Chapter 4

UML Versus Codecharts

Since the UML is the de facto industry standard modelling language, questions naturally arise about its relation to Codecharts. This subject has been treated in detail throughout our discussion in the properties of design description languages in Chapter 2 and in the guiding principles of Codecharts presented in Chapter 3. This chapter summarizes the similarities and differences between Codecharts and UML diagrams.

The UML is a rich and expressive set of notations designed to articulate a very wide range of functional and nonfunctional specifications of software as well as activities related to software development. Unlike Codecharts, the UML is not merely a *design description language* (Chapter 2), and it is not constrained to design decisions about object-oriented programs. Its charter is therefore significantly broader than that of Codecharts.

More to the point, the UML is not a formal language: It is not bound by the need for precision, nor is it restricted by the requirement for verifiability—and by implication, *automated verifiability* (§3.4). This freedom from rigour allows using the UML to articulate specifications for which any notion of design verification (let alone automated verification) is hard to conceive. For example, Use-Case Diagrams and Activity Diagrams are particularly effective in visualizing informal notions such as user requirements, whose representation often requires concepts that fall well outside the charter of any formal language. Furthermore, the UML's stereotype mechanism allows users to extend it in any way desired, offering the software engineer the flexibility required for capturing and conveying novel kinds of specifications without requiring attention to the precise meaning of any particular symbol. On the flip side, the same freedom entails the ambiguity from which UML suffers. Consider, for example, the ambiguity of whether symbols missing from the diagram imply negative information, discussed in detail §3.8. This ambiguity entails that tools that use UML for design verification and program visualization are inherently problematic.

Even if the meaning of some UML diagrams can be determined in a fairly precise manner, UML diagrams are, by and large, *undecidable* (§2.2). In effect, this means that it is virtually impossible in principle to verify conformance to UML diagrams and it is impossible in principle to build a tool that can answer the *verification question* for such diagrams. Undecidability implies that UML diagrams can be very expressive in modelling scenarios and patterns of behaviour in programs. But it rules out automated verification, which means that conformance to specifications need be checked manually, an error-prone and expensive process that rarely takes place in reality.

Finally, the UML emphasizes expressiveness, offering versatile means for modelling specifications, whereas Codecharts are guided by the principles of *elegance* (§2.4) and *minimality* (§3.7). That is, where the UML is tailored to articulate an abudance of specific types of services and modules, such as packages (namespaces), libraries, subsystems, subprograms, processes, components, connectors, ports, and so on, the vocabulary of Codecharts (p. 23) is restricted to 15 visual tokens.

Given the differences in scope and formality, the UML and Codecharts seem far apart. A detailed comparison between the languages is therefore only appropriate when the languages are narrowed down to *design description languages* (Chapter 2) for object-oriented programs. That is, only a comparison between UML Class and Package Diagrams vs. Codecharts is meaningful. Below we sketch some of the obvious differences in modelling programs, design patterns, and application frameworks.

To compare the notations' capabilities in modelling programs, contrast Codechart 1 with the Class Diagrams in Figures 2-1 and 2-2 (p. 14), all of which were reverse engineered from the same source code [the application programming interface (API) of the Java 3D class library]. Clearly, the class diagrams are not usable because they attempt to visualize a large program in terms of individual classes. The only relevant means of abstraction that UML provides are packages. But Package Diagrams do not improve the situation because they are restricted to modelling relations between the physical units that Java packages and C++ namespaces offer.

This comparison demonstrates that Class Diagrams can effectively model the implementation minutia of small programs but also that the notation does not *scale* (§2.3). It illustrates a fundamental difference between Codecharts and Class Diagrams: In Codecharts, where the emphasis is on visualizing programs at *any* level of abstraction, sets of classes, methods, and class hierarchies can be depicted regardless of their size. Consequently, Codecharts are more scalable, and visualization tools supporting Codecharts can be more effective in reverse engineering roadmaps to large programs. Furthermore, Codechart abstraction mechanisms allow software designers to use it to articulate early design decisions without premature commitment to implementation minutia, a feat that is much less achievable in the absence of generic notions such as that of a set of classes and isomorphic relations. This explains why program visualization tools are not common in the industry and why the visualization tools rarely employ UML.

Finally, let us consider the matter of tool support. The conformance of a program to a Codechart can be verified fully automatically (§3.4). This, for example, can be done with the Toolkit for Java 1.4 programs. Conversely, Class Diagrams are not formally defined, let alone automatically verifiable. For this reason, tools that claim to verify class diagrams largely end up verifying only a trivial subset of the notation.

To compare the notations' capabilities in representing design patterns, contrast the class diagram in Figure 4-1 with Codechart 7. Both describe the Composite design pattern (to which §11.1 is dedicated).

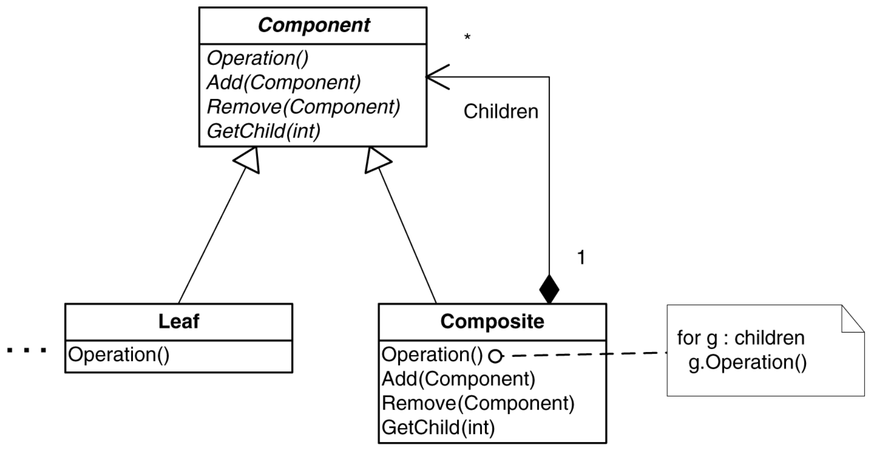
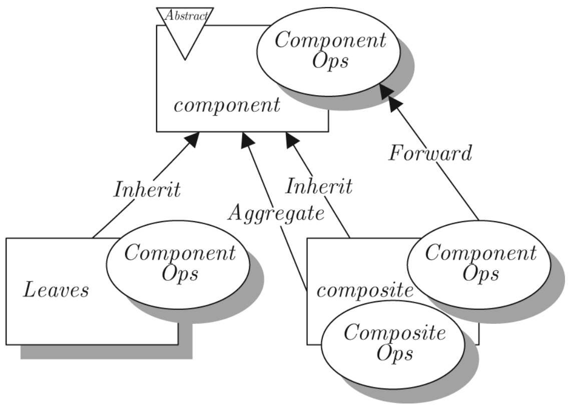


Figure 4-1. The Composite pattern modelled in the UML's Class Diagram notation (Glossary: p. 233; adapted from [Gamma et al. 1995])



Codechart 7. The Composite pattern in LePUS3

A number of obvious differences come to light from this comparison: Both diagrams attempt to depict the main *participants* in the pattern, which are classes and methods that play specific roles in the design motif that the pattern captures. But the UML depicts the Component participant as a class called Component, whereas the Codechart employs a variable for the purpose called *Component*. The difference is that a variable specifies unambiguously that any class may (in principle) play this role as long as it satisfies the formulas depicted in the Codechart. Next, observe that both diagrams seek to indicate that the Component class defines operations over the set of children (`add`, `remove`, `getChild`) and that these operations are overridden in the Composite class. The Class Diagram enumerates these operations, whereas the Codechart models them as a tribe (as a set of methods) of any size. The same applies to the number of Leaf classes (which can be one or more) as modelled in the UML using the informal ellipsis (…) notation, whereas in the Codechart it is modelled as a set of classes (*Leaves*).

Are these differences significant? The problem with the informal notation is that it leaves many questions unanswered: *Must all "leaf" classes inherit from Component?* (Yes). *Must all "leaf" classes override the methods in Component?* (No, they can inherit them). *Must all the Component operations that are overridden by the Composite class forward the call to the respective method in the children?* (Yes). *Are there only three operations over children?* (No). Answers to these questions are rigorously specified only when appropriate abstraction mechanisms are employed, such as sets of classes, sets of methods, and *isomorphic* predicate formulas.

Beyond these observations, the comparison brings to light a more fundamental difference between the notations. Symbols in Class Diagrams stand for elements of specific programs, whereas Codecharts subscribe to the principle of *genericity* (§3.6), setting apart symbols that model programs (*constants*) from symbols that model generic abstractions such as participants in design patterns (*variables*). Finally, our commitment to the principle of *automated verifiability* also dictates that the conformance of a program to Codechart 7 can be verified fully automatically.

Finally, let us compare the two notations' capabilities in documenting the use of application frameworks (see Chapter 10). Compare the Class Diagram in Figure 4-2 with Codechart 8. Both model some elements of Enterprise JavaBeans (to which §10.1 is dedicated). If a Class Diagram is used to demonstrate how programmers should write their code and how it should relate to the framework's classes, then only specific examples will do—classes `Customer` and `CustomerHome` in Figure 4-2. Such practice is likely to lead to confusion between the parts in the examples that programmers must replicate (in this case, a home interface must inherit from class `javax.ejb.EJBObject`) vs. the parts in the example that are merely demonstrative (everything else about class `Customer`). Codechart 8, on the other hand, uses variables to describe only the constraints over the user-defined classes without implying any irrelevant constraints. In large and complex application frameworks, where interactions between user-defined and prefabricated parts of the program can take a very complex form, the use of variables is indispensible (see, e.g., Codechart 85, p. 136).

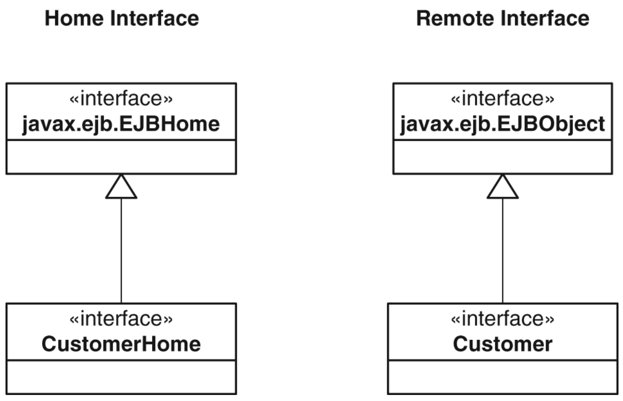
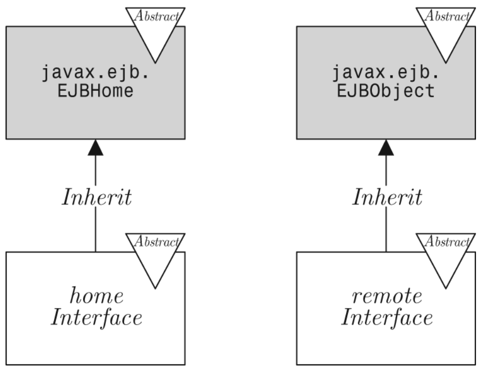


Figure 4-2. Enterprise JavaBeans™ (Table 11) elements in the UML (Glossary: p. 233, adapted from [Monson-Haefel 2001]). Classes CustomerHome and Customer are sample implementations of "home interface" and "remote interface"



Codechart 8. Enterprise JavaBeans™ (Table 11) elements in LePUS3. Variables (empty shapes) represent user-defined (yet to be implemented) entities whereas constants (filled shapes) represent prefabricated (fully implemented) entities

In conclusion, UML Class Diagrams and Codecharts are suitable for very different purposes. The appropriateness of each notation therefore depends on the circumstances in which they are used.