Software Process Models with CoCoMo techniques, and CASE tools in the context of Software engineering

Ali Tariq Bhatti[1]

[1]Department of Electrical and Computer engineering
[1]North Carolina A&T State University, Greensboro NC USA
[1]atbhatti@aggies.ncat.edu, ali_tariq302@hotmail.com, alitariq.researcher.engineer@gmail.com

***Abstract*— Clearly, a program to solve a problem and a Programming System Product to solve the same problem are two entirely different things. Obviously much more efforts and resources are required for a Programming System Product. As a rule of thumb, a programming system product costs approximately ten times as much as a corresponding program. The software industry is largely interested in developing Programming System Products and most commercial software system or packages fall in this category. Software engineering is largely concerned with the Programming System Product. In *this paper, it also aims to introduce and explain the basic CoCoMo process for software estimating. It is hoped to provide auditors, software engineers and project managers with an insight to a stable method for estimating time, cost and staff. The basic model distinguishes between three different development modes Organic, Semi-detached, and Embedded. On the other hand,* Computer Aided Software (Systems) Engineering (CASE) appears to have the potential to improve software development productivity, reduce software maintenance costs, and enhance overall product quality. Therefore, different software process models have been used for software design, validation, and testing.**

Keywords—component; Software engineering, Software processes, CoCoMo, Waterfall, CASE.

1. Software and Software Engineering

Software Engineering has to deal with a different set of problems than other engineering disciplines, since the nature of software is different. A software product is entirely conceptual entity; it has no physical or electrical properties like weight, color or voltage. Consequently, there are no physical or electrical laws to govern software engineering. In fact, one of the goals of research in software engineering is to form general laws that are applicable to software.

Software failures are also different from failures of mechanical or electrical systems. Products of these other engineering disciplines fail because of the change in the physical or electrical properties of the system caused by aging. A software product, on the other hand never "wears out" due to age. In software, failures occur due to faults in the conceptual process. Software fails because the design fails. In general, when a design fault is detected in software, changes are usually made to remove that fault so that it causes no failures in the future. Due to this, and other reasons, software must constantly undergo changes, which makes maintenance, an important issue with software. With this background we must define the software and software engineering.

SOFTWARE: is a collection of computer programs, procedures, rules, and associated documentation and data.

SOFTWARE ENGINEERING:

i) It is the systematic approach to the development, operation, maintenance, and retirement of the software.

ii) It is the application of science and mathematics by which the capabilities of computer equipments are made useful to man via computer programs, procedures, and associated documentation.

The use of the terms "*systematic approach*" or "*Science and mathematics*" for the development of software means that software engineering is to provide methodologies for developing software that are close to the scientific methods as possible.

The phrase "*usable to man*" emphasizes the needs of the user and the software's interface with the user. This definition implies that user needs should be given due importance in the development of software, and the final program should given importance to the user interface.

The basic goal of software engineering is to produce high quality software at low cost. The two basic driving factors are quality and cost. Cost of a complete project can be calculated easily if proper accounting procedures are followed. Quality of software is something not so easy to quantify and measure.

There are a number of factors that determine software quality. One can combine these different factors into "Quality metric", but the relative weights of these different factors will depend on the overall objectives of the project. Taking a broad view of software quality we can specify three dimensions of the product whose quality is to be assessed:

(i)Product Operations

(ii)Product Transition

(iii)Product Revision

The first factor of “Product Operation” deals with quality factors such as correctness, reliability, efficiency, Integrity, and usability. The second factor “Product Transition” deals with quality factors like portability, reusability, and interoperability. The “Product Revision” concerned with those aspects related to modification of programs and includes factors like maintainability and testability. These three dimensions with there factors are shown as under:

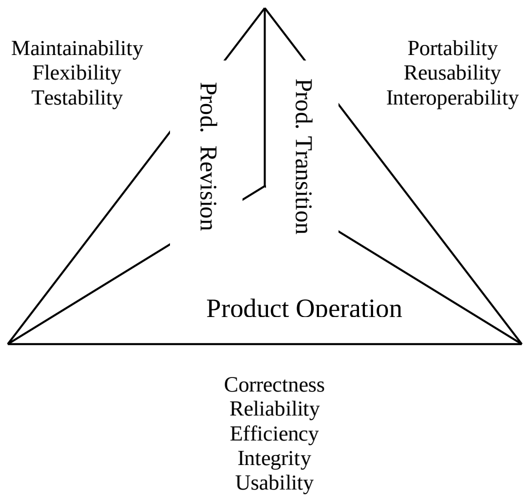

Figure 1: Factors of Software Development

Correctness is the extent to which a program satisfies its specifications. ***Reliability*** is the property which defines how well the software meets its requirements. ***Efficiency*** is a factor in all issues relating to the execution of software and includes such considerations as response time, memory requirement, and throughput. ***Integrity*** is the ability to ensure that information is not modified except by the person who is explicitly intended to modify it. ***Usability***, or the efforts required to learn and operate the software properly that emphasizes the human aspect of the system.

Maintainability is the effort required to locate and fix errors in operating programs. ***Flexibility*** is the effort required to modify an operational program (perhaps to enhance its functionality). ***Testability*** is the effort required to test to ensure that the system or a module perform its intended function properly.

Portability is the effort required to transfer the software from one hardware configuration to another. ***Reusability*** is the extent to which parts of the software can be reused in other related applications. ***Interoperability*** is the effort required to couple the system with other systems. Enabling different systems to work together and exchange data. Interoperability between different systems is achieved by using common standards and specifications. Examples of e-learning interoperability: Passing information about a student and their educational qualifications from a Student Record System in one college to a Student Record System in another college.

2. Software project Decomposion and Milestones.

The project is initially divided into the following functional blocks. Significant administrative overhead, costly for small teams and projects [2].

3. Requirement gathering.
4. Input Design and Modeling.
5. Entities modeling and their relationships
6. Database design
7. Data Security requirements
8. Team Responsibilities and their tasks.
9. Development of Interface for Foreground and Backend database.
10. Output Design
11. Maintenance requirements.

3. Software Development Methodology

Often, the customer defines a set of general objectives for software but does not identified detail input, processing, or output requirements. In these cases, and many other situations, a ***Prototyping Paradigm*** may offer the best approach. The prototyping paradigm begins with requirements gathering. Developer and Customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A “quick design” then occurs. The quick design focuses on a representation of those aspects of the software that will be visible to the customer/user (e.g., input approaches and output formats). The quick design leads to the construction of a prototype. So, we decided to adopt Prototyping methodology for the development of this software, so that customer sees what appears to be a working version of the software. The successful development of a modern day Information System (IS) which involves the writing of *software units* (sometimes referred to as modules) demands much more than a modern block structured programming language and a powerful operating system.

4. Software units

The three questions that a planner or auditor needs to know about a prospective software unit are:-

(i)How long will it take?

(ii)How much will it cost?

(iii)How many people will it need?

5. Generic Software Process Models

There are many variants of these models e.g. formal development where a waterfall-like process is used, but the specification is formal that is refined through several stages to an implementable design [1]. There are four generic software process models:

(i)The waterfall model: Separate and distinct phases of specification and development

(ii)Evolutionary development: Specification and development are interleaved

(iii)Formal systems development (example - ASML): A mathematical system model is formally transformed to an implementation

(iv)Reuse-based development: The system is assembled from existing components

5.1 Waterfall Model

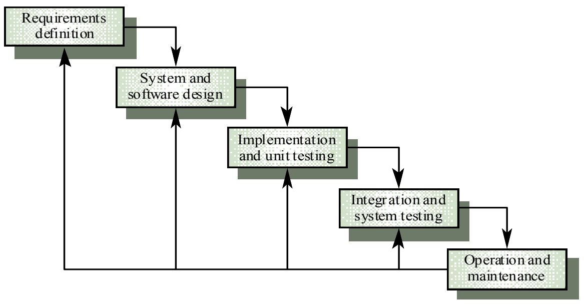
Requirements definition
System and software design
Implementation and unit testing
Integration and system testing
Operation and maintenance

Figure 2: Waterfall Model[3]

The drawback of the waterfall model is the difficulty of accommodating change after the process is underway.

5.1.1 Waterfall Model problems:

• Inflexible partitioning of the project into distinct stages

• This makes it difficult to respond to changing customer requirements

• Therefore, this model is only appropriate when the requirements are well-understood

• Waterfall model describes a process of stepwise refinement

However, it uses are

- Based on hardware engineering models
- Widely used in military and aerospace industries.

Although the waterfall model has its weaknesses, as it is instructive because it emphasizes important stages of project development. Even if one does not apply this model, he must consider each of these stages and its relationship to his own project [4]

5.2 Evolutionary development

- Exploratory development

Objective is to work with customers and to evolve a final system from an initial outline specification.

Should start with well-understood requirements.

The system evolves by adding new features as they are proposed by customer.

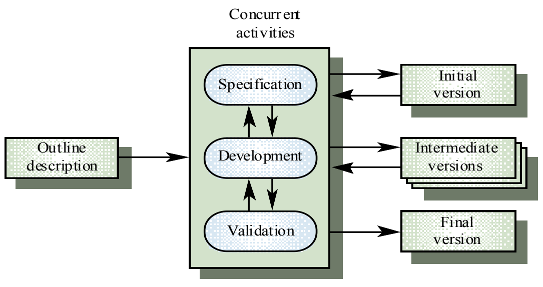

Figure 3: Evolutionary Development

5.3 Formal systems development

- Based on the transformation of a mathematical specification through different representations to an executable program.
- Transformations are 'correctness-preserving' so it is straightforward to show that the program conforms to its specification
- Embodied in the 'Cleanroom' approach (which was originally developed by IBM) to software development

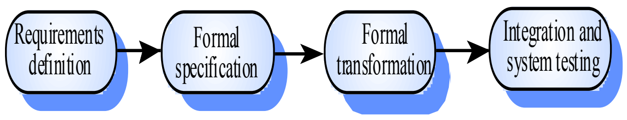

Figure 4: Formal systems development

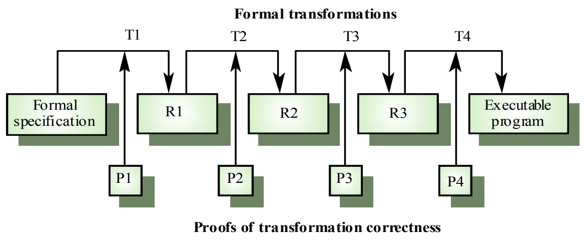

Figure 5: Formal transformations

5.4. Reuse-oriented development

• Based on systematic reuse where systems are integrated from existing components or COTS (Commercial-off-the-shelf) systems

• Process stages

• Component analysis

• Requirements modification

• System design with reuse

• Development and integration

This approach is becoming more important but still limited experience with it

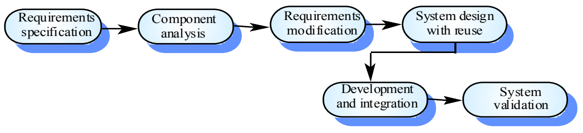
Requirements specification
Component analysis
Requirements modification
System design with reuse
Development and integration
System validation

Figure 6: Reuse -oriented development

6. CoCoMo in Software engineering

***C**onstructive* ***C**ost* ***M**odel* (CoCoMo) is an algorithmic software estimation model. The fundamental concept is that *the amount of effort required in writing a software unit will depend of the size of that unit.* The relationship is not linear i.e. a unit twice as long does not take up exactly twice the effort. The general idea is that the EFFORT required by a team of programmers to write a software unit is measured in persons and months i.e. we say a unit will take 10 person months. This is 10 people all working for 1 month, or 2 people working for 5 months or 1 person working for 10 months etc. There are obvious limitations on this. For example if a unit requires 200 person months, it is impractical to have 1 person working for 200 months which is 16 years and 8 months, similarly there is an immense problem managing 200 people working for 1 month.

6.1 CoCoMo techniques

(i)Effort

The effort required is measured in person months. We stress that the units of measurement are persons *times* months and that we are estimating at this stage "manpower effort". This ***EFFORT*** is proportional to the ***SIZE*** of the software unit measured in 1000's of lines of code; each line representing one source instruction.

The generally accepted form is:

- **EFFORT is measured in Person Months = PM**
- **SIZE is measured in = KDSI**
- **Thousands (K) of Deliverable Source Instructions (KDSI).**

Therefore the concept can be stated as

EFFORT (in Person Months) is proportional to MODULE SIZE in (KDSI)

PM $\propto$ (KDSI)

The original research showed that the model requires the (KDSI) to be raised to a power and the whole of the right hand side to be multiplied by a coefficient.

PM = A x $(\text{KDSI})^B$

This gives a first estimate of the manpower effort required to write a software module. We will see that the numbers A and B are given to us depending on further information to do with the complexity of the software itself. The important step here is that once we know the size of the software unit and can find A& B, then we can calculate an estimate of person months of effort. This will involve the use of a calculator, but we will keep the arithmetic as straight forward as possible.

(ii)Development Time

The next step is to estimate the Total Development Time. As this name implies, this is the time taken to develop a software unit from beginning to end. The **T**otal **D**evelopment time is known as TDEV and is usually measured in MONTHS. It may appear that once we have estimated the EFFORT as so-many-person-months, we could divide this figure by the number of programmers available and work out the time. So, taking a more scientific approach we say that:

Total Development time TDEV is based on Effort as follows:-

TDEV = 2.5 $(\text{PM})^C$

where the power C is also given to us depending on the software module complexity.

(iii) Development Modes

There are three basic modes of development ranging from simplest to most complex with a middle ground. These have become known as:-

(a)**Simplest** - Organic Mode

(b)**Middle Ground** - Semi-Detached Mode

(c)**Most Complex** - Embedded Mode

There are no crystal clear cut boundaries between these 3 modes of development and an experienced Project Manager will need to use his judgment when deciding which mode a new unit of software will fall into.

As a general rule the following:-

Organic Mode: Fairly simple construction, small KDSI, small team of programmers who all know the ropes.

Semi-Detached Mode: The middle ground.

Embedded Mode: Complex software, high KDSI upwards of 150 KDSI, large team with associated co-ordination problems.

(iv)Cost

Finally in this section, let us take our first look at how much a software unit will cost. We now have a rough idea of the average number of staff required and the time in months this number of people will be working. If we coupled this with the cost per month of the average staff member we would have a first estimate of staff

development costs.

Average Number of Staff x Development Time in Months x Cost per Programmer per Month.

The Reader should notice that Average Number of Staff was obtained from:

Average Number of Staff = PM/TDEV

So, Average Number of Staff x Development **Time = PM /TDEVx TDEV = PM**

In other words

Staff Costs = PM x Cost per Programmer per Month

This is sensible when we think that PM = Persons x Months and when this is multiplied by Cost per Person per Month. This will give the Cost of the software unit.

So, if we couple the development EFFORT with the monthly cost of a programmer we could estimate the total cost of this software unit.

The number of person months of EFFORT x Cost per person per month = Total cost estimate for a unit of this size.

(v)FURTHER REFINEMENTS

So far our estimates have been of a global nature and dealt only with overall Effort and overall Development Time, treating the whole software development as a single "black box" unit. The next step is to consider the software development in more detail with thought to the actual phases involved. Once a specification is received (from the Analyst/Designer for traditional life cycle development, or as a technical specification derived by the Analyst/Programmer and User as part of a prototype, or whatever other authority) there are 3 main phases to consider in turning this authorized requirement program specification into the deliverable product of a software unit. The overall software unit is now considered as being comprised of 3 main phases.

These 3 main phases are:

i. Product Design

ii. Programming

iii. Integration and Testing

Of the overall Effort calculated from the basic CoCoMo formula we now consider what % of this overall effort will be spent by the professional programmer on **Product Design**; then the % of Effort for actual **Programming** and finally the % of Effort for **Integration and Testing**.

(a)Product Design Phase

Having received a specification, the first task for the programmer is to sit down and design the software to accomplish the required result. This phase will include the logical structure and initial documentation to achieve this end. In prototyping it could include the design of the basic structure, screen layouts, color schemes. It should be done in close harmony with the User so that the final product is what was ordered and ***"does not result in any surprises at all"***. This will take 16%, 17% or 18% of the overall effort depending on development mode as we shall see.

(b)Programming Phase

When the design is complete there will then be the actual programming phase which will take between ½ and ¾ of the overall effort. The % will actually vary from 48% up to 68% and depends on the development mode and the actually size of the software unit in KDSI.

(c)Integrating & Testing Phase

Finally, once the unit has been completely coded it must now be Integrated and Tested with other software within the project. This is a test of such facets as interfaces, how our Software Unit is going to react and inter react with other Units both at interfaces, passing of parameters and variables, calling routines and boundary conditions. The actual test harness is beyond the scope of this particular text. The amount of the original overall Effort varies from 16% to 34% depending on development mode and the actual size of the code to be Integrated and Tested.

7. CASE

CASE is an acronym of Computer-Aided Software Engineering. **Software systems which are intended to provide automated support for software process activities,** such as requirements analysis, system modelling, debugging and testing

• **Upper-CASE**: Tools to support the early process activities of requirements and design

• **Lower-CASE**: Tools to support later activities such as programming, debugging and testing

On the surface, some speak of CASE tool integration in terms of the support of shared data storage [8]; others describe a fully supported development life cycle [6] where all tools interface to a common framework/database in a distributed environment [7].

7.1 CASE integration

• **Tools**; Support individual process tasks such as design consistency checking, text editing, etc.

• **Workbenches**: Support a process phase such as specification or design, Normally include a number of integrated tools

• **Environments**: Support all or a substantial part of an entire software process. Normally include several integrated workbenches

7.2 Integration Issues

With this in mind, the discussion of tool integration will be approached from several different aspects relative to the perspective of the end-user. Five areas of integration are examined here, namely:

• Single-vendor tool integration

• Multiple-vendor tool integration-

• Operating environment integration

• Development process integration

• End-user integration

7.1 Single-Vendor Tool Integration

The initial form of CASE tool integration is “internal” integration, that is, integration of the tools and data of a single vendor. Some vendor tools use a local data dictionary; others fashion a toolset joined together around a central, shared dictionary. The data dictionary is usually some form of (relational) database which offers a vendor a way to provide reliable data storage and access for the tools. This form of integration is generally of a proprietary nature [5].

7.2 Multiple-Vendor Tool Integration

Another form of integration is “external” integration. This is when a vendor integrates tools with those of another vendor. This integration can be of the form of “access control” and/or “data control.” In access control, the vendor allows the tools to be invoked/controlled by tools from other vendors and returns appropriate messages/codes to the invoking process. In data control, the vendor allows these external tools to (in)directly manipulate the data contained in the internal dictionary. This section focuses primarily on the data control aspect of external integration.

7.3 Operating Environment Integration

When discussing CASE tool integration, the operating environment should also be considered. This includes both the base computing environment and the add-on tools and utilities that compose the development support environment. Some CASE tools have versions that run on a personal computer (e.g., Excelerator (Index Technology), Teamwork (Cadre Technologies)),while others are targeted to workstations or mainframes (e.g., Software through Pictures (StP) (IDE), Procase C Environment (Procase Corporation)). Tools are also developed for use on a specific target operating system. The choice of system generally has to do with considerations for the technical (real-time) or commercial (Management Information Systems(MIS)) application to be hosted by the toolset.

7.4 Development Process Integration

CASE tools are also working into the framework of the development process. These tools are no longer targeted specifically to the analysis/design phase of software development. CASE tools are being considered to help combine the various phases of the entire life cycle (e.g.,project management, analysis and design, configuration management) in anticipation of smoothing process transitions.

7.5 End-User Integration

Finally, more emphasis is being applied to integration of CASE tools with the users themselves. The concept of integration with the end-user ranges from something as simple as maintaining a consistent user interface to something as complex as providing support for an expert system interface to aid in detailed design.

8. Conclusion:

A simplified representation of a software process presented from a specific perspective. Different generic process models to be used as Water Fall, Evolutionary Development, Formal transformation, and Integration from reusable components. A set of activities whose goal is the development or evolution of software all software processes are Specification, Development, Validation, and Evolution. Using Software engineering, CoCoMo process, and CASE process, the software should deliver the required functionality and performance to the user should be maintainable, efficient, usable, and dependable. Therefore, roughly 60% of costs are development costs, and 40% costs are testing costs. It depends on the type of system being developed and the requirements of system attributes such as performance, system reliability, and the development model that is being used. Software engineering in the 21st century faces three key challenges are (a)Legacy systems: which are Old, valuable systems must be maintained and updated. (b)Heterogeneity: Systems are distributed and include a mix of hardware and software (c)Delivery: There is increasing pressure for faster delivery of software.

References:

[1] Ian Sommerville, "Software Engineering", Addison Wesley, 7th edition, 2004.
[2] Karlm, "Software Lifecycle Models', KTH, 2006 .
[3] CTG. MFA – 003, "A Survey of System Development Process Models", Models for Action Project: Developing Practical Approaches to Electronic Records Management and Preservation, Center for Technology in Government University at Albany / Suny, 1998 .
[4] National Instruments Corporation, "Lifecycle Models", 2006 , http://zone.ni.com.
[5} Acly, E. “Looking Beyond CASE.” *IEEE Software*, 5, 2 (Mar 1988), 39-44.

[6] Martin, J. “Integrated CASE Tools a Must for High-Speed Development.” P*C Week,* 6, 3 (Jan

1990), 78.
[7] Phillips, B. “A CASE for Working Together.” *ESD: The Electronic System Design Magazine,* 1912 (Dec 1989), 55-58.

[8] Wasserman, A. I. “Integration and Standardization Drive CASE Advancements.” *Computer Design*, 27, 22 (Dec, 1988), 86.

BIOGRAPHY

Ali Tariq Bhatti received his Associate degree in Information System Security (Highest Honors) from Rockingham Community College, NC USA, B.Sc. in Software engineering (Honors) from UET Taxila, Pakistan, M.Sc in Electrical engineering (Honors) from North Carolina A&T State University, NC USA, and currently pursuing PhD in Electrical engineering from North Carolina A&T State University. Working as a researcher in campus and working off-campus too. His area of interests and current research includes Coding Algorithm, Networking Security, Mobile Telecommunication, Biosensors, Genetic Algorithm, Swarm Algorithm, Health, Bioinformatics, Systems Biology, Control system, Power, Software development, Software Quality Assurance, Communication, and Signal Processing. For more information, contact **Ali Tariq Bhatti** alitariq.researcher.engineer@gmail.com.