Evolutionary Process Models

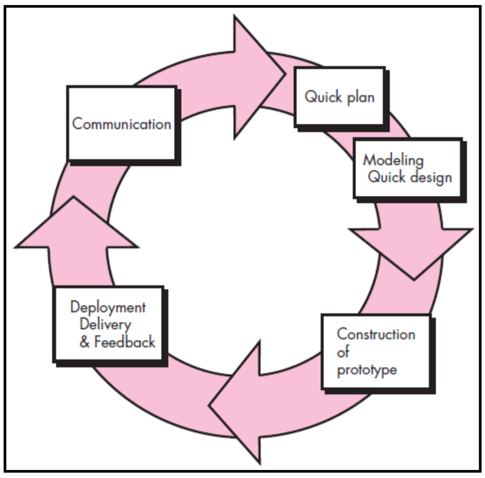
When core set of requirements is known but details of the product or systems extensions are not yet defined, one can use Evolutionary Process Models.

They are characterized in a manner that enables you to develop increasingly more complete versions of the software i.e. the product (software) evolves over time.

The popular evolutionary models are:
1. Prototyping process model
2. Spiral process model

Prototyping Model

A prototype is an initial version of a software system that is used to:
- Demonstrate concepts
- Try out design options
- Find out more about the problem and its possible solutions



1. **Communication**: Meet stakeholders to define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory.

2. **Quick plan + Modeling Quick Design**: A quick design focuses on a representation of those aspects of the software that will be visible to end users (e.g., human interface layout or output display formats)

3. **Construction** of prototype

4. The prototype is **deployed and evaluated** by stakeholders, who provide **feedback** that is used to further refine requirements.

In most projects, the first system built is barely usable. It may be too slow, too big, and awkward in use or all three. There is no alternative but to start again, painful but smarter, and build a redesigned version in which these problems are solved.

The prototype can serve as “the first system.” The one that Brooks recommends you throw away. But this may be an idealized view. Although some prototypes are built as **“throwaways”**, others are evolutionary in the sense that the prototype slowly evolves into the actual system.

Benefit:
Users get a feel for the actual system, and developers get to build something immediately.

Drawback:

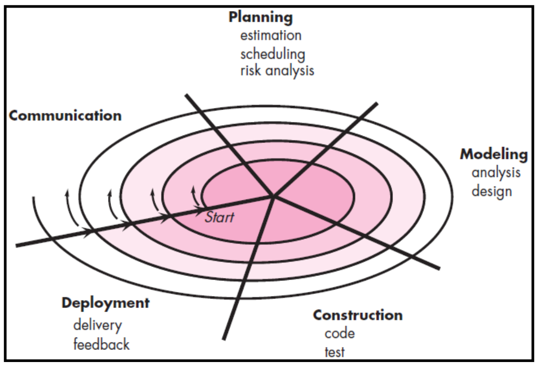
1. Prototype appears to be a working version of software to stakeholders.
   They might refuse to understand that product needs to be rebuilt and demand to fix the prototype to make it a working product.
   Many times software development management relents to this.

2. A software engineer may make 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, he may become comfortable 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.

Spiral Model

The spiral model is an **evolutionary software process model** that couples the **iterative** nature of **prototyping** with the **controlled and systematic aspects** of the **waterfall model**.

It provides the potential for rapid development of increasingly more complete versions of the software.



Definition of Spiral Model by Barry Boehm:

- The spiral development model is a *risk*-driven *process model* generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems.
- It has two main distinguishing features:
  - One is a **cyclic approach** for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk.
  - The other is a **set of anchor point milestones** for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.

- **Anchor point** is the point where spiral crosses horizontal line.
- Each anchor point marks end of a spiral and beginning of the next spiral.
- At every anchor point, definition is enhanced (evolved) and risk is reduced.

- A spiral model is divided into a **set of framework activities** defined by the software engineering team.
- Each of the framework activities represents one **segment** of the spiral path.
- As this **evolutionary process** begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center.
- **Anchor point milestones** (a combination of work products and conditions that are attained along the path of the spiral) are noted for each evolutionary pass.

- The first circuit around the spiral might result in the development of a product specification;
- subsequent passes around the spiral might be used to develop a prototype and
- Then progressively more sophisticated versions of the software.

- Each pass (through the planning region) results in adjustments to the project plan.
- Cost and schedule are adjusted based on feedback from the customer
- The project manager adjusts the planned number of iterations required to complete the software.

"The spiral model can be adapted to apply throughout the entire life cycle of an application, from concept development to maintenance."

The spiral model is a realistic approach to the development of large-scale systems and software.
- The spiral model uses prototyping as a risk reduction mechanism.
- It also enables application the prototyping approach at any stage in the evolution of the product.
- It maintains the systematic stepwise approach suggested by the classic life cycle but incorporates it into an iterative framework that more realistically reflects the real world.
- The spiral model demands a direct consideration of technical risks at all stages of the project.

But still it is not the ultimate solution…
Drawbacks:
1. It may be difficult to convince customers (particularly in contract situations)
2. It demands considerable risk assessment expertise and relies on this expertise for success.

Concurrent Development Model

The concurrent development model (concurrent engineering) allows a software team to represent **iterative and concurrent elements** of any of the process models.
The concurrent model is often more appropriate for product engineering projects where different engineering teams are involved.

In the figure, one element of the concurrent process model shows different states.
A **state** is some externally observable mode of behavior.

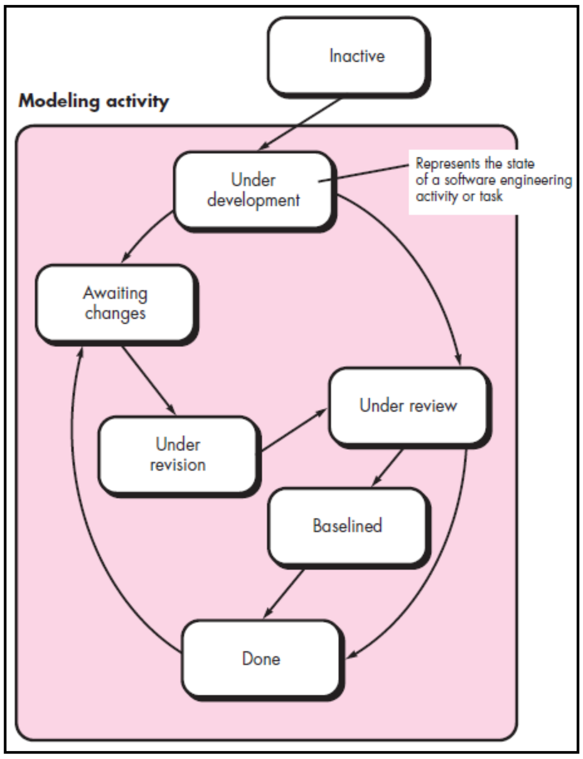


Fig. provides a schematic representation of one software engineering activity within the modeling activity using a concurrent modeling approach.

- The activity—**modeling**—may be in any one of the states noted at any given time.
- Similarly, other activities, actions, or tasks (e.g., **communication** or **construction**) can be represented in an analogous manner.
- All software engineering activities exist concurrently but reside in different states.

For example:
1. The **communication activity** exists in the **awaiting changes state**.
2. The **modeling activity** initially existed in the **inactive state.** Now, it makes a transition into the **under development state**.
3. If the customer indicates that changes in requirements must be made, the **modeling activity** moves from the **under development state** into the **awaiting changes state**.

Concurrent modeling defines a series of events that will trigger transitions from state to state for each of the software engineering activities, actions, or tasks.

- Rather than confining software engineering activities, actions, and tasks to a sequence of events, it defines a process network.
- Each activity, action, or task on the network exists simultaneously with other activities, actions, or tasks.
- Events generated at one point in the process network trigger transitions among the states.

Specialized Process Models

1. Component Based Development

A **component** is a modular, deployable, and replaceable part of a system that encapsulates implementation.

An individual software component is a software package, a web service, a web resource, or a module that encapsulates a set of related functions (or data).

Examples of component models are:

a) Enterprise JavaBeans (EJB) model
b) Component Object Model (COM) model
c) .net model
d) Common Object Request Broker Architecture (CORBA) component model

The components can be:

- conventional software modules
- object oriented classes
- packages of classes

The component-based development model incorporates the following steps (implemented using an evolutionary approach):

1. Available component-based products are **researched** and **evaluated** for the application domain in question.
2. Component **integration issues** are considered.
3. **Software architecture** is designed to accommodate the components.
4. Components are **integrated** into the architecture.
5. Comprehensive **testing** is conducted to ensure proper functionality.

Benefits of component-based development model:

- Software reuse
- Reduction in development time
- Reduction in cost (if component is frequently reused)
- Reduction in requirement of resources

Limitation:
This model can be used when the requirements are generic. If requirements are specific, then it cannot be used.

2. The Formal Methods model

- The formal methods model encompasses a set of activities that leads to **formal mathematical specification** of computer software.

- A formal method enables to **specify, develop, and verify** a computer-based system by applying a rigorous, mathematical notation.

- **Ambiguity, incompleteness, and inconsistency** can be **discovered and corrected** more easily by the application of mathematical analysis.

- When formal methods are used during design, they serve as a basis for program verification and therefore enable you to discover and correct errors that might otherwise go undetected.

But these are not widely used due to the following **reasons**:

- The development of formal models is currently quite **time consuming and expensive**.
- Because few software developers have the necessary background to apply formal methods, **extensive training is required**.
- It is difficult to use the models as a communication mechanism for **technically unsophisticated customers**.

Even then this model is popular among:

- Software developers building **safety-critical software** (e.g., developers of aircraft avionics and medical devices)
- Developers that would suffer **severe economic hardship** should software errors occur.

3. Aspect oriented software development

- **Aspect-oriented software development** (AOSD) is a software development **technology** that seeks **new modularizations** of software systems in order **to isolate secondary or supporting functions from the main program's business logic**.
- AOSD allows **multiple concerns** to be <u>expressed separately</u> and <u>automatically unified</u> into working systems.
- A **concern** is a particular set of information that has an effect on the code of a computer program; customer requires properties or areas of technical interest for the entire architecture.
  - Some concerns are **high-level properties** of a system (e.g. security, fault tolerance)
  - Some affect **functions** (e.g. the application of business rules)
  - Some are **systemic** (e.g. task synchronization or memory management)

When concerns cut across multiple system functions, features, and information, they are often referred to as **crosscutting concerns**.

Aspectual requirements define those crosscutting concerns that have an impact across the software architecture.

AOCE (aspect-oriented component engineering) uses a concept of **horizontal slices** through **vertically-decomposed** software components, called **"aspects,"** to characterize cross-cutting functional and non-functional properties of components.

An aspect of a program is a feature linked to many other parts of the program, but which is not related to the program's primary function.

For example:
Logging code can crosscut many modules.
So, the aspect of logging should be separate from the functional concerns of the module it cross-cuts.

Isolating aspects is the aim of AOSD.

Examples of aspects: persistence, logging, security, authentication, authorization, performance monitoring, etc.

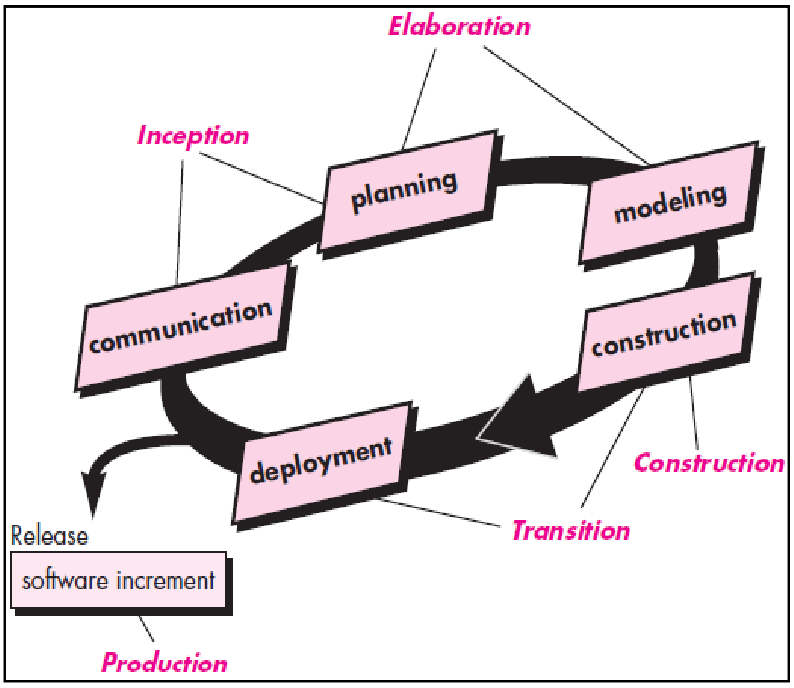
Persistence means saving the state.

- This is achieved in practice by storing the state as data in computer data storage.
- Picture editing programs or word processors, for example, achieve state persistence by saving their documents to files.
- Programs have to transfer data to and from storage devices and have to provide mappings from the native programming-language data structures to the storage device data structures. (Object-Relation Mapping)

- Aspect oriented process model is not well-established.
- The process flow will be both evolutionary and concurrent.
  Evolutionary: as the aspects are identified and then constructed
  Concurrent (parallel development): as the aspects are engineered independently of localized software components.

Unified Process Model

- Ivar Jacobson, Grady Booch, and James Rumbaugh: "use case driven, architecture-centric, iterative and incremental" software process.
- Best features and characteristics of **traditional software process models** along with implementations many of the best principles of **agile software development**.
- The Unified Process recognizes the importance of customer communication and streamlined methods for **describing the customer's view of a system** (which is known as **use case**).
- It suggests a process flow that is **iterative** and **incremental**, providing the **evolutionary** feel that is essential in modern software development.
- UML—a unified modeling language contains a robust notation for the modeling and development of object-oriented systems.
- UML became a de facto industry standard for object-oriented software development.



1. The **inception** phase encompasses the **communication and planning** activities.
2. The **elaboration** phase encompasses the **communication and modeling** activities.
3. The **construction** phase is identical to the **construction** activity.
4. The **transition** phase encompasses the latter stages of the generic **construction** activity and the first part of the generic **deployment (delivery and feedback)** activity.
5. The **production** phase coincides with the **deployment** activity of the generic process.

Inception Phase

- Functional business requirements are identified. (represented as usecase diagram)
- Tentative architecture for subsystems
- identifies resources
- assesses major risks
- defines a schedule

Elaboration Phase

- Architectural representation is expanded to the use case model, the requirements model, the design model, the implementation model, and the deployment model.
- Modifications to the plan

Construction Phase

- develops or acquires the software components that will make each use case operational for end users
- unit tests are designed and executed
- Integration activities (component assembly and integration testing) are conducted.
- Use cases are used to derive a suite of acceptance tests that are executed prior to the initiation of the next UP phase.

Transition Phase

- Software is given to end users for beta testing
- user feedback reports both defects and necessary changes
- user manuals, troubleshooting guides, installation procedures are made
- a usable software release at end of this phase

Production Phase

- the ongoing use of the software is monitored
- support for the operating environment (infrastructure) is provided
- defect reports and requests for changes are submitted and evaluated

- The five Unified Process phases do not occur in a sequence, but rather with staggered concurrency.
- A **workflow** (like taskset previously discussed) identifies the tasks required to accomplish important software engineering action and the work products that are produced as a consequence of successfully completing the tasks.
- Not every task identified for a Unified Process workflow is conducted for every software project.
- The team adapts the process (actions, tasks, subtasks, and work products) to meet its needs.