EFFICIENT DEVELOPMENT OF HIGHLY REUSABLE DISTRIBUTED SYSTEMS USING THE TCAO

Santokh Singh
Centre for Software Innovation
and Dept of Computer Science
The University of Auckland
Private Bag 92019
New Zealand
santokh@cs.auckland.ac.nz

Sheri Zidie Xu
Dept of Computer Science
The University of Auckland
Private Bag 92019
New Zealand
zxu018@ec.auckland.ac.nz

Harveen Kaur
Department of Mathematics
The University of Auckland,
Private Bag 92019
New Zealand
hkau017@ec.auckland.ac.nz

ABSTRACT
Currently there are neither structured nor efficient ways to develop and reuse non-trivial and complex distributed systems despite the fact that there exist numerous sources of software, components and knowledge relating to this field. In this paper we describe how we use a novel all-encompassing development methodology called the Total Components Aspect-Oriented (TCAO) methodology to efficiently and rapidly design and develop reusable distributed systems of any size, functionality and complexity. We further describe our other novel concepts of using Early Aspect-Oriented Components (EAOC) and Early Aspect-Oriented Software (EAOS), collectively known as early systemic-entities, to get an early head-start in the development process starting from the requirements engineering phase itself and spanning throughout all the phases. TCAO also encompasses the activities of deployment, maintenance and refactoring to add any further remoting functionalities if the need arises. We further describe how we use TCAO to identify, isolate and use aspects, which are systemic cross-cutting concerns, to make the analysis, design, modelling and implementations of reusable distributed systems easier to understand, manage and control. With the aid of an exemplar software system, we also describe how a non-trivial reusable distributed system can be efficiently and rapidly developed by using our methodology to facilitate technology transfer and reuse.

KEY WORDS
Distributed Systems, Software Engineering, Aspects and Components

1. Introduction

Distributed applications have become increasingly popular, essential and important but at the same time have evolved into complex and sophisticated systems. The need for such systems stems from their power to provide the much needed capabilities of allowing wide scale sharing of resources over spatially separated machines, higher performance through the use of processors on participating computers, increased reliability in terms of fault tolerance and catering for human-to-human, human-to-machine or machine-to-machine communication through comprehensible interfaces [10], [14]. However, the development of distributed systems has become increasingly cumbersome and complicated because the development process is tedious, difficult control, costly to manage and the resulting software scattered, hard to understand and maintain. Current development techniques, including current Component Based Software Development (CBSD) methodologies, still cannot address these issues though these methodologies have achieved limited success pertaining to the design, development and maintenance of non-distributed software systems [2], [22]. This type of support is glaringly lacking in the development of distributed systems whose development is further complicated due to the fact that the software parts reside on different machines and communicate with each other through a variety of connection mechanisms and protocols like SOAP, HTTP and WAP.

In this paper we propose a novel all-encompassing development methodology called the Total Components Aspect-Oriented (TCAO) methodology to efficiently and rapidly design and develop reusable distributed systems of any size, functionality and complexity. This methodology can be applied to the whole distributed software development process, starting early in the Requirements Engineering phase, stretching through the analysis, design and implementation stages and extending through the delivery and deployment phases. It also encompasses all testing stages and even allows development to be carried out piecemeal-by-piecemeal and iteratively. It is flexible and can be applied equally well in Agile environments like eXtreme Programming techniques [1] or eXtreme AOCE [20], and effectively takes into consideration any subsequent maintenance or refactoring of the remote systems that were developed using TCAO. Above all, it allows for the effective identification and extraction of code, components and subsystems for reuse in other distributed systems to increase productivity and efficiency in their development [9]. We will first describe the motivation behind this research for developing highly reusable distributed software systems and the need for a methodology to support their development.

2. Motivation

Distributed systems that are non-trivial, for instance those used for e-Businesses, are almost always large, complex and currently difficult to develop, maintain and reuse. Such distributed systems use either the internet and/or intranet technologies as the fundamental part of their architecture. The more popular and current infrastructures and technologies for distributed system are either based on CORBA, DCOM, RMI, web services, EDI and XML over TCP/IP, or an extension or combination of these technologies [24]. These infrastructures have advanced mechanisms to abstract remote component interfaces to support cross-organisational communication in a Service Oriented Architecture. Building such systems from scratch can expend vast amounts of development time, skills and other resources including money, manpower and material. A lot of these precious resources can be saved if we can identify, extract and recycle software parts when we develop new remote software systems. But currently there are neither consistent ways nor proper reusable techniques to identify and reuse software from distributed systems. We are as such motivated to expound our concepts of developing highly reusable distributed systems to achieve the objectives of producing understandable, maintainable, reliable and reusable software systems and components more efficiently and productively. We develop such systems by using our novel development methodology called the Total Components Aspect-Oriented (TCAO) methodology that can support our development ideas about efficient software reuse for remoting purposes.

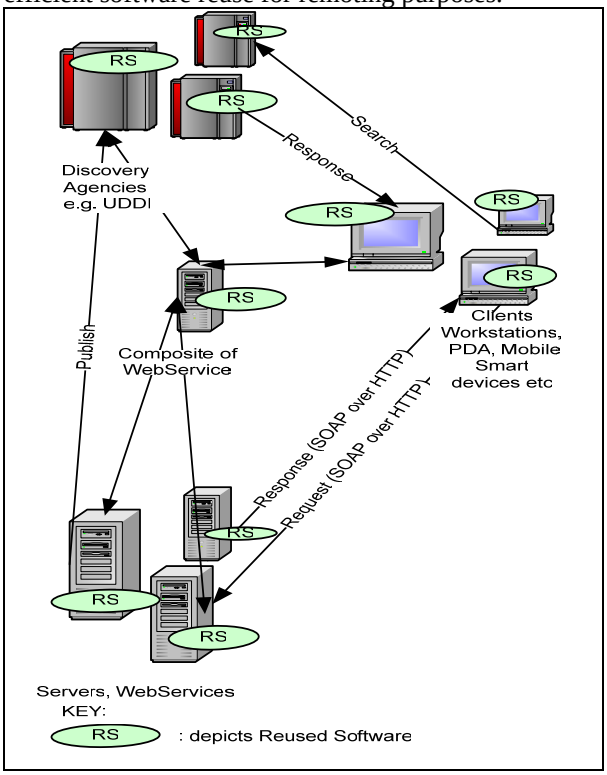

Figure 1: Service Oriented Architecture of a distributed system composed of highly reused software

The Service Oriented Architectural diagram of a reusable distributed system is shown in Figure 1. This system itself can be composed of highly reused software taken from other distributed software systems. It has client-server architecture with discovery agencies to locate servers and enable integration between the clients and the servers. Clients connect to the servers through connection channels and send their requests to the servers using particular protocols, e.g. the Simple Object Access Protocol (SOAP) over HTTP [6], [16].

The reused composite of web services is a more complex subsystem that links to multiple web services and acts as an intermediary between the clients and the multiple web-services that are consumed [18]. The clients call a single composite object which in turn calls the web services linked to it. This essentially makes the task of consuming web services easier for the clients as they do not need to know which web service the composite object is calling. Each client just makes a request to the composite object and gets a response back from the composite, the transactions between the composite and the web services it consumes is hidden from the clients [21].

To be highly reusable, all the designs and implementations of the distributed system and its sub-systems, components and functions need to be well described, precisely documented and capable of indexing so that accurate discovery for software reuse is possible [6], [19]. All the subsystems shown in figure 1 above are composed of some reused software, and the resultant distributed system itself can be reused for developing other distributed systems, hence it is called a reusable distributed system. Development of reusable distributed systems using TCAO increases efficiency and productivity while at the same time bringing about savings in costs, with fewer developers and less machines and resources required. The software produced is also more manageable, controllable, maintainable and understandable compared to that produced using existing methodologies, thus allowing for further efficient and effective reuse of distributed software and its components. In the next section we explore, discuss and critically analyse some of the existing and more popular development methodologies and related research work.

3. Related Work

Component Based Software Engineering (CBSE) techniques have been used, albeit with limited success, to address the issues to make software development easier to manage and control, with the aim of also making the resulting software more reusable for future purposes [4]. With the increase in size, functionality and complexity of current software systems, especially in remotely distributed systems, the currently available CBSE methodologies are unable to achieve these goals. Examples of the more popularly used CBSE methodologies include the Topcoder [2], Aspect-Oriented Component Engineering [13], OMG's Model Driven Architecture [5], The Select

Perspective and the Architecture Based Component (ABC) Composition Approach [17].

TopCoder^{R} is a very comprehensive CBSE methodology, and it has four stages to each release of a component, i.e. the specification, architecture/design, development/testing, and certification stages. If any of these phases fail an acceptance test, the phase has to be restarted. TopCoder^{R} as such is a tedious development methodology and focuses on the lower-level features of the components and software system. The other CBSE methodologies like ABC, OMG's Model Driven Architecture (MDA) and Select perspective also do not focus on the high level features of the components and their services. These can make the designs and their analysis hard to understand at higher levels or during maintenance and refactoring. Higher level systemic component descriptions such as transaction processing, persistency, security, user interfaces, collaboration, configuration etc. are all not addressed in these CBSE methodologies. Such high-level features are important for understanding and using systemic components and their functionalities, especially in distributed systems which can be complex.

On the other-hand the AOCE methodology uses a concept of different cross cutting systemic concerns or aspects (e.g. user interface, persistency, security, transaction processing, resource utilisation, configuration aspects etc.) which are used to categorize and reason about provided and required services of software components in the system [13], [20]. AOCE caters for the identification, manipulation, description and reasoning about the software component's high-level functional and non-functional requirements. These requirements may be grouped by different aspects or a composite of aspects, with "aspect details" and "aspect detail properties" providing an ontology and descriptive language to describe constraints relating to the provided and required properties between components and their compositions [12]. Components in AOCE are implemented using aspect categorisations and characterizations to support aspect-component description, discovery and adaptation. Though AOCE may have extensive description and support for aspects, it does not, and cannot, cater for early component and software discovery and identification for reuse purposes. This hinders efficiency and effectiveness of using components early in the development phases and leads to wastage in terms of time and other resources because these issues are only addressed later on in the development cycle.

All the current CBSE methodologies have attempted to use the best approaches from existing traditional software development methodologies, but have failed to sufficiently utilize the power of community-based development and have not sufficiently addressed the all-important issues of code and designs reusability, understandability and scalability [3]. With the exception of AOCE, all the methodologies tend to focus more on low level issues and features of the software components rather than the components' high-level requirements and inter-component relationships. None of the currently existing CBSE methodologies, including AOCE, have proper steps to identify reusable software for distributed systems nor do they deal with early components and early software. Furthermore none of these development methodologies can be adequately or efficiently applied to all the stages of the development life cycle.

4. Total Components Aspect-Oriented Methodology

To overcome the obstacles and setbacks inherent in the current methodologies described above, we formulated, refined and tested a new and novel development methodology called the Total Components Aspect-Oriented (TCAO) methodology. It is a total development methodology because it can be used in totality for each and every phase of the development process, i.e. including from the inception, analysis, design, implementation stages right through to the software's delivery, deployment, and subsequent maintenance (if any) of software systems. TCAO covers and caters for all the stages and aspects of the software development process and cycle, and can also be applied for the discovery, identification and reuse of the whole software system or its components and subsystems. The concept of early components, early software and cross-cutting issues (aspects) used in TCAO and its software development procedures are discussed in the following subsections.

4.1 Early Aspect-Oriented Components and Software

Early Aspect-Oriented Components and Software are respectively software components and subsystems which are identified and used very early in the Requirement Engineering stage itself. These components and software are aspectized to address the issue of cross-cutting concerns that arise starting from the Requirements Engineering phase. An example of these concepts used in a practical situation is mentioned below whereby we reuse the software from a collaborative Travel Planner system to rapidly develop an Online Banking system using TCAO.

Figure 2 above show the Early AO-Components and Early AO-Software that was identified, extracted and reused from the collaborative Travel Planner prototype for the new Online Banking System. Any reused software that needed adaptation was refactored and modified so that it can be made fit for use.

The aspects in both the corresponding components and software from the Travel Planning system and the Online Banking system are the same as such the software can be more easily adapted to suit the new system. The customer and staff components shown in both systems have same use, aspects and support same operations; therefore they can be practically reused verbatim, except with minor additions like the inclusion bank account objects. Our approach can also be applied to the reuse of generic components from various different software applications, provided the components have the requisite functionalities needed for the new software.

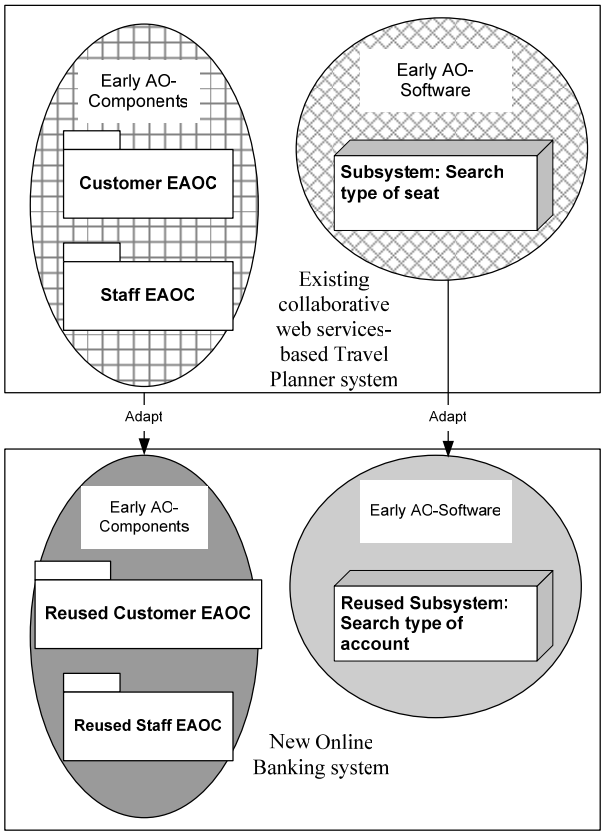

Figure 2: Early AO-Components and Early AO-Software identified, extracted and reused for the Online Banking System from the Collaborative Travel Planner.

The search subsystem of the Travel Planner could also be beneficially reused through modifications/alterations to the functional name and input parameters, but the business logic, aspect types and details remain essentially the same. This example showed us that we can salvage and reuse software from systems provided we create and document them consistently and clearly.

4.2 Procedures involved in TCAO

TCAO allows for the effective identification and reuse of recyclable software or components very early in the development process, i.e. starting from the requirements engineering phase itself. This is done after the development team has ascertained the specifications of the software to be constructed. The steps involved in developing software systems using TCAO are described below:

1. First, the initial Requirements Engineering (RE) diagrams based on the client's specifications and through discussions with all the relevant parties are drawn and agreed upon by all concerned.
2. Next, based on the RE diagrams drawn in (1.) above, we identify all Early AO-Components and Early AO-Software from existing distributed software that has the possibility to be reused in the new distributed system. We also componentize the various use cases in 1. We search for suitable and matching components and subsystems from our repository of existing and available reusable components, software etc. The functions and location of all discovered software that is identified as reusable is be recorded.
3. If the components, aspects, software etc. cannot be found in our repository, we can search for them in other sources, for example, the internet, from other developers or Commercial Off The Shelf (COTS) software [7], [23].
4. After finally ascertaining the most suitable software parts we document and record the location of the identified reusable software parts collectively called Early Systemic Entities and mark this information into the RE diagram, to complete the TCAO use case diagrams.
5. We then draw the TCAO analysis diagram based on the use cases and Early Systemic Entities.
6. The TCAO design diagrams, sequences diagram, collaborative diagram are then drawn.
7. All the drawings and designs are well documented, regularly updated and stored for any current and future reuse purposes.
8. The reusable early systemic entities are then extracted and incorporated into the library folders and packages of the system to be developed. The distributed system is then implemented by auto-code generation.
9. The generated code is then further debugged, refined and tweaked. The TCAO process uses iterative testing and refactoring until the distributed system is developed based on the required specifications.
10. The system is then delivered and deployed for the client.
11. Any further maintenance, upgrading and refactoring of the distributed system can be carried out more consistently and coherently as the code, patterns, architectural, analyse and design diagrams are well documented and easily accessible to those who are responsible for the activities [8]. The documentation also aids in the reuse of the software and its parts. All documentation is kept updated and current, using two styles of documentation, one for machines to assist in automatic discovery and the other more descriptive with more elaborate instructions for human consumption.

```
<?xml version="1.0" encoding="utf-8" ?>
<definitions…..
xmlns:wsdl="http://localhost/MyWebService/
bin/mywsdlSchema.xml" ……>
.....
                    <aowsdl:ReusedComponents
Name="Persistency_SearchAccountTypeCompo
nent">
            <wsdl:Documentation
Information="Exposes aspects to find types
from a particular bank. After finding the
account type users can make specific
transactions depending on… All human
readable information go here. This includes all
instructions and high level documentation
about the reused components/web service for
human consumption. " />
        <wsdl:Description Description="To
search for account types to make financial
transactions" />
```

Figure 3: XML styled documentation for the reused early components and software

Documentation is of paramount importance in TCAO though the code itself is written in a consistent and clear manner and can act like the documentation for very good developers. The documentation is written in XML format because it can be used as the universal data format for integrated electronic business solutions, is consistent, extensible and can be written so that is can understood by other machines to allow for dynamic discovery of components.As shown in Figure 3, TCAO provides two levels of documentation i.e. one called the Information element is human readable, and is very verbose and written in high level language. The other one is the Description element, which is machine readable and is compact, crisp and written with fewer words, mostly keywords. This machine readable format allows for automatic discovery for the software and components. The software is also classified according to proper categories and subcategories to enable easier search. Examples of the top level categories include database functionalities, security, higher level components like a Person Component etc. while the subcategory example include login and password (for the security category), Transactions and Connections (for Database) and Customer, Staff etc, components (for the Person Component).

If the client decides, alterations and modifications to the software's specifications, designs and implementations can be made during any of the development stages because the software is composed of components that are more understandable, pluggable and playable. The designs, components and software from the newly built reusable distributed system can also be efficiently reused in other software applications. Besides being an efficient and effective CBSE methodology, TCAO also better modularize and divide the distributed system into components coherently and consistently across different platforms, technologies and domains.

5 Using TCAO to develop an Online banking system

To show that the TCAO methodology is efficient, effective and practicable, we will illustrate how we can apply it to design and develop an Online Banking prototype system from existing reusable components and software. We used an existing distributed software system, a collaborative web services-based Travel Planner system to extract the Early Aspect-Oriented Components (EAOC) and Early Aspect-Oriented Software (EAOS), which we reused in our new system.

In all our diagrams and designs, we abbreviated the aspect types according to the following notations.

Aspect type	Notation	Provided/required
Security	[[Sec]]	A positive '+' sign prefixing the aspect (e.g. [[+Sec]]) indicates that the aspect is provided by the component while a negative '-' sign (e.g. [[-Sec]]) means that the aspect is required by the component.
Persistency	[[Pers]]	
Transaction Processing	[[TP]]	
User Interface	[[UI]]	
Resource Utilization	[[RU]]	
Collaboration	[[Col]]	
Configuration	[[Conf]]	
Performance	[[Perf]]	

Table 1: Some aspect types identified in our reusable distributed systems

Some of the identified aspect types and their abbreviations used in our distributed systems are shown in Table 1 above. Each aspect type has aspect details and properties associated to the aspect. For instance, the aspects that are "provided" by software components are prefixed with a "+" sign and those "required" with a "-"sign. We further use a consistent naming convention to write the aspects/functions by prefixing the sign and aspect type to the aspect name separated by an underscore. For instance, [[+Sec_authenticate()]] means that the particular component provides an aspect of the type security which enables authentication functions to be performed. This function cross-cuts many parts of code and components in both the Travel Planner and the Online Banking System, and as such is required by numerous components in the software. Using our conventions, it will be written as [[-Sec_authenticate()]], the negative sign to signify that the aspect is required by the component/system.

The high level specifications of the Online Banking prototype is that it should be a distributed system capable of handling secured transactions over the internet for managing an authorised person's finances wherever and whenever they need to do it, i.e. as long as they have internet access. All transactions are also to be done in real time and capable of rollback if the transaction does not complete or is aborted.

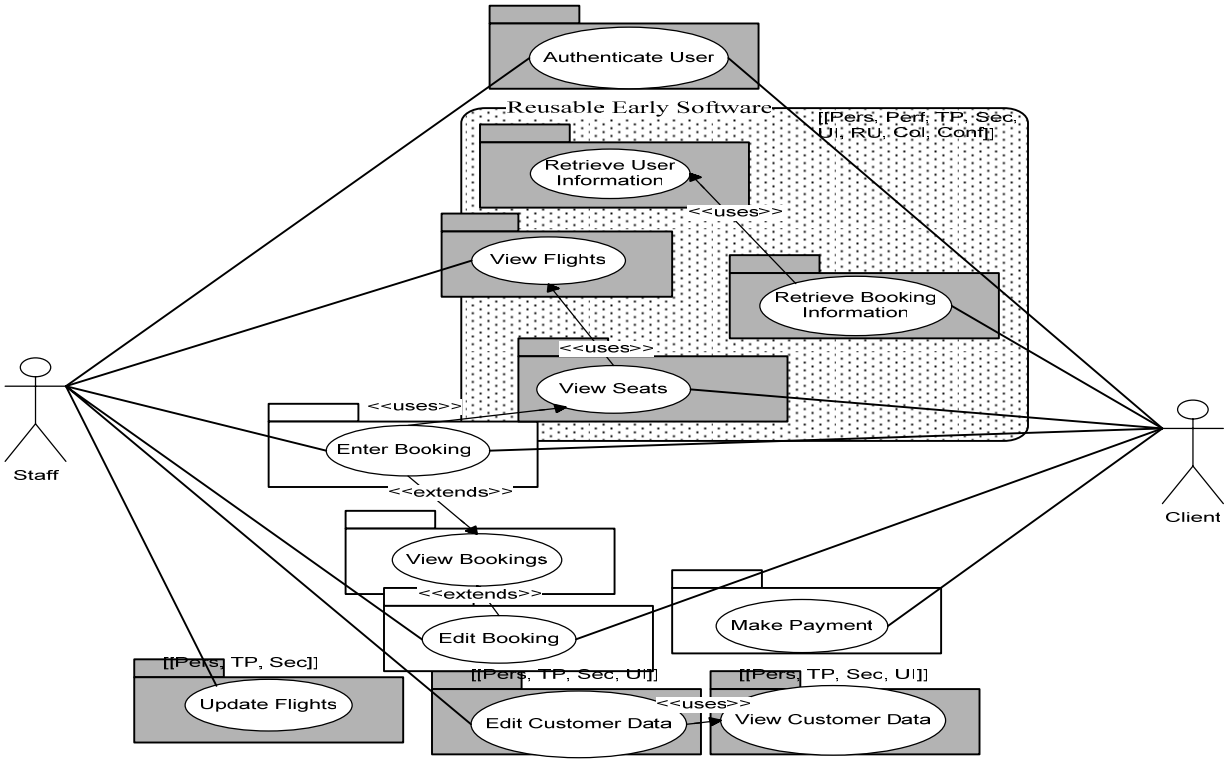

Fig. 4 Enhanced TCAO use case diagram of the collaborative Travel Planner system with the reusable Early Systemic Entities shown shaded.

Figure 4 shows the enhanced TCAO Requirements Engineering diagram from the Flights system of the collaborative Travel Planner application with the reusable Early Systemic Entities clearly marked out for consideration. The reusable early components are shaded grey and the reusable early software is placed within a dotted-box. The aspects involved are shown in square brackets e.g. [[Pers, TP, Sec]]. The unshaded components are not reused in the Online Banking System.

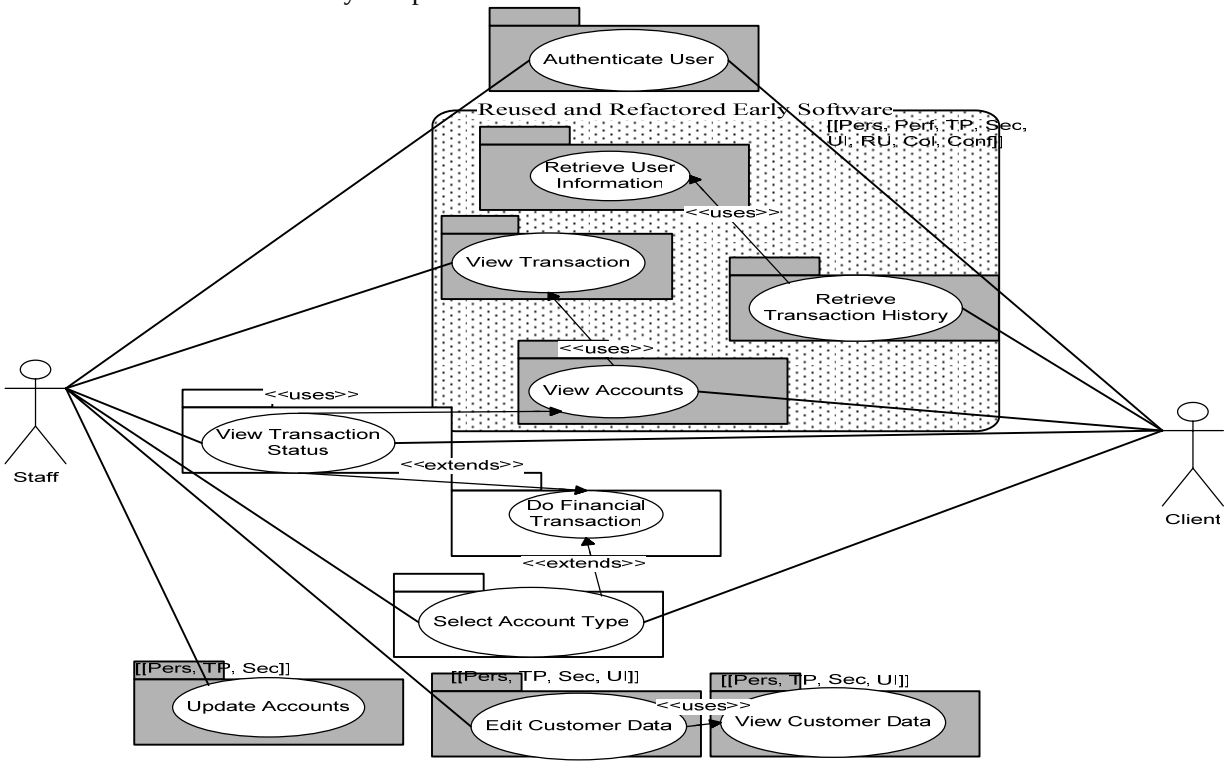

Fig. 5 Enhanced TCAO use case diagram of the new online banking system with the Early Systemic Entities reused and refactored from the Travel Planner

Where necessary, we also adapted and modified those identified early systemic entities so that they can be reused beneficially in the Online Banking System. Figure 5 shows the Requirement Engineering diagram of the Online Banking system with the reused individual components shaded grey and the reused early software within the dotted-box. This concept of identifying and using early systemic entities during the Requirement Engineering phase gives developers a head-start and allows for shorter development time, thus increasing productivity. The unshaded components and software parts are not retrieved from the Travel Planner system, but if available, they can be extracted from other existing software or Commercial off the shelf (COTS) components. If such components are unavailable, then they need to be designed and developed from scratch as per the specifications of the application.

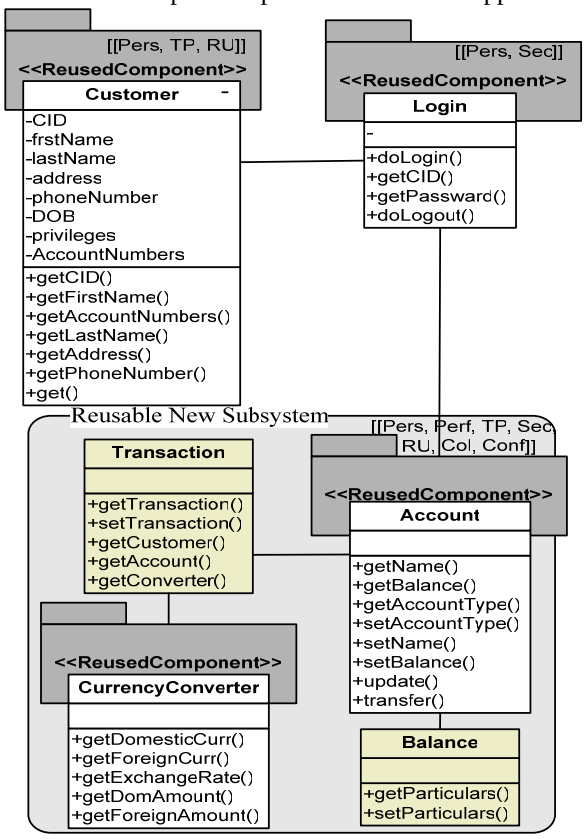

Figure 6: Design diagram showing the reused components and example of the construction of new reusable subsystems.

The design diagram in Figure 6 shows the reused components and an example of how new reusable subsystems are constructed so that they can be further reused in other distributed systems. The aspects are identified and specified for each and every component and subsystem. The XML-based documentation is consistently updated at all times so that we can refactor and modify the application more easily if the need arises. Good documentation also aids developers to understand the designs/implementations faster if they join the development team after the project has already been launched. This has the effect of making the developers more productive at the earliest possible time by reducing their time taken to learn and familiarise themselves with the well-documented system.

6 Discussions

Both the collaborative Travel Planner and the Online Banking System were built one after another by 2 pairs of different developers, each pair working together in agile conditions, i.e. taking turns to be the navigator and driver of the development process respectively and development time was measured. The collaborative Travel Planner took a very long time to develop, i.e. about 6 months (i.e. 750 hours, averaging about 30 hours per week of joint effort). The reason for the long time span was mainly because there was very little easily accessible and reusable software [15]. On the other hand the Online Banking System took only about 2 months to develop, i.e. about 300 hours of total development time, in large part due to the availability of the well-documented and reusable distributed systemic parts from the Travel Planner System.

The Travel Planner was practically built from scratch and there were no proper reusable software parts besides code snippets that had to be aspectized and componentized. There were very few early systemic entities in the form of early components and early software available to be plugged into its requirements engineering phase. Existing code also had very poor documentation associated with it and the developers' motivation level at times was very low as there were not many avenues to search for existing designs and implementations.

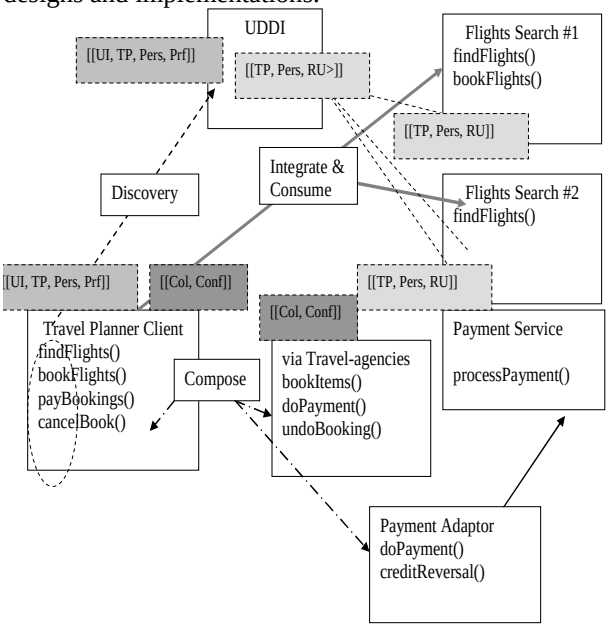

Figure 7 The Travel Planner System that has reusable software

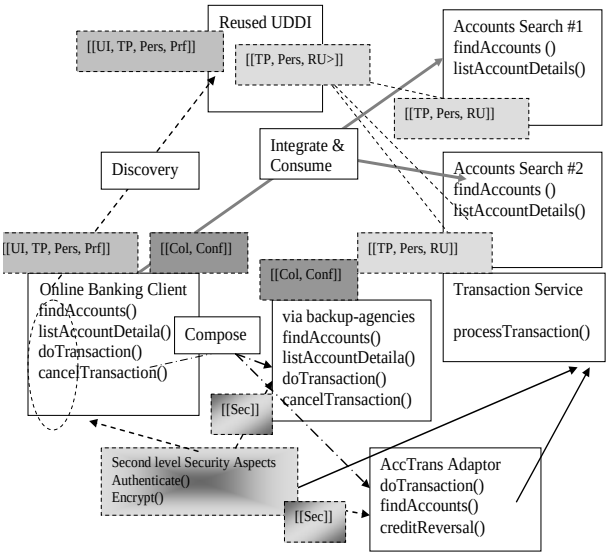

Figure 8 The Online Banking System built from reusable software

Figure 7 shows the inter-relationships between the various distributed sub-systemic parts in the collaborative Travel Planner system, including the aspects and operations involved. Figure 8 illustrates the inter-relationships between the various distributed sub-systems in the Online Banking System. As can be seen, there exists similarities and differences between the two systems, the similarities are more pronounced in the overall architecture as both are distributed systems and are built using the same technology, i.e. both use the Universal Description, Discovery and Integration tool as the discovery agency, use web services for remote operations, SOAP over HTTP as the transport and communication protocol etc. These similarities were found to have a very important impact in enhancing the reusability of software in distributed systems.

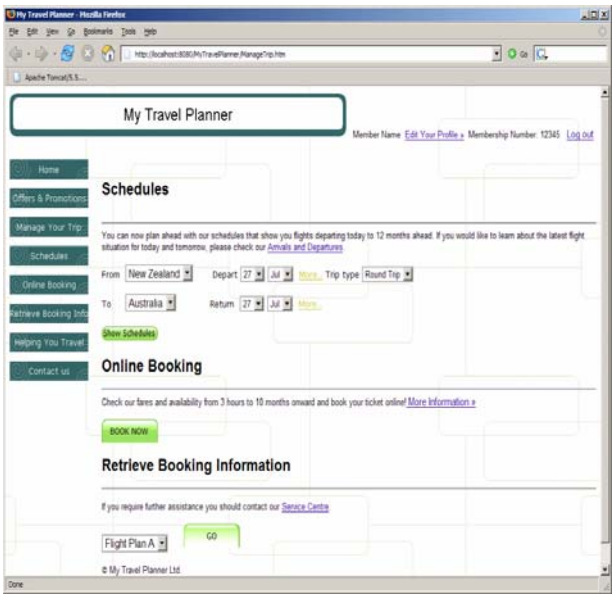

Figure 9: The collaborative Travel Planner providing the reusable early systemic entities

Figure 10 The Online Banking System built from reusable software

Figures 9 and 10 depict the Browser User Interfaces (BUI) of the collaborative Travel Planner and Online Banking System respectively. The Human Computer Interface ideas, designs and code for the Travel Planner BUI were reused in the Online Banking System. This is because the concepts and actions involved in both applications, e.g. the submit buttons, text fields action events etc. were essentially the same and could be efficiently migrated across with adjustment to the underlying business logic.

One of the problems that we discovered associated with utilising early software is trying to reuse software that finally turned out to be not of the correct type. Going along wrong tracks can cause wastage in terms of time and resources as the software may ultimately be unsalvageable and not of the type that we wanted. This was mainly due to the fact that the software was obtained from third party vendors and was either incorrectly or insufficiently documented and in most cases only the byte code was available with the source code missing. One proposed solution to this is to use only software from trusted vendors or developers that is well documented with all source code supplied.

We may be encountered some difficulty if we try to reuse very large early systemic entities, because the bigger the component or subsystem being considered for reuse, the more adaptation and modification work there may be that needs to be done on the entity. An example of a very large entity that we reused from the Travel Planner in the Online Banking System is shown in Figure 11 below. It is a hotel room's booking subsystem that was reused to make and confirm financial transactions in the Online Banking System. Though the adaptation and refactoring of large reusable pieces software like this can take a fair amount of time and resources, we found that the time spent refactoring and modifying such existing reusable software so that it is fit for reuse is still less than that would have been required to build it from scratch.

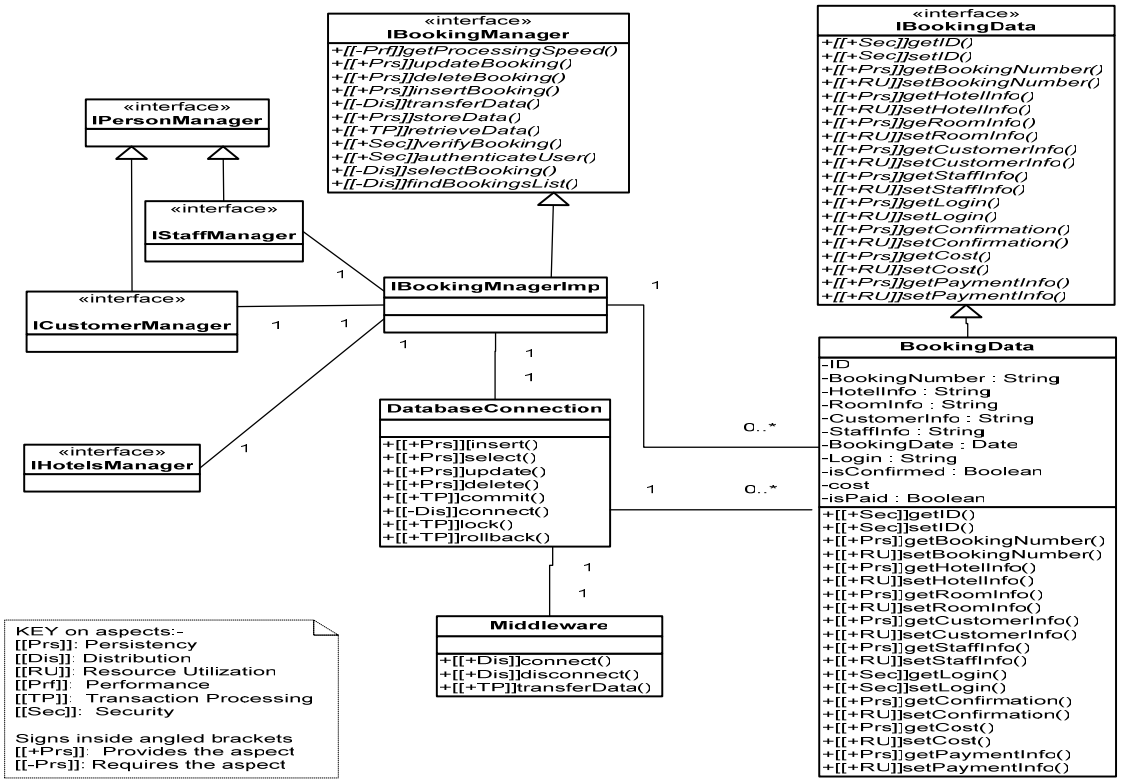

Figure 11: High level TCAO designs of a large reusable early systemic entity from the collaborative Travel Planner system.

Our experience of development by using TCAO and reusable software in the Online Banking System also showed us some other exciting findings. We realised that through our clear coding conventions and good XML-based documentation practices, we were able to easily identify already developed software parts and components within the Online Banking System itself that can be further duplicated, adapted and reused within this system itself. For instance, the component for home loan calculations was duplicated, refactored and reused in other components namely for the budget calculating component, personal lending component and business lending component. This made the development work much easier, efficient and productive.

7 Conclusions

We described how we can use a novel all-encompassing development methodology called the Total Components Aspect-Oriented (TCAO) methodology to efficiently and rapidly design and develop reusable distributed systems of any size, functionality and complexity. The distributed systems are developed using identified Early Aspect-Oriented Components (EAOC) and Early Aspect-Oriented Software (EAOS), collectively known as early systemic-entities. These entities are used starting from the requirements engineering phase itself so as to get a head-start in the development process. TCAO is a total development methodology that can be applied to all the development phases of distributed software development, i.e. starting from the inception phases and spanning throughout all the phases, including the deployment, maintenance and refactoring of the system to add any further remoting functionalities. We can also use TCAO to identify, isolate and use aspects, which are systemic cross-cutting concerns, to make the analysis, design, modelling and implementations of reusable distributed systems easier to understand, manage and control.

To have more control and to increase efficiency and productivity in developing distributed systems, our development methodology supports iterative development cycles and can also be employed using test-driven agile approaches. Through our clear and consistent coding conventions and good XML-based documentation practices, we were able to easily identify software parts and components in distributed for reuse. We also demonstrated how we developed a non-trivial reusable distributed system efficiently and rapidly through reusing early systemic-entities. The development of such systems using reused software slashed the development time by more than half. Our future works include building tool support for TCAO and using these tools to develop, refactor, componentize and document distributed software systems to make them more reusable. This will also help expand our repository and library of reusable software parts and designs that will be beneficial to developers of distributed systems in particular and the software community as a whole.

8 References

[1] P. Abrahamsson, Extreme programming: first results from a controlled case study. *Proc. 29th Euromicro Conference "New Waves in System Architecture"*, 2003, 259-266.

[2] P. Allen, & S. Frost, *Component-based development for enterprise systems: Applying the Select Perspective* (Cambridge University Press, NY: SIGS Books, 1998).

[3] G. Bastide, A refactoring-based tool for software component adaptation. *Proc. 10th European Conference on Software Maintenance and Reengineering*, 2006.

[4] A. Bertolino, & R. Mirandola, Towards component-based software performance engineering, *Proc. 6th Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction at ICSE*, 2003, 1-6.

[5] J. Bezivin, S. Hammoudi, D. Lopes, & F. Jouault, Applying MDA approach for Web service platform. *Enterprise Distributed Object Computing Conference, Proc. Eighth IEEE International*, 2004, 58-70.

[6] E. Cerami, *Web services essentials - distributed applications with XML-RPC, SOAP, UDDI & WSDL* (Sebastopol, CA: O'Reilly, 2002).

[7] K. Crowston, J. Howison and H. Annabi, Information systems success in free and open source software development: theory and measures, *Software Process Improvement and Practice, 11*(2), 2006, 123-48.

[8] M. Duell, Looking beyond software to understand software design patterns, *Computer Software and Applications Conference, Proc. The Twenty-Third Annual International*, 1999, 312-313.

[9] W. Frakes, & C. Fox, Sixteen questions about software reuse, *Communications of the ACM, 38*(6), 1995, 75-87.

[10] D. Garmus, and D. Herron, *Function point analysis, measurement practices for successful software projects* (Addison-Wesley Information Technology Series, 2001).

[11] Gómez, M., Plaza, E. (2004): Extending matchmaking to maximize capability reuse. *Proc. 3rd International Joint Conference in Autonomous Agents and Multiagent Systems*, 2004, 114-151.

[12] J.C. Grundy, Multi-perspective specification, design and implementation of software components using aspects. *International Journal of Software Engineering and Knowledge Engineering, 10*(6), 2000, 713-734.

[13] J.C. Grundy, & J.G. Hosking, Developing software components with aspects: some issues and experiences. *Chapter 25 in Aspect-Oriented Software Development*. Prentice-Hall, 2004, 585-604.

[14] A. Knight, & N. Dai, Objects and the Web. *IEEE Software, 19*(2), 2002, 51-59.

[15] A. Liotta, C. Ragusa, & M. Ballette, A novel framework for the evaluation, verification and validation of distributed applications and services, *World Scientific and Engineering Academy and Society Transactions on Computers, 2*(4), 2003, 1108-13.

[16] M. Litoiu, Migrating to Web services - latency and scalability, *Proc. Fourth International Workshop on Web Site Evolution, IEEE CS Press*, 2002, 13-20.

[17] H. Mei, ABC: supporting software architectures in the whole lifecycle, *Proc. 2nd International Conference on Software Engineering and Formal Methods*, 2004, 342-343.

[18] R.S. Moreira, G.S. Blair, & E. Carrapatoso, A reflective component-based and architecture aware framework to manage architecture composition. *Proc. of 3rd Int'l Symp. On Distributed Objects and Applications*, 2001, 187- 196.

[19] M. Niazi, D. Wilson, & D. Zowghi, Critical success factors for software process improvement implementation: an empirical study, *Software Process Improvement and Practice, 11*(2), 2006, 193-211.

[20] S. Singh, H. C. Chen, O. Hunter, J. C. Grundy and J. G. Hosking, Improving agile software development using eXtreme AOCE and Aspect Oriented CVS, *APSEC 2005*, 752-762.

[21] M. Stearns, & G. Piccinelli, Managing interaction concerns in web-service systems, *Proc. 22nd International Conference on Distributed Computing Systems Workshops*. Vienna, Austria, 2002, 424-429.

[22] P. Vitharana, F. Mariam, & H. Jain, Design, retrieval, and assembly in component-based software development, *Communications of the ACM, 46*(11), 2003, 88-97.

[23] T. Wanyama, & B.H. Far, Towards providing decision support for COTS selection, *Electrical and Computer Engineering Canadian Conference*, 2005, 908-911.

[24] L.J. Zhang, H. Li, H. Chang, & T.Chao, XML-based advanced UDDI search mechanism for B2B integration, *Proc. the Fourth IEEE International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems*, 2002, 9-16.