2.5 Product Design

The product design model specifies the composition of the product including the product architecture, architectural verification, design rationale, and product realization. This model defines how the capabilities specified in the product requirements specification are realized and the means by which the product interacts with its operational environment as defined in the product environment specification. This model is augmented with a constraints and conventions specification that imposes limits on the realization of product requirements based on program/project management and customer guidance.

The product realization specifies the means for building a software executable that will operate on a specified computational platform in a specified operational environment. This includes the means to build a subset of a specified product, even to the level of building the functionality of a single component.

Many different products, reflecting differing tradeoffs, could be built that would satisfy a given product requirements specification. The product design model identifies alternatives in how a product can be built and how these alternatives are evaluated so as to realize a best fit to the behavior specified by the requirements model.

The product design model defines the elements and dependencies that are sufficient to implement a product. This model specifies an architecture, with policies and conventions, to guide the realization of needed components. Based on the architecture and conformant components, this model specifies the means by which a product that satisfies the product requirements specification, or any subset thereof, can be realized.

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 organizes the software into a set of independently created components. A coherent structure realizing coherent architectural integrity in the form of a well-conceived set of interdependent software components provides the medium for a shared understanding of the software as a whole. Each component has specified responsibility in realizing the behavior of a product.

Each component is specified to provide particular capabilities. A component is realized either as software or as software-encapsulated hardware. Each software component is realized as a "module" expressed 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 collection of modules in object form are combined in accordance with the software architecture to create executable software appropriate to a specified operational platform and environment. A software-encapsulated hardware component is realized as a virtual device consisting of software functionality that defines how it is accessed by other software and hardware (physically realized or software emulated) mechanisms that implement its essential characteristic behavior within connections provided by the product's computational platform.

A software architecture is specified from three different perspectives. Each of these perspectives is described in a corresponding view, each view being expressed in the form of multiple defining structures:

- A "static" view defines how the software is organized into independently implemented software components
- A "dynamic" view defines processes and threads that specify how the software implements concurrent behaviors including coordination and communications among concurrent processing elements
- A "physical" view defines how software components are combined and transformed into executable units for deployment into 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. Components are implemented as *modules*. Each component may be expressed in alternate modules that differ in their realization of behavioral quality factors.

The decomposition hierarchy organizes components according to similarity of purpose. Each component is realized either as an independently constructed software module or as a partitioning of its responsibilities into more specialized components. This hierarchy is a guide for locating the module responsible for any particular behavior of the product.

A component hierarchy partitions the software to be built into collections of increasingly specialized components according to similarities in the specific responsibilities assigned to each. 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 (Figure 2.5-1).

Each module is uniquely responsible for a coherent ensemble of computational behavior. Combined with other modules that it depends upon, according to the dependency structure, it is translated into an object representation that realizes its specified conceptually coherent behavior. 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.

Environment components specify the means for interactions with the product's specified operational environment. This includes the product's operational platform and the entities (users, devices, and systems) that are accessible within the product's ecosystem. The product's platform provides the means for computation, for persistent data storage, and for communication with entities in the product's environment; this component encapsulates devices, operating systems/hypervisor, programming

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 retrospective and prospective behavior}*
- 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/tranformation}*
- Reasoning *{math/physical, knowledge/expertise, deductive/heuristic, and hybrid models}*
- Media *{static and dynamic forms, composition, accessibility/dynamics/interaction control}*

Environment *{needed for the product to operate in its encompassing ecosystem}*

- Entity *{capabilities for interacting with users, devices, and systems operating in the ecosystem}*
- Platform *{foundational capabilities for computation, communication, and data storage}*

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

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 entities. A component may enhance the inherent capabilities of the 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 product requirements. These modules 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 provide 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, either directly (i.e., by referencing their interfaces) or indirectly (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 types 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 process or thread is associated with a responsible component for its implementation.

Each instance of a process is an autonomous activity initiated based on specified dynamic criteria (e.g., externally or internally triggered events, data value changes, or periodically, each with conditional constraints). Each process instance defines and

operates exclusively within its own separately managed data space, initiating subordinate thread-defined tasks or communicating with separately initiated tasks as needed.

Instances of a specified thread 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 partition 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 concurrently active processes and threads interact to coordinate activity and share data.

For computational efficiency, process or thread instances that are specified as being initiated concurrently based on the same criteria or referencing the same data may be combined based on the coordination-communications structure to be implemented as a single runtime 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 a single module, along with all modules that it depends upon. Software executables are any composition of one or more software objects to implement all or a subset of the capabilities specified by product requirements.

The computational platform structure specifies the physical devices and connections that comprise a computational platform as the product's target computing environment.

The deployment mapping specifies how executable units are mapped onto computational devices in the target computing environment.

Architectural Quality

The architectural quality element specifies the reasoning and rationale by which the product is known to satisfy the behavioral qualities as specified in the product requirements model.

(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?)

Behavioral Quality

Behavioral quality factors constrain product design alternatives.

Design Rationale

Design rationale specifies 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.

Project management, consistent with program guidance, defines the methods, practices, and conventions to be used. A program may impose policies with the intent of achieving consistency among its projects based on similarities in their chartered objectives.

Customer policies reflect consideration of how the product will fit into the customer environment (e.g., indicating expected use of existing or planned facilities or continued use of familiar practices). These policies affect the platform on which the product is built to operate and the practices applied in building it. Prescribed policies may influence the resolution of tradeoff analyses in the architecture as well as in individual component designs.

These policies may constrain developer alternatives that would otherwise be left to engineering judgement based on identification and analyses of alternatives or other past experience. Examples of potential policies that could be imposed include:

- designated use of specific COTS products (tools, data storage, computational platform)
- computational platform standards and conventions regarding messaging, transaction protocols, and logging practices
- availability guidance involving fault handling and degraded processing practices
- platform management protocols (startup/shutdown, hardware health and diagnostic conventions, concurrency techniques)
- Training practices
- Real-time partitioning (-> OS, processor/virtualization, localization & partitioning/allocation of resources)
- platform integrity guidance regarding system, transactional, and data access security practices
- Safety partitioning (hdw partitioning, direct hardware controls/overrides, single authority designation)
- Methods of device interconnection (e.g., distributed servers versus point-to-point)
- Data distribution (e.g., buffering) services
- Hardware-software configuration, compatibility, and reconfiguration/ reprogramming
- Accommodations for future computational environment evolution

Analyses of Design 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.

Conventions and Constraints

Customer-imposed Constraints

The customer needs specification (in the delivery model) identify constraints arising from customer enterprise conventions to which the product will need to conform. These constraints are addressed in the product requirements model. These 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).

Product Realization

The product realization element specifies the mechanisms, provided as part of the development computational platform, by which product components are 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 specified environment.

Product Instrumentation

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.