SOFTWARE ENGINEERING PROCESS DEVELOPMENT

M.Sc. Ivanova Milka.
Faculty of Mechanical Engineering – Technical University of Sofia, Bulgaria

Abstract: A software engineering (SE) process is a set of activities that leads to the production of software product. These activities may involve the development of software from scratch in a standard program language. However new software is developed by extending and modifying existing systems and by configuring and integrating off-the-self software or systems components.

In the report they have understand the concept of software engineering, software engineering process models and when these models might be used.

Keywords: Software, software product, software engineering, software process, process models

Many people equate the term software as a computer programs. But software is not just the programs also all associate documentation and configuration data that is needed to make these programs operate correctly. A software system usually consist of a number of separate programs, configuration files which are used to set up these programs, system documentation witch describe the systems' structure and user's documentation which explains how to use the system and web sites for users to download recent information.

Software that can be sold to the customer named a software product; There are fundamental types of software product:

- *Generic product* –stand-alone systems that are produced by development organization and sold on the open market;
- *Customized (bespoke) product* – systems which are commissioned by a particular customer (developed especially for that customer).

Software engineering is an engineering discipline that is concerned with all aspects of software production from early stage of system specification to maintaining the system after it has gone into use. Software engineering adopt a systematic and organization approach to work as this is often the most effective way to produce high quality production. However engineering is all about selecting the most appropriate method for a set of circumstances and more creative less formal approach to development may be effective in some circumstances. Less formal development is particularly appropriate for the development of web-based systems which required a blend of software and graphical design skills.

A software process is the set of activities and associated results that produce a software product. There are four fundamental process activities that are common to all software processes:

- *Software specification* – where customers define the software to be produced and constraints on its operation.
- *Software development* – where software is designed and programmed.
- *Software validation* – where software is checked to ensure that it is what the customer requires.
- *Software evolution* – where software is modified to adapt it to changing customer and market requirements.

A software process model is a simplified description of a software process that presents one view of that process. It may include activities that are part of a software process, software products and the roles of people involved in software engineering. Most software process models are based on one of three general models or paradigms of software development: *The waterfall approach* This takes the above activities and represents them as separate process phases such as requirements specification, software design, implementation, testing and so on. After each stage is defined it is 'signed-off, and development goes on to the following stage. *Iterative development* This approach interleaves the activities of specification, development and validation. An initial system is rapidly developed from very abstract specifications. This is then refined with customer input to produce a system that satisfies the customer s needs. The system may then be delivered. Alternatively, it may be reimplemented using a more structured approach to produce a more robust and maintainable system. *Component-based software engineering (CBSE)* This technique assumes that parts of the system already exist. The system development process focuses on integrating these parts rather than developing them from scratch.

Some examples of the types of software process model that may be produced are:

I. THE LINEAR SEQUENTIAL MODEL

Sometimes called the *waterfall model,* the *linear sequential model* suggests a systematic, sequential approach to software development that begins at the system level and progresses through analysis, design, coding, testing, and support. Figure 1 illustrates the linear sequential model for software engineering.

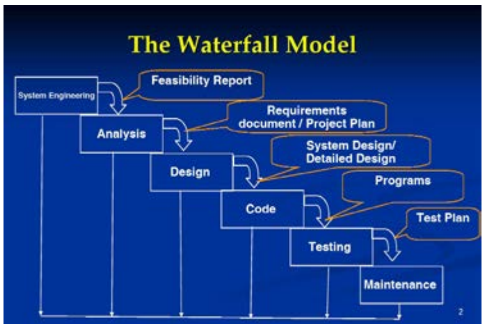

***Fig.1** "Waterfall" model*

Modeled after a conventional engineering cycle, the linear sequential model encompasses the following activities:

System/information engineering and modeling Because software is always part of a larger system (or business), work begins by establishing requirements for all system elements and then allocating some subset of these requirements to software. This system view is essential when software must interact with other elements such as hardware, people, and databases. System engineering and analysis encompass requirements gathering at the system level with a small amount of top level design and analysis. Information engineering encompasses requirements gathering at the strategic business level and at the business area level.

Software requirements analysis The requirements gathering process is intensified and focused specifically on software. To understand the nature of the program(s) to be built, the software engineer ("analyst") must understand the information domain for the software, as well as required function, behavior, performance, and interface. Requirements for both the system and the software are documented and reviewed with the customer.

Design Software design is actually a multi-step process that focuses on four distinct attributes of a program: data structure, software architecture, interface representations, and procedural (algorithmic) detail. The design process translates requirements into a representation of the software that can be assessed for quality before coding begins. Like requirements, the design is documented and becomes part of the software configuration

Code generation The design must be translated into a machine-readable form. The code generation step performs this task. If design is performed in a detailed manner, code generation can be accomplished mechanistically.

Testing Once code has been generated, program testing begins. The testing process focuses on the logical internals of the software, ensuring that all statements have been tested, and on the functional externals, that is, conducting tests to uncover errors and ensure that defined input will produce actual results that agree with required results.

Support Software will undoubtedly undergo change after it is delivered to the customer (a possible exception is embedded software). Change will occur because errors have been encountered, because the software must be adapted to accommodate changes in its external environment (e.g., a change required because of a new operating system or peripheral device), or because the customer requires functional or performance enhancements. Software support/maintenance reapplies each of the preceding phases to an existing program rather than a new one.

The linear sequential model is the oldest and the most widely used paradigm for software engineering. Among the problems that are sometimes encountered when the linear sequential model is applied are:

- Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.
- It is often difficult for the customer to state all requirements explicitly. The linear sequential model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.
- The customer must have patience. A working version of the program(s) will not be available until late in the project time-span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

Each of these problems is real. However, the classic life cycle paradigm has a definite and important place in software engineering work. It provides a template into which methods for analysis, design, coding, testing, and support can be placed. The classic life cycle remains a widely used procedural model for software engineering. While it does have weaknesses, it is significantly better than a haphazard approach to software development

II. PROTOTYPING MODELS

The prototyping paradigm (Figure 2) 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 I he prototype) is evaluated by the customer/user and used to refine requirements tor the software to be developed Iteration occurs as the prototype is tuned to satisfy the needs of I ho customer, while at the same time enabling the developer to better understand what needs to ho done

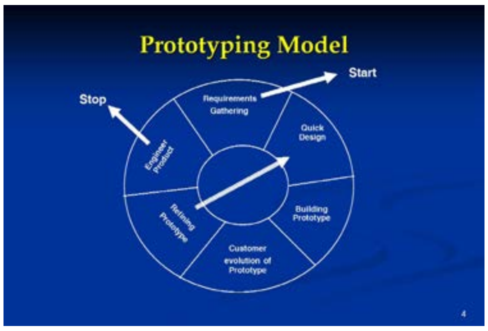

Fig.2 *Prototyping model*

Ideally, the prototype serves as a mechanism for identifying software requirements II a working prototype is built, the developer attempts to use existing program fragments or applies tools (e.g., report generators, window managers) that enable working programs to lie generated quickly

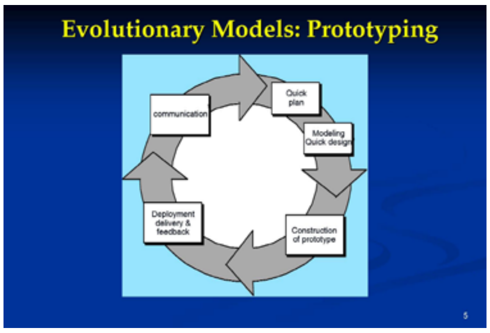
Evolutionary Models: Prototyping
communication
Quick plan
Modeling Quick design
Construction of prototype
Deployment delivery & feedback
5

Fig. 3 *Evolutionary prototyping model*

The prototype can serve as "the first system." The one that Brooks recommends we throw away. But this may be an idealized view. It is true that both customers and developers like the prototyping paradigm. Users get a feel for the actual system and developers get to build something immediately. Yet, prototyping can also be problematic for the following reason:

- The customer sees what appears to be a working version of the software, unaware that the prototype is held together "with chewing gum and baling wire," unaware that in the rush to get it working no one has considered overall software quality or long-term maintainability When informed that the product must be rebuilt so that high levels of quality can be maintained, the customer cries foul and demands that "a few fixes" be applied to make the prototype a working product. Too often, software development management relents
- The developer often makes implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability. After a time, the developer may become familiar with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system.

Although problems can occur, prototyping can be an effective paradigm for software engineering. The key is to define the rules of the game at the beginning; that is, the customer and developer must both agree that the prototype is built to serve as a mechanism for defining requirements. It is then discarded (at least in part) and the actual software is engineered with an eye toward quality and maintainability.

III. RAD MODEL

Rapid application development (RAD) is an incremental software development process model that emphasizes an extremely short development cycle. The RAD model is a "highspeed" adaptation of the linear sequential model in which rapid development is achieved by using component-based construction If requirements are well understood and project scope is constrained, the RAD process enables a development team to create a "fully functional system" within very short time periods (e.g., 60 to 90 days). Used primarily for information systems applications, the RAD approach encompasses the following phases :

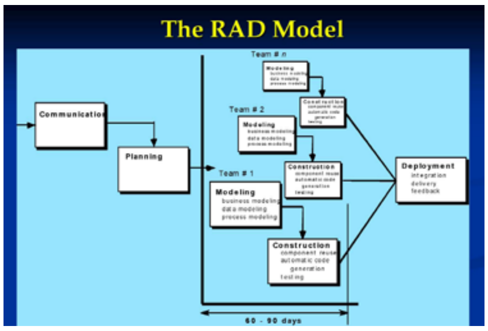
The RAD Model
Communication
Planning
Team # n
Team # 2
Team # 1
Modeling business modeling data modeling process modeling
Construction component reuse automatic code generation testing
Deployment integration delivery feedback
60 - 90 days

Fig. 4 *RAD model*

Business modeling The information flow among business functions is modeled in a way that answers the following questions: What information drives the business process? What information is generated? Who generates it? Where does the information go? Who processes it?

Data modeling The information flow defined as part of the business modeling phase is refined into a set of data objects that are needed to support the business The characteristics (called attributes) of each object are identified and the relationships between these objects defined.

Process modeling The data objects defined in the data modeling phase are transformed to achieve the information flow necessary to implement a business function. Processing descriptions are created for adding, modifying, deleting, or retrieving a data object.

Application generation RAD assumes the use of fourth generation techniques. Rather than creating software using conventional third generation programming languages the RAD process works to reuse existing program components (when possible) or create reusable components (when necessary). In all cases, automated tools are used to facilitate construction of the software.

Testing and turnover Since the RAD process emphasizes reuse, many of the program components have already been tested. This reduces overall testing time. However, new components must be tested and all interfaces must be fully exercised.

If a business application can be modularized in a way that enables each major function to be completed in less than three months (using the approach described previously), it is a candidate for RAD. Each major function can be addressed by a separate RAD team and then integrated to form a whole.

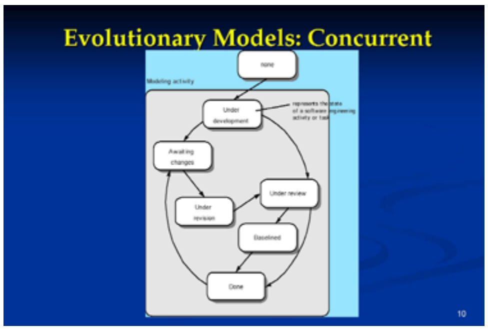

***Fig. 5** Evolutionary RAD model*

Like all process models, the RAD approach has drawbacks :

• For large but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams.

• RAD requires developers and customers who are committed to the rapid-fire activities necessary to get a system complete in a much abbreviated time frame. If commitment is lacking from either constituency, RAD projects will fail

• Not all types of applications are appropriate for RAD If a system cannot be properly modularized, building the components necessary for RAD will be problematic. If high performance is an issue and performance is to be achieved through tuning the interfaces to system components, the RAD approach may not work.

• RAD is not appropriate when technical risks are high. This occurs when a new application makes heavy use of new technology or when the new software requires a high degree of interoperability with existing computer programs.

IV. <u>THE EVOLUTIONARY SOFTWARE PROCESS MODEL</u>

Evolutionary models are iterative. They are characterized in a manner that enables software engineers to develop increasingly more complete versions of the software.

<u>The Incremental Model</u>

The **incremental model** combines elements of the linear sequential model (applied repetitively) with the iterative philosophy of prototyping. The incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces a deliverable "increment" of the software. It should be noted that the process flow for any increment can incorporate the prototyping paradigm. When an incremental model is used, the first increment is often a **core product.** That is, basic requirements are addressed but many supplementary features (some known, others unknown) remain undelivered The core product is used by the customer (or undergoes detailed review). As a result of use and/or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced

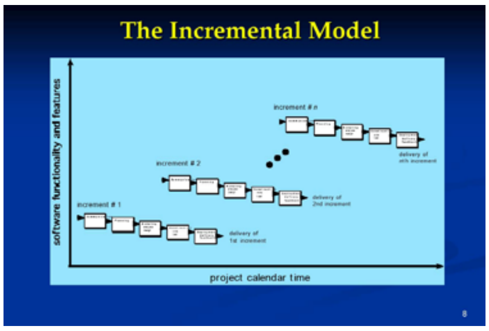
The Incremental Model
software functionality and features
increment # n
delivery of nth increment
increment # 2
delivery of 2nd increment
increment # 1
delivery of 1st increment
project calendar time
8

***Fig. 6** Incremental model*

The incremental model focuses on the delivery of an operational product with each increment. Early increments are stripped down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user.

<u>The Spiral Model</u>

The **spiral model** is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the linear sequential model. It provides the potential for rapid development of incremental versions of the software. Using the spiral model, software is developed in a series of incremental releases. During early iterations, the incremental release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

A spiral model is divided into a number of framework activities, also called **task regions.** Typically, there are between three and six task regions. Figure 2.8 depicts a spiral model that contains six task regions:

•**Customer communication** tasks required to establish effective communication between developer and customer.

•**Planning** tasks required to define resources, timelines, and other project related information.

•**Risk analysis** tasks required to assess both technical and management risks.

•**Engineering** tasks required to build one or more representations of the application

•**Construction and release** tasks required to construct, test, install, and provide user support (e.g., documentation and training).

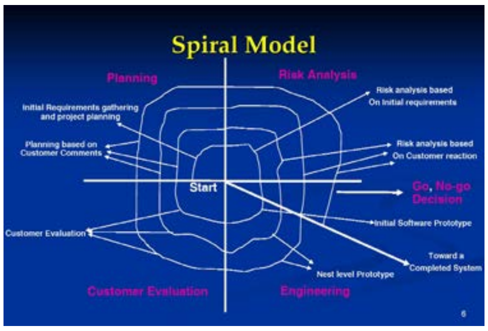
Spiral Model
Planning
Risk Analysis
Initial Requirements gathering and project planning
Risk analysis based On initial requirements
Planning based on Customer Comments
Risk analysis based On Customer reaction
Start
Go, No-go Decision
Customer Evaluation
Initial Software Prototype
Toward a Completed System
Nest level Prototype
Customer Evaluation
Engineering
6

***Fig.7** Spiral model*

Each of the regions is populated by a set of work tasks, called a **task set,** that are adapted to the characteristics of the project to be undertaken. For small projects, the number of work tasks and their formality is low. For larger, more critical projects, each task region contains more work tasks that are defined to achieve a higher level of formality.

The WIN WIN Spiral Model

The spiral model suggests a framework activity that addresses customer communication The objective of this activity is to elicit project requirements from the customer. In an ideal context, the developer simply asks the customer what is required and the customer provides sufficient detail to proceed. Unfortunately, this rarely happens. In reality, the customer and the developer enter into a process of negotiation, where the customer may be asked to balance functionality, performance, and other product or system characteristics against cost and time to market.

The best negotiations strive for a "win-win" result. That is, the customer wins by getting the system or product that satisfies the majority of the customer's needs and the developer wins by working to realistic and achievable budgets and deadlines.

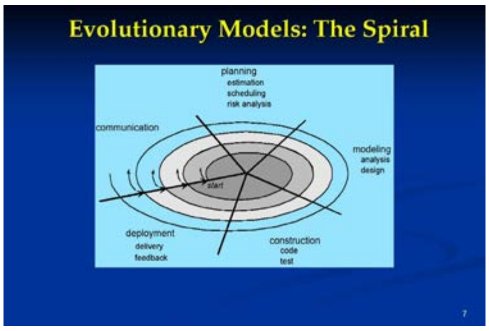

Fig.8 *Evolutionary spral midel (WINWIN model)*

Bohem's WINWIN spiral model defines a set of negotiation activities at the beginning of each pass around the spiral. Rather than a single customer communication activity there are some activities

- Identification of the system or subsystem's key "stakeholder"
- Determination of the stakeholders' "win conditions."
- Negotiation of the stakeholders' win conditions to reconcile them into a set of win-win conditions for all concerned (including the software project team).

Successful completion of these initial steps achieves a win-win result, which becomes the key criterion for proceeding to software and system definition.

In addition to the emphasis placed on early negotiation, the WINWIN spiral model introduces three process milestones, called **anchor points,** that help establish the completion of one cycle around the spiral and provide decision milestones before the software project proceeds.

In essence, the anchor points represent three different views of progress as the project traverses the spiral. The first anchor point, **life cycle objectives** (LCO), defines a set of objectives for each major software engineering activity. For example, as part of LCO, a set of objectives establishes the definition of top-level system/product requirements. The second anchor point, **life cycle architecture** (LCA), establishes objectives that must be met as the system and software architecture is defined. For example, as part of LCA, the software project team must demonstrate that it has evaluated the applicability of off-the-shelf and reusable software components and considered their impact on architectural decisions. **Initial operational capability** (IOC) is the third anchor point and represents a set of objectives associated with the preparation of the software for installation/distribution, site preparation prior to installation, and assistance required by all parties that will use or support the software.

Classes created in past software engineering projects are stored in a class library or repository. Once candidate classes are identified, the class library is searched to determine if these classes already exist. If they do, they are extracted from the library and reused. If a candidate class does not reside in the library, it is engineered using object-oriented methods. The first iteration of the application to be built is then composed, using classes extracted from the library and any new classes built to meet the unique needs of the application. Process flow then returns to the spiral and will ultimately re-enter the component assembly iteration during subsequent passes through the engineering activity.

The **unified software development process** is representative of a number of component-based development models that have been proposed in the industry. Using the **Unified Modeling Language** (UML), the unified process defines the components that will be used to build the system and the interfaces that will connect the components. Using a combination of iterative and incremental development, the unified process defines the function of the system by applying a scenario-based approach (from the user point of view). It then couples function with an architectural framework that identifies the form the software will take.

Literature:

1. Boehm B., Software Engineering Economics, Prentice Hall, 2003
2. Boehm B., Software Engineering, John Wiley & Sons, 2007
3. Pressman R., Software Engineering; A Practioner's Approach, McGraw Hill, 2010.
4. Sommerville I, Software Engineering, Pearson Edition, 2007