4.2 An Abstraction-Based Environment

An abstraction-based environment is a reuse-based software product factory.[1] Such an environment, conceived in the 1980's as a precursor to the producibility vision, provides a means to generate a software product corresponding to a specification of its requirements expressed in a standardized semi-formal notation.[2]

The motivation for using a semi-formal notation was to support creating a precise, concise, unambiguous, complete, and consistent specification for a software product. This notation was built upon and logically organized a set of abstractions corresponding to recurring computational elements seen as being common to many software products being developed. A product to be built was envisioned as being composed of customized instances of such computational elements.

To build a particular product, its envisioned elements are each identified in the product's specification as an instance of a corresponding abstraction and elaborated with associated information that is needed to enable a concrete realization of that element. This information corresponds to element-related decisions that the developer must resolve in order to build a product that provides a solution to the given problem.

Such realization is enabled with the representation of each abstraction by an adaptable component, as the medium for systematic reuse. Information associated with an element in the requirements is then applied to its abstraction's adaptable component to derive a concrete realization of the element. The resulting set of realized elements are then composed to form the envisioned software product, according to an underlying standardized design that determines how abstraction-categorized elements go together to form a coherent whole.[3]

---

[1] G H Campbell, *Abstraction-based Environments*, Software A & E, Arlington, VA, 1988. <www.domain-specific.com/PDFfiles/ABE-Spectrum.pdf>

[2] K.L.Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Application", IEEE Trans on Sw Eng SE-6(1), Jan 1980, 2-13.

[3]D.L.Parnas, P.C.Clements, D.M.Weiss, "The Modular Structure of Complex Systems", *IEEE Transactions on Software Engineering* SE-11 (3), March 1985, 259-266.

With this approach, non-software elements of a software product can also be built with adaptable components for requirements-based derivation and composition of customized parts (e.g., to derive customized design specifications, deployment and evaluation materials, or user documentation and training materials that are consistent with software content derived based on the same requirements information).

In all cases, systematic reuse based on a framework of abstractions defined by adaptable components reduces the cost of development, even accounting for the investment needed to build and evolve the supporting environment. Ameliorating this further, an abstraction-based environment—as a software product in its own right—can be built as a composition of many of the same abstractions and associated adaptable components that it uses to build other products and is further the basis for a family of such environments.

The Nature of an Abstraction-based Environment

As noted above, the essential concepts of an abstraction-based environment are:

- A requirements notation organized upon the abstractions that are sufficient to specify the elements of any software product that can be built.
- Information associated with each abstraction that the developer must provide for its instantiation as parts of a given product.
- An adaptable component, associated with each abstraction as its basis for systematic reuse, with which developer-provided information can be used to derive customized parts for the requirements-described product.
- An underlying product design that specifies the architecture and additional components needed to construct a product, consistent with the concepts of the requirements notation, with elements that are optionally included or customized based on requirements content.
- A simple process by which developers specify requirements, to then derive and evaluate a product with respect to those requirements

An abstraction-based environment is iteratively built to allow for incremental improvements and changes in the scope of products it can be used to build:

- Identify products that should be buildable with the environment.
- Identify abstractions with associated information that are sufficient to describe from a user perspective each element found in identified buildable products.
- Build an adaptable component that encompasses all content needed to derive customized instances of each abstraction based on associated information.
- Organize abstractions into a product design structure that reflects how the corresponding elements occur in the identified buildable product.
- Define a requirements notation and provide software that allows a developer to select and add information for each abstraction following the abstraction-organized product structure to describe the content of an envisioned product.
- Apply a completed requirements specification (i.e., product model) to the set of abstraction-linked adaptable components to derive customized parts that are combined following the product design to form a software product.
- Evaluate the software product against the requirements specification and repeat the process as needed to refine or modify the product.

During development of a product, its requirements specification is evaluated for consistency and completeness:

- Requirements may be constrained by limitations in the available components, to be expressed in developer guidance and imposed during specification of information or exposed during the verification and generation of the product.
- Criteria are applied for identifying any incompleteness in the specification and prevent inconsistency in content among elements. Support for an incomplete specification to result in a partial product may be provided if inconsistent elements can be resolved in some fashion or disregarded.

Capabilities of an Abstraction-based Environment

The capabilities of an abstraction-based environment are realized through two structures—external and internal. The external structure is the view presented to a developer as the activities to be performed in building a product:

- A product specification activity that supports the developer in elaborating abstractions with associated information to create a product model that describes the elements of the product to be built,
- A product generation activity that, using adaptable components, derives a customized product that conforms to the specified product model,
- A product evaluation activity that includes validation of the product model to customer needs and verification of the generated product to the product model.
- A product delivery activity for deploying the product into operational use.

The internal structure is the underlying design (the architecture and associated components) of the environment as a software product and, with limited differences, of any buildable product:

- A product behavior level – the conceptual capabilities exhibited in the external operation of the product.
- A software services level – generic services needed to implement other software components.
- A virtual hardware level – services for access to hardware capabilities (a computational platform and interface devices) by means of which product behavior is effected.

The Spectrum Abstraction-Based Environment

The Spectrum environment[4] was built in the mid-1980's as a formative reuse-based software product factory. The dual process of constructing Spectrum and using it to

[4] Campbell, *Abstraction-Based Environments*, 4-6.

build software products served as a direct precursor to the Synthesis and successor DsE methodology.

The organizing premise of Spectrum was that the requirements specification for a product could take the form of a stylized presentation of developer decisions that had to be made in building a product. A completed specification was sufficient to enable the instantiation of an adaptable design and associated set of adaptable components. The same adaptable design and most components, as applicable, were used both in building the Spectrum environment and in building any product that could be described using provided requirements specification concepts.

The adaptable design was instantiated based on a product's specification. The resulting design identified a relevant subset of adaptable components, also customized according to the specification. The resulting modules—instantiated components—were composed according to the instantiated design to realize a product customized to the specification.

Product Development in Spectrum

The Spectrum environment was a minimally essential precursor [Figure 4.2-1] to the generalized DsE product manufacturing model (refer to Figure 3.3-2). It directly supported three activities of an abstraction-based environment:

1. Product specification – Creating a description, in terms of supported computational constructs, of a product that would fit customer needs
2. Product generation – Generating a product based on the product specification using an associated set of adaptable components
3. Product verification – Testing of the generated product, informed by review of the generated product model, to identify any flaws

The Spectrum environment did not include any automated support for model validation or product delivery activities and only provided support for static analyses of product model content with testing of a generated product performed separately.

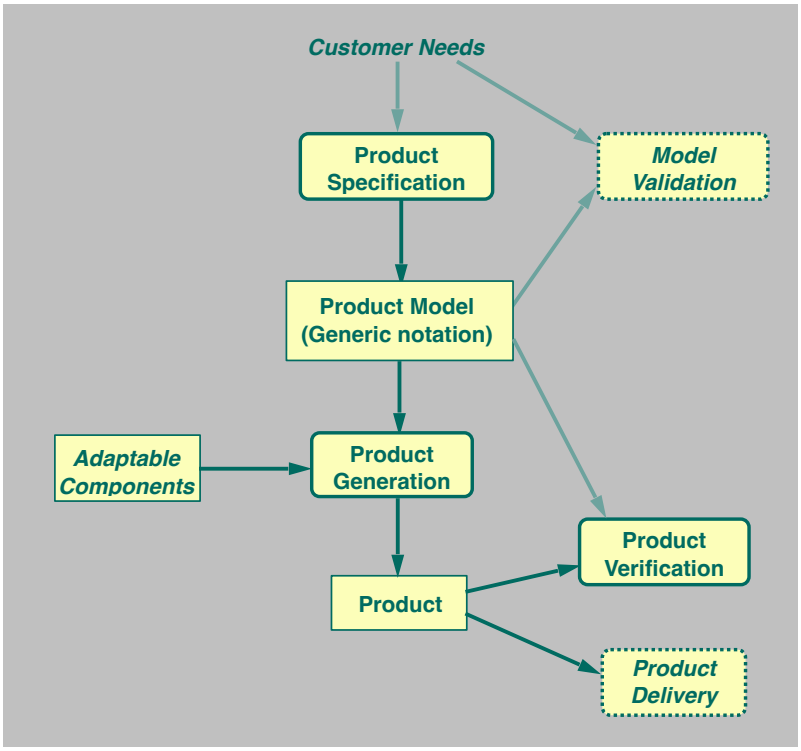

Figure 4.2-1: Building a Software Product in Spectrum

The Internal Structure of Spectrum and Its Generated Products

Spectrum, as an abstraction-based environment, supported the concept of a "generic" family of software products. (The capabilities of Spectrum were limited due both to its lacking a well-defined scope of applicability as a domain and to its being built on the limited capabilities of 1980's computer technology. The concept of a domain-specific layer above the Spectrum specification layer was anticipated but not realized in the environment as built.)

Products that could be built with Spectrum were implicitly determined "bottom-up", limited by the abstractions and associated realizations that the Spectrum product model supported. If a product could be described in terms of these abstractions and their realizations, it could be built with Spectrum; otherwise, this required modifying the Spectrum model by adding or enhancing its abstractions or their realizations.

The Spectrum product and any product built with it was composed of instances of abstractions defined by components in a 3-level architectural hierarchy [Figure 4.2-2]:

- application software *{product behavior}*
  - function definition (world model, external interface)
  - shared functionality (user state, user assistance, interaction conventions, common specification elements)
- system software *{generic services}*
  - data abstraction (application data types, semantic data storage)
  - logic abstraction (knowledge representation and inferencing, strategy control)
  - user interfaces (textual documents, structured forms, graphical forms, interaction control)
  - code generation (modules, component instantiation, product configuration)
- hardware hiding *{virtual computer and devices}*
  - a computer (data types, sequential processing, concurrency, communication)
  - a set of logical devices (user displays, printers, data storage, network access)

The hardware and software services abstractions were instantiated to build both applications and Spectrum itself. Behavior abstraction instances differed according to each product's purpose.

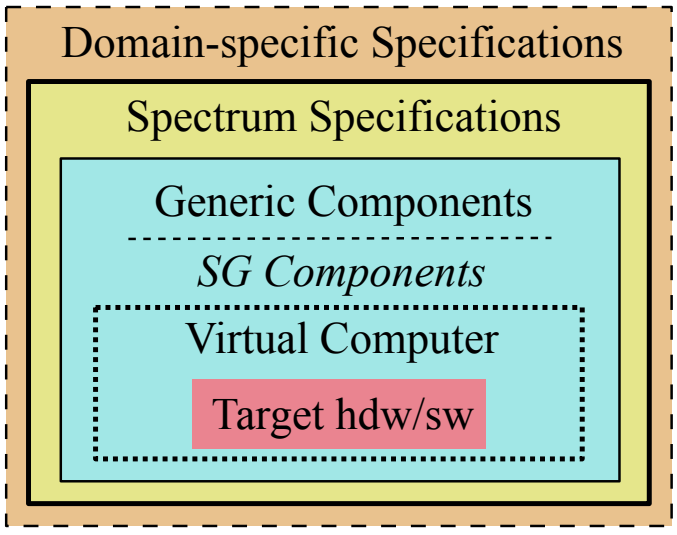

Figure 4.2-2 Levels of Product Representation in Spectrum (to be revised)

The Structure of a Spectrum Product Specification

A product was described in Spectrum from two perspectives: (1) a "world model" specification of the product's information content, describing the static and dynamic nature of the world in which the product would operate, and (2) an "external interface" specification of the mechanisms that the product was to use to obtain information for maintaining its world model as a basis for determining what actions it would take using associated virtual hardware mechanisms.

World Model Concepts

A Spectrum world model specified the information that the product needed about the environment in which it was to operate. This represented its information space corresponding to its natural physical or virtual environment and associated entities.

The world model was realized by an active semantic data model[5] having 3 elements:

- Problem-specific abstract data types.
- An object-oriented data model defining an inheritance hierarchy of object classes and typed value and relation attributes of associated objects.
- Strategies associated with classes and object attributes for determining data model content, based on user or other device input data, computations, and inferred class memberships and attribute values.

Strategies could be associated with a class—controlling the membership of classes and subclasses by creating, reclassifying, or removing instance objects based on an object's attribute values. By this mechanism, an object could become a member of multiple subclasses at any level, reflecting a proper view of an object's actual complexity.

Similarly, strategies could be associated with attributes of objects in a class—determining how the value of an attribute was determined for an object based on the values of its own and related object's attributes. Attributes could represent a data value of an object or a relation with other objects. Strategies operated automatically to

---

[5] Michael Hammer and Dennis McLeod, "Database Description with SDM: A Semantic Database Model", ACM Transactions on Database Systems 6:3, Sep. 1981, 351-386.

maintain data model consistency as attribute content of an object or of its related objects changed.

External Interface Concepts

The Spectrum external interface of a product was specified as a set of "logical" devices, each being realized by means of an identified physical device (i.e., user displays, external data storage, or network-based communications).

A logical user display device was defined for each of a set of user roles. Each user role corresponded to a set of window-based displays that presented information represented in the world model. Each display presented the information content of objects in a designated world model class and of other objects accessible via its relation attributes. Information was displayed in any of several user interface formulations (e.g., data-customized textual documents and various structured forms such as tables and lists). The customized content of each particular display was generated to reflect the content and structure of the world model information being displayed. User displays were managed according to a set of interaction strategies for selection, sequencing, and embedded nesting of display forms and dynamically updated as world model content changed.

A logical network access device provided the means to send and receive messages corresponding selectively to the content of objects in a specified world model class including content of other objects linked to them by relation attributes. Similarly, a logical data storage device provided the means to exchange data stored in the world model with an external relational database.