The Less Well Known UML

A Short User Guide

Bran Selic

Malina Software Corp., Nepean, Ontario, Canada
selic@acm.org

Abstract. The general perception and opinion of the Unified Modeling Language in the minds of many software professionals is colored by its early versions. However, the language has evolved into a qualitatively different tool: one that not only supports informal lightweight sketching in early phases of development, but also full implementation capability, if desired. Unfortunately, these powerful new capabilities and features of the language remain little known and are thus underutilized. In this article, we first review how UML has changed over time and what new value it can provide to practitioners. Next, we focus on and explain one particularly important new modeling capability that is often overlooked or misrepresented and explain briefly what is behind it and how it can be used to advantage.

Keywords: Unified Modeling Language, model-driven development, computer language semantics, architectural description languages.

1 Introduction

Available evidence suggests that the Unified Modeling Language (UML) is the most widely used modeling language in industrial practice [4]. Based on that and the fact that it is taught in most software engineering and information technology curricula around the world, it would be reasonable to expect that it is also a very well understood language, with all its capabilities known and exploited as appropriate. However, it seems that the reality is different. There is much misunderstanding and misinformation about the true nature of UML so that some of its more powerful features are unknown or underexploited.

The reasons behind this are due in large part to the way the language evolved. When it was first launched and then standardized in the mid 1990's, UML was (perhaps rightly) perceived as a mostly commercial initiative rather than a technical contribution. It was promoted as an attempt to consolidate the confusing explosion of diverse object-oriented analysis and design methods and notations that preceded it. While many of these notations were supported by computer-based commercial tools, very few were supported by multiple vendors. This created a problem for practitioners, since they not only had to make the difficult decision of choosing a suitable method and tool, but also had to contend with the potentially risky circumstance of being inextricably bound to a single tool vendor and a single narrowly-specialized method.

M. Bernardo, V. Cortellessa, and A. Pierantonio (Eds.): SFM 2012, LNCS 7320, pp. 1–20, 2012.
© Springer-Verlag Berlin Heidelberg 2012

Other than its object-oriented base, the original conception of UML was very much in the tradition of earlier software engineering notations: a graphical notation with informal semantics intended primarily to assist in early conceptual design of software. In other words, it originated as a *descriptive* tool, in contrast to programming languages, which are formal *prescriptive* tools. Despite subsequent efforts at tightening up the language definition in the original standardized version through the use of meta-modeling and the formal Object Constraint Language (OCL) [17], its primarily descriptive and informal nature remained. Adoption by the Object Management Group (OMG) in 1996 drew a lot of public attention to UML, resulting in the publication of a number of popular UML books intended to provide user friendly guides for practitioners. It is fair to say that these early books and contemporary technical debates reported in various professional publications were formative, leaving lasting impressions and opinions of the language that have mostly persisted in spite of over 15 years of dramatic changes.

The most significant of these changes occurred with the release of the first major revision of the language, UML 2, which was adopted by the OMG in 2005. This was followed by several important standardized increments to the language definition, all of which were designed to provide a more precise specification of both its syntax and semantics. The net result is that UML can now even be used as a programming language, if desired.

Unfortunately, neither the authors of the original best-selling textbooks[1] nor the broader public have taken sufficient note of these important qualitative differences. Consequently, the understanding of current UML is inadequate and many of its important new capabilities remain little known and little used. It is the intent of this article to shed light on some of the lesser known but very useful features and modeling capabilities of UML.

2 The Progressive Formalization of UML

It is probably true that, despite being the most widely used modeling language, UML must also qualify among the most heavily criticized computer languages (e.g., [3] and [6]). As noted above, in its earlier incarnations, UML was intended primarily as an informal tool, to be used for descriptive purposes, providing support for analysis and design and for documentation. Consequently, the semantics of the original language were very weak and highly ambiguous, prompting one recognized language design expert to comment, alluding to UML, that "…bubbles and arrows, as opposed to programs, never crash…"[7]. The original conception of the language by its two primary authors, Grady Booch and Jim Rumbaugh, (joined subsequently by Ivar Jacobson), was as a kind of power-booster facility already widely represented in the previous generation of software design methodologies such as structured analysis and structured design (SASD). These early methodologies typically involved informal

[1] More up-to-date UML user textbooks have been published, such as [8], but, unfortunately, the original outdated books still remain as the most frequently used and cited references.

graphical (modeling) languages, and were designed to raise the level of abstraction above that of traditional programming language technology. Their imprecise nature meant that interpretation of the meaning of models depended on users' intuitions, leaving space for much confusion and misinterpretation.

As a result, programmers quickly lost confidence in these informal high-level descriptions of the software, since they often bore little resemblance to what was actually implemented in the programs. The net result is that modeling has a very negative reputation among many experienced practitioners, who perceive it as mostly a waste of time and effort. This view that code is the only trustworthy design artifact, the only one worth bothering with, is still quite prevalent and is echoed by many adherents of present-day agile methods, such as "eXtreme Programming" [1][2].

This line of thinking, of course, denies the value of abstraction as a means of coping with complexity, which is really the essence of what modeling provides. The semantic gap between application domain concepts and programming technology used to implement those concepts is still quite significant, and, with the exception of trivial programs, it presents a major hurdle to the design of reliable software. In other words, we *need* abstract representations of our solutions, since the complexity and technological bias of programs can easily overwhelm us. This holds even for modern object-oriented languages, which do allow the definition of application-domain concepts (i.e., via the class construct) but which, unfortunately, still contain an excess of implementation-level detail that gets in the way of comprehension and reasoning.

It is worth noting that there are actually very sound reasons for high degrees of informality in modeling: until an idea is properly understood and validated, it is inappropriate to burden designers with bureaucratic niggling about low-level syntactical details such as missing punctuation marks. The process of design almost invariably starts with informal and vague notions that are gradually firmed up and made more precise over time. If a seemingly promising idea turns out to be inappropriate once it has been elaborated and understood, the effort expended on specifying such detail will have been wasted. (Alternatively, it may lead to an even worse predicament whereby a bad design is retained because so much effort was vested in constructing a working prototype that there may be an understandable reluctance to discard it.)

But, as design firms up it is necessary to eventually transition from informal sketchy models to fully formal computer implementations. Ideally, to ensure preservation of design intent from inception to implementation, a good computer language should enable a gradual progression through this continuum, as free of error-inducing discontinuities as possible. This notion of enabling a smooth transition from descriptive to prescriptive modeling by using a single language throughout the process was one of the primary motivations for introducing UML 2 and is also the principal characteristic that distinguishes it from its predecessor.

The progression towards a more precise and more formal language definition occurred in four major steps:

[2] In the author's view this is a very narrow and somewhat distorted interpretation of agility and not one necessarily intended by the designers of such methods.

1. The first step was the definition of a supplement to the original UML 1 specification: the UML Action Semantics specification. The Action Semantics added the capability to specify fine-grained behavior, such as the sending and receiving of messages, the reading and writing of classifier features, or the creation and destruction of objects. As an integral part of the definition of these actions, a much more precise, albeit still informal, definition of the dynamic semantics of the core of UML was also defined.
2. The definition and adoption of UML 2 [10], which fully incorporated the Action Semantics, and which, in addition to a more precise and more modular language specification, provided a small number of new modeling capabilities[3]. Perhaps most significant was the addition of advanced structural modeling features taken from several widely used architectural description languages. These little known and often misunderstood capabilities are explained in more detail in section 7.2 below.
3. The "Semantics of a Foundational Subset for Executable UML Models" specification [11], which included a fully formal definition of a key subset of the UML 2 actions, using a variant of first-order predicate logic. In addition, it included an operational semantics specification of a UML virtual machine for executing those actions. A more detailed discussion of this specification is provided in section 5.
4. The "Concrete Syntax for a UML Action Language" specification [12], recently adopted by the OMG, which provides a precise textual surface syntax for UML actions,

With the last increment above the progression to a fully-fledged implementation quality language was completed. Note, however, that—in contrast to traditional programming languages—the degree of formality to be used with UML is at the discretion of the modeler. This means that it is possible to use UML both informally for rapid and lightweight "sketching" of early design ideas as well as for implementation—a full-cycle language.

In conclusion, UML is very far from being a "notation without semantics", as it is often characterized by those who are less familiar with it. In fact, when it comes to a mathematically formal specification, UML is ahead of most popular programming languages—at least for its executable subset.

3 UML "versus" Domain-Specific Modeling Languages

Another common criticism directed at UML is that, being a "general-purpose" language, it is (a) too big and unwieldy and (b) too blunt an instrument to adequately cope with the kinds of domain-specific subtleties encountered in highly-specialized software applications. In such discussions UML is often pitted against so-called "domain-specific modeling languages" (DSMLs) [3] [16]. These are typically compact high-level languages designed specifically to address a relatively narrow application domain.

[3] To be fair, the current version of UML (UML 2.4 at the time of this writing) still suffers from numerous technical flaws. However, the latest version of the language currently under development, UML 2.5, is designed with the sole objective of eliminating as much as possible remaining ambiguities and imprecision in the language specification.

As is argued below, this particular controversy, like many similar ones in the history of computing, is more a conflict of commercial marketing messages than of technical visions. That is, it should be fairly obvious that a custom domain-specific language will, usually, produce more concise and more direct specifications and, therefore, more effective solutions for problems in its domain than a more general language. Getting closer to the application domain and its concepts, is precisely what is meant by "raising the level of abstraction" when modeling languages are discussed. The real technical issue is to find the most effective method of realizing a DSML.

There are three different ways in which this can be done:

1. A completely new language can be designed from scratch
2. An existing (base) language can be *extended* with domain-specific concepts
3. The concepts of a general existing (base) language can be *refined* (specialized) to represent domain-specific concepts

The latter two methods may seem similar, but they are fundamentally different. Namely, *extending* a language involves adding completely new concepts to the language—concepts that are unlike any of the concepts in the base language—whereas *refining* a language means *narrowing* the definition of existing base language concepts, so that they match application semantics. In principle, the refinement approach provides some important advantages over the other two approaches, but, these are often overlooked in heated theological debates. This is because they are not technical in nature, although they may actually be more important in practical industrial settings.

The primary advantage of the refinement approach to DSMLs is the ability to take advantage of several factors:

1. Reuse of the expertise that went into designing the base language (and, we should point out that, given the absence of a sound and proven theory, modeling language design expertise is still quite scarce),
2. Reuse of the tooling for the base language as well as any training materials.
3. If the base language is standardized and widely taught—as is the case with UML—it should be much easier to find experienced practitioners who are familiar with it and who will more readily absorb the specialized DSML based on it. This also reduces the training cost, which can be a significant for new languages.

Of the above, perhaps the most important is the ability to reuse existing base language tools. Although modern DSML design tools, such as those offered by *itemis AG,*[4] provide the very useful ability to automatically or semi-automatically generate some language-specific tooling directly from the language definition (e.g., model editing tools, code generators), this is usually far from sufficient for more complex applications. Industrial-scale software development needs a very wide variety of tools such as debuggers, model validators of different kinds, simulators, test generators, document generators, version management tools, and the like. The effort required to develop industrial-strength tools of this type that are scalable, usable, and robust is not

[4] `http://www.itemis.com/`

trivial and should never be underestimated (this is why there are commercial development tool vendors). Moreover, tool development is rarely a one-time cost, because tools will typically need to be maintained and upgraded over time. In balance, the impact of these costs may in some cases exceed any technical benefits of having a custom language.

From this perspective, refinement-based approaches have an advantage over the other two, since they offer a lot of opportunity for reuse, particularly for widely-used languages such as UML, for which there already exists a rich choice of tools.

This is not to say that custom DSMLs have no role to play, but it seems that their "sweet spot" lies with smaller highly-specialized stand-alone applications ("small languages for small problems"). The case of complex multi-faceted systems is different, however, even when they can be decomposed into multiple specialized sub-domains. This is because these sub-domains often overlap, so that some parts of the system may be represented in multiple models that are written in different DSMLs. These different views then have to be reconciled, which can be difficult if the individual languages are designed independently of each other (this is sometimes referred to as the "fragmentation problem" of DSMLs). In such situations, the problem is easier to manage if the various overlapping languages share a common semantics foundation. Once again, refinement-based approaches hold the advantage here, since the various DSMLs can all be evolved from the same base language.

But, refinement-based approaches do have one fundamental disadvantage: the domain-specific concepts of a DSML must have a corresponding base concept in the base language. If no suitable base concept can be found, then either an extension-based or a new language alternative must be used. (The former is usually preferred since it is more likely to have more potential for reuse in general than a completely new language.) This is why a concept-rich general-purpose language may be the best base for DSMLs, paradoxical as this may sound. Moreover, as explained earlier, such a language can also help us deal with the fragmentation problem, since all the derived DSMLs would be sharing the same semantic core.

This brings us back to UML and its ability to serve as a source language for defining refinement-based DSMLs via its *profile* mechanism. A UML profile comprises a set of domain-specific refinements of standard UML language constructs and corresponding constraints. Because such refinements are semantically aligned with their base concepts, existing UML tools may be directly applicable for the DSML defined by the profile[5]. This capability has been used extensively to produce numerous UML-based DSMLs, many of which have been standardized by the OMG as well as other standardization bodies [9].

Unfortunately, the profile mechanism of UML is far from perfect[6]. This is due to the fact that, when the approach was first proposed, there was little practical experience with it. For example, it is not easy to determine with precision whether or not a

[5] Needless to say, it is best if the UML tool itself be sensitized to the added domain-specific semantics, which is why many current UML tools are designed to be highly customizable.

[6] Improvements to the profile mechanism are being considered within the OMG at the present time.

particular specialization of a UML concept is semantically aligned with its base concept so that a standard UML tool will treat it correctly. Nevertheless, despite its technical shortcomings, because it is a refinement-based approach to DSML design, it has the advantages that come with that approach. It has proven adequate in practice, although, admittedly, not universally so.

In addition to serving as a mechanism for defining DSMLs, the profile mechanism has one additional and important capability. This is the ability to use profiles as an *annotation mechanism* for re-interpreting UML models or profile-based DSML models. Namely, using stereotypes of a profile, it is possible to attach customized annotations to elements of a UML or DSML model. These annotations are like overlays that do not affect the underlying model in any way and can, therefore, be dynamically applied or removed as required. Such annotations are typically used to provide custom supplementary information not supported in standard UML and which can be exploited by various model analyzers or model transformers. For example, the standardized MARTE profile [13], provides facilities for adding information that is useful for certain types of real-time systems, such as timing information (deadlines, durations, processing times, delays, etc.), which can be used by specialized tools to analyze the timing characteristics of a design.

In summary, UML and the idea of DSMLs are not mutually exclusive as is often suggested. In fact, UML may provide the best solution to supporting a DSML in many practical situations, particularly for more complex systems.

4 The Structure of UML 2

Since it was designed to cover a broad spectrum of application types, UML is, undoubtedly, a large computer language. This makes it difficult to master in its full extent. (Although, it should be noted that the intellectual effort required to master UML pales in comparison to the effort required to master modern programming languages such as Java, which, although relatively compact, is only truly useful if it is combined with numerous standard class libraries and other utilities.) But, is it really “too big” as its critics often like to repeat?

It is probably fair to say that it is “too big”, *if the language is approached without a particular purpose in mind*, meaning that one would need to digest all of it in a single sweep. Fortunately, in UML 2, the structure of the language has been modularized so that it is rarely a need master the full language. In fact, UML consists of a set of distinct *sub-languages*, each with its own concrete syntax (notation), but which, fortunately, share a common foundation (Fig. 1[7]). With a few exceptions, these languages are independent of each other and can be used independently or in combination. Thus, one only needs to learn the sub-languages of direct interest and the sub-languages that these depend on, while ignoring the rest. For instance, users interested in capturing event-driven behavior via state machines, need only to know the State Machine language, a subset of the Activities language, and the Foundations they both rest on.

[7] Note that this diagram depicts a language user’s view of the various sub-languages and their relationships. The actual internal structure of the UML metamodel is somewhat different.

UML 2 consists of the following sub-languages:

- The *Foundations* module contains a *Structural Kernel,* which covers basic structural concepts such as classes, associations, instances, values, etc., and a *Behavioral Kernel*, which in turn depends on the Structural Kernel, and which covers essential behavioral concepts, such as events, messages, and the like. The Structural Kernel alone is sufficient for certain basic forms of software modeling provided via class diagrams. The Foundations provide the common core that is shared by all other sub-languages
- The *Structured Classes* language was added in UML 2 and supports the modeling complex classes representing complex architectural entities. It is a distillation of a number of architectural description languages. It is described in more detail in section 7.2.
- The *Deployment* language is used for capturing the allocation of software modules (e.g., binaries) to underlying hardware or software platforms[8].
- The *Collaborations* language is used to describe complex structural patterns of collaborating objects. Despite its name, this language is used to specify structure and not behavior. However, it is often used to define the structural setting for interactions. It is covered in section 7.1.
- The *Interactions* language serves to model interactions between multiple collaborating entities and the actions that occur as a result. In UML 2, interactions can be specified using three different graphical syntactical forms as well as one tabular one.
- *Actions* are used for specifying fine-grained behavioral elements comparable to traditional programming language instructions. UML actions depend on the Activities language, which provides facilities for combining actions into more complex behavioral fragments as well as to control their order of execution. Note that, except for a generic syntactical form (which does not differentiate between the various types of actions), there is no concrete syntax for representing actions. Instead, behavior at this level can be specified using the ALF language [12], which has a textual concrete syntax reminiscent of conventional programming languages such as Java. (The raw UML Actions can be thought of as a kind of UML assembler, whereas the ALF language is of a higher order.)
- The *Activities* language is used to model complex control or data flow based behaviors, or even combinations of the two. It is inspired by a colored Petri Net formalism and is well-suited to the modeling parallel processes, such as complex business processes.
- The *State Machine* language is used for specifying discrete event-driven behaviors, where the responses to input events are a function of history. In UML 2 a special variant called *protocol state machines* was added to support the specification of interface protocols.

---

[8] This language provides a relatively simple model of deployment that is suitable for some applications. A much more sophisticated deployment modeling capability is provided by the MARTE profile [14].

- The *Use Case* language captures use cases and their relationships, as well as the actors that participate in those use cases.
- The *Information Flow* language serves to capture the type and direction of flow of information between elements of a system at a very abstract level.
- The *Profiles* language, as already explained, is not used for modeling systems but for defining DSMLs based on UML.

In addition to the above set that are part of UML proper, two other languages can be used when modeling with UML:

- The *Object Constraint Language* (OCL), which is used to write formal logic constraints either in user models or in profiles.
- The *Action Language for Foundational UML (ALF)*, a high-level language with a concrete textual syntax used for specifying detailed behaviors in the context of UML Activities. The semantics of ALF constructs are expressed in terms of UML actions and activities and are fully compatible with the dynamic semantics of UML itself. As noted earlier, the combination of UML and ALF (including its libraries) is sufficient to make UML an implementation language.

Naturally, all of these languages can be combined as needed. This is facilitated by the fact that they are all based on the same Foundations, including OCL and ALF.

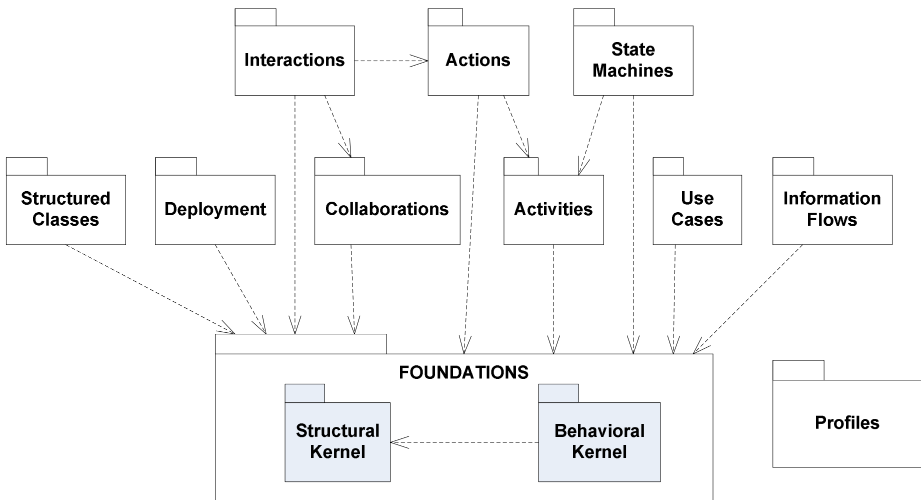

Fig. 1. The UML 2 sub-languages and their relationships

This new modularized architecture of the UML 2 language provides a lot of flexibility, enabling modelers to select an appropriate subset of the language suited to their needs. Combined with the profile mechanism, such a subset can be specialized even further to suit specific application domains.

5 The Specification of UML Semantics

As noted above, the definition of UML semantics, originally expressed almost exclusively in natural language, has been significantly tightened with the adoption of the "Semantics of a Foundational Subset for Executable UML Models" specification [11]. Although it does not cover the full UML language, this specification provides a foundation for the semantic core of UML, upon which the rest of the language rests. The subset of UML covered is referred to as "*foundational UML*" or *fUML* for short. It covers the following four major groupings of UML concepts from Fig. 1:

- The *Structural Kernel of UML* (part of the Foundations package).
- The *Behavioral Kernel of UML* (also part of the Foundations package).
- A major subset of the UML *Activities* sub-language
- A major subset of the UML *Actions*

This subset was carefully chosen because it was deemed sufficient for describing the semantics of the remainder of UML. Note, however, that it is not a minimal set. Instead, a tradeoff was made between minimality and expressiveness, providing for a relatively compact and understandable specification.

The approach taken for defining the semantics of fUML is an *operational* one; that is, the dynamic semantics of the various fUML concepts are specified in terms of the operation of a fUML virtual machine (in effect, an interpreter capable of executing fUML models). The language used to specify this virtual machine is a subset of fUML. This subset of the subset is referred to as "*base UML"*, or *bUML*. The relationship between these different flavors of UML is depicted by the Venn diagram in Fig. 2. Note that bUML is only slightly smaller than fUML.

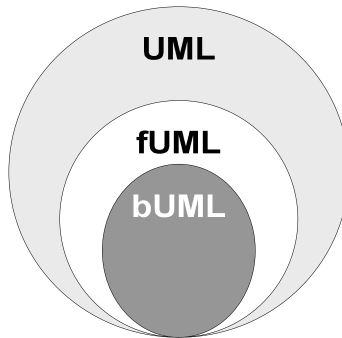

Fig. 2. Relationship between full UML, fUML, and bUML

Of course, to avoid a fully circular definition of the semantics, it is necessary to describe the semantics of bUML using some other formalism, preferably one that is well recognized and well understood. Therefore, a separate *formal* definition of the semantics of bUML is provided in the fUML specification. This definition uses a *declarative* approach, using a special formalism for modeling concurrent systems called

Process Specification Language (PSL), which has been adopted as a standard by the International Standards Organization (ISO) [5]. PSL is based on first-order mathematical logic.

The details of the fUML virtual machine and how PSL is used to capture bUML semantics are outside the scope of this article. However, the following section provides an informal overview of the semantics of UML, which incorporate the semantics of fUML.

6 The Dynamic (Run-Time) Semantics of UML

The run-time semantics of UML, as defined in the general standard and as further refined within fUML, are relatively straightforward. The basic behavioral paradigm is a discrete event driven model.

The prime movers of all behavior in the system are active objects, which interact with each other by sending messages through links. The sending and receiving of messages, result in event occurrences. The reception of an event by an active object may cause the execution of appropriate behaviors associated with the receiving object.

An *active object* in UML is an object that, once created, will commence executing its *classifier behavior* (a specially designated behavior associated with the class of the object). This behavior will run until it either completes or until the object is terminated. In a given system, the classifier behaviors of multiple active objects can be running concurrently. It is sometimes said that an active object "runs on its own thread", but this formulation can be misleading. One problem is that the concept of a "thread" is technology specific and has many different realizations and interpretations in different systems. (It is, of course, preferable if the semantics of UML are defined precisely and independently of any particular technology.) It is also inappropriate, since there can be many such "threads" associated with the behavior of an active object. For example, the composite states of UML state machines may include multiple concurrent regions, or, a UML activity may fork its control or data flows into multiple concurrent flows.

When an active object needs to interact with another active object, it sends a message through a link to the object at the opposite end of the link. The message is a carrier of information and may represent either a synchronous invocation of an *operation* of the target object, or an asynchronous signal corresponding to a *reception*[9]. Once the message arrives at its destination, it is placed in the *event pool* associated with the receiving object. The message will remain in the event pool until it is dispatched according to a scheduling policy. To allow modeling of different kinds of systems, the scheduling and dispatching policies are semantic variation points in fUML, although a default first-come-first-serve policy is supplied.

[9] In UML a reception is a behavioural feature similar to an operation, except that it is invoked (by sending a signal) and executed asynchronously. Only active objects can have receptions.

Messages are extracted from the event pool only when the receiving object executes a "receive" action[10]. Which message is selected and dispatched at that point depends on the scheduling policy. Once the message is received, it is processed by the active object according to its classifier behavior.

As part of executing its classifier behavior, an active object may access the features of passive objects. Passive objects are created by active objects and their operations and attributes are accessed synchronously. Note that, unless care is taken, it is possible for conflicts to occur when multiple active objects access the same passive object concurrently.

7 Advanced Structure Modeling in UML 2

Empirical evidence suggests that of all the diagram types provided by UML, class diagrams are by far the most widely used in practice [2], [4]. This is generally a positive outcome, since class diagrams are an excellent example of the power of abstraction and the benefits that it can bring to the design of complex systems. Nevertheless, there is much confusion about the precise meaning of these diagrams (undoubtedly due in part to the imprecise and vague descriptions provided in the standard itself), leading to frequent misuse.

One common mistake is to treat class models as instance models. A class in UML, as in most object-oriented languages, is a specification of what is common to all instances of that class (e.g., the number and types of its features), that is, a set of rules that define what constitutes a valid instance of the class. Using mathematical terminology, we say that a class is an *intentional* specification. (Corresponding to it is an extension represented by the set of all possible instances of that class.) Consequently, a class says nothing about characteristics that are unique to individual instances—those are abstracted out in class models. This level of modeling is sufficient in some types of applications, particularly in databases. In fact, class modeling evolved from standard entity-relationship modeling that originated in database theory.

But, there are many applications where it is necessary to capture instance-specific information. For example, consider the two distinct systems depicted in the two instance diagrams in Fig. 3(a) and Fig. 3(b). That these are two different systems should be clear, since they contain a different number of elements. However, note that they share the same class diagram (Fig. 3(c))[11].

Clearly, class diagrams are not suitable for this purpose and we need something more. In UML, there are two types of structure sub-languages specifically designed to support instance-based modeling: collaborations and structured classes. In contrast to simple instance (object) diagrams, which merely represent snapshots of systems at some point in time, these sub-languages provide a means for specifying rules for what constitutes *valid configurations of instances*. In other words, they do for instances what class modeling does for classes.

---

[10] In case of state machines, this action is implicit and occurs upon the full completion of a transition when a steady state is reached – hence, the term "run-to-completion".

[11] In fact, the class diagram represents a potentially infinite number of different systems.

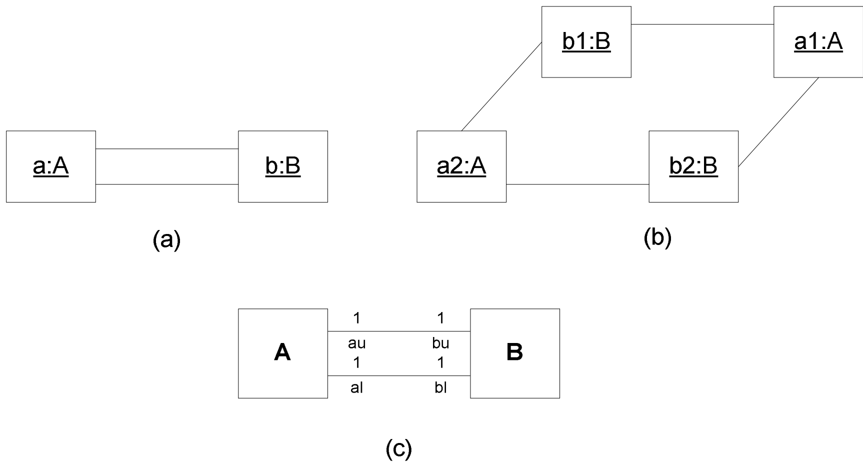

Fig. 3. Class versus instance models

7.1 Collaborations and Collaboration Uses

A common example of an instance-based pattern (configuration of instances) is the well-known Model-View-Controller architectural pattern shown in Fig. 4 [18]. This is a general pattern that can take on many concrete forms. In it, we differentiate the individual participants based their responsibilities, or *roles,* within the pattern rather than by their identities [14]. Thus, we need to represent instances in a structural context while abstracting away their identities. For example, the fact that in some realizations of Model-View-Controller a single object might be filling both the Model and the Controller roles is irrelevant to the specification of the pattern.

When working with roles, we may not be interested in the type of the object playing a particular role, leaving it undefined. However, in UML, we also have the option to specify the type of a role, a constraint which signifies that only instances of the designated type or its compatible subtypes can fill that role.

A role only makes sense in a greater structural context in which interacts with other roles. In UML, the structural context containing roles is called a *collaboration*[12]. UML.

(*Practical tip*: Note the use of the rectangle notation for classifiers in Fig. 4, to represent the collaboration rather than the more widely known dashed oval notation found in most UML textbooks. The rectangle notation is the default UML notation for any kind of classifier, including collaborations. This is usually a much more convenient and efficient form than the enclosing oval, since it provides for more effective use of scarce screen surface area.)

[12] This is a rather unfortunate choice of name, since the term “collaboration” has dynamic connotations, although the UML concept is structural in essence. (In fact, in UML 1, collaborations were misclassified as a kind of behavioural modelling.).

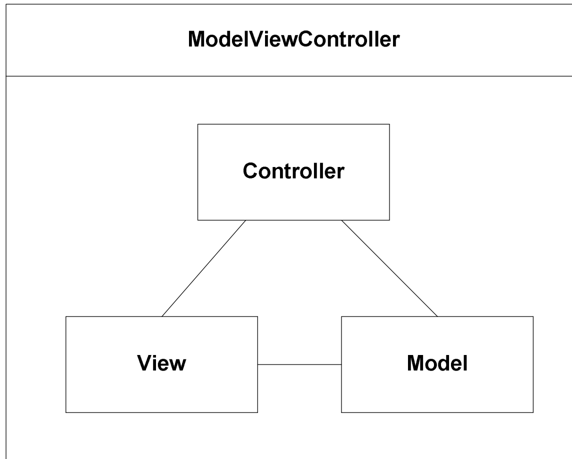

Fig. 4. Model-View-Controller pattern expressed as a collaboration

The roles of a collaboration may be linked to each other via *connectors*, representing communication paths by which the roles interact with each other. Connectors explicitly identify which roles are mutually coupled and which ones are not. Since deciding on the coupling between components of a system is a critical architectural design decision, the presence of connectors provides a concrete manifestation of design intent. Since inter-object communications in UML is accomplished via links, a connector denotes a link instance, in the same way that a role denotes an object instance. Note that it is not necessary to define associations for such implicit links, although it is possible, particularly if all the connected roles are typed.

Standard UML does not define fully the semantics of connectors. Specifically, the communication properties of such links (i.e., whether they are order preserving, non-duplicating, and non-lossy) are left as semantic variation points. If a stronger definition is required, it can be provided through a profile[13].

Collaborations are frequently combined with interactions, providing the structural setting over which message exchange sequences are overlaid. (In fact, it is this combination of two distinct UML sub-languages that gave rise to the term "collaboration".[14]) In those cases, the lifelines of an interaction are associated with the roles of the underlying collaboration. Clearly, messages between two lifelines should only be permitted if the corresponding roles are connected (although standard UML does not enforce this constraint).

(*Practical tip*: In support of this combination of collaborations and structures, UML provides the *communications diagram* notation, which shows what appears to be a collaboration diagram with roles corresponding to the lifelines of the interaction and messages shown as numbered arrows running parallel to the connectors they traverse. While this notational form is widely advertised in many UML textbooks,

---

[13] fUML is more constrained and does assume perfect communication properties for links.

[14] It is only in UML 2 that these two sub-languages were separated.

practical experience has shown that it is not particularly useful due to graphical limitations, and is probably best avoided. In practice, message names are often longer than the space provided between the roles of the graph so that they do not fit in the diagram. Moreover, for anything but trivial message sequences, it is very difficult to follow the flows of messages by keeping track of message sequence numbers, especially when concurrent sequences are involved.)

Collaborations that capture structural patterns, such as Model-View-Controller, can be treated as a kind of macro definition that can be reused whenever a pattern needs to be applied in a given model. In UML this can be achieved through a mechanism called *collaboration use*. Consider, for example, the case where we would like to capture a special variant of the Model-View-Controller pattern such that the Model and the Controller roles are filled by the same object. We can represent this by a new collaboration, shown by the collaboration diagram in Fig. 5.

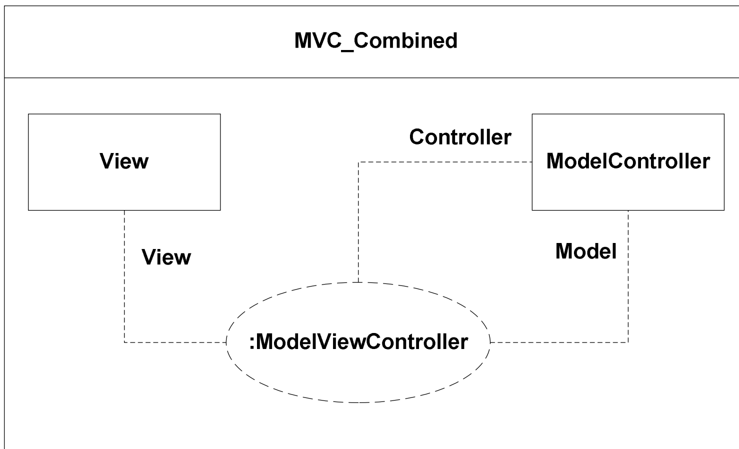

Fig. 5. Collaboration use example

The dashed oval in Fig. 5 is the notation for a collaboration use, and it represents a reference to (i.e., application of) the original ModelViewController pattern. The dashed lines emanating from it represent the roles of the corresponding collaboration (ModelViewController). Of course, in this particular case, we could have done this by simply having a simple collaboration with just two roles, View and ModelController, avoiding the collaboration use. However, that would have obscured the design intent, which was to take advantage of the well-known design pattern. The collaboration use makes that explicit.

Although collaborations have been a part of UML from the very first release of the standard[15], evidence indicates that, in contrast to class diagrams, they are little used by practitioners [4]. This is surprising at first, since one would normally expect instance

[15] Although the technical definition of collaborations has changed somewhat between UML 1 and UML 2, the essence has remained unchanged.

based modeling to be the more intuitive form of representing structure to most people. Class modeling requires an extra inductive reasoning step: abstracting from the particular to the general. The most likely explanation is that most software developers trained in object-oriented programming are familiar with the class concept, whereas the notion of a role is less well known and is not supported explicitly in common object-oriented programming languages.

This is unfortunate, since there are many cases where collaborations are the most natural modeling technique. Instead, it often happens that attempts are made to capture instance-specific structures unsuccessfully using class diagrams. For example, Fig. 6 illustrates one of the most common mistakes. In attempting to capture the structure shown in Fig. 6(a) even experienced modelers might define a class diagram like the one in Fig. 6(b). However, this is incorrect since that particular class model supports a variety of different instance patterns, including the one shown in Fig. 6(c).

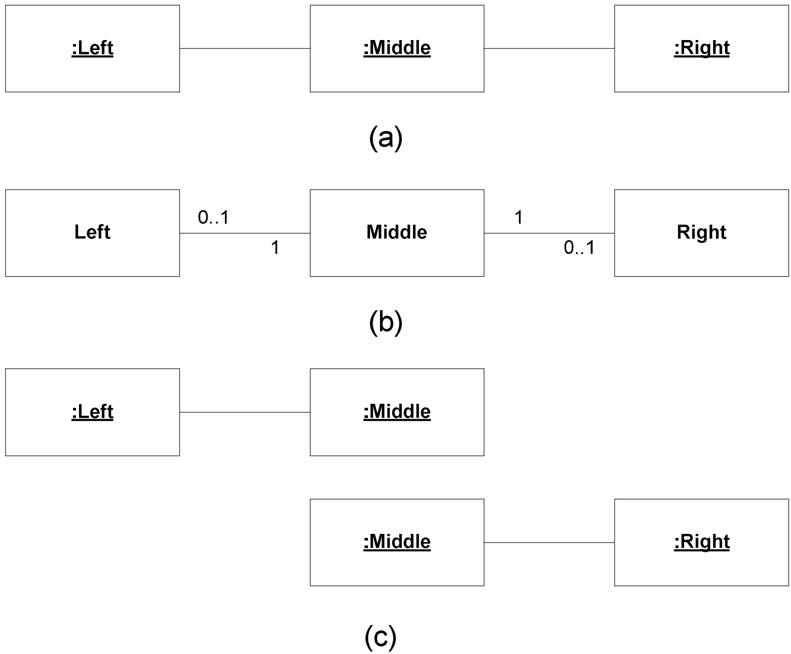

Fig. 6. Inappropriate use of class modeling

7.2 Structured Classes and Components

UML 2 added another form of instance-based modeling via the *structured class* concept. Structured classes were inspired by various architectural description languages used to represent the architectural structure of software systems.

Structured classes are distinguished from "simple" classes by virtue of the following two features:

- The possible presence of one or more communication *ports* on its interface.
- The possible encapsulation of an *internal structure* consisting of a network of collaborating objects.

Ports. Ports are a common feature of most architectural description languages but they are rarely encountered in programming languages. Ports are analogous to the pins of a hardware chip: instead of a single interface, it is possible to have a set of distinct interface points each dedicated to a specific purpose. Ports in UML add two important modeling capabilities:

1. Like any interface, a port provides an explicit and focused *point of interaction* between the object that owns it (i.e., an instance of a structured class) and its environment, thereby isolating each from the other. An important characteristic of ports is that they can be *bi-directional*. This means that a port has (a) an *outward face*, which it presents to its collaborators and which defines the services that the object provides to its collaborators, and (b) an *inward face*, which can be accessed by its internal components and which reflects the services that are expected of the collaborators on the outside. Thus, a port represents a full two-way contract, with the obligations and expectations of each party explicitly spelled out.
2. Since an object can have multiple ports, they allow an object to *distinguish between multiple, possibly concurrent collaborators* by virtue of the port (interface) through which an interaction occurs.

Reflecting the bi-directional nature of ports, in UML a port can be associated with UML Interfaces[16] in two different ways. For its outward face, a port may *provide* zero or more Interfaces, which specify its offered services. For its inward face, a port may *require* zero or more Interfaces, which define what is expected of the party at the opposite end of the port.

(*Practical* tip: It is generally better to associate no more than one provided Interface with a port, to avoid possible conflicts when the definitions of two or more Interfaces overlap (e.g., they provide the same service). If different Interfaces are desired, it is always possible to add a separate port for each Interface.)

Note that it is possible for an object to have multiple ports that provide the same Interface. This allows an object to distinguish between multiple collaborators even though they require the same type of service. This capability is particularly useful when Interfaces have associated protocols, that is, when the interactions between an Interface user and an Interface provider must conform to a particular order. For example, a database class may impose a two-phase commit protocol to be used when data is being written. This usually means that at any given time, an interface (i.e., port) instance may have a state, corresponding to the phase of the protocol that it is in. With multiple ports supporting Interfaces of the same type with each dedicated to a separate client, it is possible for individual ports to be in different states, allowing thus full decoupling of concurrent clients all using the same Interface.

---

[16] To distinguish between the generic notion of “interface” and the specific UML concept with the same name, the latter is capitalized in the text.

The dynamic semantics of UML ports are quite straightforward: whatever messages (operation calls, signals) come from the outside they are simply relayed inwards, and vice versa. In other words, they are simply relay devices and nothing more.

Ports must be connected to something on either face; otherwise, whatever comes in from a connected side will simply be lost. There are two possible ways in which a port can be connected: (a) to the end of a connector or (b) to the classifier behavior of the object that owns the port. In the latter case, the port is called a *behavior port*. It is only when a message arrives at a behavior port that anything actually happens (see section 6 above).

Of course, only compatible ports can be connected through a connector. The specific rules for compatibility are not defined in standard UML (i.e., it is a semantic variation point). But, generally speaking, two ports are compatible if the services required by one are provided by the other and vice versa. If protocols are associated with the ports, compatibility rules are more complex except in the trivial case where the associated protocols are exact precise complements of each other.

The standard UML notation for ports is shown in Fig. 7. Port Pa is a non-behavior port, whereas Pb is a behavior port (indicated by the small "roundtangle" attached to it). Note that ports do not necessarily have to appear on the boundary of the classifier icon, although that is the usual convention.

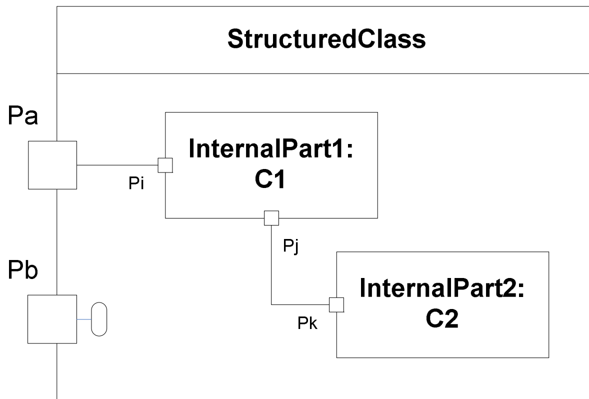

Fig. 7. A structured class with parts and ports

Internal Structure. In a sense, as shown in Fig. 7, the internal structure of a structured class is very much like a collaboration that is contained within the confines of an object. The only difference is that the *parts* comprising the internal structure do not represent roles, but the structural features of the class, such as attributes and ports. (Naturally, some of these parts may be typed by structured classes with their own ports and internal parts, as the example in Fig. 7 illustrates.) The extra modeling capability provided by internal structure is that the parts can be linked via connectors. In other words, in addition to a set of structural features (e.g., attributes), structured

classes also own an interconnection topology that shows explicitly how the various parts interact.

The full run-time semantics of structured classes are not fully specified in standard UML (nor are structured classes covered by fUML). Are the parts and connectors automatically created when an instance of the class is created? That would certainly be quite useful, since: (a) then there would be no need to specify the tedious housekeeping code for creating the internal objects and connections, and (b) it would result in more reliable implementations, since the implementation code would be automatically generated from the class definition. Moreover, this would also ensure that only possible couplings between elements of the system would be those that are explicitly specified by the modeler. In fact, a number of UML profiles, such as the UML-RT profile [15], define such semantics.

The addition of concepts such as ports, connectors, and structured classes means that UML can be used as an architectural description language extending the range of UML from the highest levels of a system down to very fine-grained detail (e.g., using ALF).

Components. There a common misconception that in order to take advantage of UML's architectural modeling constructs such as ports, connectors, and internal structures one must use the UML Component construct. As explained above, this is not the case. In fact, the Component construct in UML 2 is merely a specialization of the structured class concept, with a few additional features that are included primarily for backward compatibility with previous versions of UML. Namely, the UML 1 definition of Component attempted to cover a variety of different interpretations of that widely used term with a single concept. Consequently, a component was used to denote both a unit of software reuse (e.g., the source code and binary modules) residing in some design repository as well as an instantiable run-time entity similar to a class. While UML 2 has retained this hybridized concept (to smooth the transition of UML 1legacy models to UML 2), it is probably best to avoid it, using instead structured classes which have clear and unambiguous semantics.

8 Summary

UML has evolved a long way from its informal early versions, characterized by ambiguous and imprecise semantics. It has gradually emerged as a full-cycle computer language, equally capable of being used in a highly informal lightweight design sketching mode as well as a fully-fledged implementation language. Unfortunately, this progression from a purely descriptive tool to a prescriptive one, has received little attention so that there is much misunderstanding of its role and capabilities.

In this article, we have briefly summarized the nature of this dramatic evolution, pointing out how the tightening of the semantics of UML has been achieved and what benefits that provides.

In addition, we have focused on an important new capability provided by UML 2, but one which is little known and often misinterpreted: the mechanisms for capturing

and enforcing instance-based structural patterns. This adds an important new dimension to UML by providing the standard constructs found in architectural description languages, extending thus the scope of abstractions that can be represented.

References

1. Beck, K.: Extreme Programming Explained. Addison-Wesley, Boston (2000)
2. Dobing, B., Parsons, J.: How UML is Used. Communications of the ACM 49, 109–113 (2006)
3. Greenfield, J., Short, K., et al.: Software Factories. Wiley Publishing, Inc., Indianapolis (2004)
4. Hutchinson, J.: An Empirical Assessment of Model Driven Development in Industry. PhD Thesis, School of Computing and Communications, Lancaster University, UK (2011)
5. International Standards Organization (ISO): Industrial automation systems and integration – Process specification language (Part 1: Overview and basic principles). ISO standard 18629-1:2004 (2004), http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=3543
6. Kelly, S., Tolvanen, J.-P.: Domain-Specific Modeling. John Wiley & Sons, Hoboken (2008)
7. Meyer, B.: UML: The Positive Spin (1997), http://archive.eiffel.com/doc/manuals/technology/bmarticles/uml/page.html
8. Milicev, D.: Model-Driven Development with Executable UML. Wiley Publishing Inc., Indianapolis (2009)
9. Object Management Group (OMG): Catalog of UML Profile Specifications, http://www.omg.org/technology/documents/profile_catalog.htm
10. Object Management Group (OMG): OMG Unified Modeling Language (OMG UML) Superstructure. OMG document no. ptc/10-11-14 (2010), http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/
11. Object Management Group (OMG): Semantics of a Foundational Subset for Executable UML Models (fUML). OMG document no. formal/2011-02-01 (2011), http://www.omg.org/spec/FUML/1.0/PDF/
12. Object Management Group (OMG): Action Language for Foundational UML. OMG document no. ptc/2010-10-05 (2010), http://www.omg.org/spec/ALF/1.0/Beta2/PDF
13. Object Management Group (OMG): UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems. OMG document no. formal/2011-06-02 (2011), http://www.omg.org/spec/MARTE/1.1/PDF
14. Reenskaug, T., Wold, P., Lehne, A.: Working With Objects. Manning Publications Co., Greenwich (1996)
15. Selic, B., Rumbaugh, J.: Using UML for Modeling Complex Real-Time Systems. IBM developerWorks (1998), http://www.ibm.com/developerworks/rational/library/content/03July/1000/1155/1155_umlmodeling.pdf
16. Völter, M.: From Programming to Modeling—and Back Again. IEEE Software, 20–25 (November/December 2011)
17. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley Professional, Reading (2003)
18. Wikipedia, Model-View-Controller, http://en.wikipedia.org/wiki/Model-view-controller