Chapter 3

An Overview of Codecharts

The vices and virtues of existing modelling and specification languages examined in Chapter 1 motivate our choice of the underlying principles of LePUS3, the language of Codecharts, as follows:

1. Object-orientation
2. Visualization
3. Rigour
4. Automated verifiability
5. Scalability
6. Genericity
7. Minimality
8. Information neglect

Let us examine each one of these principles and illustrate how each manifests itself in Codecharts.

3.1 OBJECT-ORIENTATION

LePUS3 is an object-oriented design description language: It is a language of statements about the design of programs encoded in object-oriented programming languages called Codecharts. More specifically, we may divide the subjects of LePUS3 specifications into three broad categories:

1. Programs and Class Libraries. Codecharts can serve as roadmaps for existing implementations and blueprints for hypothetical ones encoded in various class-based programming languages. Verification proceeds by indicating the meaning of the terms in the symbols (e.g., what CollectionsHrc stands for).
2. Design Motifs. Codecharts were tailored to articulate object-oriented design patterns, in particular the Gang of Four design patterns (Chapter 11). Verification normally proceeds by indicating which parts of the program are intended to serve as the implementation of the pattern (e.g., which classes implement the Iterators).

3. Application Frameworks. The combination of constants (testCase, setUp) and variables (userTest, FixtureClasses) captures the interactions between the prefabricated (existing) and the user-defined (yet-to-be-implemented) elements in specifications of application frameworks. Verification normally proceeds by indicating how each variable is assigned its intended implementation.
Presently, object-oriented languages are the most popular programming languages in academia and industry. But this is not the only reason for focusing on the object-oriented programming paradigm in this book. More importantly, the abstraction mechanisms underlying this programming paradigm offer the most effective means for constructing large, complex, and versatile software systems. In other words, the tools that such languages provide offer the most promising instruments for addressing the problem of software complexity.

Let us consider abstraction mechanisms underlying the object-oriented paradigm and see how exactly they relate to LePUS3:

(i) **Encapsulation** (also modularity) allows data and related operations to be divided into classes. Each class is a unit associated with a set of operations, referred to as methods. Classes and methods are therefore our primitive (or atomic) entities. In other words, Codecharts abstract programs in terms of classes, methods, and their properties as the elementary units in the representation.

(ii) **Inheritance** is a powerful abstraction mechanism which allows a class (the subclass) to be defined incrementally by importing data and operations from another (the superclass). Inheritance is also a mechanism of subtyping: Operations defined for a class are also defined for classes that inherit therefrom. Inheritance in LePUS3 has a special status, as a special symbol is reserved for sets of classes that constitute inheritance class hierarchies (§7.6).

(iii) **Dynamic binding** is a mechanism which allows the selection of the appropriate method to be called at runtime. In Codecharts, a special status is reserved to sets of methods that share the same signature (name and argument types) throughout an inheritance hierarchy, to which we refer as clans (§7.3).

---

1 The numerous merits of object-oriented programming have been discussed extensively in the literature; see, for example, Wirfs-Brock et al. [1990].

2 What follows are not definitions. We recommend Wirfs-Brock et al. [1990] and Craig [2000] as sources of detailed information about these terms.
These abstraction mechanisms motivate our answer to the ontological question (p. 15). In Table 6 we list the building blocks of object-oriented design, that is the rudimentary elements with which our descriptions are concerned and the means by which they are represented. Restricting ourselves to the list in Table 6 means that the only statements that can be represented using Codecharts are those that describe “things” made of one of the four categories of these specific building blocks. The advantages of restricting ourselves this way is the prospect of an elegant design description language (p. 17) that is committed to a minimal ontology (p. 15). The disadvantage is that Codecharts are not as expressive. In particular, our language cannot effectively model architectural styles or the design of programs encoded in other programming paradigms, such as functional, logic, and procedural programs.\(^3\)

Table 6. Building-Blocks of Object-Oriented Design

| Classes and methods | are the primitive, atomic entities of our representation. For example, the constant testCase in Codechart 4 (p. 21) stands for a class, whereas the superimposition of setUp over testCase stands for a method. |
| Relations | between entities are also represented. For example, the edge marked Inherit in Codechart 4 represents the inheritance (in Java: extends) relation between userTest and testCase. |
| Sets of classes/methods | are called higher dimensional classes/methods. For example, the variable Elements in Codechart 3 (p. 20) stands for a class of dimension 1 (a set of classes). Particular attention is given to sets of classes that constitute inheritance hierarchies. For example, the constant CollectionsHrc in Codechart 2 stands for the set of Java’s collection classes. Also, sets of dynamically bound methods (clans) are also represented. For example, the set of methods with the signature next() that are defined in classes of the IteratorsHrc is represented in Codechart 2 by superimposing the signature constant (next) over the hierarchy constant (IteratorsHrc). |
| Correlations between sets | of classes/methods associate entities between the two sets. For example, the edge marked Aggregate in Codechart 2 specifies that every concrete class in the Collections hierarchy has an aggregate of type Object. |

Finally, let us note that the term “object-oriented” has many different interpretations in the literature. These vary largely by the interpretation given in each programming language. Craig [2000] divides object-oriented programming languages into prototype-based and class-based languages. We

\(^3\)Note that the programming language is not the only factor: One may write procedural programs in an object-oriented programming language such as Java. The design of such a program will not be amenable to representation in LePUS3.
shall focus our attention on the class-based languages: the bigger and more popular category of programming languages that includes Simula 67, Smalltalk (in its various versions), C++, Object Pascal, Beta, Java, Ada 95, CLOS, and C#. Nonetheless, since the term “object-oriented” is more widely recognized than the term “class-based”, we shall employ the former even in contexts when the latter is more precise.

3.2 VISUALIZATION

The language of Codecharts, LePUS3, is a visual language for the same reasons that roadmaps and blueprints are visual: because visual cues offer effective representations of very complex objects and their relations. Like the languages of roadmaps and blueprints, the principle of visualization dictates that visual tokens (such as shapes and edges) and their visual properties (such as fill and shade) effectively capture and convey the subject of representation—in our case, modular units and their correlations. When used correctly, Codecharts are effective visualizations that allow us to “see” programs in any appropriate level of abstraction.

There are many advantages to visual languages: First, diagrams are more intuitive than most formulas. Therefore, using it does not require special mathematical training. Furthermore, since specifications are Codecharts, they can also be used for the purpose of program visualization. That is, diagrams generated by reverse engineering the source code depict the program in various levels of abstraction.

For example, the Design Navigator [Gasparis 2009], which is part of the Two-Tier Programming Toolkit [ttp.essex.ac.uk], can generate visualizations of Java programs by analyzing their source code. Program visualization reverse engineering source code can significantly promote the understanding
of the structure and organization of arbitrarily large programs. The tool can be used to unfold the organization of large collections of classes and methods in a gradual process of concretization [Gasparis et al. 2008], a step wise, user-guided process of refining each visualization—or any part thereof—using an effective tool such as the Design Navigator. Let us demonstrate the effectiveness of such a process with a specific example.4

The Closeable hierarchy in package java.io consists of all the classes and interfaces that inherit (possibly indirectly) from the interface Closeable. The Design Navigator can visualize the entire hierarchy using one symbol, a hierarchy constant, as demonstrated in Figure 3-1. The Codechart in this screenshot demonstrates the power of effective abstraction, hiding the richness of an entire hierarchy, and the notion of scalability that our design description language supports.4

![Screenshot of the TTP Toolkit visualizing an abstraction of the Closeable hierarchy in package java.io](image)

Figure 3-1. Screenshot of the TTP Toolkit visualizing an abstraction of the Closeable hierarchy in package java.io

The Codechart in Figure 3-1 is only useful as a first abstraction. The Design Navigator may be used to concretize the CloseableHrc, a process which replaces any abstraction—essentially, any visual depiction of a set—with a finer grained and more detailed representation. For example, to reveal the organization of the classes that comprise this class hierarchy, the user clicks on the CloseableHrc symbol and then clicks on a concretization operator, the list of which appears on the left panel in Figure 3-1. The outcome of this process is demonstrated in Figure 3-2.

4Steps in this sequence were omitted from this description and the layout of the symbols in the figures have been adapted to this demonstration.
3.2 Visualization

Figure 3-2. Screenshot of the TTP Toolkit visualizing a slightly more concrete representation of the Closeable hierarchy.

The complexity of the Closeable class hierarchy further unfolds as we concretize the Codechart, demonstrating that the Design Navigator is indispensible in creating such representations. For example, after a number of “concretization” steps, the Design Navigator will break down each one of the hierarchies depicted in Figure 3-2 to several sets of classes and sub hierarchies, as illustrated in Figure 3-3.

Figure 3-3. Screenshot of the Toolkit visualizing an even more concrete representation of the Closeable hierarchy.
Rather than concretizing entire Codecharts, concretization can visualize the details of any part thereof. For example, Figure 3-4 depicts a Codechart produced by concretizing only the `OutputStreamHrc` hierarchy in Figure 3-3. The resulting Codechart details some of the subclasses of `OutputStream` and their methods. At this level, individual methods, classes, and relations become clearly visible.

Figure 3-4. Screenshot of the TTP Toolkit visualizing the subclasses of `OutputStream` in the Closeable hierarchy

3.3 RIGOUR

In §2.1 we emphasized the need to reconcile practical with theoretical demands. The first requirement from a sound scientific theory is mathematical rigour. But our theoretical demands are dictated by pragmatic concerns: Rigour prevents us from “unsound and clumsy” practical work done in computing (p. 9). Formalizing Codecharts equipped us with many insights into the properties of such formal languages, in particular, decidability (§2.2). Rigour affords clarity and confidence in reasoning at levels that otherwise cannot be achieved. Only precise definitions can ensure that designers and programmers indeed share the same understanding of specifications. Such confidence is essential for effective tool support in specification and verification. Indeed, the verification question (p. 11) demands that we define precisely the relation between specifications and programs.

Which mathematical framework offers us the most appropriate tools for unpacking our language? Robin Milner [1986] motivates mathematical logic as follows:
Almost everyone who builds computing systems is convinced that all systems design—software or hardware—needs to be done within a rich conceptual frame, which is articulated by rigorous methodologies. ... The conceptual frame provided just by programming languages is too narrow; a wider frame is needed in which to understand the specifications of systems and the methodology which articulates this understanding must be based on some form of logic.

For these reasons we choose to employ mathematical logic in defining LePUS3 as a formal specification language. Below we briefly recap elements in the formalization of LePUS3. A detailed and systematic presentation of LePUS3 as a formal language is given in Part II of this book.

A Codechart is a specification in LePUS3, which consists of a set of formulas that can be unpacked as well-formed formulas in the first-order predicate logic (FOPL). As a first-order language, symbols in the vocabulary of LePUS3 (Legend 1, p. 23) can be categorized by the conventional elements of formal grammars as follows:

- **A formula** (such as `Inherit(userTest,testCase)` in Codechart 4) is a combination of a relation symbol (such as `Inherit`) with one or more terms and occasionally also with a predicate symbol.
- **Terms** (constants and variables) stand for entities, such that:
  - **0-dimensional terms** (such as `userTest` and `testCase` in Codechart 4) stand for individual entities, whereas
  - **1-dimensional terms** (such as `Elements` and `Iterators` in Codechart 3) stand for sets of entities
- **Variables** (`userTest` and `Elements`) range over classes and methods whereas **constants** (`testCase`, `setUp`) stand for specific program entities.

An intuitive introduction to these symbols is given throughout Part I of this book. Although precise definitions are offered in the opening of each section, an informal tone is kept throughout this part. Part II offers a more rigorous approach to specifications, whereas formal definitions are summarized in Appendix II. More specifically, Part II offers a formal semantics for Codecharts, defined as finite structures in model theory. A decidable abstract semantics function maps every “program” (which is taken to be defined as an expression in the programming language) into a finite structure (Chapter 14). These can be enriched with higher order (namely sets of) classes and methods to become design models, which are axiomatized in the first-order logic.

In Part II of this book we also define precisely the relation between (the decidable, non-functional, LePUS3) specifications and programs. In particular, we can resolve conclusively the question of whether a particular program indeed constitutes an implementation of a specific design pattern as formalized in our design description language. We also show that automated verifiability affords us to delegate the responsibility for resolving such questions to a fully automated process. For example, in Proposition 3 (p. 186) we prove that package `java.util` implements the Iterator pattern, and in Proposition 4 (p. 186) we prove that package `java.awt` implements the Composite pattern; both of which can be automated using the Two-Tier Programming Toolkit (see next section).
Rigour also allows us to carry out some reasoning on Codecharts at a level of confidence that otherwise cannot be achieved. For example, in Part II we prove that any visual token can be safely removed from a chart without invalidating it (the principle of information neglect). We also prove a number of propositions concerning the relationships between design patterns. For example, in Proposition 5 (p. 203) we prove that the Iterator pattern is a “special case” of the Factory Method pattern (Iterator ⊑ FactoryMethod).

3.4 AUTOMATED VERIFIABILITY

Rigour enables us to use a mathematical theory to understand the properties of LePUS3, including the property of decidability (p. 11). Unlike most design description languages, LePUS3 is fully turing decidable. By this we mean that, at least in principle, the verification question (p. 11) can be resolved fully automatically. Loosely speaking, this notion implies that given a LePUS3 specification (a Codechart) S and a program p we can determine the maximal (computation) time and space resources which require a computer program to compute the answer to the question whether p satisfies S. In fact, not only can this be done in principle, but also it is easy to show that it can be done in a relatively short time.5

To answer the verification question (p. 11), we must establish formally the relation between specifications and programs. The relation between Codecharts and Java programs is formally established in Part II of this book. Specifically, in Chapter 9 we define a set of truth conditions under which a specification is satisfied: Each formula imposes a set of Tarski-like truth conditions on design models. A specification S is satisfied by program p if and only if a design model M can be constructed such that M is an appropriate representation of p and M satisfies (the truth conditions in) S.

Having established these conditions, we can turn to the question of whether automated design verification is possible in practice. The Two-Tier Programming Project at the University of Essex has been concerned with investigating the feasibility of a tool implementing the verification algorithm. Our research has shown that, with a design model, the complexity of the verification algorithm is at most squared in the number of entities in the interpretation of the constants in a closed specification. In an open specification, the complexity is at most squared in the number of entities in the range of the assignment provided. Furthermore, we have implemented the verification algorithm in version 0.5.1 of the Two-Tier Programming Toolkit [http://ttp.essex.ac.uk/], which we shall simply refer to as the Toolkit (see §15.4). For example, at a click of a button, the Toolkit can establish whether a given Java 1.4 program implements the Composite design pattern. Verification of design pattern implementations typically requires less than a second to produce a definitive answer, as demonstrated in Figure 3-5.

5That is, in computational complexity that is squared in the size of the relations in the design model.
3.4 Automated Verifiability

Figure 3-5. Screenshot of the TTP Toolkit showing the results of verifying that package `java.awt` implements the Composite pattern (§15.3)

If violated, the Toolkit indicates which parts of the specification have been violated, thereby enabling the programmer to restore the consistency between design and implementation. In §15.4 we expand on this facility.

As a consequence of our commitment to automated verifiability (more precisely, to decidability), Codecharts cannot always be used in the same way as other design description languages. For example, the statement which requires that a certain event (creating an instance of class `a`) shall precede another (creating an instance of class `b`) under all possible executions, which can be articulated in modelling languages such as UML and formal specification languages such as Z, is undecidable and cannot be represented using a Codechart. Indeed, as a consequence of the decidability requirement, LePUS3 is unsuitable for representing a range of specifications concerning the behaviour of programs. In exchange for giving up such statements, Codecharts can be verified automatically. Have we made a wise choice?

To answer this question, let us return to our analysis of the inherent properties of contemporary software technology (Chapter 1). Conformity relates to the difficulty that arises from our inability to enforce specifications. Programs do not conform to our expectations at the most basic level because there are no effective tools that can help us enforce them. Undecidable modelling languages have so far crippled development and CASE tool manufacturers because, even if the meaning of specifications is well established, there can be no effective means for enforcing them.
And even when conformance is established for yesterday's software by whatever means (e.g., using testing and program inspection), changeability dictates that today's software may not conform to our specifications. Software is in a continuous state of flux. Inconsistencies between design and implementation must be detected early in the development process and repeatedly sought for throughout each and every stage of evolution. Such inconsistencies must be resolved by either fixing the implementation or changing the specifications. Unless the process is fully automated, problems such as architectural erosion and architectural drift are bound to arise.

Finally, automated verification is the ultimate tool for managing complexity. Only a continuous, repeated process of detecting and resolving inconsistencies between design and implementation can ensure that specifications are current and correct, thereby providing the kind of roadmaps for programs which are absolutely essential for managing the complexity of software systems of industrial scale.

### 3.5 SCALABILITY

A crucial trait of design description languages is their capacity for abstraction (p. 12), in particular their scalability, which we discussed extensively in Chapter 2 (see in particular p. 13). To ensure scalability we subscribe to the Feynman–Tufte Principle given in Table 7. Economy of expression is therefore a leading concern. By this criterion, a design description language is measured by its ability to represent “a lot of information” about a program compactly using as few symbols as possible.

**Table 7.** Feynman–Tufte Principle

| A visual display of data should be simple enough to fit on the side of a van. |
| Source: Shermer [2005]. |

In Codecharts we take scalability to require our design description language to allow effective representation of arbitrarily large systems with as few symbols as we choose. As we show in detail in §3.2, scalability is particularly useful in software visualization: Not unlike roadmaps, visual specifications generated by program visualization tools must be of any scale.

Scalability of a design description language is achieved, amongst others, by using abstractions that capture commonly recurring regularities in software design. For example, in Codecharts, the *ISOMORPHIC* predicate is particularly useful for conveying an abstract visualization of the one-to-one correlations between sets of methods. Consider, for instance, the use of this abstraction in modelling 23 of the methods\(^6\) in class *java.awt.Container*, each of which forwards the method’s arguments to a method with the same

---

\(^6\)There are many more methods in classes *Container* and *Component* which match this description but only 23 of them fit here.
signature in class `java.awt.Component`. The `ISOMORPHIC` predicate symbol in Codechart 5a (a double-headed arrow) captures the regularity between each pair of methods with matching signatures concisely and precisely.

Scale in Codechart 5b is achieved using abstractions such as sets (shaded shapes) and the `ISOMORPHIC` predicate symbol (double-headed arrow). Specifically, the two sets of methods are represented by superimposing a representation of the set of their common signatures (ContainerOps) over a representation of the two classes (`java.awt.container` and `java.awt.component`). The 1:1 correlation between methods in both sets is specified by the `ISOMORPHIC` predicate symbol.

**Codechart 5a.** Each of the 23 individual methods in `Container` forwards the call to the method it overrides in `Component`.

**Codechart 5b.** Each method in a set of methods in class `Container` forwards the call to the method it overrides in `Component`.
3.6 GENERICITY

Genericity is a principle most commonly associated with the abstraction of type information in programming languages like Java and C++. But genericity also plays a very important role at the specification level. We consider the principle of genericity to be that principle which dictates the distinction between the representation of concrete programs and abstract ideas. It dictates an explicit distinction between constants and variables. This distinction has two uses: abstraction for early design and the distinction between specific programs and design motifs. Below we demonstrate these uses, whereas Chapter 9 is dedicated to explaining the genericity mechanisms in the language of Codecharts.

Premature commitment to implementation minutia is one of the costliest mistakes of inexperienced software designers. As a language for representing blueprints of programs, a design description language must therefore allow us to represent only the constraints we wish to impose on the intended implementation without committing us any more detail than absolutely necessary. This requirement is referred to as the principle of abstraction in early design (Table 8).

Table 8. Principle of Abstraction in Early Design

| A design description language should focus on specifying constraints, not implementation detail. It must offer the software designer the means to refrain from making premature commitments to implementations during the early design stages (the blueprint metaphor). |

Some of the instruments required for representing the abstractions needed during early design, such as higher-order terms and predicates, have already been presented under the principle of scalability (§3.5). In addition, Codecharts may employ variables for describing nonspecific entities in the implementation. For example, in combination with the signature constant push, the variable some in Codechart 6 specifies that a class inheriting from Stack will eventually be implemented such that (a) it will contain a method that overrides Stack.push(), and (b) this method will contain a return statement with an expression of type Stack (or subtypes thereof). Note, however, that Codechart 6 does not commit us to any additional implementation detail: We are free to choose the name of that class, the number of levels of inheritance that separate it from class Stack, the number of fields and methods it implements in any way we see fit, and even the precise type of the return expression it contains (as long as it is a subtype of Stack).

Another shortcoming of informal modelling notations is their lack of clear distinction between specific implementations and abstractions, such as design motifs. In the absence of variables, modellers resort to specific examples. Why exactly are such specifications inadequate?

Which is not surprising given that most notations were tailored for the purpose of modelling specific programs.
In a nutshell, the level of abstraction that variables afford is indispensable if a line is to be drawn between a category of implementations—a design motif (such as a design pattern)—and programs. Consider, for example, the difference between the constant `testCase` in Codechart 4, which stands for a specific Java class called `TestCase`, and the variable `userTest`, which stands for some implementation chosen by the programmer that uses the JUnit application framework. `userTest` has no specific interpretation: Mapping it to class `MyUserTest` therefore constitutes the claim that class `MyUserTest` indeed satisfies the requirements that JUnit imposes on its clients. The introduction of variables therefore allows us to articulate specifications that are not tied in to a particular implementation.

![Codechart 6. Method Stack.push will be overridden by a method that returns an instance of Stack](image)

The failure to distinguish abstractions from implementations is also a hindrance during early design stages, because it forces software developers to make premature commitments to implementation detail. Consider, for example, how to capture the design decision that “method `Stack.push()` will be overridden by a method that returns an instance of `Stack`” without prematurely committing to details such as which class defines the overriding method, how many levels of inheritance separate it from `Stack`, and what is the actual return type of the method `push`. Codechart 6 demonstrates how variables in our language provide the abstraction mechanism by which this statement is precisely captured without making neither one of these unnecessary commitments (see Chapter 12: Modelling early design revisited).

### 3.7 MINIMALITY

We seek to accommodate Tony Hoare’s [1975] notion of *elegance* (p. 17) and thus have made every effort to reject of “the temptations of features and facilities”, and adopt “a passionate devotion to the principles of purity, simplicity and elegance”. In this spirit, the number of symbols in our design description language must be kept to a minimum, so long as it can be used to
capture and convey the building blocks of object-oriented design (described on p. 22):

- **Individual classes** and methods are represented using 0-dimensional terms.
- **Sets of entities** are represented using 1-dimensional (shaded) terms.
- **Relations between entities** are represented using arrow-headed edges.
- **Correlations between sets** of entities are also represented using edges.

Consequently, the vocabulary of Codecharts consists of only 15 visual tokens, depicted in Legend 1 (p. 23).

### 3.8 INFORMATION NEGLECT

To be rigorously defined, a design description language must specify not only what each symbol represents but also what exactly the absence of information means. But while this issue may first appear insignificant, it can be the source of considerable ambiguity. Let us demonstrate this point with an example.

Consider, for example, the class diagram in Figure 3-6: Does class **LinkedList** inherit from **Observable**? Does **LinkedList** define more than two methods? Does a program which contains more than the three classes modelled in Figure 3-6 satisfy it? In which one of these cases does the absence of a representation stand for negative information, and in which does it not?

Since the language of class diagrams is not well defined, none of these questions have definite answers. Clearly, in some cases the absence of a symbol implies negative information (e.g., **LinkedList** does not inherit from **Observable**). In other cases it implies nothing (e.g., there may be other classes in the program). Finally, in some cases it is not at all clear whether the specification is complete or partial. For example, is the list of methods in **LinkedList** exhaustive?

The principle of information neglect dictates that the absence of information from a Codechart implies nothing (Table 9). In other words, each specification in our design description language merely imposes a set of constraints on conforming implementations, and if an element of the program is not represented, this should be taken to require its absence.
Table 9. Principle of Information Neglect

| Absence of information in a specification does not imply negative information. |

There is one interesting consequence for the principle of information neglect: Every Codechart can be simplified by removing symbols from it. Any program that satisfies the first Codechart will also satisfy the simplified Codechart. In other words, if the simplified Codechart is valid, then it models a superset of the set of programs that the original Codechart described.

The principle of information neglect follows naturally from the principles of scalability and abstraction in early design. It greatly simplifies the semantics of the language. For example, it implies that the semantics are preserved by removing parts of the Codechart, namely by applying the operator of abstraction viz. information neglect (p. 202).

* For example, if a term is removed, then all formulas containing it must also be removed.