PADT: SSAD-Streamlined and Component-Based Development

Perspective, Activity, and Documentation Transitioning Approach

Sheldon X Liang
Department of Computer Science, Azusa Pacific University
701 E Alosta Ave., Azusa, CA 91702

and

Samuel Sambasivam
Department of Computer Science, Azusa Pacific University
701 E Alosta Ave., Azusa, CA 91702

ABSTRACT

*CBSE (**C**omponent-**B**ased **S**oftware **E**ngineering) claims to offer a radically new approach to the design, construction, implementation and evolution of software applications. However because of unclear and ambiguous requirements, it is very difficult hence to design and build components successfully. To the contrary, SSAD (**S**tructured **S**ystems **A**nalysis and **D**esign) is considered a pragmatic Requirements Engineering method that can be well documented and educated, therefore represents a pinnacle of the rigorous documentation-driven approach.*

After years of research on software architecture and teaching Capstone Project through SSAD, the authors have realized that there has existed the gap between many aspects throughout SDLC (System Development Life Cycle). This paper presents the PADT framework that provides software engineering with a new transitioning view, and that accommodates multi-systems perspective, developmental activity, and deliverable documentation in support of streamlining SSAD through to CBSE.

1. INTRODUCTION

CBSE (**C**omponent-**B**ased **S**oftware **E**ngineering) offers a radically new approach to the design, construction, implementation and evolution of software applications. The modular structure of a CBSE solution allows individual components to be replaced easily at design time or run time, i.e. if a software application is assembled from components, it is easier to reconfigure the components to support desired changes in the business process. CBSE views the system as a set of off-the-shelf components integrated within an appropriate architecture [1,2,3]. However, there is no evidence that components are more natural to think than functionality especially when an information system is under development from scratch. So with unclear and ambiguous requirements, it would be difficult to design and build components successfully.

1.1. Why Structured Systems Analysis?

SSAD (**S**tructured **S**ystems **A**nalysis and **D**esign) is a straightforward and pragmatic Requirements Engineering method that can be well documented and educated [3, 5], despite both supporting arguments and criticisms on the waterfall model. SSAD has an important feature that stakeholders are intensively involved in the requirements analysis stages and usually required to approve the deliverables at all stages as they are completed to ensure the system meets their needs. Currently, SSAD represents the pinnacle of a rigorous documentation-driven approach, which plays a great role in software engineering education. SSAD provide working design documentation to help team members understand each other.

After years of educating Capstone Project through SSAD [6], the deliverables for stand-alone software projects have been well-developed and standardized at Azusa Pacific University, such as PRD (problem requirements document), PSD (project specification document), and SDD (software design document). With the standardized documentation, engineering software has become *traceable*, *teachable,* and *team-workable*.

1.2. Why Systems Perspective Transitioning?

Software projects involve different types of people who see the system from heir own perspective through SDLC. Along with the waterfall model, SSAD *promotes* pragmatic Requirements Engineering approach at the analysis stage, but *discourages* frequent changing of requirements at the later development stage [8,9]. CBSE uses the modular structure solution to achieve individual components to be replaced with ease at design time or run time [1, 2,3].

Actually, SSAD and CBSE complement each other because SSAD focuses on the Analysis and Design stages of SDLC, while CBSE stresses modular structures that enable components substitution. The transitioning from SSAD to CBSE itself reflects a shift in the focus from a conceptual perspective (requirements) to a technical implementation.

1.3. Previous Work and Contribution

After years of architecting reliable architectures for software-intensive distributed systems [10, 11, 12, 13, 14, 15,16], cohort-educating Capstone Project at Azusa Pacific University [6], the authors understood that software architecture plays a crucial role in bridging the gap between system requirements and implementation [7,8]. Besides the conceptual requirements, and the operational implementation, the technical architecture is

used to force the architect or architecture team to consider the key design aspects early and across the whole system. One of the most recent accomplishments is an architectural description language (re-ADA) that supports to generate the prototype into the intended product system [11,12].

We have expected all products executable across stages to involve stakeholders to review and revise the documentation [10,17] via documentation-driven approach [11,15] in order to automate engineering software. This paper aims to bridge the gap between SSAD and CBSE via systems perspective transitioning to make full use of the combined advantages – effectiveness of acquiring requirements at early stages, and of adjusting requirements at later stages.

The main contribution is the PADT framework that provides software engineering with a new transitioning view. The traditional layered technology view [9] stresses the supportive across borders between layered technologies, instead, the PADT transitioning effects the transitioning from perspective to perspective via automated developmental activities/tools. The specific contribution is the bridge between SSAD and CBSE via the descriptive architecture in re-ADA. Another important contribution to software engineering is the fulfillment of products executable throughout SDLC – as the systems perspective proceeds in transitioning, multi-staged products (from the prototype, through the midway products, to the complete product).

2. TRANSITIONING VIEW

Software engineering emphasizes quality by integrating multi-technologies into the subject in a well-layered way, including tools, methods, process models, and at last, a "quality focus"[9]. From an educator's and a software engineer's perspective, a breakthrough is made among those layers through systems **perspective**, developmental **activity**, and deliverable **documentation transitioning** approach.

Three systems **perspective** is incorporated into the PADT framework reflect different concerns of the stakeholders [4,7,8]. The *conceptual* system is introduced to address "What-To-Do" showing the functional view of the system whose major focus is about functional requirements and constraints, the *technical* system to specify "How-To-Do" showing a structural view of the system with the interior structure "exploded", and the *operational* system to fulfill "Way-To-Go" with all functional requirements and constraints implemented within a complete system.

A set of development **activity** is associated with major aspects of PADT framework including systems perspective, such as *require* / *define* (by the customer / analyst) the conceptual system, *design* / *review* (by the designer / maintainer) the technical system, and *develop* / *run* (by the programmer / operator) the operational system.

A set of **documentation** has been successfully integrated in the PADT framework, as part of software engineering education curriculum at Azusa Pacific University. *Capstone Project* [6], on the basis Adaptable Process Model [18] consists of the following standardized documents:

- PRD: Project Requirements Document;
- PSD: Project Specification Document;
- SPMP: Software Project Management Plan;
- SDD: Software Design Document;
- ATP: Acceptance Testing Plan;
- SOM: Software Operator's Manual;
- WDSC: Well-Documented Source Code

The perspective **transitioning** from SSAD to CBS itself reflects a shift of focus on from the conceptual requirements, through technical architecture, to operational implementation.

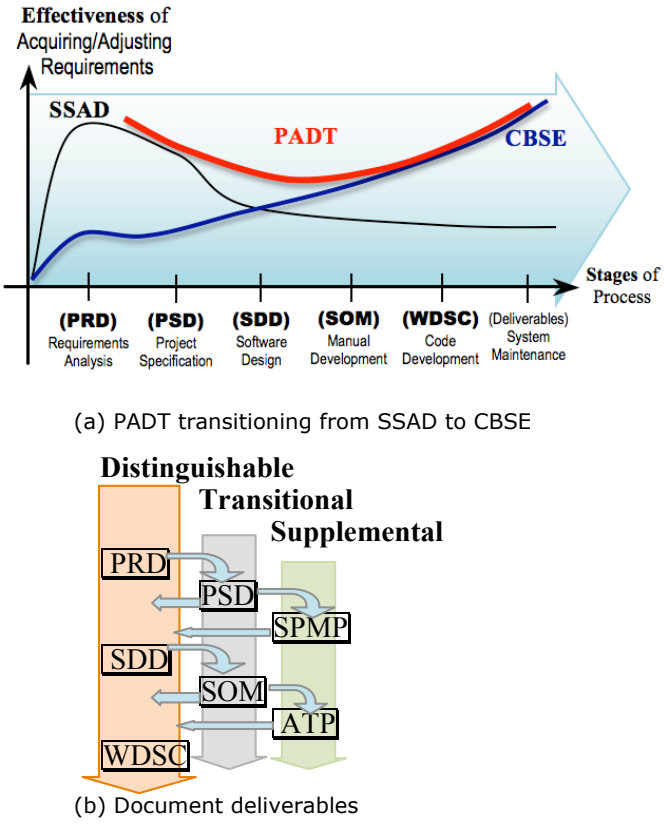

Figure 1. Perspective Transitioning Deliverable Evolution

Figure. 1(a) illustrates the perspective transitioning view in support of engineering software via documentation-driven perspective transitioning throughout SDLC: PADT pursues *to streamline SSAD through to CBSE* in order to make full use of the combined advantages – effectiveness of acquiring requirements (with SSAD at early stages) and effectiveness of adjusting requirements (with CBSE at later stages).

3. TRANSITIONING PROCESS

To engineer software systems is the process of manufacturing software systems, intended to try the best-practice processes to create and/or maintain software. A software system consists of executable computer code and the supporting documents needed to manufacture, use, and maintain the code [19].

3.1 Engineering Documentation-Driven Process

According to the PADT transitioning view, there exist gaps to be bridged between different methodologies – SSAD and CBSE, deliverable documents – requirements and specification, etc, all which leads to systems perspective transitioning issue:

How to evolve a product system from a prototype?

Documentation-driven approach will do the trick in support of perspective transitioning while engineering software systems. Documentation or documenting everything plays a crucial role in engineering software systems, so the question "must everything be documented?" has an unequivocal answer "YES" [21]. The software development process (SDLC) itself is usually divided into phases whose ordering, and the interactions between the phases specify a software life-cycle model that refers to developmental activities, such as requirements analysis, design, implementation, and maintenance. And all the result from the activities should be documented. The process by its definition [22] reflects the sequence of interdependent and linked procedures that (at every stage) consume one or more resources to convert inputs into outputs -- in particular, the output of each phase serves as the input to the next [23].

3.2 Deliverable Evolution

Starting with requirements analysis, SSAD has much to be documented, such as structured analysis diagram, HIPO, flow chart, data dictionaries, program comments etc. Based on three systems perspective in PADT, deliverable evolution from document to document not only keeps track of the key content that dominantly characterizes the systems perspective, but also reflects focus shifting against different systems perspective. That is, the key conent evolves or expands to the extent that new document can be derived.

Figure. 1 (b) also illustrates the deliverable evolution through Capstone Project with a set of documentation well standardized as follows:

- Three *distinguishable* documents as major deliverables for three systems perspective;
- Two *transitional* documents for transitioning across systems perspective;
- Two *supplemental* documents in support of management and test plan.

Table 1. Perspective, Activity and Documentation associated with Tools

Perspective	Activities		Documents	Tools
Conceptual →What-To-Do	Require Review	Define Decompose	PRD	• Requirement acquiring • SCD (system context diagram) • GUI (Graphical User's Interface) • RSP (rapid systems prototyping)
Perspective Transitioning	Review Revise	Define/ Decompose/ Plan	PSD SPMP	• High-level DFD (data flow diagram) • Top-level architecture • Solid Structured Chart • ERD (Entity-Relationship Diagram) • Gantt Chart (for scheduling)
Technical →How-To-Do	Revise Refine	Design/ Redesign Refine	SDD	• Hierarchical DFD (data flow diagram) • Architectural design • CBD (Component-based development) • Pseudo code (for algorithms)
Perspective Transitioning		Describe/ Plan	SOM ATP	• MID (Module Interface Description) • Operator's manual template • Black-box testing plan
Operational →Way-To-Go	Run Repeat	Develop Debug	WDSC	• Flow chart • Programming languages • Application Programming Interface

With three systems perspective, PRD is used to define the conceptual system with the emphasis on functionality (behavior), SDD design the technical system on structure (organization), and WDSC develop the operational system on functional procedures and constraints within a complete system.

With regard to two transitional documents, PSD refines PRD into the extent that SDD can start to add design details; SOM describes functional procedures under certain constraints so that WDSC can start to implement procedures in a certain programming language.

SPMP is used to plan the software project in terms of management including scheduling, cost control and budget management, resource allocation, collaboration software, communication, quality management and documentation or administration systems. And ATP in software engineering is black box testing performed on a system prior to its delivery.

3.3 Perspective Transitioning via Development Tools

SSAD provides an analysis and design framework or set of development tools that can be adopted by people with sufficient experience and expertise. So tools are applied to analysis and design in accordance with deliverable evolution associated with Perspective, Activity and Documentation

Table 1 highlights tools in support of perspective transitioning process from document to document. For instance, the deliverable evolution from PRD, through PSD, to SDD refers to system context diagram, high-level DFD, top-level architecture, and architectural design.

4. PERSPECTIVE TRANSITIONING

With the aim of automating the systems perspective transitioning (as shown in Fig. 1), PADT takes substantial action to convert the interior structure (represented by high-level DFD) into top-level architecture via substitutable interoperation with re-ADA [11,12]. This is a crucial step moving toward CBSE with software architecture constructed. Substitutable interoperation is an interoperable architecture under which all the components (decomposed by high-level DFD and hierarchical-level DFD) are substitutable in the first place, and secondly the component substitution is fulfilled by extending a new component to substitute the old one. Substitutability and extensibility [24,25] of components play important role in support of CBSE.

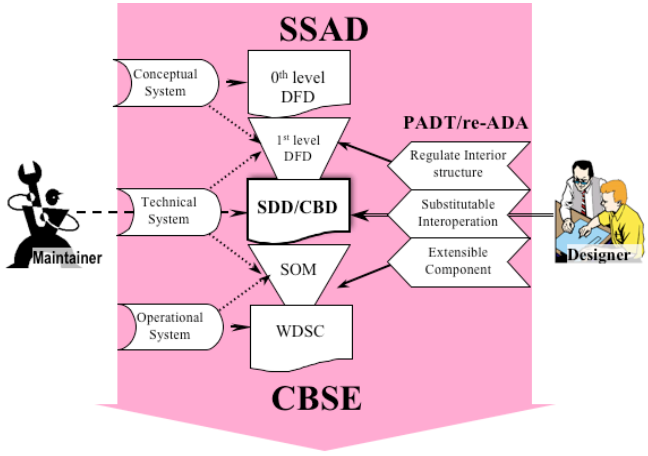

Figure 2. CBD via Substitutable Interoperation

4.1 re-ADA Streamlining SSAD through to CBSE

The software architecture is a structural plan that describes the elements of the system in how they work together to fulfill the system's requirements [17]. The central to PADT framework in support of streamlining SSAD through to CBSE is the reliable

Ada-based Descriptive Architecture Language (re-ADA) that introduces software architecture to software design.

Fig. 2 illustrates that substitutable interoperation plays an important ole in bridging the gap between SSAD and CBSE. 1st level DFD represents the interior structure, and this can be architected by the substitutable interoperation in re-ADA. Based on this, SDD and CBD (component-based development) are introduced with the emphasis on technical system (a structural view), a shift from the conceptual system (a functional view). With this, all the components are made within the architecture extensible so that the design result of SSAD is easily transferred to that of CBSE. So the dominating methodology (CBSE) is naturally introduced to make the technical system substantial to accommodate extensible components [12] for operational system.

4.2 DFD-regulated Perspective Transitioning

Substitutable interoperation represents the ability of systems to provide interoperable services to and from the distributed and collaborative components. With high-level DFDs developed through SSAD, we have to consider the possibility that there are diverse types of data flow that is attached to service processes. To regulate DFD is to use substitutable interoperation mechanism to support diverse data flow communication between components.

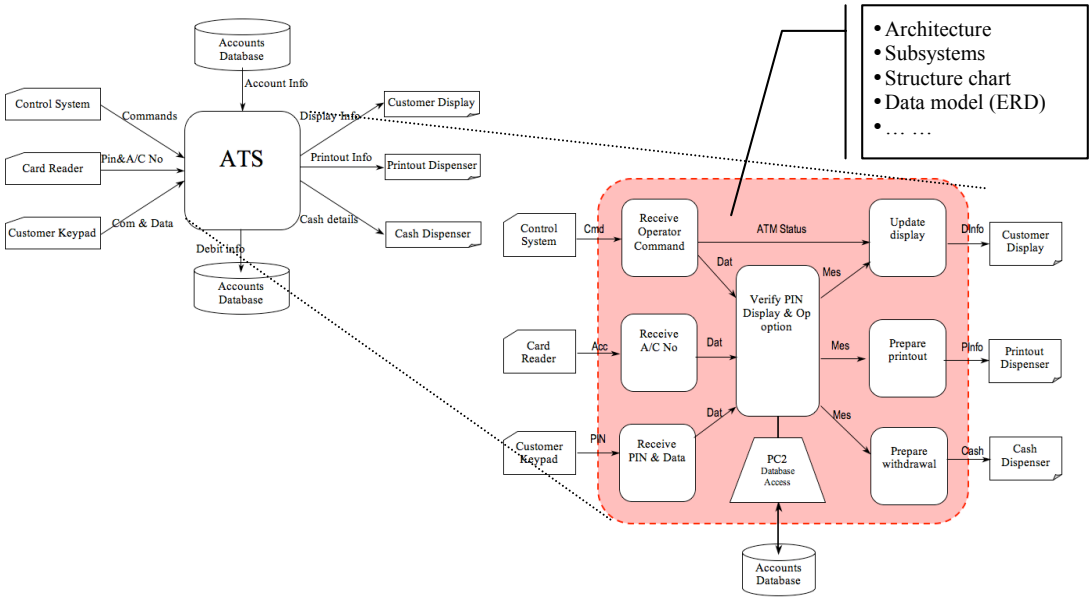

Figure 3. DFD-regulated Architecture via Substitutable Interoperation

The Automatic Teller System (ATS) is presented as a case study that provides the customers of a financial institution with access to financial transactions in a public space without the need for a human clerk or bank teller [26]. The simplified ATS is stated as follows:

- Project Requirements Documents (PRD) is to define the conceptual system with emphasis on 0th level DFD, based on which the prototype can be developed.
- Project Specification Document (PSD) is to refine the conceptual system with emphasis on the interior structure (1st level DFD), the extended version of conceptual system toward technical system .
- Software Design Document (SDD) is to design the technical system with emphasis on substitutable interoperation among components that constructs the architecture for operational system.
- Well Documented Source Code (WDSC) is written in re-ADA and runtime foundation with emphasis on interoperable components and synthetic collaboration.

Figure 3 illustrates document-driven perspective transitioning from PRD (0th level DFD) to PSD (1th level DFD). The interior structure is derived from the monolithic process (component) including and introducing following issues in analysis and design:

- System Architecture for interior structure (the highlighted part) derived from 0th level DFD
- Subsystem Overview for the decomposed components
- Structure Chart for organizing multiple components
- Data model via Entity-Relationship Diagram

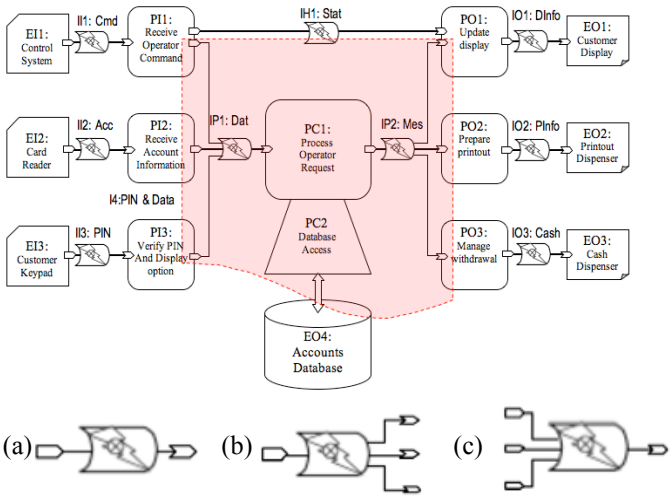

Figure. 4. Architectural Framework for ATS (1st DFD)

Figure 4 illustrates the application of substitutable interoperation in support of architecting the interior structure to a top-level architecture. After hiring substitutable interoperation mechanism, all the external devices, and interior components can be interconnected, with the support of re-ADA runtime foundation a prototype is therefore built to be able to execute. There are three types of substitutable interoperation are hired to architect interior structure: (a) pipeline, (b) client-server, and (c) event-driven mechanism, which regulates the interior structure (Fig. 3) into top-level architecture (Fig. 4).

Introducing software architecture to software design splits stakeholders' concerns into two categories: functionality and non-functional properties. The software architecture is the structure of the system, which comprises software components (functionality), the externally visible properties of those components (abstraction and information hiding), and the relationships between them, and constraints on both components and relationships (non-functional properties) [7]. The design principle of abstraction and information hiding (that separates the externally visible properties of the components from the implementation of the components) introduces CBSE to software design.

Table 2: Substitutable interoperation:

PI1, PI2, PI3 —IP1 → PC1 —IP2 → PO1, PO2, PO3

(a) CBSE through extensibility

```
with IP1_Collaborator_P; use  IP1_Collaborator_P;

package IP1_Client_P is

  class Client_Pub1 is new Publisher
     procedure Prepare is overridden;
  end Client_Pub1;
  class Client_Pub2 is new Publisher ... end;
  class Client_Pub3 is new Publisher ... end;

end IP1_Client_P;
...

with IP2_Collaborator_P;
use  IP2_Collaborator_P;
package IP2_Event_P is

  class Event_Sub1 is new Subscriber
     procedure Response is overridden;
  end Event_Sub1;
  class Event_Sub2 is new Subscriber ... end;
  class Event_Sub3 is new Subscriber ... end;

end IP2_Listener_P;
...

with IP1_Collaborator_P, IP2_Collaborator_P;
use              IP1_Collaborator_P,
IP2_Collaborator_P;
package IP1_IP2_P is

  class PC1_Pub_Sub is new
IP1_Collaborator_P.Subscriber,
        IP2_Collaborator_P.Publisher
     procedure Response is overridden;
     procedure Prepare  is overridden;
  end PC1_Pub_Sub;

end IP1_IP2_P;
...
```

(b) Synthetic and Executable Operational System

```
with IP1_Client_P;use IP1_Client_P;
with IP2_Event_P; use IP2_Event_P;
with IP1_IP2_P;   use IP1_IP2_P;
EXE_SYS:
declare
   PI1: aPublisher := new Client_Pub1;
        -- aPublisher stands for access type
   PI2: aPublisher := new Client_Pub2;
   PI3 : aPublisher := new Client_Pub3;
   COM_IP1   : IP1_Collaborator;

   PO1 : aSubscriber := new Event_Sub1;
   PO2 : aSubScriber := new Event_Sub2;
   PO3 : aSubscriber := new Event_Sub3;
   COM_IP2   : IP2_Collaborator;
   PC1  : PC1_Pub_Sub;  -- PC1 plays dual role of both Subscriber and Publisher via
multiply inheritance
begin
   COM_IP1.connect ( (PI1, PI2, PI3), PC1 );
     --* PI1, PI2, PI3 ==> PC1
   COM_IP2.connect ( PC1, (PO1, PO2, PO3) );
     --* PC1 ==> PO1, PO2, PO3
   loop
     delay 1;
   exit when SYS_Terminated;
     --:   PI1.Prepare(d)◊MET(met_amt);
     --:   PI2.Prepare(d)◊MET(met_amt);
     --:   PI3.Prepare(d)◊MET(met_amt);
     --: *⌈PI1.Deliver(d)◊LAT(lat_amt)   → PI1.Prepare(d)◊MET(met_amt);
     --:  [⌈ PI2.Deliver(d)◊LAT(lat_amt) → PI2.Prepare(d)◊MET(met_amt);
     --:  [  PI3.Deliver(d)◊LAT(lat_amt) → PI3.Prepare(d)◊MET(met_amt);
     --:  [] PC1.Observe(d)◊LAT(lat_amt) → PC1.Respond(d)◊MRT(mrt_amt);
     --: ⌋
           --: | |
     --:   PC1.Prepare(d)◊MET(met_amt);
     --: *⌈PC1.Deliver(d)◊LAT(lat_amt)   → PC1.Prepare(d)◊MET(met_amt);
     --:  [⌈PO1.Observe(d)◊LAT(lat_amt)  → PO1.Respond(d)◊MRT(mrt_amt);
     --:  [ PO2.Observe(d)◊LAT(lat_amt)  → PO2.Respond(d)◊MRT(mrt_amt);
     --:  [ PO3.Observe(d)◊LAT(lat_amt)  → PO3.Respond(d)◊MRT(mrt_amt);
     --: ⌋
   end loop;
   COM_IP1.disconnect;
   COM_IP2.disconnect;
end;
```

4.3 Executable Products across Systems Perspective

With the conceptual prototype that is derived from 1st level DFD and architected with substitutable interoperation in re-ADA, all the components are communicable with each other by playing different roles (publisher / subscriber). With the components to be extended / refined, the technical system is also executable. The evolution of systems perspective by means of components extension and substitution can reach the extent that the functional component is fulfilled the complete functionality, which leads to the operational system.

PADT supports all the products across systems perspective in such an evolutionary way that the intended system starts with SSAD, then turns to CBSE, and are executable. In order to architect the shadow of Figure 5, there essential substitutable interoperators in re-ADA [11,12] are formally described and shown in Table 2.

5. ACKNOWLEDGMENTS

An appreciation and understanding of software engineering concepts is best gained by participating in a real software engineering project. The selection, definition, specification, development, design, coding, implementation, documentation, and defense of a substantial software engineering product is the Capstone Project. The Capstone Project is a software project through which students learn software engineering principles and theory. They put them into practice by creating Problem Requirements Documents (PRD) and Product Specification Documents (PSD) that describe the objectives and expected outcomes of the project.

We are deeply grateful to Azusa Pacific University, where teaching software engineering becomes a new chapter in our academic career. Software engineering has developed us in the philosophical depth of recognizing the world. Over the past 20 years of working on software engineering, we have experienced the beauty of creation. Therefore, our faith and belief are strengthened not because of our creativity in software engineering, but because of our humility in awe of the Creation of the universe. So we owe an ultimate word of thanks to the Most High for letting us be talented for enjoying Software Engineering as part of our life.

6. CONCLUSION

SSAD is a straightforward and pragmatic Requirements Engineering method that can be well documented and educated [4,5], while CBSE favors adjusting requirements at later stages. PADT framework provides software engineering (education) with a new perspective transitioning view in such a pragmatic way that SSAD is streamlined through to CBSE -- engineering software systems is a documentation-driven perspective transitioning process.

One of the main drawbacks while applying PADT/re-ADA through Capstone Project is the loss of flexibility of drawing DFD. Based on the previous research accomplishments [11,12,27], further work focuses on the automated tool to be developed to architect the interior structure (described in DFD) with substitutable interoperation mechanism in re-Ada. The difficulty may be the intelligent regulation of diverse dataflow with the DFD, and the unification of data representation for dataflow. The DFD model within SSAD is able to represent the interior structure for the system, but may result in some ambiguity because of the lack of a mathematical foundation.

In conclusion, with emphasis on Perspectives from different stakeholders, Activity throughout software life cycle, Documentation for everything, and Transitioning from phase to phase, the PADT framework streamlines SSAD through to CBSE is t0 regulate DFD to a top-level architecture. The gap between SSAD and CBSE is bridged by re-ADA, so as to fulfill products executable throughout SDLC.

7. REFERENCES

[1] Wikipedia, Component-Based Software Engineering, http://en.wikipedia.org/wiki/Software_componentry

[2] What is CBD good for? CBD, frequently asked questions http://www.users.globalnet.co.uk/~rxv/CBDmain/cbdfaq.htm#benefits

[3] S. Schach, Object-Oriented Software Engineering, McGraw Hill Higher Education, 1st ed, 2008

[4] Wikipedia, SSADM, http://en.wikipedia.org/wiki/Structured_Systems_Analysis_and_Design_Methodology

[5] Heinrich HuBmann, Formal Foundations for Software Engineering Methods, LNCS 1322, Springer, 1997

[6] Capstone Project, http://www.apu.edu/caps/cismis/details/requirements/, Azusa Pacific University

[7] IEEE Standard Board, Recommended Practice for Architectural Description of Software-Intensive Systems (IEEE-std-1471 2000), September 2000

[8] DoD Joint Technical Architecture (JTA Version 4.0, 2002), http://www-jta.itsi.disa.mil/

[9] R. Pressman, Software Engineering: A Practitioner's Approach, McGraw-Hill S/E/M, 6 edition, 2004

[10] S. Liang L. Zhang, Luqi, Automatic Prototype Generating via an Optimized Object Model, Ada Letters, Vol. XXIII (2), June 2003

[11] S. Liang, L. Reibling, S. Sambasivam, "Automatic Prototype Generating" Restated with re-ADA, Proc. of SIGAda09, Nov. 1-5, 2009, Tampa Bay, FL.

[12] S. Liang, L. Reibling, J. Betts, re-ADA: Reliable Ada-based Descriptive Architecture for C4ISR via a Quantitative Interoperating Model, Proceedings of SIGAda'08, Oct 26-31, 2008, Portland, OR.

[13] Reibling, L.A., "Background Discussion on Mission Management Systems," *Trade Study Report to Boeing DARPA Unmanned Combat Air Vehicle ATD Program*, GDE Systems, Inc. Publication No. STS-98-0003A, 8 September 1998.

[14] Liang, X. Puett J. Luqi, Synthesizing Approach for Perspective-based Architecture Design, Proceedings of 14th IEEE International Workshop on Rapid Prototyping, June 9-11, 2003, San Diego, CA USA

[15] X. Liang, Software Documentation-Driven Manufacturing, Proc. of COMPSAC 2003, Nov.3-6, Dallas, TX

[16] S. Liang, J. Puett, Luqi, Quantifiable Software Architecture for Dependable Systems of Systems, LNCS: Architecting Dependable Systems II, Springer Verlag, 2004

[17] Luqi, Berzins, et.: A Prototyping Language for Real-Time Software, IEEE TSE, Vol. 14(10), Oct 1988

[18] Adaptable Process Model, R.S. Pressman & Associates, Inc. http://www.rspa.com/docs/

[19] http://www.practicalprocess.com/seyp/definition.html

[20] http://encyclopedia2.thefreedictionary.com/Software+engineering

[21] Diana Patterson, The challenges of documenting everything, ACM SIGDOC Asterisk Journal of Computer Documentation, Volume 7, Issue 2 (March 1981)

[22] http://www.businessdictionary.com/definition/process.html

[23] ISO/IEC 12207 Software Life Cycle Processes

[24] Wikipedia, Substitutability, http://en.wikipedia.org/wiki/Substitutability

[25] Wikipedia, Extensibility, http://en.wikipedia.org/wiki/Extensibility

[26] Automated teller machine, http://en.wikipedia.org/wiki/Automated_teller_machine

[27] Luqi, Computer-Aided Prototyping for a Command-and-Control System using CAPS IEEE Software, V.9 n.1, January 1992.