COMPARISON BETWEEN AGILE METHODOLOGY AND PLAN DRIVEN AS METHOD FOR ERP DEVELOPMENT

RUFMAN IMAN AKBAR

STT Pelita Bangsa
rufman_ia@yahoo.com.sg

Abstrak *: Metode Agile dan Plan Driven adalah dua methode dalam rekayasa perangkat lunak yang saling berbeda. Masing-masing memiliki tahapan dan proses yang berbeda mulai dari perencanaan, dokumentsi hingga implementasi aplikasinya. Penelitian ini mencoba membandingkan penggunaan metode Agile dengan metode Plan Driven dalam pengembangan perangkat lunak ERP (Enterprise Resources Planning). Perbandingan dilakukan secara kualitatif untuk melihat perbedaan yang terjadi dalam penggunaan ke dua metode tersebut. Penelitian dilakukan dalam kerangka Rekayasa Perangkat Lunak Empiris – berdasarkan catatan yang ada dalam proyek pengembangan ERP yang dilakukan oleh team pengembang dari satu rumah produksi terhadap dua proyek yang berbeda. Dalam penelitian ditemukan bahwa masing-masing metode memiliki kelebihan dan kelemahan – berdasarkan variabel yang diamati. Secara keseluruhan, metode Agile memiliki keunggulan di banding Plan Driven dalam pengembangan ERP.*

Kata kunci: *Metode Pengembangan Perangkat Lunak, Metode Agile, Plan Driven, ERP, Rekayasa Perangkat Lunak Empiris*

1. INTRODUCTION

Main purposes of systems analysis and design, as performed by systems analysts and designers, seeks to understand what users need to analyze data input or data flow systematically, data to process or transform, data to store, and output information in relation to a particular organization or enterprise. By doing thorough analysis, system analysts and designers seek to identify and solve the right problems. Furthermore, systems analysis and design is used to analyze, design, and implement improvements in the support of users and the functioning of businesses that can be accomplished through the use of computerized information systems (Kendall & Kendall, 2011).

Developing a system without proper planning tends to leads to great user dissatisfaction and frequently causes the system to fall into disuse. Process of systems analysis and design lends structure to the analysis and design of information systems, a costly endeavor that might otherwise have been done in a haphazard way. It can be thought of as a series of processes systematically undertaken to improve a business through the use of computerized information systems.

Users' involvement throughout the systems project is critical factor to the successful development of application software. Systems analysts and designers are the other essential component in developing useful information systems. Users are moving to the forefront as software development teams in their capacity. This means that there is more emphasis on working with software users; on performing analysis of their business, problems, and objectives; and on communicating the analysis and design of the planned system to all involved

It is surprising how few IT organizations utilize a formal methodology. Although the vast majority employ a wide variety of automated tool sets to assist their programmers in developing and testing complex code, the "process" of systems

development is still largely chaotic in most organizations. A systems methodology guides the activities of developing and evolving systems starting from the initial feasibility study and culminating only when the system is finally retired. Use of a methodology assures the organization that its process of developing and maintaining systems is sustainable and repeatable (Keyes, 2005).

Choice would be simple if there were only one methodology. Unfortunately, or can be fortunately, we need to choose from so many kind of methodologies. Some are industry standard when the others are proprietary to a particular consulting organization. Given this vast choice, it is important that we are able to determine whether a systems methodology will meet the specific needs of our organization. The way to do this is by evaluating the methodology.

Since software systems are abstract and intangible, they are not constrained by the properties of materials, governed by physical laws, or by manufacturing processes. This simplifies software engineering, as there are no natural limits to the potential of software. However, because of the lack of physical constraints, software systems can quickly become extremely complex, difficult to understand, and expensive to change (Sommerville, 2011).

Developing an organizational information system is completely different from developing a controller for a scientific instrument. Both of these applications need software development methods; they do not all need the same software engineering techniques. There are still many reports of software projects going wrong and ‘software failures’. Software engineering is criticized as inadequate for modern software development. Some of these so-called software failures are a consequence of two factors (Sommerville, 2011):

The first factor is increasing in demands. New software development method can help us to build larger, more complex systems, but the demands still can change due to change in business process. Systems have to be built and delivered more quickly; larger, even more complex systems are required; systems have to have new capabilities that were previously thought to be impossible or not included in the systems. Existing software engineering methods cannot cope and new software engineering techniques have to be developed to meet new these new demands.

The second factor is about low expectations. It is relatively easy to write computer programs without using software engineering methods and techniques – or using Cowboy programming techniques. Many companies have drifted into software development as their products and services have evolved. They do not use software engineering methods in their everyday work. Consequently, their software is often more expensive and less reliable than it should be. So, we need better software engineering education and training to address this problem. We also need to have better matching method with the requirement and environment condition.

Software engineering is an intensely people-oriented activity, yet little is known about how software engineers perform their work. In order to improve software engineering tools and practice, it is therefore essential to conduct field studies, i.e., to study real practitioners as they solve real problems. To aid this goal, we conduct study based on the real project on a software house that often handle ERP custom development project. We choose one project that use Extreme Programming technique (as one kind of Agile method) and the other use System Development Life Cycle (SDLC) technique to represent Plan Driven. We use SDLC from Shelly and Rosenbelt, since there is several kind of SDLC format. For each technique, we collect documents

from real practice, gather an analysis of some of its advantages and disadvantages from team members opinion.

We choose ERP system development in this study because of size of the system, resources, and time period needed in the project. It is not a short time development life cycle, but also not so long development period. To have apple to apple comparison, we choose two projects with mostly the same size, resources and time needed to finish the project. Both projects also apply to same industry (in this case – manufacture).

2. BASIC THEORY

Software process model is a simplified representation of a software process. Each process model represents a process from a particular perspective, and thus provides only partial information about that process. For example, a process activity model shows the activities and their sequence but may not show the roles of the people involved in these activities (Sommerville, 2011).

These generic models are not definitive descriptions of software processes. Rather, they are abstractions of the process that can be used to explain different approaches to software development. We can think of them as process frameworks that may be extended and adapted to create more specific software engineering processes.

The first method to evaluate is SDLC model. This takes the fundamental process activities of specification, development, validation, and evolution and represents them as separate process phases such as requirements specification, software design, implementation, testing, and so on. The first published model of the software development process was derived from more general system engineering processes. This model is illustrated in Figure 1. Because of the cascade from one phase to another, this model is known as the 'waterfall model' or software life cycle (Royce, 1970). The waterfall model is an example of a plan-driven process—in principle, we must plan and schedule all of the process activities before starting work on them.

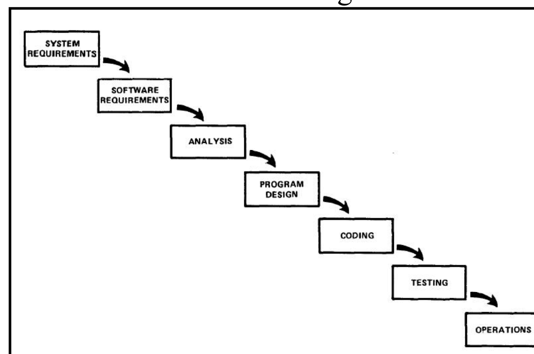


Figure 1. Implementation Step To Develop Large Computer Program – Waterfall Model

In practice, iterative interaction between the various phases is confined to successive steps (see Fig. 2). Ordering of steps is based on the following concept: that as each step progresses and the design is further detailed; there is iteration with the preceding and succeeding steps but rarely with the more remote steps in the sequence. The virtue of all of this is that as the design precedes the change process is scoped down to manageable limits. At any point in the design process after the requirements analysis is completed there exists a firm and close up, moving baseline to which to return in the event of unforeseen design difficulties. What we have is an effective fallback position

that tends to maximize the extent of early work that is salvageable and preserved (Royce, 1970).

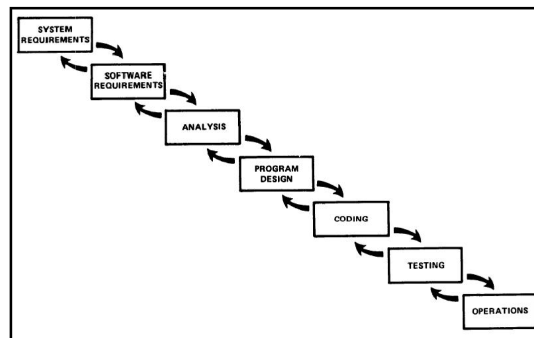


Figure 2. Iterative Interaction between Phases

Figure 1 and 2 shows the very basic model of Waterfall technique developed by Winston Royce in 1970. From time to time, this model had been evolved to form new Waterfall model or Neo Waterfall (SDLC). Easy to understand and time-tested are the main reason of this model still exist and evolved until current time. Figure 3 show one kind of Neo Waterfall model (Shelly & Rosenblatt, 2012). This model also known as Two-ways Waterfall models with emphasis on iteration and user input.

The SDLC model usually includes five steps, which are described in the following paragraphs: systems planning, systems analysis, systems design, systems implementation, and systems support and security (Shelly & Rosenblatt, 2012).

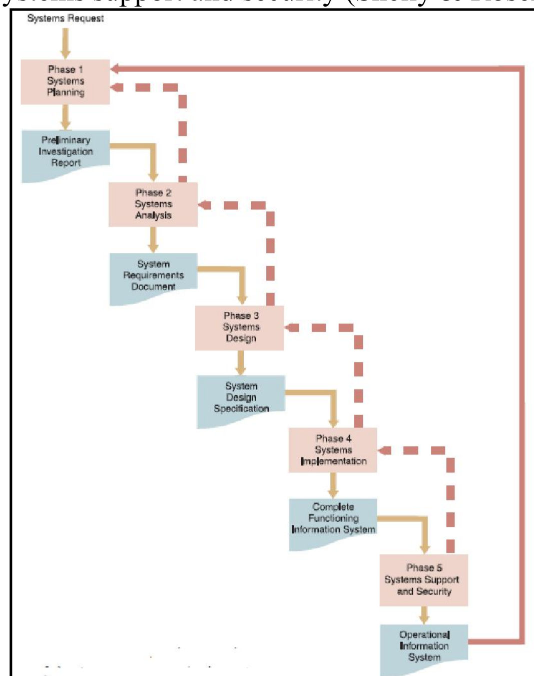


Figure 3. SDLC Model from Shelly and Rosenblatt

SYSTEMS PLANNING. The systems planning phase usually begins with a formal request to the IT department, called a systems request, which describes problems or desired changes in an information system or a business process. In many companies, IT

systems planning is an integral part of overall business planning. When managers and users develop their business plans, they usually include IT requirements that generate systems requests. A systems request can come from a top manager, a planning team, a department head, or the IT department itself. The purpose of this phase is to perform a preliminary investigation to evaluate an IT-related business opportunity or problem. The preliminary investigation is a critical step because the outcome will affect the entire development process.

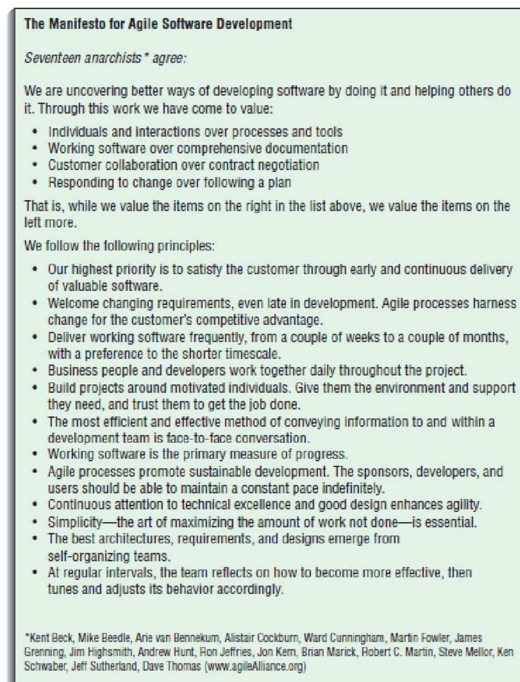
SYSTEMS ANALYSIS. The purpose of the systems analysis phase is to build a logical model of the new system. The first step is requirements modeling, where you investigate business processes and document what the new system must do to satisfy users. Requirements modeling continues the investigation that began during the systems planning phase. To understand the system, we perform fact-finding using techniques such as interviews, surveys, document review, observation, and sampling. Then we use the factfinding results to build business models, data and process models, and object models. The deliverable for the systems analysis phase is the system requirements document. The system requirements document describes management and user requirements, costs and benefits, and outlines alternative development strategies.

SYSTEMS DESIGN. The purpose of the systems design phase is to create a physical model that will satisfy all documented requirements for the system. At this stage, we design the user interface and identify necessary outputs, inputs, and processes. We also design internal and external controls, including computer-based and manual features to guarantee that the system will be reliable, accurate, maintainable, and secure. During the systems design phase, we need to determine the application architecture, which programmers will use to transform the logical design into program modules and code. The deliverable for this phase is the system design specification, which is presented to management and users for review and approval. Management and user involvement is critical to avoid any misunderstanding about what the new system will do, how it will do it, and what it will cost.

SYSTEMS IMPLEMENTATION. During the systems implementation phase, the new system is constructed. Whether the developers use structured analysis or O-O methods, the procedure is the same — programs are written, tested, and documented, and the system is installed. The objective of the systems implementation phase is to deliver a completely functioning and documented information system. At the conclusion of this phase, the system is ready for use. Final preparations include converting data to the new system's files, training users, and performing the actual transition to the new system. The systems implementation phase also includes an assessment, called a systems evaluation, to determine whether the system operates properly and if costs and benefits are within expectations.

SYSTEMS SUPPORT AND SECURITY. During the systems support and security phase, the team development maintains, enhances, and protects the system. Maintenance changes correct errors and adapt to changes in the environment. Enhancements provide new features and benefits. The objective during this phase is to maximize return on the IT investment. Security controls safeguard the system from both external and internal threats. System must be secure, reliable, maintainable, and scalable. A scalable design can expand to meet new business requirements and volumes. Information systems development is always a work in progress. Since business processes change rapidly, most information systems need to be updated significantly or replaced after several years of operation.

Other method to be used in this comparison is Extreme programming, which are a part of Agile Method. Many different individual methodologies come under the umbrella of Agile Methodologies, including the Crystal family of methodologies, Adaptive Software Development, Scrum, Feature Driven Development, and eXtreme Programming. In February 2001, many of the proponents of these alternative approaches to systems analysis and design met in Utah in the United States to reach a consensus on many of the underlying principles their various approaches contained. This consensus turned into a document they called "The Agile Manifesto" (Figure 4). The Agile Methodologies share three key principles: (1) a focus on adaptive rather than predictive methodologies, (2) a focus on people rather than roles, and (3) a self-adaptive process (Valacich, et.al., 2012).



The Manifesto for Agile Software Development

Seventeen anarchists agree:*

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

That is, while we value the items on the right in the list above, we value the items on the left more.

We follow the following principles:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity—the art of maximizing the amount of work not done—is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

*Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas (www.agileAlliance.org)

Figure 4. The Manifesto for Agile Software Development

The Agile Methodologies group argues that software development methodologies adapted from engineering generally do not fit well with the reality of developing software. In the engineering disciplines, such as civil engineering, requirements tend to be well understood. Once the creative and difficult work of design is completed, construction becomes more predictable. One of the best-known and most written-about Agile Methodologies is called Extreme Programming. Developed by Kent Beck in the late 1990s, extreme Programming illustrates many of the central philosophies of this new approach to systems development. We use extreme Programming as an example of the central ideas common to many Agile methods.

Extreme Programming (XP) was conceived and developed to address the specific needs of software development conducted by small teams in the face of vague and changing requirements. This new lightweight methodology challenges many conventional tenets, including the long-held assumption that the cost of changing a piece of software necessarily rises dramatically over the course of time. XP recognizes

that projects have to work to achieve this reduction in cost and exploit the savings once they have been earned (Beck, 1999). Fundamentals of XP include:

1. Distinguishing between the decisions to be made by business interests and those to be made by project stakeholders.
2. Writing unit tests before programming and keeping all of the tests running at all times.
3. Integrating and testing the whole system--several times a day.
4. Producing all software in pairs, two programmers at one screen.
5. Starting projects with a simple design that constantly evolves to add needed flexibility and remove unneeded complexity.
6. Putting a minimal system into production quickly and growing it in whatever directions prove most valuable.

Beck defines a set of five values that establish a foundation for all work performed as part of XP—communication, simplicity, feedback, courage, and respect. Each of these values is used as a driver for specific XP activities, actions, and tasks (Beck, 1999).

In order to achieve effective communication between software engineers and other stakeholders (e.g., to establish required features and functions for the software), XP emphasizes close, yet informal (verbal) collaboration between customers and developers, the establishment of effective metaphors for communicating important concepts, continuous feedback, and the avoidance of voluminous documentation as a communication medium (Pressman, 2010).

To achieve simplicity, XP restricts developers to design only for immediate needs, rather than consider future needs. The intent is to create a simple design that can be easily implemented in code). If the design must be improved, it can be refectories at a later time.

Feedback is derived from three sources: the implemented software itself, the customer, and other software team members. By designing and implementing an effective testing strategy, the software (via test results) provides the agile team with feedback. XP makes use of the unit test as its primary testing tactic. As each class is developed, the team develops a unit test to exercise each operation according to its specified functionality. As an increment is delivered to a customer, the user stories or use cases that are implemented by the increment are used as a basis for acceptance tests. The degree to which the software implements the output, function, and behavior of the use case is a form of feedback. Finally, as new requirements are derived as part of iterative planning, the team provides the customer with rapid feedback regarding cost and schedule impact.

Extreme programming applies four rules in developing the software project:

1. Communication. The programmer must communicate with the customer and elicit his requirements, thus the emphasis on customer satisfaction. The programmer also needs to communicate with fellow workers, thus the emphasis on team work.
2. Simplicity. The design is maintained as simply as possible.
3. Feedback. The software is tested from its early stages, feedback is obtained, and changes are made. This is a cyclical process.
4. Courage. The programmer can make changes even at the last stages and implement new technologies as and when they are introduced.

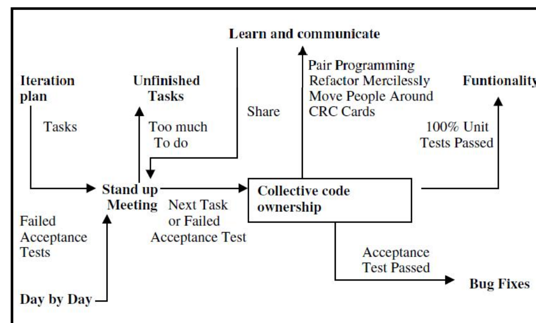


Figure 5. Extreme Prog. Process Of Software Dev

The components of extreme programming include: a) User stories are written by the customer and describe the requirements of a system. The customer need not specify his requirements using any particular format or technical language; he merely writes these in his own words. Aside from describing what the system must be, the user stories are used to calculate the time estimates for release planning. At the time of the implementation of the user stories the developer obtains detailed information from the customer. The time estimate is usually in the form of ideal development time — defined as how long it would take to implement the story in code if there were no distractions, no other assignments, and the programmer knew exactly what to do. Typically, each story will get one to three weeks. The user stories are also used to produce test scenarios for acceptance testing by the customer as well as to verify that the user stories have been implemented correctly; and b) Release planning produces the release plan followed during development of the system; it is also called the "planning game." During release planning a meeting is set up with the customers and the development team. During this meeting a set of rules is set up by the customers and developers to which all agree. A schedule is then prepared. A development team is selected to calculate each user story in terms of ideal programming weeks, which is how long it would take to implement that story if absolutely nothing else needed to be done.

3. MATERIAL AND METHOD

Since Software engineering is an intensely people-oriented activity, yet little is known about how software engineers perform their work. In order to improve software engineering tools and practice, it is therefore essential to conduct field studies, i.e., to study real practitioners as they solve real problems (Forest, 2008).

This study combined Direct Technique and Independent Technique as part of empirical software engineering research. For Direct Technique, we use Participants Observation to get deep understanding, goals and rational for action, time spent or frequency over long periods. For Independent Technique, we use Documentation Analysis to get design and documentation practices. We also use Dynamic and static Analysis to get understanding about Design and Programming Practices.

Visual Basic.net 2003 and MS SQL Server 2000 are programming language and data provider used in both project. For the design tools, team use Microsoft Office (include Visio) as standard tools. Both systems run under Windows XP on client side, developer side, and Windows Server 2003 for Server Operating System. Total member of team development is about 20, which contain of 14 programmer/system analysts and designer, and 6 users key man of the system.

The sub system for each ERP systems includes; a) Management system module; b) Sales; c) PPIC; d) Material Warehouse; e) Shop Floor/Work shop; f) QC; g) Finish Goods Warehouse; h) Delivery System; i) Purchasing; and j) AR Function.

Factors to compare based on each technique of software development are;

a. Resources Needed : Man Month for programming (A.1); and Software and Hardware (A.2)
b. Efforts: Meeting with Users (qty) (B.1); Man month for system design (B.2); and Testing with users (B.3)
c. Documentation : Design documentation (C.1); Testing documentation (C.2); and System documentation (C.3)
d. Error : Logical Error (D.1); and Runtime Error (D.2)
e. Aggregate: Total time schedule (E.1); and Rate of user's dissatisfaction (E.2)

Man Month for programming will show time needed for programming time – this factor include routine test, unit test, and integration test. Software and hardware will be counted based on concurrent usage – on average basis.

Meeting with users will reflect number of meeting – formal meeting, on design and evaluation phase. Man month of system design will show effort of system designer on design phase to translate idea of solution to design of application. Testing with users will show number of event to make test together with users – in formal time.

Design Documentation will show type of design and total number of page of design document. Testing documentation will shown number of document for testing, including test plan, scenario, result, and test report. System Documentation will record documentation of running system – including user manual and running manual.

Logical Error will show number of error based on misinterpretation between designer – user, programmer – designer, and user programmer. Runtime error will record number of error on running system – after production phase.

Total Time Schedule will show day needed to finish system – starting from Kick Off Meeting until hand over system. Rate of Users Satisfaction will show evaluation from users based on some variabel on system development and run time phase.

All factors will be indexed based on FGD before compared. Based on total index, we will calculate weight of each factor. Values of each weight will be used to multiply the value of each related factor.

Table 1. Indexing and Weighted

No	Factors	Index	Weight
1	A1	-	-
2	A2	-	-
3	B1	-	-
4	B2	-	-
5	B3	-	-

Index = from FGD

Weight = index/Sigma of Index

Table 2. Comparison Factors

No	Factors	W/F	XP
1	A1	-	-
2	A2	-	-
3	B1	-	-
4	B2	-	-
5	B3	-	-

Index SDLC and XP = real value * wight

4. RESULT AND DISCUSSION

Based on data from document and FGD, we found the result as follows;

Table 3. Indexing and Weighted

No	Factors	Index	Weight
1	A1	8	0.11
2	A2	6	0.08
3	B1	6	0.08
4	B2	4	0.05
5	B3	4	0.05
6	C1	5	0.07
7	C2	5	0.07
8	C3	6	0.08
9	D1	7	0.09
10	D2	7	0.09
11	E1	9	0.12
12	E2	8	0.11

Table 4. Indexing and Weighted

No	Factors	SDLC	XP
1	A1	6.4	5.33
2	A2	0.64	0.72
3	B1	3.36	2.28
4	B2	0.16	0.13
5	B3	1.92	1.39
6	C1	5.33	2.27
7	C2	2.33	5.22
8	C3	17.6	16.00
9	D1	0.65	0.19
10	D2	0.19	0.09
11	E1	43.2	36
12	E2	0.21	0.32
		82.00	70.65

Total index for Extreme Programming lower than SDLC, it mean based on data comparison XP still better than SDLC. Except for A2 and E2, we found that index for SDLC lower than XP.

The Result just based on comparison from two projects on ERP in Manufacture Company. In the next – if possible, better to have more case for comparison. Also for other business type, need to expand i.e. on trading or services company. For index, also still can be expand to cover more detail aspect.

REFERENCES

Kendall., Kenneth; Kendall, Julie, 2011, *Systems Analysis and Design* , Prentice Hall.

Keyes, Jessica, 2005, *Software Engineering Handbook*, Auerbach Publications.

Sommerville, Ian, 2011, *Software Engineering* 9th Edition, pearson Education.

Royce, Winston., 1970, Managing The Development of Large Software System*, Proceeding IEEE Wescon*, 1970.

Shelly, Gery; Rosenblatt, Harry., 2012, *System Analyst and DesignApplication*. Course Technology.

Pressman, Roger, 2010, *Software Engineering,* 7th Edition, McGraw Hill.

Valacich, Joseph; Georgr, Joey; Hoffer, Jeffrey, 2012, *System Analyst and Design* . Pearson.

Beck, Kent, 1999, *Extreme Programming Explainde* . Embrace3 Change.

Forest,Shull et.al, 2008, *Guide to Advanced Empirical Software Engineering* . Springer