1. ESTENSIONE DELLA PROGRAMMAZIONE AD OGGETTI

Limiti della programmazione ad oggetti

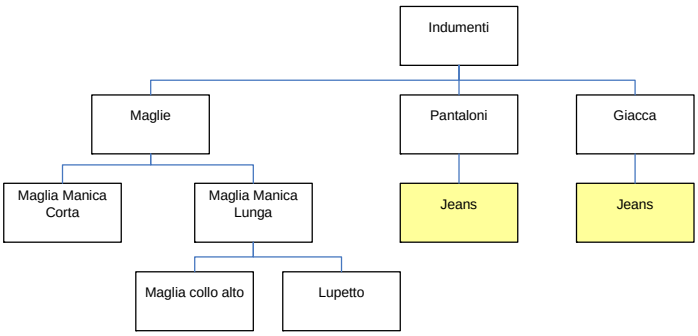
Sistemi di classificazione gerarchici

Dato un criterio o una particolare funzione, è possibile definire un sistema gerarchico di classificazione. In genere non esiste un criterio migliore di un'altro, tutto dipende dall'aspetto che si vuole evidenziare, e quindi dallo scopo della classificazione.

La classificazione degli esseri viventi è basata su un sistema gerarchico che tiene conto di fattori strutturali e morfologici. I cetacei sono una sotto-categoria dei mammiferi perchè sono dotati di polmoni, hanno il cuore a quattro cavità, e allattano al seno. Poiché i cetacei si sono dovuti adattare all'ambiente acquatico, possiedono dei tratti strutturali che li rendono più simili ai pesci che ai mammiferi per quello che riguarda gli arti e la capacità di movimento. Cambiando sistema di classificazione, ad esempio tenendo conto dell'ambiente e habitat di appartenenza (aria, terra, mare), squali e balene apparterrebbero allo stesso gruppo.

In questo sistema, ogni oggetto appartiene ad uno e una e una sola classe, e ogni classe è legata ad una classe padre, dalla quale "eredita" tutte le caratteristiche. Il sistema può quindi funzionare, se e solo se le caratteristiche aggiuntive di ogni classe figlia sono strettamente dipendenti dalle caratteristiche della classe padre dalla quale deriva, e normalmente questo vincolo impone che si possa classificare gerarchicamente un solo aspetto per ogni classe di oggetti.

lo schema a fianco illustra un ipotetico sistema di classificazione per gli indumenti. La classe "Maglia Manica Corta" è strettamente dipendente dalla classe padre "Maglia", in quanto sole le maglie hanno il concetton di manica, e in particolare, di manica corta. La classe jeans invece non è una particolarità strettamente legata al pantolone: essendo una caratteristica del tessuto può essere valida per ogni classe di indumenti, e quindi non dovrebbe essere rappresentata in questo albero.



Natura composta degli oggetti

La maggior parte degli oggetti del mondo reale è costituito da un aggregato di elementi, e ogni elemento dell'aggregato ha generalmente una propria origine e storia evolutiva. La natura composta di un oggetto rende difficile la classificazione gerarchica, in quanto ogni componente rappresenta un aspetto che potrebbe essere preso in considerazione per il raggruppamento.

Secondo la suddivisione classica motocicli e bicicli appartengono a categorie diverse. Nonostante le evidenti somiglianze tra questi due mezzi, (la trasmissione a catena, il numero di ruote, il sistema frenante, il manubrio, etc), spesso risulta utile separare i veicoli a motore da quelli senza motore. Cambiando il sistema di classificazione, ad esempio mettendo assieme i veicoli in base alla loro "forma", bicicli e motocicli apparterrebbero allo stesso gruppo (veicoli a due ruote), ma motocicli e quad sarebbero in gruppi diversi, nonostante condividano praticamente tutto eccetto il numero di ruote

Spesso gli oggetti vengono modellati tenendo conto solo della parte esterna e funzionale e ignorando la composizione interna. Questo porta a creare insiemi e gerarchie di classi che si basano sul ruolo / uso che viene fatto dall'oggetto, piuttosto che sulla sua reale struttura. Attraverso l'ereditarietà, è possibile creare una nuova classe (sottoclasse) partendo da una classe esistente (superclasse), dando un'implementazione diversa ai metodi esistenti (overriding) o aggiungendone di nuovi. Ma nel mondo reale un oggetto "evolve" se e sole se uno dei suoi elementi costitutivi viene aggiunto o modificato. Lo scopo di sostituire / aggiungere elementi ad un oggetto è quello di aggiungere / modificare funzionalità.

Il telefono cellulare si evolve e diventa in grado di scattare foto. Dal punto di vista concettuale la classe "TelefonoConFoto" deriva dalla classe "Telefono", ed introduce il nuovo metodo "ScattaFoto".

Dal punto di vista fisico, il telefono ha una nuova funzione (scattare foto) perché è stato aggiunto un nuovo elemento costitutivo (la fotocamera)

Uno degli scopi principali dell'ereditarietà è quello del riuso del codice: nel momento in cui un elemento costitutivo è presente in più oggetti appartenenti a gerarchie diverse risulta inevitabile la duplicazione del codice, in quanto non è possibile individuare una classe base comune che contenga la caratteristica condivisa.

Macchina fotografica e telefono sono oggetti appartenenti a gruppi distinti. La classe "TelefonoConFoto" potrebbe trarre beneficio dalla classe "Macchina fotografica" in quanto molte delle funzioni implementate nella macchina fotografica sono utili anche al telefono. Purtroppo "TelefonoConFoto" deriva da "Telefono" e non ha nessuna classe base in comune con "Macchina fotografica" che può essere utilizzare per implementare le funzioni condivise.

Usando alcune tecniche di composizione / aggregazione[1] è possibile limitare la duplicazione del codice, ma costringendo a definire delle infrastrutture il cui costo è a volte superiore dei benefici portati. Uno dei problema di fondo è che non è possibile ignorare la natura composta di un oggetto, né è possibile appesantire il sistema creando centinaia di micro-classi / wrapper in modo da mantenere la descrizione fedele alla realtà.

Errori di definizione delle classi

Il sistema attributi / metodi attraverso il quale vengono definite le classi, tende ad unire due concetti che sono profondamente diversi e distinti: cos'è l'oggetto e cosa fa l'oggetto. Ogni definizione dovrebbe basarsi su elementi che sono oggettivi e immutabili, poiché ciò che un qualcosa è non può variare in base al contesto, o un criterio soggettivo. Il *cosa può fare* un oggetto è fortemente dipendente dal contesto, e dalla tecnologia, in quanto è sempre possibile trovare un nuovo modo per usare un oggetto. Gli aspetti che sono di natura dinamica e variabile non dovrebbero far parte di una definizione, che per sua natura è statica. La non distinzione tra questi due concetti, porta molto spesso a creare gerarchie di classi basate sul ruolo che un oggetto ha, piuttosto che su quello che l'oggetto è.

Una bicicletta è un mezzo di trasporto. Sembra ragionevole far derivare un'ipotetica classe "Bicicletta" da una classe base "MezzoDiTrasporto". Una bicicletta può essere collegata ad una dinamo, e tramite la forza esercitata sui pedali, produrre corrente. In questo caso la bicicletta si trasforma un generatore di corrente, e dovrebbe derivare da un'ipotetica classe "Generatore". Il modello è inesatto, entrambe le definizioni sono errate: la bicicletta è un oggetto con due ruote, con una catena che collega ruota a pedali, etc. L'essere mezzo o generatore è un possibile uso o servizio che l'oggetto bicicletta può fornire.

Nel definire un sistema ad oggetti, accade spesso che alcune funzioni individuate in fase di progettazione vengono tradotte in metodi delle rispettive classi, mentre altre, individuate in un secondo momento (a causa di una prima analisi superficiale o al sopraggiungere di nuove specifiche) vengano aggiunte usando le seguenti tecniche:

1. Utilizzando l'ereditarietà, creando una nuova classe che contiene la funzione aggiuntiva.
2. Creando una classe delega o wrapper che prende in input la classe da estendere e implementa l'estensione.
3. Modificando il codice sorgente.

La prima soluzione è profondamente sbagliata, e fallisce ad esempio nel seguente caso:

Una classe A viene specializzata in una classe A1 per poter implementare una funzione che A non ha previsto. In un secondo momento la classe A viene specializzato in una classe A2, sempre per poter implementare una funzione che A non ha previsto, diversa da quella di A1. In un terzo momento, è necessario avere una classe A3, che possieda le funzionalità di A1 e A2, più alcune specifiche di A3. Se si eredita da A, bisogna duplicare il codice di A1 e A2, se si eredita da A1, bisogna duplicare il codice di A2, se si eredita da A2, bisogna duplicare il codice di A1, se si eredita da entrambe, A1 e A2 si ritrovano con una classe base in comune (la A).

In genere non è corretto aggiungere funzionalità ad un oggetto attraverso l'ereditarietà, che dovrebbe essere utilizzata per creare una nuova entità, e non un'estensione. Un problema analogo si ha con il caso della persona studente o lavoratore, quando si tenta di esprimere questa specializzazione con l'ereditarietà (per dettagli consultare un qualsiasi testo che tratti OOP). Sia il *lavoratore* che lo *studente* possiedono funzionalità aggiuntive che in qualche modo estendono la *persona*, ma una non esclude l'altra, una persona può essere entrambe le cose. Il problema si presenta perché l'essere *studente* o *lavoratore*, non è una condizione statica (che quindi sarebbe corretto includere in una definizione), ma è una condizione dinamica, che dipende da come entra in relazione l'ipotetica classe *persona* con altre entità. L'essere *studen*te non implica essere "diverso" da una nomale *persona*, è una condizione temporanea dovuta al fatto che si è iscritta ad almeno un corso di studio. E' pur vero che l'essere *studente* in qualche modo conferisce alla

[1] Delgation pattern - http://en.wikipedia.org/wiki/Delegation_pattern

classe *persona* delle funzioni aggiuntive che normalmente non avrebbe, ma questo non può essere comunque espresso con l'ereditarietà.

La seconda soluzione sarebbe formalmente scorretta, poiché alcune funzioni rimarrebbero sottoforma di metodi della classe "originale", e altre sottoforma di metodi di una classe di supporto.

Un'azienda fornisce una libreria di image processing nella quale esiste una classe chiamata "Immagine". Questa classe contiene i metodi classici di elaborazione delle immagini, tra cui rotazione, ridimensionamento, correzione colore etc. Il programmatore necessita di implementare alcune funzioni avanzate di elaborazione delle immagini, che non sono presenti nella classe. Concettualmente, essendo metodi che manipolano l'immagine, dovrebbero apparire nella classe immagine, ma non possedendo i sorgenti e non potendo usare l'ereditarietà per i motivi sopra citati, sarebbe costretto a fare una classe con metodi che prendono come "input" l'immagine da elaborare.

Usare una classe con metodi statici che operano su un oggetto è equivalente ad avere delle funzioni che operano su una struttura dati, tornando, di fatto, ad un approccio a procedure / funzioni.

La terza soluzione non è sempre applicabile, poiché il sorgente non è sempre disponibile. Nel caso in cui lo fosse fallisce nel momento un cui il sorgente viene utilizzato da più sviluppatori indipendenti che non vogliono essere vincolati ad un progetto di gruppo:

Due sviluppatori indipendenti A e B applicano delle modifiche ad una classe. Se lo sviluppatore A volesse includere le modifiche effettuate da B, sarebbe costretto ad effettuare il merge manuale tra i due codici sorgenti per le seguenti ragioni:

B potrebbe avere modificato alcuni metodi che A vuole mantenere inalterati.

B potrebbe avere aggiunto delle estensioni che ad A non interessano e non vuole includere.

Il sorgente scritto da B potrebbe essere con uno stile totalmente diverso rispetto quello scritto da A

Le modifiche applicate da B potrebbero essere in concorrenza, o concettualmente simili con quelle effettuate da A

Ogni manutenzione al codice scritto da B da parte di B dovrebbe essere riportata manualmente da A

Anche nel caso in cui tutte le possibili estensioni potessero essere applicate facilmente a livello di sorgente, come risultato si avrebbero classi pesanti con decine di metodi, che in un uso "normale" risulterebbero assolutamente superflui.

Le estensioni devono essere applicare da moduli indipendenti che non alterano in alcun modo la struttura esistente, in modo da poter essere inclusi, scelti e aggiornati con facilità.

Uso improprio degli attributi

C'è differenza tra attributi che descrivono le proprietà intrinseche di un oggetto, e attributi che in qualche modo tengono traccia dell'uso e delle relazioni instaurate con altri oggetti.

Le proprietà intrinseche di un edificio sono quelle che ne descrivono la struttura, dimensioni, composizione, materiale di costruzione, collocazione geografica, etc. Il proprietario dell'edificio o la tipologia (magazzino, negozio, appartamento, etc) non dovrebbero essere proprietà dell'edificio, perché sono elementi che fanno riferimento all'uso che ne viene fatto, e non a quello che rappresenta.

Nella OOP non c'è alcuna distinzione tra queste due tipologie di attributi. Analogamente alle funzioni, anche le relazioni sono elementi dinamici e dipendenti dal contesto. Non è possibile durante la definizione di una classe prestabilire quali siano le relazioni, perché nel futuro potrebbero nascere nuove classi in grado di instaurare nuovi legami con essa. Inoltre ogni relazione coinvolge sempre due elementi, e sarebbe concettualmente scorretto se solo uno di essi ne conservasse traccia sottoforma di attributi.

Un album contiene dei brani musicali. Dal punto di vista della classe "album" i brani contenuti sono un attributo di tipo "collezione di brani". Invertendo le parti, un brano appartiene ad un album. Dal punto di vista del brano, l'album di appartenenza è un attributo di tipo "album".

Quale delle due classi deve dichiarare l'attributo-relazione? Formalmente è il contenitore che deve tenere traccia del contenuto, anche se spesso è altrettanto utile che il contenuto indichi qual è il suo contenitore.

Oggetti e servizi

Ci sono due principali ragioni per definire una nuova classe. La prima, è avere un'entità in cui raggruppare un insieme di attributi, la seconda è avere un'entità attraverso la quale compiere determinate operazioni, e quindi fornire determinati servizi. Non solo è sempre possibile trovare un nuovo modo per usare un oggetto, ma più oggetti possono avere lo stesso uso (o fornire lo stesso servizio). Riflettendo attentamente, i processi umani non sono mai vincolati ad un particolare utensile, ma bensì al servizio che l'utensile è grado di fornire, cosicché siamo in grado di adattare lo stesso processo anche disponendo di utensili diversi.

Bisogna dividere un foglio in 4 parti uguali. Attraverso una forbice è possibile effettuare un taglio verticale e successivamente altri due tagli orizzontali nelle due porzioni ottenute dal primo taglio. Se al posto di una forbice, avessimo un cacciavite piano, effettuando pressione su uno degli spigoli, potremmo tagliare il foglio, usando la medesima tecnica. Cacciavite e forbice sono oggetti diversi, ma entrambi, se usati opportunamente, possono fornire il servizio di "Taglio" sul quale si basa questo algoritmo.

Nella vita di tutti i giorni, ogni persona è perfettamente in grado di prendere un oggetto e utilizzarlo in modo diverso rispetto alla sua funzione originaria, senza alterarne la sua struttura: lo stesso dovrebbe valere nella programmazione.

Le interfacce permettono di esprimere parzialmente questi concetti, ma con due forti limitazioni. La prima è che, a differenza di una classe, l'interfaccia non ha associata alcuna implementazione, ma è una semplice collezione di metodi e poiché alcuni metodi possono avere implementazioni simili (o identiche) in oggetti diversi, la duplicazione del codice è spesso inevitabile. E' possibile ridurre un'interfaccia all'essenziale e delegare ad una classe esterna l'implementazione delle caratteristiche avanzate e condivise, ma questo approccio, come sarà meglio definito più avanti, può portare a diversi problemi. La seconda è che, se non è possibile mettere mano direttamente al sorgente, l'implementazione di interfacce in classi già definite è possibile solo attraverso l'ereditarietà, con tutti i problemi che questo tipo di approccio comporta.

Servizi dipendenti

Una classe difficilmente lavora isolata, per operare ha spesso bisogno di interagire con altri elementi del sistema, ovvero, necessita di alcuni servizi. Questi servizi possono essere mantenuti tramite attributi della classe stessa, essere passati come argomenti ai singoli metodi, essere esposti come metodi di classi statiche / singleton, oppure essere forniti da classi che vengono istanziate all'occorrenza.

Istanziare le classe-servizio all'occorrenza e passarle sottoforma di attributi / argomenti ha tre svantaggi. Il primo è che il programma rimane vincolati staticamente ad un'implementazione del servizio, scelta dal programmatore. Il secondo è che, ogni classe / metodo usa la propria istanza del servizio, nonostante sia possibile condividere una singola istanza tra più classi. Il terzo, è che ogni classe deve memorizzare sottoforma di attributo l'istanza della classe contenete il servizio necessario, aumentando le richieste di memoria.

Utilizzando le classi statiche, potrebbero essere risolti i problemi di memoria e di condivisione, ma i metodi delle classi statiche non sono polimorfici (non possono essere ridefiniti dalle sottoclassi), e sono di fatto equivalenti a funzioni all'interno di un namespace. L'approccio ibrido, ovvero creare una e una sola istanza della classe, e rendere accessibile questa istanza attraverso un metodo / proprietà statica all'interno della classe stessa [2], risolverebbe il problema del polimorfismo, ma non quello del vincolo statico ad una particolare implementazione.

Rappresentando ogni servizio con un'interfaccia e usando una classe-factory per istanziare la classe "giusta" che implementa l'interfaccia permetterebbe lo svincolamento da una particolare implementazione, ma una factory non può essere a conoscenza del contesto in cui la richiesta viene fatta, ed è comunque vincolata alle scelte che il programmatore ha fatto riguardo tipologie di classi che possono essere costruite.

Aldilà dei problemi di prestazioni, e delle scomodità di alcune di queste tecniche, l'approccio statico fallisce per due motivi: primo perché i sistemi e le tecnologie evolvono, e lo stesso servizio fornito da un oggetto può essere fornito in modo diverso in sistemi e tecnologie diverse. Secondo, perché in base al contesto in cui la richiesta del servizio viene fatta, ci possono essere implementazioni migliori di altre, e la scelta di "cosa è meglio in quel momento" dovrebbe essere presa dalla macchina che esegue il programma, piuttosto che dal programmatore. Collegando un'applicazione ad un servizio, e non ad una particolare implementazione, il sistema rimane all'avanguardia rispetto a futuri cambiamenti / ottimizzazioni.

Il linguaggio di programmazione dovrebbe permettere di formalizzare queste dipendenze, in modo che il sistema (in accordo con l'utente e il programmatore) possa decidere di volta in volta la classe "giusta" da assegnare per tale implementazione.

---

[2] Pattern singleton - http://it.wikipedia.org/wiki/Singleton

Operazioni

Nella OOP la classe incapsula nella stessa entità una struttura dati (attributi) e una serie di funzioni (metodi) in grado di operare su di essa. Questo modello presuppone che un metodo sia strettamente collegato ad una e una sola struttura dati, e che altre eventuali dipendenze debbano essere passate come argomenti. La classe / struttura che dichiara il metodo assume un ruolo decisamente più forte rispetto alle classi / strutture passate come argomento ai singoli metodi. Questa assunzione non è sempre possibile, poiché esistono alcune categorie di operazioni che agiscono su un gruppo di strutture, nelle quali nessuna ha un ruolo predominante sulle altre.

Bisogna stampare un oggetto. L'azione di stampa può essere vista da due punti di vista: il primo, è quello dell'oggetto che potrebbe avere un metodo "stampa", e aspettarsi come argomento la stampante sul quale effettuare la stampa. Il secondo è quello della stampante, che potrebbe anch'essa avere un metodo "stampa", e aspettarsi come argomento l'oggetto da stampare.

Delegando l'operazione alla stampante, questa potrebbe stampare solo gli oggetti e lei noti, dei quali conosce formato e formattazione. Delegando l'operazione all'oggetto, questo sarebbe costretto ad invocare solo operazioni "primitive" condivise tra tutte le stampanti, e non potrebbe usufruire di eventuali funzioni ottimizzate presenti su qualche modello di stampante (es. stampa di tabelle, codice a barre, etc). L'operazione *stampa* non appartiene a nessuno dei due oggetti in particolari, opera indistintamente con entrambi (e quindi dovrebbe comparire in entrambe).

Il linguaggio di programmazione dovrebbe prevedere il concetto di "operazione" fuori dal contesto di una particolare classe. L'operazione, una volta indicate le classi coinvolte, dovrebbe essere disponibile sottoforma di metodo di quest'ultime, e ciascun metodo richiedere come argomento le altre classi necessarie.

L'operazione O lavora sulle classi A, B, C. Nella classe A, l'operazione diventa un metodo che accetta come argomenti B, C, nella classe B un metodo che accetta come argomenti A, C, nella classe C un metodo che accetta come argomenti A,B.

Con le operazioni concettualmente svincolate da una particolare classe, il programmatore può facilmente dare un'implementazione diversa alla stessa operazione, in base alla tipologia di parametri che vengono passati [3]. Riprendendo il problema della stampante, potrebbe esistere sia la generica operazione *stampa*, che opera con un particolare oggetto e una stampante generica, sia quella specifica per una stampante particolare.

Un metodo, che concettualmente esegue una certa operazione, può avere comportamenti diversi in base al contesto in cui esso è invocato:

Il metodo "ToString" converte un oggetto in una stringa di testo. In una classe "Data" se invocato in Italia dovrebbe restituire la data in formato "giorno-mese-anno", mentre negli Stati Uniti "mese-giorno-anno".

Sempre il metodo "ToString" in un classe "Contatto", se invocato all'interno di una rubrica telefonica, dovrebbe restituire"nome – telefono", mentre all'interno di un campo "mittente di posta", dovrebbe restituire "nome – indirizzo di posta".

Il "contesto" non è qualcosa che può essere passato come argomento, perché dipende da fattori "dinamici", come ad esempio da quale punto è stato richiamato il metodo da *contestualizzare*. Il linguaggio di programmazione dovrebbe quindi prevedere la possibilità di definire contesti, e di poter dare implementazioni di un metodo specifiche in ogni contesto.

Un problema concreto

A conclusione di questa introduzione che critica i metodi proposti dalla attuale OOP, viene fornito un esempio nel linguaggio c# in cui vengono evidenziati in modo chiaro tutti i problemi sollevati.

Si vuole generalizzare il processo di IO da un generico canale. A tal fine viene definita una classe astratta `Stream`, con la seguente struttura:

```
public abstract class Stream
{
    protected long _position;
    protected long _length;

    public bool IsEof
```

[3] L'overloading dei metodi, permette di avere metodi con lo stesso nome, che si differenziano in base alla tipologia di argomenti, ma per poterne usufruire è necessario modificare la classe originale (quindi disporne dei sorgenti), o utilizzare l'ereditarietà (con i problemi elencati in precedenza)

```
    {
        get
        {
            return _position >= _length;
        }
    }

    public int Read(byte[] buffer, int count)
    {
        int i;
        for (i = 0; !IsEof && i < count; i++)
            buffer[i] = ReadByte();
        return i;
    }

    public string ReadLine()
    {
        char c;
        StringBuilder line = new StringBuilder();
        while (!IsEof && (c = (char)ReadByte()) != '\n')
            line.Append(c);
        return line.ToString();
    }

    public abstract byte ReadByte();
    ...
}
```

La classe non è completa, per semplificazione sono stati omessi i metodi che non sono necessari al fine di questo esempio. Come è possibile notare, la classe è astratta perché è necessario implementare il metodo `ReadByte` sul quale sono basati i metodi `Read` e i metodi `ReadLine`. La classe `File` può derivare da `Stream`, e implementare il metodo `ReadByte` invocando le funzioni fornite dal sistema operativo:

```
public class File : Stream
{
    protected string _fileName;
    protected int _fileDescriptor;

    public File(string fileName)
    {
        _fileName = fileName;
        _fileDescriptor = os_open(_fileName);
    }

    public override byte ReadByte()
    {
        byte[] buffer = new byte[1];
        if (os_read_file(_fileDescriptor, 1, buffer) == 0)
            throw new EofException();
        _position++;
        return buffer[0];
    }
    ...
}
```

Il metodo `ReadByte` fa uso della funzione `os_read_file`, che dato un file descriptor, permette di leggere un insieme di byte dal file, e restituirli su un buffer passato come argomento. Attualmente il metodo `Read` è basato su `ReadByte`, ma poiché l'implementazione di `ReadByte` è basata su una funzione che può leggere un insieme di byte, è possibile utilizzare questa funzione anche su `Read` anziché eseguire tante letture di un byte. Non è stato previsto che il metodo `Read` potesse essere indipendente da `ReadByte`, e quindi non è stato definita come virtuale. Nella classe `Stream` viene portata la seguente modifica:

```
public virtual int Read(byte[] buffer, int count)
```

Mentre nella classe `File` viene effettuato l'override del metodo `Read` e implementato nel seguente modo:

```
    public override int Read(byte[] buffer, int count)
    {
        int readCount = os_read_file(_fileDescriptor, buffer, count);
        _position += readCount;
        return readCount;
    }
```

A questo punto il metodo `ReadByte` potrebbe basarsi su `Read`, esattamente al contrario di come era stato pensato nella classe base:

```
    public override byte ReadByte()
    {
        byte[] buffer = new byte[1];
        if (Read(buffer, 1) == 0)
            throw new EofException();
        return buffer[0];
    }
```

Fino a questo punto tutto sembra funzionare correttamente, il sistema sembra robusto e in grado di far fronte alle possibili estensioni: attraverso l'ereditarietà ogni classe può sfruttare l'implementazione base di `Stream`, e fornire la propria versione specializzata.

Il sistema evolve, e ora è necessario modellare un insieme di classi per rappresentare le periferiche hardware. Viene definita una classe `Device` che contiene alcuni metodi e attributi comuni a tutte le periferiche.

```
public abstract class Device
{
    protected string _id;
    protected string _name;
    protected DeviceCategory _category;
    public abstract void StandBy();
    public abstract void Disable();
    public abstract void Enable();

    ...
}
```

Nel progettare la classe che rappresenta la porta seriale si conclude che questa è uno stream dal quale poter leggere / scrivere byte.

```
public class SerialPort : Device
{
    public SerialPort(int portNumber)
    {
        _id = "COM" + portNumber;
    }

    public Byte ReadByte()
    {
        return os_serial_port_read(_id);
    }

    ...
}
```

A questo punto sorge il primo problema. La porta seriale è una periferica, e quindi dovrebbe estendere `Device`, ma è anche uno stream, quindi dovrebbe estendere `Stream`. Il linguaggio OOP è ad ereditarietà singola, e non verranno presi in considerazione quelli ad ereditarietà multipla per i noti problemi che questi possono causare (es, quando l'insieme delle classi utilizzate per l'estensione hanno una classe base in comune[4]). Come prima soluzione sarebbe possibile restituire il comportamento di stream della classe `SerialPort` sottoforma di attributo.

```
public class SerialPortStream : Stream
{
    protected SerialPort _serialPort;

    public SerialPortStream(SerialPort serialPort)
    {
        _serialPort = serialPort;
    }

    public override byte ReadByte()
    {
        return _serialPort.ReadByte();
    }
}

public class SerialPort : Device
{
    protected SerialPortStream _stream;

    public SerialPort(int portNumber)
    {
        _id = "COM" + portNumber;
        _stream = new SerialPortStream();
```

[4]Diamond problem - http://it.wikipedia.org/wiki/Diamond_problem

```
    }

    public SerialPortStream Stream
    {
        get
        {
            return _stream;
        }
    }

    public Byte ReadByte()
    {
        return os_serial_port_read(_name);
    }
}
```

Questa soluzione presenta un errore concettuale, in quanto la porta seriale non contiene uno stream ma è uno stream: infatti la specializzazione in `SerialPortStream` è basata solo ed esclusivamente su metodi forniti da `SerialPort`. Molto probabilmente la classe `SerialPortStream` dovrà utilizzare attributi / metodi privati di `SerialPort`, ai quali non dovrebbe aver accesso. I linguaggi di programmazione hanno previsto "trucchi" per invadere lo spazio privato e protetto di una classe (friend in c++, e visibilità internal di c#, etc), annullando il concetto di black-box che dovrebbe essere alla base della programmazione ad oggetti.

Una seconda soluzione potrebbe essere di trasformare la classe `Stream` in un'interfaccia, e di implementare tale interfaccia all'interno della classe `SerialPort`.

```
public interface IStream
{
    byte ReadByte();
    int Read(byte[] buffer, int count);
    string ReadLine();
    bool IsEof { get; }
    ...
}
```

`Stream` deve essere modificata nel seguente modo:

```
public class Stream : IStream
{
    ...
}
```

A questo punto `SerialPort` può implementare `IStream`. L'implementazione dei metodi `Read` e `ReadLine`, è assolutamente identica a quella fornita in `Stream`. La duplicazione del codice è sempre da evitare, perché in caso di manutenzione / estensione / correzione bug, bisogna porre attenzione nel sincronizzare manualmente tutte le parti duplicate. I metodi condivisi potrebbero essere implementati da una classe delega per poi essere invocati nel seguente modo:

```
public static class StreamUtils
{
    public static string ReadLine(IStream stream)
    {
        char c;
        StringBuilder line = new StringBuilder();
        while (!stream.IsEof && (c = (char)stream.ReadByte(stream)) != '\n')
            line.Append(c);
        return line.ToString();
    }

    public static int Read(IStream stream, byte[] buffer, int count)
    {
        int i;
        for (i = 0; !stream.IsEof && i < count; i++)
            buffer[i] = stream.ReadByte();
        return i;
    }
}

public class SerialPort : Device, IStream
{
    public SerialPort(int portNumber)
    {
        _id = "COM" + portNumber;
    }

    public Byte ReadByte()
```

```
    {
        return os_serial_port_read(_name);
    }

    public virtual string ReadLine()
    {
        return StreamUtils.ReadLine(this);
    }

    public virtual int Read(byte[] buffer, int count)
    {
        return StreamUtils.Read(this, buffer, count);
    }
}
```

In questo modo la duplicazione di codice sarebbe limitata all'invocazione di un metodo di un'altra classe. E' possibile evitare questa sovrastruttura provando a semplificare l'interfaccia `IStream`, riducendola al minimo indispensabile. Dopo una analisi è possibile concludere che forse solo il metodo `Read` è veramente utile e importante, e che gli altri possono essere in qualche modo implementati basandosi su questo. L'interfaccia viene trasformata in questo modo:

```
public interface IStream
{
    int Read(byte[] buffer, int count);
    bool IsEof { get; }
    ...
}
```

E la classe `SerialPort` assume questo aspetto:

```
public class SerialPort : Device, IStream
{
    public SerialPort(int portNumber)
    {
        _id = "COM" + portNumber;
    }

    public int Read(byte[] buffer, int count)
    {
        for (int i = 0; i < count; i++)
            buffer[i] = os_serial_port_read(name);
        return count;
    }

    public bool IsEof
    {
        get { return false; }
    }
}
```

La classe base `Stream` cosi come pensata non ha più significato, e le funzioni `ReadByte` e `ReadLine` devono essere implementate da una classe esterna. Come prima soluzione sarebbe possibile utilizzare una classe statica `StreamReader`, simile a quella vista in precedenza:

```
public static class StreamReader
{
    public static string ReadLine(IStream stream)
    {
        char c;
        StringBuilder line = new StringBuilder();
        while (!stream.IsEof && (c = (char)ReadByte(stream)) != '\n')
            line.Append(c);
        return line.ToString();
    }

    public static byte ReadByte(IStream stream)
    {
        byte[] buffer = new byte[1];
        if (stream.Read(buffer, 1) == 0)
            throw new EofException();
        return buffer[0];
    }
}
```

Analizzando questa classe è possibile notare che il metodo `ReadLine` chiama ripetutamente `ReadByte`, e per ogni lettura è costretto ad allocare dinamicamente un buffer di un byte, con un notevole calo di prestazioni. Una possibile soluzione è quella di allocare staticamente il buffer, e renderlo fruibile sottoforma di attributo privato

```
public static class StreamReader
{
    protected static byte[] _buffer = new byte[1];

    public static byte ReadByte(IStream stream)
    {
        if (stream.Read(_buffer, 1) == 0)
            throw new EofException();
        return _buffer[0];
    }

    ...
}
```

ma in questo modo la classe non può essere utilizzata in concorrenza tra più thread. Oltre al problema di performance, c'è il fatto che ogni funzione necessita come argomento lo stream su cui operare, tornando di fatto ad un approccio "funzionale". Come seconda soluzione potrebbe essere creata una classe non statica, che riceva in fase di costruzione lo stream su cui operare (da istanziare quando è necessaria):

```
public class StreamReader
{
    protected byte[] _buffer = new byte[1];
    protected IStream _stream;

    public StreamReader(IStream stream)
    {
        _stream = stream;
    }

    public byte ReadByte()
    {
        if (_stream.Read(_buffer, 1) == 0)
            throw new EofException();
        return _buffer[0];
    }

    public string ReadLine()
    {
        char c;
        StringBuilder line = new StringBuilder();
        while (!_stream.IsEof && (c = (char)ReadByte(stream)) != '\n')
            line.Append(c);
        return line.ToString();
    }
}
```

Poiché ogni istanza è collegata al proprio stream è possibile applicare l'ottimizzazione sopra descritta. Ora senza tener conto della scomodità di dover creare due classi per eseguire una lettura "normale" da stream i problemi sembrano risolti, il programmatore può usufruire delle funzione "avanzate", ed è stata evitata la duplicazione del codice.

Ora è necessario fornire l'astrazione di stream anche per i buffer di memoria. A questo fine viene creata la classe `MemoryStream`, che implementa `IStream`, con la seguente struttura.

```
public class MemoryStream : IStream
{
    protected byte[] _buffer;
    protected int _position;

    public MemoryStream(byte[] buffer)
    {
        _buffer = buffer;
    }

    public int Read(byte[] buffer, int count)
    {
        int maxRead = Math.Min(count, _buffer.Length - _position);
        Buffer.BlockCopy(_buffer, _position, buffer, 0, maxCount);
        _position += maxRead;
        return maxRead;
    }

    public bool IsEof
    {
        get { return _position >= _buffer.Length; }
    }
}
```

Con l'accesso diretto alla memoria, le operazioni `ReadLine` e `ReadByte` potrebbero essere fatte con assoluta efficienza, ma purtroppo queste sono delegate alla classe `StreamReader`, e quindi non possono essere ridefinite in `MemoryStream`. Effettuare uno controllo all'interno di `StreamReader` per eseguire operazioni diverse a seconda del tipo di stream passato sarebbe a dir poco orribile, non è possibile applicare questo genere di estensioni modificando il sorgente (anche perché questo potrebbe non essere disponibile). E' possibile estendere `StreamReader` e creare la classe `MemoryStreamReader`, la quale può effettuare l'override di questi metodi ed implementarli in modo efficiente per la classe `MemoryStream`.

```
public class MemoryStreamReader : StreamReader
{
    public MemoryStreamReader(MemoryStream memStream)
        : base(memStream)
    {
    }

    public override string ReadLine()
    {
        MemoryStream memStream = _stream as MemoryStream;
        int index = memStream._position;
        while (index < memStream._buffer.Length && memStream._buffer[index] != '\n')
            index++;
        memStream._position += index;
        return BufferToString(memStream._buffer, memStream._position, index - memStream._position);
    }

    public override byte ReadByte()
    {
        MemoryStream memStream = _stream as MemoryStream;
        if (memStream.IsEof)
            throw new EofException();
        return memStream._buffer[memStream._position++];
    }
}
```

Se un metodo generico che opera su stream volesse utilizzare lo `StreamReader`, in base a quale criterio dovrebbe scegliere la classe giusta? Lavorando con interfacce generiche non è possibile conoscere se esistono e quali sono eventuali versioni "ottimizzate" specifiche per un tipo di stream.

La prima soluzione consiste nel modificare l'interfaccia `IStream` e definire un metodo `CreateStreamReader`, attraverso il quale ogni stream può costruire lo `StreamReader` più adatto.

```
public interface IStream
{
    int Read(byte[] buffer, int count);
    bool IsEof { get; }
    StreamReader CraeteStreamReader();
    ...
}
```

Modificare un interfaccia non è sempre possibile, sia per non disponibilità del sorgente, sia perché la libreria che dichiara l'interfaccia potrebbe essere già stata distribuita, e per ovvie ragione deve rimanere retro-compatibile. La seconda soluzione è delegare ad una factory la creazione dello `StreamReader` in base allo stream passato

```
public class StreamReaderFactory
{
    public static readonly StreamReaderFactory Instance = new StreamReaderFactory();

    protected StreamReaderFactory()
    {
    }

    public StreamReader Create(IStream stream)
    {
        if (stream is MemoryStream)
            return new MemoryStreamReader(stream);
        return new StreamReader(stream);
    }
}
```

Ma anche in questo caso, definendo nuove classi che implementano `IStream`, potrebbero esserci nuovi `StreamReader`, e sarebbe sempre necessario modificare il sorgente della classe factory. Una possibile modifica alla factory, potrebbe essere quella di introdurre una tabelle attraverso la quale il programmatore può associare a ciascun tipo di stream il suo `StreamReader`,

```
public class StreamReaderFactory
{
    protected Dictionary<Type, Type> _table;

    public static readonly StreamReaderFactory Instance = new StreamReaderFactory();

    protected StreamReaderFactory()
    {
        _table = new Dictionary<Type,Type>();
    }

    public void Register(Type streamType, Type streamReaderType)
    {
        _table[streamType] = streamReaderType;
    }

    public StreamReader Create(IStream stream)
    {
        Type streamReaderType;
        if (_table.TryGetValue(stream.GetType(), out streamReaderType))
            return (StreamReader)Activator.CreateInstance(streamReaderType, stream);
        return new StreamReader(stream);
    }
}
```

In alternativa, è possibile dichiarare un'interfaccia `IStreamReaderProvider`, con un unico metodo `CreateStreamReader` e da affiancare all'interfaccia `IStream`. La factory potrebbe controllare la presenza di questa interfaccia nello stream passato, e in caso affermativo utilizzare il metodo `CreateStreamReader`, altrimenti restituire lo `StreamReader` di default;

```
public interface IStreamReaderProvider
{
    StreamReader CreateStreamReader();
}

public class StreamReaderFactory
{
    public static readonly StreamReaderFactory Instance = new StreamReaderFactory();

    protected StreamReaderFactory()
    {
    }

    public StreamReader Create(IStream stream)
    {
        if (stream is IStreamReaderProvider)
            return ((IStreamReaderProvider)stream).CreateStreamReader();
        return new StreamReader(stream);
    }
}

public class MemoryStream : IStream, IStreamReaderProvider
{
    ...

    public StreamReader CreateStreamReader()
    {
        return new MemoryStreamReader(this);
    }
}
```

Per risolvere il problema in modo ottimo, è stato necessario definire 2 interfacce (`IStreamReaderProvider`, `IStream`), e 3 classi di supporto (`StreamReader`, `MemoryStreamReader`, `StreamReaderFactory`) .

Conclusioni

La OOP non riesce ad esprimere formalmente situazioni complesse, che sono comunque comuni e frequenti all'interno di un progetto. Non si rivela robusta nei sistemi dinamici, che evolvono e si estendono nel tempo. Ci sono molte carenze nel sistema di tipi che devono essere colmate da pattern spesso complessi e macchinosi che costringono il progettista a costruire decine di classi che distolgono l'attenzione dal vero problema da risolvere. Il fatto che con gli strumenti a disposizione sia stato comunque possibile risolvere ogni problema emerso, non significa che il modello sia di per se ottimo o giusto. Infatti ad esempio, nonostante sia possibile programmare "ad oggetti" anche usando un linguaggio che mette a disposizione solo funzioni e strutture dati, difficilmente si può affermare che tale linguaggio implementa il paradigma della programmazione ad oggetti.

Introduzione alla programmazione basata sui servizi

Il concetto di “programmazione ad oggetti” non è di per se sbagliato, ma nella sua attuale formulazione è stata data scarsa considerazione ai seguenti aspetti:

1. *Si può sempre trovare un nuovo modo per usare un oggetto:*
    a. *Un oggetto può stabilire sempre nuovi legami con altri oggetti*
    b. *Un oggetto può essere utilizzato anche per fini diversi rispetto a quelli per il quale è stato concepito*
2. *Un programma dipende quasi sempre da servizi generici, non da oggetti e piattaforme particolari*
3. *Un oggetto viene quasi sempre utilizzato per quello che fa, non per quello che è*
4. *Un oggetto fornisce servizi, e lo stesso servizio può essere fornito da oggetti diversi*
5. *Un oggetto è formato da un aggregato di elementi, ognuno dei quali gli conferisce una caratteristica o una funzionalità.*
6. *Lo stesso elemento costitutivo può comparire anche in più oggetti diversi e non correlati.*

Per avere una rappresentazione fedele alla realtà, ogni componente costitutivo di un oggetto dovrebbe essere a sua volta un oggetto. Questo tipo di modello non è concretamente realizzabile, e oltretutto risulta spesso superflue superare un certo livello di dettaglio. In base al punto 5, è possibile vedere questi componenti non tanto per la loro natura fisica, ma piuttosto per quella funzionale, cosi quello che nella realtà è un aggregato di oggetti, nel modello può diventare un aggregato di servizi.

Classi e servizi

Ogni oggetto appartiene ad un classe, ma la classe di per se non ne definisce le caratteristiche: poiché lo stesso oggetto può fornire servizi diversi, le caratteristiche (attributi e metodi) sono fortemente legati all’aspetto (o servizio) preso in considerazione.

In un programma di gestione magazzino, l’oggetto “stampante” viene trattato come un “articolo”, e quindi descritto con i soli attributi utili allo stoccaggio (codice articolo, dimensioni, peso, collocazione). Lo stesso oggetto stampante, in un programma di vendita online , viene descritto con attributi che evidenziano le specifiche tecniche (tecnologia di stampa, numero di colori, velocità, tipo di cavo, etc)

La classe rappresenta un insieme aperto nel quale è possibile raggruppare i servizi forniti dalla stessa entità. Una classe può essere estesa, ovvero in qualsiasi punto del programma è possibile implementare un nuovo servizio in una classe esistente, anche non disponendo della definizione originale. In questo modello l’ereditarietà tra classi perde di significato, in quanto le classi non sono altro che un insieme di servizi.

Ogni servizio è rappresentato da un interfaccia che ne descrive la struttura in termini di metodi, proprietà ed eventi. Un metodo rappresenta un’operazione, una azione che è possibile compiere. Un *metodo* può essere parametrizzato da una serie di argomenti e può restituire un valore per indicare l’esito o il risultato ottenuto. Una proprietà rappresenta una caratteristica, un aspetto di natura descrittiva e non funzionale. Le proprietà possono essere di sola lettura, o di lettura / scrittura. Ad una *proprietà* può essere associata una variabile in memoria o un blocco di codice definito dal programmatore per poter recuperare / impostare dinamicamente il valore (es. calcolandola in base ad altre proprietà). Un *evento* è un mezzo attraverso il quale è possibile notificare ad altri il verificarsi di un particolare avvenimento, dovuto normalmente ad un cambio di stato. In modo molto simile al metodo, anche l’evento può essere accompagnato da argomenti, per indicare con maggior dettaglio i cambiamenti avvenuti. Alcuni eventi sono generati direttamente dal sistema operativo, e corrispondono a segnali inviati da componenti hardware. Altri sono definiti dal progettista, e sono generati normalmente quando una proprietà cambia valore, o quando viene invocata una particola funzione. Attraverso questo meccanismo, è possibile eseguire codice solo nel momento in cui è veramente necessario, e reagire in tempo reale alle richieste e i cambiamenti.

Una volta definito il servizio attraverso la sua interfaccia, questo può essere implementato. L’implementazione può essere effettuata direttamente all’interno della classe che fornisce il servizio o esternamente nel caso in cui la stessa implementazione possa essere utile a più classi. Poiché lo stesso servizio può essere implementato in modi diversi, ogni implementazione esterna alla classe deve essere identificata da nome. Un servizio rappresenta un singolo aspetto di un oggetto, e quindi può essere implementato senza problemi attraverso un sistema gerarchico. In alternativa all’ereditarietà tra classi, è possibile definire delle implementazioni (anche parziali) “basi” per ogni servizio, dalle quali successivamente ereditare, ed eseguire l’overriding dei metodi Durante l’implementazione di un servizio è possibile definire anche variabili e metodi non pubblici da utilizzare come elementi di supporto, che potranno essere opzionalmente visibili anche quando l’implementazione è usata come base di altre implementazioni

Contesti e dipendenze

Ad ogni passo di esecuzione corrisponde un contesto, nel quale è presente un catalogo che associa ad ogni servizio, l'attuale implementazione. I contesti sono organizzati in un sistema gerarchico, e sono nell'ordine: sistema, modulo, classe, servizio, implementazione, metodo, blocco. In ognuno di questi contesti è possibile formalizzare i servizi dipendenti ed eventualmente esprimere una preferenza sulle classi che li devono fornire. Le dipendenze a livello di modulo sono risolte al caricamento del modulo in memoria, quelle a livello di classe ogni qual volta ne viene creata un istanza (e chiaramente dipendono dal contesto in cui questo avviene), mentre quelle di metodo ad ogni invocazione. Le preferenze vengono prese in considerazione solo se, nel momento in cui si entra in un contesto, nel catalogo quel servizio non ha ancora alcuna implementazione associata. Ogni volta che si entra in un nuovo contesto, quello precedente viene salvato in uno stack, per poi essere ripristinato all'uscita. Il contesto può cambiare in base a diversi fattori, e in particolare cambia passando da un metodo ad un'altro. I servizi dichiarati dalla classe che contiene il metodo hanno la priorità rispetto alle indicazioni contenuto nel contesto precedente, e quindi la classe diventa il fornitore di quei servizi, fin quando si resta in quel contesto.

E' possibile creare delle entità "contesto", in cui dichiarare un insieme di coppie servizi-implementazioni. Grazie a questo sistema, in qualsiasi punto del programma è possibile cambiare il contesto "normale", ed entrare in un contesto personalizzato, in cui magari per ragioni di sicurezza l'accesso ad alcuni servizi viene inibito, o controllato attraverso proxy. Nel contesto sistema è il sistema operativo a scegliere per ogni servizio, l'attuale implementazione, scelta che può essere anche personalizzata dall'utente.

Struttura del linguaggio

Nei paragrafi che seguono verrà fatto uso della parola contesto per indicare un dominio di validità. Ogni contesto ha un contesto padre, e può avere uno o più contesti figli o sotto-contesti. A meno di indicazioni diversi, ogni elemento dichiarato all'interno di un contesto ha validità solo in quel contesto e nei suoi contesti figli. Qui sotto un esempio dei principali contesti:

```
//Contesto modulo

var var1 : Int32;

function Function1()
{
    //Contesto funzione
}

interface IIntreface1
{
    //Contesto interfaccia
    function Function2();
}

implement IInterface1 as Interface1Implementation
{
    //Contesto implementazione interfaccia
}

on IInteface.Event()
{
    //Contesto evento
}

class Class1
{
    //Contesto classe
    implement IInterface1
    {
        //Contesto implementazione interfaccia
        var var2 : String;

        function Function2()
        {
            //Contesto funzione
            var var3 : String;
            if ( var2 > 10)
            {
                //Contesto blocco
                var var4 : String;
                implement IInterface2
                {
                    //Contesto implementazione interfaccia
                }
            }
        }
    }
}
```

Inoltre verrà spesso fatto uso della parola membro, per indicare un qualsiasi elemento in grado di contenere un valore (variabile, argomento o proprietà)

Primo programma, "Hello World"

Come tradizione in ogni linguaggio, il primo programma che sarà analizzato è quello per stampare "Hello world" a video.

```
require IModule

on IModule.load()
{
    require IConsole as c;
    c.WriteLine("Hello world");
}
```

Attraverso l'istruzione **`require`** viene fatta richiesta al sistema di una particolare interfaccia. Le richieste a livello di modulo vengono risolte al momento del caricamento del modulo in memoria, e generalmente sono rivolte verso quei servizi utili a più classi. Nell'esempio viene fatta richiesta del servizio `IModule`. Questo servizio, a meno di indicazioni

diverse, ha una implementazione predefinita generata dal compilatore. Successivamente è stato scritto un blocco di codice in ascolto sull'evento `load` del servizio `IModule`. Questo evento viene generato dal sistema operativo al caricamente del modulo, e corrisponde a tutti gli effetti al punto di ingresso del programma. All'interno del gestore del evento, è stata fatta richiesta del servizio console (`IConsole`) e tramite la clausola **`as`** è stato specificato un alias attraverso il quale accedere al servizio. E' stato infine invocato il metodo `WriteLine` del servizio console, per emettere la stringa di testo "Hello world" nella console.

Funzioni

Attraverso l'istruzione **`function`** è possibile dichiarare una funzione. L'istruzione è valida sia nel contesto di modulo, sia in quello di classe o interfaccia (nel quale prende il nome di metodo). Una funzione può avere uno o più argomenti, e restituire un valore. Ogni argomento può avere un valore predefinito, e in tal caso può essere omesso durante la chiamata. Ad esempio:

```
function Int32ToHex(value : Int32, digitCount : Int32 = 8) : String
{
    ...segue implementazione...
}

on IModule.load()
{
   require IConsole;
   IConsole.WriteLine(Int32ToHex(28));
}

Output: 0000001C
```

E' stata dichiarata una funzione di nome `Int32ToHex` con due argomenti di tipo `Int32`, che ritorna una stringa. L'argomento `digitCount` ha come valore predefinito 8, e quindi può essere omesso durante la chiamata effettuata nell'evento `load`.

Due funzioni possono avere lo stesso nome all'interno dello stesso contesto, a patto che differiscano nel numero o tipologia di argomenti. Con l'istruzione **`return`** è possibile uscire dalla funzione, restituendo il controllo al chiamate e opzionalmente specificare un valore di ritorno. Il valore di ritorno è sempre mantenuto nello pseudo-argomento **`retvalue`**, al quale è possibile accedere anche in scrittura, per poter memorizzare valori temporanei / predefiniti senza uscire dalla funzione. Nel caso in cui la funzioni ritorni un oggetto, l'oggetto può essere creato o dal chiamante o dalla funzione stessa. Ad esempio:

```
struct Point
{
   X : Int32;
   Y : Int32;
}

function ParsePoint(text : String) : Point
{
   if (retvalue is null)
      retvalue = new Point();
   retvalue.X = Int32.Parse(text.Substring(0, text.IndexOf(“,”) - 1));
   retvalue.Y = Int32.Parse(text.Substring(text.IndexOf(“,”)));
}

on IModule.load()
{
    var p1 : Point
    var p2 : Point;

    ParsePoint(“10, 20”) in p1;
    p2 = ParsePoint(“10, 20”);
}
```

E' stata dichiarata una struttura `Point` e una funzione `ParsePoint` che accetta come argomento una stringa, e restituice l'oggetto `Point` corripondente (la stringa è costituita da una coppia di numeri separati da virgola). La funzione controlla se è stato passato un oggetto in cui memorizzare il valore di ritorno e in caso contrario ne crea uno nuovo. Nell'evento `load`, viene invocata `ParsePoint` indicando con la clausola **`in`** che il valore di ritorno deve essere memorizzato in `p1`. Nella seconda chiamata non viene fornita alcuna indicazione, per cui la funzione `ParsePoint` crea un nuovo oggetto `Point`, successivamente copiati nella variabile `p2`.

Interfaccie, classi e strutture

Le interfacce possono essere dichiarate solo nel contesto di modulo attraverso l'istruzione **`interface`**. Ad esempio:

```
interface IPlaneLocation
{
    property X : Int32;
    property Y : Int32;
    function GetDistance(other : IPlaneLocation) : Int32;
}
```

Con l'interfaccia `IPlaneLocation` viene individuato un oggetto nel piano attraverso una coppia di coordinate. Sono state dichiarate due proprietà `X` e `Y` di tipo `Int32` , e un metodo `GetDistance` che prende come argomento una locazione, e restituisce la distanza dalla locazione corrente.

Un'interfaccia può ereditare da un'interfaccia esistente. L'ereditarietà tra interfacce deve essere intesa nel seguente modo: se l'interfaccia A estende l'interfaccia B, significa che chi implementa A deve implementare anche B.

```
interface IStream
{
     function Write(text : String);
}

interface IFile : IStream
{
    property Path : String;
}
```

L'interfaccia `IFile`, estende `IStream`: chi implementa `IFile`, deve implementare anche `IStream`.

E' possibile implementare un'interfaccia in qualsiasi contesto attraverso l'istruzione **`implement`**. Nel esempio che segue, viene mostrato l'uso di **`implement`** all'interno di una classe:

```
class Point
{
    public implement IPlaneLocation
    {
        function GetDistance(other : IPlaneLocation) : Int32
        {
            return (sqrt(pow(X - other.X, 2) + pow(Y - other.Y, 2)));
        }
    }
}
```

La classe `Point` implementa `IPlaneLocation`. Il modificatore **`public`** indica che i metodi e le proprietà dell'interfaccia sono visibili all'esterno della classe. E' possibile notare che le proprietà `X`, `Y` non sono state esplicitamente dichiarate: in questo caso il compilatore genera automaticamente l'implementazione standard che legge e scrive la rispettiva variabile privata auto-generata. Poiché questa implementazione di `IPlaneLocation` potrebbe essere utilizzata anche da altre classi, è possibile dichiarala a livello modulo e assegnargli un nome con la clausola **`as`**:

```
implement IPlaneLocation as BasePlaneLocation
{
    ... (segue l'implementazione) ...
}
```

coloro che vogliono usare una implementazione esistente, possono specificarla con la clausola **`use`**:

```
class Point
{
    public implement IPlaneLocation use BasePlaneLocation;
}
```

I metodi definiti nell'implementazione base possono essere ridefiniti utilizzando la clausola **`override`**. E' possibile accedere al metodo della versione base, utilizzando **`base`** prima del nome. Ad esempio:

```
class Point
{
    public implement IPlaneLocation use BasePlaneLocation
    {
        override function GetDistance(other : IPlaneLocation) : Int32
        {
             if (X == other.X)
                 return abs(other.Y – Y);
             return base.GetDistance(other);
        }
    }

}
```

Anche un rettangolo potrebbe implementare `IPlaneLocation`, considerando o il punto centrale o l'angolo in alto a sinistra. Nell'esempio che segue viene dichiarata l'interfaccia `IRect`, e una classe `Rect` che implementa sia `IRect` che `IPlaneLocation`:

```
interface IRect
{
    X : Int32;
    Y : Int32;
    Width : Int32;
    Height : Int32;
}

class Rect
{
    public implement IRect;

    implement IPlaneLocation use BasePlaneLocation
    {
        override property X : Int32
        {
            get { return IRect.X + IRect.Width / 2; }
            set { IRect.X = value – IRect.Width / 2; }
        }

        override property Y : Int32
        {
            get { return IRect.Y + IRect.Height / 2; }
            set { IRect.Y = value – IRect.Height / 2; }
        }
    }
}
```

L'implementazione di `IPlaneLocation` è basata su `BasePlaneLocation`, ma le proprietà `X` e `Y` sono state ridefinite. Tramite il blocco **`get`** è possibile specificare il codice per recuperare il valore della proprietà. Il valore deve essere restituito con l'istruzione **`return`** analogamente a quello che avviene in un metodo. Tramite il blocco **`set`** è invece possibile specificare il codice per assegnare il valore della proprietà. Il valore attuale da assegnare è passato nello pseudo-argomento **`value`**. Per ambiguità con le omonime proprietà di `IPlaneLocation`, è stato necessario specificare il nome dell'interfaccia per accedere alle proprietà `X` de `Y` di `IRect`. L'interfaccia `IPlaneLocation` può essere implementata anche considerando l'angolo in alto a sinistra anziché il punto mediano. In questo caso è sufficiente associare `X` e `Y` di `IRect` a `X` e `Y` di `IPlaneLocation` utilizzando la keyword **`use`**:

```
class Rect
{
    public implement IRect;

    implement IPlaneLocation use BasePlaneLocation
    {
        override property X : use IRect.X;
        override property Y : use IRect.Y;
    }
}
```

Anche in questo caso, questa implementazione di `IPlaneLocation` potrebbe essere utilizzata non solo nella classe `Rect`, ma in tutte le classi che implementano `IRect`. Poiché l'implementazione di `IPlaneLocation` fa uso di elementi di `IRect`, una eventuale implementazione esterna deve indicare la dipendenza da `IRect` attraverso l'istruzione **`require`**.

```
implement IPlaneLocation use BasePlaneLocation as MiddleRectPlaneLocation
{
    require local IRect;
    ... (segue l'implementazione) ...
}

class Rect
{
    implement IRect;
    implement IPlaneLocation use MiddleRectPlaneLocation;
}
```

La clausola **`local`** indica che il requisito (servizio dipendente) deve essere risolto localmente, ovvero nello stesso contesto in cui avviene la richiesta. Nel caso specifico significa che una classe può usare questa implementazione di `IPlaneLocation` se e solo se implementa anche l'interfaccia `IRect`. Lo svantaggio di questa tecnica è che ogni classe che implementa `IRect` e vuole implementare anche `IPlaneLocation`, deve esplicitamente dichiarare

`IPlaneLocation`. In alternativa è possibile dichiarare che l'implementazione è rivolta ad un particolare gruppo di classi / interfacce attraverso la clausola **`for`**:

```
implement IPlaneLocation for IRect use BasePlaneLocation
{
    ... (segue l'implementazione) ...
}
```

In questo modo, tutti coloro che implementano `IRect` implementano automaticamente anche `IPlaneLocation`. Non è stato necessario specificare la dipendenza da `IRect`, perché questa è implicita nel **`for`**.

L'istruzione **`implement for`** può essere utilizzare anche per assegnare un'interfaccia solo ad un'istanza di una classe:

```
function Sample()
{
    var r : Rect;
    implement IPlaneLocation for r use MyImplementation;
    ...
}
```

Nell'esempio, solo `r` implementa `IPlaneLocation` attraverso `MyImplementation`.

L'implementazione di un'interfaccia può essere effettuata nel contesto di metodo, quando non ha valenza generale, ed è specifica per una particolare situazione. Ad esempio:

```
enum LogLevel
{
    Info,
    Warning,
    Error
}

interface ILogger
{
   function Log(message : String,  logLevel : LogLevel);
}

interface IProcess
{
   function Run();
   ...
}

class MyProcess
{
    implement IProcess
    {
       function Run()
       {
          require ILogger;
          ILogger.Log("Loading...", LogLevel.Info);
          ...
       }
    }
}


on IModule.load()
{
    require IConsole;
    var errors : String;

    implement ILogger
    {
        function Log(message : String,  logLevel : LogLevel)
        {
            if (logLevel == LogLevel.Error)
                errors.Append(message).Append(" ");
        }
    }

    var p = new MyProcess();
    p.Run();
    if (errors.Length > 0)
       IConsole.Write("Errors in process execution.");
}
```

La classe `MyProcess` implementa `IProcess`, e il metodo `Run` richiede `ILogger` per scrivere informazioni sull'avanzamento del processo. Nell'evento `load` dell'interfaccia `IModule`, è stata implementata `ILogger` accodando tutti i messaggi di tipo *Error* nella variabile `errors`. Poiché il metodo `Run` della classe `MyProcess` viene invocato all'interno dell'evento `load`, l'implementazione di `ILogger` associata a `Run` è propria quella di `load`. Al termine dell'esecuzione di `Run`, viene controllato se si sono verificati errori, e in caso affermativo viene stampato un messaggio a console.

Nell'istruzione **`implement use`** è possibile specificare anche una classe, a patto che questa implementi l'interfaccia richiesta. La creazione / distruzione della classe avviene all'ingresso / uscita dal contesto in cui è stata dichiarata l'istruzione **`implement`**. Ad esempio:

```
implement ILogger for IStream
{
    function Log(message : String, logLevel : LogLevel)
    {
       IStream.Write(logLevel)
       IStream.Write(": ")
       IStream.Write(message)
       IStream.Write("\r\n");
    }
}

class File
{
    public implement IStream
    {
        var _fileName : String;

        ...segue l'implementazione...
    }

    public constructor(fileName : String)
    {
         _fileName = fileName;
    }
}

on IModule.load()
{
    implement ILogger use class File("log.txt")

    var p = new MyProcess();
    p.Run();
}
```

Attraverso **`constructor`** è stato dichiarato un costruttore nella classe `File`, un particolare metodo chiamato dal compilatore per inizializzare la classe ogni volta che ne viene creata un'istanza. Nell'esempio il costruttore richiede come argomento il nome del file. Poiché la classe `File` implementa `IStream` e l'interfaccia `ILogger` è implementata per ogni `IStream`, anche la classe `File` implementa automaticamente `ILogger`. Nella clausola **`use`** gli argomenti del costruttore della classe `File` sono specificati come avviene nell'invocazione di un metodo.

Utilizzando **`new`** dopo **`use`**, viene chiesto di costruire una nuova istanza della classe ad ogni richiesta. La classe creata viene automaticamente distrutta all'uscita dal contesto in cui è avvenuta la richiesta.

```
implement ILogger use new class File("log.txt")
```

Analogamente, anche nell'istruzione **`require`** è possibile richiedere esplicitamente (dove è applicabile) una nuova istanza ad ogni richiesta:

```
require new ILogger
```

Nella clausola **`use`** è possibile specificare anche un'istanza di una classe memorizzata in una variabile.

```
var f = new File("log.txt");
implement ILogger use var f;
```

L'istruzione **`implement for`** può essere utilizzata anche per estendere classi dichiarate in moduli esterni. L'istruzione è contestuale, e non ha effetto sulle classi create prima di entrare nel contesto in cui compare l'istruzione:

```
class Circle
{
   implement ICircle;
}
```

```
implement IShape for Circle
{
   ...segue l'implementazione...
}
implement ICircle for Circle
{
   ...segue l'implementazione...
}
```

In alternative, se la modifica è globale, o riguarda più elementi è possibile utilizzare l'istruzione **`extend`**:

```
extend class Circle
{
   implement IShape
   {
       ...segue l'implementazione...
   }

   override implement ICircle
   {
       ...segue l'implementazione...
   }
}
```

Poiché l'interfaccia `ICircle` è già stata implementata in `Circle`, è necessario specificare **`override`** prima di una nuova implementazione.

Non è possibile estendere una classe che è stata definita nello stesso modulo in cui viene dichiarata l'estensione, e non è possibile estendere più di una volta la stessa classe nello stesso modulo. Le estensioni sono applicata in cascata in base alla priorità del modulo (i moduli sono caricati con un ordine specifico). Nel caso in cui la stessa interfaccia è implementata più volte dalla stessa classe, verrà utilizzata l'implementazione del modulo a priorità più alta.

Attraverso l'istruzione **`struct`** viene dichiarata una struttura dati. A differenza delle classi, le strutture non possono implementare alcuna interfaccia, e contengono solo un insieme di proprietà. L'accesso alle proprietà è diretto, e non attraverso metodi **`get`** e **`set`**:

```
struct Point
{
    X : Int32;
    Y : Int32;
}
```

Variabili, proprietà, argomenti e tipi primitivi

Attraverso l'istruzione **`var`** viene dichiarata una variabile. Il linguaggio è strettamente tipizzato, di conseguenza ogni variabile deve avere un tipo. Il tipo della variabile può essere esplicitato in fase di dichiarazione, oppure dedotto in base alla sua prima assegnazione. Le variabili possono essere dichiarate in qualunque contesto, e hanno validità solo in quel contesto o nei *contesti figli*. Qui sotto alcuni esempi di dichiarazione:

```
var a1 : Int32;
var b1 = 45;
var c1 = "Hello";
var d1 : String = "Sample";
```

La variabile `a1` è esplicitamente di tipo `Int32`, `b1` è implicitamente di tipo `Int32` e vale 45, `c1` è implicitamente di tipo `String` e vale "Hello", `d1` è esplicitamente di tipo `String` e vale "*Sample*".

Per ridurre il rischio di errori, non è possibile dichiarare una variabile con lo stesso nome di una dichiarata in un contesto padre. Ad esempio:

```
var a : Int32;

function Sample1()
{
    var a : Int32;
    var b : String;
}

function Sample2()
{
    var b : Byte;
}
```

La funzione `Sample1` non può dichiarare una variabile di nome a, poiché è già stata dichiarata nel modulo. La funzione `Sample2` può dichiarare la variabile di nome b, anche se è dichiarata in `Sample1`, poiché sono in *contesti fratelli*.

Il linguaggio fornisce alcuni tipi primitivi, riassunti in tabella:

Tipo	Descrizione	Dominio
`SByte`	Intero 8 bit con segno	da –128 a 127
`Int16`	Intero 16 bit con segno	da –32.768 a 32.767
`Int32`	Intero 32 bit con segno	da –2.147.483.648 a 2.147.483.647
`Int64`	Intero 64 bit con segno	da –9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
`Byte`	Intero 8 bit senza segno	da 0 a 255
`UInt16`	Intero 16 bit senza segno	da 0 a 65.535
`UInt32`	Intero 32 bit senza segno	da 0 a 4.294.967.295
`UInt64`	Intero 64 bit senza segno	da 0 a 18.446.744.073.709.551.615
`Float32`	Reale virgola mobile 32 bit (singola precisione)	3.4E +/- 38 (7 cifre)
`Float64`	Reale virgola mobile 64 bit (doppia precisione)	1.7E +/- 308 (15 cifre)
`Char`	Carattere unicode (Intero16 bit)	(stesso `UInt16`)
`Boolean`	Vero o falso	`true` / `false`
`String`	Stringa di testo a lunghezza variabile	N/A
`DateTime`	Data e ora (Intero 64 bit)	(stesso `UInt64`)
`Object`	Oggetto generico (anche i tipi primitivi possono essere convertiti in Object)	
`Type`	Tipo, incapsula le informazioni su un tipo	

Analogamente alle classi, anche i tipo primitivi possono implementare interfacce: E' possibile creare un nuovo tipo a partire da un tipo primitivo attraverso l'istruzione **`type`**:

```
type Percentage : Float32
{
    implement IFormatter
    {
        function Format() : String
        {
            return value.Format() + "%";
        }
    }
}
```

Se un tipo B estende un tipo A, le assegnazioni tra membri di tipo A e B possono essere fatte in entrambi i sensi, ma perdendo eventuali informazioni sul tipo. Ad esempio:

```
require IConsole;

on IModule.load()
{
    var a1 : Float32 = 0.5;
    var b1 : Percentage = 0.1;

    IConsole.WriteLine(a1.Format());
    IConsole.WriteLine(b1.Format());
    a1 = b1;
    IConsole.WriteLine(a1.Format());
}

Output: 0.5 10% 0.1
```

L'assegnazione di `b1` in `a1` è stata possibile (entrambi sono `Float32`), ma dopo l'assegnazione `b1` viene convertito in un normale `Float32`. In caso di assegnazione opposta (`b1 = a1`), `a1` sarebbe stato convertito in `Percentage`.

Quando viene assegnata un'interfaccia ad un membro vengono memorizzate anche le informazioni sul tipo di appartenenza, anche se questo è un tipo primitivo. Ad esempio:

```
require IConsole;

function CompareObject(a, b: IComparable)
{
    var text : String;
    if (a > b)
        text = " is greater than ";
    else
        text = " is less than ";

    IConsole.Write((a as IFormatter).Format() + text + (b as IFormatter).Format());
}

on IModule.load()
{
    var v1 : Percentage = 0.5;
    var v2 : Float32 = 0.2;
    CompareObject(v1, v2);
}
```

```
Output: 50% is greater than 0.2
```

Come ogni tipo primitivo, `Float32` implementa `IComparable`, di conseguenza anche `Percentage` implementa `IComparable` poiché estende `Float32`. Grazie a questa interfaccia, è possibile stabile una relazione d'ordine tra due oggetti, e quindi utilizzare l'operatore maggiore (>). In `CompareObject` tramite l'operatore **`as`**, viene richiesta ad `a` e `b` l'interfaccia `IFormatter`. Poiché `IComparable` non è convertibile in `IFormatter` la richiesta passa al tipo associato a `IComparable`, il quale implementa `IFormatter` (in questo caso ad `a` è associato `Percentage`, e a `b` `Float32`).

Normalmente variabili, proprietà e argomenti mantengono una copia del valore assegnato, che viene automaticamente distrutto all'uscita del contesto di dichiarazione. Attraverso la keyword **`ref`** è possibile creare un membro che mantiene un riferimento al valore assegnato, anziché duplicarlo. All'uscita dal contesto, il riferimento viene azzerato, ma l'elemento referenziato non viene distrutto. Ad esempio:

```
require IConsole;

on IModule.load()
{
    var r1 : Rect;
    var r2 : ref Rect;
    var r3 : Rect;

    r1 = new Rect();
    r1.X = 100;
    r2 = r1;
    r3 = r1;
    IConsole.WriteLine(r1.X);
    r2.X = 50;
    IConsole.WriteLine(r1.X);
    r3.X = 0;
    IConsole.WriteLine(r1.X);
}
```

```
Output: 100 50 50
```

Nell'esempio `r1` e `r3` sono di tipo `Rect`, mentre `r2` è di tipo riferimento a `Rect`. Dopo l'assegnazione `r1 = r2`, queste variabili fanno riferimento allo stesso valore, modificare `r2` implica modificare anche `r1` e viceversa. Questo non vale per `r3`, il cui valore è stato duplicato da `r1`.

A meno di indicazione diverse, il riferimento avviene attraverso il puntatore alla memoria in cui è attualmente allocato l'oggetto. Questo significa che per avere un riferimento valido, l'oggetto referenziato deve essere stato prima creato. In molti cosi è utile mantenere un riferimento solo attraverso alcune proprietà, e creare l'oggetto solo in un secondo momento. In questo caso è possibile specificare nella clausola **`ref`** attraverso quali proprietà realizzare il riferimento. La classe referenziata deve implementare `IRefBinding`, e attraverso il metodo `Bind` restituire l'oggetto giusto in base al valore delle proprietà di riferimento.

```
interface ICity
{
    property Name : String;
    property Cap : Char[5];
    property District : Char[2];
}

interface IPerson
{
```

```
    property BirthCity : ref ICity(Name);
}

class City { public implement ICity; }

class Person { public implement IPerson; }

static implement IRefBinding for ICity
{
    require IDbService;

    function Bind(propRef : IProperty) : object
    {
        var sql = "SELECT * FROM City WHERE " + propRef.Name + " = " + propRef.Value;

        return IDbService.Query(typeof(City), sql);
    }
}
```

Sono state dichiarate due interfacce (`ICity`, `IPerson`)e le rispettive classi. `IPerson` dichiara la proprietà `BirthCity` come un riferimento a `ICity`, attraverso la proprietà `Name`. L'interfaccia `ICity` implementa `IRefBinding` in modo statico (l'interfaccia è collegata a `ICity`, ma non usa `ICity`). Nel metodo `Bind` viene creato un oggetto `City` facendo una query su un ipotetico database.

E' sempre possibile accedere alle proprietà che fanno parte del riferimento, anche se il riferimento non è stato ancora risolto. La risoluzione del riferimento (ovvero la creazione dell'oggetto referenziato), avviene al primo accesso ad una proprietà non inclusa nel riferimento, quando viene invocato un metodo, o quando viene passato l'oggetto referenziato ad un membro. Nel momento in cui viene assegnato il valore ad una delle proprietà che fanno parte del riferimento, se il riferimento era stato precedentemente risolto, questo viene azzerato e l'oggetto creato distrutto. Riprendendo l'esempio precedente:

```
on IModule.load()
{
    var p1 : ref IPerson;
    var c1 : ref ICity;

    p1 = new Person();
    p1.BirthCity.Name = "Milano";

    IConsole.WriteLine(p1.BirthCity.Cap);
    c1 = p1.BirthCity;

    p1.BirthCity.Name = "Roma";
}
```

Al momento della creazione di `p1`, la proprietà `BirthCity` non fa riferimento ad alcun oggetto `ICity`, tuttavia è possibile accedere alla proprietà `Name` perché questa fa parte del riferimento. Nel momento in cui viene stampata la proprietà `Cap` (che non fa parte del riferimento), o viene assegnato a c1 `BirthCity`, il compilatore invoca il metodo `Bind` per creare l'oggetto `City` conforme al riferimento. Se il riferimento è stato risolto, quando viene riassegnata la proprietà `Name` di `BirthCity`, l'oggetto `City` viene distrutto e il riferimento azzerato

Indipendentemente che l'accesso sia per riferimento o per valore, è possibile assegnare ad ogni membro il valore nullo attraverso la keyword **`null`**. Il valore nullo indica che il membro non ha valore, e che il valore non è stato ancora assegnato. Tutte i membri valgono `null` fin quando non gli viene assegnato un valore. Non è possibile accedere in lettura ad una variabile se questa vale `null`. E' possibile controllare se un membro è nullo attraverso l'istruzione **`is null`**

```
var b1 : Int32 = 45;
b1 = null;
if (b1 is null)
   IConsole.WriteLine("b1 has no value");

Output: b1 has no value
```

Array e collezioni

Gli array possono essere a lunghezza fissa o variabile. Per dichiarare un array a lunghezza fissa è sufficiente specificare tra parentesi quadre le dimensioni, subito dopo il nome del tipo. Se le dimensioni vengono omesse, viene dichiarato un array a lunghezza variabile. Gli array a lunghezza variabile mantengono gli elementi su un buffer le cui dimensioni crescono / diminuiscono in base al numero degli elementi contenti. Ad esempio:

```
var a1 : String[5];
var a2 : Int32[];
```

`a1` è un array fisso di 5 stringhe, mentre `a2` è un array a lunghezza variabile di interi.

Gli array a lunghezza fissa implementano implicitamente `ICollection`, mentre quelli a lunghezza variabile implementano anche `IList`. Queste interfacce hanno la seguente struttura:

```
interface IReadOnlyCollection
{
   property Count : Int32;
   function GetItem(index : Int32) : object;
}

interface ICollection : IReadOnlyCollection
{
   function SetItem(index : Int32, value : object);
}

interface IList : ICollection
{
   function Add(item : object) : Int32;
   function InsertAt(index : Int32, item : object);
   function Clear();
   function Remove(item : object);
   function RemoveAt(index : Int32);
   function IndexOf(item : object) : Int32;
   function Contains(item : object) : Boolean;
   property Capacity : Int32;
}
```

Per accedere agli elementi di un array è possibile richiamare il metodo `GetItem` / `SetItem`, oppure utilizzare l'operatore [], che è implicitamente implementato in ogni array. Gli indici hanno base 0, ovvero il primo elemento ha indice 0, e non 1. Ad esempio:

```
require IConsole;
on IModule.load()
{
    var values : Int32[10];
    var i : Int32;
    for (i = 0; i < 10; i++)
    {
       IConsole.WriteLine("Insert value number " + i.Format() + ": ");
       values[i] = Int32.Parse(IConsole.ReadLine());
    }
    IConsole.WriteLine("The last value is: " + values[9].Format());
}
```

E' stato dichiarato un array di 10 interi, successivamente riempito da console. Al termine dell'inserimento, viene stampato il valore dell'ultimo elemento. In alternativa è possibile utilizzare un array a lunghezza variabile:

```
require IConsole;
on IModule.load()
{
    var values : Int32[];
    IConsole.WriteLine("Insert values, press enter to stop:");
    while (true)
    {
       var text : String;
       text = IConsole.ReadLine();
       if (text == "")
           break;
       values.Add(Int32.Parse(text));
    }
    IConsole.WriteLine("You insert " + values.Count.ToString() + " values.");
}
```

In questo caso vengono richiesti un numero arbitrario di elementi, e mano a mano accodati all'array. Al termine del processo viene stampato il numero degli elementi attualmente contenuti.

E' possibile inizializzare un array in fase di dichiarazione, specificando gli elementi contenuti tra parentesi graffe, separati dalla virgola. Il numero degli elementi e il tipo possono essere omessi, in quanto deducibili dall'espressione di assegnamento. Ad esempio:

```
var a1 = {3, 4, 5);
var b1 : String[2] = { "Dog", "Cat" };
```

`a1` è un vettore statico di tre interi (implicitamente dedotto), mentre `b1` è un vettore statico di due stringhe, esplicitamente dichiarato

Controllo di flusso

Analogamente ai linguaggi c, c++, java, c# esistono due istruzioni di selezione / controllo flusso. Attraverso l'istruzione **`if`** viene eseguito il blocco di istruzioni sottostante se la condizione specificata tra parentesi tonde è vera. Facoltativamente è possibile specificare anche un blocco **`else`**, che viene eseguito solo se la condizione è falsa. Ogni blocco di istruzioni deve essere racchiuso tra parentesi graffe, a meno che non sia costituto da una sola istruzione. Ad esempio:

```
function Sample(a1 : Int32)
{
    if (a1 > 50)
    {
        if (a1 < 80)
            IConsole.WriteLine("a1 is between 50 and 80");
        else
        {
            IConsole.WriteLine("a1 is greater than 80");
            return;
        }
    }
}
```

Utilizzando l'istruzione **`break`**, è possibile uscire immediatamente dal blocco **`if`** / **`else`**, senza eseguire le istruzioni successive. L'istruzione **`if`** può comparire anche sottoforma di espressione nella forma `<condizione> ? <vero> : <falso>`. Ad esempio:

```
require IConsole;

on IModule.load()
{
    var a1 = 30;
    var b1 = a1 > 15 ? "greater" : "less";
    IConsole.WriteLine(b1);
}

Output: greater
```

Poiché `a1` è maggiore di quindici, a `b1` viene assegnato "greater".

Attraverso l'istruzione **`switch`**, è possibile eseguire blocchi di codice in base al valore che assume l'espressione specificata tra parentesi tonde. I valori sono controllati tramite una sequenza di istruzioni **`case`**. Facoltativamente, è possibile dichiarare un blocco con l'istruzione **`default`**, che verrà eseguito solo se il valore non rientra in nessuno dei casi specificati. A differenza di altri linguaggi, è possibile specificare qualunque espressione all'interno di case, e non solo valori costanti. Il valore attuale è memorizzato nello pseudo-argomento **`value`**, che può essere omesso in caso di espressioni di uguaglianza. Ad esempio:

```
function Sample(a1 : Int32)
{
    switch (a1)
    {
        case 10 //a1 uguale a 10
        {
        }
        case (value == 15) //a1 uguale a 15
        {
        }
        case (value in [16..30]) //a1 compreso tra 16 e 30
        {
        }
        case (value > 30 && value % 2 == 0) //a1 maggiore di trenta e pari
        {
        }
        case (value > 30) //a1 maggiore di 30
        {
        }
        default //Tutti gli altri casi
        {
        }
    }
}
```

I casi non devono essere esclusivi, il valore può entrare in più casistiche. Per uscire da un caso, senza controllare i casi successivi, è necessario utilizzare l'istruzione **`break`**.

Cicli e iterazioni

Attraverso l'istruzione **`foreach`** è possibile eseguire un blocco di istruzioni, per ogni elemento di un insieme. Ad ogni iterazione l'elemento corrente viene passato in una variabile definita dall'utente. Ad esempio

```
on IModule.load()
{
    var a1 = new String[];
    a1.Add("Dog");
    a1.Add("Cat");

    foreach (var item : String in a1)
    {
        if (position(item) > 0)
           IConsole.Write(",");
        IConsole.Write(item);
    }
}
Output: Dog, Cat
```

Con l'operatore **`position`**, viene restituito l'indice dell'elemento all'interno collezione.

La variabile che contiene l'elemento corrente può essere una variabile esistente o può essere dichiarata all'interno dell'istruzione. Il tipo può essere omesso quando è deducibile dall'insieme passato. E' possibile utilizzare come insieme qualsiasi classe che implementa `IEnumerator` o `ICollection` (esiste un'implementazione di `IEnumratore` per `ICollection`) o un qualsiasi insieme statico. Segue la definizione di `IEnumerator`:

```
interface IEnumerator
{
    function Reset();
    function Next() : object;
    property readonly Current : object;
}
```

Il metodo `Reset` azzera l'enumeratore, `Next` restituisce l'elemento successivo o `null` in caso non ci siano ulteriori elementi, mentre `Current` mantiene l'elemento corrente.

E' possibile definire un insieme statico racchiudendo gli elementi tra parentesi graffe, separati da virgola. In alternative, se gli elementi sono interi consecutivi, è possibile definire un intervallo [a..b] (dove a e b possono essere costanti o identificatori). Ad esempio:

```
on IModule.load()
{
    var tot = 0;

    foreach (var i in [0..5])
         tot += i;
    IConsole.WriteLine(tot);
}

Output: 15;
```

Utilizzando le parentesi quadre inverse, si esclude l'estremo specificato.

Con l'istruzione **`while`** viene eseguito ciclicamente un blocco di istruzioni, fin quando la condizone specificata è vera. Ad esempio:

```
on IModule.load()
{
    require IFileSystem;
    var stream = IFileSystem.Open("test.dat") as IStream;
    while (!stream.IsEof)
    {
        var line = (stream as ITextReader).ReadLine();
        IConsole.WriteLine(line);
    }
    stream.Close();
}
```

E' stato dichiarato un ciclo che ad ogni iterazione legge una riga da un file precedentemente aperto, per poi stamparla a video. Il ciclo continua fin quando non si è raggiunta la fine del file.

E' possibile evalutare la condizione anziché in testa al ciclo, in coda utilizzando la coppia **`do while`**. Questa variante è utile nel caso il blocco debba essere eseguito almeno una volta. Ad esempio:

```
on IModule.load()
{
    do
    {
        var c : Char;
        IConsole.WriteLine("Press 0 to exit.");
        c = IConsole.ReadChar();
        switch (c)
        {
            case '1'
            {
                IConsole.WriteLine("You press one.");
                break;
            }
            case '2'
            {
                IConsole.WriteLine("You press two.");
                break;
            }
        }
    }
    while (c != '0');
}
```

L'istruzione **`for`** è simile all'istruzioine **`while`**, ma in aggiunta è possibile specificare, oltre che la condizione, anche un'espressione di inzializzazione e un'espressione da eseguire alla fine di ogni iterazione (tipicamente utilizzata per incrementare / decrementare una variabile). L'espressione di inizializzazione deve essere specificata come primo argomento, la condizione come secondo, mentre l'espressione di fine iterazione come terzo. Ad esempio:

```
on IModule.load()
{
   for (var i = 0; i < 5; i++)
      IConsole.WriteLine(i);
}

Output: 0 1 2 3 4
```

Nel ciclo è stata dichiara una variabile `i` ed è stato specificato che l'iterazione deve continuare fin quando `i` è minore di 5 e che la variabile `i` deve essere incrementata ad ogni iterazione.

All'interno di ogni tipo di iterazione, è possibile utilizzare l'istrzione **`break`**, per uscire immediatamente dal blocco senza eseguire ulteriori cicli o istruzioni, oppure l'istruzione **`continue`** per saltare dirattamente all'iterazione successiva, senza eseguire le istruzioni sottostanti.

Eventi

Gli eventi possono essere dichiarati solo all'interno di un'interfaccia con l'istruzione **`event`**. Ogni evento è identificato da un nome, e in modo del tutto analogo alle funzioni, può essere accompagnato da uno o più argomenti. L'evento, essendo un segnale rivolto a più ascoltatori, non può avere valore di ritorno. L'evento può essere lanciato attraverso l'istruzione **`raise`**, solo all'interno del contesto in cui è stata implementata l'interfaccia. Ad esempio:

```
interface IJob
{
    property Name : String;
    function Start();
    function Stop();
    event Started();
    event Terminated(success : Boolean);
}

class PrintJob
{
    public constructor(printer : IPrinter, document : IDocument)
    {
        _printer = printer;
        _document = document;
    }

    public implement IJob
    {
        protected var _printer : IPrinter;
```

```
        protected var _document : IDocument;

        function Start()
        {
            raise Started();
            _printer.Print(_document);
            raise Terminated(true);
        }

        function Stop()
        {
             raise Terminated(false);
        }
    }
}
```

Attraverso l'istruzione **on** è possibile dichiarare un blocco di codice da eseguire al verificarsi dell'evento specificato. E' possibile restare in ascolto di eventi generati da qualunque interfaccia, sia quelle indicate nella clausola **require**, sia quelle memorizzate in variabili / argomenti / proprietà. L'istruzione **on** può essere usata in qualunque contesto. L'assegnazione dell'ascoltatore avviene non appena il membro riceve un oggetto valido. In caso di assegnazione multiple allo stesso membro, l'ascoltatore assegnato al vecchio oggetto viene rilasciato, per essere assegnato al nuovo. L'ascoltatore viene in ogni caso rilasciato all'uscita dal contesto in cui è dichiarata l'istruzione **on**.

```
var job : IJob;

function Sample()
{
    on job.Terminated(success : Boolean)
    {
       IConsole.WriteLine("Job terminated.");
    }

    job = new PrintJob();
    job.Name = "Job 1";
    job.Start();

    job = new PrintJob();
    job.Name = "Job 2";
    job.Start();
}
```

L'ascoltatore per l'evento `Terminated` di job entra in funzione non appena viene assegnato un valore a job, Alla seconda assegnazione, viene rilasciato l'ascoltatore per l'oggetto "Job 1", per essere assegnato all'oggetto "Job 2.". All'uscita dal corpo della funzione, viene rilasciato anche per l'oggetto "Job 2".

Nel caso in cui sia necessario eseguire la stessa procedura per più eventi, o non si conosca a priori quale sarà l'oggetto sul quale restare in ascolto, è possibile utilizzare l'istruzione **attach**. Con questa istruzione è possibile utilizzare una funzione / metodo come ascoltatore per un evento dell'oggetto specificato. Il primo argomento della funzione deve essere di tipo `object` e conterrà l'oggetto che ha generato l'evento. E' possibile utilizzare l'istruzione **attach** solo se il membro è stato inizializzato con un oggetto valido. Utilizzando **attach** il rilascio dell'ascoltatore non è mai automatico, e deve essere fatto esplicitamente attraverso l'istruzione **detach**. Ad esempio:

```
interface IJobScheduler
{
    function AddJob(job : IJob);
}

class SquentialJobScheduler
{
    public implement IJobScheduler
    {
        protected var _jobs : ref IJob[];
        protected var _currentJob : IJob;

        function AddJob(job : ref IJob)
        {
            _jobs.Add(job);
            attach OnJobTerminated to job.Terminated;
            Schedule();
        }

        protected function OnJobTerminated(sender : object, success : Boolean)
        {
```

```
            detach OnJobTerminated to (sender as IJob).Terminated;
            _jobs.Remove(sender as IJob);
            _currentJob = null;
            Schedule();
        }

        protected function Schedule()
        {
            if (_jobs.Count > 0 && _currentJob != null)
            {
               _currentJob = _jobs[0];
               _currentJob.Start();
            }
        }
    }
}
```

E' stato dichiarata un interfaccia che rappresenta uno scheduler per alcune attività. L'interfaccia è stata implementata dalla classe `SequentialJobScheduler`, in modo da eseguire le attività in modo sequenziale. Ad ogni attività aggiunta nel metodo `AddJob`, viene associato l'ascoltatore `OnJobTerminated` per l'evento `Terminated`. Ad attività conclusa, nel metodo `OnJobTerminated`, viene rilasciato l'ascoltatore precedentemente aggiunto.

Anche in questo caso il numero di argomenti della funzione deve essere identico a quelli specificati nell'evento, e ogni argomento dell'evento deve essere convertibile nel corrispettivo argomento della funzione.

Operatori

In tabella sono raggruppati gli operatori per categoria, molti dei quali derivanti dai linguaggi c, c++, java, c#.

Operatore	Descrizione
`< <= == >= > != in`	Operatori di confronto (minore, minore uguale, uguale, maggiore uguale, maggiore, diverso, contenuto)
`+ - * / %`	Operatori aritmetici (addizione, sottrazione, moltiplicazione, divisione, modulo)
`&& || !`	Operatori logici (and, or, not)
`& | ^ << >> ~`	Operatori binari bit a bit (and, or, xor, shift a sinistra, shift a destra, complemento a uno)
`= += -= *= /= &= |= ^= <<= >>=`	Operatori di assegnazione (assegna, assegna e somma, assegna e sottrai, ecc)
`++a a++ --a a-- a(b) a[b] a.b`	Operatori vari (Pre incremento, post incremento, pre decremento, post decremento, chiamata a funzione, indicizzazione array, accesso a membro)
`is isnot as new delete typeof sizeof empty`	Operatori controllo tipo

Con l'operatore **new** si costruisce una nuova istanza di un oggetto. L'operatore deve essere seguito dal nome del tipo, più eventuali argomenti del costruttore, se definito. E' possibile utilizzare come tipo anche un'interfaccia, a patto che nel contesto corrente esista una regola che indichi quale sia l'implementazione predefinita per quell'interfaccia. Ad esempio:

```
interface IButton
{
    ...segue definizione...
}

class SkinnedButton
{
    public implement IButton
    {
        ...segue implementazione...
    }
}

implement IButton use new SkinnedButton;

on IModule.load()
{
```

```
    var button = new IButton();
}
```

Nel contesto di modulo è stata associata a `IButton` una nuova istanza della classe `SkinnedButton`. Utilizzando l'operatore new verrà quindi costruita una classe `SkinnedButton`

Il comportamento dell'operatore **`new`** varia in base a come è dichiarato il membro che riceverà il nuovo oggetto. Nel caso il membro memorizzi per valore, non sarà allocato un nuovo blocco di memoria, ma verrà utilizzato quello già allocato dal membro. Nel caso il membro memorizzi per riferimento, sarà allocato un nuovo blocco di memoria nell'heap, che dovrà essere deallocato manualmente attraverso l'operatore **`delete`**.

Con gli operatori **`is`** e **`isnot`** è possibile controllare se un membro è di un certo tipo o implementa un'interfaccia.

In aggiunta alle interazioni standard previste tra operatori e tipi, è possibile ridefinire alcuni operatori in modo che possano essere applicati anche a tipi definiti dall'utente. Possono essere ridefiniti tutti gli operatori, ad eccezione di quelli contrassegnati in rosso nella tabella. Attraverso l'istruzione **`operator`** si dichiara un operatore. Ogni operatore accetta uno o più argomenti, e può restituisce un valore. La sintassi per argomenti e valore di ritorno è del tutto analoga a quella utilizzata per le funzioni / metodi. Gli operatori che operano con più di un tipo possono essere dichiarati solo nel contesto di modulo. Ad esempio:

```
function mcm(a : Int32, b : Int32) : Int32 //Massimo comun divisore
{
    ...implementazione...
}

function mcd(a : Int32, b : Int32) : Int32 //Minimo comune multiplo
{
    ...implementazione...
}

interface IFraction
{
    property Num : Int32;
    property Den : Int32;
    function Simplify();
}

class Fraction
{
    public constructor(num : Int32, den : Int32)
    {
        Num = num;
        Den = den;
    }

    public implement IFraction
    {
        function Simplify()
        {
            var divisor = mcd(Num, Den);
            Num /= divisor;
            Den /= divisor;
        }
    }

    public implement IFormatter
    {
        function Format() : String
        {
            if (Den == 1)
               return Num.Format();
            return "(" + Num + "\" + Den + ")";
        }
    }
}

operator + (a : Fraction, b : Fraction) : Fraction
{
    var retValue = new Fraction();
    retValue.Den = mcm(a.Den, b.Den);
    retValue.Num = (retValue.Den / a.Den) * a.Num + (retValue.Den / b.Den) * b.Num;
    retValue.Simplify();
    return retValue;
}
```

```
on IModule.load()
{
    var a1 = new Fraction(2, 4);
    var a2 = new Fraction(3, 2);
    var r1 = a1 + a2;
    IConsole.WriteLine(r1.Format());
}

Output: 2
```

E' stata definita un'interfaccia `IFraction` e la rispettiva classe `Fraction`, dopodiché è stato ridefinito l'operatore + tra due oggetti di tipo `Fraction`.

Un operatore tipicamente ridefinito è quello di indicizzazione array. Essendo un operatore associato ad un solo tipo può essere dichiaro anche all'interno di una classe / interfaccia. In caso venga definito all'esterno, il primo argomento rappresenta il tipo al quale l'operatore deve essere applicato. Poiché attraverso l'operatore [] è possibile sia leggere, che scrivere dei valori, il corpo dell'operatore è del tutto simile a quello di una proprietà. Ad esempio:

```
interface IMatrix
{
    property readonly Rows : Int32;
    property readonly Cols : Int32;
    function GetItem(rowIndex, colIndex : Int32) : object;
    function SetItem(item : object, rowIndex, colIndex : Int32);
}

class Matrix
{
    public constructor(rows, cols : Int32)
    {
        _buffer = new Int32[cols * rows];
        Cols = cols;
        Rows = rows;
    }

    public constructor(buffer : ref Int32[], cols : Int32)
    {
        _buffer = buffer;
        Cols = cols;
        Rows = buffer.Count / Cols;
    }

    public implement IMatrix
    {
        protected var _buffer : ref Int32[];

        function GetItem(rowIndex, colIndex : Int32) : object
        {
            return _buffer[(rowIndex * Rows) + colIndex];
        }

        function SetItem(item : object, rowIndex, colIndex : Int32)
        {
            _buffer[(rowIndex * Rows) + colIndex] = item;
        }
    }
}

operator [] (matrix : ref IMatrix, rowIndex, colIndex : Int32) : object
{
    get
    {
        return matrix.GetItem(rowIndex, colIndex);
    }
    set
    {
        matrix.SetItem(value, rowIndex, colIndex);
    }
}

on IModule.load()
{
    var m1 : new Matrix(100, 20);
    m1[0, 0] = 5;
}
```

E' stata definita una generica matrice di oggetti (interfaccia `IMatrix`) e la rispettiva implementazione in `Matrix`. Dopodiché è stato definito esternamente l'operatore [] per lavorare sulla generica interfaccia `IMatrix`, che accetta come argomenti gli indici di riga / colonna.

Eccezioni

La gestione delle eccezione è del tutto simile a quello già adottato da altri linguaggi (java, c#, c++, etc). Un eccezione è una condizione anomala che impedisce il normale flusso del programma. Le eccezioni possono essere generate dal microprocessore (ad esempio, durante un accesso non valido in memoria) oppure dall'utente tramite l'istruzione **`throw`**. Un'eccezione è rappresentata da una qualunque classe che implementa `IException`. Ad esempio:

```
class ArgumnentException
{
    implement IException;
}

function Sample(val : Int32)
{
    if (val < 0)
        throw new ArgumentException();
    ...
}
```

E' stata definita una classe che rappresenta l'eccezione di argomento non valido. Successivamente in `Sample` viene lanciata `ArgumentException` se il valore passato è negativo.

Le eccezioni possono essere catturate e gestite all'interno di un blocco **`try cacth`**. L'istruzione **`catch`** accetta opzionalmente come argomento il tipo di eccezione da gestire, e può comparire anche più volte dopo un blocco **`try`**, in modo da poter gestire in modo diverso eccezioni diverse. Quando viene generata un eccezione, l'esecuzione viene interrotta e il controllo passato al primo blocco **`catch`** compatibile. Gestita l'eccezione, l'esecuzione riprende dalla prima istruzione esterna al blocco **`try catch`** appena eseguito. E' possibile dichiarare un numero arbitrario di blocchi **`try cacth`** all'interno di un metodo / funzione, anche annidati tra di loro. Ad esempio

```
function DeleteAuthorSong(path, auhtor : String)
{
    require IConsole;
    require IFileSystem;
    try
    {
        foreach (string fileName in IFileSystem.List(path + "*.mp3"))
        {
            try
            {
                ISong song = new Mp3Media(fileName);
                if (audio.Author == auhtor)
                    IFileSystem.Delete(fileName);
            }
            catch
            {
                 IConsole.WriteLine("Cannot delete " + fileName);
            }
        }
    }
    catch (SecurityException)
    {
        IConsole.WriteLine("Access to path " + path + " is denied");
    }
    catch (IOException as ex)
    {
        IConsole.WriteLine("IO error: " + ex.Message);
    }
}
```

E' stata dichiarata una funzione (`DeleteAuthorSong`) che elimina tutti i file mp3 di un dato autore e path. Il blocco **`try catch`** più interno cattura qualunque tipo di eccezione ad ogni interazione, cosi in caso di errore è possibile proseguire con il file successivo. Il blocco **`try catch`** più esterno, controlla due tipi di eccezioni, la prima un errore di sicurezza (`SecutrityException`), la seconda un generico errore di IO (`IOException`), di cui è possibile accedere ai dettagli attraverso `ex`.

In coda ad un blocco **`try catch`** è possibile dichiarare anche un blocco **`finally`**. Il blocco **`finally`** viene sempre eseguito all'uscita dal blocco **`try`**, anche se l'uscita è causata da un'eccezioni o un'istruzione **`break`** / **`return`**;

```
function Sample()
{
    require IFileSystem;
    var fileReader = IFileSystem.Open("test.txt") as ITextReader;
    try
    {
       ...uso di reader...
    }
    finally
    {
        fileReader.Close();
    }
}
```

Nell'esempio, la chiusura del file deve esser fatta anche in caso di eccezione.

Conversioni di tipo

Le conversioni tra un tipo e un altro sono implicite quando non vi è perdita di informazioni. E' quindi possibile assegnare ad un tipo numerico di maggior precisione, un tipo numerico di minor precisione (ma non viceversa). E' inoltre possibile assegnare ad un membro di tipo interfaccia, un qualsiasi oggetto che implementa tale interfaccia (ma non viceversa). Poiché alcuni oggetti possono subire estensioni, o cambiare comportamento a seconda del contesto, non è sempre possibile stabile staticamente in fase di compilazione se un oggetto implementa un'interfaccia. In questo caso è possibile utilizzare l'operatore **`as`** per tentare una conversione dinamica a runtime. Le conversioni dinamiche possono essere molto costose, e vanno evitate nel limite del possibile.

Attraverso l'istruzione **`conversion`** è possibile definire una conversione personalizzata tra due tipi. Le conversioni possono essere implicite, ovvero eseguite automaticamente, o esplicite, ovvero eseguite con l'ausilio dell'operatore **`as`**. E' consigliato utilizzate conversioni implicite solo quando il costo di conversione è minimo L'istruzione **`conversion`** è valida solo nel contesto di modulo. Il tipo convertito deve essere restituito tramite l'istruzione **`return`**, mentre l'oggetto da convertire è accessibile attraverso lo pseudo-argomento **`value`**. Riprendendo l'esempio della classe frazione:

```
conversion implicit IFraction to Float32
{
    return (value.Num as Float32) / (value.Den as Float32);
}

on IModule.Load()
{
    var a1 = new Fraction(1, 2);
    var f1 : Float = a1;
}
```

E' stata dichiara una conversione tra `IFraction` e `Float`, semplicemente dividendo numeratore con denominatore

Un altro semplice esempio di conversione potrebbe essere tra oggetti che implementano `IFormatter` e `String`:

```
conversion implicit IFormatter to String
{
    return value.Format();
}
```

Metodi condizionali e multi-tipo

Alcune categorie di funzioni hanno comportamenti diversi in base al contesto, o al valore degli argomenti passati. Alcune di queste varianti possono essere gestite in modo statico all'interno della funzione tramite blocchi **`if`** / **`switch`**, altre non sono gestibili staticamente poiché il dominio dei valori possibili di un argomento può variare con il tempo. Attraverso la clausola **`condictional`** prima dell'istruzione **`function`**, è possibile dichiarare un metodo condizionale. Il metodo verrà eseguito se e solo se la condizione specificata nella clausola **`where`** è vera. La condizione può essere una qualunque espressione, e può comprendere controlli anche sugli argomenti passati. Le interfacce che contengono un metodo condizionale, non possono dichiarare altri membri. E' possibile implementare più volte un metodo condizionale, anche all'interno della stessa classe, ma specificando condizioni diverse. Ad esempio:

```
interface ICustomFormatter
{
    condictional function Format(format : String);
}

implement ICustomFormatter for Int32
{
    condictional function Format(format : String) where (format == "x")
```

```
    {
        return Int32ToHex(value);
    }

    condictional function Format(format : String) where (format.StartWith("b"))
    {
        var digitCount = Int32.Parse(format.Substring(1));
        return Int32ToBinary(value, digitCount);
    }
}

on IModule.load()
{
    var a1 : Int32;
    a1 = 12;
    IConsole.WriteLine(a1.Format("x"));
    IConsole.WriteLine(a1.Format("b8"));
}

Output: C 00001100
```

E' stata dichiarata l'interfaccia `ICustomFormatter`, con un metodo condizionale `Format` utilizzato per trasformare un oggetto in stringa, fornendo come argomento il formato desiderato. L'interfaccia è stata implementata per `Int32`, e il metodo `Format` ha due varianti: una è valida nel caso in cui l'argomento `format` valga `x`, l'altra nel caso `format` inizi per `b`.

E' possibile rendere accessibile una funzione sottoforma di metodo, anche da più tipi contemporaneamente. I tipi sui quali dovrà essere mappata devono essere dichiarati come argomenti, preceduti dalla clausola **`member`**. Questo tipo di funzione è utile quando l'operazione che si deve descrivere coinvolge più tipo, e non uno in particolare. Ad esempio:

```
interface IPrinter
{
    ...segue la definizione...
}

interface IDocument
{
    ...segue la definizione...
}

function Print(printer : member IPrinter, doc : member IDocument, settings : IPrintSettings)
{
    ...segue l'implementazione...
}

on IModule.load()
{
    var p1 : IPrinter;
    var d1 : IDocument;

    ...assegnazione di p1 e d1...

    p1.Print(d1, null);
    d1.Print(p1, null);
}
```

Il metodo `Print` è accessibile sia dall'interfaccia `IPrinter` che dall'interfaccia `IDocument`. Quando viene invocato da `p1`, non deve essere passato l'argomento `printer`, quando viene invocato da `d1`, non deve essere passato l'argomento `document`.

Contesti definiti dall'utente

Attraverso l'istruzione **`context`** è possibile definire un contesto personalizzato, nei quale poter indicare come dovranno essere implementante alcune interfacce. Le uniche istruzioni valide all'interno di un blocco **`context`** sono **`require`**, **`implement`**, e **`var`**. I contesti possono essere dichiarati solo a livello di modulo, e possono essere statici o dinamici. I contesti statici devono avere obbligatoriamente un nome. I contesti dinamici sono accompagnati da una clausola **`where`**, usata dal sistema per stabile quando è il momento di entrare in quel contesto. L'ingresso è automatico, non appena l'espressione specificata in **`where`** risulta vera. Ad esempio:

```
context when (IUserInterface is GraphicUserInterface)
{
    implement IButton use new class GraphicButton();
    implmenet ITextBox use new class GraphicTextBox();
}
```

```
context when (IUserInterface is HtmlUserInterface)
{
    implement IButton use new class HtmlButton();
    implement ITextBox use new class HtmlTextBox();
}

function Sample()
{
    var textBox1 = new ITextBox();
    var textBox2 = new ITextBox();
}
```

Sono stati dichiarati due contesti dinamici, entrambi vincolati a come è stato implementato il servizio di interfaccia grafico (`IUserInterface`). Il primo, dichiara che se il servizio di interfaccia utente è implementato dalla classe `GraphicUserInterface`, le interfaccia `ITextBox` e `IButton` devono essere implementate da una nuova istanza delle classi `GraphicTextBox` , `GraphicButton`. Analogamente il secondo contesto. La funzione `Sample`, non fa richiesta di una classe specifica, e l'assegnazione avverrà in base al contesto in cui viene invocata la funzione.

Per forzare l'ingresso in un particolare contesto è possibile utilizzare l'istruzione **`enter context`**, seguita dal nome del contesto. Il nome può essere un qualsiasi modulo, classe, metodo, o contesto statico definito dall'utente, oppure essere omesso nel caso si voglia entrare in un nuovo contesto. Il cambio di contesto può essere utilizzato quando è necessario controllare l'accesso a determinati servizi. Ad esempio:

```
implement ILog use class File("log.txt");

context Startup
{
    require IConsole;
    implement ILog use IConsole;
}

on IModule.load()
{
    enter context Startup
    {
        require ILog;
        ILog.Log("Inzializzazione...");
        ...
    }
    ...
}
```

E' stato dichiarato a livello di modulo che il servizio di log (`ILog`) deve essere implementato dalla classe `File`. E' stato dichiarato un contesto di nome `Startup`, in cui è stato richiesto il servizio di console e dichiarato che il servizio di log deve essere implementata da `IConsole`. Nella fase di inizializzazione del programma, viene effettuato l'ingresso nel contesto `Startup`, cosicché ogni eventuale messaggio di log possa essere visualizzato su console, anziché su file.

Il cambio di contesto può essere anche utilizzato quando è necessario agire come parte di una classe / metodo. Ad esempio:

```
on IModule.load()
{
    require IConsole;
    enter context IConsole
    {
         WriteLine("First Message");
         WriteLine("Second Message");
    }
}
```

Poiché si è entrati nel contesto dell'interfaccia `IConsole`, è possibile utilizzare i metodi dell'interfaccia senza alcun prefisso, esattamente come se ci trovassimo al suo interno.

Attraverso l'istruzione **`lock context`** si congela il contesto corrente. Fin quando si rimarrà dentro al blocco **`lock`**, non sarà possibile eseguire alcun cambio di contesto, e verrà mantenuto il contesto presente prima di eseguire l'istruzione. Ad esempio:

```
function Sample()
{
    lock context
    {
        ...
    }
}
```

Codice nativo

Il compilatore non emette codice assembly, ma ANSI C. Le ragioni di questa scelta sono la grande vicinanza di questo linguaggio al codice macchina, e l'enorme diffusioni di compilatori C che oramai esistono per qualunque tipo di piattaforma o microprocessore. Ogni compilatore C possiede delle estensioni che permettono al programma di poter invocare particolari funzioni di sistema, o di accedere a risorse hardware. Attraverso il blocco **`#native`** / **`#endnative`**, è possibile scrivere un blocco di codice C che passerà inalterato nel processo di compilazione. L'accesso a variabili / proprietà all'interno di un blocco nativo è possibile solo utilizzando l'istruzione **`__extern`**.

```
class PosixConsole
{
    implement IConsole
    {
        function WriteLine(text : String)
        {
            #native
                printf("%s\n", __extern(text));
            #endnative
        }
    }
}
```

Il comportamento dell'istruzione **`__extern`** varia a seconda del tipo passato ed è riassunto in tabella:

Operatore	Descrizione
Tipi numerici	Valore numerico inalterato
Stringhe	Il puntatore al primo carattere in formato unicode.
Array	Il puntatore al primo elemento
Date	Intero a 64 bit (numero di 100 nanosecondi trascorsi dalle 12 del primo gennaio del 1601)
Booleani	Intero a 8 bit 1 in caso di vero, 0 in caso di falso;
Classi / Interfacce / Strutture /	Puntatore alla memoria in cui è allocato

Namespace, moduli e dipendenze

E' possibile raggruppare tipi tra loro correlati all'interno di namespace. I namespace sono identificati da nomi, e hanno una struttura gerarchica (ci può essere un namespace all'interno di un'altro namespace). Tipi appartenenti a namespace diversi possono avere lo stesso nome. Tutti i tipo dichiarati all'interno di un blocco **`namespace`**, faranno parte di quel namespace. Ad esempio

```
namespace System
{
    interface ISample
    {
        ...
    }
}
```

l'interfaccia `ISample` è nel namespace `System`

L'istruzione **`namespace`** può essere utilizzata solo a livello di modulo, anche più volte. A differenza dei package di java, non c'è alcuna correlazione tra namespace e collocazione fisica dei file sorgenti. Classi che fanno parte di file o moduli diversi, possono comunque appartenere allo stesso namespace. Ogni namespace è sempre relativo al namespace padre. Ad esempio

```
namespace System
{
    namespace IO
    {
         interface IFile;
    }
}

namespace System.IO
{
    class File;
}
```

Sia la classe `File`, che l'interfaccia `IFile` si trovano dentro al namespace `System.IO`

Per accedere ad un tipo che si trova all'interno di un namespace, è necessario specificare prima del nome del tipo il namespace di appartenenza, ad esempio:

```
var a1 : System.ISample;
```

Il namespace può essere omesso se il tipo viene utilizzato nello stesso namespace di definizione.

Attraverso l'istruzione **`using`** seguita dal nome del namespace, è possibile utilizzare tutti i tipi del namespace senza obbligo di prefisso. Ad esempio:

```
using System;

on IModule.load()
{
    var a1 : ISample; //ISample è stata dichiarata nel namespace System
}
```

Per utilizzare una classe o un interfaccia che è stata definita in un modulo esterno è necessario utilizzare l'istruzione **`import`** seguita dal nome del modulo. L'istruzione **`import`** può essere utilizzata anche una sola volta all'interno di un qualsiasi file che compone il modulo. Ad esempio:

```
import MyModule;
```

Il comportamento di questa istruzione (dove o come cercare i moduli richiesti), è strettamente vincolata al sistema operativo.

Un problema concreto

Viene proposta la soluzione dello stesso problema del capitolo 1. Qui in seguito la definzione delle interfaccie `IStream`, `IDevice`:

```
type DeviceCategory extend String;

interface IStream
{
    function ReadByte() : Byte;
    function Read(buffer : ref Byte[], count : Int32) : Int32;
    function ReadLine() : String;
    property readonly IsEof : Boolean;
}

interface IDevice
{
    function Enable();
    function Disable();
    property DeviceName : String;
    property DeviceCategory : DeviceCategory;
}
```

La loro implementazione condivisa:

```
implement IDevice as BaseDevice
{
    function Enable()
    {
        ...implememtazione...
    }

    function Disable()
    {
        ...implememtazione...
    }
}

implement IStream as BaseStream
{
    function Read(buffer : ref Byte[], count : Int32) : Int32
    {
        var i = 0;
        while (!IsEof && i < count)
            buffer[i++] = ReadByte();
        return i
    }
```

```
    function ReadLine() : String
    {
        var c : Char;
        var line = new String();
        while (!IsEof && (c = ReadByte()) != '\n')
            line.Append(c);
        return line;
    }
}
```

La classe `File` implementa `IStream` utilizzando `BaseStream` e definendo il metodo `ReadByte` utilizzando il servizio `IFileSystem`

```
public class File
{
    require IFileSystem as fs;

    public constructor(fileName : String)
    {
        _fileDescriptor = fs.Open(fileName);
    }

    public implement IStream use BaseStream
    {
        protected var _fileDescriptor : Int32;
        protected var _buffer : Byte[1];

        function ReadByte() : Byte
        {
            fs.ReadFile(_fileDescriptor, _buffer, 1);
            return _buffer[0];
        }
    }
}
```

La classe `MemoryStream` implementa da zero `IStream`

```
public class MemoryStream
{
    public constructor(buffer : ref Byte[])
    {
        _position = 0;
        _buffer = buffer;
    }

    public implement IStream
    {
        protected var _position : Int32;
        protected var _buffer : ref Byte[];

        function ReadByte() : Byte
        {
            return _buffer[position++];
        }

        function Read(buffer : ref Byte[], count : Int32) : Int32
        {
            count = min(count, _buffer.Count - _position);
            _buffer.CopyTo(buffer, _position, count);
            return count;

        }

        function ReadLine() : String
        {
            var read = 0;;
            while (read + position < _buffer.Count && _buffer[read + _position] != '\n')
                read++;
            var retValue = new String(buffer, _position, count, Encoding.ASCII);
            _position += read;
            return retValue;
        }

        property readonly IsEof : Boolean
        {
            get { return _position >= _buffer.Count; }
        }
    }
}
```

La classe `SerialPort` implementa `IDevice` utilizzando `BaseDevice`, `IStream` utilizzando `BaseStream` e definendo il metodo `ReadByte`. Inoltre fa richiesta dei servizi di io da porta.

```
public class SerialPort
{
    require IPortService;

    public constructor(portId : String)
    {
        DeviceName = portId;
        IPortService.Open(portId);
    }

    public implement IDevice use BaseDevice;

    public implement IStream use BaseStream
    {
        function ReadByte() : Byte
        {
            return IPortService.Read(PortName);
        }

        override property readonly IsEof : Boolean
        {
            get { return false; }
        }
    }
}
```