Glossary

abstract class A class that defines a partial implementation for a set of objects.

abstraction (1) An abstraction (noun) is a description of something that omits some details that are not relevant to the purpose of the abstraction; the converse of *refinement*. Types, collaborations, and action specs are different kinds of abstraction. (2) Abstract (verb) means to create an abstraction; also called *generalize*, *specify*, and sometimes *analyze*.

action occurrence A related set of changes of states in a group of objects between two specific points in time. An action occurrence may abstract an entire series of interactions and smaller changes.

action spec A specification of an action type. An action spec characterizes the effects of the occurrences on the states of the participating objects (for example, with a postcondition).

Actions can be joint (use cases): They abstract multiple interactions and specific protocols for information exchange and describe the net effect on all participants and the summary of information exchanged.

Actions can also be localized, in which case they are also called *operations*. An operation is a one-sided specification of an action focused entirely on a single object and how it responds to a request, without regard to the initiator of that request.

action type The set of action occurrences that conform to a given action spec. A particular action occurrence may belong to many action types.

architectural style An architectural style (or *type*) defines a consistent set of architectural elements, patterns, rules for using them, and stereotype or other notational shorthand for expressing their use, all within a package.

architecture (system) The architecture of a system consists of the structure(s) of its parts (including design-time, test-time, and runtime hardware and software parts), the nature and relevant externally visible properties of those parts (modules with interfaces, hardware units, and objects), and the relationships and constraints between them (there are a great many possibly interesting such relationships).

architecture The set of design decisions and rules about any system (or smaller component) that keeps its implementors and maintainers from exercising needless creativity. Every refinement can have such architectural rules.

architecture implementation(s) The way each category of connector works internally, including the protocol of interactions between ports. Component or object-oriented frameworks are effective ways to *implement* an architecture.

association A pair of attributes that are inverses of each other, usually drawn as a line connecting two types.

attribute A named property of an object whose value describes information about the object. An attribute's value is itself the identity of an object. In software, an attribute can represent stored or computable information. An attribute is part of a model used to help describe its object's behavior and need not be implemented directly by a designer.

class (1) A language-specific construct defining the implementation template for a set of objects, the types it implements, and the other classes or types it uses in its implementation (including by class inheritance). (2) An implementation concept that defines the stored data and associated procedures for manipulating instances of the class; the implementation construct can be mapped to OO languages and to procedural and even assembly language.

collaboration A set of related actions between typed objects playing defined roles in the collaboration; these actions are defined in terms of a common type model of the objects involved. A collaboration is frequently a refinement of a single abstract action or a design to maintain an invariant between some objects.

collaboration spec A collaboration is specified by the list of actions between the collaborators, an optional list of actions considered "outside" the collaboration, action specs, static and effect invariants that may apply to either set of actions, and an optional sequence constraint on the set of actions.

component (general) A coherent package of software artifacts that can be independently developed and delivered as a unit and that can be composed, unchanged, with other components to build something larger.

component (in code) A coherent package of software implementation that (a) has explicit and well-specified interfaces for the services it provides; (b) has explicit and well-specified interfaces for services it expects from others; and (c) can be composed with other components, perhaps customizing some of their properties, without modifying the components themselves. As a consequence of these properties, a component can be independently developed, delivered, and deployed as a unit.

component architecture type, or style The categories of connector that are permitted between components, what each of them does, and the rules and constraints on their use. Some (unary) connector types can even be used to define standard infrastructure services that will always be provided by the environment.

component-based design The mind-set, science, and art of building with and for components and ensuring that the result of plugging components together has the expected effect.

component-based development (CBD) An approach to software development in which all artifacts—from executable code to interface specifications, architectures, and business models, and scaling from complete applications and systems down to individual components—can be built by assembling, adapting, and "wiring" together existing components into a variety of different configurations.

component instance The object, set of objects, or predetermined configuration of such a set of objects that is the runtime manifestation of a component when composed within a particular application.

component spec A specification of the external behavior of a component, covering the services provided and required and the underlying component technology.

components Units of software that can be plugged in to a wide variety of others. They range in scale from small user-interface widgets to large transaction-processing applications.

conformance One behavioral description conforms to another if (and only if) any objects that behave as described by one are also behaving as described by the other (given a mapping between the two descriptions). A *conformance* is a relationship between the two descriptions, accompanied by a *justification* that includes the mapping between them and rationale for choices made. Refinement and conformance form the basis of traceability and document the answer to the question, Why is this design done thus?

connectors The connections between ports that build a collection of components into a software product (or larger component). A connector imposes role-specific constraints on the ports that it connects and can be refined to particular interaction protocols that implement the joint action.

convenience attribute A redundant attribute (possibly parameterized) that is introduced to simplify the specification of actions or invariants—for example, age defined as well as birthday.

dialect A package that contains a useful and agreed-on set of mutually consistent stereotypes, together defining a particular "dialect" of the modeling language. All modeling work is done in the context of selected dialect(s).

dictionary The collected set of definitions of modeling constructs; the definitions must include not only the formal modeling and specification bits (relating the formal names and symbols to each other) but also the (usually informal) glossary of descriptions that relate the symbols and names to things in the problem domain. Dictionary definitions are scoped according to package scope rules.

dynamic invariant *See* effect invariant.

effect A convenience postcondition introduced (and named) to factor parts of postconditions that are common across more than one action. Unlike ordinary predicates, an effect can contain the special postcondition operator @pre.

equality A generic relation on a type, in which the relation must satisfy certain mathematical properties; defined as a standard framework.

framework application An import of a framework with substitutions; usually depicted graphically using a UML "pattern" symbol, with labeled lines for the type substitutions and text annotations for finer-grained substitutions (attributes, actions, and so on).

framework, model A template package; a package that is designed to be im-ported with substitutions: It "unfolds" to provide a version of its contents that is specialized based on the substitutions made. (Note that our usage of *framework* is somewhat broader than its traditional usage as a collection of collaborating abstract classes.)

A framework can abstract the description of a generic type, a family of mutually dependent types, a collaboration, a refinement pattern, the modeling constructs themselves, and even a bundle of fundamental generic properties (associative, commutative, and so on). Frameworks are themselves built on other frameworks; at the most basic level, the structure of frameworks represents the basis for the organization of all models.

implementation Program code that conforms to an abstraction; requires no further refinement (strictly speaking, it still goes through compilation and so on).

import, extension An extension import is a relation between packages whereby all names and definitions exported by the imported package are accessible in the importer, together with any new elements and added statements about the imported elements that the importer may introduce. A package exports all introduced elements as well as all elements accessible via extension imports. Import by extension is the default rule for import.

import, usage A usage import is a relation between packages whereby all names and definitions exported by the imported package are accessible in the importer, together with any new elements and added statements about the imported elements that the importer may introduce. However, elements accessible only via usage imports are not exported by a package.

invariant effect A transition rule that applies to the postcondition of every action in the range of the invariant; by writing a conditional (`eff1 ==> eff2`), you can impose the rule selectively on those actions that have effect `eff1`—for example, "all operations that alter `x` must also notify `y`."

object Any identifiable individual or thing. It may be a concrete, touchable thing, such as a car; or an abstract concept, such as a meeting, relationship, number, or a computer system. Objects have individual identities, characteristic behaviors, and (perhaps mutable) states. In software, an object can be represented by a combination of stored state and executable code.

object behavior The effects of an object on the outcomes of the actions it takes part in and their effects on it.

package A named container for a unit of development work. All development artifacts—including types, classes, compiled code, refinements, diagrams, documentation, change requests, code patches, architectural rules and patterns, tests, and other packages—are in a package. A package is treated as a unit for versioning, configuration management, reuse, dependency tracking, and so on. It also provides a scope for unique names of its elements.

package, nested A package whose name is itself scoped within a containing package. The contained package implicitly imports its container.

package, virtual A named package that informally denotes (as opposed to actually containing) a set of terms and definitions that you want to refer to from other packages. A virtual package can be "virtually" imported by a "real" package.

parameterized attribute An attribute with parameters such as `priceOf(Product)`. Its value is a function from a list of parameters to an object identity. Unlike an operation,

it is used only as an ancillary part of a behavioral description and need not be implemented directly. A partial parameterized attribute has a precondition.

ports The exposed interfaces that define the "plugs" and "sockets" of components: Those places at which the component offers access to its services and from which it accesses services of others. A plug can be coupled with any socket of a compatible type using a suitable connector.

provisions A set of prerequisites associated with a framework; any elements substituted when applying this framework must meet the prerequisites in order for the framework to be applicable. Provisions are analogous to design-time preconditions.

quoted actions A postcondition can refer to another action by naming it within brackets: [[action(...)]]. This is called quoting, and it means that the effect specified for that action is a part of this postcondition. If written as [[->action(...)]], then the action must actually be invoked as part of the postcondition; if further prefixed with sent, it indicates that an asynchronous invocation must be made.

redundant specs A specification (including invariants and pre- and postconditions) that is implied by other parts of the model but is included for emphasis or clarity. Such specs are prefixed with a "/".

refinement (1) A refinement (noun) is a more-detailed description that conforms to another (its abstraction). Everything said about the abstraction holds, perhaps in a somewhat different form, in the refinement. (2) *Refinement*, or *refinement of*, is also used to mean the relationship between the abstract and detailed descriptions rather than to only the detailed description itself. (3) Refine (verb) is to create a refinement; also called *design*, *implement*, or *specialize*.

retrieval A function that determines the value of an abstract attribute from the stored implementation data (or otherwise detailed attributes); used with a conformance to show how the attributes map to the abstraction, as a prerequisite to showing how the behavior specifications are also met. Also called an *abstraction function.*

Reuse Law 1 Don't reuse implementation code unless you also intend to reuse the specification. Otherwise, you have no reason to believe that a revised version of the implementation won't break your code.

Reuse Lemmas (1) If you reuse a specification, try a component-based approach: implement against the interface and defer binding to the implementation. (2) Reuse of specifications leads to reuse of implementations. In particular, whenever you can implement standardized interfaces, whether domain-specific or for infrastructure services, you enable the reuse of all other implementations that follow those standards. (3) Successful reuse needs thorough interface specifications. (4) If you can "componentize" your problem domain descriptions themselves and reuse domain models, you greatly enhance your position to reuse interface specifications and implementations downstream.

scenario A prototypical trace of interactions, showing a set of action occurrences starting from a known initial state. Usually described as narrative steps, with accompanying interaction diagrams, and accompanying snapshots of an evolving state.

sequence expression A textual representation of temporal composition of actions; some can be translated into an equivalent state chart.

snapshot A depiction (usually as a drawing) of a set of objects and the values of some of their attributes at a particular point in time.

specification type versus design type A *specification type* is one that is introduced as a part of the type model of another type to help structure its attributes and effects in terms closer to the problem domain. The behaviors of the spec type are not themselves of external interest, and it may never be implemented directly. A *design type*, in contrast, is one that participates directly in actions; its behaviors are of primary importance, and it is not just a means to factor the specification of some other type.

state A Boolean attribute that is drawn on a state chart. The structure of the states defines invariants on those attributes (such as mutually exclusive states, inclusive states, or orthogonal states); additionally, you should write explicit invariants relating the state attributes to other attributes in the type model.

state chart A graphical description of a set of states and transitions.

state transition A partial specification of an action drawn as a directed edge on a state chart. The initial and final states are part of the pre- and postcondition in the spec, and additional pre/post specs are written textually on the transition.

state type A set of objects defined by a predicate: Unlike a true type, objects can move into and out of it during their lives. The predicate is defined within a parent true type; for example, caterpillar is a state within lepidopter.

static invariant A predicate that forms part of a type model and that should hold true on every permitted snapshot—specifically, before and after every action in the model. Some static invariants are written in text; other common ones, such as attribute types and associations as inverse attributes, have built-in notations.

static model A set of attributes, together with an invariant, constitutes the static part of a type model. The invariant says which combinations of attribute values make sense at any one time and includes constraints on the existence, ranges, types, and combinations of individual attributes.

stereotype A shorthand syntax for applying a framework; a stereotype is used by referring to its name as «name», attached to any model element. Frameworks provide an extensibility mechanism to the modeling language; stereotypes provide a syntax for using this mechanism.

subclass A class that inherits some of its implementation from its superclass(es).

subtype A type whose members form a subset of its supertype; all the specifications of the supertype are true of the subtype, which may add further specifications. (Note that we use *subclass* to mean inheritance of implementation.)

testing The activity of uncovering defects in an implementation by comparing its behavior against that of its specification under a given set of runtime stimuli (the *test cases* or *test data*). Any refinement can have corresponding tests.

traceability The ability to relate elements in a detailed description with the elements in an abstraction that motivate their presence and vice versa; the ability to relate implementation elements to requirements.

type A set of objects that conforms to a given type spec throughout their lives.

type constant A named member of the type—for example, 7 is a type constant of Integer. Type constants can be globally referred to by type_name.member_name.

type expression An expression denoting a type using set-like operators—for example, Women + Men.

type intersection A combination of two specifications, each of which must be fully observed (without restricting the other) by an object that belongs to the resulting type. The designer must guarantee each postcondition whenever its precondition is met regardless of the other's pre/post condition.

Type intersection—or subtyping—happens when a component or object must satisfy different clients. Each specification must be satisfied independently of the other. A type specification defines a set of instances: the objects that satisfy the spec. Subtyping is about forming the intersection of two sets: those objects that happen to satisfy both specifications.

type join A combination of two views of the same type; each view may impose its own restrictions on what the designer of the type must achieve (conjoining postconditions) or what a client must ensure (conjoining preconditions).

Joining happens when two views of the same type are presented in different places; they might be in the same package or imported from different ones. Joining is about building the text and drawings of a specification from various partial descriptions.

type spec A description of object behavior. It typically consists of a collection of action specs and a static model of attributes that help describe the effects of those actions. A type spec makes no statement about implementation.

unfolding Depicting the results of an import, possibly including substitutions, in the context of the importing package with the appropriate elements substituted.

use case A joint action with multiple participant objects that represent a meaningful business task, usually written in a structured narrative style. Like any joint action, a use case can be refined into a finer-grained sequence of actions. Traditionally, the refined sequence is described as a part of the use case itself; we recommend it be treated as a refinement even if the presentation be as a single template.