2.5 Product Design

The product design model specifies the composition of the product including a product architecture, design rationale, and product realization. The design specifies the construction of a product that will realize the behavior specified in the product requirements. This model identifies alternate solutions that would produce the expected behavior in an operational context specified in the product environment model and the tradeoffs entailed in arriving at a preferred solution.

The product architecture element specifies structures that describe the internal organization of the product. These structures provide static, dynamic, and physical views of that organization. The conveyer for all of these views is a set of software components specified in the static view.

The design rationale element specifies the reasoning by which the product architecture is expected to realize the observable behavior specified in the product requirements model. This element characterizes alternatives considered and tradeoffs among them traceable to quality criteria and constraints. This specification includes analyses of (1) how potential changes in customer needs or technology (i.e., over the useful life of the product) would affect architecture alternatives and associated product quality and (2) how architectural alternatives comport with constraints and conventions, noting any potential changes that could enable improvements in developmental or product quality.

The product realization element specifies the building of candidate software product versions (i.e., any subset of the product architecture, composed of one or more top-level components and all referenced subordinate components) to be employed for evaluation or use on a specified computational platform and operational environment. The product can be realized with instrumentation for enhanced observability in evaluating product behavior.

Product Architecture

The product architecture element partitions the software as a whole into a set of independently realized components, defining architectural integrity upon which shared understanding can be established. This element specifies the internal structure of the product, both as a prescriptive "to-be" guide to its construction and as a descriptive "as-built" aid for explaining observed versus expected behavior and for exploring the implications of potential changes. This specification of product structure provides a basis both for predicting the degree to which the resulting product will satisfy specified behavioral quality criteria and for building subsets and alternative versions of the product using available components[1].

This element specifies a set of characteristic architectural structures for the product, organized into three views:

- A "static" view defines how the software is organized into independently realized components, including logical dependencies among them;
- A "dynamic" view defines how the software is organized into processing elements—processes and threads—that create concurrent behaviors, including coordination and communications among these elements;
- A "physical" view defines how components are to be combined and transformed into executable units for deployment onto a computational platform.

Static View

The purpose of the static view is to specify a collection of components that are sufficient in aggregate to construct software that satisfies product requirements. Any coherent subset of this view can be applied to construct limited-capability versions of a product. This view consists of two structures: a decomposition hierarchy and a dependency relation. Together, these structures are a guide for identifying the component that is uniquely responsible for any particular behavior of the product.

The decomposition hierarchy partitions the software, according to the separation of concerns principle, into components[2]. Each component corresponds to an abstraction

[1] D.L.Parnas, "Designing software for ease of extension and contraction", ACM ICSE '78: Proceedings of the 3rd international conference on Software engineering, May 1978, 264–277.

[2] D.L. Parnas, P.C. Clements, D.M. Weiss, The modular structure of complex systems, IEEE Trans. Software Engineering SE- 11 (3), 1985, 259–.266

- Behavior *{observable behavior as specified in the product requirements model}*
  - External Interface *{how the product creates behavior via the entities in its ecosystem}*
  - Information Model *{the product's model of the ecosystem's composite past/present/future information content as context for its behavior}*
  - Introspection Model *{the product's model of its past/present/future behavior for explanation and planning}*
  - Enablement *{software capabilities for common elements of behavior such as conventions for entity-tailored presentation of data and media}*
- Services *{software capabilities needed to implement other components}*
  - Data *{abstract types, structures, object management, analysis, transformation}*
  - Reasoning *{math/physical, knowledge/expertise, analytic/statistical/heuristic, and composite/fusion models}*
  - Media *{static and dynamic forms, composition, accessibility/dynamics/interaction-control}*
- Environment *{the elements of the product's operational environment}*
  - Entities *{capabilities for interacting with users, devices, and systems operating in the product's ecosystem}*
  - Platform *{the product's foundational capabilities for computation, communication, and data storage}*

Figure 2.5-1. Example Upper Levels of a Decomposition Hierarchy

that delineates its purpose and computational responsibilities, mutually exclusive to other components.

The responsibilities of each component can be either physically realized or apportioned among a subsumed set of components—each corresponding to a more limited abstraction that represents a more specialized purpose and capabilities than its parent component. A physically-realized component is implemented by one of a set of alternative modules, each satisfying the component's specified purpose but differing in the details of each module's interface, internal design, implementation of behavior and quality tradeoffs, and dependence on other components' modules.

The topmost level of a typical decomposition hierarchy may be defined to have three components: environment, behavior, and services. Each of these is recursively decomposed to form a hierarchy of increasingly specialized components (e.g., as illustrated to a second and informally suggested third level [Figure 2.5-1]).

Environment components specify the product's computational platform and components, which provide the means to interact with entities as specified in the product environment model. The platform component provides the means for computation, persistent data storage, and communication with entities in the product environment. This component encapsulates behavior of enabling devices, operating systems/hypervisor, programming languages, and associated utilities.

Each entities component serves as a proxy for interactions with associated (physical or virtual) instances of the type, specifying access to the capabilities of a given type of environment-specified entity. A component may encapsulate an entity's intrinsic capabilities—physical or virtualized—with software functionality specific to the entity type that enhances those capabilities to better support product needs. Similarly, an entity component may specify a virtualized entity type, combining multiple entities as a single composite entity or partitioning a single entity to be accessed as multiple logically-distinct entities. In any case, an entity component can enhance a corresponding entity's intrinsic capabilities with services component-provided functionality (using services components) to, for example, filter, validate, or transform received data, maintain an audit or diagnostic history, retain values for delayed or composite access, or infer missing or future content based on prior values or environment model-specified behavior.

Behavior components are responsible for realizing observable behavior as specified in the product requirements model[3]. External interface components are organized into three categories corresponding to entity type components: managing interactions with devices, managing interactions with users, and managing interactions with systems.

[3] Note that an off-the-shelf application that is to be embedded into the product may be separately implemented as a self-contained behavior component that does not depend on any other components but instead duplicates, in whatever form, the capabilities of the other components it would otherwise need.

Information model components specify the product’s awareness of ecosystem information content whereas introspection model components specify the product’s awareness of its own behavior. Enablement components specify product-specific shared services needed by other behavior components.

Services components specify specialized computational capabilities needed for the implementation of other components. The interface for a services component defines the services through which other components can employ the component’s provided capabilities. Services provided by the computational platform or as part of externally supported tools should be encapsulated with product-specific abstract interfaces to support portability across platforms, enable customizations, or limit exposure to changes in the native interfaces of those services.

The dependency relation specifies the dependencies each component has on the inclusion of other components in a product realization. The dependencies of each module, differing in their realizations of component responsibilities, may differ in which modules of other components that must be included. A dependency can be direct (by referencing a component’s interface) or indirect (only dependent on the effects of an included component’s behavior). Subsets of a product’s capabilities can be built as long as all dependencies among components are satisfied. In general, each external interface component depends on a corresponding entity component, behavior and entity components may depend on services components, and all components depend on platform components.

Dynamic View

The purpose of the dynamic view is to define how concurrent activities of the product are realized. It consists of two structures: a concurrency structure and a coordination-communications structure. These structures are revised iteratively as processes and threads are implemented in components.

The concurrency structure specifies the processes and threads that software can activate as tasks to achieve concurrent activity. A task is the runtime actualization of a process or thread definition. Each process or thread is the responsibility of a designated component for its implementation. The product requirements model specifies processes that represent the observable concurrent behavior of the product; each responsible component defines threads that dependent processes need to achieve their associated concurrent behavior.

Each process defines an autonomous activity to be initiated based on specified dynamic criteria—externally or internally triggered events, data value changes, or periodically repeated—with conditional criteria for activation, continuation, and termination and all associated actions. Each process-defined task operates exclusively within its own separately managed data space, initiating subordinate thread-defined tasks or communicating (synchronously or asynchronously) with separately initiated tasks as needed.

Each thread-defined task can be initiated as determined by and under the control of another active process- or thread-based task. For example, a process or thread may define the partitioning of a homogeneous data structure into elements, each of which is assigned to a separately initiated task to be independently processed, with the results of all of these tasks then being combined by the initiating task for a composite result.

The coordination-communications structure specifies how tasks associated with each process and thread interact with other tasks to coordinate activity and share data. This structure supports analyses of the expected and actual concurrent behavior of the product. For computational efficiency, processes or threads that are specified as being initiated in a fixed sequence or concurrently based on the same criteria or referencing the same data may be merged based on the coordination-communications structure, to be actualized as a single task[4].

Physical View

The purpose of the physical view is to define how the software product is realized from components and mapped onto a physically-realized computational platform. It consists

[4] H. Gomaa, “Structuring criteria for real time system design”, ICSE '89: Proc 11th Intl Conf on Software Engineering, May 1989, 290–301 *(Section 5).*

of three structures: a composition mapping, a computational platform structure, and a deployment mapping.

The composition mapping structure specifies how software modules are selected, configured, and composed to form object units (e.g., binary code, text content, and operational data spaces) that are then combined to form executable units. A software object is built from one or more “root” modules, along with all modules that these depend upon. A software executable is any composition of one or more software objects to implement (all or a subset of) the capabilities specified as product requirements.

The computational platform structure specifies the logical (physical or virtualized) devices and connections (required and optional) provided by each computational platform on which the product can operate. Each such device is encapsulated by a corresponding entities component module that defines its accessible capabilities.

The deployment mapping structure specifies how a specified set of executable units can be mapped—statically or dynamically—onto the elements of each of one or more targeted computational platforms. A product can be partitioned into multiple executable units for selective deployment onto different physical or virtualized elements of a computational platform.

Design Rationale

The design rationale element specifies how elements of the requirements model have influenced the product design and how potential changes would affect the design. This includes criteria used in evaluating among alternative solutions, tradeoffs made in achieving quality criteria, and interface conventions that had to be accommodated.

Key design decisions should be described with respect to requirements content. Traceability of design elements to requirements elements should be made apparent.

Additional constraints may have been imposed on the design to make an otherwise imperfect solution acceptable. Constraints that have required the exclusion of otherwise preferred alternative solutions are characterized so that those solutions can be reconsidered if constraints change over time.

Analyses of Alternatives

Tradeoff analyses concern the options developers can identify regarding alternative designs for a solution. Provider program experience building past products and expectations concerning future products can influence how best to resolve among alternatives. This may in part include the existence of software that implements aspects of a preferred approach that developers can employ in building the current product.

Any imposed conventions that lead to cost, schedule, feasibility, or quality concerns are identified, with alternatives and tradeoffs, for consideration at the requirements or customer relationship level as appropriate. Customer policies may constrain the resolution of tradeoff analyses in the architecture as well as in individual component designs. The product architecture may need to accommodate the independent customization and change of these aspects to conform to current and future customer circumstances.

Architectural Quality

Architectural quality is a composite measure of the degree to which product behavioral quality can be made to conform to requirements-specified quality criteria. Reasoning and rationale are given as to how the need to satisfy quality factors have influenced design choices, considering tradeoffs made to address conflicting factors.

The relative significance in the product requirements of the various factors provides a basis for tradeoffs among alternative solutions. This measure takes into account both the degree to which different elements of the architecture differ in their influence and effects on different quality factors and the dependencies that exist among quality factors. This is in good part a derived view of how specified product components, individually and in combination, contribute to addressing requirements-specified quality criteria.

The content of this element derives from directed expert reviews, empirical product evaluations, and product analytics model evaluations of the effects of potential design alternatives on behavioral quality factors. Based on a multi-dimensional tradeoff analysis reflecting the importance and contribution of each product component in satisfying overall quality factor criteria, rationale can be given for expecting that the product as a whole should satisfy aggregate quality criteria.

Interface Conventions

The architecture defines components that implement interface specifications defined in the product environment model for interactions with external entities. Specific interfaces must conform to conventions specified—unilaterally, collaboratively, or collectively—by the enterprises responsible for each entity. These conventions provide guidance to the project concerning the realization of interfaces required to achieve specified product behavior. Interface conventions may further restrict the form in which an interface is realized.

The product must be built to satisfy the customer- or provider-designated computational environment, ancillary tools or devices, data security protocols, monitoring and auditing practices, other organizational conventions, or governmental regulations. Interface conventions may be imposed to accommodate the constraints that are applicable in the customer's envisioned use of the product.

Interface conventions prescribe appropriate consistency across a product and among related products, specifically concerning how control and data are to be represented, according to the nature of each interface. Ideally, components that support the realization of commonly used protocols and representations will be built and shared across a program's projects as appropriate.

Interfaces are grouped into three broad categories: user interfaces, edge device interfaces, and system interfaces. Each of these are limited by the capabilities of the devices by which these interfaces are realized.

- User interfaces can differ based on the various forms of media that exist for this purpose (e.g., audio/aural, video/visual, textual, image/graphical, touch/haptic). Each logical user interface is characterized in terms of a "role" that corresponds to the particular capabilities and information that such a user needs within an enterprise. The purpose of policy is to specify the conventions by which those capabilities and information are expressed for consistency over all user interface forms, differing only due to substantive differences in expressed content and user roles in use of the product.

- Edge device interfaces are constrained by the physical interfaces provided by each device. Such specifics should be isolated within a software component that encapsulates how other components are able to access the capabilities provided by each type of device.
- System interfaces provide communications among related systems, constrained by the media through which those systems are able to exchange messages (control and data). Each type of needed communications media is encapsulated in a software component for that purpose. The content and format of messages is established by the enterprise/system primarily responsible for the associated content (ideally as a result of negotiation with developers of communicating systems).

Product Realization

The product realization element specifies the means by which a product package is created—component-associated modules selected, optionally instrumented, transformed, and composed according to the physical view structures, along with other associated operationally-useful product model elements—to be employed for evaluation or deployment. Product realization uses developmental platform mechanisms to build a deployable product that conforms to (all or a coherent subset of) the physical view of the product architecture element. The resulting product will be suitable for injection into the actual or an optionally-instrumented virtualization of the product's computational platform.

Platform Virtualization

A product is designed to operate on a specified computational platform in a specified operational environment. The product architecture encapsulates the computational platform and operational environment in components that abstract their actual behavioral capabilities. The computational platforms for development, evaluation, and operations may differ; due to these differences, each of these may be represented by an alternative module of the computational platform component.

The product may be built to run on all of these alternatives—on the actual operational platform or on a virtualized facsimile of that platform and the operational environment. During evaluations as part of development or delivery, using a facsimile can be necessary either because an actual or equivalent platform realization is unavailable or because evaluations of product behavior may need capabilities that are precluded within the actual operational platform.

Product Instrumentation

A product can be built with instrumentation that enables platform-supported monitoring and control of its operation for empirical—quantitative and qualitative—evaluations of its behavior. Such instrumentation can provide analytic and diagnostic capabilities, including data collection and profiling, spatial and temporal virtualization, operator control of computation and data values, and analyses of behavioral quality including dynamic properties.

Instrumentation can be used to track and control the progress of product functionality and to monitor and modify data values that control or derive from otherwise unobservable internal behavior. Such instrumentation can be dynamically engaged on an evaluation platform or set to record corresponding computational tracking and data change events as they occur.

Although the use of instrumentation can provide insights into the order in which processing occurs and causes of data changes, it may distort aspects of product behavior that are sensitive to computational resources availability and timing-dependent effects—potentially avoidable within a virtualized environment that can emulate resources and time.