensato per essere eseguito su un thread separato, è stata introdotta la variabile `stopFlag` per permettere di terminarne l'esecuzione.

```
public class Program
{
    private static System.Threading.AutoResetEvent stopFlag = new
    System.Threading.AutoResetEvent(false);

    public static void Main()
    {
        ServiceHost svh = new ServiceHost(typeof(ServiceImplementation));
        svh.AddServiceEndpoint( // Binding
        typeof(WCFSimple.Contract.IService),
        new NetTcpBinding(),
        "net.tcp://localhost:8000");
        svh.Open();
        Console.WriteLine("SERVER - Avvio...");
        stopFlag.WaitOne();

        Console.WriteLine("SERVER - Terminando...");
        svh.Close();

        Console.WriteLine("SERVER – Terminato.");
    }

    public static void Stop()
    {
        stopFlag.Set();
    }
}
```

Il client, in questo esempio, svolge due operazioni: avvia il server e lo chiama. Viene istanziato un oggetto `ChannelFactory` per la creazione di un canale di comunicazione tra client e server. `ChannelFactory` può creare un canale per uno specifico contratto di servizio, binding e address. Il resto del codice è la logica di frontend del client: un loop che chiede di inserire un nome, lo passa al servizio e mostra il saluto ritornato.

```
public class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Esempio WCF");

        // Avvia il server
        System.Threading.Thread thServer = new
        System.Threading.Thread(WCFSimple.Server.Program.Main);

        thServer.IsBackground = true;
```

```
        thServer.Start();
        System.Threading.Thread.Sleep(1000); // Attendi che il server sia attivo

        // Avvia il client
        ChannelFactory<WCFSimple.Contract.IService> scf;
        scf = new ChannelFactory<WCFSimple.Contract.IService>(
                                                           new NetTcpBinding(),
                                                           "net.tcp://localhost:8000");

        WCFSimple.Contract.IService s;
        s = scf.CreateChannel();

        while(true)
        {
            Console.Write("CLIENT - Nome: ");
            string name = Console.ReadLine();
            if (name == string.Empty)
            {
                break;
            }

            string response = s.Ping(name);
            Console.WriteLine("CLIENT – Risposta dal servizio: " + response);
        }

        (s as ICommunicationObject).Close();

        // Termina il server
        WCFSimple.Server.Program.Stop();
        thServer.Join();
    }
}
```

3 Overview di YAST

Con questo capitolo inizia la trattazione dell'architettura YAST. Lo scopo del capitolo è di chiarire quale sia l'ambiente all'interno del quale YAST opera e di come l'architettura sia strutturata.

3.1 Introduzione

L'ambiente denominato Sportello è, ad oggi, composto da una serie di transazioni sviluppate in linguaggio VB6 che possono essere invocate direttamente attraverso un componente detto Portale o da una pagina web, oppure indirettamente da altre transazioni. Fino ad ora, il Portale ha svolto il ruolo di padre di tutte le transazioni lanciate, le quali, essendo eseguibili ActiveX, necessitano di un parent. Ognuna di queste funzioni di Sportello è quindi un processo distinto che viene eseguito sul client e che si appoggia a DLL comuni. Con la nuova infrastruttura YAST, i cui componenti principali sono evidenziati in rosso nello schema in Fig. 3.1, ci si è posti l'obiettivo di rinnovare la tecnologia delle transazioni bancarie portandole in ambiente managed .NET. Attualmente, per l'operatività di Sportello, il cliente fornisce al personale di filiale delle workstation con Windows XP o Windows 7.
Come è possibile vedere nello schema, l'inglobamento dell'architettura nell'ambiente di Sportello prevede il riutilizzo da parte di quest'ultima di molti componenti infrastrutturali tuttora in uso.

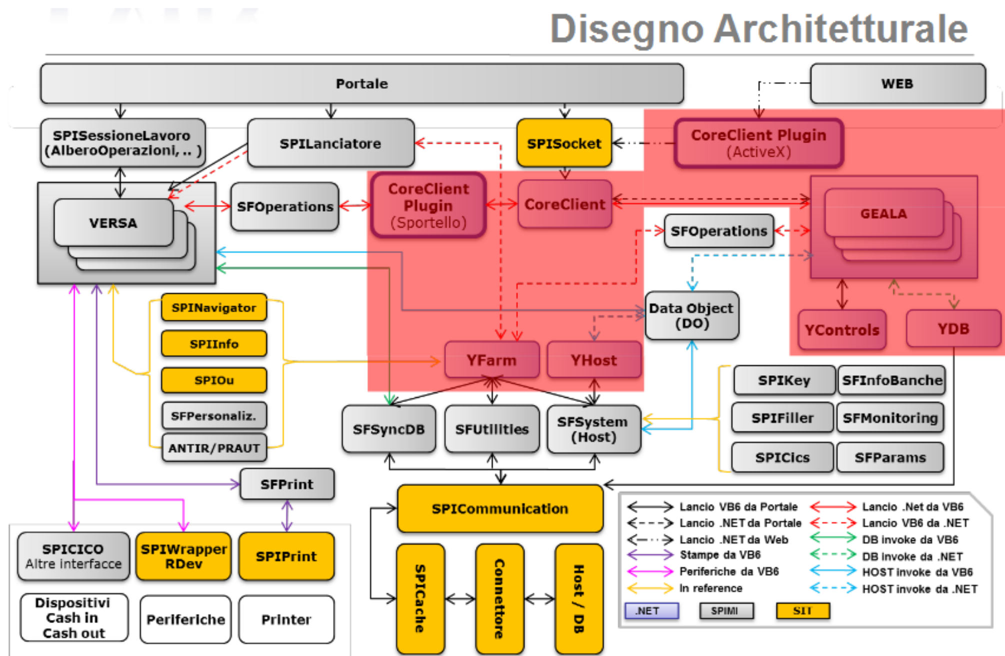

Fig. 3.1 – Ambito di YAST nell'ambiente di Sportello

3.2 Schema dell'architettura

Lo schema dell'architettura in Fig. 3.2 mostra l'idea di fondo su cui si basa l'architettura. Il Portale è il componente da dove è possibile avviare transazioni VB6/COM e, grazie alla mediazione del CoreClient, adesso anche transazioni .NET. Lo scopo del CoreClient è appunto quello di gestire le richieste di lancio di una transazione .NET a fronte di una richiesta del Portale (Fig. 3.3) o del CoreClientPlugin. Quest'ultimo, invece, è un ActiveX che permette il lancio di transazioni .NET in seguito a comandi compiuti all'interno di pagine web. Come è possibile vedere, il CoreClient è il nucleo principale dell'architettura .NET.

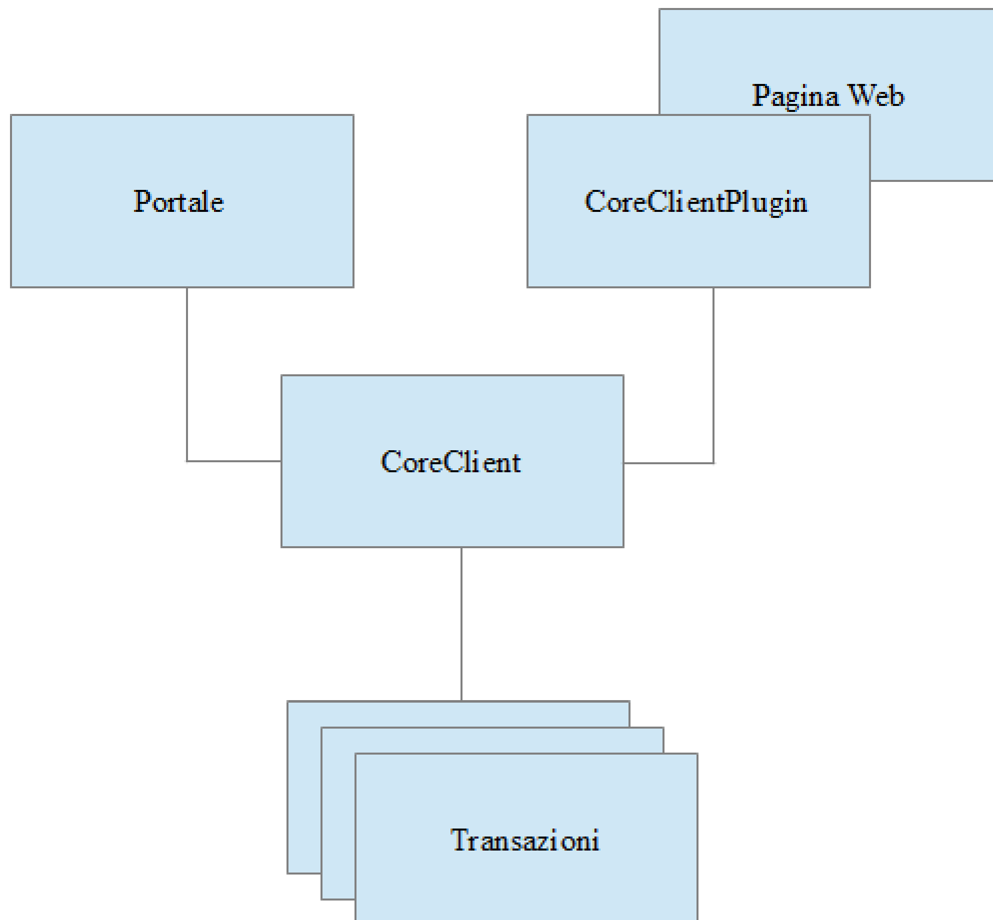

Fig. 3.2 – Schema dell'architettura

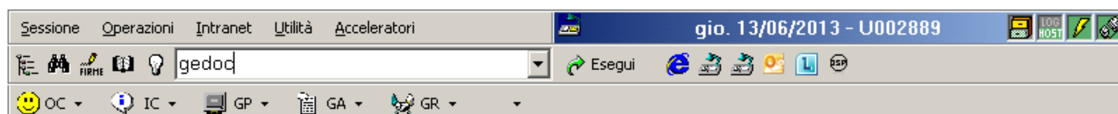

Fig. 3.3 – Il Portale

4 Struttura di una transazione .NET

Il capitolo descrive la struttura delle nuove transazioni .NET modellate attraverso il pattern Model-View-Controller, la gestione del flusso di esecuzione attraverso degli step di navigazione atomici e il funzionamento delle chiamate Nested (nidificate) tra transazioni.

4.1 Il pattern Model-View-Controller

Fino ad ora abbiamo parlato di transazioni distinguendole tra VB6/COM e .NET. Le transazioni VB6 sono state sviluppate prima dell'avvento di YAST e, benché siano supportate dalla nuova architettura, un approfondimento sulla loro struttura va oltre lo scopo di questa documentazione.
Le transazioni .NET soddisfano il pattern architetturale denominato *Model-View-Controller* (MVC). Si tratta di un pattern molto diffuso nello sviluppo di interfacce grafiche di sistemi software Object-Oriented che si basa sulla separazione dei compiti tra i componenti software che interpretano i tre ruoli principali (Fig. 4.1):

- MODEL: componente incaricato di fornire i metodi per accedere ai dati utili dell'applicazione;
- VIEW: componente incaricato della visualizzazione dei dati contenuti nel Model e della logica di interazione con l'utente;
- CONTROLLER: componente incaricato della gestione e dell'attuazione dei comandi ricevuti dall'utente attraverso il View e del conseguente cambio di stato degli altri due componenti.

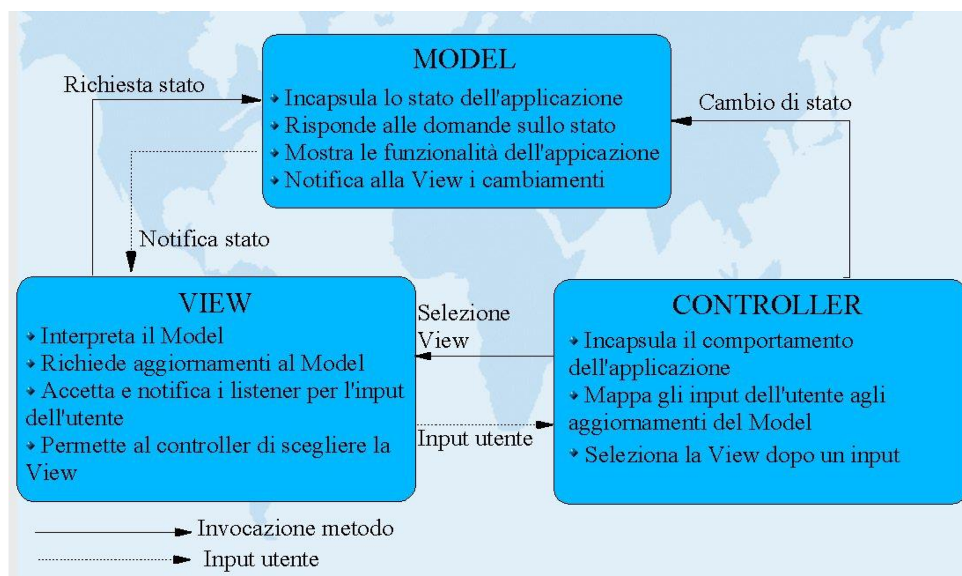

Fig. 4.1 – Pattern architetturale Model-View-Controller (MVC)

Questo schema implica anche la tradizionale separazione fra la logica applicativa, chiamata anche *logica di business*, a carico del Controller e del Model, e l'interfaccia utente a carico del View. Ogni transazione sviluppata sul Framework YAST incapsula questo pattern architetturale attraverso la gestione di 4 progetti (Fig. 4.2):

- Controller: è l'entry-point della transazione; è un eseguibile destinato alla comunicazione con il CoreClient;
- Datamodel: è una DLL che costituisce il modello dati della transazione e che imposta il binding dei dati sulle mappe;
- Presentation: è un eseguibile che costituisce l'insieme della mappe WPF della transazione, richiamabile esclusivamente dal Controller;
- Service: libreria dei servizi (logica di business, DB, Host, webservices) utilizzati dalla transazione.

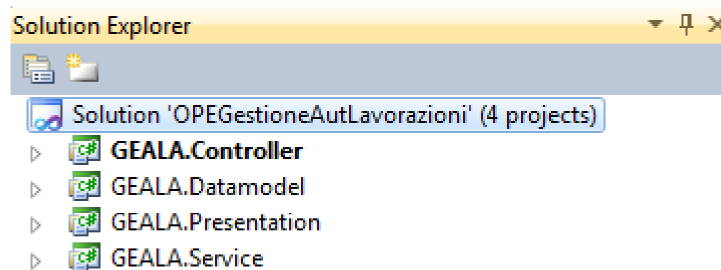

Fig. 4.2 – Struttura di una transazione

4.2 Gestione del flusso di navigazione

Il flusso di navigazione tra le finestre (dette *mappe*) delle transazioni e il lancio dei servizi che le compongono, si riconduce ad una sequenza di *step di navigazione* innescati da delle *actions*. Le actions sono fondamentalmente tutte le operazioni che l'utente può compiere su una mappa di transazione, che nella maggior parte dei casi si riconducono a click su un bottone. La logica di navigazione è implementata nelle due classi `YManager` e `YNavigation`, che vedremo più nel dettaglio nei paragrafi successivi. Ogni step di navigazione deve implementare necessariamente una delle seguenti classi astratte:

- `NavStart`: step che identifica il nodo di partenza del flusso di navigazione;
- `NavMap`: step che identifica un nodo di navigazione di tipo Mappa (Presentation);
- `NavService`: step che identifica un nodo contenente della logica di servizio (chiamate ad Host, a WebServices, a DB, logica applicativa, etc..).
- `NavNested`: step che identifica il lancio di una transazione (.NET o COM) in modalità Nested;
- `NavExcel`: step che lancia una esportazione dei dati in formato Excel;
- `NavEnd`: step che identifica il nodo di conclusione del flusso di navigazione.

A loro volta, queste classi sono un'implementazione della classe astratta `NavStep`, che rappresenta un oggetto generico di navigazione:

```
public abstract class NavStep
{
    // enum di tutti i possibili step di navigazione
```

```
    public enum StepType
    {
        Start,
        End,
        Map,
        Service,
        Excel,
        Nested
    }

    // properties
    public abstract StepType Type { get; }
    public IDatamodel ModelloDati { get; set; }

    // metodi
    public abstract NavStep Next();
}
```

Il diagramma riportato in Fig. 4.3 rappresenta un semplice flusso di navigazione. All'avvio viene mostrata la mappa principale, chiamata MAPPA 1: questa mappa rende disponibili 4 *actions* che possono essere intraprese, ognuna delle quali porta ad un altro step di navigazione:

- MostraDettagli: blocca l'esecuzione della MAPPA 1 e mostra MAPPA 2;
- RicercaAnagrafica: chiama la transazione IDENT in modalità nested;
- VerificaSaldo: invoca la BS (Business Service) di verifica saldo;
- Annulla: esce dalla transazione.

Per quanto riguarda la MAPPA 2, si può uscire da essa attraverso l'*action* Conferma.

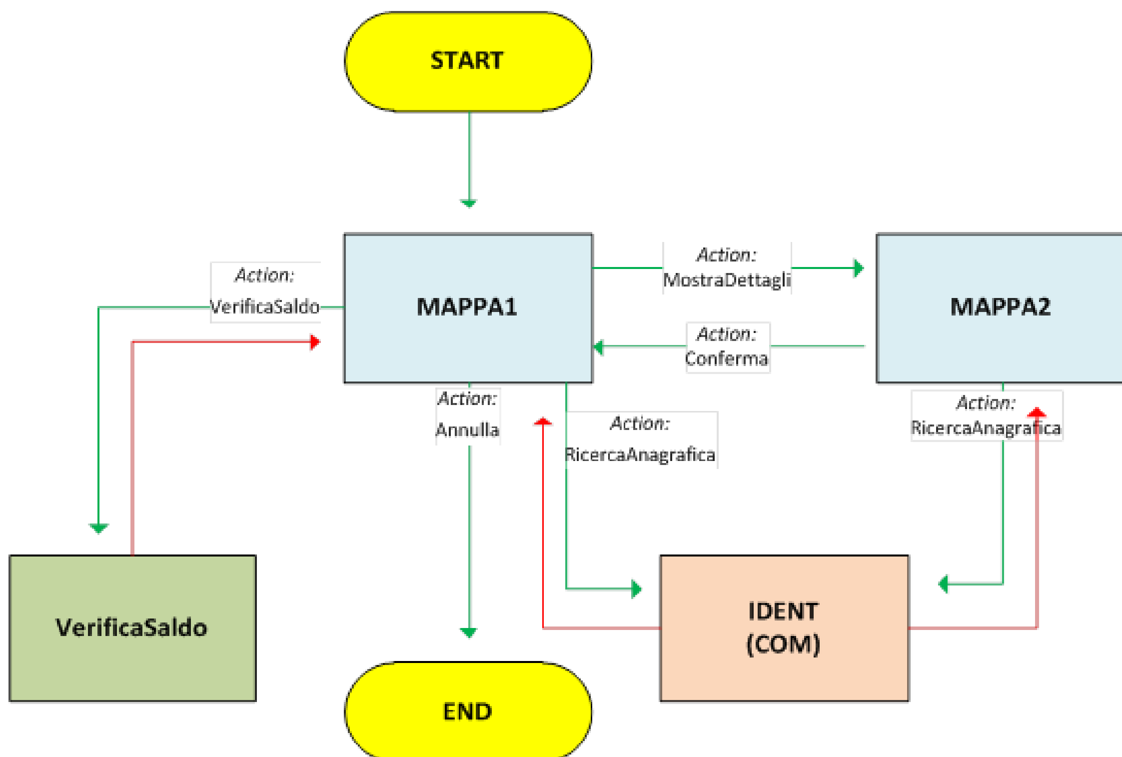

Fig. 4.3 – Schema di un semplice flusso di navigazione

L'implementazione di questo workflow comporta:

1. la creazione di due classi *Window* nel Presentation della transazione, una corrispondente alla MAPPA 1 e l'altra alla MAPPA 2 (ad esempio *Window1.xaml* e *Window2.xaml*);

2. il censimento delle due mappe nella classe del Presentation denominata `MapManager`, che estende `YManager`:

```
public enum Maps
{
    [Mappa(typeof(Window1))]
    [Titolo("ACRON", "Mappa 1")] // facoltativo
    MAPPA1,

    [Mappa(typeof(Window2))]
    [Titolo("ACRON", "Mappa 2")] // facoltativo
    MAPPA2
}
```

   il censimento viene effettuato inserendo una nuova voce nell'enum `Maps`. Ogni voce identifica (attraverso l'attributo `Mappa`) e qualifica (attributo `Titolo`) una mappa del Presentation. Il tipo `Mappa` corrisponde al nome del file xaml della classe che implementa la mappa, mentre `Titolo`, se presente, comporta l'override dei dati di intestazione della mappa, che di default vengono recuperati dall'albero operazioni;

3. il censimento delle *actions* che è possibile intraprendere in queste due mappe nell'enum `Actions`:

```
public enum Actions
{
    [ConnectedButton("btnDettagli")]
    MostraDettagli,

    [ConnectedButton("btnRicAnaMappa1")]
    [ConnectedButton("btnRicAnaMappa2")]
    RicercaAnagrafica,

    [ConnectedButton("btnConferma")]
    Conferma,

    [ConnectedButton("btnAnnulla")]
    Annulla,

    [ConnectedButton("btnVerificaSaldo")]
    VerificaSaldo
}
```

   come è possibile vedere, per una singola action può essere eseguito il bind a più controlli (in questo caso bottoni), anche di mappe diverse;

4. l'inoltro degli eventi *ButtonClick* delle due mappe al gestore centralizzato `ButtonClickManager` nelle classi che implementano la logica delle mappe:

```
{MAPPA 1}
btnDettagli.Click += new RoutedEventHandler(ButtonClickManager);
btnRicAnaMap1.Click += new RoutedEventHandler(ButtonClickManager);
btnAnnulla.Click += new RoutedEventHandler(ButtonClickManager);
btnVerificaSaldo.Click += new RoutedEventHandler(ButtonClickManager);

{MAPPA2}
```

```
btnConferma.Click += new RoutedEventHandler(ButtonClickManager);
btnRicAnaMap2.Click += new RoutedEventHandler(ButtonClickManager);

private void ButtonClickManager(object sender, RoutedEventArgs e)
{
    Cursor = Cursors.AppStarting;
    MapManager.EventForward((Maps)this.GetType().getConnectedMap(typeof(Maps)),
                                         ((YControls.YButton)sender).Name);
    Cursor = Cursors.Arrow;
}
```

il metodo ButtonClickManager non è centralizzabile, e deve essere quindi presente in ognuna delle classi delle mappe utilizzate, per essere utilizzato come gestore dell'evento Click sui bottoni delle mappe. Il suo scopo è quello di inviare l'evento scatenato dal click su un bottone al gestore centralizzato presente nella classe Navigation del Controller attraverso il metodo EventForward.

4.3 YNavigation

La classe YNavigation implementa il motore di navigazione di YAST attraverso il metodo goNextStep:

```
// *** motore di navigazione ***

public void goNextStep(NavStep step)
{
    switch (step.Type)
    {
        case NavStep.StepType.Start:
        goNextStep(step.Next());
        break;

        case NavStep.StepType.End:
        // forzo chiusura di tutte le mappe rimaste
        // aperte (magari per eccezione non gestita)
        // di una data transazione
        presentation.EndAllMaps(((NavEnd)step).ModelloDati.OPF.CodiceOPF);
        ((NavEnd)step).DispatchFinalization();
        break;

        case NavStep.StepType.Map:
        presentation.Show(((NavMap)step).FormObject);
        break;

        case NavStep.StepType.Service:
        // chiama il servizio definito nello step
        ((NavService)step).ServiceObject();
        // continua con lo step successivo
        goNextStep(step.Next());
        break;

        case NavStep.StepType.Nested:
        // lancia l'operazione nested dichiarata
        ((NavNested)step).LaunchNested();
        break;

        case NavStep.StepType.Excel:
        // lancia una esportazione in Excel
        ((NavExcel)step).ExportExcel();
        break;
```

```
        default:
        // non si dovrebbe mai arrivare qui
        // invoco il metodo Log della classe Logger
        break;
        }
}
```

Il metodo `goNextStep` accetta come parametro un oggetto di tipo `NavStep`. La logica è semplice: il metodo `goNextStep` è invocato nei blocchi case delle varie actions gestite dal `Navigation`, per passare allo step successivo. L'azione è compiuta mediante un blocco switch-case sul tipo di step di navigazione (*Start*, *End*, *Nested*, *Service*, …) che invoca un metodo specifico del tipo di step. Ad esempio, se lo `StepType` è di tipo `Nested`, allora si invoca il metodo `LaunchNested`, implementato nella classe di navigazione del Controller della transazione. La classe `YNavigation` espone anche un metodo per l'inizializzazione del motore di navigazione. Si tratta del metodo astratto `Start`, il cui override deve essere effettuato nel Controller della transazione. All'interno di questo metodo viene inizializzato il Datamodel, viene effettuata la sottoscrizione agli eventi sul Presentation ed infine viene dato il via al flusso di navigazione richiamando il nodo di navigazione `start`:

```
public override void Start()
{
    // Inizializzazione gestore mappe Presentation
    // e sottoscrizione eventi
    presentation = new MapManager();
    presentation.DataModel = CurrentDataModel;

    MapManager.onPresentationAction += new
    MapManager.PresentationActionEventHandler(MapManager_onPresentationAction);

    // start navigation
    goNextStep(new start((Datamodel)CurrentDataModel));
    // Destroy gestore mappe presentation e de-sottoscrizione eventi
    MapManager.onPresentationAction -= new
    MapManager.PresentationActionEventHandler(MapManager_onPresentationAction);
    presentation.DataModel = null;
    presentation = null;
}
```

4.4 YManager

La classe `YManager` implementa tutta la logica riguardante il ciclo di vita delle mappe, esponendo dei metodi che permettono di agire su di esse. I metodi che svolgono i ruoli più importanti sono i seguenti:

`public void Show(object map)`:

il metodo `Show` controlla se la mappa passata come parametro è presente nel dizionario `activeMaps`, che tiene traccia delle mappe già create/mostrate. Se è presente in tale dizionario non occorre creare una nuova istanza della mappa, ma è sufficiente richiamare su di essa il metodo `UnFreezeMap` passando come parametro lo stesso ricevuto in input da `Show`. Nel caso in cui, invece, la mappa che è richiesto mostrare non sia presente nel dizionario, viene lanciato il metodo privato `ShowSingleMap` (passando sempre in input lo stesso parametro di `Show`) utilizzando il metodo `Invoke` della classe `Dispatcher` del framework .NET.

```
public void Show(object map)
{
    // la mappa era già stata creata/mostrata?
    if (activeMaps.ContainsKey(map))
    {
        // Sblocca la mappa
        UnFreezeMap(map);
     }
     else
     {
        // Page o Window?
        Type mapType = getMapType(map);
        if (mapType.BaseType == typeof(Page))
        {
           // La mappa è situata all'interno del Browser
           // Non è ancora stato implementato
           MessageBox.Show(string.Format("Si è cercato di visualizzare una mappa in
           modalità PAGE. L'unica modalità supportata è
           WINDOW.", Environment.NewLine),
           "YAST - Errore",
           MessageBoxButton.OK,
           MessageBoxImage.Error);
        }

        if (mapType.BaseType == typeof(Window) || getMapType(map).BaseType == typeof(YWindow))
        {
           Dispatcher.CurrentDispatcher.Invoke(new Action(() => { ShowSingleMap(map); }));
        }
    }
}
```

private void ShowSingleMap(object startingMap):

il metodo `ShowSingleMap` valorizza il titolo della mappa e crea l'oggetto Window (WPF) descritto dal parametro `startingMap`. Questo parametro rappresenta la voce corrispondente alla mappa censita in precedenza nell'enum `Maps`.
Il titolo della mappa, per default, è della forma 'AcronimoTRX – DescrizioneTRX' (ricavato da inquiry sull'albero operazioni), a meno che non sia stato specificato diversamente nel campo apposito nell'enum `Maps`. Un'altra operazione svolta dal metodo è quella di gestire la possibilità di ridurre ad icona la finestra: questo è possibile solo se la mappa è l'unica ad essere presente nel dizionario `activeMaps`, quindi la mappa principale. La finestra viene successivamente attivata attraverso Reflection con il richiamo del metodo `InvokeMember("ShowDialog",…)` sull'oggetto di tipo `Type` che identifica la mappa. E' da notare che ogni mappa è di un determinato `Type` definito dallo sviluppatore, che estende il controllo `YWindow` il quale è, a sua volta, derivato dalla classe nativa `Window`, che possiede il metodo `ShowDialog` passato come parametro al metodo `InvokeMember`. La classe che descrive il tipo di una mappa è `MappaAttribute`.

```
public class MappaAttribute : Attribute
{
    public Type map { get; protected set; }

    // Costruttore con 1 parametro
    public MappaAttribute(Type value)
    {
        map = value;
    }
}
```

```
private void ShowSingleMap(object startingMap)
{
    try
    {
        // type per l'attivazione via Reflection della mappa
        Type t = null;

        // recupero i campi (fields) dell'oggetto startingMap
        FieldInfo fieldInfo = startingMap.GetType().GetField(startingMap.ToString());

        // 1. recupera l'attributo "Mappa" collegato a startingMap
        // ritornandone il type per l'attivazione
        MappaAttribute[] ma = fieldInfo.GetCustomAttributes(typeof(MappaAttribute), false)
                                                                as MappaAttribute[];

        t = (ma.Length > 0 ? ma[0].map : null);

        // 2. recupera l'attributo "Titolo" collegato a startingMap
        // ritornandone il type per l'attivazione
        TitoloAttribute[] ta = fieldInfo.GetCustomAttributes(typeof(TitoloAttribute), false)
                                                                as TitoloAttribute[];
        TitoloAttribute titolo = (ta.Length > 0 ? ta[0] : null);

        // creo istanza dell'oggetto mappa (Presentation) passando il Datamodel
        object tInstance = Activator.CreateInstance(t, new object[] { DataModel });

        // imposta il titolo della finestra: default: acronimo - descrizione
        // Se MapAcronimo e MapDescrizione sono valorizzati, allora uso questi
        // valori per il titolo

        string windowTitle = string.Format("{0} - {1}",
                            DataModel.OPF.CodiceOPF.Trim().ToLower(),
                            DataModel.OPF.Descrizione.Trim().ToLower());
        if (titolo != null)
        {
            windowTitle = string.Format("{0} - {1}",
            titolo.Acronimo.Trim().ToLower(),
            titolo.Descrizione.Trim().ToLower());
        }

        PropertyInfo pi = t.GetProperty("Title");
        pi.SetValue(tInstance, windowTitle, null);

        // devo valorizzare la proprietà ResizeMode DINAMICAMENTE:
        //  - se prima mappa => ResizeMode = CanMinimize
        //  - tutte le altre => ResizeMode = NoResize
        PropertyInfo rm = t.GetProperty("ResizeMode");
        rm.SetValue(tInstance, (activeMaps.Count == 0 ? ResizeMode.CanMinimize :
                                                 ResizeMode.NoResize), null);

        // aggiungo la mappa alla lista delle mappe attive
        activeMaps.Add(startingMap, (Window)tInstance);

        // lancio mappa
        t.InvokeMember("ShowDialog", BindingFlags.InvokeMethod | BindingFlags.Instance |
        BindingFlags.Public, null, tInstance, null);

        // distruggo l'handle
        if (activeMaps.ContainsKey(startingMap))
        {
            activeMaps.Remove(startingMap);
        }
    }
    catch (Exception ex)
    {
        Logger.Log(Assembly.GetExecutingAssembly(), TraceEventType.Error,
```

```
                                                "ShowSingleMap()", ex);
    }
}
```

public void Close(object map):

il metodo Close effettua la chiusura della mappa specificata dall'oggetto passato come parametro richiamando il metodo Close dell'oggetto Window (ottenuto dal dizionario activeMaps) attraverso il metodo BeginInvoke del Dispatcher.

```
public void Close(object map)
{
    if (getMapType(map).BaseType == typeof(Window) ||
               getMapType(map).BaseType == typeof(YWindow))
    {
        Window w = (Window)activeMaps[map];
        // chiudo la mappa
        w.Dispatcher.BeginInvoke(new Action(() => { w.Close(); }));
    }
}
```

public static void showWaitPanel():

il metodo showWaitPanel configura e avvia un processo denominato YWaitPanel. Si tratta di un pop-up di attesa contenente una gif animata (Fig. 4.4). Il motivo per cui si è preferito utilizzare un processo indipendente per la visualizzazione del pannello di attesa riguarda l'interoperabilità tra le nuove transazioni .NET e quelle più vecchie in VB6. In sostanza, essendo state alcune di queste ultime modificate per avere la possibilità di lanciare transazioni .NET (e viceversa), e dovendo la grafica del pannello rimanere inalterata, si è optato per questa soluzione. Il processo adotta la mutua esclusione per evitare che più YWaitPanel siano attivi contemporaneamente.

Fig. 4.4 – Il pannello di attesa di YWaitPanel

```
public static void showWaitPanel(double parentHeight, double parentWidth)
{
    Process wPanel = new Process();

    // Configurazione Processo
    wPanel.StartInfo.CreateNoWindow = true;
    wPanel.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
    wPanel.StartInfo.UseShellExecute = false;
    wPanel.StartInfo.FileName = @"C:\Programmi\XXXXX\YAST\YWaitPanel.exe";
    wPanel.StartInfo.Arguments = string.Format("/W:{0} /H:{1}",
                                     parentWidth.ToString(),
                                   parentHeight.ToString());
    // Esecuzione processo
    wPanel.Start();
}
```

public static void hideWaitPanel():

Il metodo hideWaitPanel() è il duale di showWaitPanel(): dopo aver cercato l'istanza di YWaitPanel attiva tra i processi in esecuzione, lo termina. Importante è l'utilizzo che viene fatto dell'istruzione lock sulla variabile statica di tipo object locker: la zona compresa tra le parentesi graffe dell'istruzione lock contrassegna una sezione critica del codice, che impedisce ad un thread di accedervi mentre un altro thread è già all'interno della sezione. Per ognuno dei processi di YWaitPanel in esecuzione viene prima di tutto controllata la proprietà HasExited, che segnala se il processo è già stato terminato: se non è così viene terminato. A questo punto, volendo fare in modo che la terminazione di YWaitPanel sia sincronizzata con la prosecuzione della transazione, è stato introdotto un ciclo while che, dopo un'attesa di 500 millisecondi ricontrolla (fino ad un massimo di 7 volte consecutive) la proprietà HasExited del processo al quale è appena stata inviata la richiesta di terminazione. Dopo il settimo tentativo, se il processo non è ancora stato terminato, il metodo getta la spugna e l'esecuzione del programma prosegue.

```
public static void hideWaitPanel()
{
    int MaxRetry = 6;
    int retryDelay = 500;
    int retryCounter = 0;

    try
    {
        lock (locker)
        {
            foreach (Process pWait in Process.GetProcessesByName("YWaitPanel"))
            {
                // Per evitare che sia in chiusura da un altro processo
                if(!pWait.HasExited)
                {
                    pWait.Kill();
                    while (!pWait.HasExited && retryCounter <= MaxRetry)
                    {
                        // attende 500 ms prima di riprovare
                        Thread.Sleep(retryDelay);
                        retryCounter++;
                    }
                }
            }
        }
    }
    catch (Exception ex)
    {
        Logger.Log(Assembly.GetExecutingAssembly(), TraceEventType.Warning,
                                                    "hideWaitPanel()", ex);
    }
}
```

private static Type getMapType(object map):

il metodo statico getMapType restituisce il tipo dell'oggetto generico mappa passato come parametro. Il recupero del tipo è effettuato leggendo l'attributo Mappa della voce corrispondente nell'enum Maps della classe MapManager, i cui elementi ricordiamo essere strutturati in questo modo:

```
[Mappa(typeof(Window1))]
[Titolo("ACRON", "Mappa 1")] // facoltativo
```

```
MAPPA1,
…
```

In caso di fallimento durante il recupero del tipo, l'eccezione viene loggata e il tipo restituito sarà impostato a `null`.

```
private static Type getMapType(object map)
{
    Type t = null;

    try
    {
       // Recupero i campi (fields) dell'oggetto startingMap
       FieldInfo fieldInfo = map.GetType().GetField(map.ToString());

       // Recupera l'attributo "Mappa" collegato a startingMap
       // ritornandone il type per l'attivazione
       MappaAttribute[] ma = fieldInfo.GetCustomAttributes(typeof(MappaAttribute),
                                                  false) as MappaAttribute[];
       t = (ma.Length > 0 ? ma[0].map : null);
    }
    catch(Exception exception)
    {
        t = null;
        Logger.Log(Assembly.GetExecutingAssembly(), TraceEventType.Warning,
                                              "getMapType()", exception);
    }

    return t;
}
```

public void EndAllMaps(string opfName):

il metodo `EndAllMaps` effettua la chiusura di tutte le mappe attive (presenti quindi nel dizionario `activeMaps`) della transazione specificata dal parametro in ingresso, nello stesso modo visto per il metodo `Close`.

```
public void EndAllMaps(string opfName)
{
    foreach (KeyValuePair<object, DependencyObject> map in activeMaps)
    {
    try
    {
        if (((DependencyObject)map.Value).DependencyObjectType.SystemType.
                               FullName.Contains(opfName.ToUpper()))
        {
            Window w = ((Window)map.Value);
            w.Dispatcher.BeginInvoke(new Action(() => { w.Close(); }));
        }
    }
    catch (Exception ex)
    {
         Logger.Log(Assembly.GetExecutingAssembly(), TraceEventType.Error,
                                               "EndAllMaps()", ex); }
    }
}
```

public void FreezeMap(object map):

il metodo FreezeMap esegue la ricerca della mappa specificata dal parametro in input nel dizionario delle mappe attive (activeMaps), quindi la disabilita impostando le proprietà ShowInTaskbar e IsEnabled a false.

```
public void FreezeMap(object map)
{
    if (getMapType(map).BaseType == typeof(Page))
    {
        // blocco mappa e cambio cursore in "attesa"
        Page w = (Page)activeMaps[map];
        w.Dispatcher.BeginInvoke(new Action(() =>
                                 {
                                     w.Cursor = Cursors.AppStarting;
                                     w.IsEnabled = false;
                                  }));
     }
     if (getMapType(map).BaseType == typeof(Window) ||
           getMapType(map).BaseType == typeof(YWindow))
     {
        // blocco mappa e cambio cursore in "attesa"
        Window w = (Window)activeMaps[map];

        w.Dispatcher.BeginInvoke(DispatcherPriority.Normal, new Action(() =>
                                {
                                    w.ShowInTaskbar = false;
                                    w.Cursor = Cursors.AppStarting;
                                    w.IsEnabled = false;
                                 }));
     }
}
```

private void UnFreezeMap(object map):

il metodo UnFreezeMap è il duale di FreezeMap. Dopo aver eseguito la ricerca della mappa specificata dal parametro in input nel dizionario delle mappe attive (activeMaps) la abilita impostando le proprietà ShowInTaskbar e IsEnabled a true.

```
private void UnFreezeMap(object map)
{
    if (getMapType(map).BaseType == typeof(Page))
    {
        // sblocco mappa e cambio cursore in "normale"
        Page w = (Page)activeMaps[map];
        w.Dispatcher.BeginInvoke(new Action(() =>
                                {
                                    w.Cursor = Cursors.Arrow;
                                    w.IsEnabled = true;
                                 }));
    }
    if (getMapType(map).BaseType == typeof(Window) ||
              getMapType(map).BaseType == typeof(YWindow))
    {
        // sblocco mappa e cambio cursore in "normale"
        Window w = (Window)activeMaps[map];

        w.Dispatcher.BeginInvoke(DispatcherPriority.Normal, new Action(() =>
                                {
                                    w.ShowInTaskbar = true;
                                    w.Cursor = Cursors.Arrow;
```

```
                                w.IsEnabled = true;
                            }));
    }
}
```

public void RefreshMap(object map, params string[] propertyNamesFilter):

il metodo `RefreshMap` ha come scopo l'aggiornamento dei controlli della mappa passata come primo parametro con i rispettivi valori del Datamodel. Il refresh viene eseguito lanciando il metodo `bindDataModelToPresentation` della classe `YBinder`. Questo metodo cicla su tutte le property del Datamodel corrente, cercando quelle che espongono l'attributo `[BindWith(...)]` (le quali, come vedremo, sono bindate a qualche controllo sulle mappe) ed eseguendo l'associazione tra valore della property nel Datamodel corrente e il valore mostrato dal controllo su mappa. E' prevista la possibilità di passare una lista di proprietà che si vogliono aggiornare come secondo parametro; se invece il parametro passato è solo l'oggetto `map`, allora sarà aggiornato l'intero Presentation della mappa.

```
public void RefreshMap(object map, params string[] propertyNamesFilter)
{
    YBinder.bindDataModelToPresentation(activeMaps[map], DataModel, propertyNamesFilter);
}
```

public void RefreshGrid(object map, string gridName):

il metodo `RefreshGrid` esegue l'aggiornamento dei dati contenuti in una griglia di tipo `YDataGrid` lanciando il metodo `refreshGrid` della classe `YBinder`. Il metodo cerca, tra tutti i campi del Datamodel che abbiano un bind ad un controllo del tipo `YDataGrid`, ad esempio:

```
[BindWith("ExampleWindow", "grdExample", WidgetType.Grid)]
public List<ExampleListGridRow> Elenco { get; set; }
```

quello specificato dal nome passato come secondo parametro (nell'esempio, verrà passato `"grdExample"`). Successivamente, esegue il caricamento della griglia con i valori presenti nella lista di tipo `List<DefinedGridRow>` cui il controllo è bindato.

```
public void RefreshGrid(object map, string gridName)
{
    YBinder.refreshGrid(activeMaps[map], gridName, DataModel);
}
```

public void RefreshDataModel(object map, params string[] propertyNamesFilter):

il metodo `RefreshDataModel` aggiorna il Datamodel con i valori attualmente contenuti nei controlli della mappa specificata come primo parametro. Il refresh viene eseguito invocando il metodo `bindPresentationToDataModel` della classe `YBinder`.

```
public void RefreshDataModel(object map, params string[] propertyNamesFilter)
{
    YBinder.bindPresentationToDataModel(activeMaps[map], DataModel, propertyNamesFilter);
}
```

public bool doValidation(object map):

il metodo doValidation esegue la validazione formale dei controlli della mappa passata come parametro. I controlli della libreria YControls YTextBox, YDatePicker, YComboBox e YIbanMultiBox espongono infatti le proprietà *Mandatory* ed *Invalid*. Quando un controllo in una mappa presenta la proprietà *Mandatory* impostata a true, l'architettura non procede con gli step di navigazione fino a quando questo non viene valorizzato. L'architettura si occupa anche di avvertire l'utente dell'obbligatorietà di un controllo impostando, a seguito del blocco del flusso di navigazione, la proprietà *Invalid* del controllo a true se questo non è stato valorizzato. La proprietà Invalid può essere utilizzata dallo sviluppatore per avvertire l'utente del fallimento di un controllo di coerenza dei dati inseriti (come ad esempio la correttezza di un codice fiscale). doValidation richiama il metodo doValidation della classe YBinder, che esegue dapprima un controllo sulla proprietà *Mandatory* di tutti i controlli della mappa passata come parametro. Il metodo doValidation restituisce true se la validazione è stata eseguita con successo, false altrimenti.

```
public static bool doValidation(DependencyObject map, out List<Control> invalidWidgets)
{
   invalidWidgets = new List<Control>();
   Hashtable widgets = new Hashtable();

   GetWidgets(ref widgets, map);

   foreach (var item in widgets.Keys)
   {
       // 1. validazione Mandatory Textboxes
       // la widget deve essere visibile, Usabile e non valorizzata
       if (widgets[item].GetType() == typeof(YControls.YTextBox))
       {
           YControls.YTextBox t = ((YControls.YTextBox)widgets[item]);
           if (t.Usable && t.Visible && t.Mandatory && string.IsNullOrEmpty(t.Text))
           {
               t.Invalid = true;
           }

           // salvo lista widget invalide popolata sia dalla widget
           // appena invalidata che da quelle invalidate in precedenza
           if (t.Invalid)
           {
               invalidWidgets.Add(t);
            }
       }

       // 2. validazione Mandatory DatePicker
       // la widget deve essere visibile, Usabile e non valorizzata
       if (widgets[item].GetType() == typeof(YControls.YDatePicker))
       {
           YControls.YDatePicker t = ((YControls.YDatePicker)widgets[item]);
           if (t.Usable && t.Visible && t.Mandatory && !t.HasText)
           {
               t.Invalid = true;
           }

           // salvo lista widget invalide popolata sia dalla widget
           // appena invalidata che da quelle invalidate in precedenza
           if (t.Invalid)
            {
                invalidWidgets.Add(t);
             }
```

```
            }

            // 3. validazione Mandatory Comboboxes
            // la widget deve essere visibile, Usabile e non valorizzata
            if (widgets[item].GetType() == typeof(YControls.YComboBox))
            {
                YControls.YComboBox t = ((YControls.YComboBox)widgets[item]);
                if (t.Usable && t.Visible && t.Mandatory && !t.HasText)
                {
                    t.Invalid = true;
                }

                // salvo lista widget invalide popolata sia dalla widget
                // appena invalidata che da quelle invalidate in precedenza
                if (t.Invalid)
                {
                   invalidWidgets.Add(t);
                }
        }

        // 4. validazione Mandatory IBAN
        // la widget deve essere visibile, Usabile e non valorizzata
        if (widgets[item].GetType() == typeof(YControls.YIbanMultiBox))
        {
            YControls.YIbanMultiBox t = ((YControls.YIbanMultiBox)widgets[item]);
            if (t.Usable && t.Visible && t.Mandatory && (t.GetIbanCode().Length < 27))
            {
               t.Invalid = true;
            }

            // salvo lista widget invalide popolata sia dalla widget
            // appena invalidata che da quelle invalidate in precedenza
            if (t.Invalid)
            {
               invalidWidgets.Add(t);
             }
    }

    // validazione OK se lista delle widgets invalide non ha elementi
    return (invalidWidgets.Count == 0);
}
```

4.5 Nodi di navigazione NavMap

Gli step di navigazione di tipo `NavMap` identificano la volontà di visualizzare una mappa presente nel progetto *Presentation* della soluzione della transazione diversa da quella corrente.

```
internal class mapMappa2 : NavMap
{
    public override object FormObject
    {
       // La mappa da mostrare è Mappa2
       get { return Maps.Mappa2; }
    }

    public override NavStep Next()
    {
       // Le mappe normalmente NON hanno un next
       return null;
    }
}
```

4.6 Nodi di navigazione NavService

Gli step di navigazione di tipo `NavService` identificano la volontà di utilizzare la logica di servizio della transazione, come chiamate a webservices o a DB. Generalmente la logica di servizio è confinata nel progetto Service della transazione.

```
internal class srvExample : NavService
{
    // invoco costruttore parametrico infrastrutturale
    // passando l'interfaccia del modello dati
    public srvExample (Datamodel datamodel) : base(datamodel) { }

    public override void ServiceObject()
    {
        // invoca il webservice
        new OneServiceClass(ModelloDati).serviceMethod();
    }

    public override NavStep Next()
    {
        // Per esempio, il next step potrebbe mostrare una mappa
        return new mapMain();
    }
}
```

4.7 Nodi di navigazione NavExcel

I nodi di navigazione di tipo `NavExcel` sono stati introdotti per dare la possibilità di produrre esportazioni in Excel dei dati di griglia presenti in alcune mappe.

```
internal class ExcelExporter : NavExcel
{
    // Costruttore di classe
    public ExcelExporter (Datamodel datamodel,
                          string xmlColumnDefinitions,
                          string excelFileName) : base(datamodel,
                          xmlColumnDefinitions,
                          excelFileName){ }

    // Metodo di conversione degli elementi nel tipo object (Boxing)
    // attraverso l'utilizzo di un delegato
    public override List<object> GetDataSource(YDatamodel datamodel)
    {
        Datamodel currentDataModel = (Datamodel)datamodel;

        return currentDataModel.ElencoOperazioni.ConvertAll<object>(
        delegate(ExampleListGridRow item) { return (object)item; });
    }

    public override NavStep Next()
    {
        return new mapMain ();
    }
}
```

Al costruttore della classe sono passati come parametri il Datamodel associato alla transazione, un file XML contenente le definizioni per gli stili di colonna, e il nome del file Excel che verrà prodotto. Il metodo `GetDataSource` si occupa del boxing degli elementi componenti le righe della griglia di tipo definito dallo sviluppatore (in questo esempio si tratta di elementi `ExampleListGridRow`). Da notare l'utilizzo del delegato per la conversione

del singolo elemento ExampleListGridRow. Il motore di navigazione della classe YNavigation (vedi paragrafo 4.2) si occupa di invocare il metodo ExportExcel che crea il foglio elettronico e, se Microsoft Excel è installato nel sistema, di visualizzarlo all'istante.

4.8 Nodi di navigazione NavNested

I nodi di navigazione di tipo NavNested consentono il lancio di una transazione nested dalla transazione chiamante e presentano delle differenze di implementazione a seconda del tipo di transazione che necessita di essere lanciata come figlia. I casi possibili sono elencati in Fig. 4.5.

Caso	Transazione Chiamante	Transazione Nested	Ritorno Dati
1	.NET	COM	No
2	.NET	COM	Sì
3	.NET	.NET	No
4	.NET	.NET	Sì
5	COM	.NET	No
6	COM	.NET	Sì
7	COM	COM	No
8	COM	COM	Sì

Fig. 4.5 – Casistiche possibili per il lancio di transazioni nested

I casi 7 ed 8 non saranno trattati, poiché estranei all'ambito dell'architettura YAST.
La classe astratta di definizione di un generico oggetto di navigazione nested è la seguente:

```
public abstract class NavNested : NavStep
{
    public delegate void ClosingOperationEventHandler(NavStep Next);
    public ClosingOperationEventHandler onClosingOperation;

    // Costruttore di classe
    public NavNested(IDatamodel datamodel)
    {
        ModelloDati = datamodel;
    }

    public override StepType Type
    {
        get { return StepType.Nested; }
    }

    public abstract void LaunchNested();
}
```

Un ruolo importante è quello giocato dalla definizione dell'evento onClosingOperation e dal tipo delegato ClosingOperationEventHandler che accetta metodi con un parametro di tipo NavStep che ritornano il tipo void. La sottoscrizione all'evento onClosingOperation permette la continuazione del flusso di navigazione della transazione chiamante alla chiusura della nested. La sottoscrizione viene effettuata nel blocco switch-case della classe Navigation del Controller, che gestisce la logica per ognuna delle Actions censite.
Il codice contenuto all'interno di una clausola case riguardante un'azione di lancio di una nested avrà la forma seguente:

```
case Actions.NestedEsempio :
    // Blocco la mappa attiva della transazione chiamante
    presentation.FreezeMap(senderMap);

    // Aggiorno datamodel con valori widget
    presentation.RefreshDataModel((Maps)senderMap);
    // Creo oggetto nested
    opeESEMPIO trxEsempio = new opeESEMPIO((Datamodel)CurrentDataModel);

    // Sottoscrizione all'evento di chiusura nested
    trxEsempio.onClosingOperation += new
    opeESEMPIO.ClosingOperationEventHandler(closing_nested);
    // Invoco la nested
    goNextStep(trxEsempio);

    // alla chiusura della nested eseguo un
    // refresh selettivo della mappa
    presentation.RefreshMap((Maps)senderMap, new string[3]{ "field1","field2","field3" });
break;
```

Come si può vedere, è necessario sottoscriversi all'evento `onClosingOperation` prima di invocare la nested, passando come gestore dell'evento il metodo `closing_nested` della classe `YNavigation` (questo è possibile, poiché la classe `Navigation` eredita da `YNavigation`):

```
public void closing_nested(NavStep next)
{
    goNextStep(next);
}
```

4.8.1 NavNested COM senza ritorno dati

Uno step NavNested COM identifica la volontà di lanciare una transazione di tipo COM (stiamo quindi parlando di una transazione VB6) in modalità nested. Il caso presentato in questo paragrafo, senza ritorno dati da parte della transazione chiamata verso la transazione chiamante, è indicato nella tabella in Fig. 4.5 con il numero 1.

```
internal class opeGESCO : NavNested
{
    // Costruttore di classe
    public opeGESCO(Datamodel datamodel) : base(datamodel){}

    public override void LaunchNested()
    {
        // Inizializzazione dell'oggetto di scambio
        SFOperations.GESCO gesco = new SFOperations.GESCO();
        // Valorizzazione dell'oggetto di scambio
        gesco.parametro1 = true;
        gesco.parametro2 = "esempio";
        gesco.trxChiamante = "GEALA";

        // Boxing oggetto di scambio
        object scambio = (object)gesco;

        // Lancio nested in modalità sincrona
        YInterop.LanciaCOM(ModalitaLancioCOM.SINCRONO, "GESCO", ref scambio);

        // Unboxing su oggetto SFOperation.object
        gesco = (SFOperations.GESCO)scambio;

        // Release forzata oggetto scambio dati
```

```
            System.Runtime.InteropServices.Marshal.ReleaseComObject(gesco);

            // Sollevo evento di continuazione navigazione
            if (onClosingOperation != null)
            {
                onClosingOperation(Next());
            }
        }

        // Ritorna un nodo di tipo NavStep
        public override NavStep Next()
        {
            // Il next step è ritornare il controllo a mapMain
            return new mapMain();
        }
}
```

Il metodo `LaunchNested` valorizza un oggetto di scambio del tipo `SFOperations.ACRONIMO`, (`SFOperations` è un componente architetturale VB6) necessario per poter passare dei dati in input alla transazione COM da lanciare. Successivamente, la transazione figlia viene lanciata attraverso il metodo `LanciaCOM` della classe `YInterop`, specificando che l'esecuzione dell'operazione deve avvenire in maniera sincrona e passando l'oggetto di scambio specificato per riferimento (il quale è preventivamente sottoposto a boxing per questioni di compatibilità tra tipi). E' pratica comune passare alla transazione nested anche una variabile, generalmente una stringa, che renda possibile il riconoscimento della transazione chiamante (`gesco.trxChiamante = "GEALA"` nell'esempio).
Il lancio della transazione nested può avvenire anche in modalità asincrona, tutte le modalità di lancio sono elencate nel tipo enumerato `ModalitaLancioCOM`:

```
public enum ModalitaLancioCOM
{
    SINCRONO,
    ASINCRONO
}
```

A differenza della modalità di lancio sincrona, la modalità asincrona permette di poter operare sulla transazione chiamante anche dopo la chiamata della nested.
Alla chiusura della transazione COM nested, come prima cosa viene chiamato il metodo `ReleaseComObject` della classe `Marshal` (nativa del Framework .NET) per decrementare il contatore dei riferimenti all'oggetto appena chiamato attraverso il *Runtime Callable Wrapper* (RCW), successivamente viene sollevato l'evento che segnala la chiusura della transazione nested: ad `onClosingOperation` viene passato il valore di ritorno del metodo `Next()`, pertanto il gestore dell'evento `closing_nested` riceverà come parametro (in questo esempio) un nodo di tipo `mapMain`; al suo interno, la chiamata al metodo `goNextStep` farà proseguire il flusso di navigazione.

4.8.2 NavNested COM con ritorno dati

Il caso che prendiamo ora in considerazione è quello riguardante il lancio di una transazione COM con ritorno di dati alla transazione chiamante (indicato nella tabella in Fig. 4.5 con il numero 2). La struttura della classe è molto simile al caso precedente:

```
internal class opeIDENT : NavNested
{
    // Costruttore di classe
    public opeIDENT(Datamodel datamodel) : base(datamodel) { }
```

```
    public override void LaunchNested()
    {
        // Inizializzazione dell'oggetto scambio
        SFOperations.IDENT ident = new SFOperations.IDENT();
        // Valorizzazione dell'oggetto di scambio
        ident.RicercaRapporto = true;
        ident.ABI = "XXXXX";
        ident.trxChiamante = "GEALA";

        // Boxing oggetto di scambio
        object scambio = (object)ident;

        // Lancio della nested in modalità sincrona
        YInterop.LanciaCOM(ModalitaLancioCOM.SINCRONO, "IDENT", ref scambio);

        // Unboxing su oggetto SFOperation.object
        ident = (SFOperations.IDENT)scambio;

        // Aggiornamento del Datamodel - solo se è stato premuto il
        // bottone 'Conferma' sulla UI della transazione nested
        if (ident.ExitConferma)
        {
            // verifico di aver effettivamente selezionato un rapporto
            if (!string.IsNullOrEmpty(ident.Filiale))
            {
                ((Datamodel)ModelloDati).Filiale = Convert.ToInt16(ident.Filiale);
            }
        }

        // Release forzata oggetto scambio dati
        System.Runtime.InteropServices.Marshal.ReleaseComObject(ident);

        // Sollevo evento di continuazione navigazione
        if (onClosingOperation != null)
        {
            onClosingOperation(Next());
        }
    }

    public override NavStep Next()
    {
        // Il next step è ritornare il controllo a mapMain
        return new mapMain();
    }
}
```

Al ritorno dalla transazione figlia viene eseguito l'unboxing dell'oggetto di scambio e ne viene letto il campo `ExitConferma`. Questo controllo è necessario per sapere se l'operazione di aggiornamento del modello dati è lecito, dal momento che ha senso solo nel caso in cui sulla mappa della transazione figlia sia stata eseguita dall'utente un'azione di conferma. Successivamente avviene (oppure no, come appena visto) l'aggiornamento del Datamodel con i parametri restituiti dalla transazione chiamata. Nell'esempio, per poter valorizzare il campo `Filiale` del Datamodel viene eseguito un cast al tipo `Datamodel` sull'oggetto `ModelloDati` del tipo interfaccia `IDatamodel` (ricordiamo che quando una classe implementa un'interfaccia, essa può essere utilizzata attraverso un riferimento a quell'interfaccia). L'oggetto `ModelloDati` viene inizializzato nel costruttore della classe `NavNested` in questo modo:

```
public NavNested(IDatamodel datamodel) { ModelloDati = datamodel; }
```

dove il parametro passato `datamodel` è solitamente un oggetto di tipo `Datamodel` contenente tutte le proprietà definite nel Datamodel della transazione chiamante. La richiesta che il parametro del costruttore sia di tipo `IDatamodel` obbliga lo sviluppatore a passare un oggetto che sia coerente con questa interfaccia, evitando così spiacevoli sorprese a runtime. L'interfaccia `IDatamodel` dichiara solo un oggetto `OPF` di tipo `Operazione`, una classe che descrive tutte le proprietà più importanti della transazione

```
public interface IDatamodel
{
    Operazione OPF { get; set; }
}
```

```
public class Operazione
{
    public string CodiceOPF { get; set; }
    public string Descrizione { get; set; }
    public TipoOperazione TipoOPF { get; set; }
    public string FullName { get; set; }
    public bool NeedApplicationDomain { get; set; }
    public eLaunchMode ExecutingMode { get; set; }
}
```

Al termine della valorizzazione del Datamodel della transazione chiamante, la classe è strutturata nello stesso modo di un nodo di navigazione `NavNested` COM senza ritorno dati.

4.8.3 NavNested .NET senza ritorno dati

Uno step NavNested .NET senza ritorno dati (caso indicato nella tabella in Fig. 4.5 con il numero 3) identifica la volontà di lanciare una transazione di tipo .NET in modalità nested, senza la necessità di dover ricevere dei dati di ritorno alla chiusura della transazione chiamata. La struttura utilizzata per il passaggio dei dati in input alla nested è un `Dictionary`, non essendoci problemi di compatibilità dei tipi tra due transazioni .NET.

```
internal class opeINSTO : NavNested
{
    // Costruttore di classe
    public opeINSTO(Datamodel datamodel) : base(datamodel) { }

    public override void LaunchNested()
    {
        // Cast datamodel
        Datamodel d = ((Datamodel)ModelloDati);

        // Dichiarazione e valorizzazione
        // oggetto scambio dati con nested
        Dictionary<string, object> nestedInput = new Dictionary<string, object>();
        nestedInput["Categoria"] = d.Categoria;
        nestedInput["Filiale"] = d.Filiale;
        nestedInput["Rapporto"] = d.Rapporto;

        // Sottoscrizione all'evento di chiusura della
        // nested intercettato e gestito dal CoreClient
        Core.onClosingNestedOperation += new
                    Core.ClosingNestedOperationEventHandler(finalize_nested);

        Core.LaunchNestedOperation("INSTO", AppDomain.CurrentDomain, nestedInput);
```

```
    }

    public void finalize_nested(string acronimo)
    {
        if (onClosingOperation != null)
        {
            onClosingOperation(Next());
        }
    }

    public override NavStep Next()
    {
        Core.onClosingNestedOperation -= new
                    Core.ClosingNestedOperationEventHandler(finalize_nested);
        // Il next step è ritornare il controllo a mapMain
        return new mapMain();
    }
}
```

Mentre il lancio delle transazioni VB6 era delegato alla classe `YInterop` del componente architetturale `YFarm`, nel caso di lancio di transazioni nested .NET senza ritorno dati è il CoreClient ad occuparsi del lancio attraverso il metodo `LaunchNestedOperation`.
Dopo la valorizzazione del dizionario dei parametri `nestedInput`, è necessaria la sottoscrizione all'evento `onClosingNestedOperation` della classe statica `Core` del CoreClient, utilizzando come gestore dell'evento il metodo `finalize_nested`. Al termine di queste operazioni si lancia l'operazione con il metodo `LaunchNestedOperation`, passando come parametri l'acronimo della transazione, il dominio di esecuzione della chiamante e il dizionario `nestedInput`.

4.8.4 NavNested .NET con ritorno dati

Lo step di navigazione di tipo NavNested .NET con ritorno dati (caso indicato nella tabella in Fig. 4.5 con il numero 4) utilizza il metodo `finalize_nested` come gestore dell'evento `onClosingNestedOperationWithArgs` per aggiornare il modello dati con i valori ritornati dalla nested.

```
internal class opeINSTO : NavNested
{
    // Costruttore di classe
    public opeINSTO(Datamodel datamodel) : base(datamodel) { }

    public override void LaunchNested()
    {
        // Cast datamodel
        Datamodel d = ((Datamodel)ModelloDati);
        internal bool aspetta = true;

        // Dichiarazione e valorizzazione
        // oggetto scambio dati con nested
        Dictionary<string, object> nestedInput = new Dictionary<string, object>();
        nestedInput["Categoria"] = d.Categoria;
        nestedInput["Filiale"] = d.Filiale;
        nestedInput["Rapporto"] = d.Rapporto;

        // Sottoscrizione all'evento di chiusura della
        // nested intercettato e gestito dal CoreClient
        Core.onClosingNestedOperationWithArgs += new
        Core.ClosingNestedOperationWithArgsEventHandler(finalize_nested);
        Core.LaunchNestedOperation("INSTO",
                            AppDomain.CurrentDomain,
```

```
                                ModelloDati.OPF.PipeCore.PipeID,
                                nestedInput);

        // Attende che i dati siano stati trasferiti dalla nested alla chiamante
        while (aspetta)
        {
            Dispatcher.CurrentDispatcher.Invoke(DispatcherPriority.Background,
                                          new Action(delegate { }));
        }
    }

    public void finalize_nested(string acronimo, object output)
    {
        // Aggiorno modello dati corrente con dati ritornati dalla nested
        // NB: i campi specificati nel DEVONO esistere nel datamodel della chiamante
        if (output != null)
        {
            ((Datamodel)ModelloDati).FillWith(output);
        }

        if (onClosingOperation != null)
        {
           onClosingOperation(Next());
        }
    }

    public override NavStep Next()
    {
         Core.onClosingNestedOperationWithArgs -= new
                    Core.ClosingNestedOperationWithArgsEventHandler(finalize_nested);
         aspetta = false;
         // Il next step è ritornare il controllo a mapMain
         return new mapMain();
    }
}
```

Il meccanismo di ritorno dei dati dalla nested verso la chiamante richiede un controllo di sincronizzazione, dal momento che il dizionario ritornato può contenere molte coppie chiave-valore e l'aggiornamento del Datamodel della transazione chiamante può non essere immediato. A questo scopo è stata introdotta la variabile booleana `aspetta`, posta inizialmente a `true` , che determina quanto è necessario aspettare. Il ciclo while che la utilizza come condizione svolge un compito simile a quello che svolgeva la funzione `DoEvents` in Visual Basic 6, permette cioè di attendere che il trasferimento dei dati dal dizionario al Datamodel sia terminato senza bloccare l'interfaccia utente:

```
while (aspetta)
{
    Dispatcher.CurrentDispatcher.Invoke(DispatcherPriority.Background,
                                   new Action(delegate { }));
}
```

La variabile `aspetta` cambia stato solo nel momento in cui viene chiamato il metodo `Next`; questo accade alla chiusura della transazione nested, quando viene scatenato l'evento `onClosingNestedOperationWithArgs`, che viene gestito dal metodo `finalize_nested`. All'interno di quest'ultimo, la chiamata alla funzione `FillWith` si occupa di valorizzare i campi del Datamodel corrente:

```
public void FillWith(object input)
{
    // tento la valorizzazione del datamodel SOLO nel caso di
    // chiamata nested (input != null)
```

```
    if (input != null)
    {
        foreach (KeyValuePair<string, object> kvp in (Dictionary<string, object>)input)
        {
            try
            {
                // es: <"ABI", "03069">  ==>  datamodel.ABI = "03069";
                Utils.setPropertyValueWithType(this, (GetType()).GetProperty(kvp.Key), kvp.Value);
            }
            catch (Exception ex)
            {
                // potrebbe accadere quando ci sono versioni differenti di DataModel
                // (properties mancanti): non è bloccante, loggo solamente
                Logger.Log(Assembly.GetExecutingAssembly(), TraceEventType.Warning,
                                                                "FillWith()", ex);
            }
        }
    }
}
```

La funzione `FillWith` è definita all'interno della classe `YDatamodel` di `YFarm`.

4.8.5 Chiamate Nested .NET da transazioni COM

Nel caso in cui la chiamata verso una transazione .NET sia effettuata da una transazione COM (casistiche 5 e 6 della tabella in figura 4.5), è necessario creare preventivamente una classe specifica per il componente SFOperations una cui istanza possa essere utilizzata come oggetto di scambio:

```
VERSION 1.0 CLASS

BEGIN
   MultiUse = -1  'True
   Persistable = 0  'NotPersistable
   DataBindingBehavior = 0  'vbNone
   DataSourceBehavior  = 0  'vbNone
   MTSTransactionMode  = 0  'NotAnMTSObject
END

Attribute VB_Name = "GEALA"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = True
Attribute VB_PredeclaredId = False
Attribute VB_Exposed = True

Option Explicit

Private mstrFunzione As String
Private mstrMatricola As String
Private mstrABI As String
Private mstrFILIALE As String
Private mstrDataContab As String

Public Property Get Nest_DATA_CONTAB() As String
   Nest_DATA_CONTAB = mstrDataContab
End Property

Public Property Get Nest_TIPO_CTR() As String
   Nest_TIPO_CTR = mstrFunzione
End Property

Public Property Get Nest_MATR_OPER() As String
```

```
   Nest_MATR_OPER = mstrMatricola
End Property

Public Property Get Nest_FILIALE() As String
   Nest_FILIALE = mstrFILIALE
End Property

Public Property Get Nest_ABI() As String
   Nest_ABI = mstrABI
End Property

Public Property Let Nest_TIPO_CTR(ByVal strNewValue As String)
   mstrFunzione = strNewValue
End Property

Public Property Let Nest_MATR_OPER(ByVal strNewValue As String)
   mstrMatricola = strNewValue
End Property

Public Property Let Nest_FILIALE(ByVal strNewValue As String)
   mstrFILIALE = strNewValue
End Property

Public Property Let Nest_ABI(ByVal strNewValue As String)
   mstrABI = strNewValue
End Property

Public Property Let Nest_DATA_CONTAB(ByVal strDataCont As String)
   mstrDataContab = strDataCont
End Property

Public Function ListProperties() As String
   ListProperties ="Nest_TIPO_CTR;Nest_MATR_OPER;Nest_FILIALE;
   Nest_ABI;Nest_DATA_CONTAB"
End Function

Public Sub Clone(objParams As GEALA)
   'DA UTILIZZARE NEL CASO DI SCAMBIO DATI CON TRX .NET
End Sub
```

Nella classe presentata come esempio, le variabili per lo scambio sono gestite attraverso delle proprietà; per ognuna di esse sono definiti i metodi `Let` e `Get`, utilizzati rispettivamente per l'assegnazione della variabile e per recuperarne il valore.
Una volta creata un'istanza della classe di scambio ed averne valorizzati i campi, è possibile lanciare la nested .NET con l'istruzione `ExecuteSyncYAST` della classe `Sys` del componente architetturale `SFSystem`:

```
Private Function startGEALA() As Boolean
   Dim objParams As SFOperations.GEALA
   On Error GoTo ErrorHandler

   Set objParams = New SFOperations.GEALA
   objParams.Nest_TIPO_CTR = "CHIUF"
   objParams.Nest_MATR_OPER = ""
   objParams.Nest_ABI = mstrAbi
   objParams.Nest_FILIALE = m_strFiliale
   objParams.Nest_DATA_CONTAB = Widgets.Value(widAllaDataSolare)

   Sys.ExecuteSyncYAST "GEALA", objParams

   FuncSubExit:
     Set objParams = Nothing
     Exit Function
```

```
   ErrorHandler:
     startGEALA = False
     Call errorTraceEventLog(TypeName(Me), "startGEALA", Err)
     GoTo FuncSubExit
End Function
```

L’introduzione del metodo `ExecuteSyncYAST` ha comportato la modifica di un componente architetturale dell’ambiente di Sportello preesistente, ma garantisce la possibilità di lanciare una transazione nested .NET da una transazione VB6 in modalità sincrona, favorendo in questo modo la convivenza di YAST con l’ambiente COM che lo circonda.

```
Public Function ExecuteSyncYAST(ByVal strAcronimo As String, ByVal objParams As Object) As Integer
   On Error GoTo hError

   Dim objLauncherYAST As CoreClientPlugin.Sportello
   Set objLauncherYAST = New CoreClientPlugin.Sportello
   ExecuteSyncYAST = objLauncherYAST.Launch(strAcronimo, objParams)
   Set objLauncherYAST = Nothing

   Exit Function

   hError:
    'Tracciatura Log
End Function
```

Come è possibile vedere dal blocco di codice, il metodo istanzia un oggetto della classe `Sportello` del componente architetturale di YAST denominato `CoreClientPlugin`. La classe definisce anche un metodo per il lancio di una transazione nested .NET in modalità asincrona. Nel caso in cui sia necessario un passaggio di informazioni dalla transazione.NET richiamata verso la transazione COM chiamante (caso 6 della tabella in figura 4.5), è sufficiente che la nested restituisca un dizionario le cui chiavi devono corrispondere ai nomi delle proprietà dell’oggetto di scambio. Al termine della chiamata, il `CoreClientPlugin` si occuperà della valorizzazione dei campi di quest’ultimo recuperandoli dalle voci corrispondenti del dizionario. Il componente `CoreClientPlugin` sarà trattato in maniera più approfondita nel capitolo ad esso dedicato.

5 Inter Process Communication (Pipes)

Il capitolo spiega come avviene la comunicazione tra due processi diversi in esecuzione sulla stessa macchina attraverso l'utilizzo di una named pipe. Nell'ambito di YAST, le named pipes sono utilizzate principalmente per la comunicazione tra il CoreClient e una transazione con finalità di logging sulle operazioni eseguite dall'utente.

5.1 Named Pipes

In informatica l'espressione Inter Process Communication si riferisce a tutte quelle tecnologie software il cui scopo è di consentire a diversi processi di comunicare tra loro scambiandosi dati e informazioni. Nei sistemi operativi, una pipe è uno degli strumenti disponibili per far comunicare tra loro dei processi. Le pipe, diversamente dai socket, offrono un canale di comunicazione monodirezionale, e quindi occorre impiegarne due per ottenere una comunicazione bidirezionale. Se due processi che devono comunicare tra loro attraverso una pipe sono del tutto separati (ovvero non hanno modo di scambiarsi descrittori di file o handles), è possibile creare nel *file system* un file speciale, detto FIFO o named pipe, che funge da punto di accesso alla pipe: scrivendo su questo file speciale si inviano dati alla pipe, mentre leggendolo si possono prelevare. In questo caso, i dati non sono memorizzati temporaneamente nel *file system*, ma transitano da un processo all'altro tramite un buffer.
In questo modo, un processo può offrire un servizio ad altri processi aprendo una named pipe in una posizione nota del *file system*. In YAST, la logica di creazione, gestione e scambio dati è implementata nelle classi *PipeClient* e *PipeServer* del componente YFarm.
Per utilizzare una pipe è necessario:

1. decidere chi farà da server e chi da client. Un criterio di scelta è, ad esempio, la persistenza dell'oggetto (il CoreClient è un processo sempre attivo in background, quindi farà da server per le pipes);

2. inizializzare il pipe server associandoci un nome univoco. A questo scopo viene utilizzato un numero casuale generato dal metodo getUniqueRandomNumber del CoreClient:

```
private static string getUniqueRandomNumber()
{
    // Considerando ddHHmmssff come formatmask, il valore massimo possibile è
    // 3123595999 => necessario utilizzare una variabile long
    long uid = -1;
    long.TryParse(DateTime.Now.ToString("ddHHmmssff"), out uid);
    // Non dovrebbe servire, ma visto che stiamo usando un dictionary meglio
    // controllare che l'uid appena generato non sia già usato come chiave
    if (uid != -1)
    {
        while (pipeTrxs.ContainsKey(uid.ToString()))
        {
            uid++; // aggiungo un'unità all'id appena generato
        }
```

```
        }

        return uid.ToString();
    }
```

quindi si istanzia un nuovo oggetto di tipo PipeServer e lo si avvia attraverso il metodo `Start(string pipename)`;

3. sottoscriversi all'evento "*ricezione messaggio dal client*":

```
operationPipe.MessageReceived += new
        PipeServer.MessageReceivedHandler(operationPipe_MessageReceived);
```

4. inizializzare il pipe client e connetterlo al pipe server utilizzando lo stesso nome utilizzato per identificare il pipe server attraverso il metodo

```
Connect(string pipename);
```

5. sottoscriversi all'evento "*ricezione messaggio dal server*":

```
pipeCore.MessageReceived += new PipeClient.MessageReceivedHandler(pipeCore_MessageReceived);
```

6. per inoltrare un messaggio, sia da server che da client, si usa il metodo:

```
public bool SendMessage(string message).
```

5.2 PipeServer

La classe `PipeServer` contiene i metodi per il lancio e la successiva gestione di un pipe server e la definizione del corpo dei thread di ascolto per le connessioni di client e per la lettura dei dati passati alla pipe. Il metodo che avvia il pipe server è `Start`:

```
public void Start(string pipename)
{
    // Il pipe name DEVE essere nella forma: \\.\pipe\<nome>
    pipeID = pipename;
    PipeName = string.Format(@"\\.\pipe\{0}", pipename);
    // Avvia il thread di ascolto connessioni alla pipe
    listenThread = new Thread(ListenForClients) { IsBackground = true };
    listenThread.Start();
    Running = true;
}
```

Il parametro `pipename` passato specifica il nome della pipe che, dopo essere formattato nel formato corretto per il suo utilizzo (il pipe server non può creare una pipe su un'altra macchina, quindi è necessario utilizzare un "." come nome del server) verrà assegnato alla proprietà pubblica `PipeName` della classe. Successivamente viene lanciato il thread di ascolto per le connessioni al pipe server. Il thread utilizza il codice del metodo `ListenForClients` con la proprietà `IsBackground` impostata a `true`, in modo da non impedire al processo di terminare in caso di chiusura.

```
void ListenForClients()
{
    SECURITY_DESCRIPTOR sd = new SECURITY_DESCRIPTOR();
    // Security Descriptor: abilito tutti i permessi
    // SECURITY_DESCRIPTOR_REVISION = 1
    InitializeSecurityDescriptor(ref sd, SECURITY_DESCRIPTOR_REVISION);
```

```
    SetSecurityDescriptorDacl(ref sd, true, IntPtr.Zero, false);

    IntPtr ptrSD = Marshal.AllocCoTaskMem(Marshal.SizeOf(sd));
    Marshal.StructureToPtr(sd, ptrSD, false);

    SECURITY_ATTRIBUTES sa = new SECURITY_ATTRIBUTES
                            {
                                nLength = Marshal.SizeOf(sd),
                                lpSecurityDescriptor = ptrSD,
                                 bInheritHandle = 1
                            };

    IntPtr ptrSA = Marshal.AllocCoTaskMem(Marshal.SizeOf(sa));
    Marshal.StructureToPtr(sa, ptrSA, false);

    while (true)
    {
    // Crea un'istanza della pipe per ogni client collegato
    // DUPLEX | FILE_FLAG_OVERLAPPED = 0x00000003 | 0x40000000
    SafeFileHandle clientHandle = CreateNamedPipe(PipeName,
                                              0x40000003,
                                              0,
                                              255,
                                              BUFFER_SIZE, // 4096
                                              BUFFER_SIZE, // 4096
                                              0,
                                              ptrSA);
      // Gestione errore connessione, continuo
      if (clientHandle.IsInvalid)
      {
         continue;
      }

      int success = ConnectNamedPipe(clientHandle, IntPtr.Zero);

      // Non è stato possibile instaurare la pipe col client
      if (success == 0)
     {
         // rilascio l'handle e mi metto in ascolto per
         // altre richieste di creazione pipe
         clientHandle.Close();
         continue;
      }

      Client client = new Client { handle = clientHandle };

      lock (clients)
      {
         clients.Add(client);
      }

       Thread readThread = new Thread(Read) { IsBackground = true };
       readThread.Start(client);
    }
}
```

Un descrittore di sicurezza è una struttura atta a fornire informazioni di sicurezza per gli oggetti del sistema operativo, che sono identificati da un ID univoco, l'handle. Ogni oggetto che può avere un security descriptor è detto *securable* e può essere protetto con dei permessi. Tutti gli oggetti di Windows con nome e anche alcuni senza nome sono *securable*. Un esempio del funzionamento dei descrittori di sicurezza è rappresentato dal messaggio “Accesso Negato” che compare quando si tenta di terminare alcuni processi dal Task Manager di Windows. Per accedere ai membri della struttura

SECURITY_DESCRIPTOR, nel codice sono utilizzati i metodi InitializeSecurityDescriptor e SetSecurityDescriptorDacl e non un accesso diretto, perché il formato della struttura può variare. Il primo dei due metodi inizializza un descrittore di sicurezza, il secondo imposta le informazioni in una DACL (*Discretionary Access Control List*) allo scopo di specificare che livello di accesso all'oggetto si può avere. Nel caso del codice sopra, il flag *pDacl* (il terzo parametro del metodo SetSecurityDescriptorDacl) è settato con il valore IntPtr.Zero, pertanto non sono impostate politiche di accesso particolari, e l'oggetto sarà disponibile per l'accesso da qualunque applicazione. Successivamente viene allocato un blocco di memoria delle dimensioni della struttura SECURITY_DESCRIPTOR, alla quale si farà riferimento attraverso il puntatore ptrSD. La seconda struttura, SECURITY_ATTRIBUTES, conterrà il descrittore di sicurezza per l'accesso al client e specifica, con il parametro bInheritHandle impostato a 1, che l'handle recuperato attraverso la struttura è ereditabile.
Il ciclo di ascolto del thread tenta di stabilire una connessione full duplex attraverso una pipe per ognuno dei client che dimostra di voler connettersi e, in caso di successo, viene aggiornata la lista dei client connessi (previo lock della lista) e viene lanciato un thread specifico per la lettura dei dati provenienti del client.
Al thread di lettura dei messaggi dalla pipe viene passato come parametro l'oggetto Client che identifica l'handle del client che si è connesso e del quale si vogliono leggere i messaggi:

```
void Read(object clientObj)
{
    Client client = (Client)clientObj;
    client.stream = new FileStream(client.handle,
                                   FileAccess.ReadWrite,
                                   BUFFER_SIZE,
                                   true);

    byte[] buffer = new byte[BUFFER_SIZE];

    while (true)
    {
       int bytesRead = 0;
       using (MemoryStream ms = new MemoryStream())
       {
          try
          {
             // Stream length
             int totalSize = client.stream.Read(buffer, 0, 4);

             // Il client si è scollegato
             if (totalSize == 0)
             {
                 break;
             }

             totalSize = BitConverter.ToInt32(buffer, 0);

             do
             {
                 int numBytes = client.stream.Read(buffer, 0,
                                              Math.Min(totalSize - bytesRead, BUFFER_SIZE));
                 ms.Write(buffer, 0, numBytes);
                 bytesRead += numBytes;

              } while (bytesRead < totalSize);
          }
          catch
          {
              // Al verificarsi di un errore esco
```

```
                break;
            }

            // Il client si è scollegato
            if (bytesRead == 0)
            {
                break;
            }

            // Sollevo evento ricezione dati dal client
            if (MessageReceived != null)
            {
                ASCIIEncoding enc = new ASCIIEncoding();
                MessageReceived(enc.GetString(ms.ToArray(), 0, ms.ToArray().Length));
            }
        }
    }

    // IPC guidelines: clients deve essere locked altrimenti
    // stream.close() lo potrei lanciare da un altro thread
    // con conseguenti IOException

    lock (clients)
    {
        // Rilascio risorse impegnate
        DisconnectNamedPipe(client.handle);
        client.stream.Close();
        client.handle.Close();
        clients.Remove(client);
    }

    // Sollevo evento disconnessione client
    if (ClientDisconnected != null)
    {
        ClientDisconnected();
    }
}
```

La dimensione del buffer dedicato al messaggio è di `4096` bytes ed è definita dalla costante `BUFFER_SIZE`. Successivamente si inizializza un oggetto `FileStream` specificando il tipo di accesso in lettura/scrittura. Il ciclo di lettura funziona in questo modo: il thread legge i primi 4 bytes del messaggio, che ne identificano la lunghezza: se questa è pari a zero, significa che il client ha terminato la connessione, il thread interrompe il ciclo e la pipe viene disconnessa attraverso il metodo `DisconnectNamedPipe(client.handle)`, quindi vengono rilasciate le risorse utilizzate da `client`, l'oggetto viene eliminato dalla lista dei clients collegati alla pipe e viene sollevato l'evento che segnala la disconnessione di un client, gestito attraverso un delegato. Nel caso in cui la lunghezza del messaggio ritornata sia maggiore di zero, avviene la lettura del messaggio e il sollevamento dell'evento di ricezione di un messaggio da parte del client, che viene gestito tramite un delegato che accetta un parametro di tipo `string` (il messaggio).
Per concludere la panoramica sulle funzionalità della classe `PipeServer`, esaminiamo il metodo che si occupa dell’invio di un messaggio a tutti i client collegati alla pipe:

```
public void SendMessage(string message)
{
    // Conversione string => byte[]
    ASCIIEncoding enc = new ASCIIEncoding();
    byte[] messageBuffer = enc.GetBytes(message);

    lock (clients)
    {
```

```
        byte[] messageLength = BitConverter.GetBytes(messageBuffer.Length);

        foreach (Client client in clients)
        {
            // Lunghezza
            client.stream.Write(messageLength, 0, 4);

            // Dati
            client.stream.Write(messageBuffer, 0, messageBuffer.Length);
            client.stream.Flush();
        }
    }
}
```

Il primo passo è quello di convertire la stringa del messaggio passata come parametro in un array di byte. Successivamente, utilizzando il metodo statico `GetBytes` della classe `BitConverter`, il valore intero indicante la dimensione dell'array di byte contenente il messaggio viene copiato in un altro array di byte denominato `messageLength`.
L'ultima operazione è quella di scrivere nel buffer il messaggio (prima i 4 byte che ne definiscono la lunghezza, poi il messaggio vero e proprio) attraverso il metodo `Write` dell'oggetto `FileStream`. Al termine della scrittura si esegue un `Flush()` dello stream, forzando così la scrittura del messaggio su disco.

5.3 PipeClient

La classe `PipeClient` gestisce la ricezione e l'invio di messaggi attraverso la pipe nella stessa maniera appena vista per la classe `PipeServer`, ma è necessario fare le dovute precisazioni per quanto riguarda connessione e disconnessione ad un pipe server.

```
public void Connect(string pipename)
{
    if (Connected)
    {
        throw new Exception("Pipe server già collegato");
    }

    // Il pipe name DEVE essere nella forma: \\.\pipe\<nome>
    pipeID = pipename;
    PipeName = string.Format(@"\\.\pipe\{0}", pipename);
    // GENERIC_READ | GENERIC_WRITE = 0x80000000 | 0x40000000
    handle = CreateFile(PipeName, 0xC0000000, 0, IntPtr.Zero, 3,
                        0x40000000, IntPtr.Zero);
    // Non è stato possibile creare l'handle (server non attivo)
    if (handle.IsInvalid)
    {
       return;
    }

    Connected = true;

    // Avvia il thread di ascolto dei messaggi
    readThread = new Thread(Read) { IsBackground = true };
    readThread.Start();
}
```

Al metodo `Connect(string pipename)` viene passato un parametro che specifica il nome della named pipe da utilizzare, quindi viene effettuata una chiamata alla funzione `CreateFile` delle API di Windows allo scopo di aprire un handle per connettersi al server. I parametri accettati da questa funzione sono, in sequenza:

- *nome della named pipe*, il nome della pipe alla quale connettersi;
- *tipo di accesso*: nel nostro caso, eseguendo l'OR tra i valori 0x80000000 (GENERIC_READ) e 0x40000000 (GENERIC_WRITE) si ottiene 0xC0000000, quindi un accesso sia in lettura che in scrittura;
- *modalità di condivisione*: il valore 0 previene l'apertura del file nel caso in cui altri processi ne richiedano l'eliminazione oppure l'accesso in lettura o in scrittura;
- *puntatore a struttura* SECURITY_ATTRIBUTES: facoltativo, per il client non sono utilizzati attributi di sicurezza;
- *creazione file/apertura file esistente*: questo parametro impostato a 3 specifica l'apertura di una pipe esistente;
- *flags e attributi*: la presenza del flag 0x40000000 specifica che il file è stato aperto per operazioni di I/O asincrone; il settaggio di questo flag è di importanza fondamentale, poiché permette che più operazioni di lettura/scrittura avvengano simultaneamente;
- *puntatore al template file*: facoltativo, non utilizzato. Si tratta di un puntatore ad un file contenente attributi e specifiche supplementari destinato ad estendere gli attributi del file appena aperto.

Se un handle valido è ritornato dalla funzione CreateFile, la connessione è instaurata e viene lanciato il thread di lettura, altrimenti il metodo di connessione getta semplicemente la spugna.

```
public void Disconnect()
{
    if (!Connected)
    {
        return;
    }

    // Rilascio delle risorse
    Connected = false;
    PipeName = null;

    if (stream != null)
    {
        stream.Close();
    }

    handle.Close();
    stream = null;
    handle = null;
}
```

Il metodo Disconnect() si occupa di disconnettere il client dal pipe server e di liberare le risorse utilizzate dalla connessione, richiamando il metodo Close sull’oggetto FileStream utilizzato per la lettura/scrittura dei dati dalla pipe e sull’handle ritornato dalla funzione CreateFile.

5.4 Stack Tracing

Lo scopo finale dell’utilizzo delle named pipes è quello di rendere possibile una gestione centralizzata, da parte del CoreClient, dell’attività di log delle azioni eseguite su tutte le mappe. Le azioni che è possibile compiere sui controlli e i comportamenti del controllo soggetti a tracciatura sono definiti nel tipo enumerato TrackAction:

```
// Enum delle azioni "tracciabili"
public enum TrackAction
{
    ButtonClick = 0,   // click su Button
    ValueChange = 1,   // valuechange su textbox/combobox
    Exception = 2      // eccezione
}
```

Nei costruttori delle classi che caratterizzano le mappe di una transazione dovrebbe essere sempre presente il seguente codice:

```
public EsempioWindow(Datamodel datamodel) : base(datamodel)
{
    InitializeComponent();
    …

    // Recupero tutti i controlli della mappa di tipo YControls
    List<DependencyObject> widgets = new List<DependencyObject>();
    if (((DependencyObject)this).TryGetChilds(out widgets))
    {
        // Per ogni controllo eseguo la sottoscrizione all'evento
        // onChangeToStack (ammesso che  esista, es: YLabel non lo espone)
        foreach (DependencyObject widget in widgets)
        {
            // Sottoscrizione all'evento via Reflection
            EventInfo evInfo = widget.GetType().GetEvent("changeToStack");

            if (evInfo != null)
            {
                Type evType = evInfo.EventHandlerType;
                MethodInfo miHandler = this.GetType().GetMethod("onChangeToStack",
                                       BindingFlags.NonPublic | BindingFlags.Instance);
                // Creo un delegato
                Delegate d = Delegate.CreateDelegate(evType, this, miHandler);
                // Sottoscrivo all'evento
                MethodInfo addHandler = evInfo.GetAddMethod();
                Object[] addHandlerArgs = { d };
                addHandler.Invoke(widget, addHandlerArgs);
             }
        }
    }
}
```

Questo blocco di codice è trasparente rispetto al nome della mappa e dei controlli presenti su di essa, pertanto può essere utilizzato in ogni costruttore senza problemi. La funzione svolta è quella di effettuare, per ognuno dei controlli di tipo `YControls` presenti in mappa, la sottoscrizione all'evento `changeToStack`. Questa sottoscrizione permetterà alla transazione, attraverso il gestore dell'evento `onChangeToStack`, di comunicare attraverso la pipe (con il metodo `writeStack` la classe statica `Logger` del componente `YFarm`) quali operazioni sono state svolte dall'utente sulla mappa. Il codice inizia richiamando il metodo statico `TryGetChilds` della classe `YBinder` di `YFarm`, per il recupero dei controlli presenti in mappa. Una volta terminata l'esecuzione del metodo, la lista di dei controlli `widgets` sarà valorizzata con tanti oggetti di tipo `DependencyObject` (ogni controllo WPF deriva dalla classe `DependencyObject`) quanti sono i controlli in mappa. Successivamente, attraverso Reflection, è possibile recuperare le informazioni sull'evento `changeToStack` (se esistente) per ognuno di essi e, se questo è stato reso disponibile nel controllo, si effettua la sottoscrizione ad esso.
Naturalmente, questo meccanismo implica che anche il gestore dell'evento `changeToStack` sia presente in ognuna delle mappe della transazione:

```
void onChangeToStack(object sender, ChangeToStackEventArgs args)
{
    // Leggo la property Name del controllo via Reflection
    String name = sender.GetType().GetProperty("Name").GetValue(sender, null).ToString();
    // Composizione e scrittura Stack Trace
    string mess = string.Format("{0}.{1} {2}",
    this.GetType().getConnectedMap(typeof(Maps)).ToString(), name, args.dettagli);

    Logger.writeStack(CurrentDataModel.OPF, (TrackAction)args.trackAction, mess);
}
```

Il gestore `onChangeToStack` si occupa di recuperare il nome del controllo su cui è stata eseguita un'azione da parte dell'utente e lo assegna alla variabile `name`. Il messaggio che sarà inviato attraverso la pipe è composto di tre argomenti:

- il nome della mappa che contiene il controllo;
- il nome del controllo;
- i dettagli sull'operazione eseguita sul controllo o su eventuali eccezioni da questo sollevate.

L'utilizzo effettivo della named pipe è gestito dal metodo `writeStack`, che accetta tre parametri: un oggetto di tipo `Operazione` contenente delle informazioni sulla transazione (tra le quali il nome della pipe di comunicazione che dovrà essere utilizzata, passato dal CoreClient), il tipo di azione/eccezione da tracciare (deve essere definita nel tipo enumerato `TrackAction`) e il messaggio appena composto:

```
public static void writeStack(Operazione opf, TrackAction action, string dettagli)
{
    // Inizializzazione pipe client
    Pipe.PipeClient pipe = opf.PipeCore;

    if (pipe != null)
    {
       try
       {
          StackTraceItem sti = new StackTraceItem();
          sti.Transazione = opf.CodiceOPF.Trim().ToUpper();
          sti.TimeStamp = DateTime.Now;
          sti.Azione = action;
          sti.Dettagli = dettagli;

          if (pipe.Connected)
          {
              pipe.SendMessage(sti.Encode(pipe.PipeID));
          }
       }
       catch (Exception ex)
       {
          Logger.Log(Assembly.GetExecutingAssembly(), TraceEventType.Warning,
                                                    "writeStack()", ex);
       }
    }
    else
    {
        Logger.Log(Assembly.GetExecutingAssembly(), TraceEventType.Information,
                     "writeStack()", "PipeCore NULL! No StrackTracing...");
    }
}
```

6 WCF in YAST

Il sottosistema applicativo WCF del Framework .NET è utilizzato in YAST per migliorare le prestazioni in fase di avvio di una transazione in particolare, denominata GEALA. WCF viene utilizzato per la creazione di un endpoint con il quale la transazione mette a disposizione di altri processi (nel nostro caso, il CoreClient) dei metodi propri con un contract. L'idea che consente di elevare le prestazioni di caricamento è semplice: il processo della transazione non viene terminato con la chiusura del flusso di navigazione, ma viene mantenuto in vita e risvegliato con una segnalazione al momento opportuno.

6.1 Utilizzo di WCF in YAST

In YAST, la modalità di comunicazione attraverso WCF è stata introdotta solo nelle ultime versioni dell'architettura, su consiglio di un consulente Microsoft, per risolvere alcuni problemi di performance riguardanti una transazione in particolare. Nel momento in cui scrivo WCF è utilizzato solo nel caso di avvio della transazione da pagina web. La transazione con problemi di performance, denominata GEALA, presentava dei tempi di caricamento molto lunghi.
Si tratta di una transazione nested, che non può quindi essere lanciata direttamente da Portale, ma solo come figlia di un'altra transazione o da pagina web. La soluzione che ha permesso di migliorarne le prestazioni (e che utilizza WCF) è quella di tenere, dopo il primo avvio, il processo GEALA.Controller.exe in esecuzione anche quando effettivamente non viene utilizzato, in uno stato idle. Nel momento in cui è necessario che la transazione si risvegli, attraverso WCF il CoreClient invia un segnale di risveglio al Controller di GEALA, della quale viene lanciato il metodo `Launch`.
Una volta che l'utente avrà terminato di utilizzare il programma, il processo ritorna in stato idle, in attesa di ricevere il prossimo messaggio di risveglio.
Iniziamo analizzando il comportamento della transazione GEALA all'avvio, ricordando che nelle proprietà del progetto Controller della transazione è specificato come oggetto di avvio la classe `Program` (Fig. 6.1).

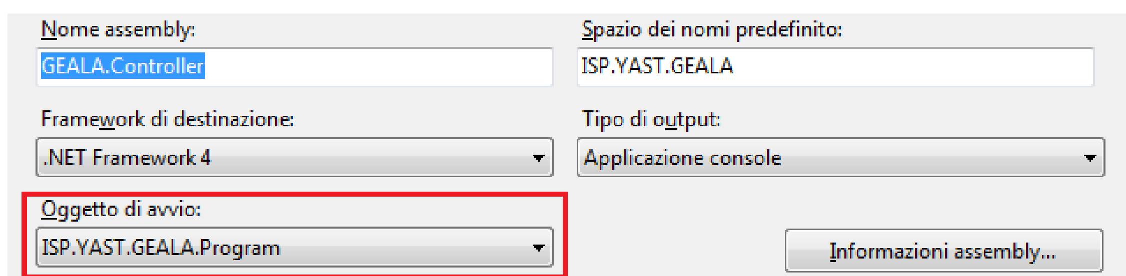

Fig. 6.1 – La classe Program impostata come entrypoint della transazione

```
class Program
{
    static ServiceHost host = null;
    public static AutoResetEvent waitForInput = new AutoResetEvent(false);
```

```
    [STAThread]
    static void Main(string[] args)
    {
       if (ConfigurationManager.AppSettings["LaunchMode"].Trim().ToUpper() == "WCF")
       {
           // LANCIO WCF HOST DEL PROCESSO
           host = ClientTaskService.StartServiceHost(Assembly.GetExecutingAssembly().
                                                FullName.Substring(0, 5).ToUpper());
           while (true)
           {
               // Faccio "respirare" la coda dei messaggi ogni 500ms
               while (!waitForInput.WaitOne(500, false));
               ClientTaskService.StartMain();
               waitForInput.Reset();
           }
       }
       else
       {
           // LANCIO TRADIZIONALE DEL PROCESSO:
           // Recupero le informazioni della transazione dall'albero operazioni
           Operazione opf = YInterop.LeggiDettagliOperazione(Assembly.
                                 GetExecutingAssembly().FullName.Substring(0, 5));

           // args[0] valorizzato dal CoreClient durante il lancio del processo
           string pipeName = string.Empty;

           try
           {
               pipeName = args[0];
           }
           catch
           {
               pipeName = Core.startPipe(opf);
            }

            // Invoco Controller della transazione
            // NB: GEALA PUO' ESSERE CHIAMATA SOLO NESTED QUINDI
            // LE DEVONO ESSERE PASSATI DEI PARAMETRI
            Dictionary<string, object> input = null;

            try
            {
               input = Utils.convertActiveXParams(args[1]);
            }
            catch
            {
               input = null;
            }

           opf.ExecutingMode = eLaunchMode.ACTIVEX;
           Controller.Launch(opf, input, pipeName);
       }
    }
}
```

All'avvio della transazione viene prima di tutto eseguito un controllo per verificare se la modalità di avvio è WCF. La modalità di avvio è registrata nell'App.Config del progetto Controller, all'interno del tag appSettings:

```
<!--PARAMETRI CONFIGURAZIONE-->
  <appSettings>
    <!-- Modalità lancio operazione [WCF o Process] -->
    <add key="LaunchMode" value="WCF"/>
    <!-- Attributo di Lanciabilità dell'operazione da Portale -->
```

```
    <add key="StartableFromPortale" value="False"/>
    <!-- Attributo di Multi-Instaziabilità dell'operazione -->
    <add key="MultipleInstance" value="False"/>
    <!-- Eseguibilità operazione in modalità LOG (Offline) -->
    <add key="Storable" value="False"/>
  </appSettings>
```

E’ possibile cambiare la modalità di lancio della transazione da WCF a standard semplicemente cambiando il valore della key `LaunchMode` in un qualsiasi altro valore.
Nel caso in cui la modalità di avvio sia WCF, il metodo statico `StartServiceHost` della classe `ClientTaskService` si occupa dell'hosting del servizio istanziando un nuovo oggetto statico `ServiceHost` (denominato `host`):

```
public static ServiceHost StartServiceHost(string AppName)
{
    Uri appUri = new Uri(string.Format(ClientTaskCommon.CoreClientListenerUriFormat, AppName));

    ServiceHost host = new ServiceHost(typeof(ClientTaskService), new Uri[] { appUri });
    host.AddServiceEndpoint(typeof(IClientTask), ClientTaskCommon.GetBinding(), appUri);
    host.Open();

    return host;
}
```

al metodo viene passato l’acronimo della transazione, recuperato via Reflection.
I parametri passati al costruttore specificano, nell’ordine:

- la classe che implementa i metodi definiti nel Contract del servizio WCF (`ClientTaskService`);
- l’URI di default della named pipe da utilizzare, recuperato attraverso la proprietà `CoreClientListenerUri` della classe `ClientTaskCommon` di `YFarm`.

Dalla classe `ClientTaskCommon` viene recuperato anche il tipo di binding da effettuare, che nel nostro caso sarà di tipo `NetNamedPipeBinding`:

```
public class ClientTaskCommon
{
    public const string CoreClientListenerUriFormat = "net.pipe://xxxxxxx/App/{0}";

    public static System.ServiceModel.Channels.Binding GetBinding()
    {
        return new NetNamedPipeBinding(NetNamedPipeSecurityMode.None);
    }
}
```

Dopo aver inizializzato la parte di hosting della connessione attraverso la named pipe con WCF, due cicli while annidati tengono il thread del Controller in attesa utilizzando l'oggetto `waitForInput` di tipo `AutoResetEvent`:

```
while (true)
{
    // Faccio "respirare" la coda dei messaggi ogni 500ms
    while (!waitForInput.WaitOne(500, false))
    {
        ClientTaskService.StartMain();
    }

    waitForInput.Reset();
}
```

La classe `AutoResetEvent` consente ai thread di comunicare tra loro tramite delle segnalazioni. In genere si tratta di comunicazioni relative ad una risorsa per la quale i thread necessitano dell'accesso esclusivo, ma, in questo caso, la “risorsa” è il metodo statico `StartMain` della classe `ClientTaskService` del Controller di GEALA.
`WaitOne` mette il thread in attesa, ma il metodo non è bloccato in attesa del segnale, bensì viene eseguito periodicamente ogni 500 ms fino a quando il valore da esso restituito non sarà `true`. Ora, lo scopo di tutto questo è fare in modo che il CoreClient, su richiesta del Portale o di una pagina web, possa segnalare alla transazione GEALA “dormiente” di risvegliarsi.
La segnalazione che permette il risveglio di GEALA è effettuata dal CoreClient attraverso il metodo:

```
proxy.StartOperation(string.Empty, ((Dictionary<string, object>)input)["params"].ToString());
```

dove l'oggetto `proxy` è un'istanza della classe `ClientTaskProxy` che, dopo essere stata istanziata passando al costruttore l'acronimo della transazione, con il metodo `StartOperation` permette l'inizializzazione della named pipe da parte del CoreClient (recuperando l'URI ed il Binding di default della named pipe dalla classe `CoreClientCommon`) ed il successivo richiamo del metodo `StartOperation` della classe `ClientTaskService` del Controller di GEALA utilizzando WCF.

```
public void StartOperation(string pipe, string parameters)
{
    IClientTask client = null;
    ChannelFactory<IClientTask> cf = null;

    int RetryCounter = 0;
    int MAX_RETRY = 3;
    int RetryDelay = 1000;
    bool EndPointSetted = false;

    while (!EndPointSetted)
    {
        try
        {
            CreateFactoryAndChannel(out client, out cf);
            client.StartOperation(pipe, parameters);
            EndPointSetted = true;
        }
        catch (EndpointNotFoundException e)
        {
            Thread.Sleep(RetryDelay);
            if (++RetryCounter > MAX_RETRY)
            {
                throw;
             }
         }
         finally
         {
             Helpers.SafeDispose((IClientChannel)client, (ChannelFactory)cf);
          }
     }
}
```

`StartOperation` crea la connessione al servizio WCF named pipe avviato precedentemente richiamando il metodo privato `CreateFactoryAndChannel` della stessa classe, il quale specifica Address e Binding da utilizzare recuperandoli sempre dalla classe `CoreClientCommon` di supporto. Una volta impostata la connessione al servizio, viene avviato il metodo `StartOperation` dello stesso.

```
public void StartOperation(string pipe, string parameters)
{
    _pipeFromCore = pipe;
    _parametersFromCore = parameters;
    // Se esiste già un'istanza attiva => non faccio il set
    // ma nascondo il waitPanel e mostro un avviso
    if (_semaforo)
    {
        // Chiusura waitpanel
        YManager.hideWaitPanel();
        Process wMsgBox = new Process();
        // configuro processo che mostra YMessageBox (passo testo
        // e caption come parametri da linea di comando)
        wMsgBox.StartInfo.CreateNoWindow = true;
        wMsgBox.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
        wMsgBox.StartInfo.UseShellExecute = false;
        wMsgBox.StartInfo.FileName = @"C:\Programmi\XXXXXX\YAST\YWarning.exe";
        wMsgBox.StartInfo.Arguments = "\"Attenzione! Risulta ancora aperta una Lista
                              eventi. Uscire dalla Lista e riprovare. \" \"Attenzione!\"";
        // Lo Eseguo
        wMsgBox.Start();
    }
    else
    {
        Program.waitForInput.Set();
    }
}
```

Come prima cosa il metodo controlla che non vi siano altre istanze di GEALA in esecuzione (non idle, ma risvegliate e correntemente visualizzate dall'utente) attraverso il flag `_semaforo`. Se è questo il caso, viene lanciato un nuovo processo, `YWarning`, il quale non fa altro che restituire una MessageBox i cui parametri (messaggio e caption) sono passati come parametri all’oggetto `Process` che identifica il processo. Se, invece, la transazione GEALA non è utilizzata ma comunque attiva tra i processi, `waitForInputSet()` segnala al ciclo while interno dei due visti in precedenza la possibilità di procedere. A quel punto viene avviato il metodo statico `ClientTaskService.StartMain()`, che avvia a tutti gli effetti la transazione rendendola disponibile all’utilizzo da parte dell’utente:

```
public static void StartMain()
{
    _semaforo = true;
    string[] args = new string[] { _pipeFromCore, _parametersFromCore };

    // Recupero le informazioni della transazione dall'albero operazioni
    Operazione opf = YInterop.LeggiDettagliOperazione(Assembly.
                                  GetExecutingAssembly().FullName.Substring(0, 5));

    // args[0] valorizzato dal CoreClient durante il lancio del processo
    string pipeName = string.Empty;
    try { pipeName = args[0]; }
    catch { pipeName = Core.startPipe(opf); }

    // Sono stati specificati dei parametri di input (da ActiveX)?
    if (args.Length == 2)
    {
        // SI: lancio da ActiveX, input diverso da null
        opf.ExecutingMode = eLaunchMode.ACTIVEX;
        Controller.Launch(opf, Utils.convertActiveXParams(args[1]), pipeName);
    }
    else
    {
        // NO: lancio da Portale, input = null
        Controller.Launch(opf, null, pipeName);
```

```
    }

    _semaforo = false;
}
```

Il metodo StartMain è quello che effettivamente avvia il flusso di navigazione della transazione. Dopo aver recuperato tutte le informazioni riguardanti l'operazione con l'ausilio dell' *albero operazioni* e dopo aver impostato i parametri per la comunicazione attraverso named pipe, viene avviato il Controller attraverso il metodo statico Launch. I parametri passati al metodo forniscono le informazioni raccolte dall'albero operazioni come oggetto di tipo Operazione, eventuali parametri da pagina web e il nome della named pipe da utilizzare per la comunicazione con il CoreClient. Da notare è l'impostazione del flag _semaforo a true, che comporta il rifiuto di eventuali altre chiamate al metodo StartMain; questo sarà nuovamente possibile solo al ritorno dal metodo Launch del Controller al termine della transazione, quando il valore di _semaforo tornerà a false.
Esaminiamo ora il caso in cui il LaunchMode definito per la transazione sia quello standard. In questo caso come prima cosa vengono recuperate, con il metodo LeggiDettagliOperazione, le informazioni di base sulla transazione: descrizione e TipoOpF. TipoOpF può assumere due possibili valori, OPNET (se la transazione è .NET) oppure OPCOM (se la transazione è VB6). Se la transazione è di tipo .NET, sono passate anche informazioni sul nome esteso della transazione ed un flag che determina se necessita di un dominio di applicazione particolare.
Successivamente viene inizializzata la named pipe da parte della transazione (senza l'utilizzo di WCF), assegnandole come nome il primo degli argomenti passati dal CoreClient nel momento della creazione dell'oggetto Process:

```
trxControllerProcess.StartInfo.Arguments = pipeID.ToString() + " " +
                    ((Dictionary<string, object>)input)["params"].ToString();
```

L'ultima operazione svolta prima del lancio del Controller della transazione è la valorizzazione di un dizionario con i parametri passati sotto forma di stringa dal controllo ActiveX che gestisce le richieste di avvio da pagina web.

```
Dictionary<string, object> input = null;
try
{
    input = Utils.convertActiveXParams(args[1]);
}
catch
{
    input = null;
}
```

7 CoreClient

Il componente di YAST denominato CoreClient è il nucleo principale dell'architettura .NET: ad esso è infatti delegata la gestione del lancio di una transazione .NET a fronte di una richiesta del Portale, del CoreClientPlugin (ActiveX) oppure di un'altra transazione. Il componente permette inoltre una gestione centralizzata della tracciatura delle operazioni eseguite dall'utente sulle mappe delle transazioni e di eventuali eccezioni.

7.1 Lancio di transazioni .NET

Il CoreClient è il componente architetturale di YAST che si occupa del lancio delle transazioni .NET e del logging centralizzato delle operazioni effettuate dall'utente durante il loro utilizzo. Si tratta di un processo che viene avviato conseguentemente all'avvio del Portale (quindi, salvo casi particolari, all'avvio del sistema) e, in ambiente di Integration/System, la sua presenza tra i processi è segnalata da un'icona nella tray-area.

Fig. 7.1 – Tray-icon del CoreClient

Il funzionamento del CoreClient è basato sui membri di un'unica classe statica, denominata `Core`, la quale contiene anche i metodi per l'impostazione del pipe server per la comunicazione tra processi.
L'avvio di una transazione .NET, come abbiamo già visto, può essere effettuata in due modalità diverse:

1. lancio *diretto* – lancio diretto da Portale o da pagina web di una transazione;

2. lancio *nested* – lancio di una transazione come figlia di un'altra transazione;

In entrambi i casi, il lancio è delegato al metodo statico `LaunchOperation(string acronimo, AppDomain dominio = null, object input = null, string callerPipeID = string.Empty)` i cui parametri identificano, nell'ordine:

- l'acronimo della transazione da lanciare (ad esempio `"GEALA"`);
- il dominio applicativo da usare per la transazione;
- il dizionario contenente i dati di scambio in caso di chiamata di tipo *nested*;
- l'ID della pipe della transazione chiamante (usata solo nel caso di chiamata *nested*).

```
internal static void LaunchOperation(string acronimo, AppDomain dominio = null, object
                                    input = null, string callerPipeID = string.Empty)
{
    int MaxRetry = 6;
```

```
int retryDelay = 500;
int retryCounter = 0;

// Determino la modalità di esecuzione della richiesta
eLaunchMode launchMode = eLaunchMode.DIRETTA;

if (!string.IsNullOrEmpty(callerPipeID))
{
    launchMode = eLaunchMode.NESTED;
}
else
{
    // input != null e AppDomain = null => activeX
    if (dominio == null && input != null)
    {
        launchMode = eLaunchMode.ACTIVEX;
    }
}

// recupero le informazioni della transazione dall'albero operazioni
Operazione opf = YInterop.LeggiDettagliOperazione(acronimo);
opf.ExecutingMode = launchMode;

if (opf == null)
{
    // non è stato possibile recuperare le informazioni
    // della transazione dell'albero operazioni
    MessageBox.Show(log, "YAST - Lancio operazione", MessageBoxButton.OK, MessageBoxImage.Stop);
}
else
{
    try
    {
        // Nested o Avvio diretto?
        string pipeID = string.Empty;

        switch (launchMode)
        {
            case eLaunchMode.ACTIVEX:

            Configuration configSection = ConfigurationManager.
            OpenExeConfiguration(string.Format(@"C:\Program
                                Files\XXXXXXX\YAST\{0}\{1}.Controller.exe", opf.FullName,
                                opf.CodiceOPF.ToUpper()));

            if (configSection.AppSettings.Settings["LaunchMode"].Value == "WCF")
            {
                Process[] Processes = Process.GetProcessesByName(
                              string.Format("{0}.Controller", acronimo.Trim().ToUpper()));
                Process process = null;

                if (Processes.Length > 0)
                {
                    process = Processes[0];
                }

                // Controllo sull'utilizzo della memoria della transazione
                if (process != null)
                {
                    long memoryLimit = Convert.ToInt64(ConfigurationManager.
                                                        AppSettings["MemoryLimit"]);

                    long privateBytes = process.PrivateMemorySize64;
                    // Se supero la soglia, termino il processo. Poi verrà
                    // riavviato nel codice che segue
                    if (privateBytes >= memoryLimit)
```

```
        {
            process.Kill();

            // Da tempo al processo di chiudersi, il kill è
            // asincrono, invia solo un segnale al processo
            while (!process.HasExited && retryCounter <= MaxRetry)
            {
                // 0,5 secondi per come è configurato retryDelay
                Thread.Sleep(retryDelay);
                retryCounter++;
            }

                process = null;
        }
    }

    // MODALITA' WCF
    // è già attivo il processo?
    if (Process.GetProcessesByName(string.Format("{0}.Controller",
                                    acronimo.Trim().ToUpper())).Length == 0)
    {
        // NO: lo lancio
        Process trxControllerWCF = new Process();

        // Configura processo
        trxControllerWCF.StartInfo.CreateNoWindow = true;
        trxControllerWCF.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
        trxControllerWCF.StartInfo.UseShellExecute = false;
        trxControllerWCF.StartInfo.FileName = string.Format(@"C:\
                            Programmi\XXXXXXX\YAST\{0}\{1}.Controller.exe",
                                            opf.FullName, acronimo);
        // Esegue il processo
        trxControllerWCF.Start();
    }

    // Segnala il risveglio della transazione
    ClientTaskProxy proxy = new ClientTaskProxy(acronimo);
    proxy.StartOperation(string.Empty, ((Dictionary<string,
                                object>)input)["params"].ToString());
}
else
{
    // MODALITA' PROCESS
    // Creo la pipe di comunicazione e salvo l'ID
    pipeID = startPipe(opf);

    Process trxControllerProcess = new Process();

    // Configura processo
    trxControllerProcess.StartInfo.CreateNoWindow = true;
    trxControllerProcess.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
    trxControllerProcess.StartInfo.UseShellExecute = false;
    trxControllerProcess.StartInfo.FileName = string.Format(@"C:\
                        Programmi\XXXXXXX\YAST\{0}\{1}.Controller.exe",
                                        opf.FullName, acronimo);

    trxControllerProcess.StartInfo.Arguments = pipeID.ToString() +
            " " + ((Dictionary<string, object>)input)["params"].ToString();

    // Esegue il processo
    trxControllerProcess.Start();
}
break;

case eLaunchMode.DIRETTA:
```

```
                pipeID = startPipe(opf);
                Process trxController = new Process();

                // Configura il processo
                trxController.StartInfo.CreateNoWindow = true;
                trxController.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
                trxController.StartInfo.UseShellExecute = false;
                trxController.StartInfo.FileName = string.Format(@"C:\Programmi\
                XXXXXXX\YAST\{0}\{1}.Controller.exe", opf.FullName, acronimo);

                trxController.StartInfo.Arguments = pipeID.ToString();

                // Esegue il processo
                trxController.Start();
                break;

                case eLaunchMode.NESTED:

                // Lancio la transazione nel dominio indicato (dominio della
                // chiamante) passando in input l'oggetto di scambio dati
                // con chiamante ed il nome della pipe di comunicazione
                // CoreClient <=> transazione
                dominio.CreateInstanceFrom(string.Format(@"C:\Programmi\
                          XXXXXXX\YAST\{0}\{1}.Controller.exe", opf.FullName, acronimo),
                          string.Format("XXX.YAST.{0}.RemoteObject", acronimo),
                          false, 0, null, new object[] { opf,
                                                         input,
                                                         callerPipeID.ToString() },
                                                         null, null);
                break;

                default:

                // Log: modalità di lancio sconosciuta
                break;
            }
        }
        catch (Exception ex)
        {
            // Log: Errore critico
            MessageBox.Show(ex.Message, "YAST - Errore Critico", MessageBoxButton.OK, MessageBoxImage.Stop);
        }
    }
}
```

Dopo un primo controllo che decide se la transazione è richiamata tramite un lancio diretto, un lancio nested o attraverso ActiveX valutando i parametri `callerPipeID` e `dominio` (rispettivamente ID della pipe e dominio applicativo dell'eventuale transazione chiamante), l'acronimo viene passato come parametro al metodo `LeggiDettagliOperazione` della classe `YInterop`, il quale, come abbiamo già visto nel capitolo precedente, si occupa di recuperare delle informazioni specifiche per la transazione operando su una struttura chiamata *albero operazioni*. Tra queste vi sono:

- Nome della transazione;
- Codice della transazione;
- Modalità di esecuzione della transazione;
- ID della named pipe utilizzata per la comunicazione con il CoreClient.

A seguito del recupero delle informazioni dall'albero operazioni, salvate nell'oggetto `opf` di tipo `Operation`, si entra in un blocco switch-case sulla variabile `launchMode`.

7.2 Modalità di lancio ACTIVEX

La modalità di lancio ActiveX viene invocata quando la transazione viene avviata da pagina web e, a tutti gli effetti, si tratta di un lancio *diretto* dell'operazione. Il metodo `LaunchOperationFromActiveX` del CoreClient non contiene codice proprio, ma la logica di avvio, ma una chiamata al principale metodo di avvio delle transazioni: `LaunchOperation`.

```
public static void LaunchOperationFromActiveX(string acronimo, object input)
{
    LaunchOperation(acronimo, null, input, null);
}
```

L'avvio inizia con la lettura del file XML di configurazione del Controller della transazione da lanciare, che specifica la modalità di lancio (WCF o Process) ed il limite superiore di memoria privata che può essere utilizzata dal processo. La strategia utilizzata per il lancio è rappresentata nel diagramma in Fig. 7.2.

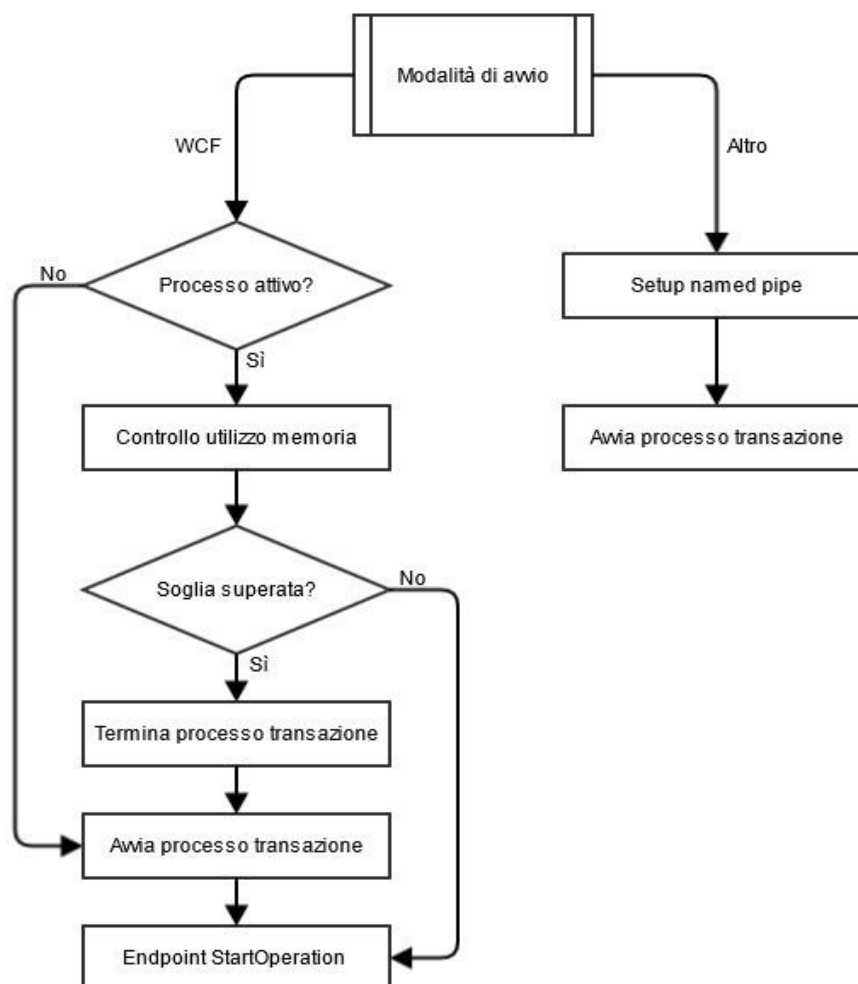

Fig. 7.2 – Schema della modalità di lancio ActiveX

Se la modalità di avvio richiesta è WCF, come prima cosa il CoreClient controlla se il processo della transazione è già attivo, in tal caso viene effettuato un ulteriore controllo sull'utilizzo di memoria di tale processo, terminandolo se supera la soglia definita dal campo `MemoryLimit` dell'XML di configurazione della transazione per poi riavviarlo. Lo scopo di questo controllo e quello di evitare eventuali memory leaks (la transazione

GEALA è stata soggetta a questo genere di problemi, durante lo sviluppo). Successivamente, a prescindere dal fatto che il Controller fosse già in esecuzione o che sia stato avviato o riavviato, viene istanziato l'oggetto della classe `ClientTaskProxy` di `YFarm` necessario per lanciare l'interfaccia utente della transazione.

7.3 Modalità di lancio DIRETTA

La modalità di lancio *diretta* crea un pipe server per la comunicazione con la transazione con la chiamata a `startPipe`, quindi avvia direttamente il *Controller* dell'operazione:

```
case eLaunchMode.DIRETTA:

    pipeID = startPipe(opf);
    Process trxController = new Process();

    // Configura il processo
    trxController.StartInfo.CreateNoWindow = true;
    trxController.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
    trxController.StartInfo.UseShellExecute = false;
    trxController.StartInfo.FileName = string.Format(@"C:\Programmi\XXXXXXX\YAST\{0}\{1}.Controller.exe",
                                                      opf.FullName, acronimo);

    trxController.StartInfo.Arguments = pipeID.ToString();

    // Esegue il processo
    trxController.Start();

break;
```

Alla transazione chiamata viene passato come parametro l'ID della named pipe sulla quale il CoreClient sarà in ascolto.

7.4 Modalità di lancio NESTED

Per quanto riguarda il lancio di un'operazione nested, il CoreClient gestisce solamente il caso dell'avvio di una transazione .NET da una chiamante anch'essa .NET. Infatti, il caso in cui la transazione chiamante sia .NET e la figlia sia una transazione VB6, è gestito senza che il CoreClient faccia da intermediario, direttamente dalla transazione madre. Questo è possibile grazie al componente architetturale `YFarm` ed in particolare alla classe `YInterop` che contiene tutta la logica di interfacciamento con i componenti preesistenti dell'architettura, tra i quali SPILanciatore, che permette l'avvio di transazioni VB6. Anche l'altro caso, che riguarda l'avvio di una operazione .NET da una VB6, non richiede la mediazione del CoreClient ed è stato già discusso nel paragrafo 4.8.5.

7.4.1 Nested .NET → .NET

Per il lancio nested di una transazione .NET da un'altra transazione .NET il CoreClient dispone del metodo statico `LaunchNestedOperation`. Tale metodo viene richiamato, previa valorizzazione del dizionario dei dati di scambio (per convenzione chiamato sempre

nestedInput) e sottoscrizione all'evento di chiusura della nested, nel nodo di navigazione di tipo NavNested della transazione chiamante:

```
// Sottoscrivo all'evento di chiusura della nested
// intercettato e gestito dal CoreClient
Core.onClosingNestedOperationWithArgs += new
                 Core.ClosingNestedOperationWithArgsEventHandler(finalize_nested);
Core.LaunchNestedOperation("GEAST", AppDomain.CurrentDomain, "pipeID", nestedInput);
```

I parametri passati sono, nell'ordine, l'acronimo della transazione nested da avviare, il dominio entro il quale questa è destinata ad essere eseguita (per le chiamate nested .NET → .NET è generalmente lo stesso della chiamante), l'ID della named pipe utilizzata dalla transazione chiamante e il dizionario dei dati passati in input alla transazione figlia. E' importante notare che il terzo parametro, quello relativo all'ID della named pipe deve necessariamente avere un valore, diverso da null e da string.Empty: questo perché il metodo LaunchOperation, del quale LaunchNestedOperation non è altro che un wrapper, riconosce la modalità di lancio nested solo se questo parametro è valorizzato. Nel caso in cui non si voglia utilizzare la comunicazione attraverso la named pipe, è sufficiente passare un qualunque valore di tipo string che non sia string.Empty oppure "". Il lancio della nested è compiuto dal metodo CreateInstanceFrom classe AppDomain, che crea una nuova istanza della classe definita nell'assembly specificato:

```
dominio.CreateInstanceFrom(string.Format(@"C:\Programmi\XXXXXXX\YAST\{0}\{1}.Controller.exe",
                                         opf.FullName, acronimo),
                                         string.Format("XXX.YAST.{0}.RemoteObject", acronimo),
                                         false,
                                         0,
                                         null,
                                         new object[]
                                         {
                                             opf, input, callerPipeID.ToString()
                                          },
                                          null,
                                          null);
```

I parametri passati al metodo sono, nell'ordine:

- nome (incluso il path) del file che contiene l'assembly con la classe richiesta;
- nome della classe del quale si richiede venga creata un'istanza (incluso il namespace di appartenenza);
- ignoreCase: variabile booleana che specifica se deve essere eseguita una ricerca della classe con distinzione tra lettere maiuscole e minuscole;
- bindingAttr: posto a 0 impone una ricerca di costruttori pubblici con distinzione tra lettere maiuscole e minuscole;
- binder: posto a null specifica l'utilizzo del gestore di associazione predefinito;
- args: array di elementi object da passare come argomenti al costruttore del tipo richiesto. Nel nostro caso, il primo argomento è la struttura Operazione contenente informazioni sulla chiamante, il secondo il dizionario di scambio e il terzo il pipe ID della transazione chiamante;

- culture: informazioni specifiche delle impostazioni cultura che regolano l'assegnazione forzata di args ai tipi formali dichiarati per il costruttore. Non è utilizzato, pertanto viene valorizzato a `null`;

- activationAttributes: matrice di uno o più attributi che possono prendere parte allo scambio. Non utilizzato.

Da quanto visto, abbiamo quindi un entrypoint alternativo per ogni transazione che supporti l'avvio in modalità nested, definito dalla classe pubblica `RemoteObject` nel *Program.cs*:

```
// entrypoint per quando istanziati in NESTED
public class RemoteObject : System.MarshalByRefObject
{
    public RemoteObject(Operazione opf, object inputParam, string pipeName)
    {
        Controller.Launch(opf, inputParam, pipeName);
    }
}
```

il costruttore di questa classe avvia il *Controller* della transazione figlia, passando le informazioni necessarie all'avvio dell'operazione.

7.4.2 Nested VB6 → .NET

Il lancio di una transazione .NET da VB6 è piuttosto raro, ed è possibile in due modalità:

1. Sincrona, attraverso il metodo `LaunchOperationFromVB6` del CoreClient;

2. Asincrona, attraverso il metodo `LaunchOperationFromVB6Async` del CoreClient.

Il metodo `LaunchOperationFromVB6` del CoreClient viene utilizzato sia per lanciare transazioni .NET da Portale in modalità sincrona che per il lancio di nested VB6 → .NET. L'elemento architetturale che interfaccia il mondo COM con quello .NET è il controllo ActiveX denominato CoreClientPlugin. Ricordiamo che quest'ultimo svolge anche il compito di avviare una transazione .NET da pagina web, come avremo modo di approfondire nel capitolo successivo.
Il diagramma in Fig. 7.3 rappresenta il procedimento di avvio di una transazione .NET da una chiamante VB6.

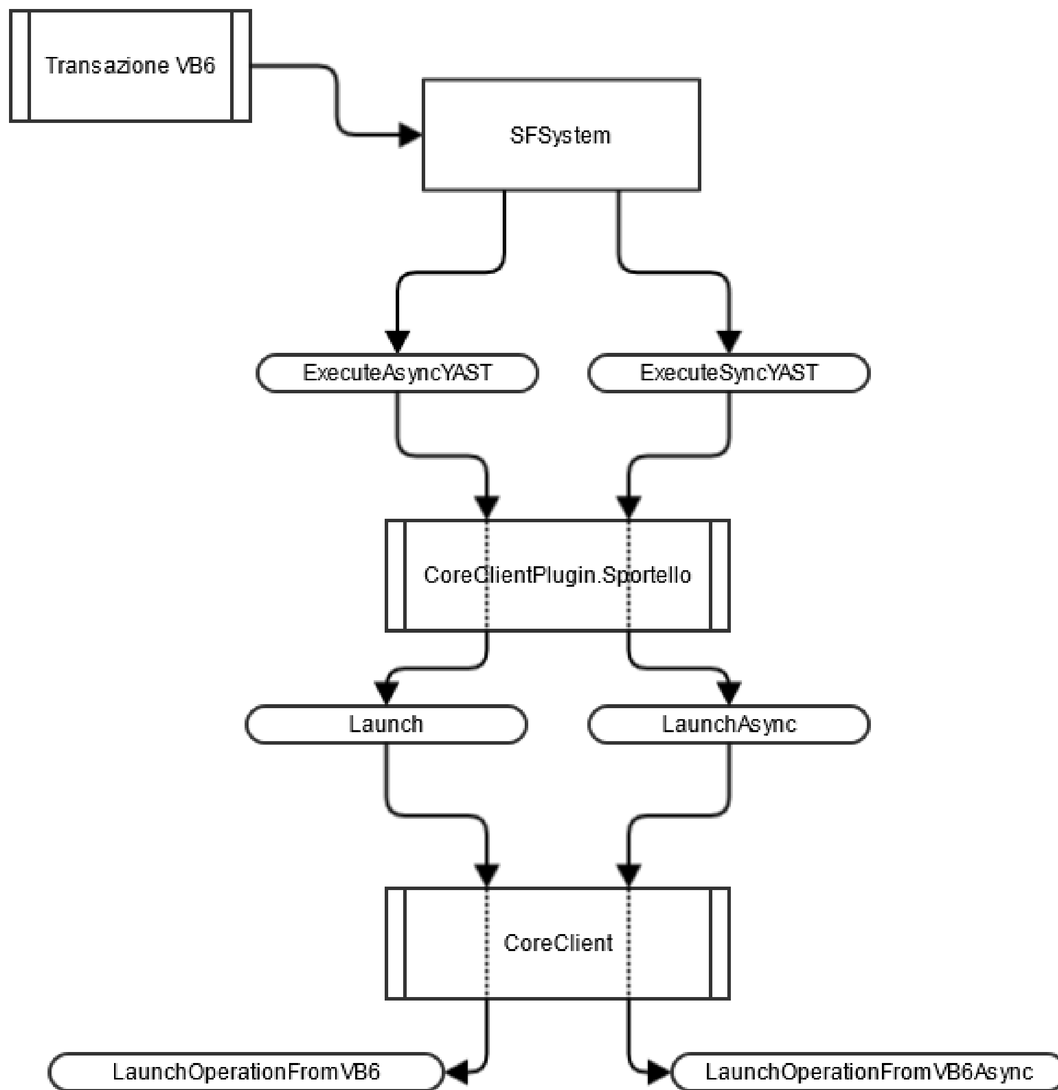

Fig. 7.3 – Lancio nested VB6 → .NET

Come abbiamo già visto nel paragrafo 4.7.5, la transazione chiamante utilizza, per il lancio di transazioni .NET, i metodi `ExecuteSyncYAST` ed `ExecuteAsyncYAST` del componente SFSystem. Questi, una volta chiamati, istanziano un oggetto Sportello del CoreClientPlugin, la cui interfaccia è visibile da SFSystem:

```
// Interfaccia visibile da SFSystem (COM)
public interface ISportello
{
    int Launch(string acronimo, ref object scambio);
    int LaunchAsync(string acronimo, ref object scambio);
}
```

I metodi `Launch` e `LaunchAsync` sono strutturati nello stesso modo: dopo aver eseguito un controllo sulla presenza del CoreClient tra i processi attivi (ed averlo avviato se per qualche motivo non fosse presente), svolto dalla chiamata a `CheckCoreActivity`, il metodo tenta di valorizzare la stringa `listProperties` invocando la funzione ListProperties() dell’oggetto di scambio. La funzione ritorna la concatenazione dei nomi delle variabili

dell'oggetto di scambio in input che sarà possibile utilizzare, separate dal carattere ";". Naturalmente, se per qualche motivo tale lista risulta essere vuota, la chiamata nested termina e il metodo segnala il fallimento ritornando il valore -1. Se, al contrario, la stringa non è vuota o nulla, si passa alla valorizzazione del dizionario `input`, dove le key sono costituite dai nomi delle proprietà recuperate da `listProperties`.

```
public int Launch(string acronimo, ref object scambio)
{
    // CoreClient è attivo? se non lo è, lo rilancio.
    Utils.CheckCoreActivity(true);

    string listProperties = string.Empty;

    try
    {
        listProperties = Convert.ToString(scambio.GetType().InvokeMember(
                "ListProperties", BindingFlags.InvokeMethod, null, scambio, null));
    }
    catch (Exception ex)
    {
        // Log dell'eccezione
     }

     if (string.IsNullOrEmpty(listProperties))
     {
         return -1; // SFOperations NON è conforme => non lancio la transazione
                    // e ritorno alla chiamante -1
     }

     // Conversione SFOperations.<trx> ==> input [Dictionary<string, object>]
     Dictionary<string, object> input = new Dictionary<string, object>();
     try
     {
         foreach (string property in listProperties.Split(';'))
         {
             input[property] = scambio.GetType().InvokeMember(property,
                                  BindingFlags.GetProperty, null, scambio, null);
         }
     }
     catch (Exception ex)
     {
         return -1; // se si verifica un errore => non lancio la transazione alla
                    // e ritorno alla chiamante -1
     }

     // Lancia la transazione
     Dictionary<string, object> output = Core.LaunchOperationFromVB6(acronimo, input);
     // Conversione output [Dictionary<string, object>] ==> SFOperations.<trx>
     try
     {
         foreach (string property in listProperties.Split(';'))
         {
             scambio.GetType().InvokeMember(property, BindingFlags.SetProperty, null,
                                    scambio, new object[] { output[property] });
         }
     }
     catch (Exception ex)
     {
          return -1; // se si verifica un errore => non lancio la
                     // transazione e ritorno alla chiamante "-1"
     }

     return 1;
}
```

Dopo la valorizzazione del dizionario per il passaggio dei dati avviene la chiamata al metodo LaunchOperationFromVB6 oppure LaunchOperationFromVB6Async del CoreClient.
Sia l'uno che l'altro accettano come parametri l'acronimo della transazione .NET da lanciare ed un oggetto object contenente i dati di input per l'operazione (il dizionario input sottoposto a boxing):

```
public static Dictionary<string, object> LaunchOperationFromVB6(string acronimo, object input)
{
    Dictionary<string, object> output = null;

    // recupero le informazioni della transazione dall'albero operazioni
    Operazione opf = YInterop.LeggiDettagliOperazione(acronimo);
    if (opf == null)
    {
        // LOG + MessageBox
    }
    else
    {
        try
        {
            opf.ExecutingMode = eLaunchMode.NESTED;
            // Creo pipe comunicazione e salvo ID
            string pipeID = startPipe(opf);

            Assembly nested = Assembly.LoadFrom(string.Format(@"C:\Programmi\XXXXXXX\YAST\
                                    {0}\{1}.Controller.exe", opf.FullName,
                                    acronimo));

            Type nestedType = nested.GetType(string.Format("XXX.YAST.{0}.RemoteObject",
                                    acronimo),
                                    false,
                                    true);

            Activator.CreateInstance(nestedType, new object[] {opf, input, pipeID.ToString()});

            output = new Dictionary<string, object>();

            foreach (KeyValuePair<string, object> kvp in (Dictionary<string, object>)input)
            {
                // esiste un valore aggiornato?
                object currValue = kvp.Value;

                if (temporaryOutputDict != null)
                {
                    if (temporaryOutputDict.ContainsKey(kvp.Key))
                     {
                        currValue = temporaryOutputDict[kvp.Key];
                     }
                }

                output[kvp.Key] = currValue;
            }

            nested = null;
        }
        catch (Exception ex)
        {
            output = new Dictionary<string, object>();
            MessageBox.Show(ex.Message, "YAST - Errore Critico", MessageBoxButton.OK,
                                                  MessageBoxImage.Stop);
        }
    }

    return output;
```

```
}
```

`LaunchOperationFromVB6` inizia con il recupero delle informazioni sulla transazione da lanciare dall'albero operazioni. Se l'operazione di recupero fallisce, la procedura termina restituendo in output un oggetto `Dictionary` posto a `null`. Se invece i dati riguardanti la transazione sono recuperati, viene inizializzato il pipe server e si lancia la transazione .NET recuperando il tipo della classe `RemoteObject` per poi crearne un'istanza con `CreateInstance`. Il dominio applicativo dell'operazione sarà lo stesso del CoreClient. Nel caso di ritorno dati da parte delle transazione chiamata, viene valorizzato il dizionario temporaneo `temporaryOutputDict` nel metodo `EndOperation`, il quale viene chiamato automaticamente quando l'operazione raggiunge il nodo di chiusura `NavEnd`. A questo punto viene creato un clone del dizionario utilizzato per l'input, `output`, e se ne aggiornano solamente i valori che hanno subito modifiche.
A differenza di quanto appena visto, `LaunchOperationFromVB6Async` presenta una modalità di lancio del processo differente e non permette il ritorno di dati dalla transazione chiamata:

```
public static void LaunchOperationFromVB6Async(string acronimo, object input)
{
    // Recupero le informazioni della transazione dall'albero operazioni
    Operazione opf = YInterop.LeggiDettagliOperazione(acronimo);
    opf.ExecutingMode = eLaunchMode.ACTIVEX;

    // MODALITA' PROCESS
    // creo pipe comunicazione e salvo ID
    string pipeID = startPipe(opf);
    Process trxControllerProcess = new Process();

    // Configura processo
    trxControllerProcess.StartInfo.CreateNoWindow = true;
    trxControllerProcess.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
    trxControllerProcess.StartInfo.UseShellExecute = false;
    trxControllerProcess.StartInfo.FileName = string.Format(@"C:\Programmi\XXXXXXX\
                                  YAST\{0}\{1}.Controller.exe", opf.FullName, acronimo);

    // Nome della pipe di comunicazione CoreClient <=> transazione
    // più stringa parametri ActiveX
    trxControllerProcess.StartInfo.Arguments = pipeID.ToString() + " " +
    Utils.DictToBase64String((Dictionary<string, object>)input);

    // Esegui processo
    trxControllerProcess.Start();
}
```

La transazione figlia viene lanciata come un processo separato. Gli argomenti passati sono l'ID della named pipe di comunicazione con il CoreClient e il dizionario contenente i parametri di input preventivamente convertito in una stringa Base64.
Il motivo di questa conversione è legato al fatto che `Arguments` è di tipo `string`. Inoltre, essendo una stringa che definisce i parametri passati all'applicazione, eventuali spazi all'interno di essa dichiarano, senza che ciò sia voluto, più parametri diversi. Il metodo ausiliario `DictToBase64String` della classe `Utils` del componente `YFarm` permette di generare una stringa contenente una sequenza di coppie key-value da passare come unico parametro in questo modo:

```
dict[key1] = val1, dict[key2] = val2    →    key1=val1; key2=val2
```

Esiste però ancora un problema da risolvere: cosa accade se uno dei valori del dizionario è una stringa che contiene uno spazio? Viene generato involontariamente un

secondo parametro e si corrompe l'integrità del primo. Allo scopo di evitare questo spiacevole inconveniente si codifica la stringa generata dal dizionario con il sistema di numerazione Base64, che elimina ogni spazio dalla stringa, pur incrementandone le dimensioni di circa il 33%.

7.5 Terminazione di una transazione

Ora che abbiamo visto quali sono i meccanismi utilizzati per il lancio delle transazioni nelle varie modalità, passiamo ad esaminare come viene effettuata dal CoreClient la chiusura di una transazione. A questo scopo, il CoreClient espone tre metodi statici:

```
EndOperation(Operazione opf)
EndOperation(Operazione opf, object output)
EndOperationWithErr(Operazione opf, Exception ex)
```

Il metodo che termina una transazione senza ritorno di dati è `EndOperation(Operazione opf)`:

```
public static void EndOperation(Operazione opf)
{
    if (opf.ExecutingMode == eLaunchMode.ACTIVEX)
    {
        string xml = Encode("NP", "NESSUNA");

        // Connetto e Disconnetto la pipe
        PipeClient pipeCCPlugin = new PipeClient();
        pipeCCPlugin.Connect("CoreClientPlugin");
        pipeCCPlugin.SendMessage(xml).ToString();
        // Se lancio la Nested senza leggere i dati
        // ritornati => ritorno al Plugin esito="NP"
        // (Non Pervenuto) e segnalazione="NESSUNA"
        pipeCCPlugin.Disconnect();
    }

    if (opf.ExecutingMode == eLaunchMode.DIRETTA)
    {
        // Se arriva segnalazione di chiusura (corretta) tolgo da
        // lista pipe attive e cancello lo stackTrace
        if (pipeTrxs.ContainsKey(opf.PipeCore.PipeID))
        {
            pipeTrxs.Remove(opf.PipeCore.PipeID);
        }
    }

    if (onClosingNestedOperation != null)
    {
        onClosingNestedOperation(opf.CodiceOPF);
    }
}
```

Per il momento tralasciamo quanto viene eseguito dal metodo nel caso in cui l'operazione sia lanciata da ActiveX, lo vedremo in maniera più approfondita nel prossimo capitolo, dedicato al CoreClientPlugin. Nel caso in cui la transazione sia stata lanciata in modalità diretta, è necessario rimuovere l'ID della named pipe utilizzata per la comunicazione con il CoreClient dal dizionario `pipeTrxs`, contenente un riferimento ad ogni pipe utilizzata. Questo non è necessario nel caso di lancio annidato, poiché in

questo caso il pipe ID è ereditato dalla transazione chiamante. Come ultima operazione se la transazione era di tipo nested, viene sollevato l'evento onClosingNestedOperation, che segnala alla chiamante la chiusura della transazione figlia.
Se la transazione necessita di ritornare dei dati, è stato creato un apposito override del metodo EndOperation, avente un parametro aggiuntivo specifico per questo scopo:

```
public static void EndOperation(Operazione opf, object output)
{
    try
    {
        temporaryOutputDict = (Dictionary<string, object>)output;
    }
    catch (Exception ex)
    {
        // LOG dell'eccezione
    }

    if (opf.ExecutingMode == eLaunchMode.ACTIVEX)
    {
        // output è un dictionary che DOVREBBE contenere due campi:
        // "axEsito" e "axSegnalazione"
        string xml = string.Empty;
        try
        {
            // Unboxing del dizionario di output
            Dictionary<string, object> response = (Dictionary<string, object>)output;

            string axEsito = "NP";
            if (response.ContainsKey("axEsito"))
            {
                axEsito = response["axEsito"].ToString();
            }

            string axSegnalazione = "NESSUNA";
            if (response.ContainsKey("axSegnalazione"))
            {
                axSegnalazione = response["axSegnalazione"].ToString();
            }

            xml = Encode(axEsito, axSegnalazione);
        }
        catch
        {
            xml = Encode("ERR", "NESSUNA");
        }

        // Connetto e Disconnetto la pipe
        PipeClient pipeCCPlugin = new PipeClient();
        pipeCCPlugin.Connect("CoreClientPlugin");
        pipeCCPlugin.SendMessage(xml).ToString();
        // Se lancio la Nested senza leggere i dati
        // ritornati => ritorno al Plugin esito="NP"
        // (Non Pervenuto) e segnalazione="NESSUNA"
        pipeCCPlugin.Disconnect();
    }

    if (onClosingNestedOperationWithArgs != null)
    {
        onClosingNestedOperationWithArgs(opf.CodiceOPF, output);
    }
}
```

Come nel caso precedente, tralasciamo il caso in cui il lancio sia avvenuto da ActiveX; inoltre, essendo il ritorno dei parametri permesso solamente alle transazioni annidate, non è stato necessario inserire un controllo sul tipo di lancio. Il metodo tenta di valorizzare preventivamente il dizionario temporaryOutputDict attraverso l'unboxing dell'oggetto output, nell'eventualità di lancio nested da transazione VB6. Al termine, il metodo solleva l'evento onClosingNestedOperationWithArgs, passando come secondo parametro il dizionario dei dati ritornati ancora sottoposto a boxing.
Se la transazione deve terminare a causa di un'eccezione, viene utilizzato il metodo statico EndOperationWithErr, che accetta come secondo parametro un oggetto di tipo Exception contenente informazioni sulla natura dell'eccezione:

```
public static void EndOperationWithErr(Operazione opf, Exception ex)
{
    // Per permettere il completamento del flusso di navigazione
    // devo necessariamente sollevare uno tra i due eventi
    // onClosingNestedOperation e onClosingNestedOperationWithArgs

    if (onClosingNestedOperation != null)
    {
        onClosingNestedOperation(opf.CodiceOPF);
    }
    else if (onClosingNestedOperationWithArgs != null)
    {
        // Di norma viene ritornato un dictionary => ne creo uno
        // contenente l'eccezione
        Dictionary<string, object> nestedOutput = new Dictionary<string, object>();
        nestedOutput["NEST_Exception"] = ex;
        onClosingNestedOperationWithArgs(opf.CodiceOPF, nestedOutput);
    }
}
```

In questo caso non si fanno differenze sul tipo di chiamata: si controlla a quale dei due eventi di chiusura la transazione si è sottoscritta (onClosingNestedOperation o onClosingNestedOperationWithArgs), quindi solleva l'evento trovato. Nel caso in cui la scelta sia ricaduta su onClosingNestedOperationWithArgs, viene valorizzato un campo del dizionario (NEST_Exception) contenente l'eccezione. E' importante notare che, per leggere tale campo, il Datamodel della transazione chiamante deve necessariamente possedere una proprietà con lo stesso nome.

8 CoreClientPlugin

Come appena visto nel capitolo precedente, il CoreClientPlugin non è solo un componente che consente l'avvio di transazioni da pagine web, ma fornisce anche supporto per l'interoperabilità nel caso di lancio di transazioni annidate dal mondo COM a quello .NET. In questo capitolo ci concentreremo maggiormente sul primo aspetto.

8.1 Componenti ActiveX

Dalla versione 3.0 di Internet Explorer, gli utenti hanno la possibilità di aggiungere del codice alle loro pagine web sotto forma di controlli ActiveX. Per posizionare un controllo ActiveX su una pagina web, si utilizza il tag HTML `<OBJECT>`, al cui interno vengono forniti al browser gli estremi per l'identificazione del controllo e tutte le specifiche per una corretta inizializzazione dei relativi parametri. Un esempio potrebbe essere:

```
<OBJECT ID="ControlloEsempio"
   WIDTH="157px"
   HEIGHT="47px"
   CLASSID="CLSID:2CD4FB81-E7BD-12CE-A3DC-422663580000">

   <PARAM NAME="_Version" VALUE="1.0">
   <PARAM NAME="_Caption" VALUE="Esempio">
   <PARAM NAME="_ForeColor" VALUE="">
   <PARAM NAME="_BackColor" VALUE="">
</OBJECT>
```

Quando un client carica la pagina web in cui è presente il tag `<OBJECT>`, il browser si assicura che il controllo esista sulla macchina, altrimenti lo carica attraverso la rete e lo installa. Quando il browser è certo della presenza del controllo, ne lancia setta i parametri utilizzando le proprietà specificate dai tag `<PARAM>`, quindi lo avvia. Il controllo sarà gestito dal browser attraverso un linguaggio di script che permette di utilizzare un controllo ActiveX come un normale oggetto. Al termine dell'esecuzione, il controllo rimane installato sul sistema client, rendendo più agevole e rapido un eventuale prossimo utilizzo.
Il meccanismo di lancio di una transazione da pagina web prevede che in questa sia registrato il componente ActiveX CoreClientPlugin:

```
<OBJECT ID="CoreClientPlugin"
     WIDTH = "0px"
     HEIGHT = "0px"
     CLASSID = "CLSID:A738ECEC-9EFE-4FBA-8A7B-C4872AF6FFF1"/>
```

Dove il `CLASSID` deve coincidere con quello specificato nelle proprietà di progetto del CoreClientPlugin, come è possibile vedere in Fig. 8.1:

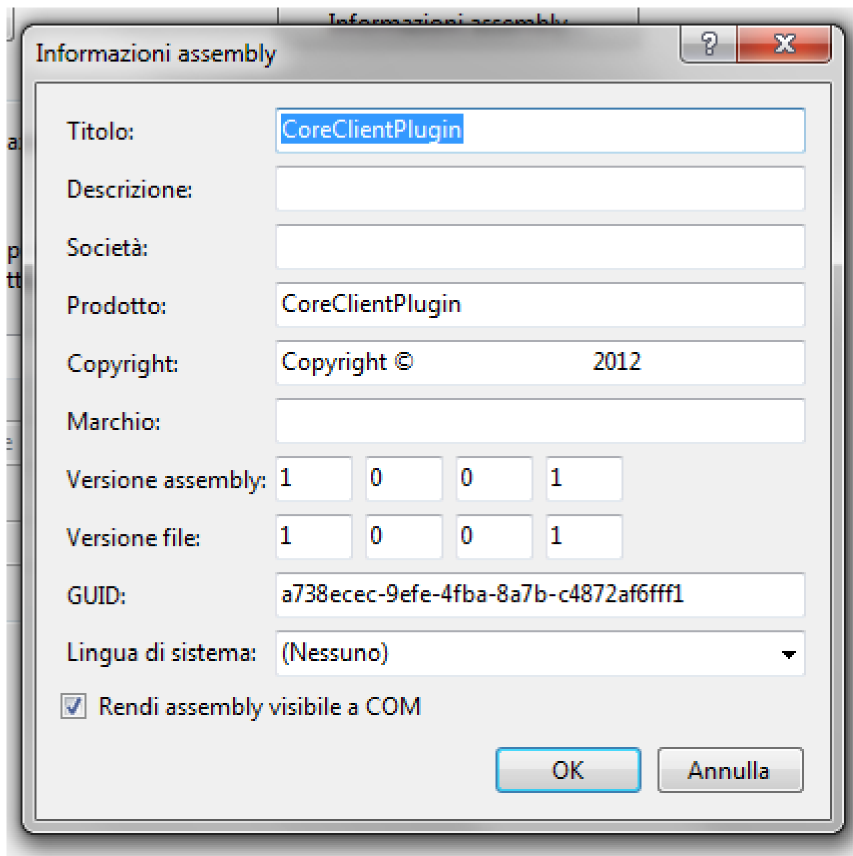

Fig. 8.1 – Informazioni di progetto del CoreClientPlugin

Il `CLASSID` è necessario per recuperare la classe che effettivamente implementa l'interfaccia esposta dall'oggetto ActiveX, denominata `IPluginEvents`:

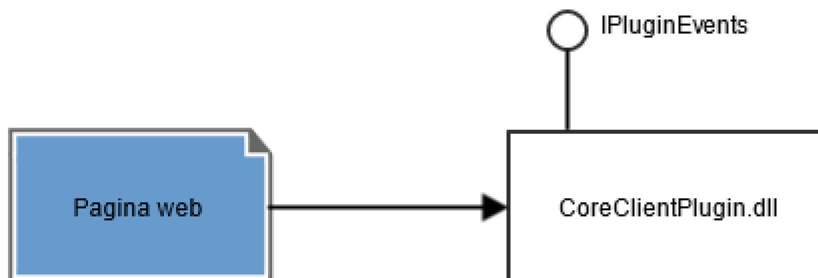

Fig. 8.2 – Interfaccia IPluginEvents esposta dall'oggetto ActiveX

L'interfaccia `IPluginEvents` definisce la firma dei metodi e delle proprietà richiamabili da script:

```
[ComVisible(true)]
[Guid("34D0843D-6D37-4D88-B334-34CEB51FD569")]
[InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
public interface IPluginEvents
{
    #region DATI FISICI DELLA FILIALE

    [DispId(1)]
    void setCloneFisico(string cloneFisico);
    [DispId(2)]
```

```
void setFilialeFisica(string filialeFisica);
 [DispId(3)]
void setSportelloFisico(string sportelloFisico);

#endregion

#region DATI LOGICI DELLA FILIALE

[DispId(4)]
void setCloneLogico(string cloneLogico);
[DispId(5)]
void setFilialeLogica(string filialeLogica);
[DispId(6)]
void setSportelloLogico(string sportelloLogico);
[DispId(7)]
void setMatricola(string matricola);
[DispId(8)]
void setDataSolare(string dataSolare);
[DispId(9)]
void setDataContabile(string dataContabile);

#endregion

#region CAMPI DI SERVIZIO

[DispId(10)]
void setCloneOperante(string cloneOperante);
[DispId(11)]
void setCodABI(string codABI);
[DispId(12)]
void setFiliale(string filiale);
[DispId(13)]
void setMatricolaOperante(string matricolaOperante);
[DispId(14)]
void setTipologiaCTR(string tipologiaCTR);
[DispId(15)]
void setCodicePTF(string codicePTF);
[DispId(16)]
void setTipoPTF(string tipoPTF);
[DispId(17)]
void setCodTipoIntervento(string codTipoIntervento);
[DispId(18)]
void setFLMultib(string flMultib);
[DispId(19)]
void setFLAltaPrioritastring(string flAltaPriorita);

#endregion

#region CAMPI USI FUTURI

[DispId(20)]
void setCodFunzione(string codFunzione);
[DispId(21)]
void setDataGestioneCompletate(string dataGestioneCompletate);
[DispId(22)]
void setFilialeCC(string filialeCC);
[DispId(23)]
void setCategoriaCC(string categoriaCC);
[DispId(24)]
void setConto(string conto);
[DispId(25)]
void setIBAN(string iban);
[DispId(26)]
void setNDG(string ndg);
[DispId(27)]
void setProgrFunzione(string progFunzione);
```

```
    #endregion

    #region CAMPI DI OUTPUT

    [DispId(28)]
    string getSegnalazione();

    #endregion

    [DispId(29)]
    void Launch(string acronimo);

    [DispId(30)]
    void EndOperationEvent();

    [DispId(31)]
    void setVersione(string versione);
}
```

La classe del CoreClientPlugin che implementa l'interfaccia `IPluginEvents` è denominata `Plugin`. Una particolarità degna di nota riguarda il fatto che la classe implementa anche l'interfaccia `IObjectSafety`, una scelta resa necessaria dalla natura "untrusted" dei linguaggi di script. Gli elementi dell'interfaccia che spaziano da quello etichettato con `[DispId(1)]` fino a quello con `[DispId(28)]` sono tutti metodi che permettono il `get/set` delle proprietà private della classe, e hanno tutti la stessa forma:

```
public void setCloneFisico(string cloneFisico) { _cloneFisico = cloneFisico; }
```

Questi campi sono legati alla logica funzionale che riguarda il passaggio delle informazioni da pagina web a transazione e viceversa. Dal punto di vista tecnico, i metodi più interessanti sono due:

- Plugin_Load(object sender, EventArgs e)
- Launch(string acronimo)

`Plugin_Load` viene chiamato all'avvio del CoreClientPlugin. In esso ritroviamo il controllo, effettuato con `CheckCoreActivity`, sull'attività del CoreClient: se per qualche motivo quest'ultimo non fosse presente tra i processi attivi, viene avviato.

```
private void Plugin_Load(object sender, EventArgs e)
{
    // 1. CoreClient è attivo? se non lo è, lo rilancio.
    Utils.CheckCoreActivity(true);
    // 2. init ABC con CoreClientPlugin
    pipeCore = new PipeServer ();
    pipeCore.MessageReceived += new PipeServer.MessageReceivedHandler(pipeCore_MessageReceived);
    pipeCore.Start("CoreClientPlugin");
}
```

Successivamente viene inizializzato un pipe server, del quale ci si sottoscrive all'evento `MessageReceived`, in attesa di eventuali segnalazioni da parte della transazione che si è in procinto di avviare. La sottoscrizione all'evento `MessageReceived` è utilizzata attualmente al solo scopo di sollevare un evento di terminazione della transazione che sia intercettabile da Javascript. Tale evento è denominato `EndOperationEvent`, ed è ovviamente definito nell'interfaccia `IPluginEvents`.
Il processo di connessione alla named pipe del CoreClientPlugin è diverso rispetto a quello solitamente utilizzato per gli altri metodi di lancio. In effetti, una volta

richiamato `pipeCore.Start`, il pipe server appena creato rimane in ascolto per eventuali connessioni, ma in quale momento queste vengono effettuate?
Per come è stato pensato l'avvio da pagina web, la risposta è: alla fine. La connessione al pipe server viene effettuata nei metodi dedicati alla terminazione della transazione del CoreClient, al solo scopo di passare due parametri: `axEsito` e `axSegnalazione`:

```
// Connetto e Disconnetto la pipe
PipeClient pipeCCPlugin = new PipeClient();
pipeCCPlugin.Connect("CoreClientPlugin");
pipeCCPlugin.SendMessage(xml).ToString();
// Se lancio la Nested senza leggere i dati
// ritornati => ritorno al Plugin esito="NP"
// (Non Pervenuto) e segnalazione="NESSUNA"
pipeCCPlugin.Disconnect();
```

la variabile `xml` è una stringa contenente una struttura XML della seguente forma:

```
<CORECLIENTRESPONSE>
    <ESITO>…</ESITO>
    <SEGNALAZIONE>…</SEGNALAZIONE>
</CORECLIENTRESPONSE>
```

I cui valori dei tag `<ESITO>` e `<SEGNALAZIONE>` variano al variare del tipo di lancio (nested con parametri, nested senza parametri). Se il lancio non prevede parametri di ritorno, `<ESITO>` conterrà `"NP"` (Non Pervenuto), mentre `<SEGNALAZIONE>` avrà valore `"NESSUNA"`.
L'altro caso, quello di chiamata nested con parametri di ritorno, prevede una valorizzazione dei tag da dizionario:

```
string xml = string.Empty;
try
{
    // cast dictionary output
    Dictionary<string, object> response = (Dictionary<string, object>)output;

    string axEsito = "NP";
    if (response.ContainsKey("axEsito"))
    {
        axEsito = response["axEsito"].ToString();
    }

    string axSegnalazione = "NESSUNA";
    if (response.ContainsKey("axSegnalazione"))
    {
        axSegnalazione = response["axSegnalazione"].ToString();
    }

    xml = Encode(axEsito, axSegnalazione);
}
catch
{
    xml = Encode("ERR", "NESSUNA");
}
```

Il metodo `Launch` avvia la transazione da pagina web, generando una stringa contenente i valori dei dati da passare alla transazione e utilizzandola come parametro per il metodo `InsertValue` di un'istanza della classe `YSocket`:

```
public void Launch(string acronimo)
{
    // Mostro il pannello di attesa
```

```
    YManager.showWaitPanel(644, 1024);

    // Init
    _segnalazione = string.Empty;

    // Log parametri ricevuti
    StringBuilder trace = new StringBuilder(800);
    trace.AppendLine("Dati ricevuti in input:");
    trace.AppendLine("---------------------------------------------");
    trace.AppendFormat("Clone Fisico:{0}{1}", _cloneFisico, Environment.NewLine);
    trace.AppendFormat("Filiale Fisica:{0}{1}", _filialeFisica, Environment.NewLine);

        .
        .
        .

    Logger.Log(Assembly.GetExecutingAssembly(), TraceEventType.Information,
                                            "Launch()", trace.ToString());
    trace.Clear();

    // Creazione stringa parametri per YSocket
    StringBuilder sb = new StringBuilder(120);
    sb.AppendFormat("{0}@", acronimo.Trim().ToUpper());  // GEALA@
    sb.AppendFormat("{0};", _cloneFisico);
    sb.AppendFormat("{0};", _filialeFisica);
    sb.AppendFormat("{0};", _sportelloFisico);
    sb.AppendFormat("{0};", _cloneLogico);
    sb.AppendFormat("{0};", _filialeLogica);
    sb.AppendFormat("{0};", _sportelloLogico);
    sb.AppendFormat("{0};", _matricola);
    sb.AppendFormat("{0};", _dataSolare);
    sb.AppendFormat("{0};", _dataContabile);
    sb.AppendFormat("{0};", _cloneOperante);
    sb.AppendFormat("{0};", _codABI);
    sb.AppendFormat("{0};", _filiale);
    sb.AppendFormat("{0};", _matricolaOperante);
    sb.AppendFormat("{0};", _tipologiaCTR);
    sb.AppendFormat("{0};", _codicePTF);
    sb.AppendFormat("{0};", _tipoPTF);
    sb.AppendFormat("{0};", _codTipoIntervento);
    sb.AppendFormat("{0};", _flMultib);
    sb.AppendFormat("{0};", _flAltaPriorita);
    sb.AppendFormat("{0};", _codFunzione);
    sb.AppendFormat("{0};", _progFunzione);
    sb.AppendFormat("{0};", _dataGestioneCompletate);
    sb.AppendFormat("{0};", _filialeCC);
    sb.AppendFormat("{0};", _categoriaCC);
    sb.AppendFormat("{0};", _conto);
    sb.AppendFormat("{0};", _iban);
    sb.AppendFormat("{0};", _ndg);
    sb.AppendFormat("{0};", _versione);

    YSocket socket = new YSocket();
    socket.InsertValue(sb.ToString(), "LAUNCH_OPNET");
    sb.Clear();
}
```

YSocket è il wrapper di un componente architetturale preesistente, chiamato SPISocket. Il suo scopo è quello di passare l'evento di avvio della transazione al CoreClient. A tale scopo, quest'ultimo si sottoscrive, all'avvio, all'evento onValueChanged di YSocket:

```
// Sottoscrizione all'evento onValueChanged di SPISocket
YSocket socket = new YSocket ();
socket.onValueChanged += new YSocket.YSocketValueChangedEventHandler(socket_ValueChanged);
```

L'evento, come abbiamo visto, viene sollevato nel metodo Launch del CoreClientPlugin, ma viene gestito dal CoreClient che, come prima cosa, controlla se la modifica dello stato di SPISocket è dovuta ad una richiesta di lancio da parte di pagina web:

```
void socket_ValueChanged(string bsLabel, string bsValue)
{
    if (bsLabel.Trim().ToUpper() == "LAUNCH_OPNET")
    {
        // Test Activex : se contiene @, quello che segue sono
        // i parametri per il lancio della transazione
        string[] splitted = bsValue.Split('@');
        string codopf = splitted[0];
        string parms = null;
        bool activex = false;

        if (splitted.Length == 2)
        {
            parms = splitted[1];
            activex = true;
        }
      .
      .
      .

        // Lancio da Portale o da ActiveX?
        if (activex)
        {
            // Converto stringa parametri in dictionary
            Dictionary<string, object> input = new Dictionary<string, object>();
            input["params"] = parms;

            Core.LaunchOperationFromActiveX(codopf, input);

            input.Clear();
        }
        else
        {
            // Lancio da Portale
             Core.LaunchOperation(codopf);
        }
    }
  .
  .
  .
}
```

Al termine di una serie di controlli sulla lanciabilità dell'operazione (non riportati perché estranei al meccanismo di lancio), il gestore dell'evento onValueChanged avvia la transazione utilizzando il metodo LaunchOperationFromActiveX visto nel capitolo 7, riguardante il CoreClient.

9 Conclusioni

L'architettura YAST presentata in questa relazione è funzionante ed è attualmente utilizzata nelle workstation delle filiali. Questa presenta dei notevoli punti di forza rispetto alla precedente infrastruttura, uno dei quali è certamente quello di ridurre quello che viene normalmente definito "Inferno delle DLL". Naturalmente, come per tutte le cose (e a maggior ragione nell'informatica), il semplice fatto che funzionino non implica necessariamente anche una perfetta esplicazione dei compiti alla quali sono deputate. Questo è in parte dovuto a dei limitanti vincoli hardware e software delle workstation messe a disposizione, che in più occasioni si sono rivelate uno dei fattori di maggior impatto sulle prestazioni dell'architettura. D'altra parte, anche se YAST agevola notevolmente lo sviluppo e la gestione delle transazioni, non sono mancate le occasioni per rendersi conto che l'architettura può ancora essere migliorata. Ad esempio, molti passaggi necessari allo sviluppo di una transazione sono obbligatori, ma non sono stati resi parte di un contratto definito da un'interfaccia, o implementati fin dall'inizio con dei metodi virtuali. Un altro punto degno di nota sono i controlli custom creati appositamente per le transazioni: alcuni di essi presentano tutt'oggi delle piccole imperfezioni che rendono lo sviluppo di certe funzionalità un po' complicato. La creazione di controlli customizzati che rispondano a particolari esigenze degli utilizzatori è un compito da non sottovalutare, poiché una volta che un controllo è utilizzato in più transazioni già distribuite una eventuale modifica può portare a comportamenti inattesi proprio nel punto più delicato delle applicazioni: l'interfaccia utente. Questo, beninteso, è valido per il rilascio di qualsiasi componente architetturale.
Il periodo di tirocinio è stato fondamentale per insegnarmi questo modo di progettare "distribuito", un allenamento a ragionare su quali impatti possa portare una semplice (o ritenuta tale) modifica sull'intera struttura. Ho avuto inoltre l'occasione di conoscere meglio il mondo COM, una tecnologia che, con l'avvento del Framework .NET è stata ampiamente sorpassata, ma che è ancora di importanza fondamentale per il largo utilizzo che ne è stato fatto, senza contare che i componenti COM sono tuttora più prestanti dei loro corrispondenti .NET, e possono quindi essere utilizzati in applicazioni dove le prestazioni devono necessariamente essere ottimali.
Il lavoro di sviluppo in team necessita di strumenti appositi per la gestione del codice e dei progressi effettuati dal singolo su software in lavorazione da più persone, ad esempio permettendo di eseguire il merge delle modifiche apportate da più sviluppatori sullo stesso file senza doverlo fare a mano. Ho quindi utilizzato strumenti per il controllo della versione come TortoiseSVN, che più di una volta si è rivelato provvidenziale per il recupero di lavoro che altrimenti sarebbe andato perduto, e per la gestione della configurazione del software come IBM Rational ClearCase.
Un ringraziamento è dovuto alle persone che mi hanno assistito durante questa mia prima esperienza lavorativa, le quali hanno dimostrato una grande professionalità e hanno saputo coinvolgermi nelle attività, insegnandomi con infinita pazienza non solo a lavorare, ma anche tutto quello che riguarda la vita in azienda.

[1] Christian Nagel, Bill Evjen, Jay Glynn, Karli Watson, Morgan Skinner, *Professional C# 4 e .NET 4*, Wiley Publishing, Inc. Indianapolis, Indiana, 2010 (Ulrico Hoepli Editore, Milano, 2010).

[2] Francesco Balena, *Microsoft Visual Basic 6*, Arnoldo Mondadori Editore, Trento, 2001.

[3] Matthew Cochran, *C# Heap(ing) Vs Stack(ing) in .NET: Part I* - http://www.c-sharpcorner.com/UploadFile/rmcochran/chsarp_memory401152006094206AM/chsarp_memory4.aspx.

[4] Mickey Williams, *Inside the .NET Managed Heap* - http://www.codeguru.com/columns/dotnet/article.php/c6593/Inside-the-NET-Managed-Heap.htm.

[5] Bharat Kinariwala, Tep Dobry, *Programming in C* - http://www-ee.eng.hawaii.edu/~tep/EE160/Book.

[6] Steven Hollidge, *Garbage Collector .NET 4* - http://stevenhollidge.blogspot.it/2012/04/garbage-collector-net-4.html.

[7] Rob Paveza, *Speedy C#, Part 2: Optimizing Memory Allocations – Pooling and Reusing Objects* - http://geekswithblogs.net/robp/archive/2008/08/07/speedy-c-part-2-optimizing-memory-allocations---pooling-and.aspx.

[8] Abhishek Sur, *Memory Management in .NET* - http://www.codeproject.com/Articles/38069/Memory-Management-in-NET.

[9] Jeffrey Richter, *Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework* - http://msdn.microsoft.com/en-us/magazine/bb985010.aspx.

[10] Shivprasad Koirala, *.NET Best Practice No: 2 – Improve garbage collector performance using finalize/dispose pattern* - http://www.codeproject.com/Articles/39246/NET-Best-Practice-No-2-Improve-garbage-collector.

[11] Joydip Kanjilal, *When and How to Use Dispose and Finalize in C#* - http://www.devx.com/dotnet/Article/33167.

[12] Ralf Sudelbücher, *A truly simple example to get started with WCF* - http://weblogs.asp.net/ralfw/archive/2007/04/14/a-truely-simple-example-to-get-started-with-wcf.aspx.

[13] Dino Esposito, *C# 4.0, the Dynamic Keyword and COM* - http://msdn.microsoft.com/en-us/magazine/ff714583.aspx.

[14] www.esri.com, *Introduction to COM* -
http://resources.esri.com/help/9.3/arcgisdesktop/com/COM/IntroToCOM.htm.

[15] Microsoft Corporation and Digital Equipment Corporation, *The Component Object Model Specification, Draft Version 0.9*, Microsoft Corporation, 1995 -
http://www.daimi.au.dk/~datpete/COT/COM_SPEC/html/com_spec.html.

[16] Giuseppe Dio, *I controlli ActiveX* - http://users.libero.it/giudio/tech/activex.html.

[17] Riccardo Golia, *Un'introduzione alla Reflection* -
http://www.aspitalia.com/articoli/asp.net/reflection.aspx.