2.5 Product Design

The product design model specifies the composition of the product including its architecture, architectural quality, 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 is augmented with evaluated tradeoffs among considered alternative solutions that would produce that behavior in an operational context specified in the product environment model.

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 physically realized components.

The design rationale element specifies criteria that influenced the form of the design including alternatives and tradeoffs considered and limitations on the solution due to constraints and conventions dictated by program or project management or by customer guidance.

The architectural quality element specifies how the design satisfies product quality criteria specified in the product requirements model. This specification is augmented with insights into how product quality would be affected by considered alternatives in the product architecture (e.g., in response to changes in customer needs or technology over the life of the product).

The product realization element specifies the means of building a deployable software product, or any executable subset conformant with the product architecture, to operate on a specified computational platform and operational environment. This provides for instrumentation of the product to support observability for evaluation or introspection of product behavior beyond such capabilities to be included in the deployed product (e.g., for operational monitoring of health, quality factors, or history).

Product Architecture

The product architecture 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 determining the implications of potential changes. It is a basis both for predicting the degree to which the resulting product will satisfy specified behavioral quality criteria and for building product versions/subsets using available components.

The product architecture partitions the software as a whole into a set of independently realized components. These components are organized into a coherent structure that gives the product architectural integrity upon which shared understanding can be established.

Each component is specified to have specific responsibilities, in providing particular capabilities needed to realize intended product behavior. A component is realized as a "module" either as software (in source or object form) or as software-encapsulated hardware (i.e., as a "virtual device" in physically-realized or software-emulated form). A module expressed as software is realized in the form of programming language constructs, defining data constructs and processing logic (sequential and concurrent) intended to realize specified computational effects in software object form. A module expressed as software-encapsulated hardware consists of hardware mechanisms and software functionality that extends its hardware capabilities and defines its interfaces with software and with other entities. A module implements its characteristic behavior by means of capabilities and connections supported by the product platform.

A collection of modules in object form are combined in accordance with the software architecture to create an executable package that, with any associated physically realized hardware, will operate in a specified computational platform and operational environment.

The specification of a software architecture comprises a set of characteristic structures, organized into three views of the architecture:

- 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 processes and threads that implement concurrent behaviors, including coordination and communications among concurrent processing 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. This view consists of two structures: a decomposition hierarchy and a dependency relation.

The decomposition hierarchy partitions the software into collections of increasingly specialized components according to similarities in their designated responsibilities. This organizes components according to similarity of purpose.

The topmost level of this hierarchy defines three categories of components: environment, behavior, and services. Each of these is further decomposed to form a hierarchy of increasingly specialized components (e.g., as in Figure 2.5-1). This hierarchy is a guide for locating the component responsible for any particular behavior of the product.

Each component is realized either as an independently constructed software *module* or as a partitioning of its responsibilities into more specialized components. Each component is uniquely responsible for a coherent ensemble of computational behavior. Each component may be expressed in alternate modules that differ in their internal design and behavioral quality tradeoffs.

A module is specified in terms of the design of its interface and its internals. A module interface is a specification of the information that the implementer of a dependent (client) module can assume is not expected to change, even as the module internals may be modified. Each module, combined with other modules that it depends upon

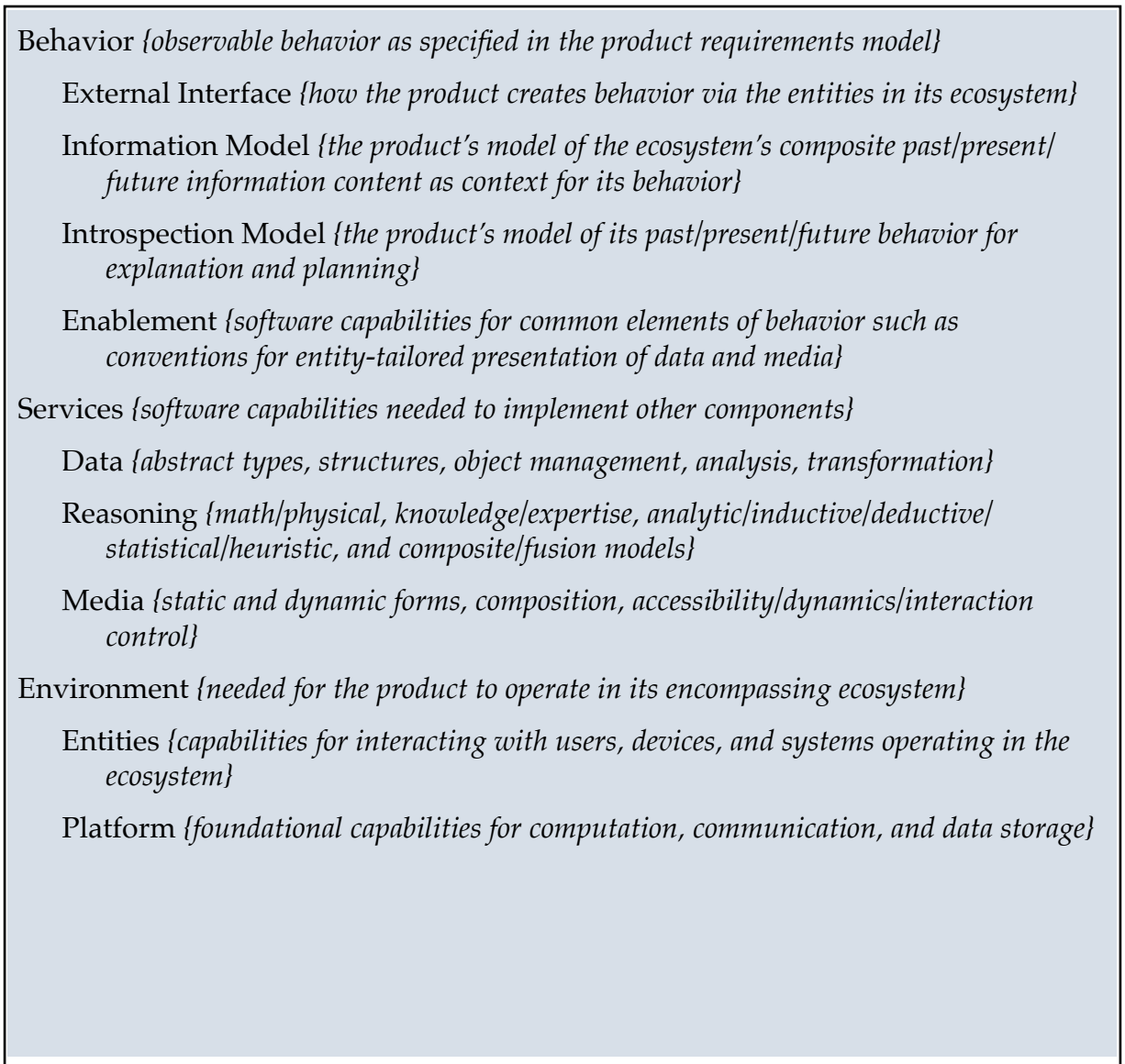

Figure 2.5-1. Notional Example of Component Hierarchy Upper Levels

according to the dependency relation, is translated into an object representation that realizes its specified conceptually coherent behavior.

Environment components specify the product's computational platform and the means for the product to interact with entities (users, devices, and systems) specified in the product environment model. The product's platform component provides the means for computation, for persistent data storage, and for communication with entities in the product's environment; this component encapsulates enabling devices, operating

systems/hypervisor, programming languages, and associated utilities. Each type of physically-realized entity of the product's ecosystem is represented as encapsulated in a software component that serves as a proxy for interactions with associated instances of the type. A component may enhance the inherent capabilities of entities it represents (e.g., retaining a history of an entity's behavior, enhancing its diagnostic and prognostic capabilities, inferring temporarily inaccessible data based on interpolation of past content). An entity may be virtualized, either presenting multiple physical entities as a single composite entity or presenting a single physical entity as multiple logically distinct entities.

Behavior components are responsible for realizing observable behavior as specified in the product requirements model. These components are organized into three categories corresponding to entity types represented in environment components: modules that manage interactions with hardware devices, modules that manage interactions with users, and modules that manage interactions with other systems.

Service components specify computational software that provides specialized capabilities with which other components can be implemented. A service component defines an interface by which other components can employ the component's provided capabilities. Each such component may be implemented by developers or may encapsulate other separately developed software (e.g., implemented as a framework in a runtime library) obtained from another provider having appropriate competence in needed capabilities.

The dependency relation is a specification of the dependencies each module has on other modules. A module's dependency can be direct (i.e., by referencing their interfaces) or indirect (i.e., dependent on effects of their independent behavior), for correct behavior. Executable subsets of a product's capabilities can be built as long as all dependencies of included components are satisfied. In general, each behavior component depends on a corresponding entity component, both behavior and entity components depend selectively on services components, and all depend on platform components.

Dynamic View

The purpose of the dynamic view is to define how concurrent activities of the product are accomplished. The dynamic view consists of two structures: a concurrency structure and a coordination-communications structure.

The concurrency structure specifies the processes and threads that software can activate as tasks to achieve concurrent activity. Each task is the runtime actualization of a process or thread definition. Each process or thread is associated with a responsible component for its implementation.

Each process defines an autonomous activity to be initiated based on specified dynamic criteria (e.g., externally or internally triggered events, data value changes, or periodically, each with conditional constraints). Each process-based task defines and operates exclusively within its own separately managed data space, initiating subordinate thread-defined tasks or communicating with separately initiated tasks as needed.

Each thread defines tasks that 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.

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.

Physical View

The physical view consists of three structures: a composition mapping, a computational platform structure, and a deployment mapping. This view defines how the software product is realized from components and mapped onto a physically-realized

computational platform. The specifics of the computational platform is defined here, subject to constraints imposed through the product requirements model.

The composition mapping structure specifies how software modules are configured and composed to form object units 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) capabilities as specified by product requirements.

The computational platform structure specifies the (required and optional) (physical) devices and connections that any platform on which the product can operate will provide.

The deployment mapping specifies how executable units are mapped, statically or dynamically, onto computational devices in a targeted computational platform.

Design Rationale

The design rationale element specifies the factors on which the design is based, in particular how prescribed provider and customer polices have influenced the product design and how potential changes in these policies would affect the design. Constraints define guidance necessary to satisfy legal, regulatory, or technical standards established by governmental or industry authority. Conventions define factors that ensure consistency across related projects of a provider or customer program and corresponding products.

Analyses of Alternatives

Design tradeoff analyses concern the options developers have regarding alternative ways of implementing a solution. The program based on experience building past products and expectations concerning future products provides guidance on how such options should be resolved. This may in part include the existence of software that implements essential aspects of preferred approaches that developers should employ in building their product.

Requirements-specified Constraints

The product requirements model specifies imposed constraints on the resolution of product design alternatives. Any constraints that lead to cost or feasibility concerns would need to be identified, with alternatives and tradeoffs, for consideration at the requirements or customer relationship level as appropriate.

These constraints and implications should be noted as part of design rationale, particularly those that are decisive in excluding otherwise preferred alternative resolutions. In addition, for any unconstrained design decisions, the implications of potential future restriction or standards changes should be similarly noted. Any alternatives may need to be reconsidered if constraints are modified in the future.

Customer policies may constrain the resolution of tradeoff analyses in the architecture as well as in individual component designs. These constraints may designate interface conventions associated with tools and technologies or systems that are related in some way with the enterprise's envisioned use of the product. These may entail expected use of the customer's computational environment, particular commercial tools or devices, data security protocols, monitoring and auditing practices, other organizational conventions, or governmental regulations that the product must be built to satisfy. The product architecture must accommodate the independent customization and change of these aspects to suit current and future customer circumstances.

Interface Conventions

Interface specifications for interactions with external entities are defined in the product environment model; interface conventions may limit the realization of those specifications. Specific interfaces must conform to conventions imposed by the authorities responsible for each entity. These conventions provide guidance to the project concerning the design of interfaces that are its responsibility.

These conventions prescribe appropriate consistency 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 associated

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 broadly constrained 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, video, textual, graphical, touch). Each logical user interface is characterized in terms of a "role" that corresponds to the particular capabilities and information that such a user needs. The purpose of policy is to specify the conventions by which those capabilities and information are expressed for consistency over all user interfaces, differing only due to substantive differences in their use of the product.

Edge device interfaces are constrained by the physical interfaces provided by each device but such specifics are isolated within a software component that encapsulates how other components are able to access the capabilities provided by each device (or category of device).

Communications among related systems is constrained by the media by 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 system primarily responsible for the associated content (optionally as a result of negotiation between developers of communicating systems).

Architectural Quality

The architectural quality element specifies the reasoning and rationale by which the product has been shown to satisfy the behavioral quality factors as specified in the product requirements model and how these have constrained design alternatives.

(describe nature of multi-dimensional tradeoff analyses, need for criteria for multi-factor tradeoffs for best fit to needs (is this redundant to sys eng or analytics content? use radar chart for each major quality factor to show satisfaction relative to subjective importance of each?)

Product Realization

The product realization element specifies the mechanisms, provided as part of the developmental platform, by which product component instances are selected and transformed into optionally instrumented object form and packaged to create an operable product. The resulting product will express a coherent subset of the product architecture and will be suitable for injection into a realization (actual or optionally instrumented facsimile) of the product's operational environment.

Platform Emulation

A product is designed to operate on a specified operational platform. The operational platform may differ from the developmental platform. The design encapsulates the platform in a component that abstracts the actual behavior of the operational platform. The product may be built to run either on the actual operational platform or on an extended or emulated facsimile of that platform. During development, using a facsimile is necessary either because the actual-equivalent platform is not available or because evaluations of product behavior may need additional capabilities that are not provided by the operational platform.

Product Instrumentation

A product can be built with instrumentation that supports its evaluation. The operation of an instrumented product can be monitored and controlled as an aid to evaluating its behavior in operation. Such instrumentation can provide analytic and diagnostic capabilities, including data profiling, spatial and temporal virtualization, operator control of computation and data values, and analyses of 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 internal behavior. Such instrumentation can be dynamically engaged from within a testing environment or set up to record corresponding computation tracking and data change events as they occur. Although the use of instrumentation can provide insights into the order in which processing occurs and the causes of data changes, it may distort aspects of product behavior that depend on the use of resources including time-dependent effects.