Productivity Improvement With Evolutionary Development

Dr. Anneliese von Mayrhofer Barbara Hirsh
Illinois Institute of Technology
Chicago, IL 60616

Abstract
Productivity is usually associated with development time. Too often, unreasonably high maintenance costs overshadow gains in development productivity. An evolutionary approach to software development can realize lifetime productivity improvements. The purpose of this paper is to explain the natural evolutionary process of software. Negative evolutionary influencers can erode the useful life of software, but these can be regulated. Software evolves through three stages: (1) elaboration, (2) adaptation and (3) mutation, progressive expansion/growth. Differing life cycle models are contrasted for (1) how they consider these three evolutionary stages, (2) how well they meet customer needs, and (3) how they take into account evolution regulators.

1. Introduction
What do a western ghost town, a dodo bird, and VISICALC software have in common? They were all unable to withstand changes in their environment and succumbed to evolutionary decay. When evolutionary forces are not understood nor regulated, it is analogous to blindly traveling down dark alleys, not knowing that current decisions influence our final destination. Similarly, not taking evolutionary forces during software development can result in a premature death. Taking an evolutionary course is like taking a safe and well-lit path. The course is carefully predetermined but can be easily revisied. We can decide on the best direction at cross roads. Decisions are not just based on the original plan but on added information learned while traveling the road. However, accurate initial decisions must be made so that the journey will not fail due to the lack of provisions.

We must consider quality during a product's useful life (beyond its development time) to arrive at an accurate productivity measure. Too often productivity is measured in terms of a product's original development time; "How much code can be written in how little time?" Such disregard for quality leads to reduced quality levels and unmaintainable products, and severe limitations on the product's longevity. Evolutionary development emphasizes "How can we develop quality software that withstands the test of time and changing customer demands?"

Evolution is a natural phenomenon. When ignored, it will decay software systems to a state of obsolescence and discount any development-oriented productivity measures. When understood, planned and managed properly, evolution makes changes not only effective but desirable.

2. Productivity Measures
Productivity is often defined for software development as the following:

\[
\text{productivity} = \frac{\text{size of products}}{\text{development time}}
\]

Productivity within the limited scope of "size over time" measure overlooks factors such as cost, quality, marketability and maintenance effort. Productivity can be artificially inflated over the short term. Product size can be artificially extended. This may not only affect the performance of the product but may reduce the clarity of the code and its maintainability. Development time can be reduced by limited testing and documentation efforts. Development effort is transferred to the maintenance phase. Such actions increase productivity during development but greatly decrease the productivity during the rest of the software's life. Shortcuts to reduce development time will, therefore, have an adverse effect on maintenance cost. [1]

Productivity gains during development must hold up during the lifetime of the product. A majority of the lifetime expenditure of software is incurred not in its creation but in its subsequent evolution. Managers of the software development environment need to consider and continually assess product quality, customer satisfaction, and cost of development not only during initial development but throughout the software's life. A valid productivity measure in an evolutionary context is:
productivity = \frac{(\text{lifetime}) \text{ product revenue}}{(\text{lifetime}) \text{ production cost}}

We can increase productivity by either developing more products capable of producing revenue or by reducing costs throughout the product's lifetime (development and maintenance). Thus, this measure prevents shifting costs from development into maintenance and considers lifetime costs. It also implies that developing the right product at the right time for the right market is important, thus targeting customers' needs and appropriate quality levels as important ingredients to high productivity.

Software quality can be measured using a tree-based scoring method for factors contributing to software quality (e.g. [1], [2], [3]). Marketability and salability relate to the quality of the work done by marketing and sales and are measured separately using appropriate measures. All costs can be measured through simple aggregation of costs per product in dollars. Evolutionary development promises longer lifetime (and associated revenue) at less cost. Both increase our productivity measure.

3. Evolutionary Stages

Software evolution can be segmented into stages. Levels of evolution were first discussed by Lehman [4]. Software evolution happens in three stages:

Stage 1: Elaboration
Software development evolves as the developer (or the project team) grasps a better understanding of the problem or role the system is to play. Just as the developer's knowledge evolves, so does the software product. As the problem is clarified, the developer also increases knowledge about possible design and implementation solutions. The process of elaborating and further refining the initial basic concepts to an operational system can be considered process dependent evolution. Feedback from the customer is critical at this stage to ensure correct conceptualization of the system is consistent. Productivity measures traditionally use the time to complete this stage with little consideration for software quality. Productivity measures must consider how the current operational system is meeting current needs of the user's environment.

Stage 2: Adaptation and Mutation
Deficiencies in processes employed in Stage 1 and changes within the user environment cause a transition to the second stage of evolution. Stage 2 occurs after the system is operational. It is characterized by activities such as fixing problems and adding enhancements. Fixing problems denotes inadequacies of the development process. Adding additional functionality is usually a result of the user adapting to a changing environment or plan. This results in releases of modified code and documentation. This evolution can be considered intrinsic evolution as it mirrors the evolution of the environment in which it is placed. Feedback is directed at understanding new user demands and the evolving system, in order to incorporate changes without deteriorating it.

Stage 3: Progressive Expansion and Growth
Software enters the third stage of evolution when it has been modified and expanded to such a degree that continual evolution of the system is no longer possible. The complexity of the system may render continual evolution, in terms of expansion, infeasible. The complexity is often a result of the intradependencies within the system and the interdependencies with the external environment. The system may have grown in a direction unconducive to new technologies in the user environment or to enhancement of the system's software and/or hardware. This evolution is comparable to the Darwinian principles of natural selection and mutation. The system mutated itself to a state of obsolescence. Feedback at this stage is understanding the system and the alternatives to that system. It also involves transferring the knowledge and experience gained to the successor system. We want to avoid entering this stage, in order to increase the software's life.

4. Evolutionary Influencers

Evolution is a natural decay process, ultimately resulting in obsolete software. Environment influencers erode the software product, placing limitations on its continued viability. Controlling
these influencers will prolong software life. Examples of evolutionary influencers are:

- the quality of managerial decision-making
- advances in technology
- software developed and maintained in a "fire-fighting mode" environment
- marketplace demands and the lack of anticipation of those demands
- volatility of the customer environment
- disregard for feedback
- not planning for evolution

These influencers can cause software systems to fail prior to their completion, or, if they are completed, they limit the software system's longevity. All software products are affected by these evolutionary influencers. Evolution is intrinsic to the very nature of computer usage and of the associated programs, once they exceed some minimal capability [4]. The environment of most software systems is not static, but in a state of change, or evolution. Therefore, requirements of a software system are, at most, a temporary actualization of its place within this changing environment. The software system itself alters the user environment which then alters the customers' expectations and needs of that software product. As the needs of the user evolve, the product itself must evolve to meet these needs.

5. Evolutionary Regulators

5.1. Description of Evolutionary Regulators

Software does not have to evolve on a predetermined course of self destruction. Actions can be taken to counterbalance negative decay influencers. We can regulate the growth of the software product in a positive manner. These regulators derive from Gilb's critical characteristics of evolutionary delivery planning [5].

EVOLVING LIFE PLAN - Development must be viewed as a continual process within a product's life. This requires a life plan that also evolves when understanding increases. Preconceived notions about the course to follow create a closed mindset that will not allow the product to continually evolve. We need an attitude away from "create the perfect product, forever" to "design the product so that it can continually evolve." Future steps are based on the results of the last step. We need to evaluate direction and adjust course when changes in the original goals and conceptions are found inappropriate or when a better alternative path becomes obvious. Each crossroad is evaluated with redirection possible at any time. Change is planned by enforcing a design methodology that emphasizes adaptable, modifiable, and expanded designs. Change results from proactive choices instead of reactive limitations.

DEVELOPMENT BY CHUNKS - Development proceeds in small manageable sections. Development is a continual process and the best way to control it is to chunk it into small pieces. This makes risk manageable. When change is incremental, we can easily see the effect of the current action. If it degrades the system, we need only step backwards to the last crossroad. We return to a previous state and try to find a better solution to the current problem. This regulator also uncovers ineptness of managerial decisions.

FEEDBACK CHANNELS - Procedures, techniques, and tools are needed for eliciting, evaluating and incorporating feedback. Increasing the number of software development cycles becomes a mechanism for increasing the amount of feedback and the time frame in which it becomes incorporated into the software. Evolutionary development relies heavily on feedback channels. Feedback directs development and evolution to manipulate evolving customer needs. Company and customer goals are fed into the development environment. Feedback channels to and from the customer are open throughout the life of the product. External sources also add valuable information for incorporating technological advances.

CUSTOMER-ORIENTED CULTURE - Development and support organizations must be customer oriented (not developer oriented). Company goals must not be in direct conflict with customer goals. We must understand what the end user needs and why. Customer satisfaction can then be maximized. This allows control of volatile requirements and changing marketplace demands. It really doesn't matter how wonderful and innovative software is, if it does not meet the customer's current needs. Functional, quality and resource requirements are based upon the customer. The entire software development environment is accountable for a satisfactory software system. This approach emphasizes results, instead of relying on the process and hoping for the results. Demarco [6] states that methodology should not be
standardized but must adapt to the project (or, actually, to customer goals). When methodology is strictly enforced, the methodology, not the developers, makes the decisions. Developers become responsibility free. They need an acceptable degree of freedom to control their success. Process or methodology must never become more important than the resulting product.

SYSTEM ARCHITECTURE - The system is evolving and changing, but it should be easy to understand the current system at any time. Developers must understand how each correction to the system is changing not only that one error but the overall system.

QUALITY ENGINEERED INTO THE PRODUCT - The quality of the product is determined up front, a factor in all decisions. Quality is designed and planned into the product. Global product quality objectives are clearly identified.

5.2. The Effect of Evolutionary Regulators

Evolutionary regulators reduce the force of evolutionary influencers towards obsolescence of the product. These are summarized in Figure 1. "H" indicates that the evolutionary regulators have a high positive effect on evolutionary influencers. "M" indicates a medium positive effect. "L" indicates a low positive effect on evolutionary influencers.

<table>
<thead>
<tr>
<th>EVOLUTIONARY REGULATORS</th>
<th>EVOLUTIONARY INFLUENCERS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Evolving Life Plan</td>
<td>H</td>
</tr>
<tr>
<td>Development by Chunks</td>
<td>H</td>
</tr>
<tr>
<td>Feedback Channels</td>
<td>H</td>
</tr>
<tr>
<td>Customer Orientation</td>
<td>M</td>
</tr>
<tr>
<td>System Architecture</td>
<td>L</td>
</tr>
<tr>
<td>Quality &quot;Engineered&quot;</td>
<td>M</td>
</tr>
</tbody>
</table>

Figure 1: Impact of Evolutionary Regulators on Evolutionary Influencers

The managerial decision process is likely to improve because the risks are more manageable. Smaller chunks make the intra-dependencies within the project easier to define. The effect of wrong decisions is limited to the scope of the chunk. Course correction is possible during all stages of the project without losing all prior benefits. Accountability improves: results can be seen at the end of each chunk.

The evolutionary development process caters to a proactive environment which can handle external forces, such as technological advances of the development environment and advances within the customer's environment, market demands caused by competition or a customer environment in a state of flux. With a product that is maintainable, adaptable and changeable, the development environment can handle the impact of change to the product. The evolutionary approach uncovers errors earlier when actions can be taken to remedy them.

6. Incorporating Evolutionary Concepts

Within Life Cycle Models

6.1. Linear, Sequential Software Development Models

Understanding the development process brought insight into how to manage and control the first stage of evolution. Understanding software development as an elaboration or refinement process of the original conceptualization of the problem triggered ways of defining this process. One of the basic concepts is that, if enough planning and information are done up front, a perfect product would be built. Royce proposed the classic waterfall model (not given that name until a later date) [7]. The waterfall process model emphasizes the creation of documentation (phase deliverables) as a way to mark the gap between the developers conceptualization and creating the "perfect product." This emphasis overshadows other factors that influence the quality of the final product. For example, important feedback chains may be ignored:

- Experience gained by proceeding through the process is not fed back.
- Inevitable changes in the customer environment due to either the new product itself or changes from external forces are not considered in the process. They are "oversights."

Changes to early phase deliverables are difficult. Similarly, it is more difficult to paddle upstream than to continue to go with the flow, even though it may mean that a wrong fork was taken and the final destination may not be suitable. The waterfall life cycle model emphasizes the transference of information forward. The process does not allow feedback from later phase deliverables or customers to influence in the planned target system. The waterfall model allowed feedback from the
current phase to the immediately preceding phase for the purpose of correcting errors. A common conception of the waterfall model (by Boehm [8]) consists of these phases: system feasibility, software plans and requirements, product design, detailed design, code, integration, implementation, and operation/maintenance.

The waterfall model is development oriented. It does not reexamine the original "marching orders." There is user influence in the early phases of the waterfall life cycle model, but it assumes that, after the developer studies the user's environment and gets a conceptualization of the correct solution (software system) the implementation of that initial conceptualization becomes more important than any reexamination or redefinition of the initial conceptualization to fit changing needs.

This process model does not consider the changing nature of the system due to error corrections, nor is there much thought given to how the environment changed because of the addition of the system. This usually creates greater demands on the functionality of the system than previously conceptualized. Since inherent evolutionary concepts are often ignored during development, the support phase is a reactive phase. Preventive measures to prolong the health of the system are not part of the process because all the initial planning supposedly created a "perfect system." The user's growth and sophistication (due in part to the system) and the continual mutation of the system through additions and modifications would eventually cause the system to obsolescence. The second stage of evolution (mutation and expansion) is not considered in the framework of the waterfall model. Evolution is only considered during the operation and maintenance phase. How sequential development models handle the evolutionary stages can be conceptualized as shown in Figure 2.

Evolutionary Development - Tom Gilb's Evolutionary Delivery Model is an example of evolutionary process models [5]. See Figure 4.

Like the traditional waterfall model, it began with a set of objectives that specify quality and cost associated with reaching goals. Instead of doing "the ultimate product" at once, we select chunks or "critical success features." Each product at the end of a cycle represents the most useful features to date. The product grows in a user-directed fashion. Building a product in successive cycles requires a basic design that is easy to change and adapt. Reusability, modularity, and adaptability are important characteristics of software built in evolutionary cycles. Advantages of this incremental approach are:
* users see something useful earlier than if entire

6.2. Nonsequential Software Development Models
The next generation of life cycle models takes into account other facets of evolution. The second stage of evolution is as important as the first stage. The pitfalls of looking at software development as merely a linear activity seriously limits the applicability of previous models. The emphasis on evolution has become evident in evolving software development process models that look at software development as a nonsequential activity. Figure 3 shows how evolutionary stages are handled in non-sequential software development models.
system had been built in one full sweep
* before new features are added, feedback loops afford the user repeated input into development
* limit amount of change for any development cycle
* minimize danger of developing product past the user.

The four steps per iteration through the smaller loop closely relate to specification and implementation phases in the waterfall model. The difference lies in their size and the frequent feedback built into the evolutionary model. Unlike the waterfall model, evolutionary development explicitly considers transition between evolutionary stages.

**Iterative Enhancement** - Basili and Turner first introduced iterative enhancement [9] in 1975. The basic principle of this method is application of stepwise refinement. Initially, only a subset of the total system is implemented. New versions are added to the initial core. New versions include both extensions and design modifications. This non-sequential process model emphasizes the principle of building on past experience. Designs are continually altered as the implementor learns from the path taken.

**Eternal Development Cycle** - The Eternal Development Cycle Model by Deming [10] is essentially the evolutionary delivery process model. The main difference is that the emphasis is not on small manageable sections. The sectioning is variable and determined by user feedback.

**Embedded Phased Approach** - The Embedded Phased approach, as explained by von Mayrhauser [1], combines the idea of evolution with well-defined sequential development steps. It is specifically geared towards all development phases.

### 6.3. Future Software Development Models
Software development process models will continue to evolve. They will guarantee that the initial conceptualizations become accurate actualizations. Software will be able to handle various types of modifications, including adaptations to meet the evolving environment. New process models must consider that software will eventually enter Stage 3. This can be conceptualized as shown in Figure 5.

They must provide direction for reusing the software with minimal effort. Then useful lifetime will span generations instead of a few years.

### 6.4. Software Development Models and Evolution Regulators
Regardless of the software development model, software passes through the three evolutionary stages. Software development models incorporate evolutionary regulators to differing degrees. How much a software development model considers evolutionary regulators has a great impact on the longevity of the software product. Figure 6 summarizes the extent to which the above life cycle models consider evolutionary regulators.

<table>
<thead>
<tr>
<th>EOLVING LIFE PLAN</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>Stage 2</td>
<td></td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>Stage 3</td>
<td></td>
<td></td>
<td>H</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&quot;ENGINEERED QUALITY&quot;</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>Stage 2</td>
<td></td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>Stage 3</td>
<td></td>
<td></td>
<td>H</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUSTOMER-ORIENTED CULTURE</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1</td>
<td>L</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>Stage 2</td>
<td>L</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>Stage 3</td>
<td></td>
<td></td>
<td>H</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FEEDBACK CHANNELS</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1</td>
<td>L</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>Stage 2</td>
<td>L</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>Stage 3</td>
<td></td>
<td></td>
<td>H</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SYSTEM ARCHITECTURE</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1</td>
<td>L/M</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>Stage 2</td>
<td></td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>Stage 3</td>
<td></td>
<td></td>
<td>H</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEVELOPMENT BY CHUNKS</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>Stage 2</td>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>Stage 3</td>
<td></td>
<td></td>
<td>H</td>
</tr>
</tbody>
</table>

H = High
M = Medium
L = Low
N = Non-existent

1 = Linear Waterfall
2 = Non-Sequential Model
3 = Perfect Evolutionary Model

Figure 6 - Occurrence of Evolution Regulators in Life Cycle Models
6.5. Meeting Customer Goals

One of the main differences of sequential and non-sequential development is the time frame required to meet customer goals and the total life span of the product. These are compared in the following figures.

The gap between the customer need line and the product's current state line is determined by the amount of influence the evolutionary influencers have on the product's growth direction. In the perfect evolutionary software development environment, customer goals would be continually met no matter how fast they change or increase.

7. Conclusions

Productivity must be looked at in terms of product quality, lifetime cost, and customer satisfaction. To meet these end results, the software products need to be conceived and understood, developed, and allowed to evolve within an evolutionary approach. Evolutionary regulators ensure that the evolution direction will be positive.

By using an evolutionary approach to software development, the benefits of the effort spent on development are actualized over a longer period of time. Productivity increases because the life span of the product increases. Gains in development productivity are not eaten up by support cost. Development productivity gains are not lost because of a lack of understanding of evolution. The incorporation and understanding of the evolutionary stages within software development has itself been an evolving process. Software life cycle models have been evolving to meet the challenges of complex software development.

Bibliography