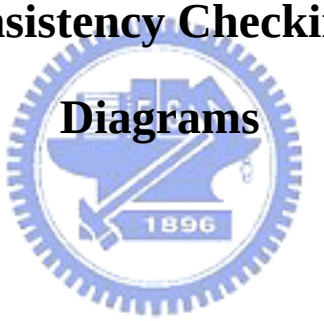
國 立 交 通 大 學

資 訊 工 程 學 系

碩 士 論 文

UML 圖形間一致性檢查之研究

The Study of Consistency Checking between UML Diagrams



研 究 生：劉昆灝

指導教授：鍾乾癸

中 華 民 國 九 十 四 年 七 月

UML 圖形間一致性檢查之研究

The Study of Consistency Checking between UML Diagrams

研 究 生：劉昆灝　　Student : Kun-Hao Liu

指導教授：鍾乾癸　　Advisor : Chyan-Goei Chung

國 立 交 通 大 學

資 訊 工 程 學 系

碩 士 論 文

A Thesis

Submitted to Department of Computer Science and Information Engineering College of Electrical Engineering and Computer Science National Chiao Tung University in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science and Information Engineering

July 2005

Hsinchu, Taiwan, Republic of China

中 華 民 國　九十四　年　七　月

UML 圖形間一致性檢查之研究

研究生：劉昆灝　指導教授：鍾乾癸
國立交通大學資訊工程研究所

摘要

物件導向分析與設計方法已被大多數軟體開發人員及組織所採用，UML 語言也已成為今日軟體設計文件的主要塑模工具。在軟體複雜度日益提升之今日，軟體文件之不一致常導致軟體開發成本之增加，因此軟體設計文件的追溯性日益重要；然而今日之物件導向軟體開發環境對於跨階段 UML 圖形之一致性檢查功能卻嚴重不足，導致軟體開發團隊常需花費甚多人力來做不同階段軟體一致性的檢查，本研究旨在探討跨階段(需求與分析階段間及分析與設計階段間)的 UML 圖形之一致性檢查問題，並提出有效機制以提升軟體文件之可追溯性。

本研究首先分析跨階段 UML 圖形元素間的關係，進而提出下列的輔助機制：

(1) 將 use case 的 flow of event 以表格式描述，方便分析階段 collaboration diagram 之 class 名稱及 responsibility 之制定。

(2) 由 analysis classes、middleware 及 system software 給予使用者建立關聯 design classes 之導引，以方便設計階段 design classes 及 sequence diagram 之設計。

透過上述輔助機制，可更方便進行(1) use case diagram 與 collaboration diagram，(2) collaboration diagram 與 sequence diagram，及(3) analysis class diagram 與 design class diagram 之一致性檢查，並在 ArgoUML 開發工具加入上述輔助及檢查機制以驗證其實用性。

藉由本研究所提出之輔助機制及檢查方法確有助於在開發前期提早發現軟體產物間不一致的問題，減少設計的錯誤，因而降低日後偵錯與修改的成本，提升軟體開發之生產力。

The Study of Consistency Checking between UML Diagrams

Student: Kun-Hao Liu Advisor: Chyan-Goei Chung
Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Chiao-Tung University

Abstract

Object-Oriented Analysis and Design have been widely used by most software developers and organizations. Unified Modeling Language (UML) becomes a major modeling tool used to describe software documents. By the improving of the complexity of information related product, inconsistencies between software documents would result in the increasing of development costs. Hence, the traceability between software documents becomes more and more important. However, the function of consistency checking between cross-phase UML diagrams is not well developed in most development environments at the present. Software development teams have to spend extra human resources in order to check consistencies between software artifacts produced in different development phases. In this research, we focus on the problem of consistency checking between cross-phase UML diagrams including those artifacts produced in requirement capturing phase to analysis phase and analysis phase to design phase. We also propose an efficient mechanism to improve the traceability between software documents.

After analyzing the relationships between elements appeared in UML diagrams, we proposed the mechanism to assist developers as follows:

1. We use a table to describe the flow of event of each use case scenario, this helps the developers identify class names and responsibilities in collaboration diagrams more easily during the analysis phase.
2. Associate the design classes with analysis classes, middlewares, and system softwares, it helps the developers identify design classes and sequence diagrams during the design phase.

It would be more convenient to check the consistency between (1) use case scenario and collaboration diagram, (2) collaboration diagram and sequence diagram, and (3) analysis class diagram and design class diagram with the proposed mechanism described above. We also implement this mechanism on the development tool called ArgoUML to verify the practicality. By means of this proposed mechanism, developers can find out the inconsistencies between software artifacts during the earlier stage of development cycles, and reduce mistakes and costs in design workflow to improve the productivity of software development.

誌謝

這本論文的完成首先要感謝指導老師鍾乾癸教授的教導，老師博學的知識以及對作學問的嚴謹態度使我受益良多，老師總是不厭其煩的指導我在研究上的缺失以及適時的提醒我做人處世的道理，讓我在碩士班的兩年生活中不僅是學習到研究上的相關知識，最重要的還有實事求是的精神。

感謝靜紋學姊一直扮演著好老師的角色，每每對於我提出的問題都耐心的講解，並且提出建議改善研究過程中遭遇到的問題。謝謝 610 實驗室所有一起為論文努力的志宇、嘉鑫、維孝、志忠，總是不斷的為彼此加油打氣，還有同樣在新竹唸書的基辰、彥錚、智瑋，陪我一路走來的戴昀、郁芳、宇穎、永昇、志銘、澤凱以及許許多多的好朋友們，謝謝你們用歡笑紓解我做研究煩悶的心情，使得我能堅持下去。

最後，感謝我的家人們一直以來對我的包容與關心，僅以此成果與所有關心我的家人及朋友們分享，謝謝你們。

目錄

圖目錄

第1章、緒論

1.1 研究背景與動機

1990 年代初期，物件導向技術逐漸成熟，物件導向軟體軟體設計方法與傳統結構化軟體設計方法相較，其最大優點是將封裝與資料隱藏的概念導入開發系統中，使所設計的軟體更具彈性、可移植性與再利用性，OO based 的軟體開發方法也提供了軟體架構建置的基礎以及利用各種元件(component)來組合成所需的系統。

軟體開發團隊使用建立模型的方法來描述欲解決的問題或是系統，這個過程稱為塑模(modeling)，在塑模的過程中開發人員將系統用文字、符號、圖形或圖表等元素組合成用來描述系統面相(view)的文件，藉由將抽象的系統以模型的方式來分析並表現問題。各階段的流程分別產生描述系統面相的模型，並透過各種模型來表示系統，塑模的過程在現今的軟體開發流程中已被認為是對系統開發工作有重要助益的技術之一，它讓軟體開發人員對所開發的系統有完整的了解，並可幫助軟體開發達成以下四個目標[1]：

1. 幫助開發人員將系統的架構視覺化(visualize)，有助於了解系統的發展概況。
2. 各個模型互相配合可以完整詳述系統的架構及運作方式(behavior)。
3. 模型提供一個開發的樣板(template)，開發人員可以藉由模型的引導來完成系統的建置。
4. 模型可以完整呈現開發過程中開發人員所做過的決定，紀錄系統開發的過程。

1989 年開始，各種不同機構或學者所提出的物件導向軟體開發方法陸續出現，如 Booch 提出的 Booch 1993[2], Jacobson 的 OOSE(Object-Oriented Software

Engineering)[3]、Rumbaugh 的 OMT(Object Modeling Technique)[4][5]等，每個提出的方法都是一套完整的架構，並使用其各自的塑模方法，也各有其優點與缺點。然而這些多樣的塑模方法造成使用上的雜亂與不方便，於是 Grady Booch, Ivar Jacobson, 與 James Rumbaugh 三人開始著手將三人所提出的方法擷取優點與各方法的相似之處並加以整合，進而制定了統一模型化語言(Unified Modeling Language，即 UML[1])並定期討論 UML 語言之適宜性與實用性。UML 語言的內容主要由 things 及 relations 這兩個基本元素所組成，並訂定有九種 diagrams，這些不同類型的 diagrams 可以用來描述軟體發展各階段的模型，而由於 UML 的出現使得各種新技術的研究成果得以整合及累積，也因為 UML 被有計畫性的推廣及許多軟體開發單位的採用使得物件導向技術更加成熟與廣泛應用。

在 UML 語言逐漸成熟之後，Grady Booch 等人進而提出一套與 UML 互相搭配的軟體開發流程，即統一軟體發展流程(Unified Software Development Process，或稱 Rational Unified Process，簡稱 USDP[6])。USDP 具有 use-case driven、architecture-centric、與 iterative and incremental 等三個主要的特色，並依序規劃了 requirement capturing、analysis、design、implementation、與 test 五個階段，在此五個階段的執行過程中均會產生不同的產物，這些產物大部分是以 UML 語言所規範的各種 diagram 來表示，而軟體開發是一種滾動式的、循序累進的設計活動，所以每個階段的產物會與上階段的產物有關聯，也就是各產物間具有一致的關係。

由於軟體開發團隊對於使用 UML 與 USDP 以協助軟體開發的需求，市面上出現數種開發環境如 Rational Rose、Visual Paradigm、ArgoUML 等等，這些工具有些只提供基本的 UML 製圖功能，有些則包括軟體開發流程的導引與輔助。因為這些開發環境的出現，使用者得以系統化地建立模型與文件，有助於提升軟體開發的品質與效率。然而現有的軟體開發工具多著重於模型建立的功能以及軟體開發流程的順暢，卻鮮少考慮各階段產物間一致性的問題。隨著軟體的規模越來越大，伴隨著將產生更多的產物以及團隊參與成果，反易產生不同階段產物內容

不同、不完整或多餘等現象，也就是開發流程各階段產物有不一致的情形發生。

現有軟體之產物間不一致的研究，大都屬同一個階段或是在整個開發流程之後才檢查其產物間是否有問題，對跨軟體開發階段所產生的不一致現象鮮少討論。跨階段產物間的不一致將影響整體軟體品質甚鉅，只要有一個產物沒有完整的將資訊傳送到下個階段，將使得整個流程產生連帶的錯誤或缺失，最後導致測試錯誤，需要耗費更多的人力、時間以及開發成本找出及修正錯誤，因此除了同階段產物間的一致性檢查之外，跨階段產物的檢查更是整個開發流程中所不可或缺的。若能在軟體開發流程中依跨階段產物的關聯性適度輔助軟體開發人員，並自動進行跨階段產物的一致性檢查，對軟體品質與效率之提升有重要助益。

本研究分析 USDP 相鄰階段產物間的相關性與追溯性，並利用所得到的相關性提出跨階段產物間一致性檢查的方法，並將此方法整合於 UML 製圖的開發環境中，讓使用者可以在系統上直接進行繪圖與檢查的工作。

本研究分為四個主要部分：

1. 各階段產物性質之探討：配合軟體開發流程的各個階段來敘述 UML 在各階段所產生的產物，並探討產物的性質與產物的產生目的。
2. 各階段產物間的相關性分析：分為同階段間的產物與跨階段的產物來討論，將研究產物間的一致性、完整性與追溯性(traceability)。
3. 跨階段一致性檢查之輔助：提出方法來幫助使用者藉由與系統的互動來檢查出產物間的不一致，同時規範產物的描述方式以提供額外的一致性檢查資訊。
4. 輔助系統之設計：將所提出的檢查方法實做一雛型系統。

1.2 各章節介紹

本論文章節安排如下，首先在第二章介紹軟體開發的流程以及各階段產物一致性檢查的相關研究。第三章敘述需求取得階段到軟體設計階段產物間追溯性的關係，進而提出增進各個階段產物追溯性的機制。第四章敘述所提各階段產物間

一致性檢查的方法。第五章說明依上述方法建置的開發環境之系統架構與使用介面。第六章則為本研究之結論及未來研究方向。

第2章、背景知識與相關研究

2.1 USDP (Unified Software Development Process)與 UML

利用物件導向技術(Object-Oriented Technology)來設計軟體已是軟體發展的主流趨勢，目前軟體開發常用的程式語言如 JAVA、C++、Visual Basic、C#等等均為物件導向程式語言，物件導向軟體發展方法及物件導向軟體發展環境也漸趨成熟。回顧１９９０年代初期，各類物件導向分析與設計方法陸續推出，帶動物件導向技術的風潮，但各人所採用的 model 不同，孰優孰劣見仁見智，經過幾年的討論，大家共識趨於一致，由 Grady Booch、Ivar Jacobson 及 Jim Rumbauth 等物件導向技術研究先驅領導而制定 Unified Modeling Language(UML)物件導向軟體塑模語言，以作為軟體塑模文件標準，並成立 UML consortium 定期討論 UML 語言之適宜性及實用性，有了共同的軟體塑模語言，各種新技術的研究成果得以整合與累積，再經過 UML consortium 成員的仔細討論使其更為完整與周嚴，因而加速物件導向技術的成熟與廣泛應用。

Grady Booch 等人在 UML 語言漸趨完整後，進而提出一套與 UML 互相搭配的軟體開發流程，即 Unified Software Development Process (USDP)，此流程具有三個主要的特質：

- Use-case driven：以 use cases 來推導各種模式之完整建立。
- Architecture-centric：以系統架構為優先建立的開發方法。
- Iterative and incremental：以反覆漸進方式來構建完整軟體系統。

USDP 流程將軟體開發分為四個主要的階段：inception、elaboration、construction 及 transition，每個階段包含一次或多次的 workflow 循環 (iteration)，每個循環的開發工作包含一或多個下列步驟：

a. Requirement Capturing：定義系統的需求、參與系統的各個角色(actor)、

系統功能的 use cases、及系統非功能性需求。

b. Analysis：依據系統的 use cases 定義系統應有的 packages、analysis classes、完成 use case 的相關 analysis classes 的互動關係(以 collaboration diagram 表示)、及各 analysis classes 間的關係(以 class diagram 表示)。

c. Design：定義系統的硬體架構(以 deployment diagram 表示) 、應有的 subsystems 及相互關係、使用的 middleware 與系統軟體、系統應有的 design classes 以及其 attribute 與 operation、每一 use case 內各 design classes 的互動關係(以 sequence diagram 表示)、及定義各 design classes 間的關係。

d. Implement：依 Design 的結果運用程式語言實作軟體系統，系統的組成包含 component、source code、scripts、binaries 與可執行檔案等，工作重點包括 component 的配置、及 subsystem 與 class 的實作與 test。

e. Test：主要工作為設計與規劃系統的整合方式與測試流程、規劃 test case、執行軟體及系統整合測試。

執行上述步驟的過程都會有相對應的產物(artifact)產出以供品管人員驗證其正確性、完整性及一致性，這些產物大多以 UML 語言來表示撰寫此階段的模式，這些產物將在第三章詳細介紹。

UML 語言之基本元素為 things 及 relations，由 things 及 relations 可構建九種不同 diagrams 以描述軟體發展各階段的 models，利用這九種 diagrams 及相關說明文件即可構建出完整的軟體產物，這些 diagrams 甚至可透過 XML 語言表示以供軟體設計文件交換之用，更加速軟體元件及軟體設計文件之再利用，對軟體生產力與品質的提升有極大助益。UML 之九種 diagrams 可分為 structural diagram 與 behavior diagram 兩大類，此九種 diagrams 簡要說明如下：

- **Structural Diagram**

  1. Class Diagram: 由 classes、 interfaces、collaboration 及 relations 等組成，以表示 classes 之間的關係。在 USDP 過程是透過 class diagram

以表現 analysis classes(analysis 階段)或是 design classes(design 階段)之間的關係。

2. Object Diagram: 由一組 objects 與 objects 之間關係所組成，Objects 通常來自 class diagram 中 classes 的 instances，用來代表設計階段特定 objects 間之互動關係。
3. Component Diagram: component 是系統中一個密不可分的運作單元，透過 component diagram 可表示一群 components 之間的互動關係及各 component 的組成元素。
4. Deployment Diagram: 包含各類實體運算資源 nodes 的分佈與連接關係，及各 node 上置放的 components，代表系統軟硬體佈署關係，在 design 及 implement 階段，可利用 deployment diagram 來表示各 node 上的 subsystems 及/或 components。

- **Behavior Diagram**

5. Use-Case diagram: 包含 use cases、actors 及兩者間的關係，actors 代表與系統相關之周邊環境元件，use case 表示一種系統功能，actor 與 use case 的關係代表周邊環境元件允許執行的功能。Requirement capturing 階段利用 Use-case diagram 來描述系統中所有 actors 與所有 use cases 及其相互關係。
6. Sequence Diagram: 代表一組 objects 執行的順序及互動關係，可用來表示軟體系統中一種執行行為的全部或局部。依 USDP 的規範，在 design 階段採用 sequence diagrams 描述各 use case 中參與 objects 的互動關係及執行順序。
7. Collaboration Diagram: 以 classes 或 objects 為主體，描述 classes 或 objects 之間的互動關係，主要強調互動之間的結構性。依 USDP 的規範，在 analysis 階段採用 collaboration diagram 以描述 use case 中參與 analysis classes 間的互動關係。

8. Statechart diagram: 以 state machine 組成，包含 states, transitions, events, activities，以 event-driven 的方式來描述系統的狀態變化。依 USDP 規範，在 design 階段可用 statechart diagram 來描述一個複雜 design class 的狀態及其變化情形。
9. Activity diagram: 描述活動流程，描述 activities 和 activities 間執行順序關係，依 USDP 規範，其可用來描述複雜動作的執行流程。

2.2 UML 圖形間的一致性問題與相關研究

軟體發展過程中，依階段及系統複雜度可運用上述九種 diagrams 之某些 diagrams 來描述該階段的設計文件部份內容，例如需求階段可用 use case diagrams 來描述系統的環境、應有的功能、及各環境角色與 use cases 間的關係。一些複雜 use cases 可用 activity diagram 來描述系統與內部動作之執行順序，或用 statechart diagram 表示在不同 use cases 間系統狀態之轉換關係；在分析階段則可用 collaboration diagram 來表示每一 use case scenario 所用到 analysis classes 及這些 classes 間的互動結構關係，及用 class diagram 來顯示 analysis classes 間的相互關係等等。

從上一個章節我們知道在 USDP 中每個 workflow 的階段都會有不同的產物，這些產物大部分是以 UML 所規範的形式來表示(如 class diagram、collaboration diagram 等)，同一階段的 UML 圖形之元件需維持一致；軟體開發是一種循序累進式設計活動，每個階段的產物與上個階段的產物間有一定的追溯關係(traceability relation)，利用這種關聯性可檢查不同階段產物間之一致性。

目前已有許多以 UML 語言為基礎之軟體開發環境問世，使用者可很方便的繪製各種 UML 圖表以建構軟體系統，然而由於使用者經驗不足或疏忽常造成圖形的不完整或圖形間發生不一致的現象，這些不一致的問題可以歸納成以下兩種：

a. 同階段 UML 圖形間相互的不一致：指同一個階段中(如同在 design 階

段)，產生的產物間有不一致的情形存在。例如 design 階段某一 sequence diagram 中，object A 呼叫另一個 object B 所不存在的 operation，即 class diagram 中 object B 所屬的 class 並沒有定義該 operation。

b. 跨階段 UML 圖形相互間的不一致：指跨階段中(如 analysis 至 design 階段)，產物間有不一致的情形存在。例如 analysis 階段中被定義的某個 analysis package 並沒有在下個階段(design 階段)被 design subsystem 所實現，此時就代表 design 階段並沒有完整的把上個階段的結果延續下來，所以就產生了不一致。

產物的 UML diagram 若發生不一致的錯誤將導致開發者設計出錯誤的系統。目前的軟體開發環境如 Rational 公司的 Rational Rose、The Regents of the University of California 的 ArgoUML 等除可提供繪製 UML 圖形功能，也會自動檢查單一 UML 圖形中資料之完整性、一致性、及模糊性，並提出警告訊息，但其功能僅限制於同一個 diagram 形式內資料的檢查或是同一個開發階段，尚無法作到跨圖表與跨階段間資料一致性的檢查，換言之，目前的工具尚無法作到跨 UML 圖形間追溯性的自動檢查，因此在發展大型軟體的過程仍然會發生不同階段的設計文件間有 inconsistent、incomplete、及 ambiguous 的問題，仍需靠人檢查以找出，花費甚多人力與時間，若有適當方法可自動或半自動檢查圖形間不一致現象，對軟體發展的效率與品質的提升有重要助益。

有關跨 UML diagram 間的 traceability 與 consistency 檢查的研究成果雖已有人提出，但仍有甚多不足之處，如 Padmanabhan Krishnan 提出將 UML 的 diagram 轉換成一套以 state 為基礎的 PVS 表示式[7]，再藉著 PVS 檢查 state sequence 的架構來判斷這些由 diagram 轉換成的 states 是否一致。這項研究之不足之處有：

1. 需另外引用一套新的語法
2. 僅能將 sequence diagram, activity diagram, use-case diagram 中部分的 UML 轉成 PVS 語法。

3. 缺乏充足的實例證明能有效解決 traceability 與 consistency 的問題。

Oliver Laitenberger 提出一套"reading techniques"來幫助 reviewer 更有效的找出文件中不符合 consistency 之處[8]，不過僅針對需求與分析階段中部分 diagrams 提供一些 review 時該注意的 guidelines，不能作到自動或半自動檢查。Pascal Fradet 等人則提出利用圖論(graph theorem)的關係(relation) 來檢查 diagram 與 diagram 間的圖形結構有沒有相似[9]，配合正規語言將 diagram 的衍生關係展成語言的形式以作檢查，這種利用數學的方法來做驗證雖然正確性高，但是一般人不易使用。

此外還有學者自訂出一套定義圖與圖之間關係的圖形語言，Stuart Kent 等人將 diagram 間的元素利用他們所訂出的語言將圖形間對應關係以視覺化的方式表現出來[10]，然而較複雜的 UML diagram 所產生對應圖形與原本 UML 的 diagrams 糾結在一起使得可讀性大幅降低，徒增軟體設計師的困擾。另有些學者利用工具將 UML diagram 先轉成中間碼，再利用這些中間碼去檢查有沒有符合他們所訂定的一些 constraint rules，以檢查 UML diagram 間的一致性[11][12]，這種方式的前提是軟體設計師本身要懂得這些軟體與中間碼的使用方式，軟體設計師為了檢查 consistency 的問題而要另外學一套工具或語言，甚為麻煩。另外 Engels 等人則以 UML-RT 為基礎，利用 CSP 作為 Consistency Check 的工具來處理 Behavioral Model 的 Consistency Check[13]。UML-RT 是一種為了描述 Real-time 系統的複雜行為，從 UML 延伸出來的語言，CSP(Communicating Sequential Processes)提供一個數學模型來描述 Concurrency。此研究成果的主要缺點是只能作 Behavioral Model diagrams 的 consistency Check，無法適用於其他圖形。

從以上的相關研究成果可了解目前有下列不足之處：

1. 檢查的對象沒有包含所有的 diagram，將導致資訊上的遺漏。
2. 使用太過複雜的數學語言，一般開發人員學習不易。
3. 沒有配合軟體開發的流程來逐步檢查，僅能片面式的將某一個時間點所繪製出的 diagram 做檢查，無法避免在軟體開發中錯誤的發生。
4. 沒有合理的解釋圖形間的 traceability 與相關性，僅著重在某個元素是否

出現在正確的位置來研究，並沒有完全表現每個 diagram 間的演進關係。

5. 缺乏完整的協助開發工具。

利用物件導向軟體發展方法 USDP 最大的優點是 seamless，前階段所建立的資料在下一階段仍可引用，可引用此性質來進行跨 UML diagram 間的 traceability 與 consistency 檢查，若能提供一套工具來幫助使用者在軟體開發的過程中持續檢查跨階段間產物的一致性對軟體系統的正確性將有重要助益，可提升軟體開發的生產力；因此，本主題之研究目的是探討跨 UML diagram 間的 traceability 與 consistency 的性質，進而設計跨 UML diagram 間的 traceability 與 consistency 自動或半自動機制，並在 ArgoUML 工具上實作以驗証其可行性及實用性，本系統將對以 UML 語言作為物件導向軟體發展文件標準的軟體設計文件之檢查有重大助益，進而提升軟體發展之生產力與品質。

第3章、UML 圖形間相關性之分析

本章主要探討 USDP 工作流程中自 requirement capturing 階段到 design 階段間各階段產物的關係，包括同階段間產物的一致性與跨階段間產物的一致性，並藉由這些相關性進一步提出如何透過適當增加產物的相關資訊來協助進行跨圖形間的一致性檢查之設計。

本章討論 USDP 中的前三個流程，包含了該階段流程的描述以及其所產生的產物，另外也依據該階段產物間的關係提出本系統所能提供使用者之一致性檢查功能。以下共分為三個小節，3.1 節討論 USDP 流程中各階段的產物以及在同一個階段間產物的相關性，3.2 節討論跨階段產物的相關性。

3.1 USDP 各階段之產物

UML 語言所規範的九種 diagrams 用於描述 USDP 的各式文件，requirement capturing 階段使用 use case diagram 描述系統中 actors 與 use cases 的關係、activity diagram 描述一個 scenario 的活動流程；在 analysis 階段使用 class diagram 來說明系統內 analysis class 之間的關係、使用 collaboration diagram 來說明特定之 analysis class 間如何互動以實現 use case 之某一 scenario；在 design 階段使用 class diagram 來說明系統內 design class 間的關係、使用 sequence diagram 說明特定 design objects 間之互動流程以實現一 use case 之特定 scenario、使用 deployment diagram 表示系統各 node 之關係及各 node 上佈署的 subsystem、使用 statechart diagram 描述一個複雜 design class 的狀態變化情形；在 Implementation 階段使用 component diagram 來描述系統 component 內之 classes 及 component 間的關係、使用 deployment diagram 來表示各 node 上 component 的佈署。

為發掘 USDP 各階段之 UML 圖形之關係，以下先描述 USDP 流程中的前三個階段的工作流程、產物及使用之 UML 圖形。

3.1.1 Requirement Capturing 階段

Requirement Capturing 階段是開發流程中的第一個階段，根據 USDP 的定義本階段的主要工作包含下列五個：

1. *Find actors and use cases*：

找出系統中的 actors 與系統所提供的 use case，並簡單描述每個 use case 的作用。

2. *Prioritize use cases*：

訂出 use case 開發的優先順序，較為重要的 use case 必須要在系統開發初期先行開發。

3. *Detail a use case*：

詳細描述 use case 裡面的 flow of event，包括 actor 如何與系統互動、如何起始該 use case、以及 use case 裡的 basic scenario 與 alternative scenario。

4. *Prototype User Interface*：

描繪使用者介面的雛型。

5. *Structure the use-case model*：

建立 use case 間的各種關係，如 include、extend、generalization 等關係，並建立 use case diagram。

此階段之產物包括：

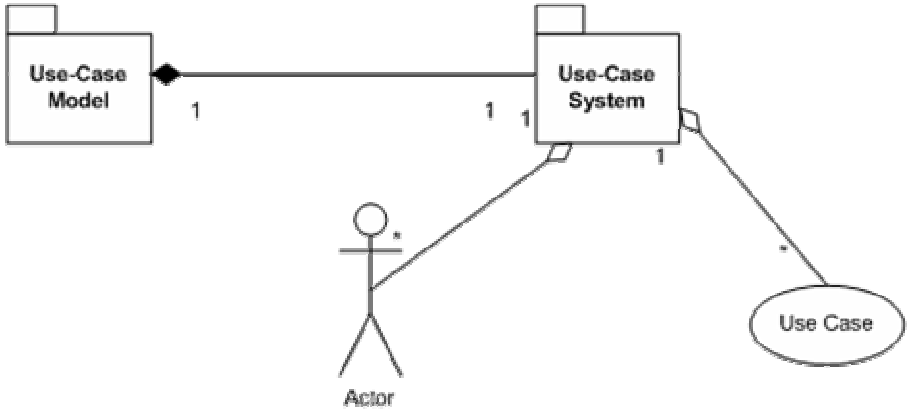


圖 3-1 Requirement Capturing 階段之產物

- *Use-Case Model*：由 actor、use case、use-case system 所組成。
- *Actor*：use-case model 描述系統要對何種類型的 user 提供何種功能，每一個不同類型的 user 稱為一個 actor (human actor)，而與系統溝通的 External system 也是一個 actor (system actor)。Actor 是表示一個在系統外的角色如何與系統做互動。
- *Use Case*：每一種能讓 actor 與系統做互動的方法稱作 use case，一個 use case 含一連串的系統與 actor 間之相互動作流程，每一種可能動作流程稱為 scenario，可分為 basic path 與 alternate paths。
- *Architectural Description*：architectural description 包含 use-case model 的架構面(architectural view)描述，並說明在架構上有重要影響的 use cases。系統的架構面必須同時將處理較重要或是較關鍵功能的 use cases 加以描述，這些 use cases 將在系統的開發流程前期優先開發。
- *Glossary*：glossary 被用來註明一些重要的或是常用的語詞(term)，這些語詞將被系統開發人員或是系統分析師在描述系統時使用。它在整個開發文件上扮演重要的角色，可以避免造成文字使用上的誤會以及定義不明的情況。
- *User-Interface Prototype*：在 requirement capturing 階段可以先建立使用者介面的雛型，這將對了解 human actor 與系統間之互動關係有助益，這不只能幫助開發人員設計更好的使用者介面，也能讓開發人員更了解 use case。

use-case model 是以 use-case diagram 來表示各個 use case 與各個 actor 間的關係，use-case diagram 是 UML 標準所規定的圖形之一，用來描述系統的 static use case view，它提供了以下的資訊：

1. *Use cases*：系統所定義且必須完成的所有 use case。
2. *Actors*：用來起始 use case 的 actor，或是參與 use case 的其他 actor。
3. *Relationships*：包含了 dependency、generalization 以及 association，用來

表現 use case 間的關係。

一 use case 包含一個以上的 scenario，當系統複雜時，可透過 package 來組合以展成階層式關係，方便使用者閱讀。以 Pay Invoice[6]這個 use case 為範例，其詳細互動流程可描述如下：

----------Start of Pay Invoice Use Case----------

Use Case: PAY INVOICE

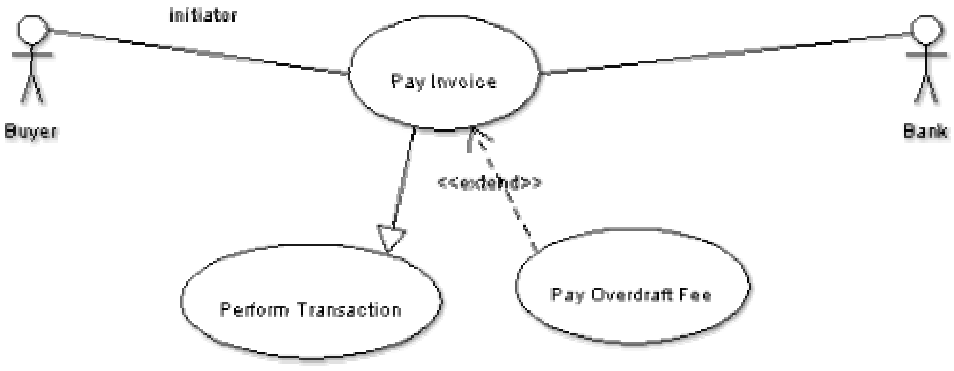
Initiator: Buyer

Actor: Buyer and Bank

Brief Description

The use case Pay Invoice is used by a Buyer to schedule invoice payments. The Pay Invoice use cases then effects the payment on the due date.

Context Diagram



Preconditions

The buyer has received the goods or services ordered and at least one invoice from the system. The buyer now plans to schedule the invoice for payment.

Flow of Events

Basic Path

1. The buyer invokes the use case by beginning to browse the invoices received by the system. The system checks that the content of each invoice is consistent with the order confirmations received earlier (as part of the Confirm Order use case) and somehow indicates this to the buyer. The order confirmation describes which items will be delivered, when, where, and at what price.
2. The buyer decides to schedule an invoice for payment by the bank, and the system generates a payment request to transfer money to the seller's account. Note that a buyer may not schedule the same invoice for payment twice.
3. Later, if there is enough money in the buyer's account, a payment transaction is made on the scheduled date. During the transaction, money is transferred from the buyer's account to the seller's account.
4. The system set the status of this invoice to "paid".
5. The use-case instance terminates.

Alternative Paths

1. In Step 2, the customer may instead ask the system to send an invoice rejection back to the salesperson.
2. In Step 3, if there is not enough money in the account, the use case will cancel the payment and notify the customer.

Postconditions

The use-case ends when the invoice has been paid or the invoice payment was canceled and no money was transferred.

----------End of Pay Invoice----------

以上的 use-case model、use case、actor 及 use-case scenario 是以階層式的方式來呈現的，use-case model 由 use case diagram 來表示，包含系統所有定義的 use

case 與 actor，而每個 use case 都必須有一個 use case specification，其中包含了一個以上的 scenario，每一個 use case specification 裡的 scenario 將會在 analysis 階段各自發展出一個 collaboration diagram。

3.1.2 Analysis 階段

根據 USDP 的定義 Analysis 階段包含有以下主要四個工作項目：

1. *Architectural Analysis*

1.1 定義系統中的 analysis package, service package, 與建立 analysis package 間的依存關係。

1.2 定義 obvious entity class。

1.3 定義 common special requirement。

2. *Analyze a Use Case*

2.1 定義 analysis classes。

2.2 描述 analysis object 間的 interaction。

2.3 Capturing special requirement

3. *Analyze a Class*

3.1 定義 analysis classes 所負責的 responsibility。

3.2 定義 analysis class 的 attributes。

3.3 定義 analysis class 之間的 associations 與 aggregations

3.4 定義 analysis class 之間的 generalizations

3.5 Capturing special requirement

4. *Analyze a Package*

Analysis 階段的主要產物如下：

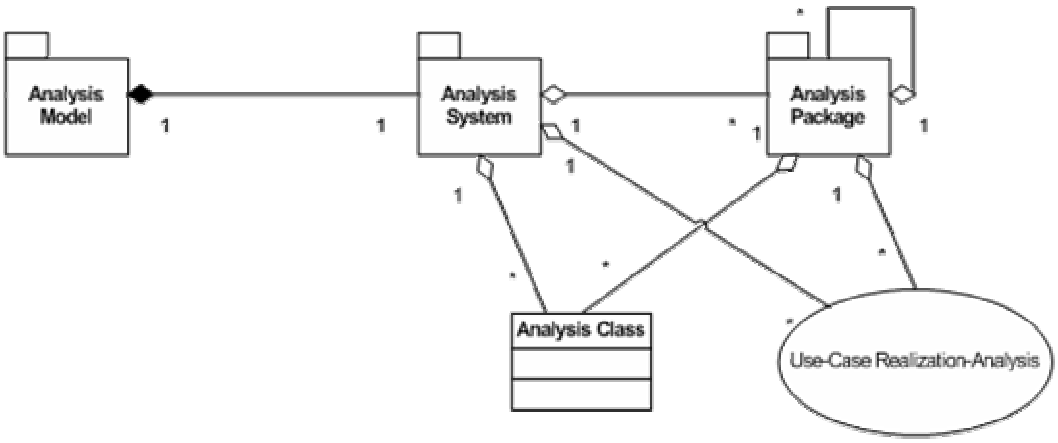


圖 3-2 Analysis 階段之產物

- *Analysis Model*：用來描述系統最 top-level 的模型，由 analysis class、analysis package、use-case realization-analysis 所構成。
- *Analysis Class*：依據 use case 而尋出之系統功能基本組件，其可負責處理系統中的部分 functional requirement，analysis class 上定義了 attribute 與 responsibility。Analysis class 依照使用特性的不同分為下列三種類型：
  I. Boundary Class：boundary class 的工作是扮演 actor 與 system 間的溝通橋樑，每個 actor 都必須要透過至少一個 boundary class 才能與系統溝通，如 ATM 上的人機介面。
  II. Entity Class：entity class 通常是代表一個會長時間存在且內容不常更動的 object，如資料庫中的訂單紀錄。
  III. Control Class：control class 負責複雜的運算邏輯、協調、排程、與控制系統內其他 object 的活動，如銀行信用卡交易系統中的個人安全碼計算單元。
- *Use-Case Realization-Analysis*：analysis 階段的 use-case realization 是由 analysis class 間的互動所構成，可使用 collaboration diagram 來透過那些 analysis class 之 responsibility 來完成一特定 use case 之某一 scenario 所指定的功能，一個 use-case realization 可以直接對應到一個 requirement

capturing 階段的 use case scenario。

- *Analysis Package*：analysis package 是將系統分為多個容易管理產物的集合，它可以包含 analysis classes、use-case realizations、或是其他的 analysis packages。

Analysis 階段產生的 UML 圖形有 class diagram 與 collaboration diagram 兩種，每一個 collaboration diagram 均是透過數個 analysis class 的互動來實現一個 scenario，而 class diagram 則是描述所有定義的 analysis class 的關係，由以上的特性可以歸納出此兩種 UML 圖形間具有以下的相關性：

- 所有 collaboration diagram 中出現的非重複 analysis class 數量總數必須與 class diagram 中的 analysis class 數量總數相同，這是因為所有被定義的 analysis class 都必須出現在 class diagram 中，而這些 analysis class 也必須至少在一個 collaboration diagram 中參與互動。
- 在任何一個 collaboration diagram 中有 link 相連的 analysis class 也會在 class diagram 中有 relationship 相連，因為 collaboration diagram 中的 link 代表著要求另一 class 執行某一 responsibility，因此會在 class diagram 的對應 analysis class 間有 relationship 存在。

同時在本階段的第一個流程會產生 analysis package，用來描述系統的架構，analysis package 與上述的 collaboration diagram 及 class diagram 具有以下的相關性：

- 在 class diagram 中有關係的一對 analysis classes，必須是位於同一個 analysis package 中，或是彼此有關係相連的兩個 analysis package 中。這是由於 analysis class 間的關係也會在 analysis package 間找到相對應的關係，否則就是不一致。

3.1.3 Design 階段

Analysis 階段完成後緊接著的就是 Design 階段，根據 USDP 定義 Design 階

段的主要流程有以下四個：

1. *Architectural Design*：

1.1 定義系統的 node 與 network 並畫成 deployment diagram

1.2 定義系統的 subsystems 以及其提供的 interface, 這些 subsystems 大部份可以參考到 analysis 階段產生的 package, 也有部分的 subsystem 是被用於銜接 middleware 與 system-software layer 差異的 software module，之後並定義各個 subsystem 間的關係並配置到 deployment diagram 上。

1.3 定義 architecturally significant design class。參考 analysis class 先定義對於架構有重要影響的 design class，或是與 middleware 或 system software 有關之 design class。

1.4 Identifying Generic Design Mechanism

2. *Design a Use Case*：

2.1 參考 analysis class 定義參與 use case 的 design class。

2.2 描述 use case 中各個 object 如何互動來實現此 scenario。

2.3 定義 use case 中參與的 subsystems 與 interfaces。

2.4 描述 use case 中各個 subsystem 如何互動來實現此 scenario。

2.5 註記 implementation 階段的 nonfunctional requirement。

3. *Design a Class*：

3.1 Outlining the Design Class。

3.2 定義 design class 的 operation。

3.3 定義 design class 的 attributes。

3.4 定義各 design class 間的 associations 與 aggregations

3.5 定義 design class 間的 generalizations

3.6 描述 operations 的實現方法。(Describing methods)

3.7 透過 statechart diagram 的建立來描述 design object 的 state。

3.8 處理 special requirement

4. *Design a Subsystem*：

4.1 維護 subsystems 間的 dependencies

4.2 維護 subsystems 所提供的 interface

4.3 維護 subsystem 中的 design class

Design 階段最主要的工作是參照 analysis model 建立系統的結構來設計本階段的 design model 並做為 implementation 階段的藍圖，有別於 Analysis 階段較概念化的表示，Design 階段會考慮到實作的 physical model，使用者會因為程式語言選擇的不同而產生具有多樣 stereotype 的 design class。

在經過 Analysis 的階段後，緊接著的 Design 階段就會將上個階段產生的產物當作輸入，加以進一步處理後產生屬於 Design 階段的新的產物，Analysis 階段所產生的 analysis model, analysis class(包含其 attributes 與 responsibilities), collaboration diagram, analysis class diagram, 以及 analysis package 都會在 Design 階段依照系統需求演進成其他的產物，圖 3-3 為此階段主要產物的關係圖，

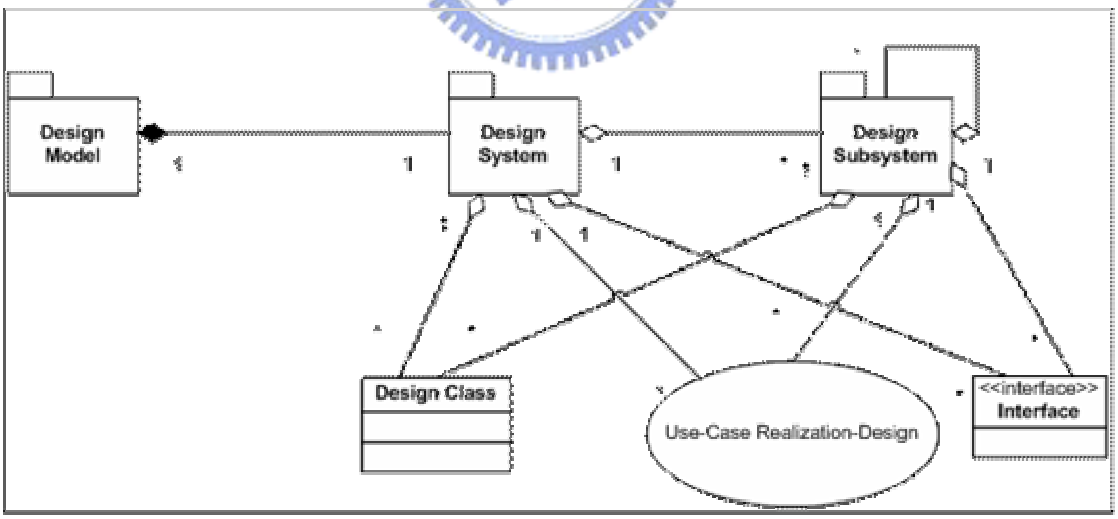


圖 3-3 Design 階段之產物

這些產物包括有：

- *Design model*：用來描述 use case 的 physical realization 方法以及 functional 與 nonfunctional 的需求。

- *Design class*：由上個階段產生的 analysis class 演進而來，本階段中的 design class 將會參考 analysis class 而產生用來實現該 analysis class 的 design class。
- *Use-Case realization-Design*：用這個階段產生的 design class 取代上個階段 interaction diagram 中的各個 analysis class，並建構以 design class 為主的 interaction diagram。
- *Design subsystems*
- *Interface*
- *Deployment model*：用來描述系統的各個 design subsystem 如何分布在各個運算節點(computational node)上，這是在 design 階段所創造出來的產物，與前幾個階段並沒有 traceability 的關係。
- *Architectural description*

本階段產生的產物大多數都能以 UML 標準所規範的圖形來表示，這些圖形包含了 deployment diagram、class diagram、sequence diagram 等，以下將敘述每個圖形在本階段的功能以及所提供的資訊：

- Deployment diagram：被用在本階段的第一個流程，描述系統的 node 架構以及 node 間的 communication，主要提供的內容有兩項：
  a. nodes：系統定義的 node，包含 node 的名稱以及配置在其上的 design subsystems。
  b. relationships：連接兩個 node，通常是用來註明這兩個 node 之間有存在互動或是依存的關係，relationship 上可以標示其特殊的 stereotype，例如<<intranet>>、<<internet>>等。
- Class diagram---design：被用於描述本階段所定義的 design class，其作用與上個階段的 class diagram---analysis 類似，提供的資訊有：
  a. Design classes：系統在本階段所定義的所有 design class。
  b. Relationships：design class 間的 relationships，包括 associations、

aggregations 與 generalizations，用來描述連接的 design class 間的關係。

- Sequence diagram：描述本階段 use case scenario 如何透過 design object 的互動來實現功能，主要所提供的資訊有：
  a. Design objects：參與此 sequence diagram 的 design object。
  b. messages：design object 與另一個 design object 的訊息交換，本階段中 message 主要是由 design object 所屬類別的 responsibility 或 operation 所構成，被呼叫的 design object 必須負責完成該 message 上的 responsibility，本階段完成後所有的 message 都必須是用被呼叫的 design object 所屬類別的 operation 來表達。
- Statechart diagram：在本階段的第三個流程中被用來描述某一個 design object 的狀態轉換，圖中所提供的資訊有：
  a. States：該 design object 在執行過程中所會出現的狀態。
  b. Transitions：包含 events 與 actions，transition 由該 design object 所屬類別的 operation 所構成，並以箭頭指向發生此 event 時 design object 的目標狀態。

根據以上的特性可以歸納出以下幾種相關性：

a. sequence diagram 與 class diagram 的相關性

- 任何一個 sequence diagram 中有互動的 design object 間，在 class diagram 中所屬的對應 class 間必須有關係存在。
- class diagram 中所有的 design class 其 instance 都必須至少參與一個 sequence diagram 的互動。

b. design subsystem 與 class diagram 的相關性

- 每個 class diagram 內的 design class 都必須屬於一個 design subsystem。
- 在 class diagram 內有關係相連的一對 design classes 必須屬於同一個

design subsystem，或是有關係相連的一對 design subsystem 上。

c. design subsystem 與 sequence diagram 的相關性

- 任何一個 sequence diagram 中有互動的 design object 間，其所屬的對應 subsystem 間必須有關係存在。

d. design subsystem 與 deployment diagram 的相關性

- 所有定義的 design subsystem 都必須要出現在 deployment diagram 上，這是因為每個 design subsystem 根據定義都必須被配置在一個 node 上。
- 有關係的一對 design subsystems 必須被配置在同一個 node 上或是有 link 相連的兩個 node 上。

3.2 跨階段產物間相關性之分析

3.2.1 from Requirement Capturing to Analysis

從 requirement capturing 階段進到 analysis 階段時，主要工作之一是藉由每一個 use case 的 detail specification 來產生 analysis 相對應的 collaboration diagram。

每一個 collaboration diagram 代表著一個 use case 裡定義的 scenario，此 diagram 的主要組成元素為 analysis classes 以及該 analysis class 參與此活動的 responsibilities，以 Pay Invoice 這個 use case 為例，圖 3-4 為 Pay Invoice use case 的 basic path 的 collaboration diagram。

任一 collaboration diagram 的 analysis class 與其 responsibility 都是依據 use case scenario 的 specification 來定義的，因此兩者是一一對應的，然而 scenario 的執行流程是以敘述式的語句來表達，這些句子以口語敘述，因此會夾雜著多餘的語詞如 therefore、later 等，開發人員不易從中判斷需用哪些 analysis classes 及 responsibilities 來表示其合作關係，因此有必要定義一種新的 use case 表示方式以幫助開發人員易於定義 analysis class 及 responsibility，最好使其能做到一一對應。

Collaboration diagram 中的每一個 message 代表一個動作，而 analysis class 主要有 boundary class、entity class 及 control class 三種，因此將圖 3-1 的 scenario 改寫為圖 3-5 的形式，表格的欄(Column)代表的是 actor 或是 system，列(Row)中敘述為所做的動作，每個動作代表一個 responsibility，動作前的數字則是用來標明動作的順序，巢狀化的動作(nested actions)則以階層式的 sequence number 來表示。表格內只記載了 flow of event 中的重要動作，在 actor 行內的動作代表是此 actor 對 system 輸入的資料，而在 system 行內的動作代表此系統所必須負責完成的動作。表格中的動作可分為下列兩種：

1. 一般性的動作，即由(動詞+名詞)或(動詞+片語)的形式所構成的 responsibility，如 Pay Invoice 範例中的 Get invoice、Generate payment request，這種類型的動作其本身就是一個 responsibility。
2. Loop, branch 等於流程控制有關的動作，如 Pay Invoice 範例中的

if (there is enough money in the buyer's account)
then schedule payment for this invoice

這種與流程控制有關的動作是由 condition 與 responsibility 所組成，在這個例子中的 condition 條件是必須滿足"there is enough money in the buyer's account"，才會啟動其 responsibility (即 schedule payment for this invoice)。

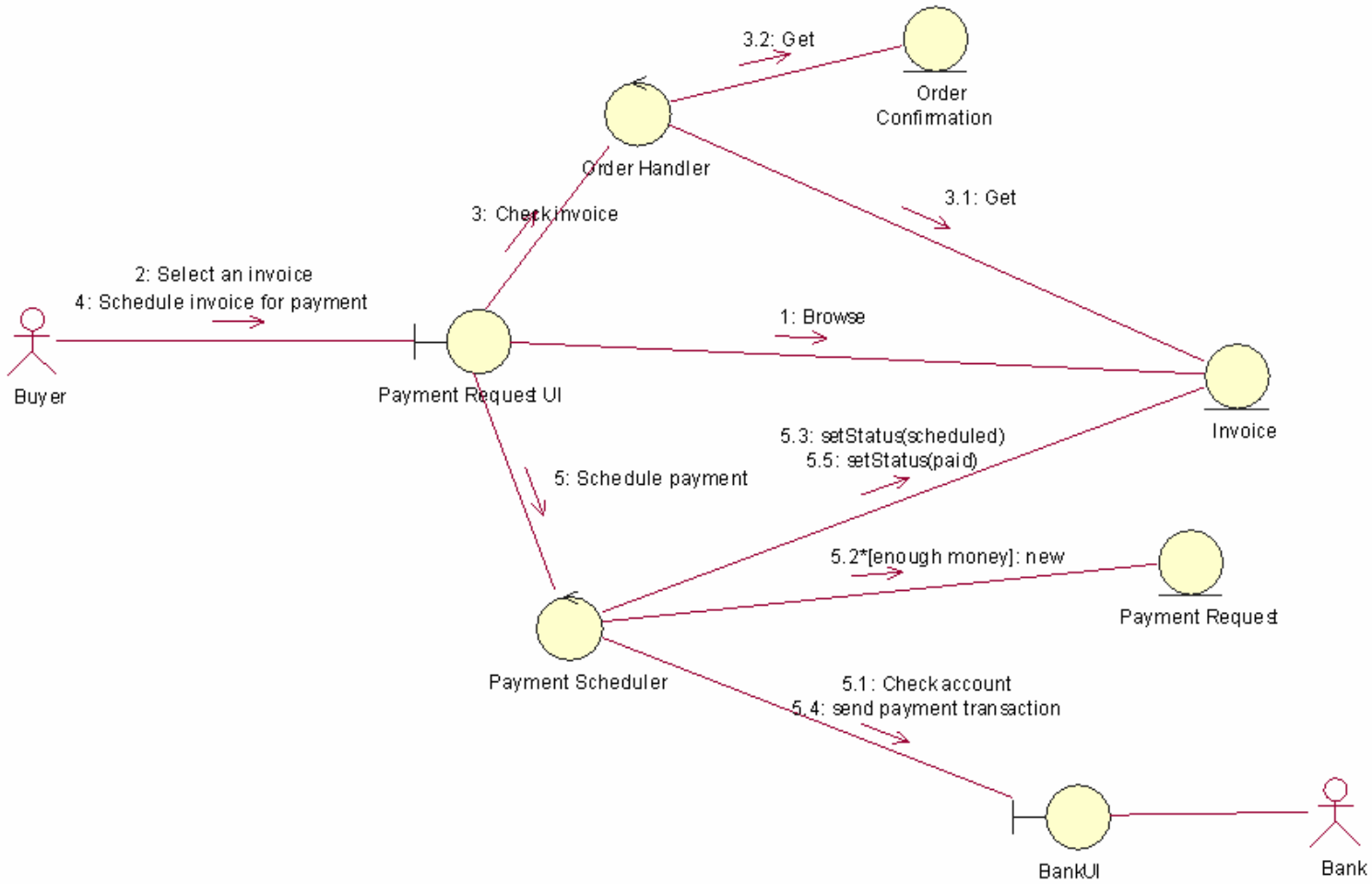


圖 3-4 Pay Invoice use case—basic path 的 collaboration diagram

Buyer	*System*
	1. Display invoice list
2. Select an invoice	
	3. Check invoice to make sure goods were received 3.1 Get invoice 3.2 Get Order Confirmation of this invoice
4. Schedule invoice for payment	
	5. Schedule payment 5.1 Check account 5.2 *if* (there is enough money in the buyer's account) *then* generate a payment request 5.3 set the status of this invoice to "scheduled" 5.4 send payment transaction to bank 5.5 set the status of this invoice to "paid"

圖 3-5 Pay Invoice use case—basic path 的 scenario 表格

collaboration diagram 的 entity class 代表系統裡 long-lived 且 persistent 的資料，以名詞的形式描述於 scenario 中，因此 entity class 的名稱可以藉由參考 scenario 裡的重要名詞而得到，為了讓系統可以自動檢查 entity class 名稱是否來自於 scenario 裡的名詞，使用者可在表格內標示重要的名詞，這些名詞將做為 entity class 名稱的 candidate。圖 3-6 以粗體字的方式標示出 Use Case：Pay Invoice---Basic Path 表格內的重要名詞。

Buyer	*System*
	1. Display **invoice** list
2. Select an **invoice**	
	3. Check **invoice** to make sure goods were received 3.1 Get **invoice** 3.2 Get **Order Confirmation** of this **invoice**
4. Schedule **invoice** for payment	
	5. Schedule payment 5.1 Check account 5.2 *if* (there is enough money in the buyer's account) *then* generate a **payment request** 5.3 set the status of this **invoice** to "scheduled" 5.4 send payment transaction to bank 5.5 set the status of this **invoice** to "paid"

圖 3-6 Pay Invoice use case—basic path 的 scenario 表格(標記名詞)

另外由圖 3-6 的 step3.2 可發現 invoice 的內容需有 order number，才能檢查此貨物是否已送達，因此在表格旁加一欄來做註解如圖 3-7 所示，註解內亦標示了重要名詞，對未來定義 entity class 的 attribute 有重要助益。

Buyer	*System*	/*Comment*/
	1. Display **invoice** list	
2. Select an **invoice**		
	3. Check **invoice** to make sure goods were received 3.1 Get **invoice** 3.2 Get **Order Confirmation** of this **invoice**	Invoice should have **order number**, **good lists**, **total price**, **seller's account** and a **status** field. Order Confirmation should have **order number**, **good lists**, **total price**, **delivery information** and **buyer's personal information**.
4. Schedule **invoice** for payment		
	5. Schedule payment 5.1 Check account 5.2 *if* (there is enough money in the buyer's account) *then* generate a **payment request** 5.3 set the status of this **invoice** to "scheduled" 5.4 send payment transaction to bank 5.5 set the status of this **invoice** to "paid"	Payment request should have **seller's account**, **buyer's account**, **amount of money**, and **scheduled date**.

圖 3-7 Pay Invoice use case—basic path 的 scenario 表格(加入註解)

Boundary class 負責 actor 與系統的溝通，在 scenario 的表格中若出現 actor 與 system 間有緊鄰著的動作出現(如 Pay Invoice---Basic Path 中的"2. Select an invoice 與"3. Check invoice"，或" 4. Schedule invoice for payment"與"5. Schedule payment")，則代表為有 actor 與 system 執行 I/O 的動作出現，必須藉 boundary class 作為兩者間的溝通介面，I/O 動作有輸出後接著輸入，及只有輸出兩種，此 boundary class 的 responsibility 必須能執行此動作。預先定義一些 I/O 動作類別的動詞將有助於判斷該 responsibility 是否為屬於 boundary class 的 responsibility，這些動詞包含了 system to actor 與 actor to system 兩類，圖 3-8 列出幾個常用的 I/O 動詞。

system to actor 的 I/O 動詞	actor to system 的 I/O 動詞
• Display • Show • Output • Present	• Select • Input • Enter

圖 3-8 常用的 I/O 動詞列表

第三類 analysis class 為 control class，control class 所扮演的是 controller、coordinator 或是執行複雜動作的 worker 等角色，系統中的所有 coordination、transaction、sequencing 以及對其他 object 的控制都是 control class 的 responsibility。由於使用者對於 control class 的產生數量以及每個 control class 的任務分配並不一定，所以無法藉由參考 scenario 來決定 control class 的數量與名稱，但從例子中觀察得知若是 scenario 裡的 responsibility 有具有巢狀的形式，則 root 的那個 responsibility 必定是 control class 的 responsibility。

3.2.2 from Analysis to Design

從 analysis 階段進入 design 階段的主要工作是將上個階段的 analysis class 以 design class 來實現、參考 analysis package 以產生 design subsystem、產生各 use case scenario 的 sequence diagram 以描述 design object 互動的關係、定義系統架構並以 deployment diagram 來表示此架構。

由於 sequence diagram 也是代表一 use case scenario 之 object 互動順序關係，因此與上階段的 collaboration diagram 間有密切關係；analysis class 在設計階段也

需出現，因此兩階段間之 class diagram 也有關係存在。

以下分別由 sequence diagram 及 class diagram 為出發點討論此兩階段產物間的相關性：

I. *Sequence diagram*

Sequence diagram 由 design objects 及 messages 兩個元素所組成，圖 3-9 是 pay invoice 的 basic path 的 sequence diagram，sequence diagram 上共有九個 design objects，這些 design objects 所屬的 design class 是為了要實現 analysis class 的 responsibility 而產生，比較圖 3-9 與圖 3-4 可發現，design class 的數量比 analysis class 的數量多，如多了處理 invoice 的”Invoice Manager”與處理 order confirmation 的”OrderConfirmation Manager”，這兩個新增的 design class 都是屬於 container class 的類型。這種衍伸結果可知一個 analysis class 會在 design 階段由一個以上的 design classes 所實現，其中有一個 design class 會繼承原先 analysis class 的名稱以及 responsibility，其他新產生的 design class 則各有其產生的原因，詳細說明如下。

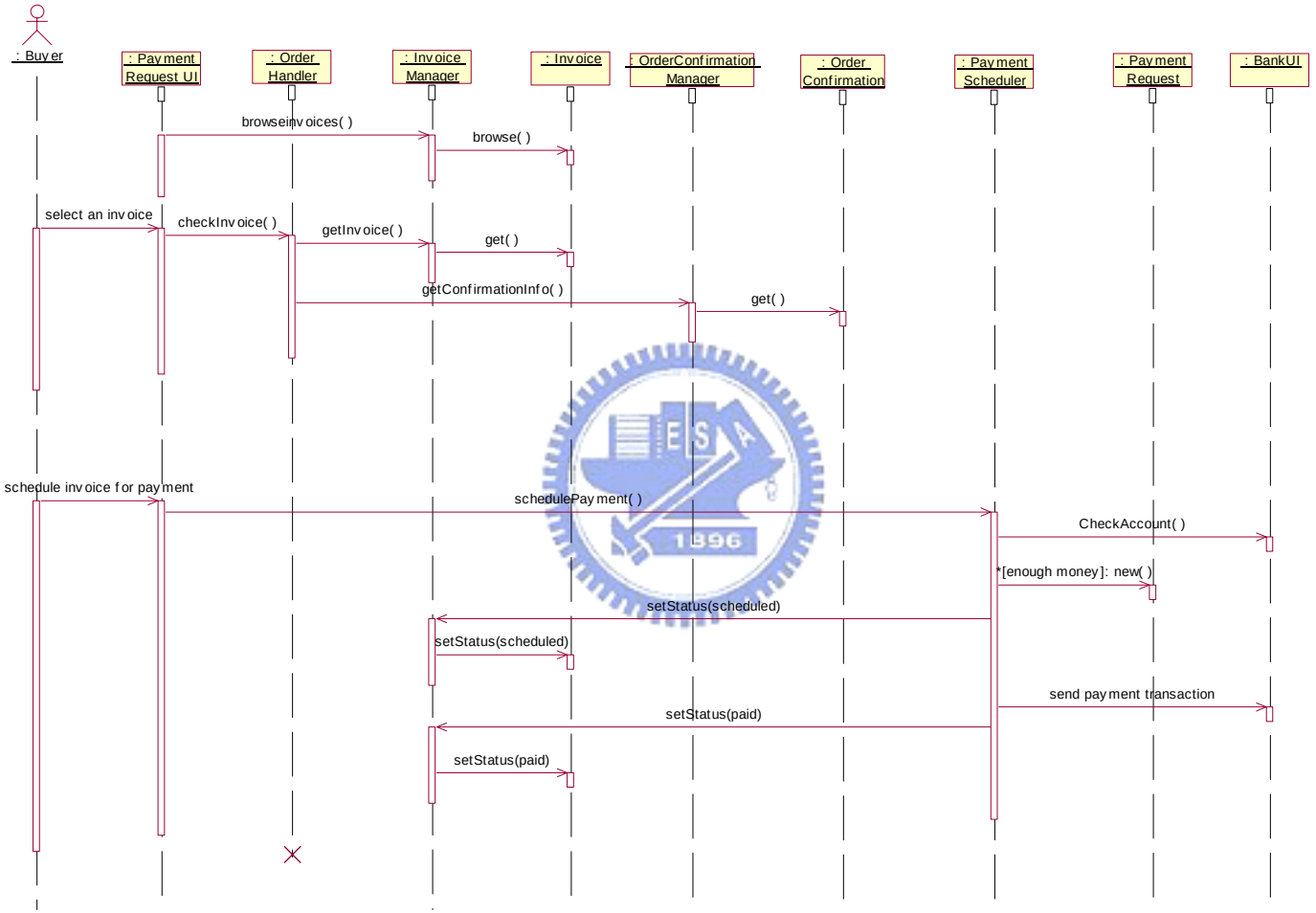


圖 3-9 Pay Invoice use case—basic path 的 sequence diagram

Sequence diagram 中 design object 所屬的 design class 有部分可以追溯回上階段的 analysis class，另一部分則是在本階段新增而無法追溯回上個階段的，首先探討 design class 的產生來源，可先將 design class 分為下列三類：

1. primary design class：可直接 trace 回 analysis class 的 design class 稱為 primary design class。每個 analysis class 都會在 design 階段產生一個 primary design class，這個 primary design class 必須負責實現 analysis class 的 attribute 與 responsibility，例如圖 3-9 中的 Payment Requirement UI、Order Handler 等。Primary design class 的名稱必須與其所追溯回的 analysis class 名稱相同。
2. 輔助 primary design class 的 design class：由於 primary design class 之 responsibility 所採用的 algorithm 過於複雜或是不同 data type 之間的轉換等原因而產生輔助性質的 design class。
3. 採用既有的 middleware 或 system-software 時，為了將其中的資料型態轉換為本系統使用的資料型態，或是為了系統可攜性(portability)而加入的 adaptor classes：這些 design class 是來自於轉換或協調 middleware 或 system-software subsystem 與發展中的系統之差異，或是為了提高系統可攜性與相容性、降低對於系統外部 underlying infrastructure 的依存度而產生。

在 analysis 階段有三種類型的 analysis class，分別是 boundary class、control class、與 entity class，以下將分別敘述在 design 階段實現三種不同類型 analysis class 所產生的輔助性 design class：

1. Control class：具有以下兩種輔助性質的design class。

   (1)為了輔助primary design class完成operation所產生的design class。

   Primary design class 中有些較複雜的 responsibilities，將分解為多個 design classes 以降低單一 class 的複雜度，此 responsibility 需要借助其他新

增 design class 的 operation 來完成。

例如某個 control class 具有將一篇文章中某些特定字元出現次數編碼為 Huffman code 的 responsibility，此 responsibility 必須同時考量計算的效能與相關演算法的使用，若直接將這個複雜的功能放進原本的 primary design class，容易造成實作與維護時的複雜度；為了減少 class 的複雜度，使用者可能會新增一個負責做 Huffman coding 處理的 design class，並藉由 primary design class 呼叫此新增的 design class 所提供的 operation 來完成這個複雜的 responsibility。

(2)為了協調兩個design class工作而產生的coordinator class。

這種 design class 的產生情況通常是因為 remote procedure call 或分散式運算，多個 design classes 被 deploy 到不同的 nodes，此時需要產生新的 design class 負責 coordinate 的工作。例如本地端的 design class A 須要藉由溝通互動與另一個網路節點上的 design class B 共同完成某一個 responsibility，design class A 必須等待 design class B 完成某項工作後傳遞資料回來才能繼續執行下去，這時候使用者可以選擇在本地端新增一個 coordinator class 來觀測不同節點上 design class 目前的工作處理情形。產生 coordinator class 的優點是當遠端的 design class 執行發生異常狀態時，本地端的 design class A 不會無謂地一直等待，而是透過 coordinator class 的輔助來提供 design class A 繼續執行所需的預設資料，或是提供 design class A 另一個可以獲得資料的來源。

2. Entity class：具有以下兩種輔助性質的design class。

(1)輔助管理多筆資料的container class。

依照單一 transaction 中屬於某 entity class 的 object 在系統中所出現的數量可將 entity classes 分為兩類。

第一種類型的 entity class 是單一 transaction 可明確指定單一特定 object

的 entity class，這種類型的 object 通常用 object ID 來查詢，而且只會找到一個 object，例如報表或是提交的表單。

第二種類型的 entity class 則是在單一 transaction 中需要由多個 objects 選取符合指定條件的 objects，通常用數個 attribute 的集合來查詢這類 objects，查詢結果可為多個 objects，例如資料庫中的訂單紀錄或是會員資料。這種類型的 entity class 需要輔助性質的 container class 來負責查詢的工作。在 Pay Invoice 範例中的 Invoice 就是這種類型的 entity class，所以必須產生 container class “Invoice Manager”。

使用者進入 design 階段時，依照這兩種不同的特性來產生 entity class 的輔助性 container class，負責做 object 的 select 與查詢。

例如若有一個第二種類型的 entity class 代表客戶的訂單資料，因為訂單資料具有很多個資料欄位，查詢時常常需要對各個欄位設定條件以查詢到符合的資料，此時使用者就必須產生一個從事查詢客戶訂單工作的資料庫 container class，所有的查詢都必須經由這個 container class 才能完成。

(2)為了實現attribute type的design class。

因為 entity class 或 container class 的 attribute 可能是較複雜的資料結構，使用者所使用的程式語言本身並不支援，所以使用者必須新建立 design class 來實現此 attribute type。

同樣重覆上個例子，訂單資料的 entity class 其資料欄位除了單純的訂單編號、客戶姓名、送貨地址等字串類型的欄位資料外，也可能會有表列式(list)的資料結構，如訂貨明細等，這些資料結構可能不是所採用的程式語言原本就有提供的資料結構，必須產生新的 design class 來負責完成這個表列式的 attribute type。

3. Boundary class：具有以下三種輔助性質的design class。

(1)提高系統對不同components相容性的design class。

一個 boundary class 所提供的 service 可能在 design 階段被分割成許多 components，每個 component 可能對應到實際硬體設備或軟體元件。為了方便系統維護與升級，降低對這些 component 的依存性，會產生新的 design classes。

比方說 analysis 階段的 SysOutput boundary class 可能會在 design 階段形成 printer 與 monitor 兩個 components，以及一個 design class 負責 driver 的更新與維護。

(2)為了實現attribute type的design class。

提供 service 的 design class 可能含有非程式語言支援的 attribute type 或是 data structure, 使用者必須新建立 design class 來實現此 attribute type。

比方說有一個負責讀入晶片卡的 boundary class，因為該 class 要讀取的資料可能是一個類型為字串欄位的集合的個人資料，而這種 attribute type 並不是使用者所採用的程式語言所支援的類型，所以使用者必須產生一個新的 design class 來負責實現這個字串集合的 attribute type。

(3)為了輔助primary design class的operation所產生的design class。

Primary design class 中有些較複雜的 responsibilities，可能分解為多個輸出入形式以對應不同的 actor、不同的輸出方式、或複雜資料的處理，classes 為降低單一 class 的複雜度，會透過呼叫其他新增 design class 的 operation 來完成。

延用上一個例子，讀取晶片卡資料的 boundary class 其中有一個 responsibility 是讀入晶片卡上的個人資料，因為讀入資料的 responsibility 可能會先經過安全保證的演算法來計算取得資料的合法性，所以被使用者視為是較複雜的 responsibility，而為了不增加 primary design class 本身的複雜度，使用者可能會考慮新增一個 design class 來負責做讀取的工作，而 primary design class 僅需要呼叫該輔助的 design class 所提供的 operation 就能實現 responsibility。

除了以上說明的 design class 主要產生原因外，在 analysis 階段中使用者可能會在 analysis class 上加註 special requirement，這些 requirement 大多數為 nonfunctional requirement，例如因為 fault-tolerance 的需求而產生新的 design class 來處理資料儲存及復原，像這類的 design class 無法明確直接地 trace 回 analysis class，但有增強其功能之用途，因此必須在一致性確認的時候提醒使用者。

為方便之後的討論，定義一名詞 *class group* 為由一個 design class 與其他輔助它的 design class 所形成的集合，class group 的產生來源主要有兩個：

1. 直接從 analysis class 發展成 primary design class，並與輔助其功能的 design class 形成一個 class group。
2. 由 user 定義 depend on middleware 或 system-software 的 design class 與其它因此 design class 而產生的 design class 所組成，此種 class group 中不含有 primary design class。

class group 具有下列特性：第一，每個 design class 都必須屬於至少一個 class group，而如果該 class group 只有一個 design class，則代表那個 design class 並不需要輔助的 design class。例如 design class A 若可以 trace 回 analysis class B，但 A 的所屬 class group 除了 A 以外並沒有其他的 design class，則可以得知 analysis class B 在經過前述輔助性 design class 產生與否的檢查後並沒有產生其他輔助的 design class。而若此 design class 無法 trace 回任何一個 analysis class，那就代表這個 design class 是 depend on middleware 或 system-software 所產生的 design class，且不需要其他輔助性質的 design class。

第二，一個 design class 如果同時屬於兩個以上的 class group，那這個 design class 一定具有以下的特性之一：

1. 該 design class 所提供的 attribute type 被兩個以上的 class group 內的 design class 所繼承或使用到。
2. 該 design class 所提供的 operation 被兩個以上的 class group 內的 design class 的 operation 所呼叫到。

而 sequence diagram 也與 collaboration diagram 有相關性，這些相關性歸納如下：

Sequence diagram 與 collaboration diagram 同為 UML 語言內所規範的 interaction diagram，collaboration diagram 著重在描述 class 之間的組織與互動關係，sequence diagram 上的 messages 不再如 collaboration diagram 一樣是 class 的 responsibility，而是所要呼叫的 operation，但這個 operation 卻可以追溯到 analysis 階段 collaboration diagram 內的 responsibility，這些相關性有：

1. 兩個原本在 analysis 階段的 collaboration diagram 內有 link 相連的 analysis class，會在 design 階段形成兩個對應的 class group 內之 design class 的 link。圖 3-10 是擷取自 Pay invoice 在 analysis 階段的 collaboration diagram，從 boundary class “Payment Request UI”會呼叫 entity class “Invoice”的 responsibility “Browse”，這個 responsibility 是用作查看系統內的 invoice 資料；

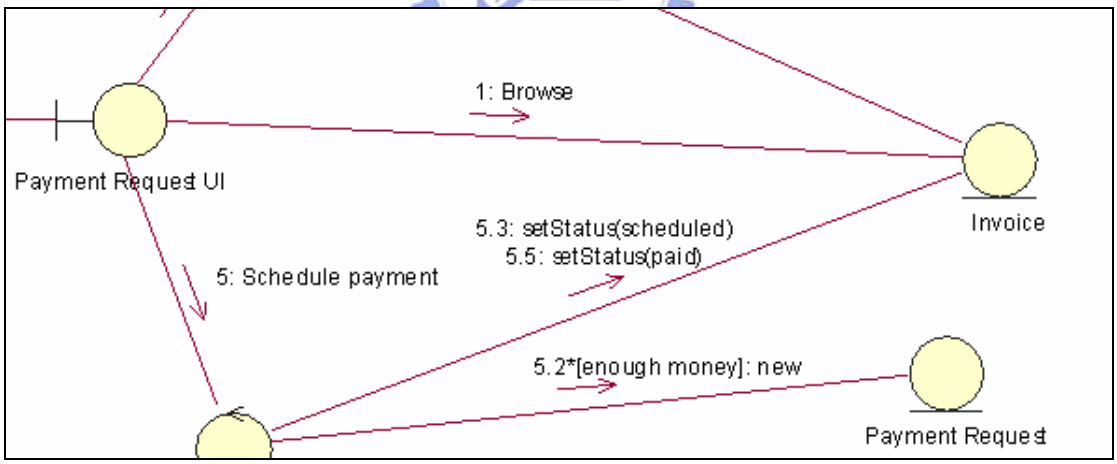


圖 3-10 Pay Invoice use case—basic path 的 collaboration diagram(部分)

而圖 3-11 是擷取自該 scenario 在 design 階段的 sequence diagram，圖中的 Payment Request UI 屬一個 class group，Invoice Manager 與 Invoice 則是屬於另一個 class group，原先在上個階段 boundary class “Payment Request UI”與 entity class “Invoice”間的 link 會在本階段由 Payment Request UI 與 Invoice Manager 間的 link 所繼承。

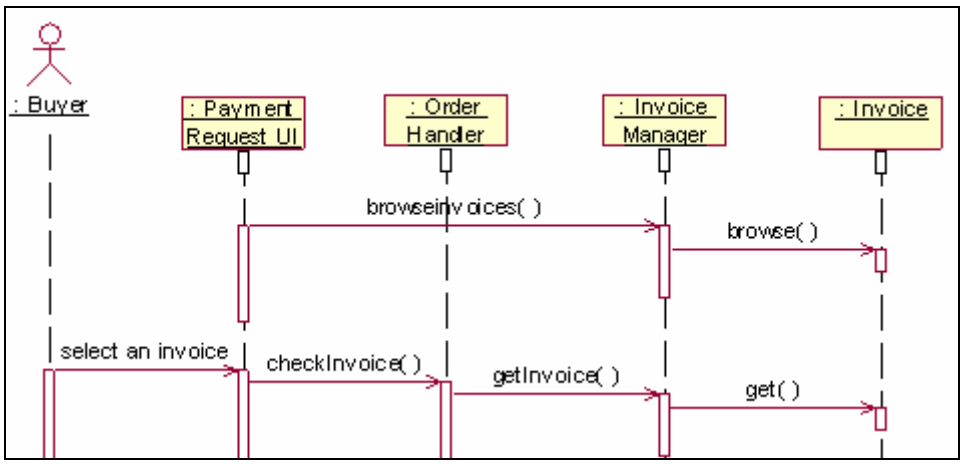


圖 3-11 Pay Invoice use case—basic path 的 sequence diagram(部分)

2. Analysis 階段 collaboration diagram 內的每個 responsibility 都會在 design 階段形成一連串的 operation 動作(a sequence of operations)，而這一連串的動作中會有一個 operation(也只有一個)是跨 class group 來傳遞，此 operation 直接繼承自 responsibility，為了解釋方便我們稱這個 operation 為 *primary operation*，如圖 3-11 中，object Payment Request UI 呼叫 Invoice Manager 的 operation “browseInvoices”就是一個 primary operation。由 primary operation 的定義可得知 primary operation 的數量必須與 responsibility 的數量相同，並為 1：1 的對應關係，而 primary operation 的 callee class 會依其所追溯回的 analysis class 類型而有所不同，以下分兩個 case 來討論。

當在 analysis 階段被呼叫的 analysis class 是 control class 或 boundary class 之一時，原 analysis class 間的 responsibility 傳遞會由 caller class 的 class group 內的 design class 傳送到被呼叫的 class 對應的 class group 內的 primary design class，即 primary operation 的接收者必須是 primary design class。

而如果在 analysis 階段被呼叫的 analysis class 是 entity class，此時兩種類型的 entity class 會有不同的發展模式，若該 entity class 沒有對應的 container class，其 responsibility 由 primary design class 內的 operation

所負責；但若該 entity class 有 container 產生，則 primary operation 由 container class 所接收，因為該 responsibility 必須透過 container class 才能完成。

總結以上分析，在 sequence diagram 中的 design class 可歸納為以下數種：

1. 與 analysis class 對應的 primary design class。
2. 輔助 primary design class 的 design class，這些 design class 必須是因為輔助 primary design class 而產生的 design class。
3. Depend on middleware 或是 system software 的 design class 與輔助它的 design class。

II. *Class diagram*

Design 階段的 class diagram 由 design class 與其間的 relationships 所組成，同樣的此階段的 class diagram 也可以參考 sequence diagram 及上階段的 analysis class diagram 來建立 relationships，歸納此兩階段間 class diagram 的相關性如下：

Analysis 階段所產生之 class diagram 的 relationship 如何傳遞到 design 階段的 design class diagram 呢？有了 class group 的劃分，就可以將原本 analysis class 間的關係轉成 class group 間的關係，也就是這兩個相對應的 class group 內必須有至少兩個 analysis class 有 link 相連。

比方說 class group A (trace back)-> analysis class A，

class group B (trace back)-> analysis class B，

則 analysis class A 與 B 之間的關係由 class group A 與 group B 裡面的 design class 所繼承，即 group A 中若無 design class 與 group B 的 design class 有關係，則會發生不一致，將產生 inconsistency message 提醒使用者。

上述分析，已將 requirement capturing 階段到 design 階段各階段的產物分析

出跨階段各個產物間的相關性、一致性、追溯性與完整性，下一章將依照此章節所提出的各產物特性來設計輔助軟體開發流程的系統。

第4章、具跨階段產物一致性檢查及輔助機制之設計

上一章已對將各個階段產物的相關性做了詳細的分析，本研究擬在現有的 open source 開發工具「ArgoUML」加入產物的相關性進行一致性檢查的機制，並提供使用者一些開發輔助機制，以避免產物間發生不一致造成錯誤。本章 4.1 節說明設計策略，4.2 至 4.4 節分別說明需求、分析、及設計階段之機制設計。

4.1 設計策略

依據上一章對跨階段產物間的相關性分析，可了解產物間之一致性關聯關係分為兩大類，(1)同階段 UML 圖形產物之關聯性，如 class diagram 上的 class 名稱必須與 collaboration diagram 對應的 class 名稱一致；(2)跨階段 UML 圖形間的追溯性所產生的關聯性，如 design class 與 analysis class 之間的關聯性，因此 USDP 流程中的 UML 圖形產物間一致性檢查的策略可制定如下：

1. 根據原有的 UML 圖形關聯性設計一致性檢查方法：根據上一章對同階段與跨階段產物間關聯性的分析與整理，各階段的 UML 圖形與其上階段的 UML 圖形或是同階段的 UML 圖形間原本就存在著關聯性，這類型的關聯性大多是指某兩個 UML 圖形間的元素的出現數量、種類、名稱必須一致，不能有拼字錯誤、遺漏、或多餘的情形發生；此類型的一致性檢查是當使用者完成一個流程時針對圖形的完整性進行檢查。
2. 針對跨階段產物間的追溯性提出檢查方法：這類型的檢查需要使用者提供額外的相關資訊，本研究將提出這些額外資訊應包含的項目及將既有的 UML 產物表現方式加以改寫。例如，藉由改寫 use case 的 scenario 表現方式以及對 scenario 裡的重要名詞進行標示，就可以讓系統自動檢查下個階段所產生的對應 collaboration diagram，及用 checklist 的方式，讓使用者標註本階段產物與上個階段產物內元素的對應關係以達到產

物間追溯性的目的，如圖 4-1 所示。

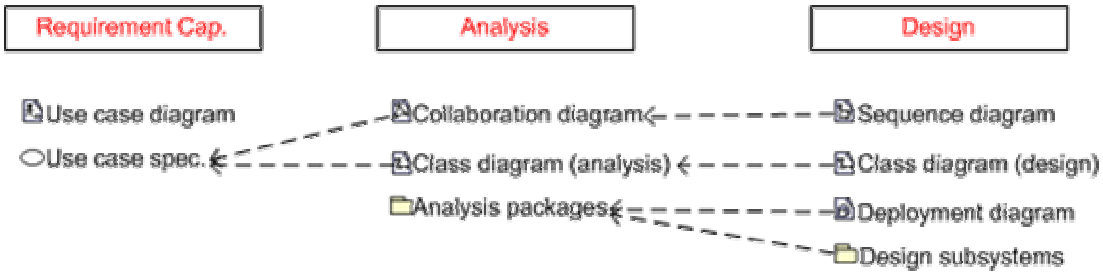


圖 4-1 USDP 各階段主要產物之追溯關係

依據此策略，本系統所設計的功能機制可分為下列三類：

1. 流程進行輔助機制：在使用者進行設計而畫 UML 圖形時，適時提供已有元素供使用者選用，以避免錯誤的發生，如提供使用者 responsibility 的清單，讓使用者從中挑選合適的 responsibility。
2. 同階段性 UML 圖形一致性檢查機制：當使用者完成某一個流程階段或完成一個圖形繪製，系統自動檢查此流程產生的產物與其他已產生的產物間是否有不一致的情形，例如使用者畫完一個 use case 中的 collaboration diagram 時，檢查該圖中的 class 是否在已有的 analysis class 清單中。
3. 階段產物的完整性檢查：當使用者完全完成 analysis 或 design 整個階段時，系統所進行的整體檢查，確定使用者確實完成這個階段所必須產生的產物，產物若不齊全就不允許使用者進入下一個流程，例如幫助使用者在 analysis 階段完成時檢查是否每個 use case 裡的 scenario 都已經有建立相對應的 collaboration diagram。

4.2 需求階段(Requirement Capturing)

Requirement capturing 階段是 USDP 軟體開發工作流程中的第一個階段，本階段使用者必須以 use case diagram 來表示系統所定義的 use cases 與 actors，同時註明各個 use case 的 specification。在 requirement capturing 階段系統提供了使

用者建立 use-case model，並依照上一章提到的標記方法標記各 scenario 裡的關鍵名詞，這些關鍵名詞將輔助使用者進行 analysis 階段的流程。

Requirement capturing 階段的流程共分為五個項目，分別是 find actors and use cases、prioritize use cases、detail a use case、prototype user interface 與 structure the use-case model，各流程詳細的輔助功能茲列述如下：

1. Find actors and use cases

系統提供使用者藉由繪製 use case diagram 來表示其所定義的 actor 與 use cases，使用者在 use diagram 中加入並同時定義 use case 與 actor 的名稱，以及 use case 與 actor 間的關係。繪製 use case diagram 的功能是由 ArgoUML 所提供。

2. Prioritize use cases

給予每個 use case 一個優先度欄位，使用者於本流程可設定各 use case 的優先度，系統會以優先度排序各個 use case，同時提醒使用者要在開發流程的前期先進行具有高優先度 use case 的開發。

3. Detail a use case

本流程中使用者將註明各個 use case 的 specification，上一章已說明了 specification 裡的欄位，包含了 use case 的各種相關資訊如 use case name、precondition、flow of event、postcondition 等，其中 flow of event 記載的是 use case 內各種可能發生的 scenario。本研究提出以表格的方式描述 use case 內的 scenario，故系統中將提供使用者 scenario 表格以書寫其中的動作，同時提供使用者標記 scenario 裡的關鍵名詞。

4. Prototype User Interface

使用者於本流程中可為系統的介面建立雛型，因使用者介面並不會對下一個階段的產物產生追溯性的影響，故本系統並沒有在這個流程中提供特別的輔助功能。

5. Structure the use-case model

本流程中使用者必須為各個 use case 建立其間的關係，依據 UML 語言的規範，use case 間具有 generalization、extend、與 include 三種關係，系統提供使用者在 use case diagram 中以拖曳的方式建立 use case 間的關係。

4.3 分析階段(Analysis)

Analysis 階段有四個核心活動，分別是 architectural analysis、analyze a use case、analyze a class 與 analyze a package，各核心活動之輔助功能分述如下：

1. Architectural Analysis

1.1 定義系統中的analysis package、service package，與建立analysis package間的依存關係

系統在這個流程中提供使用者建立 packages 的工具，要求使用者為每個 package 命名及建立 packages 間的 dependency 關係，若 package 間有巢狀(nested)的關係，則要求使用者在 package 的性質內註明該 package 所屬的 parent package 及包含的 child package。

1.2 定義obvious entity class

使用者應定義對於該系統架構有重要影響的 entity class，系統會提供使用者在 use case scenario 中標記的關鍵名詞及其出現次數，供使用者選擇產生這類型的 entity class。

1.3 定義Common special requirement

使用者應定義一些共通的 special requirement，系統將會產生文字表格供使用者填寫。

2. Analyze a Use Case

2.1 定義analysis classes

使用者定義一個 use case scenario 之 analysis class 的名稱時會參考此 use case scenario 中使用者已標記的重要名詞，這些名詞將是 entity class 名稱的

candidate。系統將提供 scenario 上的關鍵名詞讓使用者選擇當作 entity class name，若該名詞已被定義為 entity class name，也將自動標明，當 scenario 內的名詞被用來當 entity class 名稱，將特別標記。

因為 boundary class 的名稱一般不會出現在 scenario 裡，因此無法提供 boundary class 名稱的協助，但能提供此 scenario 中共有幾次的 I/O 動作予使用者參考。而 control class 是使用者依系統要執行的動作之複雜度而決定是否由 control class 來負責，並由使用者依工作性質來命名，因此無法協助 control class 的命名。

2.2 描述analysis object間的interaction

使用者需建立 collaboration diagram 來描述該 scenario 內的 analysis class 如何互動。

系統將提供使用者已定義的 analysis class 供使用者選擇，scenario 敘述句中名詞若為 analysis class 的名稱，將作標記以提醒使用者。當使用者加入 boundary class 時，將提供使用者在 scenario 上出現的 actor 名稱供使用者建立 actor 與該 boundary class 之關係，並提供 I/O 相關的 scenario 敘述句供使用者建立 link 及制定 boundary class 之 responsibility。

當使用者加入由 control class 或 boundary class 與 entity class 間的 link 時，系統將提供 scenario 有該 entity class 的敘述句供使用者建立 entity class 之 responsibility 之用。當 boundary class 或 control class 與另一 control class 間建立 link 時，則提供其他 scenario 中仍未被使用到的敘述句供使用者建立該 control class 之 responsibility 之用。

當使用者繪製完一個 scenario 的 collaboration diagram 時，系統會提供以下的檢查：

(1) 檢查 scenario 中是否有尚未被加入 collaboration diagram 內的 responsibility，若有則提醒使用者。

(2) 檢查 collaboration diagram 中 responsibility 的數量是否與 scenario

中列在 system column 內的 responsibility 數量相同，若不相同則代表有尚未被加入圖形中的 responsibility，必須提醒使用者。

(3) 檢查 scenario 中是否有已定義的 analysis class 沒有出現在 collaboration diagram 中，若有則提醒使用者。

2.3 Capturing special requirement

定義此 use case 的 nonfunctional requirement 供 design 或是 implement 階段之用，系統將會產生文字表格來讓使用者填入這些資訊。

3. Analyze a Class

3.1 定義analysis classes所負責的responsibility

列出該 analysis class 在上一階段各個 use-case realization 所代表的 collaboration diagram 中的 responsibilities 讓使用者檢閱。這部分是系統自動完成的，直接對應到各個 scenario 的 collaboration diagram。

3.2 定義analysis class的attributes

Scenario 中經過標示的名詞有些已經被指定為 entity class 的名稱，而剩下未被指定的名詞將當作 analysis class 的 attribute 或是 attribute type，而 entity class 的 attribute 可從 comment 欄位中的關鍵名詞取得，這些資訊將自動產生以供使用者選用。

3.3 定義analysis class之間的associations與aggregations

將 collaboration diagram 中與該 analysis class 有 link 相連的 analysis class 列出供使用者參考建立 association 與 aggregation 之關係。

3.4 定義analysis class之間的generalizations

系統將列出所有 analysis class 裡有相同的 attribute 或 responsibility 者給使用者參考，供使用者建立 analysis class 間 generation 的關係。

3.5 Capturing special requirement

analysis class 之 nonfunctional requirement 將提供 design 階段或

implement 階段處理之用。系統將會產生文字表格來讓使用者填入這些資訊。

4. Analyze a Package

此階段的工作是將所有已定義的 analysis class 分配到適當 package 中，系統將會提供下列輔助：

(1) 列出所有的 analysis class 與該 analysis class 所參與的 collaboration diagram 及 scenario 名稱。

(2) 當 analysis class 被配置進某一 package 中時，系統提供與該 analysis class 有 link 的 analysis class 及出現的 collaboration diagram 名稱，供使用者選擇分配 package 之參考，這些相互間有 link 的 analysis class 會有相當大的機會被分配到同一個 package。

(3) 當全部的 analysis class 都配置進 package 後，系統自動列出每個 package 間的 analysis class 跨 package 呼叫的次數，以提醒使用者將次數高的 analysis class 重新分配所屬的 package。

在完成上述四個 analysis 階段的核心活動後，系統會進行整體檢查以保證能無誤的進入下一個階段，本階段的整體一致性檢查包括：

(1) 系統若在各 use case scenario 中發現有未被使用到的關鍵名詞，則必須告知使用者檢查是否有未定義的 attribute 或 attribute type。

(2) 系統若發現在 collaboration diagram 中有 link 相連的 analysis class 但相互間卻沒有 relationships 的關係相連，則提醒使用者有未定義的關係。

(3) 系統會檢查是否每個在 collaboration diagram 中互相有溝通的 analysis class 其所屬的 package 之間是否也有關係存在。

4.4 設計階段(Design)

Design 階段的核心工作包括 architectural design、design a use case、design a class 與 design a subsystem，各核心工作之輔助功能分述如下：

1. Architectural Design

1.1 定義系統的node與network

ArgoUML 已有提供使用者建立 deployment diagram 之功能，包括每個 node 的名稱以及 node 間的溝通方式(如 internet、intranet 等)。

1.2 定義系統的subsystems以及interface並配置deployment diagram

系統將對每一個 analysis package 詢問使用者該 package 的執行將由哪些 node 來完成，每一個 node 傳遞需配置一個 subsystem。

此外使用者也將定義每一個將利用 middleware 與 system-software 的 subsystems，若有必要，將定義一(組)design class 或一 subsystem 當作呼叫的介面，以保障應用系統的獨立性，而後定義各個 subsystem 之間的依存關係(dependency)，系統將提供各 analysis package 間的關係給使用者參考，有關係的 package 其所對應到的 subsystem 間也必須至少存在一組 dependency，並提醒使用者要定義該 subsystem 的 interface。

1.3 定義architecturally significant design class

這個階段應定義對於系統架構有重要影響的 design class，系統會提供使用者分析階段所定義之 architecturally significant analysis class 供使用者選用。

1.4 Identifying Generic Design Mechanism

此階段需靠使用者自行設計，系統不提供輔助的機制。

2. Design a Use Case

2.1 參考analysis class定義參與use case的design class

系統將列出該 scenario 已定義的 analysis class 當作建立 primary design

class，並要求使用者一一檢查 class 的性質，對於每一 entity class 則詢問是否有必要建立其 container class，或其 attribute type 是否有必要建立 design class 以代表之，若有則建立相關的 design class，並定義其屬同一個 class group。

對於 middleware 或 system-software 的 subsystem，系統也一一詢問是否要新增 design class 來當做呼叫介面，若必要則提供介面供其建立。

對於 boundary class 來說，boundary class 有三種產生輔助性 design class 的原因。系統會針對每個 boundary class 一一詢問使用者是否有必要分割 boundary class 的 component，若有則詢問其數量以新增 design class；詢問其 attribute type 是否有必要建立 design class 以代表之；另詢問其 operation 是否因為複雜度的考量而有必要新增其他的 design class 來處理，若有則建立輔助該 operation 完成的 design class。

而 control class 有兩種產生輔助性 design class 的原因，系統會針對每個 control class 一一詢問是否有必要產生 coordinator class，或其 operation 是否因複雜度的考量而有必要新增其他的 design class 來處理，若有則建立 design class。

2.2 描述use case中各個object的互動

依據指定的 use case scenario，系統將首先列出其 collaboration diagram 及參與其中的 class 之相關 class group，並依 collaboration diagram 之 analysis class 互動順序，由左而右列出其 typical object 供使用者繪製 sequence diagram，使用者對每個 object 檢查是否需插入 class group 內成員之 typical object；當使用者選擇其中的 design class 時，列出該 class group 的 analysis class 在上個階段呼叫其他 analysis class 的 responsibility 清單。使用者必須選擇其中的 responsibility，然後建立實現該 responsibility 的(一連串)messages。當加入一個 message 於 sequence diagram 時，被呼叫的 design class 需定義其對應 operation 名稱。

當結束此 sequence diagram 繪製時，將執行下列一致性檢查：

(1) 檢查 sequence diagram 中的 message 是否都是 operation。

(2) 檢查 analysis 階段的 collaboration diagram 中的 responsibilities 是否都被實現了。

(3) 檢查與每一個 responsibility 對應的一連串 message 中，是否(只)有一個跨 class group 的 message(即 *primary operation*)。

(4) 檢查與每一個 responsibility 對應的一連串 message 裡是否第一個參與的 design class 與最後一個被呼叫的 design class 分別是在 analysis 階段 caller 與 callee 相對應的 class group 中。

(5) 檢查與每一個 responsibility 對應的一連串 message 裡所有參與的 design class 都必須是屬於 caller 或 callee 的 class group，若不是則必須是 depend on middleware 或 system-software 的 design class，不滿足此條件則告知使用者有不一致的情形發生。

2.3 定義use case中參與的subsystems與interfaces

與 2.4 描述use case中各個subsystem的互動

由使用者自行繪製 subsystem 的 sequence diagram，系統不提供支援。

2.5 註記implementation階段的requirements

提供文字表格讓使用者輸入 implementation 階段的 nonfunctional requirement。

3. Design a Class

3.1 Outlining the Design Class

列出已定義之 design class 供使用者選擇進行定義工作。

3.2 定義design class的operation

列出選定 class 已定義之 operation 名稱供使用者定義其參數、型態等。

3.3 定義design class的attributes

若該 design class 為 primary design class 則列出其所在 analysis class 的

attribute，供使用者參考定義 attribute，並提供 attribute 定義機制。同時並檢查是否 analysis class 的 attribute 都被 primary design class 所實現了。

3.4 定義各design class間的associations與aggregations

列出指定之 design class 所屬 class group 之間關係，及在所屬 analysis class 之間的關係，包含：

(1) 列出該 design class 在 sequence diagram 中與之有 link 相連的 design class，供使用者建立兩者間的關係。

(2) 列出實現該 design class 的 attribute type 的 design class，供使用者建立兩者間的關係。

(3) 列出 primary design class 與輔助它的 design class，供使用者建立其關係。

當結束時將自動檢查有上述 relationship 的 design class 是否已被使用者定義了 relationships。

3.5 定義design class間的generalizations

提供上個階段 analysis class 間的 generalization 關係，analysis class 對應本階段的 class group 內的 design class 供使用者參考定義。

3.6 描述operations的實現方法

提供每個 operation 描述其 pseudo code 機制。

3.7 透過statechart diagram的建立來描述design object的state

提供使用者繪製指定 class 之 statechart diagram 的機制，diagram 上的 transition 必須是該 design class 已定義的 operation。

3.8 處理special requirement

列出該 design class 所參與的 use-case realization 裡的所有 nonfunctional requirement 給使用者參考。同時也列出該 design class 所屬 analysis class 的 special requirement 供使用者進行定義之參考。

4. Design a Subsystem

4.1 維護subsystems間的dependencies

在 architectural design 所有的 subsystem 都已被定義出來了，同時使用者也已經建立了其間的 dependency，因此系統應自動提供使用者檢驗 subsystem 內的 design class 是否與該 subsystem 的 dependency 關係一致。

4.2 維護subsystems所提供的interface

有被 design class 參考到的 subsystem 系統將檢查是否已提供 interface。系統也會檢查每個 interface 是否都有對應的提供者。

4.3 維護subsystem中的design class

同時系統也會提供每個 subsystem 內 design class 參考到其他 subsystem (即 external reference)的數量給使用者參考，可以讓使用者選擇是否要重新分配 subsystem 的 design class。

在完成上述四個 design 階段的核心工作後，系統會進行整體檢查以保證能無誤的進入下一個階段，本階段的整體一致性檢查包括：

(1) 檢查是否每個 package 都已經產生對應的 subsystem 了，而如果有未完成的 package 則必須提醒使用者。

(2) 檢查是否有被其他 subsystem 或 design class 使用到的 subsystem 其 interface 是否都被定義了，若有未定義的 interface 也必須提醒使用者

(3) 對於定義 design class，系統會檢查下列事項是否已經完成了：

- 檢查是否每個 analysis class 都有建立相對應的 primary design class。
- 檢查是否每個 design class 內的 attribute type 都已經被實現了。
- 檢查是否每個 design class 都有其出現的原因，若不是 primary design class 或是為了輔助 primary design class 所產生的 design class，就必須是 depend on middleware 或是 system-software subsystem 的 design class，如果都不是則代表不一致。

第5章、跨階段產物一致性檢查系統之架構

本章說明「跨階段產物一致性檢查系統」的設計原理及應用實例，將第三章所提的檢查方法架在 open source 的「ArgoUML」系統上並整合為一系統。「ArgoUML」project 是一開放原始碼的軟體專案，由各地的有興趣的軟體工程師共同開發而成，目前仍持續開發中，現有的功能主要提供繪製 UML 圖形為主，其主要功能列舉如下：

1. 製圖功能：可繪製 Class diagram、Statechart diagram、Use case diagram、Activity diagram、Collaboration diagram、Deployment diagram 共六種圖形，而有些 UML 語言所訂定的圖形如 Sequence diagram 或 Component diagram 目前並未實作於 ArgoUML 中。除了提供製圖功能外，也允許讓使用者將圖形匯出成 XML 或是其他圖檔的格式。
2. 輔助設計的提醒功能：在使用者建立圖形時，給予使用者適當的提醒或設計指南，避免使用者因不熟悉 UML 圖形而誤用。如加入一個名為「Invoice」的 class 時，ArgoUML 會以 checklist 的介面來提醒「Does the name ‘Invoice’ clearly describe the class？」，避免使用者產生不適宜的 class 名稱。
3. Todo list 功能：由使用者自行建立尚未完成的工作，並依照優先順序來排列，方便使用者整理與安排工作表。

從以上的功能可知，ArgoUML 開發平台只是一個單純的 UML 製圖輔助工具，但並沒有將 USDP 的工作流程與各階段的順序概念融入 ArgoUML 的設計中。ArgoUML 是以 Java 語言所設計而成，Java 語言具有的設計彈性使得其他開發人員可以透過新增模組的方式為 ArgoUML 設計新的功能，所以我們選擇 ArgoUML 為基礎，在其上實作跨階段產物間一致性檢查的系統。

本章共分為三個小節，5.1 節簡要說明 ArgoUML 的系統架構，5.2 節說明本研究所提之一致性檢查方法在 ArgoUML 上之設計架構，5.3 節則是部份功能的系統介面解說。

5.1 ArgoUML 之系統架構

5.1.1 General architecture

ArgoUML 的主要架構由三個 package 所組成，分別是控制 UML 圖形內部架構的 NSUML (Novosoft UML metamodel library)、控制 UML 圖形繪製表現的 GEF(Graph Editing Framework)以及處理整合運用的 ArgoUML。其中 NSUML 包含所有表現及處理 UML model 的 classes，GEF 負責將 UML diagram 的模型視覺化並表現在 UI 上，而 ArgoUML 則是將這兩部分的功能連結，並加入應用邏輯。

ArgoUML 採用 Model-View-Controller(MVC)的設計架構，MVC 架構將 UI 與系統內部處理的資料分開，同一個 model 會有一至多個與之相連的 views，當 model 內容需做更改，將透過 controller 呼叫 model 內的 method 來改變其資料，同時 model 也通知所有 views 改變其狀態內容。

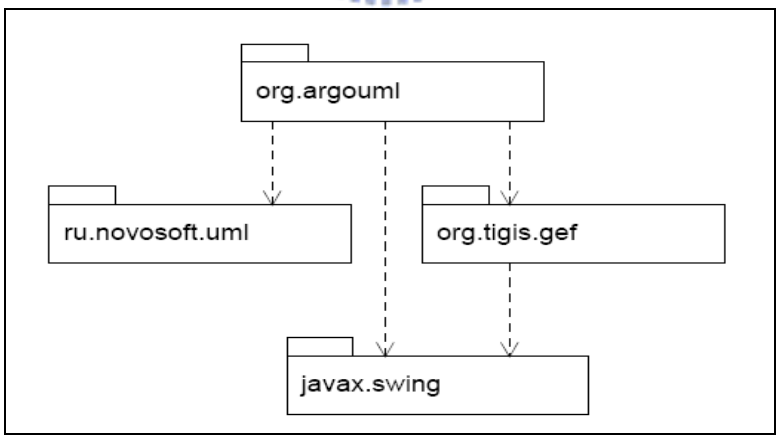


圖 5-1 ArgoUML architecture[14]

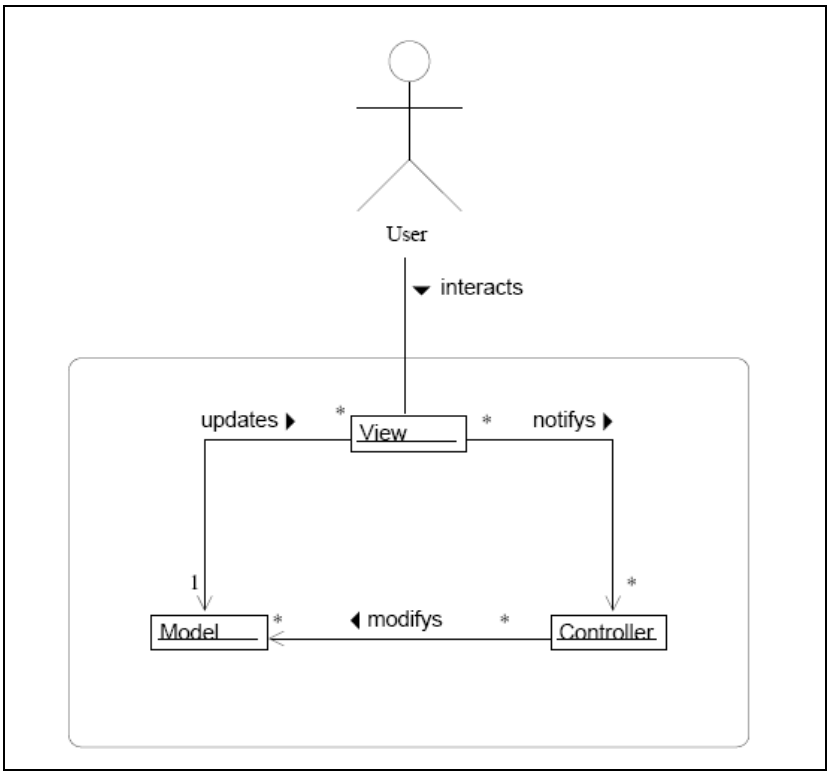


圖 5-2 Model-View-Controller Pattern[14]

圖 5-3 為 ArgoUML 的 framework，四層的架構(由下而上為 Layer 0 至 Layer 3)讓開發人員可以在上兩層(Application-specific layer 與 Application-generic layer)進行開發，由於 ArgoUML 是以 JAVA 做為開發系統的程式語言，所以最底層為 Java Virtual Machine(JVM)，而介於上兩層與最底層間的 Layer 1 則是 ArgoUML 所使用的函式庫。而 ArgoUML 將每一層的任務由數個 subsystem(或稱 component)來實現，5.1.2 節將說明這些 subsystem 的功能。

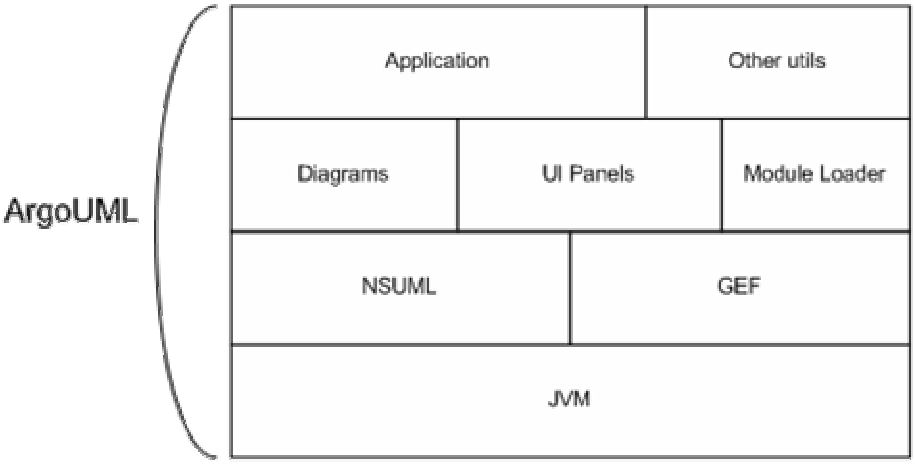


圖 5-3 ArgoUML Framework

5.1.2 Packages

ArgoUML 將系統功能分為許多個 subsystems(又稱為一個 argouml 內的 component)，各有其負責的任務，這些 subsystems 以層狀(layered)的架構所集合成，圖 5-4 簡介說明每個 subsystem 內的架構，subsystem 具有 interface 以及擴充的功能，可以透過 plugin 以彌補其他不足。

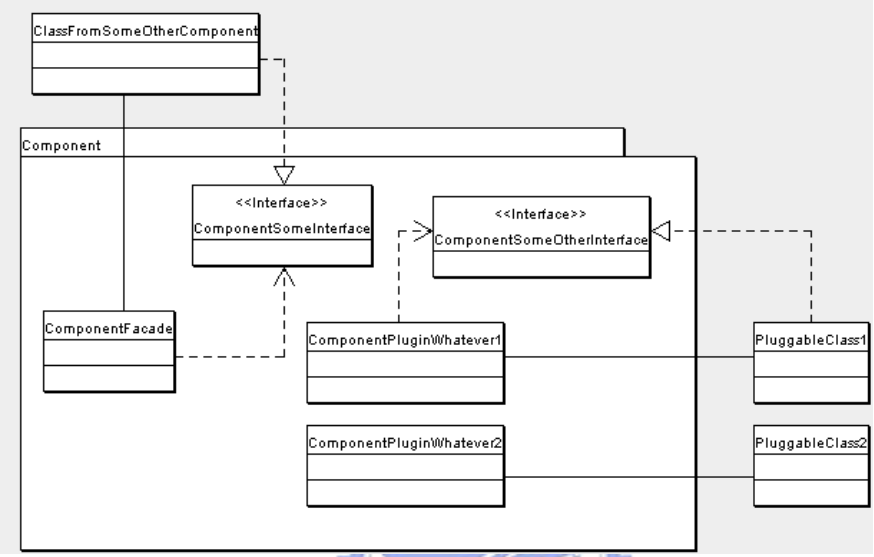


圖 5-4 subsystem's architecture[15]

主要的 subsystems 架構分為四個 layer：

- Layer 0

  由負責紀錄的 Logging subsystem、負責地區性字串轉換的 Internationalization subsystem、及負責提供 java runtime 支援的 JRE(with utils) subsystem，Layer 0 的這三個 subsystem 包含了其他三層所會使用到的一般性功能，自 Layer 1 到 Layer 3 都會 depend on layer 0。

圖 5-5 Layer 0 packages[15]

- Layer 1

  Argouml subsystem 架構中的最底層，Layer 1 除了 depend on Layer 0 之外並沒有其他參考的 subsystem。其中包含了 Model、Todo Items、GUI Framework、Help System 四個 subsystem，其中 Model subsystem 提供 UML 各種圖形的資料結構，使用上一節提到的 NSUML 來實作 UML model，並提供較上層的 subsystem 使用。

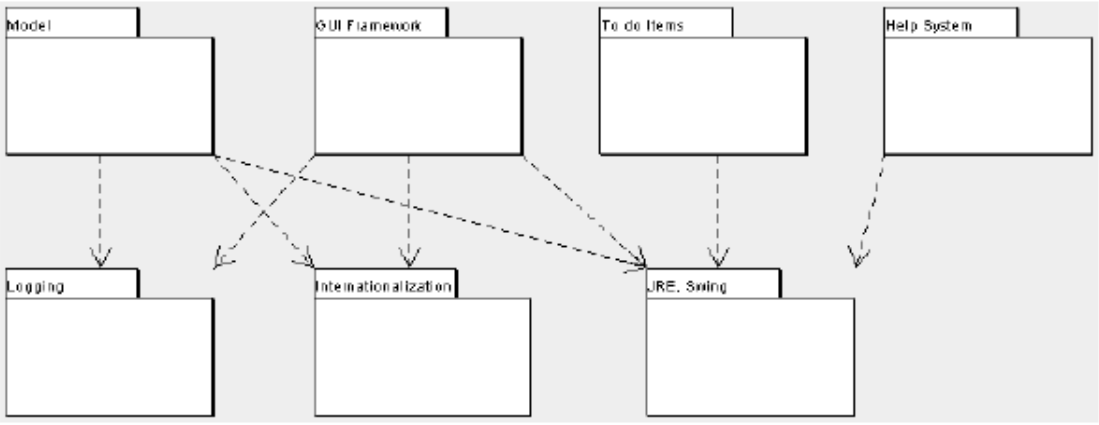


圖 5-6 Dependencies between Layer 1 and Layer 0[15]

- Layer 2

  包含以下六個 subsystems：

  1. Diagrams：負責將 model 內的 diagram 以圖形化的方式表現出來，並包含 diagram 上的工具列。
  2. Property Panels：提供顯示 model 內 diagram 及 object 的 properties。
  3. Explorer：提供樹狀的顯示結構以表達 model elements、diagrams 及其他 objects 間的關係。
  4. Code Generation：提供 code generation 輔助功能的 subsystem。
  5. Reverse Engineering：提供 reverse engineering 輔助功能的 subsystem。
  6. Module Loader：負責讀取 Layer 3 上具有延伸功能的 modules 或 plugins。

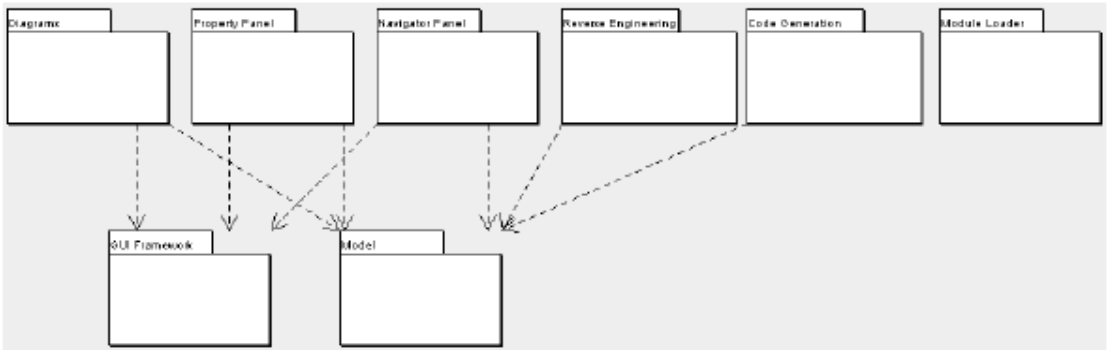


圖 5-7 Dependencies between Layer 2 and Layer 1[15]

- Layer 3

  ArgoUML subsystem 架構的最上層，包含以下五個 subsystems：

  1. Java-code generation and reverse engineering：處理 java code 的 code generation 與 reverse engineering。
  2. Other languages-code generation and reverse engineering：處理其他程式語言的 code generation 與 reverse engineering。
  3. Critics and checklists：提供使用者對於繪製 diagram 上的輔助(如 check list 提醒該注意的事項)。
  4. OCL：提供處理與編輯 OCL(Object Constraint Language)的字串。
  5. Application：進入 ArgoUML 應用程式的起始進入點(entry point)。

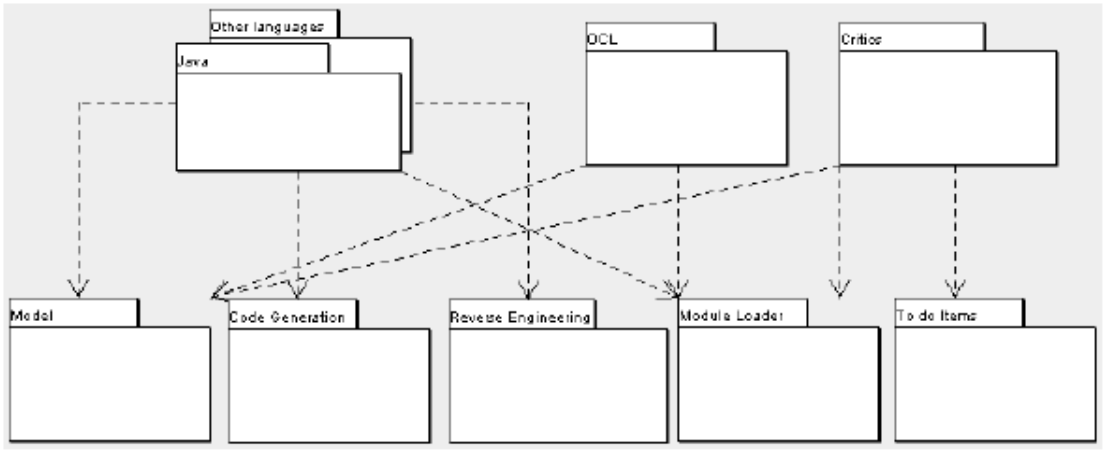


圖 5-8 Dependencies between Layer 3 and Layer 2[15]

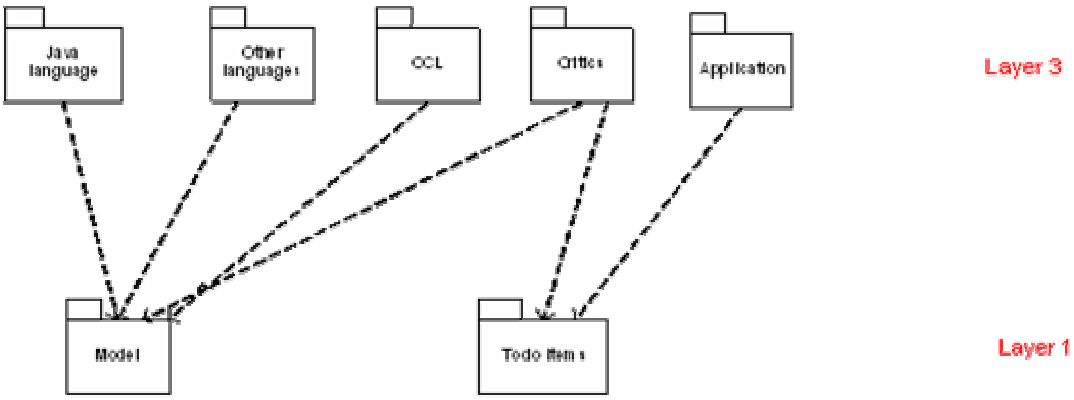


圖 5-9 Dependencies between Layer 3 and Layer 1

5.1.3 ArgoUML 介面設計

圖 5-10 為 ArgoUML 開發工具在執行時的主要畫面，畫面一共分為下列四個部分：

1. Explorer：在畫面的左上方，顯示目前的 project(或 model)的開發狀態與產物，是一階層性的樹狀架構。
2. Editor：在畫面的右上方，由 Diagram 的編輯區域與 toolbar 所組成，
3. ToDos：在畫面的左下方，依優先度的順序來排列使用者自行建立的工作清單，並以階層的方式表示。
4. Details Panel：在畫面的右下方，為此開發工具的核心功能之一，當使用者選擇 diagram 中的物件或是 todo list 內的工作項目時，這個面板將詳細列出該物件的所有 property 資料，使用者透過該面板上的欄位來更改物件的內容。

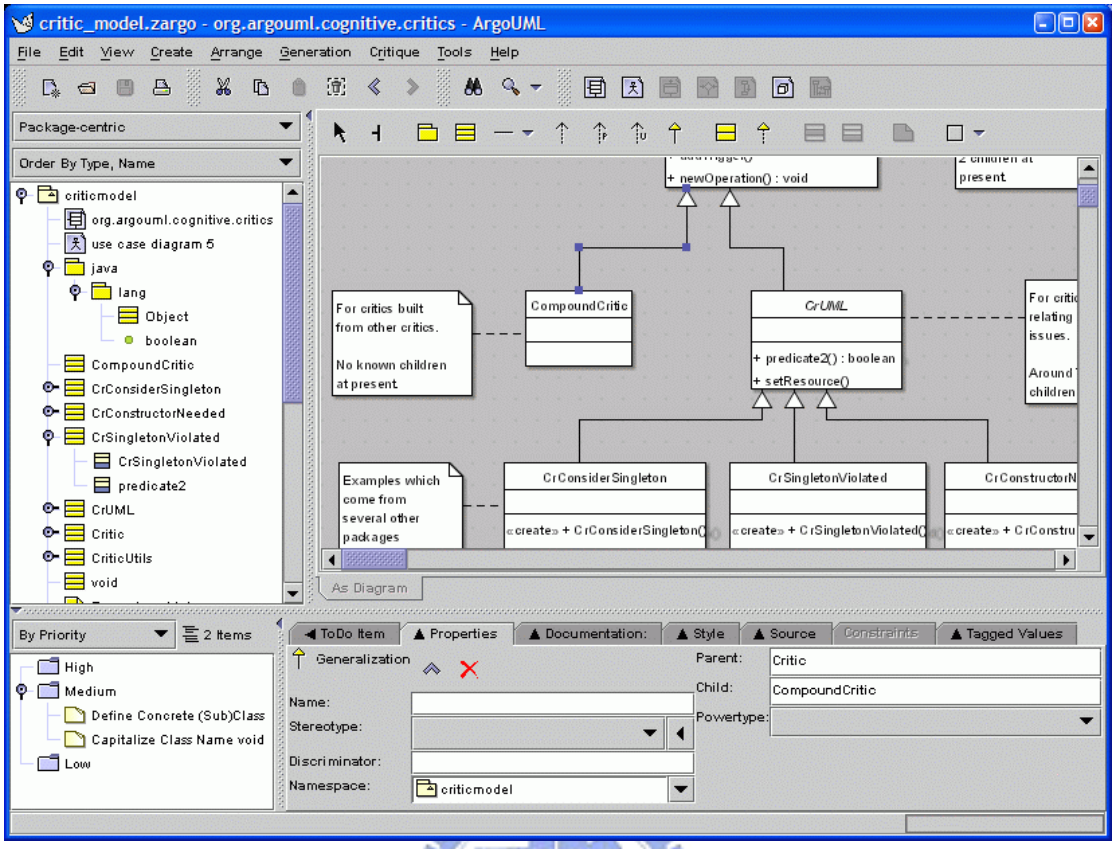


圖 5-10 ArgoUML main window[16]

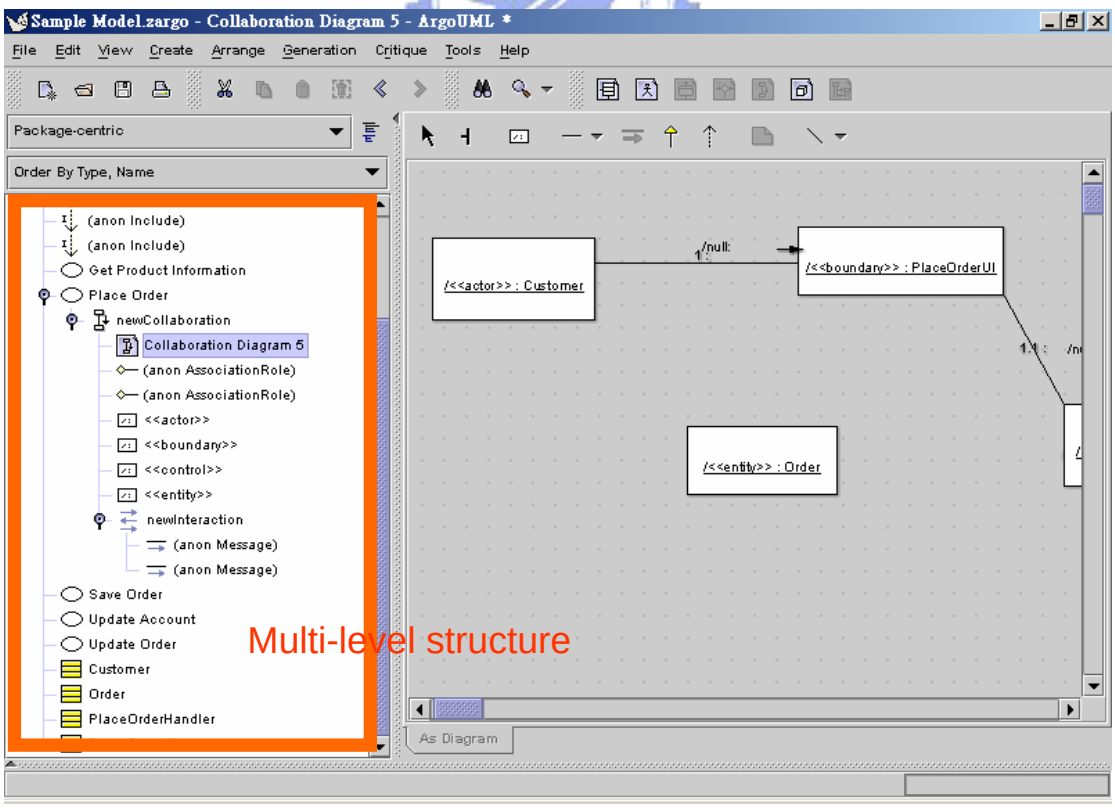


圖 5-11 ArgoUML-multilevel structure

5.2 一致性檢查方法在 ArgoUML 上之設計架構

本研究將提出的產物一致性檢查方法(稱為 ArgoUSDP module)建置在 ArgoUML 上，使用 plugin 的方式來延伸 ArgoUML 的功能，使其支援軟體開發流程上的產物檢查方法。藉由實作 ArogUML 所提供的延伸功能介面" org.argouml.application.api.Pluggable"讓 Layer 2 的 Module Loader subsystem 在程式啟動時將 ArgoUSDP 載入執行環境中。

ArgoUSDP 的起始模組是位在 ArgoUML 架構中的 Layer 3 中，並有少部分功能更動到 ArgoUML 的原始碼，圖 5-12 是 ArgoUSDP 與 ArgoUML 主程式的 framework 關係圖。

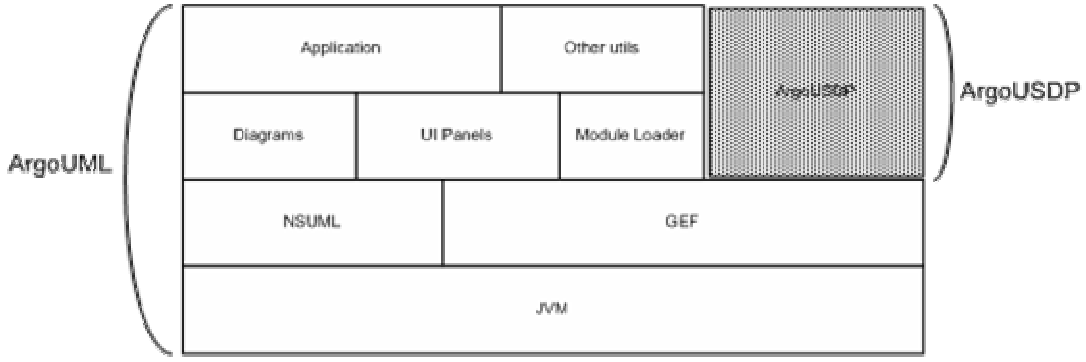


圖 5-12 ArgoUSDP 與 ArgoUML 主程式之 framework

圖 5-13 為 ArgoUSDP 模組的 subsystem 架構，共分為以下四個重要的 subsystems：

1. Main module：ArgoUSDP 模組的進入點，實作 ArgoUML 所提供的 pluggable 介面，並由 Module Loader 在啟動時載入。
2. Scenario Editor：新加入的 scenario 編輯面板，與 ArgoUML 中原有的 use case diagram 整合，讓使用者可以在繪製 use case diagram 時直接建立 scenario 表格。
3. Artifact Checker：負責檢查各產物之間的相關性，其中包含了一個 Rule Set，Rule Set 內含有 consistency checking 的各項 rules，在進行檢查時

會檢查是否每條 rule 都正確成立，上一章所敘述的相關性檢查都是由本 subsystem 來負責實現。

4. Process Assistant：負責提供使用者進行流程上的輔助，例如提醒使用者是否在該階段中有仍未建立的 artifact。

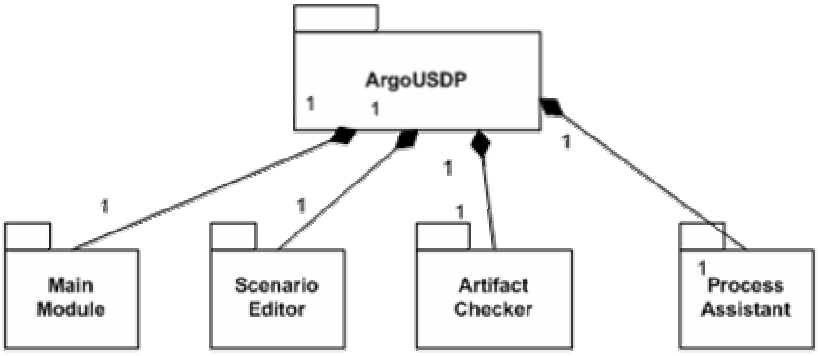


圖 5-13 ArgoUSDP subsystems

圖 5-14 說明 ArgoUSDP 內各 subsystem 在 ArgoUML framework 中所在的 layer，其中除了 Main module 在 Layer 3 外，其餘都位於 Layer 2 中。而圖 5-15 則說明 ArgoUSDP 與其他 subsystem 間的依存關係。

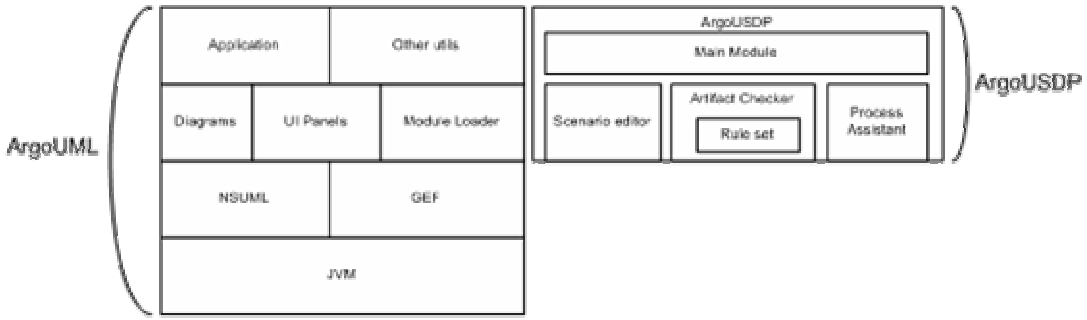


圖 5-14 ArgoUSDP framework

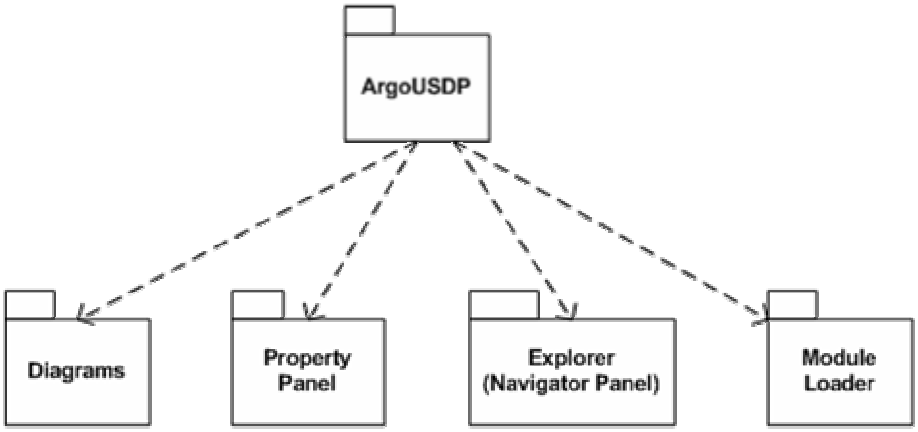


圖 5-15 ArgoUSDP 與其他 subsystem 間之關係

5.3 系統介面

ArgoUSDP 的輔助功能分為 Requirement Capturing 階段、Requirement Capturing 至 Analyis、Analyis 至 Design 部分，第一部分的核心產物為 use case diagram 與 use case scenario 的 specification，第二部分的核心產物為 collaboration diagram，第三部分的核心產物則是 sequence diagram。由於 ArgoUML 為持續在開發中的計畫，有部分的產物目前並沒有實作於 ArgoUML 中而僅被規劃為未來之開發重點，其中缺少 sequence diagram 的支援使得本研究的檢查方法無法將 Analysis 至 Design 階段的部分建置於 ArgoUML 中，而其他部分的系統介面節錄如下。

5.3.1 Requirement Capturing 階段

使用者必須先定義系統的 use case model 並以 use case diagram 表示之，圖 5-16 為 Pay Invoice use case 的 use case diagram。

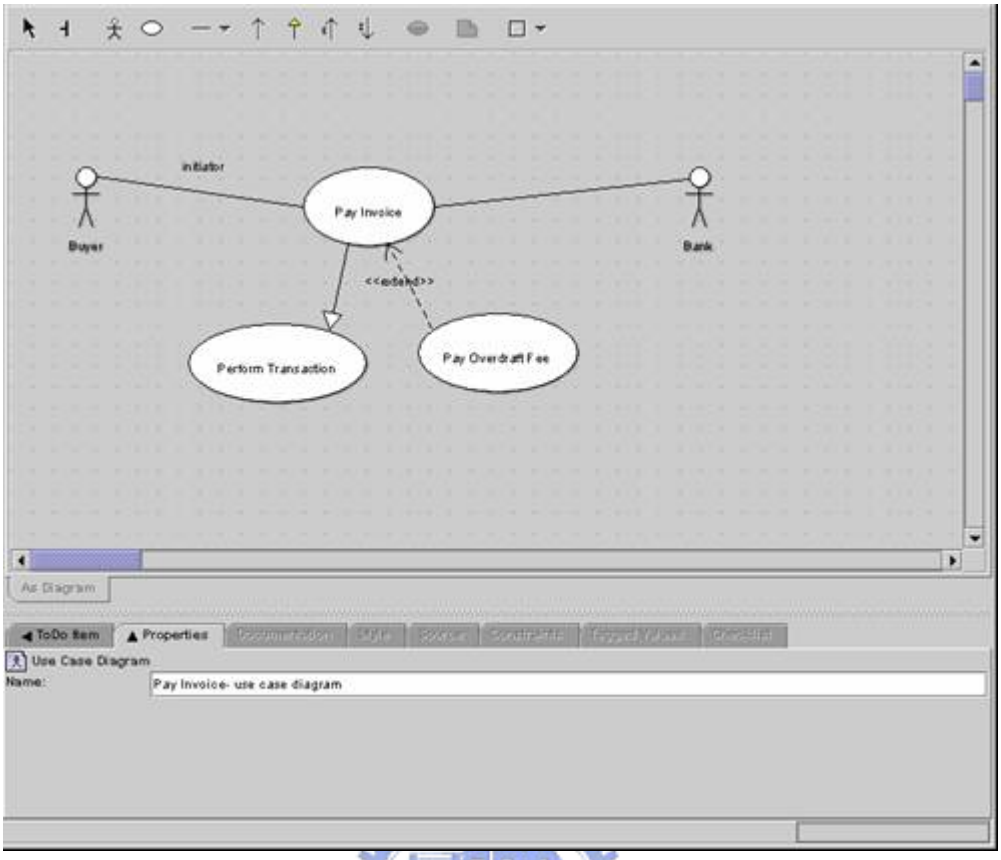


圖 5-16 使用者建立 Pay Invoice 的 use case diagram

有了 use case diagram 之後使用者必須為每個 use case 的 scenario 建立 specification 表格，使用者點選 property panel 上的”View Scenario”標籤來進入 scenario 編輯模式(如圖 5-17)。

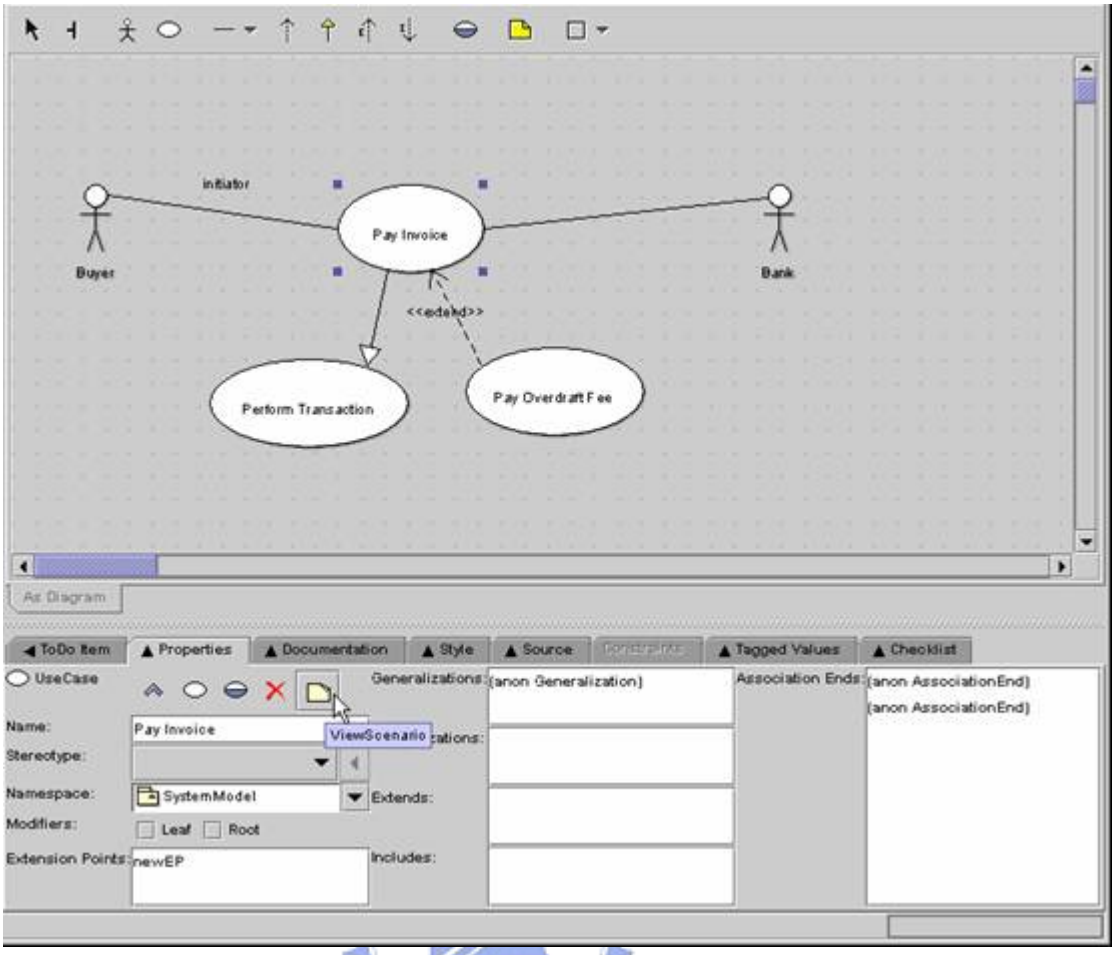


圖 5-17 選擇 View Scenario

圖 5-18 是 scenario 的編輯視窗，視窗共分為三個部分，上方為按鈕列，使用者可選擇新增 actor 欄位或是將此 scenario 存為 html 格式的檔案；中間部分為表格編輯視窗，使用者可點選表格中的欄位以加入 responsibility；右邊是關鍵名詞的列表，會列出該 scenario 中被使用者標示為關鍵名詞的字。同時使用者可以點選欄位標頭以變更 actor 的名稱(如圖 5-19)。

使用者點選表格中的任一欄位後會開啟更詳細的 scenario 動作編輯視窗(如圖 5-20)，在該視窗中有較大的文字編輯空間，可以編輯較複雜格式的動作如 branch 或 loop。

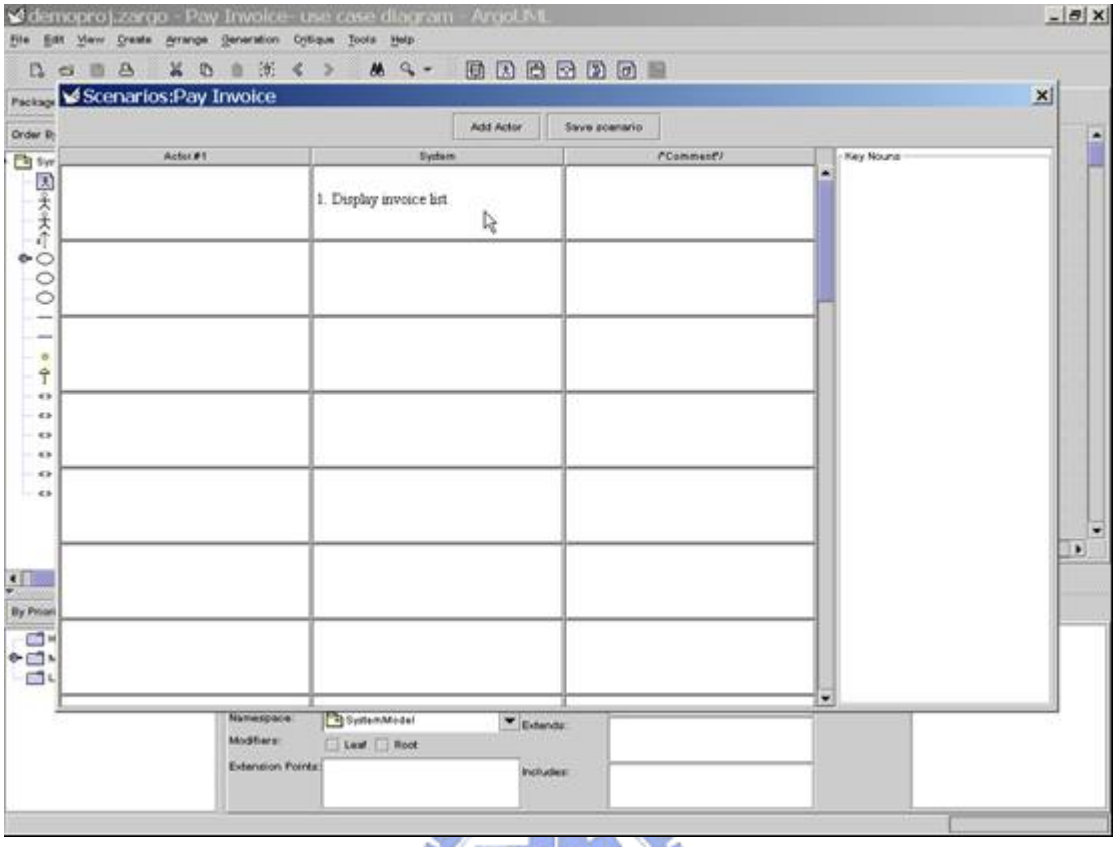


圖 5-18 Scenario 編輯視窗

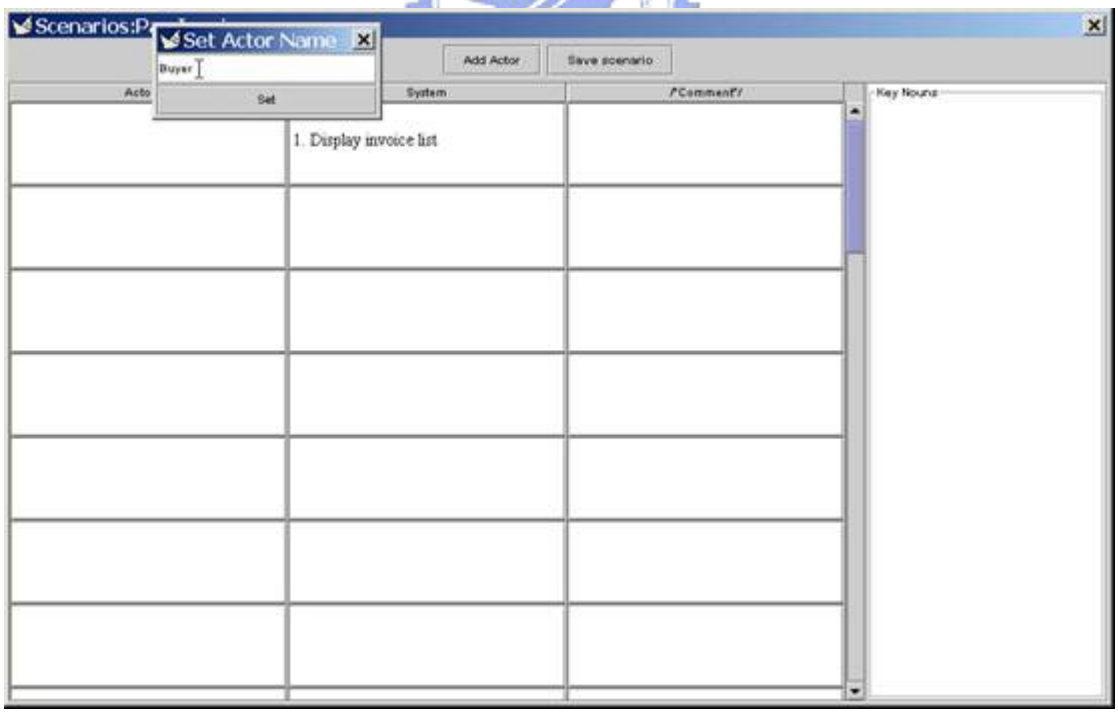


圖 5-19 變更 actor 名稱

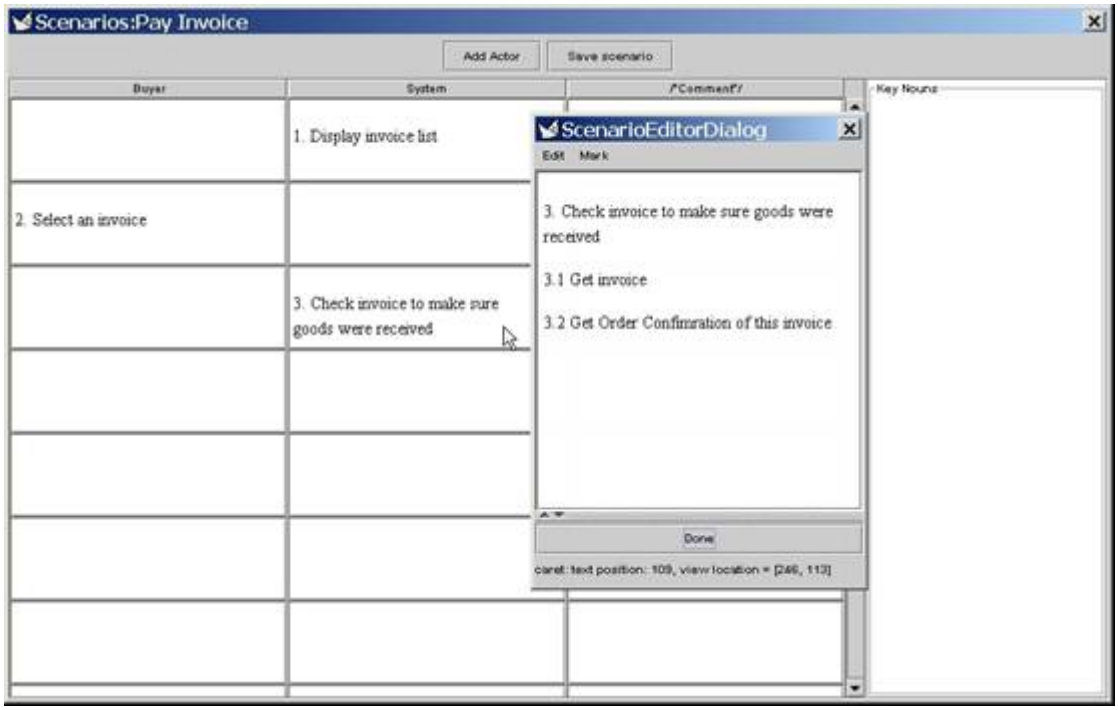


圖 5-20 scenario 動作編輯視窗

圖 5-21 的動作編輯視窗中使用者選擇了其中的一段文字(invoice)，並點選選單上的 mark 以標記為關鍵名詞，同時該段文字也會以粗體字型表示。

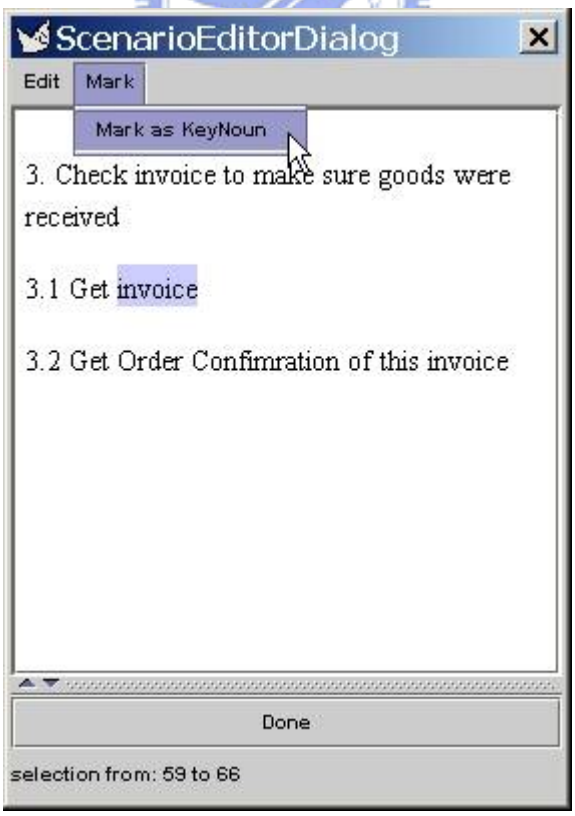


圖 5-21 標記關鍵名詞

圖 5-22 為 Pay Invoice use case 的 basic path scenario，右方的關鍵名詞列表以將使用者所標示的重要名詞列出。

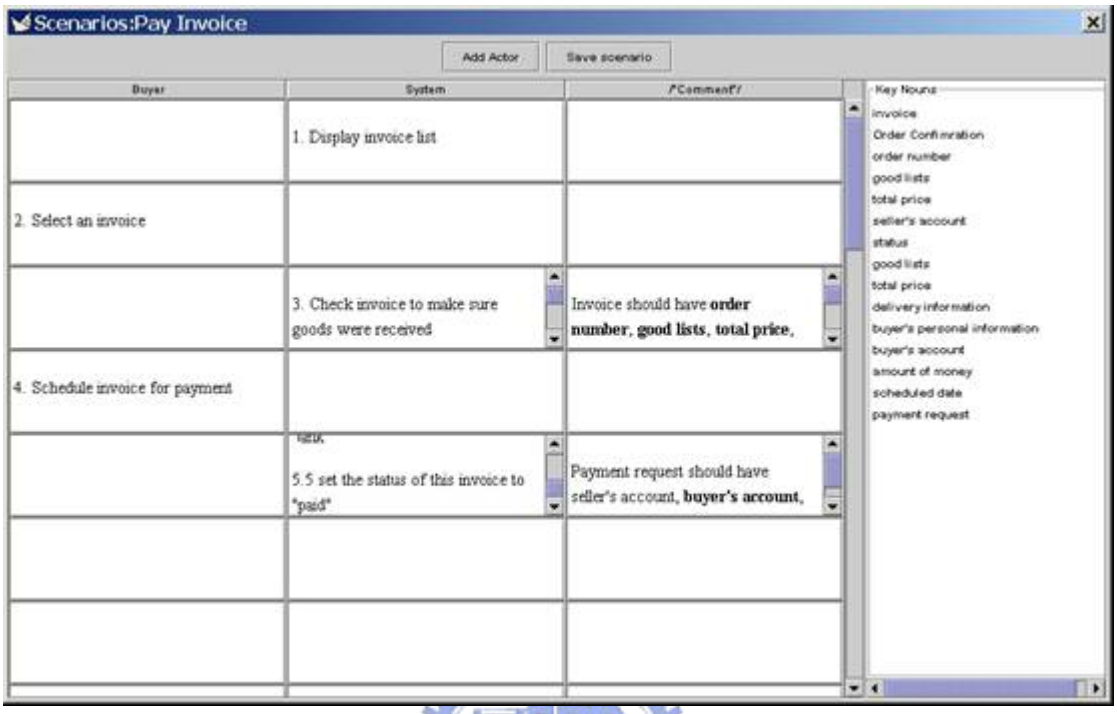


圖 5-22 Pay invoice use case 的 basic path scenario

5.3.2 Requirement Capturing to Analysis

在 ArgoUSDP 的設計中，每個 use case 底下具有 Analysis Phase 與 Design Phase 兩個分支，在 Explorer 列表也可看見這兩個子分支(如圖 5-23)。

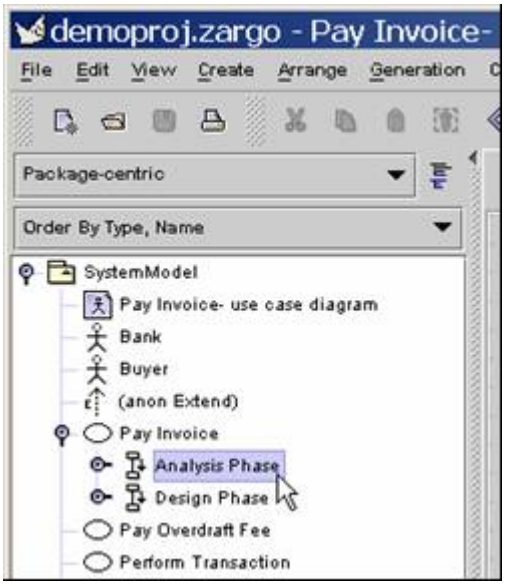


圖 5-23 Analysis Phase 分支

在這個流程中會於 analysis phase 的分支下建立該 use case 的 collaboration

diagram，本小節擷取部分輔助使用者產生 collaboration diagram 的功能如下。

首先必須先定義參與該 scenario 的 analysis class(如圖 5-24)，同時在圖 5-25 使用者透過 analysis class 的 property panel 選擇該 analysis class 的類型，本階段中使用者僅能選擇 entity、control、與 boundary 三種 stereotype。

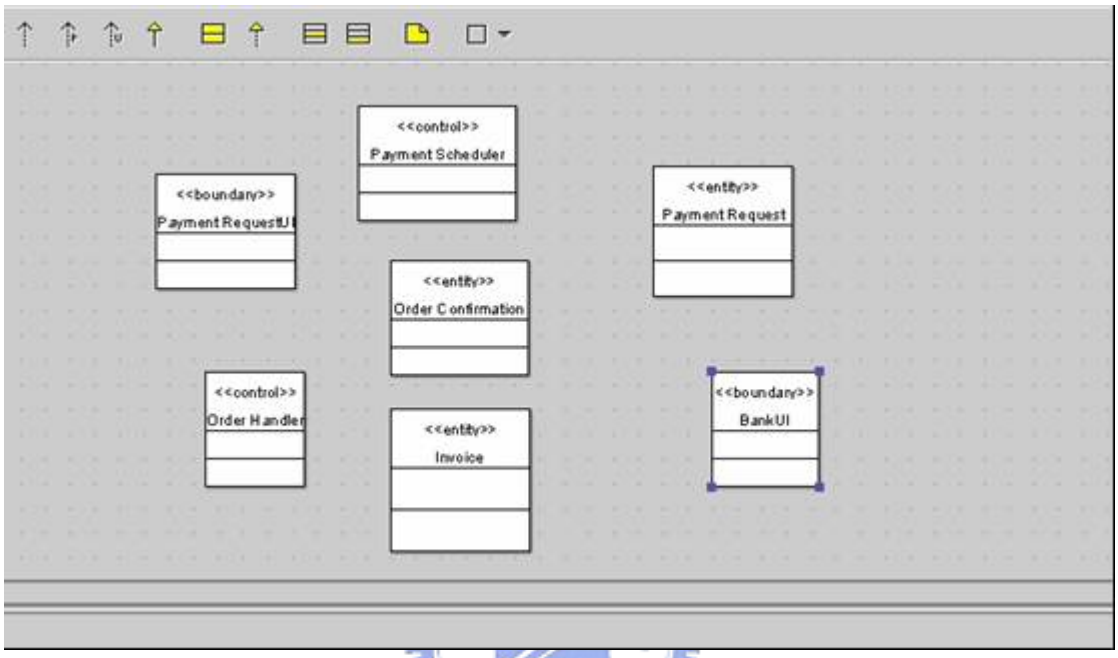


圖 5-24 定義 analysis class

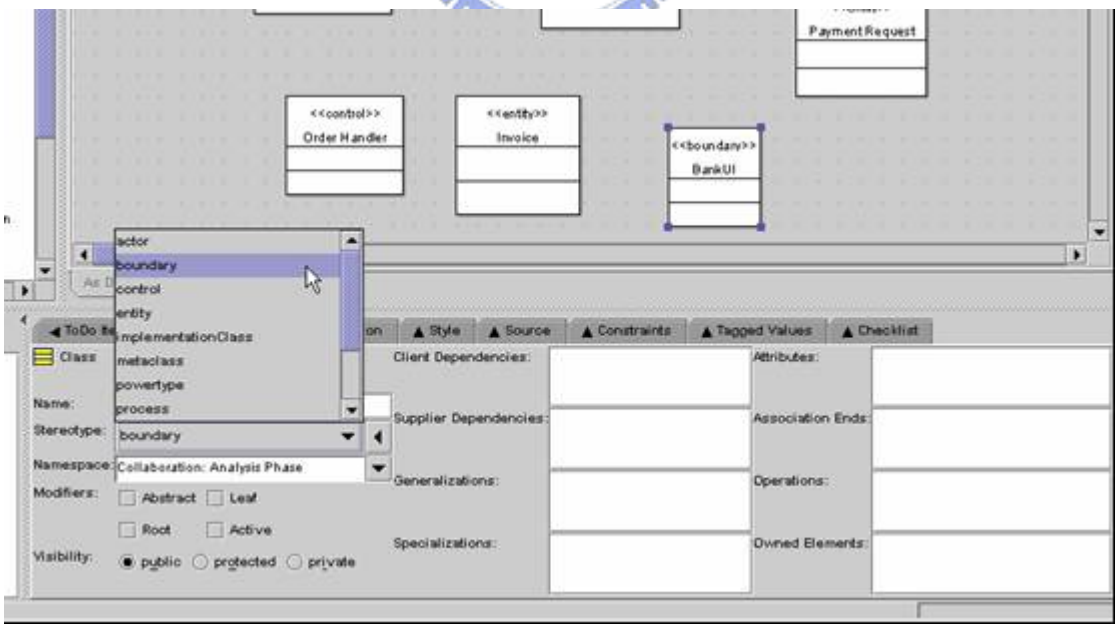


圖 5-25 選擇 analysis class 的類型

定義 analysis class 之後使用者便可在 analysis phase 下建立 collaboration diagram(如圖 5-26)，collaboration diagram 中出現的 analysis class 必須是先前已

定義過的 analysis class，否則無法加入。

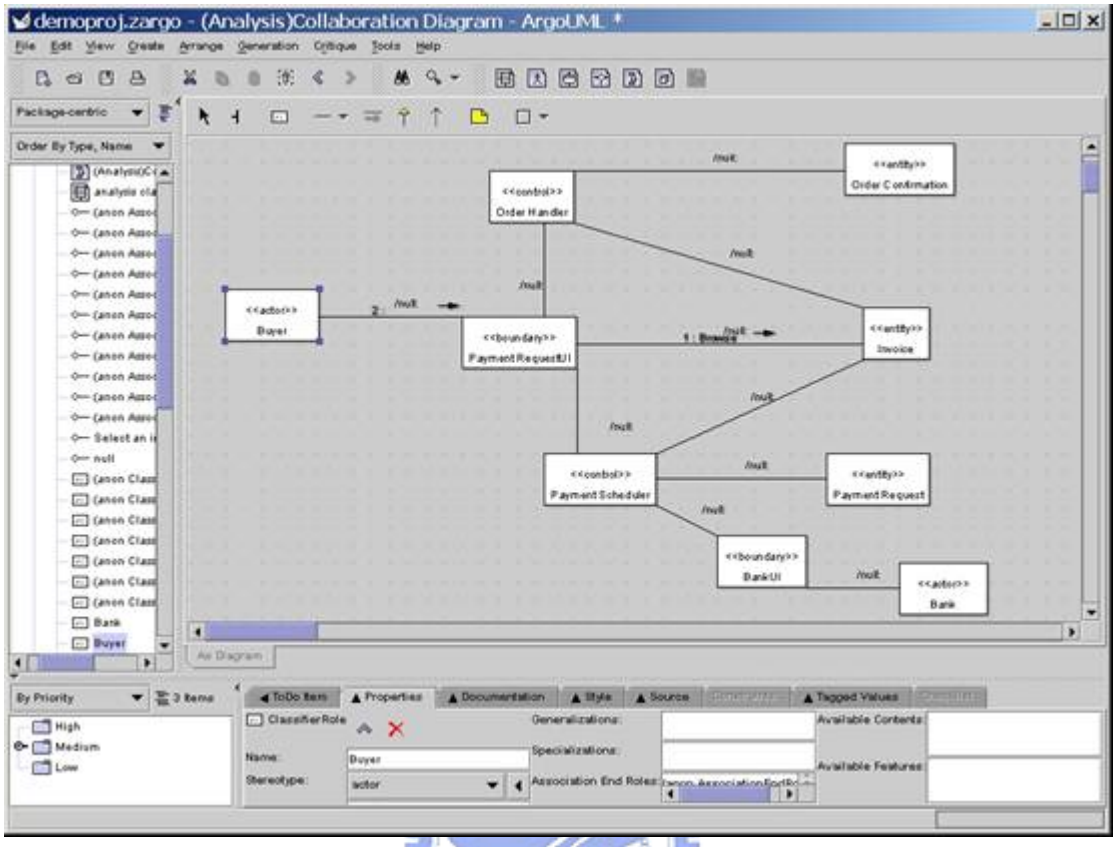


圖 5-26 建立 collaboration diagram

在建立 collaboration diagram 上的 message 時，系統將提供先前建立的 scenario 裡的動作來讓使用者參考，而考慮到 message 上的名稱不一定會與 scenario 上的動作名稱相同，所以是以 checklist 方式提供使用者協助。當使用者輸入 message 名稱時，同時選擇與其對應的動作(如圖 5-27)。而這個新加入的 message 也會被列在被呼叫的 analysis class 裡的 responsibility list 中(如圖 5-28)。其他檢查的功能如關鍵名詞與 entity class 名稱的對應，以及關鍵名詞與 attribute 的對應也是以 checklist 機制來設計，故在此不詳細敘述介面內容。

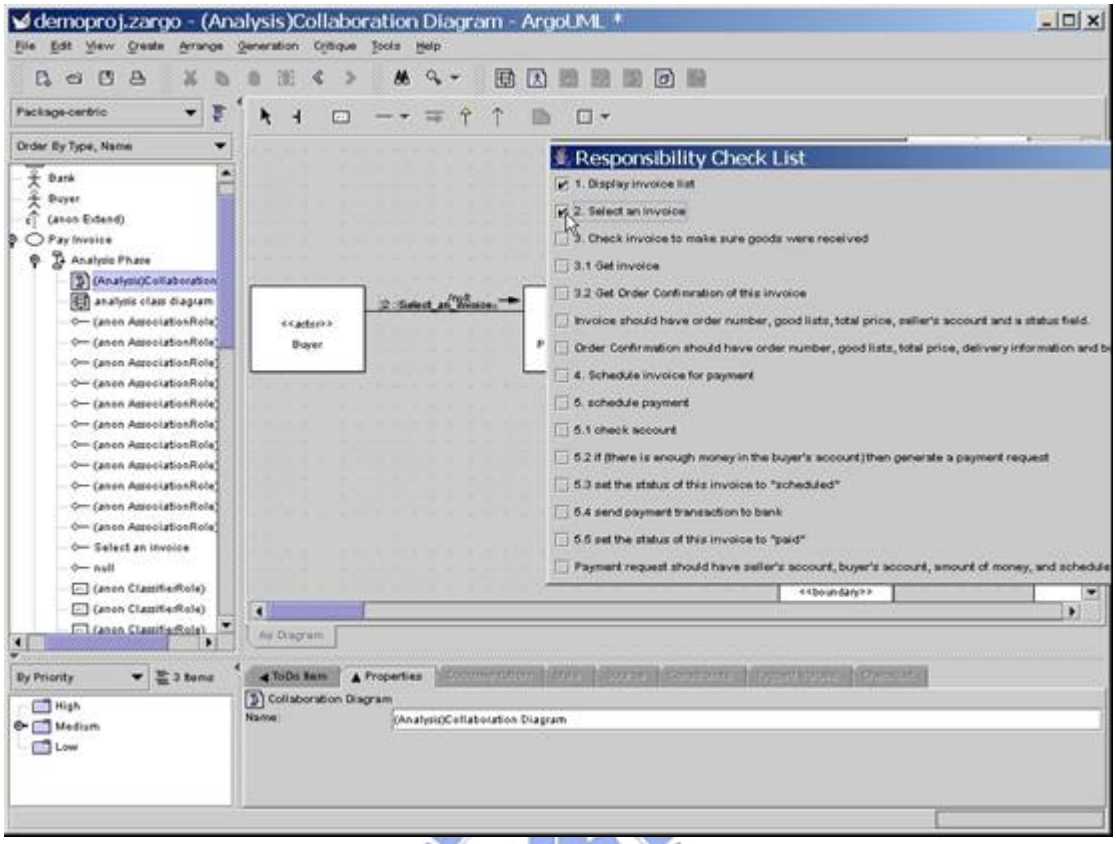


圖 5-27 Responsibility checklist

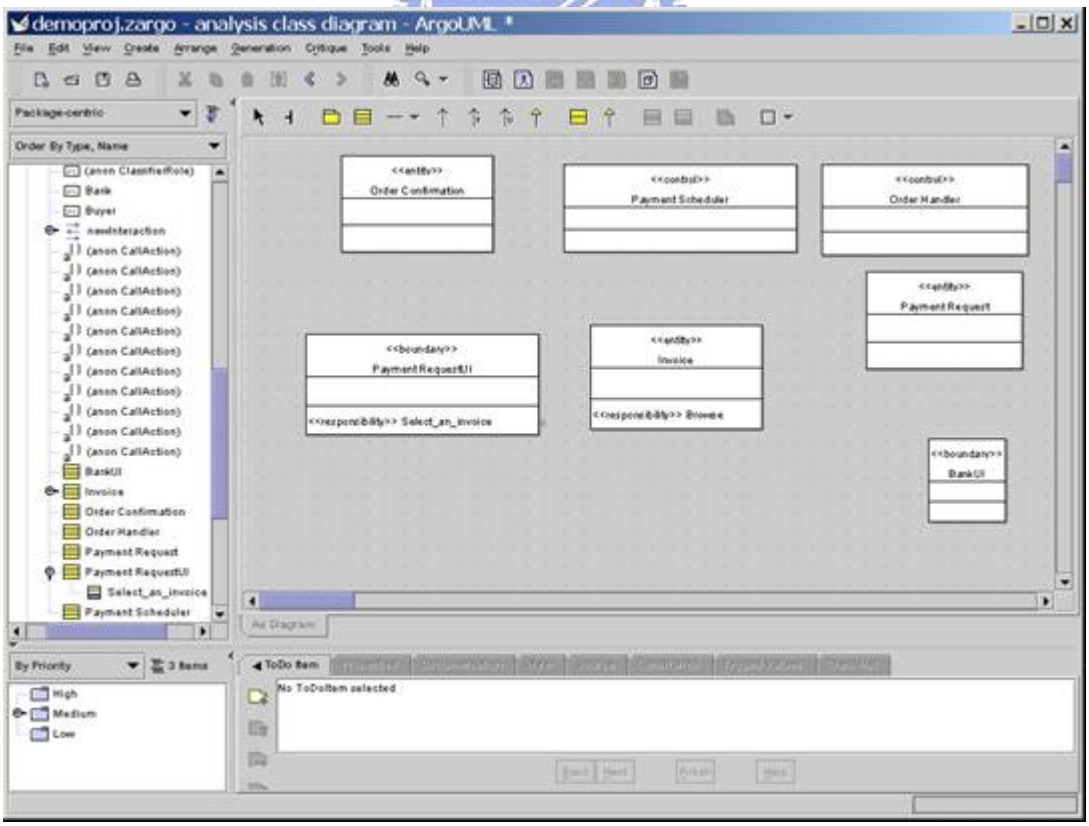


圖 5-28 Analysis class 的 responsibility

第6章、結論

UML 圖形是軟體開發流程中各階段所產生的產物，隨著系統日益複雜及軟體規模漸趨龐大，跨階段產物間可能因人為疏失或是團隊合作的問題而導致發生不一致的情事，若無法即時找出將導致後續階段的錯誤，但僅依靠人力來發現錯誤是件不容易的事，有必要藉助工具適時的輔助開發人員檢查流程中的產物是否具備一致性。現有的 UML 輔助開發工具大都偏重軟體發展流程的步驟導引及提供製圖功能為主，因此僅能輔助使用者在開發流程中建置各式 diagrams，在跨 diagram 間內容檢查並不完整，無法檢測跨階段產物間的不一致，實有必要提升其功能，因此本研究的目的是在探討跨階段軟體產物 UML 圖形之檢測機制，以降低其發生不一致現象的機率。

本研究首先分析開發流程各階段產物的性質及相關性，追蹤產物元素在階段更迭時的演進變化，以建立各產物及產物內容(Properties)的追溯性，進而提出下列機制：

1. use case scenario 的表格化表示方式

   提出以表格的方法來表示 use case scenario 的 flow of event，使 flow of event 易於檢視與理解，有助於分析階段 collaboration diagram 之建立，且更方便進行 use case diagrams 之一致性檢查。

2. 藉由 analysis classes 與 design classes 之關聯性及 collaboration diagram 與 sequence diagram 之關聯性，提出 design class 及 sequence diagram 之輔助發展導引機制，以提升 design 階段之完整性，及易於進行兩階段產物的一致性檢查。

3. 將所提出的檢查方法及輔助開發機制加入 ArgoUML 開發工具中，完成軟體開發系統雛型，以驗證其實用性。

本研究的成果對軟體發展而言具有下列效益：

1. 配合軟體開發流程的階段，適時輔助使用者建立產物間的關係，以提升其完整性。
2. 每個階段結束時可自動檢視各個產物的完整性與一致性，減少因人為疏失而造成的錯誤。
3. 提升 ArgoUML 開發環境之效能，使用者可以同時在建立 UML 圖形時檢視系統中已定義的各項產物，並給予使用者建立圖形元素之建議及自動檢查 UML 圖形產物之完整性，改善以往 UML 圖形與檢查工具分開執行所造成的不方便。

本研究的實作是建構在 ArgoUML 開發工具之上，該開發工具架構龐大而複雜且功能尚未完備，限於時間與研究人力，仍有甚多不足之處，未來可加強的地方如下：

1. 後續階段產物的檢查

   軟體開發流程在設計階段之後還有 Implementation 階段與 Test 階段，Implementation 階段主要是將 Design 階段的 design class 以 component 的方法呈現，Test 階段主要是建構 Test case 以及測試計畫的進行。這兩個階段的產物同樣也包含 UML 語言所規範的圖形(如 Implementation 階段的 component diagram)，以及以文件表示的產物(如 Test 階段的測試計畫)。本研究仍未探討到這兩個階段產物間的相關性。
2. 更多 UML 產物檢查的實作

   礙於配合 ArgoUML 的開發進度，本研究無法將所有的產物一致性檢查機制整合進該開發工具中，例如因為 ArgoUML 尚未具有繪製 sequence diagram 的功能而無法完整實作 sequence diagram 與相關產物的檢查。
3. 建構具有產物檢查功能與配合流程的完整開發環境

   目前實作的開發環境僅是在開發流程的階段中利用使用者所提供的資

訊以及已經產生的產物來進行一致性的檢查，而更完備的軟體開發流程工具應該要包含了進行流程的提示、個別使用者的開發進度、團隊合作的產物開發、團隊分工的流程協調等，這些具有整合人力流程控管、產物檢查與文件建立功能的開發工具目前並沒有在商業市場上出現，未來應進一步的將這些多元化的功能整合成一個開發環境，以提升軟體開發的便利性與實用性。

參考文獻

[1] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*, ADDISON-WESLEY 1998

[2] Grady Booch.Object-*Oriented Analysis and Design with Applications*, 2nd edition.Redwood City, CA: The Benjamin/Cummings Publishing Company, 1993.

[3] Ivar Jacobson, et al. Object-*Oriented Software Engineering-A Use Case-Driven Approach*, Wokingham, England: Addison Wesley Longman, 1992.

[4] James Rumbaugh, et al. *Object-Oriented Modeling and Design*, Upper Saddle River, NJ: Prentice-Hall, 1991.

[5] James Rumbaugh. *OMT Insights*, New York: SIGS Books, 1996.

[6] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Software Development Process*, ADDISON-WESLEY 1999

[7] Padmanabhan Krishnan. *Consistency Check for UML*, 1530-1362/00,IEEE 2000

[8] Oliver Laitenberger, *Perspective-based Reading of Code Documents: Special Issue on Information and Software Technology*, Nov. 1997

[9] Pascal Fradet, Daniel Le Métayer, Michaël Périn. *Consistency Checking for Multiple View Software Architectures*, ESEC / SIGSOFT FSE 1999: 410-428

[10] Jan Hendrick Hausmann, Stuart Kent. *Visualizing Model Mappings in UML*, SOFTVIS 2003: 169-178

[11] Tom Mens, Ragnhild Van Der Straeten, Jocelyn Simmonds. *Maintaining Consistency between UML Models with Description Logic Tools*.

[12] Christian Nentwich, Wolfgang Emmerich, Anthony Finkelstein, Ernst Ellmer. *Flexible consistency checking*, ACM Trans. Softw. Eng. Methodol. 12(1): 28-63

(2003)

[13] Gregor Engels, Jochen M. Kuster, Reiko Heckel, and Luuk Groenewegen. *A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models.*

[14] Martin Skinner. Enhancing *an Open Source UML Editor by Context-Based Constraints for Components*, University of Berlin, Thesis, December 2001

[15] Linus Tolke and Markus Klink. *Cookbook for Developers of ArgoUML*

[16] ArgoUML Tour. http*://argouml.tigris.org/tours/index.html*