

Restructuration du projet Fudaa

Suite à la réunion du 11 Mai, voici un document qui devrait permettre à tous les développeurs de Fudaa de partager autour de la restructuration du projet. De la part des développeurs Fudaa, il est attendu une relecture du document, des propositions quant aux nommages des modules/dossiers, des choix d'outils,...

1. Objectif du document	2
2. L'état actuel	2
3. Objectifs de la restructuration	2
4. Macro-Planning de la restructuration	3
1. Phase 1: version 1.0	3
2. Phase 2: version 1.1	3
3. Phase 3: version 1.2	3
5. Glossaire	3
6. Les 2 arborescences	4
7. Objectifs du découpage	5
8. L'outil Maven	5
9. Organisation d'un module commun	6
1. Les fichiers de configuration des IDE et des autres outils	6
2. Les classes d'exemples.....	7
3. Les classes de tests	7
4. Les tests graphiques	7
10. Organisation d'un module applicatif	7
1. Note sur le vocabulaire serveur/dodico/métier et fudaa/client	7
2. Arborescence d'un module applicatif	8
11. Proposition de découpage	9
1. Ctulu	9
2. Dodico	9
1. Les modules commun de dodico	9
2. Les modules applicatif	9
3. Ebli.....	10
4. Fudaa	10
12. Gestion des versions	10
1. Gestion des inter-dépendances.....	10
2. La notion de SNAPSHOT	11
3. Usage du gestionnaire de sources: tags/branches	12
13. Convention de nommage/Formatter les sources	12
1. Convention de nommage	12
2. Formateur de source.....	12
14. Nouveaux outils	13
1. Gestion des bugs/exigences	13
2. Moteur d'intégration continue	13
3. Suivi de la qualité du code	13
15. Nouvelles librairies	13
1. Gestion des logs.....	13
2. Outils Apache.....	14
3. Nouveau framework swing.....	14
4. Format d'échange des documents	14

Objectif du document

Ce document est un document de travail et n'est pas une proposition définitive.

L'objectif de ce document est de présenter la restructuration du projet Fudaa et de faire valider cette proposition par tous les développeurs gravitant autour de Fudaa.

L'état actuel

Tout le projet Fudaa se trouve dans la même arborescence sur le gestionnaire de source. Dans cette arborescence, les modules communs (les briques de base utilisées par plusieurs applications) et les modules applicatifs se trouvent dans le même espace projet. Cela avait pour avantage de simplifier la propagation des modifications. Par contre, toute application était impactée par des modifications sur les modules communs et chaque développeur devait avoir une copie de la totalité du projet afin de travailler sur son application.

Cette organisation a atteint ces limites et c'est suite à ces divers constats qu'une réorganisation de Fudaa a été décidée.

Objectifs de la restructuration

Les objectifs sont multiples et ils sont détaillés dans le CR de la réunion du 11 Mai 2009.

- Se séparer de l'architecture monolithique du projet actuel qui demande de télécharger tous les projets Fudaa afin de développer son application
- Modulariser le développement
- Simplifier la prise en main du projet
- Limiter les effets de bord induits par les développements sur les modules communs
- Se séparer des modules désuets facilement

La restructuration devra permettre d'isoler les modules communs des modules applicatifs. Elle permettra à chaque application d'avoir son propre cycle de vie: elle sera indépendante du développement des autres applications et des évolutions des modules communs. Avec cette architecture, c'est le(s) responsable(s) d'une application qui prend la décision de mettre à jour son application vis-à-vis des évolutions des modules communs de Fudaa.

La restructuration consiste à créer un espace projet pour chaque module commun ou applicatif. Les inter-dépendances se feront pas la mise à disposition des jars.

C'est l'outil [Maven](#) qui sera utilisé comme outil de build. Cet outil permet de gérer les inter-dépendances directes et indirectes (transitives). Ainsi, si un module applicatif dépend de ebli-common version 1.0.0 qui dépend de ctulu-common en version 1.1.0, Maven téléchargera (si nécessaire) les jars ebli-common-1.0.0.jar et ctulu-common-1.1.0.jar. Il compilera ensuite le projet en utilisant ces 2 dépendances.

Cette restructuration demandera plus de rigueur à chaque développeur surtout si ce dernier intervient sur un module commun. Avant toute modification (d'importance je dirais. On peut peut être s'en passer pour des modifications mineure), il faudra faire part de ses besoins aux autres développeurs, s'assurer de la non-régression sur le module (en lançant les tests unitaires du module), s'assurer que les dépendances restent homogènes vis à vis des autres modules communs et d'une compatibilité ascendante ou

d'une migration simple avec une documentation adéquate. Finalement, il devra livrer la nouvelle version tout en fournissant la liste des bugfixs/améliorations.

Pour chaque module commun, une roadmap sera établie (via un outil de gestion de bogues comme Trac ou Jira).

Macro-Planning de la restructuration

Phase 1: version 1.0

La première partie de la restructuration consistera à redécouper les projets actuels mais sans toucher au code source. Cette première phase doit permettre de conserver le fonctionnement actuelle des applications et minimiser les régressions.

Cette première phase commencera dès que tous les acteurs Fudaa auront validé ce document. Pendant cette phase, il faudra limiter le travail sur l'ancienne arborescence fudaa_devel. Dans le cas contraire, il est fortement conseillé d'avertir Frédéric Deniger afin de prendre en compte les modifications et de les reporter dans la nouvelle structure.

Phase 2: version 1.1

La deuxième partie prendra en compte les modifications apportées par la branche Modeleur.

Cette phase commencera dès que Deltacad aura reversé son travail dans

Phase 3: version 1.2

Des actions pourront être menées afin de rationaliser les dépendances et d'introduire de [nouveaux outils](#).

En parallèle à ces 3 phases, le CETMEF va commencer la mise en place du serveur dédié. Dans l'état actuel, la restructuration n'est pas dépendante de ce travail et nous disposons des outils minimaux (serveur FTP) pour mettre en place la restructuration.

Glossaire

- **package (ou paquetage en français)**: notion java qui permet de grouper des classes java. Voir http://en.wikipedia.org/wiki/Java_package
- **Module ou projet**: un ensemble de package(s) homogène(s) permettant de fournir un ou plusieurs services (un module commun) ou une application autonome. Un module peut dépendre d'un ou plusieurs autres modules et de librairies externes. Un module doit permet de générer un ou plusieurs jars. Par exemple, il est possible de générer en même temps un jar contenant les classes propres du module, un jar contenant les tests,...
- **Librairie externe**: un jar fourni par une entité tiers (Apache, geotools,...). Cette librairie doit avoir une licence compatible avec la licence GPL v2.
- **Module applicatif**: un module permettant de construire une application finale.
- **Module commun**: module non applicatif offrant des services de base utilisables par plusieurs autres modules. Par exemple, le module ebli-1D fournit des composants permettant de visualiser des graphes/courbes.

- **Espace d'un module ou espace projet:** répertoire contient tous les ressources nécessaire à un module: les sources, les sources des tests, les ressources nécessaires, la documentation de développement,....
- **SVN:** raccourci pour désigner l'outil de gestion de sources Subversion
- **Arborescence:** URL d'un espace projet dans le référentiel Subversion. Par exemple l'arborescence de l'aide dans le référentiel actuel est http://fudaa.svn.sourceforge.net/viewvc/fudaa/trunk/fudaa_devel/aide/. Il est aisé de faire le lien entre l'URL précédente et le paramétrage nécessaire au client SVN pour récupérer les sources. Pour simplifier les notations, les chemins pourront être données à partir de trunk. Ainsi, nous parlerons de fudaa_devel/aide au lieu du lien complet.
- **Couche (ou Tiers):** on pourra parler de couche pour désigner un ensemble de module homogène: la couche ctulu, la couche dodico, Voir http://fr.wikipedia.org/wiki/Architecture_3-Tiers . Attention au terme Tiers (anglicisme) qui veut plus dire Couche que 1/3...
- **Branche:** c'est une notion du gestionnaire de source qui définit une "dérivation" dans les versions d'un fichier/dossier. Voir [http://fr.wikipedia.org/wiki/Branche_\(gestion_de_configuration\)](http://fr.wikipedia.org/wiki/Branche_(gestion_de_configuration)) . Ainsi http://fudaa.svn.sourceforge.net/viewvc/fudaa/branches/Fudaa_Mascaret_3_0/fudaa_devel/aide/ représente le projet fudaa_devel/aide dans la branche Fudaa_Mascaret_3_0.
- **Tag:** identifiant apporté à un ensemble de sources. Utiliser pour marquer des sources à l'origine d'une livraison d'une version d'un module/projet. Par la suite, un tag est utilisé pour créer une branche si un bugfix doit être apporté sur une version donnée.
- **Le tronc (ou trunk):** c'est la branche principale dans laquelle tous les derniers développements sont ajoutés. <http://fudaa.svn.sourceforge.net/viewvc/fudaa/trunk/>

Est-ce qu'il ne faudrait pas parler de projet pour module applicatif et garder le terme module pour les modules communs ?A modifier/compléter -Frédéric Deniger 15/05/09 14:09

Actuellement, on hésite souvent entre les termes anglais ou français. Il existe des normes comme les javaBeans (<http://fr.wikipedia.org/wiki/JavaBeans>) ou les design pattern qui nous obligent à utiliser des termes anglais. D'un autre coté, il est beaucoup plus simple d'utiliser les termes français lorsque nous manipulons des termes métiers (Maillage, Branche, digue). Il est proposé d'utiliser les termes anglais dès qu'il s'agit de terme informatique (package, abstract, ...) sauf si le terme français est couramment utilisé et compris par tous (j'ai pas d'exemple (Fichier source peut être). Pour les commentaires et les termes métier, le français serait favorisé. Ensuite, chaque projet peut contenir un dictionnaire de données; si un consensus entre les partenaires est trouvé, ce dernier peut être "monté" dans la partie commune et sera appliqué (je préfère conseillé) à toutes les applications utilisant des notions couvertes par ce dictionnaire de données.

Personnellement, je pense qu'un glossaire commun serait interessant pour tous les developpeurs qui souhaitent travailler avec Fudaa, comme base pour bien comprendre les données manipulées. Je ne sais par contre pas si ce glossaire doit être commun a tous les sources ou documentations ou commentaires. Chaque application peut être limité a un usage bien particulier pour une organisation bien spécifique qui utilise son propre jargon, qui peut légèrement différer d'une autre organisation.Mettre au point un jargon commun peut s'averer un travail fastidieux. -Marchand 18/05/09 15:32

Les 3 arborescences

Il y aura clairement 2 arborescences:

- les modules communs seront rassemblés dans **framework** . Ainsi les modules communs serait réunis dans <http://fudaa.svn.sourceforge.net/viewvc/fudaa/trunk/framework>.
- Les modules métiers appartiendront à la partie **business**.
- Les modules applicatifs serait réunis dans **soft**. Ainsi le projet Fudaa-Crue est accessible depuis `soft/fudaa-modeleur` soit <http://fudaa.svn.sourceforge.net/viewvc/fudaa/trunk/soft/fudaa-modeleur>.
- En général, les arborescences sont écrites en minuscule et sans espace.

Voici une vision de l'arborescence finale:

```
|--framework
| |--ctulu-fu - Le module contenant les classes fu
| |--ctulu-bu - Le module contenant les classes bu
| |--dodico-common
|--soft
| |--fudaa-crue
| |--etc...
```

Les noms des projets des parties framework et business reprennent l'architecture initiale de Fudaa avec les projets:

- `ctulu-*` qui constitue les librairies de bases
- `dodico-*` qui correspond à la couche métier. `dodico` dépend de `ctulu`
- `ebli-*` qui est la couche graphique 1d,2d et 3d. `ebli` ne dépend que de `ctulu`
- `fudaa-*` qui correspond à la couche applicative. `fudaa` s'appuie sur les 3 autre projets.

Le projet aide sera dispatché dans les modules applicatifs. A priori, il n'y a pas d'interdépendances entre les différentes aides des applications.

Objectifs du découpage

Ils sont effectivement multiples:

- créer un module pour chaque applications
- créer des modules homogènes
- isoler les packages qui ne sont plus utilisés (la couche CORBA, des packages d'`ebli`)
- isoler les packages qui demandent des dépendances spécifiques (`ebli 3d` avec une dépendance vers `java3d`,...)
- séparer les modules "serveur" des modules "client"

Client/Serveur:

par module "serveur", il est sous-entendu qu'il s'agit de modules qui pourraient être utilisés par des serveurs web ou d'applications (EJB). Ce sont donc des modules qui doivent être indépendants de la couche graphique. Par opposition, Les modules "client" sont les modules graphiques basés sur Swing.

L'outil Maven

Des liens:

- <http://maven.apache.org/guides/getting-started/index.html>

- <http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>
- <http://maven.apache.org/plugins/>
- <http://java.developpez.com/faq/maven/>

L'outil Maven est devenu un standard dans le développement logiciel. Il permet de gérer facilement les dépendances entre les modules, de générer facilement des rapports (javadoc, résultats de tests unitaires, résultat d'analyse statique,...) sur un projet.

Cet outil se base sur un repository distant dans lequel les livrables peut être déposés (uploadés) et être disponibles en téléchargement. L'outil Maven est entièrement accessible en ligne de commande.

Afin de le configurer, un fichier pom.xml est requis pour chaque module. Ce fichier décrit le projet : les développeurs, la version de java, les dépendances, les plugins maven à utiliser, les rapports à produire. On peut dire que l'équivalent du fichier ant build.xml. Ce descripteur peut être également utilisé pour configurer un ensemble de module: Maven dispose d'une gestion hiérarchique des pom.

Cet outil définit des conventions quant à l'organisation des répertoires.

Dans notre cas, un projet Maven est organisé de la façons suivante :

```

|-- src
| |-- main
| | |-- java > Contient le code source java
| | |-- resources > Contient les ressources à inclure dans le jar produit
| | (configuration, ...)
| | |-- config > Fichier de configurations extérieurs au jar. Contient en général les
| | fichiers de conf pour PMD, eclipse
| |-- test
| | |-- java > Contient le code source java des tests
| | |-- resources > Contient les ressources pour les tests
| |-- site > Le site de documentation: http://maven.apache.org/guides/mini/guide-site.html
|-- target > Répertoire où Maven génère ses fichiers/artefacts
|-- pom.xml > Descripteur du projet

```

Voir <http://java.developpez.com/faq/maven/?page=utilisation#utilisation1> pour la description complète avec les parties web et les filters.

Note: dans cette arborescence, toutes les ressources (images, fichier de traduction,...) seront réunies dans le répertoire `src/main/resources`. Dans la configuration actuelle de Fudaa, cette séparation n'est pas appliquée.

Organisation d'un module commun

Chaque module commun sera organisé sur le modèle maven.
La documentation (html, doc,...) se trouvera dans le le répertoire `site`.

Les fichiers de configuration des IDE et des autres outils

Le répertoire `src/main/config` contiendra les fichiers de configuration utilisés pour l'IDE, les analyseurs de code statiques (PMD,...) et des fichiers de configuration nécessaires pour la distribution (jnlp, Manifest,...). Il n'y a pas de règles pour l'organisation de ce dossier mais un nommage clair des fichiers devrait permettre au développeurs de connaître le rôle de chaque fichier. Par exemple, pour Fudaa-Crue, ce dossier contient un fichier `eclipse-format-import.xml` qui est le fichier de configuration pour eclipse pour

formater/organiser les imports d'une classe. Est-ce qu'il faut utiliser des répertoire par IDE, outils ? -Frédéric Deniger 15/05/09 14:08

Les classes d'exemples

Les classes d'exemples (qui peuvent être des supports aux didacticiels) se trouveront dans le répertoire de `test` et dans un package `example`. Ainsi pour `ebli3d`, le répertoire `/src/test/java/org/fudaa/ebli/3d/example` contiendra des exemples d'utilisation du projet `ebli3d`.

Les classes de tests

L'outil de test utilisé est [JUnit](#) version 4.8. Cette version s'appuie sur les annotations. Actuellement les tests unitaires de Fudaa sont basés sur JUnit 3.8 (sans annotations). question ouverte. Je pense que si on fait l'effort de restructurer Fudaa autant prendre les dernières versions des outils -Frédéric Deniger 15/05/09 14:07

Les classes de tests sont suffixées par `Test`. Elles appartiennent au même package que la classe testée. Cela a pour avantage de clarifier l'organisation et de permettre aux classes de test d'accéder aux champs/méthodes qui ne sont visibles qu'au niveau package (visibilité `protected` ou par défaut).

Les tests graphiques

Des outils de tests automatisés des interfaces graphiques seront à intégrer dans Fudaa. Pour l'instant 2 outils ont été testés (hors projet Fudaa) par les développeurs:

- FEST-Swing: <http://easytesting.org/swing/wiki/pmwiki.php>
- Abbot: <http://abbot.sourceforge.net/doc/overview.shtml> qui serait intéressant notamment grâce à `costello`

Point en attente... -Frédéric Deniger 15/05/09 14:07

Organisation d'un module applicatif

L'arborescence des modules applicatifs doit contenir un niveau supplémentaire. En effet, il faut séparer la partie métier de la partie cliente dans les applications pour des raisons évidentes de design et pour permettre à la partie métier de ces applications d'être facilement utilisable dans une application web ou dans un serveur d'application.

Un module applicatif sera constitué de 3 sous-modules:

1. `aide`: contient l'aide applicatif. Est-ce que cette partie doit toujours être considérée comme faisant partie du développement ? Souvent ce ne sont pas les mêmes intervenants/entités qui interviennent sur ce point ? -Frédéric Deniger 15/05/09 14:07
2. `métier`: les classes `dodico`
3. `client`: les classes `fudaa`

Note sur le vocabulaire serveur/dodico/métier et fudaa/client

Dans l'arborescence actuelle, les applications sont découpées en 2 parties:

1. la partie `dodico` qui contient toutes les classes métier et les classes CORBA.
2. la partie `fudaa` qui dépend de `dodico` dans laquelle l'application graphique est construite.

Dans la nouvelle structure la partie cliente (fudaa) a toujours la bonne dénomination. Par contre, la partie dodico, il faudrait plus parler de couche métier que de couche serveur. La partie CORBA n'est désormais plus utilisée et l'aspect serveur sera désormais apporté par un sous-projet supplémentaire. Par exemple, dans les projets EJB (serveur d'application), le module applicatif devra contenir un nouveau sous-module ejb. Ce sera la même chose pour la partie web.

Arborescence d'un module applicatif

```
fudaa-crue
|-- pom.xml > Descripteur du projet fudaa-crue
|-- src
| | |-- resources > Contient les ressources à inclure dans le jar produit
| | (configuration, ...)
| | |-- config > Fichier de configurations extérieurs au jar. Contient en général les
| | fichiers de conf pour PMD, eclipse
| |-- site > le site du projet fudaa-crue. Il contiendra les sites des sous-modules de
| fudaa-crue
|-- dodico (ou metier): contient le sous-module métier de fudaa-crue
|-- fudaa (ou client): contient le sous-module client/swing de fudaa-crue
```

Le nommage des sous-modules dodico/fudaa reste à discuter (but de ce document). Devons-nous garder les anciens noms pour conserver les dénominations du projet Fudaa initial. Est-ce que le nommage dodico/fudaa ne doit pas être réservé aux modules communs et il faut utiliser metier/client pour les modules applicatifs ? -Frédéric Deniger 15/05/09 14:06

La réponse à cette question permettra également de spécifier les noms de packages. Par exemple pour le sous-module métier de fudaa-crue, quel est le nom de package à utiliser:

```
org.fudaa.dodico.crue ou org.fudaa.metier.crue ou org.fudaa.crue.metier.A
discuter... personnellement la 3eme solution me semble la mieux mais la
discussion est ouverte -Frédéric Deniger 15/05/09 14:06
```

Je m'interroge : si les sous modules dodico/fudaa contiennent les sources des classes metier et client d'une application, que contient src ? Est ce une nécessité d'avoir des sous modules a un module applicatif (je parle de sous modules qui ne soit pas communs ?) Peut être que c'est nécessaire pour certaines appli, et pas pour d'autres. -**Marchand** 18/05/09 15:45

Le dossier `src` contient les fichiers généraux du projet (fichiers de configuration des IDE, de déploiement) et le répertoire `site` qui contient les sources du site web du projet. Il n'est pas nécessaire d'avoir des sous-modules mais ils permettent de bien structurer le projet. Ils permettront de migrer facilement vers des architectures distribuées (client-serveur). Il n'est pas exclue que certaines applications très simples ne suivent pas cette organisation et ne comportent pas de sous-modules.

Note importante: le renommage des packages ne sera fait que pour les nouvelles applications. Il faudra faire attention aux applications comme Fudaa-Mascaret qui utilise la réflexion.

Proposition de découpage

Ctulu

Tous les projet issus de ctulu se trouveront dans le répertoire `framework`.

- `fu`: classes utilitaires non graphiques issues de `com.memoire.bu` (Guillaume Desnoix). Ce module sera nommé **ctulu-fu**, information donnée dans le descripteur de projet soit le fichier `pom.xml`
- `bu`: classes graphiques de de `com.memoire.bu` (Guillaume Desnoix). Ce module sera nommé **ctulu-bu**
- `common`: les classes common de ctulu. **ctulu-common**
- `sig`: les classes SIG: **ctulu-sig**
- `video`: les classes video **ctulu-video**
- `gui`: les classes swing de ctulu **ctulu-ui**

Ce découpage est celui qui sera effectif pour la phase 1. Pour la phase 3 ou 4, on pourra se poser des questions sur les packages swing qui ne devraient être intégrés qu'à partir de la couche `ebli` (partie graphique).

Dodico

L'arborescence de dodico sera basée sur la même logique que celle de ctulu.

Pour dodico, il y aura un projet `dodico-corba` qui contiendra l'ensemble des fichiers IDL et des classes nécessaires aux applications basée sur CORBA. Ce projet fera partie de **business**.

Les modules commun de dodico

- **common** qui contiendra les packages commun, fichier et fortran: intégré à **framework**
- **ef**: les classes gérant les éléments finis. Les packages `olb` et `trigrd` sont fortement liés à ce module-> **fudaa-ef**
- **trigrd.io, serafin.io,...**: les packages qui permettent de lire des fichiers de maillage sont intégrés au module **fudaa-ef**
- **dico**: gestion des projets EDF basés sur des fichiers dictionnaire (télémac, mascaret, edamox,...). Sera intégré à **framework** car pas de dépendance métier.
- **geodesie**: package deprecated: **supprimé**.
- **h1d**: partie commune d'hydraulique1d. Intégré au projet **fudaa-mascaret (soft)**.
- **h2d**: partie commune d'hydraulique2d. Intégré dans la partie business car utilisé par le modeleur et prepro.

Les autres classes "communes" (mesure, mathématiques) sont basées sur CORBA et elles seront incluses dans le projet `dodico-corba` dans un premier temps. Ensuite, elles seront sorties de ce package si besoin.

Les modules applicatif

Les autres packages de dodico seront réunis dans leur module applicatif respectif.

Ebli

Le découpage d'ebli serait le suivant:

- ebli-common qui contient les ressources communes
- ebli-3d
- ebli-2d
- ebli-1d
- ebli-graphe: package de dessin 1d deprecated. A voir car il me semble que ce package à encore son utilité pour certains cas. -Frédéric Deniger 15/05/09 14:14

Fudaa

Les modules communs de Fudaa seraient les suivi:

- mesh: gestion des maillages -> laissé dans le projet **fudaa-prepro**
- fdico: interface graphique pour la gestion des fichiers dico -> **fudaa-dico** de la partie business
- sig: GUI pour les modules SIG -> **fudaa-sig** de la partie business
- common: classes communes de fudaa
- autre ?

Gestion des versions

Pour tous les modules, les versions seront basés sur 3 nombres
<version_majeure>.<version_mineure>.<bugfix ou buildnumber>.

En général, la version majeure est incrémentée si un nombre important de fonctionnalités est apporté à la version en cours ou si l'API est remaniée. La version mineure est incrémentée si l'API n'a pas changé et si les améliorations sont peu nombreuses. Ces incréments pourront être décidés suite à un consensus commun.

Le dernier nombre ne doit être utilisé que pour les bugfix. Dans certains projets, ce derniers nombre correspond plus à un buildnumber qui est automatiquement incrémenté dès qu'un version est diffusée (hors version SNAPSOT).

Ce qui est proposé pour les incréments:

- la version majeure est incrémentée si l'API n'est pas compatible avec les version précédentes (suppression de classes, de méthodes, ce qui permettrait de faire le ménage de temps en temps)
- la version mineure est incrémentée si l'API est compatible (avec notion de deprecated, API augmentée).

Gestion des inter-dépendances

Le versionning des modules applicatifs sera relativement simple car aucun autre module ne dépendrait de ce dernier. Par contre, pour le versionning des modules commun, il faudra apporter une attention particulière à l'homogénéité des dépendances transitives. Par exemple, si le module ebli-2d-1.0.0 dépend du module ctulu-sig-1.0.0, il faudra s'assurer que le module ebli-3d-1.0.0 ne dépende pas de ctulu-sig-2.0.0. Dans ce cas, une application qui aura besoin de ebli-2d et ebli-3d aura 2 dépendances incompatible vers ctulu-sig. Il faut s'assurer que tous les modules de la même couche dépendent d'un ensemble cohérent de modules de la couche inférieure. Si une dépendance vers un sous-module est modifiée dans une couche, cela demandera au développeur de faire la même modification pour tous les modules de la même couche. Cela demandera un effort de

qualification supplémentaire mais c'est une étape obligatoire.

Il n'y a pas un risque que quelqu'un ne connaissant pas un module frère d'une même couche soit dans l'impossibilité de faire évoluer ce module frère de manière satisfaisante ? -Marchand 18/05/09 15:59

A priori non, car l'architecture proposée (avec un pom parent qui est commun à tous les modules d'une même couche voir ci-dessous), le développeur est obligé de redéployer tous les modules d'une couche s'il modifie l'un d'eux. C'est le cote contraignant mais c'est pour l'instant le seul moyen (trouvé) pour assurer la cohérence d'une couche et pour éviter d'oublier des modules "frère". -Frédéric Deniger 20/05/09 23:27

La gestion des inter-dépendances entre les modules commun est donc la partie délicate. Une solution (partielle ?) à ce problème réside dans l'utilisation d'un fichier de configuration commun (on dit pom parent dans le langage Maven) au niveau de chaque couche. Par exemple:

```
|--common
| |--ctulu
| | |--pom.xml - descripteur de la couche ctulu dans lequel toutes les
dépendances et leur version sont déclarées.
| | |--fu - Le module contenant les classes fu
| | |--bu - Le module contenant les classes bu
| | |--etc...
```

Ce fichier pom.xml regrouperait toutes les déclarations des dépendances vers d'autres librairies/modules. Ainsi, il assure que toute la couche ctulu est homogène en terme de dépendances. Il faut bien prendre en compte le fait que si un développeur modifie une dépendance, il doit vérifier que tous les autres modules impactés fonctionnent avec cette nouvelle dépendance.

Il faudra vérifier que cette solution (à priori non) n'oblige pas un développeur d'un module commun à récupérer toutes les sources des modules de la couche en question. De la même manière, il serait avantageux d'utiliser un fichier pom.xml au niveau commun qui assurera que toute la partie common forme un ensemble homogène.

L'outil Maven permettra tout de même au développeur d'application de choisir précisément ses dépendances en excluant certaines dépendances transitives. Il est également possible d'être plus permissif dans la déclaration des dépendances en autorisant une dépendances vers un ensemble de version (par exemple, un projet peut dire qu'il a besoin de ebli-3d-1.0.*) plutôt que vers une version fixe.

Note: attention, il se pourrait que des modules applicatifs dépendent les uns des autres. Par exemple, Fudaa-prepro dépendra des modules applicatifs de Reflux,Rubar et Telemac. Faut-il interdire ces interdépendances entre module applicatif en regroupant des module (les modules reflux, rubar et telemac seraient regroupés dans un seul module applicatif) ou considérer ces modules applicatifs mais "communs" comme de vrais modules communs ? -Frédéric Deniger 15/05/09 14:08

La notion de SNAPSHOT

Les paragraphes précédents se focalisent sur la livraison de version finale utilisable en production. Par contre, il faut aussi prendre en compte le livraison de version de développement. Par exemple, une équipe projet peut intervenir en même temps sur un

module commun et sur un module applicatif. Au cours du développement, le responsable du module commun doit pouvoir diffuser une version de développement de son module commun afin que ces partenaires puisse le tester dans l'application finale. En général, on parle de version Alpha, Beta ou Release Candidate. Avec Maven, il existe la notion de SNAPHSOT: <http://java.developpez.com/faq/maven/?page=terminologie#documentation5>

Ainsi les versions qui sont destinées à être diffusée en test uniquement doivent être suffixée par -SNAPHSOT.A compléter/commenter.... -Frédéric Deniger 15/05/09 14:08

Usage du gestionnaire de sources: tags/branches

Tous les développements se font sur la branche principale (le tronc). Un tag est posé sur un ensemble de sources dès qu'un livrable est généré. Si un bugfix est nécessaire, une branche est créée à partir du tag. Le bugfix est ensuite reporté sur la branche principale.

Pour simplifier la gestion des branches et la maintenance, il est nécessaire de ne corriger que la dernière version livrée. Si un projet utilise la version n-2, il faudra essayer de le faire migrer vers la dernière version diffusée afin de limiter les branches à maintenir. Il sera toutefois possible de corriger une version n-2 mais cette opération doit être exceptionnelle.

Note: avec SVN, les notions de tags et de branches sont quasiment équivalentes: un tag est géré par SVN comme une branche. Il faudra être strict sur la gestion des branches et n'autoriser la modification d'une branche que si c'est un bugfix.

Convention de nommage/Formatter les sources

Convention de nommage

Actuellement, les conventions de nommage de fudaa sont regroupées dans le document http://fudaa.fr/devel/index.php?option=com_content&task=view&id=36&Itemid=33 . Il était demandé d'utiliser le caractère _ en préfixe ou suffixe pour les paramètres de méthodes et les attributs de classes. Cette convention permettait de différencier facilement les portées des variables. Or, avec les nouveaux IDE, cette notation devient inutile et elle apporte du "bruit" inutile.

Il est proposé de partir des [conventions de SUN](#) et les compléter éventuellement pour coller au mieux aux habitudes de chacun. Par la suite, les outils d'analyse du code (PMD, Checkstyle) seront utilisés afin de vérifier que le code suit bien les règles mises en place pour tous les projets de la plate-forme Fudaa.

Formateur de source

Les IDE proposent des formateurs internes. Il est tout à fait possible de partager la configuration de ces formateurs entre les développeurs utilisant le même outil. Par contre, le partage de cette configuration entre différent IDE semble difficile. Une solution serait d'utiliser un formateur externe mais l'offre semble être minime et non maintenue: par exemple, [Jalopy](#) ne semble plus être maintenu.

Ce serait de la responsabilité de chaque développeur de configurer correctement ses outils afin de suivre les conventions établies. Si possible, ce dernier peut partager sa configuration (grâce à un export) en la plaçant dans le répertoire config du module.

Lors de la restructuration, un répertoire common/src/config sera rajouté. Ce dernier contiendra tous les fichiers de configuration partageables.

Nouveaux outils

Gestion des bugs/exigences

2 solutions sont envisagées:

- Trac: <http://trac.edgewall.org/>
- Jira: <http://www.atlassian.com/software/jira/>

Ces 2 solutions semblent être équivalentes avec Trac qui serait plus simple à installer. Par contre, Jira semble être plus complet. Cet outil est gratuit pour les projets open source.

Choix ouvert. Ne pas hésiter à aller faire un tour sur les sites. -Frédéric Deniger 15/05/09 22:46

En raison d'une impossibilité de coupler Trac avec un serveur Subversion distant, le gestionnaire retenu est Jira.

Moteur d'intégration continue

L'outil le plus simple et qui donne satisfaction:

- Hudson: <https://hudson.dev.java.net/>

Suivi de la qualité du code

- Sonar: <http://www.sonarsource.com/>

Nouvelles librairies

Gestion des logs

Dans l'état actuel, les logs sont gérés par FuLog. Cette classe a l'avantage d'être légère mais elle dispose de limitations multiples: configuration du formatage impossible, pas de filtre des logs aux niveaux des classes, impossible d'ajouter des paramètres supplémentaires. Ce sujet est largement couvert par des logiciels libres:

- [log4j](#) : très répandue, contient de nombreux appenders (sortie console, sortie fichier, sortie fichier html) et formateurs. Licence Apache
- [logback](#) : la suite de log4j ? Licence Apache
- [java.util.logging](#) : api livrée avec le JDK mais elle est assez limitée en terme de formatage et d'appenders

Il serait également plus sûr d'utiliser un outil comme <http://www.slf4j.org/> qui est une surcouche à ces outils de logging et permet de s'affranchir de l'implémentation. Ainsi, à partir du même code, il serait tout à fait envisageable de changer de logger.

Outils Apache

A l'origine, il y avait des problème de licence mais ces derniers sont résolus avec la v3 de la licence GPL: <http://www.apache.org/licenses/GPL-compatibility.html>

Si fudaa utilise la version 3 de la GPL (ce qui semble peu contraignant), il sera possible d'utiliser les librairies Apache qui seraient très utiles et notamment le projet commons: <http://commons.apache.org/>

Nouveau framework swing

Fudaa se base su Bu. Or, cette surcouche à Swing date de 1998 et d'autres solutions sont désormais plus riches:

- [Spring-rcp](#) : framework basé sur spring
- [Netbeans](#)
- [Jnode](#)
- <https://appframework.dev.java.net/>

Format d'échange des documents

Doit-on statuer sur le format d'échange des document : openOffice, Zoho ou un autre. Il est vrai que Google Doc a le grand avantage de permettre l'édition en ligne et gère les versions d'un document.

Faut-il un Wiki ? Le site Mambo est suffisant ?

L'avantage du wiki, c'est sa souplesse d'édition et d'organisation. Contrairement à Mambo, il dispose souvent (au moins Dokuwiki, Trac) d'un historique des modifs. Et je ne suis pas sur qu'on puisse facilement faire un lien vers un autre article dans Mambo ?? Mais bon, mambo permet au moins l'édition, et c'est vrai que si les catégories suffisent, alors pourquoi pas ? A noter que Trac intègre aussi en standard un wiki. Est ce que google doc offre les mêmes possibilités d'organisation, de liens, etc ? -Marchand 5/25/09 2:26 PM