PolyPublie
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Titre: Title:	Recovery of traceability links in software artifacts and systems
Auteur: Author:	Giuliano Antoniol
Date:	2003
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Antoniol, G. (2003). Recovery of traceability links in software artifacts and systems [Ph.D. thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7272/

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/7272/
Directeurs de recherche: Advisors:	Ettore Merlo
Programme: Program:	Unspecified

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

NOTE TO USERS

This reproduction is the best copy available.

UMI®

UNIVERSITÉ DE MONTRÉAL

RECOVERY OF TRACEABILITY LINKS IN SOFTWARE ARTIFACTS AND SYSTEMS

GIULIANO ANTONIOL
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIAE DOCTOR (Ph.D.)
(GÉNIE ÉLECTRIQUE)
DÉCEMBRE 2003

© Giuliano Antoniol, 2003.

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisisitons et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-92151-4
Our file Notre référence
ISBN: 0-612-92151-4

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this dissertation.

While these forms may be included in the document page count, their removal does not represent any loss of content from the dissertation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de ce manuscrit.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée:

RECOVERY OF TRACEABILITY LINKS IN SOFTWARE ARTIFACTS AND SYSTEMS

présentée par: ANTONIOL Giuliano

en vue de l'obtention du diplôme de: Philosophiae Doctor

a été dûment acceptée par le jury d'examen constitué de:

M. DAGENAIS, Michel, Ph.D., président

M. MERLO, Ettore, Ph.D., membre et directeur de recherche

M. PIERRE, Samuel, Ph.D., membre

M. GALLAGHER, Keith, Ph.D., examinateur externe

To my Parents and my Sister Mema

ACKNOWLEDGEMENTS

I would like to thank the many people that helped me in many different ways. In particular I would like to thank Gerardo Canfora, Bruno Caprile, Gerardo Casazza, Aniello Cimitile, Michel Dagenais, Andrea De Lucia, Sara Gradara, Massimiliano Di Penta, Alessandra Potrich, Jean Sylvain Sormani, Paolo Tonella and Umberto Villano for their valuable suggestions and help in coauthoring publications directly or indirectly related to traceability recovering.

Thanks to the `Albergate` programming team: Claudia Ciccone, Andrea Colombari, Francesca Danzi, Daria Girelli, Roberto Martini, Mauro Meneghini, Andrea Porta, and Paola Vincenti, who kindly provided the source code, related documentation, and requirements to code traceability matrix.

I would like to express my gratitude to the people who inspired this research: Eliseo Mambella who first brought to my attention the relevance of the traceability recovery problem in 1995; Renato De Mori who introduced me to statistical pattern recognition; and Fabio Brugnara, whose suggestions were invaluable in developing the stochastic language model for traceability link recovery.

I would like to thank my friends for their encouragement. In particular, I owe a debt of gratitude to Licia, without whose support I would not have achieved all of my objective of the past twelve years.

Finally, a special thanks to Ettore Merlo, not only my advisor, but also one of my best friends: we share much more than research interests we share a vision of life.

RÉSUMÉ

La demande de production d'une grande quantité de systèmes logiciels promeut l'adoption de techniques et d'approches visant à augmenter la productivité de développement. La pression élevée pour augmenter la productivité favorise des approches inspirées de la fabrication, telle que l'utilisation de composantes et de normes; l'hypothèse fondamentale est que l'industrie du logiciel pourrait beaucoup améliorer sa productivité en appliquant de telles approches.

Malgré une certaine adoption de la réutilisation, des intergiciels, des générateurs de code, du logiciel personnalisé, des cadres de référence et des outils de développement rapides, l'industrie du logiciel est toujours loin d'atteindre l'augmentation promise de productivité.

De plus, le développement d'un logiciel couvre juste une fraction du cycle de vie du logiciel: les systèmes logiciels évoluent sans interruption pour satisfaire les besoins toujours changeants des utilisateurs; les changements peuvent être dictés par la pression du marché, par l'adaptation à la législation ou par les améliorations requises.

Dans le marché compétitif d'aujourd'hui, la pression à augmenter la productivité porte à des processus plus sophistiqués (c.-à-d. les processus de développement et d'évolution), qui éliminent les activités improductives.

Établir et maintenir la traçabilité et la cohérence entre les objets logiciels produits ou modifiés le long du cycle de vie du logiciel sont parmi les activités coûteuses et fastidieuses fréquemment négligées.

Cependant, les liens de traçabilité entre la documentation en texte libre associée au cycle de développement et d'entretien d'un système de logiciel et de son code source sont indispensables dans un nombre de tâches comme la compréhension du code et l'entretien du logiciel.

En plus, le maintien des liens de traçabilité entre les versions subséquentes d'un

système logiciel est important pour évaluer les variations incrémentielles entre versions, pour identifier les incohérences de rapport entre effort de développement et variation de code et évaluer pour l'historique des changements.

Cette thèse propose des approches pour identifier et évaluer semi-automatiquement des liens de traçabilité entre des objets logiciels afin de réduire l'effort exigé pour maintenir, pour faire évoluer ou pour vérifier ces liens le long du cycle de vie du logiciel.

La documentation des systèmes logiciels est presque toujours exprimée de façon informelle, en langage naturel et texte libre. Les exemples incluent les spécifications des caractéristiques, les pages de manuels, les journaux de développement de systèmes, les journaux d'erreurs et les rapports relatifs à l'entretien. D'un autre côté, le code source doit adhérer à la grammaire des langages de programmation adoptés pour développer les logiciels. De plus, des paradigmes différents de développement peuvent avoir été adoptés (par exemple, objet orienté versus l'approche procédurale traditionnelle). En conséquence, plusieurs méthodes sont nécessaires pour faire face à la nature différente des objets produits, des paradigmes de développement et des différents langages de programmation. Avec plus de précision, la taxonomie suivante a été définie et appliquée:

- retrouver ou valider les liens de traçabilité à partir de la documentation en texte libre;
  - code cible: appliquer des méthodes inspirées de la recherche d'informations;
- retrouver ou valider les liens de traçabilité à partir de la conception;
  - conception cible: exprimée en texte ou selon des méthodes formelles, appliquer des méthodes basées sur les graphes et les algorithmes pertinents;
- retrouver ou valider les liens de traçabilité à partir du code source;
  - code cible: comme pour la conception pour logiciel orienté objet ou les approches basées sur la détection des clones dans le code procédural.

Les différents cas de taxonomie sont basés sur une définition de similitude entre entités logicielles. Il vaut la peine de noter que, si la similitude entre entités est définie sur la base de l'ensemble des propriétés communes, deux entités ne sont pas distinguables si et seulement si elles partagent le même nom et possèdent le même ensemble de propriétés.

Cependant, chaque objet logiciel le long du cycle de vie devrait être l'évolution ou l'amélioration des objets précédents. Il devrait être comparé aux objets précédents de sorte qu'il soit possible de tracer l'information le long du cycle de vie du logiciel. De ce fait, en général, imposer la correspondance exacte des propriétés est inutilement rigoureux et peut mener à des résultats insatisfaisants. Ainsi, une étape préliminaire est la définition d'une mesure moins rigoureuse de similitude avec un comportement plus réaliste. Une fois qu'un tel critère de similitude a été défini, chaque objet peut être tracé dans l'ensemble de ceux qui lui ressemblent le plus.

L'exactitude du rétablissement des liens de traçabilité a été empiriquement évaluée sur différents systèmes logiciels. En particulier, des modèles de recherche documentaire ont été appliqués dans deux études pour tracer les liens entre le code source en C++ et des pages de manuels et pour rétablir les liens entre le code source en Java et ses spécifications fonctionnelles. 29 logiciels industriels, qui avaient été développés selon le paradigme par objets et pour lesquels la conception était disponible, ont été employés pour évaluer la faisabilité de récupérer les liens de traçabilité entre la conception et le code source en utilisant des algorithmes de graphes. Le rétablissement de la traçabilité entre les systèmes logiciels a aussi été évalué en utilisant plusieurs versions de deux systèmes écrits en C++. La faisabilité d'extraire des mesures de similitude de niveau élevé à partir de grands systèmes logiciels a été démontrée sur environ 400 versions du noyau de Linux.

En conclusion, cette thèse démontre la faisabilité du rétablissement des liens de traçabilité, mesure la performance et précision des approches proposées sur différents systèmes logiciels et identifie une taxonomie des facteurs (par exemple, le code réutilisé, l'architecture externe, etc.) qui peuvent affecter le rétablissement des liens de

traçabilité entre la documentation en texte libre et le code source.

Mots-Clés: génie logiciel, analyse de logiciels, traçabilité, évolution de logiciels, similarité de logiciels

ABSTRACT

Software production for the mass market requires the adoption of techniques and approaches to increase software development productivity. The great pressure to produce has promoted approaches derived from manufacturing, such as component-based development and standardization. The underlying assumption is that the software industry may improve productivity by applying similar approaches to those used in manufacturing. Despite the adoption of reuse, availability of commercial off-the-shelf components, and existence of code generators, middleware, frameworks, and rapid application development tools and techniques, the software industry is still far from the anticipated productivity level. Moreover, software development represents a small part of the software life-cycle; software systems continuously evolve to meet ever changing user needs which may be driven by market pressure, adaptation to legislation, or improvement needs.

In today's competitive market, the pressure to increase productivity results in the tailoring of software processes, especially development and evolution processes to eliminate unproductive activities. Establishing and maintaining traceability links and consistency between software artifacts produced or modified in the software life-cycle are costly and tedious activities that are crucial but frequently neglected in practice. Traceability links between the free text documentation associated with the development and maintenance cycle of a software system and its source code are crucial in a number of tasks such as program comprehension and software maintenance. Finally, maintaining traceability links between subsequent releases of a software system is important for evaluating relative source code deltas, highlighting effort/code variation inconsistencies, and assessing the change history.

This thesis proposes to recover and assess traceability links between and within software artifacts and systems in a semi-automatic way, thus reducing the effort re-

quired to maintain, evolve or verify traceability links throughout the software life-cycle.

Software system documentation is almost always expressed informally, using natural language and free text. Examples include requirement specifications, manual pages, system development journals, error logs and related maintenance reports. On the other hand, source code must adhere to the grammars of the programming languages adopted to develop the software. Furthermore, different development paradigms may have been adopted (e.g., object-oriented versus the traditional procedural approach). As a result, several methods are needed to cope with the various software artifacts, development paradigms and programming languages. More precisely, the following taxonomy has been defined and applied to recover or validate traceability links from:

- free text documentation to code: apply methods adapted from information retrieval;
- design to design or code: apply methods relying on graph algorithms;
- source code to source code: apply methods similar to the previous item (object-oriented software) or clone-based approaches (procedural code).

The different taxonomy cases are based on the definition of a similarity between two software artifacts. It is worth noting that, if the similarity between two artifacts is defined as the set of common properties, the two artifacts are indistinguishable if and only if they share the same name and possess the same collection of properties. However, each software artifact in the software life-cycle could be the evolution or the refinement of previous artifacts. Artifacts should be consistent so that it is possible to trace information along the software life-cycle. In general, imposing the exact correspondence of properties is unnecessarily stringent and may lead to unsatisfactory results. Thus, a preliminary step is the definition of a less stringent similarity measure with a more realistic behavior. Once such a similarity criterion has been defined, each given artifact may be traced into the set of those artifacts most similar to it.

The accuracy of traceability recovery has been empirically evaluated on different software systems. In particular, information retrieval models were applied in two case studies. The first study investigated tracing C++ source code onto manual pages. The second mapped Java code to functional requirements.

The evaluation of the feasibility of recovering a traceability mapping between design and source code was based on 29 industrial software components. The components had been developed in the object-oriented paradigm and each one was documented with its design.

Traceability recovery between subsequent releases of C++ software systems was investigated using several releases of two C++ systems. The feasibility of extracting high level similarity measures from large software systems was demonstrated by analyzing about 400 releases of the Linux kernel.

Main contributions of this thesis can be summarized as the original definition, development and assessment of technologies, methods and approaches to support traceability links recovery between different level of abstraction: from textual documentation to source code, from design to code, and between different releases of a software system.

This thesis demonstrates the feasibility of the presented traceability links recovery approaches, quantifies the accuracy of such approaches on different software systems, and identifies a taxonomy of factors such as reused code, external architecture, frameworks which potentially affect the accuracy of the traceability recovery process between free text documentation and source code.

Foreseeable application of the presented approaches to industrial problem will help to reduce the effort required to recover, maintain and validate traceability links.

Keywords: software engineering, software analysis, traceability, re-documentation, software evolution,program comprehension, similarity definition

CONDENSÉ

La demande de production d'une grande quantité de systèmes logiciels promeut l'adoption de techniques et d'approches visant à augmenter la productivité de développement. La pression élevée pour augmenter la productivité favorise des approches inspirées à la fabrication, telle que l'utilisation de composantes et de normes; l'hypothèse fondamentale est que l'industrie du logiciel pourrait beaucoup améliorer la productivité en appliquant de telles approches.

Malgré une certaine adoption de la réutilisation, des générateurs de code, du logiciel personnalisé, des cadres de référence et des outils de développement rapides, l'industrie du logiciel est toujours loin d'atteindre l'augmentation promise de productivité.

De plus, le développement d'un logiciel couvre juste une fraction du cycle de vie du logiciel: les systèmes logiciels évoluent sans interruption pour satisfaire les besoins toujours changeants des utilisateurs; les changements peuvent être dictés par la pression du marché, par l'adaptation à la législation ou par les amélioration requises.

Dans le marché compétitif d'aujourd'hui, la pression à augmenter la productivité porte à des processus plus sophistiqués (c.-à-d. les processus de développement et d'évolution), qui éliminent les activités improductives.

Établir et maintenir la traçabilité et la cohérence entre les objets logiciels produits ou modifiés le long du cycle de vie du logiciel sont parmi les activités coûteuses et fastidieuses fréquemment négligées.

Cependant, les liens de traçabilité entre la documentation en texte libre associée au cycle de développement et d'entretien d'un système de logiciel et de son code source sont indispensables dans un nombre de tâches comme la compréhension du code et l'entretien du logiciel.

En plus, le maintien des liens de traçabilité entre les versions subséquentes d'un

système logiciel est important pour évaluer les variations incrémentielles entre versions, pour identifier les incohérences de rapport entre effort de développement et variation de code et pour évaluer l'historique des changements.

Cette thèse propose des approches pour identifier et évaluer semi-automatiquement des liens de traçabilité entre des objets logiciels afin de réduire l'effort exigé pour maintenir, pour faire évoluer ou pour vérifier ces liens le long du cycle de vie du logiciel.

La documentation des systèmes logiciels est presque toujours exprimée de façon informelle, en langage naturel et texte libre. Les exemples incluent les spécifications des caractéristiques, les pages de manuels, les journaux de développement de systèmes, les journaux d'erreurs et les rapports relatifs à l'entretien. D'un autre côté, le code source doit adhérer à la grammaire des langages de programmation adoptés pour développer les logiciels. De plus, des paradigmes différents de développement peuvent avoir été adoptés (par exemple, par objets versus l'approche procédurale traditionnelle). En conséquence, plusieurs méthodes sont nécessaires pour faire face à la nature différente des objets produits, des paradigmes de développement et des différents langages de programmation. La taxonomie suivante a été définie et appliquée:

- approches déterministes
- approches probabilistes

Avec plus de précision:

- retrouver ou valider les de liens traçabilité à partir de la documentation en texte libre;
  - code cible: appliquer des méthodes inspirées de la recherche d'informations;
- retrouver ou valider les liens de traçabilité à partir de la conception;
  - conception cible: exprimée en texte ou selon des méthodes formelles appliquer des méthodes basées sur les graphes et les algorithmes pertinents;

- retrouver ou valider les liens de traçabilité à partir du code source;
  - code cible: comme pour la conception de logiciels par objets ou les approches basées sur la détection des clones dans le code procédural.

Les différents cas de taxonomie sont basés sur une définition de similitude entre entités logicielles. Il vaut la peine de noter que, si la similitude entre entités est définie sur la base de l'ensemble des propriétés communes, deux entités ne sont pas distinguables si et seulement si elles partagent le même nom et possèdent le même ensemble de propriétés.

Cependant, chaque objet logiciel le long du cycle de vie devrait être l'évolution ou l'amélioration des objets précédents. Il devrait être comparé aux objets précédents de sorte qu'il devrait être possible de tracer l'information le long du cycle de vie du logiciel. De ce fait, en général, imposer la correspondance exacte des propriétés est inutilement rigoureux et peut mener à des résultats insatisfaisants. Ainsi, une étape préliminaire est la définition d'une mesure moins rigoureuse de similitude avec un comportement plus réaliste. Une fois qu'un tel critère de similitude a été défini, chaque objet peut être tracé dans l'ensemble de ceux qui lui ressemblent le plus.

Selon la taxonomie présentée, plusieurs méthodes sont développées dans la thèse permettant de récupérer des liens de traçabilité entre entités logicielles. Chaque méthode alternativement a été empiriquement évaluée en terme de précision (et performance) sur les objets logiciels disponibles. Les méthodes et les résultats principaux peuvent être récapitulés comme suit.

Liens entre la documentation textuelle et le code source

Des approches déterministes aussi bien que probabilistes ont été développées et appliquées à plusieurs systèmes logiciels (Albergate - Chapitre 3, LEDA - Chapitre 5, Linux - Chapitre 6, etc.). En particulier, les espaces vectoriels et les modèles

probabilistes de langage ont été appliqués aux classes pour tracer les liens entre le code source C++ et les pages de manuels et entre le code Java et les spécifications fonctionnelles relatives.

Tel que montré au Chapitre 3, les deux modèles réalisent 100% de rappel avec presque le même nombre de documents récupérés. Cependant, les modèles probabilistes de langage atteignent les valeurs de rappel les plus élevées (moins de 100%) avec un plus petit nombre de documents repérés et montre une meilleure performance lorsqu'un rappel de 100% est requis.

L'évaluation des performances des approches par rapport à l'outil `grep` démontre les avantages des technologies plus sophistiquées. Il apparaît aussi qu'à l'augmentation de la distance entre les objets logiciels, les résultats de `grep` se détériorent.

Les meilleurs résultats ont été réalisés une fois que la normalisation des textes était appliquée, en éliminant les mots d'arrêt, en réduisant les termes à leur racine et ainsi en acceptant implicitement un critère de correspondance moins strict.

Cependant, la normalisation des textes dans certains cas peut échouer dans la reconduction des documents et du code source à un vocabulaire commun. En effet, l'idée clef est que les connaissances du domaine d'application utilisées par les programmeurs en écrivant le code aient été capturées par les identificateurs mnémoniques. Selon cette hypothèse, le vocabulaire des identificateurs dans le code source partage un nombre significatif d'éléments avec le vocabulaire de la documentation.

Bien que cette conjecture soit soutenue par les résultats obtenus dans les deux cas étudiés, l'efficacité de la méthode devient moins prononcée lorsque le nombre des mots en commun entre les identificateurs du code de source et les termes de la documentation décroît.

Pour surmonter cette limitation, une nouvelle approche basée sur des modèles bigram de langage (séquences de deux mots) a été conçue. À la base de cette méthode est l'hypothèse que les programmeurs ont une tendance à traiter les connaissances du domaine d'application de façon cohérente, en appliquant une série de règles inconnues,

durant l'écriture du code et plus précisément en choisissant les noms et les termes des programmes selon une certaine logique.

Ainsi, les termes inclus dans des fragments de code différents, liés aux mêmes concepts et aux mêmes documents de niveau supérieur, vont en toute probabilité être pareils ou très similaires.

La méthode infère à partir d'un ensemble initial de liens le comportement des programmeurs, c.-à-d. les règles adoptées par les programmeurs pour choisir les identificateurs. Ces règles sont implicitement représentées par les distributions de probabilité conjointe estimées sur les ensembles de liens connus.

La méthode a été appliquée à trois systèmes logiciels différents; les entités de code telles que les classes et les fichiers ont été mises en correspondance avec des spécifications fonctionnelles; la précision a été évaluée en comparant les résultats aux matrices de traçabilité fournies par les développeurs des systèmes. Les trois cas d'étude peuvent être pensés comme représentatifs de différentes approches de développement, de différents langages et de différents outils.

Remarquablement, dans la plupart des cas, l'augmentation des informations d'apprentissage disponibles a amélioré les performances des méthodes; en d'autres termes, la méthode apprend des données disponibles (c.-à-d. la distribution de probabilité conjointe semble efficacement capturer les règles qui ont été appliquées en choisissant les identificateurs). Par conséquent, la tâche de rétablir les liens de traçabilité peut être rendue plus facile en appliquant des normes appropriées de codage lors du développement des logiciels.

Liens entre la conception par objets et le code source, et entre différents fragments de code source

Les documents de conception sont souvent incohérents par rapport à l'implantation.

De plus, le logiciel industriel, et spécialement celui développé avec la technologie OO, est souvent construit en utilisant une stratégie par composantes ou par bibliothèques de logiciels.

Un outil de vérification de conformité entre la conception et le code doit tenir compte de la distance "physiologique" entre la conception et le code ou entre versions subséquentes; il doit aussi être robuste par rapport aux incohérences entre la conception et le code source.

Le processus de rétablissement exploite le calcul de la distance entre textes et utilise l'algorithme de concordance maximale pour déterminer les liens de traçabilité entre la conception et le code. Le résultat est une mesure de similarité associée aux paires (*classes de code*, *classes de conception*) qui peuvent être classifiées comme égales et correspondantes ou non correspondantes selon un seuil de probabilité maximale.

Le test de conformité entre code et conception a été appliqué à un système industriel et il a obtenu un niveau de traçabilité moyen de 0.971, avec une moyenne de 6.37 classes non classifiées dans la conception. Avant d'appliquer le classificateur à vraisemblance maximale, le niveau de traçabilité moyen était 0.890 et le nombre de classes supprimées 2.24.

La similarité entre versions subséquentes d'un système OO est immédiatement dérivée de l'approche conçue pour tracer la conception OO dans le code. Par la rétro-ingénierie, la conception "telle-quelle" correspondant à un logiciel donné est obtenue. Sur cette conception, la distance entre textes et la concordance maximale, détaillés dans le Chapitre 5, produisent une carte de traçabilité. Il faut noter que la seule différence significative entre l'approche proposée pour tracer la conception dans le code et pour tracer le code dans le code OO est l'adoption d'un schéma de poids différent pour les attributs et pour les méthodes qui permet ainsi d'atteindre un compromis entre les influences relatives.

La méthode a été appliquée à deux cas d'étude: le premier cas d'étude a été utilisé pour évaluer les paramètres, c.-à-d. les poids de correspondance et le seuil de taille. Le deuxième cas d'étude a été réalisé pour valider ultérieurement les poids et le seuil.

Pour les systèmes logiciels analysés, les meilleurs résultats ont été atteints avec un poids de 30 % pour les classes et les noms ($\lambda_c = 30\%$) et un poids de 70 % pour les attributs et méthodes.

Les liens de traçabilité récupérés, tant au niveau de l'interface que de l'implantation, ont été exploités pour établir des modèles d'estimation du volume des changements à partir d'un estimé des classes intéressées. Estimer la dimension d'un changement est une étape de base dans l'application des modèles d'évaluation de coûts en entretien de logiciels.

La similitude basée sur les clones

L'évolution de grands systèmes logiciels a été caractérisée par des mesures de similarité au niveau des versions ou des systèmes. Les quatre métriques présentées sont représentatives des changements et/ou des similitudes, entre des systèmes logiciels, des sous-systèmes ou possiblement des versions subséquentes d'un système logiciel.

La faisabilité du calcul des métriques a été démontrée sur environ 400 versions du noyau de Linux et sur l'évolution d'un grand logiciel observé. Linux est un système d'exploitation multiplateforme, de plusieurs millions de lignes de code, largement adoptée à travers le monde. Les données recueillies permettent de représenter avec des métriques de haut niveau les différences et les similitudes entre les versions du noyau de Linux. Des 400 versions considérées, dix-neuf d'entre elles, de 2.4.0 à 2.4.18, ont été traitées et analysées en détail, en identifiant la duplication de code parmi des sous-systèmes de Linux au moyen d'une approche basée sur les métriques. Les résultats obtenus soutiennent l'hypothèse que le système Linux ne contient pas une portion considérable de code dupliqué. De plus, la duplication de code tend à rester stable à travers les versions, suggérant ainsi une structure relativement stable qui évolue en douceur sans évidence flagrante de dégradation.

TABLE OF CONTENTS

DEDICATION iv

ACKNOWLEDGEMENTS v

RÉSUMÉ vi

ABSTRACT x

CONDENSÉ xiii

LIST OF TABLES xxv

LIST OF FIGURES xxix

LIST OF ACRONYMS xxxii

LIST OF APPENDICES xxxiv

PREFACE xxxv

CHAPTERS

1. Introduction 1
   - 1.1 Introduction 1
   - 1.2 Background Notions 5
     - 1.2.1 Definition of a Distance 7
     - 1.2.2 Definition of a Similarity 9
     - 1.2.3 Inner Product and Norm 11
     - 1.2.4 Measure and Probability Definitions 12

1.3 Feature Extraction 14
1.3.1 Source Code Processing 15
1.3.2 OO Design Processing and AOL Extraction 19
1.3.3 Informal Textual Documentation Processing 21
1.3.4 Assessing Results 22
1.4 Related Work 23
1.5 Chapter Summary 26
2. Design to Code Traceability 28
2.1 Mapping Model 28
2.1.1 Classification of Matched and Unmatched Class Pairs 31
2.1.2 Relations Traceability 33
2.1.3 Context Information 33
2.1.4 Difference Visualization 34
2.2 Experimental Results 35
2.2.1 Test Suite 36
2.2.2 Average Match Figures 37
2.2.3 Design-code Match of the Relations 41
2.2.4 Detailed Analysis of an Example Component 43
2.3 Chapter Summary 45
3. Manuals and Requirements to Code Traceability 48
3.1 Mapping Model 48
3.1.1 Probabilistic IR Based on Language Models 48
3.1.2 Traditional Probabilistic IR Model 51
3.1.3 Vector Space IR Model 54
3.2 Experimental Results 55
3.2.1 Test Suite 56
3.2.2 LEDA Case Study 56
3.2.3 Albergate Case Study 58
3.2.4 Probabilistic Versus Vector Space Model 59

3.2.5 Comparing IR Models with `grep` 62
3.2.6 Benefits of IR in a Traceability Recovery Process . . 63
3.2.7 Considerations on Effort Saving and Document Granularity 64
3.2.8 Retrieving a Variable Number of Documents 66
3.3 Chapter Summary 68
4. A Bigram Language Model 70
4.1 Mapping Model 70
4.1.1 Programmer's Cognitive Model 71
4.1.2 Mathematical Model 74
4.1.3 Model Generation 78
4.1.4 Model Assessment 78
4.2 Traceability Problems in Software Systems 79
4.2.1 Partially Automatic-Generated Code 80
4.2.2 Totally Automatic-Generated Code 81
4.2.3 COTS and Reused Code 82
4.2.4 External Architectures 82
4.2.5 Design and Implementation-Level Components . . . 83
4.3 The Traceability Recovery Process 84
4.3.1 Requirements Processing 85
4.3.2 Code Processing 86
4.3.3 Traceability Map Recovery 87
4.3.4 Tool Support 87
4.4 Experimental Results 87
4.4.1 Test Suite 88
4.4.2 Case Study Results 88
4.5 Chapter Summary 97
5. OO Code to Code Traceability 98
5.1 Mapping Model 98

5.1.1 Mapping Equations 99
5.2 Traceability Recovery Process 100
5.2.1 Software Metric Extraction 100
5.2.2 AOL Parsing and Similarity Computation 101
5.2.3 Release Comparison 102
5.2.4 Code Difference Computation 102
5.2.5 Tool Support 102
5.3 Experimental Results 103
5.3.1 Test Suite 104
5.3.2 Setting Preliminary Weights 104
5.3.3 Assessing the Similarity Value Threshold 105
5.3.4 Assessing Weights and Thresholds 106
5.3.5 Analysis of the Distributions of Similarity Values . . 109
5.3.6 DDD Validation 110
5.3.7 Estimating the Size of Changes 111
5.4 Chapter Summary 115
6. Large Procedural Systems 116
6.1 Mapping Model 116
6.1.1 Mapping Equations 117
6.2 Experimental Results 119
6.2.1 Test Suite 121
6.2.2 Linux Evolution Across Releases 124
6.2.3 Linux 2.4.18 Intra Release Evolution 128
6.2.4 Linux 2.4.18 Self Similarity 133
6.3 Chapter Summary 137
7. Conclusions 138
7.1 Conclusions 138
7.1.1 Free Text Documentation 139
7.1.2 Bigram Language Model 141

7.1.3 OO Design to Code and OO Code to Code Mapping 143
7.1.4 Clone Based Similarity 144
7.2 Chapter Summary 145

BIBLIOGRAPHY 147

APPENDICES 160

LIST OF TABLES

2.1 Deleted classes and average similarity measures for design components, as resulting from the maximum match algorithm. 37

2.2 Columns 2-4 report results on raw data; columns 5-7 those obtained by polynomial fit. 39

2.3 Deleted classes and average similarity measures for design components, as resulting from the maximum likelihood classifier. 40

2.4 Number of generalization, association, and aggregation relations of design classes found among corresponding code classes over number of relations present in the design. 41

2.5 Attribute match for class `Wct_DistItem` from component C16. For each pair of attributes the associated similarity measure is shown. . 45

3.1 The traditional probabilistic approach compared to the language model approach; Albergate results were obtained with the improved process. 53

3.2 LEDA results. 57

3.3 Albergate results with improved process. 59

3.4 Albergate results with simplified process. 62

3.5 grep results: number of queries, retrieved empty sets, mean and max sizes of the retrieved sets. 63

3.6 Average results. 64

3.7 Detailed results of Group A. 64

3.8 Detailed results of Group B. 65

3.9 Albergate results using a threshold. 67

4.1 Albergate $\lambda'_k = 0$ traceability results. 90

4.2 Albergate single words plus bigrams traceability recovery results. . . 92

4.3 Transient Meter $\lambda'_k = 0$ baseline results. 92

4.4 Transient Meter: $\lambda'_k = 0$ results after removal of automatic-generated identifiers. 93

4.5 Transient Meter: $\lambda'_k = 0$ final results after removal of non-traceable classes. 94

4.6 Transient Meter: single words plus bigrams traceability results. . . . 95

4.7 Library Management: $\lambda'_k = 0$ results after morphological analysis. . 95

4.8 Library Management: $\lambda'_k = 0$ results after removing automatic-generated identifiers. 96

4.9 Library Management: $\lambda'_k = 0$ results after removing non-traceable files. 96

4.10 Library Management: single words plus bigrams traceability results. 96

5.1 Weights used to compare the different releases of LEDA and result summary. 104

5.2 Results of comparing LEDA releases: added, deleted, modified, unchanged LOC and classes. 105

5.3 Results of comparing DDD relevant releases: added, deleted, modified, unchanged LOC and classes. 110

5.4 DDD added classes, modified classes and multivariate model parameters. 113

5.5 DDD added classes, modified classes and multivariate model cross validation performances. 113

5.6 DDD added classes and modified methods model parameters. . . . 114

5.7 DDD added classes, modified methods cross validation performances. 114

6.1 Linux kernels most important events. 123

6.2 CRs $\geq 1\%$ among major subsystems. 135

6.3 CRs $\geq 10\%$ among `mm` architecture dependent code. 135

6.4 CRs $\geq 5\%$ among `drivers`. 136

A.1 Albergate traceability matrix summary. 161

D.1 Main features of available Library of Efficient Data Types and Algorithms (LEDA) releases. 166

E.1 Main features of relevant Data Display Debugger (DDD) releases. . 168

G.1 Classes and attributes in the design and code for each component. . 171

LIST OF FIGURES

2.1 Bipartite graph. 30

2.2 Matched and unmatched entities. 31

2.3 Design to code matching process. 35

2.4 Finding the classification threshold. *(a):* the misclassification error E(t), with component *C1* excluded, is plotted as a function of the classification threshold, *t*. *(b):* quadratic fitting of E(t) in the neighborhood of t_{min}. 38

2.5 Pair difference diagram for component C16. Green classes (dark grey) are the three at the top, green to yellow ones are in the middle (light grey), while some of the red classes (dark grey) are shown at the bottom. Attributes have been omitted for clarity. 44

3.1 Traceability link recovery process. 55

3.2 LEDA precision/recall results. 58

3.3 Albergate precision/recall results with improved process. 60

3.4 Albergate precision/recall results. 61

4.1 How the programmer can map concepts into code. 72

4.2 Traceability recovery process. 84

4.3 Example of traceability links. 85

5.1 Traceability link recovery method. 101

5.2 Distributions of class names and properties σ values for the LEDA releases. 106

5.3 Distributions of the $\overline{\sigma}$ values for the LEDA releases with $\lambda_c = 30\%$ . . 107

5.4 Distributions of the $\overline{\sigma}$ values for the LEDA releases ($\lambda_c = 30\%$) computed by the Maximum-Match Algorithm 108

6.1 Linux size evolution. 120

6.2 Similarity metrics computation process. 121

6.3 CR(A,B) and OVL(A,B) for stable and unstable releases. 125

6.4 Common ratio $CR(1.0, R_k)$ k varying from 1.0 to 2.4. 126

6.5 Common ratio $CR(R_k, 2.4)$ k varying from 1.0 to 2.4. 127

6.6 $ACCS(R_k, R_{k+1})$ k varying from 1.0 to 2.4. 129

6.7 Overall evolution of common ratio. 130

6.8 Evolution of common ratio between `mm` and `fs`. 131

6.9 Evolution of common ratio between `mips` and `mips64` code inside the `mm` subsystem. 132

6.10 Two examples of clones found. 134

LIST OF ACRONYMS

ACCS	Average Common Cluster Size
AOL	Abstract Object Language
ASCS	Average Single Cluster Size
AST	Abstract Syntax Tree
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off The Shelf
CR	Common Ratio
DDD	Data Display Debugger
GUI	Graphical User Interface
IDE	Integrated Desktop Environment
IDL	Interface Definition Language
IR	Information Retrieval
LEDA	Library of Efficient Data Types and Algorithms
LOCs	Lines of Code
OAV	Overall AVerage
ODL	Object Design Language
OMT	Object Modeling Technique

OO . . . . . Object-Oriented

OOD . . . . Object-Oriented Designer

ORB . . . . Object Request Broker

RAD . . . . Rapid Application Development

REI . . . . . Recovery Effort Index

SRS . . . . . Software Requirements Specifications

UML . . . . Unified Modeling Language

LIST OF APPENDICES

A. Albergate 161

B. Transient Meter 163

C. Library Management Software 165

D. LEDA 166

E. DDD 168

F. AOL 169

G. Industrial Software Characteristics 171

PREFACE

Software systems continuously evolve to meet ever-changing user needs. Changes may be driven by market pressure, adaptation to legislation, or improvement needs.

Numerous international organizations have recommended that software maintenance and evolution activities should follow precise phases in order to ensure consistency between the different software artifacts such as between the documentation and the source code. For example, the IEEE 1219 standard suggests that a maintenance process should encompass the following phases:

- Problem or modification identification, classification and prioritization;
- Analysis;
- Design;
- Implementation;
- System testing;
- Acceptance test; and
- Delivery.

However, in today's competitive market, the pressure to increase productivity results in the tailoring of some software processes especially development and evolution processes, to eliminate unproductive activities. Thus, in practice, the outlined process, or any similar one, is rarely adopted. Maintenance or evolution activities are focused on code fixing and seldom followed by documenting the changes applied.

As a consequence, the maintainability of systems and, in general, their quality, tend to deteriorate. Documentation, executables, sub-systems, modules and files or

directories organization, the *as-is* architecture, may no longer correspond to the original software architecture. Software evolution activities rapidly become extremely difficult as any change may produce unpredictable side effects on other portions of the system. Indeed, repeated undocumented maintenance interventions on the code reduce the user's understanding of the system and tend to substituting vague perceptions for reliable information on the software.

In this situation, some form of software system re-engineering activities aimed at recovering any mapping between software artifacts (e.g., free text documentation and code), or between subsequent releases of a software system is desirable. However, establishing and maintaining traceability links and consistency between software artifacts, produced or modified in the software life-cycle, is costly and time-consuming. Thus, approaches, methods and tools are needed to help developers maintain, evolve, and assess traceability links that would reduce the need for human intervention. The research carried out and presented in this thesis stems from these considerations and from the fact that since development activities cover only a fraction of the entire software life-cycle, any method or tool which reduces the evolution costs while ensuring a higher software quality would reduce the total cost of ownership.

CHAPTER 1: Introduction

1.1 Introduction

The complexity of developing software systems is usually approached by following a phased software development process in which the activities performed within a phase refine the artifacts produced by the previous one, possibly in an iterative way. Requirement analysis, design and coding are typically present in almost any software development process. Unfortunately, a phased process does not automatically help to ensure consistency between and within software artifacts developed throughout the product life-cycle (e.g., between requirements and source code).

According to IEEE Glossary, traceability is defined as *the degree to which a relationship can be established between two or more products of the development process.* Pfleeger (Pfleeger 2001) made the distinction between horizontal traceability, which is defined as the traceability between documents at the same level of the software life-cycle (e.g., the traceability between related requirements), and vertical traceability, which is defined as the traceability between documents at different levels of the software life-cycle (e.g., the traceability between requirements and source code components). Traceability creates links between and within software artifacts. The goal of this thesis is to develop methods and approaches to recover traceability links, help evaluate traceability links, and more generally support software system evolution by ensuring the consistency of software artifacts.

Very often source code evolves, but documentation is not updated; maintaining consistency and traceability information between software artifacts is costly and time-

consuming, frequently neglected due to the pressure to reduce the time to market or move on to the next software change. Furthermore, due to outsourcing or turnover, the people who developed or maintained a system may no longer be available. Thus, the system itself is often the only reliable source of information.

Large software systems continuously evolve (Lehman 1980) to meet ever changing user needs. Changes may be driven by market pressure, adaptation to legislation, or improvement needs. Maintaining traceability links between subsequent releases of a software system is important for supporting maintenance activity, evaluating the volume of changed code, highlighting effort versus changed code inconsistencies, and assessing the change history. The activity of checking the compliance of two software versions can be greatly assisted by automatic tools that help a programmer to identify regions of code that do not match between two software releases. Context `diff` between files may be applied to establish similarity between entities (files, classes, methods) belonging to different software releases. However, results are too coarse and very difficult to summarize, visualize and interpret.

Most documentation that accompanies large software systems consists of free text documents expressed in natural languages. Examples include requirements and design documents, user manuals, error logs, maintenance journals, design decisions, reports from inspection and review sessions, and also annotations of individual programmers and teams. In addition, free text documents often capture the available knowledge of the application domain which can be expressed in the form of bodies of laws and regulations, or in the form of technical and scientific handbooks. Even when formal or semi-formal models are applied, free text is largely used, either to add semantic and context information in the form of comments, or to facilitate the understanding of the formal models for non-technical readers. In fact, diagrammatic representations are often supplemented with free text descriptions that convey information not captured by the diagrams themselves. Even Z specification documents (Potter et al. 1991) are typically an amalgam of mathematics, which is precise and supports reasoning, and explanatory texts.

Establishing traceability links between the free text documentation associated with the development and maintenance cycle of a software system and its source code can be helpful in a number of tasks. A few notable examples are:

- Program comprehension; existing cognition accept that that program comprehension occurs in a bottom-up manner (Pennington 1987b), a top-down manner (Brooks 1983; Soloway and Ehrlich 1994), or some combination of the two (Letovsky 1986; Mayrhauser and Vans 1993; Mayrhauser and Vans 1994; Mayrhauser and Vans 1996). They also agree that programmers use different types of knowledge during program comprehension, ranging from domain specific knowledge to general programming knowledge (Brooks 1983; Shneiderman and Mayer 1979; Vessey 1985). Traceability links between areas of code and related sections of free text documents, such as an application domain handbook, specification document, set of design documents, or manual pages, aid both top-down and bottom-up comprehension. In top-down comprehension, once a hypothesis has been formulated, traceability links may provide hints on where to look for signals that either confirm or refute it. In bottom-up comprehension, the main role of traceability links is to assist programmers in the assignment of a concept to a chunk of code and in the aggregation of chunks into more abstract concepts.

- Maintenance; as the software industry matured, companies built up a sizable number of legacy systems. A legacy system is an old system which is valuable for the corporation which owns and often developed it (Bennett 1995; Brodie and Stonebraker 1995). For the purpose of maintaining legacy systems, design recovery (Chikofsky and Cross 1990) may be performed requiring different sources of information, such as source code, design documentation, personal experience and general knowledge about problem and application domains (Biggerstaff 1989; Merlo et al. 1993). Central to design recovery is representation (Rugaber and Clayton 1993), for which different schemes have been used

and described (Rich and Waters 1990; Biggerstaff et al. 1993). Traceability links between code and other sources of information are helpful in performing the combined analysis of heterogeneous information and, ultimately, associating domain concepts with code fragments and vice-versa.

- Requirement tracing; traceability links between the requirement specification document and the code are a key to locate the areas of code that contribute to implementing specific user functionality (Pinhero and Goguen 1996). This is helpful in assessing the completeness of an implementation with respect to stated requirements, devising test cases, and inferring requirement coverage from structure coverage during testing. Traceability links between requirements and code can also help to identify the code areas directly affected by a maintenance request as stated by an end-user. Finally, they are useful during code inspection to provide developers with clues about the goals of the code at hand, and during quality assessment, for example, to single out loosely-coupled areas of code that implement heterogeneous requirements (Concling and Bergen 1988; Ramesh and Dhar 1992).

- Impact analysis; a major goal of impact analysis is the identification of the work products affected by a proposed change (Arnold and Bohner 1993). Changes may initially have been made to any of the documents that comprise a system, or to the code itself, and then have to be propagated to other work products (Fyson and Boldyreff 1998; Turver and Munro 1994). As an example, enhancing an existing system by adding new functions is in most cases initiated at the requirement specification level and changes are then propagated through design documents down to the source code. Conversely, a change of an algorithm or a data structure may start at the code level and then be documented in the relevant sections of the design documentation. Such activities can be greatly assisted by establishing traceability links between code and free text documentation.

- Reuse of existing software; deriving reusable assets from existing software systems has emerged as a winning approach to promote the practice of reuse in industry (Caldiera and Basili 1991; Canfora et al. 1994). Means to trace code to free text documents are helpful in locating reuse-candidate components. Indeed, since existing software often has not been produced with reuse in mind, knowledge about its functionality and the application domain concepts it implements is spread over several artifacts, including requirement specification documents, manual pages, design documents, and the code itself. In addition, traceability links between the code and an application domain handbook are helpful in indexing reusable assets for potential reuse based on a user request (Arnold and Stepowey 1987; Burton et al. 1987; Frakes and Nejmeh 1987).

1.2 Background Notions

Unlike other reverse engineering problems, recovering traceability links between software artifacts (e.g., free text documentation and source code or between subsequent releases of a given artifact) cannot simply be based on compiler technology because of the difficulty of applying syntactic analysis to natural language. Furthermore, the modeling paradigm should analyze artifacts at very different abstraction levels ranging from the source code level to design level, documentation level, and requirements level.

Bunge's ontology (Bunge 1977; Bunge 1979) has been a source of inspiration in the Object-Oriented (OO) domain. According to this ontology, objects can be viewed as *substantial individuals* which possess *properties*. Chidamber and Kemerer (Chidamber and Kemerer 1994) proposed a representation of *substantial individuals*, that is, objects, as a finite collection of properties:

$$X =< x, P(x) > \tag{1.1}$$

where the object X, is identified by its unique identifier, x, and $P(x)$ is its finite collection of properties. If similarity between two individuals is defined as the intersection of their sets of properties, two individuals are indistinguishable if and only if they share the same name and possess the same collection of properties.

In general, two individuals X and Y may possess different properties. Comparing them for similarity translates into checking the similarity of the their properties. The more properties they share, the more similar they are. If two individuals are identical, then no matter how their similarity is measured, similarity must evaluate to 1; conversely, the more differences two individual exhibit, the less similar they are (Lin 1998).

This idea is easily generalized to represent all the artifacts which characterize a software system or which are produced along the software life-cycle. Since Bunge's ontology does not specify the nature of the represented object, it can be applied to various software artifacts (i.e., requirements, design, user manual or source code), once an identity and a set of properties are defined. However, a criterion imposing *substantial individuals* identity is unnecessarily stringent and may lead to unsatisfactory results.

From a different point of view, traceability recovery may be thought of as an Information Retrieval (IR) task. In fact, IR systems are concerned with retrieval based on user information needs (Frakes and Baeza-Yates 1992) of documents from usually very large document databases. They pre-process the collection of documents for retrieval through an indexing process; user needs are captured by phrases which are themselves indexed and used to rank the documents. IR has proved useful in many areas, including the management of huge scientific and legal literature, office automation, and complex software engineering projects.

A widely used approach to retrieve documents from the document space is *ranked retrieval*, which returns a ranked list of documents (Harman 1992). Two IR models have been widely applied: the probabilistic model and the vector space model. The vector space model treats documents and queries as vectors in an n-dimensional space,

where n is the number of indexing features such as words in dictionary. Documents are ranked against queries by computing a similarity or a distance function between the corresponding vectors. In the probabilistic model, free-text documents are ranked according to the probability of relevance to a query computed on a statistical basis. In other words, the problem of recovering traceability links may be thought of as an IR problem where software artifacts play the roles of documents and queries. The problem becomes that of defining a similarity criterion or a distance between software artifacts i.e., substantial individuals.

The different models require the definition of a similarity criterion or a distance between software artifacts or, in the probabilistic model, they require the definition of the probability for any given artifact being relevant to a query. Therefore, in the following subsections the essential notions related to distance, similarity and metric spaces will be summarized.

1.2.1 Definition of a Distance

A *metric space* (X, d) consists of a set X in which is defined a *distance function* $d : X \times X \rightarrow R$, which assigns to each pair of points of X a distance between them, and which satisfies the following four axioms:

1. $d(x, y) \geq 0$ for all points x and y of X
2. $d(x, y) = d(y, x)$ for all points x and y of X
3. $d(x, z) \leq d(x, y) + d(y, z)$ for all points x, y and z of X
4. $d(x, y) = 0$ if and only if the points x and y coincide

The X set may or may not be a vector space. For example,it may not be closed under the *addition* operation. Nevertheless, X elements may be represented with multiple components as elements of a vector space usually are. The function d defines a topology over X; d is also referred to as a metric over X.

It is worth noting that sometimes only axioms 3 and 4 are required, since axioms 1 and 2 may be derived. If axiom 4 is not required, or it does not hold for the considered function, the function d is called a *pseudo-metric*, while if axiom 2 is not ensured, d is said to be a *quasi-metric*; finally, if the triangular inequality (i.e., axiom 3) does not hold it is said to be a *semi-metric*.

Clearly, a quasi-metric can be transformed into a metric, if needed, by applying the following transformation:

$$d'(x,y) = \frac{d(x,y) + d(y,x)}{2} \tag{1.2}$$

Given a metric d, it is always possible to bound its value between 0 and $1/k$ via:

$$d'(x,y) = \frac{d(x,y)}{1 + kd(x,y)} \tag{1.3}$$

In particular, for k=1, the values are bounded into the interval $[0,1]$. The most popular metric for continuous feature is the *Euclidean distance*:

$$d_2(x,y) = [\sum_{k=1}^{n}(x_k - y_k)^2]^{\frac{1}{2}} \tag{1.4}$$

which is the special case ($p = 2$) of the Minkowski metric:

$$d_p(x,y) = [\sum_{k=1}^{n}|x_k - y_k|^p]^{\frac{1}{p}} \tag{1.5}$$

Widely used distances are the *Mahanalabonis* distance and the *city block* distance, also referred to as Manhattan distance. The latter can be obtained from the Minkowski metric when p=1; while the Euclidean distance represents the shortest path between two points, the Manhattan distance is the sum of the length of the distances along each dimension.

When dealing with multidimensional representations, the Minkowki, and thus the Euclidean, distance tend to be dominated by the largest scaled vector dimension; in such a situation a weighting schema may reduce the undesired effect. Using a matrix

notation where x^T represents the transpose of the vector x and A^{-1} the inverse of the matrix A, the Mahanalabonis distance is defined as:

$$d(x,y) = \sqrt{(x-y)\Sigma^{-1}(x-y)^T} \tag{1.6}$$

where Σ is the covariance matrix. The Mahanalabonis distance tends to alleviate the problem of different scales by applying a whitening transformation to the data. When the components are extracted from independent Gaussian distributions with unitary variance (i.e., Σ is the identity matrix) the equation (1.6) is just another way to express the usual Euclidean distance.

1.2.2 Definition of a Similarity

Distance and similarity are two related concepts. The mathematical definition reflects the common agreement that the more similar two things are, the lower their *distance* is; i.e., the *closer* they are.

Let X be a set, a function $\sigma : X \times X \rightarrow [0,1]$ is said to be a similarity if and only if (Lin 1998; Dominich 2000):

1. for all points x, y in X: $0 \leq \sigma(x,y) \leq 1$
2. for all points x, y in X: $\sigma(x,y) = \sigma(y,x)$
3. for all points x, y in X if $x = y \Rightarrow \sigma(x,y) = 1$

X may be a collection of strings or a collection of arrays of integers e.g., representing the frequency of words in a text for a given dictionary.

Clearly, a distance is induced by a similarity via the transformation:

$$d(x,y) = 1 - \sigma(x,y) \tag{1.7}$$

Notice that the satisfaction of the triangular inequality is not required for a similarity and thus in general the above will represent a semi-metric. Conversely, setting $k = 1$ in equation (1.3) allows for transformation of any distance into a similarity.

A frequently adopted similarity definition is based on the cosine of the angle between two vectors:

$$\sigma(x,y) = |[\frac{\sum_k x_k y_k}{\sqrt{\sum_i (x_i)^2 \sum_j (y_j)^2}}]| \tag{1.8}$$

The quantity between square brackets is the un-centered correlation between x and y, correlation whose values range between -1 and +1. More generally, the absolute value of a correlation (i.e., Pearson, Spearman or Kendal coefficient) defines a similarity and thus a correlation may also be interpreted as semi-metric based on the equation (1.7). A related similarity is the *Dice* coefficient defined as:

$$\sigma(x,y) = |\frac{2\sum_k x_k y_k}{\sum_i (x_i)^2 + \sum_j (y_j)^2}| \tag{1.9}$$

In software engineering it is often necessary to evaluate the similarity of two artifacts, such as two different designs or an architectural design and a detailed design produced early in the software life cycle. In such a case, similarity between artifacts may be constructed based on similarity between strings of characters. If x and y are two strings, a similarity may be expressed in term of the *edit* distance d_e (Cormen et al. 1990), which is the number of characters that must be inserted, deleted, or substituted to map one string onto the other:

$$\sigma(x,y) = 1 - d_e(x,y)/(|x| + |y|) \tag{1.10}$$

where $|x|$ is the number of character composing x i.e., the length.

Since the upper boundary for the edit distance is the sum of the lengths of the strings ($|x| + |y|$), the above similarity ranges between 0 and 1; it is 0 when the two strings x and y have no character in common, and it is 1 when they coincide.

This similarity will be used to recover traceability links between design and code. High level similarities are constructed starting from this definition.

1.2.3 Inner Product and Norm

Let V be a vector space over a field F (either C or R), let x and y be vectors of V, a function $f(x,y)$, $f : V \times V \rightarrow F$, denoted as $< x,y >$ is an inner product if it satisfies the following properties:

1. for all x in V: $< x,x > \geq 0$ and $< x,x >= 0 \Leftrightarrow x = 0$
2. for all scalar $a \in F$ and for all x, y , z in V: $< z, ax+y >= a < z,x > + < z,y >$
3. for all x, y in V; $< x,y >=< y,x >^*$

Where x^* is the x complex conjugate, the inner product, alternatively said scalar or dot product, is often defined over R^n as:

$$< x,y >= \sum_{k=1}^{n} x_i y_i$$

A norm is a function $\|\| : V \rightarrow F$ such that:

1. for all x in V: $\|x\| \geq 0$ and $\|x\| = 0 \Leftrightarrow x = 0$
2. for all scalar $a \in F$ and for all x in V: $\|ax\| = a\|x\|$
3. for all x, y in V; $\|x + y\| \leq \|x\| + \|y\|$

Notice that an inner vector space becomes a normed space by posing:

$$\|x\| = \sqrt{< x,x >}$$

Any normed vector space is turned into a metric space by posing:

$$d(x,y) = \|x - y\|$$

If the norm is induced by the scalar product, the usual Euclidean distance is obtained. Alternatively, let Q be an $n \times n$ definite or semidefinite symmetric matrix;

the equation $\|x\|_Q = xQx$ defines a norm. Notice that the Mahanalabonis distance corresponds to the choice $Q = \Sigma^{-1}$. In a normed space, with the norm induced by the scalar product, the similarities defined by equations (1.8, 1.9) have a compact representation in terms of scalar product and norm. For example, equation (1.8) is equivalent to:

$$\sigma(x, y) = \frac{| < x, y > |}{\sqrt{\|x\|\|y\|}}$$

1.2.4 Measure and Probability Definitions

Measure and probability definitions are based on the σ-algebra definition. A σ-algebra X over a set S is a non-empty collection of subsets of S that is closed under complements and countably infinite unions:

1. if ϕ is the empty set then $\phi \in X$;
2. if $A \in X$ then its complement is in X;
3. if A_i $i = 0, 1, \ldots$ is a collection of X elements then $\bigcup_k A_k \in X$.

A measure μ associates a volume, size, and weight to elements of the σ-algebra X, subsets of S. A function μ is a measure if and only if:

1. $\mu(\phi) = 0$ (ϕ is the empty set);
2. for any countable collection A_k of elements of X such that $A_i \bigcap A_j = \phi \; \forall i \neq j$, then $\mu(\bigcup_k A_k) = \sum_k \mu(A_k)$;

Since X is a σ-algebra, it must contain the empty set and be closed under the complement (i.e., $S \in X$) and the countable union of X elements. A probability is a measure $Pr()$, which satisfies the following additional axioms:

1. $\forall A \in X : 0 \leq Pr(A) \leq 1$;
2. $Pr(S) = 1$;

The axiom $Pr(S) = 1$ expresses the certainty that some event in the *universe* will occur. It may be argued that when tossing a coin twice, the first result does not influence the second; the two events, say H (heads) and T (tails), have nothing in common, and the occurrence of either one does not influence the other: they are *independent* events. In such a case, the probability of the joint event, $H \bigcap T$, often written as H, T, is just the product of the individual probabilities:

$$Pr(H \bigcap T) = Pr(H)Pr(T)$$

There is often the need to restrict the probability to a particular subset D with non-null probability measure; the conditional probability or the probability of A given D is defined as:

$$Pr(A|D) = \frac{Pr(A \bigcap D)}{Pr(D)}$$

The set (event A) is weighted with respect to the event D. Clearly, if A and D are independent events then $Pr(A|D) = Pr(A)$. The two events, A and B, are conditionally independent given a third event D if:

$$Pr(A, B|D) = Pr(A|D)Pr(B|D)$$

or equivalently:

$$Pr(A|B, D) = Pr(A|D)$$

In other words if D is true, knowing B does not change the probability of A.

A useful result is Baye's theorem which restates an *a-posteriori* probability in terms of a-priori and known probabilities. Suppose we know that one, and only one, of two hypotheses must be true, and we know the probability of the first one $Pr(Hp_1)$; we also know $Pr(Hp_2)$ since they partition the *universe* hypotheses space ($Hp_1 \bigcup Hp_2 = 1$). After observing an event E, to compute $Pr(Hp_1|E)$, Baye's theorem says:

$$Pr(Hp_1|E) = \frac{Pr(E|Hp_1)P(Hp_1)}{Pr(E)} = \frac{Pr(E|Hp_1)Pr(Hp_1)}{P(E|Hp_1)Pr(Hp_1) + Pr(E|Hp_2)Pr(Hp_2)}$$

Baye's theorem is immediately derived from the conditional probability definition and can be easily extended over collections of hypotheses.

Baye's theorem is the basis of many classification systems in software engineering as well as in many other disciplines. Suppose, for example, we have a collection of functional requirements $Fr_1, \ldots, Fr_n$ and we observe a chunk of code C implementing one or more functional requirements. In order to evaluate the probability:

$$Pr(Fr_k|C) = \frac{Pr(C|Fr_k)Pr(Fr_k)}{Pr(C)}$$

There is obviously no reason to believe that $Pr(C)$ depends on any particular functional requirement i.e., it may be safely assumed constant. Furthermore, since there is no evidence that one requirement is more *likely* than the other, the probability computation may be simplified to:

$$Pr(Fr_k|C) = Pr(C|Fr_k)$$

Therefore, to recover traceability links a reasonable choice is:

$$\widehat{Fr_k} = arg \max_{Fr_k} Pr(C|Fr_k)$$

1.3 Feature Extraction

To recover a traceability mapping, a necessary preliminary activity is to identify, select and represent the properties $P(x)$ characterizing the software artifacts. These properties depend on the artifact (e.g., software requirement specification, design or code), development paradigm (e.g., procedural or OO languages), development environment or application domain. Sometimes, preprocessing may be needed to extract and represent properties of interest. For example, an OO design developed

with an Integrated Desktop Environment (IDE) tool and represented in the tool with a proprietary language may be mapped into a textual notation for further processing; C or C++ source code may need pre-processing to extract information from the system. The tools needed to process informal documentation such as requirement documents or user manuals differ from those needed to analyze the source code. In the following subsections, the relevant aspects related to the different sources of information involved in traceability recovery will be discussed. The following sources of information are considered:

- Source code.
- OO design documentation.
- Informal textual documentation.

1.3.1 Source Code Processing

Parsing programming languages such as C or C++ poses several challenges. Besides the intrinsic programming language peculiarities (e.g., `union`, `struct`, `class`, function pointers, etc.), preprocessor directives must be managed. This is usually done by a dedicated compiler component, the *preprocessor* (e.g., the GNU `cpp`). Preprocessor directives are the usual way in which programmers obtain platform portability when developing in C or C++. Parsing multi-platform code where preprocessing directives are platform-dependent is equivalent to projecting the source code onto a given hardware/software configuration.

To obtain information on several platforms, at least two approaches are feasible:

- Pre-process and parse the code sources with different configurations.
- Construct a fictitious *reference* configuration assuming that each and every preprocessor `#ifdef` condition is *true*.

The first approach is unfortunately feasible only for small or medium size systems (e.g., see Chapter 5) when the entire software configuration to pre-process and compile the source code is available.

When millions of LOCs need to be managed, or when part of the system is missing (e.g., a middleware layer) it may not be realistic or feasible to consider all the possible software configurations or even a given specific configuration (e.g., see Chapter 6); if some included files are missing pre-processing is unfeasible.

Consider for example the Linux 2.4.0 kernel which runs on ten different processors. It would be almost impossible to pre-process the configuration for the S38 IBM machine on an i386 platform. Furthermore, it contains more than 7000 files, and 400 preprocessor switches drive the actual kernel configuration. Each preprocessor switch can assume three values:

- Y: the code is included in the compiled kernel.
- N (or commented switch): the code is excluded.
- M: a dynamically loadable module is produced.

Among the $10 * 400^3$ possibilities, there are obviously many meaningless configurations. For example, it is very unlikely that a machine has multiple different sound boards.

On the other hand, by constructing a fictitious reference configuration, no specific architecture is identified. This approach is well-suited for studying the software evolution, recovering high level traceability mapping between subsequent releases or identifying function clones among several platform-dependent sub-systems without recompiling the kernel.

The heuristic adopted to manage preprocessor directives in large software systems is based on the consideration that very often only the *then* part of an `#ifdef` is present; moreover, the *then* branch almost always contains more code than the *else* branch. Among the 3243 source (.c) files of the 2.4.0 kernel, 2172 contain at least

one `#ifdef`, whereas only 1140 files have an `#else` preprocessor directive. The actual number of `#ifdef`'s is by far larger (22134) than the number of `#else`'s (3565). In terms of volume (measured in Lines of Code (LOCs)), the *then* branch (about 300 KLOCs) is an order of magnitude larger than the `#else` part (about 20 KLOCs).

Preprocessor conditional statements are conceptually identical to traditional programming language *if then else fi* statements; that is, there may be an *else* part, *if* may be nested, an *else* part is assigned to the *nearest if* and they must be balanced. Since preprocessor directives, i.e., `#ifdef`, must be balanced, a parsing of the preprocessor statements can project the source on `then` branch. In other words the `#ifdef` conditions are forced to be `true`, thus extracting the *then* branches since only the source code of the `#then` part is retained by the pre-processor.

Unfortunately, there are few cases where this heuristic produces syntactically wrong C code. Namely, a C scope i.e., `{`, may be opened in the `then` part of an `#ifdef` subject to condition `EXP`, and the scope end `}` may be located in a different preprocessor statement within the `#else` part. This also means that the scope is closed by a combination of expressions where `EXP` is negated. This is the situation found, for example, in the Linux 2.4.0 `ultrastor.c` scsi driver. Due to the very low number of such cases (in the 2.4.0 kernel, twenty on about 48000 functions), these were considered pathological situations, detected and signaled for manual intervention.

1.3.1.1 Parsing Procedural Languages

Large C systems are likely to encompass a variety of mixed programming styles, programming patterns, idioms, coding standard and naming conventions. Most noticeably, both the ANSI-C and the old Kernighan & Ritchie style may be present. A tool inspired by island-driven parsing (Corazza et al. 1991; De Mori 1998) has been implemented to localize and extract function definitions. Once islands e.g., function bodies or signatures are identified, the in-between code is scanned, and the function definition extracted by means of a hand-coded parser. Once the function has been identified several kind of information may be extracted.

Following the approach proposed in (Mayrand et al. 1996; Buss et al. 1994), the functions extracted as illustrated above were represented on the basis of software metrics accounting for layout, size, control flow, function communication and coupling. In particular, each function was modeled by an array of 54 software metrics:

- Number of passed parameters.
- Number of LOCs.
- Number of statements.
- cyclomatic complexity.
- Number of used/defined local variables.
- Number of used/defined non-local variables.
- Software metrics accounting for the number of arithmetic and logic operators (++, -, >=, <, etc.).
- Number of function calls.
- Number of return/exit points.
- Number of structure/pointer access fields.
- Number of array accesses.
- Software metrics accounting for the number of language keywords (e.g., `while`, `if`, `do`).

Different sets of metrics could be adopted such as those used in (Mayrand et al. 1996; Buss et al. 1994), and there is no particular reason to prefer one set of metrics over the others. Since several software metrics are correlated (e.g., number of LOCs and number of statements, cyclomatic complexity and maximum nesting level), a sufficiently large number of metrics capture the information needed to represent a

function; in other words, when a large system is considered, the use of different sets of metrics does not significantly influence the results.

1.3.1.2 Parsing OO Languages

Extracting information about methods and class relationships from OO code is far more difficult than it is from design: results might have some degree of imprecision. In fact, given two classes and a relation between them, there are intrinsic ambiguities due to the choice left to programmers implementing the OO design, as to whether to consider such relation an association or an aggregation, unidirectional or bidirectional. Pointers, references, templates (e.g., `list<tree>`), and arrays (e.g., `Heap a[MAX]`) can represent both associations and aggregations. A discussion on the semantics of the architectural information extracted from the code can be found in (Woods et al. 1999).

To simplify the code processing and be conservative, an aggregation is recognized from code if and only if an instance of an object is stored as a data member of another object, or a template or object array data member is declared, even if a pointer could be used to create an object chunk. All remaining cases, i.e., object pointers and references, both as data members and formal parameters of methods, are represented as associations.

C++ and Java code related information are captured in intermediate code representation that encompasses the essentials of a class diagram in an OO design description language, the Abstract Object Language (AOL) representation. Furthermore, the set of software metrics detailed in 1.3.1.1, comments, and identifiers of classes and methods, are retained if needed.

1.3.2 OO Design Processing and AOL Extraction

To ensure independence from the specific CASE tools, design documents were represented via an intermediate language, the AOL representation. The language is a general-purpose design description language capable of expressing concepts available

at the design stage of OO software development.

The AOL specification is included in Appendix F; more details on AOL can be found in (Fiutem and Antoniol 1998; Antoniol et al. 1998; Antoniol et al. 2001b; Antoniol et al. 2000c). Essentially, AOL has been designed to capture OO concepts in a formalism independent of programming languages and tools. It is a general-purpose design description language that can express concepts available at the design stage of OO software development. It is based on Unified Modeling Language (UML), the notation that is becoming the standard in object oriented design. UML (Booch et al. 1997) is a visual description language with some limited textual specifications. Hence, many parts of the language were totally new while adhering to UML where textual specifications were available. At present, AOL covers only the UML part related to class diagrams. Since for class diagrams, the UML and Object Modeling Technique (OMT) notations are almost identical, AOL is compatible with OMT designs.

AOL resembles other OO design/interface specification languages such as Interface Definition Language (IDL) (Lamb 1987; OMG 1991) and Object Design Language (ODL) (Lea and Shank 1994). OO concepts such as attribute, method, inheritance, and information hiding are available in AOL. However, AOL was thought of as a light C++ design representation language; thus it does not have the expressive power of ODL or IDL, but at the same time it allows a simple representation of all the class diagram concepts. Aggregations and associations are widely used in C++ programs and designs: AOL explicitly represents them while the above mentioned languages do not. As a result, a limited effort is required to develop a tool to translate design or C++ into AOL.

A `CASE2AOL Translator` module has been implemented for the StP/OMT (STP 1996) tool to obtain an AOL specification of the internal object models from the StP/OMT repository, while the `Code2AOL Translator` works on C++ and Java code.

1.3.3 Informal Textual Documentation Processing

Each textual document such as requirement, manual page, etc. is processed and the dictionary of its words, scored by their frequencies, is extracted. Stop words (e.g., articles) are discarded; words successively undergo a normalization step of morphological analysis. Plural and singular conjugated verbs, as well as synonyms, are brought back to the radix. In summary, the following steps are applied:

1. First all upper case letters are transformed into lower case.
2. Second stop-words such as articles, punctuation, numbers, etc. are removed.
3. Finally a morphological analysis is used to convert plurals into singulars and to transform conjugated forms of verbs into infinitives.

It must be noted that, in general, these phases cannot be fully automated, and they are likely to remain semi-automatic; it is well known that a word's semantics may be context sensitive, and currently available natural language tools cannot fully disambiguate contextual semantics, nor deal with complex forms representing metaphoric speech (e.g., ... *the process should be killed* ...).

Linguistic processing is also applied to identifiers extracted from the source code. Words are first filtered by a stopper. Stop words removal is divided into two subphases. In the first, the stopper discards the following elements:

- Language reserved keywords and language predefined types.
- Short identifiers, commonly used as cycle counters or array indexes (e.g., `x`, `y`, `j`, `k`).
- Words discarded code's associated documentation.

In the second phase, identifiers obtained by concatenating two or more words (e.g., `room_number`, `payBill`, `clear_theForm`) are segmented and the stopper phase reapplied. Finally, for text documentation, a stemming phase, described below, is carried out.

It must be noted that there is need for human intervention: short words may convey relevant application domain information.

To help carry out the analysis, the following tools were implemented:

- A stemmer that removes stop-words such as articles, numbers, preposition, certain categories of verbs etc. from requirements and code. By modifying the stop word list the maintainer is left in charge of the final decision to remove a word or not.

- A morphological analyzer, based on a thesaurus produced by the maintainer, from the dictionary of all the possible terms of the requirements and code.

1.3.4 Assessing Results

Different criteria may be applied to assess the accuracy of the results. Besides visual inspection, manual verification and qualitative assessment, a more quantitative measure is needed.

When recovering traceability links under the assumption that an oracle exists i.e., the programmer provided the correct traceability map, two widely accepted IR metrics, namely *recall* and *precision* (Frakes and Baeza-Yates 1992) can be adopted. *Recall* is the ratio of the number of relevant documents retrieved for a given query over the total number of relevant documents for that query. *Precision* is the ratio of the number of relevant documents retrieved over the total number of documents retrieved.

A suitable choice is to select the first N documents in the list i.e., the most similar artifacts. In such a case, the behavior of Recall and Precision with different values of N need to be studied. The value of N is called *cut level.*

Recovering traceability links is a semi-automatic process. The main role of tools consists of restricting the search space, while recovering all artifacts (e.g., manual pages) relevant to each query (e.g., source code component). Without tool support, all the documents must be analyzed before discovering that a given artifact is not

described by any document; with a restricted search space, the number of items to analyze is generally much smaller. This means that high recall values (possibly 100%) should be pursued; in this case, higher precision values certainly reduce the effort required to discard false positives, i.e., artifacts that are retrieved but are not relevant to a given query.

Notice that recall is undefined for queries that do not have associated relevant documents. However, these queries may retrieve false positives that have to be discarded by the software engineer. To take into account such queries, the following aggregate formulas may be used:

$$Recall = \frac{\sum_i \#(Relevant_i \wedge Retrieved_i)}{\sum_i \#Relevant_i}$$

$$Precision = \frac{\sum_i \#(Relevant_i \wedge Retrieved_i)}{\sum_i \#Retrieved_i}$$

where i ranges over the entire query set, including the queries with no associated documents. These queries do not affect the computation of the recall ($Relevant_i$ is the empty set), while they negatively affect the computation of the precision whenever $Retrieved_i$ is not the empty set. This negative influence takes into account the effort required to discard false positives.

1.4 Related Work

The issue of recovering traceability links between code and free text documentation is not yet well understood and investigated, and very few contributions have been published in the past 20 years. However, a number of related papers have been published in the area of impact analysis. For example, Turver and Munro (Turver and Munro 1994) assumed the existence of some form of ripple propagation graph, describing relations between software artifacts, including both code and documentation, and focusing on the prediction of the effects of a maintenance change request,

not only on the source code, but also on the specification and design documents.

Boldyreff et al. (Boldyreff et al. 1996) presented a method for the identification of traceability links of high-level domain concepts using use all available maintenance information. Their definition of traceabilty is the one given in the IEEE standard for software maintenance (1993):

a thread from the requirement to the implementation, with respect to the specific development and operational environment

They extended to vertical and horizontal traceability the above definition to cope with the various existing relations between software artifacts.

The requirement traceability problem has been investigated and discussed in several works. Gotel and Finkelstein (Gotel and Finkelstein 1993) investigated the nature of the requirement traceability problem and identified the reasons leading to a significant lack in the advancement of theory and practice. Among the other identified factors there is the lack of a common definition. This problem was superseded by the IEEE glossary definition adopted in this thesis.

IBIS (Concling and Bergen 1988), TOOR (Pinhero and Goguen 1996), PRO-ART (Pohl et al. 1997) and REMAP (Ramesh and Dhar 1992) are a few examples of software development tools able to build and maintain traceability links among various software artifacts. RECON (Wilde and Casey 1996) is a tool developed based on *Software Reconnaissance*, a dynamic analysis technique proposed in (Scully 1993) and (Wilde and Scully 1995). Ramesh and Jarke (Ramesh and Jarke 2001) followed an empirical approach to define, validate and apply a reference model of traceability based on the previous experiences with the deployment of REMAP. However, these tools are focused on the development phase; furthermore, they require human intervention to define links, or the adoption of naming conventions.

Maarek et al. (Maarek et al. 1991) introduced an IR method for automatically

assembling software libraries based on a free text-indexing scheme. The method uses attributes automatically extracted from IBM RISC System/6000 AIX 3 documentation to build a browsing hierarchy which accepts queries expressed in natural language. GURU (Maarek et al. 1994) is a system that automatically identifies indexes by analyzing the natural-language documentation in the form of manual pages or comments usually associated with the code. This system has been applied to construct an organized library for AIX utilities.

Several software reuse environments use IR to index and retrieve the reusable assets. The RLS system (Burton et al. 1987) extracts free-text single-term indexes from comments in source code files, by looking for keywords such as "author", "date created", etc. REUSE (Arnold and Stepowey 1987) is an IR system which stores software objects as textual documents in a reuse repository. Similarly, CATALOG (Frakes and Nejmeh 1987) stores and retrieves C components, each of which is individually characterized by a set of single-term indexing features automatically extracted from natural language headers of C programs.

Maletic and Marcus (Maletic and Marcus 2001) presented a system called PROCSSI that uses IR techniques to identify semantic similarity between pieces of code. Authors also prove that this semantic similarity measure is helpful in the comprehension task.

Antoniol et al. (Antoniol et al. 1999) presented a method to establish and maintain traceability links between code and free text documents. The method exploits probabilistic IR techniques to estimate a language model for each document or document section, and applies Bayesian classification to score the sequence of mnemonics extracted from a selected area of code against the language models.

The same method was applied in (Antoniol et al. 2000a), to recover traceability links between the functional requirements and the Java source code, extending and validating the previous results on a more complex and difficult case study. The investigation was then extended in (Antoniol et al. 2000b; Antoniol et al. 2002) to vector space model, to compare different model families and assess the relative influ-

ence of affecting factors. Latent semantic approach and IR approach were compared in (Marcus and Maletic 2003).

The case study, equations, approaches and processes proposed in this dissertation led to the publication of scientific contributions in several international journals (Antoniol et al. 2000c; Antoniol et al. 2001a; Antoniol et al. 2002) and conferences (Antoniol et al. 2000b; Antoniol et al. 2000a).

1.5 Chapter Summary

This chapter summarized the basic concepts underlying the techniques and methods developed and presented in this thesis. The traceability recovery approaches presented in the next chapters require some familiarity with key mathematical concepts such as distance, similarity, and probability. Each of the following chapters is devoted to a specific traceability recovery problem and presents methods, techniques, tools, and results obtained by applying the introduced methods and techniques.

The main contributions of this thesis can be summarized as the original definition, development and assessment of technologies and methods and approaches to support traceability links recovery between different level of abstraction: from textual documentation to source code, from design to code, and between different releases of a software system.

For example, by interpreting traceability links recovery as a pattern matching based information retrieval task, the large body of knowledge from those mentioned areas can actually be reused. Moreover, effective means to compare different approaches for traceability recovery via the traditional measures of precision and recall can be immediately derived.

- Chapter 2 presents and validates an approach to recover traceability links between OO design and C++ code. Contributions include the definition of a similarity based on the string edit distance and a graph-based approach to recover an optimal matching.

- Chapter 3 shows how approaches inspired by information retrieval can be adapted to recover traceability links between textual documentation and OO source code. The main contribution is the adoption of an information retrieval approach and the development of a novel approach inspired by stochastic language models.

- Chapter 4 contributes a new method to recover traceability links between high level documentation, such as functional requirements or use cases, and low level artifacts such as detailed design or source code. The advantage of the new method, in contrast to those presented in Chapter 3, is that it does not require that source code and textual documentation share a common vocabulary.

- Chapter 5 presents methods that can be considered as an improvement of the approaches defined in Chapter 2. Besides the traceability recovery approach, the other main contribution of Chapter 5, is the assessment of feasibility, accuracy and reliability of reverse engineering OO design from C++ source code.

- Chapter 6 deals with the evolution of large procedural software. It contributes an approach to analyzing large software systems developed with C programming language as well as four software evolution metrics which capture at a high level of abstraction some key aspects of large software evolution such as the volume of actually changed code between subsequent releases.

Detailed information of the techniques, methods and tools are provided in each chapter; details on the software system and software applications used to assess the accuracy are reported in the appendices.

CHAPTER 2: Design to Code Traceability

2.1 Mapping Model

A design represents an abstraction with respect to its implementation; a design X and its implementation Y may possess different properties. Classes may be added when writing the code, or elements specified as domain or context information may not have counterparts in the code. Thus, an essential step in the definition of a similarity measure between them is the introduction of a map m between a subset of the properties of X and a subset of the properties of Y. X and Y are *substantial individuals* represented by the equation (1.1). Properties belonging to the map are considered as matched properties. The remaining properties from $P(x)$ and $P(y)$ are unmatched properties of X and Y respectively:

$$m: \quad P(x) \rightarrow P(y) \tag{2.1}$$

$$Unmatched(X) = P(x) - Dom(m) \tag{2.2}$$

$$Unmatched(Y) = P(y) - Ran(m) \tag{2.3}$$

where m is an injective function from $P(x)$ to $P(y)$, Dom is the domain and Ran the range. Unmatched properties of X are those not in the domain of m, while unmatched properties of Y are those not in the range of m.

Given a measure of similarity, $\sigma(p, q)$, between the matched properties $p \in P(x)$ and $q \in P(y)$ of two different objects X and Y, an overall similarity measure between the objects can be obtained by applying a suitable average operator as, for example,

the arithmetic average:

$$\overline{\sigma}(X,Y) = 1/|Dom(m)| \sum_{p \in Dom(m)} \sigma(p, m(p)) \tag{2.4}$$

An overall picture of the similarity between X and Y is therefore given by the sets of unmatched properties (Unmatched(X), Unmatched(Y)) and by the average similarity measure between matched properties ($\overline{\sigma}(X,Y)$). More detailed information can be obtained by the individual similarity measures for the matched pairs of properties $(\sigma(p,q), p \in Dom(m), q = m(p))$.

The main entities present in an OO design that must be reflected and implemented in the corresponding code are classes, objects and relations. Experiments were performed on designs represented with the OMT notation. When tracing an OO design into C++ code, among the three models used in OMT to represent an OO software system (class diagram, functional model, dynamic model), the focus was centered on the class diagram. The class diagram is usually the first to be developed, common to many other OO methodologies and notations (e.g. Booch and UML), and is the only one that describes the system specifically using OO concepts.

When defining similarity between objects to trace OO design into code, classes have to be considered as basic entities whose properties are the class attributes both fields and methods. Given a pair of classes for which a similarity measure has to be determined, the similarities of the contained attributes, or properties, have to be computed first. For such a purpose, a natural choice is to consider the names of the attributes, prefixed with class scope, as strings, and to compute the complemented *edit distance* (Cormen et al. 1990) between such strings defined by equation (1.10). In other words in (1.10), x and y are the qualified names of a design or code attribute respectively.

After computing the similarity between each pair of attributes, the similarity at the class level is computed. The match function is inferred by applying the maximum match algorithm (Cormen et al. 1990) to the bipartite graph (see Figure 2.1), in

which nodes are respectively attributes from design and code and are connected by edges that are weighted with the similarity measures.

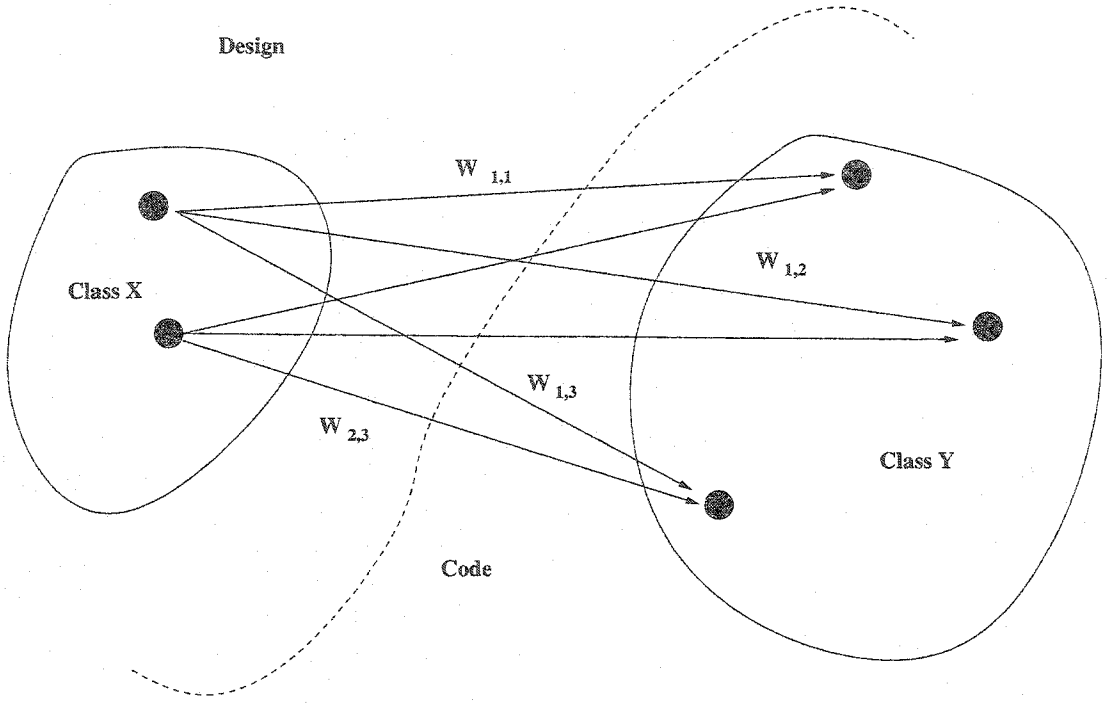

Figure 2.1: Bipartite graph.

The edges selected by the algorithm are those giving the overall maximum match and define the desired mapping. A further outcome of this algorithm is the set of unmatched attributes in the design and code. Once the detailed mapping is available, the average similarity measure is computed for the two classes according to the equation (2.4).

By repeating the above procedure for each pair of classes, it is possible to determine their respective average similarity measure. To determine the correspondence between design and code, it is possible to exploit the maximum match algorithm again. In this case, the nodes in the bipartite graph are respectively classes from the design and code, while edges are weighted with the average similarity measures. The edges extracted by the algorithm represent the traceability links between design and code. Each link is weighted with a similarity measure. In addition, an initial set of unmatched classes

is determined as those having no traceability link attached, as shown in Figure 2.1.

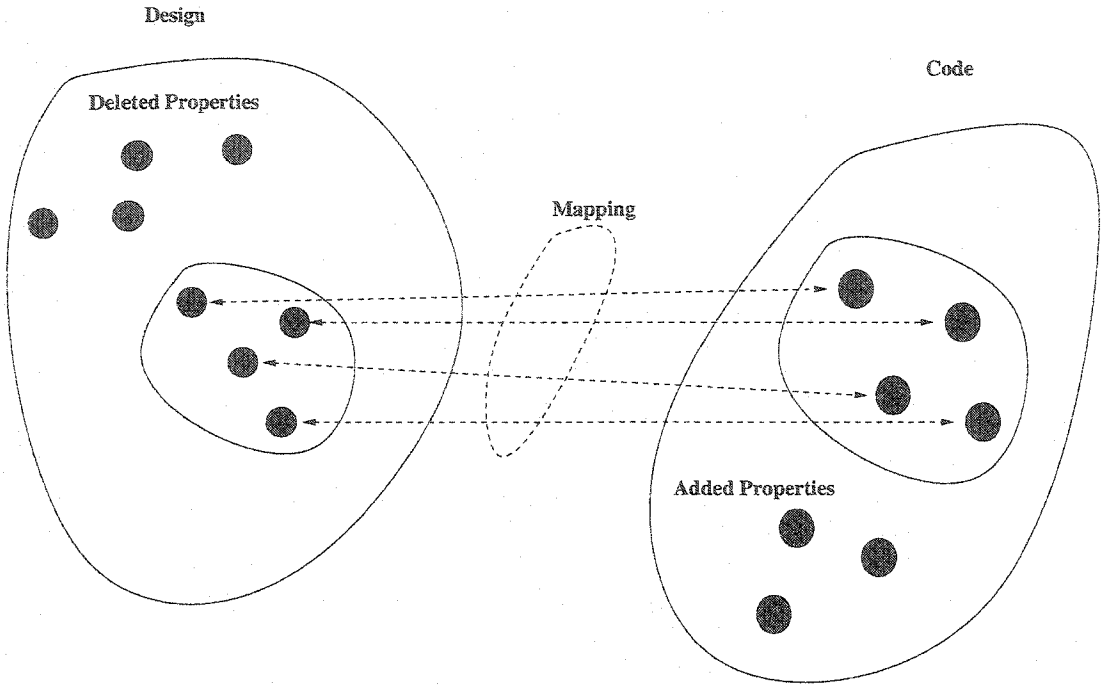

Figure 2.2: Matched and unmatched entities.

2.1.1 Classification of Matched and Unmatched Class Pairs

The presence of a traceability link between a class in the design and a class in the code is not sufficient to state that a match occurred. In fact, the similarity measure associated with the link may be very low, and the edge in the bipartite graph could have been selected by the maximum match algorithm only as an effect of maximizing the total match measure.

Therefore, the links connecting matched classes from design and code have to be classified to further distinguish the truly matched classes from the unmatched. For this purpose, a maximum likelihood classifier (Duda and Hart 1973) has been used, giving a threshold which separates low similarity class pairs, to be considered unmatched, from high similarity matched class pairs.

When a similarity threshold is adopted, two kinds of errors may occur. The first

is the classification of truly matched classes as unmatched. The parameter which accounts for this error is *recall*, computed as the ratio of correctly classified class pairs over the total number of truly matched class pairs. If recall is 1, no matched class is missed by the classifier. The second error is the classification of an unmatched class pair as matched. The parameter accounting for it is *precision*, computed as the ratio of correctly classified class pairs over the total number of classes classified as matched. If precision is 1, no unmatched class is classified as matched, see Section 1.3.4.

The maximum likelihood classifier is the one for which the sum of the two errors is minimal. While each error can be individually minimized (recall tends to 1 as the threshold is arbitrarily decreased, while precision tends to 1 when the threshold increases), the maximum likelihood classifier gives the threshold for which the likelihood of *both* errors is minimal.

The computation of the maximum likelihood threshold requires that a set of class pairs be correctly labeled as matched or unmatched. For each of the two categories, the shape of the probability density has to be estimated from the frequency and the intersection of the two curves gives the threshold. Probability densities can be estimated by assuming a Gaussian distribution and determining mean and variance, or more generally by using raw frequency data or smoothed data as a substitute for the true densities.

The accuracy of the classifier is then evaluated on a test set different from the one used to compute the threshold. If few examples are available, the evaluation can be conducted with a *cross validation* technique. Each component in turn is considered as a test case, while the remaining components are used to determine the threshold. A repetition of the test procedure is thus possible by changing the test component. Average performance and robustness can then be assessed on a wider base than a single test case.

2.1.2 Relations Traceability

In the sections above, traceability has been considered at the level of the basic design entities: classes and attributes. The concept can be further expanded by requiring that traceability holds also at the level of the *relationships* among classes. In particular, if we consider a class in the design and its corresponding class in the code, the relations of generalization, association and aggregation present in the design should be reflected in the code. Since the code is the implementation of the design, a relation appearing in the code with no counterpart in the design can be considered a detail that should be ignored when abstracted to the design level. On the contrary, all design relations are expected to be found in the code.

The traceability check procedure for the relations uses the following strategy. For each relation in the design, if two connected classes are matched by two corresponding classes in the code, a relation of the same type is sought between the two matched classes in the code. Relations between class pairs unmatched in the code are not considered.

2.1.3 Context Information

Context information is often included in the class diagram, while extra information can be in the code due to automatic generation. When tracing design into code, such information is classified as unmatched. On the design side, the typical context information is:

- Classes from other user components which are necessary to better understand the context of the current component.
- Classes from libraries which are included to represent sub-classing of, or associations with, current component classes.
- Environment components, external systems or users.

Classes in the code but not in the design typically have the following origins:

- COTS may introduce classes in source code which are not modeled in design.
- Classes automatically generated by code generators.
- Middleware such CORBA introduces new classes as a result of the process of stub and code generation.
- Components implementing Graphical User Interface (GUI) are likely to present many of the previous features. They are often automatically generated through GUI builders and usually GUIs make heavy use of libraries.
- Test code, drivers or stubs used to test the component in isolation.

The maximum likelihood classifier allows for the identification of matched and unmatched classes. Unmatched classes in the design can be retained for clarity or marked as context information.

2.1.4 Difference Visualization

To highlight similarities and differences, *pair-difference coloring*, a technique which employs different colors to contrast pairs of versions of the same information, was adopted. This technique, in which similar and different parts of the two versions are assigned different colors to represent changes in source code has been used in the GASE environment (Holt and Pak 1996). Colors have also been used in software visualization, to associate time information differentiating recent from older changes on source code (Ball and Eick 1996).

Evolution of the design of a software component can be supported by adopting such coloring techniques. To highlight similarity information, the background color of the classes in the design was modified so that those unmatched by the code are red, while those with a perfect match (similarity equal to 1) are green. Intermediate colors from green to yellow can be used for intermediate similarity levels. The programmer charged to update the design to improve its traceability with the code can prioritize the interventions. Classes with colors close to the green require few modifications in

the names of the attributes, while yellow classes may require deeper investigations to understand why the best matched attributes in the code are so different. Red classes are not matched at all, thus the question is if they are context information to be preserved or if they have to be removed from the design.

In addition, the programmer can go into deeper details and analyze the match between attributes (fields and methods) for a class of interest in the design. The traceability of the relations is also given, as the list of inter-class relations that are specified in the design but are not implemented in the code. This information is provided in textual form.

A prototype of the *difference* visualization technique was implemented on top of Object-Oriented Designer (OOD) [1], a free software design tool. The output of the tool to the printer is filtered by a program which uses the match levels determined by traceability analysis to add colors to the classes in the design.

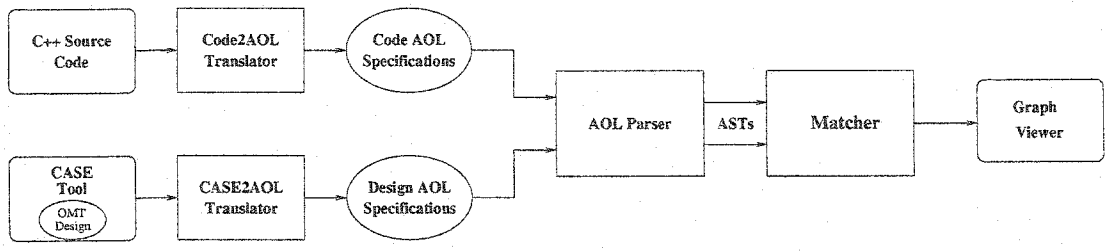

Figure 2.3: Design to code matching process.

2.2 Experimental Results

The whole design to code traceability check process is represented in Figure 2.3. It consists of the following steps:

1. AOL Representation Extraction: the AOL textual representation can be recovered from both design and code through respectively a `Code2AOL` and a `CASE2AOL` translator.

[1] OOD was developed by Taegyun Kim, at Pusan University of Foreign Studies, Pusan, Korea.

2. AOL Representation Parsing: an `AOL Parser` produces the Abstract Syntax Tree (AST) which subsequent phases rely on.

3. Match between design and code representations: a `Matcher` module implements the traceability check; it includes a function to compute the edit distance between attribute names, an implementation of the maximum matching algorithm and a maximum likelihood classifier.

4. Result Visualization: a `Pair Difference Coloring` module graphically shows the results of matching, highlighting similarities and differences between classes in the design and in the code.

The programmer is then in charge of the final step, in which the design is modified to solve all outlined differences from code. This phase cannot be completely automated since the reasons for the major differences have to be fully understood by the designer, in order to perform a meaningful update of the design.

Additional information can be obtained by the designer, by querying the traceability links and the associated measures for the attributes of individual classes, and by evaluating the traceability of the relations between classes.

2.2.1 Test Suite

The experiments were performed on 29 software components described in Appendix G. The similarity defined by equations (1.10, 2.4) was used in a first experiment without any further constraint to recover a traceability mapping. A second set of experiments (applying the same similarity) aimed at devising an optimal classification threshold, based on a maximum likelihood classifier, to eliminate wrong matches. As one would expect, the number of classes and attributes extracted from the code is often substantially higher than in the design, thus indicating that the abstraction

represented in the design typically ignores implementation details, like support classes and attributes which are introduced in the coding phase (Lorenz and Kidd 1994). In a few cases the opposite is true (e.g., C2, C4, C7), and the higher number of classes in the design can be explained by the presence of context information inserted to help the programmer in understanding the whole operative setting of the classes under development.

2.2.2 Average Match Figures

Comp.	Del.	Avg sim.	Comp.	Del.	Avg sim.
C1	0	0.829	C16	9	0.909
C2	1	0.944	C17	0	0.856
C3	0	0.992	C18	0	0.947
C4	10	0.636	C19	0	0.983
C5	0	0.986	C20	1	0.905
C6	4	0.963	C21	0	0.988
C7	8	0.748	C22	5	0.815
C8	0	0.698	C23	8	0.871
C9	0	0.884	C24	0	0.808
C10	0	0.968	C25	0	0.996
C11	0	1	C26	3	0.877
C12	1	0.907	C27	6	0.948
C13	0	0.983	C28	1	0.874
C14	1	0.628	C29	2	0.994
C15	5	0.890			

Table 2.1: Deleted classes and average similarity measures for design components, as resulting from the maximum match algorithm.

Table 2.1 shows the results of the code traceability check, before applying the maximum likelihood classifier. The number of unmatched (deleted) classes in this table only accounts for the classes in the design for which the maximum match algorithm does not produce a traceability link. The application of the maximum likelihood classifier may increase such number and it will be discussed in the following. The average similarity measure, i.e., the arithmetic average of the similarities in the recovered traceability links, is given for each component. This measure is also expected

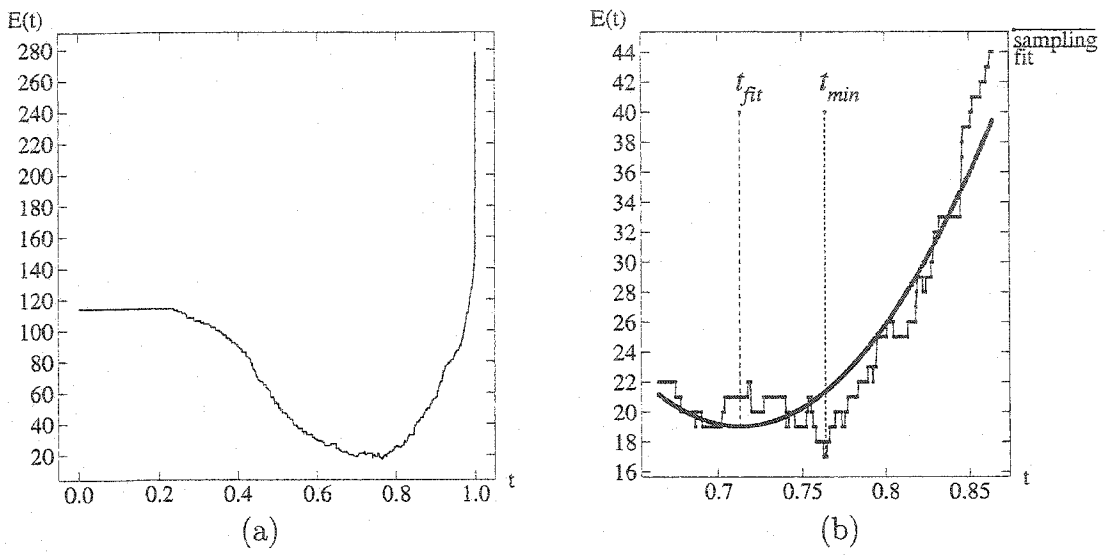

Figure 2.4: Finding the classification threshold. *(a):* the misclassification error E(t), with component *C1* excluded, is plotted as a function of the classification threshold, t. *(b):* quadratic fitting of E(t) in the neighborhood of t_{min}.

to increase after the application of the maximum likelihood classifier.

Average similarity is higher than 0.8, with the exception of components C4, C7, C8 and C14, while the maximum number of classes deleted from the design is 10. The low similarity (0.628) of component C14 was further investigated. The design of this component contains 3 classes. One of them is context information that is deleted. The remaining two classes have respectively a perfect match (similarity 1) and a very poor match (similarity 0.257) with code. A close inspection into the latter class reveals that the low similarity level is associated to a class to be considered actually unmatched, i.e., introduced to provide context information but having no counterpart in the code.

The maximum likelihood classifier was then applied to obtain a better identification of matched and unmatched classes. To evaluate its performance, the cross validation technique was used. One component is taken out from the database, and a value, t_{min}, for the classification threshold (t) is estimated on the remaining 28 components. The process is then iterated over all the components of the database, and at each iteration the accuracy of the classifier is assessed by counting the misclassification

	Raw Data			Polynomial Fit		
Component	t_{min}	Precision	Recall	t_{fit}	Precision	Recall
C1	0.764	1.0	0.926	0.738	0.954	0.971
C2	0.764	1.0	1.0	0.715	0.95	0.976
C3	0.764	1.0	1.0	0.706	0.95	0.976
C4	0.764	0.912	0.756	0.74	0.97	0.992
C5	0.765	1.0	1.0	0.717	0.951	0.977
C6	0.763	1.0	1.0	0.714	0.951	0.977
C7	0.697	0.857	0.923	0.717	0.957	0.979
C8	0.695	0.588	1.0	0.701	0.973	0.983
C9	0.765	1.0	1.0	0.714	0.947	0.975
C10	0.763	1.0	1.0	0.694	0.949	0.982
C11	0.687	1.0	1.0	0.723	0.955	0.972
C12	0.686	1.0	1.0	0.713	0.951	0.976
C13	0.764	1.0	1.0	0.714	0.952	0.977
C14	0.764	1.0	1.0	0.728	0.958	0.97
C15	0.764	1.0	1.0	0.707	0.951	0.977
C16	0.764	1.0	1.0	0.721	0.957	0.973
C17	0.695	1.0	1.0	0.705	0.951	0.977
C18	0.747	0.917	1.0	0.717	0.953	0.976
C19	0.764	1.0	1.0	0.688	0.948	0.983
C20	0.763	1.0	1.0	0.71	0.951	0.977
C21	0.694	1.0	1.0	0.726	0.958	0.974
C22	0.764	1.0	1.0	0.698	0.952	0.983
C23	0.764	1.0	1.0	0.696	0.952	0.983
C24	0.763	1.0	1.0	0.7	0.951	0.983
C25	0.694	1.0	1.0	0.711	0.952	0.977
C26	0.763	1.0	1.0	0.724	0.957	0.973
C27	0.764	1.0	1.0	0.732	0.956	0.969
C28	0.763	1.0	1.0	0.733	0.957	0.97
C29	0.764	1.0	1.0	0.709	0.951	0.977

Table 2.2: Columns 2-4 report results on raw data; columns 5-7 those obtained by polynomial fit.

errors made on the excluded component.

As shown by the plot in Figure 2.4.a, the large-scale behavior of the error $E(t)$ is quite smooth; however, its "true" point of minimum (that is, abstracted from the sampling noise) can be only approximately localized. This may suggest that a more robust estimate of the classification threshold could be obtained by locally approximating $E(t)$ with a (low-degree) polynomial, and by setting the threshold to a value, t_{fit} which corresponds to the point of minimum of such polynomial. In Figure 2.4.b, a second degree fit of $E(t)$ is shown.

Results collected in Table 2.2 show how the estimate of the classification threshold via polynomial fit does not improve the overall accuracy of the classifier; yet, it typically gives a more balanced occurrence of false positives and false negatives (see, for example, component C8), resulting in a higher robustness.

Comp.	Del.	Avg sim.	Comp.	Del.	Avg sim.
C1	11	0.956	C16	10	0.982
C2	2	0.984	C17	1	0.966
C3	0	0.992	C18	0	0.947
C4	74	0.839	C19	0	0.983
C5	0	0.986	C20	2	0.996
C6	4	0.963	C21	0	0.988
C7	15	0.856	C22	6	0.990
C8	18	0.909	C23	9	0.988
C9	1	0.900	C24	5	0.984
C10	1	0.995	C25	0	0.996
C11	0	1	C26	4	0.999
C12	3	0.996	C27	7	0.999
C13	0	0.983	C28	2	0.999
C14	2	1	C29	2	0.994
C15	6	1			

Table 2.3: Deleted classes and average similarity measures for design components, as resulting from the maximum likelihood classifier.

The average threshold computed by the maximum likelihood classifier using raw data (Table 2.2, second column) is 0.74. This value is extremely stable with respect to the choice of the components used for its computation (standard deviation 0.03) and its value is not critical in the error minimization process (see Figure 2.4). Therefore it can be used as a reference value for new components designed and developed by the same company.

If 0.74 is used to classify the matched class pairs resulting from the application of the maximum match algorithm (see Table 2.1), the number of classes deleted from the design and the average similarity measure are those shown in Table 2.3.

The average similarity is substantially higher, has a minimum of 0.839, and is above 0.9 with the exception of two components (C4 and C7). A perfect match is achieved for 3 components (C11, C14, C15), of which only one was already present in Table 2.1. This result was obtained by considering class pairs below the threshold as unmatched. Consequently, the number of classes deleted from the design increases, but in several cases a unitary increase suffices to improve the design-code match (e.g.,

C2, C9, C10, C14).

A final revision of the traceability links is required from the designer in any case, since the maximum likelihood classifier may in some infrequent cases be wrong, as shown by the values of precision and recall in Table 2.2.

2.2.3 Design-code Match of the Relations

Starting from the matching between the design and code classes obtained with the maximum match algorithm, the subset of the truly matched classes was considered. For each pair of matched classes, the relations of generalization, association and aggregation in the design have been sought in the code. Table 2.4 shows for each component and type of relation, the number of matching relations found in the corresponding code classes over the total number of relations between matched classes in the design.

Comp.	Gen.	Ass.	Agg.	Comp.	Gen.	Ass.	Agg.
C1	1/1	1/4	3/3	C16	0/0	0/7	0/0
C2	9/9	0/7	1/1	C17	2/2	0/2	0/1
C3	8/8	0/3	1/1	C18	9/9	0/5	0/0
C4	7/11	1/13	0/11	C19	5/5	0/5	0/0
C5	5/5	1/1	0/0	C20	5/5	0/1	0/0
C6	0/0	3/4	0/0	C21	0/0	0/0	0/0
C7	0/0	0/12	5/5	C22	0/0	0/1	0/1
C8	0/0	0/7	1/4	C23	0/0	0/2	0/3
C9	20/20	4/5	1/2	C24	2/2	2/2	1/7
C10	14/14	13/24	0/0	C25	1/1	0/1	0/0
C11	12/12	0/1	2/5	C26	2/2	0/2	0/0
C12	5/5	0/1	0/2	C27	9/9	0/1	1/1
C13	0/0	0/0	0/0	C28	2/2	0/2	0/0
C14	0/0	0/0	0/0	C29	0/0	0/1	2/3
C15	3/3	0/0	0/3				

Table 2.4: Number of generalization, association, and aggregation relations of design classes found among corresponding code classes over number of relations present in the design.

The traceability of the generalization relation is nearly total. The C4 component

is the only one with four design generalizations that do not appear in the code. Less traceable are the association and aggregation relations (although the latter are better). The poor traceability of associations and aggregations has a twofold explanation. On one hand, the meaning of such relations as intended by the designer is loosely coupled with its implementation, since it is mainly associated with a vague concept of collaboration between design entities. On the other hand, some relations are missing in the code due to the limitations of the reverse engineering tools, which are not always able to recognize them in complex constructs and data structures, and which do not therefore retrieve some of the actually implemented relations. Generalization is the simplest relation to reverse engineer. Those missing in the code correspond to an incorrect representation of the underlying class hierarchy in the design. Association is the most difficult relation to recover, since it may be implemented by using *key-identifiers*, typically integers or strings which are not directly related to the type of the associated objects. Aggregations are usually simpler to extract than associations because they are implemented through more standard programming constructs such as arrays.

Component C4 is responsible for the graphical user interface and, consequently, it contains automatically generated code, library classes and user code. Its traceability level is the lowest (0.839), with the maximum number of deleted classes (74), even after applying the maximum likelihood classifier. The traceability of its relations is again the lowest, with four generalizations and almost all associations and aggregations missing. The difficulties in separating relevant information from automatically generated code and library classes suggest that results from this component should be carefully interpreted. The true traceability level of this component would be expected to increase if the designers and developers of this component were involved in identifying and comparing the relevant portion of the design and the actually implemented code.

In summary, 121/125 (96.8%) generalizations, 25/114 (21.9%) associations and 18/53 (33.9%) aggregations could be automatically traced from the design to the

code. If component C4 is excluded, 114 generalizations (100%), 24 associations out of 101 (23.7%), and 18 aggregations out of 42 (42.8%) could be found in the code.

2.2.4 Detailed Analysis of an Example Component

Average match figures discussed in the previous sections do not account for the more precise information from which they have been computed. For example, for each pair of matched classes, the detailed map of attributes and methods is constructed. Programmers can use such information to update designs and make them consistent with the implementation. On the basis of the similarity values of the matched classes and attributes, they can decide to update names and to add or remove classes and attributes.

To gain a deeper insight into the process of evolving a design to improve its code traceability, component C16 was considered in more detail. Its average traceability index is 0.982 with 10 unmatched classes, resulting from the maximum likelihood classifier. The classifier increased the number of deleted classes from 9 to 10 due to a very low match (0.397) between two classes. Manual inspection confirmed that such additional deletion is correct. Figure 2.5 shows the graphical result of code traceability analysis for C16. Dark grey boxes(plotted as green in the visualization tool), used to represent a perfect match of classes in the design and in the code, are shown at the top. Light grey (i.e., light-green to yellow) represents the intermediate similarity levels and accounts for the four classes in the middle (similarity levels 0.957, 0.985, 0.991 and 0.940). Dark grey (i.e., red in the tool) is used for the boxes at the bottom associated with some of the unmatched classes. One of them is an outcome of the maximum likelihood classification (similarity 0.397).

It can be noted that every non red class has a prefix `Wct_` in its name. Therefore, the hypothesis can be made that the red classes at the bottom are actually context information.

The match between design and code for the four light-green to yellow classes was then considered in more detail. Class `Wct_DistItem` has an average similarity measure

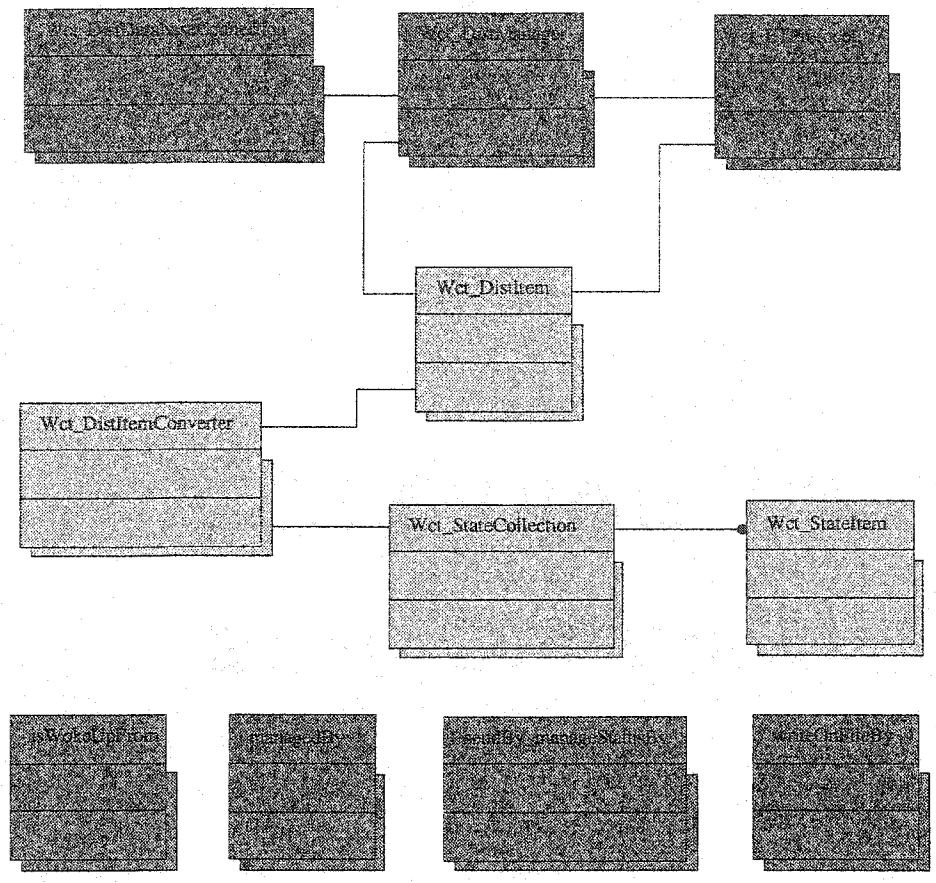

Figure 2.5: Pair difference diagram for component C16. Green classes (dark grey) are the three at the top, green to yellow ones are in the middle (light grey), while some of the red classes (dark grey) are shown at the bottom. Attributes have been omitted for clarity.

equal to 0.957 with a same name class in the code. The matched attributes have been examined as shown in Table 2.5. In the last column the individual similarity measures are given for them.

For the first six attributes, a perfect match with code could be retrieved. The last but one attribute, `setItemStatusToSENT`, is matched with `setItemStatus` (similarity 0.900). The difference between the two names is in the suffix `ToSENT` specified in the design and omitted in the code. The operation implemented in the code is probably more general than the one in the design and an additional parameter can be used to obtain the specialization represented in the design. It is likely that programmers took the opportunity to generalize with only minor overhead. The relatively low similarity

Design attribute	Code attribute	Sim.
Wct_DistItem	Wct_DistItem	1
~Wct_DistItem	~Wct_DistItem	1
deleteItemFromDirectory	deleteItemFromDirectory	1
extraction	extraction	1
init	init	1
loadItemStatus	loadItemStatus	1
setItemStatusToSENT	setItemStatus	0.900
deleteItemFromDatabase	loadItemNumberFromDatabase	0.763

Table 2.5: Attribute match for class Wct_DistItem from component C16. For each pair of attributes the associated similarity measure is shown.

level (0.763) of the last attribute suggests a possible deletion of an operation when moving to coding. An examination of the attribute names seems to confirm such a hypothesis. To obtain a perfect match between design and code only minor changes of Wct_DistItem design are required. Attribute setItemStatusToSENT is replaced with setItemStatus and attribute deleteItemFromDatabase deleted. Similar problems were observed for the other three classes in the middle of Figure 2.5. In one case (class Wct_StateCollection) one of the non perfect matches between attributes was due to a typing error.

2.3 Chapter Summary

The chapter presented and validated an approach to recover traceability links between OO design and C++ code. Main contributions are the definition of a similarity between OO classes and an approach which applies the bipartite graph matching algorithm to recover an optimal mapping between design and code.

OO similarity is defined at the class interface level and abstracts details from the implementation of classes. An implementation which significantly deviates from the design will obtain a low similarity score and will be undetected. To overcome this limitation, the adoption of a classifier which pinpoints false matching is essential. Reported experimental results show that a Bayesian classifier with a similarity

threshold of about 75 % achieves minimization of the misclassification error. Clearly, other software systems may or may not attain the optimal misclassification error with the above threshold, so a threshold calibration will generally be required.

Bipartite graph matching is essentially linear in the graph size (i.e., number of nodes and edges), thus the main limitation is in the quadratic complexity of computing a similarity measure for each pair of classes, the first of which belongs to the design and the second of which to the source code. Foreseeable improvements are in the area of approximate matching to limit the number of required comparisons or the definition of heuristics to reduce the time complexity for very large software systems. Finally, the similarity computation can be easily parallelized; thus , if necessary, the similarity evaluation can be split over a number of machines.

x

CHAPTER 3: Manuals and Requirements to Code Traceability

3.1 Mapping Model

To recover or assess a traceability map between informal textual documentation such as that contained in user manuals or requirement documents and the source code, techniques more suitable are those inspired by Information Retrieval (IR) approaches. Indexing the documents, the queries, and ranking documents against queries certainly depend on the particular IR model adopted. In the following subsections, two IR models are described: the probabilistic model and the vector space model. In the probabilistic model, free text documents are ranked according to the probability of relevance to a source code component. The vector space model treats documents and queries as vectors; documents are ranked against queries by computing some similarity functions between the corresponding vectors.

3.1.1 Probabilistic IR Based on Language Models

This model computes the ranking scores as the probability that a document D_i is related to the query Q, which represents the source code component S_Q:

$$\sigma(D_i, Q) \;=\; Pr(D_i|Q)$$

Notice that the stochastic language model approach is completely different from the traditional probabilistic IR method, as described, for example, in (Baeza-Yates and Ribeiro-Neto 1999); in the following, whenever there is no possibility of confusion,

it will be referred to as a probabilistic model instead of a probabilistic language model whereas the traditional approach will be referred to as a *traditional* probabilistic IR model.

Applying Baye's rule (Bain and Engelhardt 1992), the conditioned probability above can be transformed into:

$$Pr(D_i|Q) = \frac{Pr(Q|D_i)Pr(D_i)}{Pr(Q)} \tag{3.1}$$

For a given source code component, $Pr(Q)$ is a constant and the model can further simplified by assuming that all system documents have the same probability. Therefore, for a given source code component Q, all documents D_i are ranked by the conditioned probabilities $Pr(Q|D_i)$.

For each document D_i, these conditioned probabilities are computed by estimating a stochastic language model (De Mori 1998). Indeed, under the hypothesis that the source code components and the documents share a common vocabulary V, the source code component S_Q can be mapped into a query Q, which is composed of the sequence of m words $w_1, w_2, \ldots, w_m$ (the identifiers of S_Q) belonging to vocabulary V. The conditional probability:

$$Pr(Q|D_i) = Pr(w_1, w_2, \ldots, w_m \mid D_i) \tag{3.2}$$

can be estimated on a statistical basis by exploiting a stochastic language model for document D_i. The estimation process relies on statistics about the frequency of the occurrences of sequences of words of V in D_i. However, the above probability can be written as:

$$Pr(w_1, w_2, \ldots, w_m \mid D_i) = Pr(w_1 \mid D_i) \prod_{k=2}^{m} Pr(w_k \mid w_1, \ldots, w_{k-1}, D_i) \tag{3.3}$$

When m increases, the probabilities involved in equation (3.2) quickly become difficult to estimate due to the exponential increase of the possible sequences of words

which can be constructed over the vocabulary. A simplification can be introduced by conditioning the dependence of each word to the last $n-1$ words (with $n < m$), as follows:

$$\begin{aligned} Pr(w_1, w_2, \ldots, w_m \mid D_i) \quad &\simeq \quad Pr(w_1, \ldots, w_{n-1} \mid D_i) * \\ &\prod_{k=n}^{m} Pr(w_k \mid w_{k-n+1}, \ldots, w_{k-1}, D_i) \end{aligned} \tag{3.4}$$

This n-gram approximation, which formally assumes a time-invariant Markov process (Cover and Thomas 1992), greatly reduces the volume of statistics to be collected in order to compute $Pr(Q|D_i)$; clearly this also introduces an imprecision. However, n-gram models are still difficult to estimate, because if $|V|$ is the size of the vocabulary, all possible $|V|^n$ sequences of words in the vocabulary have to be considered; indeed, the estimation can be very demanding even for a 2-gram (bigram) model. In a bigram model, $Pr(w_1, w_2, \ldots, w_m \mid D_i) \simeq Pr(w_1 \mid D_i) \prod_{k=2}^{m} Pr(w_k \mid w_{k-1} D_i)$. Since the occurrence of any sequence of words in a document D_i is a rare event most of the sequences will never occur due to the sparseness of data. To alleviate the problem, a reasonable choice is a unigram approximation ($n = 1$) that corresponds to considering all words w_k to be independent. Therefore, each document D_i is represented by a language model in which unigram probabilities are estimated for all words in the vocabulary as:

$$\sigma(D_i, Q) = Pr(Q|D_i) = Pr(w_1, w_2, \ldots, w_m \mid D_i) \simeq \prod_{k=1}^{m} Pr(w_k \mid D_i) \tag{3.5}$$

Unigram estimation is based on the term frequency of each word in a document. However, using the simple term frequency would turn the product $\prod_{k=1}^{m} Pr(w_k \mid D_i)$ to zero whenever any word w_k is not present in the document D_i. This problem, known as the zero-frequency problem (Witten and Bell 1991), can be avoided by using different approaches (see (De Mori 1998)). As in speech recognition, the adopted approach consists of smoothing the unigram probability distribution by computing new probabilities as follows:

$$Pr(w_k \mid D_i) = \begin{cases} \frac{c_k - \beta}{N} + \lambda & \text{if } w_k \text{ occurs in } D_i \\ \\ \lambda & \text{otherwise} \end{cases} \tag{3.6}$$

where N is the total number of words in the document D_i and c_k is the number of occurrences of words w_k in the document D_i. The interpolation term λ is:

$$\lambda = \frac{n}{N * |V|}\beta \tag{3.7}$$

where n is the number of different words of the vocabulary V occurring in the document D_i. The value of the parameter β is computed according to Ney and Essen (Ney and Essen 1991) as follows:

$$\beta = \frac{n(1)}{n(1) + 2 * n(2)} \tag{3.8}$$

where $n(j)$ is the number of words occurring j times in the document D_i.

3.1.2 Traditional Probabilistic IR Model

In the traditional probabilistic IR approaches (Baeza-Yates and Ribeiro-Neto 1999), a query Q (i.e., its index terms) is scored with respect to a set R of documents known, or guessed, to be relevant. If $\bar{R}$ is the set of non relevant documents, similarity between the document D_i and the query Q is defined as:

$$\sigma(D_i, Q) = \frac{Pr(R|D_i)}{Pr(\bar{R}|D_i} \tag{3.9}$$

Given a set of binary weights $w_{i,j}$ and $w_{q,j}$, the above equation can be expanded into:

$$\sigma(D_i, Q) \simeq \sum_{j=1}^{N} w_{i,j} w_{q,j} log\left(\frac{Pr(k_j|R)}{1 - Pr(k_j|R)}\right) + log\left(\frac{1 - Pr(k_j|\bar{R})}{Pr(k_j|\bar{R})}\right) \tag{3.10}$$

where $Pr(k_j|R)$ $(Pr(k_j|\bar{R}))$ stands for the probability that the term k_j is present in a document randomly selected from R $(\bar{R})$ and $w_{r,s}$ is defined as follows:

$$w_{r,s} = \begin{cases} 1 & \text{if } k_s \in D_r \\ \\ 0 & \text{otherwise} \end{cases} \tag{3.11}$$

The main drawback of the model is that the set R (and thus $\bar{R}$) are unknown, at least in the initial phase. Usually, the following simplifying conditions are assumed:

$$Pr(k_j|R) = 0.5 \tag{3.12}$$

$$Pr(k_j|\bar{R}) = \frac{n_j}{N} \tag{3.13}$$

At the beginning, $Pr(k_j|R)$ is supposed to be uniform and constant, while $Pr(k_j|\bar{R})$ is approximated by the ratio between the size of the document set and the number of documents containing k_j.

Let V be the subset of documents retrieved based on equations (3.10, 3.12, 3.13) and ranked above a given threshold r, the probability $Pr(k_j|R)$, $Pr(k_j|\bar{R})$ can be refined by taking:

$$Pr(k_j|R) = \frac{|V_j| + 0.5}{|V| + 1} \tag{3.14}$$

$$Pr(k_j|R) = \frac{n_j - |V_j| + 0.5}{N - |V| + 1} \tag{3.15}$$

where $|V|$ is the size of the V set, and V_j is the subset of V containing k_j. The set V of retrieved and ranked document can be thus recomputed. As pointed out in (Baeza-Yates and Ribeiro-Neto 1999), main drawbacks of the traditional probabilistic model are that it does not take into account the frequency of terms ($w_{i,j}$ and $w_{q,j}$ are binary terms), and it further requires the initial guess of R. The results obtained by applying the traditional probabilistic approach in a preliminary assessment of the models were not as satisfactory as those obtained by applying the vector space or the language model approaches. As an example, Table 3.1 shows the accuracies obtained on the Albergate system by the language model and the traditional probabilistic IR model.

		Language model			Traditional Prob. model		
Cut	Retrieved	Relevant	Precision	Recall	Relevant	Precision	Recall
1	60	29	48.33 %	50.00 %	17	28.33 %	29.31 %
2	120	41	34.16 %	70.68 %	30	25.00 %	51.72 %
3	180	45	25.00 %	77.58 %	35	19.44 %	60.34 %
4	240	51	21.25 %	87.93 %	36	15.00 %	62.06 %
5	300	57	19.00 %	98.27 %	41	13.67 %	70.69 %
6	360	58	13.80 %	100.00 %	46	12.77 %	79.31 %
7	420	58	13.80 %	100.00 %	51	12.14 %	87.93 %

Table 3.1: The traditional probabilistic approach compared to the language model approach; Albergate results were obtained with the improved process.

The accuracy obtained while using as initial probability guesses the values returned by equations (3.10,3.12) is lower for the traditional IR approach. Recall is 100 % only when 16 documents are retained i.e., exactly the number of requirements. Clearly, re-estimating the probability according to equations (3.14, 3.15) once the initial set is retrieved increases the precision for a fixed number of retrieved documents (e.g., the top three) at the expense of further lowering the recall. The traditional probabilistic IR model will not be further investigated, given the difficulties in producing a reliable initial set R, the relatively low performance, and the objective to

recover all traceability links

3.1.3 Vector Space IR Model

Vector space IR models map each document and each query onto a vector (Harman 1992). In our case, each element of the vector corresponds to a word or term in a vocabulary extracted from the documents themselves. If $|V|$ is the size of the vocabulary, then the vector $[d_{i,1}, d_{i,2}, \ldots d_{i,|V|}]$ represents the document D_i. The j-th element $d_{i,j}$ is a measure of the weight of the j-th term of the vocabulary in the document D_i. Different measures have been proposed for such a weight. In the simplest case it is a boolean value, which is 1 if the j-th term occurs in the document D_i (0 otherwise); in other cases, more complex measures are constructed based on the frequency of the terms in the documents.

To quantify the similarity of documents, a well known IR metric called *tf-idf* (Salton and Buckley 1988) was used. According to this metric, the j-th element $d_{i,j}$ is derived from the *term frequency* $tf_{i,j}$ of the j-th term in the document D_i and from the *inverse document frequency* idf_j of the same term over the entire set of documents. The term frequency $tf_{i,j}$ is the ratio between the number of occurrences of the j-th word over the total number of words contained in the document D_i. The inverse document frequency idf_j is defined as:

$$idf_j = \frac{Total\ Number\ of\ Documents}{Number\ of\ Documents\ containing\ the\ j^{th}\ term}$$

The vector element $d_{i,j}$ is:

$$d_{i,j} = tf_{i,j} * log(idf_j)$$

The term $log(idf_j)$ acts as a weight for the frequency of a word in a document: the more the word is specific to the document, the higher the weight.

The list of identifiers extracted from a source code component S_Q is represented in a similar way by a vector $[q_1, q_2, ...q_{|V|}]$. The similarity between a document D_i and a query Q is computed as the cosine of the angle between the corresponding vectors according to the equation (1.8).

3.2 Experimental Results

Figure 3.1 shows the approach for traceability link recovery using IR models. The figure highlights two paths of activities: one to prepare the document for retrieval (document path) and the other to extract the queries from code (code path).

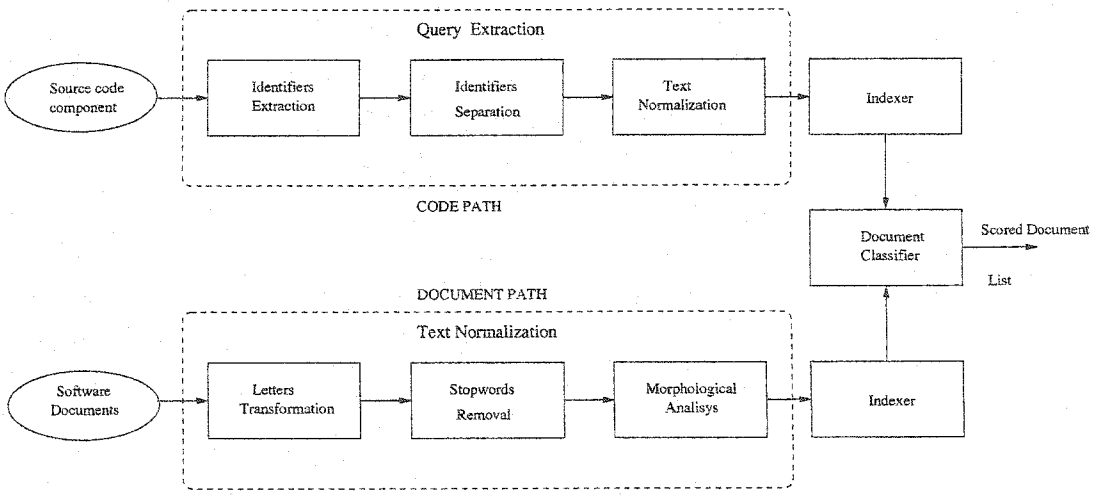

Figure 3.1: Traceability link recovery process.

In the document path, documents are indexed based on a vocabulary that is extracted from the documents themselves. The construction of the vocabulary and the indexing of the documents are preceded by a text normalization phase performed in the steps described in Section 1.3.3.

Finally, a classifier computes the similarity between queries and documents and returns a ranked list of documents for each source code component. Documents are ranked against a source code component by decreasing similarity.

3.2.1 Test Suite

Two case studies were carried out to assess the feasibility of recovering traceability links via vector space and stochastic language models and to evaluate the process shown in Figure 3.1.

3.2.2 LEDA Case Study

The first case study is based on a C++ library of foundation classes, called LEDA release 3.4, which is described in Appendix D.

The documentation of release 3.4 consists of 248 pages. Often, more than one page describes the same structure (e.g., the graph class takes about 10 pages); thus the manual can be understood as organized in 88 *logical pages*, simply referred to as *pages* in the following.

The LEDA manual pages contain a high number of identifiers that also appear in the source code. Actually, the LEDA team wrote scripts which generate manual pages extracting *special* comments from the source files. The comments constituting the manual pages are identified by a markup language. Function names, parameter names, and data type names contained in these comments appear in the manual pages, thus making the traceability link recovery task easier. For this reason, a simplified version of the method shown in Figure 3.1 was applied. The simplification concerned the identifier separation and the text normalization activities. In particular, identifier separation only consisted of splitting identifiers containing underscores, while text normalization was performed only at the first level of accuracy, i.e. the transformation of capital letters into lower case letters.

To validate the results, a 208 $\times$ 88 traceability matrix linking each class to the manual page describing it (Antoniol et al. 1999) was used. Each class is described in at most one manual page, and many classes (110) are not described at all. The number of links in the traceability matrix is 98. Ten manual pages do not describe LEDA classes, but rather basic concepts and algorithms; thus there are 78 relevant manual pages. This means that some manual pages described more than one class. For example,

very often an abstract class and its derived concrete classes were described by the same manual page (there are 20 of such cases).

Table 3.2 shows the results. The first two columns show the number of documents retained for each query (first N documents in the ranked list) and the total number of documents retrieved by all queries for each cut level. The table also shows for each IR model and each cut level the total number of relevant documents retrieved by all queries, the aggregate precision, and recall values (see Section 1.3.4).

		Probabilistic IR model			Vector Space IR model		
Cut	Retrieved	Relevant	Precision	Recall	Relevant	Precision	Recall
1	208	81	38.94 %	82.65 %	52	25.00 %	53.06 %
2	416	88	21.15 %	89.79 %	71	17.06 %	72.44 %
3	624	93	14.90 %	94.89 %	79	12.66 %	80.61 %
4	832	93	11.17 %	94.89 %	82	9.85 %	83.67 %
5	1040	93	8.94 %	94.89 %	85	8.17 %	86.73 %
6	1248	93	7.45 %	94.89 %	89	7.13 %	90.81 %
7	1456	94	6.45 %	95.91 %	90	6.18 %	91.83 %
8	1664	94	5.64 %	95.91 %	93	5.58 %	94.89 %
9	1872	95	5.07 %	96.93 %	95	5.07 %	96.93 %
10	2080	95	4.56 %	96.93 %	96	4.61 %	97.95 %
11	2288	95	4.15 %	96.93 %	96	4.19 %	97.95 %
12	2496	96	3.84 %	97.95 %	98	3.92 %	100.00 %

Table 3.2: LEDA results.

The poor results are due to the fact that most of the queries (110) were derived from classes without relevant associated manual pages. These queries contribute to the total number of retrieved documents. The main difference between the two IR models is that the probabilistic model retrieves most of the documents with smaller cut values.

However, the vector space model achieves 100 % recall when cutting the ranked list of documents at 12 candidates while the probabilistic model achieves 97.95 %. The probabilistic model achieves 100 % recall at a cut level of 17 candidates (level not reported in Table 3.2). As shown in Figure 3.2, the vector space model obtains 100 % recall faster than the probabilistic one.

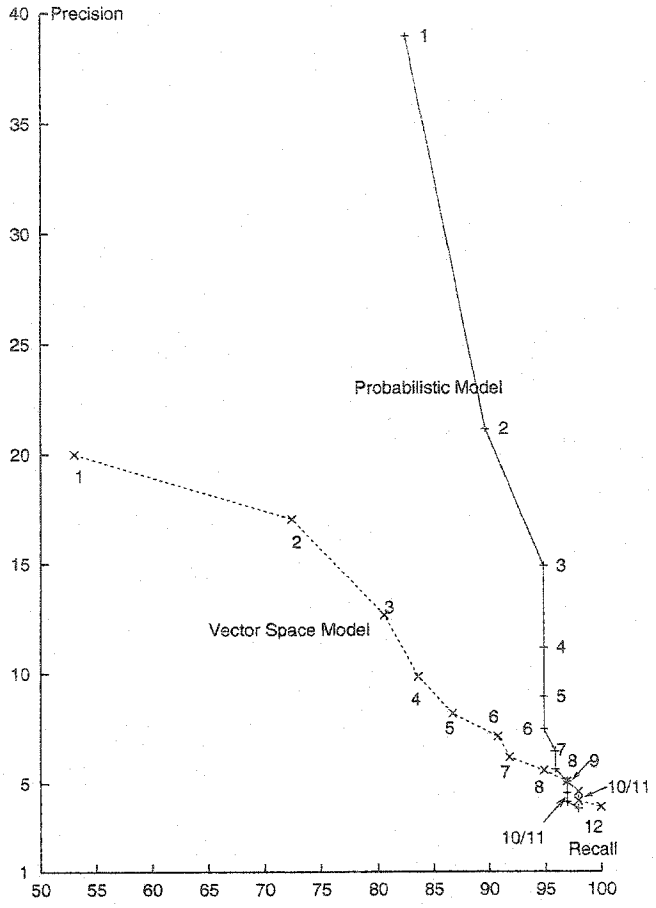

Figure 3.2: LEDA precision/recall results.

3.2.3 Albergate Case Study

The second case study involves Albergate, a software system described in Appendix A. The case study focuses on recovering traceability links between functional requirements written in Italian and Java source code. Most of the functional requirements are implemented by a small number of classes: on the average, a requirement was implemented by about 4 classes with a maximum of 10. Most classes are associated with one requirement, only 6 classes are associated with two requirements, and 8 classes are not associated with any. The total number of links in the traceability matrix is 58.

In this case study, the full version of the text processing steps described in Sub-

section 1.3.3 is applied. The relative distance between source code and documents is higher than in the LEDA case study. Common words between requirements and classes are quite infrequent in the Albergate system. In fact, unlike LEDA manual pages, Albergate functional requirements were produced in the early phases of the software development life cycle. Moreover, the Italian language has a complex grammar: verbs have many more conjugated variants than English verbs, plurals as well as adverbs and adjectives often have irregular forms.

Table 3.3 shows the results of this case study (the meaning of the columns is the same as in Table 3.2). Unlike the LEDA case study, the results of the vector space model are not very different form those produced by the probabilistic model (see Figure 3.3). However, for the probabilistic model, 100 % recall was obtained by considering the first 6 documents for each class, while for the vector space model all traceability links were recovered by considering the first 7 documents for each class.

		Probabilistic IR model			Vector Space IR model		
Cut	Retrieved	Relevant	Precision	Recall	Relevant	Precision	Recall
1	60	29	48.33 %	50.00 %	29	48.33 %	50.00 %
2	120	41	34.16 %	70.68 %	34	28.33 %	58.62 %
3	180	45	25.00 %	77.58 %	46	25.55 %	79.31 %
4	240	51	21.25 %	87.93 %	51	21.25 %	87.93 %
5	300	57	19.00 %	98.27 %	54	18.00 %	93.10 %
6	360	58	13.80 %	100.00 %	55	15.27 %	94.82 %
7	420	58	13.80 %	100.00 %	58	13.80 %	100.00 %

Table 3.3: Albergate results with improved process.

3.2.4 Probabilistic Versus Vector Space Model

The two case studies suggest that both IR models (vector space and probabilistic) are suitable for the problem of recovering traceability links between code and documentation. The results are very similar, in particular with respect to the number of documents a software engineer needs to analyze to get very high values of recall. However, the data show that the probabilistic model achieves higher values of recall with smaller cut values and makes little progress toward 100% recall. On the

other hand, the vector space model starts with lower recall values and makes regular progress with higher cut values toward 100% recall.

A possible explanation lies in the nature of the two models. The probabilistic model associates a source code component (in our case studies a class) with a document based on the product of the unigram probabilities with which each code component identifier appears in the software document (Antoniol et al. 1999; Antoniol et al. 2000b).

These probabilities are estimated via equations (3.6, 3.7, 3.8); code component identifiers that do not appear in the document are assigned a very low probability. Conversely, the similarity measure of a vector space model only takes into account the code identifiers that also appear in the document and weight the frequencies of the occurrences of such words in the document (code component) with respect to a measure of their distribution in other documents (code components, respectively).

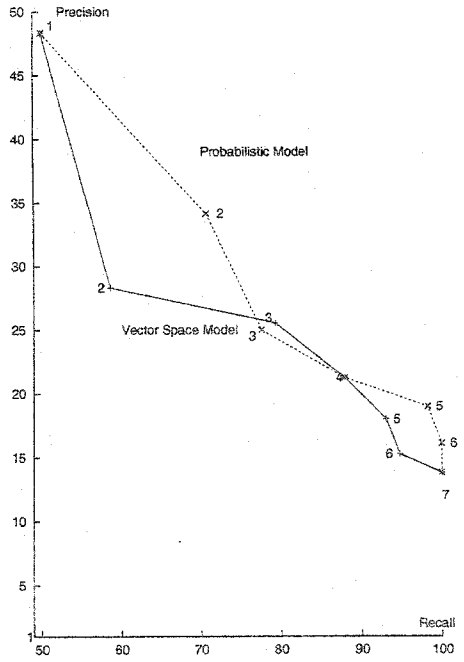

Figure 3.3: Albergate precision/recall results with improved process.

Therefore, the probabilistic model is more suitable for cases where the presence

of code component identifiers that do not belong to the software document is low: this is also the reason why, with respect to the best match, the probabilistic model performs better in the LEDA case study (82.65 % recall) than in the Albergate case study (50 % of recall). It should be noted that the probabilistic model exploited is also used in speech recognition (De Mori 1998) and information theory fields (Cover and Thomas 1992) in which the objective is to associate a received sentence with a possible transmitted sentence, at a very low error probability. Conversely, the vector space model fits cases in which each group of words is common to a relatively small number of software documents (Antoniol et al. 2000a). The vector space model does not likely aim for the best match, but rather for the maximum recall with a moderate number of retained documents.

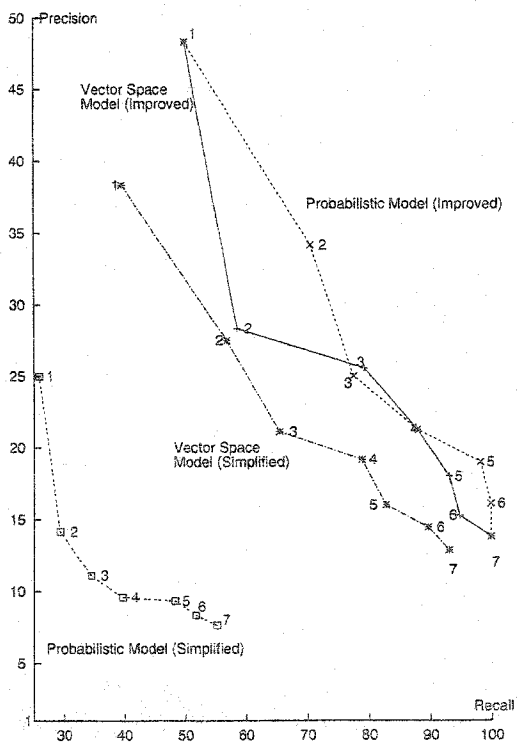

Figure 3.4: Albergate precision/recall results.

This hypothesis is supported by the results obtained by applying the simplified version of the text processing steps in Figure 3.1 and by applying it, with both

the probabilistic and the vector space models, to the Albergate case study. The simplified versions of the identifier separation and text normalization steps produce code components and software documents with a higher number of different words. Table 3.4 shows the results achieved, while Figure 3.4 shows the Precision versus Recall curves of the two IR models, in both simplified and improved processes. For the vector space model, the results of the simplified and improved processes are not very different. Conversely, the differences are evident when applying the two versions of the text processing steps with the probabilistic model (Antoniol et al. 2000b). This means that, unlike the probabilistic model, the vector space model is able to achieve higher recall values based on a smaller number of relevant words in a source code component.

		Probabilistic IR model			Vector Space IR model		
Cut	Retrieved	Relevant	Precision	Recall	Relevant	Precision	Recall
1	60	15	25.00 %	25.86 %	23	38.33 %	39.65 %
2	120	17	14.16 %	29.31 %	33	27.50 %	56.89 %
3	180	20	11.11 %	34.48 %	38	21.11 %	65.51 %
4	240	23	9.58 %	39.65 %	46	19.16 %	78.86 %
5	300	28	9.33 %	48.27 %	48	16.00 %	82.75 %
6	360	30	8.33 %	52.72 %	52	14.44 %	89.65 %
7	420	32	7.61 %	55.17 %	54	12.85 %	93.10 %

Table 3.4: Albergate results with simplified process.

3.2.5 Comparing IR Models with grep

More insight on the usefulness of IR approaches is obtained by comparing those approaches with a more *brute-force* search. The results achieved in the two case studies with the probabilistic and vector space IR models were compared with the results obtained by using the `grep` UNIX utility, as proposed in (Maarek et al. 1991). In fact, `grep` provides the simplest way to trace source code components (e.g., classes) onto high level documentation e.g., manual pages and/or requirements. The search can be done in at least two ways: in the first approach, each class identifier is used as the string to be searched in the files of high level artifacts, while the second

approach considers the *or* of the class identifiers.

Table 3.5 shows the results of the `grep` approach: it should be noted that for the Albergate system 94% of the single item queries gave empty results while, if items were *or* combined, 94% of classes were traced to 10 or more requirements. Empty sets are less frequent for `LEDA`; however, the average number of traced manual pages is quite high (20 and 75, respectively). Even worse, the `grep` approach does not offer any way to rank the retrieved requirements. From a practical point of view, this means that maintainers have to examine a large number of candidates with the same priority.

	Single Code Item				Code Items *or* Combined			
	#Queries	#Empty	Mean	Max	#Queries	#Empty	Mean	Max
Albergate	4834	4575	5	14	60	0	11	13
LEDA	4670	451	20	88	208	1	75	88

Table 3.5: `grep` results: number of queries, retrieved empty sets, mean and max sizes of the retrieved sets.

3.2.6 Benefits of IR in a Traceability Recovery Process

To assess the effectiveness of the proposed approach, a preliminary *in-field* study was designed. The study concerned the Albergate system and involved eight students: four final year undergraduates and four postgraduates. All were familiar with the procedural and the object-oriented programming paradigms; however, their experience with Java was quite different. The undergraduates had been introduced to Java just six months before the experiment took place. In particular, they attended a course during which they developed a small project using Java. On the other hand, the postgraduates had significant experience with Java. They had learned it in undergraduate courses and used it to develop their graduation thesis. Moreover, at the time of our experiment, they were involved in other Java-based projects.

Two groups were formed: Group A (three undergraduates and one postgraduate) and Group B (three postgraduates and one undergraduate). The same task was

	Recall	Precision
Probabilistic model	50%	48%
Group A	65%	60%
Group B	57%	53%

Table 3.6: Average results.

	Recall	Precision
Undergraduate 1	65%	63%
Undergraduate 2	65%	56%
Undergraduate 3	58%	44%
Postgraduate 1	72%	76%

Table 3.7: Detailed results of Group A.

assigned to both groups: the reconstruction of the Albergate traceability matrix. A copy of the requirements document and of the Albergate source code was given to each student. In addition, students of Group A also received, for each source code class, the ranked list of requirements obtained by applying the traceability link recovery method to the probabilistic IR model. However, no indication of where the ranked lists had to be cut was provided.

On average, the performance of the two groups was better than the performance achieved by the probabilistic IR model on the best match, as shown in Table 3.6. However, the best results were obtained on the average by the students that exploited the results of the traceability link recovery method (Group A). It should be pointed out that this group mainly consisted of undergraduate students who were less experienced with Java. Also, the best performance within Group A was obtained by the postgraduate student, as shown in Table 3.7. Table 3.8 shows the results achieved by students of Group B.

3.2.7 Considerations on Effort Saving and Document Granularity

Although it is not possible to generalize from the limited sample, the preliminary data demonstrate the benefits of supporting a software engineer with an automated

	Recall	Precision
Postgraduate 2	53%	56%
Postgraduate 3	70%	40%
Postgraduate 4	53%	59%
Undergraduate 4	55%	58%

Table 3.8: Detailed results of Group B.

approach based on IR models. In the previous sections, the results were compared using the IR metrics *recall* and *precision*. To achieve an indication of the benefits of using an IR approach in a traceability link recovery process, it is necessary to quantify the potential effort saving; for this purpose, the Recovery Effort Index (REI) metric was defined as the ratio between the number of documents retrieved and the total number of documents available:

$$REI = \frac{\#Retrieved}{\#Available}\% \tag{3.16}$$

This metric $(1-REI)$ can be used to estimate the percentage of the effort required to manually analyze the results achieved by an IR tool (and discard false positive), with respect to a completely manual analysis when the recall is 100 %. The lower the REI, the higher the benefits of the IR approach.

At present, there is no statistical evidence of a correlation between the REI and the effort required to accomplish the recovery task. The student Group A on average accomplished the task in less time; however, the t-test on the available data does not reject the null hypothesis i.e., we cannot prefer one method over the other.

It is interesting to observe that the REI also measures the ratio between the precision of the results achieved by a completely manual process, namely P_m, and a semi-automatic IR tool-based process, namely P_t, on the same software system when the recall is 100 %:

$$\frac{Precision_m}{Precision_t} = \frac{\#(Relevant \wedge Retrieved_m)}{\#(Relevant \wedge Retrieved_t)} \frac{\#Retrieved_t}{\#Retrieved_m} \tag{3.17}$$

Note that the number of relevant documents retrieved is the same in both processes (all relevant documents) and that the documents retrieved with a manual analysis are just all documents available. Thus:

$$\frac{Precision_m}{Precision_t}\% = \frac{\#Retrieved_t}{\#Available}\% \tag{3.18}$$

The values of REI measured in the two case studies for the vector space IR model are rather different: Albergate requires 43.75 % REI to achieve 100 % recall, whereas LEDA only requires 13.63 % REI. A possible explanation is that the set of available documents in the Albergate case study is smaller (16 functional requirements versus the 88 manual pages of LEDA); to get the same REI as in the LEDA case study, the maximum recall would have to be achieved with about two documents retrieved (also about 50 % precision). However, this is very unlikely to be achieved with IR methods which generally aim to retrieve a small percentage of a huge document space. Therefore, it is possible that greater benefits (and lower values of REI) are achieved for document spaces of greater size. Larger document spaces are also achieved when different concepts are included in different documents. Therefore, it is likely that greater benefits from this approach are achieved when the granularity of the concepts included in the documents is finer.

Alternatively, the REI could be computed with respect to a manual analysis supported by grep (queries with or combined items). In this case, the REI is computed as the ratio between the number of relevant documents retrieved with an IR approach and the number of documents retrieved by grep. This requires the grep based approach to achieve 100 % recall, as in our case. For the vector space model the values for REI are 54.54 % in the Albergate case study and 16 % in LEDA.

3.2.8 Retrieving a Variable Number of Documents

In the case studies described, a fixed number of documents for each query was retained. The results achieved for the recall can be considered good, as in both case

studies 100 % recall was achieved with a moderate number of retained candidates per query.

Inspired by the traditional probabilistic IR model which used a threshold to locate the R set, one could argue that a variable number of retained candidates per query could improve precision and REI, while maintaining a maximum recall. The approach adopted to explore this hypothesis consisted of using a threshold on the similarity values to prune the ranked list of documents retrieved by a query. In particular, for each query Q we computed the value of such a threshold t_Q as a percentage of the similarity measure of the best match:

$$t_Q = c * [\max_i \sigma(D_i, Q)] \tag{3.19}$$

where $0 \leq c \leq 1$. A query Q returns all and only the documents D_k such that $\sigma(D_k, Q) \geq t_Q$. Of course, the higher the value of the parameter c, the smaller the set of documents returned by a query.

Table 3.9 shows the results achieved with the vector space IR model for the Albergate case study using different values of the parameter c. The results are not very encouraging, as the maximum recall is achieved when setting the threshold to only 10 % of the highest similarity measure. Using this percentage, the average number of retrieved documents per query is 9, while 3 documents are retrieved in the best case, and 15 documents in the worst case.

c	Retrieved	Relevant	Precision	Recall
90 %	59	29	49.15 %	50.00 %
70 %	101	38	37.62 %	65.51 %
50 %	158	50	31.64 %	86.20 %
30 %	265	55	20.75 %	94.82 %
10 %	484	58	11.98 %	100.00 %
min(10 %, best 7)	329	58	17.62 %	100.00 %

Table 3.9: Albergate results using a threshold.

Although the results for the precision are worse than the results achieved with

a fixed cut (first 7 documents in Table 3.3), they still demonstrate the benefits of using an IR approach. When the recall is 100 % (c = 10 %), 484 documents are retrieved, whereas a completely manual approach requires examination of 960 possible queries (16 requirements times the 60 classes), and the resulting REI is 50.41 %. This means that presumably about 50 % of the effort can be saved by only discarding the documents whose similarity measure is below 10 % of the best match.

The results can be improved by combining the paradigm of variable and fixed cut: each query retrieves only the documents with a similarity measure greater than a given threshold, but no more than a fixed number. As an example, the last row in Table 3.9 shows the results achieved by considering as the number of documents retrieved by each query the minimum between 7 and the number of documents whose similarity value is higher than 10 % of the best match. In this case, the results are better than the results achieved with a fixed cut (the first 7 documents in Table 3.3): the average number of retrieved documents is 6 and the REI is 34.27 %, that means that the percentage of effort saved might be more than 65 %.

3.3 Chapter Summary

The chapter presented two novel approaches inspired by information retrieval to recover traceability links between source code and textual documentation. Using the available data, the traditional information retrieval probabilistic approach attained a significantly lower accuracy than the vector space model and the approach derived from stochastic language models. Stochastic language models were first applied in computational linguistics and spoken language recognition, but they also proved to be an effective means to recover traceability in software engineering too.

Both models achieve 100% recall with almost the same number of retrieved documents. However, the probabilistic model achieves high recall values (less than 100%) with a smaller number of retrieved documents and then performs better when 100% recall is required. Benchmarking the mentioned approaches against a `grep` brute force traceability link recovery demonstrates the benefits of the more sophisticated

technologies. As in (Maarek et al. 1991), `grep` is overwhelmed by IR approaches.

Two main limitations can be identified. First vector space and stochastic language models need to be further validated on larger systems to assess their relative performance. Reported results were obtained on two systems whose representativeness of the industrial practice should be assessed.

The second limitation is common to software engineering data sets: training material is often scarce and the adequacy of a probabilistic model can be questioned. A quick comparison between the size of software engineering documentation and the size of natural language processing corpora makes it clear that obtained results suffer from poor training. Thus, system sizes, application domains, and volume of documentation suggest caution in generalizing the obtained accuracy to other software systems.

CHAPTER 4: A Bigram Language Model

4.1 Mapping Model

Existing cognitive models share the theory that program comprehension occurs in a bottom-up manner (Pennington 1987a; Pennington 1987b), a top-down manner (Brooks 1983; Soloway and Ehrlich 1994), or some combination of the two (Letovsky 1986; Mayrhauser and Vans 1993; Mayrhauser and Vans 1994; Mayrhauser and Vans 1996). Researchers also agree that programmers use different types of knowledge ranging from domain specific to general programming knowledge (Shneiderman and Mayer 1979; Brooks 1983; Vessey 1985). Programmers need to build a mental model of the software system, i.e., high-level abstractions preliminary to development or maintenance activity.

In this chapter, we assume that such a mental model exists, in other words, that programmers tend to process application-domain knowledge and use high-level abstractions consistently. Thus, program item names of different code regions, or words of a given text document related to the same concepts, are likely to be the same or at least very similar. Following this assumption, the knowledge of existing traceability links can be exploited by modeling the mental map adopted by programmers to link high-level concepts present in textual documentation with source code regions.

In Chapter 3 and in articles (Antoniol et al. 2000d; Antoniol et al. 2002) stochastic modeling and maximum likelihood classification were adopted to model programmers as stochastic communication channels. The main limitation was the simple idea that concepts are captured by single words. To overcome this, the math-

ematical model was redefined to accommodate subsets of words. The number of possible word sequences for any given text document is the power set of the words. To avoid a combinatorial explosion, the method was limited to sequences of adjacent words because for each document, the number of terms considered, i.e., single words plus pairs of adjacent words, is upper bounded by twice the number of words in the document itself.

4.1.1 Programmer's Cognitive Model

Programmers consistently map problem and domain concepts in software artifacts: both low-level and high-level artifact terms reflect programmer knowledge. In other words, the terms in different parts of software artifacts at a given stage of the software life cycle (e.g., the design documents or the code components) bound to a given concept are likely to be similar, while they are potentially different from those used in higher-level documents such as Software Requirements Specifications (SRS) and produced in early phases.

Consider, for example, a requirement from a library management application and an excerpt of its implementation as shown in Figure 4.1. Assume this hypothetical requirement state:
To insert a new book into the library the user must insert the following information:

- **title**
- **author's** *name*
- **number** *of* **pages**

After **confirming** *the data inserted, they are stored into the database. In case of* **abort**, *nothing shall happen.*

In an OO implementation, programmers will map this requirement into different classes including a class containing the book data structure (an excerpt of such a class is shown in Figure 4.1-a) and a class implementing a new book insertion form (Figure 4.1-c and Figure 4.1-b). Let us highlight, for instance, these different situations:

1. The same terms appear in the requirements and in the code (`title`).

2. The identifiers `Add` and `Canc` are associated, respectively, with the words `confirming` and `abort`;

3. A different term (`Npages`) is associated with a word (`pages`) used in the requirement and used consistently in the code.

4. The requirement words `number` and `pages` may be mapped to the identifier `Npages`. This means that in some cases programmers map a set of words corresponding to a domain or problem concept into a composite program item name.

5. The requirement identifier `authors` is mapped ambiguously into two different terms `Auths` and `Authors`.

a)

```
public class Book {
  String Auths;
  String Title;
  int NPages;
  ...
  public Book() {
  }
  ...
}
```

b)

```
public class AddBook extends JFrame {
  JPanel contentPane;
  GridBagLayout gridBagLayout1 = new GridBagLayout();
  JTextField Authors = new JTextField();
  JLabel jLabel1 = new JLabel();
  JTextField Title = new JTextField();
  JLabel jLabel2 = new JLabel();
  JTextField Npages = new JTextField();
  JLabel jLabel3 = new JLabel();
  JButton Canc = new JButton();
  JButton Add = new JButton();
  Border border1;
  ...
  /**Construct the frame*/
  public AddBook() {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  ...
}
```

c)

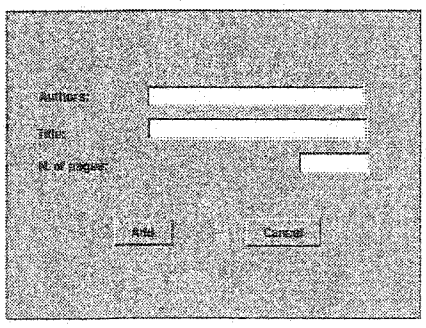

Figure 4.1: How the programmer can map concepts into code.

The main idea is that, as in point 4, several words of a requirement may contribute to creating a single program item name. However, a concept or object may

often be described by a single word or a couple of adjacent words. Although some previous research suggested that a concept should always be expressed by a single word, generally a noun, see (Booch and Rumbaugh 1998), documents and source code inspection revealed a non-negligible fraction of concepts mapped into multiple words (see Section 4.4.2).

This observation led to the hypothesis that, when reading requirements, programmers tend to use a mental "sliding window" spanning a limited number of words (not counting *stop words* i.e., articles, prepositions) to create program item names.

A challenging issue is to identify the window size, i.e., the maximum number of adjacent words representing a concept. One may decide to consider the power set of the words in a requirement. However, this is unfeasible, and in practice, a concept is often well located in the text. Thus, a sliding window of a few words should be considered. This follows the general consensus concerning short-term memory and the ability to concurrently process about seven different information chunks (Miller 1963; Miller 1975).

Another issue is to weight single words and words sequences differently; it is possible that concepts are retained in a single word with a higher probability than in multiple words. In our model, this is formalized by different weights associated with terms of different size (see Section 4.1.2). Details on calibrating such weights can be found in (De Mori 1998).

Figure 4.1-b highlights a class containing identifiers automatically generated by a tool. In this case the Java class, implementing a frame has been drawn using Borland JBuilder^{TM}. These identifiers are not tied to the problem domain, programmer experience, or mental model. They represent one of the factors negatively affecting traceability and traceability recovering processes.

Finally, it has been noticed that, if identifiers associated with a concept in requirements are used ambiguously (`Auths` and `Authors` associated with the word `authors`), the performances of a recovery process are likely to deteriorate as the percentage of these inconsistencies increases.

4.1.2 Mathematical Model

Stochastic models are extensively used in several areas: automatic speech recognition, machine translation, spelling correction, text compression, etc. (Brown et al. 1994; Kukich 1992; Witten and Bell 1991). In the software development and maintenance life cycle, programmers may be thought of as stochastic channels transforming a high-level text document into high-level abstractions and knowledge and finally into *observations* i.e., chunks of code. Development or maintenance activities are modeled by a stochastic channel where the sequence of words representing high-level text document W_k, which is generated by a source S with the *a-priori* probability $Pr(W_k)$ and which is mapped by a programmer into an abstraction A_k, is transformed into the sequence of observations O with probability $Pr(O \mid W_k)$, or more precisely with probability $Pr(O \mid A_k)$. In traditional pattern recognition applications, O could represent, for example, the acoustic signal produced by uttering W_k, the translation of W_k from Italian to English, or a typewritten version of W_k with possible misspellings. In software engineering, to model programmer activities, W_k may represent a functional requirement or a maintenance change request, while O may correspond to the program item names i.e., the mnemonics for code identifiers chosen by the programmers. Notice that any word of the W_k document, as well as other sources of knowledge, will possibly influence the entire observation O.

As stated in Section 4.1.1, the method's underlying assumption is that programmers tend to build code identifiers, variables and function names, by applying unknown rules and domain and problem knowledge. These rules do not change significantly over time or from one programmer to another since they depend on corporate culture, adopted coding standards, and programmers' skills. Hence, the program item names of different code regions are likely to share semantic meaning with high level abstractions and concepts, or at least they have been consistently created by applying a set of unknown rules. Thus, program item names from different code regions related to a given high-level abstraction or concept possibly corresponding to a given text

document are likely to be the same, or at least very similar.

To formulate the traceability recovery model, it was assumed that concepts embodied into a fragment of a high-level document can be represented and captured in a high-level abstraction by terms appearing in the text fragment. A term corresponds to:

1. A single word (e.g., the unigrams as in Chapter 3).

2. Two or more consecutive words.

3. More generally, a subset of the words belonging to the text fragment.

Given a document W_k, its abstract representation A_k is modeled by the superposition of terms $t_{k,i}$:

$$A_k = \bigcup_i \{t_{k,i}\} \tag{4.1}$$

In general, the difficulty in precisely locating concepts and words triggering concepts leads to a combinatorial explosion. Thus, it is assumed that concepts are represented and triggered by a single word, or by at most two adjacent words; hereafter called *bigram*. The probability $Pr(A_k)$ can be expanded as:

$$Pr(A_k) = Pr((\bigcup_{i=1}^{n} \{w_{k,i}\}) \bigcup (\bigcup_{i=1}^{n-1} \{w_{k,i} w_{k,i+1}\})) \tag{4.2}$$

where n is the requirement vocabulary size. $\{w_{k,i}\}$ were assumed independent events in Chapter 3 and in other related documents (Antoniol et al. 2000d; Antoniol et al. 2002). However, $\{w_{k,i}\}$ or $\{w_{k,i} w_{k,i+1}\}$, $i = 1, \ldots, n$ independence can be questioned. The document specific weight $\lambda_k \in [0, 1]$ can be introduced to rewrite the equation (4.2) as:

$$Pr(A_k) = Pr(\bigcup_{i=1}^{n} \{w_{k,i}\}) + \lambda_k Pr(\bigcup_{i=1}^{n-1} \{w_{k,i} w_{k,i+1}\}) \tag{4.3}$$

The problem involves decoding the observation O, e.g., program item names, into the original high-level document W_k. This requires finding $\widehat{W_k}$ which maximizes the *a-posteriori* probability $Pr(W_k \mid O)$ under the assumptions $Pr(W_k \mid O) = Pr(A_k \mid O)$ and $Pr(W_k) = Pr(A_k)$ (i.e., under the hypothesis that the document probability matches the abstraction probability). By applying the Bayes' rule, the following identity is obtained:

$$Pr(W_k \mid O) = Pr(A_k \mid O) = \frac{Pr(O \mid A_k)Pr(W_k)}{Pr(O)} \tag{4.4}$$

$Pr(W_k)$ is the high-level document *a-priori* probability. Since there is no reason to prefer a particular high-level document i.e., there is no *a-priori* information on the document distribution, it can be safely assumed that documents have the same probability. Furthermore, in the above equation, $Pr(O)$ is a constant with respect to k and could be eliminated. Hence, decoding is equivalent to find:

$$\widehat{A_k} = arg\max_{A_k} Pr(O \mid A_k) \tag{4.5}$$

The high-level document W_k, e.g., a functional requirement, is mentally mapped by programmers into an abstraction A_k, which in turn is transformed into an observation $O = \bigcap_{j=1}^{m}\{o_j\}$, i.e., a chunk of source code. Thus:

$$Pr(O|A_k) = Pr(\bigcap_{j=1}^{m} o_j | \bigcup_{i} t_{k,i}) \tag{4.6}$$

By further assuming $\{o_1\}\bigcap\{o_2\}\ldots\bigcap\{o_m\}$ as conditionally independent from A_k, i.e., dependencies between $\{o_p\}, \{o_q\}$ are modeled by the conditional probability of events $\{o_p|A_k\}, \{o_q|A_k\}$, the above equation (4.6) can be rewritten as:

$$Pr(O|A_k) = \prod_{j=1}^{m} Pr(o_j|A_k) \tag{4.7}$$

In other words, each o_j depends on the entire abstraction A_k. The conditional probability $Pr(o_j|A_k)$ can be simplified as:

$$Pr(o_j|A_k) = \frac{Pr(o_j A_k)}{Pr(A_k)} = \frac{Pr(o_j \bigcap (\bigcup_i t_{k,i}))}{Pr(A_k)} \tag{4.8}$$

The event $o_j \bigcap (\bigcup_i t_{k,i})$ can be rewritten as $\bigcup_i (o_j t_{k,i})$. However, since $t_{k,i}$ were not assumed independent events, when considering single words and couples of adjacent words, the following relation holds:

$$\begin{aligned} Pr(o_j A_k) &\simeq Pr((\bigcup_{i=1}^{n} o_j\{w_{k,i}\}) \bigcup (\bigcup_{i=1}^{n-1} o_j\{w_{k,i}w_{k,i+1}\})) \\ &= Pr(\bigcup_{i=1}^{n} o_j\{w_{k,i}\}) + \lambda_k Pr(\bigcup_{i=1}^{n-1} o_j\{w_{k,i}w_{k,i+1}\}) \end{aligned} \tag{4.9}$$

λ_k may also be considered as a scaling factor *weighting* our belief that a concept is mapped by a bigram. To obtain a tractable form, a λ'_k may be chosen so that event dependences are accounted for (e.g., $o_j\{w_{k,i}w_{k,i+1}\}$ versus $o_j\{w_{k,j}w_{k,j+1}\}$):

$$\begin{aligned} Pr(o_j|A_k) &\simeq \sum_{i=1}^{n} Pr(o_j|w_{k,i}) Pr(w_{k,i}) + \\ &\quad \lambda'_k \sum_{i=1}^{n-1} Pr(o_j|w_{k,i}w_{k,i+1}) Pr(w_{k,i}w_{k,i+1}) \end{aligned} \tag{4.10}$$

By substituting equation (4.10) into equation (4.7) the following expression of $Pr(O|A_k)$ is obtained:

$$\begin{aligned} Pr(O|A_k) &\simeq \prod_{j=1}^{m} (\sum_{i=1}^{n} Pr(o_j|w_{k,i}) Pr(w_{k,i}) + \\ &\quad \lambda'_k \sum_{i=1}^{n-1} Pr(o_j|w_{k,i}w_{k,i+1}) Pr(w_{k,i}w_{k,i+1})) \end{aligned} \tag{4.11}$$

Equations (4.5) and (4.11) are central to the method as they provide a means to effectively recover traceability links based on the *a-priori* knowledge of a subset of traceability links. The latter are used to provide an initial estimate for the probabilities involved in the equation (4.11).

The involved probabilities $Pr(o_j|w_{k,i})$ and $Pr(w_{k,i})$, as well as $Pr(o_j|w_{k,i}w_{k,i+1})$, $Pr(w_{k,i}w_{k,i+1})$, and λ'_k, need to be estimated on a *labeled* training set, i.e., on a subset known traceability relations. Given the labeled training set, for example, the unigram probabilities may be approximated with frequencies:

$$Pr(o_j|w_{k,i}) \simeq \frac{c(o_j w_{k,i})}{c(w_{k,i})} \; i = 1 \ldots n \; j = 1 \ldots m \tag{4.12}$$

$$Pr(w_{k,i}) \simeq \frac{c(w_{k,i})}{|W_k|} \tag{4.13}$$

where $c(h)$ is the number of times that the h word appears in the texts. In the same way, $c(hl)$ is the number of times that the couple h, l appears (h in the observation O and l in the document W_k), while $|W_k|$ is the number of words in W_k i.e., $|\ |$ gives the set cardinality. A similar approximation allows us to estimate the remaining probabilities. Probabilities and λ'_k are chosen so that they minimize the error over the training set.

4.1.3 Model Generation

As in Chapter 3, probability estimation was based on word frequency. This process suffers from the same problem of zero-frequency and smoothing already addressed in Chapter 3 and the same strategies were applied. In other words, results were obtained with a closed vocabulary where probabilities were smoothed according to the shift-β smoothing techniques.

4.1.4 Model Assessment

To obtain an unbiased estimate of the performance of the traceability recovery method, a cross-validation approach (Stone 1974) was applied to three different software systems. Given the traceability maps of components, these were divided in two non-overlapping sets: a training set and a test set. More precisely, the following sets were considered:

- CS_i, the set of software components implementing the ith requirement.
- TR_i, the Training Set related to the ith requirement; TR_i *components* were used to estimate the probabilities required ($TR_i \subseteq CS_i$).
- TS_i, the Test Set related to the ith requirement ($TS_i \subset CS_i$ and $TR_i \cap TS_i = \emptyset$), TS_i *components* were used as queries to evaluate the (4.5).
- ES_l, the set of couples (TR_{il}, TS_{il}) related to the lth experiment.

In a software system such as *Albergate*, an exhaustive cross-validation on a whole software system was not feasible because there are millions of different and consistent choices of ES_l. For this reason, the experiments aimed at mimicking the incremental reconstruction of traceability links. The probabilities $Pr(o_j|w_{k,i})$, $Pr(o_j|w_{k,i}w_{k,i+1})$ and λ'_k were thus estimated using TR_i and performance measured over TS_i.

4.2 Traceability Problems in Software Systems

Software systems often contain elements that may affect traceability and understandability. Industrial software developed with Commercial Off The Shelf (COTS) components e.g., database access components, automatically generated code e.g., by GUI builders or report generators, and middleware, Common Object Request Broker Architecture (CORBA), pose new challenges to traceability link recovery. Thus the following taxonomy of traceability recovery affecting factors was defined:

1. Partially automatic-generated code.
2. Totally automatic-generated code.
3. COTS and reused code.
4. External Architectures (e.g., middleware, network infrastructures).
5. Design and Implementation-level components.

The above affecting factors (i.e., more precisely the automatically generated identifiers and labels) act as *noise* superimposed on the *signal* (i.e., the program item names chosen by programmers) confusing the traceability recovery process.

In the following sub-sections, each factor will be analyzed and relation to traceability issues established.

4.2.1 Partially Automatic-Generated Code

Partially automatic-generated code are components and classes often related to the GUI of the system, containing both automatically generated code (and identifiers) and code written by programmers. For example, identifiers and names of GUI widgets (labels, text fields, buttons, etc.) are automatically generated following precise naming conventions (e.g., `TButton1, TButton2, TTextField1`, etc.). Naming conventions may be customized, with widget identifiers and names changed from a property window of the Rapid Application Development (RAD) IDE to better reflect high-level documents or application domain concepts. However, programmers tend to give significant and domain consistent names only to a small fraction of automatically generated component elements that they perceive as essential to understanding the application while developing the system. The results are that default names dominate over programmer specific names, particularly for labels, shapes and any visual component having a constant value and behavior.

Similar considerations may be applied to those visual components, encapsulating specific functionalities and dragged into user interfaces. For example, any kind of *visual COTS*, such as Microsoft ActiveX or OCX and Borland VCL, belongs to this category. As a second example, consider a component implementing a database management system connection and queries or encapsulating network services. The component may have a default generated name (e.g., `Table1, Table2, and Query1`) or a user-defined name; if the name is not consistent with the related requirements, traceability will be lost. Notice that these components may encapsulate complex behaviors (update a database table, retrieve a page from the web, print a PDF docu-

ment, etc.). Thus, the more complex the encapsulated behavior, the more severe the traceability problem is.

To maximize traceability, these default-generated identifiers should be filtered out. For example, two automatically generated windows containing the same number and types of widgets will exhibit very similar or identical identifier frequency distribution, although they are associated with different domain concepts and requirements. Even worse, generated identifiers are likely to dominate identifiers chosen by programmers, thus reducing the traceability link recovery precision.

4.2.2 Totally Automatic-Generated Code

Totally automatic-generated code are components and classes automatically generated e.g., created by parser generators, without any need for coding; programmers' intervention, if any, is almost irrelevant (e.g., fill the printer spooler device property of the visual component).

Although these components and classes implement functionalities specified in high-level documents, traceability recovery is unfeasible, for the same reasons described in the previous subsection. Suppose the requirement document contains the following sentence:

"The system must allow the printing of the list of customers from a particular town, specifying names, address..."

Such a requirement is simply implemented by an automatic-generated report containing only identifiers such as `QRBand1`, `QRBand2`, `QRLabel1`, `QRDBText1`, `QRLabel2`, `QRDBText2`.

To ensure traceability, manual intervention is required, either by recording traceability links or assigning consistent names while using the report generator. For example, the fields `QRDBText1` and `QRDBText2` should be mapped into `Customer_name`, `Customer_address`.

4.2.3 COTS and Reused Code

COTS source code is usually not available; the only traceable elements are the uses of COTS resources such as classes, functions. If it is available, there is not guarantee that COTS identifiers reflect the application domain concepts and high-level document names. This happens because the component identifiers are mapped to concepts related to the specific solution. For example, a component for image compression contains identifiers related to Fourier transform, quantization, etc., while the system using it may contain, for example, identifiers related to a geographic information system. Moreover, the COTS could have been developed in a different organization adopting different naming conventions. As for automatically generated code, manual intervention is required.

Reused components are equivalent to COTS except for the availability of source code; their code may have been developed in previous projects possibly related to different application and problem domains; thus consistency between program item names and requirement terms is very unlikely. In other words, reused components and classes should be excluded from the traceability recovery process.

4.2.4 External Architectures

This category encompasses middleware and frameworks such as CORBA or the Microsoft .Net. External architectures behave similarly, at a different level of granularity, to the partially generated code. System components or system class hierarchies are related to other external hierarchies of classes. When an external architecture is integrated, there may be automatically generated source code files or classes with or without programmer hooked code, partially generated classes derived from the external hierarchies with or without overloading, or reused classes from a framework.

A typical example is the use of an Object Request Broker (ORB) in a distributed software system to guarantee interoperability between application and remote objects. A node of the distributed system may contain the following families of classes:

- Classes that implement portions of CORBA. As for other reused code, the traceability link recovery is generally not possible.

- Automatically generated classes: these classes are generated by the Interface Definition Language (IDL) compiler, typically to implement data structures passed around by distributed objects. Since original variable names, as defined in the IDL specification files by programmer, are modified and several other variables are generated, there is no guarantee to recover traceability links.

- Partially generated classes: these are, for example, the CORBA stubs and skeletons containing methods and identifiers with names as defined by programmer in the IDL. These classes can be successfully traced to requirements specifying information communicated to and from remote systems implemented using CORBA.

4.2.5 Design and Implementation-Level Components

Because these components do not appear in requirements, there is no guarantee to ensure their traceability. Consider, for example, the *splash screen* or the *about window* of a GUI application. There is usually no requirement related to these classes. The motivation is grounded into the object-oriented development process itself. Requirements are well mapped into classes reflecting the domain-component view (Coad and Yurdon 1991) or the conceptual view of the system (Booch and Rumbaugh 1998; Jacobson et al. 1999; Bruegge and Dutoit 2000). At the design level, the class hierarchy is modified (Coad and Yurdon 1991; Booch and Rumbaugh 1998; Jacobson et al. 1999; Bruegge and Dutoit 2000) by refining domain objects and by adding details such as reused classes, human interaction classes, data management classes, and task management classes. These components and classes cannot be traced directly to high-level documents. An intermediate step tracing them into design may therefore be required.

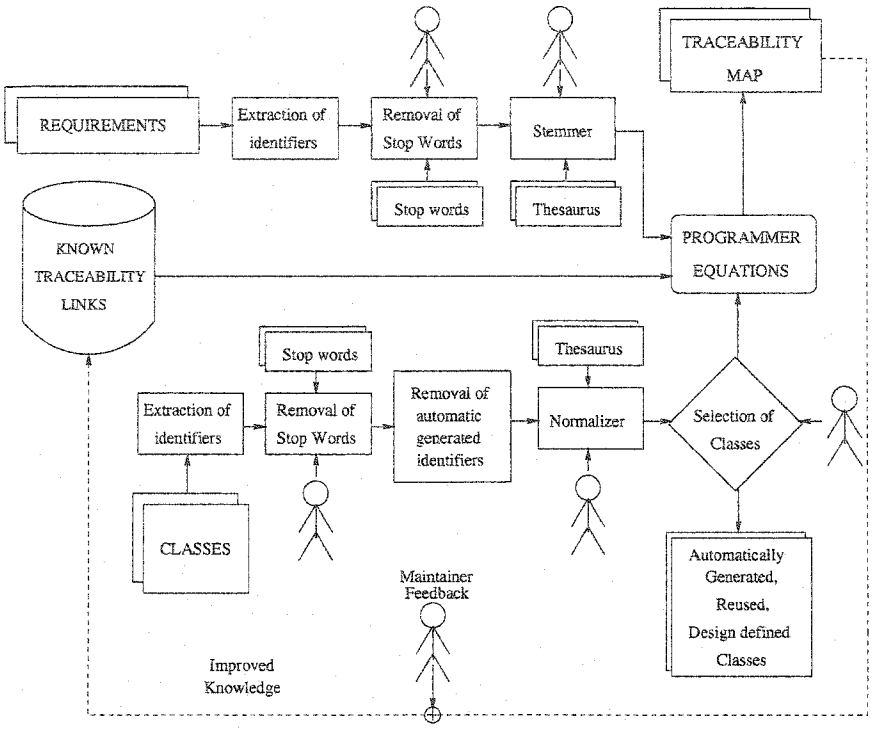

Figure 4.2: Traceability recovery process.

4.3 The Traceability Recovery Process

Figure 4.2 shows the generic process defined to recover traceability links between high-level documents and source code. The process accounts for the affecting factors identified in the previous section; it was instantiated in the context of traceability link recovery between functional requirements and the corresponding OO code.

The underlying approach assumes the considerations of Section 4.1.1, and it further assumes that for each requirement at least one traceability link is available (continuous lines in Figure 4.3). This a-priori information is recovered by a manual analysis, supported by different approaches such as traceability link recovery based on vector spaces or probabilistic language models described in Chapter 3. The knowledge of existing traceability links bootstraps the recovery of remaining unknown links (dashed lines in Figure 4.3).

The recovery process can be considered as consisting of three main blocks, each

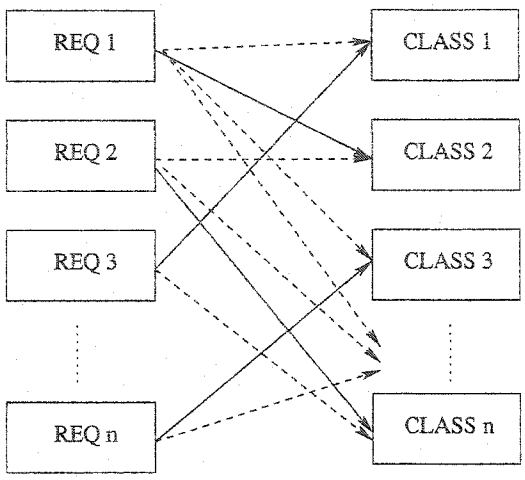

Figure 4.3: Example of traceability links.

one further divided into sub-blocks:

1. Requirements processing.

2. Code processing.

3. Traceability map recovery by means of the Bayesian classifier.

4.3.1 Requirements Processing

Each requirement is processed and the set of its unigrams and bigrams is extracted. Stop words, e.g., articles, are discarded; words have undergone a successive normalization step of morphological analysis (the *Stemmer* box in Figure 4.2). Plural and singular conjugated verbs, as well as synonyms, are brought back to the radix.

These phases, as already stated in Section 1.3.3 cannot usually be fully automated, and are likely to remain semi-automatic; currently available natural language tools cannot fully disambiguate contextual semantics, nor deal with complex forms representing metaphoric speech (e.g., ... *since zombies cannot be killed the system must be restarted* ...). Finally, the vocabulary is built, weighting each word by its frequency.

4.3.2 Code Processing

For each class, a list of identifiers and the associated occurrences is extracted from both the interface and implementation files according to Section 1.3.3. Unlike the previous Chapters 2, 3 and the publications (Antoniol et al. 2000d; Antoniol et al. 2002), comments are not discarded. Extracted program item names are filtered by a stopper. Stop words removal is divided into two sub-phases. In the first sub-phase, exactly the same process as Section 1.3.3 is applied.

The first sub-phase is followed by the removal of automatically generated identifiers because as highlighted in Section 4.2, they may confuse the traceability recovery. Pruning is automatically performed once the list of classes available in the RAD tool library is known; identifiers will have a name generated following well-defined rules. For example, for the class `TLabel` generated identifiers will have names such as `TLabel1`, `TLabel2`.

Morphological analysis is subsequently applied to the remaining words. However, the applied analysis is slightly different from that performed on requirements. Plural and singular verb forms are first normalized; furthermore, event method names dispatched by the same object are brought back to a radix equal to the object name. For example, if there is a text field named `Address`, methods named `AddressClick`, `AddressChange`, etc. are brought back to the radix `Address`. This phase (the box tagged as *"Normalizer"* in Figure 4.2) can be automatically performed when RAD tools are used; the method names have the object name as prefix, followed by the event name (the list of possible events is known).

Finally, as emphasized in Section 4.2, entirely generated classes that cannot be traced into high-level documentation are removed from the system under analysis. This step can be performed automatically once the tools used to develop the system are known.

Reused code and external architectures are processed as untraceable classes and are removed from further consideration. Unfortunately, given such a granularity level,

the removal process requires human intervention. However, file names, comments, documentation (if available) or methods such as those presented in (Antoniol et al. 2001c) may serve to facilitate the activity.

4.3.3 Traceability Map Recovery

Equations reported in Section 4.1.2 are applied to the *a-priori* known links, and probabilities are estimated on training material. Finally, the Bayesian classifiers score traceability links with probability. Once maintainers recover new links, these are in turn used as new inputs to enrich the classifier training set; the re-trained classifier is then applied to the remaining class-requirement pairs.

4.3.4 Tool Support

To automate the traceability recovery process, a number of tools have been developed:

1. A script extracting bigrams, single words, and their occurrences from requirements.

2. A script that parses the source code (written in Java, C++ or Visual Basic) and extracts the list of couples identifier/occurrence keywords, language types and language's symbols are removed.

3. A script that, given the names of library classes used by a RAD or present in a middleware, removes all automatically generated identifiers from the code.

4. A tool that implements the estimation of probabilities $Pr(o_j|w_{k,i})$, $Pr(w_{k,i})$, $Pr(o_j|w_{k,i}w_{k,i+1})$, $Pr(w_{k,i}w_{k,i+1})$ and λ'_k with the shift-β smoothing and closed vocabulary.

4.4 Experimental Results

4.4.1 Test Suite

A feasibility study and method assessment were performed by recovering traceability links between functional requirements and source code of three software systems having different characteristics:

- the Albergate Java system, developed following the Boehm waterfall model (Boehm 1981) and implemented without making use of RAD tools.
- the Transient Meter C++ system, developed following a prototyping approach, and using of RAD tools, COTS and external software architectures.
- A Visual Basic system, developed following a waterfall model, but implemented using of the Visual Basic IDE.

Details on Albergate, Transient Meter and the Visual Basic Library Management Software can be found in the appendices. These systems are quite different; Transient Meter suffered from almost all traceability problems outlined in Section 4.2: it was built using a RAD, reused components, and was implemented with a CORBA architecture. The traceability recovery process was applied to the central node, designed as the node that performs complex activities such as system startup, configuration monitoring, waveform processing (e.g., Fast Fourier Transform), and visualization.

4.4.2 Case Study Results

This section reports the case study results obtained applying the proposed traceability recovery process to the software systems described in Section 4.4. To make the comparison easier, results presented in Chapter 3 are also summarized here, since they constitute the comparison baseline.

For all the three systems studies were also performed, considering up to three-word terms (i.e., single words, plus bigrams, plus trigrams). However, adding trigrams did not improve performances with respect to bigrams. This may indicate that, for the

three systems analyzed, most of the concepts were retained in terms consisting of at most two adjacent words.

Results are reported in tables showing the precision and recall for different sizes of the *training set* (i.e., one, two or more known links for each requirement). Precision and recall were computed for different positions in the ranking list the best one, two, three classes associated to a requirement. The maximum number of classes used as *training set* for each query depends on the average number of classes associated with each requirement: up to five classes for *Albergate*, three for *Transient Meter* and only two for the *Library Management* software.

To imitate the real world process application and to obtain an average estimate of the performance under different conditions (i.e., material included and excluded from the *training set*), experiments were replicated several times (see Section 4.1.2); each table entry corresponds to the mean computed over 100 random experiments; the number of experiment replications was chosen to guarantee a standard deviation of precision and recall below 3%.

Finally, in order to compare different training sets and to evaluate the influence of bigram probability, t-tests (significance level $\alpha = 5\%$) were performed. In the remainder of the chapter, the terms "significantly affect" or "significant improvement" will be used to state that statistical evidence, according to the t-test, was obtained.

4.4.2.1 Choosing the λ'_k Factor

As highlighted in equation (4.3) in Section 4.1.2, performances of the method are influenced by the weight λ'_k associated with the multi-word term probability ($\lambda'_k = 0$ for unigram probability). To reduce the computational load, a preliminary phase to assess the effectiveness and influence of λ'_k was performed.

Analysis performed for different values of λ'_k, varying from zero to one, showed that, on the available case studies, values of λ'_k between 0.3 and 1 did not significantly affect results. For $\lambda'_k < 0.3$, results are very close to those for $\lambda'_k = 0$. Moreover, a more complex model having different λ'_k for different documents i.e., for different

values of k did not significantly influence the results. Hence, for all documents (i.e., all k) results were computed assuming $\lambda'_k = 0$ for unigrams and $\lambda'_k = 1$ for unigrams plus bigrams.

4.4.2.2 Albergate

Table 4.1 reports results obtained with $\lambda'_k = 0$, and by applying the Bayesian classifier once:

- Stop words were removed and morphological analysis was applied to requirements.
- Stop words and normalization were applied to program item names.

The experimental conditions are the same as those giving the best results as presented in Chapter 3; results differ slightly due to fluctuations caused by randomly generating the traceability recovery experiments and due to the experiment organization (training and testing phase).

Score:		**Best 1**	**Best 2**	**Best 3**
1 Class	Precision (%):	33	22	17
Training	Recall (%):	33	45	53
2 Classes	Precision (%):	46	30	22
Training	Recall (%):	46	60	68
3 Classes	Precision (%):	51	32	24
Training	Recall (%):	51	65	72
4 Classes	Precision (%):	52	32	24
Training	Recall (%):	52	64	73
5 Classes	Precision (%):	55	34	24
Training	Recall (%):	55	67	76

Table 4.1: Albergate $\lambda'_k = 0$ traceability results.

The subsequent step (see Table 4.2) considers $\lambda'_k = 1$ corresponding to the situation where both single words and bigrams are modeled.

The comparison between accuracy obtained (see Table 4.2) and the figures of Chapter 3 is very encouraging and supports the newly proposed equations. There is

an increase in precision and recall: Table 4.2 results outperform Table 4.1 due to the bigrams contribution. Detailed comparison with Table 3.3, however, is somehow more difficult, in that the approach is almost completely different; in Chapter 3, results have been obtained using a unique dictionary: program item names are forced to belong to requirements. Moreover, the equations of Chapter 3 cannot be re-estimated once new training material is available. Furthermore, since there is the need for training material, any requirement mapped into a single class (and vice-versa) cannot be used (missing testing material). Table 4.2 results are significantly better than those in Table 3.3, once sufficient training material is available (i.e., two or more links).

Current results improve recall and precision: although using one-class *training set* does not significantly improve the performance, results significantly improve with training sets of two or three classes for the Best 1 and Best 2 ranking scores. However, with respect to the $\lambda'_k = 0$ model, when increasing the training set, performances may deteriorate due to the following factors:

- When considering bigrams, the number of possible terms is upper bounded by twice the number of words in the text documents (i.e., the denominator of the equation (4.13) doubles). In particular, when a single class is available for training, bigram probabilities did not improve the results.

- Adding new traceability links may introduce useless information that deteriorates the *signal to noise* ratio. In other words, when four or five classes are used to train the system, most of the requirements are already mapped to the respective implementing classes. Therefore, the test set includes only a few classes and, furthermore, these classes tend to have terms in common, thus decreasing the discriminant power of the method.

4.4.2.3 Transient Meter

The baseline was established by applying the process with $\lambda'_k = 0$ and results were calculated:

Score:		Best 1	Best 2	Best 3
1 Class	Precision (%):	33	22	20
Training	Recall (%):	33	45	57
2 Classes	Precision (%):	54	32	23
Training	Recall (%):	54	64	69
3 Classes	Precision (%):	57	33	24
Training	Recall (%):	57	66	72
4 Classes	Precision (%):	54	33	24
Training	Recall (%):	54	67	72
5 Classes	Precision (%):	53	32	23
Training	Recall (%):	53	64	70

Table 4.2: Albergate single words plus bigrams traceability recovery results.

1. After morphological analysis (including stop words removals).

2. After removal of automatically generated identifiers.

3. After excluding non-traceable classes.

Table 4.3 results were obtained with an experimental setup corresponding to the first item; at first glance, it appears that traceability recovery is quite poor when compared to results of Table 4.1 and Table 4.2.

Score:		Best 1	Best 2	Best 3
1 Class	Precision (%):	7	8	9
Training	Recall (%):	7	16	26
2 Classes	Precision (%):	4	8	9
Training	Recall (%):	4	16	28
3 Classes	Precision (%):	6	8	11
Training	Recall (%):	6	16	33

Table 4.3: Transient Meter $\lambda'_k = 0$ baseline results.

Differences can be explained in terms of the adopted software development approaches. *Transient Meter* was developed using RAD IDE, COTS, communication middleware, reused code, and other traceability affecting factors described in Section 4.2. On the other hand, *Albergate* was entirely *manually* coded from scratch, without reusing components, but integrating a relational database.

Data reported in Table 4.3 are puzzling since they seem to contradict the learning effect experienced in the *Albergate* case study: adding information decreases traceability. As new training material was added and new links recovered, precision and recall worsened.

The counterintuitive phenomenon happens when the training material added has few or no common words (i.e., identifiers) with the pre-existing *training set* composed of classes previously associated with the given requirement. The situation violates the assumption that knowledge is processed in a consistent way; the new training material (i.e., program item names) adds *noise* rather than useful information. Probability distributions are flattened, causing classes of the *test set* to be easily associated with a wrong requirement.

Although the steps described in Figure 4.2 to process the source code do not always improve precision, each step is an essential component of the traceability recovery process when dealing with COTS, middleware, RAD IDE, etc. As emphasized, automatically generated identifiers need to be pruned to avoid flattening probability distribution with spurious information. Results obtained after this step are shown in Table 4.4.

Score:		**Best 1**	**Best 2**	**Best 3**
1 Class	Precision (%):	5	9	10
Training	Recall (%):	5	18	31
2 Classes	Precision (%):	1	6	11
Training	Recall (%):	1	15	32
3 Classes	Precision (%):	2	9	13
Training	Recall (%):	2	18	40

Table 4.4: Transient Meter: $\lambda'_k = 0$ results after removal of automatic-generated identifiers.

The next step, the normalization phase, helps to associate classes whose identifiers (attributes, methods or comments) share common radixes. However, in presence of automatically generated code and COTS, the normalization tends to add noise, i.e., confounding links. Thus, automatically generated code and COTS need to be removed

to improve retrieval accuracy.

Transient Meter contains classes that cannot be effectively traced into requirements; these classes consist of one splash screen, a window to display aggregated data from the database, and a class for report printing. It was also discovered that, in *Transient Meter*, reused classes have no way to be traced into requirements. Thus, seven classes (two for handling wave files, five for signal processing), which all mapped to a single requirement, were excluded prior to computing new results. Moreover, classes belonging to the CORBA architecture (three classes) except the stub were removed. Finally, some low-level design classes, which cannot be directly mapped to requirements, were identified. A total of four classes, three handling data structures and one implementing an adapter to reused classes, were removed. Thus, the final number of classes traced into requirements was 19 (out of the initial 36).

Results in Table 4.5 were obtained by applying the process of Figure 4.2 which encompasses the described activities accounting for the traceability affecting factors (see Section 4.2). They show that the system significantly learns as the training set increases.

Score:		**Best 1**	**Best 2**	**Best 3**
1 Class	Precision (%):	26	24	20
Training	Recall (%):	26	48	60
2 Classes	Precision (%):	60	38	28
Training	Recall (%):	60	76	83
3 Classes	Precision (%):	71	46	31
Training	Recall (%):	71	92	93

Table 4.5: Transient Meter: $\lambda'_k = 0$ final results after removal of non-traceable classes.

Finally, the analyses corresponding to $\lambda'_k = 1$ were run on the same material used to obtain Table 4.5. In other words, traceability recovery was computed based on the equation (4.3), where bigram terms play a fundamental role; results shown in Table 4.6 confirm the ability of the approach to significantly learn.

Moreover, with respect to the unigrams, there is a significant performance improvement for the top ranking score (Best 1). This means that some classes that

Score:		Best 1	Best 2	Best 3
1 Class	Precision (%):	32	25	21
Training	Recall (%):	32	50	62
2 Classes	Precision (%):	64	39	27
Training	Recall (%):	64	77	82
3 Classes	Precision (%):	77	46	31
Training	Recall (%):	77	92	93

Table 4.6: Transient Meter: single words plus bigrams traceability results.

ranked in second or third position using unigrams, obtained the Best 1 ranking score due to bigram contribution. On the other hand, bigrams did not help too much in correctly ranking more classes than unigrams among the top three scores.

4.4.2.4 Library Management Software

The traceability recovery process followed for this system was the same adopted for *Transient Meter*. In particular, after removing non-traceable files (3 out of 25 files, implementing utility functions, were removed).

Results are shown in Table 4.7, 4.8 and 4.9. The performance improvement after the different steps basically confirms results obtained for the two previous analyzed systems.

Score:		Best 1	Best 2	Best 3
1 File	Precision (%):	26	24	21
Training	Recall (%):	26	47	64
2 Files	Precision (%):	33	26	21
Training	Recall (%):	33	52	62

Table 4.7: Library Management: $\lambda'_k = 0$ results after morphological analysis.

Finally, Table 4.10 reports results obtained while considering bigrams. This system is an example in which considering bigrams did not significantly help (on the contrary, performances for the top score tend to decrease).

Statistical tests showed that bigrams behaved differently for the three systems analyzed. The general lesson to be learned is that the influence of bigrams depends on

Score:		Best 1	Best 2	Best 3
1 File	Precision (%):	33	27	23
Training	Recall (%):	33	55	68
2 Files	Precision (%):	34	32	24
Training	Recall (%):	34	64	72

Table 4.8: Library Management: $\lambda'_k = 0$ results after removing automatic-generated identifiers.

Score:		Best 1	Best 2	Best 3
1 File	Precision (%):	38	29	24
Training	Recall (%):	38	58	72
2 Files	Precision (%):	34	37	27
Training	Recall (%):	34	73	82

Table 4.9: Library Management: $\lambda'_k = 0$ results after removing non-traceable files.

the way requirement terms are mapped to code. If, in most cases, single words from requirements are mapped to code variables (e.g., *Library Management*, the smallest and simplest system), then bigrams confuse the Bayesian classifier. On the other hand, when more adjacent words from requirements are generally associated to variables (e.g., *Transient Meter*), the bigram-based classifier tends to outperform the unigram-based classifier.

Moreover, it has been learned that the maximum number of adjacent words to be considered is limited. In particular, experiments performed on the available systems revealed that training with more than two adjacent words did not improve accuracy.

Attention should also be paid when increasing the *training set* since the amount of multiple-word terms increases not only the useful information, but also the *noise*.

Score:		Best 1	Best 2	Best 3
1 File	Precision (%):	38	30	26
Training	Recall (%):	38	60	77
2 Files	Precision (%):	36	31	27
Training	Recall (%):	36	63	82

Table 4.10: Library Management: single words plus bigrams traceability results.

Finally, as highlighted in Section 4.4.2.1, a conservative choice of $\lambda'_k = 1$ ensures good performances in the case studies presented. The choice of the best λ'_k, i.e., the best weighting factor for the multiple word term probability, is relevant to improving accuracy.

4.5 Chapter Summary

This chapter presented a novel approach based on stochastic language models to recover traceability links between high level documentation, such as functional requirements or use cases, and low level artifacts such as detailed design or source code. The novel method improves the approaches presented in Chapter 3 since it does not require that source code and documentation share a common vocabulary. Furthermore, chapter 4.1 presented a taxonomy of factors such as the presence of automatically generated code, COTS and middleware, which may affect the traceability recovery process. By applying a customized process and thus eliminating the *noise* effect due to COTS, reused code, communication middleware and, more generally, components that are difficult or even impossible to trace into requirements, the recovery accuracy was substantially improved on the available case studies.

Main limitations are not tied to the approach but rather are the same problem and limitation highlighted for the approach presented in Chapter 3.

Reported results were obtained on systems whose representativeness of the industrial practice should be assessed.

As stated in Chapter 3, a second limitation is common to software engineering data sets: training material is often scarce and the adequacy of a probabilistic model can be questioned. A quick comparison between the size of software engineering documentation and the size of natural language processing corpora makes it clear that obtained results suffer from poor training. Thus, system sizes, application domains, and volume of documentation suggest caution before generalizing the obtained accuracy to other software systems.

CHAPTER 5: OO Code to Code Traceability

5.1 Mapping Model

As a system evolves, new functionalities are added and existing ones are removed or modified. Therefore, in OO systems, there will be added and deleted classes as well as classes whose interfaces change or remain unchanged, when the class code undergoes modifications. Added and/or deleted classes may be easily identified e.g., by comparing the interfaces of the classes, whereas modified classes may be more difficult to trace.

The activity of checking the evolution between two software versions can be greatly assisted by automatic tools that help a programmer to identify regions of the software that underwent modifications. Context `diff` between files may be applied to establish similarity between entities belonging to different software releases (files, classes, methods). However, results may be too coarse grained and very difficult to summarize, visualize and interpret.

The same ideas and similarity introduced in Chapter 2 can be adapted to deal with different versions of an OO software system. Given two releases of an OO software system one may be thought of as the evolution of the other; thus the *oldest* release plays the role of the *design* while the newest is considered the *implementation*. In other words, the following steps are carried out:

- Recover the "as is" design representation of the two releases (in AOL).
- Compute similarities between the two AOL representations.

- Build the mapping between the releases (e.g., added, deleted, and modified classes and methods).

5.1.1 Mapping Equations

Let $\sigma'(p,q)$ denote the similarity between properties $p \in P(x)$ and $q \in P(y)$. As introduced in Chapter 2, a bijective function $m_{X,Y} : P'(x) \rightarrow P'(y)$, which maps each property in $P'(x) \subseteq P(x)$ onto a property of $P'(y) \subseteq P(y)$, can be derived from σ'. Thus, the similarity between classes is defined as:

$$\overline{\sigma}(X,Y) = 1/|P'(x)| \sum_{p \in P'(x)} \sigma'(p, m_{X,Y}(p)) \tag{5.1}$$

Let V_i and V_j $(i \neq j)$ be two releases of a software system. Based on the similarity function $\overline{\sigma}$, a mapping function $m'_{V_i,V_j} : V'_i \rightarrow V'_j$ can be constructed that associates classes in $V'_i \subseteq V_i$ with classes in $V'_j \subseteq V_j$. Classes belonging to $V_i - V'_i$ are deleted classes, classes in $V_j - V'_j$ are added classes, while each class $C_{i,k} \in V'_i$ evolves into class $m'_{V_i,V_j}(C_{i,k}) \in V'_j$. For any given class $C_{i,k} =< c_{i,k}, P(c_{i,k}) >$ in V_i and any given class $C_{j,l} =< c_{j,l}, P(c_{j,l}) >$ in V_j the similarity between individual properties is defined as follows:

$$\sigma'(x,y) = \lambda_c * \sigma(c_{i,k}, c_{j,l}) + (1 - \lambda_c) * \sigma(x,y) \tag{5.2}$$

where $x \in P(c_{i,k})$, $y \in P(c_{j,l})$, $\lambda_c \in [0,1]$ is the weight associated with class name matching, and $\sigma(u,v)$ is the complemented edit distance defined by equation (1.10).

The matching problem between the two releases V_i and V_j can be represented as a graph $G = (V,E)$. The nodes V are the classes $C_{i,k}$ and $C_{j,l}$ belonging to V_i and V_j respectively. Edges E connect each class in V_i to each class in V_j and are weighted by the similarity of the connected classes (i.e., $\sigma'(C_{i,k}, C_{j,l})$). Thus if $|C_i|$ and $|C_j|$ are the number of classes of V_i and V_j, respectively, a bipartite graph with $|C_i| + |C_j|$ nodes and $|C_i| \times |C_j|$ weighted edges is constructed. The optimum match $m_{C_{i,k},C_{j,l}}$ between $C_{i,k}$ and $C_{j,l}$ is inferred via the maximum match algorithm (Cormen et al.

1990) to a bipartite graph as in Chapter 2. The similarity between $C_{i,k}$ and $C_{j,l}$ is then defined as the average optimum weight (equation 5.1) between properties as computed by the maximum match algorithm.

5.2 Traceability Recovery Process

The overall method to recover traceability links is represented in Figure 5.1. It consists of the following activities:

1. AOL Representation Extraction: an AOL system representation is recovered from code through a `Code2AOL` extractor.

2. Software Metrics Extraction: class level and function level software metrics are computed.

3. AOL Parsing and Similarity Computation: the similarity $\overline{\sigma}(X, Y)$ is computed for any given class X in release V_i and any given class Y in release V_j.

4. Release Comparison: an optimum match is computed between classes of releases V_i and V_j (i.e., inference of function m'_{V_i,V_j}) by means of a maximum matching algorithm (Cormen et al. 1990).

5. Difference Computation: code of corresponding classes and methods is compared to identify added, deleted, and modified LOC.

The following subsections highlight the key issues of these activities and the related implications.

5.2.1 Software Metric Extraction

Software metrics are extremely appealing when a large software system has to be assessed and no a-priori documentation and/or information are available. Several papers (Henry and Kafura 1984; Chidamber and Kemerer 1994; Fenton 1994; Pearse and Oman 1995; Daly et al. 1995; Mayrand and Coallier 1996) and books (Moller

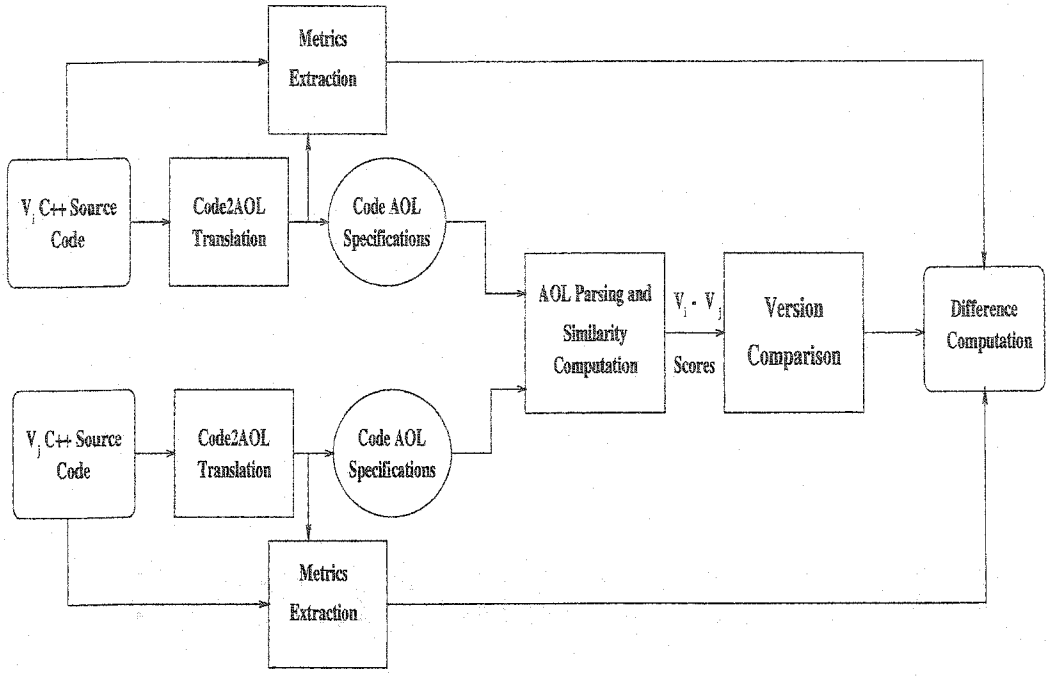

Figure 5.1: Traceability link recovery method.

and Paulish 1993; Lorenz and Kidd 1994) investigated software metrics attempting to draw conclusions on the relation between measured values and software characteristics (e.g., reliability, testability, maintainability, etc) with special focus on to quality issues (Basili et al. 1996; Mayrand and Coallier 1996).

From an estimate of the impacted classes during maintenance, software metrics are used to build models that predict the size of changes in terms of added or modified LOC.

Moreover, software metrics can be used to enforce coding standards, to evaluate code complexity and sizes, and refine traceability links.

5.2.2 AOL Parsing and Similarity Computation

AOL parsing and similarity computation is a two step process. The first step parses the AOL representation of two software releases and assigns weights to each class property. The second step computes an optimum match between the given classes.

5.2.3 Release Comparison

Similarities among classes tend to be significant over a given threshold that may depend on the subject system. Thus, the concept of a pruning threshold is introduced to remove edges that are unlikely to represent a real mapping between classes. Removing these edges may produce a graph with isolated nodes which represent either classes deleted from the old release or classes added to the new one. Finally, on the pruned bipartite graph, the maximum match algorithm is applied to induce the mapping function m'_{V_i,V_j} between V_i and V_j. The pairs of nodes resulting from this mapping represent classes in common in the two releases i.e., classes which evolved from the old to the new release.

Similarity is measured on the basis of the string matching between class names, attributes and their types, and method signatures. The consequence is that, even if two classes obtained a 100% similarity, there is no guarantee that no modifications occurred.

5.2.4 Code Difference Computation

The final step of the method is to identify changes at the statement level. However, the aim is to mask minor changes. For example, if comments are added to a chunk of code or the code is indented to increase readability it would be desirable to hide by default such a detail unless explicitly required.

This is achieved in the difference computation step, in which the differences in the code (excluding comments) of corresponding methods are identified. Differences are summarized in terms of number of added, deleted, and modified LOC. To compute the overall difference between two releases, these numbers are augmented with the number of LOC of added/deleted classes and methods.

5.2.5 Tool Support

The traceability link recovery method shown in Figure 5.1 has been completely automated for the C++ language. As a preliminary step, source code undergoes

a double preprocessing. First, the gnu `cpp` preprocessor expands include directives and macro definitions (retaining comments); then, a custom program removes system includes and comments maintaining the correct line numbering. Finally, class bodies and method bodies are saved into distinct files.

A `Code2AOL Extractor` module was developed to extract the AOL representation from code; more details can be found in Section 1.3.1.2.

The class metrics extracted are: the number of public, private and protected attributes and methods, the number of direct subclasses, the number of direct superclasses, etc. Function level metrics include cyclomatic complexity, number of statements, number of passed parameters, number of operators, number of function calls, and return points.

The matching phase relies on an AOL parser. Once the abstract syntax trees of the compared releases are available, they are traversed and similarities are computed. They are then used to build a bipartite graph, which is passed to the maximum match algorithm. The AOL parser and the edit distance computation were implemented in C, while the maximum match algorithm is written in C++.

Finally, given the recovered mapping, a script computes the differences in the code of corresponding pairs of methods. For each pair the GNU tool `diff` is exploited to compare method bodies. The result is summarized as the number of added, deleted, and modified LOC; the number of added or deleted LOC is augmented with LOC of added or deleted classes and methods computed by the metric extractor.

5.3 Experimental Results

The traceability link recovery method was tested on with two freely available C++ systems: the `LEDA` library and the `DDD` debugger. In both cases, only the OO code was analyzed while procedural parts were disregarded. In both cases the procedural part was limited to a small fraction of the overall size.

5.3.1 Test Suite

LEDA was used to assess the parameters of the process (λ_c and the pruning threshold), which were further validated on the source code of different releases of DDD. DDD is a graphical user interface to GDB and DBX, which are popular UNIX debuggers. DDD was developed and maintained by Andreas Zeller at the Technische Universität Braunschweig, Germany; it is freely available and can be downloaded from several sites (e.g., http://www.cs.tu-bs.de/softech/ddd/). In particular, 31 different releases, ranging from release 2.0beta1 to release 3.1.3 were analyzed.

DDD seems to have evolved smoothly, since from the first to the last release, its source code doubled from 52 to 107 KLOC of C++ and the number of classes only changed from 123 in the first release analyzed to 135 in the last release.

The main DDD site stores and gives access to almost the entire DDD evolution; only the oldest releases are no longer available. On the other hand, it was not possible to access those LEDA releases which were not currently distributed e.g., releases 3.6.x.

More details on LEDA and DDD can be found in the appendices.

5.3.2 Setting Preliminary Weights

The first step of the study consisted of devising a heuristic for the identification of a pruning threshold on the similarity measures to remove edges that are unlikely to represent a real mapping between classes.

λ_c	uncertainty range (u.r.)	% pairs in u.r.	threshold	error rate
30%	54%-78%	11%	70%	2%
50%	42%-75%	13%	50%	4%
70%	30%-79%	16%	-	-

Table 5.1: Weights used to compare the different releases of LEDA and result summary.

For this purpose, consecutive releases of LEDA were compared by assigning different weights to class name matching and to property matching. A first comparison was

conducted by assigning the same weight ($\lambda_c = 50\%$) to both. The result was quite interesting: in most cases less than 6% of pairs had a similarity value lower than 90%.

5.3.3 Assessing the Similarity Value Threshold

The second step of the study consisted of analyzing source code to assess the findings of the method. Code analysis was performed on the small subset of pairs with a similarity value lower than 90%. Indeed, because of the use of the weight $\lambda_c = 50\%$ in the string matching, all pairs with a similarity measure greater than 90% showed a high degree of similarity both in class name side and in property. They were then considered as a case of evolution or, sometimes, unchanged classes.

To discard false positives, three software engineers independently analyzed the pairs with a similarity value lower than 90%, i.e., pairs of classes in which one is not one the evolution of the other. A group consensus approach was taken to solve the cases where the three software engineers disagreed.

Release	3.0	3.1.2	3.2.3	3.4	3.4.1	3.4.2	3.5.2	3.7.1
Added #	47	70	10	40	4	4	35	178
Added LOC	1894	4946	363	4193	200	148	2941	20679
Deleted #	7	3	3	10	0	4	9	1
Deleted LOC	334	113	95	372	0	173	307	63
Modified #	61	91	52	73	20	56	62	27
old LOC	18551	21019	16874	20367	10605	34853	37734	12859
new LOC	19668	22699	17772	23716	10951	36283	41669	12456
Unchanged #	2	16	122	96	189	152	140	208
old LOC	185	4029	25262	21386	53238	30164	28149	69025
new LOC	184	4071	25460	22136	53916	30309	28072	60923
Similarity value	89%	94%	96%	95%	99%	98%	96%	98%

Table 5.2: Results of comparing LEDA releases: added, deleted, modified, unchanged LOC and classes.

This manual inspection showed that only pairs with a similarity value higher than 75% were always to be considered as cases of class evolution and only pairs with a similarity value lower than 42% were always false positives. This means that there was a quite large range of uncertainty and most of the analyzed pairs were in this

range. However, the results also showed that a threshold 50% discriminates false matchings. Class pairs with a similarity values below 50% are unlikely to represent a real mapping; this threshold in the worst case, which was comparing releases 3.0 and 3.1.2, gave an error rate lower than 4%, but error rate was usually lower than 1%.

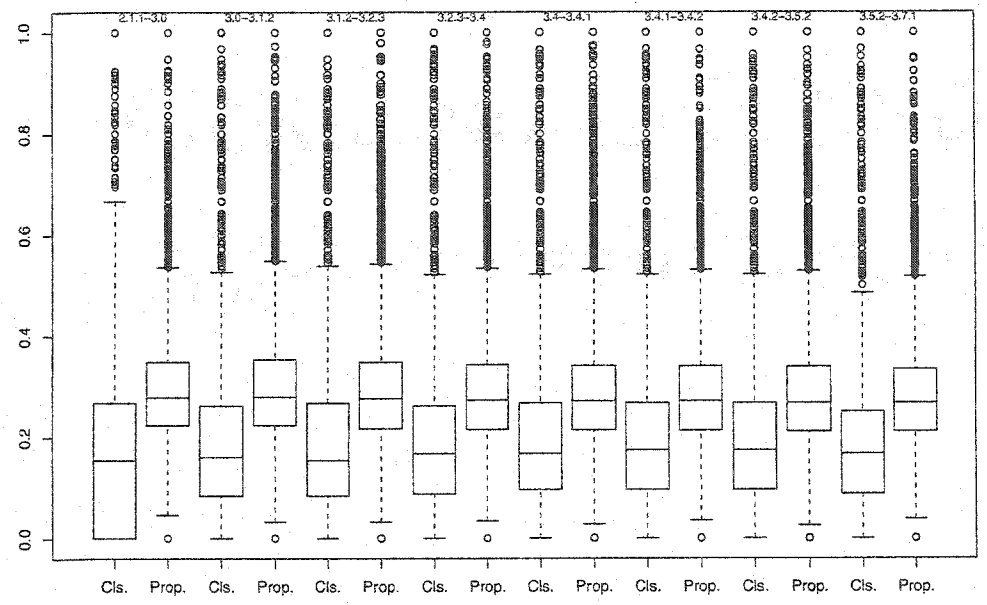

Figure 5.2: Distributions of class names and properties σ values for the LEDA releases.

5.3.4 Assessing Weights and Thresholds

Although the obtained results were quite encouraging, it was decided to compare the releases of LEDA using different weights for class name and property matching. The objective was to assess the thresholds on the similarity values with respect to the adopted weights. Table 5.1 shows the different values used for λ_c together with the corresponding uncertainty range and the percentage of pairs falling in this range in the worst case; the table also shows the threshold adopted with the different values of λ_c and the corresponding error rate in the worst case.

The results showed that, in the case of the analyzed software releases, the pairs

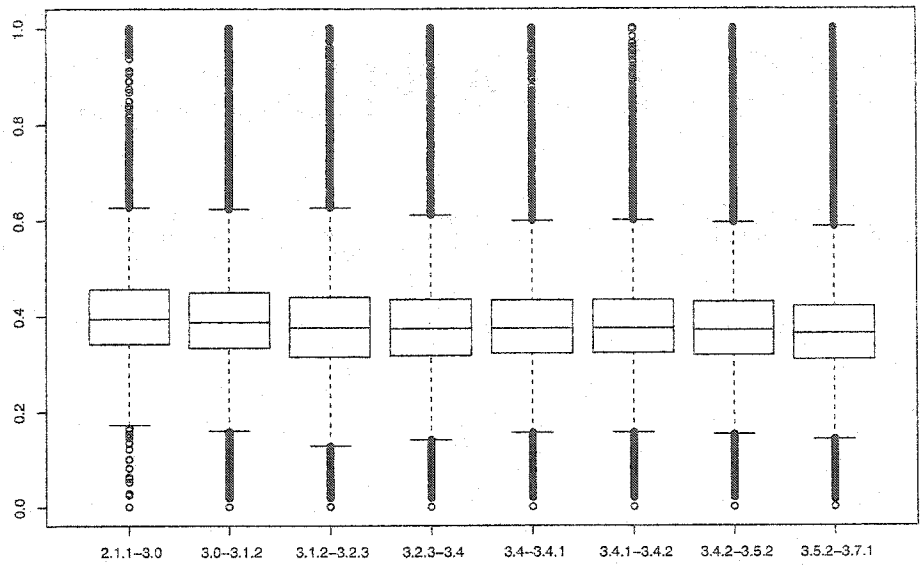

Figure 5.3: Distributions of the $\overline{\sigma}$ values for the LEDA releases with $\lambda_c = 30\%$

produced by the matching tool are almost the same, regardless of the weights. In the worst case, matching between releases 2.1.1 and 3.0, the pairs obtained by using different weights differed in less than 6% of cases. Changing λ_c usually affected less than 3% of matches. Furthermore, the similarity values of these pairs were generally low.

However, the similarity values associated with pairs in the uncertainty range (42%-75%) were in most cases significantly different. The best results were obtained with a lower weight for the class name matching ($\lambda_c = 30\%$). In this case, the uncertainty range was 54%-78%. Moreover, the threshold 70% was able to discriminate between similar and different pairs, with a lower error rate (less than 2% in the worst case). Conversely, giving a higher weight to class name matching ($\lambda_c = 70\%$) did not produce good results: the uncertainty range was larger (30%-79%) and it was not possible to identify a threshold giving performance comparable to the previous cases. The main reason for this behavior was the significant number of classes in the different releases

that changed their name, without changing significantly their properties.

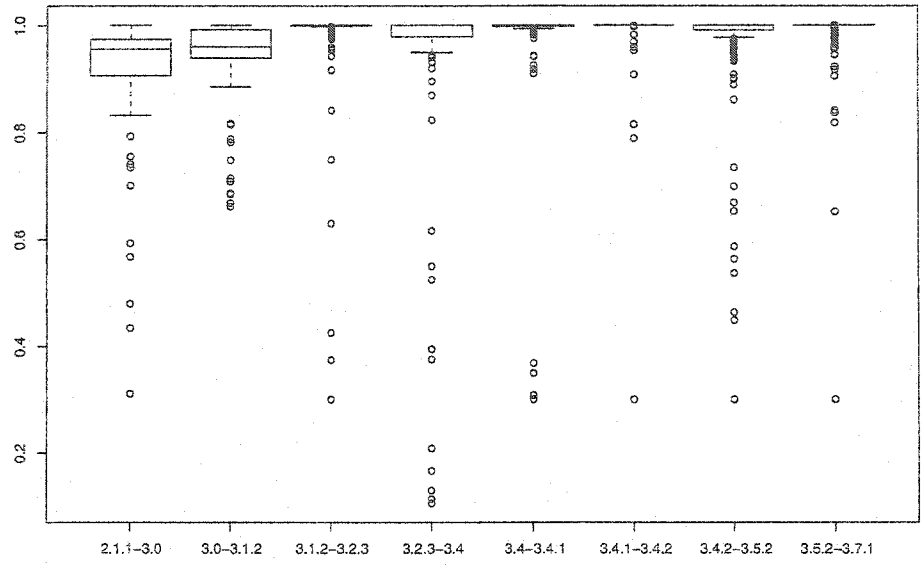

Figure 5.4: Distributions of the $\overline{\sigma}$ values for the LEDA releases ($\lambda_c = 30\%$) computed by the Maximum-Match Algorithm

Table 5.2 shows the results of comparing the different releases of LEDA using the weight 30% for class name matching: the first column contains the comparison of releases 2.1.1 and 3.0. The remaining columns contain the comparison of the current column (new release) with the previous one (old release). In particular, for each release, the table outlines the number of classes and their total size (in terms of LOC) added, deleted, modified, and unchanged (in the interface) with respect to the previous release. Classes in the new or old release are considered added or deleted if they do not match any class in the old or new release, or if they match some class with similarity value lower than 70%. Classes are considered unchanged in the interface if they match some class in the previous release with a similarity value of 100%.

5.3.5 Analysis of the Distributions of Similarity Values

Figure 5.2 shows the box plots of the distributions of the complemented edit distances (σ values of equation (1.10)) of class names (Cls.) and properties (Prop.) for LEDA consecutive releases. Each class name's σ values are obtained by comparing each class name of a LEDA release with all class names of the subsequent release. Similarly, the distributions of the class σ values are obtained by comparing each class property of a LEDA release with all class properties of the subsequent release. Therefore, these values are independent of the λ_c weight used in the equation to compute the σ' values.

For each graphic, the upper and lower ends of the rectangles are located at the lower and upper quartiles of the data, respectively. The dotted "whiskers" represent the upper and lower extremes of the distributions, by excluding outliers, which are indicated by empty circles. By default, anything over 1.5 times the inter-quartile range is considered an outlier. A line crossing the box locates the median of the distribution. Figure 5.2 indicates that the overwhelming majority of class and property similarity values are below 50%; this means that the σ values that more likely correspond to class or property evolution are the outlier of the distributions. This confirms the finding that a threshold 50% is largely able to discriminate false positives in the case $\lambda_c = 50\%$.

The distributions of the σ values are reflected in the distributions of the $\overline{\sigma}$ values as shown in Figure 5.3, for the case of $\lambda_c = 30\%$. The similarity values of pairs of classes belonging to the traceability relation are likely to be located in the outliers of the upper part of the distribution (more than 60% similarity value).

Figure 5.4 shows the distributions of $\overline{\sigma}$ values after the maximum match algorithm was applied. In this case, the data suggests that false matchings most likely belong to distribution outliers i.e., similarity values below 80-90%. Figures 5.3 and 5.4 support the finding that a threshold 70% reliably discriminate false positives when $\lambda_c = 30\%$.

5.3.6 DDD Validation

The final step of the study consisted of applying traceability recovery to 30 DDD available releases, using the best set of parameters achieved on the LEDA study which were 30% as weight for class name matching and 70% as threshold on the pair similarity values. The goal of this second analysis was to demonstrate the applicability of the process and the accuracy of the inferred parameters on a different software system.

Table 5.3 shows the results of comparing the different DDD releases of Table E. These results are representative of the overall DDD history. On the average, the similarity values between subsequent releases are higher, as the major changes are included in the releases shown in the table.

Release	2.0	2.1b1	2.1.90	2.2.3	2.99	3.0.90	3.0.91	3.1.3
Previous Release	2.0beta3	2.0	2.1.1	2.2.2	2.2.3	3.0	3.0.90	3.1.2
Added Classes	0	2	4	0	2	4	1	0
Added LOC	0	486	139	0	890	103	26	0
Deleted Classes	0	0	0	2	0	0	0	0
Deleted LOC	0	0	0	1293	0	0	0	0
Modified Classes	3	6	11	59	10	36	3	0
old LOC	4294	10443	13901	9951	17313	28916	6018	0
new LOC	4463	11511	16206	9715	20806	34488	6070	0
Unchanged Classes	120	117	115	69	118	94	131	135
old LOC	30874	24929	24113	33057	25375	18568	47897	54839
new LOC	30909	25816	24883	32973	25397	19064	47925	54826
similarity value	0.99	0.99	0.99	0.92	0.99	0.97	0.99	1

Table 5.3: Results of comparing DDD relevant releases: added, deleted, modified, unchanged LOC and classes.

The results were encouraging, since manual analysis of the code revealed that added, deleted and modified classes were correctly classified i.e., the error rate for this system was almost null.

5.3.7 Estimating the Size of Changes

Cost and effort prediction is an important aspect of the management of software projects. Experience shows that accurate prediction is difficult: an average error of 100% may be considered "good" and an average error of 32% "outstanding" (Vicinanza et al. 1991). Most methods for predicting effort require an estimate of the size of the software. Once it is available, models can be used that relate size to effort.

Cost estimation is not a one-time activity performed in early phases, but rather estimates should be refined continually throughout a project (DeMarco 1982). Furthermore, estimates also have to be provided for post-release maintenance activities. Thus, it is necessary to repeatedly predict size throughout the entire software life-cycle.

Most research on size prediction has dealt with traditional applications and traditional software development practices. Few methods have been proposed for OO software development. The results of the traceability process can be used to build models to predict the size of changes in terms of added and modified LOC from an estimate of the number of impacted classes. LOC were measured as the number of non-blank lines, excluding comments and pre-processor directives.

5.3.7.1 Empirical Data Analysis Approach

Several regression techniques were considered to model the relationships between the size of the changes and metrics about the evolution of OO entities, such as the number of classes with modified interface.

A leave-one-out cross-validation procedure (Stone 1974) was used to measure model performance. Each given model was trained on $(n-1)$ points of the data set L (sample size was $n = 30$ for the DDD study) and accuracy was tested on the withheld datum. This step was repeated for each point in L and accuracy measures averaged over n. This method gives an estimate of future performance on novel data and it is thus recommended in the design of predictive models. Moreover, it enables comparisons among different families of models, different choices of parameters, or

data preprocessing. Here, the model error was estimated as the cross-validation version of the *normalized mean squared error* (NMSE), which is the mean squared error normalized over the variance of the sample. Let y_k be a data point belonging to a set of observations of the dependent variable Y and $\hat{y}_k$ be its estimate:

$$NMSE = \frac{\sum_{k \in L}(y_k - \hat{y}_k)^2}{\sum_{k \in L}(y_k - \mu_y)^2} \tag{5.3}$$

where $\mu_y = mean(Y)$ is the mean of the observed values in the sample L. To assess the accuracy, the cross-validation estimates of the sample standard error, s, the residuals $y_k - \hat{y}_k$ and the r-squared R^2 of the fit were also computed. The size of the database suggested the use of models with a reduced number of free parameters. Multivariate linear models:

$$Y = b_0 + b_1 X_1 + \ldots + b_n X_n \tag{5.4}$$

with n at most 2 were therefore considered. Moreover, the use of resistant regression techniques was investigated to handle non-obvious outliers and extreme points.

5.3.7.2 DDD Results

A preliminary set of experiments were performed to model the size of modified code (i.e., added and modified LOC) by means of number of added classes, modified and added, or modified classes. These variables can be estimated from the impact analysis of the change request.

A point of concern is whether an intercept term b_0 should be included in the model. It may be reasonable to suppose the existence of support code which is not directly related to the modifications being counted. However, there is no statistical evidence to believe that on DDD data an intercept value is needed (intercept p-value 0.2, 0.258, 0.4). It will therefore be disregarded in the following.

As shown in Table 5.4, a multivariate model accounting for classes with modified interface (b_1) and added classes (b_2) seems to better explain the data.

Model	b_1 (b_2)	p-value	R^2
Added Classes	1815	6.0e-12	0.80
Modified Classes	309	4.4e-11	0.77
Modified and Added Classes	156 (1075)	2.628e-11	0.86

Table 5.4: DDD added classes, modified classes and multivariate model parameters.

To assess prediction error on future observations, the three models were compared with cross validation. Table 5.5 reports cross validation results: best predictive capability is achieved when the multivariate model is considered. Table 5.5 also shows the $\overline{RSE}$, the mean residual square error, i.e., the total squared difference between the predictions and the observations for the given model averaged over the number of observations. The assumption of a Gaussian distribution for the dependent variable, is not theoretically correct, as the size of a change cannot be negative. However, data of Table 5.5 obtained with this assumption clearly demonstrates that the multivariate model is preferable.

The results summarized in Table 5.5 are encouraging, since with two independent variables an NMSE of 28% was obtained, meaning that the square error variance was less than half of the sample variance. From another point of view, the model based on the added and modified (in the interface) classes achieves a cross validation average error of 86 %, which can be considered good (Vicinanza et al. 1991).

Model	NMSE	$s(error)$	$\overline{RSE}$	$\overline{R}^2$
Added Classes	0.40	1456	2110899	0.81
Modified Classes	0.35	1379	1864648	0.77
Added and Modified Classes	0.28	1225	1482992	0.87

Table 5.5: DDD added classes, modified classes and multivariate model cross validation performances.

Figures in Table 5.5 were compared with models based on the number of classes modified both in the interface and in the implementation. The results obtained were consistently poorer than those obtained from the models based on the number of classes with modified interface. This can be explained considering that, for the DDD

study, the number of added LOC is consistently higher than the number of modified LOC. Indeed, changes in the interface of a class likely induce modifications in its body and in other classes.

Model	b_0(p-value)	b_1 (b_2)	Model R^2
Methods	-364 (0.00309)	26	0.9558
Methods and Added Classes	-318 (0.00586)	23 (306)	0.9617

Table 5.6: DDD added classes and modified methods model parameters.

Robust regression techniques were also investigated to handle non-obvious outliers. The applied robust fit uses Huber's M-estimator and it initially uses the median absolute deviation scale estimate based on the residuals. The estimates obtained for the most promising model, i.e., the multivariate model without intercept, improved (NMSE of 24 %, $s(error)$ 1136.132), in support of the hypothesis that influential points and/our outliers may actually be included in the data set.

Model	NMSE	$s(error)$	$\overline{R}^2$
Modified Methods	0.05	532	0.96
Added Classes and Modified Methods	0.05	546	0.96

Table 5.7: DDD added classes, modified methods cross validation performances.

Finally, as shown in Table 5.6, analysis which considered the number of modified methods as independent variable, was performed. A simple model based only on the number of modified methods outperformed the previous one. It was further improved if a multivariate model including added classes was considered. Notice that, for these models the intercept is statistically significant, as shown in Table 5.6. The models' p-value is not reported in the table because it is zero. Cross validation (see Table 5.7) clearly demonstrates that a simple model suffices because extremely good predictions can be obtained based only on the number of modified methods. Unfortunately, a prediction of the number of methods impacted by a maintenance request is much

more difficult to obtain than the number of impacted classes.

5.4 Chapter Summary

This chapter presented an original extension of the ideas introduced in Chapter 5 to model the evolution and the recovery of traceability links between subsequent releases of an OO software system.

Different experiments were carried out while differently weighting the class name and the names of attributes and methods. On the available case studies, the highest traceability recovery accuracy was obtained when the class name has a lower weight (i.e., 30%) than that of properties names. Clearly, other software systems may or may not attain the optimal misclassification error with the same wights and, in general, a threshold calibration will be required.

Since the *as is design* was recovered from the C++ code, more fine grained information was available. Thus, information besides the class interface matching was used to build a model which predicted the system evolution in terms of added and modifies LOCs. The number of added and modified classes was chosen as independent variables. In fact, experienced programmers should not have difficulties to predict the expected number of added and modified classes involved in an evolution task.

Besides the traceability recovery approach, the other main contribution of the chapter is the proof of feasibility that an accurate and reliable design can be reverse engineered from C++ source code. The chapter's main limitations are the same as those outlined for methods and technologies in Chapter 2. Bipartite graph matching is linear in the graph size. The main limitation comes from the quadratic complexity of computing a similarity measure for each pair of classes in the old and new releases. Foreseeable improvements are in the area of approximate matching to limit the number of required comparisons or the definition of heuristics to reduce the time complexity for very large software systems. Finally, the similarity computation can be easily parallelized; thus , if necessary, the similarity evaluation can be split over a number of machines.

CHAPTER 6: Large Procedural Systems

6.1 Mapping Model

Large multi-platform software systems are likely to encompass a variety of programming languages, coding styles, idioms and hardware-dependent code. Analyzing multi-platform source code is a challenging task since assembler code is often mixed with high-level programming language. Furthermore, scripting languages, configuration files, and hardware specific resources are typically used.

Systems were often originally conceived as a single platform application, with a limited number of functionalities and supported devices. They subsequently evolved by adding new functionalities and were ported on new product families. In other words, new devices and target platforms were added. When writing a device driver or porting an existing application on a new processor, developers may decide to copy an entire working subsystem and then modify the code to deal with the new hardware. This technique ensures that their work will not have any unplanned effect on the original piece of code they have just copied. However, this practice promotes the appearance of duplicated code fragments, also called *clones*.

When a large multi-million LOCs system evolves, recovering and presenting to developers the detailed traceabilty recovery mapping between subsequent releases may easily end up in a situation of information overflow. Many publications have proposed various ways of identifying a very detailed mapping by identifying similar code fragments and/or components in a software system (McCabe 1990; Johnson. 1993; Buss et al. 1994; Baker. 1995; Kontogiannis et al. 1996; Mayrand et al. 1996;

Baxter et al. 1998). However, the gathered information accounts for local similarities and changes; as a result the overall picture describing the macro system changes are difficult to obtain. Moreover, if chunks of code migrate via copy and remove or cut-and-paste among modules or sub-systems, the code may not be easily distinguished from freshly developed one.

As a software system evolves, new code fragments are added, and certain parts are deleted, modified or remain unchanged. The overall evolution is difficult to represent with fine grained similarity measures such as the number of functions, files or lines of code. For example, the decision to restructure a large software system should be supported by high level information which represents abstraction and a summary of system overall picture.

6.1.1 Mapping Equations

The characterization of software system evolution, with a limited number of parameters, has been grounded on code-level similarity, and more precisely, on techniques that detect function duplication among code fragments. Intuitively, the larger the fraction of code fragments shared by two systems, the higher their similarity, while two completely different systems are likely to have a very low similarity and share very few source code fragments.

Given two different software releases, systems, or sub-systems, say A and B, high level information about the extent of their similarity is computed starting from the definition of a similarity between functions. In the following, two functions or code fragments are considered similar if they exhibit exactly the same values of a set of metrics describing them. This definition of similarity was first proposed in (Mayrand et al. 1996).

Let $\mathbf{M}_f =< m_1(f), \ldots, m_n(f) >$ be the tuple of metrics characterizing a function f, belonging to A; $m_i(f)$ $(i = 1 \ldots n)$ stands for the i-th software metric, listed in Section 1.3.1.1 and chosen to describe f, and n is equal to the number of metrics describing the function.

For any given function f, $f \in A$, let $C_f(A,B)$ be the f *cluster*. $C_f(A,B)$ is the subset of a function belonging to B, which exhibits metric values identical to $f \in A$:

$$C_f(A,B) \stackrel{\text{def}}{=} \{g | g \in B \ \wedge f \in A \ \wedge \\ m_i(f) = m_i(g) \ \wedge \ i = 1 \ldots n \\ \wedge m_i(f) \in \mathrm{M}_f\}$$

The above equation defines a cluster, which is a point in a multidimensional software metric space and which represents indistinguishable functions. The above definition may be changed, for example, by imposing that the differences among functions do not exceed a set of thresholds, thus, representing an ellipsis.

The cluster collection $\mathcal{C}_A(B) = \{C_f(A,B) | f \in A\}$ contains a cluster for each function $f \in A$, that is it defines a detailed mapping between two artifacts. Based on this detailed mapping, a set of four high level metrics capturing the high level similarity and evolution are defined.

Let $|S|$ be the number of elements in the set S. High level similarity between A and B is captured by the Common Ratio (CR) between A and B, which is the ratio of the number of functions belonging to A, having $|C_f(A,B)| \neq 0$, with respect to the number of functions contained in A. Formally:

$$CR(A,B) = \frac{|\{f | \ (\ f \in A \ ) \wedge (\ |C_f(A,B)| \neq 0 \ ) \ \}|}{|\mathcal{C}_A|}$$

where $|\mathcal{C}_A|$ is the cluster cardinality i.e., the number of functions in A. The common ratio may be thought of as the percentage of functions in A having similar functions in B.

In a software system, even between two subsequent snapshots, it is not uncommon for code fragments to disappear. The Average Single Cluster Size (ASCS) is the average size of fragment clusters in A not having a *counterpart* in B:

$$ASCS(A,B) = \frac{\sum_{f \in A \wedge |C_f(A,B)|=0} |C_f(A,A)|}{|\{f \in A \wedge |C_f(A,B)| = 0\}|}$$

Conversely, it may be useful to define a measure of the fraction of the unchanged code. The Average Common Cluster Size (ACCS) represents the average A's cluster dimension for functions having at least one similar function in B:

$$ACCS(A,B) = \frac{\sum_{f \in A \wedge |C_f(A,B)| \neq 0} |C_f(A,A)|}{|\{f \in A \wedge |C_f(A,B)| \neq 0\}|}$$

A coarser view is obtained by considering the average cluster size i.e., the average number of functions in a cluster, regardless of any condition also expressed in the following Overall AVerage (OAV):

$$OAV(A,B) = \frac{1}{|\mathcal{C}_A|} \sum_{f \in A} |C_f(A,B)|$$

OAV and CR may be computed with respect to the same system i.e., $OAV(A,A)$, $CR(A,A)$. In this case they represent internal code duplication. They are not symmetric because functions may be added or deleted when passing from A to B.

6.2 Experimental Results

The Linux kernel, a system consisting of million lines of codes, was used as case study. About 400 releases, 365 from 1.0 to 2.4.0 and sub-releases 2.4.x up to 2.4.18, were analyzed to assess the process of quantifying similarity at the system level. Code-level similarity was defined at the function level, i.e., procedures/functions were considered the elementary code fragments, and computed using a *metric-based* approach (Mayrand and Coallier 1996).

The process to compute the similarity consists of the following, subsequent phases:

1. Handling of preprocessor directives;

2. Function identification;

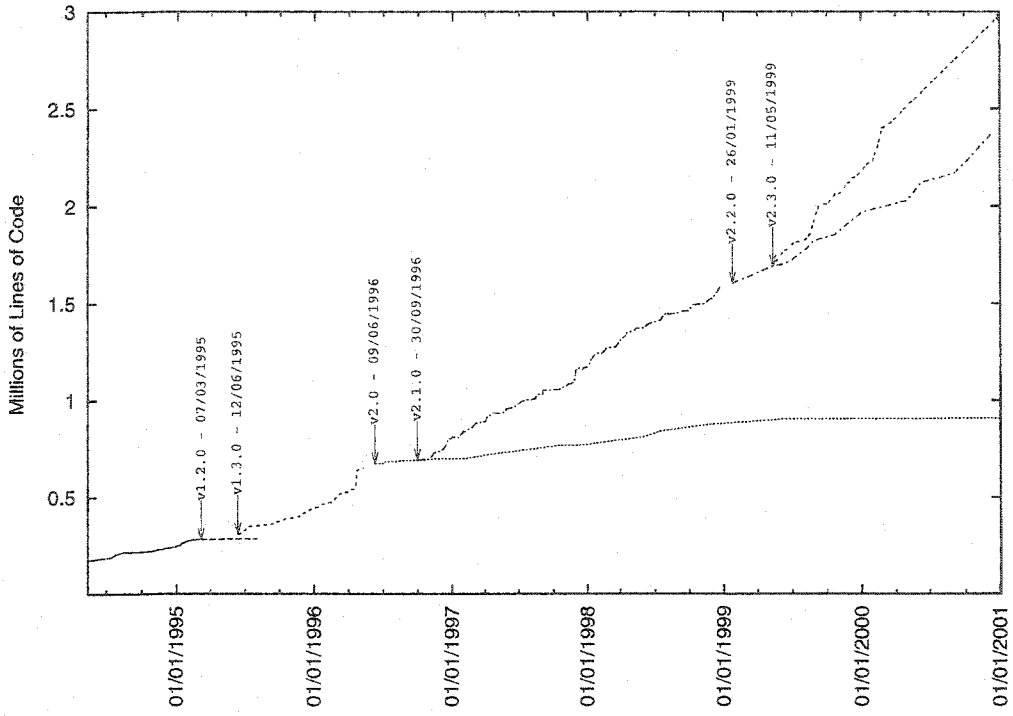

Figure 6.1: Linux size evolution.

3. Metrics extraction; and

4. Cluster identification and computation of the cloning ratio.

Metrics extraction can be performed in linear complexity with respect to system size. However, since the metric extractor used was not optimized, the extraction of metrics for each Linux release required about one hour on a Pentium III (850Mhz 128 Mbytes RAM).

Once metrics were available, clone detection was performed. Clone detection i.e., clustering has $O(n^2)$ worst case complexity, where n is the number of functions. The entire process required about one day for all the Linux releases. Details on the issues related to analyzing software developed with the C language, e.g., handling preprocessor directives, were discussed in Section 1.3.1.

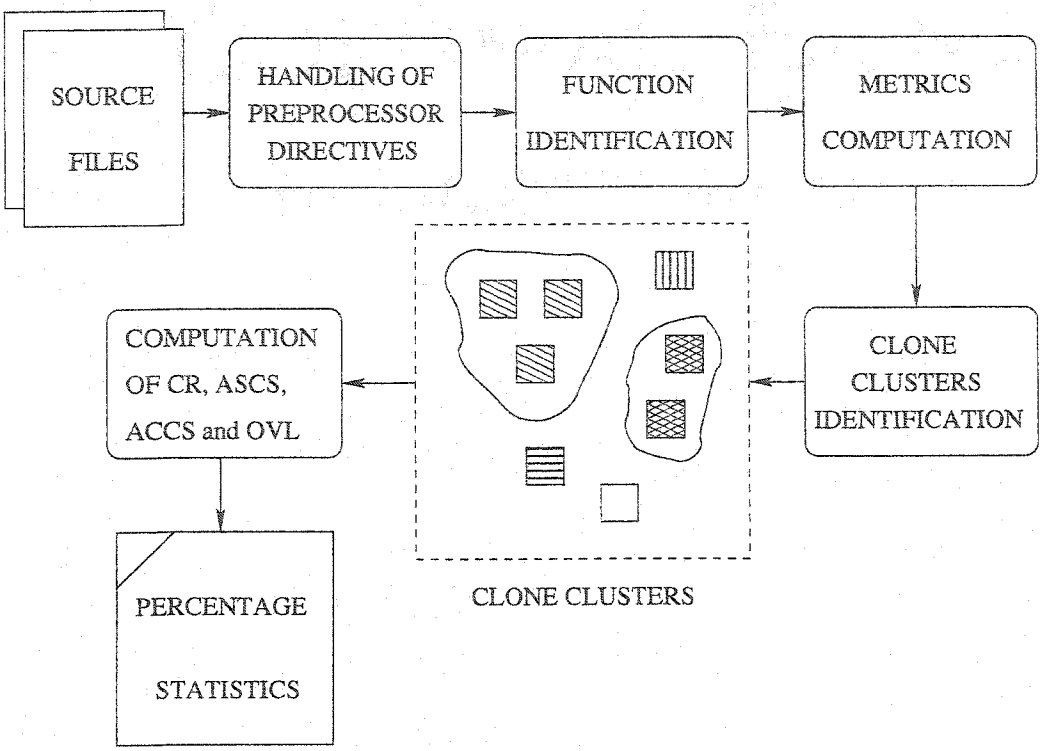

Figure 6.2: Similarity metrics computation process.

6.2.1 Test Suite

Linux is a Unix-like operating system that was created in Finland by Linus Torvalds (Torvalds 1999) when he was a student. The first Linux version, 0.01, was released in 1991. Since then, the system has been developed by many people collaborating over the Internet with Torvalds' support. In 1994 version 1.0 of the Linux Kernel was released, and in January 2001 version 2.4 was released.

Version 1.0 had about 175,000 lines of code. Linux version 2.0, released in June 1996, had about 780,000 lines of code. Version 2.4 has more than two millions lines of code (MLOCs). Figure 6.1 shows the trend in size evolution from the release 1.0 up to the 2.4. For example, one of the latest stable releases, the 2.4.18 release, is composed of about 14000 files; its size is about 3 MLOCs (.c and .h). Counting the LOCs contained in .c files (i.e., excluding include files), it is about 2.5 MLOCs (.c files only). The architecture-specific code accounts for 422 KLOCs. In platform-

independent drivers (about 1800 files) there are about 1.6 MLOCs. The core kernel and file systems contain 12 KLOCs and 235 KLOCs respectively.

Even-numbered releases (e.g. 1.0.x, 1.2.x, 2.4.x) represent the stable branches and only incorporate bug fixes and well-tested, low-disturbance, highly demanded new functionality. Figure 6.1 shows the still ongoing stable release 2.0.x (dotted line).

An odd-numbered branch, unstable branch, is created soon after a new stable branch appears. For example, 1.3.x releases, the dashed line leading to release 2.0 (v2.0). New features are added and evaluated in unstable releases until enough features have been included or too much time has elapsed since the last stable branch. Thereafter, the unstable branch gets into feature freeze and all the important bugs are fixed. The last version of the unstable branch ends the unstable branch and is followed by the initial version of a new stable branch.

The Linux kernel is an ideal candidate as testbed for automated code examination and comprehension tools. It is based on the Open Source concept, so source code is available and can be inspected and studied. It is representative of real-world software systems. It is also too large to be examined manually.

Unlike other Unixes (e.g., FreeBSD), Linux it is not directly related to the Unix family tree, in that its kernel was totally new, not written by porting existing Unix source code. The very first version of Linux was targeted at the Intel 386 (i386) architecture. When the Linux project started, research community generally believed that high operating system portability could be achieved only by adopting a microkernel approach. The fact that Linux, which relies on a traditional monolithic kernel, now runs on a wide range of hardware platforms, including palmtops, Sparc, MIPS and Alpha workstations, as well as IBM mainframes, clearly points out that portability can also be obtained by the use of clever code structure.

Linux is based on the Open Source concept: it is developed under the GNU

Release Series	Initial Initial	Number of Releases	Time to Start of Next Release Series	Duration of Series
0.01	9/17/91	2	2 months	2 months
0.1	12/3/91	85	27 months	27 months
1.0	3/13/94	9	1 month	12 months
1.1	4/6/94	96	11 months	11 months
1.2	3/7/95	13	6 months	14 months
1.3	6/12/95	115	12 months	12 months
2.0	6/9/96	34	24 months	32 months
2.1	9/30/96	141	29 months	29 months
2.2	1/26/99	19	9 months	still current
2.3	5/11/99	60	12 months	12 months
2.4	1/4/01	18	–	still current
2.5	22/11/01	8	–	still current

Table 6.1: Linux kernels most important events.

General Public License and its source code is freely available to anyone who wishes it.

A special characteristic of Linux is that it is not issued from an organizational project but has evolved through the efforts of volunteers from all over the world who contributed code, documentation and technical support. This effort involved more than 3000 developers in 90 countries on five continents (Moon and Sproull 2000). Due to the nature of the decentralized and voluntary basis of development effort, no formalized development processes has been adopted.

A key point in Linux structure is modularity. Without this, it would be impossible to use the open-source development model and to let many developers work at the same time on different modules. High modularity means that people can cooperate on the code without conflicts. Possible code changes have an impact confined to the module into which they are contained, without affecting other modules. To this aim the Linux kernel architecture was redesigned with one common code base that could simultaneously support a separate specific tree for any number of different machine architectures,

The use of loadable kernel modules (Goyeneche and Sousa 1999), which are dynamically loaded and linked to the rest of the kernel at run-time, was introduced

with the 2.0 kernel version. Kernel modules further enhanced modularity by providing an explicit structure for writing hardware-specific code (e.g., device drivers). Besides making the core kernel highly portable, this enabled many programmers to work simultaneously on the kernel without central control.

An important management decision was made in 1994 which allowed a parallel release structure for the Linux kernel. Even-numbered releases were the development versions on which programmers could experiment with new features. Once an odd-numbered release series incorporated sufficient new features and became sufficiently stable through bug fixes and patches, it would be renamed and released as the next higher even-numbered release series and the process would begin again. The principal exception to this release policy has been the complete replacement of the O.S. virtual memory system in the 2.4 version series (i.e., within a *stable* release). More details on the Linux history can be found in (Bar 2001; Bar 2002).

Table 6.1, which is an updated version of that published in (Moon and Sproull 2000), shows the most important events in the Linux kernel development time table, along with the number of releases produced for each development series.

6.2.2 Linux Evolution Across Releases

To monitor and quantify the similarity across releases, the first 365 Linux kernel releases (66 stable and 299 unstable) were used. Releases are available at *http://www.memalpha.cx/L* and can be freely downloaded. $OAV(A,B)$, $CR(A,B)$, $ASCS(A,B)$, and $ACCS(A,B)$, described in Subsection 6.1.1, were computed between subsequent releases (release R_k versus R_{k+1}) and between release R_k and one of two *extreme points*: the first and the last considered releases, release 1.0 versus release R_k and release R_k versus 2.4 respectively.

Figure 6.3 displays the histograms of the $OAV(R_k, R_{k+1})$ and $CR(R_k, R_{k+1})$ for stable/unstable releases. The Y axis represents the frequency, i.e., the count of the

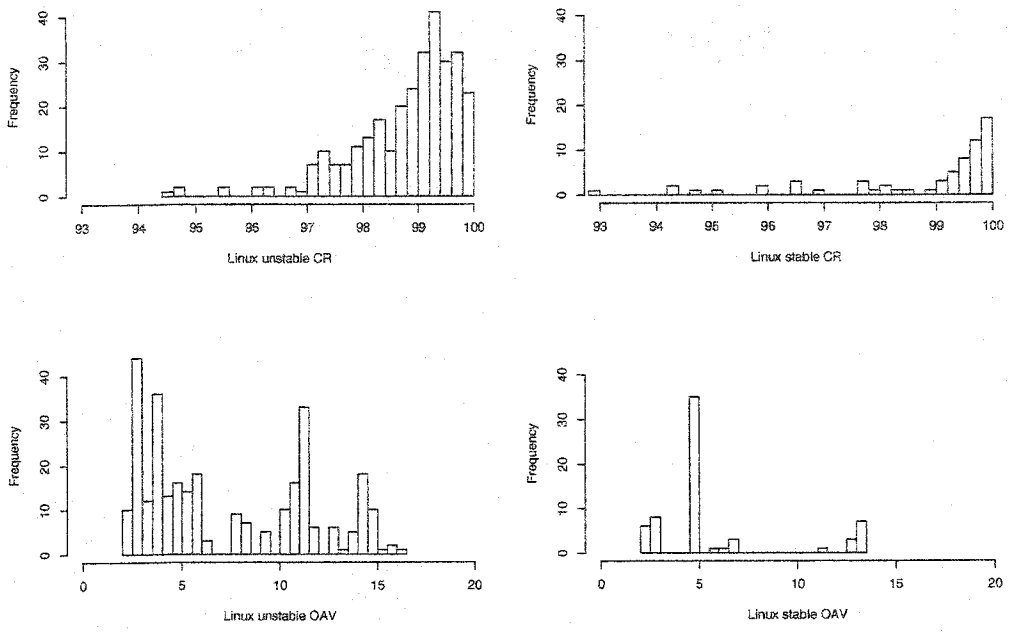

Figure 6.3: CR(A,B) and OVL(A,B) for stable and unstable releases.

elements in the histogram X axis cell; thus, there are about 45 unstable releases with $CR(R_k, R_{k+1})$ between 99.2 and 99.4 %.

It can be observed that $CR(R_k, R_{k+1})$ values are more concentrated toward 100% for the stable release, while the $OAV(R_k, R_{k+1})$ exhibits lower values. This is not surprising since a stable release is rarely modified. Modification occurs if bugs are discovered or if a new important and *stable* piece of code, developed in the unstable product line, is judged worthwhile to be introduced it in the next stable release. Conversely, unstable releases experience a more turbulent evolution: new drivers may be added by copy-and-paste pre-existing drivers, entire files may appear and disappear or may be moved from one sub-system to another. Experimental code coexists with stable code until the decision to create a new stable release is undertaken.

In other words, programmers tend to evolve the system by copy-and-paste and then modify the code. For example, when a new driver is added, a similar driver

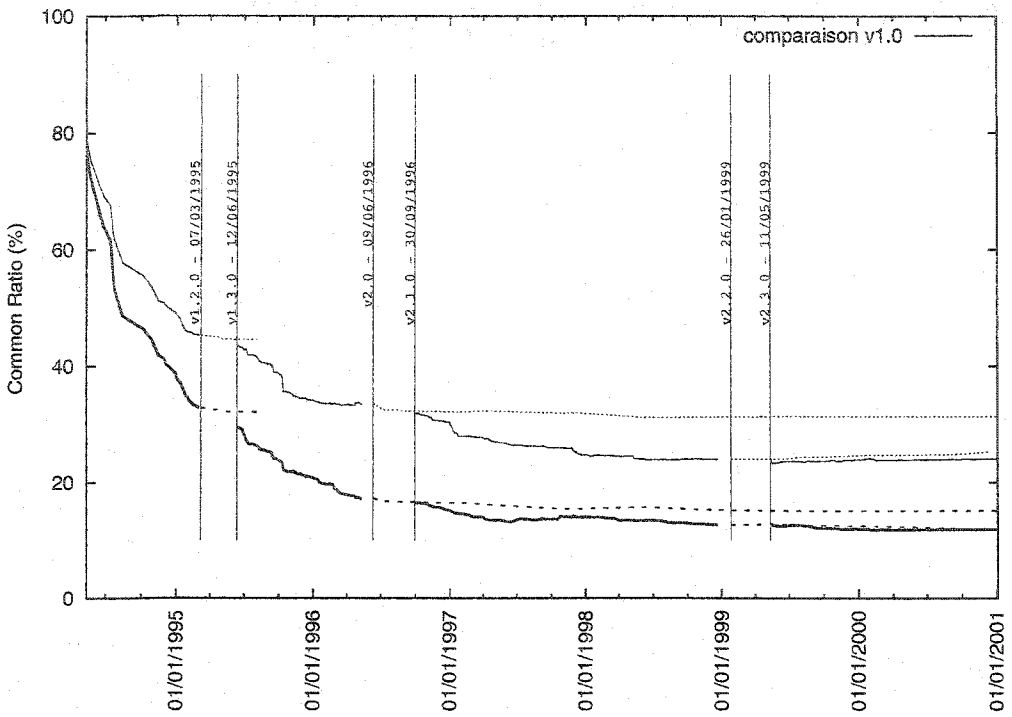

Figure 6.4: Common ratio $CR(1.0, R_k)$ k varying from 1.0 to 2.4.

may be used as a starting point. Similarly, when a new architecture is introduced (e.g., mips64 versus mips or sparc64 versus sparc), large chunks of code and entire sub-systems may be reused. The development practice of using the two product lines (stable and unstable) promotes high values of the OAV in unstable releases. As shown in Figure 6.3, experimental releases contain several clusters with more than 10 copies of the same function. Stable releases do not exhibit the same pattern. Linux kernels undergo a refactoring activity which removes system level duplication before releasing the new stable release. Also, high CR values are more frequent between unstable release. This may be explained by the differences between stable and unstable releases. Daily or weekly unstable snapshots are very common, while the time between two stable releases is considerably higher. Linux developers cannot deliver a very large amount of code in a very short time, thus two subsequent unstable releases generally contain the same code. A new stable release adds significant new features with respect to the previous stable release, as a result, among unstable releases there

is much higher code commonalities than among stable releases as shown in Figure 6.3.

More insight on the proposed metrics as well as on Linux evolution is gained by considering Figure 6.4 and Figure 6.5 where $CR(1.0, R_k)$ and $CR(R_k, 2.4)$ are plotted. Stable and unstable releases were not differentiated and were considered only with reference to the time scale. As illustrated in Figure 6.1, the evolution of $CR(1.0, R_k)$ and $CR(R_k, 2.4)$ show the different branches of stable and unstable releases (i.e., v1.2, v1.3, v2.0, v2.1, v2.2, v2.3 and v2.4). CR was computed for LOC and plotted as the upper family of lines, and it was also computed for C statements and plotted as lower bold lines. LOC and statements CR exhibit the same pattern: the diagrams may be thought of as representative of the rate at which source code fades away and is dismissed, and as the rate at which new code is respectively added. These are complementary phenomena. As the code evolves, only a fraction of the original code is retained, while new code is added.

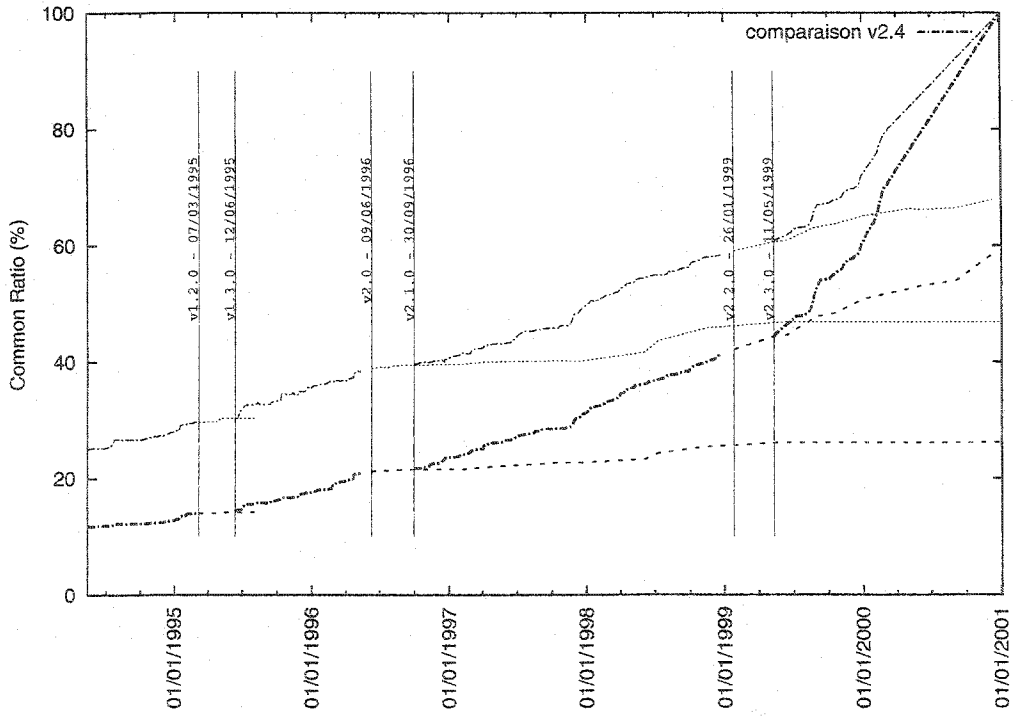

Figure 6.5: Common ratio $CR(R_k, 2.4)$ k varying from 1.0 to 2.4.

In particular, Figure 6.4 clearly demonstrates that there is a *core* part of the system that does not evolve and dates back to the early Linux design and development stages. As the size increases, the percentage of source code retained from version 1.0 decreases and it is limited to a 12 % of 2.4 release (lower bold line of Figure 6.4). Meanwhile, as shown by Figure 6.5, freshly developed code is added and substantially contributes to the newest releases. Consider for example the releases v2.2 - 26/01/1999, its code constitutes about 40 % of the newest 2.4.0 release.

Figure 6.6 shows the temporal evolution of $ACCS(R_k, R_{k+1})$ as well as tendency to increase the cluster size as a function of the time. Further investigation is required to better understand the results. It is well known that a relevant number of platforms and drivers were recently added; for example, all SCSI drivers are likely to share several functions and thus several fragments of code; therefore, adding new drivers may increase the cluster size. This observation seems to be in partial agreement with the findings of other researchers (Godfrey and Tu 2000).

Interestingly, in our data sample, the $ASCS(R_k, R_{k+1})$ is almost constant with a mean of 1.074240 and a standard deviation of 0.13. $ASCS(R_k, R_{k+1})$ represents the fraction of code removed passing between two consecutive releases. Unstable releases dominate over stables (mean 1.078563 versus 1.049089), since more code is removed from unstable releases.

6.2.3 Linux 2.4.18 Intra Release Evolution

This subsection investigates the cloning ratio *micro-evolution* in the Linux kernel from release 2.4.0 to release 2.4.18. The analysis has been performed at different levels of granularity:

1. The overall cloning on the entire Linux kernel;

2. The cloning among major subsystems; and

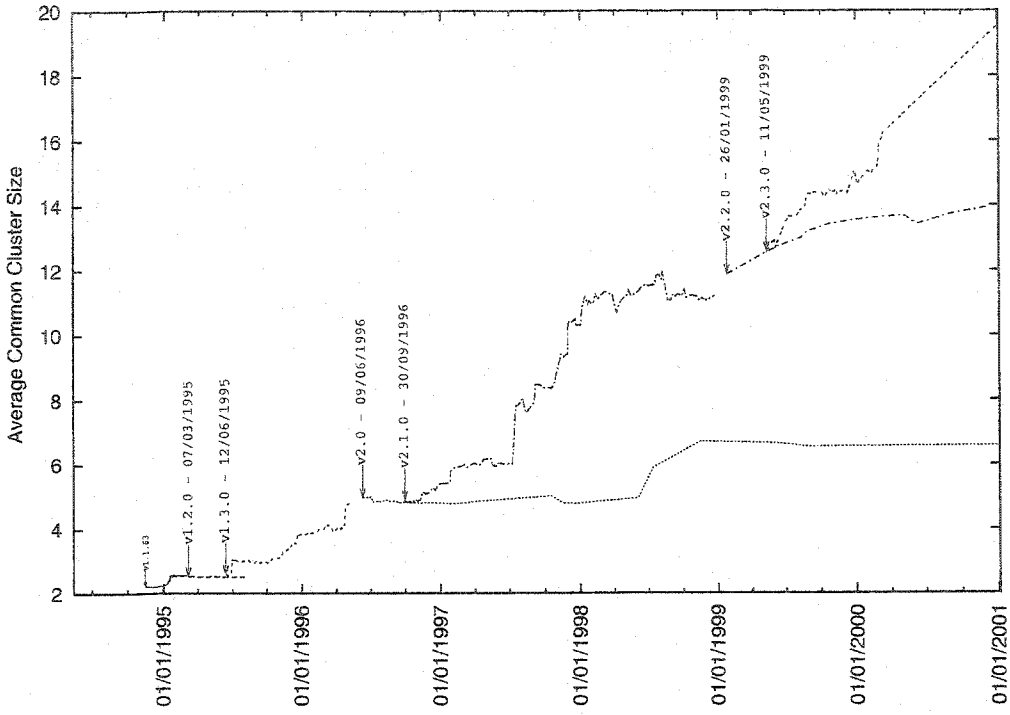

Figure 6.6: $ACCS(R_k, R_{k+1})$ k varying from 1.0 to 2.4.

3. The cloning among architecture-dependent code of some subsystems.

In (Casazza et al. 2001), the clones were identified considering all functions contained in the system regardless of their sizes measured as the number of LOCs of the function body. Small functions such as setting functions or getting the value of a structure very often clustered together. However, it may be argued that these functions do not really represent relevant clones, and thus that the resulting CR is biased by false positives that significantly affect the micro evolution trend.

To study the influence of short functions on CR, this index was computed for two different configurations. The first configuration corresponds to the assumptions made in (Casazza et al. 2001), namely, all functions, regardless of their sizes, were considered. In the second configuration, however, all functions with a body shorter

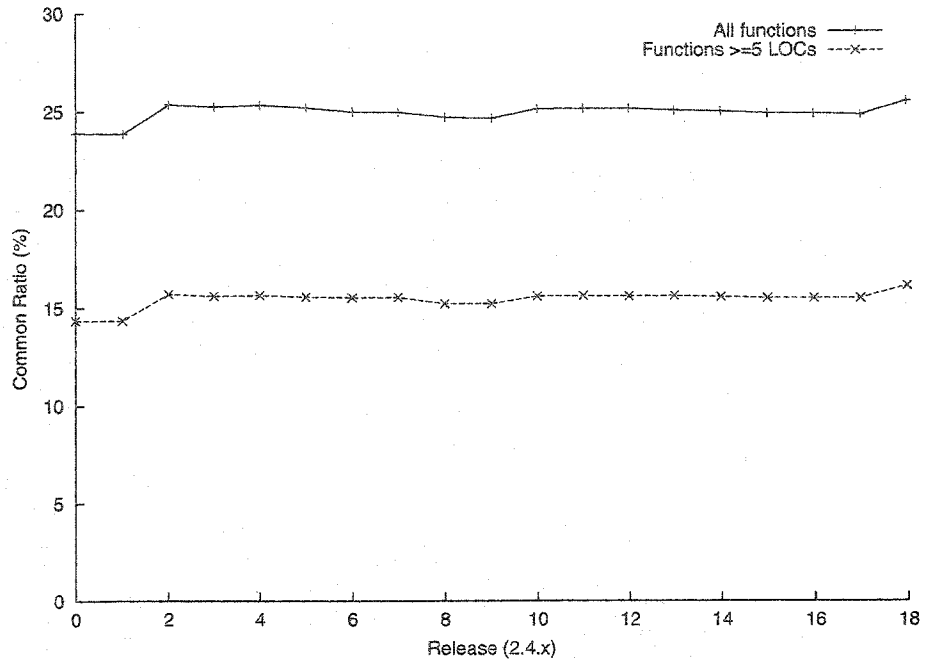

Figure 6.7: Overall evolution of common ratio.

that five LOCs were discarded, detecting clone clusters and computing the CR only on the remainders.

While analyzing CRs on several Linux releases, it was observed that CRs among all possible combinations of Linux subsystems were often null or very low and that they corresponded to sparse cloning matrices. Furthermore, according to the definition of CR, a high CR value does not necessarily imply a high number of replicated code fragments. A 50% CR may correspond only to a couple of cloned functions if small subsystems are considered. On the other hand, if the analyzed subsystems contain a high number of functions, say 1000, even a CR as low as 1% will be interesting. In the analysis that follows, only the most significant results are reported i.e., high CR values or high number of cloned functions.

Figure 6.7 reports the evolution of the overall CR, computed considering both all functions and only functions ≥ 5 $LOCs$. The figure shows that results are very

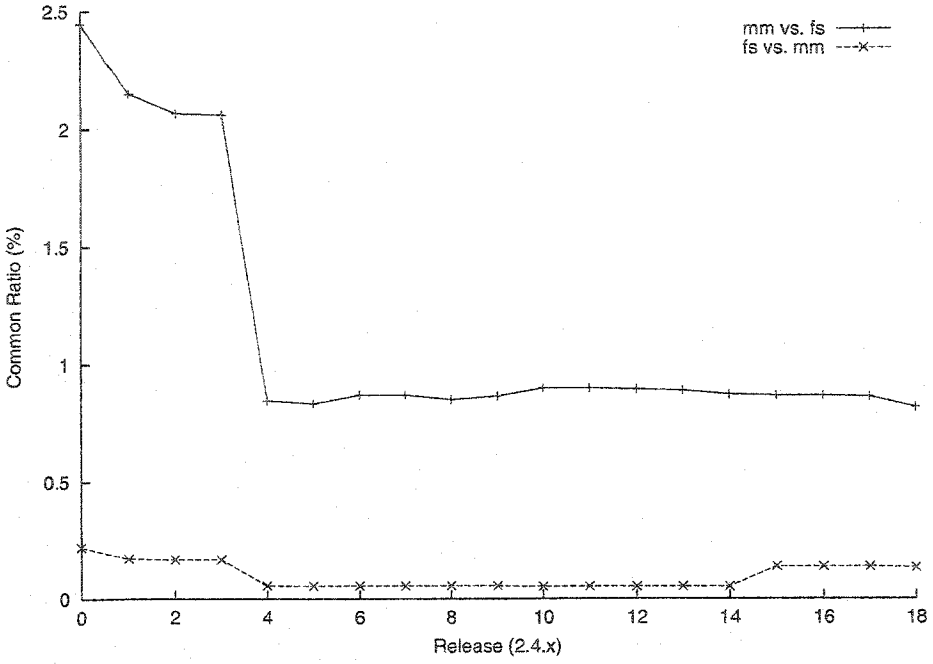

Figure 6.8: Evolution of common ratio between `mm` and `fs`.

different if small functions are filtered. In both cases, the cloning variation over releases is not relevant. Focusing our analysis on CR for functions ≥ 5 LOCs (as well as in all the further analyses presented in this subsection), the CR varies from 14.33% to 16.11%, i.e., a maximum difference of about 2%, and its standard deviation is 0.03. This supports the hypothesis that no considerable refactoring was performed across 2.4.x releases.

The analysis of CR evolution among major subsystems confirms the previous impressions. Even in this case, no variation higher than 2 % in the CR has been detected. Figure 6.8 shows the evolution of cloning between `fs` and `mm` subsystems. From release 2.4.0 to release 2.4.4, the CR in `mm` decreased by about 1.6% (about 20 functions), indicating a possible refactoring activity.

Similar to the results presented in the previous subsection, the most interesting behavior of CR evolution was found in relation to the `mips64` and `mips` architecture-

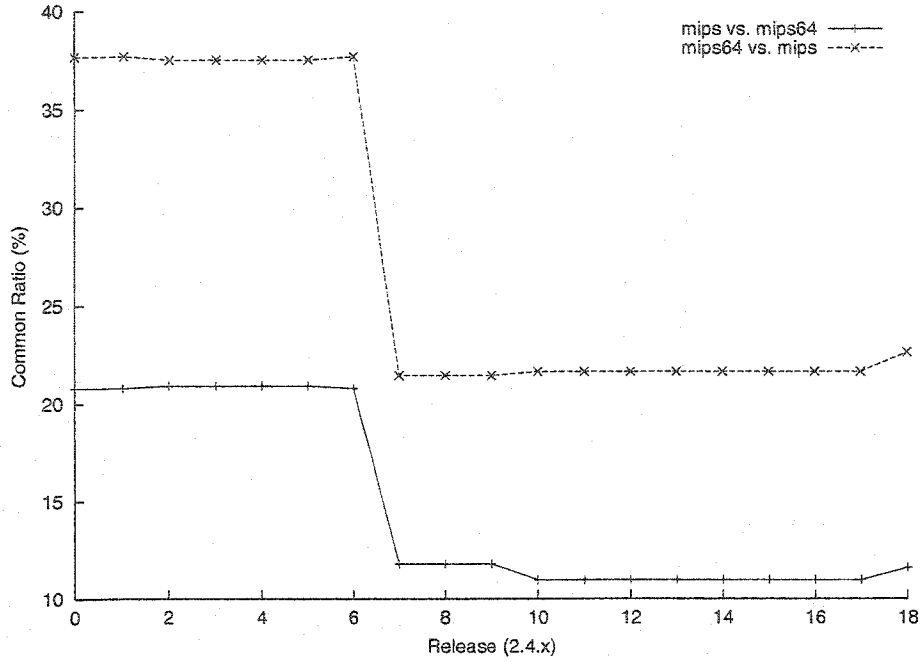

Figure 6.9: Evolution of common ratio between `mips` and `mips64` code inside the `mm` subsystem.

dependent code in the `mm` subsystem.

The values of CR are plotted in Figure 6.9. It shows that the CR ranged from 37.68%, for release 2.4.0 which slightly different from the 38.4% reported in (Casazza et al. 2001) and computed considering all functions, to 22.60% for release 2.4.18.

One may argue that the programmers first ported the `mm` subsystem to the `mips64` architecture by cloning portions of the `mips` code, and then performed a refactoring. However, a more detailed analysis demonstrated the exact opposite. In fact, the number of functions (≥ 5 $LOCs$) composing the `mips64` portion of `mm` varied from 69 in release 2.4.0 to 115 in release 2.4.18, in that the number of cloned functions remained constant to:

$$37.68\% \; of \; 69 = 22.60\% \; of 115 = 26$$

In other words, like any relative measure, CR should be used with great care, always resorting to the examination of absolute values.

6.2.4 Linux 2.4.18 Self Similarity

Computing the cloning ratio among major Linux architectural components and computing the percentage of duplicate code among different supported platforms assumes an architectural description of the software under study. However, such documentation does not exist, since no document describes the system at a high level of abstraction. Bowman et al. derived both the conceptual architecture (the developers' system view) and the concrete architecture (the implemented system structure) of the Linux kernel (Bowman 1998; Bowman et al. 1999). They started from a manual hierarchical decomposition of the system structure which consisted of the assignment of source files to subsystems and of subsystems to subsystems in a hierarchical way. As shown in (Bowman et al. 1999), the extracted subsystems usually correspond to directories in the source code directory tree (often simply referred to as the source tree) as produced by expanding the Linux distribution tarball. For simplicity's, in the performed analysis, it has been assumed that each directory of the source tree contains a subsystem at a proper level of the system hierarchy. Thus the search for cloned code was performed by comparing the code contained in any two directories.

Figure 6.10 shows two different examples of identified function clones. The first clone pair (top of the figure), is an example of a function *copied* from `mips` to `mips64` memory management subsystem. On the other hand, the second clone pair is a cross-system example: although the accessed data structure has different field names, the action actually performed is the same, namely, the removal of an item from a concatenated list.

Table 6.2 reports the CRs higher than 1% among Linux major subsystems which are the twelve top-level directories, *documentation* and *include* directories excluded. CRs are reported along with the corresponding number of cloned functions, both for the considered configurations of functions longer than five LOCs and for all functions.

Table 6.2 clearly shows that:

```
linux-2.4.0/arch/mips/mm/init.c
mips
pte_t *get_pte_slow(pmd_t *pmd,
                    unsigned long offset)
{
        pte_t *page;

        page = (pte_t *) __get_free_page(GFP_KERNEL);
        if (pmd_none(*pmd)) {
                if (page) {
                        clear_page(page);
                        pmd_val(*pmd) =
                            (unsigned long)page;
                        return page + offset;
                }
                pmd_set(pmd, BAD_PAGETABLE);
                return NULL;
        }
        free_page((unsigned long)page);
        if (pmd_bad(*pmd)) {
                __bad_pte(pmd);
                return NULL;
        }
        return (pte_t *) pmd_page(*pmd) + offset;
}
```

```
linux-2.4.0/arch/mips64/mm/init.c
MIPS64
pte_t *get_pte_slow(pmd_t *pmd,
                    unsigned long offset)
{
        pte_t *page;

        page = (pte_t *) __get_free_pages(GFP_KERNEL, 0);
        if (pmd_none(*pmd)) {
                if (page) {
                        clear_page(page);
                        pmd_val(*pmd) =
                            unsigned long)page;
                        return page + offset;
                }
                pmd_set(pmd, BAD_PAGETABLE);
                return NULL;
        }
        free_pages((unsigned long)page, 0);
        if (pmd_bad(*pmd)) {
                __bad_pte(pmd);
                return NULL;
        }
        return (pte_t *) pmd_page(*pmd) + offset;
}
```

```
fs/dquot.c
static inline
       void remove_inuse(struct dquot *dquot)
{
        if (dquot->dq_pprev) {
                if (dquot->dq_next)
                        dquot->dq_next->dq_pprev =
                        dquot->dq_pprev;
                *dquot->dq_pprev = dquot->dq_next;
                dquot->dq_pprev = NULL;
        }
}
```

```
arch/arm/mm/small_page.c
static void
       remove_page_from_queue(struct page *page)
{
        if (page->pprev_hash) {
                if (page->next_hash)
                        page->next_hash->pprev_hash =
                        page->pprev_hash;
                *page->pprev_hash = page->next_hash;
                page->pprev_hash = NULL;
        }
}
```

Figure 6.10: Two examples of clones found.

- The table contains only seven rows out of 144 possibilities; in other words, only very few subsystem comparisons gave raise to appreciable clone extents;
- The difference between the results obtained considering all functions and those obtained with a 5-LOCs threshold is relevant;
- Though CRs among major subsystems is not very high, even a small ratio (e.g. 1.43% between `arch` and `drivers`) corresponds to 152 cloned functions, as these subsystems are very large.

In the two configurations, CRs were computed by considering the ratio to the total number of retained functions. This may lead to two counterintuitive phenomena: higher CR for functions ≥ 5 *LOCs* and different CRs corresponding to the same number of cloned functions are observed because of the lower number of functions that are assumed to belong to the system.

	Functions ≥ 5 *LOCs*		All Functions	
Subsystems Compared	**Common Ratio**	**Functions Cloned**	**Common Ratio**	**Functions Cloned**
arch-drivers	1.43%	152	13.46%	1821
fs-drivers	2.06%	93	10.38%	549
ipc-arch	1.45%	1	1.35%	1
kernel-arch	2.11%	114	13.17%	902
lib-arch	2.90%	9	2.86%	14
lib-net	1.45%	4	1.43%	7
mm-drivers	1.36%	18	4.80%	78

Table 6.2: CRs $\geq 1\%$ among major subsystems.

	Functions ≥ 5 *LOCs*		All Functions	
Subsystems Compared	**Common Ratio**	**Functions Cloned**	**Common Ratio**	**Functions Cloned**
i386-mips	11.11%	1	10.34%	1
i386-s390	11.11%	1	10.34%	1
i386-sh	14.81%	3	17.24%	3
mips64-mips	22.61%	6	28.57%	8
mips-mips64	11.59%	2	17.38%	3
s390-arm	10.00%	1	13.64%	2
s390-i386	15.00%	2	13.64%	2
s390-mips	10.00%	1	9.09%	1
s390-sh	15.00%	2	13.64%	2
sh-i386	10.00%	1	11.63%	1
sparc64-sparc	12.77%	2	14.00%	2

Table 6.3: CRs $\geq 10\%$ among `mm` architecture dependent code.

A similar approach was followed to evaluate the cloning extents within the subsystems related to the different supported platforms. The `arch` directory contains fifteen sub-directories, each corresponding to a supported processor architecture (e.g., `i386`, `s390`, `sparc`). Each platform has, among others, its own `kernel` and memory management `mm` implementations. In particular, Table 6.3 shows the CRs among `mm` for the architectures supported by Linux 2.4.18. A different threshold (10%), higher than the 1% used for Table 6.2, was used to avoid reporting meaningless data. Only 10 rows out of 225 were retained and, as can be readily seen in Table 6.3, the `mm` subsystems contain only few cloned functions even if the CR values are not very low.

	Functions ≥ 5 *LOCs*		All Functions	
Subsystems Compared	Common Ratio	Functions Cloned	Common Ratio	Functions Cloned
sbus-char	6.62%	53	14.48%	138
sgi-char	6.80%	7	15.83%	19
tc-char	9.38%	6	21.69%	18
i2c-parport	5.44%	8	10.45%	23
input-usb	5.88%	3	11.86%	7
sgi-macintosh	5.83%	6	11.67%	14
tc-macintosh	12.50%	8	25.30%	21
zorro-pci	8.33%	1	8.33%	1
sgi-sbus	10.68%	11	17.50%	21
tc-sbus	10.94%	7	22.89%	19
sgi-tc	7.77%	8	11.67%	14
tc-sgi	12.50%	8	20.48%	17

Table 6.4: CRs $\geq 5\%$ among `drivers`.

The data for Linux 2.4.18 confirmed the results obtained on different Linux releases (Casazza et al. 2001; Bowman et al. 1999). In most cases, the implementation of similar functionalities was carried out by introducing function dependencies across different subsystems to reuse code rather than cloning. This is clearly shown by the small number of subsystems which exhibit a non-negligible number of cloned functions.

There are a few exceptions, however. Among these, is the CR between the `mips64` and `mips mm` subsystems (22.61%, with six cloned functions). The ratio obtained without filtering out functions smaller than five LOCs was slightly higher (28.57%), but considerably smaller than the 38.4% computed on the Linux Kernel 2.4.0 and reported in (Casazza et al. 2001). However, even in this case, the absolute number of cloned functions is low.

Table 6.4 reports data on CR and cloned functions among Linux drivers. Driver subsystems (e.g, the `SCSI` and `IDE` drivers, the `char` and `USB` or the `PCI` drivers) are the largest part of the kernel code and are subject to continuous evolution. CR among driver subsystems is fairly low, and in general only a few functions are duplicated. An exception seems to be the number of duplicated functions between the `char` and

sbus subsystems, where 53 clone clusters were identified.

6.3 Chapter Summary

Large procedural system analysis requires the adaptation of methods and tools to trace and monitor system evolution. This chapter presented an approach based on the detection of code duplications to study the evolution of large systems developed with the C programming language. The main contributions are the approach itself, which is not tied to the C programming language, and the four high level software metrics to capture software evolution. The defined software metrics abstract some key aspects of large software evolution such as the volume of actually changed code between subsequent releases. Methods and techniques were applied to more than 400 releases of the Linux kernel showing the high performance and scalability of the approaches.

CHAPTER 7: Conclusions

7.1 Conclusions

Traceability links between software artifacts either at different abstraction levels or belonging to subsequent releases of the same software artifact are often inconsistent or absent. Automatic tools have been developed to support vertical traceability and horizontal traceability checks pinpointing potential discrepancies and lack of traceability between the artifacts. Reports about compliance, graphical layouts such as the pair-difference diagrams or high level abstractions, and figures summarizing changes (e.g., the CR plotting) may prove useful both for development and evolution activities.

Several approaches have been investigated. Depending on the goals and available sources of information, deterministic approaches (IR - vector spaces, graph based, clone) or probabilistic approaches (unigram and bigram language model) can be applied. More precisely, the following taxonomy was applied to recover or validate traceability links from:

- free text documentation such as manual pages or requirement documents
  - target code: apply vector spaces or unigram and bigram language model
- design
  - target design or code: apply graph based approaches
- source code

- target source code: apply graph based approaches (OO software) or clone based approaches (procedural code)

The following subsections consider each identified option.

7.1.1 Free Text Documentation

Deterministic as well as probabilistic approaches have been developed and applied to several software systems (Albergate, LEDA, Linux, etc). In particular, IR methods were adapted to recover traceability links between code and free text documentation; IR methods were applied to trace C++ and Java source classes to manual pages and functional requirements, respectively.

The results achieved in the two case studies with IR models support the hypothesis that IR provides a practicable solution to the problem of semi-automatically recovering traceability links between code and documentation.

As shown in Chapter 3, both models achieve 100% recall with almost the same number of documents retrieved. However, the unigram language model achieves highest recall values (less than 100%) with a smaller number of documents retrieved and performs better when 100% recall is required. On the other hand, the vector space model shows regular progress in the recall values when the number of retrieved documents increases. Also, it requires less effort in the preparation of queries and document representations.

Vector space and probabilistic models need to be further validated on larger systems to assess the relative performance: on the available case studies, the probabilistic approach may be preferred when high recall values close to 100 % recall are required with low cut values. In these cases effort saving may be preferred over the recovery of a complete mapping.

However, different results may be obtained by other researchers on different systems. Indeed, no statistical evidence was obtained to support one approach over the other.

Concerning the adequacy of a probabilistic model, a quick comparison between the size of software engineering documentation and natural language processing corpora makes it clear that the software engineering arena suffers from the problem of scarce training material. Sparseness of data and zero frequency may be alleviated by smoothing techniques. Different smoothing techniques were tested (De Mori 1998; Ney and Essen 1991; Witten and Bell 1991). In the Albergate and LEDA case studies, shift-β gave the best results. It should be emphasized that smoothing gives very low non-zero probabilities to unseen words; as a result, a query is sometimes dominated by the weight of words unseen in the training material.

Benchmarking the approaches against a `grep` brute force traceability link recovery demonstrates the benefits of the more sophisticated technologies. As in (Maarek et al. 1991), `grep` is overwhelmed by IR approaches. It also appears that, as the distance between software artifacts increases, the `grep` performance decreases.

The best results were achieved once text normalization was applied, thus implicitly introducing a more tolerant matching criterion. In the Albergate case study, the higher distance between artifacts makes the recovery task more difficult and not surprisingly, the effect of text normalization was considerably higher.

However, text normalization in some cases can fail to reconnect software documents and source code to a common vocabulary. Indeed, the key idea of the method based on the unigram model is that the application-domain knowledge processed by programmers is captured by the identifiers. Under this assumption, the source code identifier vocabulary shares a significant number of items with the documentation vocabulary. Although this conjecture is supported by the results obtained in both case studies, the effectiveness of the method becomes less pronounced when the number of common words between the source code component identifiers and the documentation items decreases.

7.1.2 Bigram Language Model

The unigram language model limitations can be overcome by extending the approach, as shown in Chapter 4. Indeed, it seems reasonable that programmers tend to process application-domain knowledge in a consistent way when writing code: program item names of different code regions related to a given text document are likely to be the same or very similar. Under this assumption, the information about existing traceability links can be exploited to recover new traceability links, even when the number of common words between the source code component identifiers and the documentation is very low or null. In other words, once programmer behavior is modeled, no matter where the knowledge comes from, few links suffice to recover all the other traceability links. Programmer behavior can be captured through stochastic modeling by learning the rules programmers adopt to map high level documentation and domain concepts into low level artifacts such as program item names. Once a subset of existing traceability links is known, for any given link the joint probability distribution of the high level document and the linked source code components is estimated together with the marginal probability distributions. The estimated probability distributions are used in a Bayesian classifier to score sequences of mnemonics extracted from a not yet *classified* code component i.e., a component not belonging to the subset of known traceability links. Higher scores suggest the existence of links between the component from which a particular sequence of mnemonics is extracted and the document that generated the marginal probability distribution. Results show that this approach represents a valid alternative to the method presented in Section 3 when documentation and code do not share a common vocabulary.

The bigram language model was applied to three different software systems. Code regions i.e., classes and files were traced into the functional requirements. Accuracy was evaluated by comparing results with the traceability matrices compiled by the developers of the system. The three case studies can be considered representative of different development approaches, languages and tools. Traceability links were

recovered in systems which were developed with RAD IDE or code generators and which incorporated databases, middleware and reused code.

In most cases, as the training set increased, the method performance improved. In other words, on the available data, the bigram language model learns. In other words, the joint probability distribution seems to effectively capture the consistency rules applied when creating identifiers. Clearly, the task of recovering traceability links may be eased by enforcing appropriate coding standards.

When using a RAD IDE environment, reused code, external architectures or middleware, programmers tend to assign meaningful names only to a fraction of the identifiers. As a consequence, non-domain specific names may dominate over domain related names, thus confusing the traceability recovery process. Untraceable elements should be removed from the analysis. In the case of automatically generated code, heuristics may be adopted to discard classes that, after pruning automatic generated identifiers, exhibited a list of identifiers which was empty or smaller than a fixed threshold.

While performing the traceability recovery process, it turned out that comments are a valuable source of information. These were exploited to recover traceability links contrary to the approach of Chapter 3. Clearly, this required coding standards or heuristics to help associate comments with classes and methods.

Text normalization is fundamental. However, as highlighted in Chapter 4, to obtain benefits from normalization, it is mandatory to enforce compliance with the proposed process. Moreover, particular attention should be paid to normalizing identifiers, such as bringing back object names and event handlers to the same radix.

Finally, it has been observed that enriching the training set generally increases the precision and recall of subsequent steps. This fact is very relevant, since other methods, such as those proposed in Chapter 3, do not allow the re-estimation of the model parameters when further information is available.

7.1.3 OO Design to Code and OO Code to Code Mapping

Design documents are often inconsistent with source code implementation. Furthermore, industrial software, especially that developed with OO technology, is often based on a component-based strategy or COTS and libraries. A similarity criterion imposing the exact matching of entities is unlikely to produce meaningful results. A design-code compliance check tool must take into account the "physiological" distance between design and code or between subsequent releases. It must also be robust with respect to the inconsistencies between design and code caused by reuse and COTS. To properly handle COTS, libraries and reused code, a maximum likelihood classifier was applied after the maximum match computation to obtain a more accurate classification of classes.

The concept of *similarity* between entities in design and code, while relaxing the constraint of exact name matching by introducing an edit distance, was the key to finding the best match in cases in which modifications were introduced in the names of the attributes.

The design-code compliance check was applied to an industrial system and obtained an average traceability of 0.971, with an average of 6.37 unmatched classes in the design. Before applying the maximum likelihood classifier, the average traceability was 0.890 and the number of deleted classes 2.24.

The traceability of the relations between matched classes gives an overall value of 96.8% generalizations, 33.9% aggregations, and 21.9% associations which were specified in the design and implemented in the code. The lower traceability of aggregations and associations is in part due to the intrinsic limitations of the reverse engineering tool described in Subsection 1.3.1.2 which extracts them from C++ code.

The similarity between subsequent releases of an OO system is immediately derived from the approach devised to trace OO design into code. By reverse engineering the source code, the as-is design corresponding to the given software release is obtained. On those designs, the edit distance and maximum matching detailed in

Chapter 5 recover a traceability map. The only significant difference between the approaches proposed in Chapters 2 and 5 is the adoption of a different weighting schema which allows a compromise between the relative influences of the class name and the properties.

The traceability link recovery approach was applied to two case studies. The first was used to assess the parameters, i.e., the matching weights and the pruning threshold. The second case study was performed to further validate the proposed method. The parameters of the method were calibrated on freely available and public domain software written in C++. Using the method on different systems is likely to require the re-calibration of parameters. For the software analyzed, the best results were achieved when giving a lower weight to class-name matching ($\lambda_c = 30\%$) than to attribute or method matching.

On both systems, a pruning threshold value 70% successfully discarded false matching. Not surprisingly, some failures (less than 2% error rate) were registered for LEDA. Two reasons can explain the differences between the two software packages analyzed:

- LEDA experienced a high growth rate in the number of classes, while the number of classes of the DDD was more stable;
- several LEDA releases were not available, while the DDD history was complete.

The recovered traceability mapping, both at the interface and implementation levels, was used to build models to estimate the size of changes from an estimate of the impacted classes. Estimating the size of a change is a basic step in maintenance cost estimation models.

7.1.4 Clone Based Similarity

In Chapter 6, the evolution of large software systems was been characterized with four similarity measures at the release or system levels. Software metrics are representative of the changes and similarities, among subsystems or, possibly subsequent,

releases of a software system.

The feasibility of the metrics computation was demonstrated on about 400 versions of the Linux kernel. Linux is a widely adopted multi-platform operating system with multi-million lines of code. The collected data analyzes the differences and similarities among Linux kernel releases with high level software metrics.

Linux has not been developed through a well-defined software engineering process, but rather by the cooperative work of relatively independent programmers. Nevertheless, the overall CR, as well the common ratios of its subsystems, are remarkably low, especially if small functions are not taken into account. Linux cloning ratio among sub-systems can be considered at a physiological level; recently-introduced architectures tend to exhibit a slightly higher cloning ratio. The reason for this is that a subsystem for a new architecture is often developed incrementally with respect to a similar one (e.g. `mips64` from `mips`). In general, the evolution of *common ratio*, at the overall level, tends to be fairly stable, thus suggesting that the software structure is not deteriorating due to copy-and-paste practice.

A relatively high *common ratio* value very often corresponds to a small number of duplicated functions. Code duplication may be considered relevant only among few major subsystems (e.g., `arch` versus `drivers`). Even in this case, due to the high number of functions in the subsystems, a *common ratio* value of about 1-2% correspond to just 100-150 duplicated functions.

7.2 Chapter Summary

This thesis presented a spectrum of methods, technologies and approaches to support the recovery of traceability links in software systems. Each chapter addressed specific problems, introduced and applied methods and technologies together with an evaluation of the accuracy on available case studies.

The several novel and original contributions given are summarized in the Introduction (Section 1.5) and at the end of each chapter. The main contributions of the thesis are technologies and tools to support traceability recovery. Foreseeable application

of such methods and techniques to industrial problem will certainly help developers and reduce the effort required to recover, maintain and validate traceability links.

The main limitations of the presented approaches can be summarized as follows:

- Traceability recovery between OO design and code is mainly limited by the quadratic time complexity required to compute the similarities between classes in the design and classes in the code. However, heuristics can be applied if needed, and the tasks can be easily parallelized on a network of machines. In reality, there are few limitations for the software systems of today's size due to the available computational power and modern local area network.

- The main limitation to recovery traceability links between textual documentation and source code is tied to the external validity of the reported results. Their general application in large industrial software should be assessed. The system sizes and the volume of the documentation suggest caution generalizing the obtained accuracy to other software.

- The main limitations of Chapter 4 are the same as outlined in the previous item. Although results are very encouraging, new experiments and larger systems are required to further validate the approach.

- The last two chapters deal with recovering traceability links and study the source code evolution of large OO and procedural systems. As for the methods presented in Chapter 2 performance and scalability are ensured by low computational complexity.

The limitations and threat to external validity outlined above open new research directions in the area of validation as well as in the definition of similarity measures and approaches with linear or almost linear computational complexity.

BIBLIOGRAPHY

[Antoniol et al. 2000a] ANTONIOL, G., CANFORA, G., CASAZZA, G., DELUCIA, A. 2000a. «Information Retrieval Models for Recovering Traceability Links between Code and Documentation». In *Proceedings of IEEE International Conference on Software Maintenance*, pages 40–49, San Jose CA USA. IEEE Comp. Soc. Press.

[Antoniol et al. 2001a] ANTONIOL, G., CANFORA, G., CASAZZA, G., LUCIA, A. D. 2001a. «Maintaining Traceability Links during Object-Oriented Software Evolution». *Software - Practice and Experience*, 31:331–355.

[Antoniol et al. 2000b] ANTONIOL, G., CANFORA, G., CASAZZA, G., LUCIA, A. D., MERLO, E. 2000b. «Tracing Object-Oriented Code into Functional Requirements». In *Proceedings of the 8th International Workshop on Program Comprehension*, pages 227–230. Limerick Ireland.

[Antoniol et al. 2002] ANTONIOL, G., CANFORA, G., CASAZZA, G., LUCIA, A. D., MERLO, E. 2002. «Recovering Traceability Links between Code and Documentation». *IEEE Transactions on Software Engineering*, 28(10):970–983.

[Antoniol et al. 1999] ANTONIOL, G., CANFORA, G., DELUCIA, A., MERLO, E. 1999. «Recovering Code to Documentation Links in object-oriented Systems». In *Proceedings of the IEEE Working Conference on Reverse Engineering*, pages 136–144, Atlanta Georgia. IEEE Comp. Soc. Press.

[Antoniol et al. 2000c] ANTONIOL, G., CAPRILE, B., POTRICH, A., TONELLA, P. 2000c. «Design-Code Traceability for Object Oriented Systems». *The Annals of Software Engineering*, 9:35–58.

[Antoniol et al. 2000d] ANTONIOL, G., CASAZZA, G., CIMITILE, A. 2000d. «Traceability Recovery by Modeling Programming Behavior». In *Proceedings of IEEE Working Conference on Reverse Engineering*, Brisbane Australia. IEEE Computer Society Press.

[Antoniol et al. 2001b] ANTONIOL, G., CASAZZA, G., PENTA, M. D., FIUTEM, R. 2001b. «Object-Oriented Design Patterns Recovery». *Journal of Systems and Software*, 59:181–196.

[Antoniol et al. 2001c] ANTONIOL, G., CASAZZA, G., PENTA, M. D., MERLO, E. 2001c. «A Method to Re-organize Legacy Systems via Concept Analysis». In *Proceedings of the IEEE International Workshop on Program Comprehension*, Toronto ON Canada. IEEE Press.

[Antoniol et al. 1998] ANTONIOL, G., FIUTEM, R., CRISTOFORETTI, L. 1998. «Using metrics to identify design patterns in object-oriented software». In *Proc. of the Fifth International Symposium on Software Metrics*, pages 23–34.

[Arnold and Bohner 1993] ARNOLD, R. S. , BOHNER, S. A. 1993. «Impact Analysis - Towards a Framework for Comparison». In *Proceedings of IEEE International Conference on Software Maintenance*, pages 292–301, Montreal Quebec Canada.

[Arnold and Stepowey 1987] ARNOLD, S. P. , STEPOWEY, S. L. 1987. «The Reuse System: Cataloging and Retrieval of Reusable Software». In Tracz, W., editor, *Software Reuse: Emerging Technology*. IEEE Computer Society Press.

[Baeza-Yates and Ribeiro-Neto 1999] BAEZA-YATES, R. , RIBEIRO-NETO, B. 1999. *Modern Information Retrieval*. Addison-Wesley Publishing Company, Reading, MA.

[Bain and Engelhardt 1992] BAIN, L. , ENGELHARDT, M. 1992. *Introduction to Probability and Mathematical Statistics*. Duxburry Press, Belmont CA.

[Baker. 1995] BAKER., B. S. 1995. «On finding duplication and near-duplication in large software systems.». In *Proceedings of IEEE Working Conference on Reverse Engineering.*

[Ball and Eick 1996] BALL, T. , EICK, S. G. 1996. «Software Visualization in the Large». *IEEE Computer*, 29(4):33–43.

[Bar 2001] BAR, M. 2001. «Linux Kernel Pillow Talk». *Byte magazine.*

[Bar 2002] BAR, M. 2002. «A forest of kernel trees». *Byte magazine.*

[Basili et al. 1996] BASILI, V. R., BRIAND, L. C., MELO, W. L. 1996. «A Validation of Object-Oriented Design Metrics as Quality Indicators». *IEEE Transactions on Software Engineering*, 22(10):751–761.

[Baxter et al. 1998] BAXTER, I. D., YAHIN, A., MOURA, L., SANT'ANNA, M., BIER., L. 1998. «Clone detection using abstract syntax trees.». In *Proceedings of IEEE International Conference on Software Maintenance*, pages 368–377.

[Bennett 1995] BENNETT, K. 1995. «Legacy Systems: Coping with Success». *IEEE Software*, pages 19–23.

[Biggerstaff 1989] BIGGERSTAFF, T. 1989. «Design Recovery for Maintenance and Reuse». *IEEE Computer.*

[Biggerstaff et al. 1993] BIGGERSTAFF, T., MITBANDER, B., WEBSTER, D. 1993. «The Concept Assignment Problem in Program Understanding». In *Proceedings of the International Conference on Software Engineering*, pages 482–498, IEEE Computer Society Press.

[Boehm 1981] BOEHM, B. W. 1981. *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, NJ.

[Boldyreff et al. 1996] BOLDYREFF, C., BURD, E. L., HATHER, R. M., MUNRO, M., YOUNGER, E. J. 1996. «Greater Understanding Through Maintainer Driven

Traceability». In *Proceedings of the International Workshop in Program Comprehension*, Germany. IEEE Press.

[Booch et al. 1997] BOOCH, G., JACOBSON, I., RUMBAUGH, J. 1997. *Unified Modeling Language for Object-Oriented Development, Version 1.0.* Rational Software Corporation.

[Booch and Rumbaugh 1998] BOOCH, G. , RUMBAUGH, I. J. J. 1998. *The Unified Modeling Language User Guide.* Addison-Wesley Publishing Company.

[Bowman 1998] BOWMAN, I. 1998. «Conceptual Architecture of the Linux Kernel». Technical report, Technical Report University of Waterloo, http://plg.uwaterloo.ca/ itbowman/CS746G/a1/. Page accessed July 2003.

[Bowman et al. 1999] BOWMAN, I., HOLT, R., BREWSTER, V. 1999. «Linux as a case study: its extracted software architecture». In *Proceedings of the International Conference on Software Engineering*, pages 555–563. IEEE Computer Society Press.

[Brodie and Stonebraker 1995] BRODIE, M. L. , STONEBRAKER, M. 1995. *Migrating Legacy Systems.* Morgan Kaufmann, San Francisco, CA, USA.

[Brooks 1983] BROOKS, R. 1983. «Towards a Theory of the Comprehension of Computer Programs». *International Journal of Man-Machine Studies*, 18:543–554.

[Brown et al. 1994] BROWN, P. F., CHEN, S. F., PIETRA, S. A. D., PIETRA, V. J. D., KEHLER, A. S., MERCER, R. L. 1994. «Automatic speech recognition in machine-aided translation». *Computer Speech and Language*, 8:177–187.

[Bruegge and Dutoit 2000] BRUEGGE, B. , DUTOIT, A. H. 2000. *Object-Oriented Software Engineering: Conquering Complex and Changing Systems.* Prentice-Hall.

[Bunge 1977] BUNGE, M. 1977. *Treatise on Basic Philosophy: Vol. 3: Onthology I: The Furniture of the World.* Reidel, Boston MA.

[Bunge 1979] BUNGE, M. 1979. *Treatise on Basic Philosophy: Vol. 4: Onthology II: A World of Systems.* Reidel, Boston MA.

[Burton et al. 1987] BURTON, B. A., ARAGON, R. W., BAILEY, S. A., KOELHER, K., MAYES, L. A. 1987. «The reusable Software Library». In Tracz, W., editor, *Software Reuse: Emerging Technology*, pages 129–137. IEEE Computer Society Press.

[Buss et al. 1994] BUSS, E., MORI, R. D., GENTLEMAN, W., HENSHAW, J., JOHNSON, H., KONTOGIANNIS, K., MERLO, E., MULLER, H., PAUL, J. M. S., PRAKASH, A., STANLEY, M., TILLEY, S., TROSTER, J., WONG, K. 1994. «Investigating Reverse Engineeering Technologies for the CAS Program Understanding Project». *IBM Systems Journal*, 33(3):477–500.

[Caldiera and Basili 1991] CALDIERA, G. , BASILI, V. R. 1991. «Identifying and Qualifying Reusable Software Components». *IEEE Computer*, pages 61–70.

[Canfora et al. 1994] CANFORA, G., CIMITILE, A., MUNRO, M. 1994. «RE2: Reverse Engineering and Reuse Re-Engineering». *Journal of Software Maintenance - Research and Practice*, 6:53–72.

[Casazza et al. 2001] CASAZZA, G., ANTONIOL, G., VILLANO, U., MERLO, E., PENTA, M. D. 2001. «Identifying Clones in the Linux Kernel». In *SCAM*, pages 90–97.

[Chidamber and Kemerer 1994] CHIDAMBER, S. R. , KEMERER, C. F. 1994. «A Metrics Suite for Object Oriented Design». *IEEE Transactions on Software Engineering*, 20(6):476–493.

[Chikofsky and Cross 1990] CHIKOFSKY, E. J. , CROSS, J. H. 1990. «Reverse Engineering and Design Recovery: A Taxonomy». *IEEE Software*, 7(1):13–17.

[Coad and Yurdon 1991] COAD, P. , YURDON, E. 1991. *Object Oriented Analysis - Second edition.* Prentice-Hall, Englewood Cliffs, NJ.

[Concling and Bergen 1988] CONCLING, J. , BERGEN, M. 1988. «IBIS: a Hypertext Tool for Exploratory Policy Discussion». *ACM Transaction on Office Information Systems*, pages 303–331.

[Corazza et al. 1991] CORAZZA, A., MORI, R. D., GRETTER, R., SATTA, G. 1991. «Computation of Probabilities for a Stochastic Island-Driven Parser».

[Cormen et al. 1990] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L. 1990. *Introductions to Algorithms.* MIT Press.

[Cover and Thomas 1992] COVER, T. M. , THOMAS, J. A. 1992. *Elements of Information Theory.* Wiley Series in Telecommunications John Wiley & Sons., New York, NY 10158-0012.

[Daly et al. 1995] DALY, J., BROOKS, A., MILLER, J., ROPER, M., WOOD, M. 1995. «The Effect of Inheritance on the Maintainability of Object-Oriented Software: An Empirical Study». In *Proceedings of IEEE International Conference on Software Maintenance*, pages 20–29, Opio-Nice France.

[DeMarco 1982] DEMARCO, T. 1982. *Controlling Software Projects.* Yourdon Press.

[Dominich 2000] DOMINICH, S. 2000. *Mathematical Foundation of Information Retrieval.* Kluwer Accademic Publishers, Boston/U.S.A; Dordrecht/Holland; London/U.K.

[Duda and Hart 1973] DUDA, R. O. , HART, P. E. 1973. *Pattern Classification and Scene Analysis.* John Wiley & Sons, New York, NY 10158-0012.

[Fenton 1994] FENTON, N. 1994. «Software Measurement: A Necessary Scientific Basis». *IEEE Transactions on Software Engineering*, 20(3):199–206.

[Fiutem and Antoniol 1998] FIUTEM, R. , ANTONIOL, G. 1998. «Identifying Design-Code Inconsistencies in Object-Oriented Software: A Case Study». In *Pro-*

ceedings of IEEE International Conference on Software Maintenance, pages 94–102, Bethesda Maryland.

[Frakes and Baeza-Yates 1992] FRAKES, W. B. , BAEZA-YATES, R. 1992. *Information Retrieval: Data Structures and Algorithms.* Prentice-Hall, Englewood Cliffs, NJ.

[Frakes and Nejmeh 1987] FRAKES, W. B. , NEJMEH, B. A. 1987. «Software Reuse Through Information Retrieval». In *Proceedings of 20-th Ann. HICSS Kona (HI)*, pages 530–535.

[Fyson and Boldyreff 1998] FYSON, M. J. , BOLDYREFF, C. 1998. «Using Application Understanding to Support Impact Analysis». *Journal of Software Maintenance - Research and Practice*, 10:93–110.

[Godfrey and Tu 2000] GODFREY, M. W. , TU, Q. 2000. «Evolution in Open Source Software: A Case Study». In *Proceedings of IEEE International Conference on Software Maintenance*, San Jose California. IEEE Press.

[Gotel and Finkelstein 1993] GOTEL, O. C. Z. , FINKELSTEIN, A. C. W. 1993. «An Analysis of the Requirements Traceability Problem». Technical report, Department of Computing Imperial College.

[Goyeneche and Sousa 1999] GOYENECHE, J. D. , SOUSA, E. D. 1999. «Loadable Kernel Modules». *IEEE Software*, 16(1):65–71.

[Harman 1992] HARMAN, D. 1992. «Ranking Algorithms». In *Information Retrival: Data Structures and Algorithms*, pages 363–392. Prentice-Hall, Englewood Cliffs, NJ.

[Henry and Kafura 1984] HENRY, S. , KAFURA, D. 1984. «The Evaluation of Systems' Structure Using Quantitative Metrics». *Software Practice and Experience*, 14(6).

[Holt and Pak 1996] HOLT, R. , PAK, J. Y. 1996. «GASE: Visualizing Software Evolution-in-the-Large». In *Proceedings of IEEE Working Conference on Reverse Engineering*, pages 163–166, Monterey CA.

[Jacobson et al. 1999] JACOBSON, I., BOOCH, G., RUMBAUGH, J. 1999. *The Unified Software Development Process.* Addison-Wesley Publishing Company.

[Johnson. 1993] JOHNSON., J. H. 1993. «Identifying redundancy in source code using fingerprints.». In *CASCON*, pages 171–183.

[Kontogiannis et al. 1996] KONTOGIANNIS, K., MORI, R. D., BERNSTEIN, R., GALLER, M., MERLO, E. 1996. «Pattern Matching for Clone and Concept Detection». *Journal of Automated Software Engineering.*

[Kukich 1992] KUKICH, K. 1992. «Techniques for automatically correcting words in text». *ACM Computing Surveys*, 24(4):377–439.

[Lamb 1987] LAMB, D. A. 1987. «IDL: Sharing Intermediate Representations». *ACM Transactions on Programming Languages and Systems*, 9(3):297–318.

[Lea and Shank 1994] LEA, D. , SHANK, C. K. 1994. «ODL: Language Report». Technical Report Draft 5, Rochester Institute of Technology.

[Lehman 1980] LEHMAN, M. M. 1980. «Programs life cycles and laws of software evolution». *Proceedings of the IEEE*, 68(9):1060–1076.

[Letovsky 1986] LETOVSKY, S. 1986. *Cognitive Processes in Program Comprehension: First Workshop. E. Soloway and S. Iyengar eds.* Ablex Publisher.

[Lin 1998] LIN, D. 1998. «An Information-theoretic definition of similarity». In *the 15th ICML*, pages 296–304, Madison WI.

[Lorenz and Kidd 1994] LORENZ, M. , KIDD, J. 1994. *Object-Oriented Software Metrics.* Prentice-Hall, Englewood Cliffs, NJ.

[Maarek et al. 1991] MAAREK, Y., BERRY, D., KAISER, G. 1991. «An Information Retrieval Approach for Automatically Constructing Software Libraries». *IEEE Transactions on Software Engineering*, 17:800–813.

[Maarek et al. 1994] MAAREK, Y., BERRY, D. M., KAISER, G. E. 1994. «GURU: Information Retrieval for Reuse». *Landmark Contributions in Software Reuse and Reverse Engineering.*

[Maletic and Marcus 2001] MALETIC, J. I. , MARCUS, A. 2001. «Supporting Program Comprehension Using Semantic and Structural Information». In *Proc. of 23rd International Conference on Software Engineering*, pages 103–112, Toronto.

[Marcus and Maletic 2003] MARCUS, A. , MALETIC, J. I. 2003. «Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing». In *Proceedings of the International Conference on Software Engineering*, pages 125–135, Portland Oregon USA.

[Mayrand and Coallier 1996] MAYRAND, J. , COALLIER, F. 1996. «System Acquisition Based on Software Product Assessment». In *Proceedings of the International Conference on Software Engineering*, pages 210–219, Berlin.

[Mayrand et al. 1996] MAYRAND, J., LEBLANC, C., MERLO, E. 1996. «Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics». In *Proceedings of IEEE International Conference on Software Maintenance*, pages 244–253, Monterey CA.

[Mayrhauser and Vans 1993] MAYRHAUSER, A. V. , VANS, A. 1993. «From Program Comprehension to Tool Requirements for an Industrial Environment». In *Proceedings of IEEE Workshop on Program Comprehension*, pages 78–86, Capri Italy. IEEE Comp. Soc. Press.

[Mayrhauser and Vans 1994] MAYRHAUSER, A. V. , VANS, A. 1994. «Dynamic Code Cognition Behaviours for Large Scale Code». In *Proceedings of IEEE Work-*

shop on Program Comprehension, pages 74–81, Washington DC USA. IEEE Comp. Soc. Press.

[Mayrhauser and Vans 1996] MAYRHAUSER, A. V. , VANS, A. M. 1996. «Identification of Dynamic Comprehension Processes During Large Scale Maintenance». *IEEE Transactions on Software Engineering*, 22(6):424–437.

[McCabe 1990] MCCABE, T. J. 1990. «Reverse engineering reusability redundancy: the connection». *American Programmer*, 3:8–13.

[Merlo et al. 1993] MERLO, E., MCADAM, I., MORI, R. D. 1993. «Source Code Informal Information Analysis Using Connectionist Models». In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1339–1344, Chambery France.

[Miller 1963] MILLER, G. A. 1963. «The Magical Number Seven Plus or Minus Two: Some Limits on Our Capacity for Processing Information». *Psychological Review.*

[Miller 1975] MILLER, G. A. 1975. *The Magical Number Seven after Fifteen Years.* John Wiley & Sons.

[Moller and Paulish 1993] MOLLER, K. H. , PAULISH, D. J. 1993. *Software Metrics a Practicioner's Guide to Improved Product Development.* Chapman Hall, Boundary Row London SE1 8HN.

[Moon and Sproull 2000] MOON, J. , SPROULL, L. 2000. «Essence of Distributed Work: The Case of the Linux Kernel». Technical report, First Monday vol. 5 n. 11, http://firstmonday.org/issues/issue5_11/moon/index.html. Page accessed July 2003.

[De Mori 1998] MORI, R. D. 1998. *Spoken dialogues with computers.* Academic Press, Inc., Orlando, Florida 32887.

[Ney and Essen 1991] NEY, H. , ESSEN, U. 1991. «On Smoothing Techniques for Bigram-Bases Natural Language Modelling». In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, volume S12.11, pages 825–828, Toronto Canada.

[OMG 1991] OMG 1991. «The Common Object Request Broker: Architecture and Specification». Technical Report 91.12.1, OMG.

[Pearse and Oman 1995] PEARSE, T. , OMAN, P. 1995. «Maintainability Measurements on Industrial Source Code Maintenance Activities». In *Proceedings of IEEE International Conference on Software Maintenance*, pages 295–303, Opio-Nice France.

[Pennington 1987a] PENNINGTON, N. 1987a. *Comprehension Strategies in Programming. In: Empirical Studies of Programmers: Second Workshop. G.M. Olsen S. Sheppard S. Soloway eds.* Ablex Publisher Nordwood NJ, Englewood Cliffs, NJ.

[Pennington 1987b] PENNINGTON, N. 1987b. «Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs». *Cognitive Psychology*, 19:295–341.

[Pfleeger 2001] PFLEEGER, S. L. 2001. *Software Engineering: Theory and Practice.* Prentice-Hall, Englewood Cliffs, NJ.

[Pinhero and Goguen 1996] PINHERO, F. A. C. , GOGUEN, J. A. 1996. «An Object-Oriented Tool for Tracing Requirements». *IEEE Software, IEEE Computer Society Press*, pages 52–64.

[Pohl et al. 1997] POHL, K., DOMGES, R., JARKE, M. 1997. «Toward selective model-driven trace capture». In *Proc. of the 7th International Conference on Advanced Information Systems Engineering*, Barcellona.

[Potter et al. 1991] POTTER, D. L., SINCLAIR, J., TILL, D. 1991. *An Introduction to Formal Specification and Z.* Prentice-Hall, Englewood Cliffs, NJ.

[Ramesh and Dhar 1992] RAMESH, B. , DHAR, V. 1992. «Supporting Systems Development Using Knowledge Captured During Requirements Engineering». *IEEE Transactions on Software Engineering*, pages 498–510.

[Ramesh and Jarke 2001] RAMESH, B. , JARKE, M. 2001. «Towards References Models for Requirements Traceability». *IEEE Transactions on Software Engineering*, 27(1):58–93.

[Rich and Waters 1990] RICH, C. , WATERS, R. 1990. *The Programmer's Apprentice.* Addison-Wesley Publishing Company, Reading, MA.

[Rugaber and Clayton 1993] RUGABER, S. , CLAYTON, R. 1993. «The Representation Problem in Reverse Engineering». In *Proceedings of IEEE Working Conference on Reverse Engineering*, pages 8–16. IEEE Computer Society Press.

[Salton and Buckley 1988] SALTON, G. , BUCKLEY, C. 1988. «Term-Weighting Approaches in Automatic Text Retrieval». *Information Processing and Management*, 24(5):513–523.

[Scully 1993] SCULLY, M. C. 1993. «Augmenting Program Understanding With Test Case Based Methods». Technical Report SERC-TR-68-F, Software Engineering Research Center University of Florida Gainesville.

[Shneiderman and Mayer 1979] SHNEIDERMAN, B. , MAYER, R. 1979. «Syntactic/Semantic Interactions in Programmer Behaviour: A Model and Experimental Results». *IJCIS*, 8(3):219–238.

[Soloway and Ehrlich 1994] SOLOWAY, E. , EHRLICH, K. 1994. «Empirical Studies of Programming Knowledge». *IEEE Transactions on Software Engineering*, SE-10(5):595–609.

[Stone 1974] STONE, M. 1974. «Cross-validatory choice and assesment of statistical predictions (with discussion)». *Journal of the Royal Statistical Society B*, 36:111–147.

[STP 1996] STP 1996. *Software Through Pictures manuals.* Interactive Development Environments.

[Torvalds 1999] TORVALDS, L. 1999. «The Linux Edge». *Communications of the ACM*, 42(4):38–39.

[Turver and Munro 1994] TURVER, R. J. , MUNRO, M. 1994. «An Early Impact Analysis Technique for Software Maintenance». *Journal of Software Maintenance - Research and Practice*, 6(1):35–52.

[Vessey 1985] VESSEY, I. 1985. «Expertise in Debbugging Computer Programs: A Process Analysis». *IJMMS*, 23:459–494.

[Vicinanza et al. 1991] VICINANZA, S., MUKHOPADHYAY, T., PRIETULA, M. 1991. «Software-Effort Estimation: an Exploratory Study of Expert Performance». *Information Systems Research*, 2(4):243–262.

[Wilde and Casey 1996] WILDE, N. , CASEY, C. 1996. «Early Field Experience with Software Reconnaissance Technique for Program Comprehension». In *Proceedings of IEEE Working Conference on Reverse Engineering.*

[Wilde and Scully 1995] WILDE, N. , SCULLY, M. C. 1995. «Software Reconnaissance: Mapping Program Features to Code». *Journal of Software Maintenance - Research and Practice*, 7(1):49–62.

[Witten and Bell 1991] WITTEN, I. H. , BELL, T. C. 1991. «The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression». *IEEE Trans. Inform. Theory*, IT-37(4):1085–1094.

[Woods et al. 1999] WOODS, S., CARRIERE, S. J., KAZMAN, R. 1999. «A Semantic Foundation for Architectural Reengineering and Interchange». In *Proc. of the International Conference on Software Maintenance*, Oxford England.

APPENDICES

APPENDIX A

Albergate

Albergate is a software system developed in Java using a waterfall process. All the documentation required by the software development process is available for this system including requirement documents, design documents, test cases, etc.

Requirement	#Classes	Overlap
Req. 1	10	Req. 9 (4)
Req. 2	4	None
Req. 3	2	None
Req. 4	4	None
Req. 5	6	None
Req. 6	7	None
Req. 7	2	None
Req. 8	3	None
Req. 9	4	Req. 1 (4)
Req. 10	2	None
Req. 11	7	None
Req. 12	2	Req. 13 (1)
Req. 13	2	Req. 12 (1)
Req. 14	1	None
Req. 15	1	None
Req. 16	1	None

Table A.1: Albergate traceability matrix summary.

Albergate implements all the operations required to administer and manage a small to medium size hotel (room reservation, bill calculation, etc.). It was entirely developed by a team of final year students at the University of Verona, Italy, starting

from 16 functional requirements. These requirements, as well as all the other system documentation, is written in Italian. Albergate exploits a relational database (MySQL) and consists of 95 classes and about 20 KLOC.

To trace source code classes onto functional requirements, experiments had to focus on the 60 classes implementing the user interface of the software system. Other classes, automatically generated or extended from external frameworks such as panels, buttons, etc. had to be disregarded.

To validate the results, the original developers were required to provide a 16 $\times$ 60 traceability matrix linking each requirement to the classes implementing it. Most of the functional requirements were implemented by a small number of classes, on average, four classes with a maximum of 10. Most classes were associated with one requirement, only six were associated to two requirements, and eight were not associated with any functional requirement. The total number of links in the traceability matrix was 58.

APPENDIX B

Transient Meter

Transient Meter is a distributed measurement system for power quality monitoring. The system consists of nodes placed near monitored sites (e.g., generators, motors, industrial plants) and connected by a trigger circuit that detects power disturbance and by a digital acquisition board. These nodes detect power transients and process them by performing classification and measurements. A central node retrieves data from distributed nodes and stores it in a database.

1. The system was developed using a RAD tool (*Borland C++ Builder*^{TM}); therefore it contains both totally-automatic generated classes (e.g., produced by report generators) and partially generated classes containing some identifiers (e.g., user interface classes).

2. The system reused existing class hierarchies (e.g., to read and write data from/to `.wav` files). Moreover, COTS components were used for handling complex signal visualizations, implementing some user interface widgets typical of a measurement instrument (LED, switches, seven-segment displays, etc.), and handling Fourier transforms.

3. Communication between the central node and the distributed nodes was implemented by a CORBA architecture.

The SRS document contains 18 requirements, and the source code consists of 63 classes (about 25 KLOCs). The analyzed subsystem corresponding to the central

node was developed starting with 13 requirements and contained 36 classes for a total of about 15 KLOCs. Three classes were automatically generated by the RAD tool, seven were reused classes, four were design-level classes, and three belonged to the CORBA architecture. Other classes generated by the IDL compiler to define data structures exchanged between distributed objects (identified by a well-known postfix: `_var`, `_forany`), were disregarded.

APPENDIX C

Library Management Software

The library management software was developed for a university library. The system consists of two modules:

- A client application, basically a GUI front-end used by dedicated library personnel.
- A web application, accessible to all users to check book availability.

The traceability recovery experiments were performed on the first application developed from 10 functional requirements. It consists of 35 Visual Basic files (about 16 KLOCs). The intent of the experiment, in this case, was to map requirement onto Visual Basic files. Each file, on its own, may be:

- A frame (having extension `.frm`) implementing a window of the system, containing the layout of the window, and the event handlers of the widgets (contained into the window) and of the window itself.
- A module composed of some procedures or functions implementing specific functionalities.

APPENDIX D

LEDA

Release	2.1.1	3.0	3.1.2	3.2.3	3.4	3.4.1	3.4.2	3.5.2	3.7.1
KLOC	35	34	61	69	95	100	111	123	153
Classes	69	109	176	178	208	211	210	235	410
Methods	1649	2388	3519	3695	4967	5104	5197	6124	10260
Attributes	201	245	346	371	510	543	589	740	1177
Associations	96	116	180	186	272	275	278	336	537
Aggregations	5	16	47	52	114	118	142	181	410
Generalizations	10	43	87	87	98	99	98	114	206

Table D.1: Main features of available LEDA releases.

LEDA is a library of foundation classes developed and distributed by Max-Planck-Institut für Informatik, Saarbrücken, Germany (freely available for academic research and teaching from http://www.mpi-sb.mpg.de/LEDA/). The LEDA project started in 1988 and a first release of the library (1.0) was available in 1990. Since then there have been several other releases, the latest (3.7.1) in 1998. Table D.1 shows the main characteristics of the different releases obtained by static analysis of the source code. The table shows that LEDA evolved considerably from the first to the last release (from 35 to 153 KLOC and from 69 to 410 classes). The LEDA library provides implementation of the most common structures such as lists, stack, heap, trees, graphs as well as geometrical entities such as points, segments, polygons, etc.

LEDA is documented with a detailed user manual (the 3.4 release user manual has of 258 pages) which summarizes the concepts and describes in detail the implemented

functionalities and classes interfaces, and provides examples of usage. Often, more than one page describes the same structure e.g., the graph class takes about 10 pages. The manual can thus be thought of as organized into 88 *logical* sections or *logical* manual pages.

APPENDIX E

DDD

Release	2.0	2.1b1	2.1.90	2.2.3	2.99	3.0.90	3.0.91	3.1.3
Previous	2.0b3	2.0	2.1.1	2.2.2	2.2.3	3.0	3.0.90	3.1.2
KLOC	89	103	102	127	139	165	167	170
Classes	123	125	130	128	130	134	135	135
Methods	467	497	547	398	470	552	557	562
Attributes	2020	2122	2239	2306	2416	2710	2721	2747
Associations	147	148	158	80	89	102	102	103
Aggregations	191	197	199	198	201	213	215	215
Generalizations	72	74	77	76	76	80	80	80

Table E.1: Main features of relevant DDD releases.

DDD is a graphical user interface to GDB and DBX, which are the popular UNIX debuggers. They were developed at the Technische Universität Braunschweig, Germany, protected by the GNU general public license, and are available from http://www.cs.tu-bs.de/softech/ddd/. In particular, 31 different releases, ranging from 2.0beta1 to 3.1.3 were analyzed. Table E shows the main characteristics of several selected DDD releases. For the sake of space, only releases that exhibit major changes with respect to their immediate predecessor are shown in the table.

DDD seems to have evolved smoothly. From the first to the last release the code doubled (from 52 to 170 KLOC of C++) and the number of classes did not change very much (from 123 in the first release analyzed to 135 in the last).

It should be noted that the main DDD site stores and gives access to almost the entire DDD evolution; only the oldest releases are no longer available.

APPENDIX F

AOL

```
Meta-characters extended BNF notation:

        {}   means zero or more times
        []   means an optional element, so 0 or 1 time
        ""   means a terminal symbol
        |    means the boolean symbol OR
        ()   means a block of elements, useful to group them

AOL_design_description::= list_AOL_declarations
list_AOL_declarations ::= {AOL_decl ";"}
AOL_decl              ::= class
                        | association
                        | generalization
                        | aggregation

/*------------------------- CLASSES ------------------------*/

class             ::= CLASS class_name scope
                        [ATTRIBUTES attribute_list]
                           [OPERATIONS operation_list]
class_name        ::= id
scope             ::= "{" (EXTERNAL | ABSTRACT) "}"
attribute_list    ::= [attribute {"," attribute}]
attribute         ::= visibility [SHARED]
                        attribute_name ":" type
visibility        ::= PUBLIC
                    | PRIVATE
                    | PROTECTED
                    | UNDEF_SCOPE
attribute_name    ::= id
operation_list    ::= [operation {"," operation}]
operation         ::= visibility [SHARED] operation_name
                        "(" operation_arg_list ")" [":" type]
                           [an_annotation]
operation_name    ::= id
opeation_arg_list::= [argument {"," argument}]
argument          ::= arg_name [":" type]
an_annotation     ::= "{" (ABSTRACT | other_annotations) "}"
other_annotations::= string

/*------------------------- RELATIONS  ---------------------*/

association       ::= RELATION [relation_name] ROLES roles
                        [ATTRIBUTES attribute_list]
                           [IS_A_CLASS assoc_class_id_ref]
```

```
relation_name     ::= id
roles             ::= role "," role {"," role}
role              ::= [NAME role_name] CLASS class_id_ref
                        MULT multiplicity
                          [QUALIFIER qualifier_name]
role_name         ::= id
class_id_ref      ::= id_ref
multiplicity      ::= string | ONE | MANY
                        | ONE_OR_MANY  | OPTIONALLY_ONE

qualifier_name    ::= string
assoc_class_id_ref ::= id_ref

aggregation       ::= AGGREGATION [NAME aggregation_name]
                        CONTAINER role PARTS part_roles
aggregation_name  ::= id
part_roles        ::= role {"," role}

generalization    ::= GENERALIZATION
                       [DISCRIMINATOR discriminator_name]
                        super_class_id_ref SUBCLASSES
                          sub_classes_ids

discriminator_name ::= string
super_class_id_ref ::= id_ref
sub_classes_ids   ::= sub_class_id {"," sub_class}
sub_class         ::= id

id                ::= IDENTIFIER
id_ref            ::= IDENTIFIER
```

APPENDIX G

Industrial Software Characteristics

Design and code of industrial software for telecommunications was provided by Sodalia SpA, Trento, Italy. 29 C++ components (about 308 KLOC) were available for the experiments. The components were developed following an incremental process model and the final release of each component, together with the final design, was analyzed and processed. All components were developed using C++.

	Design		Code				Design		Code		
Comp.	Classes	Attr.	Classes	Attr.	LOCs	Comp.	Classes	Attr.	Classes	Attr.	LOCs
C1	38	280	53	1229	27398	C16	17	49	8	105	6116
C2	17	374	16	416	19863	C17	6	29	6	64	438
C3	13	134	17	216	6190	C18	12	346	15	335	20781
C4	113	438	103	1227	42781	C19	7	143	11	159	9913
C5	7	82	26	188	15031	C20	9	131	8	233	9412
C6	9	0	5	101	3428	C21	2	40	5	58	2561
C7	29	139	21	496	21335	C22	9	25	4	26	2532
C8	35	109	44	957	21796	C23	14	33	6	52	2514
C9	29	329	39	644	11639	C24	16	219	16	229	9680
C10	24	165	28	244	15319	C25	3	28	3	39	1422
C11	18	260	18	249	14297	C26	9	57	6	55	3059
C12	12	159	11	198	16847	C27	19	174	13	118	4922
C13	1	19	2	50	2619	C28	7	54	6	51	2858
C14	3	15	2	13	1081	C29	7	60	5	64	1447
C15	12	0	7	174	11028						

Table G.1: Classes and attributes in the design and code for each component.

Table G.1 gives the number of classes and attributes in each analyzed component resulting from the design and code. LOCs figures are shown in the last column under the *Code* heading.