1.3 Foundational Concepts

Along with the preceding formulation of terminology, an appreciation of several general concepts provides a foundation for understanding the approach being described.

The Evolution of Software Development as a Discipline

Every systematic discipline of practice has started as an opaque and individualized handcraft. As experience is gained, tools and techniques are developed that reduce routine and more mechanical aspects of the effort. Fabricated parts are created to further reduce the detail work required to create a finished product. At this point, it is still a fundamentally "manual" effort that relies on the skills and efforts of individuals to fashion more or less custom products. As understanding grows of an overall standard process for creating a product, machinery is developed for automation and mass production in which efforts begin to focus more on improving productivity and product quality. Improvements in mass production ultimately lead to mass customization for products that are individually tailored to specific needs and preferences, partially regaining some of the benefits of handcraft but with greater consistency and reduced labor.

Software development is following this same progression. It is now at the stage of being a tool- and parts-supported craft within a somewhat standardized process. This is encumbered by a mentality focused on building unique, statically-defined products that solve well-understood problems.

This craft-based approach has been augmented over time, with more disciplined practices, introduction of tools supporting more mechanical aspects of development (e.g., configuration management and builds, documentation, regression testing), and greater use of previously developed parts (i.e., commonly useful functionality). Although the basic principles and practices of software engineering are well-established based on 50 years of experience, the form these take will continue to progressively change. These and other advances are a foundation for an industrial mindset that will

supplant systemically-flawed assumptions of traditional practice leading to the ability to mass produce customized products.

Separation of Concerns

Separation of concerns is the principle of organizing a complex matter into a constituent set of elements (i.e., of action or information). Each such element is conceived to have singular authority over a specified scope of responsibility and can be understood and realized in terms of its unique contribution to the matter as a whole.

This principle guides the structuring of, for example, a process into activities, an enterprise into programs, human activity into roles, and hardware and software functionality into discrete components. This is the basis for the concept of information hiding that Parnas conceived as the foundation for designing software for ease of change.

This principle motivates distinguishing between engineering and manufacturing as distinct elements of product development. Manufacturing is seen as the realization of products whereas engineering is seen as providing the means for effective and efficient manufacturing. A third element, management, is seen as the direction and coordination of engineering and manufacturing efforts within an enterprise.

The Need for an Engineering Discipline of Software

In early days of building software, when problems to be solved were simpler, code could be written using a simple notation to just mimic the steps a person would take for a solution. Code was written in discrete sections (e.g., blocks or subroutines) that provided sufficient computational separation of concerns. Such code did not have a complex structure and could be quickly understood by others. The translation from a human description of the problem to a machine-processable solution was simple. The existing way of working was affected only by the selective automating of discrete, previously manual tasks.

Software was and still is built by devising a human expression of intended behavior that is then translated into a notation that a computational device can follow to enact that behavior. The original conception of this required developers to define behavior in

the native language of computational hardware; subsequently, stylized notations (i.e., programming languages) were created to allow developers to describe behavior in a more human-readable form that could then be mechanically translated into a computational device's native notation.

In due course, as problems have become more complex and solutions more long-lived, code needed to have a more clearly defined structure for subsequent understanding of how it should be expected to behave and what to change if different behavior was needed. With obscure relationships among different parts of the code, changes in one part can indirectly affect other parts. Behavior of the code has become harder to predict and more likely to be different than expected. As a result, more time has to be spent trying to be sure the software works correctly. After all that, the software may still work differently than users had imagined or needed. As users' needs change over time, the code becomes still more complicated due to unplanned changes and yet more difficult to get correct and then change further. Of greater concern, the time from recognizing the need for a product and having that product available for use has grown increasingly long and even then often requires users to change how they prefer to work for the solution to work effectively.

In best cases, software is developed within an enterprise that has a specific focus and experience building particular types of software. Practitioners in those enterprises have the competence (knowledge, experience, and expertise) to build such software more efficiently and effectively than those who lack such competence. In this context, a product can often be built using a mix of new and previously developed components. A conventional approach makes no overt accommodation for this, leaving it to individual developers to recognize and exploit such opportunities, or not. This is a highly undisciplined and inefficient way of leveraging past work and misses the opportunity to lay a proper foundation for building and evolving software-based products.

With the growing complexity of problems and solutions, an engineering discipline is needed for building software-based products. Engineering is the systematic application of knowledge and expertise to achieve a solution to a problem, considering properties and tradeoffs among alternatives to determine a solution that best satisfies all aspects of

the problem. In building a software-based product, the problem to be solved is often initially not fully understood or properly communicated. As potential solutions are explored, understanding of the problem improves until alternatives can be sufficiently reduced and criteria for an acceptable solution determined. Over time, as experience with similar problems increases, systematic methods arise for more clearly defining a problem at the start and reliably focusing on and then resolving among alternative viable solutions. With similar problems, less effort is needed to determine and build customized solutions to each, with a focus on having to address only important differences among them.

Perspectives on Process Variation

A defined process specifies how actions are organized to accomplish some purpose; this includes defining a partially ordered set of activities and the practices (or methods) to be used for each activity. The effectiveness of an effort differs depending on the suitability of the process used and the ability of practitioners to effectively perform that process. There are three useful perspectives on process variation that inform achieving a higher level of productivity in building a given level of product value:

- ***Performance*** – The productivity achieved by practitioners following the process
- ***Capability*** – The (minimally acceptable .. maximally achievable) range in productivity that characterizes a given process
- ***Maturity*** – The degree to which projects are consistently able to achieve a targeted level of capability in the performance of a specified process

These all will differ depending on intended product value (utility and quality) varying further due to the associated complexity and uncertainty in the problem-solution. In any case, productivity can be improved by (1) automating aspects of the process to reduce manual effort or product defects requiring rework, (2) improving the maturity of practitioners in applying the process such as through training in better use of practices, or (3) modifying the process or practices to support higher level of achievable capability.

Process and Product Quality

The "value" of a thing is its utility for a given purpose and its quality in serving that purpose. Quality is the degree to which the thing represents the "ideal" form of such a thing for a given use. Quality is ultimately a subjective discernment but, for an enterprise to prosper, it benefits from having the means to continually improve the quality of its efforts. Quality improvement involves having measurable goals for its endeavors, mechanisms for determining how well those goals are being achieved, and actions to be taken when ways are found to improve both guidance for and associated practice of endeavors.

An enterprise is concerned with two sorts of quality: process quality and product quality. ***Process quality*** is the efficiency and effectiveness (equivalent to "productivity") of an effort in achieving some envisioned result. ***Product quality*** is the degree to which a result (i.e., a product or service) approximates its achievable best form.

Process quality represents the degree to which a defined process is effective in supporting repeatably achieving an intended result. The performance of a well-defined process can be improved by improving the ability of practitioners to perform the process or by improving automated support of activities to require less effort, for a result of fewer errors and less repetition. Process quality can be improved by modifying the process in ways that will result in higher productivity for a given level of effort, such as by eliminating or streamlining activities to reduce manual effort or achieving higher levels of automation that simplify the process.

In developing a software-based product, the goal of productivity is to reduce the delay between recognizing the need for a product and the consequent effective operational use of that product. This goal persists as needs change over the useful life of the product, requiring repeated modifications. Process quality for development of a software-based product (defined in section 2.0 as "developmental" quality) is evaluated in terms of four aggregate properties: feasibility, sustainability, conformability, and verifiability.

Product quality represents the degree to which a product is suitable for its intended use. Product quality is dependent on the process with which the product is realized in that the process determines the criteria and attention given to achieving the various elements of product quality. Product quality can be improved either by increasing the effort spent on evaluating and achieving product quality goals or by enhancing the process with more effective means of addressing product quality. Product quality for a software-based product (defined in section 2.3 as "behavioral" quality) is evaluated in terms of four aggregate properties: functionality, performance, dependability, and usability.

Expending greater effort on product quality can be viewed as reducing developer productivity unless the resulting improvement in product quality is recognized as requisite to product viability, whether initially or over its useful life; the challenge is to find a proper balance between adequate quality and timely cost-effective deployment of a needed product. The process cost of improved product quality can be mitigated by improvements in the process that increase potential productivity, reducing the effort needed to achieve given product quality goals.

Two other more indirect dependencies exist between product and process quality. One dependence is between product quality and the quality of the customer's operational process: the effectiveness of that process depends on the capabilities and use of the product being a proper fit with the customer's process and practices. Another dependence is between product quality and the quality of the provider enterprise business process (i.e., measured in terms of efficient use of resources, costs, and ability to meet schedules). *{business/operational quality (time to market, cost, competence (fit to business charter), ?)}*

The Causes of Product Change

A product is built to provide capabilities that support the needs of customers in a targeted market. Although it is natural to think of a product as being the fixed realization of some associated solution to a specific problem, both the problem and its

solution are susceptible to change. A product is better viewed as a changing solution to a changing problem.

Three factors lead to a product being changed: problem-solution uncertainty, product deficiencies, and problem-solution changes. A realistic approach to development must account for each of these factors for a product to be sustained over its expected useful life. These three are all interrelated in that each of them can arise even as either of the others are being addressed (e.g., a deficiency that is the result of an incorrect resolution of some uncertainty, an identified problem-solution change that changes how a problem-solution uncertainty should be resolved, or a problem-solution uncertainty that is not recognized until a problem-solution change occurs.)

Problem-solution uncertainty is a consequence of problem-solution complexity and the limits of human capability (or quoting Dijkstra: "On our inability to do much"). Product deficiencies express when the product is found to be flawed, having defects that degrade its ability to satisfy its envisioned purpose and capabilities, or when the product as released is acceptable but incomplete. Problem-solution changes are the result of changing customer needs, including changes in operational circumstances or enabling technology.

Problem-Solution Uncertainty

A product is a particular solution to an understanding of a problem. However, based on two of the precepts of software development (from section 1.1), uncertainty is a concern in two respects: is the actual problem properly understood and which of its many potential solutions is a best fit for the problem? The essential challenge, and substantial effort, of product development is resolving problem-solution uncertainty.

An effective solution requires a reasonably correct and complete understanding of the problem and the behavior that the product must exhibit to provide the customer with the capabilities needed. Certainty is an unobtainable ideal but uncertainty has many sources potentially subject to mitigation for improved confidence in the problem and a suitable solution. Uncertainty about the problem arises from unknowns (including excluded information due to security/proprietary/competitive concerns), tacit or

overlooked assumptions, inherent complexity, miscommunications, ambiguity, and differing user or expert opinions about the problem or solution based on prior experience. These uncertainties can usually be satisfactorily resolved through customer-developer collaboration in exploring subject matter knowledge, customer objectives and practices, and circumstances related to operational context.

Uncertainties about the problem and its possible solution demands an iterative "learning" process of development through which understanding of the problem improves and may change and the nature and tradeoffs of alternative solutions are explored and comparatively evaluated. A premature or ill-founded resolution of uncertainties gives the illusion of problem-solution understanding but results in development of a deficient product.

Development of a product begins with a general conception of capabilities that the product should provide for its intended purpose. Such a conception lacks specificity in that many different products could in principle satisfy an insufficiently precise description of the problem or its solution. The process of creation is one of refining this conception by identifying and comparing alternatives in the problem and solution, considering issues of feasibility, practicality, constraints, convention, and preference imposed by the context of usage, to determine what form the product will take. The effort involved in achieving a satisfactory product is dependent on the complexity of the actual problem and the viability of creating an appropriate solution to that problem.

Achieving a good solution is expedited with competence gained from solving similar problems but uncertainty can remain when alternative solutions fail to adequately resolve problem-specific conflicts among related aspects of product quality. Uncertainty about the solution concerns what combination of development choices will result in the right functionality with appropriate quality criteria, while satisfying any extrinsic constraints on the solution. Determining the best solution entails understanding the nature of the context in which the product operates and the interactions that result, how the solution addresses problem-imposed quality criteria and solution constraints, and how the computational platform affects product behavior with different solutions. Each

of these aspects brings some degree of uncertainty that must be resolved to determine an acceptable solution.

A particular type of uncertainty relates to how understanding of the actual problem can change as development proceeds. An inaccurate understanding of the problem can result in resolving solution uncertainty that ends up as a deficiency in the product. Furthermore, the actual problem itself or recognition of possible solution alternatives can change even as the product is being developed, warranting changes in the solution to provide the customer with more effective capabilities.

Solution Deficiencies

A product is typically a satisfactory solution to the problem it addresses, seldom a perfect solution. The complexity of most software, and the challenges entailed in its development, means that it is going to have undiscovered defects. Furthermore, tradeoffs made among quality factors can lead to unforeseen emergent effects that could be improved with changes in the solution to achieve a closer fit to the problem. (quality factors are not absolute, every solution is an approximation to an envisioned ideal)

In addition, a sound practice of software development is to build a product that is a useful but incomplete solution to the full problem being addressed. The practice is to incrementally develop an effective solution to a partial problem, incrementally addressing more aspects of the problem for an augmented solution in a series of product releases. The initial release provides a coherent set of capabilities but subsequent releases improve or add capabilities that the customer needed. Developing a product in this incremental way may result in the observable behavior of each release of the product purposefully differing and, in some cases, expand the market to which the product applies.

A customer's use of a product sometimes reveals that their endeavors would be improved with changes in product capabilities. Such changes may be needed for the customer to change how they perform an endeavor rather than simply to better support existing practices. This experience may be considered by the customer as being a

deficiency in the product when it is actually a needed change in the problem or solution that the product has been properly built to realize.

Problem-Solution Changes

The suitability of a product can degrade over time unless the product is changed to accommodate changing user needs or to fit changing technology or changing operational circumstances. This mirrors the issues of problem-solution uncertainty in deciding how the product needs to change. Furthermore, such change can occur even as a product is being developed, with implications for reconsidering how aspects of uncertainty have already been resolved.

Recognizing the potential for change, building a product that is a fine solution to a properly understood problem is a good start but not the end. Changes can occur even as a product is being developed due to how uncertainties and tradeoffs in the problem or its solution are resolved or changes in the operational context in which the product is to be used. Over its expected useful life, changes in context will repeatedly occur in ways that will require changing the product so it continues to serve its intended purpose.

Modifying an existing product is substantially the same as dropping into the middle of initial development, recognizing that prior resolutions of some uncertainties are not easily modified. When a customer's needs or operational circumstances change, motivating reconsideration of the problem-solution for the product, the same considerations of uncertainty and tradeoffs of the original development continue to apply. This will be an easier task if past uncertainties were resolved in expectation of these changes.

(relation to product deficiencies: changed perception of needs) In using a product, customers often discover improvements in how their enterprise operates that would benefit from changes in the product.

A particular concern in both creating and modifying a product is maintaining the ability to make further changes as customer and market needs change. This requires judgement concerning the uncertainty regarding what types of changes are most likely. To be able to anticipate such changes, information needs to be provided, by problem

and solution experts, as to what changes are more likely to be encountered during the expected life of the product.

Failing to identify likely potential changes is equivalent to assuming that nothing about a product will ever need to be changed. An informed but imperfect projection of potential change is generally more accurate than an (implicit) assumption of no change. Developing a product without considering likely future changes results in a product for which any change is unpredictably difficult and may be unnecessarily costly to make.

The degree of certainty concerning the nature of specific changes guides how much effort should be taken to prepare for those changes, recognizing that some potential changes may never occur or may take a form that differs from current expectations. Considering potential future change does not necessarily incur additional cost in that software developers often explore alternatives in order to resolve uncertainties and augment missing information but then discard resulting information and insights that might save time when future changes do occur. More information about potential future changes serve to better inform developer explorations and allow them to better organize their work so as to localize likely changes.

The Duality of Product Change and Market Diversity

A fourth cause of product "change" (or more broadly, the rationale for the existence of multiple similar products) is diversity of needs. Product change concerns needs changing over time whereas market diversity concerns the co-existence of similar but differing needs within a market.

Product change and market diversity are both manifestations of *variability*. Product variability represents the different realizations of a product that result from problem-solution uncertainty, deficiency, and change. Every customer is susceptible to the implications of product variability.

Market variability reflects essential differences in customers' needs, including as market composition changes over time. A simple market is one in which only product variability, not market variability, is a concern. The implication of variability in a more

diverse coherent market is the need to provide multiple products to address the needs of the market as a whole.

Customers in a coherent market, by definition, will have similar needs but those needs may nevertheless differ sufficiently to preclude both customers being properly satisfied by a single product. The idea of diversity is that different products are needed either because they address different needs or because they address the same needs differently. Two products are considered to be "similar" to the degree that they exhibit similar behavior (with products that exhibit essentially identical behavior being considered "equivalent"). Such diversity may exist among customers in a market or even for a single customer facing diverse operational circumstances (e.g., differing legal or regulatory constraints, computational configurations, or ecosystem compositions).

Any given product realization may be the result of either product or market variability or both as the composition of simple markets within a coherent market changes due to the needs of different customers converging or diverging over time. (Whether a particular product realization is characterized as a "version" of a previously existing product versus as a different product is only a matter of whether it is addressing the changed needs of a single customer or the differing needs of different customers. Both cases exemplify variability.)

Resolving either product or market variability can be viewed as choosing among a set of alternative realizable products. The product that best fits a customer's current needs may differ from the product that best fits their needs at some future time. Similarly, the product that best fits one customer's needs may differ from the product that best fits another customer's needs. Both types of variability can be expressed in the form of a product family.

The Concept of a Domain

A ***domain*** is the "product" of an engineering effort to develop a product family that will meet the needs of a targeted market and an associated means for deriving customized instance products from it. It is the platform that a manufacturing effort can use to build a customized product for a designated customer in the targeted market.

In building a product, a developer faces a set of decisions about the problem and its solution that determine what product is being built for a given purpose. For a product family, the abstraction that characterizes the family resolves many of the essential decisions (in that these have the same resolution for all instances). Remaining decisions are limited to those that concern the differences that distinguish among instances of the family. Each decision that is resolved eliminates some of the instances that could have been built. With answers to all of these decisions, a competent developer will know enough to build the instance of the product family that is a best fit to the problem being solved. (Alternatively, as will be explored further in chapter 4, a concrete representation of a family can be created that enables the mechanical derivation of an instance that matches these decisions.)

Customization: From Family to Instance

A family is a set of instances that are characterized as being similar. A set of decisions are associated with the family that are sufficient to distinguish among its instances. Each decision reduces the membership of the family to a subset of instances. After decisions are resolved sufficiently to describe a particular problem and its solution, a single instance is left as the best fit among the instances of the family as defined.

If the best resolution of all needed decisions cannot be made with due certainty, decisions can be repeatedly resolved differently so as to identify multiple instances of the family that may be a good fit to the perceived problem and solution. Those instances can then be empirically compared to help in deciding which instance is the best fit for the intended purpose.

~~The members of a product family are differentiated according to criteria that represents the ways in which the behaviors (functionality and properties) of the corresponding set of similar products can differ. A specified characteristic set of unresolved decisions concerning needed capabilities and engineering tradeoffs define the differences among instances of the product family.~~