ASTROBRICK

Guide de développement

Michel Pujol
23/03/2015

Table des matières

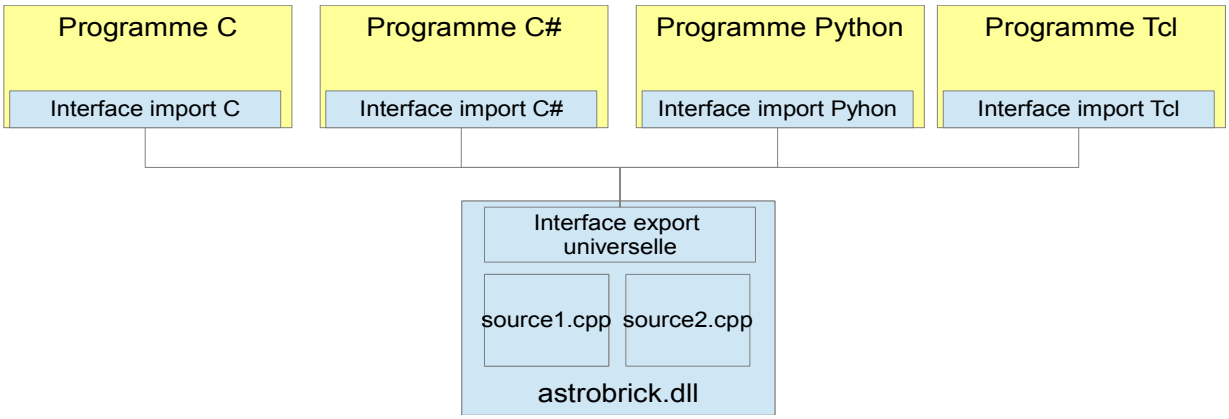
1. Architecture d'une brique

Ce guide s'adresse aux développeurs de AudeLa qui souhaitent publier une librairie d'AudeLa sous forme d'*astrobrick*.

Une *astrobrick* est une librairie dynamique offrant une interface pour différents langages. Cette interface se décompose en une **interface d'export** faisant partie de la librairie et une **interface d'import** à insérer dans le programme utilisateur.

1.1 Solution proposée

Cette solution consiste a créer une seule interface d'export universelle et une interface d'import pour chaque langage.



Avantage:

> Une seule libraire par système d'exploitation suffit pour tous les langages.
>
> Ce nombre réduit d'interfaces d'export permet de coder manuellement ces interfaces sans obligatoirement faire appel à un générateur d'interface externe comme SWIG.

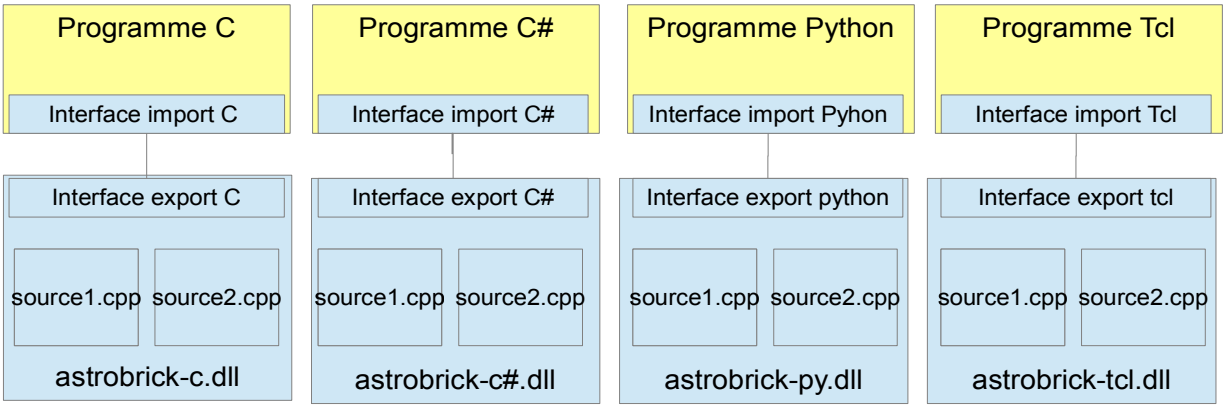
Inconvénient:

> La communication entre le programme utilisateur et la brique utilise des principes génériques communs à tous les langages, sans exploiter les possibilités spécifiques à chaque langage.
>
> Nécessite de restreindre les techniques de programmation des interfaces en appliquant des règles de programmation compatibles avec tous les langages décrites ci-dessous.

➔ **C'est la solution retenue pour les Astrobrick dans ce document.**

1.2 Autre solution (non retenue)

Une autre solution consiste a créer des paires d'interfaces export/import pour chaque langage.



Avantage:

permet de répondre à un très grand nombre de besoins divers.

Chaque paire d'interfaces export/import dédiée à chaque langage exploite au mieux les possibilités offerte par chaque langage pour communiquer entre le programme utilisateur et la brique.

Inconvénient:

nécessite de compiler une librairie pour chaque langage et pour chaque système d'exploitation.

Par exemple, il faut fabriquer 12 librairies pour une Astrobrick utilisable par des programmes en 4 langages (C, C#, Python, TCL) et sous 3 système d'exploitation (Windows, Linux, Macosx) .

Cette solution est mise en œuvre par l'outil **SWIG** qui génère un wrapper d'export et un wrapper d'import pour chaque langage.

2. Atelier de fabrication d'une brique

Une brique est fabriquée dans un sous projet de AudeLA.

Ce projet est créé dans le répertoire **audela/src/audela** si la brique fait partie du cœur d'AudeLa , ou sinon dans le répertoire **audela/src/contrib**

2.1 Projet source de la brique

Le projet de la brique est constitué

- du sous projet de fabrication de la brique avec sont interface universelle pour les 3 systèmes d'exploitation,
- des interfaces d'import pour les langages
- d'un programme de test systématique de la brique, indispensable pour publier une brique qui fonctionne.
- des makefile de déploiement

Exemple de la brique **absimple**

```
audela/contrib/absimple  // répertoire global de la brique (*)

     /absimple                    // projet de fabrication de la brique
       /src
         absimple.h               // sources de l'interface d'export
         absimple.cpp             //  universelle
         absimple_ld.h            // interface d'import C chgt dynamiqu

         CLibsimple1.cpp                |
         CLibsimple2.cpp                |
         CCalulator.h                   |  sources internes de la brique
         CCalulator.cpp                 |
         CCalendar.h                    |
         Calendar.cpp                   |
       /linux
         makefile
       /macosx
         makefile
       /vc90
         absimple.vcproj

     /absimple_tcl                // projet interface d'import tcl
       /src
         absimple_tcl.h
         absimple_tcl.cpp
         pkgindex.tcl
         absimple.tcl
       /linux
         makefile
       /macosx
         makefile
       /vc90
         absimple_tcl.vcproj

     /absimple_test          // projet programme de test de la brique
       /src
         absimple_test.cpp
       /linux
         makefile
       /macosx
         makefile
```

```
      /vc90
        absimple_test.vcproj

    /linux                          // makefile déploiement sous linux
      makefile
    /macosx                         // makefile déploiement sous macosx
      makefile
    /vc90
      makefile                      // makefile déploiement sous windows
```

(*) si la brique fait partie du cœur d'AudeLA, son projet est créé dans /audela/src/**audela**/absimple au lieu de /audela/src/**contrib**/absimple

2.2 Déploiement de la brique

2.2.1 Répertoire de déploiement

La brique est déployée dans le répertoires d'AudeLA **astrobrick_user.**

```
/audela
   /astrobrick_user
      /csharp
        absimple.cs             // interface d'import C# de la brique
        absimple_test.cs        // exemple de programme d'utilisation de la brique
        libabsimple.dll         // librairie de la brique
        libabsimple.so          // librairie de la brique

      /doc
        /html                   // documentation des briques
        astrobrick.html        // point d'entrée de la documentation

      /python
        absimple.py             // interface d'import Python de la brique
        absimple_test.py        // exemple de programme d'utilisation de la brique
        libabsimple.dll         // librairie de la brique
        libabsimple.so          // librairie de la brique

   /bin
        libabsimple_tcl.dll     // interface d'import TCL de la brique
        libabsimple.dll         // binaire de la brique

   /lib                         // librairies TCL
      /absimple1.0
        pkgindex.tcl            // interface d'import TCL de la brique
        libabsimple_tcl.dll     // interface d'import TCL de la brique (*)
        libabsimple.dll         // binaire de la brique (*)

   /src
      /include
        absimple.h              // interface d'import C de la brique
                                // de la brique
      /lib
        absimple.lib            // interface d'import C++ par chargement statique
                                // de la brique dans un programme utilisateur C++
```

2.2.2 Déploiement sous Windows

La solution **astrobrick.sln** permet de déployer les Astrobricks dans le répertoire de déploiement

sous Windows.

1. Ouvrir la solution avec Visual Studio C++
2. Générer la solution en mode **Release**
3. Vérifier dans la fenêtre de sortie de Visual C++ que les libraires sont ajoutées dans le répertoire de chaque langage

2.2.3 Déploiement sous Linux

Le fichier **audela/src/astrobrick/linux/makefile** permet de déployer les Astrobricks dans le répertoire de déploiement sous Linux.

```
$ cd audela/src/astrobrick/linux

$ make
```

2.2.4 Déploiement sous Macosx

Le fichier **audela/src/astrobrick/macosx/makefile** permet de déployer les Astrobricks dans le répertoire de déploiement sous Macosx.

```
$ cd audela/src/astrobrick/linux

$ make
```

2.2.5 Génération de la documentation

Prérequis : installer doxygen sur le poste de travail.

La documentation est générée automatiquement par la commande de déploiement des astrobrick (cf. paragraphes précédents) dans le répertoire **audela/astrobrick_user/doc/html**

La documentation est générée avec l'outil **doxygen** en utilisant le fichier de configuration

audela/src/astrobrick/astrobrick.doxygen

et à partir des fichiers suivants:

- les fichiers header **audela/src/include/*.h**
- la présentation générale **audela/doc/astrobrick/astrobrick-user-guide.html**

Génération «manuelle» de la documentation sous Windows

1. Lancer Doxywizard
2. Ouvrir le fichier de configuration **audela/src/astrobrick/astrobrick.doxygen**
3. Cliquer sur le bouton **Run doxygen** dans l'onglet **Run**

Génération « manuelle » de la documentation sous linux ou Macosx

```
$ cd audela/src/astrobrick

$ doxygen -g astrobrick.doxygen
```

2.3 Version 32 bits et 64 bits

À étudier

3. Interface d'export universelle d'une brique

3.1 Règles générales de l'interface d'export

L'interface d'export est composée de deux fichiers portant le nom de la brique

- un fichier header (.h) qui contient les déclarations des fonctions exportables d'une brique.
- un fichier source (.cpp) qui contient le corps des fonctions d'export ou leur alias vers des fonctions internes de la brique

Exemple: l'interface d'export de la brique **absimple** est composée des fichiers

- ✔ absimple.h
- ✔ absimple.cpp

Remarques:

Le fichier header **absimple.h** contient les déclarations des ressources exportables de la brique.

Le ressources sont déclarée dans un namespace portant le nom de la brique
Exemple :

```
namespace absimple
```

Important: L'interface d'export doit masquer l'implémentation interne des fonctions. En particulier le fichier header d'export ne doit pas inclure des fichiers header interne de la brique.

Le fichier source **absimple.cpp** contient des fonctions destinées à l'export

- l'implémentation des fonctions exportées ou un alias vers une implémentation interne de ces fonctions.
- les constructeurs et destructeurs des classes exportées par la brique
- l'implémentation des fonctions exportables pour gérer les exceptions registerErrorCallback et throwExternalError .
- l'implémentation de la fonction exportable releaseCharArray pour détruire les chaînes de caractères retournées par certaines fonctions.

3.1.1 Ressources exportables

Export autorisé

fonctions C,

structures et classe d'interface,

pointeurs de fonction C.

Export interdit

variables globales,

structures et classes internes de la brique

Remarque: la définition des structures/classes interne ne doit pas être exportée. Seules les redéfinitions de ces structures/classes ne contenant que des champs ou des méthodes publiques peuvent être exportées afin de faciliter leur utilisation par un programme écrit en C ou C++ en

particulier le programme AudeLa.

3.1.2 Gestion de la mémoire

Le programme utilisateur ne doit pas pouvoir passer en paramètre des objets complexes créés dans le programme principal avec leur constructeur par défaut.

➔ Toute création et destruction d'objet complexe (tableau, structure, classe) échangé avec la brique doit être faite via une une fonction spécifique de la brique par exemple

Exemple:

```
// destruction d'une chaine de caractères créée par la brique
void absimple_releaseCharArray( const char* charArray)
```

➔ L'interface d'export de la brique ne doit pas publier les constructeurs des objets complexes afin de les rendre inaccessibles aux programmes utilisateurs lorsque ce n'est pas nécessaire

3.2 Export d'une fonction

La déclaration d'un fonction (appelée aussi prototype d'une fonction) est une ligne composée:

1. de la décoration du nom de la fonction
2. des directives d'export
3. du type de retour de la fonction
4. de la convention d'appel de la fonction
5. du nom de la fonction
6. des paramètres de la fonction

3.2.1 Décoration du nom de la fonction (mangling)

Les compilateurs n'utilisent pas la même manière de nommer les fonctions dans les binaires générés.
Pour pouvoir utiliser une brique sans être obligé de la recompiler avec différentes version des compilateurs d'un système donnée, il faut utiliser la directive **extern "C"** qui impose la décoration basique du langage C compatible avec toutes versions de compilateurs sous un système donné.

Exemple Windows:

```
extern "C" __declspec(dllexport) int absimple_processAdd(int a, int b);
```

Exemple Linux:

```
extern "C" __attribute__ ((dllexport)) int absimple_processAdd(int a, int
b);
```

3.2.2 Directives d'export

Directive d'export sous Windows

Par défaut aucune fonction n'est exportée.

Chaque fonction exportable doit être déclarée avec la directive **__declspec(dllexport)**

```
extern "C" __declspec((dllexport)) int absimple_processAdd(int a, int b);
```

Directive d'export sous Linux et Macosx

Par défaut, toutes les fonctions sont exportées. Cela présente un risque de conflit quand un programme utilise plusieurs briques .

Pour éviter ce risque, la brique doit être compilée avec l'option **-fvisibility=hidden** qui annule l'export par défaut et chaque fonction exportable doit être déclarée avec la directive **__attribute__((visibility ("default")))**

Remarque: visibility ("default") ne signifie pas visible par défaut ! , mais signifie que la fonction doit être toujours visible de l'extérieur de la librairie.

Exemple

```
extern "C" __attribute__((visibility ("default"))) int
absimple_processAdd(int a, int b);
```

Directive d'import

Voir le principe de réutilisation du hearder d'export de la brique pour servir de header d'import dans un programme utilisateur écrit en C++ dans le chapitre « Interface d'import pour C++ »

3.2.3 Type de retour d'une fonction

Autorisé:

- les types de base
- les pointeurs de chaîne de caractère char *
- les pointeurs de structure ou d'objet
- les pointeurs de fonction

Interdit :

- les objets STL

Remarque: (à compléter)

retourner un variable sans nom qui sera libérée automatiquement par le compilateur à la fin de l'exécution de la fonction appelante .

convertir en const char * avec une fonction wrapper pour les autres langages

3.2.4 Convention d'appel

Les compilateurs permettent d'utiliser différentes convention d'appel __cdecl, __stdcall, __fastcall, etc...

Il est recommandé d'utiliser la convention d'appel **__cdecl** et d'éviter de panacher les conventions pour éviter d'avoir à préciser celle-ci dans les interfaces d'import de chaque langage.

Remarque: Le compilateur utilise par défaut la convention **__cdecl** si la convention d'appel n'est pas précisée dans la déclaration de la fonction. On peut donc omettre de préciser la convention d'appel dans la déclaration des fonctions des briques.

Exemple

```
extern "C" __declspec(dllexport) int absimple_processAdd(int a, int b);

équivalent à  :

extern "C" __declspec(dllexport) int __cdecl absimple_processAdd(int a, int
b);
```

3.2.5 Nom d'une fonction

Le nom des fonctions doit être préfixé par **le nom de la librairie** suivi d'un souligné **"_"** pour éviter les conflits quand plusieurs brique sont utilisées par un programme.

Exemple: le nom d'export de la fonction **processAdd** de la brique **absimple** est **absimple_processAdd**

```
extern "C" __declspec(dllexport) int absimple_processAdd(int a, int b);
```

3.2.6 Paramètres d'une fonction

Types de paramètres autorisés:

- les types de base (char, int, long, float, double)
- les pointeurs de chaîne de caractère (char *)
- les pointeurs d'instance de structure préalablement créés par la brique
- les pointeurs d'instance de classe préalablement créés par la brique.
- les pointeurs de fonction dont le type est prédéfini par la brique.

Types de paramètres interdit:

- pointeurs d'objets dont la structure ou la classe n'est pas définie dans la brique.
- les objets STL sauf si la fonction fait une recopie complète de l'objet et ne conserve pas de pointeur vers l'objet passé en paramètre.

3.3 Export d'une structure

3.3.1 Interface de structure

Une structure doit être exportée sous la forme d'une **interface structure.**

Exemple: la structure **SDateTime** est définie à l'intérieur de la brique. Cette définition n'est pas exportée.

```
struct SDateTime : public IDateTime {
   int year;
   int month;
   int day;
   int hour;
   int minute;
   int second;
   double julianDay;
};
```

La brique exporte les champs de la structure en définissant une nouvelle structure dite **structure d'interface** en appliquant les règles suivantes:

- seuls les champs que la brique veut bien export figurent dans la structure d'interface
- une fonction accesseur en lecture (get) est définie pour chaque champ que l'on veut autoriser à lire par le programme utilisateur.
- une fonction accesseur en écriture (set) est définie pour chaque champ que l'on veut autoriser à modifier par le programme utilisateur.

Par exemple l'interface de structure **IDateTime** exporte les champs exportables de **SDateTime** sous la forme suivante:

- la structure exportée **IDateTime** est préfixé par la lettre I , et contient seulement les

champs que la brique veut bien rendre visible pour un programme utilisateur écrit en C.

- les fonctions accesseurs pour les programmes utilisateurs écrits dans d'autres langages que C. Les noms des fonctions sont préfixés avec le nom de la structure **IDateTime_** . Le premier paramètre de chaque fonction est un pointeur de la structure d'interface pour identifier l'instance concernée.

```
struct IDateTime {
   int year;
   int month;
   int day;
};
```

3.3.2 Accesseurs aux champs de la structure

Les accesseurs aux champs d'une structure permettent de lire ou de modifier les champs de la structure.
Ces accesseurs sont en général facultatifs si les champs de la structure sont

Exemples d'accesseurs

```
extern "C" ABSIMPLE_API void IDateTime_year_set(IDateTime* dateTime,int
value);
extern "C" ABSIMPLE_API int  IDateTime_year_get(IDateTime* dateTime);
extern "C" ABSIMPLE_API void IDateTime_month_set(IDateTime* dateTime,int
value);
extern "C" ABSIMPLE_API int  IDateTime_month_get(IDateTime* dateTime);
extern "C" ABSIMPLE_API void IDateTime_day_set(IDateTime* dateTime,int
value);
```

3.3.3 Destruction d'une instance de structure

Si une instance structure est crée et retournée par une fonction de l'astrobrick, le programme utilisateur doit ensuite libérer la zone mémoire occupée par l'instance de la structure lorsqu'il n'en plus besoin.
Dans ce cas l'astrobrick doit fournir une fonction pour libérer la mémoire.

Exemple

```
extern "C" ABSIMPLE_API void IDateTime_releaseInstance(IDateTime* dateTime);
```

3.4 Export d'une classe

Une classe doit être exportée sous la forme

- d'une **classe d'interface**
- d'un **ensemble de fonctions** permettant d'accéder méthodes et aux attributs pour les langages autres que le C++
- d'une fonction constructeur et d'une fonction destructeur si la brique autorise la création et la destruction d'objets à l'extérieur de la brique.

Exemple: la classe **CCalculator** est définie à l'intérieur de la brique. Cette définition n'est pas exportée

```
class CCalculator : public ICalculator
{

public:
   CCalculator(void);
   void Release();
```

```
   double set(double x);
   double add(double x);
   double sub(double x);
   double clear();
   double clearMemory(void);
   double getMemory(void);
   double setMemory(void);
   double setMemoryPlus(void);
   double setMemoryMinus(void);

private:
   double current;
   double memory;
};
```

La brique exporte les méthodes de la classe qu'elle veut bien rendre visible en définissant une nouvelle classe dite **interface de classe** en appliquant les règles suivantes:

- toutes les méthodes de l'interface de classe sont abstraites
- les constructeurs et les destructeurs ne sont pas exportés.
- Il est recommandé de ne pas exporter les attributs, même public, pour des raison de compatibilité et d'évolutivité du code mais de les rendre accessibles uniquement via des accesseurs

Par exemple la classe d'interface **ICalculator** exporte les méthodes exportables de **CCalculator** ainsi qu'un constructeur et un destructeur.
Les noms des fonctions sont préfixés avec le nom de la classe **ICalculator_**
Le premier paramètre de chaque fonction est un pointeur de la classe d'interface pour identifier l'instance concernée, sauf pour la fonction constructeur.

```
class ICalculator
{
public:
   virtual void Release() = 0;
   virtual double add(double x)=0;
   virtual double sub(double x)=0;
   virtual double set(double x)=0;
   virtual double clear()=0;
};

// constructor
extern "C" ABSIMPLE_API ICalculator*  Icalculator_createInstance();
// destructor
extern "C" ABSIMPLE_API void ICalculator_releaseInstance(ICalculator*);

// declare class methods as standalone functions (used by other langages)
extern "C" ABSIMPLE_API double    ICalculator_add(ICalculator*, int a);
extern "C" ABSIMPLE_API double    ICalculator_sub(ICalculator*, int a);
extern "C" ABSIMPLE_API double    ICalculator_set(ICalculator*, int a);
extern "C" ABSIMPLE_API double    ICalculator_clear(ICalculator*, int a);
```

3.5 Export des codes d'erreur et des exceptions

Tous les codes d'erreur pouvant être retournés par les fonctions d'une brique sont listés décrite dans la classe d'interface **IError**. Chaque brique doit posséde une liste d'erreur spécifique selon son contenu applicatif.

Dans l'exemple ci-dessous, la brique **absimple** peut retourner 3 erreurs: GenericError, MemoryError, BatteryLow

Les exceptions sont retournée au programme utilisateur par une fonction callback à laquelle le programme utilisateur s'abonne avec la fonction **registerErrorCallback** .

Dans la brique, les fonctions exportables qui retournent une exception doivent appeler la fonction **throwExternalError** de la brique au lieu de la commande standard **throw**

```
class IError {
public:
   typedef enum {
     GenericError,
     MemoryError,
     BatteryLow
   } ErrorCode;

   virtual char * gets(void)=0;
   virtual unsigned long  getCode(void)=0;
};

typedef void (* ErrorCallbackType)(int code, const char * message);
extern "C" ABSIMPLE_API void absimple_registerErrorCallback(ErrorCallbackType
callback);
```

3.6 Options de compilation

3.6.1 Options de compilation sous Windows

Compilateur C++
Code generation > Runtime library= Multi-threaded (/MT)

- avantage: /MT permet d'éxécuter une astrobick compilée avec VisualC++ 2008 sur une machine ou est installé une version de runtime différente (runtime de Visual C++ 2012 par exemple)
- inconvénient: la librairie est plus volumineuse (100 Ko au lieu de 27 Ko)

3.6.2 Options de compilation sous Linux

3.7 Programme de test des exports

Le compilateur de la brique ne vérifie pas si toutes les fonctions implémentées sont réellement implémentées. Le problème n'apparaît qu'au moment de l'exécution et fait planter le programme utilisateur quand il appelle la fonction.

Pour éviter il faut écrire un programme de test qui appelle toutes les fonctions exportées et permet de s'assurer qu'il ne manque pas de fonction (ou que le nom n'est pas erroné).

Très important
Ce programme de test doit être exécuté **systématiquement** après chaque compilation de la brique sous peine de faire une belle brique inutilisable.

4. Interface d'import pour C++

La brique fournit :

- la librairie dynamique ex : **libabsimple.dll libabsimple.so libabsimple.dylib**
- le header d'import **absimple.h** que le programme utilisateur doit inclure dans ses sources avec la directive #include
- le fichier **absimple.lib** des adresses des fonctions à importer par un chargement statique
- le fichier **absimple_ld.h** des adresses des fonctions à importer par un chargement dynamique . le programme utilisateur doit inclure ce fichier dans ses sources avec la directive #include .

4.1 Header d'import de la brique

Le header d'import doit contenir la déclarations des fonction avec la directive d'import **__declspec(dllimport)** sous windows

Remarque : sous Linux et Macosx il n'existe de directive d'import. Toutes les fonctions visibles de la librairie sont utilisablesdans un programme utisateur.

Ces déclarations sont identiques à celles faites dans le header d'export de la brique à la seul différence de la directive d'import/export.

Exemple: déclaration d'import de la fonction absimple_processAdd

```
extern "C" __declspec(dllimport)  int absimple_processAdd(int a, int b);
```

à comparer avec la déclaration d'export qui a été faite dans le header d'export de la brique

```
extern "C" __declspec(dllexport)  int absimple_processAdd(int a, int b);
```

Aussi est-il possible d'utiliser le même fichier header avec une directive conditionnelle de compilation pour éviter d'avoir à maintenir deux fichiers header quasiment identiques.

Exemple de header **absimple.h** pouvant être utilisé à la fois dans l'interface d'import et dans l'interface d'export pour un programme C :

```
// absimple.h

#pragma once

#ifdef WIN32
#if defined(ABSIMPLE_EXPORTS)
#   define ABSIMPLE_API    __declspec(dllexport)  // inside DLL
#else
#   define ABSIMPLE_API    __declspec(dllimport)  // outside DLL
#endif

#else

#if defined(ABSIMPLE_EXPORTS)
#   define ABSIMPLE_API  __attribute__((visibility ("default"))) // inside DLL
#else
#   define ABSIMPLE_API       // outside DLL
#endif

#endif

extern "C" ABSIMPLE_API int  absimple_processAdd(int a, int b);
extern "C" ABSIMPLE_API int  absimple_processSub(int a, int b);
```

Le principe consiste à définir la variable de compilation ABSIMPLE_EXPORTS dans la commande de compilation de **l'interface d'export de la brique** faite par le développeur de la brique :

```
$ make
*** Compiling absimple.cpp
g++ -O2  -fPIC  -fno-stack-protector -c -Wall -DABSIMPLE_EXPORTS
-fvisibility=hidden  -I../../absimple/src -o absimple_test.o
../src/absimple_test.cpp
```

Tandis que cette variable ABSIMPLE_EXPORTS **n'est pas définie** dans la commande compilation de la l'**interface d'import du programme utilisateur**:

```
$ make
*** Compiling absimple_test.cpp
g++ -O2  -fPIC  -fno-stack-protector -c -Wall -I../../absimple/src -o
absimple_test.o ../src/absimple_test.cpp
```

Conclusion: En appliquant ce principe, le fichier header d'import est le même fichier que **absimple.h** qui sert de header d'export dans la brique.

4.2 chargement statique de la brique

Tous les systèmes d'exploitation proposent un mécanisme de chargement automatique de librairies dynamique appelé aussi « chargement statique d'une librairie dynamique »

Pour cela il suffit de déclarer le chargement automatique dans la commande d'édition de lien du programme utilisateur:

Exemple

```
*** Linking library  absimple_test
gcc absimple_test.o -ldl -lm -lstdc++ -Wl,-rpath,. -L../bin  -labsimple -o
absimple_test
$
```

Sous Windows

Il faut fournir à l'éditeur de lien le fichier .lib contenant les adresses des fonctions à importer de la brique et l'ajouter à liste des fichiers en entrée de la commande de l'éditeur de lien .

Remarque: si le fichier absimple.lib n'est pas fourni avec la brique, utilisateur peut le générer à partir de la librairie dynamique avec les utilitaires dumpbin.exe et lib.exe fournis avec le compilateur C++

```
dumpbin /exports libabsimple.dll > libabsimple.def
lib /def:DEF_libabsimple.def /out:LIB_libabsimple.lib /machine:x86
```

Conclusion: le fichier **libabsimple.lib** doit être fourni avec la brique pour que l'utilisateur puisse lier son programme avec la brique, sinon les utilisateurs doivent le générer eux-mêmes.

Sous Linux et Macosx

L'éditeur de lien de GCC extrait automatiquement les adresses des fonctions à importer à partir de la librairie dynamique.

Il suffit donc de fournir la librairie dynamique libabsimple.so ou libabsimple.dylib

4.3 chargement dynamique de la brique

Tous les systèmes d'exploitation permettent le «chargement dynamique d'une librairie dynamique» .

Le principe consiste à réécrire soit même dans le programme utilisateur un bout de code qui fait ce que font le compilateur et l'éditeur de lien dans la solution précédente.

Ce bout de code doit charger la librairie avec la commande **LoadLibrary** (ou **dlopen** sous Linux et Macosx) et récupérer les pointeurs de chaque fonction à importer avec la commande **GetProcAddress** (ou **dlsym** pour Linux et Macosx).

Pour aider l'utilisateur de la brique, la brique peut fournir le fichier **absimple_ld.h** qui contient une fonction qui fait toutes ces commandes.

TODO: je ne suis pas entièrement convaincu qu'il faille mettre en œuvre ce mécanisme car l'écriture du fichier **absimple_ld.h** peut prendre du temps au développeur de la brique en particulier pour les tests … à revoir si c'est vraiment nécessaire.

5. Interface d'import pour C#

La brique fournit :

- la librairie dynamique de la brique ex : **libabsimple.dll libabsimple.so libabsimple.dylib**
- le header d'import **absimple.cs**

Le header d'import est le même pour Windows, Linux et Macosx

Le header d'import utilise le mécanisme de chargement dynamique d'une librairie dynamique.

Le header doit contenir :

- la commande de chargement dynamique de la librairie
- les commandes de récupération des adresses des fonctions à importer
- les commandes de définition des numéros d'erreur et récupération des exceptions générées par la brique
- le commande de destruction automatique des chaînes de caractères retournées par certaines fonction de la brique

5.1 Chargement de la librairie

La commande **DllImport** permet de charger une librairie dynamique dans un programme C#.
Cette commande est répétée à l'import de chaque fonction.

5.2 Récupération des adresses des fonctions à importer

5.2.1 Fonction retournant un type de base

Le prototype de la fonction C# commence par « **public static extern** » suivi du type de base retourné par la fonction

```
[DllImport("libabsimple")]
public static extern int absimple_processAdd(int a, int b);
[DllImport("libabsimple")]
public static extern int absimple_processSub(int a, int b);
```

5.2.2 Fonction retournant un type complexe

Le prototype de la fonction C# commence par « **public static extern IntPtr**»

Exemple : la fonction retourne une structure

```
[DllImport("libabsimple"]
public static extern IntPtr ICalendar_convertIntToStruct(HandleRef
instancePtr, int year, int month, int day, int hour, int minute, int
second);
```

Exemple : la fonction retourne une chaîne de caractères

```
[DllImport("libabsimple", CharSet = CharSet.Ansi)]
public static extern IntPtr ICalendar_convertIntToString(HandleRef
instancePtr, int year, int month, int day, int hour, int minute, int
second);
```

5.3 Import d'une structure

5.3.1 Import d'une structure (méthode générique)

L'interface d'import d'une structure est une classe C# représentant la structure.

- Le constructeur doit conserver un pointeur **cPtr** sur l'instance C++ créé par l'astrobrick sous la forme d'une variable **HandleRef instancePtr**
- le destructeur doit appeler la fonction de libération de la mémoire **IDateTime_releaseInstance(instancePtr)**
- les accesseurs aux champs doivent appeler les accesseurs de l'interface de la structure.

Exemple d'import de la structure **IdateTime** :

```
public class DateTime : System.IDisposable
    {
        private HandleRef instancePtr;
        internal DateTime(System.IntPtr cPtr)
        {
            instancePtr = new System.Runtime.InteropServices.HandleRef(this,
cPtr);
        }

        ~DateTime()
        {
            Dispose();
        }

        public void Dispose()
        {
            lock (this)             {
                if (instancePtr.Handle != System.IntPtr.Zero)
                {
                    ISimple.IDateTime_releaseInstance(instancePtr);
                    instancePtr = new
System.Runtime.InteropServices.HandleRef(null, System.IntPtr.Zero);
                }
                System.GC.SuppressFinalize(this);
            }
        }

        public int year
        {
            set { ISimple.IDateTime_year_set(instancePtr, value); }
            get { return ISimple.IDateTime_year_get(instancePtr); }
        }

        public int month
        {
            set { ISimple.IDateTime_month_set(instancePtr, value); }
            get { return ISimple.IDateTime_month_get(instancePtr); }
        }

        public int day
        {
            set { ISimple.IDateTime_day_set(instancePtr, value); }
            get { return ISimple.IDateTime_day_get(instancePtr); }
        }
    }
```

5.3.2 Import d'une structure par copie d'instance

Si une structure est utilisée uniquement par une fonction de l'astrobrick pour retourner une valeur volatile, il est possible de faire une déclaration plus simple contenant seulement des attributs :

```
  [StructLayout(LayoutKind.Sequential)]
    public class DateTimeVolatile
    {
          public int  year;
          public int month
          public int day;
     }
```

Cependant la fonction retournant l'instance de la structure doit copier les données de l'instance de la structure créée par l'astrobrick avec **Marshal.PtrToStructure** puis supprimer cette instance de la mémoire avec **IDateTime_releaseInstance**

Exemple de méthode retournant une instance volatile de structure

```
public DateTimeVolatile getNow()
{
    IntPtr result = ICalendar.ICalendar_getNow(instancePtr);
    DateTimeVolatile dateTime = new DateTimeVolatile();
    Marshal.PtrToStructure(result, dateTime);
    Isimple.IDateTime_releaseInstance(result)
    return dateTime;
}
```

5.4 Import d'une classe

L'interface d'import d'une classe est une classe C#:

- Le constructeur doit conserver un pointeur **cPtr** sur l'instance C++ créé par l'astrobrick sous la forme d'une variable **HandleRef instancePtr**
- le destructeur doit appeler la fonction de libération de la mémoire **IDateTime_releaseInstance(instancePtr)**
- les méthodes de la classe C# doivent appeler les méthodes de l'interface de la structure. Les méthode qui retournent une valeur complexe volatile doivent copier la valeur **Marshal.PtrTo...** puis supprimer la variable de retournée par l'astrobrick avec une fonction **absimple_release.**.. fournie par l'astrobrick.

```
public class Calendar : System.IDisposable
{
    private HandleRef instancePtr;
     // constructor
    public Calendar()
    {
        System.IntPtr cPtr =
ISimple.ICalendar_createInstance(ErrorHelper.errorDelegate);
        instancePtr = new System.Runtime.InteropServices.HandleRef(this,
                                                                   cPtr);
    }

    ~Calendar()
    {
        Dispose();
```

```
    }

    public void Dispose()
    {
        lock (this)
        {
            if (instancePtr.Handle != System.IntPtr.Zero)
            {
                ISimple.ICalendar_releaseInstance(instancePtr);
                instancePtr = new
System.Runtime.InteropServices.HandleRef(null, System.IntPtr.Zero);
            }
            System.GC.SuppressFinalize(this);
        }
    }

    public string convertIntToString(int year, int month, int day,
                                     int hour, int minute, int second)
    {
        IntPtr charArrayPtr = Isimple.ICalendar_convertIntToString(
                        instancePtr, year, month, day, hour, minute, second);
        if (PendingError.Pending) throw PendingError.Retrieve();
        string value = Marshal.PtrToStringAnsi(charArrayPtr);
        ISimple.absimple_releaseCharArray(charArrayPtr);
        return value;
    }

    public DateTime convertIntToStruct(int year, int month, int day,
                                       int hour, int minute, int second)
    {
        IntPtr intPtr =Isimple.ICalendar_convertIntToStruct(
               instancePtr, year, month, day, hour, minute, second);
        if (PendingError.Pending) throw PendingError.Retrieve();
        return new DateTime(intPtr);
    }
}
```

5.5 Gestion des exceptions générées par l'astrobrick

6. Interface d'import pour Python

La brique fournit :

- la librairie dynamique de la brique ex : **libabsimple.dll libabsimple.so libabsimple.dylib**
- le header d'import **absimple.py**

Le header d'import est le même pour Windows, Linux et Macosx

Le header d'import utilise le mécanisme de chargement dynamique d'une librairie dynamique.

Le header doit contenir :

- la commande de chargement dynamique de la librairie
- les commandes de récupération des adresses des fonctions à importer
- les commandes de définition des numéros d'erreur et récupération des exceptions générées par la brique.
- le commande de destruction automatique des chaînes de caractères retournées par certaines fonction de la brique

6.1 Chargement dynamique de la librairie

La commande **LoadLibrary** permet de charger une librairie dynamique dans un programme python.

Exemple

```
import ctypes

_absimple = ctypes.cdll.LoadLibrary(ctypes.util.find_library('libabsimple'))
```

6.2 Récupération des adresses des fonctions à importer

6.2.1 Fonction retournant un type de base

Exemple

```
def processAdd(*args):
      return _absimple.absimple_processAdd(*args)

def processSub(*args):
      return _absimple.absimple_processSub(*args)
```

6.2.2 Fonction retournant un type complexe

6.2.3 Gestion des structures

6.2.4 Gestion des classes

6.3 Définition des numéros d'erreur

6.4 récupération des exceptions générées par la brique

7. Interface d'import pour Tcl

La brique fournit :

- la librairie dynamique de la brique
  ex : libabsimple.dll libabsimple.so libabsimple.dylib
- la librairie dynamique d'interface pour le TCL
  ex : libabsimple_tcl.dll libabsimple_tcl.so libabsimple_tcl.dylib
- le fichier **pkgindex.tcl** de chargement de la librairie d'interface

7.1 Librairie d'interface TCL

La librairie d'interface TCL est écrite en C et a pour but de traduire les commandes lancées depuis l'interpréteur TCL en appel des fonctions de la brique.

Pour cela il faut développer une fonction C de traduction pour chaque fonction à importer de la brique :

- déclarer la fonction de traduction en lui associant un nom dans l'interpréteur TCL , par exemple « absimple_processAdd"
- implémenter le corps de la fonction qui doit
  1. Parser les paramètres TCL et les convertir en type de base ou pointeur d'objet complexe
  2. Appeler l'exécution de la fonction de la librairie
  3. Analyser le résultat et éventuellement intercepter les exceptions
  4. Convertir le résultat en chaîne de caractère et le retourner à l'interpréteur TCL , ou en cas d'errer récupérer le code d'erreur et le message de l'exception et le retourner à l'interpréteur TCL

Exemple : traduction pour la fonction **absimple_processAdd**

```
// déclaration de la fonction de traduction
Tcl_CreateCommand(interp,"absimple_processAdd",(Tcl_CmdProc
*)cmdSimpleProcessAdd,(ClientData)NULL,(Tcl_CmdDeleteProc *)NULL);


/**
   additionne deux nombres
   @return retourne le résultat de l'addition
*/
int cmdSimpleProcessAdd(ClientData clientData, Tcl_Interp *interp, int argc,
const char *argv[]) {
   const char *usage = "Usage: absimple_processAdd a b";
   if(argc != 3) {
      Tcl_SetResult(interp,(char *) usage,TCL_VOLATILE);
      return TCL_ERROR;
   }

   int tclResult;
   try {
      int result = absimple_processAdd(copyObjToInt(interp, argv[1], "a"),
copyObjToInt(interp, argv[2], "b") );
      Tcl_SetObjResult(interp, Tcl_NewIntObj(result));
      tclResult = TCL_OK;
   } catch(std::exception e ) {
      Tcl_SetResult(interp,(char*)e.what(),TCL_VOLATILE);
      tclResult = TCL_ERROR;
   }
```

```
    return tclResult;
}
```

7.2 Fonction retournant un type complexe

7.3 Gestion des structures

7.4 Gestion des classes

7.5 Définition des numéros d'erreur