POLYTECHNIQUE
MONTRÉAL

LE GÉNIE
EN PREMIÈRE CLASSE

Polytechnique Montréal

LOG8430-Architecture logicielle et conception avancée

TP2 - Ré-usinage Architectural du code Ring

Auteurs:
Ait Younes Mehdi,
Barbez Antoine,
Ouenniche Farouk,
Sierra Juan Raul,
Zins Pierre

Novembre 4, 2016

Contents

1 Introduction

L'objectif de ce TP est d'analyser la qualité du code du logiciel Ring. Pour cela, nous avons dans un premier temps analysé des métriques intéressantes. Ensuite, nous avons essayé d'identifier les patrons de conception logicielle utilisés par les développeurs de Ring. Nous avons également tenté de comprendre l'intérêt de ces patrons et de voir à quels problèmes ils répondaient. Enfin, dans une troisième partie, nous avons détecté des mauvaise pratiques ou anti-patrons dans le code de Ring. Nous avons essayé d'expliquer les problèmes causés par chaque anti patron, avant de proposer une solution. Pour réaliser ce TP, nous avons pu nous baser sur la connaissance du logiciel Ring acquise lors du premier TP. Dans le troisième TP, nous essaierons de nous appuyer sur notre analyse de ce TP, afin d'implémenter nos solutions aux problèmes détectés.

2 Analyser de la qualité du code de Ring

Dans cette partie, nous nous attacherons à évaluer la qualité du code du logiciel Ring dans son ensemble. Il s'agit ici d'avoir une vue globale de Ring et de dégager les caractéristiques communes à toutes les parties du logiciel.

2.1 Métriques utilisées

Tout d'abord, dans cette étude, nous utiliserons exclusivement des métriques statiques pour l'analyse de la qualité logicielle. En effet, il s'agit ici d'analyser la qualité du code en termes de bonne utilisation des concepts de la programmation orientée objet et de le juger sur sa lisibilité et sa capacité à être facilement retravaillé. Nous ne jugerons donc pas ici ses caractéristiques pendant l'exécution.

Ensuite, étant donné que nous ne nous intéressons ici à aucune partie du logiciel en particulier, mais bien à Ring dans son ensemble. Nous ferons d'abord appel à des métriques de base, qui sont de simples données résultant du comptage de certains éléments du code. Nous utiliserons donc le nombre de lignes de code (SLOC) tout en prenant en compte la nature de ces lignes (code, commentaire, blank line...), ainsi que le nombre de classes.

Nous nous intéresserons cependant par la suite à dégager des comportements extrèmes présents dans le code de Ring. Nous utiliserons donc pour celà des métriques normalemnt utilisées pour décrire des classes ou parties bien particulières d'un logiciel. Ces métriques sont les suivantes : SLOC, nombre de méthodes, taille des méthodes et complexité cyclomatique. Nous étudierons également les rapports qu'une classe entretient avec les autres : nombre de dépendances avec d'autres classes , nombre d'attributs publics , nombre d'include...

2.2 Analyse

2.2.1 Analyse de Ring dans son ensemble

Une première chose à dire est que ring est un projet assez conséquent, puisqu'il comptabilise au total près de 150000 lignes de code, 738 fichiers et implémente plus de 550 classes.

Un code relativement bien commenté : On remarque en effet que dans son ensemble, le projet ring est composé de 87000 lignes de code et 29000 lignes de commentaire, ce qui veut dire que un quart des lignes du projet, sont des lignes

de commentaire. Cependant, on peut voir que la plupart des fichiers du projet contiennent tout en haut plusieurs lignes de commentaire concernant les auteurs et la licence. Ces commentaires ne concernent pas le code du logiciel et ne devraient donc pas être pris en compte dans notre calcul. Malgré tout, cela peut être compensé par le fait que les noms des variables sont en général très explicite. Ainsi, les commentaires ne sont pas toujours utiles. Dans l'ensemble c'est un premier bon point pour ring, qui s'assure que son code soit rapidement compréhensible.

2.2.2 Valeurs moyennes et extrèmes

SLOC : Nombre de methodes : **Complexité cyclomatique :** En effet, on remarque en faisant une analyse de la complexité cyclomatique moyenne dans la couche du daemon, que la complexité cyclomatique atteint un maximum à 52. Même si il est vrai que une forte complexité cyclomatique n'est pas toujours sinonyme de défauts de conception, on peut néamoin penser qu'un travail d'amélioration peut être fait en décomposant certaines méthodes trop longues. Relations entre les classes :

3 Identification des design patterns

3.1 Patron 1 : Façade

3.1.1 Définition et emplacement

Le patron de conception façade est identifié dans la "classe" Callmanager. Le Callmanager permet de fournir un point d'accès pour faciliter la communication entre les différentes couches (dans ce cas entre les couches LRC et daemon) et permet une flexibilité d'adaptation avec différentes technologies. L'objectif est de regrouper les interfaces d'un ensemble d'objets en une interface unifiée rendant cet ensemble plus simple à utiliser pour un client.

Emplacement : `Callmanager.cpp`

3.1.2 La raison d'utilisation

Ce patron de conception est utilisé parce qu'il va permettre de fournir une interface de haut niveau qui réduit amplement la complexité de la communication entre les différents sous-systèmes. Il permet aussi de fournir un point d'accès à un sous-système en encapsulant ce dernier avec une interface unifiée. Dans la couche LRC, c'est la classe CallManagerInterface du fichier "callmanager_wrap" qui sera connectée à la façade.

3.1.3 Schématisation du patron

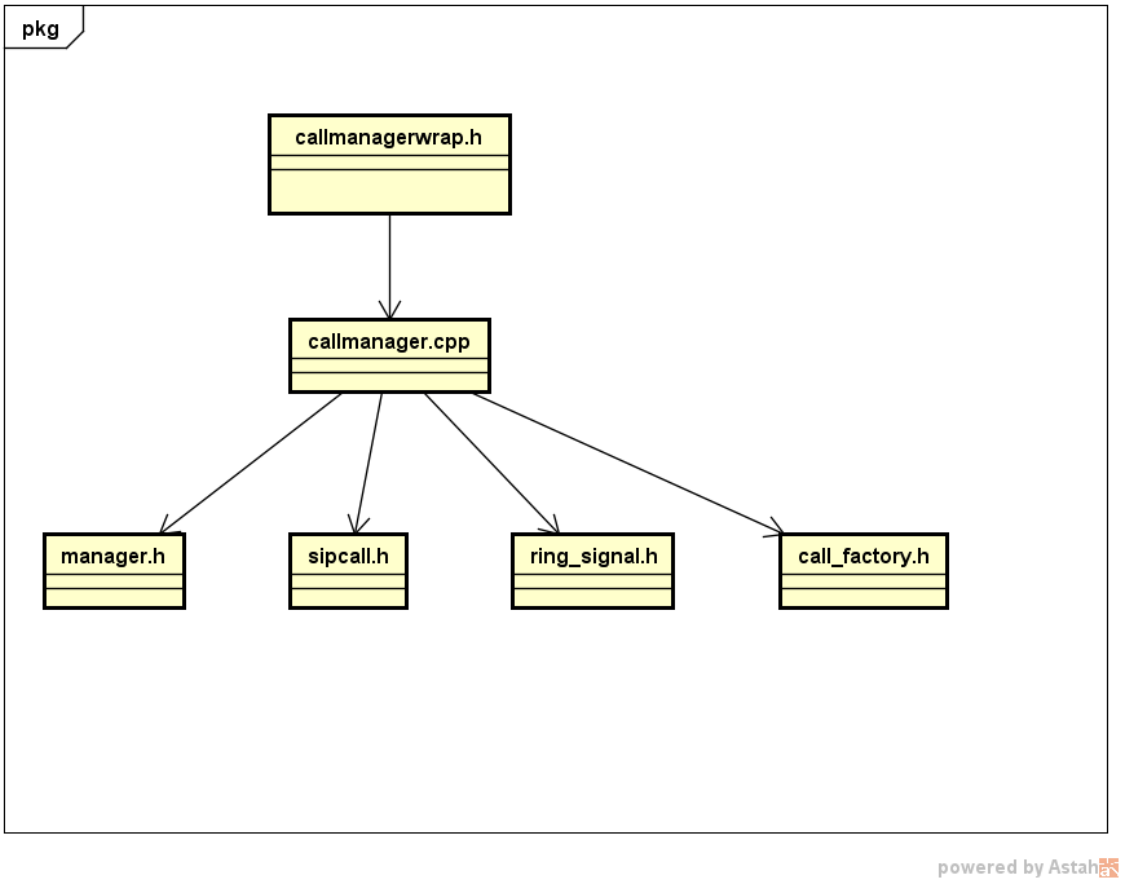

Figure 1: Schéma du patron Façade CallManager

3.2 Patron 2 : Singleton

3.2.1 Définition et emplacement

Nous avons identifié le patron singleton dans la classe Manager. Le Manager est en charge de la coordination entre les différentes classes dans le module daemon. Ce patron de conception a pour but d'assurer qu'une classe ne possède qu'une seule instance et fournit une méthode unique retournant cette instance. A partir de l'analyse des scénarios dans le TP1 on a clairement pu remarquer la présence de la classe Manager comme singleton dans l'application Ring. Cette classe participe dans presque toutes les activités du logiciel.

Emplacement : `Manager.cpp`

3.2.2 La raison d'utilisation

Ce patron de conception est utilisé parce qu'il permet une implémentation plus simple pour contrôler les activités dans l'application Ring. Le Manager va jouer le rôle de contrôleur pour l'ensemble du module daemon (vidéo, appel, message) et pour cette raison il est instancié qu'une seule fois. Il s'agit vraiment de la classe probablement la plus importante qui va tout gérer. On peut donc difficilement envisager l'instanciation de deux ou plus contrôleurs. En effet, cela pourrait amener à des blocage ou conflits entre les contrôleurs. Le manager offre par exemple des services aux classes AccountFactory, CallFactory et Call Manager.

3.2.3 Schématisation du patron

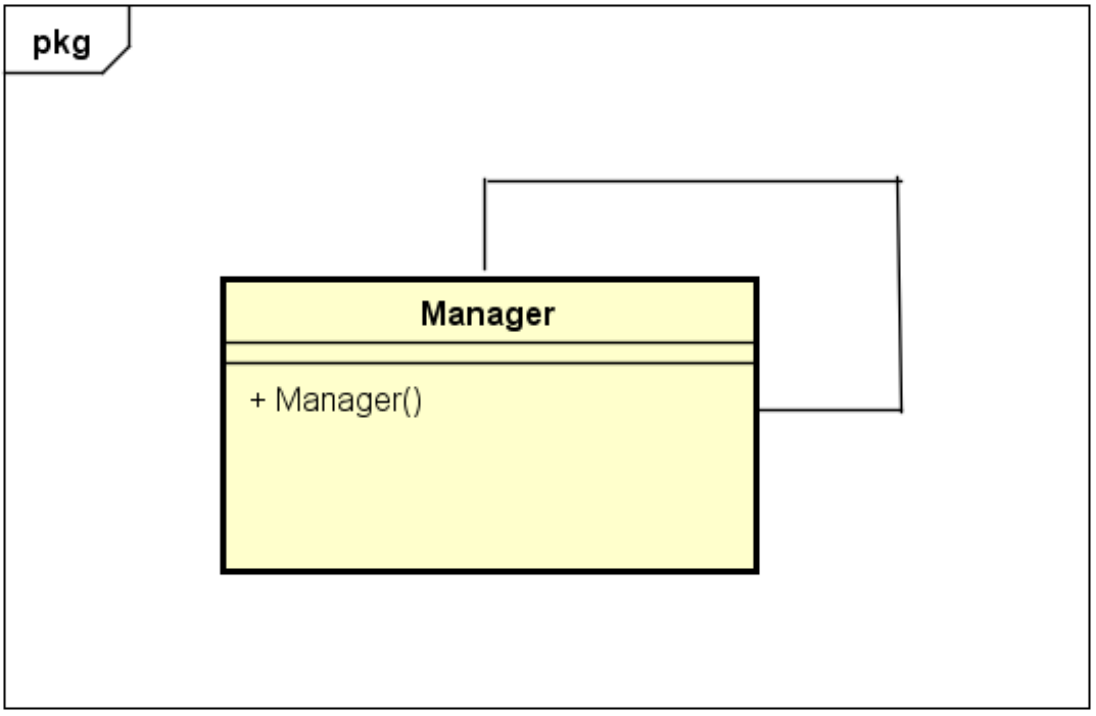

Figure 2: Schéma du singleton Manager

3.3 Patron 3 : Abstract Factory

3.3.1 Définition et emplacement

Nous avons identifié le patron de conception abstract factory, pour la création des comptes "Account" : **AccountFactory**

Ce patron de conception propose une interface pour créer des familles d'objets liés, dépendants entre eux, sans spécifier les classes concrètes.

Nous avons principalement identifié ce patron en analysant le code et le modèle. Nous avions vu dans le TP1 que Ring disposait de plusieurs comptes (ringAccount et SIPAccount). Ainsi, il nous semblait possible de trouver ce genre de patron de conception pour la création des comptes. De plus, nous avons identifié une classe *AccountFactory* avec la méthode *"createAccount()"*. Son nom suggère fortement l'utilisation du patron Factory. Enfin, en analysant le code de cette classe ainsi que les classes Account, RingAccount, SIPAccountBase et SIPAccount, nous avons bien identifié le patron de conception "Abstract factory".

3.3.2 La raison d'utilisation

D'après nous, les développeurs de Savoir Faire Linux ont utilisé ce patron de conception, car il propose une solution efficace et reconnue pour le problème de création d'objets de différents types. De plus, c'est la fabrique qui déterminera dynamiquement, pendant l'exécution, la classe concrète de l'objet à créer. Ce patron va permettre d'isoler la création des objets Account de leur utilisation. Concernant la maintenance du code, il aura un intérêt fort. En effet, il permettra aux développeurs de rajouter de nouveaux types de comptes (XXXAccount), sans modifier le code qui utilise l'objet de base. Ainsi on le code va respecter un élément fondamental du principe SOLID : l'ouverture aux extensions.

La lisibilité du code est également amélioré, puisque la partie concernant la création des objets est isolée et peut être utilisé pour les autres objets de la fabrique. Cela simplifie et réduit vraiment le code.

3.3.3 Schématisation du patron

La classe *AccountFactory* va donc permettre de créer les différents objets Account. La classe *Account* est une classe de base abstraite. La classe *SIPAccountBase* va en dériver et les classes concrètes des deux types de comptes *SIPAccount* et *RingAccount*

dériveront de *SIPAccountBase*. L'objet AccountFactory, possède un membre *generators_* qui sera initialisé dans le constructeur de AccountFactory. Ce dernier sera un en fait une map, qui associera le type du compte avec une fonction pour le créer. Un objet compte sera créé par *AccountFactory* au travers de la méthode *createAccount*. Le type de compte à créer sera passé en paramètre, puis dynamiquement à l'aide de *generators_*, le bon compte sera créé et retourné par la méthode.

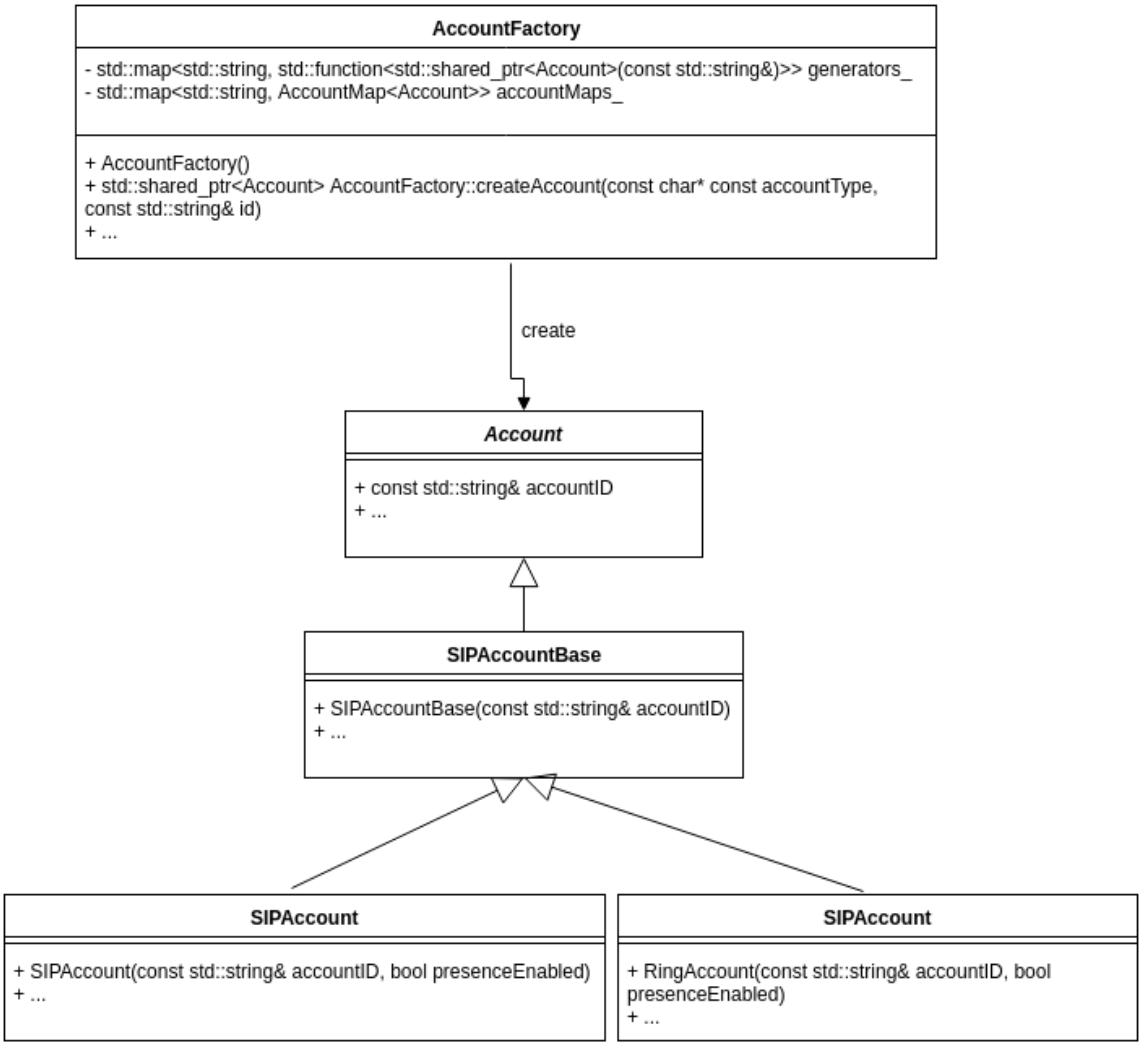

Figure 3: Schéma du patron de conception "Abstract Factory" utilisé dans Ring

4 Identification des défauts de conceptions

4.1 Défaut 1 : God class (Blob)

4.1.1 Définition et emplacement

La *God class* ou *Blob* est la conception d'une classe monopolisant le traitement. Ce défaut de conception est caractérisée par un diagramme de classe composé d'une seule classe (souvent appelée "*contrôleur*") complexe entourée par des classes de données simples. Le principal problème ici est que la majorité des responsabilités sont déléguées à cette seule classe.

Emplacement : `daemon/src/manager.cpp`.

Afin d'identifier cet anti-patron nous avons utilisé notre expérience acquise lors du *TP1* ainsi que la métrique de *nombre de lignes*. En effet la classe (la déclaration dans le header et la définition de ses méthodes) compte plus de ***2800 lignes***. Bien évidement, nous ne nous sommes pas uniquement basés sur cette métrique, nous avons analysé la classe : ses responsabilités et son rôle. Dans le cas du `Ring`, la classe `manager.cpp` est effectivement utilisée comme contrôleur, néanmoins la classe a trop de responsabilités : `Passer les appels, créer les comptes, vérifier l'état des drivers ....`

4.1.2 Impact

La *god classe* va à l'encontre du principe Orienté Objet (diviser les gros problèmes en plusieurs plus petits, ou encore "diviser pour régner") et donc de ses avantages. La *god classe* limite la possibilité de modifier l'application sans affecter la fonctionnalité d'autres objets encapsulés. Les modifications apportées à une *god classe* affectent l'application complète en raison des responsabilités et des relations qu'elle a avec les autres classes. De plus, la *god classe* est trop **complexe** à **réutiliser** et à **tester**, ce qui réduit donc l'**évolutivité** de l'application. La *god classe* peut par ailleurs être coûteux en mémoire à la compilation et donc augmenter considérablement le temps de compilation.

4.1.3 Solution et schématisation des modifications

Afin de réduire le nombre de responsabilités de la *god classe*, il suffit de séparer ces dernières dans des classes ayant des responsabilités plus spécifiques. Deux solutions possibles :

Solution 1 :
La première solution propose de séparer le `manager` principal en quatre sous-manager afin que chaque `manager` soit dédié à un service (ou responsabilité) spécifique (Figure 4).

- `accountManager` : S'occupe de la gestion et de la création des différents comptes (`RingAccount et SIPAccount`)
- `messageManager` : S'occupe de l'envoi et de la réception des messages entre comptes
- `callManager` : S'occupe de la gestion des appels (passer un appel, répondre, mettre en attente, transférer, raccrocher ...)
- `conferenceManager` : S'occupe de la gestion des conférences (Créer un conférence, ajouter des participants, retirer des participants, mettre en attente ...)
- `manager` : S'occupe des initialisations et joue le role de façade avec les autres managers. La fonction `pollEvent()` (méthodes qui est périodiquement appelées dans la main *loop* du `daemon` afin de gérer les événements) resterait implémentée au sein du `manager` principal.

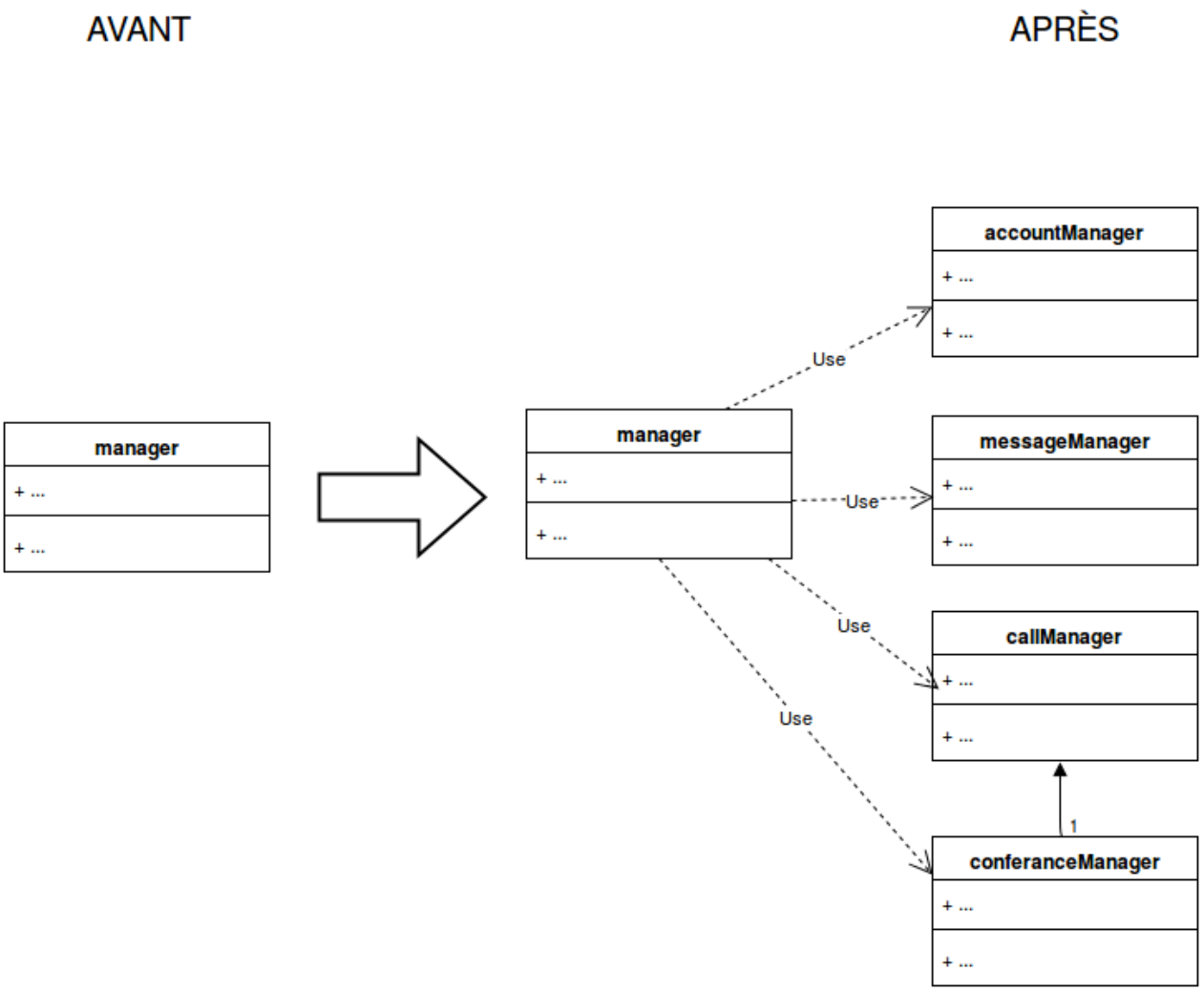

Figure 4: Schéma de re-factoring du *manager*, proposition 1

Solution 2 :
Cette deuxième solution comme la précédente propose de séparer le `manager` principal en sous-manager (de manière plus poussée que la solution précédente) afin que chaque `manager` soit dédié à un service (ou responsabilité) spécifique (Figure 5).

- `accountManager` : S'occupe de la gestion et de la création des différents comptes (`RingAccount et SIPAccount`)
- `messageManager` : S'occupe de l'envoi et de la réception des message entre comptes.
- `configurationManager` : S'occupe de la gestion des configurations et des vérifications de drivers et périphériques (audio, vidéo)
- `callManager` : S'occupe de la gestion des appels (passer un appel, répondre, mettre en attente, transférer, raccrocher ...)
- `conferanceManager` : S'occupe de la gestion des conférences (Créer un conférence, ajouter des participants, retirer des participants, mettre en attente ...)
- `callActions` : S'occupe de lier les méthodes communes à une action d'appel/conférence (répondre, raccrocher, appeler ...)
- `manager` : S'occupe des initialisations et joue le role de façade avec les autres managers. La fonction `pollEvent()` (méthodes qui est périodiquement appelée dans la main *loop* du `daemon` afin de gérer les événements) resterait implémentée au sein du `manager` principal.

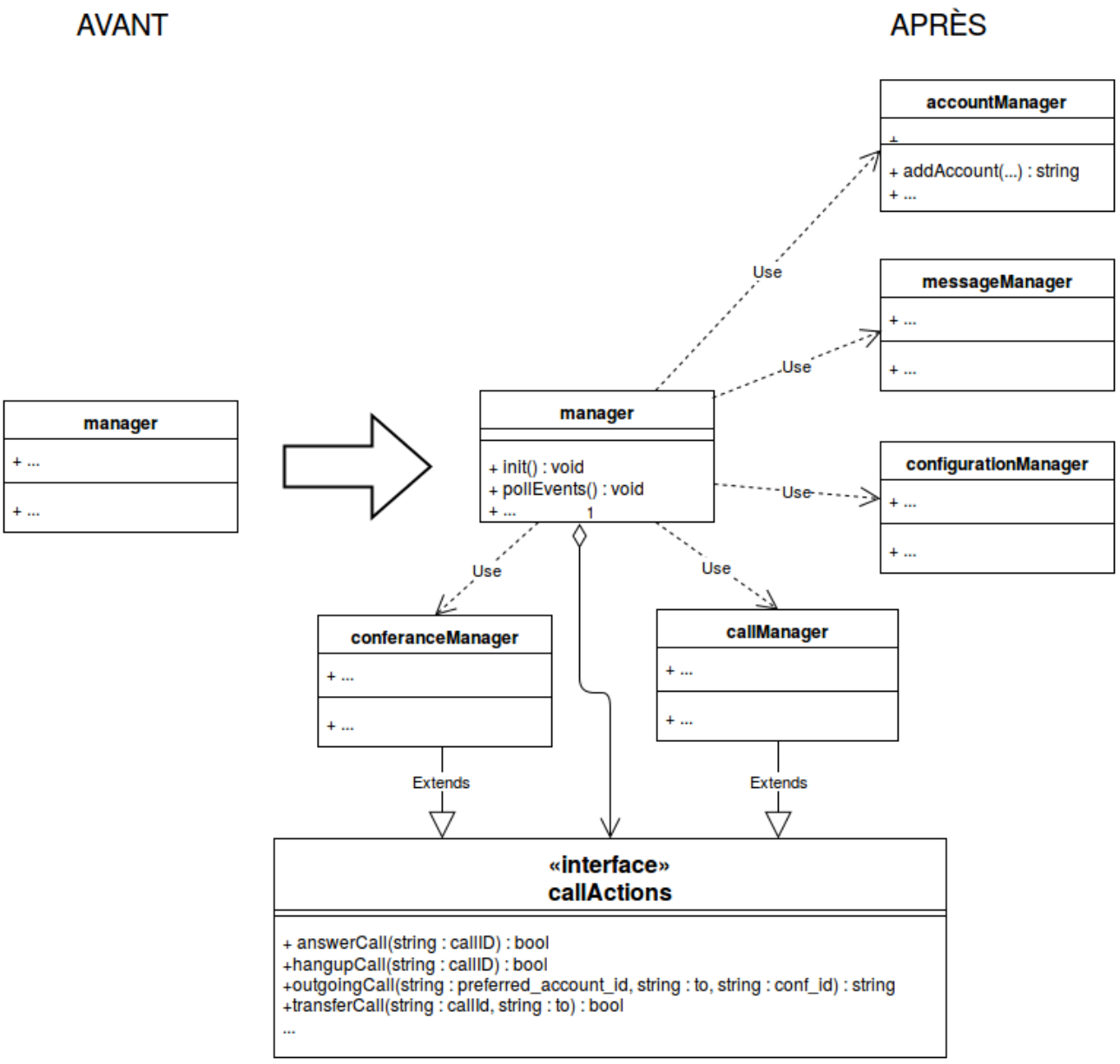

Figure 5: Schéma de re-factoring du *manager*, proposition 2

4.2 Défaut 2 : Longue Méthode

4.2.1 Définition et emplacement

La longue méthode comme son nom l'indique est une méthode qui contient trop de lignes de code. C'est une méthode qui fait plus que la tache qui lui est généralement attribuée. Souvent une astuces utilisée pour savoir si une méthode est trop longue est de voir si plusieurs commentaires sont présents au sein de la même méthode. En effet, les commentaires sont utilisés pour documenter des procédures complexes ou des sous-procédure. Ainsi, si une méthode contient beaucoup de commentaires, on peut imaginer que l'anti patron "longue méthode" est présent.

Emplacements :

- void Manager::init(...) : La fonction permet d'initialiser l'instance du manager et d'autres propriétés tels que : `PJSIP, GNUTLS, ICE, et LOGDHT`
  - Nombre de lignes : 80
  - Complexité Cyclomatique : 12
  - Lignes de commentaires : 8
- void RingAccount::doRegister_() : La fonction permet d'enregistrer un utilisateur(Ring Account) sur le `DHT` et d'écouter les appels entrants.
  - Nombre de lignes : 240
  - Complexité Cyclomatique : 14
  - Lignes de commentaires : 5
- `daemon/src/RingAccount.cpp -> bool RingAccount::SIPStartCall(...)`
  - Nombre de lignes : 81
  - Complexité Cyclomatique : 8
  - Lignes de commentaires : 10

Afin d'identifier ce défaut nous avons utilisé notre expérience acquise lors du *TP1* ainsi que les métriques de *nombre de lignes*, *complexité cyclomatique* et *nombre de lignes de commentaire* La métrique nombre de lignes nous a aidé à avoir une vue générale de la méthode alors que la métrique complexité cyclomatique nous a aidé à avoir une idée plus structurelle de cette dernière. Quant à la métrique de nombre de lignes de commentaire, il s'agissait davantage d'une métrique secondaire

comparativement aux deux précédentes. Elle nous a permis de voir éventuellement (et proportionnellement au nombre de ligne de code) les séparations de taches dans une méthode. Néanmoins, afin de re-sortir avec ces méthodes ils nous a fallu analyser manuellement les méthodes pour bien comprendre leurs responsabilités.

Note : Le nombre de longues méthodes n'a pas pu être identifié car nous n'avons pas utilisé de logiciel de détection automatique. Nous avons choisis les 3 précédentes méthodes à titre d'exemple. Nous avons par la suite choisi de proposer des solutions de re-factoring sur deux d'entre elles.

4.2.2 Impact

Plus une méthode ou une fonction est longue, plus il devient difficile de la comprendre et de la maintenir. Cela peut également être contraignant pour les évolutions futures du code. Enfin, les longues méthodes offrent un endroit propice pour de la duplication de code.

4.2.3 Solution et Schématisation des modifications

La solution est de *spliter* la méthode selon les différents comportements et taches qu'elle effectue, ou plus précisément prendre les parties de la méthode qui semble indépendantes et les extraire dans des méthodes à part puis de les rappeler dans la méthode principale. Cette méthode est appelée : "*Extract Method*"[2]

init() :

- `initPJSIP()`: Cette méthode s'occupera des initialisations faites sur `PJSIP`
- `initLog()`: Cette méthode sera en charge d'initialiser les logs sur du `DHT et GNUTLS`. Elle s'occupera aussi du `reset` sur le `iceTransport` et de la gestion de fichiers de configuration.

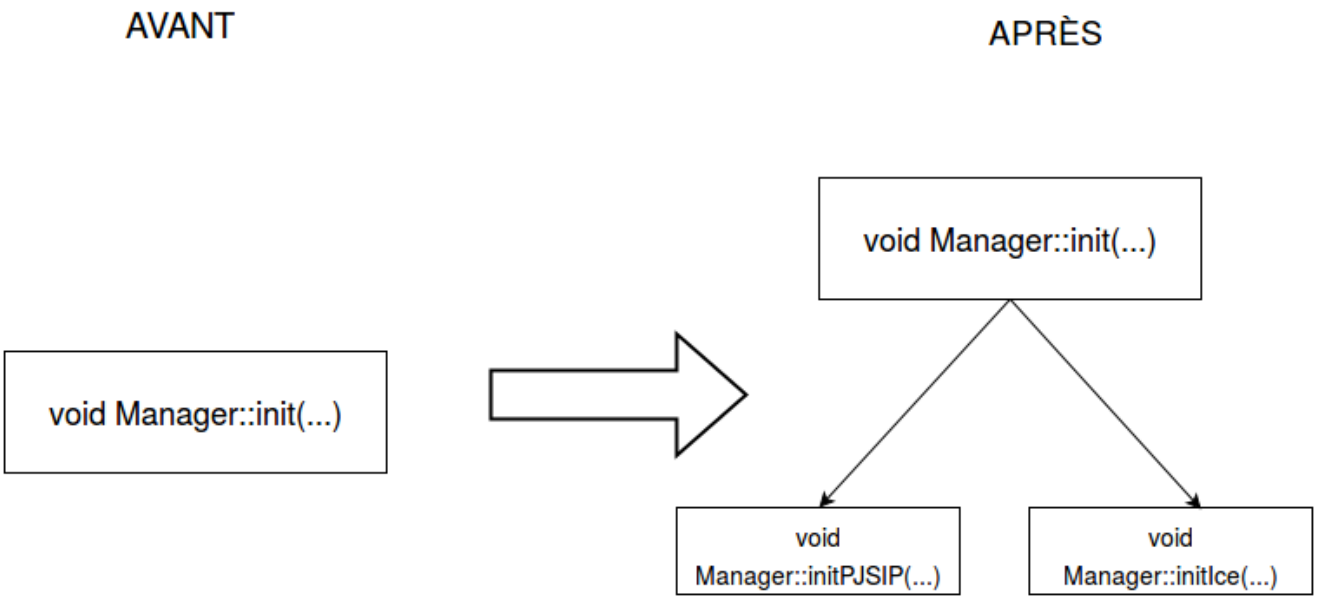

Figure 6: Schéma des changements effectués sur la méthode *init*

doRegister_() :

- `dhtState()`: Cette méthode sera en charge d'enregistrer le compte sur le `DHT`.
- `dhtLog()`: Cette méthode s'occupera de récupérer les différents niveaux de log (Error, Warning, Debug, ...)
- `deviceAnoncement()`: Cette méthode mettra également à jour l'état du périphérique et du compte sur le `DHT`.
- `callListner()`: Cette méthode écoutera les appels entrants.

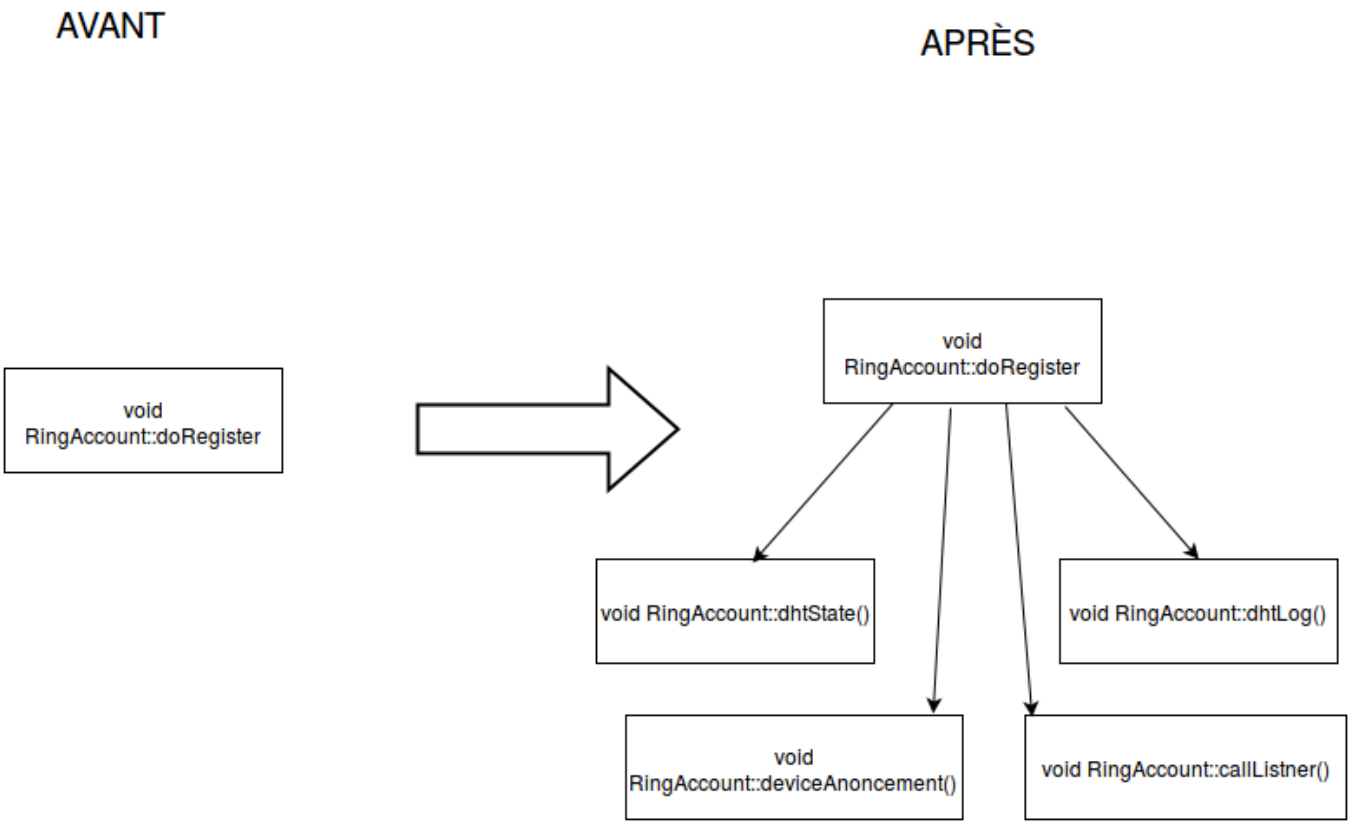

Figure 7: Schéma des changements effectués sur la méthode *doRegister*

4.3 Défaut 3 : Complexité Cyclomatique : `The Arrowhead Anti-Pattern`

4.3.1 Définition et emplacement

"Le nombre cyclomatique, la complexité cyclomatique ou la mesure de McCabe est un outil de métrologie logicielle développé par Thomas McCabe en 1976 pour mesurer la complexité d'un programme informatique. Cette mesure comptabilise le nombre de « chemins » au travers d'un programme représenté sous la forme d'un graphe." [3] De façon plus simple, ce nombre permet d'identifier le nombre de chemins possibles que peut prendre l'exécution d'une méthode.

`"The Arrowhead Anti-Pattern"` tire son nom de la forme prise par le code, lors de l'utilisation de plusieurs structures conditionnels [1], des instructions de commutation ou tout ce qui permet au flux de prendre plusieurs chemins. Ces instructions sont généralement imbriquées les unes dans les autres, créant ainsi la flèche ou la forme du triangle (ou tranche de pizza). Évidement, cet anti-patron ne repose pas toujours sur cette forme de flèche, mais plutôt sur la ***complexité cyclomatique***.

Emplacement : `daemon/src/call.cpp -> std::string Call::getStateStr() const`.

La Figure 8 représente la méthode `getStateStr` qui a une complexité cyclomatique de 17. Nous nous somme basé sur les travaux de Thomas McCabe [5] afin de définir le seuil acceptable de la complexité cyclomatique d'une méthode qui est de **10**.

Afin d'identifier ce patron de conception, nous nous sommes référé à la métrique de complexité cyclomatique. Nous avons ensuite analysé les méthodes ayant un fort nombre d'imbrications conditionnelles (qui vont créer la forme de flèche). Cette partie a été faite en analysant manuellement les méthodes récupérées lors du premier tri basé sur la complexité cyclomatique. Néanmoins, la méthode `getStateStr` n'est bien évidement pas la seul méthode a excéder ce nombre. Nous l'avons choisi à titre d'exemple et car la correction de cette dernière ne devrais pas impacter les reste du comportement de l'application.

- Nombre de lignes : 53
- Complexité Cyclomatique : 17

```
std::string
Call::getStateStr() const
{
    using namespace DRing::Call;

    switch (getState()) {
        case CallState::ACTIVE:
            switch (getConnectionState()) {
                case ConnectionState::PROGRESSING:
                    return StateEvent::CONNECTING;

                case ConnectionState::RINGING:
                    return isIncoming() ? StateEvent::INCOMING : StateEvent::RINGING;

                case ConnectionState::DISCONNECTED:
                    return StateEvent::HUNGUP;

                case ConnectionState::CONNECTED:
                default:
                    return StateEvent::CURRENT;
            }

        case CallState::HOLD:
            if(getConnectionState() == ConnectionState::DISCONNECTED)
                return StateEvent::HUNGUP;
            return StateEvent::HOLD;

        case CallState::BUSY:
            return StateEvent::BUSY;

        case CallState::INACTIVE:
            switch (getConnectionState()) {
                case ConnectionState::PROGRESSING:
                    return StateEvent::CONNECTING;

                case ConnectionState::RINGING:
                    return isIncoming() ? StateEvent::INCOMING : StateEvent::RINGING;

                case ConnectionState::CONNECTED:
                    return StateEvent::CURRENT;

                default:
                    return StateEvent::INACTIVE;
            }

        case CallState::OVER:
            return StateEvent::OVER;

        case CallState::MERROR:
        default:
            return StateEvent::FAILURE;
    }
}
```

Figure 8: bloque `switch/case` de la méthode `getStateStr`

4.3.2 Impact

La complexité cyclomatique d'une méthode a un impact direct sur sa compréhension et sa maintenance. Les études montrent une corrélation entre la complexité cyclomatique d'un programme et sa fréquence d'erreur, une faible complexité cyclomatique contribue à la compréhensibilité d'un programme et indique qu'il est susceptible de le modifier à moindre risque. [4]. La complexité cyclomatique d'un module est également un indicateur fort de sa testabilité [4].

4.3.3 Solution schématisation des modifications

En plus d'une forte complexité cyclomatique, on peut très clairement remarquer la ressemblance entre les blocs conditionnels internes des *case* `CallState::Active et CallState::Inactive` : ***Duplication de code***.

La solution proposée va d'une partie réduire la complexité de la méthode à **9** et en plus supprimer la duplication de code présente (d'une pierre deux coups). Nous avons encore une fois utilisé la méthode de "*Extract Method*" afin de réduire la complexité de la méthode, de supprimer la duplication de code, et de retirer le `"The Arrowhead Anti-Pattern"`. Nous avons crée deux méthodes `onActiveState & onInActiveState` (Figure 10) qui vont implémenter respectivement les sous-blocs `switch/case` des cas `Active & Inactive` (Figure 9).

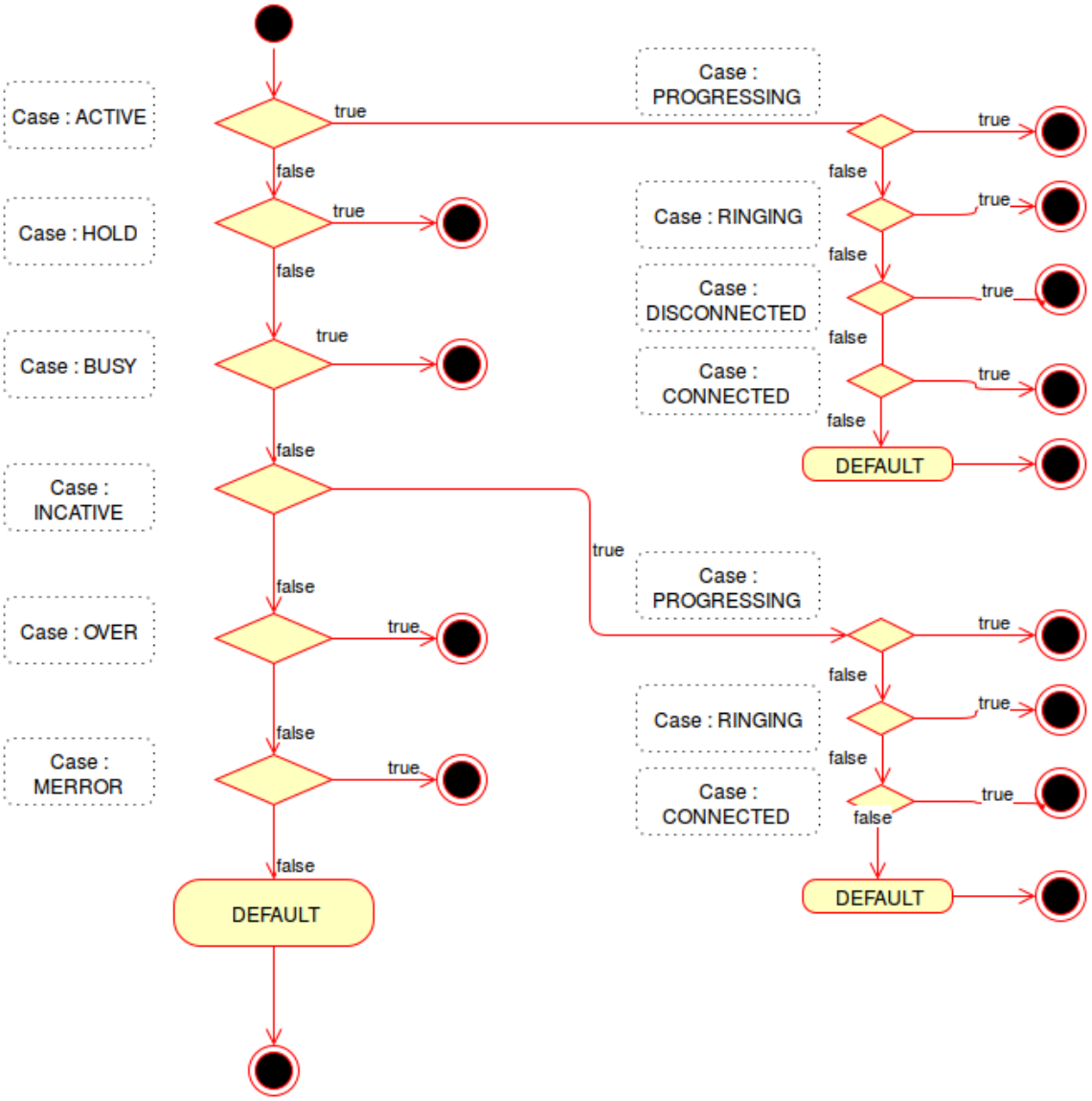

Figure 9: Diagramme d'activé de la méthode `getStateStr` avant re-factoring

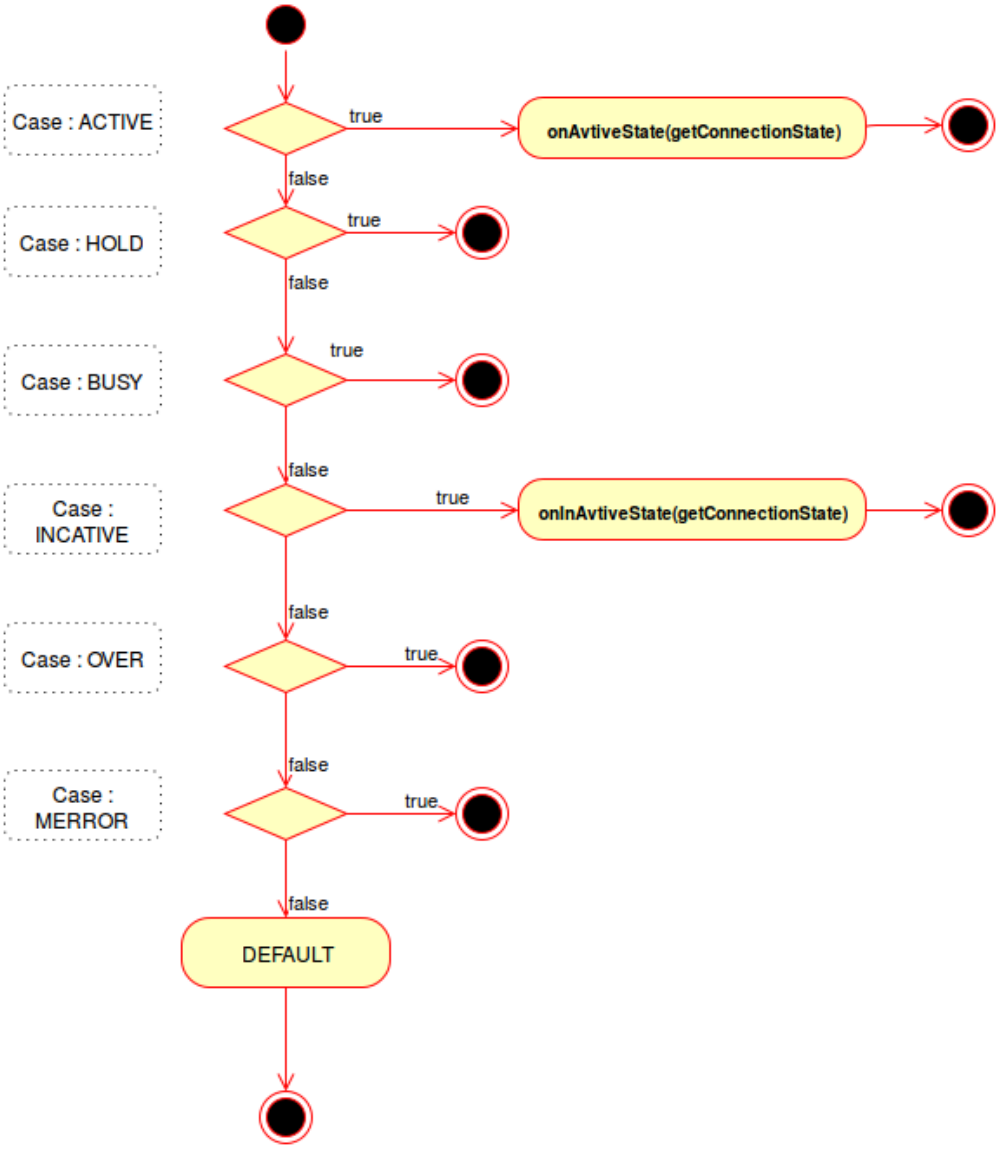

Figure 10: Diagramme d'activité de la méthode `getStateStr` après re-factoring

4.4 Défaut 4 : Duplication de code (non utilisation du *"State Pattern"*)

4.4.1 Définition et emplacement

La duplication de code (connus aussi sous le nom de *"Cut-And-Paste Programming")*, comme son nom l'indique est la répétition de deux ou plus fragments de code dont l'apparence et le comportement sont identiques (ou presque identique).

Emplacement : `daemon/src/manager.cpp -> bool Manager::joinParticipant(...)`.

Afin de localiser cette répétition nous nous sommes basés sur la métrique de nombre de lignes. Néanmoins cette métrique nous a uniquement permis de faire un premier tri (de façon générale) sur les longues méthodes qui sont les endroit ou apparaît très fréquemment la duplication de code. Afin de localiser plus précisément ce défaut de code, nous nous sommes appuyé sur les connaissances acquises lors du TP1. Nous avions déjà lors de ce TP localisé cette duplication. Par ailleurs, en analysant plus en détails la méthode, nous nous sommes aperçu que il y a un comportement qui est dépendant d'un changement d'états. Ce dernier a été implémenté en dur, alors que l'utilisation du ***State Patern*** et très fortement recommandé dans ce genre de situation.

```
call1->setConfId(conf->getConfID());
getRingBufferPool().unBindAll(callId1);

call2->setConfId(conf->getConfID());
getRingBufferPool().unBindAll(callId2);

// Process call1 according to its state
std::string call1_state_str(call1Details.find("CALL_STATE")->second);
RING_DBG("Process call %s state: %s", callId1.c_str(), call1_state_str.c_str());

if (call1_state_str == "HOLD") {
    conf->bindParticipant(callId1);
    offHoldCall(callId1);
} else if (call1_state_str == "INCOMING") {
    conf->bindParticipant(callId1);
    answerCall(callId1);
} else if (call1_state_str == "CURRENT") {
    conf->bindParticipant(callId1);
} else if (call1_state_str == "INACTIVE") {
    conf->bindParticipant(callId1);
    answerCall(callId1);
} else
    RING_WARN("Call state not recognized");

// Process call2 according to its state
std::string call2_state_str(call2Details.find("CALL_STATE")->second);
RING_DBG("Process call %s state: %s", callId2.c_str(), call2_state_str.c_str());

if (call2_state_str == "HOLD") {
    conf->bindParticipant(callId2);
    offHoldCall(callId2);
} else if (call2_state_str == "INCOMING") {
    conf->bindParticipant(callId2);
    answerCall(callId2);
} else if (call2_state_str == "CURRENT") {
    conf->bindParticipant(callId2);
} else if (call2_state_str == "INACTIVE") {
    conf->bindParticipant(callId2);
    answerCall(callId2);
} else
    RING_WARN("Call state not recognized");
```

Figure 11: Duplication de code et zone de d'implémentation du "*State Pattern*" au niveau de la méthode `joinParticipant`

4.4.2 Impact

La duplication de code rend généralement la modification de la méthode/classe plus difficile en raison des augmentations inutiles de sa complexité et de sa longueur. Ce défaut peut aussi facilement augmenter les coûts et efforts de maintenance et de ré utilisabilité. De plus, la duplication de code augmente les chances d'erreur de la part des développeurs (ex : un mauvais copier/coller). Il aussi très fréquent de voir des bouts de code oubliés et non utilisés (code mort : anti patron Lava Flow) suite à la duplication de code.

4.4.3 Solution et Schématisation des modifications

Deux solutions possibles :

Solution 1 :
La première solution est la plus simple est plus rapide, il suffit de retirer les deux blocs conditionnels (`call1_state_str et call2_state_str`) et de les isoler dans un méthode séparée. Il suffira par la suite d'appeler la méthode `accordingCall` deux fois en lui passant en paramètre la première fois : `conf et call1Details` et la deuxième fois : `conf et call2Details` (Figure 12)

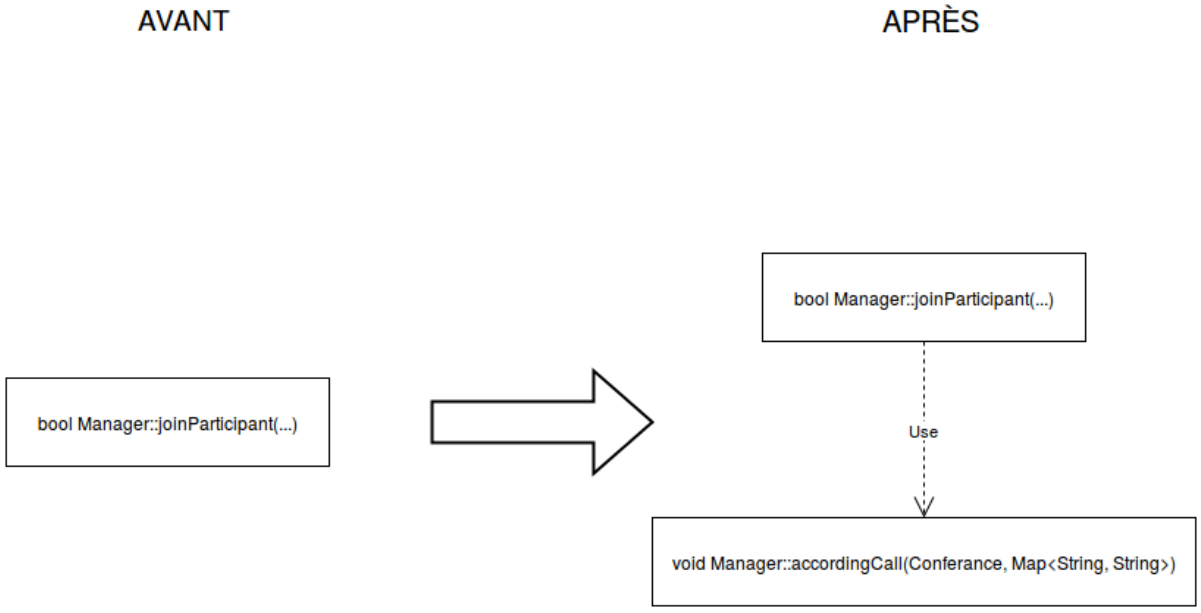

Figure 12: Séparation de méthode au niveau de `joinParticipant` afin de supprimer la duplication de code

Solution 2 :
Comme expliqué dans la définition, il est aussi possible d'utiliser le patron de conception `State`. Ce patron de conception peut être implémenté quand le comportement d'un objet dépend d'un état, et que cet objet doit changer son comportement lors de l'exécution en fonction de cet état. Ce cas se présente dans la fonction `joinParticipant`, l'état étant représenté par les variables `call1_state_str et call2_state_str`. Dépendamment de l'état de ses variables, une méthode spécifique est appelée.

De plus, en implémentant ce patron on retire aussi la duplication de code, car il suffira d'appeler deux fois la méthode `doOnchange` de la classe `Call` au niveau de la méthode `joinParticipant`. La méthode `doOnchange` re-implémentera le précédent bloc conditionnel de la méthode `joinParticipant`, et en fonction de la *conférence* et *callDetails* (paramètres de la fonction) qu'elle recevra, elle lancera la méthode associée dépendamment de `state`. Quant à la méthode *changeState*, elle sera en charge de modifier l'état de la variable *state* dépendamment de l'état de l'appel.

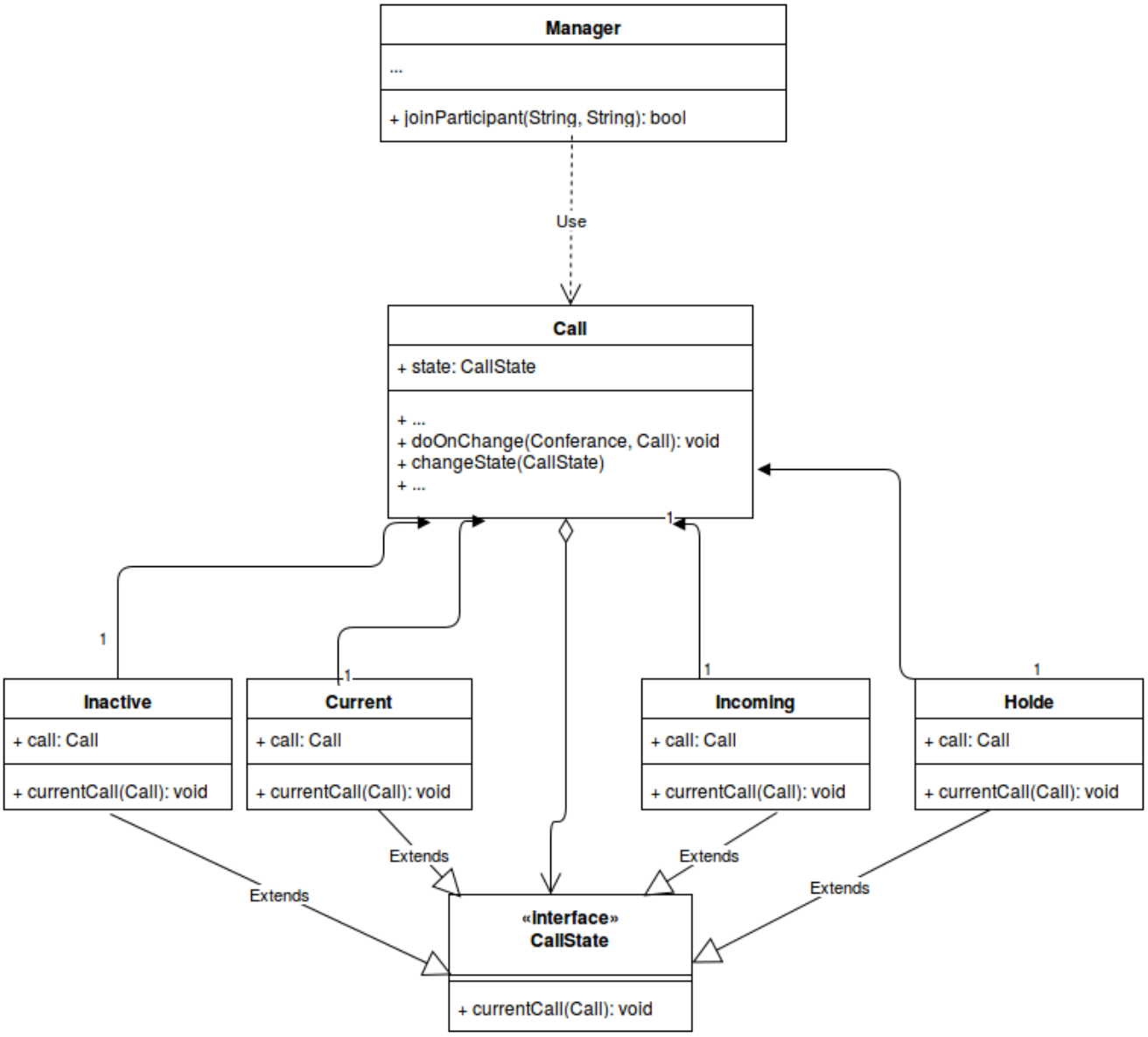

Figure 13: Schéma du patron de conception "*State*" applicable dans Ring

5 Conclusion

References

[1] The arrowhead anti-pattern. `http://wiki.c2.com/?ArrowAntiPattern`. Accessed: 2016-11-01.

[2] Extract method solution. `http://wiki.c2.com/?ExtractMethod`. Accessed: 2016-11-01.

[3] Nombre cyclomatique. `https://fr.wikipedia.org/wiki/Nombre_cyclomatique`. Accessed: 2016-11-01.

[4] Tamás Bécsi and SZ ARADI. Reliability of data transfer and handling in railway telemonitoring systems. In *Proc. of Symposium FORMS/FORMAT*, volume 2008, pages 185–192. Citeseer, 2008.

[5] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976.