Polytechnique Montréal

LOG8430-Architecture logicielle et conception avancée

TP3 - Mise en œuvre d'une architecture logicielle

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

Decembre 2, 2016

Contents

1 Introduction

L'objectif de ce TP est de finaliser notre travail effectué au cours des deux TPs précédents. Nous allons donc implémenter notre solution pour corriger un des problèmes détectés lors du TP précédent. Dans un premier temps, nous rappellerons rapidement les problèmes que nous avions découvert et leurs conséquences. Puis nous expliquerons le choix du problème pour lequel nous avons implémenter notre solution. Ensuite, nous détaillerons notre implémentation en C++, accompagnée de différents tests pour s'assurer que nos modifications n'aient pas d'impact sur le fonctionnement de Ring. En enfin, nous terminerons avec le "pull request" que nous avons fait, pour proposer notre solution aux équipes de SFL travaillant sur Ring.

2 Implémentation en C++

2.1 Problèmes détectés dans Ring

Lors du TP2, nous avions détecté plusieurs problèmes dans le code de Ring.

God class Ce problème concernait la classe Manager. Elle pouvait pouvait poser des problèmes de maintenance, d'extensibilité, d'évolutivité, de tests ou encore des difficultés de compréhension du code. Cependant, modifier cette classe qui est réellement le cœur du logiciel ne nous semblait pas raisonnable.

Longues méthodes Ces dernière consistaient le second problème:

- *void Manager::Init(...)*,
- *void RingAccount::doRegister()*
- *bool RingAccount::SIPStartCall(...)*

Ces méthodes trop longues rendaient la compréhension et la maintenance du code plus difficile. Cependant, notre solution à ce problème était simplement de découper les longues méthodes en plusieurs sous-fonctions plus petites et ayant chacune un rôle particulier. Cela demandait très peu de travail, et semblait être un peu trop simple pour le TP3. Nous n'avons donc pas choisi d'implémenter notre solution à ce problème.

ArrowHead Anti Pattern Nous avions identifié cet anti-patron au niveau de la méthode *std::string Call::getStateStr() const* (fichier `dameon/src/call.cpp`). Cette dernière présentait graphiquement une forme de "flèche" en raison du trop grand nombre de structures conditionnelles imbriquées. Au niveau des métriques, cela reflète une complexité cyclomatique trop importante. Voici le code de cette fonction.

```
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 1: bloc `switch/case` de la méthode `getStateStr`

A nouveau cet anti-patron rendait la compréhension et la maintenance du code plus compliquée. De plus, les études montrent une corrélation entre la complexité cyclomatique d'un programme et la fréquence d'erreur. Un programme ayant une faible

complexité cyclomatique est plus compréhensible et peut être modifié à moindre risque. Notre solution consistait à diviser cette fonction de manière à réduire la complexité cyclomatique. Voici deux diagrammes d'activité présentant l'état initial de la méthode et celui après l'application de notre solution.

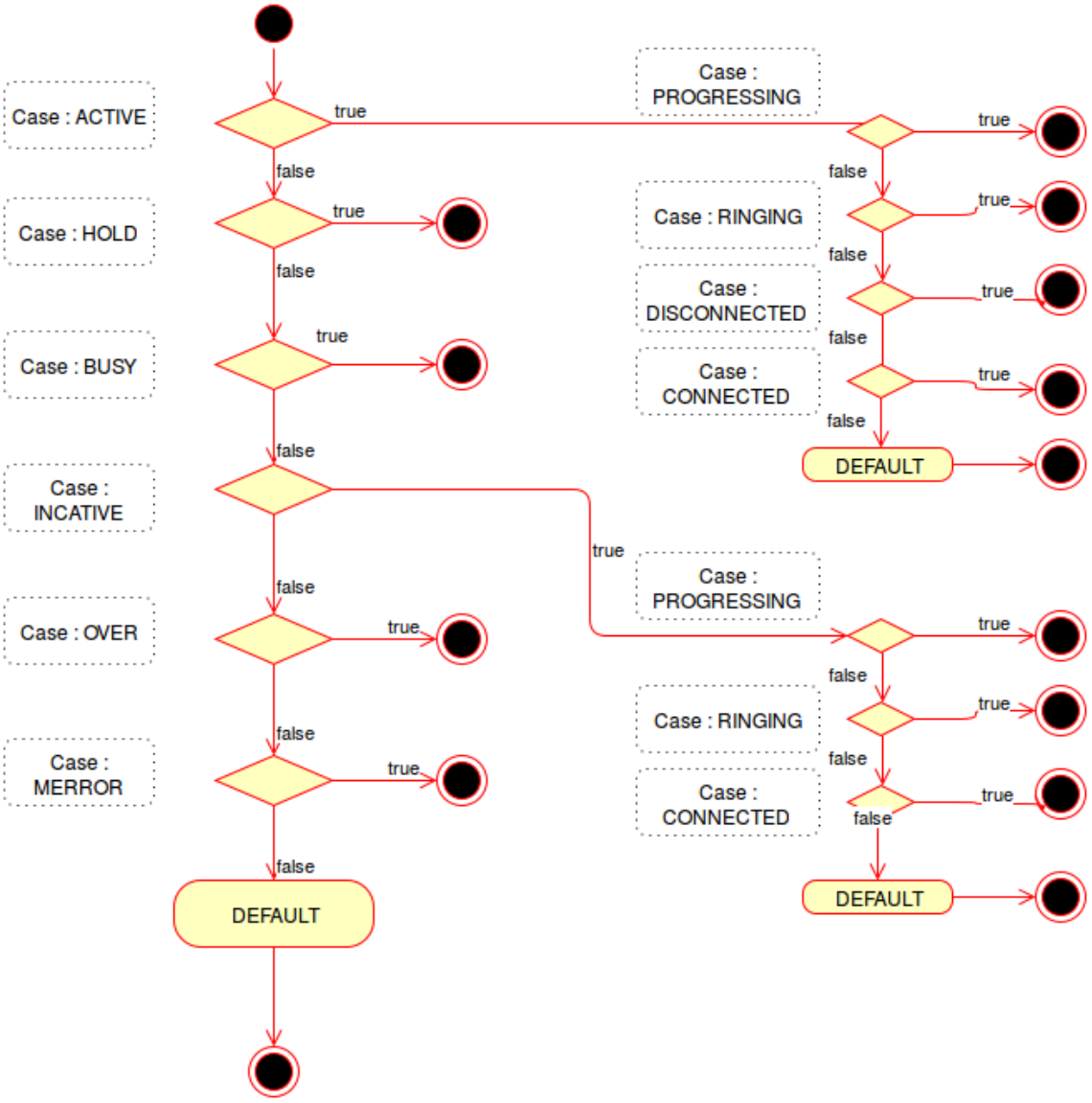


Figure 2: Diagramme d'activé de la méthode `getStateStr` avant re-factoring

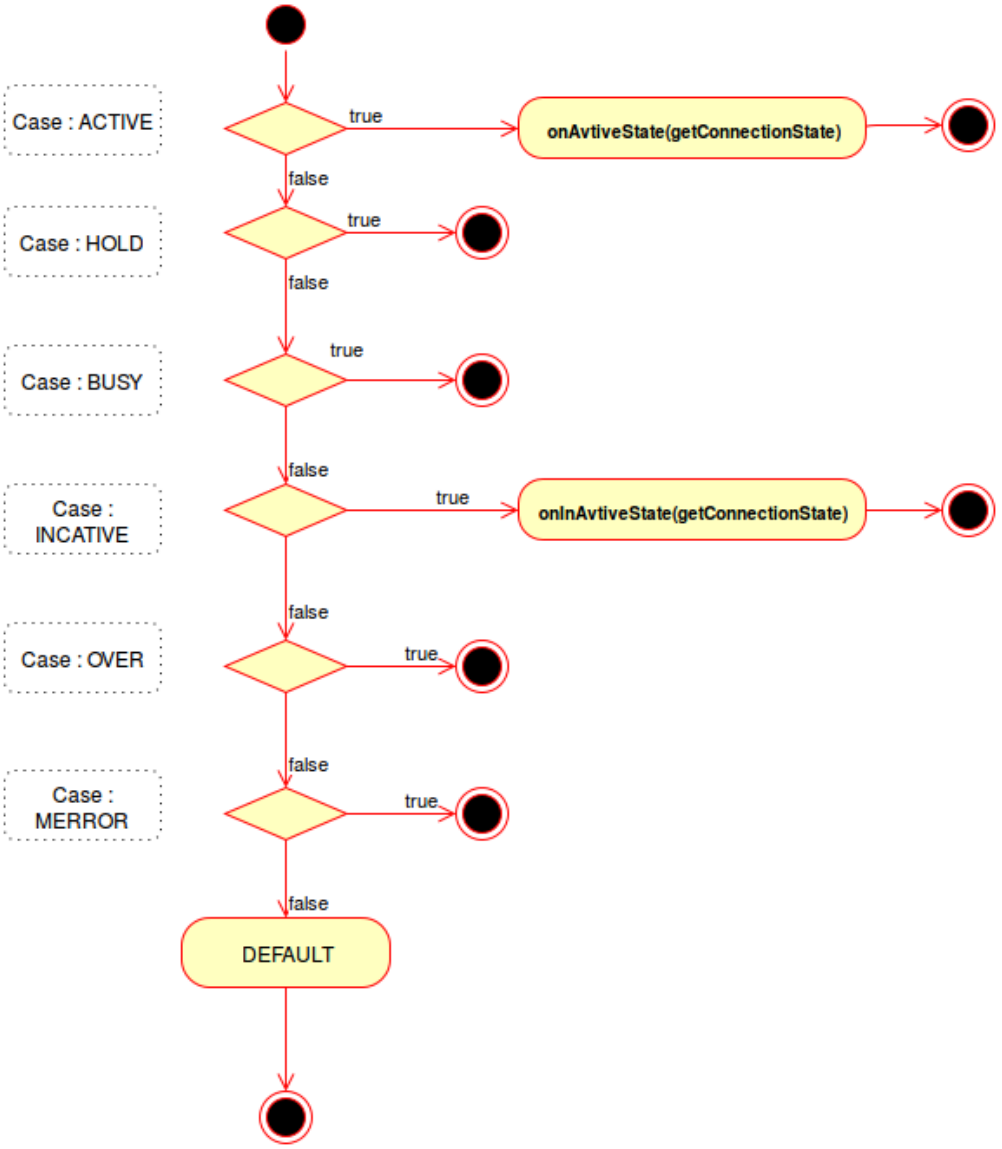


Figure 3: Diagramme d'activité de la méthode `getStateStr` après re-factoring

Notre solution semblait relativement simple comme pour les longues méthodes. Cependant, nous l'avons implémentée car elle était directement liée à notre vraie implémentation qui concernait le quatrième problème détecté.

Duplication de code En effet, nous avons décidé d'implémenter notre solution au problème de duplication de code présent au niveau de la méthode *bool Manager::joinParticipant(...)*, du fichier `daemon/src/manager.cpp`.

La duplication de code (connue 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). 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 est 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.

Dans le TP2, nous avions proposé deux solutions. Une première très simple, qui était de diviser la méthode *joinParticipant(...)* et d'isoler les blocs conditionnels dans des méthodes séparées. Cette solutions avait un apport minime pour le programme et ne permettait pas d'appliquer d'outils de conception logicielle. Au contraire notre seconde solution (celle choisie) consistait en l'utilisation du patron de conception **State** (ou **Etat**). L'application de ce patron va réellement améliorer la qualité du code tout en résolvant le problème de duplication de code. En implémentant cette solution, nous avons remarqué que nous allions "toucher" à la méthode *std::string Call::getStateStr() const* qui était l'origine du troisième problème identifié : "ArrowHead Anti-Pattern". Ainsi, comme dit précédemment, nous avons implémenté une solution globale pour les deux problèmes : ArrowHead et duplication de code.

2.2 Description de la solution envisagée

Nous avons donc choisi d'implémenter une solution qui répondra à deux des problèmes détectés. Notre solution consiste à implémenter le patron de conception **State** ainsi que de diviser et isoler certains blocs conditionnels de la méthode *Call::getStateStr()*.

Notre solution concerne les quatre fichiers suivant :

- manager.h
- manager.cpp
- call.h
- call.cpp

Fichiers *manager.h* et *manager.cpp* Au niveau de ces fichiers, nous avons ajouté une méthode *CallState_* Manager::getCallState(const std::string &callID)* qui va reprendre l'objectif de la méthode *Manager::getCallDetails(...)* mais de

manière à être adaptée au patron State. Elle pourrait par la suite remplacer la fonction *Manager::getCallDetails*, mais comme cette dernière est utilisée à de nombreux endroits et que nous souhaitions nous concentrer uniquement sur la fonction *joinParticipant()*, nous avons conservé la fonction *Manager::getCallDetails* et simplement rajouté notre propre fonction.

Nous avons légèrement modifié le code de *Manager::joinParticipant(...)*, afin de réduire la complexité cyclomatique et de supprimer la duplication de code qui était présente au niveau des blocs conditionnels en fonction de l'état des call1 et call2. Le comportement global de la méthode a été conservé.

Fichiers *call.h* et *call.cpp* Concernant ces fichiers, nous avons effectué plus de modifications et notamment le patron State. Tout d'abord, nous avons du créer une nouvelle classe **CallState_**. Cette dernière est abstraite et plusieurs autres classes concrètes vont en dériver afin de représenter les différents états des appels. Ces classes constitueront le cœur du parton State. La déclaration des classes se fait dans le fichier "call.h" et l'implémentation dans le fichier "call.cpp". Nous aurions souhaité créer deux nouveaux fichiers (un .cpp et un .h) qui contiendraient l'ensemble de ces classes. Cependant, même en ajoutant ces fichiers dans le Makefile, nous ne sommes pas parvenu à recompiler le projet. Nous pensons qu'il y a peut-être d'autres étapes à faire afin d'ajouter un nouveau fichier au projet. C'est pour cela, que nous avons mis les nouvelles classes directement dans les fichiers "call.h" et "call.cpp". De cette manière la compilation ne pose pas de problème.

Dans la classe Call, nous avons ajouté une méthode *Call::getCallState()* qui aura le même rôle que la fonction *Call::getDetails()* mais sera adaptée à notre implémentation du patron State. A nouveau nous avons préféré conserver la fonction *Call::getDetails()* et ajouter notre propre fonction car *Call::getDetails()* est utilisée à de nombreux endroits et nous voulions limiter l'impact de nos modifications et ne pas avoir à faire des modifications partout où *Call::getDetails()* était appelée. Enfin, nous avons adapté la méthode *Call::getStateStr()* afin de faire les bons appels pour mettre en place le patron State. Cette fonction va désormais renvoyer des *pair* contenant les "string" de départ (avant nos modifications) et des pointeurs sur nos objets représentant les états des Call (CallState_*). Dans ce cas, nous avons donc simplement modifié le retour de la fonction en utilisant des *pair* et ainsi il suffit de faire un appel `valeur_retour.first` pour récupéré la "string" et `valeur_retour.second` pour accéder à nos objets CallState_. Par ailleurs, comme dit précédemment, cette fonction présentait l'anti-patron "ArrowHead". Dans notre implémentation, nous l'avons également résolu en isolant certains blocs conditionnels et en ajoutant deux autres méthodes *onInactiveState()* et *onActiveState()*. De cette façon nous avons pu réduire la complexité cyclomatique de la fonction getStateStr() tout en faisant

disparaître la forme de flèche qui avait tendance à favoriser des erreurs.

De façon générale, nous avons tenté de ne pas faire trop de modifications afin de s'assurer que le logiciel fonctionnera encore correctement après notre refactoring. Nous avons simplement voulu appliquer le design pattern State dans le cas de la fonction *Manager::joinParticipant()* afin de l'améliorer. Pendant l'implémentation, nous avons pu remarquer que des CallStates étaient déjà présents sous la forme de "enum class" et d'un membre "callstate_" de la classe Call. De plus, les états des Call étaient utilisés à de très nombreux endroits. Ainsi, nous avons essayé d'implémenter notre solution tout en conservant au maximum les éléments existants pour d'autres fonctionnalités. Par la suite, nous pensons qu'il serait possible de réutiliser nos nouvelles classes **CallState_** dans l'ensemble du daemon, pour les états des appels. Les "enum class" actuelles pourraient alors être supprimées. Cependant, cela demande une connaissance de l'architecture, du code et du fonctionnement du daemon bien plus poussée que la notre ainsi qu'une certaine charge de travail.

3 Intégration et testing

3.1 Testing

Après avoir implémenté notre solution et recompilé le daemon Ring avec nos modifications, nous sommes passé à la phase de tests. Pour cela, nous avons imaginé trois cas de test. La solution que nous avons implémentée concernait le mode "conférence" de Ring, c'est à dire quand il y a plus que deux participants à une conversation vidéo. En effet, la méthode *Manager::joinParticipant()* n'est appelée que lors de l'ajout de participants à un conférence. Cependant, afin de nous assurer que les autres fonctionnalités de Ring fonctionnaient encore, nous avons également pris en compte des tests d'appel simple entre deux personnes et d'envoi de messages.

Appel simple Ring fonctionnait bien pour un appel vidéo simple entre deux personnes.

Figure 4: Appel simple

Message De même, il était encore possible d'envoyer un message à une autre personne.

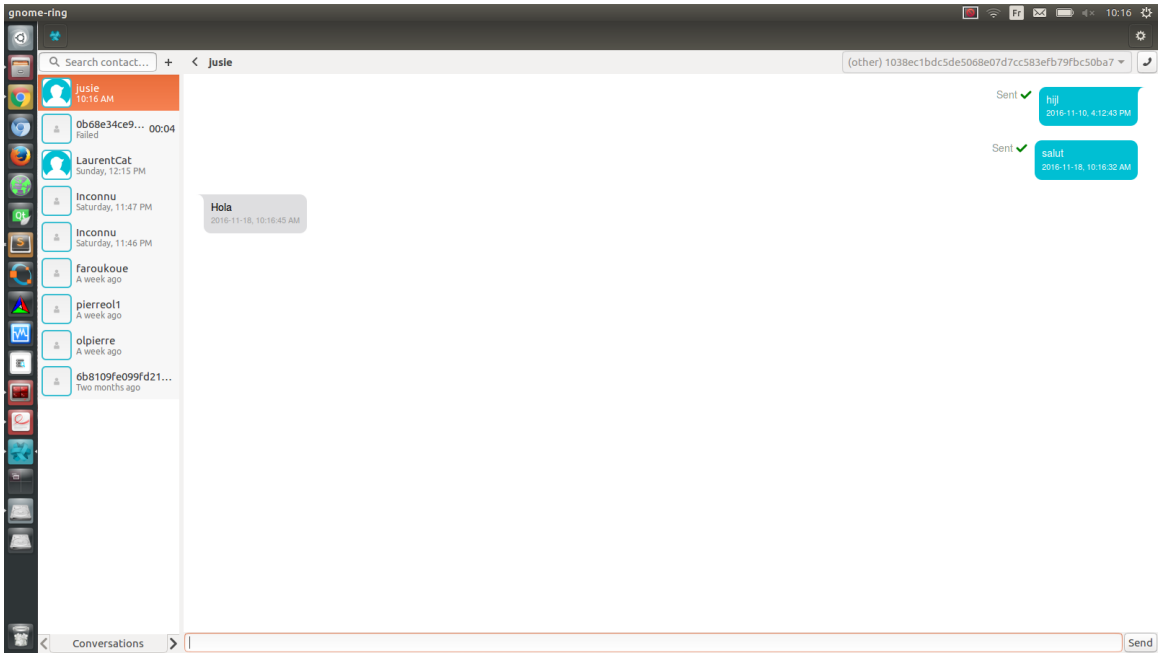


Figure 5: Messages

Conférence Le cas le plus important et plus intéressant était le mode de conférence. Le fait de pouvoir créer une conférence en ajoutant un participant à un appel existant entre deux personnes, n'est disponible qu'à partir du client Windows. Cela a rendu la phase de test un peu plus complexe, puisque le daemon avait été recompilé pour Linux. Nous avons fait le test suivant.
User_A (client Windows) appelle User_B (Linux daemon modifié) et User_C (smartphone Android dans notre cas). A ce moment là, User_A gère deux appels simultanément, mais l'un des deux est toujours mis en attente. Puis à partir de l'appel avec le User_C, User_A va ajouter le participant User_B. De cette manière, une conférence va être créée à partir des deux appels simples. On peut bien remarquer que 3 écrans apparaissent à l'écran (User_A, User_B et User_C). Comme la conférence était créée à partir du client Windows (daemon non modifié) et que notre daemon modifié tournait sous Linux, nous n'étions pas totalement sûr que les fonctions modifiées étaient bien appelées. Cependant, nous avons refait le même test, en changeant complètement la fonction *Manager::joinParticipant()*. Nous lui avons fait retourner directement le booléen *false*, sans faire aucun traitement. Ensuite, lors du test, il était impossible de créer la conférence à partir du client Windows. Cela montrait donc bien que même si la création de la conférence n'est pas faite à partir du client Linux (ayant le daemon modifié), les fonctions que nous avons modifiées sont malgré tout appelées. Si la méthode *Manager::joinParticipant()* est appelée, alors tout nos

autres modifications seront également utilisées. C'est pourquoi nous nous sommes surtout concentré sur le fait que *Manager::joinParticipant()* soit appelée.

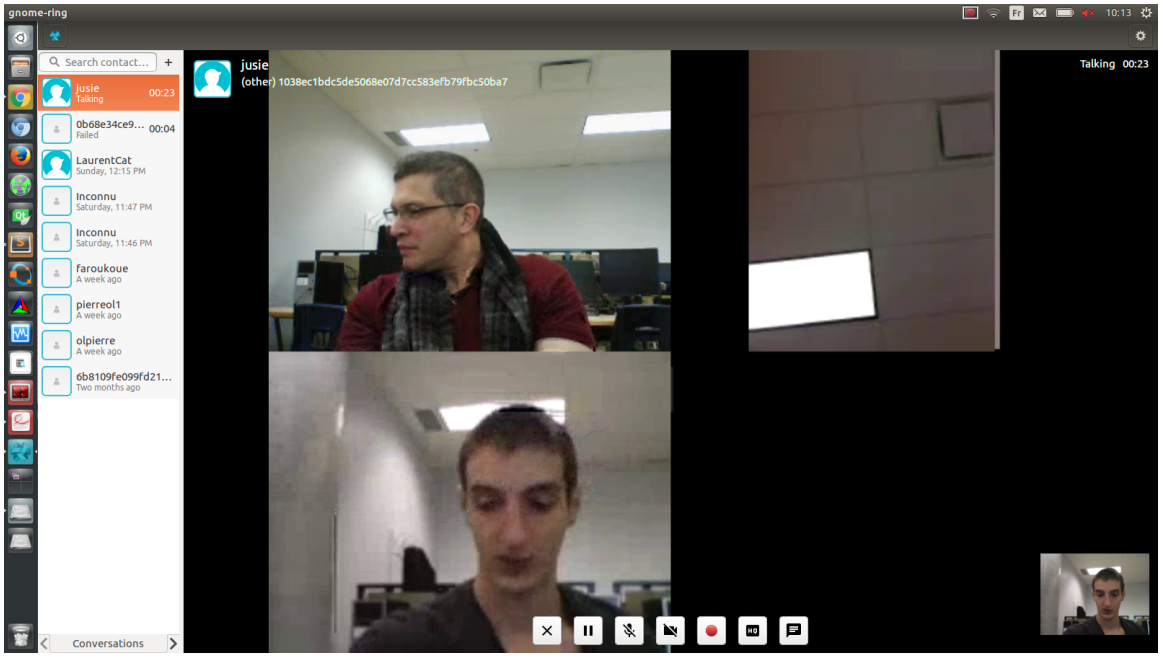


Figure 6: Conférence à 3

3.2 Métriques

Le problème initial concernait la métrique complexité cyclomatique qui était trop importante ainsi qu'une duplication de code. Notre implémentation a permis de réduire la complexité cyclomatique des méthodes *Manager::joinParticipant()* et *Call::getStateStr()*. Pour la première méthode, la complexité cyclomatique a été réduite de 8 puisque l'on a "transféré" les deux blocs conditionnels de taille 4 vers notre patron **State**. La complexité cyclomatique de la seconde méthode est passée de 17 à 9. Une complexité de 9 est bonne puisque d'après les travaux de Thomas McCabe [1],la limite acceptable est de 10. Notre implémentation a également un impact sur les métriques "nombre de lignes de code" et "nombre de classes". Il y a plus de classes ce qui pourrait rendre le code plus compliqué, mais par la suite, elles pourront remplacer d'autres éléments comme l'enum class CallState.

3.3 Améliorations apportées

Globalement hormis la réduction de la complexité cyclomatique, les métriques ne sont pas réellement significatives pour notre implémentation. Le point essentiel est d'avoir pu éliminer la duplication de code et d'avoir désormais un code plus simple, compréhensible et facilement extensible grâce au patron **State**. Par exemple, l'ajout de nouveaux états pour les Call se fera très facilement.

4 Pull Request

Pour faire la pull request, nous avons suivi les instructions de cette page : `https://tuleap.ring.cx/plugins/mediawiki/wiki/ring/index.php/Working_with_Gerrit`

Comme la pull request pour Ring doit se faire sur Gerrit, nous avons lié un de nos compte Github avec Gerrit. Nous avons fait un clone du dépôt "ring-daemon" de gerrit de l'utilisateur zinspierre.
git clone https://zinspierre@gerrit-ring.savoirfairelinux.com/ring-daemon
comme indiqué, nous avons ajouté les clés SSH pour l'utilisateur et configuré Git afin d'avoir des *"change-id"*. Enfin, nous avons effectué le commit de nos modifications en expliquant notre solution dans le message du commit. Enfin, nous avons effectué le push.
Voici le lien de notre pull request : `https://gerrit-ring.savoirfairelinux.com/#/c/5651/`

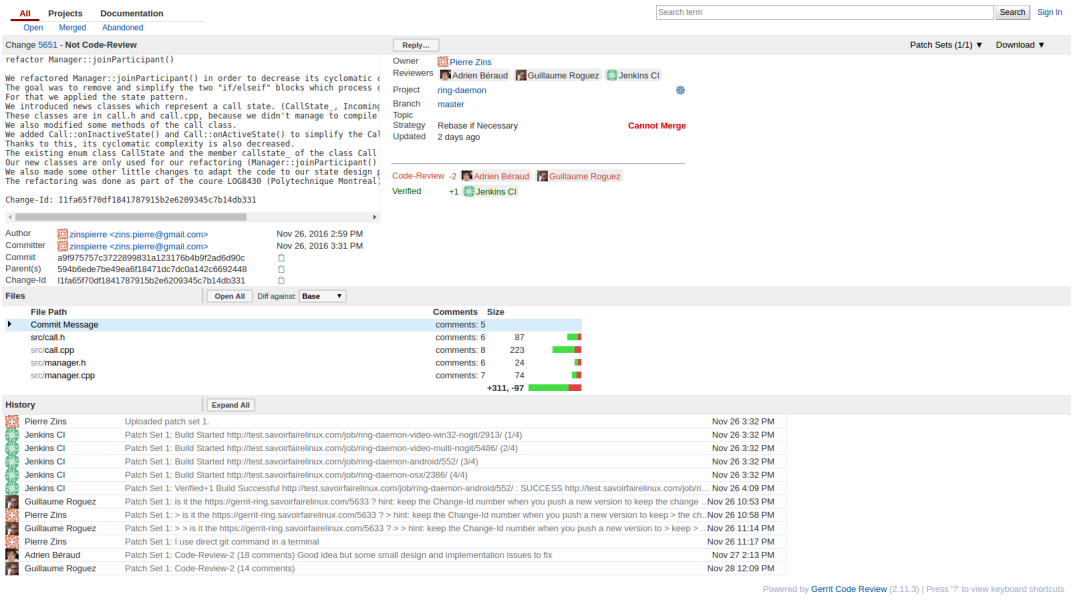


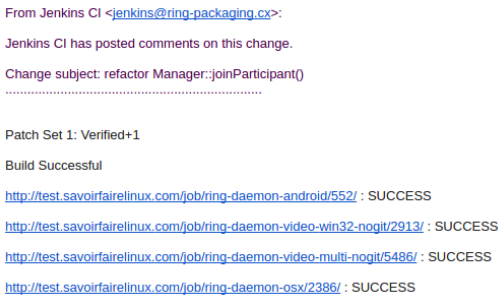
Figure 7: Pull Request Gerrit

Après avoir fait notre Pull Request, le système d'intégration continue ("Jenkins") va vérifier notre code. Dans notre cas, l'ensemble a bien fonctionné (SUCCESS partout) :

History | Expand All

Pierre Zins	Uploaded patch set 1.	Nov 26 3:32 PM
Jenkins CI	Patch Set 1: Build Started http://test.savoirfairelinux.com/job/ring-daemon-video-win32-nogit/2913/ (1/4)	Nov 26 3:32 PM
Jenkins CI	Patch Set 1: Build Started http://test.savoirfairelinux.com/job/ring-daemon-video-multi-nogit/5486/ (2/4)	Nov 26 3:32 PM
Jenkins CI	Patch Set 1: Build Started http://test.savoirfairelinux.com/job/ring-daemon-android/552/ (3/4)	Nov 26 3:32 PM
Jenkins CI	Patch Set 1: Build Started http://test.savoirfairelinux.com/job/ring-daemon-osx/2386/ (4/4)	Nov 26 3:32 PM
Jenkins CI	Patch Set 1: Verified+1 Build Successful http://test.savoirfairelinux.com/job/ring-daemon-android/552/ : SUCCESS http://test.savoirfairelinux.com/job/ri...	Nov 26 4:09 PM
Guillaume Roguez	Patch Set 1: is it the https://gerrit-ring.savoirfairelinux.com/5633 ? hint: keep the Change-Id number when you push a new version to keep the change ...	Nov 26 10:53 PM
Pierre Zins	Patch Set 1: > is it the https://gerrit-ring.savoirfairelinux.com/5633 ? > hint: keep the Change-Id number when you push a new version to keep > the ch...	Nov 26 10:58 PM
Guillaume Roguez	Patch Set 1: > > is it the https://gerrit-ring.savoirfairelinux.com/5633 ? > > hint: keep the Change-Id number when you push a new version to > keep > ...	Nov 26 11:14 PM
Pierre Zins	Patch Set 1: I use direct git command in a terminal	Nov 26 11:17 PM
Adrien Béraud	Patch Set 1: Code-Review-2 (18 comments) Good idea but some small design and implementation issues to fix	Nov 27 2:13 PM
Guillaume Roguez	Patch Set 1: Code-Review-2 (14 comments)	Nov 28 12:09 PM

Figure 8: Jenkins



From Jenkins CI <jenkins@ring-packaging.cx>:

Jenkins CI has posted comments on this change.

Change subject: refactor Manager::joinParticipant()

Patch Set 1: Verified+1

Build Successful

http://test.savoirfairelinux.com/job/ring-daemon-android/552/ : SUCCESS

http://test.savoirfairelinux.com/job/ring-daemon-video-win32-nogit/2913/ : SUCCESS

http://test.savoirfairelinux.com/job/ring-daemon-video-multi-nogit/5486/ : SUCCESS

http://test.savoirfairelinux.com/job/ring-daemon-osx/2386/ : SUCCESS

Figure 9: Jenkins mail

Quelques jours plus tard, nous avons eu le retour de deux développeurs de SFL (Adrien Béraud et Guillaume Roguez). Notre pull request n'a pas été acceptée, mais ils sont plutôt d'accord avec notre idée et la trouve intéressante. Ils ont proposé différents conseil, bonnes pratique et corrections à notre implémentation afin qu'elle puisse peut-être être acceptée. Voici une liste de quelques remarques :

- une duplication de constante avec un autre élément StateEvent
- ajout de lignes vides inutiles
- nom de la méthode getStateStr() suggère de retourner une string or nous retournons une "pair". Il faudrait donc changer le retour ou le nom de la méthode
- Utilisation des pointeurs intelligents (unique_ptr ou shared_ptr) plutôt que les raw pointers.
- coding rules
- ...

Ainsi, il faudrait apporter quelques modifications et corrections à notre implémentation. cependant, nous n'avons pas pu le faire par manque de temps.

5 Conclusion

Au travers de ce TP3, nous pu implémenter notre solution et proposer une contribution au projet open source Ring. Ce TP vient conclure le travail que nous avions effectué lors des deux TPs précédents. Ces derniers étaient essentiels afin de bien préparer notre solution. De cette manière, l'implémentation faite lors de ce TP a été plus facile et rapide. Nous avons pu tester nos modification, en recompilant le projet ring et en testant trois cas d'utilisation. Ainsi, notre refactoring n'a pas ajouté de nouvelle fonctionnalité mais a permis de supprimer une duplication de code et de réduire la complexité cyclomatique d'une fonction tout en appliquant le patron de conception "state". Le code semble donc plus compréhensible et plus adapté à une évolution. Malgré tout, notre pull request n'a pas été acceptée, et notre implémentation nécessiterait quelques corrections afin d'être acceptée et intégrée à Ring.

References

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

6 Annexes

Voici le code source de l'implémentation de notre solution. 4 fichiers de ring-daemon ont été modifiés. `src/manager.h`, `src/manager.cpp`, `src/call.h` et `src/call.cpp`. La majorité des changements sont présents ci-dessous, seuls quelques *includes* et *forward-declarations* nécessaires à certains endroits pour la compilation, n'apparaissent pas.

Listing 1: manager.h

```
1  //...
2  /** Manager (controller) of Ring daemon */
3  class Manager {
4      private:
5          std::unique_ptr<PluginManager> pluginManager_;
6
7      public:
8
9          //new
10         CallState_* getCallState(const std::string& callID);
11 //...
```

Listing 2: manager.cpp

```
1  //...
2  //new
3  //fonction ayant le meme comportement que getCallDetails()
4  //mais qui est adaptee a notre implementation
5  CallState_*
6  Manager::getCallState(const std::string &callID)
7  {
8      if (auto call = getCallFromCallID(callID)) {
9          return call->getCallState();
10     }
11     else {
12         RING_ERR("Call is NULL");
13         return 0;
14     }
15 }
16
17 bool
18 Manager::joinParticipant(const std::string& callId1,
```

```
19                                  const std::string& callId2)
20 {
21     if (callId1 == callId2) {
22         RING_ERR("Cannot join participant %s to itself", callId1.c_str())
23         return false;
24     }
25
26     // Set corresponding conference ids for call 1
27     auto call1 = getCallFromCallID(callId1);
28     if (!call1) {
29         RING_ERR("Could not find call %s", callId1.c_str());
30         return false;
31     }
32
33     // Set corresponding conderence details
34     auto call2 = getCallFromCallID(callId2);
35     if (!call2) {
36         RING_ERR("Could not find call %s", callId2.c_str());
37         return false;
38     }
39
40     // ensure that calls are only in one conference at a time
41     if (isConferenceParticipant(callId1))
42         detachParticipant(callId1);
43     if (isConferenceParticipant(callId2))
44         detachParticipant(callId2);
45
46
47     //debut des modifications
48     //recuperation des etats des calls 1 et 2
49     CallState_* call1state = getCallState(callId1);
50     CallState_* call2state = getCallState(callId2);
51     //verification que les call ne sont pas nuls
52     if(!call1state || !call2state)
53     {
54         //delete pour eviter les fuites de memoires car nous avons alloue
55         if(call1state)
56             delete call1state;
57         if(call2state)
58             delete call2state;
```

```
59          return false;
60      }
61
62
63      std::string current_call_id(getCurrentCallId());
64      RING_DBG("Current Call ID %s", current_call_id.c_str());
65
66      // detach from the conference and switch to this conference
67      if ((current_call_id != callId1) and (current_call_id != callId2)) {
68          // If currently in a conference
69          if (isConference(current_call_id))
70              detachParticipant(RingBufferPool::DEFAULT_ID);
71          else
72              onHoldCall(current_call_id); // currently in a call
73      }
74
75
76      auto conf = createConference(callId1, callId2);
77
78      call1->setConfId(conf->getConfID());
79      getRingBufferPool().unBindAll(callId1);
80
81      call2->setConfId(conf->getConfID());
82      getRingBufferPool().unBindAll(callId2);
83
84      // Process call1 according to its state
85
86      //configuration de la conference pour callID1
87      call1state->configureConference(conf, callId1, this);
88
89
90
91      // Process call2 according to its state
92
93      //configuration de la conference pour callID2
94      call2state->configureConference(conf, callId2, this);
95
96
97      // Switch current call id to this conference
98      switchCall(conf->getConfID());
```

```
99      conf->setState(Conference::ACTIVE_ATTACHED);
100
101     // set recording sampling rate
102     conf->setRecordingAudioFormat(ringbufferpool_->getInternalAudioFormat
103
104     //delete pour eviter les fuites de memoires car nous avons alloue de
105     delete call1state;
106     delete call2state;
107     return true;
108 }
109 //...
```

Listing 3: call.h

```
1  //...
2
3  //new
4  //declaration des classes representant les etats des calls
5  //pour notre patron State
6  class CallState_ {
7  public:
8      virtual const std::string getState() const =0;
9      virtual ~CallState_(){}
10     virtual void configureConference(std::shared_ptr<Conference>& conf,
11         const std::string& callId, Manager* manager);
12 };
13 class IncomingCall : public CallState_ {
14 public:
15     virtual const std::string getState() const;
16     virtual void configureConference(std::shared_ptr<Conference>& conf,
17         const std::string& callId, Manager* manager);
18 };
19 class HoldCall : public CallState_ {
20 public:
21     virtual const std::string getState() const;
22     virtual void configureConference(std::shared_ptr<Conference>& conf,
23         const std::string& callId, Manager* manager);
24 };
25 class CurrentCall : public CallState_ {
26 public:
27     virtual const std::string getState() const;
```

```
28      virtual void configureConference(std::shared_ptr<Conference>& conf,
29          const std::string& callId, Manager* manager);
30  };
31  class InactiveCall : public CallState_ {
32  public:
33      virtual const std::string getState() const;
34      virtual void configureConference(std::shared_ptr<Conference>& conf,
35          const std::string& callId, Manager* manager);
36  };
37  class ConnectingCall: public CallState_ {
38  public:
39      virtual const std::string getState() const ;
40  };
41  class RingingCall : public CallState_ {
42  public:
43      virtual const std::string getState() const ;
44  };
45  class HungupCall : public CallState_ {
46  public:
47      virtual const std::string getState() const;
48  };
49  class BusyCall : public CallState_ {
50  public:
51      virtual const std::string getState() const ;
52  };
53  class OverCall : public CallState_ {
54  public:
55      virtual const std::string getState() const ;
56  };
57  class FailureCall : public CallState_ {
58  public:
59      virtual const std::string getState() const ;
60  };
61  //...
62  class Call : public Recordable, public std::enable_shared_from_this<Call>
63      public:
64  //...
65          //new
66          std::pair<std::string, CallState_*> onInactiveState(ConnectionSta
67          std::pair<std::string, CallState_*> onActiveState(ConnectionState
```

```
68          CallState_* getCallState();
69  //...
```

Listing 4: call.cpp

```
1   //...
2   //new
3   //definition des methodes des nouvelles classes representant les etats de
4   void CallState_::configureConference(std::shared_ptr<Conference>& conf,
5           const std::string& callId, Manager* manager){
6           RING_WARN("Call state not recognized");
7   }
8   const std::string IncomingCall::getState() const {
9       return "Incoming";
10  }
11  void IncomingCall::configureConference(std::shared_ptr<Conference>& conf,
12      const std::string& callId, Manager* manager){
13      conf->bindParticipant(callId);
14      manager->offHoldCall(callId);
15  }
16  const std::string HoldCall::getState() const {
17      return "Hold";
18  }
19
20  void HoldCall::configureConference(std::shared_ptr<Conference>& conf,
21      const std::string& callId, Manager* manager){
22      conf->bindParticipant(callId);
23      manager->offHoldCall(callId);
24  }
25
26  const std::string CurrentCall::getState() const {
27      return "Current";
28  }
29  void CurrentCall::configureConference(std::shared_ptr<Conference>& conf,
30      const std::string& callId, Manager* manager){
31      conf->bindParticipant(callId);
32  }
33
34  const std::string InactiveCall::getState() const {
35      return "Inactive";
36  }
```

```
37 void InactiveCall::configureConference(std::shared_ptr<Conference>& conf,
38     const std::string& callId, Manager* manager){
39     conf->bindParticipant(callId);
40     manager->offHoldCall(callId);
41 }
42
43 const std::string ConnectingCall::getState() const {
44     return "Connecting";
45 }
46
47 const std::string RingingCall::getState() const {
48     return "Ringing";
49 }
50
51 const std::string HungupCall::getState() const {
52     return "Hungup";
53 }
54
55 const std::string BusyCall::getState() const {
56     return "Busy";
57 }
58 const std::string OverCall::getState() const {
59     return "Over";
60 }
61 const std::string FailureCall::getState() const {
62     return "Failure";
63 }
64
65
66 //new
67 //restructuration de la fonction joinParticipant() en ajoutant deux nouv
68 //onActiveState() et on InactiveState() afin de reduire la complexite cyc
69 //et adaptation des  trois fonctions pour notre patron State
70 std::pair<std::string, CallState_*>
71 Call::onActiveState(Call::ConnectionState connectionState_) const
72 {
73     using namespace DRing::Call;
74     if(connectionState_ == ConnectionState::PROGRESSING)
75     {
76         return std::make_pair(StateEvent::CONNECTING, new ConnectingCall(
```

```
77      }
78      else if(connectionState_ == ConnectionState::RINGING)
79      {
80          if(isIncoming())
81              return std::make_pair(StateEvent::INCOMING, new IncomingCall(
82          else
83              return std::make_pair(StateEvent::RINGING, new RingingCall())
84      }
85      else if(connectionState_ == ConnectionState::DISCONNECTED)
86      {
87          return std::make_pair(StateEvent::HUNGUP, new HungupCall());
88      }
89      else if (connectionState_ == ConnectionState::CONNECTED)
90      {
91          return std::make_pair(StateEvent::CURRENT, new CurrentCall());
92      }
93      else
94      {
95          return std::make_pair(StateEvent::CURRENT, new CurrentCall());
96      }
97  }
98
99  std::pair<std::string, CallState_*>
100 Call::onInactiveState(Call::ConnectionState connectionState_) const
101 {
102     using namespace DRing::Call;
103     if(connectionState_ == ConnectionState::PROGRESSING)
104     {
105         return std::make_pair(StateEvent::CONNECTING, new ConnectingCall(
106     }
107     else if(connectionState_ == ConnectionState::RINGING)
108     {
109         if(isIncoming())
110             return std::make_pair(StateEvent::INCOMING, new IncomingCall(
111         else
112             return std::make_pair(StateEvent::RINGING, new RingingCall())
113     }
114     else if (connectionState_ == ConnectionState::CONNECTED)
115     {
116         return std::make_pair(StateEvent::CURRENT, new CurrentCall());
```

```
117         }
118         else
119         {
120             return std::make_pair(StateEvent::INACTIVE, new InactiveCall());
121         }
122 }
123
124 std::pair<std::string, CallState_*>
125 Call::getStateStr() const
126 {
127     using namespace DRing::Call;
128     Call::CallState callState = getState();
129     Call::ConnectionState connectionState = getConnectionState();
130
131     if(callState == CallState::ACTIVE)
132     {
133         return onActiveState(connectionState);
134     }
135     else if(callState == CallState::HOLD)
136     {
137         if(connectionState == ConnectionState::DISCONNECTED)
138             return std::make_pair(StateEvent::HUNGUP, new HungupCall());
139         return std::make_pair(StateEvent::HOLD, new HoldCall());
140     }
141     else if(callState == CallState::BUSY)
142     {
143         return std::make_pair(StateEvent::BUSY, new BusyCall());
144     }
145     else if (callState == CallState::INACTIVE)
146     {
147         return onInactiveState(connectionState);
148     }
149     else if (callState == CallState::OVER)
150     {
151         return std::make_pair(StateEvent::OVER, new OverCall());
152
153     }
154     else if(callState == CallState::MERROR)
155     {
156         return std::make_pair(StateEvent::FAILURE, new FailureCall());
```

```
157     }
158      return std::make_pair(StateEvent::FAILURE, new FailureCall());
159 }
160
161 //...
162
163 //new
164 //fonction ayant le meme objectif que getDetails() mais qui se concentre
165 //uniquement sur nos besoins : l'etat des Calls
166 //d'ailleurs dans le code initial, une map complete etait renvoyee alors
167 //DRing::Call::Details::CALL_STATE etait reellement utilise
168 CallState_*
169 Call::getCallState()
170 {
171     return getStateStr().second;
172 }
173
174 std::map<std::string, std::string>
175 Call::getDetails() const
176 {
177     return {
178         {DRing::Call::Details::CALL_TYPE,          ring::to_string((unsigne
179         {DRing::Call::Details::PEER_NUMBER,        peerNumber_},
180         {DRing::Call::Details::DISPLAY_NAME,       peerDisplayName_},
181         {DRing::Call::Details::CALL_STATE,         getStateStr().first},
182         {DRing::Call::Details::CONF_ID,            confID_},
183         {DRing::Call::Details::TIMESTAMP_START,    ring::to_string(timestar
184         {DRing::Call::Details::ACCOUNTID,          getAccountId()},
185         {DRing::Call::Details::AUDIO_MUTED,        std::string(bool_to_str(
186         {DRing::Call::Details::VIDEO_MUTED,        std::string(bool_to_str(
187     };
188 }
189 //...
```