Farfelue: une nouvelle bibliothèque IHM

NOTA : Le présent document reprend l'ensemble des caractéristiques que devra respecter le projet tel qu'il est envisagé en l'état, mais est susceptible de modifications au plus tôt jusqu'à la mise en production définitive.

N'hésitez pas à consulter la version dernière version disponible sur le SVN lorsque le projet est lancé.

Versions:

1.0.3 : Le 12/04/2010

Modification : choix « provisoirement définitif » de la LGPL v3 comme licence pour le lancement du projet. Le choix de cette licence peut être remis en cause par la suite

1.0.2 : le 12/04/2010

ajout des convention de nommage des fichiers,

ajout des convention utilisées pour les extensions des fichiers,

ajout des convention sur l'implémentation des fontions inline et assimilées

1.0.1 : le 12/04/2010

modification: les fichiers en attente de validation sont dans la branche dev

ajout de « source d'inspiration »

adaptation de règles de formatage

ajout de Aristic Style

1.0 : le 12/04/2010

version de base du document.

Abstract

Le développement de nombreuses bibliothèques permettant de créer des interfaces homme machine (IHM ou GUI) a débuté à une époque où C++ n'en était encore qu'à ses balbutiements, où l'on ne jurait que par le « tout objet », et où les approches multi-paradigme étaient rares et difficiles à mettre en oeuvre.

Certaines d'entre elles nécessitent en outre des dépendances telles qu'il est assez difficile de les compiler soi-même sur certaines plateformes.

Bien qu'il ne soit pas question de remettre la qualité de ces bibliothèques en question, l'approche à grand pas de la version stable de C++11 (la prochaine norme à venir), l'évolution des techniques et la maitrise de plus en plus importante de l'approche multi-paradigme nous incite à revoir entièrement les bases même de la création d'IHM, et donc à lancer un projet tout à fait nouveau sur ces nouvelles bases.

Farfelue poursuit ces objectifs.

Pourquoi farfelue?

Si vous vous demandez les raisons qui ont mené au choix de farfelue comme nom de projet, elles sont toutes simples: il a été choisi suite à une discussion dans laquelle je parlais d'une « idée assez

farfelue » afin de déterminer la faisabilité et l'intérêt d'un tel projet, et que le terme est parfaitement prononçable aussi bien en français qu'en anglais.

Les fondements du projet

La décision de créer une nouvelle bibliothèque permettant de créer des IHM a été prise pour corriger certains « travers » de l'existant. Il ne s'agit donc pas d'une « énième » bibliothèque, ni de copier « bêtement » ce qui a été fait, mais bien de profiter de l'expérience acquise par les autres projet pour éviter de tomber dans leur travers.

A ce titre, Farfelue:

- Se veut portable.
- Se veut non intrusive à l'utilisation (n'incitant pas à préférer des types propres aux types fournis par la S(T)L ou par boost).
- Se veut non intrusive à la compilation (la compilation d'une application ne doit en rien modifier les habitudes de chacun en ce qui concerne la gestion d'un projet)
- Veut promouvoir un style de codage moderne, c'est à dire faisant intervenir l'ensemble des paradigmes accessibles en C++.
- doit découpler les différents aspects du problèmes, tels que le rendu ou le traitement des messages.
- Refuse le recours à un « super objet » chapeautant l'ensemble des classes que la bibliothèque repose, préférant une hiérarchie basée sur les traits et politiques template.
- Se veut accessible : L'utilisation « basique » de la bibliothèque doit être facile y compris pour ceux qui débutent en programmation, mais la bibliothèque ne doit pas brider l'imagination des développeurs chevronnés.
- Utilise l'ensemble des « nouvelles features » de la nouvelle norme (C++11).

La licence

La bibliothèque doit être fournie dans une licence open source aussi peu contaminante que faire se peut.

A l'heure actuelle, LGPL v3 a été sélectionnée, jusqu'à ce que preuve soit faite qu'elle est inappropriée.

Style de codage et conventions

Il est impossible d'envisager la tenue d'un projet de telle ampleur sans assurer un minimum de cohérence au point de vue des conventions de nommage, du style de codage ou de l'organisation des répertoires au sein de l'équipe.

C'est pourquoi, les règles suivantes seront de stricte application, le responsable de projet s'octroyant le droit de supprimer à vue et de restaurer une ancienne version de tout fichier ne les respectant pas

Style de codage

.

- Une seule instruction par ligne
- Une seule déclaration de variable par ligne
- Pas de déclaration de variable séparée par une virgule: le code
  `int i, j, k;`
  sera refusé alors que le code

```
int i;
int j;
int k;
```
est attendu

- Mise en forme selon la norme ANSI, c'est à dire sous la forme de
```
int Foo(bool isBar)
{
    if (isBar)
    {
        bar();
        return 1;
    }
    else
        return 0;
}
```
- Les accolades ne sont pas nécessaires lorsqu'il n'y a qu'une instruction à effectuer dans un test ou dans une boucle, cependant, l'instruction doit se trouver à la ligne et correctement indentée.
- Les sucres syntaxique et les formes de « shadow programming » sont proscrits, exception faite des sucres syntaxique pour lesquels une incidence réelle sur les performances a été mise en avant par des bench clairs et détaillés, ou que l'impossibilité de faire autrement ne soit avérée (par exemple: l'utilisation de l'opérateur ternaire `? Blabla : truc ;`)
- Afin d'assurer une mise en forme identique dans les différents éditeurs, la mise en forme est assurée à l'aide d'espaces plutôt que de tabulation. Un nombre de 4 (quatre) espaces successifs sera utilisé pour l'indentation
- Dans certains cas, une indentation supplémentaire peut être appliquée à seule fin de proposer un alignement facilitant la lecture comme par exemple avec l'opérateur <<
```
std::cout<< truc1 << std::endl
         << truc2 << std::endl;
```
- La largeur du code est limitée à 80 colonnes à peu près, un dépassement de deux ou trois caractères maximum est autorisé.
- L'utilisation de la directive ***using namespace*** est proscrite dans les fichiers d'en-tête ou, de manière générale, dans les fichiers destinés à être inclus dans d'autres.
- L'utilisation de la directive ***using namespace*** dans les fichiers d'implémentation est proscrite. L'utilisation d'alias d'espaces de noms est largement conseillée pour les espaces de noms imbriqués ou trop long à réécrire à chaque fois.
- L'utilisation des listes d'initialisation au sein des constructeurs est à utiliser chaque fois que possible. Seuls les cas où leur utilisation est impossible admettent de s'en passer.
- Les symboles préprocesseurs sont écrit en majuscule.
- Les gardes anti inclusion multiple définissent le nom du fichier en majuscules en remplaçant le point par un underscore (ex : `#ifndef MACLASS_H`)
- un espace est obligatoire après une parenthèse ouvrante, un opérateur ou une virgule, ainsi qu'avant une parenthèse fermante, à moins qu'il n'y ait césure
- L'utilisation de parenthèses est obligatoire si des opérateurs risquent, de par les règles de priorité, de modifier l'ordre d'évaluation des expression (ex: `if( i == 1 && j == 2 || i == 3 && j == 4 )` nécessite des parenthèses)
- Elle est souhaitable lorsque les expressions deviennent trop longues ou compliquées à comprendre. Les expressions sont alors alignées selon leur imbrication
- La césure peut survenir après une parenthèse ouvrante ou fermante, une virgule ou avant une parenthèse fermante s'il y a déjà eu césure et alignement.
- La césure d'expressions complexes est souhaitable dés qu'il y a plus de deux sous expressions. Les différentes expressions sont alors alignées. Par exemple:

`if( i == 1 && j == 2 || i == 3 && j == 4 )` devrait prendre la forme de

```
if ( ( i == 1 && j == 2 ) ||
  ( i == 3 && j == 4 ) )
```

exemple de césure suite à la longueur des expressions

```
for_each( un_truc_tellement_long_qu_il_est_illisible.begin(),
          un_truc_tellement_long_qu_il_est_illisible.begin(),
          bind2nd( truc,
                   valeur)
        )
```

Vérification de la mise en forme

L'utilisation de Artistic Style pour assurer la mise en forme du code est conseillée avant d'effectuer un commit.
La ligne de commande conseillée pour cet outil est

```
astyle --style=allman --indent=spaces=4 --indent-col1-comments
--align-pointer=name --indent-switches --indent-preprocessor
--break-blocks --pad-oper --pad-paren --pad-header --add-brackets
--convert-tabs monfichier1.cpp monfichier1.h
```

Commentaires et documentation

Chaque implémentation de fonction sera précédée d'un commentaire indiquant:

- le nom de la fonction
- le nom, le type et l'utilité des arguments transmis
- le type de retour
- Les exceptions susceptibles d'être lancées, s'il y en a
- une description des services que l'on attend de la fonction

Afin de faciliter la génération de documentation, ces commentaires seront au format « doxygen », qui semble être l'outil le plus facile à mettre en oeuvre.

Les commentaires ne nuisant jamais en rien, pour autant qu'ils apportent une précision utile quant à ce qui est fait, leur utilisation est vivement conseillée dans le corps des fonctions, cependant, l'auteur veillera en permanence à n'écrire que des commentaires utiles.

Sont considérés comme commentaires inutiles, ceux qui ne font que préciser ce que le code indique, Par exemple, dans le code:

```
void foo()
{
    /* du code avant */
    i++; // incrémente i
    /* du code après */
}
```

Le commentaire « incrémente i » est parfaitement inutile car il ne fait qu'indiquer ce que le code exprime très clairement.

Les commentaires s'écrivent soit sur une ligne précédant directement l'instruction ou le groupe d'instructions à laquelle / auquel il se rapportent, soit directement sur la même ligne, après l'instruction à laquelle ils se rapportent.

La préférence sera cependant donnée à la première possibilité.

Les éléments globaux permettant de générer la documentation (comme la description d'un espace de noms, les exemples, …) seront placé dans des fichiers *.dox se trouvant dans un dossier séparé (doxygen)

Les espaces de noms

Il est important de séparer clairement ce qui fait partie de la bibliothèque de ce qui n'en fait pas partie, tout comme il est important de clairement identifier les différents aspects au sein de la bibliothèque.

C'est pourquoi l'utilisation d'espaces de noms est primordiale dans la structure de la bibliothèque.

Un espace de noms « **Farfelue** » servira d'espace de noms global à la bibliothèque, dans lequel nous trouverons les espaces de noms

- **Kernel** : contenant tout ce qui a trait au noyau de la bibliothèque
- **Renders** : contenant tout ce qui a trait au système de rendu. Cet espace de noms peut par la suite, recevoir des espaces de noms différents pour chaque aspect de rendu envisagé (3D, OpenGl, …)
- **Signals** : contenant tout ce qui a trait à la gestion des signaux
- **Managers** : Nous aurons sans doute besoin d'un certain nombre de gestionnaires en tout genre... Ils prendront place dans cet espace de noms

Chaque espace de noms peut en outre contenir un espace de noms nommé « Prive » regroupant les classes, helpers, et fonctions destinées à un usage interne uniquement, si le besoin s'en fait sentir

D'autres espaces de noms pourront apparaître en cours de développement, par exemple, lorsqu'il s'agira de s'attaquer à la partie « declarative UI », ou à la mise au point d'un RAD.

Conventions de nommage

Il semble opportun de permettre la distinction claire et précise entre ce qui est fourni par la bibliothèque et ce qui est fourni par boost ou la STL.

Les conventions de nommage ci-dessous seront donc appliquées

- La première lettre de chaque mot des noms de classe, de structure, d'union ou d'énumération est une majuscule, ce qui nous donne `MachinChose`, par exemple. Les noms ne sont pas préfixés, exception faite des modèles de classe (`CmaClasse` ou `SmaStructure` sont interdits, par contre `template<class T> TmaClass` est admis et imposé (cf plus loin))
- Exception faite des valeurs énumérées utilisées dans les classes template, le nom d'un valeur énumérée commence par les initiales de l'énumération en minuscule suivi par le nom de la valeur, la première lettre de celui-ci étant une majuscule (ex: `meValeurEnumeree` dans l'énumération `MonEnumeration`)
- Le nom des valeurs énumérées utilisées dans les énumérations anonyme sont entièrement en minuscules et ne sont pas préfixées
- La première lettre d'une fonction, qu'il s'agisse d'une fonction membre ou d'une fonction libre, est une minuscule, l'initiale des mots suivant est en majuscule (ex: `void faitMachinChose`)
- Afin de clairement distinguer les modèles de classes, leur nom commence obligatoirement par un T majuscule, suivi du nom de la classe en respectant la première règle énoncée
- Le nom des membres privés est écrit exclusivement en minuscule et suffixé de l'underscore « _ » (ex : `membre1_`)
- Le nom des champs de structure ou d'union est écrit exclusivement en minuscule et non

suffixé

- Le nom des variables, des membres publiques et des arguments est écrit en minuscules et non préfixé, sauf pour les arguments à transmettre au constructeur
- Si nécessaire, le nom des arguments transmis au constructeur être celui du membre de la classe qui sera initialisé grâce à l'argument (ex: `MaClasse( int mavar_): mavar_(mavar_) )`

Organisation du code, des fichiers et des dossiers.

Nous serons rapidement confrontés au fait que les différents développeurs travaillent avec des outils, des compilateur ou sur des architectures différentes, et il est hors de question de privilégier un outil, un compilateur ou une architecture que ce soit pour le développement ou lors de la mise en production.

Il est donc important de séparer clairement et directement les fichiers propres à ces différents outils.

C'est pourquoi, l'arborescence suivante sera utilisée:

Dossier racine

CodeBlock : dossier contenant l'ensemble des fichiers nécessaires à l'utilisation sous codeblocks

VisualStudio : dossier contenant les fichiers dédiés aux outils propres à visual studio, avec subdivision par versions

VS2005
VS2008
VS2010
…

autotools : dossier contenant les fichiers à utiliser pour autotools (ex: script «configure » Makefile.am Makefile.in,...)

Cmake : dossier contenant les fichiers utiles à Cmake

src : dossier contenant les fichiers d'implémentation (*.cpp)

kernel : ce qui entre dans Farfelue::Kernel

render : ce qui entre dans Farfelue:: Render

signals : ce qui entre dans Farfelue::Signals

managers : ce qui entre dans Farfelue::Managers

<nom a déterminer> dossier contenant les fichiers nécessaires pour assurer la compatibilité avec une architecture ou un outil particulier

include : dossier contenant les fichiers d'en-tete

impl : dossier contenant les fichier d'implémentation de template (*.tpp)

prive : dossier contenant les déclaration de fonctions, classes et helpers à usage interne uniquement, si besoin

<nom à déterminer> : dossier contenant les fichiers d'en-tête spécifiques à un compilateur ou une architecture spécifique

doxygen : dossier contenant les fichiers nécessaires à la génération de documentation

doxygen.dox : fichier de configuration de doxygen

test : dossier contenant les sources des tests unitaires

demo (ou exemples) : dossier contenant les démos et les exemples proposés (quand il y en aura)

README : fichier qui explique un peut ce qu'est Farfelue

licence : fichier contenant la licence

ChangeLog : fichier contenant les informations de versions

INSTALL : fichier contenant les instructions pour l'installation

authors : fichier reprenant la liste des personnes ayant participé en tant qu'auteur au

développement (une section « special tanks » sera prévue pour les contributeurs qui auront permis au projet d'avancer par leur soutien et leurs idées)

Nota: Cette arborescence n'est sans doute pas complète et peut sembler complexe, mais il est à mon sens primordial de respecter une logique stricte afin d'éviter qu'un système donné ne prenne l'ascendant sur un autre, et d'en arriver à donner l'impression que cela ne fonctionne « qu'avec tel ou tel système ».

Les extensions de fichiers.

Quatre extensions sont utilisables en dehors de celles propres aux outils de compilation:

*.cpp : fichier d'implémentation de fonctions non modèles ou de fonction membre de classe non modèles et non inlinée. Ces fichiers trouvent leur place dans le dossier src

*.h pour les fichiers d'en-tête. Ces fichiers prennent place dans les dossiers include et dans les sous dossiers de include, autre que impl selon leur utilité

*.tpp : fichier d'implémentation des fonctions template et des fonctions membres de modèles de classe. Ces fichiers prennent place dans le dossier include/impl

*.dox : fichier contenant les informations générales permettant la génération autmatique de la documentation avec doxygen. Ces fichiers prennent place dans le dossier doxygen.

Convention de nommage à appliquer aux fichiers

Les noms de fichiers relatifs à une classe ou à une structures particulières sont nommés du nom de la classe ou de la structure, en respectant la même casse.

Le nom des fichier d'en-tête et les fichiers d'implémentation spécifiques à une architecture ou à un compilateur sont composés exclusivement de minuscules, en veillant à choisir le nom de manière à rappeler l'objectif de compatibilité à assurer.

Les noms de fichiers d'en-tête et d'implémentations qui ne se rapportent pas à une classe ou à une structure particulières (par exemple: le fichiers regroupant plusieurs fichiers d'en-tête pour la facilité, ou déclarant / implémentant plusieurs fonctions libres non modèles) sont composés exclusivement de minuscules, en veillant à choisir le nom de manière à rappeler l'objectif poursuivi par le contenu du fichier

implémentation des fonctions inlines et assimilées

L'implémentation des fonctions non modèles inlines se fait dans le fichier d'en-tête dans lequel elles sont déclarées.

L'implémentation des modèles de fonction ou des fonctions membre de modèles de classes peut se faire soit dans le fichier d'en-tête dans lequel elles sont déclarées soit dans un fichier d'implémentation séparé (*.impl).

Ajouts et modifications du code

Sauf cas de correction de bug flagrant et facile à résoudre, les modifications apportées au code le sont dans une branche « trunk » du projet présentant l'état journalier du projet, en attente de validation.

Seuls les ajouts et les patches dument validé passeront dans la branche accessible au publique.

La validation consiste en la revue de code et en l'exécution de tests unitaires afin de vérifier que les modifications et ajouts proposés ne provoquent pas de régression et sont exempts de problème

Lorsqu'un ajout ou une modification importante est proposé(e), il (elle) doit être accompagné(e) des tests unitaires permettant la validation de l'élément.

Les tests unitaires seront effectués en utilisant boost tests

Organisation de l'équipe

L'équipe de développement est destinée à s'enrichir au fur et à mesure en fonction des disponibilités et de l'intérêt de chacun.

Elle sera dans un premier encadrée par koala01 (Philippe Dunski), qui est l'instigateur du projet.

Toute personne est susceptible de participer au développement selon ses aptitudes et ses disponibilités.

- Par participation, on entend:
- le fait de proposer des amélioration de desing,
- le fait de proposer des patches / corrections / traductions,
- le fait de proposer l'ajout de fonctionnalités,
- le fait de faire remonter un bug,

Tout contributeur peut demander ou se voir proposer l'intégration au sein de l'équipe. Cependant, toute intégration est soumise à l'approbation de l'équipe existante et doit obtenir au moins la majorité lors d'un vote.

Le vote sera précédé d'un débat afin de juger du ressenti de l'équipe en ce qui concerne l'intégration de la personne pressentie.

En cas de partage ne permettant pas de prendre une décision uniquement, le responsable de projet peut faire valoir une voix comptant double afin de faire pencher la décision de manière claire.

Objecif prioritaire

L'objectif prioritaire lorsque le projet démarre est d'arriver à fournir une interface minimale de base permettant de présenter les bases du projet.

Il sera donc nécessaire de permettre:

- la création d'une fenêtre, de boutons, de « labels » de listes déroulantes et de menus simples
- la transmission de messages entre ces différents éléments et les données utilisées par ailleurs (entre autres dans une partie métier de l'application)
- le rendu simple (2D, utilisant les primitives de l'OS) des éléments

Pour la facilité, cette version de la bibliothèque sera essentiellement implémentée sous la forme de bibliothèque statique, mais la possibilité de créer une bibliothèque dynamique (dll / so) doit être envisagée et prise en compte.

Nota: Il est également important de prendre en compte certains objectifs à terme dans le desing général de cette première implémentation, autrement, les modifications à apportées par la suite seront beaucoup trop importantes pour nous permettre de les atteindre sans devoir « tout détruire ».

Objectif de mise en production

- Une fois l'objectif prioritaire atteint, il sera possible d'envisager la mise en production.
- Cependant, une série non négligeable d'améliorations, de tests et d'ajouts sera nécessaire.
- Parmi celles-ci, on peut citer:

- la possibilité de créer les « widget » supplémentaires et « indispensables » non implémentés pour l'objectif prioritaire.
- L'intégration aisée d'images dans les différents « widgets »
- La création de tutoriels adaptés aussi bien aux débutants qu'aux développeurs chevronnés
- Une batterie de tests systématiques en charge afin de s'assurer de la stabilité et de la qualité de l'existant
- Faire subir le « test du singe » à la bibliothèque : demander à quelqu'un de « pas trop doué » d'essayer de crér une application simple sur base des seuls tutoriaux donnés.
- Peaufiner la documentation
- …

Objectifs à terme

Il y aura à terme une série d'améliorations et de possibilités supplémentaires à envisager.

De manière non exhaustive, on peut citer:

- l'intégration du concept de « declarative UI »
- la mise au point d'un RAD adapté
- L'intégration du support de la 3D et de différentes technologies émergeantes
- Tout un travail destiné à assurer la portabilité vis à vis de compilateurs plus anciens (ne respectant pas entièrement ou pas du tout la norme C++11)
- l'intégration au sein d'outils particuliers (eclipse, Visual Studio, …)
- l'intégration de fonctionnalités demandées par les utilisateurs
- la possibilité de créer des widget « non rectangulaires »

Sources d'inspiration

Quelques bibliothèques, donc certaines sont plus ou moins abandonnées peuvent servir d'inspiration dans le design ou dans la mise en oeuvre.

Il serait dommage de se passer de leurs enseignements.

En voici une liste non exhaustive

- Le code de la STL et de boost peuvent nous donner des idées d'aménagement
- Loki : bibliothèque qui fait la part belle à la programmation générique
- ASL : une bibliothèque graphique basée sur des objets clairement distrincts du point de vue de l'héritage, visiblement orientée vers la « declarative UI »
- Notus: projet visiblement « zombie » qui semble très proche des objectifs recherchés
- Qt, WxWidget... : une série de bibliothèques bien implantées qui ont surement quelques bonnes idées