MASTER EN SCIENCES INFORMATIQUES

Les architectures des logiciels académiques: Une étude de leur spécificité

Meert, Bastien

Award date:
2016

Awarding institution:
Universite de Namur

Université de Namur
Faculté d'informatique
Année académique 2015–2016

Les architectures des logiciels académiques

Une étude de leur spécificité

Bastien Meert

Maître de stage : Dr. Alexander Serebrenik

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Dr. Wim Vanhoof

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Résumé

Malgré une littérature prolifique au sujet des architectures logicielles, il est difficile de trouver des consignes pratiques quant à la conception d'architectures de logiciels académiques. Partant de ce constat, nous pensons qu'il serait très intéressant de produire un guide d'ingénierie logicielle destiné au monde académique. Ce mémoire effectue un premier pas dans cette direction en étudiant la spécificité des logiciels académiques au travers d'un petit nombre d'outils d'analyse et de transformation de programmes développés dans le cadre de la recherche scientifique. A l'aide d'une méthode de récolte de données hybride combinant revue systématique de littérature, interviews et enquête, nous avons pu confirmer certaines spécificités du logiciel académique et identifier les tendances générales au niveau des architectures des logiciels étudiés. Cette base nous semble suffisante pour confirmer l'utilité d'un guide d'ingénierie des logiciels académiques et ouvre la porte à de nombreux travaux futurs.

Mots clés : architecture logicielle, logiciel académique, logiciel de recherche, analyse et transformation de programmes

Abstract

Despite the great amount of publications about software architectures, is is not easy to find concrete tips and advices regarding academic software architecture design. Against this background, we think it would be very interesting to produce an engineering guide dedicated to the academic world. This master thesis takes the first step toward this goal by studying academic software specificity through a few software analysis and transformation projects drawn from the research context. Using a hybrid data collection method which combines systematic literature review, interviews and survey, we were able to confirm a few specificities of academic softwares. We were also able to identify general trends regarding those project architectures. This basis appears sufficient enough to demonstrate the usefulness of an academic software engineering guide and open up many research possibilities.

Key-words : software architecture, academic software, research software, program analysis & transformation

Remerciements

Je tiens à remercier toutes les personnes ayant contribué à la rédaction de ce mémoire.

Je commence par remercier le Dr. Wim Vanhoof pour son soutien et les précieux conseils qu'il m'a donnés tout au long de la rédaction de ce travail.

Je voulais remercier ensuite le Dr. Alexander Serebrenik pour avoir encadré mon stage et pour avoir guidé ma démarche scientifique tout au long de celui-ci. Je le remercie également pour ses retours détaillés lors des différentes étapes de la rédaction.

Je remercie également le Pr. Mark van den Brand pour avoir participé à la supervision de mon stage et m'avoir donné de nombreux conseils qui ont su recadrer et améliorer mon travail.

Je n'oublierai pas de remercier les participants aux enquêtes et aux interviews sans qui je n'aurais pu récolter les données nécessaires à mon travail. Je les remercie eux, ainsi que les doctorants du département de mathématique et d'informatique de l'Université Technique d'Eindhoven pour les discussions qui m'ont beaucoup appris sur mon sujet de recherche ainsi que sur d'autres pans de l'ingénierie logicielle.

Je tenais finalement à remercier mes amis et proches pour leurs nombreuses relectures et conseils qui m'ont permis de peaufiner mon travail. Je les remercie également pour leur patience et leur soutien inconditionnel.

Table des matières

Introduction

Cela fait maintenant plusieurs décennies que l'ingénierie logicielle est étudiée en long et en large et qu'elle est le sujet de nombreuses publications. Nous pouvons d'ailleurs trouver des traces de conférences tenues à son sujet dès la fin des années 60 (Naur et Randell, 1969) pour aboutir aujourd'hui à des ouvrages exhaustifs traitant de la conception des architectures logicielles (Bass *et al.*, 2012; Spinellis et Gousios, 2009). Malgré la pluralité des publications disponibles, nous éprouvons des difficultés lorsque nous souhaitons en savoir plus sur la conception d'architectures dans le contexte académique. C'est en partant de ce constat que nous avons décidé de mener la présente étude. Le sujet est vaste et les objectifs variés allant de l'étude de la spécificité du développement de logiciels académiques à la proposition de lignes directrices et de conseils pratiques pour l'élaboration de leurs architectures. C'est l'idée de baser un guide technique sur ces conseils qui nous a motivé et poussé à nous lancer dans l'étude de ce sujet. Devant l'importance de la tâche, nous nous limitons dans ce travail à mieux comprendre la spécificité de ces logiciels académiques dans le but de trouver des solutions qui pourraient aider à leur conception. Dans cette optique, nous étudierons des logiciels d'analyse et de transformation de programmes.

Dans un premier temps, nous nous attardons sur le sujet et le contexte de ce mémoire. En effet, le peu de travaux pré-existants dans le domaine nous force à définir nous-même ce que nous entendons par logiciels académiques et ce que nous souhaitons étudier à leur propos. Nous poursuivons par une analyse du contexte afin d'envisager les différentes approches du sujet, confirmer la nécessité de notre démarche et nous assurer de l'originalité de celle-ci. Enfin nous entrons dans le vif du sujet en abordant notre démarche scientifique. Celle-ci divisée en trois étapes comprend une revue systématique de littérature, une série d'interviews et une enquête, le recoupement de ces trois sources devant nous mener à une vision plus claire des spécificités des architectures des logiciels académiques et des pistes à explorer pour aider leurs concepteurs. Avant de conclure, nous présentons une synthèse et notre interprétation des résultats. Dans la conclusion, nous tachons de prendre du recul vis-à-vis de notre protocole de recherche et revenons sur l'ensemble de cette étude dans une rétrospective qui précède notre vision des travaux futurs.

Chapitre 1

Présentation du sujet

Dans ce chapitre, nous cherchons à donner les clés qui permettent d'appréhender au mieux le sujet étudié. Nous décrivons ainsi notre sujet et nos motivations et définissons clairement la notion de logiciels académiques avant de présenter ce que nous souhaitons étudier au sujet de leur architecture. Nous terminons ce chapitre par une présentation de nos questions de recherche et de nos objectifs concrets.

1.1 Description générale et motivations

Dans le cadre de cette recherche, nous avons décidé d'étudier les architectures des logiciels académiques. En particulier, nous souhaitons en apprendre plus sur l'ingénierie logicielle dans le contexte académique. Nous espérons ainsi identifier les bonnes pratiques et les erreurs communes dans le cadre de la conception de l'architecture d'un logiciel académique. De cette manière, nous pourrons conseiller le chercheur qui est confronté à la nécessité de développer un outil supportant ses recherches sur le long terme.

Il nous est apparu que le moyen le plus efficace pour transmettre des conseils aux architectes logiciels était l'élaboration d'un guide d'ingénierie logicielle spécifique au contexte académique. Mais avant de se lancer dans la réalisation d'un tel guide, il était important de se poser la question de la pertinence d'une telle réalisation. La nécessité de ce guide apparait alors lorsque nous étudions les spécificités du contexte académique. En effet, nous retrouvons des problématiques semblables à celles rencontrées dans les autres domaines comme le renouvellement des programmeurs, la gestion d'une équipe diversifiée voire d'une communauté. Mais ce qui diffère le plus par rapport au développement logiciel usuel, ce sont les motivations des développeurs et leur expertise. En effet, l'architecte du système ne sera pas forcément informaticien et l'objectif final du projet n'est que très rarement la production du logiciel, celui-ci joue réellement le rôle de support.

Mais la nécessité d'un guide ne suffit pas à confirmer la pertinence de notre recherche. Nous devons également confronter ce projet aux publications existantes concernant la production de logiciels académiques. C'est ici que notre vision de ce guide technique entre en jeu. En effet, nous considérons qu'un guide doit remplir deux objectifs. Il doit donner des lignes directrices généralistes et les attributs logiciels les plus importants pour un contexte académique. Mais il doit également proposer un ensemble d'alternatives concrètes à des problématiques récurrentes ainsi que des contre-exemples. Et il semblerait que ce soit ce deuxième point qui soit le moins souvent pris en compte dans les publications existantes. Ceci pré-suppose que l'architecte du logiciel académique ait une connaissance déjà poussée en architecture logicielle ce qui n'est malheureusement pas toujours le cas. De plus, nos discussions avec des architectes logiciels expérimentés travaillant dans le contexte académique nous ont également appris que les solutions classiques présentées dans les livres d'ingénierie logicielle ne fonctionnent pas toujours.

Il nous est donc désormais possible de cibler nos objectifs généraux sous la forme de la production d'un guide d'ingénierie logicielle spécifique au contexte académique. Nous avons rapidement pris conscience que la production d'un tel guide représente un travail colossal. De plus, nous devons prendre en compte le travail existant. Nous avons donc défini des objectifs concrets qui permettront de constituer un point de départ à la production de recommandations concrètes qui couplées à la littérature existante permettront à terme de produire un guide complet.

Avant de spécifier nos objectifs, il nous reste à donner nos définitions de logiciels académiques et des éléments architecturaux que nous souhaitons étudier. Nous abordons ces deux points dans les deux sections suivantes avant de conclure ce chapitre sur nos objectifs concrets et nos questions de recherche.

1.2 Les logiciels académiques

Nous commençons par définir ce que nous appelons des logiciels académiques. Lorsque nous regardons dans la littérature, nous ne trouvons pas de définitions précises du logiciel académique. Ces termes désignent aussi bien des logiciels de recherche et des preuves de concept que des logiciels réalisés dans un but d'apprentissage. De plus, les logiciels académiques peuvent être à la fois des logiciels de recherche et des travaux pratiques liés à l'apprentissage. Devant cette absence de définition stricte, nous avons élaboré notre propre définition des logiciels académiques :

Les logiciels académiques sont des logiciels développés dans un contexte académique à des fins de recherche, ces logiciels doivent être des outils ou des sujets de recherche.

De cette manière, nous excluons d'une part, les projets de recherche menés dans un contexte industriel et d'autre part, les logiciels *jetables* développés uniquement en tant que projet étudiant. Mais un deuxième élément nous paraissait primordial quant à l'étude des logiciels académiques : leur longévité. En effet, alors que nous cherchons à étudier les architectures logicielles, il nous parait peu pertinent d'étudier les architectures de projets à court terme ou à usage unique comme les preuves de concept. C'est pourquoi nous avons concentré notre étude sur des projets ayant eu une activité sur plusieurs années. Cela nous donne deux avantages très intéressants. D'une part, ces projets ont souvent fortement évolué au fil des années, ils sont par là-même plus généreux en informations concernant leur architecture et il est plus facile de comprendre les choix qui les ont amenés à leur version actuelle. D'autre part, les projets matures se prêtent mieux à l'évaluation de la satisfaction des exigences ayant guidé leur conception.

Il reste un ensemble de logiciels dont le statut est encore indéfini : les logiciels de support des autres activités académiques telles que les cours et les tâches administratives. Ces logiciels ont des origines variées. Ainsi certains sont issus de travaux de recherche et ont été ré-utilisés comme support aux autres activités académiques à postériori. D'autres sont des projets étudiants sans lien avec la recherche. Pour finir, ils peuvent également avoir été produits de la même manière que les logiciels de support d'une entreprise par des équipes de développement propres à l'institution mais ne travaillant pas dans le cadre de recherche scientifique. Au vu de ces éléments, nous ne souhaitons pas étendre notre définition à l'ensemble de ces logiciels. En effet, les logiciels développés par des professionnels répondent plus aux mécaniques de l'ingénierie logicielle classique alors que ceux développés par des étudiants s'efforcent de suivre ces mécaniques dans un but de formation. Finalement nous pensons que seuls les projets qui répondent déjà à notre définition pour d'autres raisons, par exemple un ancien logiciel de recherche reconverti en support de cours ou de processus administratifs, se révèlent pertinents dans la présente étude.

Quelques exemples

Afin de mieux appréhender ce que nous considérons comme un logiciel académique, nous développons ci-dessous quelques exemples.

Les preuves de concepts

Une preuve de concept, en informatique, est un programme informatique conçu et implémenté afin de démontrer un concept, le bien fondé d'un sujet de recherche par exemple. Les chercheurs ont régulièrement recours à ce type de programmes dont il n'est pas possible d'énumérer toutes les formes, en voici quelques-unes :

— Prouver l'efficacité d'un algorithme ou d'un nouveau traitement de données
— Comparer deux implémentations d'un même concept
— Donner un exemple fonctionnel à un nouveau langage
— Démontrer l'efficacité d'une méthode ou d'une pratique de travail

Ce type de logiciels académiques est celui qui sera le plus régulièrement sujet d'étude. Mais il aboutit généralement à des projets de plus courte durée souvent déterminée. C'est pourquoi nous expliquons ci-dessus que nous ne souhaitons pas les étudier.

Les outils de recherche

Nous pouvons qualifier d'outils de recherche tous les programmes qui ont pour objectif de soutenir la recherche. Il est donc impossible de donner une liste exhaustive des catégories de logiciels qui sont composées d'outils de recherche mais nous pouvons en citer quelques-unes :

— Les *boites à outils* qui sont des ensembles de logiciels ou composants logiciels destinés à effectuer des traitements liés à un domaine d'application défini.
— Les programmes d'analyse des résultats d'expériences.
— Les programmes de simulation

Ce que nous voulons étudier

Bien que tous les logiciels académiques nous intéressent, nous nous limitons à l'étude des outils de recherche. En effet, la plupart des logiciels académiques qui répondent à notre impératif de longévité sont des outils de recherche. De plus, les autres logiciels académiques qui répondent à notre impératif de longévité peuvent également être assimilés à des outils de recherche. Par exemple, une preuve de concept qui aurait été pensée modulaire pour pouvoir reproduire plusieurs expériences se rapproche plus d'un framework d'expérimentation qui pourra être réutilisé dans d'autres études ou dans un autre but, elle correspondrait donc également à notre définition d'un outil de recherche.

1.3 Les architectures logicielles

Maintenant que nous savons quels projets nous souhaitons étudier, nous pouvons définir ce que nous voulons étudier à leur propos. Afin d'éviter toute ambiguïté, nous avons décidé de définir clairement ce qu'est une architecture logicielle. En effet, il est ressorti de nos discussions avec plusieurs personnes chargées de développer ou d'étudier des logiciels que ce concept est parfois assez flou pour les développeurs de logiciels académiques, surtout quand ils

n'ont pas une formation d'ingénieur logiciel. Nous basons notre définition des architectures logicielles sur deux éléments tirés de la littérature.

Le premier est la définition donnée dans Bass *et al.* (2012, p. 18) :

"*The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.*"

Cette définition nous apparait comme très intéressante car elle reflète bien l'aspect documentaire de l'architecture. Cette dernière doit véhiculer les informations nécessaires à la compréhension du logiciel ou du système. La définition reflète également la complexité des productions logicielles de grande ampleur en tant que système complexe composé de nombreux éléments interagissant entre eux.

La deuxième approche qui vient compléter la première et qui nous semble vraiment appropriée dans le cadre de notre recherche est l'approche de Jansen et Bosch (2005) qui nous parle de l'architecture comme d'un ensemble de décisions. En effet, la documentation classique des architectures revêt de multiples formes : diagrammes, cas d'utilisations, schémas, commentaires, etc. Alors que ces formes de documentation donnent les informations nécessaires pour comprendre le logiciel tel qu'il est actuellement, elles ne permettent pas de conserver l'histoire du logiciel et les différentes influences et exigences qui ont joué un rôle dans sa conception. L'architecture en tant qu'un ensemble de choix permet de tracer un historique bien plus fidèle et de mettre en avant les facteurs d'influence plus que la représentation schématique. Nous estimons que cette approche apporte une compréhension du logiciel de bien meilleure qualité que les approches plus classiques et permet également d'évaluer plus finement si un choix a permis de remplir les exigences qui l'ont dicté.

Nous pouvons maintenant apporter notre propre définition des architectures logicielles et définir ce que nous cherchons à apprendre à leur sujet :

Une architecture logicielle est un ensemble de choix architecturaux permettant de retranscrire à la fois l'évolution du logiciel au cours du temps mais aussi l'état de son architecture à un instant t.

Ces choix sont définis sur base de plusieurs composantes :
— La spécification de la structure architecturale choisie.
— La spécification de l'adaptation de cette structure au projet.
— Les exigences logicielles auxquelles le choix est sensé répondre.
— Les raisons du choix de cette structure plutôt qu'une autre pour répondre à ces exigences.

L'exemple de choix suivant reprend la structure présentée ci-dessus :

— Nous voulons implémenter un *pipeline* de traitement de données.
— Les filtres seront remplacés par des *plugins interchangeables*.
— Afin de remplir une exigence de *modularité*.
— Les plugins ont été préférés à des entités logicielles discrètes pour permettre une meilleure intégration de ceux-ci dans un framework.

En fonction du niveau d'abstraction, ces structures logicielles peuvent être : un style d'architecture, un design-pattern, un choix technologique ayant un impact sur l'architecture, etc.

Dans le cadre de notre recherche, nous nous concentrons sur un ensemble de structures logicielles de haut niveau. Bien que la majorité d'entre elles soient des styles d'architecture ou des patterns architecturaux, nous avons aussi pris en compte les informations fournies sur d'autres structures de haut niveau ayant eu un très fort impact sur le maintien et l'évolution du logiciel.

1.4 Objectifs et questions de recherche

Nous pouvons désormais fixer les objectifs concrets de notre recherche. En effet, nous avons déjà évoqué le travail colossal que représenterait la réalisation d'un guide technique exhaustif adapté au monde académique. Nous avons donc réduit les objectifs généraux à quelques objectifs concrets.

Comme nous ne disposons que de peu de ressources et de temps pour ce projet de recherche, nous avons décidé d'étudier un ensemble très restreint de projets afin d'en tirer la meilleure compréhension possible plutôt que de nous orienter vers une compréhension vague d'un champ d'application très large. Notre recherche se limite aux seuls logiciels académiques d'analyse et de transformation de programmes. Notre choix se porte sur ce domaine en particulier car nous pouvons facilement accéder à des développeurs et architectes travaillant sur ce type de logiciels. Un autre avantage de ce choix est l'expertise en informatique des personnes impliquées dans de tels projets, ce qui facilite grandement la récolte d'informations.

Pour commencer, il nous parait important de formuler des questions de recherche afin d'encadrer et de guider notre étude. Nous avons ainsi produit les questions de recherche suivantes en nous basant sur les définitions que nous fournissons dans les sections précédentes :

— **Quels sont les choix architecturaux de haut niveau qui ont été faits ?**
— **Les exigences auxquelles ils devaient répondre ont-elles été correctement respectées ?**

Nous avons choisi ces questions car elles permettent de représenter de manière concise l'ensemble des informations que nous cherchons à obtenir à propos

des architectures des logiciels académiques.

Premièrement, nous pouvons extraire l'ensemble des problématiques rencontrées dans l'élaboration d'un logiciel académique à partir des exigences mentionnées dans les choix architecturaux.

Deuxièmement, nous pouvons lier ces exigences à un ensemble de structures architecturales et à leurs variations. Encore une fois ces informations sont extraites des choix architecturaux.

Troisièmement, nous pouvons évaluer la pertinence des choix architecturaux. De cette manière, nous pouvons soit confirmer qu'une structure architecturale convient pour répondre à une problématique donnée ou montrer qu'au contraire elle ne convient pas, ces informations étant extraites de la réponse à la deuxième question.

Pour finir, nous pourrons nous baser sur les raisons qui ont poussé les architectes à un choix donné pour évaluer l'impact du contexte sur le choix et la réussite de celui-ci. Ceci permettra de limiter les biais qui pourraient apparaitre en cas de choix liés à un contexte trop spécifique.

C'est sur ces objectifs que s'achève notre description du sujet. Dans le prochain chapitre, nous analyserons les publications existantes et la manière dont nous envisageons de les lier à notre recherche.

Chapitre 2

Étude de l'existant

Dans ce chapitre, nous tentons d'en apprendre plus sur différentes approches adoptées par d'autres chercheurs dans leur étude des logiciels académiques. Comme nous le mentionnons plus tôt, il existe déjà des études au sujet des logiciels académiques mais presque aucune au sujet des architectures. Nous avons estimé qu'il est intéressant d'envisager les autres approches et d'essayer de voir ce que nous pourrions en retirer dans le cadre de nos propres recherches. De plus, cela nous permettra de confirmer certaines de nos intuitions vis-à-vis du domaine avant de débuter une recherche plus formelle.

Dans les sections suivantes, nous commençons par aborder la spécificité du logiciel académique avant de présenter trois approches du problème. Pour chacune d'elles, nous nous aidons d'une publication qui nous parait pertinente et représentative.

2.1 Tour d'horizon

Dans cette première section, nous souhaitons présenter quelques publications traitant de la spécificité du développement du logiciel académique. Nous pourrons donc nous rendre compte de quelle manière cette spécificité est perçue dans la littérature.

Il semblerait que le logiciel académique n'ait commencé à être étudié que récemment dans sa globalité. Comme le mentionne Segal et Morris (2008), la question du développement des logiciels scientifiques a déjà été traitée dans la littérature (Carver *et al.*, 2007) mais les études jusqu'alors portaient plus sur le calcul haute performance. Il est donc important d'attirer l'attention sur le fait que les logiciels scientifiques ne sont pas que de lourds programmes de recherche destinés aux calculateurs mais aussi des logiciels développés pour l'analyse et la visualisation de données sur des machines communes, par exemple. En faisant cette précision, l'article rejoint notre définition plus large des logiciels académiques qui englobe l'ensemble des logiciels développés dans le but ou le

support d'un travail de recherche. Il est néanmoins important de noter que les auteurs ne se limitent pas explicitement au contexte académique, mais que les problématiques abordées sont communes à tous les logiciels de recherche dont les logiciels académiques. D'après les auteurs, les spécificités des logiciels scientifiques s'articulent, entre autres, autour de deux points. D'une part, il arrive souvent que l'expertise requise pour le développement d'un logiciel de recherche soit très pointue, la participation de l'utilisateur final qui est aussi expert du domaine est alors beaucoup plus importante dans le développement d'un logiciel, et c'est souvent le chercheur qui développera son propre logiciel de recherche. D'autre part, le logiciel est souvent utilisé comme méthode d'exploration d'un domaine, il n'est donc pas toujours possible de fixer des exigences et de les analyser en suivant les pratiques d'ingénierie logicielle comme on le ferait pour un logiciel commercial.

Peu après est parue une étude au sujet du développement et de l'utilisation des logiciels scientifiques au sein de la recherche (Hannay *et al.*, 2009). Par le biais d'une enquête, les auteurs de l'article ont mis en évidence l'importance des logiciels scientifiques pour la recherche. Parmi leurs observations, ils remarquent que les scientifiques sont rarement formés correctement aux pratiques d'ingénierie logicielle et n'en voient pas forcément l'utilité. Ceci est du à l'inadéquation des méthodes classiques d'ingénierie logicielle pour la recherche scientifique et au fait que les logiciels prennent souvent de l'ampleur avec le temps. Il est alors souvent trop tard quand on remarque la nécessité d'utiliser des méthodes plus formelles. Une autre difficulté du développement de logiciels scientifiques est l'élaboration de tests car bugs et erreurs de modèle sont difficilement discernables quand il n'existe pas de résultats pré-existants auxquels se référer. Les auteurs remarquent par contre que les scientifiques ont l'air de se tourner de plus en plus vers des logiciels standards ou tentent d'en établir dans leur domaine d'application. Il est important de noter que pour différentes raisons, ils ne peuvent pas encore garantir la généralisation de ces résultats. Ils nous fournissent néanmoins des pistes intéressantes qui pourraient être explorées dans le cadre de notre recherche ou de travaux futurs.

Pour finir ce tour d'horizon, nous avons remarqué une nette tendance à encourager le rapprochement entre logiciel académique et développement open-source (Paumier, 2009; Prlić et Procter, 2012). Ce rapprochement n'est pas nouveau (Ambati et Kishore, 2004) mais nous a semblé plus fréquent dans les publications récentes bien que certains auteurs mettaient déjà en garde contre cette formule magique bien avant l'engouement actuel (Bezroukov, 1999). Nous présenterons d'ailleurs deux approches qui suivent cette tendance au rapprochement dans les sections suivantes.

2.2 10 Règles de développement

La première publication que nous avons sélectionnée, *Ten Simple Rules for the Open Development of Scientific Software* (Prlić et Procter, 2012), nous a beaucoup intéressé car elle présente des règles pour le développement de logiciels académiques, appelés logiciels scientifiques dans l'article. Les auteurs de l'article ont élaboré ces règles pour le développement de logiciels open-source. En effet, ils considèrent que les logiciels académiques devraient être fournis avec leur source conjointement aux publications qui les utilisent dans leur protocole expérimental. Procéder de cette manière permettrait une meilleure compréhension de la démarche du chercheur et une meilleure interprétation des résultats obtenus. Bien que ces règles soient destinées aux développements open-source de logiciels académiques, la plupart sont également valables pour tout développement de logiciels académiques. De plus, nous les avons trouvées suffisamment générales pour pouvoir être appliquées aussi bien aux méthodes de programmation qu'à la conception de l'architecture du logiciel. Il est également très intéressant de noter que les auteurs travaillent dans un domaine où tous les chercheurs ne sont pas experts en programmation, il arrive même que certains ne sachent pas coder. Nous avons jugé pertinent de résumer ces dix règles dans cette section car certaines d'entre elles permettent déjà d'avoir un aperçu des attributs nécessaires à la production d'un logiciel académique de qualité.

Règle 1 : Ne pas réinventer la roue

Cette règle est présente dans la majorité des guides d'ingénierie, de recherche ou de programmation. En effet, on ne le répétera jamais assez, la première chose à faire avant de démarrer un projet quel qu'il soit est de se renseigner sur les travaux similaires ou utiles qui pourraient être utilisés dans le cadre du projet.

Règle 2 : Bien coder

Bien qu'être un expert ne soit pas nécessaire, avoir des bases solides en programmation est plus que nécessaire à la production d'un logiciel de qualité. Cette règle fait justement partie de celles qui prennent plus en compte la possible méconnaissance de l'ingénierie logicielle de la part du scientifique.

Règle 3 : Être son propre utilisateur

Un point important que soulèvent les auteurs est le problème de productivité des logiciels académiques en situation concrète. Il pense donc qu'être utilisateur de son propre logiciel permettra de détecter les éléments qui pourraient freiner ou faire obstacle à la conduite d'une expérience.

Règle 4 : Être transparent

Indépendamment des contraintes légales, les auteurs remarquent que les chercheurs ont une fâcheuse tendance à protéger leur travail et ne le dévoilent pas par peur d'être spoliés de celui-ci. Or ils remarquent que la mise à disposition du dépôt [1] des sources du travail en cours a souvent l'effet inverse et augmente la productivité des différents chercheurs du domaine en les poussant à coopérer. Les auteurs ajoutent également que la publication des sources permet de réduire les erreurs et bugs des logiciels car ces derniers sont soumis à l'expertise d'un plus grand panel de développeurs.

Règle 5 : Être simple

Le concept derrière cette règle est d'utiliser autant que possible des librairies et des formats standards afin de faciliter les contributions. Dans le cadre de projet open-source en particulier, il est nécessaire que le logiciel soit simple à installer et utiliser afin de ne pas décourager les contributeurs potentiels. Cela permet aussi d'attirer plus facilement de nouveaux utilisateurs.

Règle 6 : Ne pas être perfectionniste

Le principe même du développement open-source est son perfectionnement par le biais de son aspect communautaire. Il est donc intéressant de publier le plus tôt et le plus souvent possible des versions de son logiciel afin de bénéficier de cet effet.

Règle 7 : Nourrir et faire grandir sa communauté

Avec cette règle, les auteurs mentionnent quelques points importants pour motiver les contributeurs et en attirer de nouveau, comme la reconnaissance du travail de chacun, la stabilité des interfaces et formats de fichiers, la réactivité aux feedbacks, etc.

Règle 8 : Promouvoir son projet

Passer du temps à promouvoir son projet est le meilleur moyen d'attirer plus de monde à l'utiliser et à y contribuer.

Règle 9 : Trouver des sponsors

Il ne faut pas hésiter à essayer d'attirer des sponsors publics comme privés, à demander des bourses, ou tout autre moyen qui permettrait de soutenir le travail fourni pour le logiciel. De plus, l'aspect communautaire d'un projet et

1. en anglais : repository

la richesse qu'il peut fournir est une plus value qui peut elle-même aider à attirer certains sponsors.

Règle 10 : La science compte

Cette dernière règle est plus une piqure de rappel sur le fait que tout logiciel académique est là pour soutenir la recherche et non pas se suffire à lui-même. Et même si l'attrait d'échanger avec d'autres experts dans notre domaine est vraiment fort, il est important de ne pas perdre de vue l'aspect scientifique derrière la démarche. Cela signifie aussi qu'il est parfois nécessaire d'arrêter le maintien d'un logiciel qui ne présente plus aucune pertinence dans le cadre de nos recherches.

Pertinence et utilité de cette approche

Cet article est très utile dans le cas de la présente étude pour plusieurs raisons.

Tout d'abord, bien que les conseils donnés soient destinés aux projets open-source, la plupart d'entre eux correspondent vraiment au contexte académique et peuvent être appliqués à tout développement de logiciels académiques. En effet, qu'ils soient open-source ou non, beaucoup de projets font face à la problématique de gérer un très grand nombre de contributeurs parmi lesquels des programmeurs, chercheurs ou encore étudiants. Et il est possible de considérer cet ensemble hétéroclite de contributeurs comme une communauté.

D'un autre coté, ces règles permettent d'identifier des contraintes inhérentes au modèle académique. Par exemple, la non expertise des développeurs, la nécessité de sponsors et l'importance de l'aspect scientifique sont autant de facteurs présents dans le contexte académique.

Pour finir, le rapprochement entre logiciel académique et logiciel open-source nous parait très intéressant et nous l'abordons à nouveau avec la prochaine approche.

2.3 Améliorer le logiciel académique grâce au développement open-source

Avec leur article, *How can Academic Software Research and Open Source Software Development help each other ?* (Ambati et Kishore, 2004), Ambati et Kishore nous parlent des bénéfices mutuels que les logiciels académiques et le développement de logiciels open-source pourraient tirer l'un de l'autre. Approche très intéressante d'après nous car elle permet de rapprocher les pratiques de développement des logiciels académiques de pratiques existantes pour un autre type de logiciels. Dans leur article, les auteurs se concentrent sur ce

qu'ils appellent les *Logiciels de Recherche Académiques*. Bien que proche de la notre, leur définition comporte une différence majeure : les logiciels de recherche académiques sont automatiquement des projets à grande échelle faisant intervenir de nombreux étudiants ou chercheurs sous la supervision de chercheurs plus expérimentés. De notre coté, nous avons croisé des personnes qui avaient débuté un projet seules ou avec un groupe restreint de chercheurs et étudiants pendant plusieurs années pour parfois s'étendre par la suite. Nous pouvons donc considérer les logiciels de recherche académiques de l'article comme un sous ensemble des logiciels académiques tels que nous les définissons dans le premier chapitre.

Cet article aborde un ensemble de problématiques liées aux projets académiques de grande envergure dont les différents contributeurs sont éparpillés dans plusieurs institutions et pays. Il soulève ainsi les difficultés rencontrées dans le cadre du développement de tels logiciels et tente de trouver des solutions du coté du développement des logiciels open-source. Cette approche diffère de celle que nous souhaitons adopter car il s'agit plus d'une approche méthodologique, qui cherche à trouver les bons outils et méthodes afin de solutionner les problématiques liées au développement de logiciels dans le contexte académique. Il permet néanmoins d'aborder une série de problématiques de ce type de projets.

Une des premières problématiques que les auteurs abordent est la rotation rapide des équipes qui développent le logiciel. En effet, beaucoup d'étudiants et doctorants ne vont participer que pendant leur cursus ou la rédaction de leur thèse et partiront ensuite. Un problème se pose alors car les logiciels académiques sont souvent mal documentés et il ne reste que le code source que les contributeurs ont écrit pour seule trace de leurs travaux.

Un autre problème majeur est celui de la collaboration, de la communication et du contrôle des contributions au logiciel dans le cas de la répartition des contributeurs entre plusieurs institutions.

Finalement ils abordent aussi les problèmes posés par la répartition des sous-projets entre différents groupes de recherche, chaque groupe portant plus d'importance aux objectifs scientifiques qu'il cherche à atteindre avec le logiciel plutôt qu'à son maintien. Dans cette situation, les groupes de développement prennent rarement soin de la qualité du logiciel en préférant les corrections rapides. Ils ne se soucient pas non plus de l'état des autres modules du code.

Selon les auteurs, ces problèmes peuvent être adressés grâce à l'adoption de certaines pratiques du développement open-source et de certains de leurs outils. Mais ils soulignent également le manque d'outils dédiés à certaines pratiques comme la gestion des exigences, la gestion du projet, l'estimation des métriques, la conception de suites de tests, etc.

Pertinence et utilité

Bien que l'article n'ait pas la même approche que la notre, nous pensons qu'il est très utile pour approfondir notre compréhension du contexte de développement des logiciels académiques. Ainsi nous pouvons comparer la liste des problématiques liées à ce contexte abordées dans l'article aux problématiques abordées dans les autres articles. Nous pourrons également les confronter à nos propres observations plus loin dans notre étude.

2.4 Améliorer le logiciel académique par l'évaluation de sa qualité

La troisième et dernière approche que nous abordons est soutenue par une publication s'intéressant aux attributs de qualité qui permettent d'améliorer le processus de développement de logiciels académiques. Dans *Quality-Aware Academic Research Tool Development* (Cho *et al.*, 2012), les auteurs nous présentent les attributs de qualité habituellement utilisés en ingénierie logicielle classique et dans le monde de l'entreprise tout en les adaptant au contexte académique. Leur objectif est double : permettre l'évaluation des projets académiques sur base de critères adaptés afin d'aider le développement de logiciels de qualité et faire un état des lieux de la situation actuelle en évaluant un ensemble de projets sur base de ces critères.

Pour ce faire, les auteurs fournissent une liste de *facteurs* de qualité qui doivent être évalués pour juger des qualités d'un projet. Cette liste se divise en deux catégories, les facteurs liés au développement du logiciel en lui-même, et les facteurs qui feraient du logiciel une réussite d'un point de vue *commercial*. Les facteurs qu'ils étudient sont les suivants :

Interopérabilité et intégration avec d'autres outils : les auteurs considèrent que les outils de recherche doivent être le plus interopérables possible. Ils soulèvent également la difficulté de la chose mais soutiennent que cette interopérabilité doit être autant que possible maintenue, ne serait-ce qu'au niveau des formats de fichiers. En utilisant des formats standards au domaine quand ils existent par exemple.

Capacité de développement : La capacité de développement est liée à l'évaluation de la maturité des processus de développement. Bien que le monde de l'entreprise fourmille d'outils et de standards à ce sujet, les auteurs signalent qu'une évaluation complète n'est pas toujours pertinente mais qu'il existe un ensemble minimum de paramètres à surveiller. Pour eux, les projets académiques doivent au moins gérer les points de configuration de leur logiciel [2]

2. Software Configuration Management (Berlack, 1992)

ainsi que les changements et les défauts de leur logiciel afin d'en assurer le support.

La qualité du code : Elle doit être évaluée sur base de métriques éprouvées, par exemple le nombre de lignes non commentées (ULOC) ou la complexité de McGabe, mais aussi sur base de la conformité à un style de programmation dans l'ensemble du projet (Comment déclarer tel élément, quelle est l'indentation à appliquer, etc).

Les facteurs de qualité *commerciaux* : Il s'agit des facteurs qui vont permettre la viabilité du logiciel au-delà de la recherche académique. Il s'agit majoritairement de garanties de support et de maintien du logiciel, et de la gestion correcte des licences même si le logiciel est open-source.

Après évaluation des projets selon ces critères, les auteurs sont parvenus aux constations suivantes : les outils de recherche ne sont pas assez documentés, ils intègrent rarement des systèmes de gestion des changements/défauts ou de la configuration et très peu utilisent des licences. Par ailleurs, ils soulèvent que l'inexpérience des programmeurs, le manque de fonds et la vision à court terme de ces logiciels posent également des problèmes qui doivent être surveillés de près.

Pertinence et utilité

Comme dans l'approche précédente, les conclusions nous donnent des solutions difficilement applicables au niveau de la conception de l'architecture car l'accent est mis sur les processus et les évaluations continues à mettre en place dans le projet. Elles pourraient néanmoins être prises en compte dans la rédaction des conseils généralistes à défaut d'être utiles pour guider la production de recommandations concrètes. Par contre, une partie des problématiques déjà abordées se voient confirmées comme l'inexpérience des développeurs, l'abandon rapide de certains projets académiques ou la difficulté de trouver des fonds et/ou des sponsors.

2.5 Synthèse

Nous pouvons maintenant nous demander ce qui ressort de cette analyse du contexte.

Premièrement, nos recherches préliminaires ne nous ont pas permis de trouver de travaux similaires aux nôtres dans la littérature existante, nous vérifierons néanmoins l'absence de matériaux pertinents de manière plus stricte et formelle dans le chapitre suivant.

Ensuite, nous avons pu confirmer la spécificité du développement de logiciels académiques qui prend des formes diverses en fonction de l'envergure et du type de projets. Parmi les plus récurrentes, nous pouvons parler du rôle de développeur de ces logiciels qui est souvent joué par les experts du domaine d'application en raison des exigences beaucoup plus poussées que dans les projets commerciaux. Cette situation devient problématique lorsque nous constatons que ces experts sont rarement formés à l'ingénierie logicielle et ne le jugent pas utile pour leur travail. Nous pouvons également ajouter l'inadéquation de certaines méthodes avec la réalité de la recherche scientifique. Par exemple, il est parfois impossible de spécifier des exigences précises avant le début d'un projet scientifique et dans le cadre de ces projets de *découverte*, les chercheurs attendent plutôt de voir émerger ces exigences au fur et à mesure que des prototypes sont produits. Un autre aspect posant régulièrement des problèmes dans le cadre des logiciels académiques est l'objectif scientifique qui prend le pas sur les objectifs de qualité du logiciel, problème accentué par les difficultés de collaboration et d'organisation des projets qui sont largement répartis entre différentes institutions de différents pays.

En conclusion, l'analyse du contexte nous aura surtout fourni un ensemble de pointeurs à surveiller dans notre étude et conforté dans l'idée que ce travail est nécessaire. Dans le chapitre suivant, nous abordons nos méthodes et protocoles de recherche et les résultats qu'ils ont fournis.

Chapitre 3

Recherche

Dans ce chapitre, nous décrivons notre protocole de recherche, les activités qui y sont liées ainsi que les méthodes employées. Nous explicitons également les motivations de nos choix. Enfin, nous résumons les résultats que nous avons obtenus pour chacune des étapes de notre recherche, résultats que nous synthétiserons et interpréterons dans un chapitre suivant.

3.1 Protocole général

Comme nous l'avons vu, l'objectif de la présente recherche est de réunir des informations sur les architectures des logiciels académiques destinés à l'analyse et à la transformation de programmes, espérant ainsi avoir un point de départ pour la rédaction de recommandations pratiques pour les chercheurs et étudiants qui devraient développer leur propre logiciel académique.

Afin de réunir ces informations, nous adoptons une approche hybride mêlant plusieurs méthodes de récolte de données (Easterbrook *et al.*, 2008).

Dans un premier temps, nous souhaitons vérifier qu'il n'existe pas de travaux de compilation existants dans le domaine qui pourraient nous servir de point de départ. Pour cela, nous démarrons notre recherche par une revue systématique de littérature (Keele, 2007). L'objectif de celle-ci est de rechercher des travaux de synthèse ou des guides pratiques concernant les choix architecturaux communs, conseillés ou déconseillés dans la production de logiciels académiques.

La deuxième étape de notre recherche consiste à réunir un maximum d'informations à propos de quelques projets par le biais d'interviews semi-structurées (Hove et Anda, 2005). Les objectifs de cette étape sont multiples. Nous souhaitons notamment réunir des informations sur les problématiques spécifiques à la conception d'un logiciel académique, sur les choix architecturaux qu'elles impliquent et sur les motivations propres aux logiciels académiques. Les informations concernant les problématiques nous permettront de confirmer ou infir-

mer les données que nous avons extraites de la littérature existante concernant les problèmes rencontrés par les logiciels académiques. Les choix architecturaux qu'elles impliquent nous permettront de comprendre leur impact sur les architectures et d'évaluer l'efficacité des solutions qui y sont apportées. Enfin, les motivations propres aux logiciels académiques nous permettront de mieux comprendre les facteurs influençant l'évolution du logiciel et nous permettront de confirmer le lien entre le contexte académique et les problématiques mentionnées. Ces informations permettront également de confirmer la spécificité des logiciels académiques vis-à-vis des logiciels propriétaires et open-source développés dans d'autres contextes.

La troisième et dernière étape consiste en une enquête (Seaman, 1999) destinée à acquérir des données à propos de projets supplémentaires pour confirmer nos premières impressions basées sur les interviews. Les données récoltées seront similaires à celles que nous souhaitons obtenir par le biais de nos interviews. De plus, ceci nous permettra d'évaluer l'efficacité des deux méthodes pour récolter des informations concernant les architectures logicielles ainsi que le temps nécessaire dans les deux cas, ceci pouvant s'avérer utile lors de la planification d'une recherche cherchant à étendre le travail de celle-ci.

Afin de guider le choix de projets dans les deuxième et troisième étapes, nous avons établi les critères suivants :

— Le projet doit comprendre la production d'un logiciel académique
— Le logiciel doit être destiné à l'analyse et/ou la transformation de programmes
— Le logiciel doit être utilisé comme outil ou sujet de recherche dans un minimum de deux publications
— Le logiciel doit être décrit dans au moins une publication
— Le logiciel doit avoir été actif depuis au moins cinq ans et devait encore être actif lors des premières demandes d'interviews et lorsque les formulaires furent envoyés (1er semestre 2015-2016)

Ces critères permettent de nous assurer que le travail mené serait reproductible à l'avenir. De plus, le critère d'activité garantit d'une part d'obtenir des informations récentes sur le projet et d'autre part de ne discuter que de projets ayant su faire face aux différentes problématiques du monde académique pour perdurer dans le temps.

Nous allons profiter des sections suivantes pour développer davantage les différentes étapes et les méthodes utilisées. Nous motiverons nos choix de méthodes et nous présenterons les résultats obtenus à chaque étape, le chapitre suivant étant destiné quant à lui à la synthèse et l'analyse de tous ces résultats.

3.2 Revue systématique de littérature

La revue systématique de littérature représente la première étape de notre protocole de recherche. Nous voulons vérifier dans la littérature existante si nous trouvons des travaux similaires aux nôtres. Dans les sous-sections qui suivent, nous décrivons notre protocole expérimental ainsi que les résultats que nous avons obtenus en appliquant celui-ci.

A première vue, ce travail peut sembler redondant avec l'analyse du contexte du chapitre précédent. Cependant, il existe une nette différence entre les deux. Dans le chapitre précédent, nous avons recherché de manière informelle et sans protocole strict toute information pouvant compléter notre vue d'ensemble. Dans cette section, nous cherchons à élaborer un protocole strict qui permet d'établir l'état des lieux des travaux qui auraient des objectifs similaires à ceux de la présente étude.

Nous avons commencé par élaborer notre protocole en suivant la méthode développée par l'équipe de l'Université de Keele sous la supervision de Barbara Kitchenham (Keele, 2007). Mais au fur et à mesure de nos recherches préliminaires, nous remarquons la difficulté de réunir des documents sur le sujet de manière méthodique. Nous décidons alors d'adapter la méthode présentée dans le guide de Keele avec la méthode dite du Snowballing présentée par Wohlin (2014).

3.2.1 Objectifs de la revue systématique de littérature

Pour cette revue, deux options s'offrent à nous. Nous pouvons soit tenter de réunir des informations sur un type de logiciels spécifiques et en analyser l'architecture sur base des informations présentes à leur sujet dans les publications, soit nous concentrer sur des travaux de compilation ou des guides techniques déjà rédigés. Nous avons choisi la deuxième option pour trois raisons.

Premièrement, bien que les logiciels académiques et leur fonctionnement soient souvent décrits dans des publications, leur architecture est assez rarement explicitée et les informations restent souvent partielles quand elles sont fournies.

Deuxièmement, nous ne pouvons pas réduire le champ de recherche autant que pour des interviews et des enquêtes car les critères de sélection des projets doivent être établis uniquement sur base des informations fournies dans les publications elles-mêmes. Ceci aboutit à la nécessité d'analyser énormément de publications dont la plupart ne fournissent pas d'informations pertinentes pour notre revue systématique de littérature.

Enfin, nous comptons compléter les résultats obtenus par ceux d'interviews et d'enquêtes qui sont des méthodes qui nous semblent plus adaptées à la récolte d'informations sur des projets particuliers.

Nous cherchons donc des publications fournissant des synthèses ou des compilations de données concernant les architectures employées dans les logiciels académiques ou encore des guides techniques explicitant pour certains types de logiciels quels sont les choix architecturaux conseillés. Notre objectif est double : réunir des informations pertinentes pour notre propre recherche tout en vérifiant qu'il n'existe pas déjà des travaux similaires.

3.2.2 Les questions de recherche

Pour rappel, les questions de recherche de notre étude sont les suivantes :

— **Quels sont les choix architecturaux de haut niveau qui ont été faits ?**
— **Les exigences auxquelles ils devaient répondre ont-elles été correctement respectées ?**

3.2.3 Les critères de sélection

Nous commençons par définir les critères de sélection des publications que nous souhaitons réunir au cours de cette revue systématique de littérature. Nous sommes arrivés à la liste de critères suivante :

— Nous ne considérons que les publications en anglais
— La publication doit étudier des logiciels qui répondent à la définition de logiciels académiques que nous présentons au premier chapitre.
— La publication doit étudier les architectures des logiciels académiques
— Publications retenues :
  — Travail de compilation ou de synthèse à propos des architectures logicielles dans un contexte académique.
  — Guide technique donnant une série de conseils liés à la conception d'une architecture
  — Travail d'études de cas, comparant plus de trois architectures issues de logiciels académiques.
— Publications rejetées :
  — Étude de cas ou description de logiciels ne comportant pas au moins trois logiciels
  — Publication ne présentant pas suffisamment d'informations concernant le contexte et les motivations des choix architecturaux

3.2.4 La méthode de recherche

Dans un premier temps, nous effectuons une série de recherches sur Google Scholar à l'aide des strings de recherche prédéfinis.

Les résultats obtenus par ces recherches sont ensuite triés sur base des critères de sélection présentés plus haut. L'ensemble des publications retenues forment ainsi notre premier ensemble de publications à évaluer.

Nous sélectionnons une publication à évaluer que nous considérons alors comme la publication en cours d'évaluation. Nous examinons l'ensemble des articles référencés par cette publication. Si parmi ceux-ci, certains semblent correspondre à nos critères de recherche et ne font pas encore partie des articles retenus ou à évaluer, nous évaluons leur pertinence à l'aide des mêmes critères de sélection que lors de la recherche initiale. Les nouveaux articles respectant les critères de sélection ainsi trouvés sont ajoutés à l'ensemble des articles à évaluer. Nous utilisons ensuite l'outil de citation de Google Scholar pour reproduire le même travail sur les articles citant l'article en cours d'évaluation. Une fois cela terminé, l'article en cours d'évaluation est considéré comme retenu. Nous reproduisons ces étapes jusqu'à ce qu'il n'y ai plus d'articles à évaluer.

3.2.5 Extraction des données

Au fur et à mesure que nous traitons les publications retenues, nous formons une liste des structures architecturales qui y sont référencées. Pour chacune d'entre elles nous retenons :

- Le nombre d'occurrences de la structure
- Pour chaque occurrence :
  - Adaptation de la structure
  - Raisons du choix de cette structure
  - Si possible, évaluation du respect des exigences auxquelles le logiciel tentait de répondre avec cette structure.

Ces éléments nous permettront d'établir une liste de structures architecturales ainsi que leurs forces et leurs faiblesses lorsqu'elles sont utilisées dans un contexte académique.

3.2.6 Traitement et synthèse des données

Nous évaluons les structures architecturales sur deux critères :

- Leur fréquence d'apparition dans la littérature
- Leur utilité dans le contexte académique

Le but ici est d'évaluer quelles sont les structures à conseiller ou déconseiller dans le contexte académique en fonction des exigences à respecter. Le résultat final de ce traitement est la production d'une liste de choix architecturaux typiques.

3.2.7 Résultats et discussion

Lorsque nous appliquons le protocole, nous ne parvenons pas à trouver de documents répondant à nos critères de sélection. Nous nous attendions à ne pas trouver beaucoup de résultats et c'est pourquoi nous avons utilisé le snowballing. Néanmoins nous n'avons pas trouvé une seule étude qui nous permettrait de former une base pour appliquer cette méthode.

D'après ce résultat, il semblerait qu'il n'existe pour l'heure pas d'étude similaire à la notre ni d'autres publications pouvant répondre partiellement à nos questions de recherche.

Une autre explication à ce résultat serait l'absence de définition stricte pour les logiciels académiques. C'est aussi la raison pour laquelle nous ne fournissons pas de chaînes de recherche. Nous avons essayé de nombreuses possibilités mélangeant des termes tels que *academic software*, *research software*, *research tool*, *scientific software*, *academic software engineering*, *architecture*. Mais nous n'avons jamais réussi à trouver ne serait-ce qu'une publication qui nous semble pertinente.

Nous avions aussi quelques réserves concernant une partie de nos critères de recherche mais comme nous n'avons pas su trouver d'articles pertinents, nous n'avons pas pu affiner les critères afin de tenter d'appliquer à nouveau le protocole de recherche.

Ces résultats ont encore confirmé que notre approche est originale. Nous poursuivons donc notre recherche dans la section suivante qui décrit comment nous avons mené une série d'interviews afin d'en apprendre plus sur les logiciels académiques et leur architecture.

3.3 Interviews

Dans cette section, nous présentons la première partie du travail de collecte de nouvelles données concernant les architectures des logiciels académiques : les interviews. Nous hésitions à mener une revue des publications décrivant les architectures logicielles de projets académiques. Nous n'aurions donc pas cherché des travaux de synthèse, comme lors de la revue systématique présentée dans la section précédente, mais directement des descriptions de projets individuels. Cette solution fut écartée d'une part par manque de ressources et d'autre part à cause du manque d'informations relatives à l'architecture dans les publications décrivant des logiciels académiques. Bien que l'architecture y soit parfois décrite, la description s'avère plus souvent partielle voir inexistante. Nous avons préféré à cette solution la conduite d'une série d'interviews. En effet, cette forme d'enquête permet d'avoir une meilleure compréhension de l'architecture des projets, de la façon dont elle évolue et des problématiques et motivations qui guident cette évolution.

Au total, nous avons interviewés sept personnes issues de cinq projets différents au sujet de l'architecture du logiciel développé au sein de ces projets. Ci-après, nous commençons par présenter notre protocole d'interview. Cette description sera suivie par la présentation des projets interviewés et des résultats.

3.3.1 Le protocole d'interview

Pour établir ce protocole, nous nous sommes majoritairement basé sur les travaux de Seaman (1999) et Hove et Anda (2005). Ceci nous a conduit à choisir de mener des interviews semi-structurées car cela nous semblait être la formule la plus adaptée pour récolter un maximum de données qualitatives sur un ensemble de sujets précis liés aux architectures logicielles. Ces travaux se sont aussi révélés être des sources d'informations et de conseils utiles à la bonne conduite d'une interview.

D'après Hove et Anda, l'un des points les plus importants est la planification. En effet, la plupart des tâches à effectuer sont assez chronophages et il est nécessaire de les synchroniser entre elles. Du fait de cette synchronisation nécessaire, le protocole que nous présentons ci-dessous ressemble a une liste de tâches et de points de synchronisation entre ces tâches.

La première chose que nous préparons est le guide d'interview que nous présenterons plus loin. Celui-ci reprend l'ensemble des questions que nous souhaitons aborder pour chaque projet. Il permettra également de conduire une interview test afin de s'assurer de la durée approximative nécessaire à la couverture des sujets présents dans le guide. Pendant la rédaction de ce guide, nous prenons également contact avec les participants potentiels en leur signalant une fourchette dans laquelle nous aimerions conduire l'interview.

Après la mise au point du guide d'interview vient une étape de recherche liée à chaque projet confirmé. Un fois le rendez-vous pour l'interview fixé, nous cherchons dans la littérature et sur le site du projet des informations sur son fonctionnement et son architecture. Ceci permet d'une part d'avoir une connaissance minimale du logiciel et de son fonctionnement et d'autre part de personnaliser l'interview afin de discuter plus rapidement des détails publiés à propos du projet et poser plus de questions sur les choix et motivations n'apparaissant pas dans les publications disponibles.

L'interview commence par une rapide présentation des objectifs de l'intervieweur et du type de questions qui vont être posées. C'est aussi le moment de demander l'autorisation d'enregistrer puisque nous avons opté pour l'enregistrement à des fins de retranscription. Ensuite, l'intervieweur commence à poser les questions en restant à l'affut d'éléments imprévus sur lesquels il pourrait insister ou se renseigner davantage.

Après l'interview, il reste trois étapes. Dans un premier temps, la transcrip-

tion est effectuée, ceci peut prendre jusqu'à huit heures de travail par heure d'enregistrement. Ensuite, il est nécessaire d'extraire les données, ce processus doit être le plus rigoureux possible afin d'obtenir des résultats comparables. Nous proposons un protocole d'extraction de données que nous présentons plus bas. Pour finir, les données peuvent être analysées. Dans notre cas, cette opération se fait en deux temps. Nous commençons par une analyse préliminaire pour nous aider à la rédaction de l'enquête que nous présentons dans la section suivante et attendons les résultats de celle-ci pour faire une synthèse complète de ce que nous avons appris. Vous trouverez cette synthèse dans le prochain chapitre.

3.3.2 Le guide d'interview

Le guide d'interview présenté dans l'annexe 5.3 reprend sous forme de questions types l'ensemble des sujets à traiter lors de l'interview. Il ne s'agit que rarement de lire les questions mais plutôt de les utiliser comme aide mémoire pour ne pas oublier d'aborder un sujet donné.

De plus, les questions sont adaptées en fonction des connaissances acquises lors des recherches préliminaires. Par exemple, la section globale commence parfois par un cours rappel des structures logicielles de l'architecture qui sont abordées dans une publication. Ceci permet de donner à l'interviewé un exemple de ce que nous entendons par structures et choix logiciels et fait gagner du temps en nous permettant d'aborder plus rapidement d'autres choix qui ne sont pas mentionnés dans les publications.

Nous avons également remarqué que d'un interviewé à l'autre, une même question n'est pas comprise de la même manière et que reformuler une même question plusieurs fois lors de l'interview aboutit parfois à des réponses plus complètes.

3.3.3 L'extraction des données

Au début, nous avons décidé d'établir un protocole strict pour extraire les données de manière plus rapide et efficace tout en rendant plus facile le travail de comparaison et de synthèse des résultats obtenus. Au fur et à mesure que nous rédigions ce protocole, nous avons remarqué qu'il était trop contraignant, demandant des informations trop précises et augmentant par là-même le nombre d'informations périphériques à traiter. Ce protocole a donc évolué vers un guide plus souple.

En pratique, nous prêtons une attention particulière aux éléments suivants :

— Informations pratiques relatives au projet
— Problématiques rencontrées
— Exigences principales

- Structures architecturales utilisées
  - Pour répondre à quelle problématique, quelle exigence ?
  - Pourquoi cette structure en particulier ?
  - Comment est-elle adaptée au projet ?
  - Succès de la structure à donner une solution à la problématique, à répondre à l'exigence
  - Problèmes causés par la structure
- Chronologie des choix
- Évolution des motivations
- Autres décisions ayant un impact majeur sur le projet

3.3.4 Projets interviewés

Dans les sous-sections suivantes, nous présentons succinctement les projets qui ont fait l'objet d'une interview. Nous avons découplé cette présentation des résultats afin de ne pas permettre le recoupement entre un projet donné et son architecture quand nous n'en avons pas reçu l'accord explicite. Les sites web des différents projets sont par ailleurs fournis dans la bibliographie.

3.3.5 Projets : ProM

ProM (Van Dongen *et al.*, 2005) est un framework dédié au *Process Mining*. Le principe derrière le Process Mining est d'extraire des informations à propos d'un processus sur base de ses logs d'exécution. ProM propose un grand nombre de méthodes d'analyse de logs permettant d'extraire des informations variées à propos du flux d'exécution, des performances, des données échangées, etc.

Ce logiciel nous a particulièrement intéressé parce qu'il est typique des projets à grande échelle disposant de beaucoup de contributeurs. Il est soutenu par pas moins de cinq institutions et un grand nombre de plugins sont développés par des contributeurs répartis aux quatre coins du monde.

3.3.6 Projets : mCRL2

mCRL2 (Groote et Mousavi, 2014) est un langage de spécification formel associé à un ensemble d'outils. L'objectif de ces outils et de ce langage sont de permettre la modélisation, la validation et la vérification de protocoles de systèmes concurrents. Les outils supportent de nombreuses fonctionnalités.

Bien que moins important que ProM au niveau de ses contributions, il s'agit d'un projet qui dépasse la dizaine de contributeurs (estimation donnée à 20 pour mCRL2 uniquement). De plus, sa longévité est assez intéressante, mCRL2 ayant au moins dix ans et héritant d'un projet débuté il y a 30 ans sous le nom de μCRL.

3.3.7 Projets : Rascal

Rascal (Klint *et al.*, 2009) est un langage de méta-programmation qui permet l'analyse, la transformation et la génération de code source. Il permet aussi bien la méta-programmation que la programmation classique. Le projet a été développé comme un laboratoire d'expérimentation et n'a donc pas d'objectif immédiat d'utilisation en production. Il reste cependant la base de nombreux travaux de recherche notamment au Centre de mathématique et d'informatique d'Amsterdam (CWI) où l'équipe initiatrice du projet le développe encore. Il est aussi important de mentionner que Rascal est la continuation du projet ASF+SDF (van den Brand *et al.*, 2001) également mené au CWI et qu'il hérite de certains de ses éléments.

3.3.8 Projets : Tom

L'objectif de Tom (Balland *et al.*, 2007) est d'ajouter des fonctionnalités telles que le pattern matching et la transformation de code à des langages existants. Il prend donc la forme d'un pré-compilateur pour C ou Java permettant d'exprimer dans le langage cible les fonctions représentées par les primitives de Tom. Il a initialement débuté comme un back-end pour les compilateurs Elan (2004) et ASF+SDF mais fut transformé en pré-compilateur autonome pour C et puis Java avec une équipe grandissante. Développé à l'origine par un seul chercheur, il s'est vu attribuer des ressources supplémentaires au gré de son existence.

3.3.9 Projets : srcML

srcML (Collard *et al.*, 2011) est à la fois le nom d'un format XML pour la représentation de code source ainsi que celui des outils qui permettent de convertir du code source en srcML. Sous ce format, il est assez facile d'analyser ou manipuler le code source grâce aux technologies XML et il est possible de revenir au langage d'origine sans perte car celui-ci est entièrement encapsulé dans srcML. Comme dans le cas de Tom, le projet a été écrit pendant plusieurs années par une seule personne avant de recevoir plus de ressources et de contributeurs.

3.3.10 Résultats

Nous allons maintenant présenter les résultats de nos différentes interviews. Dans ces sous-sections, nous nous focaliserons sur une présentation détaillée des différentes thématiques abordées. Nous réservons l'approche chiffrée des différentes thématiques à notre chapitre de synthèse combinant les résultats des interviews et de l'enquête. Les projets participants n'ayant pas tous accepté d'être référencés explicitement, il est normal que certains noms de projets se répètent souvent alors que d'autres ne sont jamais mentionnés.

3.3.11 Résultats : Les facteurs influençant les architectures

Nous commençons notre présentation des résultats par une description des facteurs qui influent sur les décisions de conception des projets. Ces facteurs d'influence regroupent les problématiques rencontrées, les centres d'intérêts et les qualités logicielles recherchées par les projets.

Gestion des licences

Nous abordions déjà cette problématique dans le chapitre 2 et l'avons à nouveau rencontrée lors des interviews. Dans le cadre de celles-ci, les participants ont mentionné à plusieurs reprises la gestion des licences au sein de leur logiciel. Dans certains cas, elles furent l'un des moteurs principaux du choix de la nouvelle architecture du projet. Dans d'autres cas, elles sont imposées par des choix de conception. Que ce soit pour garantir la liberté de contribution ou s'accorder avec une politique commerciale, tous les projets interrogés ayant prévu ou eu des débouchés commerciaux à un moment de leur cycle de vie, la gestion fine des licences de distribution semble être une nécessité pour les projets académiques.

Objectifs de recherche

Cette problématique est sans doute l'une des plus caractéristiques des logiciels académiques et se retrouvait par conséquent dans notre étude du contexte. Elle prend différents aspects en fonction des projets rencontrés. Dans certains cas, il s'agit juste d'un manque d'investissement, les contributeurs ne documentent pas suffisamment leurs composants ou ne fournissent pas les cas de tests nécessaires à la maintenance et la non-régression de leur contribution. Dans d'autres cas, c'est un impact négatif direct comme le non respect de certaines conventions de développement ou le contournement de certaines API, ce fut le cas avec un contributeur de mCRL2 qui souhaitait améliorer les performances de son outil. Généralement, cette dissonance entre les objectifs scientifiques et le maintien du logiciel entraine une perte de maintenabilité, voire des pertes de fonctionnalités quand un code n'est plus maintenable une fois que son auteur quitte le projet.

Une autre facette de cette problématique est la possibilité de publication, ainsi certains interviewés nous faisaient part de la frustration qu'ils ressentaient quand ils ne pouvaient pas améliorer ou optimiser leur logiciel comme ils le désireraient parce qu'il n'était pas possible de publier à ce sujet. Dans le cas de Rascal, seul 20% du temps peut être réservé à l'amélioration du logiciel.

Manque de formation

Nous ne nous attentions pas à trouver cette problématique au sein de projets liés à l'étude des programmes informatiques. Néanmoins, le fait que cer-

taines composants logiciels soient développés par des étudiants en cours de formation aboutit parfois à l'émergence de cette problématique au sein de ce type de projets. Les conséquences sont généralement similaires à celles de la problématique liée aux objectifs scientifiques. Nous n'avons cependant pas d'exemple lié au développement de l'architecture, mais plutôt des cas où les développeurs ne connaissaient pas les conventions de programmation à adopter ou ne savaient pas bien utiliser les technologies du projet.

Renouvellement des contributeurs

Cette problématique est directement liée au monde académique. En effet, les contributeurs à des projets informatiques sont assez souvent des étudiants ne restant que pour la durée du projet. Elle se recoupe avec plusieurs des problématiques précédentes. Ainsi c'est souvent à cause de cette limite temporelle qu'il est possible de constater un manque de formation ou d'investissement pour le projet. Dans certains projets, des versions multiples du logiciel sont maintenues parce que l'équipe principale ne dispose pas des ressources ou informations nécessaires pour porter certains composants d'importance d'une version à l'autre.

Répartition géographique des contributeurs

Bien qu'elle ne touche qu'un seul de nos projets, cette problématique eut une importance primordiale dans l'évolution du projet car elle nécessite de faciliter l'indépendance des différentes équipes qui travaillent sur les composants d'un même logiciel.

Inadéquation des méthodes classiques et manque d'outils adaptés

Les problématiques d'inadéquation entre méthode d'ingénierie classique et contexte académique et de manque d'outils de développement adaptés à ce contexte alors qu'elles sont présentes dans la littérature ne sont pas beaucoup ressorties lors de nos interviews. Nous pouvons néanmoins citer le cas de Rascal où les concepteurs avaient utilisé le design pattern du visiteur car celui-ci permet de séparer clairement les opérations de la structure de l'arbre syntaxique au sein d'un interpréteur. Néanmoins, le manque de support de ce pattern par les outils et les changements et évolutions très rapides liés au contexte de recherche ont rendu ce choix contre-productif. Au final, un choix jugé adéquat du point de vue de l'ingénierie logicielle s'est retrouvé mis en échec par le contexte de la recherche et des outils qui y sont liés.

Difficulté de rédaction des tests

Dans la littérature, nous apprenions que les tests sont parfois difficiles à mettre en place car il n'est pas aisé de différencier les erreurs de logique et

les bugs lors du développement d'un logiciel original. Nous n'avons cependant jamais abordé ce sujet dans les interviews. Une hypothèse que nous pouvons formuler est que dans le champ d'application réduit que nous étudions, c'est-à-dire les logiciels de transformation et d'analyse de programmes, les développeurs connaissent et appliquent plus facilement des méthodes de tests qui permettent de faire face à cette problématique. Par exemple, les développeurs de Rascal utilisent des tests aléatoires, et non des cas de tests prédéfinis, afin de détecter plus facilement les anomalies.

Qualités du logiciel

De manière générale, les projets ayant participé à nos interviews mettent l'accent sur la viabilité et l'utilisabilité de leur logiciel. Ainsi nous abordons très souvent des objectifs tels que la maintenabilité, la facilité de compréhension du code, la facilité de contribution, la facilité d'utilisation, etc, dans nos interviews. Dans certains cas, cela prend une place prépondérante, ainsi un de nos interviewés a expliqué que le principal guide à toute décision de conception était la simplicité. Afin de réduire au maximum les tâches de maintenance et d'évolution, le logiciel, son architecture et son implémentation devaient rester les plus simples possible.

Sources de financement ou sponsors

Ce facteur d'influence également mentionné dans la littérature apparait à plusieurs reprises dans nos discussions. Qu'ils s'agissent d'une bourse ou du support d'une entreprise, la présence d'un financement extérieur a toujours un impact sur l'évolution du logiciel car celle-ci est plus ou moins contrainte à une orientation donnée par la source de financement. Un des projets a, par exemple, reçu d'une entreprise un financement destiné à l'implémentation de certaines fonctionnalités spécifiques, il a du adapter son architecture pour répondre à cette demande.

3.3.12 Résultats : Les principaux choix architecturaux

Nous abordons maintenant les choix architecturaux principaux que nous avons pu relever au sein des différents projets. Pour ce faire, nous lions à chaque fois une structure architecturale à un facteur d'influence en tant que raison ayant poussé au choix. Nous commentons également le succès du choix à répondre aux problématiques l'ayant imposé.

Deux réponses à la nécessité de modularité

Beaucoup de facteurs d'influence sus-mentionnés impliquent une bonne modularité du logiciel. Ceci permet de faciliter la séparation des tâches entre les différentes équipes travaillant sur un logiciel, qu'elles soient formées sur base

d'une répartition géographique, d'objectifs de recherche communs ou pour gérer un grand nombre de contributeurs. Afin de répondre à cette nécessité, les projets étudiés adoptent systématiquement l'approche de flux de données ou des plugins, il arrive même que certains projets utilisent les deux.

Nous commençons par l'architecture en flux de données, sa représentation la plus courante est le *Pipe-and-Filter* (Bass *et al.*, 2012, p. 239). Une autre représentation est le *batch* ou exécution de type *Unix* qui reste extrêmement similaire. En effet, cette dernière est souvent utilisée pour permettre d'effectuer des traitements consécutifs dans un *pipe*, autrement dit une chaine d'instructions, exprimé en ligne de commande. Nous retrouvons alors une logique similaire au Pipe-and-Filter si ce n'est que les pipes ne font pas partie du programme et que l'interopérabilité entre filtres est garantie par le format de fichiers intermédiaires. Sur les cinq projets ayant participé aux interviews, quatre ont recours au style d'architecture en flux de données comme structure principale.

La seconde solution est l'architecture sur base de plugins. Le seul projet ne se basant pas sur le Pipe-and-Filter est d'ailleurs basé sur un framework accueillant des plugins destinés à effectuer des traitements variés. Deux autres projets utilisant le flux de données comme structure principale ont adopté un système de plugins prenant la place des filtres.

Réutilisation et génération de code

Trois projets ont été confrontés à des problèmes de duplication de code source. Par exemple, les contributeurs d'mCRL2 avaient l'habitude de copier un outil existant pour le modifier au lieu d'en créer un nouveau. Nous pouvons également citer Rascal qui avec son nombre croissant de primitives, aujourd'hui plus de 400, devait multiplier les analyses au sein de son interpréteur. Afin de pallier à ces problèmes, deux méthodes sont appliquées au sein des trois projets : la génération de code et la production de librairies. La première méthode se base généralement sur un pattern de substitution ou du visiteur qui permet de générer l'ensemble des classes structurelles lors de la création d'un nouvel outil ou d'un nouveau noeud dans un arbre syntaxique. La deuxième sert à stocker l'ensemble des routines qui sont utilisées par plus de deux composants ou plus de deux classes dans le cas de l'orienté objet. Deux des projets utilisent les deux méthodes et un se contente uniquement de la génération de code.

Le choix des design patterns

Alors que les choix architecturaux que nous avons abordés jusque ici ont globalement rempli leurs objectifs, nous voulons nous attarder sur le cas de Rascal.

Lorsque les concepteurs ont choisi d'utiliser le pattern visiteur pour géné-

rer la structure de leur arbre syntaxique, ce choix semblait tout indiqué. Ils pouvaient générer le code redondant et la propagation des modifications dans la structure des classes du visiteur devait permettre de localiser les éléments à corriger dans les classes d'opérations grâce aux erreurs de référencement.

Alors que cette méthode semblait idéale sur le papier, l'interviewé a qualifié cette décision d'erreur. En effet, le pattern visiteur est assez mal supporté par Eclipse, l'IDE dans lequel Rascal est implémenté sous forme de plugins. Par conséquent, la détection des erreurs qui devait servir à localiser le code à éditer s'est transformée en cauchemar avec la propagation des erreurs de références dans toutes les classes d'opérations, ne permettant plus de distinguer la portion de code fautive.

Une solution fut d'appliquer un pattern d'interpréteur à la place du pattern visiteur, cette transformation put être automatisée en Rascal et fit d'ailleurs l'objet d'une publication (Hills *et al.*, 2011) de la part des concepteurs de Rascal. Cette solution eut un effet relativement positif mais pas parfait. Néanmoins, le compilateur devant être distribué peu de temps après, les travaux sur l'interpréteur se sont arrêtés.

Cet exemple nous intéresse beaucoup car il montre la difficulté de choisir la bonne méthode pour une tâche donnée même en se basant sur la littérature existante.

Un interpréteur pour développer un langage

Nous nous penchons à nouveau sur Rascal qui nous a fourni un autre exemple de décision architecturale spécifique : l'implémentation successive d'un interpréteur et d'un compilateur dans le cadre du développement d'un langage.

Au début, le compilateur semblait l'idéal pour promouvoir un langage, un compilateur étant plus performant et permettant une optimisation plus efficace du code. Néanmoins, le choix fut fait de commencer par produire un interpréteur. En effet, un interpréteur permet d'expérimenter rapidement et d'appréhender des éléments partiels du langage là ou un compilateur doit comprendre un sous-langage fonctionnel avant toute utilisation.

A postériori, l'équipe de développement de Rascal fut assez satisfaite de ce choix, développer l'interpréteur avant le compilateur offrait beaucoup plus de flexibilité pour s'adapter à l'évolution permanente du langage à ce stade du projet.

La où ce choix nous parait très intéressant du point de vue des logiciels académiques c'est que dans un objectif de production, développer le compilateur immédiatement aurait abouti à un produit *fini* plus rapidement. Néanmoins, une équipe de recherche ne peut travailler plus de quelques mois sans expérimenter et la production directe d'un compilateur aurait justement bloqué cette possibilité d'expérimentation pendant une trop longue période. A nou-

veau, nous illustrons avec cet exemple l'impact du contexte académique sur les décisions au cours de la conception d'un logiciel.

3.3.13 Résultats : L'évolution des architectures

Dans cette section, nous reprenons quelques cas d'évolution qui nous ont particulièrement marqué lors des interviews. Ils sont souvent des témoignages éloquents d'une évolution des facteurs qui influent sur la conception au cours de l'existence du projet et permettent ainsi d'établir des liens supplémentaires entre décisions et facteurs d'influence.

L'évolution des facteurs d'influence, un cas pratique

L'évolution d'un des projets nous a particulièrement marqué. Alors que l'ensemble des projets interrogés évoluaient peu ou très lentement, celui-ci a radicalement changé au cours de son existence à cause de modifications fondamentales des facteurs d'influence durant le cycle de vie du logiciel.

Ce projet a commencé son existence en tant que *hobby* d'après les dires de son concepteur. Ce dernier démarre le projet seul et y travaille quand il en a le temps. A ce moment-là, l'architecture n'est pas une priorité, le créateur du logiciel veut avoir les meilleures performances possibles et y investit tous ses efforts. Même lorsqu'il commençe à travailler à temps plein sur son logiciel en explorant des possibilités de publication à son sujet, la croissance de l'architecture reste organique par ajouts successifs des composants désirés sans considération pour une structure générale.

Avec le temps, le projet finit par intéresser une entreprise qui décide d'y investir de l'argent. Le concepteur du logiciel peut alors bénéficier de ressources supplémentaires qui servent à la fois à améliorer la structure mais aussi à implémenter des fonctionnalités supplémentaires pour l'investisseur.

Le troisième changement clé est l'obtention d'une bourse de recherche. Celle-ci peut être consacrée à revoir l'architecture de fond en comble, tâche initiée avec le financement de l'entreprise, et à investir plus sur sa maintenabilité et sa capacité à évoluer.

Nous avons trouvé cet exemple parfaitement adapté pour montrer comment des facteurs extérieurs peuvent influencer l'architecture. D'une architecture organique tournant autour d'un parser très performant, l'architecture a évolué vers une architecture composée de plusieurs composants avec des APIs très claires pouvant chacune être utilisée indépendamment dans d'autres projets de recherche.

Autres cas d'évolution

Lors de nos interviews, nous avons évoqué d'autres cas d'évolution liés aux facteurs d'influence que nous évoquons ci-dessous.

Pour le premier cas, nous parlerons de la problématique des licences. Dans le cadre d'un des projets, l'architecture fut fondamentalement transformée entre deux distributions afin de permettre d'obtenir une plus grande liberté au niveau des licences. D'une distribution monolithique encapsulant des modules appelés plugins, le logiciel évolua vers une architecture basée sur des plugins distribués indépendamment du coeur du logiciel. De cette manière, les plugins pouvaient disposer de licences de distribution différentes de celle du coeur du logiciel.

Nous évoquons rapidement Rascal. En effet, l'exemple que nous avons dépeint au sujet de l'évolution d'un interpréteur vers un compilateur est aussi un cas d'évolution de l'architecture dictée par les facteurs d'influence académiques.

Enfin nous aimerions revenir sur la tendance à ajouter les plugins à postériori. Comme nous l'avons expliqué plus tôt, trois projets sur les cinq utilisent des plugins. Pourtant ces plugins ont tous été ajoutés dans des versions tardives du logiciel du point de vue de son évolution globale. Nous expliquons cette situation par la croissance des logiciels académiques. Il est naturel pour un logiciel d'évoluer et de grandir mais c'est d'autant plus vrai dans le contexte académique où les logiciels évoluent souvent au gré des travaux de recherche et des publications successifs. Souvent, le logiciel n'est pas pensé pour croitre autant qu'il ne finit par le faire et le besoin en modularité devient de plus en plus pressant. L'adoption de plugins permet souvent d'étendre le logiciel sans trop toucher à l'existant.

3.3.14 Résultats : Les autres décisions récurrentes

Finalement, nous abordons ici les différentes décisions n'ayant pas un lien direct avec l'architecture mais ayant une influence importante sur celle-ci ou sur l'ensemble du projet. Elles reprennent entre autres les questions de stratégies de tests ou de contributions des différents projets.

L'importance et l'automatisation des tests

L'avis des personnes interviewées est unanime quant à l'importance de bien tester les logiciels pour s'assurer de la non régression de ceux-ci.

Parmi les méthodes de tests les plus récurrentes, nous pouvons mentionner l'automatisation des tests qui apparait dans quatre projets sur cinq.

Vient ensuite le bootstrap, méthode consistant en l'implémentation d'un compilateur ou d'un interpréteur dans le langage à interpréter/compiler, il est

présent dans les deux projets incluant un langage permettant la transformation immédiate de programmes.

Enfin une méthode moins habituelle fournie à nouveau par Rascal est la méthode de tests aléatoires.

La gestion des contributions

Nous remarquons ici des disparités entre les différentes politiques. Chaque projet gère les contributions à sa manière, mais seul un projet a mentionné avoir utilisé des outils et du versioning pour la gestion des contributions. Les autres projets semblent plus axés sur une vérification voire un accompagnement des contributions afin de s'assurer de leur concordance avec les méthodes et pratiques du projet.

Nous remarquons également que le contexte scientifique pousse à accepter des contributions mal documentées ou ne disposant pas des tests nécessaires. Souvent ces contributions ne sont pas maintenues ou sont gérées par l'équipe de développement principal si la contribution s'avère critique pour l'utilisation du logiciel.

La gestion de la distribution et du support

Nous remarquons de manière générale que les logiciels académiques adoptent plus une distribution du type *diffusé quand c'est fini*. Néanmoins les équipes de développement tentent de maintenir une distribution régulière variant de 6 à 18 mois selon les projets.

Une particularité que nous avons relevée semble être la quasi totale absence de support des logiciels. Dans tous les cas que nous avons abordés, les logiciels distribués ne sont mis à jour ou patchés que s'ils présentent des anomalies trop graves pour en reporter la correction à la version suivante.

3.3.15 Conclusion

Nous achevons ici la présentation des résultats issus des interviews. Nous reviendrons sur ceux-ci dans le prochain chapitre afin de les synthétiser et d'en donner notre interprétation. Dans la prochaine section, nous présentons le protocole et les résultats de notre enquête.

3.4 Enquête

Cette section présente la dernière étape de notre collecte de données : l'enquête. Cette enquête doit nous permettre d'étendre les résultats de nos interviews en ajoutant des informations au sujet de projets différents. Pour la réaliser, nous utilisons les interviews afin de définir les points les plus pertinents

à aborder et la meilleure manière de formuler des questions sans ambiguïté à leur sujet.

Dans la suite de cette section, nous présentons notre protocole d'enquête et les résultats que nous avons obtenus. Nous commentons également rapidement les formulaires des deux versions de l'enquête et donnons le nombre de participants sollicités et le nombre de réponses obtenues. Le chapitre suivant permettra de synthétiser et analyser les résultats conjoints de l'enquête et des interviews.

3.4.1 Recherche de participants

Les projets participants sont recherchés via Google Scholar car les critères de sélection incluent l'existence de publications décrivant le projet. Pour le premier formulaire, la vérification des critères est effectuée avant l'envoi du formulaire afin d'avoir une quasi certitude de réponse. Afin d'élargir le nombre de projets et de réduire la charge de travail, la vérification des critères est intégrée au second formulaire, de cette manière un projet intéressé par l'étude mais ne remplissant pas les critères peut nous laisser ses coordonnées en vue de travaux futurs.

Les mots clés utilisés pour trouver les projets sur Google Scholar sont les suivants :

— "Software analysis" tool
— "Software transformation" tool
— allintitle : "analysis program"
— allintitle : "program analysis"
— allintitle : "transformation program"
— allintitle : "program transformation"

Au début, certains projets ont été trouvés sur base de recommandations mais ils ont tous été retrouvés suite aux recherches mentionnées ci-dessus.

Les critères pour retenir un projet sont que nous trouvions une publication décrivant le projet avec les mots-clés suscités, de cette manière cette recherche est assez facilement reproductible.

Comme nous nous attendons à un faible nombre de résultats, nous demandons aussi aux projets sollicités s'ils ont connaissance d'autres projets correspondant aux critères mais aucun autre projet ne nous a été indiqué de cette manière.

3.4.2 Les formulaires

Notre enquête a été diffusée deux fois avec une mise à jour du formulaire entre les deux diffusions. Le premier formulaire est mis à disposition sur Google Form et envoyé à huit projets différents pour obtenir de nouvelles réponses. Il

est également envoyé aux cinq projets qui ont fait l'objet d'une interview afin d'obtenir un retour de ceux-ci vis-à-vis des interviews.

Nous avons obtenu une réponse positive pour trois des huit projets mais le formulaire n'a été rempli que par deux d'entre eux. Des membres de deux des projets ayant participé à une interview ont également rempli le formulaire en donnant leurs commentaires à la fin de celui-ci.

Sur base des réponses et commentaires obtenus, une mise à jour du formulaire est réalisée pour lever certaines ambiguïtés et le nouveau formulaire est envoyé à quatorze projets mais nous n'avons obtenu que deux réponses positives et aucun formulaire n'a été renvoyé.

Présentation du premier formulaire

L'idée du premier formulaire présenté dans l'annexe 5.3 est de coller d'assez près aux questions et sujets abordés lors des interviews.

Après avoir présenté l'étude et demandé des informations générales sur le projet, nous demandons de décrire l'architecture du projet. Afin de guider les participants, nous mentionnons l'importance que nous portons aux choix architecturaux, aux motivations qui les guident et au succès ou non de ceux-ci.

Nous demandons ensuite quelles sont les motivations générales du projet afin de voir quels sont les facteurs qui ont influencé la conception de l'architecture.

La section suivante demande de faire une auto-évaluation de la conception du logiciel, le but étant de récolter des informations pertinentes au sujet des conseils que nous pourrions nous-mêmes donner et former dans le guide technique.

Les sections qui clôturent cette enquête sont destinées à réunir des informations périphériques qui ne sont pas forcément liées à l'architecture en elle-même mais ressortent systématiquement dans les discussions avec les personnes interviewées lorsqu'on leur parle de l'architecture et des choix de conception.

Nous finissons en demandant aux participants un feedback sur l'enquête ainsi que la manière dont ils souhaitent voir leur participation et leur projet référencés dans nos travaux.

Présentation du second formulaire

Les réponses apportées au premier formulaire et les retours à son sujet nous ont poussé à le modifier avant une plus large diffusion. Pour cette deuxième diffusion, nous adoptons le formulaire PDF, ce dernier est fourni dans l'annexe 5.3, pour permettre plus de flexibilité aux participants. Il leur est ainsi possible de ne remplir que partiellement le formulaire, de le remplir en plusieurs fois ou encore de le remplir sur papier si cela leur convient mieux.

Le premier élément revu en profondeur concerne la récolte des données générales à propos du projet. Comme nous ne vérifions plus tous les critères avant de demander une participation, cette section est enrichie afin de permettre aux participants comme à nous-mêmes de savoir facilement sur base des informations générales si le projet répond aux critères de sélection. De cette manière, une personne souhaitant participer évitera de remplir l'entièreté du questionnaire si son projet ne répond pas aux critères. Nous proposons également aux projets de signaler leur intérêt pour notre recherche. De cette manière, qu'ils répondent ou non aux critères, il sera possible de les recontacter si leur projet correspond aux critères de sélection d'une future étude.

Ensuite, nous remarquons la difficulté de certains participants à la première version de l'enquête à formuler les choix architecturaux, leurs réponses étant parfois très vagues. Nous dressons donc une liste de styles d'architecture communs dans laquelle les participants peuvent cocher ceux qui sont implémentés dans leur projet. Nous leur demandons ensuite des informations qui nous permettent de reconstruire les choix architecturaux sur base des réponses fournies.

La section à propos du design et des motivations est revue afin d'être plus concise et plus facile à remplir, la version originale étant fort redondante et assez longue ce qui la rendait quelque peu confuse.

Il n'y a pas d'autre changement majeur dans cette nouvelle version.

3.4.3 Les participants

Comme pour les interviews, nous allons maintenant présenter les participants à notre enquête. Les projets ayant participé à la fois à l'enquête et aux interviews sont mCRL2 et Rascal.

3.4.4 Projets : Marple

Marple (Arcelli *et al.*, 2008) est un logiciel qui prend la forme d'un plugin Eclipse permettant la détection de design patterns et d'activités de reconstruction de l'architecture grâce à des métriques et informations extraites du code source. Marple est un projet du laboratoire d'étude de l'évolution des systèmes logiciels et de rétro-ingénierie de l'Université de Milano-Biccoca.

3.4.5 Projets : CScout

CScout (Spinellis, 2010), comme son nom l'indique, est un explorateur de code source destiné au langage C. Ses fonctionnalités de base permettent la détection des refactoring dans le code et il dispose de mécaniques lui permettant des analyses du code plus poussées. Il s'agit d'un très petit logiciel développé par une personne seule.

3.4.6 Résultats

Nous allons maintenant résumer les résultats que nous avons obtenus dans le cadre de l'enquête. Nous suivrons une structure proche de celle de la présentation des résultats des interviews afin de mentionner si nous avons reçu des informations nouvelles ou similaires aux anciennes.

3.4.7 Résultats : Apport des participants aux interviews

Les retours obtenus pour Rascal et mCRL2 sont cohérents vis-à-vis des résultats des interviews. Un des participants a révélé le manque de précision de certaines questions mais dans l'ensemble, ils nous ont permis d'avoir un très bon retour sur la manière dont notre enquête était rédigée tout en pointant les éléments à améliorer.

3.4.8 Résultats : Les facteurs influençant l'architecture

A ce niveau, les nouveaux résultats restent fort similaires à ceux que nous avons obtenus pour les interviews. On retrouve cette fois des projets principalement guidés par les objectifs scientifiques et la facilité de compréhension et d'évolution. Il est à noter qu'un accent particulier a été mis sur les performances pour un des projets.

Nous remarquons néanmoins qu'avec une taille moindre, cinq et un contributeurs, les nouveaux projets se focalisent plus sur les capacités d'évolution du logiciel que sur sa compréhensibilité et la facilité d'y contribuer. Ce résultat ne nous a pas étonné car nous avions déjà vu des recherches avançant que ces deux objectifs pouvaient être conflictuels (Harrison *et al.*, 2000), il nous a paru logique de privilégier la capacité à faire évoluer le logiciel facilement sur celle de permettre à un improbable nouveau contributeur de participer, nous ne pouvons malheureusement pas confirmer cette intuition sur base des réponses au formulaire.

3.4.9 Résultats : Les principaux choix architecturaux

Un nouveau choix que nous avons pu extraire des formulaires est celui de l'agencement des composants d'un logiciel autour d'un modèle commun. Ainsi nous avons une structure qui prend la forme de trois unités de traitement dédiées à une tâche spécifique qui partagent leurs informations dans un modèle et dont les exécutions sont organisées par un configurateur. Cette solution nous fait penser au *Shared-Data Pattern* (Bass *et al.*, 2012, p. 252) adjoint d'un configurateur bien que l'échelle soit beaucoup plus petite. Cette situation s'explique par une intégration qualifiée d'instrumentale dans le formulaire.

Pour l'autre projet, nous retrouvons à nouveau le style Pipe-and-Filter sans plus de détails si ce n'est l'usage de classes polymorphiques.

3.4.10 Résultats : Autres éléments

Les formulaires de l'enquête ne nous ont malheureusement pas fournis d'autres informations qu'ils s'agissent de l'évolution du logiciel ou d'autres éléments particuliers. Les différentes politiques sont elles aussi similaires à celles des projets ayant participé aux interviews.

3.4.11 Conclusion

C'est sur ces résultats que se termine notre chapitre à propos de nos activités de recherche. Dans le chapitre suivant, nous synthétisons et interprétons les résultats obtenus.

Chapitre 4

Synthèse et interprétation des résultats

Dans ce chapitre, nous synthétisons et interprétons les résultats que nous avons présentés au chapitre précédent. Pour ce faire, nous procéderons en utilisant des tables pour donner une vue globale et synthétique des résultats et nous interpréterons ensuite les résultats en commentant ces tables. Nous présentons d'abord les facteurs d'influence, nous poursuivons avec les choix architecturaux et nous terminons avec les évolutions et les éléments externes à l'architecture. Afin d'éviter toute ambiguïté, nous rappelons que nous avons étudié un total de sept projets.

4.1 Les facteurs influençant l'architecture

Dans cette section, nous abordons les facteurs qui influencent les décisions architecturales au sein des projets.

Dans la table 4.1, nous retrouvons les différents facteurs d'influence et leur fréquence. La colonne *Mentions* reprend le nombre de projets pour lesquels le sujet est abordé dans les interviews ou l'enquête alors que les deux colonnes suivantes présentent le nombre de projets pour lesquels ce facteur aboutit à un choix spécifique. La première de ces deux colonnes décompte les projets pour lesquels il y a eu un impact, quel qu'il soit, alors que la deuxième ne reprend que le nombre de projets dont l'architecture a aussi été impactée par le facteur d'influence.

Dans un premier temps, nous notons l'importance donnée à la maintenabilité et à la possibilité d'évolution du logiciel. Ainsi les facteurs tels que la compréhensibilité, la modifiabilité et la facilité de contribution sont mentionnés dans toutes les interviews et formulaires d'enquête et ont très souvent un impact sur le logiciel et son architecture. Ces facteurs sont importants. Ils le sont aussi dans les autres domaines de l'ingénierie logicielle. Mais nous pen-

Facteurs d'influence	Mentions	Impacts effectifs	
		Projet	Architecture
Gestion des licences	5	2	1
Objectifs de recherche	5	4	2
Manque de formation	2	0	0
Renouvellement des contributeurs	4	3	0
Rép. géographique des contributeurs	1	1	1
Méthodes et outils	2	2	1
Utilisabilité du logiciel	2	2	1
Compréhensibilité du logiciel	7	5	5
Modifiabilité du logiciel	7	7	7
Facilité de contribution	7	7	4
Possibilité de publication	3	2	1
Source de financement	4	1	1

TABLE 4.1 – Facteurs influençant les logiciels académiques

sons que le contexte académique accentue cette situation car les contributeurs sont très nombreux et variés et que le logiciel doit pouvoir s'adapter en permanence à de nouveaux objectifs de recherche. Ces objectifs de recherche semblent d'ailleurs avoir aussi une incidence importante sur les logiciels et parfois sur les architectures.

Ensuite viennent un ensemble de facteurs qui sont abordés régulièrement mais ne donnent pas suite à des changements dans le logiciel ou son architecture. Mise à part la gestion des licences, ces facteurs sont spécifiques au monde académique comme les sources de financement, la possibilité de publication ou encore le renouvellement des contributeurs. Parmi ceux-ci, le renouvellement des contributeurs semble avoir l'impact le plus significatif sur la manière de penser les logiciels mais n'en a pas sur la conception de l'architecture. Notre explication à ce phénomène est que d'une part ces facteurs sont pris en compte par le biais des facteurs de maintenabilité cités plus haut et que d'autre part les chercheurs pensent souvent que ces problématiques ne peuvent être solutionnées par le biais de l'architecture, c'est notamment le cas pour le renouvellement des contributeurs.

Finalement, il y a quelques problématiques qui semblent être plutôt spécifiques à certains projets. Il est intéressant de souligner que ces problématiques sont souvent résolues aussi bien au travers du logiciel que de l'architecture. C'est le cas pour la répartition géographique des contributeurs ou les problèmes liés aux méthodes et outils inadaptés.

4.2 Les choix architecturaux

Dans cette section, nous revenons sur les choix architecturaux que nous avons observés dans nos résultats.

Choix architecturaux	Implémentations	Succès	Échecs
Flux de données	5	5	0
Plugins	3	3	0
Génération avec design pattern	3	3	1
Librairies	2	2	0
Interpréteur avant compilateur	1	1	0
Modèle partagé	1	-	-

TABLE 4.2 – Choix architecturaux

La table 4.2 met en évidence la diversité des solutions employées. Elle confirme la volonté de modulariser au maximum afin d'améliorer la maintenabilité et l'évolutivité des logiciels académiques avec l'utilisation du flux de données et des plugins. Nous remarquons aussi un attrait pour la génération de code grâce à des design patterns. A ce sujet, nous voyons apparaitre une mise en garde : le choix du design pattern est alors primordial. Prenons Rascal pour exemple, cette génération ne s'est pas bien déroulée à cause d'un pattern visiteur inadapté aux outils utilisés. Ce choix est donc repris à la fois comme un succès et un échec dans le tableau car d'un coté, la génération de code a été un succès mais de l'autre, elle a aussi entrainé des problèmes au niveau de l'évolution du logiciel.

Un autre élément que nous remarquons est la réussite de la quasi totalité des choix architecturaux. En effet, bien que nous demandions des informations sur l'évolution du logiciel, c'est bien souvent l'absence de choix qui causait des problèmes dans les versions antérieures. A part le cas du pattern visiteur pour Rascal, nous n'avons pas eu d'autre exemple de choix murement réfléchi ayant mal tourné.

4.3 Les évolutions des logiciels et autres décisions

Les évolutions des logiciels étant spécifiques à chaque projet, nous estimons qu'un tableau synthétique ne serait pas informatif. Dans l'ensemble, nous voyons que ces évolutions peuvent être causées par trois raisons : l'absence de structure antérieure répondant à la même problématique, l'émergence d'une nouvelle problématique ou l'échec d'une solution antérieure.

Quant aux autres décisions, nous les présentons selon quatre axes différents.

Ainsi nous retrouvons la politique de test, la politique de contribution, la politique de diffusion et la présence de débouchés commerciaux.

Test unitaire	1
Test unitaire automatisé	3
Test fonctionnel [1]	5
Test fonctionnel automatisé	4
Implémentation d'un framework de test	5
Test aléatoire	2
Test "bootstrap"	2

TABLE 4.3 – Politiques de test

La table 4.3 nous montre l'importance donnée aux tests dans les projets académiques. Alors que les tests unitaires sont légèrement en retrait, nous remarquons la forte propension à implémenter un framework de tests et à effectuer des tests fonctionnels. De plus, les tests unitaires sont presque systématiquement automatisés alors que les tests fonctionnels sont à la fois automatisés et manuels dans une bonne proportion des logiciels. Nous remarquons également que des méthodes peu communes ou spécifiquement adaptées au problème sont parfois utilisées avec les tests aléatoires et les tests "bootstrap". Cette propension à améliorer les tests semblent directement liée aux mêmes motivations que celles aboutissant à l'utilisation d'une modularisation importante.

Au niveau des contributions, nous sommes à nouveau devant un cas de statu quo entre deux approches. D'un coté, nous avons trois projets qui limitent les contributions à l'équipe principale uniquement et quatre autres qui laissent tout le monde contribuer, quitte à vérifier à postériori si la contribution peut être conservée. Même dans le cas de contributions libres, les contributeurs sont toujours soumis aux accords de licences et aux bonnes pratiques des différents projets.

Par rapport à la politique de distribution, nous avons d'une part, 3 projets qui adoptent un rythme régulier, tous les 6 à 18 mois et d'autre part, 4 projets qui sont distribués "quand ils sont prêts". De plus, il n'y a que deux équipes qui ont mentionné le maintien de branches multiples simultanément et une seule qui assure le support du logiciel distribué au-delà de l'extrême nécessité. Nous expliquons cela par le fait que les logiciels académiques sont le plus souvent destinés aux programmeurs eux-même qui les utilisent pour leurs recherches. Il est alors souvent plus intéressant de distribuer une nouvelle version disposant de nouvelles fonctionnalités destinées à la recherche que de maintenir des versions ayant déjà rempli leur rôle.

Pour finir, abordons les débouchés commerciaux. Parmi les sept projets

étudiés, cinq sont sujets à des débouchés commerciaux dont trois l'avaient planifié dés le début. Les deux projets n'étant pas sujets à ces débouchés n'excluent pas que cela arrive dans le futur. Ceci montre un peu plus l'importance de la commercialisation des logiciels même dans le cadre des logiciels académiques, ce qui peut s'expliquer par le besoin de financement de la recherche et des institutions et renforce la nécessité de revoir certains aspects des logiciels quand un débouché commercial est prévu.

4.4 Résumé

Maintenant que nous avons analysé l'ensemble de nos résultats, nous pouvons résumer les points qui nous semblent primordiaux.

Pour commencer, nous avons vu que les logiciels académiques tendent à changer énormément et la plupart des décisions de conception et des décisions stratégiques sont prises en tenant compte de cette situation. Tout d'abord ils adoptent des architectures fournissant un maximum de modularité grâce au style en flux de données et aux plugins, mais aussi ils essayent de générer une partie du code source grâce à des design patterns tels que le design pattern du visiteur ou de substitution. Ensuite ils adoptent des pratiques telles que les tests automatisés et s'abstiennent de supporter les logiciels distribués, préférant attendre la version suivante pour corriger les bugs.

Par ailleurs, nous avons également croisé un ensemble de problématiques et motivations qui rendent le logiciel académique spécifique. Toutefois, elles semblent assez peu prises en compte par les concepteurs de logiciels académiques. Ces derniers en ont pourtant conscience et chaque logiciel que nous avons étudié propose une solution à l'une ou l'autre d'entre elles. Malheureusement, les logiciels sont rarement adaptés à l'ensemble des problématiques auxquels ils sont confrontés.

Nous achevons ici notre synthèse et notre analyse des résultats, nous clôturons cette étude dans le chapitre suivant en parlant des menaces à la validité, en présentant une rétrospective et en abordant les perspectives d'avenir.

Chapitre 5

Conclusion

Avant de revenir sur le chemin parcouru dans ce travail de recherche et de tirer des conclusions, nous souhaitons prendre un peu de recul sur notre étude et envisager les éléments qui pourraient représenter une menace à la validité de nos recherches. Nous pourrons ensuite poursuivre avec notre rétrospective qui sera suivie par les perspectives d'avenir.

5.1 Menace à la validité et prise de recul

Au cours de cette recherche, nous nous sommes rendu compte à plusieurs reprises que certaines étapes auraient pu être menées plus en profondeur ou de manière différente pour obtenir de meilleurs résultats. Nous avons aussi remarqué des éléments qui nous poussent à émettre quelques réserves quant à la validité de nos résultats. C'est pourquoi nous allons vous faire part de cette prise de recul et de ces réserves avant de passer à notre rétrospective.

Commençons par un élément que nous aurions pu améliorer. Lors de l'analyse de nos résultats, nous avons remarqué une certaine inconsistance dans ceux-ci, les informations récupérées grâce aux interviews étaient fort différentes d'un projet à l'autre. Nous expliquons cela par le manque d'expérience en interview et le manque de clarté du sujet au début des interviews car nous n'avions pu réunir d'informations via la revue systématique de littérature. A cause de cela, il était difficile d'embrayer sur les bons sujets et d'être réactif. La connaissance du sujet et l'expérience étant acquises au cours des interviews, la qualité de celles-ci augmentait au fil du temps. Nous espérions pallier à ce problème en obtenant plus d'informations via l'enquête mais nous avons eu très peu de réponses à celle-ci.

Ceci nous amène au deuxième élément : l'enquête bien que mieux cadrée dépend beaucoup trop de la bonne volonté et du temps dont dispose le participant. A cause de cela, nous avons reçu deux formulaires très complets et deux formulaires fort lacunaires. Ceci a rendu le travail d'extraction des données

très difficile.

Enfin nous avons eu des problèmes au niveau de la synthèse des résultats à cause de cette inconsistance. Nous en sommes parvenu à la conclusion que la meilleure solution serait de combiner les deux méthodes de récolte de données en fournissant un formulaire court et concis reprenant les questions fermées auxquelles nous voulons obtenir des réponses pour chaque projet, suivrait une interview plus courte mais centrée sur les éléments spécifiques au projet ou demandant des explications plus poussées. Il n'était malheureusement pas possible d'envisager une telle solution au début de la recherche car nous n'aurions pas pu établir des formulaires satisfaisants. C'est grâce aux informations que nous possédons maintenant que nous pourrions le faire.

Un autre point qui pourrait être discuté sont les critères de sélection des projets. Bien que nous ayons envoyé le formulaire à une quinzaine de personnes lors de la deuxième diffusion, nous n'avons reçu que deux réponses et malheureusement elles n'ont pas débouché sur le retour d'un formulaire. Alors que nous pensions les critères raisonnables, la proportion de réponses en cas d'enquête semble trop faible pour s'en contenter. En effet, au cours de notre recherche nous avons eu le sentiment que dégager une vision globale des architectures logicielles, même dans un domaine aussi réduit que l'analyse et la transformation de programmes, serait difficile voire impossible sans une bonne quantité de résultats. Ce problème aboutit à l'impossibilité de généraliser les résultats que nous avons obtenus vu le petit nombre de participants.

C'est à ce sujet que nous émettons notre plus grande réserve : le faible nombre de participants ne nous permet pas de différencier les cas particuliers de cas qui seraient courants mais non systématiques. De plus, à l'exception des facteurs d'influence liés à la maintenabilité, nous n'avons jamais d'unanimité ce qui rend la généralisation des résultats hasardeuse vu qu'ils ne proviennent que de trois à cinq projet sur sept.

Néanmoins nous pensons que les résultats obtenus sont suffisants pour montrer un ensemble de pointeurs à surveiller et servir de base à une étude plus approfondie du sujet mais nous reviendrons sur ce point dans les perspectives d'avenir.

5.2 Rétrospective

Nous arrivons maintenant au terme de notre étude et il est temps de revenir sur le chemin parcouru et les résultats obtenus afin de pouvoir sereinement envisager les suites à donner à ce travail.

Dans cette recherche, nous avons étudié les architectures des logiciels académiques. Nous avons commencé par présenter notre vision des logiciels académiques et l'approche des architectures comme un ensemble de choix archi-

tecturaux soutenus par différentes motivations afin de répondre à un ensemble de problématiques.

Nous avons rapidement envisagé les différentes études existantes sur le sujet et nous avons constaté l'absence de guide pratique qui pourrait être utilisé comme référentiel pour l'architecte de logiciel académique. Ceci nous a permis de confirmer la nécessité d'un tel guide mais aussi de nous indiquer un ensemble de pointeurs à surveiller dans notre recherche.

Dés lors, nous avons pu commencer la recherche à proprement parler. Nous avons procédé par étapes en commençant par une revue systématique de littérature destinée à vérifier si des publications avaient déjà traité de notre sujet de recherche par le passé. N'ayant pas trouvé de publications correspondant à ce que nous voulions explorer, nous avons embrayé sur la deuxième étape de notre projet qui prit la forme d'une série d'interviews destinées à explorer quelques projets afin d'en apprendre plus sur leur architecture, les problématiques rencontrées et les autres facteurs qui influencent les décisions. Ces interviews furent complétées par une enquête orientée vers les sujets dont nous avions discuté lors des interviews. Au total, sept projets furent étudiés et nous avons ainsi pu tirer quelques conclusions des résultats obtenus.

Pour commencer nous avons remarqué que la motivation principale des architectes de logiciels académiques était la production d'architectures maintenables et évolutives. Pour ce faire, les structures que nous avons le plus rencontrées étaient le flux de données et les plugins accompagnés de la génération de code source et de l'utilisation de librairies pour encadrer la réutilisation de code source. Dans le même ordre d'idée, les tests semblent avoir une très grande importance dans ce type de projets et nous retrouvons à la fois une forte automatisation de ceux-ci et des méthodes moins communes comme le test par bootstrap et les tests aléatoires. Nous avons également constaté que les équipes de développement négligent les facteurs spécifiques aux logiciels académiques malgré une connaissance assez répandue de ceux-ci, ceci s'expliquant par leur impression que les architectures logicielles et les méthodes de développement ne sont pas des solutions forcément adaptées à ces facteurs.

Nous venons finalement de clôturer cette recherche en abordant les difficultés de généralisation sur base d'aussi peu de résultats et les améliorations possibles que l'on pourrait apporter à notre protocole, comme la conduction d'une forme hybride d'enquête correspondant à un court formulaire dirigé suivi d'une interview.

Comme nous l'expliquions au début de cette recherche, notre objectif final était de pouvoir fournir un guide d'ingénierie logicielle destiné au contexte académique. Bien que nous nous soyons concentré sur la découverte des spécificités des logiciels académiques et de leurs influences sur l'architecture de tels logiciels, nous donnons quelques pistes dans la section suivante qui permettraient de poursuivre cette étude afin de se rapprocher de cet objectif global.

5.3 Perspectives d'avenir

Les possibilités pour poursuivre ce travail sont nombreuses et nous ne pourrions toutes les énumérer. Nous souhaitons malgré tout en présenter quelques-unes qui nous paraissent plus pertinentes que les autres. Elles regroupent à la fois des travaux complémentaires aux nôtres et des travaux qui poursuivent les nôtres dans un objectif commun de se rapprocher de la rédaction d'un guide d'ingénierie des logiciels académiques.

Pour commencer, nous pensons qu'il serait plus que bienvenu de fournir un travail de compilation des approches existantes. Comme nous l'avons vu lors de notre étude du contexte, il existe déjà des modèles représentant le développement dans un contexte académique, des lignes directrices et des listes de problématiques à surveiller. Il nous paraitrait très intéressant de compiler ces travaux afin de réaliser la première partie du guide destinée à donner des lignes directrices aux architectes de logiciels académiques.

Dans un second temps, nous pensons que notre recherche doit être étendue car nous avons remarqué que les logiciels académiques et les problématiques auxquelles ils répondent sont très variés et que jusqu'à présent les architectures de ces logiciels ne sont pas suffisamment étudiées.

A ce stade, nous disposerions de la première partie du guide pour l'ensemble des logiciels académiques et de recommandations concrètes pour les logiciels d'analyse et de transformation de programmes. Si les résultats obtenus sont concluants, il serait alors très intéressant d'étendre cette recherche à d'autres domaines d'applications afin de pouvoir à terme aider l'ensemble des chercheurs s'appuyant sur l'outil informatique.

Bibliographie

Ambati, V. et Kishore, S. (2004). How can academic software research and open source software development help each other ? *In 4th Workshop on Open Source Software Engineering*, pages 5–8.

Arcelli, F., Tosi, C., Zanoni, M. et Maggioni, S. (2008). The marple project : A tool for design pattern detection and software architecture reconstruction. *In 1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1)*, pages 325–334.

Balland, E., Brauner, P., Kopetz, R., Moreau, P.-E. et Reilles, A. (2007). Tom : Piggybacking rewriting on java. *In International Conference on Rewriting Techniques and Applications*, pages 36–47. Springer.

Bass, L., Clements, P. et Kazman, R. (2012). *Software Architecture in Practice*. Addison-Wesley Professional, 3rd édition.

Berlack, H. R. (1992). *Software configuration management*. Wiley Online Library.

Bezroukov, N. (1999). Open source software development as a special type of academic research : Critique of vulgar raymondism. *First Monday*, 4(10).

Carver, J. C., Kendall, R. P., Squires, S. E. et Post, D. E. (2007). Software development environments for scientific and engineering software : A series of case studies. *In 29th International Conference on Software Engineering (ICSE'07)*, pages 550–559. Ieee.

Cho, H., Gray, J. et Sun, Y. (2012). Quality-aware academic research tool development. *In 2012 19th Asia-Pacific Software Engineering Conference*, volume 2, pages 66–72. IEEE.

Collard, M. L., Decker, M. J. et Maletic, J. I. (2011). Lightweight transformation and fact extraction with the srcml toolkit. *In Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on*, pages 173–184. IEEE.

CScout (2016). Site web du projet. `http://www.dmst.aueb.gr/dds/cscout/`. Dernier accès le 28/02/2016.

Easterbrook, S., Singer, J., Storey, M.-A. et Damian, D. (2008). Selecting empirical methods for software engineering research. *In Guide to advanced empirical software engineering*, pages 285–311. Springer.

Elan (2004). Site web du projet. `http://elan.loria.fr/overview/elan-compiler.html`. Dernier accès le 14/08/2016.

Groote, J. F. et Mousavi, M. R. (2014). *Modeling and analysis of communicating systems*. MIT press.

Hannay, J. E., MacLeod, C., Singer, J., Langtangen, H. P., Pfahl, D. et Wilson, G. (2009). How do scientists develop and use scientific software ? *In Proceedings of the 2009 ICSE workshop on Software Engineering for Computational Science and Engineering*, pages 1–8. IEEE Computer Society.

Harrison, R., Counsell, S. et Nithi, R. (2000). Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. *Journal of Systems and Software*, 52(2):173–179.

Hills, M., Klint, P., Van Der Storm, T. et Vinju, J. (2011). A case of visitor versus interpreter pattern. *In Objects, Models, Components, Patterns*, pages 228–243. Springer.

Hove, S. E. et Anda, B. (2005). Experiences from conducting semi-structured interviews in empirical software engineering research. *In Software Metrics, 2005. 11th IEEE International Symposium*, pages 10–pp. IEEE.

Jansen, A. et Bosch, J. (2005). Software architecture as a set of architectural design decisions. *In Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on*, pages 109–120. IEEE.

Keele, S. (2007). Guidelines for performing systematic literature reviews in software engineering. *In Technical report, Ver. 2.3 EBSE Technical Report. EBSE*.

Klint, P., Van Der Storm, T. et Vinju, J. (2009). Rascal : A domain specific language for source code analysis and manipulation. *In Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on*, pages 168–177. IEEE.

Lab, E. (2016). Wiki du projet. `http://essere.disco.unimib.it/wiki/`. Dernier accès le 15/08/2016.

mCRL2 (2011). Site web du projet. `http://www.mcrl2.org/release/user_manual/index.html`. Dernier accès le 28/02/2016.

Naur, P. et Randell, B. (1969). Software engineering : Report of a conference sponsored by the nato science committee, garmisch, germany, 7-11 oct. 1968, brussels, scientific affairs division, nato.

Paumier, S. (2009). Why academic software should be open source. *INFOtheca : Journal of Information and Library Science*, 10(1-2):51–54.

Prlić, A. et Procter, J. B. (2012). Ten simple rules for the open development of scientific software. *PLoS Comput Biol*, 8(12):e1002802.

ProM (2010). Site web du projet. `http://www.promtools.org/doku.php`. Dernier accès le 28/02/2016.

Rascal (2014). Site web du projet. `http://www.rascal-mpl.org/`. Dernier accès le 28/02/2016.

Seaman, C. B. (1999). Qualitative methods in empirical studies of software engineering. *Software Engineering, IEEE Transactions on*, 25(4):557–572.

Segal, J. et Morris, C. (2008). Developing scientific software. *IEEE Software*, 25(4):18–20.

Spinellis, D. (2010). Cscout : A refactoring browser for c. *Science of Computer Programming*, 75(4):216–231.

Spinellis, D. et Gousios, G. (2009). *Beautiful architecture : leading thinkers reveal the hidden beauty in software design.* " O'Reilly Media, Inc.".

srcML (2015). Site web du projet. `http://www.srcml.org/`. Dernier accès le 28/02/2016.

Tom (2013). Wiki du projet. `http://tom.loria.fr/wiki/index.php5/Main_Page`. Dernier accès le 28/02/2016.

van den Brand, M. G., van Deursen, A., Heering, J., De Jong, H., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P. A., Scheerder, J. *et al.* (2001). The asf+ sdf meta-environment : A component-based language development environment. *In International Conference on Compiler Construction*, pages 365–370. Springer.

Van Dongen, B. F., de Medeiros, A. K. A., Verbeek, H., Weijters, A. et Van Der Aalst, W. M. (2005). The prom framework : A new era in process mining tool support. *In International Conference on Application and Theory of Petri Nets*, pages 444–454. Springer.

Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. *In Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, page 38. ACM.

Guide d'interview

Ci-après vous trouverez la version originale et la version traduite du guide d'interview présenté au Chapitre 3.

Academic softwarf architectures : Expert Interview

Practical questions

- Can you present yourself ?
- Can you describe the role you play in [Project Name] ?
- Who was responsible of architecture choices in the project ?

Global architecture questions

- What are the main architectural choices in your project ?
- For each one mention :
  - Can you describe the choice and its motivations ?
- Did you use quality attributes to guide your choices ?
  - If answer is yes : Which one and why ?
- Which were the specific problems/requirement you wanted to adress with those architectural choices ?
- Does the architecture succeed in adressing those problems and requirements ?
- Does the architecture have caused some problems itself ?

Specific architecture questions

- Questions based on the preliminary research about the project being interviewed.

Evolution of the architecture

- Did the architecture encounter major changes since early versions of the project ?
- Can you describe the architecture evolution ? Which are the key choices that have lead toward those changes and which are the motivations behind those choices ?

Appreciation of the architecture

- What do you like in the architecture ?
- What do you dislike ?

Architecture des logiciels académiques : Interview d'experts

Questions pratiques

- Pouvez-vous vous présenter ?
- Pouvez-vous décrire le rôle que vous jouez dans [nom du projet] ?
- Qui était responsable des choix architecturaux dans le projet ?

Questions globales relatives à l'architecture

- Quels sont les choix architecturaux principaux dans votre projet ?
- Pour chacun d'entre-eux :
  - Pouvez-vous décrire le choix et ses motivations ?
- Avez-vous utilisé des attributs de qualité pour guider vos décisions ?
  - Si oui : Lesquels ?
- Quels étaient les problèmes spécifiques et exigences que vous souhaitiez adresser avec ces choix architecturaux ?
- L'architecture parvient-elle à répondre avec succès à ces exigences et problématiques ?
- L'architecture est-elle elle-même la cause de certains problèmes ?

Questions spécifiques au sujet de l'architecture

- Questions basées sur les recherches préliminaires concernant le projet interrogé.

Evolution de l'architecture

- L'architecture a-t-elle rencontré des évolutions majeures depuis la première version ?
- Pouvez-vous décrire l'évolution de l'architecture. Quels sont les choix clés ayant mené à ce changement et quelles sont les motivations de ces choix ?

Appréciation de l'architecture

- Qu'appréciez-vous particulièrement au sein de l'architecture ?
- Que n'appréciez-vous pas du tout au sein de l'architecture ?

Formulaires de l'enquête : première version

Ci-après vous trouverez la première version du formulaire qui fut soumise via Google Form au premier groupe de participants à l'enquête

Academic Software Architecture

Thank you for agreeing to take part in this survey about Academic Software Architecture. Depending on the complexity of your project this survey should take between 30 and 60 minutes of your time.

I am Bastien Meert, a Belgian Master student in Computer Science from the University of Namur (UNamur) in Belgium. I conduct this survey during my research internship at the University of Technology of Eindhoven (TU/e) under the supervision of Dr. A. Serebrenik and Dr. M. van den Brand.

The topic of the research are Academic Software Architecture. Academic Software are software developed as a subject or mean of research in academic context. Our main goals are to understand which are the common concerns and problems encounter in academic context and to study their impacts, solutions and evolutions in the architecture design.

The data collected will be combined with our previous results, a series of interviews, to produce a knowledge base which will later be used to give guidelines to software architects in academic projects.
We hope we will be able to help researchers like you by making the architecture design of academic software easier. We want to help making the right design decisions, avoiding the engineering traps and design more robust architectures adapted to their needs.

In addition, these research results are intended to be the base of my Master thesis.

If you are interested in the project and want to have more information, if you want to ask a question or just want to be informed later about the outcomes of this study you can send an e-mail to b.r.p.meert@tue.nl.

About confidentiality: the data is designed to be used anonymously and not linked to the project itself but we will ask you later in the survey which reference policy you want us to apply to your submission.

*Obligatoire

Practical information about the project

1. **Based on which project do you want to fill in this survey ?** *
   This information will be used in respect of the reference policy you will choose at the end of the survey.

2. **How old is the project ?** *

3. **What was your role(s) in the project ?** *
   e.g. architect, developer, etc

4. **How many people have contributed to the project ?**
   You can give an estimation or leave it blank if you do not know

5. **How many PhD students have contributed to the project ?**
   You can give an estimation or leave it blank if you do not know

6. **How many Master students have contributed to the project ?**
   You can give an estimation or leave it blank if you do not know

Global architecture description

In this section we ask you to give us some information about the global architecture of the project. As we are really interested in concerns that lead to decisions, please add the reasons behind the choices you explain in the global descriptions. If you do not know why something has been done you can mention the choice only.

If you need more guidelines about the architecture description you can give a list of architectural decisions for which you develop the following elements :
- short description (e.g. we have decided to use a pipe and filter)
- reason behind (e.g. because we could easily configure our pipe choosing the appropriate filter for a given situation)
- success evaluation (e.g. it was mitigated because configuration gives a lot of flexibility but it was difficult to handle the input/output requirements of each filters)

We really like the decisions/reasons/evaluation pattern as it allows us to understand why choices were made and gives us information about the way we can improve this decision process.

7. **Can you give an approximation of the project size ?**
   In LOC

8. **If the software is component-based, how many components are there ?**

9. **How important is the architecture design and quality in your project ?** *
   Une seule réponse possible.

   ◯ It has always been a strong concern and was carefully designed before any code was written.

   ◯ It has always been a strong concern and architecture was designed concurrently to the development

   ◯ It was not a concern in the beginning but became a priority later, the architecture was completely redesigned

   ◯ There was no global plan for the architecture, structural choices were discussed and made depending on needs at the moment of the choice

   ◯ Architecture has never been a concern during the project

10. **Can you give a description of the global architecture as it is now ?** *
    e.g. architecture styles, main structural decisions, etc

11. **What can you say about the evolutions that led to this architecture ?** *
    structural, architectural, styles changes, etc. You can also describe how it was designed first and how it has evolved.

12. **Have you encountered specific problems in the architecture design ? How have you handled them ?**

Concern that led to design decisions

In this section we introduce a list of concerns we found in most of the previously studied projects. For each one we ask you to give an estimation of its importance in the beginning of the project and its importance now.

You can explain why it was (or was not) a concern, how it has evolved and how it has impacted the architecture design.
We are also interested in the success of this impact to address the concern.

About the evaluation of the concern we have created a scale:
- Not a concern

- Minor concern: you will look at it if there are enough resources after dealing with major concerns
- Major concern: it is around these concerns that you design your architecture
- Crucial concern: it has to be reflected in the architecture without any compromise

13. **Possibility of writing publication about your architectural choices (publishability)** *
    e.g. to use a new style or structure so you can write on it
    Une seule réponse possible par ligne.

	Not a concern	Minor concern	Major concern	Crucial concern
In the beginning				
Currently				

14. **Can you formulate your thought about the aforementioned publishability ?**
    reason behind, evolution, success of the architecture to incarnate it

15. **Impact of publishability on the architecture**
    You can also give an example

16. **Request from the source of external funding** *
    e.g. the company which is funding the project want you to put specific aspects in the project architecture
    Une seule réponse possible par ligne.

	Not a concern	Minor concern	Major concern	Crucial concern
In the beginning				
Currently				

17. **Can you formulate your thought about the aforementioned external funding ?**
    reason behind, evolution, success of the architecture to incarnate it

18. **Impact of external funding on the architecture**
You can also give an example

19. **Understandability of the architecture** *
ease of learning how it is build and how it works
Une seule réponse possible par ligne.

	Not a concern	Minor concern	Major concern	Crucial concern
In the beginning				
Currently				

20. **Can you formulate your thought about the aforementioned understandability ?**
reason behind, evolution, success of the architecture to incarnate it

21. **Impact of understandability on the architecture**
You can also give an example

22. **Modifiability of the architecture** *
ease of extend, modify, improve the architecture
Une seule réponse possible par ligne.

	Not a concern	Minor concern	Major concern	Crucial concern
In the beginning				
Currently				

23. **Can you formulate your thought about the aforementioned modifiability ?**
reason behind, evolution, success of the architecture to incarnate it

24. **Impact of modifiability on the architecture**
You can also give an example

25. **Ease of contribution to the software** *
ease of building a new plugin, or adding a new feature or upgrade, etc without modifying architecture
Une seule réponse possible par ligne.

	Not a concern	Minor concern	Major concern	Crucial concern
In the beginning				
Currently				

26. **Can you formulate your thought about the aforementioned ease of contribution?**
reason behind, evolution, success of the architecture to incarnate it

27. **Impact of ease of contribution on the architecture**
You can also give an example

28. **Performance** *

e.g. we have to build a very fast parser, we have to be able to deal with concurrency, etc
Une seule réponse possible par ligne.

	Not a concern	Minor concern	Major concern	Crucial concern
In the beginning				
Currently				

29. **Can you formulate your thought about the aforementioned performance concern ?**

reason behind, evolution, success of the architecture to incarnate it

30. **Impact of performance on the architecture**

You can also give an example

31. **Contributor's needs or preferences** *

e.g. you have to supervise a PhD student and add unforeseen features to help him in his thesis
Une seule réponse possible par ligne.

	Not a concern	Minor concern	Major concern	Crucial concern
In the beginning				
Currently				

32. **Can you formulate your thought about the aforementioned contributor needs?**

reason behind, evolution, success of the architecture to incarnate it

33. **Impact of contributors needs on the architecture**
    You can also give an example

Feedback on design decisions

In this section we want to know more about what you think about decisions made in the architecture design.
We would like you to think about the most important decisions that have been made in the architecture design and to say if it was good or bad decisions. We also like you to say why a decision was taken and why it was a bad or a good decision.
In the case of bad decisions, a short description of the solutions given to the problems will be appreciated.

These good and bad decisions are crucial to put in place the guidelines about what to do and what not to do in an academic software architecture design.

Examples :
A good decision: the project is about developing a language and include a compiler. An early decision was to bootstrap the language. It was noticed that this bootstrap has improved the quality of the project by enforcing the need to be very robust and well designed to make sure the compiler works after the bootstrap.

A bad decision: the project use very independent components to decrease coupling but nothing was done about code reuse. I ended with a lot of copy/paste and caused a lot of maintainability issues.
To solve this problem a set of libraries have been developed to benefit from a shared code base that can be change at once for all similar components.

34. **Can you make a list of the best decisions that were made in the architecture design in your opinion ?**
    decisions which had a very good impact on the project, reasons behind them and why they are good

35. **Can you make a list of the worst decisions that were made in the architecture design in your opinion ?**
    decisions which had a very bad impact on the project, reasons behind them and why they are bad, a small description of the solutions are welcome

Policies

In this section we ask you more information about the policies used in your project.
There are three of them : the test policy, the release policy and the contribution policy.

Test policy

36. **In the following elements, which one did you include in your test policy ?** *
Plusieurs réponses possibles.

- [ ] Manual Unit Testing
- [ ] Automated Unit Testing
- [ ] Manual Black-box functionnal testing
- [ ] Automated black-box functionnal testing
- [ ] Implementation of a test framework

37. **Does your test policy go further than aforementioned methods ? Can you describe in which way ?**

38. **How would you describe the maturity of your test policy ?** *
Une seule réponse possible.

	0	1	2	3	4	
Not mature	◯	◯	◯	◯	◯	Totally mature

Release policy

39. **How many branches of development do you maintain? Describe them.** *

40. **What do you do when you want to release ?** *
e.g. do you freeze all developments before a release or there is only the master which is frozen, etc

41. **Do you have a release planning ?** *
Une seule réponse possible.

- ◯ No, software is released when it is ready.
- ◯ Yes, we regularly release a new version of it.

42. **What is your release rate in month ?**
let blank if you don't know. e.g. no release plan

43. **Do you maintain released versions** *
Une seule réponse possible.

- ◯ Never
- ◯ Only in case of extreme necessity
- ◯ Yes

44. **If your architecture is components/plugins based which release pattern do you use ?**
Une seule réponse possible.

- ◯ Synchronous: new/updated components/plugins are add to the next release
- ◯ Asynchronous: every element which is stable and ready to release are added to the next release

Contribution policy

45. **In the following choices, which one is the better to depict your contribution policy ?** *
Une seule réponse possible.

- ◯ Everybody can contribute without any limits
- ◯ Everybody can contribute as long as they meet some requirements
- ◯ Only core team can contribute

46. **If you have contribution rules, which are they ? How do you deal with external contributions ?**

47. **How do you deal with student's contributions ?** *
    Are they all added to the main releases, are they only use by the students themselves, etc

Commercial Offsprings

In this section we ask you if you have or will have commercial offsprings. And what are the actual impacts on the architecture.

48. **Is the project subject to commercial offsprings ?** *
    it is used in industry, sold or used to generate product
    Une seule réponse possible.

- ◯ No and it will never be.
- ◯ No but it could happen in the future.
- ◯ No but it is planned to be in the future
- ◯ Yes it was designed to be from the beginning
- ◯ Yes but it was not planned in the beginning

49. **If yes, what are the impacts it had, has or will have on the architecture ?**

Reference in later work

In this section I would like to ask you if you want your project to be referenced in my work.
There are tree possibilities :

- No reference : then the project will be mentioned as "one project" or "project X" where X would be an anonymous identifier.
- Contribution reference : the project will still be mentioned as "one project" or "project X" but a reference to it will be included in a list of contributions to my work and future work based on it.
- Explicit reference : the project will be mentioned by its name in my work and will also appear on

the list of contributions

We would also like to ask you if you want to have a more complete reference like an e-mail address, a website, a reference to a specific publication, etc.

Please know that we will not forget to correctly reference your published work that we could use if you choose no reference, we will just not link them to any result of the survey.

50. **How do you want your project to be referenced ?** *
    e.g. in results report of the internship or related presentation, in the thesis that will be based on the project
    Une seule réponse possible.

    - ( ) No reference
    - ( ) Contribution reference
    - ( ) Explicit reference

51. **If you want a more complete reference, which information do you want us to use ?**
    let blank if you have chose "no reference" or prefer to be referenced only by the name. You can also specify a reference policy to adopt with your project

Feedback

In this optional section we will ask you to answer a few feedback about the survey itself.
It will help us to improve it for further diffusion and a fortiori improve the quality and usefulness of the results.

52. **How much time did you take to complete the survey ?**

53. **Did you find some questions unclear ? Which ones ? How can we improve them ?**

54. **What do you think about the survey and how would you improve it ?**

Formulaires de l'enquête : version révisée

Ci-après vous trouverez la version revue de l'enquête qui fut envoyée aux participants sous la forme d'un formulaire PDF.

Academic Software Architecture

General Information:

Thank you for agreeing to take part in this survey about Academic Software Architecture.
Depending on the complexity of your project this survey should take between 45 and 60 minutes of your time.

I am Bastien Meert, a Belgian Master student in Computer Science from the University of Namur (UNamur) in Belgium. I first conducted this survey during my research internship at the University of Technology of Eindhoven (TU/e) under the supervision of Dr. A. Serebrenik and Pr. M. van den Brand. I am currently writing my Master Thesis about Academic Software Architecture and I would have liked to get a few more results for the survey.

The topic of the research are Academic Software Architecture. We define Academic Software as software developed as a subject or mean of research in academic context. Our main goals are to understand which are the common concerns and problems encountered in academic context and to study their impacts, solutions and evolutions in the architecture design.

The data collected will be combined with our previous results, a series of interviews, to produce a knowledge base which will later be used to give guidelines to software architects in academic projects.

We hope we will be able to help researchers like you by making the architecture design of academic software easier. We want to help making the right design decisions, avoiding the engineering traps and design more robust architectures adapted to their needs.

If you are interested in the project and want to have more information, if you want to ask a question or just want to be informed later about the outcomes of this study you can send an e-mail to bastien.meert@student.unamur.be.

About confidentiality: the data is designed to be used anonymously and not linked to the project itself but we will ask you later in the survey which reference policy you want us to apply to your submission.

1. Practical information about the project

In this section you will give some information about the project.

It will show you if your project meet all our selection criteria as we can not check all of them with public information. If your project do not meet those criteria or you can not fill it in in time, you can send us back this page only and make sure to have check

1. Project Name:
   This information will be used in respect of the reference policy you will choose at the end of the survey.
2. Age of the project:
   We are searching for project arround 5 years old and older.
3. Number of contributors:
   You can give an estimation.
4. Number of PhD students contributors:
   You can give an estimation or leave blank.
5. Number of Master students contributors:
   You can give an estimation or leave blank.
6. Number of describing publications:
   You can give an estimation. We are searching for projects at least describe in one publication.
7. Number of use in publications:
8. You can give an estimation. We are searching for projects which appears in at least two publications (as tool, subject, etc)
9. Is the project still in activity ?
   We are searching for project still in activity.
10. Project size (LOC):
    You can give an estimation.
11. ☐ I am interested in being contacted again about this project.

    This information will be usefull if you do not meet our selection criteria now but that we extend/change them in the future or you do not have time now but still would be glad to participate if an other student continue to work on this research topic in the future.

2. Architecture description

In this section we ask you to give us some information about the global architecture of the project.
As we are really interested in concerns that lead to decisions, please add the reasons behind the choices you explain in the global descriptions. If you do not know why something has been done you can mention the choice only.

If you need more guidelines about the architecture description you can give a list of architectural choices for which you develop the following elements :
- short description (e.g. we have decided to use a pipe and filter)
- incentive (e.g. because we could easily configure our pipe choosing the appropriate filter for a given situation)
- success evaluation (e.g. it was mitigated because configuration gives a lot of flexibility but it was difficult to handle the input/output requirements of each filters)

We really like the decision/incentive/evaluation pattern as it allows us to understand why choices were made and gives us information about the way we can improve this decision process.

How important are the architecture design and architecture quality in your project ?

◯ It has always been a strong concern and was carefully designed before any code was written.
◯ It has always been a strong concern and architecture was designed concurrently to the development
◯ It was not a concern in the beginning but became a priority later, the
◯ There was no global plan for the architecture, structural choices were discussed and made de
◯ Architecture has never been a concern during the project

In the following list, which are the architecture styles used in your project ?

☐ Paradigm influenced
☐ Component-basedwith components.
☐ Plugins-based
☐ Monolithic application
☐ Layered
☐ Client-server
☐ Pipes and filters
☐ Batch sequential
☐ Datacentric
☐ Blackboard
☐ Rule based
☐ Interpreter
☐ Peer-to-peer
☐ Event-based

How are these styles adapted/mixed in your architecture ?

Are there other main structures or styles you have used ? Can you descibe them ?

How could you describe the evolution of the architecture ? Which are the major changes which took places and why ?

structural, architectural, styles changes, etc. You can also describe how it was designed first and how it has evolved.

Have you encountered specific problems in the architecture design ? How have you handled them ?

3. Concerns that led to design decisions

In this section we introduce a list of concerns we found in most of the previously studied projects.
For each one we ask you to give an estimation of its importance in the beginning of the project and its importance now.

You can explain why it was (or was not) a concern, how it has evolved and how it has impacted the architecture design.
We are also interested in the success of this impact to address the concern.

About the evaluation of the concern we have created a scale:
- Not a concern
- Minor concern: you will look at it if there are enough resources after dealing with major concerns
- Major concern: it is around these concerns that you design your architecture
- Crucial concern: it has to be reflected in the architecture without any compromise

What was the situation when you started to work on the project ?

	Not	Minor	Major	Crucial
Possibility of publication about your architectural choices e.g. to use a new style or structure so you can write on it	○	○	○	○
Request from the source of external funding e.g. the company which is funding the project want you to put specific aspects in the project architecture	○	○	○	○
Understandability of the architecture ease of learning how it is build and how it works	○	○	○	○
Modifiability of the architecture ease of extend, modify, improve the architecture	○	○	○	○
Ease of contribution to the software ease of building a new plugin, or adding a new feature or upgrade, etc without modifying architecture	○	○	○	○
Performance e.g. we have to build a very fast parser, we have to be able to deal with concurrency, etc	○	○	○	○
Contributor's needs or preferences e.g. you have to supervise a PhD student and add unforeseen features to help him in his thesis	○	○	○	○

What is the situation now ?

	Not	Minor	Major	Crucial
Possibility of publication about your architectural choices	○	○	○	○
Request from the source of external funding	○	○	○	○
Understandability of the architecture	○	○	○	○
Modifiability of the architecture	○	○	○	○
Ease of contribution to the software	○	○	○	○
Performance	○	○	○	○
Contributor's needs or preferences	○	○	○	○

Can you elaborate your thought about the actual situation ?

reason behind, evolution, success of the architecture to incarnate it

Can you describe the impact of the actual situation on the architecture ?

An example of implementation, a choices you took because of it, etc.

4. Feedback on design decisions

In this section we want to know more about what you think about decisions made in the architecture design.
We would like you to think about the most important decisions that have been made in the architecture design and to say if it was good or bad decisions. We also like you to say why a decision was taken and why it was a bad or a good decision.
In the case of bad decisions, a short description of the solutions given to the problems will be appreciated.

These good and bad decisions are crucial to put in place the guidelines about what to do and what not to do in an academic software architecture design.

Examples :

A good decision: the project is about developing a language and include a compiler. An early decision was to bootstrap the language. It was noticed that this bootstrap has improved the quality of the project by enforcing the need to be very robust and well designed to make sure the compiler works after the bootstrap.

A bad decision: the project use very independent components to decrease coupling but nothing was done about code reuse. I ended with a lot of copy/paste and caused a lot of maintainability issues.
To solve this problem a set of libraries have been developed to benefit from a shared code base that can be change at once for all similar components.

Can you make a list of the best decisions that were made in the architecture design in your opinion ?

decisions which had a very good impact on the project, reasons behind them and why they are good

Can you make a list of the worst decisions that were made in the architecture design in your opinion ?

decisions which had a very bad impact on the project, reasons behind them and why they are bad, a small description of the solutions are welcome

5. Policies

In this section we ask you more information about the policies used in your project.
There are three of them : the test policy, the release policy and the contribution policy.

Test policy

In the following elements, which one did you include in your test policy ?

- ☐ Manual Unit Testing
- ☐ Automated Unit Testing
- ☐ Manual Black-box functionnal testing
- ☐ Automated black-box functionnal testing
- ☐ Implementation of a test framework

Does your test policy go further than aforementioned methods ? Can you describe in which way ?

How would you describe the maturity of your test policy on a 1 to 5 scale ?

1	2	3	4	5
○	○	○	○	○

Release Policy

How many branches of development do you maintain? Describe them.

What do you do when you want to release ?

e.g. do you freeze all developments before a release or there is only the master which is frozen, etc

Do you have a release planning ?

◯ Yes we have regular releases ◯ No, we release the software when it is ready

What is your release rate in month ?

Do you maintain released versions ?

◯ Never
◯ Only in case of extreme necessity
◯ Yes

If your architecture is components/plugins based which release pattern do you use ?

◯ Synchronous: new/updated components/plugins are add to the next release
◯ Asynchronous: every element which is stable and ready to release are added to the next release

Contribution Policy

In the following choices, which one is the better to depict your contribution policy ?

◯ Everybody can contribute without any limits
◯ Everybody can contribute as long as they meet some requirements
◯ Only core team can contribute

If you have contribution rules, which are they ? How do you deal with external contributions ?

How do you deal with student's contributions ?

Are they all added to the main releases, are they only use by the students themselves, etc

6. Commercial offsprings

In this section we ask you if you have or will have commercial offsprings. And what are the actual impacts on the architecture.

Is the project subject to commercial offsprings ?

it is used in industry, sold or used to generate product

- ◯ No and it will never be.
- ◯ No but it could happen in the future.
- ◯ No but it is planned to be in the future
- ◯ Yes it was designed to be from the beginning
- ◯ Yes but it was not planned in the beginning

If yes, what are the impacts it had, has or will have on the architecture ?

7. Reference policy and feedback

In this final section you can give us some feedback on this survey and help us to improve our work.
We also ask you in which way you want your project to be rederenced in later work.

Please know that we will not forget to correctly reference your published work that we could use if you choose no reference, we will just not link them to any result of the survey.

How do you want to be referenced in later work ?

e.g. in results report of the internship or related presentation, in the thesis that will be based on the project

◯ No reference:then the project will be mentioned as "one project" or "project X" where X would be an an

◯ Contribution reference:the project will still be mentioned as "one project" or "project X" but a reference t

◯ Explicit reference:the project will be mentioned by its name in my work and will also appear on the list o

If you want a more complete reference, which information do you want us to use ?

let blank if you have chose "no reference" or prefer to be referenced only by the name. You can also specify a reference policy to adopt with your project

How much time did you take to complete the survey ?

Did you find some questions unclear ? Which ones ? How can we improve them ?

What do you think about the survey and how would you improve it ?

Thank you for your participation. You can now send me your answer to bastien.meert@student.unamur.be. I am also open to any question or information request about my project.