國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science & Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

基於開源 C++ 專案的服務元件建構

Constructing C++ -Based Service Components From Open Source Projects

梁峻瑞

Chun-Jui Liang

指導教授： 李允中 博士

Advisor: Jonathan Lee, Ph.D.

中華民國 112 年 7 月

July, 2023

doi:10.6342/NTU202303414

國立臺灣大學碩士學位論文
口試委員會審定書

MASTER'S THESIS ACCEPTANCE CERTIFICATE
NATIONAL TAIWAN UNIVERSITY

基於開源 C++專案的服務元件建構

Constructing C++ -based Service Components from Open Source Projects

本論文係梁峻瑞君（學號 P10922002）在國立臺灣大學資訊工程學系完成之碩士學位論文，於民國 112 年 7 月 28 日承下列考試委員審查通過及口試及格，特此證明。

The undersigned, appointed by the Department of Computer Science and Information Engineering on 28 July 2023 have examined a Master's thesis entitled above presented by LIANG, CHUN-JUI (student ID: P10922002) candidate and hereby certify that it is worthy of acceptance.

口試委員 Oral examination committee:

李祈均	郭忠義	劉建宏
（指導教授 Advisor）		
陳宗禧	李文瑞	

系主任/所長 Director: 洪士灝

誌謝

首先，我要感謝我的指導教授李允中教授，這兩年以來的教導我許多知識以及做研究的方法：尋找研究的主題、相關研究的探討、問題的發掘、以及解決問題的方法。再來，我要感謝臺灣大學軟體工程實驗室的成員，陳力聖、林怡伶、許恆、劉仁軒、林辰臻、張馨尹、錢怡君等，在一些共同的專案中一同努力、互相合作。由於許多人的幫助，讓我得以完成此篇論文

doi:10.6342/NTU202303414

摘要

在現今的系統開發中，單體式架構(Monolithic)一直是主流的開發方法，但以單體式方式來開發系統，相對也會產生許多缺點，例如：後期開發成本高、程式碼依賴性高...等。然而在系統完成後，我們也會需要對系統長期的維護及持續會有新需求的開發，在新的需求一直進來時，也會導致系統變得更大也更複雜，所以在面臨這些後續開發上的問題，我們必須有個方法來防止或降低以上單體式架構所可能產生的問題。

因此在本研究中，我們提出了一個流程能夠自動化將開源 C++ 專案轉變成爲服務元件，目的是讓單體式的架構能切分到以方法爲單位，讓整個系統可以更有效維護及重組服務，此流程包含以下步驟:辨識程式間的相依關係;從原始碼解析方法資訊;切分程式主體架構;重新組裝服務元件。

關鍵詞 — 服務元件、程式碼生成、網路應用程式架構

doi:10.6342/NTU202303414

Abstracts

In modern system development, monolithic architecture has been the mainstream approach. However, developing systems using a monolithic architecture can lead to several drawbacks, such as higher post-development costs and increased code dependencies. As the system reaches completion, long-term maintenance and continuous development to meet new requirements become necessary. As new requirements keep coming in, the system can become larger and more complex, posing challenges for subsequent development. Therefore, in the face of these challenges, we need a method to prevent or reduce the potential issues arising from a monolithic architecture.

Therefore, in this research, we propose a process that can automatically transform open-source C++ projects into service components. The goal is to break down the monolithic architecture into individual methods, allowing the entire system to be more efficiently maintained and reassembled as services. This process includes the following steps: identifying interdependencies between the code, parsing method information from the source code, partitioning the program's main structure, and reassembling the service components.

doi:10.6342/NTU202303414

Index terms — Service component, Code generation, web application framework

doi:10.6342/NTU202303414

Contents

口試委員審定書 i

誌謝 ii

摘要 iii

Abstracts iv

List of Figures ix

List of Tables xi

Chapter 1 Introduction 1

Chapter 2 Related Work 3

2.1 Background Work 3

2.1.1 Clang AST 3

2.1.2 Dependency 3

2.1.3 Web Framework 4

doi:10.6342/NTU202303414

2.2 Related Work 4
2.2.1 Soot 4
2.2.2 Reveal 5
2.2.3 Repack 5

Chapter 3 Service Component from C++ Open Source 6

3.1 System Architecture 7
3.2 Generate Dependency Graph 8
3.2.1 Clang - Create AST Tree 8
3.2.2 CSoot - CPP Code Parser 9
3.2.3 CReveal - Mapping Relation 11
3.3 Code Splitting 14
3.3.1 Code Struture From AST 16
3.3.2 Code Splite 19
3.4 Code Body Database 20

Chapter 4 Repack Service Components 25

4.1 Web Framework 25
4.2 Invoker Tracing 27
4.3 Method Dependency Collection 29
4.4 Code Body Combination 30
4.5 Output 32

doi:10.6342/NTU202303414

Chapter 5 Demonstration **33**

5.1 Controller Repack Demonstration 35

5.2 UserService Repack Demonstration 37

5.3 ItemService Repack Demonstration 38

Chapter 6 Conclusion **39**

6.0.1 Benefits 39

6.0.2 Future Work 40

Bibliography **41**

doi:10.6342/NTU202303414

List of Figures

3.1 System Architecture 7

3.2 Clang AST 9

3.3 CSoot Structure 10

3.4 CSoot Dump 11

3.5 Dependency Mapping 13

3.6 Dependency Graph 14

3.7 Code Body AST 16

3.8 Code Splitting Class Diagram 18

3.9 Code Body Splitting 19

3.10 Code Body Into DB 21

3.11 Code Body Schema 23

4.1 Repack Controller Dependency 26

4.2 Repack Controller 27

4.3 Repack Invoker Example 28

doi:10.6342/NTU202303414

4.4 Repack Adjacency List 29
4.5 Repack DFS 30
4.6 Repack Ascending 31
4.7 Repack Output 32

5.1 Demo Example Code Structure 34
5.2 Repack Demo Output 35
5.3 Repack Controller 36
5.4 Repack UserService 37
5.5 Repack ItemService 38

doi:10.6342/NTU202303414

List of Tables

3.1 Dependency Table 12

3.2 Code Body Sturcture 15

doi:10.6342/NTU202303414

Chapter 1

Introduction

服務導向架構(Service Oriented Architecture)是常用的一種軟體開發模式，能利用統一的介面並透過共同的通訊標準整合各個服務元件，達成重複使用服務元件及降低元件之間耦合性(loose coupling)的目的。在之前學長們的努力下已經完成了 Java 版本的服務產出，而在這裡，我們在 Shih [7] Java 單元服務的流程基礎上，開發了 C++ 版本的服務元件。

但也因爲程式架構的不同，處理方法上也會有所不同，一樣的部分我們也採用相依(Dependency)作爲開發上的第一步，了解程式的方法(Method)之間的相依關係(Dependency Relation)，對應到之前實驗的工具及流程架構，重新撰寫了 C++ 版本。

在原先 Java 架構中使用到了 1. Soot - 作爲程式解析的工具 2. Reveal - 產生相依圖(Dependency Graph) 3. Repack - 對服務單元重新組裝 等相關的研究套件，而以上這些部分，我們也開發出 C++ 版本作爲我們產生 C++ 服務單元的的工具。

doi:10.6342/NTU202303414

1. CSoot - 程式解析抽取資訊.
2. CReveal - 產生相依圖.
3. Code Splitting - 切分程式主體(body).
4. CRepack - 服務單元重組.

在以上我們所開發的工具中，第一步透由 CSoot 的解析了解程式的整體資訊包含 Class 、 Field 、 Method 、 Stmt 、 Expr，第二步再來到 CReveal 我們把這些資訊對應出我們所定義的相依圖(Dependency Graph) ， 第三步 Code Splitting 將程式主體依照我們定義的架構做切分，在存入資料庫中，最後 CRepack 依照 Web Framework Controller 爲單位重新建立出以Controller 爲服務的獨立可執行集合。

經由以上的流程，我們可以將一個單體架構的專案，切分成以爲單位重新建立出以 Controller 服務爲單位，在未來也可以在用在建立微服務(Microsevice)上，包含其他程式語言的專案也一併套入。

doi:10.6342/NTU202303414

Chapter 2

Related Work

2.1 Background Work

2.1.1 Clang AST

此研究上，我們以抽象語法樹(Abstract Syntax Tree)作爲開發上的基礎，研究中我們選用 Clang 爲專案生成抽象語法樹的主要編譯器，Clang [1] 是 LLVM [4] 的前端，可以用來編譯 C、 C++ 、 OBJECT-C 等語言。在抽象語法樹裡有非常多程式相關的節點資料，包含所有程式碼中的 Class 、 Field 、Method 、 Stmt 、 Expr，都會是我們之後再解析流程上會需要提取的資料內容。

2.1.2 Dependency

Dependency 指的是程式之間會依賴到其他程式的物件(Object)或方法(Method)，例如方法的被呼叫，及參數的被讀取及被寫入，都會整個專案裡程式跟程式之間

doi:10.6342/NTU202303414

重要的相依關係，而在我們先前的研究中，也定義了多種的相依種類，在此研究中我們又再延伸到85種的相依關係，再者我們會找出所有的相依關係在產出對應的相依圖(Dependency Graph)。

2.1.3 Web Framework

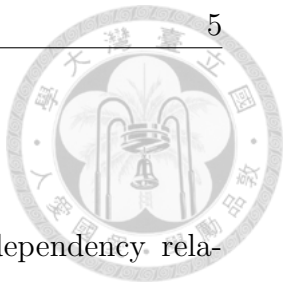
Crow [3] 是一個輕量級的 C++ 網頁應用程式框架，用於快速開發高效的網路應用。它被設計爲簡潔且易於使用，提供了基本的路由、請求處理和模板渲染等功能，並支援 RESTful 架構。Crow 是一個單項標頭 (header-only) 的框架，這意味著您只需要包含一個頭文件即可使用，無需編譯或鏈接額外的程式庫。

2.2 Related Work

2.2.1 Soot

Soot [5] 是一個針對 Java 及 Android 應用程式的程式碼靜態分析框架，其中 他的子模組: Dexpler 能將 Android APK 的 Dalvik bytecode 反組譯，進而將之轉 爲 Soot 的中間表示格式:Jimple，Scene表示整個程式，SootClass對 應到程式中的類別，SootMehod，對應到程式中的方法，SootField對應到程式中 的屬性，JimpleBody 表示 Method 中實際內容，由Locals、Traps、Units 組成，其中 Locals 代表方法中的區域變數， Traps 代表方法中例外處理的部份， Units 代表方 法中描述句。我們可以使用 Soot 提供的 API 去了解程式的全貌，進而去分析、找出程式元素與他們之間的關係，最後產生相依圖。

doi:10.6342/NTU202303414



2.2.2 Reveal

Reveal 作爲分析程式元素(program element)間相依關係(dependency relation)的工具,利用 Soot 將 Java bytecode 轉爲中間語言(Internal Rep- resentation) Jimple,再對 Jimple 進行原始碼分析,歸納出 10 種程式元素(program element)及 85 種程式關係(program relation),最終生成原始碼的 依賴圖(dependency graph)。

2.2.3 Repack

Repack 的主要目的是收到使用者所指定的 service components 之後，透過一些重組的流程來產生一個能夠穩定執行這些service components的集合，且這個集合爲能夠執行的最小集合，也就是集合裡面不會包含任何使用者用不到的檔案或是方法(Method)。

doi:10.6342/NTU202303414

Chapter 3

Service Component from C++ Open Source

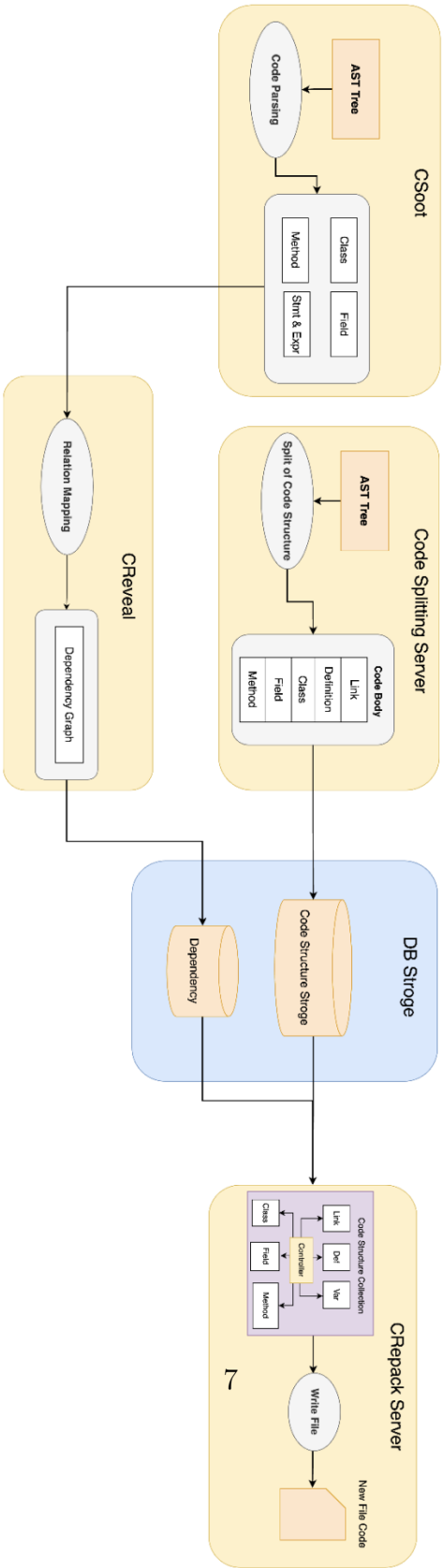
在此篇章節的主要任務，我們需要對整個專案取出服務單元，而取出的過程我們會依照先前所開發的套件，以下列出我們所開發的套件。

在系統架構圖中，我們將整個架構分爲

1. CSoot - 程式解析抽取資訊.
2. CReveal - 產生相依圖.
3. Code Splitting - 切分程式主體(body).
4. CRepack - 服務單元重組.

doi:10.6342/NTU202303414

3.1 System Architecture



doi:10.6342/NTU202303414

Figure 3.1: System Architecture

從整體系統架構圖中，我們會先從 CSoot 開始，可以看到裡面我們會先產出 AST，經由 CSoot 的解析，最後可以拿到 Class 、 Field 、 Method 、 Stmt 、 Expr 資訊， 下一步到了 CReveal 階段，拿到程式的相關資訊，經過 Mapping Relation 得到了相依圖，在將相依圖存到資料庫中，Code Splitting 也會經由 AST 給的資訊，例如節點資料及行數資料，用來切分程式主體，這些也會存到資料庫供 CRepack 使用，來到最後資一步，CRepack 將從 Controller 作爲起始點，將 Controller 所 Invoker 到的方法(Method)收集起來包括它們的相依，最終組裝再一起產出一個獨立可執行的服務集合。

3.2 Generate Dependency Graph

3.2.1 Clang - Create AST Tree

在產生抽象語法樹前我們必須確保專案是可以建立成功，如果不成功抽象語法樹則會建立失敗，當確認完後，使用 Clang 產生出整個專案所對應的抽象語法樹，從以下的展示可以知道抽象語法樹包含了非常多資訊。

doi:10.6342/NTU202303414

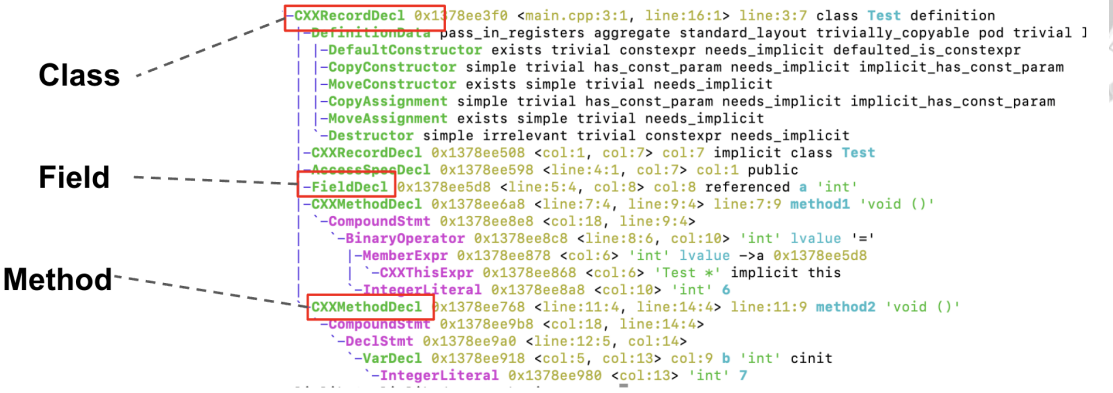


Figure 3.2: Clang AST

[Figure 3.2] - Clang AST 從圖中可以看出，針對 AST 給的相關資訊，以下爲範例程式的 AST 產出，CXXRecordDecl 代表的是 Class 相關的節點資料，FieldDecl 代表的是 Field 的節點資料 ， CXXMethodDecl 代表的是 Method 的節點資料，當然除了這些還有非常多像 Stmt 或 Expr 等重要資訊可以解析。

3.2.2 CSoot - CPP Code Parser

CSoot 是我們參考 Java Soot [5]的架構下來開發的，其中我們參考了 Soot 所定義的主體架構 Class、Field 、 Method 、 Stmt 、 Expr，我們可以清楚了解整個程式物件(Object)跟物件(Object)之間的關係及方法(Method)跟方法(Method)之間的呼叫(Invoker)關係。當 Clang 產生抽象語法樹後，我們透由 CSoot 來解析整顆抽象語法樹的節點(Node)資料，最終把這些資料收集起來再交由 CReveal 使用來產生相依圖。

當抽象語法樹產生完後，我們可以使用 CSoot 來解析我們之前所產生的 AST，將我們所需要的資訊收集起來，這其中包含了重要的 Class 、 Field 、

doi:10.6342/NTU202303414

Method 再者還有 Stmt 、 Expr，同時將這些資訊存入到 CSoot 裡。

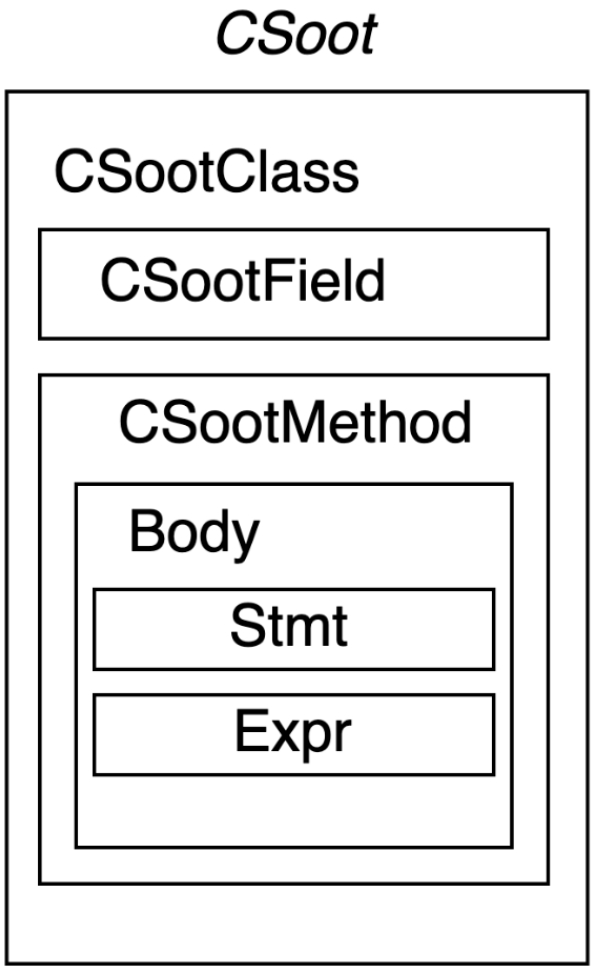


Figure 3.3: CSoot Structure

[Figure 3.3] - CSoot Sturcture 從以上的 CSoot 架構中可以發現，整體來說是將程式對於 Class 、 Filed 、 Mehtod 的外到內作爲一個容器，也就是代表我們可以輕易地了解要獲取相關資訊的規則，好比 Method 一定會在 Class 裡 Filed 也會在 Class裡，所以在使用 CSoot 可以很直觀的下 class.field 或 class.methed 取得相

doi:10.6342/NTU202303414

關的節點資訊。

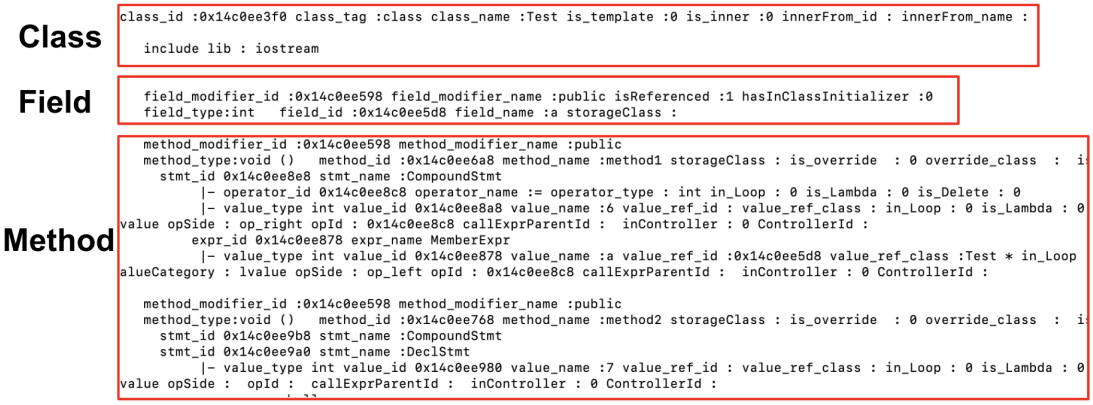


Figure 3.4: CSoot Dump

[Figure 3.4] - CSoot Dump 最後我們可以使用 Dump ， 將 CSoot 所有我們需要的節點資料展示出來，同樣的從最外部的 Class 依序可以看出內部的 Field 資料，及 Mehtod 裡的資料，圖中也可以發現 Method 內部裡也會將 Stmt 、 Expr 等相關資訊也存入其中。

3.2.3 CReveal - Mapping Relation

CReveal是我們參考實驗室 Java Reveal [6] 而開發 C++ 的版本，CReveal 是作爲分析程式元素(program element)間相依關係(dependency relation)的工具，我們利用 CSoot 取得解析過後程式的 Class 、 Field 、 Method 、 Stmt 、 Expr 等資料， CReveal 再利用訪問者模式 (Visitor Pattern) 對85種相依(Dependency)關聯起來，最後則在產出相依圖。我們也會將這些資料存入資料庫中。

doi:10.6342/NTU202303414

Encapsulation	Encapsulation	Abstraction	Delegation
include	***lambda***	***abstraction***	***access union***
include-lib	lambda-return	extends-N	read-unionfield
class	lambda-perform-op	***override***	calls-unionmethod
field-in-class	lambda-use	override	***access attribute***
method-in-class	lambda-write	override-virtural-method	access-nonStatic-nonConstant-attribute
nestedClass-of-class	lambda-flow-info	***overload***	access-Static-nonConstant-attribute
struct	lambda-loop-read	overload-nonStatic-method	lambda-access-nonStatic-nonConstant-attribute
field-in-struct	lambda-loop-write	overload-Static-method	lambda-access-Static-Constant-attribute
method-in-struct	lambda-return-type	**Delegation**	***creation***
nestedClass-of-struct	lambda-argtype-N	***invoker method***	creation-class
union	lambda-tparams-N	order-N	eager-creation-class
field-in-union	lambda-attribute-N	lambda-order-N	creation-struct
method-in-union	***method operation***	***invoker concreate method***	eager-creation-struct
nestedClass-of-union	perform-op	invoker-nonStatic-method	creation-union
template	perform-op-overload	invoker-Static-method	eager-creation-union
field-in-template	static-cast	lambda-invoker-nonStatic-method	creation-lambda
method-in-template	dynamic-cast	lambda-invoker-Static-method	***reference***
modifier	reinterpret-cast	***delegation***	lvalue-reference-type
access-modifier	const-cast	delegation	rvalue-reference-type
non-access-modifier	returns	self-delegation	***pointer***
field	use	recursive-delegation	pointer-type
initialized-by	writes	receiver	null-pointer-type
gets	delete	***access class***	pointer-to-member
read-before	flows-info	read-classfield	point-to
written-before	loop-read	calls-classmethod	
field-type	loop-write	***access struct***	
	return-type	read-structfield	
	argtype-N	calls-structmethod	
	data type		
	element-type		
	expression-type		

Table 3.1: Dependency Table

[Table 3.1] - Dependency Table 經由學長們的研究，在原本的 Java 相依圖中，我們在 C++ 的架構下又擴展到85種相依，然而這其中也因爲程式架構的不同也做了相關上的修改，好比 inner Class 及 import 等非 C++ 體系下的相依也一併拿掉，另外我們也把 Lambda 的相依也加入進來，還包含了 struct 、 union 、 templete 這些屬於只有 C++ 的相依加進來，讓整個相依資訊變得更完整。

doi:10.6342/NTU202303414

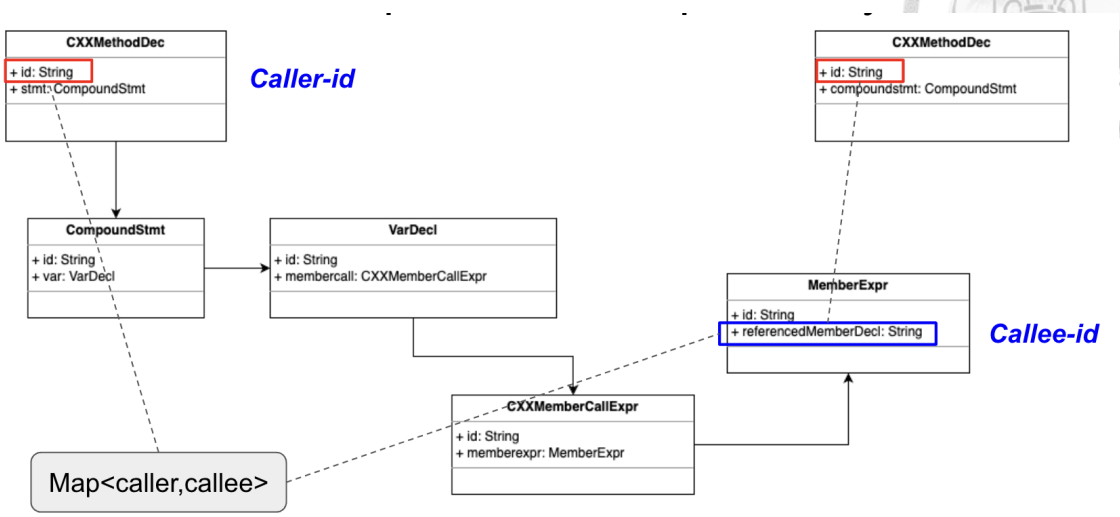


Figure 3.5: Dependency Mapping

[Figure 3.5] - Dependency Mapping 在我們定義出了相依關係後，再來就會需要把它們關聯起來，從上圖的 AST 資訊中可以發現，Invoker 的節點資料必定會存在被 Invoker 的資訊，這也是我們需依賴 AST 的主要原因，它會清楚的告訴我們每個 Method 或是 Filed Invoker 到哪個 Object 的資訊，而我們只需將這些資訊存入到 Map 中，儲存的值也就是 Caller id 跟 Callee id。

doi:10.6342/NTU202303414

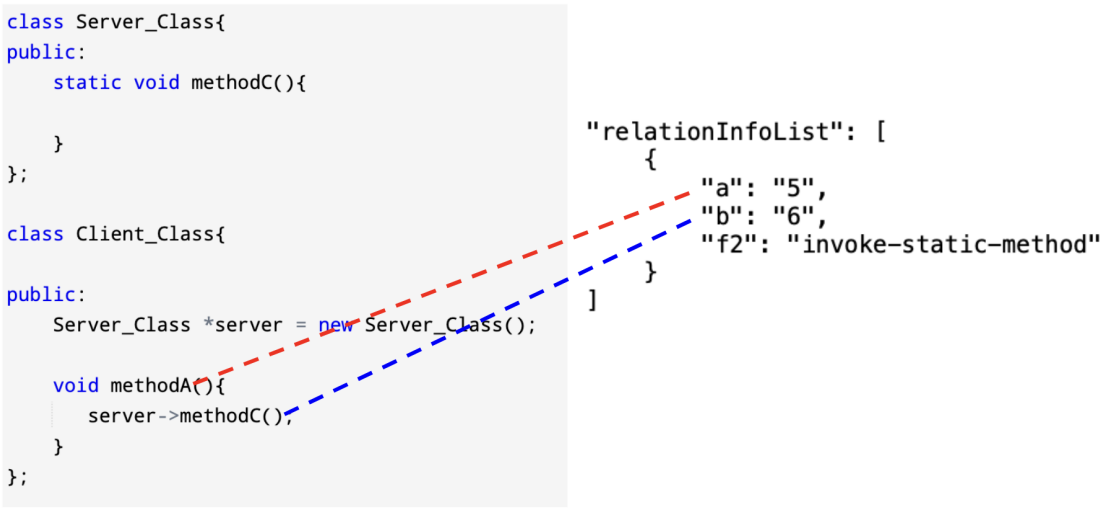


Figure 3.6: Dependency Graph

[Figure 3.6] - Dependency Graph 最後經由以上的步驟，我們產出的相依圖(Dependency Graph)就會是以上的樣子，左邊爲一個範例程式，右邊則爲 JSON 來表達的相依圖，”a” 代表著 caller ， ”b” 代表著 callee ， ”f2” 代表的是所屬的相依關係。當然實際上不會只有這些，只要我們能定義出所有的相依關係，整體的相依圖也會越完整，這也會是往後研究中會需要增加的一部分。

3.3 Code Splitting

Code Splitting 的主要目的是執行切割整個專案程式碼，目的是讓未來可以重組新的服務單元，而每個切割的主體包含了以下定義的 Link 、 Definition 、 Global 、 Class 、 Field 、Method 、 Function ，最後將這些切出來的主體依序存入資料庫中。方便以後可重組服務單元。

doi:10.6342/NTU202303414

在Code Splitting 的主要任務，是將程式碼依照我們所定義的程式主體來做切分，以下我們定義出程式主體的7個種類別，在使用訪問者模式 (Visitor Pattern) 來進行程式主體的切割。

1. Link
2. Definition
3. Global
4. Function
5. Class
6. Field
7. Method

當我們切割完以上的主體後，則會將主體(Body)新增到資料庫中。

Link	Definition	Global	Class	Method & Function	Field
#include	#define	bool	class	void	"***Fundamental***"
using namespace	typedef	char	sturct	auto	"***User-defined***"
	#undef	char8_t	union	"***Fundamental***"	"***Pointers***"
	#ifdef	char16_t	template	"***User-defined***"	"***References***"
	#ifndef	char32_t		"***Pointers***"	"***Arrays***"
	#if	const		"***References***"	
	#elif	double		"***Arrays***"	
	#else	float			
	#endif	int			
	#error	long			
	#pragma	short			
		signed			
		static			
		unsigned			
		wchar_t			

Table 3.2: Code Body Sturcture

doi:10.6342/NTU202303414

[Table 3.2] - Code Body Structure 以上圖是我們定義的 C++ 架構主體，代表著我們將會以這些定義出來的主體來分類需要被切割的主體，其中 Link 代表像是 include 函式庫相關的主體， Definition 代表的是 preprocessor 相關的主體，但因爲 preprocessor 爲產生 AST 前被處理掉，所以這部分是以 regex 來處理，Class 類別也包括了 class 、 sturct 、 union 的主體，Method 和 Funtion 則是因爲 AST 都爲 CXXMethodDecl 所以節點是一樣的，最後 Filed 屬性也會切分出來。

3.3.1 Code Struture From AST

在切割程式主體之前，我們依照上面所定義的主體種類，在從抽象語法樹的節點資料取得該主體的程式碼行數資料，在進行切割，每個主體都分別代表某一個抽象語法樹的節點(Node)。

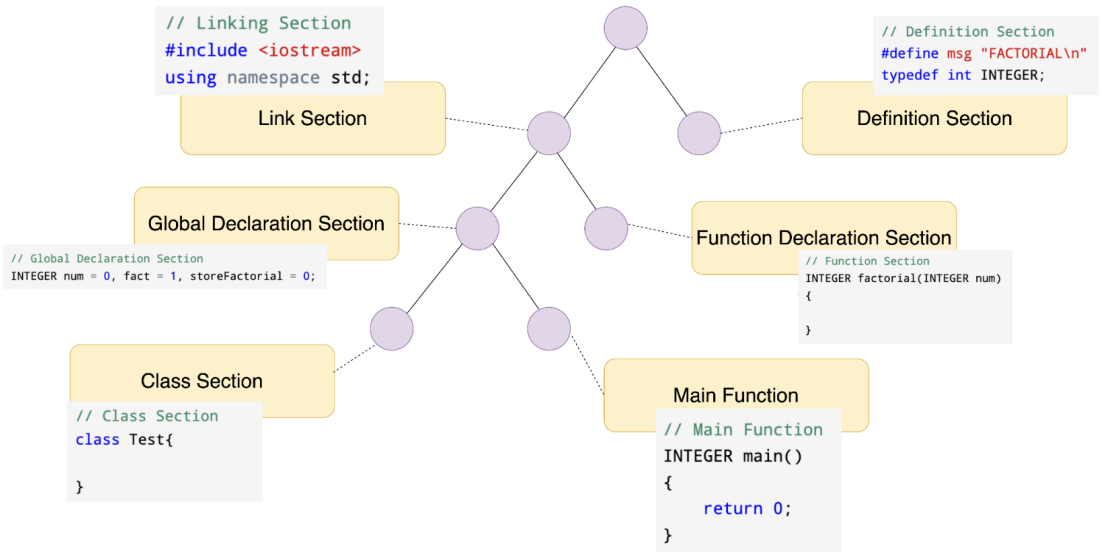


Figure 3.7: Code Body AST

doi:10.6342/NTU202303414

[Figure 3.7] - Code Body AST 在我們需要對程式碼做切割前，最主要的方法是先了解 AST 整體的架構，其中包括我們要切的主體是屬於那種 AST 表示的方法，但因爲 AST 的節點數非常多及表達的資訊也非常多，所以這部分也是我們需將我們定義出來的主體架構跟 AST 做一個對應，目的是了解我們所需要的主體分別對應到 AST 裡的哪些節點，再依序取出來。

doi:10.6342/NTU202303414

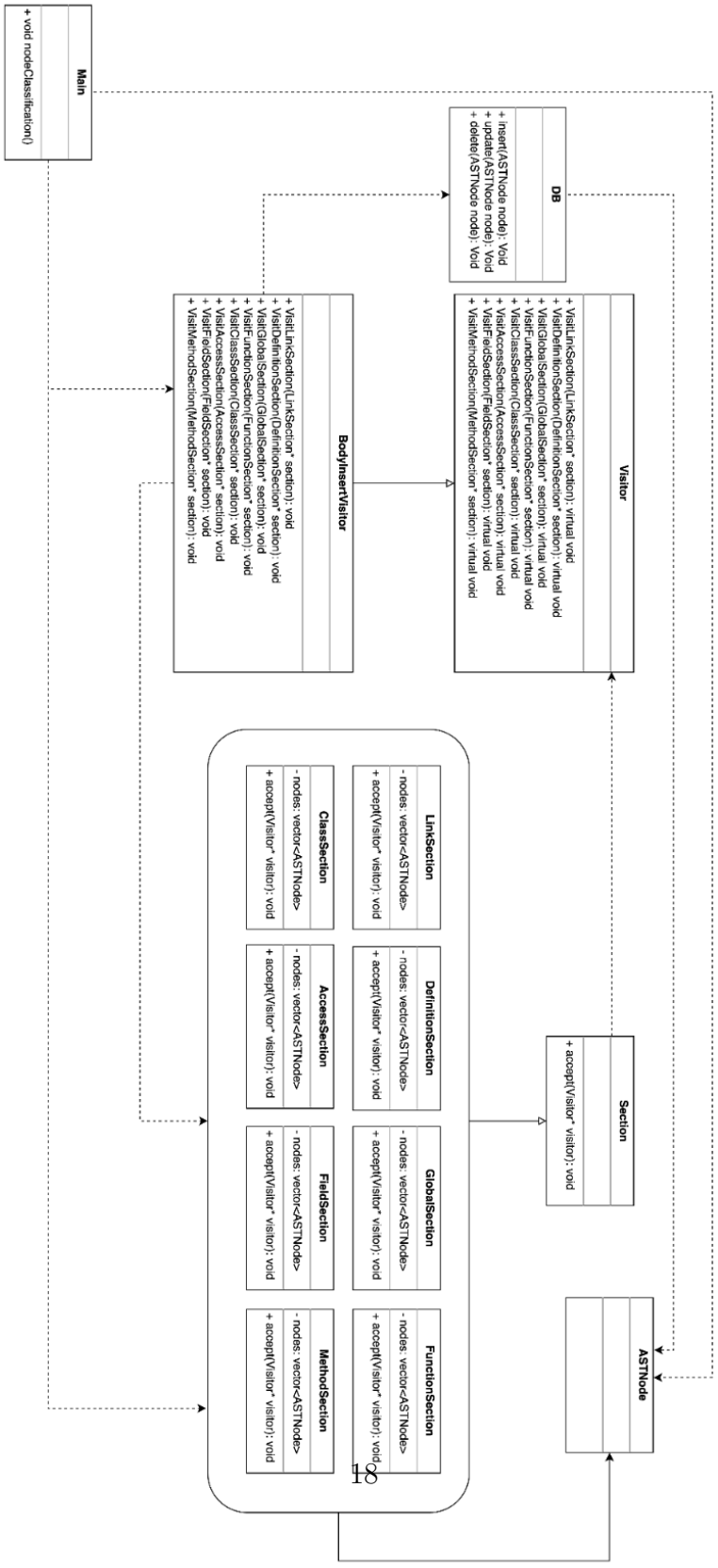


Figure 3.8: Code Splitting Class Diagram

doi:10.6342/NTU202303414

[Figure 3.8] - Code Splitting Class Diagram 在上圖 Class Diagram 我們定義了一個 visitor ， element 分別是我們之前定義的 Link 、 Definition 、 Global 、 Class 、 Field 、 Method，針對這些主體，我們將會把 AST 節點資料都走訪，只要是找到我們對應的節點，就會將它們存入資料庫中。

3.3.2 Code Splite

當我們知道要切割的節點後，我們將開始進行切割，將會依照我們需要的主體類別依序取出我們需要的主體資訊。

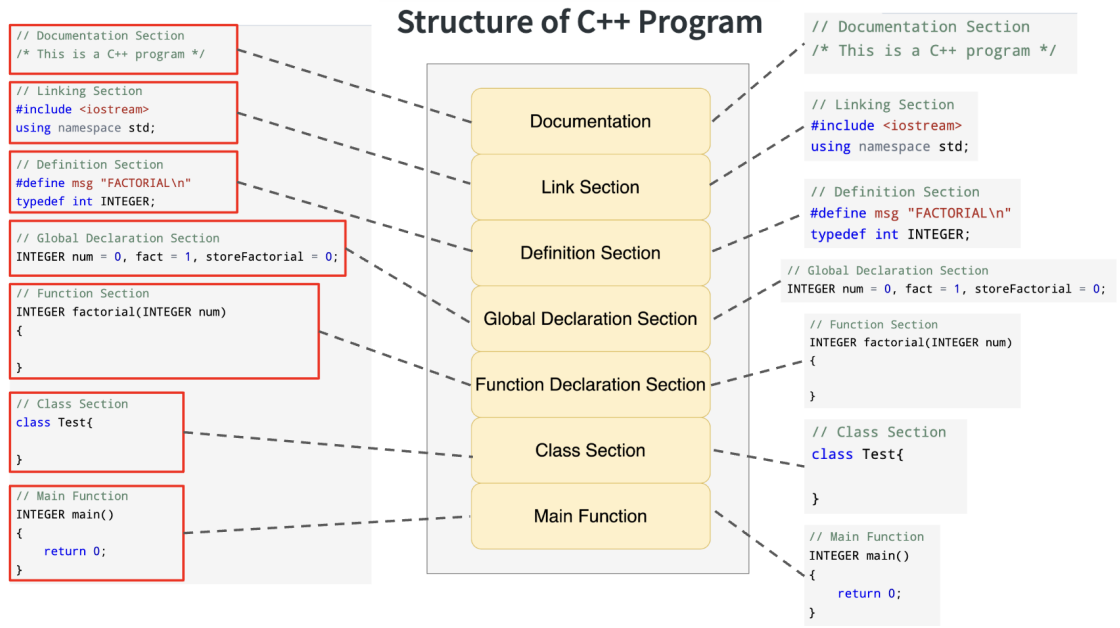


Figure 3.9: Code Body Splitting

[Figure 3.9] - Code Body Splitting 經由以上的步驟，我們就可以依照定義的主體來做切分了，左邊是一個程式範例，我們將所有的主體都先假設出來，中間則爲我們所定義的主體架構包括了 Link 、 Definition 、 Global 、 Function 、

doi:10.6342/NTU202303414

Class ，以這些定義出來的架構切割完後，就會是像右邊分別獨立出來的主體。

3.4 Code Body Database

主體資料表的建立，將會依照分類出來的主體類別新增，欄位資料也包含了 id 、 body 等重要資訊。

doi:10.6342/NTU202303414

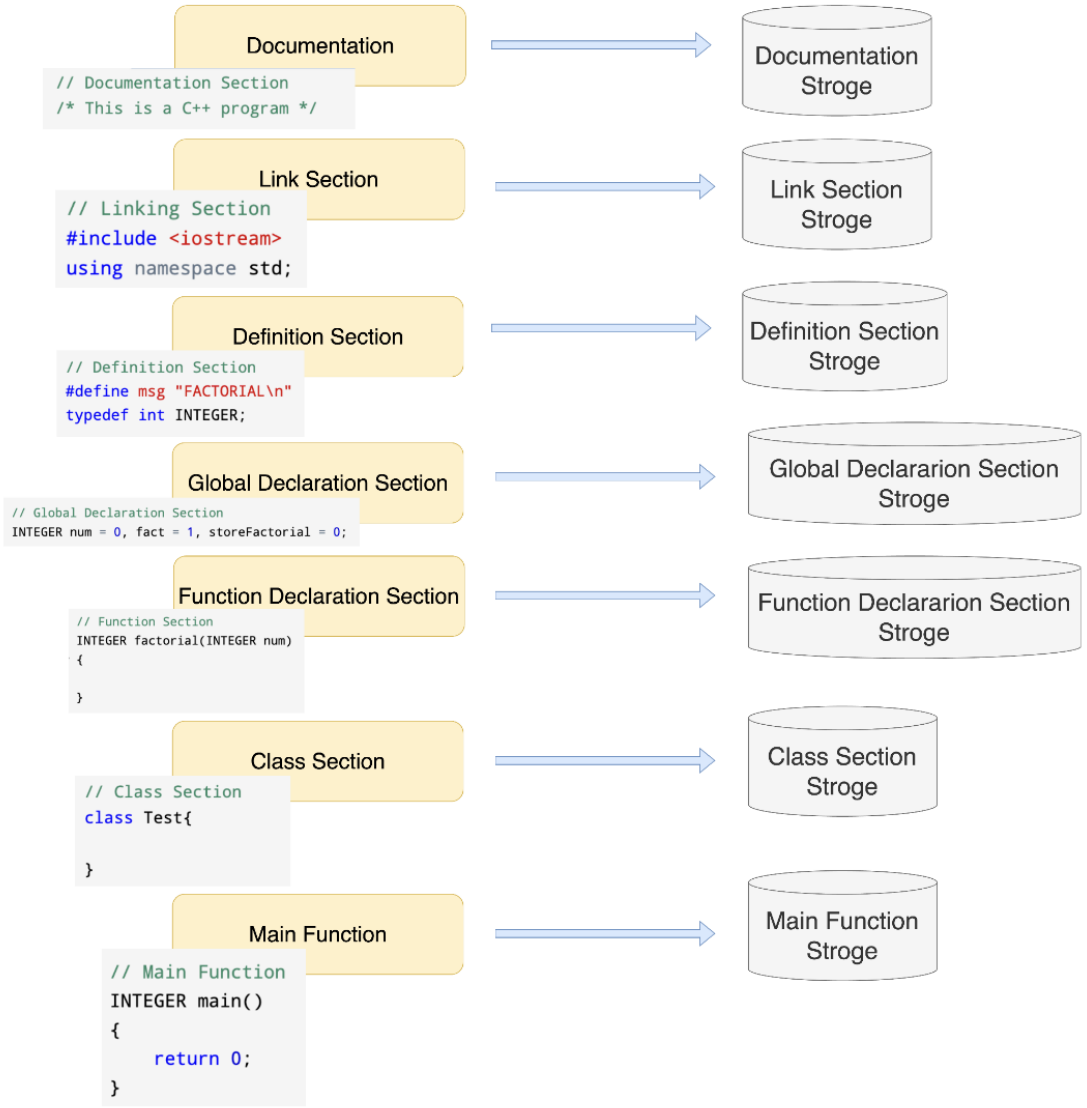


Figure 3.10: Code Body Into DB

[Figure 3.10] - Code Body Into DB 在 [Figure 3.9] 中我們已經把這些主體都切了出來，在這邊就會依序的將主體儲存到他們所對應到的主體資料庫中儲存。以下爲資料庫主體。

doi:10.6342/NTU202303414

1. Documentation Stroge
2. Link Section Stroge
3. Definition Section Stroge
4. Gobal Declararion Section Stroge
5. Function Declararion Section Stroge
6. Class Section Stroge
7. Main Function Stroge

doi:10.6342/NTU202303414

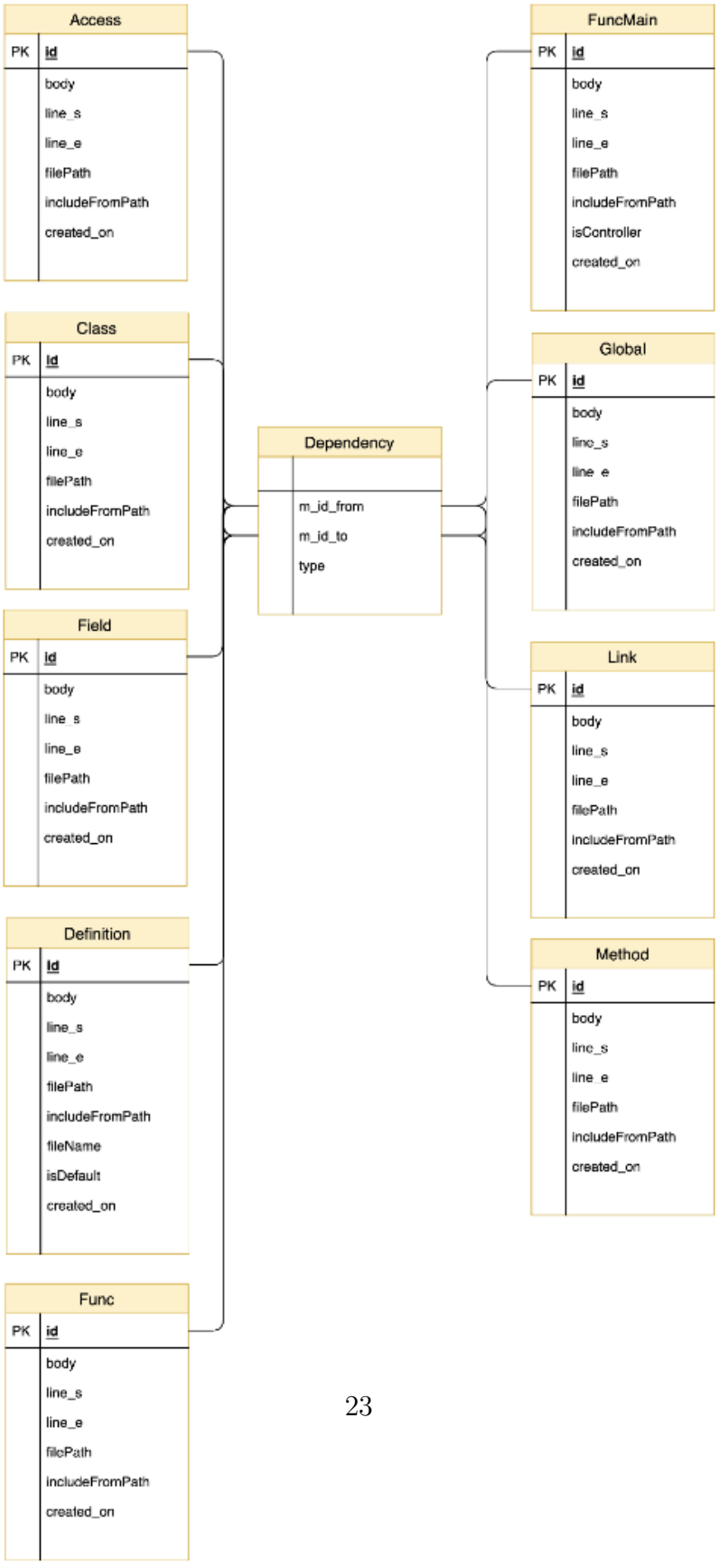


Figure 3.11: Code Body Schema

doi:10.6342/NTU202303414

[Figure 3.10] - Code Body Schema 此爲資料表的細部內容，中間爲相依，其他個別爲9個主體資料表

1. id - AST 的id
2. Body - 主體程式碼
3. line-s - 主體起始行數
4. line-e - 主體結束行數
5. FilePath - 主體的檔案位置
6. includeFromPath - 被引入的檔案位置
7. isController - 是否爲 Controller

doi:10.6342/NTU202303414

Chapter 4

Repack Service Components

CRepack 主要的目的是我們將從 Web fromework 的 Entry-Point 也就是 Controller 爲起點，以 Controller 呼叫者(Invoker)其他方法(Method) 的所有集合找出來，再重新組裝再一起，也就是將 Web fromework 以每個 Contrller 爲獨立單位一個個切割出來，每個 Controller 代表一個單獨可觸發的服務單元。

本章節我們將會介紹 CRepack 所使用到的演算法及資料結構，及 CRepack 的處理過程做一個詳細完整的介紹。

4.1 Web Framework

在這邊我們使用了 Crow 爲我們這次實驗的 Web Framework，Crow 提供了開發者 Controller 的 Function ，我們將切分出專案裡 Crow Controller 作爲我們的起始點。

doi:10.6342/NTU202303414

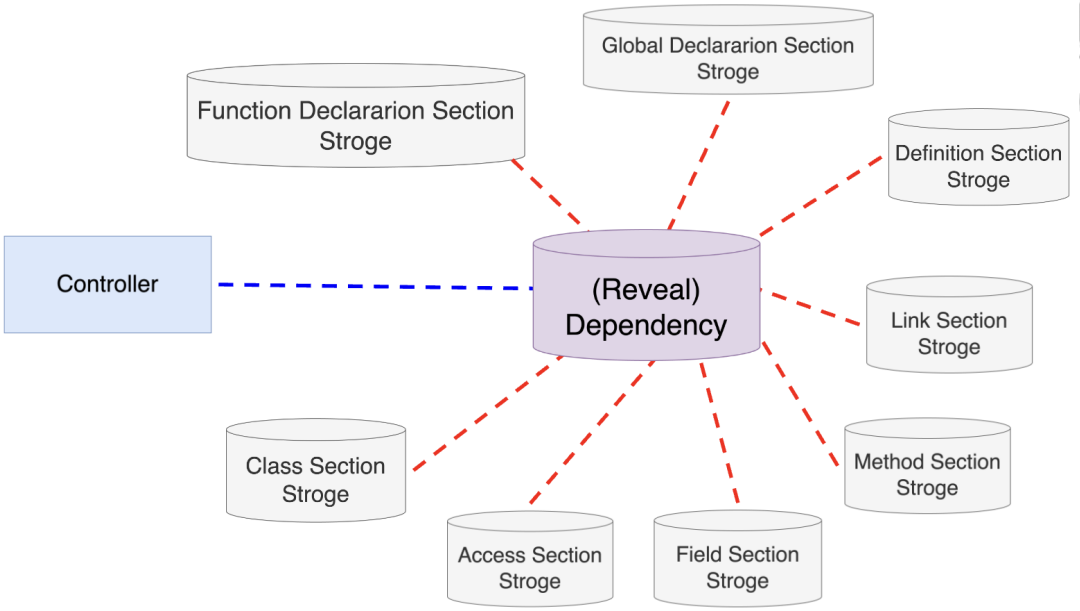


Figure 4.1: Repack Controller Dependency

[Figure 4.1] - Repack Controller Dependency 在 Repack 我們將會從每一個 Controller 挑出來，依序對這個 Controller Invoker 到的每個一個相依(Dependency)都對應起來，所以此圖表示了，我們就把資料庫的所有主體依照每個相依連起來，也就是這個 Controller 所可以單獨執行的服務集合。

doi:10.6342/NTU202303414

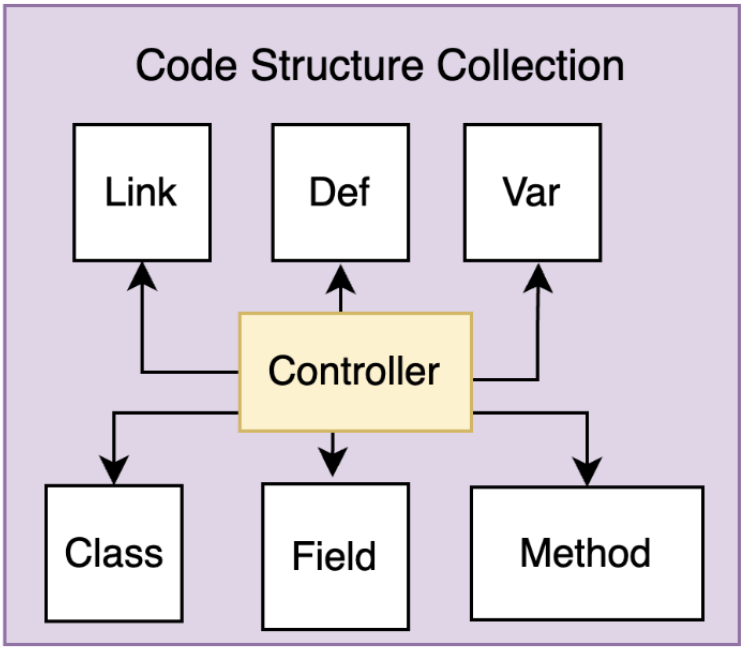


Figure 4.2: Repack Controller

[Figure 4.2] - Repack Controller 針對一個 Controller 所代表的服務集合，可以看出他必須把相關的 Link 、 Def 、 Var 、 Class 、 Field 、 Method 通通納入 Collection 中，這也才表示這個 Controller 可以被獨立出來的可執行集合。如果缺少任何一個相關主體，有可能造成編譯失敗的狀況。這也是我們前面所提出的 Dependency Table 的重要性，代表著我們必須完整的找出所有專案理所會用到的相依。

4.2 Invoker Tracing

當我們已知道 Controller 最爲我們該出發的起點，我們會先建立相鄰串

doi:10.6342/NTU202303414

列(Adjacency List) 在透過深度優先搜尋(Depth-First Search,DFS)的方法，從Controller 依序開始針對它所有的 Invoker 路徑開始搜尋，當獲得到被Invoker的方法(Method)後，我們在將這發法(Method)都存入 HashMap 中。

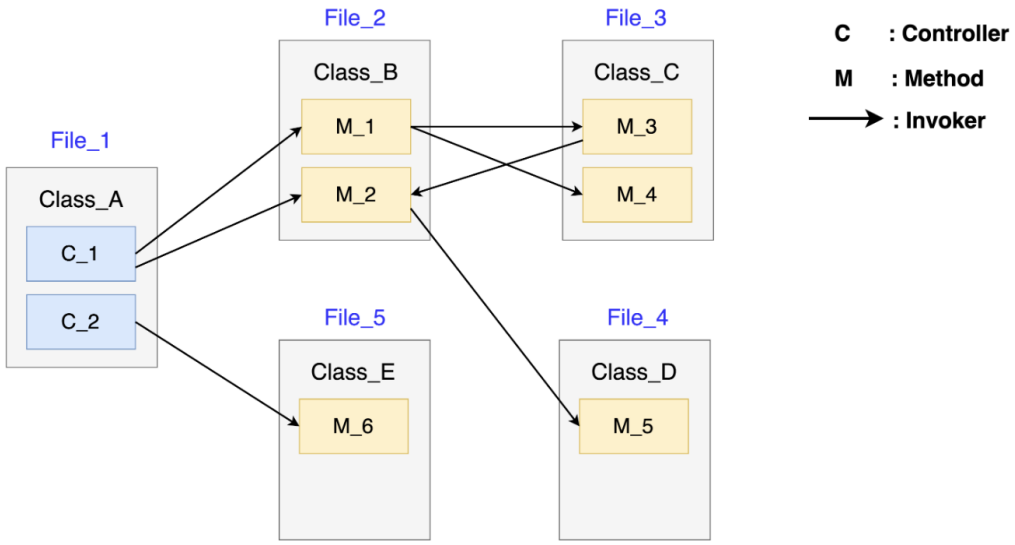


Figure 4.3: Repack Invoker Example

[Figure 4.3] - Repack Invoker Example 這邊用一個例子講解程式是之間的 Invoker 相依，我們從 File1 裡作爲起點因爲他是整個專案的entry-point 裡面有2個Controller，而C1 的 Invoker 依序爲 M1 M3 M2 M5 M4，C2代表的只有M6，從這邊來說，我們的目的就是想切分2個 Controller，分爲2個個別獨立的可執行集合。

doi:10.6342/NTU202303414

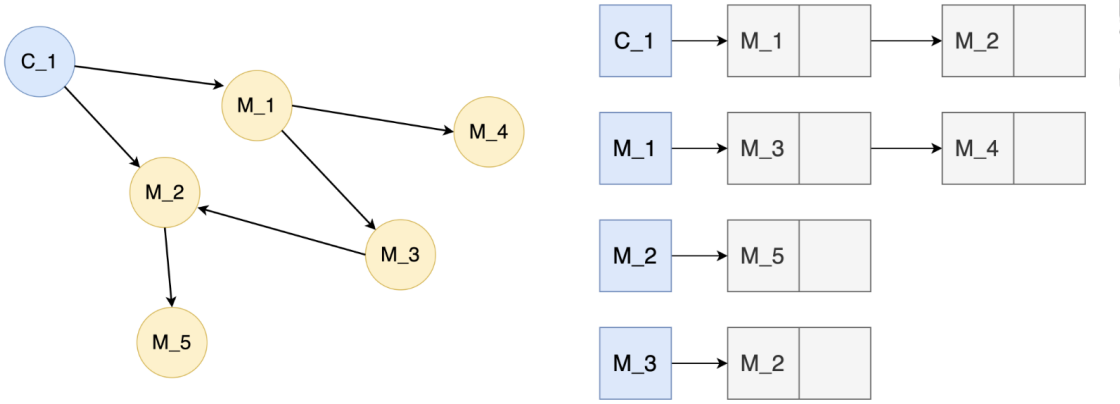


Figure 4.4: Repack Adjacency List

[Figure 4.4] - Repack Adjacncy List 在這邊我們將會使用 Adjacncy List 來表達 Method 的 Invoker 關係，所以針對剛剛的Controller1 分別 Invoker 到4個 Method M1 M2 M3 M4 M5，將會是以上這張圖所表達的資料結構。在之後我們才可使用 DFS 來走訪整個 Adjacncy List。

4.3 Method Dependency Collection

在經過方法(Method)之間的 Invoker，我們會針對被呼叫的方法(Method)透過相依圖(Dependency Graph)來找出這個方法(Method)所對應出來的所有相依(Dependency)，例如 : Class-in 、 Field-in 等，依序將這些資料存入 HashMap，其中 HashMap 的 key 值爲檔案的名稱，List 則會是針對這個檔案所有的相依(Dependency)關係(Figure 4.5)。

doi:10.6342/NTU202303414

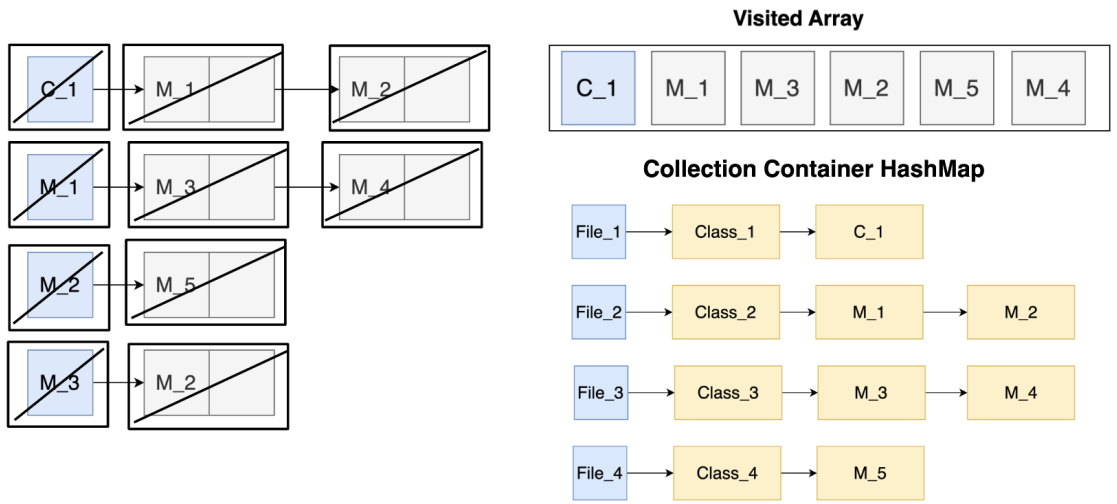


Figure 4.5: Repack DFS

[Figure 4.5] - Repack Adjacncy List 經過了 DFS 走訪，我們會將走訪的的 Method 的相關 Dependecny 通通納入 List 進來，從圖中可以看出 C1 會將 Class1 納入進來，M1 M2 會將 Class2 納入進來，這邊提到雖然 Class2 被納入進來，不過我們的資料型態是 Set 所有只要重複的直不會出現第2次，M3 M4會將 Class3 納入進來，最後 M5 會將 Class4 納入進來，最終完成了整個走訪。

4.4 Code Body Combination

當我們用 DFS 找出每個方法(Method)之間的相依(Dependency)，我們會從 Code splitting 資料庫裡，依序找出我們收集到的相依關係的程式主體(Code body)，也就是說，檔案裡會是從 Controller 到每個 Invoker 方法(Method)之間的程式主體(Code Body)，我們會將這些主體(Body)收集起來，在針對當初的行數做排序，結果就會是當初這些主體(Body)跟主體(Body)之間的相對位置關係，保

doi:10.6342/NTU202303414

證了程式當初的原始可執行架構，以下爲一個表示範例。

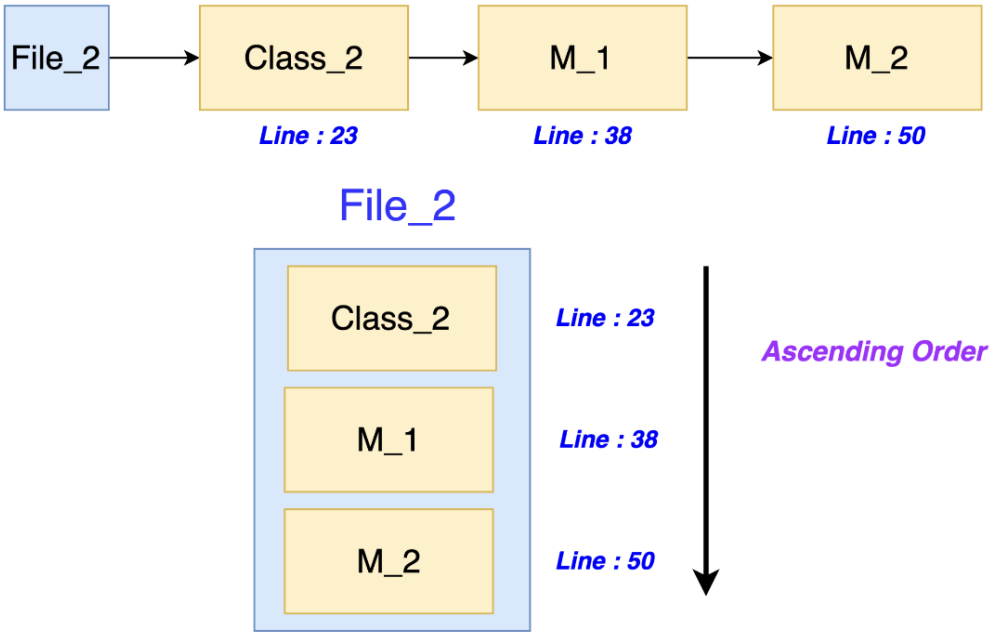


Figure 4.6: Repack Ascending

[Figure 4.6] - Repack Ascending 從上一步，雖然我們已經我完成了 Dependency 的收集，但會發生一個問題是，從新組裝的主體可能會發生錯誤，因爲我們無法可以確定每個主體的相對位置或絕對位置在哪裡，而我們在這邊解決此問題的方法是，利用 AST 告訴我們的主體行數資訊，在初期我們切割了主體資料，同時也會將行數存起來，這目的也是爲了最後重新組裝我們可以利用行數給的資訊，做相對位置的行數排序，這也確保當時程式在最初期可執行的狀況。而後只要再重新排列，也不會造成編譯錯誤。

doi:10.6342/NTU202303414

4.5 Output

在最後，所有的檔案都已經收集起來，我們會在針對這些當初在原本的檔案位置程式碼，重新產出，這其中也包括 Cmake [2]要用到的 CMakeList 檔及 Crow Web Framework 的專案套件，首先先將每個檔案的上層目錄建立出來，再將檔案位置寫入進去。

Figure 4.7: Repack Output

[Figure 4.7] - Repack Output 最終產出的新專案，分別會以原方案的 Controller 爲單位依序重新打包。以上的 crow test 是範例原方案，crow test1 跟 crow test2 是被Repack出來的新專案。

doi:10.6342/NTU202303414

Chapter 5

Demonstration

在此章節我們將展示最後的結果，這邊我們將使用實驗室所制定的 Bidding System 作爲我們簡單的範例，我們將會有 Controller 跟2個 service，第一個 service 爲 userService，是針對會員新增或修改個模組，第二個 service 是 itemService 是針對商品項目做新增修改的模組。

doi:10.6342/NTU202303414

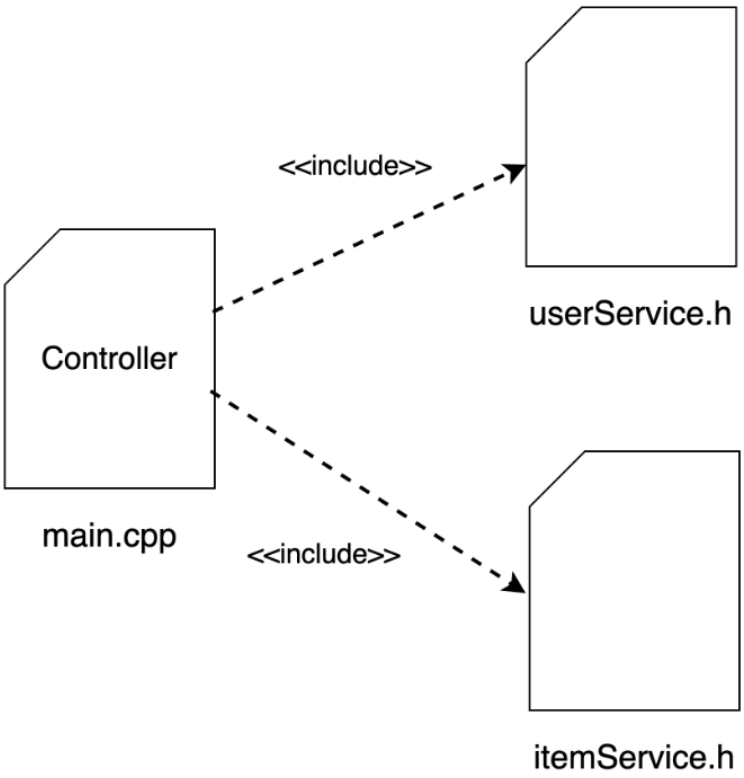


Figure 5.1: Demo Example Code Structure

[Figure 5.1] - Demo Example Code Structure 在此範例中 main.cpp 將 include 兩個 service，分別爲 userService 跟 itemServce，我們將會對整個單體式架構的系統，依照 Controller 爲單位的分解出來。

doi:10.6342/NTU202303414

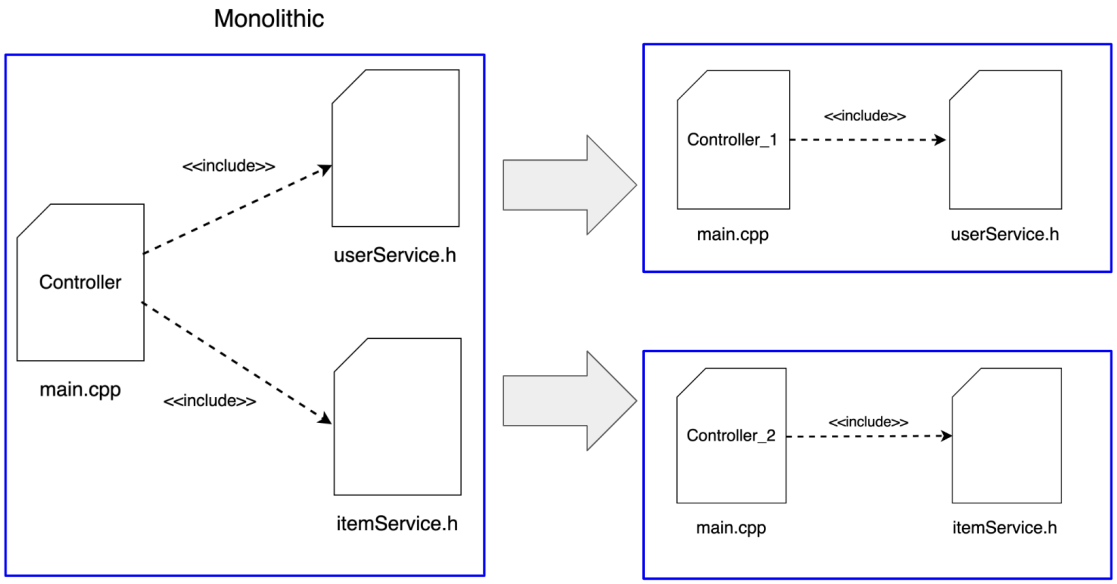


Figure 5.2: Repack Demo Output

[Figure 5.2] - Repack Demo Output 由[Figure 5.1]的範例，我們將最終 repack 成2個專案，因爲範例裡只會有2個 Controller，第一個 Controller1 會對 userService 做 Invoker，Controller2 會對 itemService 做Invocer，範例結果也就是圖中所產生的2個專案。

5.1 Controller Repack Demonstration

在最後產出的結果可以看出，根據前面提到的 Controller 將會獨立被切分出來，只有被 Invoker 到的方法(method)才會被包進來，包括這方法(Method)所對應到的相依(Dependency)。

doi:10.6342/NTU202303414

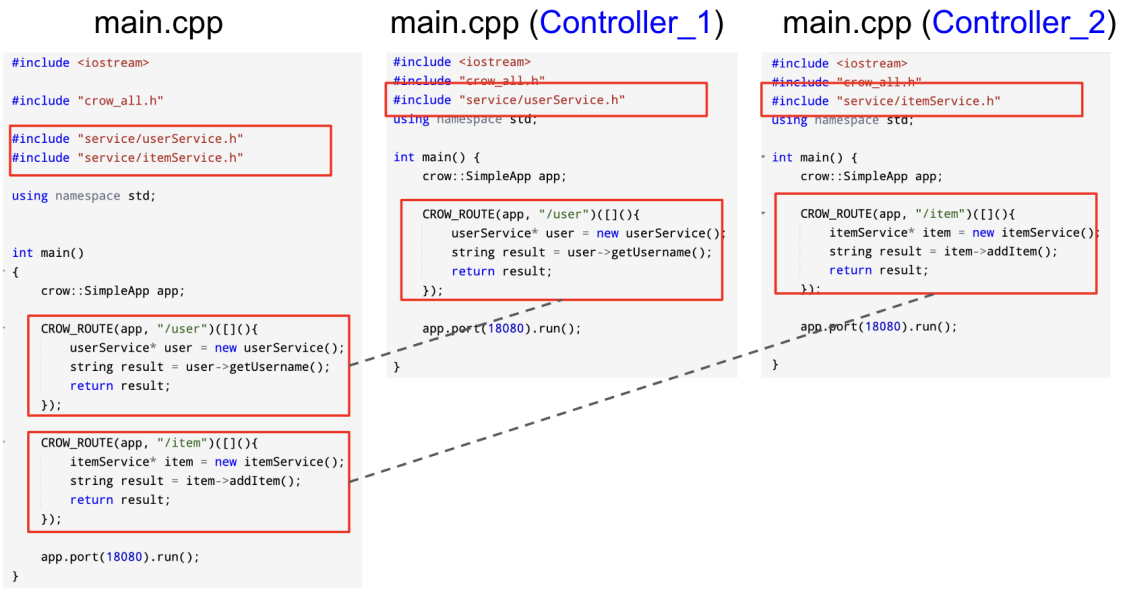


Figure 5.3: Repack Controller

[Figure 5.3] - Repack Controller 從 Demo 結果可以看出，最左邊的爲原始範例的 Controller，經過我們從新的組裝後，右邊2張圖分別代表了，原始範例被切割及重新組裝的 Controller1 跟 Controller2，而且還可以發現表頭的include 也只會將各個所被 Invoker 到的 Method 相關的 .h 檔納入進來，如果沒被 Invoker 到也不會被包進來。

doi:10.6342/NTU202303414

5.2 UserService Repack Demonstration

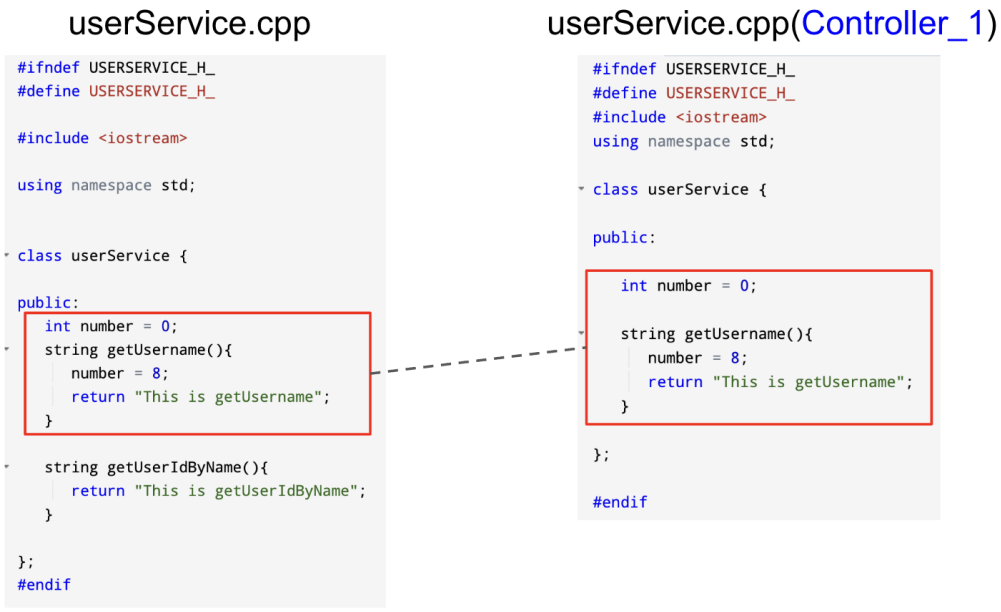


Figure 5.4: Repack UserService

[Figure 5.4] - Repack userService 從 Demo 結果可以看出，比較原本的範例程式，經由 Repack 後的 userService 只會有getUsername 這個 Method，因爲 Controller1 只 Invoker 這個 Method，所以正常情況下，我們也只會針對getUsername做重新組裝。另外也可以發現 getUsername 也對 filed 做使用的動作，這部分我們也會一併包入。

doi:10.6342/NTU202303414

5.3 ItemService Repack Demonstration

itemService.cpp

```
#ifndef ITEMSERVICE_H_
#define ITEMSERVICE_H_

#include <iostream>

using namespace std;

class itemService {

public:

    string addItem(){
        return "This is addItem";
    }

    string deleteItem(){
        return "This is deleteItem";
    }

};
#endif
```

itemService.cpp(Controller_2)

```
#ifndef ITEMSERVICE_H_
#define ITEMSERVICE_H_
#include <iostream>
using namespace std;

class itemService {

public:

    string addItem(){
        return "This is addItem";
    }

};

#endif
```

Figure 5.5: Repack ItemService

[Figure 5.5] - Repack itemService 從 Demo 結果可以看出，比較原本的範例程式，經由 Repack 後的 itemService 只會有 addItem 這個 Method，因爲 Controller2 只 Invoker 這個 Method，所以正常情況下，我們也只會針對 addItem 做重新組裝。

doi:10.6342/NTU202303414

Chapter 6

Conclusion

本篇論文提出了將複雜單體型的專案，切分成輕量的服務單元，目的是爲了減輕系統的開發複雜度及程式碼的重組及重新使用，而我們提出的方法分別爲，1. 透由AST解析程式碼，取出所有相關資訊 2. 在從中這些得到的程式碼資訊，找出所有對應的相依(Dependency)，最終產出相依圖 3. 透由AST切分程式碼主體 4. 重新包裝新的服務，以上是整篇論文所提出的產出服務單元及重新包裝的流程。

6.0.1 Benefits

1. 降低系統的複雜度.

2. 讓整體系統更好維護.

3. 程式碼可重複利用.

doi:10.6342/NTU202303414

6.0.2 Future Work

1. 加入其他種類的 CPP Web Framework.

2. 探索更多未發現相依(Dependency).

3. 嘗試解析更多 C++ 開源軟體(Open source).

doi:10.6342/NTU202303414

Bibliography

[1] Clang. https://clang.llvm.org/.

[2] Cmake. https://cmake.org/.

[3] Crow. https://crowcpp.org/master/.

[4] Llvm. https://llvm.org/.

[5] Soot. http://soot-oss.github.io/soot/.

[6] S.-W. Huang. Towards a solution to iot interoperability through reverse engineering. Master's thesis, National Taiwan University, 2017.

[7] W.-L. Shih. Construct service components from java-based open source projects. Master's thesis, National Taiwan University, 2022.

doi:10.6342/NTU202303414