Équipe FACT
Conception et développement d'applications

http://fact-team.github.io

Rédigé avec LaTeX
Version du 12 février 2015

Plan d'Assurance Qualité Logicielle

FactDev

Université Toulouse III – Paul Sabatier

— Florent Berbie
— Antoine de Roquemaurel
— Cédric Rohaut
— Andriamihary Manantsoa Razanajatovo

12 février 2015

Table des matières

Avant-Propos

Dans le cadre d'un projet, le Plan d'assurance Qualité Logicielle détermine les mesures nécessaires permettant de répondre aux exigences de qualité et de gestion inhérentes au projet. Ainsi, le Plan d'Assurance Qualité Logicielle définit les droits,les devoirs et les responsabilités de chaque partie prenante afin d'assurer le respect de ces exigences.

Il constitue un outil de travail et un référentiel commun à tous les acteurs pour leur donner une vision collective du projet.
Il est également le cahier des charges de la qualité et est réalisé en collaboration avec le client puis approuvé par celui-ci.
Enfin, il définit les procédures à suivre, les outils à utiliser, les normes à respecter, la méthodologie de développement du produit et les contrôles prévus pour chaque activité.

Le Plan d'Assurance qualité constitue un contrat entre le titulaire, le client et toutes les autres parties prenantes. Ce contrat prend effet dès son acceptation par le client et les personnes concernées et peut être, si les circonstances l'exigent, amené à être modifié au cours du projet. Dans ce cas, toute évolution future sera soumise à l'acceptation du client. En effet, au terme du projet, le Plan d'Assurance Qualité Logicielle constituera l'un des documents de résultat du projet.

1 Description du Projet

1.1 Objet du projet

Le logiciel a pour but de faciliter la création de devis et la conversion de ces devis en factures.

Ainsi, il sera possible d'enregistrer des clients dans une base de données et de leur associer des projets comportant un ou plusieurs devis ou factures.

1.2 Présentation des Parties Prenantes

1.2.1 Client : Antoine de Roquemaurel

Développeur Freelance et membre de l'équipe de développement.

☎ 06 54 33 52 93
https://antoinederoquemaurel.github.io
✉ antoine.roquemaurel@gmail.com

1.2.2 Encadrant : Fédéric Migeon

Maître de conférence à l'Université Toulouse III – Paul Sabatier

☎ 05 61 55 (62 46)
✉ Frederic.Migeon@irit.fr
IRIT1 / Niveau 3, Pièce : 361

1.2.3 Responsable de l'UE Projet : Bernard Cherbonneau

☎ 05 61 55 (63 52)
✉ Bernard.Cherbonneau@irit.fr
IRIT1 / Niveau 4, Pièce : 413

1.2.4 Titulaire : Équipe FACT

Étudiants en M1 Informatique Développement Logiciel à l'Université Toulouse III – Paul Sabatier

Florent Berbie
☎ 06 85 31 92 90
✉ florent.berbie@gmail.com

Antoine de Roquemaurel
☎ 06 54 33 52 93
✉ antoine.roquemaurel@gmail.com

Cédric Rohaut
☎ 06 74 80 12 67
✉ rohaut@icloud.com

Manantsoa Andriamihary Razanajatovo
☎ 06 01 71 53 02
✉ manantsoa.razana@gmail.com

2 Production

2.1 Méthode de développement

2.1.1 Définition et pertinence de la méthode scrum

Le développement du projet se fera selon la méthode Agile *Scrum*, comme convenu avec notre encadrant.

Cette méthode, basée sur les stratégies itératives et incrémentales, permet de produire à la fin de chaque *Sprint* (incrément/itération) une version stable et testable du logiciel. Les différents événements associés à *Scrum* accroissent la communication grâce à des réunions quotidiennes appelées *mêlées*. Ces réunions permettent une cohésion, une coopération et une homogénéité du travail fourni par les membres de l'équipe. De plus, la présence d'artefacts c'est-à-dire d'éléments à réaliser avec des niveaux de priorité contribue à la productivité du développement.

Le client Antoine de ROQUEMAUREL a défini le sujet et un ensemble de fonctionnalités pouvant être intégrées au logiciel. Les fonctionnalités majeures définissent les versions livrables (*Release*) tandis que les *Sprints*, propres à chaque *Release*, précisent les sous-fonctions à implémenter pour parvenir au résultat escompté.
La méthode *Scrum* est sujette à quelques flexibilités tel que l'ajout ou la modification de fonctionnalités durant la phase de développement afin de répondre au mieux aux besoins du client.

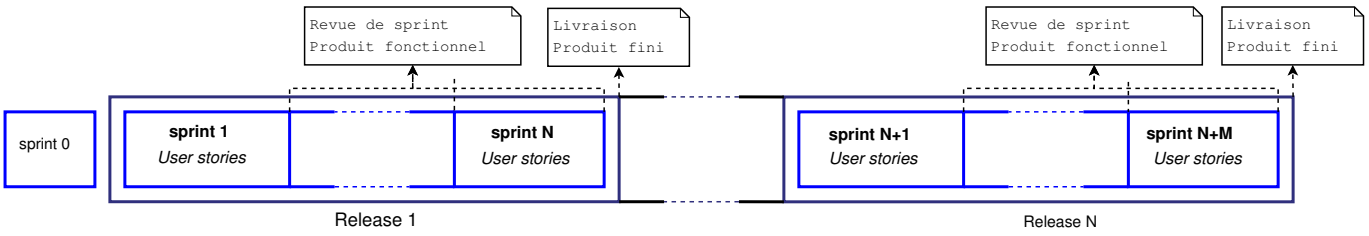

FIGURE 2.1 – Fonctionnement des *Sprints* et *Releases* de la méthode *Scrum*

2.1.2 Application de la méthode Scrum et critères de qualité

Dans le cadre de notre projet, la durée d'un *Sprint* a été défini à deux semaines. Un *Sprint* est constitué de *User Stories* et de *Technicals Stories* qui sont préalablement définies lors de réunions quotidiennes appelées *mêlées*. Ces *mêlées* sont quotidiennes et organisées durant notre temps libre. En dehors de ces réunions, nous avons mis en place un salon de discussion IRC.

Les *Technical Stories* indiquent brièvement la fonction que l'on doit réaliser. Les *User Stories* indique les fonctionnalités du logiciel sous cette forme :

> **En tant que** `[Personne(s) utilisant le logiciel]`
> **Je souhaite** `[La fonctionnalité que je désire avoir]`
> **Afin de** `[Objectif de la fonctionnalité]`

Chaque membre de l'équipe détermine un poids pour chaque *Story* c'est-à-dire une valeur indiquant sa complexité et/ou le temps nécessaire afin de la mettre en œuvre. Le poids de chaque *Story* est déterminé durant les *mêlées* au moyen d'un *Planning poker*. Ainsi, chaque membre justifie le poids qu'il a déterminé et après un éventuel débat et en commun accord, une valeur est attribuée à la *Story*. Elle possède également des niveaux de priorité précisant l'importance d'intégrer cette *Story* dans le *Sprint* :

— *Must* : La *Story* doit obligatoirement être réalisée lors du *Sprint*
— *Should* : La *Story* devra être réalisée (dans la mesure du possible)
— *Could* : La *Story* pourra être réalisée car elle n'a aucun impact sur les autres tâches
— *Would* : La *Story* ne sera pas nécessairement faite et sera alors reportée au prochain *Sprint*

Une *Story* est considérée comme terminée lorsqu'elle est fonctionnelle d'un point de vue utilisateur c'est-à-dire :
— lorsque les tests unitaires (pour la base de données ou pour les modèles) sont validés
— lorsque les tests d'intégrations sont validés
— Lorsque chaque méthode est documentée

À la fin du *Sprint*, on présente notre logiciel à notre client, Monsieur Frédéric Migeon afin qu'il le valide. Un *Sprint* fournit toujours :
— une démonstration des nouvelles fonctionnalités logicielles
— des tests unitaires
— une documentation du code (au format HTML et PDF)
— un manuel d'utilisateur à jour des nouvelles fonctionnalités

Notre projet sera réalisé en deux *Releases* c'est-à-dire qu'il y aura deux versions livrables. Chacune de ces versions est composée de trois *Sprints* ayant chacun un poids moyen de 50 pour environ 10 *Stories*.

Les versions livrables du logiciel sont définies dans un carnet des produits (*Product Backlog*) qui référence l'ensemble des *Stories* des différents *Sprints*. Ce carnet est susceptible d'évoluer en fonction des besoins du client, de nouveaux choix de conception ou encore d'éléments non prévus.

2.1.3 Organisation et rôles dans l'équipe de développement

La méthode *Scrum* possède des rôles qui lui sont propres : le *Scrum Master*, le *Product Owner* et l'équipe de développement.

2.1.3.1 Scrum Master : Florent Berbie

Le *Scrum Master* aura pour mission principale de guider les développeurs dans l'application de la méthode *Scrum*. Il veillera à ce que la méthode soit comprise de tous et appliquée de façon correcte. Il aura le rôle de meneur lors de phases importantes d'application de la méthode telles

que le *Planning Poker* ou encore les *mêlées* quotidiennes.

2.1.3.2 Product owner : Antoine de Roquemaurel

Le *Product owner* est la seule personne responsable du carnet de produit et de sa gestion. Ce carnet comprend l'expression de tous les items associés à une priorité (l'importance pour le client). La compréhension de ceux-ci ainsi que la vérification du travail fourni est sous la responsabilité du *Product owner*.

2.1.3.3 Équipe de développement : Florent Berbie, Antoine de Roquemaurel, Cédric Rohaut, Andriamihary Razanajatovo

Afin d'assurer une certaine cohésion entre les membres de l'équipe, un même niveau d'implication et un travail de qualité nous avons définis des rôles spécifiques pour chaque membre :

2.1.3.4 Directeur qualité : Cédric Rohaut

Le responsable qualité sera garant de la transposition des exigences du client sous forme de solutions techniques au sein du logiciel final. De plus il devra vérifier et valider la qualité du logiciel au travers de tests, du respect des conventions et de l'aspect général du logiciel.

2.1.3.5 Directeur documentation : Andriamihary Razanajatovo

Le responsable documentation sera chargé de la révision de l'ensemble des documents avant leur remise ou leur soumission aux parties prenantes concernées. Il veillera à la qualité des aspects fondamentaux ()le fond et la forme) des documentations à fournir.

2.1.3.6 Directeur technique Qt, C++ : Antoine de Roquemaurel

Le responsable technique Qt, C++ sert de support à l'équipe en cas de problèmes techniques liés au développement sur la plate-forme Qt. De par ses connaissances acquises dans ce domaine, il sera le plus apte à aider les membres de l'équipe projet ayant des difficultés avec cette technologie. De plus, dans le but de minimiser les couplages lors de la conception, il veillera à ce que le patron de conception MVC [1] soit correctement utilisé. Ainsi, le code sera clairement découpé en trois parties : modèle, vue et contrôleur ce qui permettra de s'assurer des contrôles d'interactions entre nos composants logiciels.

1. Modele, Vue, Contrôleur

2.1.3.7 Autres rôles, responsabilités de chaque développeur et organisation générale

Dans un souci d'assurance qualité, chaque responsable est en mesure de présenter les difficultés rencontrées, les différentes solutions possibles et de justifier le choix de la technique adoptée.

Lors de la réalisation d'un cas d'utilisation (*issue* sur *Github*), celui-ci est assigné à un développeur. Une fois que ce dernier considère sa tâche comme finie, il indique (via une *Pull Request*) aux autres membres de l'équipe que la tâche est soumise à la validation et à l'intégration. Un des autres membres vérifie que la fonction est conforme à sa description, que le code est facilement compréhensible et correctement commenté. Dans le cas ou la revue de code ne donne pleine satisfaction, le membre effectuant cette revue et les autres membres de l'équipe pourront ajouter des commentaires pour débattre de la fonctionnalité, de l'implémentation de la fonction ou des technologies à employer. Chacun pourra alors soumettre sa vision du problème et la manière avec laquelle il aurait résolu le problème.

2.2 Technologies utilisées

2.2.1 Architecture Logicielle

L'objectif du logiciel est d'éditer des devis et factures et de proposer une gestion des clients : le client possède un ou plusieurs projets auxquels sont associés un ou plusieurs devis et/ou factures. Ainsi, l'architecture logicielle comporte une base de données de type SQLite pour la gestion des clients, de ses projets, des devis et factures associés. Nous utilisons un patron de conception MVC, avec un modèle qui correspond aux objets métiers Client, Projet, Factures, Devis, Prestations. Ces objets sont instanciés par les classes associées à la base de données qui réalisent ces tâches. Enfin, nous générerons les devis et factures en LaTeX ou en PDF.

2.2.2 Environnement de développement

Le logiciel est développé à partir du *framework*[2] Qt(version 5) en langage de programmation C++ (version 11) avec l'EDI[3] Qt Creator (version 3.3).

Les membres de l'équipe développent sur différents systèmes : MAC OS (OS X 10.10), Linux (Fedora 20 et Linux Mint 17.1). Bien que développé uniquement sur des systèmes UNIX, l'application sera compatible avec Windows.

Afin de mener au mieux ce projet, l'équipe utilise l'outil *Github*, un service web d'hébergement et de gestion de développement de logiciels.

2. Ensemble de composants logiciels structurels servant à créer les fondations d'une future application. Un framework est créé par des développeurs pour d'autres développeurs. Un framework peut être vu comme une boîte à outils.

3. Environnement de Développement Logiciel

2.2.3 Outils pour la Gestion de Projet Agile

La développement de notre logiciel s'appuie sur *Github* qui permet la gestion des différentes versions de notre logiciel. Cet outil est composé d'une branche directrice (appelée *Master*), sur laquelle se greffe de nouvelles branches : une branche par *Sprint*. L'ensemble des *User Stories* que nous avons défini sont représentées dans *Github* par des *Issues*. Chaque *Issue* fait l'objet d'une nouvelle branche provenant de celle du Sprint associé. Cette gestion permet à chaque membre de développer indépendamment des autres sans entrer en conflit avec le reste l'équipe. Une fois l'issue terminée et validée par la revue de code, l'on fusionne la branche de l'issue à celle du Sprint associée afin que l'équipe bénéficie de la nouvelle version stable et fonctionnelle. On procède de la même façon en fin de *Sprint* en intégrant la branche de *Sprint* à la branche principale *Master*.

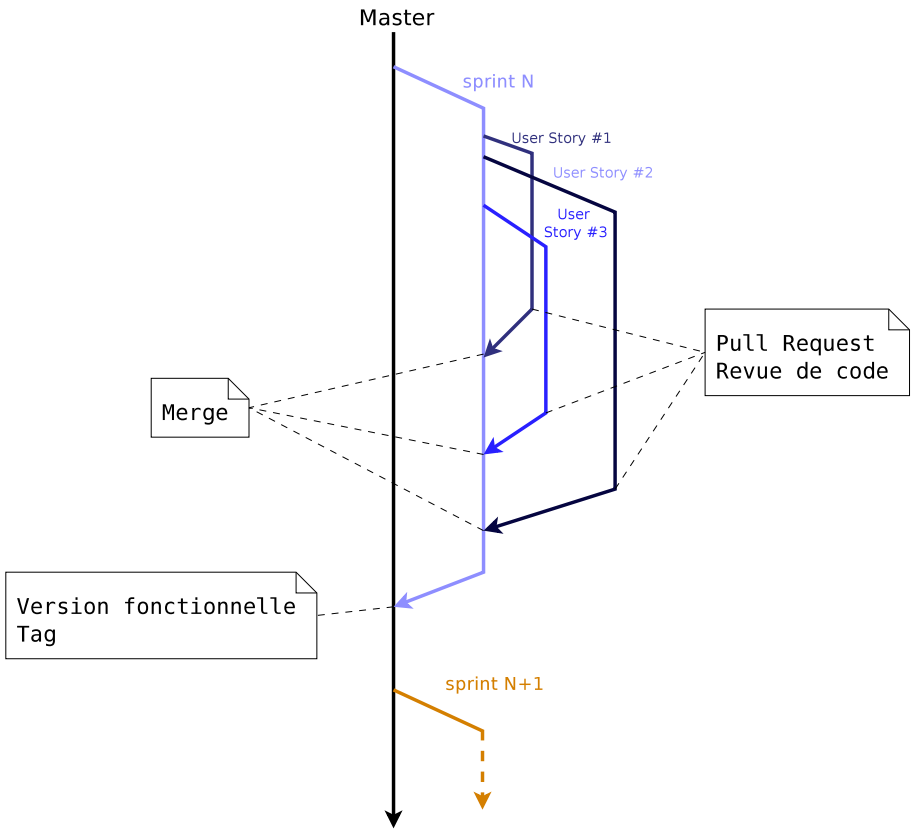

FIGURE 2.2 – Principe du « Git Branching Workflow »

Outre la gestion des différentes versions de notre code, *Github* permet un suivi de notre projet. Le site internet *Github* dédié au projet est accessible par l'équipe et notre client Monsieur Frédéric MIGEON. De plus, *Github* contient un Wiki avec les règles de bonne conduite et les conventions à respecter pour assurer l'homogénéité de notre projet.

2.2.4 Documentation

La centralisation des documents se fait au moyen du Wiki de *Github* et de Google Drive. Le Wiki de *Github* contient :

- un tutoriel sur les méthodes de travail (Git, Scrum, C++)
- une convention de code à respecter (conventions de nommage des variables, des méthodes, d'organisation du code...)

— un tutoriel sur la documentation (*Doxygen*).

Sur Google Drive, on trouve l'ensemble des documents à rédiger en équipe. Cet outil permet notamment d'éditer un même document à plusieurs et d'être facilement accessible. Parmi les documents accessibles sur le Drive, on trouve :

— le manuel de l'utilisateur
— le plan d'assurance qualité
— le Carnet des produits *Product Backlog*
— les graphiques d'avancement :
  — Burndown chart
  — Burnup chart
— les Comptes rendus mensuels
— d'autres documents concernant le projet mais n'étant parmi ceux à rendre

2.2.5 Assurance de la qualité du code

Pour assurer une meilleure homogénéité, lisibilité et maintenabilité du code, nous utilisons l'outil *SonarQube*, un logiciel permettant de mesurer la qualité du code source en continu. Celui-ci analyse et fournit diverses informations sur le code tel que :

— le pourcentage de code non documenté (ou pas assez documenté)
— la complexité générale du logiciel ou, au cas pas cas, des méthodes de notre programme
— le bon respect des conventions de codages (nommage des attributs/méthodes, indentation, ...)
— la couverture de code (via les tests unitaires)
— la duplication de code

3 Fournitures et Livrables

Durant le projet, l'équipe fournira plusieurs livrables pour l'ensemble des parties prenantes et des enseignants de l'UE Projet :

- Plan d'assurance Qualité Logicielle (signé par le client et déposé sur Moodle)
- Compte-rendu mensuel d'activité (envoyé par courriel au responsable de l'UE)
- Bilan de projet
- Graphiques d'avancement
  - Burndown chart
  - Burnup chart
- Manuel de l'utilisateur
- Carnet des produits *Product Backlog*
- Revues de sprint
- Builds[1]

Le résultat du projet sera présenté par le Titulaire au Client lors des versions livrables. La première version livrable du logiciel se fera le 26 février 2015 et la seconde le 24 avril 2014. La validité du logiciel se fera à l'appréciation de notre client, Monsieur Migeon.

L'organisation, le déroulement du projet et les résultats obtenus feront l'objet d'un oral lors de la soutenance.

1. Un *build* est un *artefact* logiciel autonome résultant de conversion de fichiers de code source en code exécutable

4 Planning

Conformément aux dates qui nous sont imposées, le projet a débuté le 27 décembre 2014 et prendra fin à la soutenance d'avril.

La première phase du projet consiste à choisir les langages et l'environnement de développement. Une fois défini, on réalise la conception globale de l'application. De plus, afin que les membres de l'équipe ait une vision précise et commune du logiciel, des maquettes de l'interface sont réalisées.

Conformément à la méthode *Scrum*, des réunions hebdomadaires avec le client ont été programmées. L'équipe de développement organisera également des *mêlées* quotidiennes afin de faire un point sur l'avancement du *Sprint* en cours.

Le Titulaire prendra compte les remarques et demandes de notre encadrant Monsieur Migeon lors des réunions. Celles-ci feront l'objet d'une ou plusieurs *issue* du *Sprint* suivant.

Le projet comportera deux *Releases* avec, pour chacune d'elle, trois *Sprints* de deux semaines comme le montre le diagramme de GANTT ci-dessous.

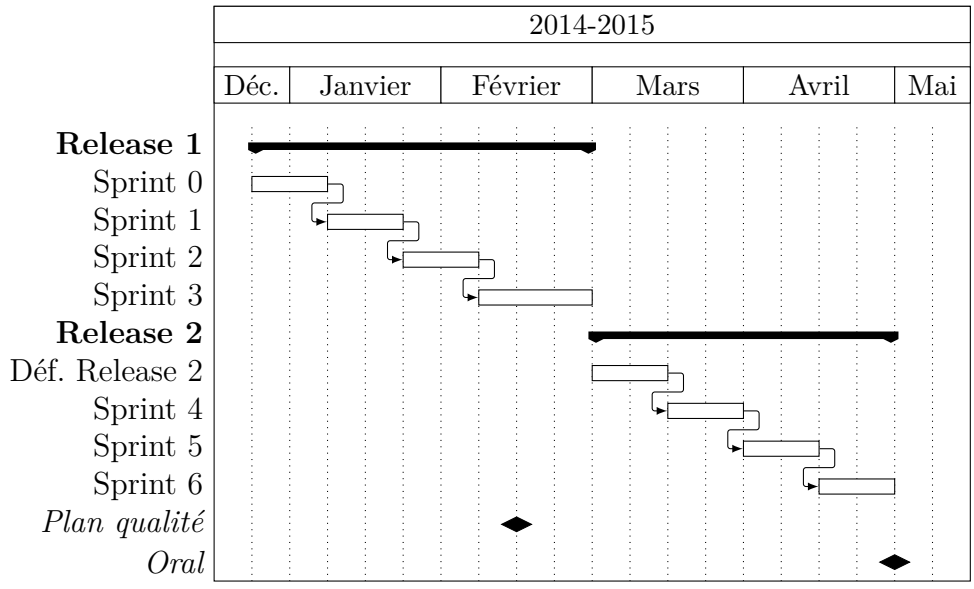

Figure 4.1 – Diagramme de Gantt du projet FactDev

5 La Démarche Qualité

La démarche qualité s'engage à offrir au client et à l'encadrant une procédure de qualité et d'amélioration continue. Dans un souci de respect de cette démarche, l'équipe à mis en place des outils et méthodes de fonctionnement.

5.1 Développement collaboratif

Comme cela fut décrit au préalable, le projet est réalisé selon la méthode *Scrum*. L'outil de gestion de version *Git* mis en place assure une certaine qualité au logiciel.

5.1.1 Outil de gestion de version : Git

Git est un outil puissant permettant de conserver les différentes versions de notre logiciel, de connaître les fonctions auxquelles sont affectées les membres de l'équipe et, en cas de difficultés, de revenir dans un état antérieur.

Git permet également, via le système des branches exposé plus haut, de travailler sur des versions séparées du code. Ainsi, il est possible de développer chacun de son côté sans créer de conflits avec le code des autres membres de l'équipe.

5.1.2 Revue de code : Github

La revue de code est un élément essentiel à la qualité d'un logiciel. Elle consiste à vérifier que la fonctionnalité spécifiée est correcte, que le code est simple, lisible et correctement documenté. Le site web d'hébergement *Github* de *Git* permet de mettre en application cette revue de code. En effet, lorsqu'un membre de l'équipe de développement considère la tâche qu'il avait affecté comme terminée, il crée une *Pull Request*. Un autre membre de l'équipe procède à la revue de code. Il est également possible pour les autres développeurs d'exposer la façon avec laquelle ils auraient traités la fonction.
Après approbation, on fusionne (*Merge*) le travail avec celui du reste du groupe.

5.2 Couverture des tests

Le logiciel *FactDev*manipule un certain nombre de données (utilisateur, clients, factures, ...) qui peuvent être ajoutées, supprimées ou modifiées. C'est pourquoi il est important de mettre en place des tests unitaires pour vérifier de l'exactitude des données avant et après manipulation. Pour cela, nous utilisons les bibliothèques fournies par Qt (*QtTest*).
En complément à ces tests, nous utilisons *lcov*, un outil graphique permettant, par analyse du code, de définir la couverture de code de nos tests unitaires.

5.3 Analyse statique de code : SonarQube

SonarQube s'avère être un outil capital à la qualité du code de l'application. En effet, il fournit des statistiques sur la qualité du code (pourcentage de documentation, duplication de code, complexité,...). Il se montre un excellent complément à la couverture du code.

6 Matrice de traçabilité

Exigences	1.		2.								3.	4.	5.			
	1.1	1.2	2.1			2.2							5.1		5.2	5.3
			2.1.1	2.1.2	2.1.3	2.2.1	2.2.2	2.2.3	2.2.4	2.2.5			5.1.1	5.1.2		
GEN01													X	X	X	X
GEN02[1]			X	X	X											
GEN03		X			X				X	X						
GEN04		X			X								X	X	X	X
GEN05									X		X					
GES01			X													
GES02				X												
GES03				X												
GES04							X	X			X		X	X		
GES05											X					
GES06											X					
GES07			X	X					X			X				
GES08		X														
GES09			X	X												
GES10				X					X							
GES11											X	X				
GES12												X				
AQ01			X			X	X	X								
AQ02					X											
AQ03										X						X
AQ04									X	X						
AQ05				X						X					X	X
AQ06				X						X					X	
AQ07											X					

Table 6.1 – Matrice de traçabilité

1. La méthode *Scrum* offre une certaine flexibilité puisque qu'après chaque revue de Sprint, le client est susceptible de demander des modifications.

A Conventions de programmation en C++

Voici les conventions écritures que nous avons fixé, il faudra les respecter pour que nous ayons un code propre et homogène, de plus elles ont été fixés pour que ce soit le plus simple pour nous (lecture rapide, propreté etc...).

Elles peuvent encore évoluer

A.1 English, of course !

Tout le code, et les `commits`, doivent être rédigés en Anglais. Soit, les noms de classes, de méthodes, de fichiers, de variables, d'attributs, et même les commentaires ;-) C'est pas compliqué, mais au moins, on se met tous d'accord, et puis voilà :)

A.2 Le nommage

A.2.1 Les attributs

Les attributs doivent toujours commencer avec une minuscule, pour séparer les mots, on les sépare avec une majuscule : utilisation de la Camel Case. Les noms de variables claires et explicite, quitte à ce qu'il soit un peu long, on a l'auto complétion que diable ! Donc les variables d'une lettre, à bannir ! (à part le i dans le cas d'un for, s'il n'est pas réutilisé après le for)

Les attributs seront préfixé par _ afin de pouvoir les reconnaître facilement.

```
int _superField;
bool _youLostTheGame;
```

A.2.2 Les noms de constantes

Les constantes doivent être tout en majuscule, les différents mots de la constante sont séparés par des underscore (_). Même remarque que pour les attributs, choisissez des noms de constante clair, compréhensible par tous, pas seulement par ceux qui sont dans votre tête !

```
const int MY_BEAUTIFUL_CONSTANT;
```

```
const bool YOU_LOST_THE_GAME_AGAIN;
const QString CONST;
```

> On évite d'utiliser les `#define` afin de garder un typage fort

A.2.3 Les noms de méthodes

Les méthodes doivent commencer par une minuscule, et séparer les différents mots par une majuscule(Camel case).

Les fonctions ne retournant rien (procédures) doivent toujours être à l'infinitif.

À l'opposé les fonctions retournant quelques choses doivent être au participe passé. Il faut décomposer au maximum, n'hésitez pas à faire une méthode `private` si besoin est, c'est toujours plus clair d'avoir une fonction, dictant explicitement ce qu'elle fait par son nom que 10 lignes de code bizarroodes avec 2-3 lignes de commentaire ! Et donc, les noms de fonctions sont essentiels ! !

```
void display(QString textToDisplay) {
    qDebug() << texteAAffiche;
}

bool test(int first, int second) {
  return (first + second);
}
```

Les accesseurs Les getters et les setters doivent être respectivement préfixés par `get` et `set` suivi du nom *exact* de l'attribut, même si le get ne respecte pas la convention de Qt.

```
int getValue();
void setValue(int);
```

A.2.4 Les noms de classes ou d'interface

Les noms de classe ou d'interface doivent tous commencer par une majuscule, les différents mots sont séparés par une majuscule, choisissez des noms de classes claires ! (oui, je me répète, mais c'est ce qui fait toute la compréhension facile, ou non, d'un programme les noms de variables, classe, paramètre, méthodes etc. . .)

A.3 L'indentation

La règle est simple, on ouvre une accolade, la ligne suivante sera décalé vers la droite(une tab = 4 caractères), on ferme une accolade, on décale l'accolade vers la gauche et tout ce qui suis.

Également, si une ligne est trop longue, on va a la ligne, et décalons d'une ligne vers la droite, une fois l'instruction finie, on redécalé vers la gauche.

Dans le cas d'un switch, le break doit s'aligner avec le case 42 : tout ce qui est entre case et break sera indenté.

Merci de mettre un espace avant chaque accolades, oui je sais je suis psychorigides, mais c'est moche sinon

```
class MaSuperClasse {
public:
  MaSuperClasse(int test, QString, machin, double _chose);
  int method();

private:
  int _test;
  QString _machin;
  double _chose;
};

MaSuperClasse::MaSuperClasse(int test, QString machin, double chose) {
  _test = test;
  _machin = machin;
  _chose = chose;
}

MaSuperClasse::method() {
  qDebug() << "Hello World";
  switch(yatta) {
    case 42:
       // ^^
       break;
    case 1337:
       // ...
       break;
    default:
       //
  }
}
```

A.4 Les accolades

Les accolades ouvrante sont positionnés à la fin de la ligne demandant une accolade (`switch`, `if`, `class`, `else`, `else if`, ...)

Les accolades fermantes sont positionnés une ligne après la dernière instruction. (avec une désindentation) Les `else` et `else if` se mettent sur la même ligne que l'accolade fermante.

```
int superMethod(void) {
  if(true) {
    // bla bla
  } else if(false) {
    // bla bla
  } else {
    //instruction
  }
}
```

A.5 Les types

Pour les types, au maximum, il vaut mieux privilégier les classes de Qt plutôt que les Types C++, autrement dit, on va utiliser les types suivants(c'est facile, ça commence pas un Q :)) :

- Primitives : `int`, `double`, `char`, `unsigned`, `bool`, ...
- `QString`, `QVariant`, `QNumber`, `QDate`, `QTime`, `QDateTime`
- Collections : `QList<Type>`, `QSet<Type>`, `QStack<Type>`, `QQueue<Type>`, `QLinkedList<Type>`, `QVector<Type>`, `QMap<Type1, Type2>` ...

Je vous ferais p'têtre un wiki sur les types cool en Qt... Mais sinon la doc est vraiment très bien fait ! Au pire, c'est assez intuitif « je veux une pile... Ça commence par Q. Comment on dit pile ? Stack ? Ah ben `QStack`. »

A.6 Les bonnes pratiques

A.6.1 Le mot clé `const`

Dès que vous pouvez... hop un `const`.

Autrement dit : si vous ne modifié jamais un paramètre, un attribut, une variable où que sait-je, on met un const. Ça permet de n'avoir aucune ambiguïté, c'est clair, et quelqu'un qui utilise la méthode sait que le paramètre ne serait pas modifié.

A.6.2 Le mot clé `void`

Une méthode ne contenant aucun paramètre *doit* contenir void, c'est comme ça.

```
void aSuperMethod(void);
```

A.6.3 Longueur du code

Une ligne ne doit pas excéder 100 caractères, une méthode ne doit pas excéder 60 lignes, un fichier doit être assez court... Mais ça on verra sur le moment ;-)

A.7 Conventions des composants graphiques

Element	Préfix
Combo	cb
Menus	mn
LineEdit	le
Buttons	btn
Table	tbl
Label	lb
TextEdit	te
Tree	tr
Dock	dck
Checkbox	chk
WIdget	wdg

A.8 Conventions de documentation

A.8.0.1 Pour une classe

```
/**
 * @author Nom de(s) la personne ayant programmé la classe sous la forme: ↩
    Prénom Nom
 * @brief The MaClass class (généré automatiquement) Role de la classe
 */
```

A.8.0.2 Pour un attribut ou une méthode

```
/**
 * @brief MaClass\dotsmaMethode Ce que fait la méthode
 * @see [optionnelle] Si cela fait référence à une autre méthode/classe/objet ↩
    alors on écrit le nom de cette
 * méthode/classe/objet
 * @param parametre1 Brève description du paramètre attendu et de ce qu'il ↩
    représente
 * @param parametre2 \ldots
 * @param parametreN \ldots
 * @return ce qui est retourné
 */
```

A.8.0.3 Remarques

On peut utiliser les balises HTML pour la documentation. Par exemple, dans @brief si on a une méthode addCustomer(int id), sa description pourrait être :

```
/**
 * @brief Customer\dotsaddCustomer Ajoute un nouveau client
 *  possédant l'identifiant <i>id</i>``.
 */
```

On utilisera alors comme convention : - Pour un paramètre « pParam » passé en paramètre :

```
/**
```

```
 * ...<i>pParam</i>
 */
```

- Pour le nom d'une classe « MaClass »

```
/**
 * ...<b>MaClass</b>
 */
```

Éviter les accents dans la documentation (Par exemple, ça sera @author Cedric Rohaut et non Cédric Rohaut)

B Table des figures

Index