Form Methods Syst Des (2013) 43:1–28
DOI 10.1007/s10703-012-0181-1

Modeling and visualizing object-oriented programs with Codecharts

A.H. Eden · E. Gasparis · J. Nicholson · R. Kazman

Published online: 10 January 2013

Abstract Software design, development and evolution commonly require programmers to model design decisions, visualize implemented programs, and detect conflicts between design and implementation. However, common design notations rarely reconcile theoretical concerns for rigor and minimality with the practical concerns for abstraction, scalability and automated verifiability. The language of Codecharts was designed to overcome these challenges by narrowing its scope to visual specifications that articulate automatically-verifiable statements about the structure and organization of object-oriented programs. The tokens in its visual vocabulary stand for the building-blocks of object-oriented design, such as inheritance class hierarchies, sets of dynamically-bound methods, and their correlations. The formalism was tailored for those pragmatic concerns which arise from modeling class libraries and design patterns, and for visualizing programs of any size at any level of abstraction. We describe *design verification*, a process of proving or refuting that a Java program (i.e. its native code) conforms to the Codechart specifying it. We also describe a toolkit which supports modeling and visualization with Codecharts, as well as a fully-automated design verification tool. We conclude with empirical results which suggest gains in both speed and accuracy when using Codecharts in software design, development and evolution.

Keywords Design notations and documentation · Object-oriented programming · Patterns · Design concepts

A.H. Eden (✉) · E. Gasparis
School of Computer Science & Electronic Engineering, University of Essex, Colchester, UK
e-mail: eden@essex.ac.uk

J. Nicholson
School of Computing, Engineering and Mathematics, University of Brighton, Brighton, UK

R. Kazman
University of Hawaii, Honolulu, USA

R. Kazman
Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, USA

The design of computing systems can only properly succeed if it is well-grounded in theory, and ... the important concepts in a theory can only emerge through protracted exposure to application.
– Robin Milner

1 Introduction

In No Silver Bullet [1], Brooks attributes the difficulties that software development projects face to four inherent properties of software.[1] We believe that these difficulties have remained amongst the practitioners' central challenges. Complexity: When combined, software systems constitute some of the most complex artifacts ever manufactured. Invisibility: Software is intangible and, in itself, invisible. This means that software is hard to visualize, and also that design flaws are particularly difficult to detect and resolve. Conformance: Apart from the problem of designing complex programs, enforcing conformance to design decisions is largely an open problem. Manual verification, if at all possible, is impractical for many projects, and automated and semi-automated verification tools are yet to enter the mainstream. Changeability: Many organizations choose to write programs in an iterative, agile process of software design, implementation and evolution, which demands that the programmers involved engage in software modeling and visualization throughout the project's lifecycle, for example by using alternating use of forward and reverse engineering tools[2] [2].

Codecharts were developed specifically to address these difficulties. The language is tailored to fit the practical needs of modeling and visualizing the structure and organization of programs written in class-based object-oriented (OO) languages such as Java, C++, C#, Object Pascal, and Eiffel. Often, these needs arise during all stages of the software engineering cycle. The language has therefore evolved from lessons drawn from research in software analysis, design, modeling, formal specification, evolution, visualization, reverse engineering and design recovery, as well as the authors' practical experience. It also seeks to combine the distinctive advantages of existing formal specification languages and modeling notations, with emphasis on abstraction, scalability, and automated verifiability.

We also sought to bridge the gap between theory and practice.[3] Our theoretical investigation was guided by the practical needs for more effective tool support in modeling and visualization activities. The language is defined in mathematical logic: Each Codechart is unpacked as a formula in a fully-decidable subset of the first-order predicate logic (Sect. 5), and the relation between design and implementation is precisely defined. Consequently, checking the conformance of programs to Codecharts can be fully automated (Sect. 6).

Codecharts address a variety of common problems in software development. Some of their benefits to specific problems have been described in technical reports [4–6], websites [7, 8], conference proceedings [9–11], and a book describing how to use them in practice [12]. This paper focuses on the contribution of the language to software design, evolution, understanding and verification, demonstrating how the use of Codecharts in each activity increases its benefits to developers.

[1] Adapted to our presentation.

[2] In particular as implied by the first ("A program must be continuously adapted or else it becomes progressively less satisfactory") and second ("As a program evolves its complexity increases unless work is done to maintain it") laws of software evolution [2].

[3] Christopher Strachey said about this dissociation: "It has long been my personal view that the separation of practical and theoretical work is artificial and injurious. Much of the practical work done in computing, both in software and in hardware design, is unsound and clumsy because the people who do it have not any clear understanding of the fundamental design principles of their work. Most of the abstract mathematical and theoretical work is sterile because it has no point of contact with real computing" [3].

Table 1 Central questions

1.	The Ontological Question (Sect. 2). Can the language be used to model and visualize the logical structure and organization of OO programs? Can its building-blocks capture such programs at any level of abstraction effectively?
2.	The Visualization Question (Sect. 3). Can representations be articulated visually? Can design decisions ('program blueprints') be visualized? Can the design of existing programs (as 'program roadmaps') be visually represented, for instance as a result from design recovery?
3.	The Scaling Questio**n** (Sect. 4). Can arbitrarily-large programs be represented abstractly, uncluttered by implementation minutiae?
4.	The Conformance Question (Sect. 5). Can we verify the conformance of an implementation to a design specification (e.g., of a Java program to a Codechart)? Can conflicts be detected and identified?
5.	The Automation Question (Sect. 6). Can the Conformance Question be answered by a program? That is, can conflicts between design and implementation be detected automatically?

1.1 Scope

To illustrate the scope and objectives of Codecharts, we pose five central questions that arise from software modeling and visualization activities throughout the software lifecycle (Table 1). The remainder of our discussion is focused on the (positive) answers that Codecharts offer to these questions.

Unlike most notations, Codecharts can be used both for modeling possible programs and design patterns, as well as for visualizing existing programs. The purposes such diagrams serve can be illustrated using the metaphors of *roadmaps* and *blueprints*:

– During forward engineering, Codecharts can be compared to 'program blueprints': visual specifications modeling design decisions concerning programs under construction and design patterns prescribed by software designer. For example, Fig. 6 depicts the blueprint of the Decorator pattern, specifying classes, sets of classes, sets of dynamically-bound methods, and their relationships in *every* (correct) implementation of this pattern.
– During reverse engineering, Codecharts can be compared to 'program roadmaps': diagrams visualizing the logical structure and organization of existing programs, generated by a design recovery tool by analyzing the implementation. For example, Fig. 10 offers a roadmap of the Java 3D API generated by our design recovery tool, depicting inheritance class hierarchies, sets of dynamically-bound methods, and their relationships in the software package.

To illustrate how Codecharts can promote forward, reverse, and round-trip engineering, we developed a set of tools modeling and visualizing programs with Codecharts called the Two-Tier Programming Toolkit (henceforth, the Toolkit). We demonstrate below how the Toolkit can be used to support these activities. We proceed with a pilot study, the results of which suggest that Codecharts improve both the speed and accuracy of developers in forward and reverse engineering tasks within the scope of the language. Using the language does not require special training or long-term commitment to the notation, specific dialects, programming languages, tools or development environments.

1.2 Context

Codecharts can be used both for modeling (or specifying) and visualizing object-oriented programs. Below we clarify this central distinction and place Codecharts in the context of related languages.

Modeling and *specification* languages represent design decisions about programs that are yet to be implemented. Such decisions can be represented using various notations such as Class and Interaction Diagrams, architectural description languages [13], design pattern specification languages [14], and Dataflow diagrams. Some modeling languages lack a means to distinguish between specific implementations (e.g., class `InputStream`) and generic participants in a design motif (e.g., the *component* participant in the Decorator pattern). Also, many notations were tailored to represent implementation minutiae, thereby limiting their use in effectively representing the abstract building-blocks of object-oriented programs such as inheritance class hierarchies and sets of dynamically-bound methods. Consequently, tools that attempt to *visualize* large programs using these modeling notations may not always provide useful insights [15]. Finally, modeling notations can capture and convey human intuitions, but those are often ambiguous and their implementation cannot be verified [16], which limits the productivity gains that programmers can expect from modeling and visualization tools.

In contrast, program *visualization* notations are specifically designed to visualize information about programs via reverse engineering. They are useful for representing metrics such as size of source code, complexity and distribution of method calls. Visualization notations however were not designed to support modeling design decisions (such as the use of patterns) for unimplemented programs.

Formal specification languages (e.g., [17–25]) are rigorously defined notations, accompanied by a mathematical framework based on well-understood theories such as first-order logic, temporal logic, set theory, and graph grammars. Specifications in formal languages such as Z [26] can be used to “fix” the requirements and analyze them. For example, RBML [27, 28] formalizes both structural and behavioral descriptions of patterns. However, the conformance of programs to specifications in many formal systems cannot be verified fully automatically (Sect. 6).

Most formal languages employ a symbolic mathematical notation whose adoption is likely to require specialized training. Visual languages, in contrast, are notations for graphical and diagrammatic representation which can depict programs in a range of level of abstractions (e.g. [17, 29, 30] and [31]). Visual notations are motivated by the mind’s natural aptitude for spatial reasoning. Roadmaps, building blueprints, floorplans, and electrical circuit diagrams visually convey complex information. It is well established that visual ‘explanations’ of complexity are more readily and intuitively grasped than symbolic formulas and require less training to be used effectively [32]. For example, Statecharts [17] is a formal and visual modeling language for specifying details about the relation between the input to the program and its internal ‘state’.

UML [31] is a powerful and expressive collection of notations suitable for many software development tasks. Many visual modeling languages base their notations on UML or dialects thereof. For example, the Design Pattern Modeling Language [33] is defined as a UML metamodel, offering a conservative extension to the language which is accessible to programmers and also capable of effectively modeling both programs and design patterns as first class objects. Tools in the DEMIMA framework [34] can check the conformance of source code to design patterns specified in the Pattern and Abstract-level Description Language (PADL), which translates UML diagrams to a constraint-based language. Specifications are expressed as Java data structures implemented using the Ptidej tool suite. However, the authors do not describe a tool that can create PADL models from visual specifications.

UML and its dialects are generally insufficient for answering some of the central questions listed in Table 1. Automated verification is hampered by the fact that “no formal definition exists of how UML maps to any particular programming language. You cannot look at a UML diagram and say exactly what the equivalent code would look like" [16]. Being a modeling language for programs rather than design motifs, its vocabulary lacks variables, hence "UML cannot be used to describe an infinite set of pattern instances because the language is not designed for that purpose" [35]. Finally, as a modeling language, it is yet to be demonstrated that UML can be effective for program visualization (Sect. 4).

1.3 Caveat

Codecharts are not supposed to replace any of the existing modeling and visualization languages. Rather, the language seeks to offer a novel conceptual tool for communicating and enforcing design decisions, and for understanding programs. Codecharts are poorly suited to expressing statements that are not visually articulated and Turing-decidable formal statements about programs in class-based languages. For example, Codecharts cannot represent functional specifications, including dynamic or behavioral properties such as constraints on sequences of events, algorithms, and objects. Owing to their minimal vocabulary, Codecharts cannot effectively represent design decisions concerning programming paradigms, architectural styles, and design patterns that are not purely object-oriented. Obviously, due to its rigorously defined semantics, Codecharts cannot represent informal and un-verifiable specifications.

1.4 Outline

Sections 2 through 6 offer answers to the ontological, visualization, scaling, conformance and automation questions (Table 1) respectively. In Sect. 7 we show how Codecharts support software evolution using our round-trip engineering tool. In Sect. 8 we discuss the empirical results of two controlled experiments with the Toolkit and explore some of the practical benefits the Toolkit stands to deliver to programmers. Section 9 concludes with a discussion of the pros and cons of Codecharts and future work.

2 The ontological question

Let us demonstrate the answer Codecharts offers to the Ontological Question using a small example drawn from Java's `java.io` package [36], depicted in Table 2. The building-blocks in the design of this program are as follows:

Individual classes, methods and method signatures are the most elementary entities. A method's *signature* is defined by its name and argument types. For example, `Input-`

Table 2 Code excerpts from the `java.io` package [36]

```
public abstract class InputStream
  implements Closeable {
   public abstract int read(); ...
}
public class FileInputStream
  extends InputStream {
  public int read() ...
}
public class ByteArrayInputStream
  extends InputStream {
  public int read() ...
}
```

```
public abstract class FilterInputStream
  extends InputStream {
  protected InputStream in;
  public int read() { return in.read(); }
}
public class BufferedInputStream
  extends FilterInputStream {
  public int read() { ...
}
public class LineNumberInputStream
   extends FilterInputStream {
  public int read() {
     ...c = in.read(); ...
}
```

`Stream` and `Int` are classes, `read()` is a signature, and `InputStream.read()` and `FilterInputStream.read()` are methods.

Relations represent properties of entities and relationships between them. For example, the entity `InputStream` is said to be in the unary relations *Class* and *Abstract*. The pair ⟨`FileInputStream,InputStream`⟩ in relation *Inherit* represents the 'extends' relation between the respective classes. Other binary relations include *Create* (indicating the relation between a method creating objects and the type of the objects created) and *Forward* (a call from one method to another). Relations can also represent other decidable properties, such as *Interface*(x) (x is a Java interface) *Private*(x, y) (x is a private member of class y), *Throws*(x, y) (x may throw an exception of type y) and so on.

Rather than treating classes as containers of methods and attributes, each class (and static type) and method is a separate entity that may be related by the binary relation *Member*. For example, the pairs ⟨`FilterInputStream,InputStream`⟩ and ⟨`FilterInputStream,FilterInputStream.read()`⟩ are both in the relation *Member*, representing the relation between class FilterInputStream, its attribute ('data member') `InputStream` and method ('function member') `read()`.

Higher-dimensional classes and methods are finite sets of entities. For example, `InputStream` is a *class of dimension 0*, whereas `DecoratorsHrc`, defined as the set

```
DecoratorsHrc = {FilterInputStream,BufferedInputStream,
                 LineNumberInputStream}
```

is a *class of dimension 1*. Java packages and C++ namespaces can be represented as *classes of dimension 1*. Methods and signatures of higher dimensions are symmetrically defined. Consider the following examples modeled in Fig. 4: The set of dynamically bound methods with signature `createRetained` defined in `SceneGraphObject` and its subclasses is a *method of dimension 1*. And the set of sets of dynamically bound methods with some signature in the set `setLive` defined in `SceneGraphObjectRetained` and its subclasses is a *method of dimension 2*.

An **Inheritance Class Hierarchy** (in short, *hierarchy*) is a class of dimension *1* which contains a 'root' class that all other classes in the set inherit from. For example, `DecoratorsHrc` is not only a *class of dimension 1* but also a *hierarchy*.

In philosophical terms, these building-blocks describe the ontological commitments [37] underlying Codecharts. In practice, they force the software engineer to think about OO design in terms of sets of dynamically bound methods, class hierarchies etc. This paper demonstrates some of the programs and patterns that can be adequately described in these terms. Additional examples can be found in [5, 9–12] and [38].

3 The visualization question

Visual notations are popular because of the advantage of diagrams in conveying complex details [39]. Consider roadmaps, which are useful because their vocabulary lends itself naturally to the reasoning of drivers. Consequently, drivers rarely need to consult with a roadmap's glossary: thin lines evidently stand for narrow roads, thick lines for motorways and pictures of petrol pumps stand for petrol stations. Similarly, floorplans typically visualize walls, doors and windows. Seeking to avoid the confusion that arises from bloated vocabularies, these notations limit themselves to a set of visual tokens directly relevant to the process of reasoning that each is designed to support. The motivation for our vocabulary is best summarized by Hoare [40]:

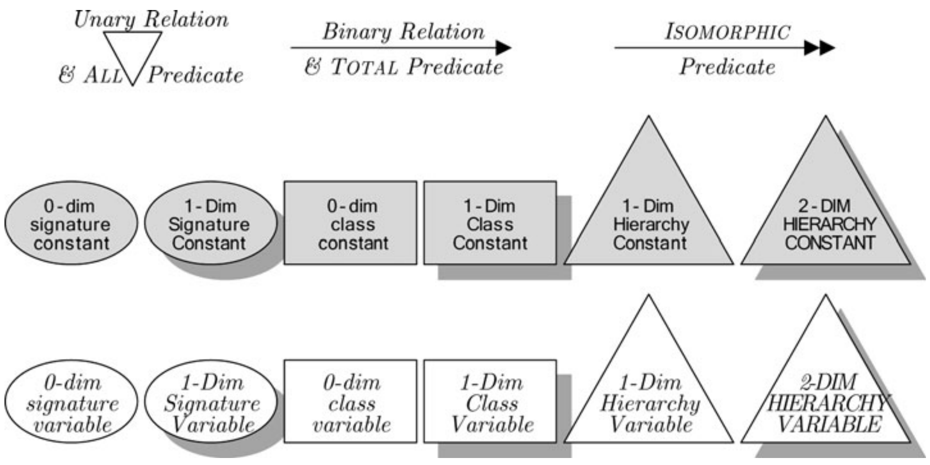


Fig. 1 The vocabulary of Codecharts

We need a puritanical rejection of the temptations of features and facilities, and a passionate devotion to the principles of purity, simplicity and elegance.

Designed after this principle, the vocabulary of Codecharts (Fig. 1) is restricted to capture and convey statements about the building-blocks of OO design (Sect. 2).

The grammar and semantics of Codecharts are laid out in precise terms in [4]. Below we sketch the vocabulary using examples drawn from the Codecharts below:

- **Relation symbols** stand for unary and binary relations. For example, the unary relation symbol *Abstract* over the signature constant `close()` in Fig. 8 indicates that the method with such signature in class `Closeable` is abstract. Similarly, the binary relation symbol *Member* connecting `FileInputStream` to `InputStream` in Fig. 9 indicates that class `FileInputStream` has an attribute (a.k.a. data member) of type `InputStream`.
- **Terms** (constants and variables) stand for entities: *0*-dimensional terms represent individual entities and *1*-dimensional terms represent finite sets of entities. For example, the class `Reader` in `java.io` is modeled in Fig. 8 using the *0*-dimensional class constant marked as such, and the set of classes that inherit from it are modeled as the *1*-dimensional class constant `SubClassesOfReader`. *2*-dimensional hierarchy terms stand for sets of hierarchies, such as OTHER-HIERARCHIES in Fig. 10.
- **Predicate symbols** stand for predicates. The *Total* predicate denotes a left total binary relation (unpacked as demonstrated by condition (f) in Table 3). The *Isomorphic* predicate is used to indicate a bijective functional relation.
- **Formulas** combine a relation symbol with terms and a predicate symbol.

Methods are represented by superimposing the respective (set of) signature(s) over the respective (set of) classes (or class hierarchies). For example, the method with signature `close()` in `Closeable` is modeled in Fig. 8 by superimposing the signature constant over the class constant, respectively. The set of dynamically-bound methods with the signature `createRetained()` defined in classes in the hierarchy `NodeHrc` are similarly modeled in Fig. 10 by superimposing the respective constants. And the set of methods with any one of the signatures in the set represented by the *1*-dimensional signature constant

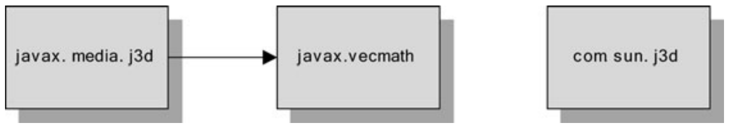


Fig. 2 Codechart modeling top level packages in the Java 3D API

`ReaderOps` in class `Reader` is modeled in Fig. 8 by superimposing the respective constants.

Variables and constants are used in the usual manner. Constants may refer to any entity of same type and dimension whose identity is fixed by an interpretation function. Variables can be assigned to any number of instances of same type and dimension as demonstrated in Sect. 5.

Each Codechart is unpacked as a formula in a subset of the first-order predicate logic (FOPL) as defined in Chap. 9 in [4].[4] The unpacking is demonstrated in the examples below. However, since checking conformance to Codecharts is fully automated (Sect. 6), details of the formalism need not concern most programmers, who can use the notation without mathematical training.

Below we demonstrate how the vocabulary of Codecharts is used in software modeling and visualization.

3.1 Example: modeling Java 3D

We now show how Codecharts can be used to encode design decisions by demonstrating how informal descriptions of the API to the Java 3D library are specified. The Java 3D library was developed at Sun Microsystems for rendering interactive 3D graphics. It consists of several hundred classes and learning to use it effectively poses a challenge even for experienced programmers.

The following passage describes the organization of the API into three top level packages:

> The core Java 3D API classes are in the javax.media.j3d package. Java 3D also includes a collection of vector and matrix classes that ... are in the javax.vecmath package. ... A set of useful utility classes is included in the com.sun.j3d package tree. In practice, all three packages are available to all Java 3D programs, and the set of packages is considered the Java 3D API. [41]

Figure 2 illustrates how this simple description is encoded formally as a Codechart, which also indicates some dependency between two of the packages.

For example, the class constant `com.java.j3d` in Fig. 2 is unpacked in the FOPL as the following sentence:

$$\forall x \in \texttt{com.java.j3d} \bullet \mathit{Class}(x)$$

Next let us consider the organization of classes that support the scene graph, a data structure central to the use of the library which comprises a representation of all the objects in the graphical scene. These classes are described in the Javadoc documentation of the library as follows:

[4]This work was further developed into a Theory of Classes in the Typed Predicate Logic [38].

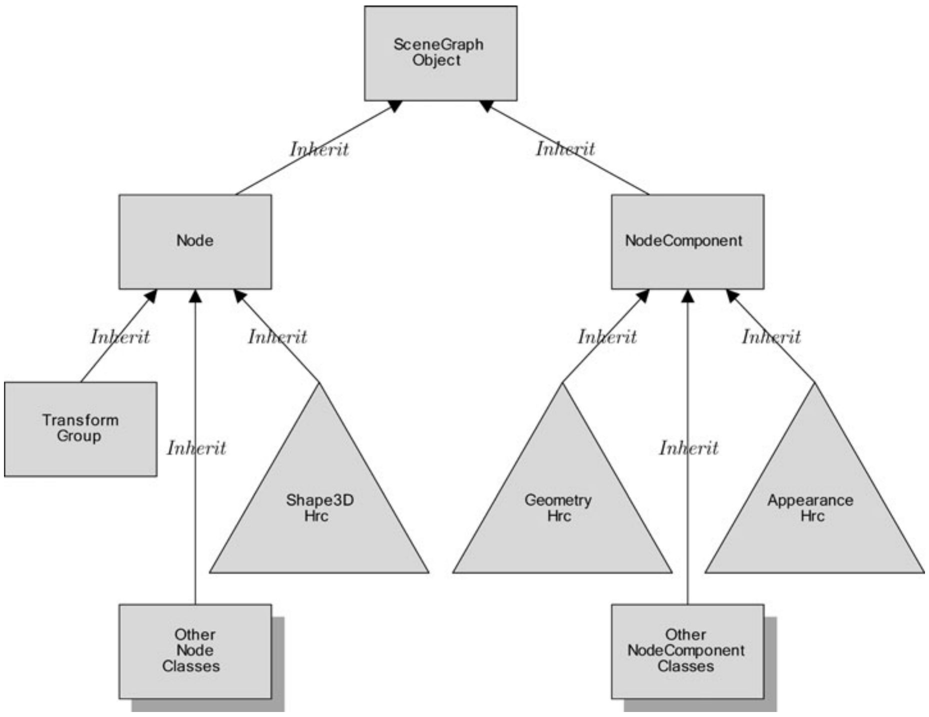


Fig. 3 Codechart modeling class `SceneGraphObject` and some of its subclasses

> SceneGraphObject is the common superclass for all scene graph objects. Scene graph objects are classified into two main types: nodes and node components. The Node object is the common superclass of all nodes, which includes TransformGroup [and] Shape3D... The NodeComponent object is the common superclass of all node components, which includes Geometry [and] Appearance.

This description is formalized by the Codechart depicted in Fig. 3.

The formulas represented by the combination of the constants `SceneGraphObject` and `Node` and the relation symbol *Inherit* in Fig. 3 are unpacked in the FOPL as the following sentences:

Class`(Node)`
Class`(SceneGraphObject)`
Inherit`(Node,SceneGraphObject)`

Where $\mathit{Inherit}(x, y)$ is understood as the transitive closure of the relation *Inherit*, i.e.

$$\mathit{Inherit}(x, y) \triangleq \langle x, y\rangle \in \mathit{Inherit} \vee \exists z \bullet \langle x, z\rangle \in \mathit{Inherit} \wedge \mathit{Inherit}(z, y)$$

Finally, let us consider the following descriptions related to the set of 'retained' classes, which mirror the set of scene graph object classes. According to [42], the motivation for 'duplicating' the scene graph object hierarchy is to maintain an internal delegate class separate from the public implementation class defined by the specification, which allows the developers more leeway in modifying the implementation without breaking the API or exposing protected or package level access to methods or fields. The author also explains that each

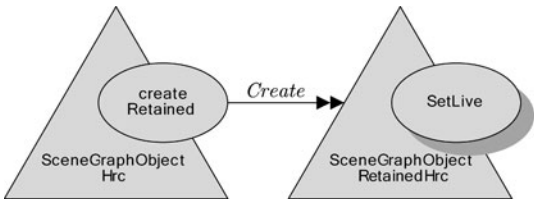


Fig. 4 Codechart modeling the scene graph and scene graph retained twin hierarchies, a set of dynamically-bound factory methods in the first hierarchy, and a set of sets of dynamically-bound methods in the second hierarchy. The *double-headed arrow* represents an isomorphism between three sets: Each class in the set of scene graph classes defines a `createRetained` method which creates entities of exactly one class in the retained hierarchy, and each class in the retained hierarchy is created by exactly one such method

`SceneGraphObjectRetained` class implements several `setLive` methods that can be overridden to respond in a specific manner. The isomorphism between the two hierarchies is visualized by the Codechart in Fig. 4.

For example, the formulas represented by the combination of the constants `SceneGraphObjectHrc`, `SceneGraphObjectRetainedHrc`, `createRetained` and the isomorphic predicate *Create* depicted in Fig. 4 are unpacked in the FOPL as the following sentences:

Hierarchy `(SceneGraphObjectHrc)`
Hierarchy `(SceneGraphObjectRetainedHrc)`
ISOMORPHIC(*Create*,
  `createRetained` ⊗ `SceneGraphObjectHrc`,
  `SceneGraphObjectRetainedHrc`)

where—

$$\mathit{Hierarchy}(\mathit{Hrc}) \triangleq \exists \mathit{root} \in \mathit{Hrc} \bullet \forall \mathit{cls} \in \mathit{Hrc} \bullet \mathit{Class}(\mathit{cls}) \wedge (\mathit{cls} \neq \mathit{root} \Rightarrow \mathit{Inherit}^{+}(\mathit{cls}, \mathit{root}))$$

$$\mathit{ISOMORPHIC}(\mathcal{R}, X, Y) \triangleq \forall x \in X \exists y \in Y \bullet (\mathcal{R}(x, y) \vee x, y \in \mathit{Abstract}) \wedge (\mathit{ISOMORPHIC}(\mathcal{R}, X - \{x\}, Y - \{y\}) \vee X, Y = \{\})$$

3.2 Example: modeling the Decorator pattern

An accurate, meaningful and appropriately understood representation of design decisions about programs under construction is useful not only when implementation commences but throughout the software lifecycle. Properly communicated, such a representation can be taken to represent a 'contract' between the designers and programmers and between different teams of programmers. In absence of clarity and precision, design decisions are typically violated in practice [43].

Codecharts can be used to represent certain design decisions concisely and unambiguously, such as the structural aspects of design patterns as demonstrated in the Decorator (Fig. 5). Consider the design decisions modeled in the Codechart in Fig. 6 that represent a 'blueprint' of the Decorator pattern. Rather than unpacking it in long and complex formulas in the FOPL we sketch its *truth conditions* [12, 44] in Table 3. In the next sections we show how to prove that `java.io` conforms to this Codechart and how the Toolkit automates this process.

Note that all the terms occurring in Fig. 6 are variables. This reflects the fact that what is being modeled is a pattern: an abstraction representing a design motif, not a specific

Intent: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Participants:
- **Component** defines the interface for objects that can have responsibilities added to them dynamically.
- **ConcreteComponent** defines an object to which additional responsibilities can be attached.
- **Decorator** maintains a reference to a Component object and defines an interface that conforms to Component's interface.
- **ConcreteDecorator** adds responsibilities to the Component.

Collaborations: Decorator forwards requests to its Component object. It may optionally perform additional operations before and after forwarding the request.

Structure:

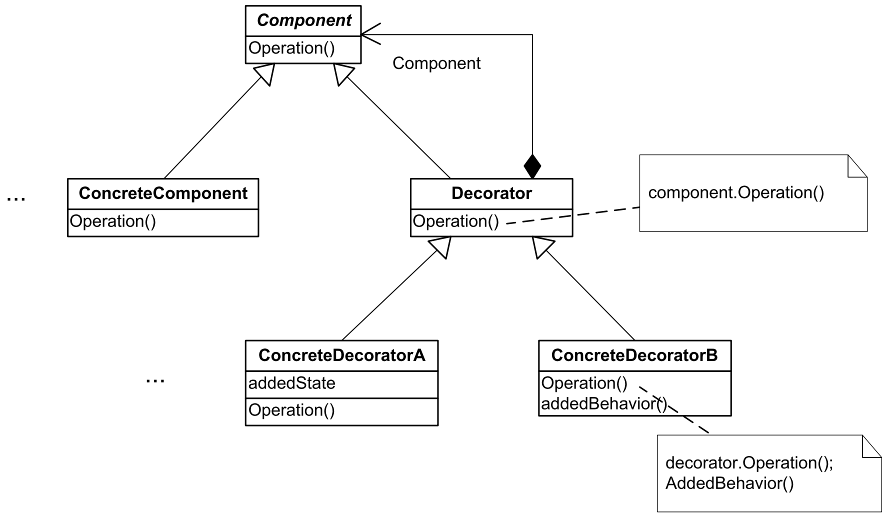


Fig. 5 The Decorator pattern (adapted from [45])

program. The distinction between generic (‘participant’) and specific (a specific class) is significant from many perspectives. Contrast this with the respective UML class diagram (Fig. 5) where such a distinction is not maintained.

3.3 Example: visualizing the `InputStream` classes

Many design recovery tools are reverse engineering tools that analyze source code, generate representations that are more abstract than the source code [46], and visualize the results. Design recovery is useful for understanding the structure and logical organization of an unfamiliar program: While physical organization (e.g. files and folders) is easy to obtain using existing tools, the logical organization of OO programs indicating correlations and clusters of cooperating classes and methods is typically less evident. Any implicit notion of ‘design’ can take a long time to extract manually from source code. Therefore, program ‘roadmaps’ that are produced by a design recovery tool can help program understanding.

The language of Codecharts was designed to facilitate software visualization using design recovery tools. Such tools can create Codecharts that analyze source code and create roadmaps at any level of abstraction we choose. For example, the Codecharts in Sect. 4

Fig. 6 A Codechart modeling the Decorator pattern

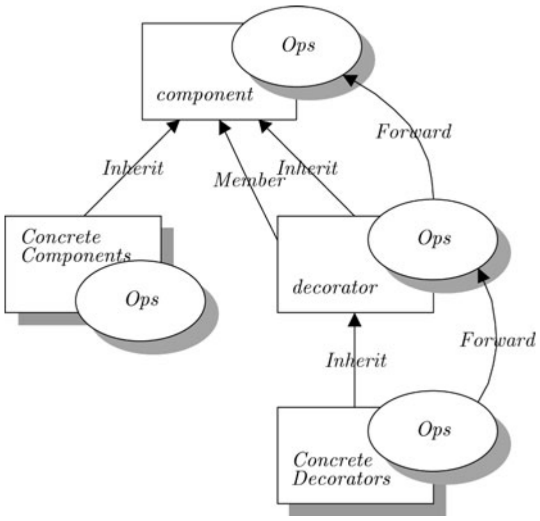


Table 3 Truth conditions for the Codechart in Fig. 6 (Decorator pattern)

Terms

(a) The variables *component* and *decorator* range over individual types (in Java: class, interface, or primitive type)

(b) The variables *ConcreteComponents* and *ConcreteDecorators* range over sets of types

(c) The variable *Ops* ranges over sets of method signatures

Formulas

(d) Every class in *ConcreteComponents* inherits (Java: extend or implement; possibly indirectly) from *component*

(e) *decorator* inherits from *component*

(f) Every class in *ConcreteDecorators* inherits from *decorator*

(g) *decorator* has a data member (aka *field*) of type *component*

(h) *component* defines a method for each signature in the set *Ops*

(i) *decorator* defines a method for each signature in *Ops*, and every such method forwards the call to the method it overrides

(j) Every class in *ConcreteComponents* defines (or inherits) a method for each signature in *Ops*

(k) Every class in *ConcreteDecorators* defines a method for each signature in *Ops*, and every such method forwards the call to the method it overrides

were generated by such a tool, the Design Navigator [9, 11, 47] (part of the TTP Toolkit), by analyzing the source code of package `java.io`.

The Codechart depicted in Fig. 7 ('Top Chart') offers the most abstract view of any program (or class library): `AllClasses` represents the entire set of classes and static types in the implementation—package `java.io` in our case. As such, a Top Chart is not particularly useful except as a starting point to design recovery. From here, the Design Navigator offers its users a range of operators (left panel, Fig. 7), each seeking to uncover how various parts of the software relate and visualize them in more detail. For example, if the user is interested in understanding the `Closeable` interface and its subclasses, the Design Navigator can produce a more detailed Codechart.

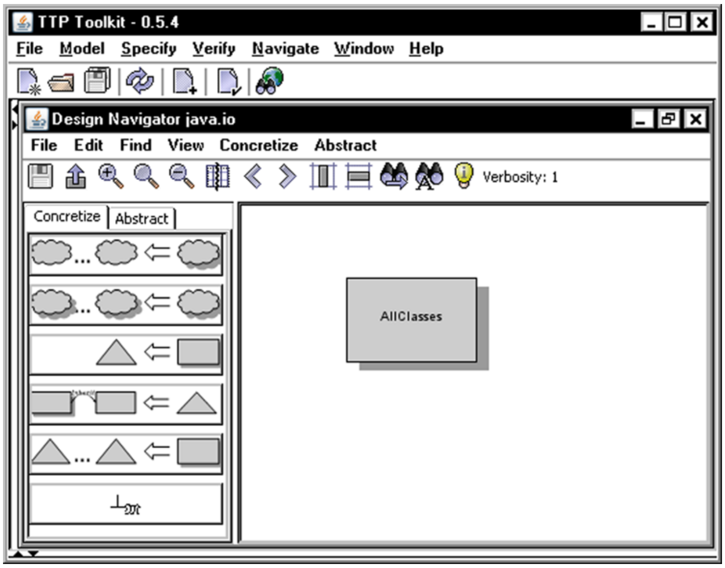

TTP Toolkit - 0.5.4
File Model Specify Verify Navigate Window Help
Design Navigator java.io
File Edit Find View Concretize Abstract
Verbosity: 1
Concretize
Abstract
AllClasses

Fig. 7 Top Chart

However, there are over 140 classes and subtypes that inherit from the Closeable interface. Since it is quite a large number, visualizing them as individual classes would be pointless. Instead, the Codechart depicted in Fig. 8 offers a compact abstraction: It shows that the Closeable hierarchy consists of four nested class hierarchies, each composed of subclasses of `Reader`, `Writer`, `InputStream` and `OutputStream`. It visualizes the principal division of labor between the 140 classes in this set. Other views of the same set of classes can also be generated according to needs.

It is often useful to visualize a smaller number of classes in greater detail, namely at even lower levels of abstraction. For example, programmers may be interested in visualizing the organization of the individual classes that handle input streams. The Codechart that the Toolkit generates for this purpose, shown in Fig. 9, depicts class `InputStream`, five of its subclasses, and the relations between them. For instance, it shows the relations between `FilterInputStream` and `InputStream`: class `FilterInputStream` has a member of type `InputStream`, inherits from `InputStream`, and has methods that forward the call to methods in `InputStream`.

Recovering information of this sort can be useful for a variety of software engineering tasks. Codecharts can also bring to the fore structural properties that the programmer may recognize. For example, the pattern of relations between `FilterInputStream` and `InputStream` depicted in Fig. 9 bears similarity to the Decorator design pattern (Fig. 6). Does `InputStream` conform to the Decorator? In Sect. 5 we show how this question is answered.

4 The scaling question

Some notations restrict themselves to a vocabulary of individual classes, fields, and methods. Such representations can visualize implementation minutiae in great detail, but are less useful for visualization tools when larger programs are concerned. To demonstrate this, Fig. 11

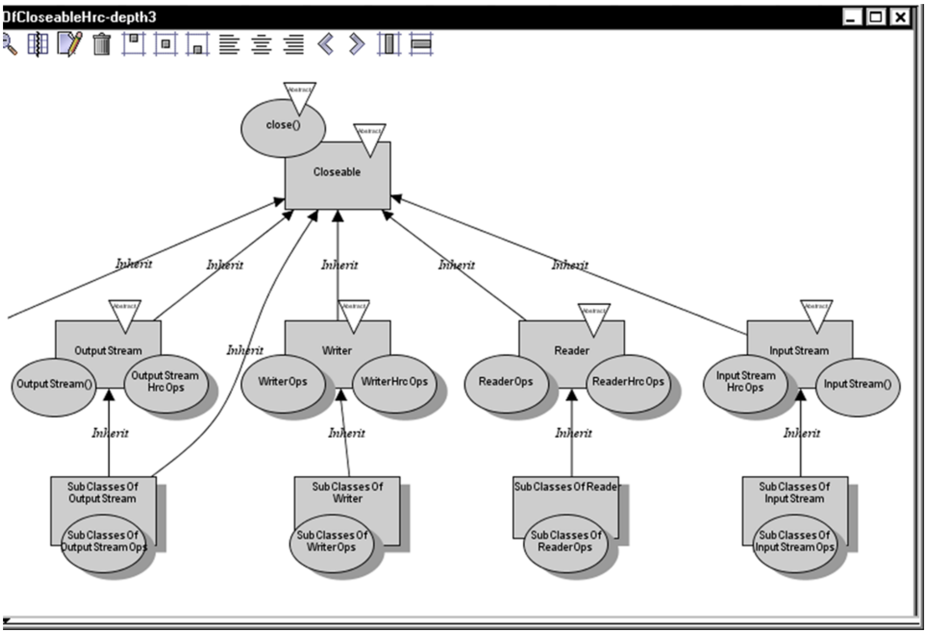


Fig. 8 Closeable hierarchy

(created using Fujaba) and Fig. 12 (created using NetBeans) depict the results of attempting to visualize the individual elements of a program of several dozen Java files.

Scalability is not idiosyncratic to software modeling. Roadmaps can cover entire continents, and blueprints can be used effectively in building gargantuan monuments. When modeling highly complex domains, it is useful to apply the Feynman-Tufte Principle [48], which requires that "A visual display of data should be simple enough to fit on the side of a van". In other words, scalability in the theory of information visualization [32] translates in our context to the demand that, regardless the size of the program, a visual representation must remain legible and uncluttered.

Some notations scale by using a separate symbol for each kind of module: subsystems, subprograms, processes, components, connectors, packages, libraries, namespaces, ports, sockets, layers, data structures, and so forth. Such notations can contribute to the communication between programmers. However, a profusion of symbols limits the support that tools can provide programmers and confuse them.

Codecharts trades expressiveness for the elegance afforded by a more minimal approach. Its vocabulary is limited to the mechanism of abstraction in its underlying ontology: zero- and higher-dimensional classes and methods. For example, a higher-dimensional class constant (shaded rectangle) can either be used to represent a Java package (or C++ namespaces), all the classes and interfaces that implement some Java interface, or an entire class hierarchy. It can also represent any set of special interest. For example, a higher-dimensional method can stand for a set of factory methods, or for the set of dynamically-bound methods in a class hierarchy, as well as a set of the above.

Consider for example the Codechart in Fig. 10, which models the same set of classes (the API to the Java3D library) as the diagrams in Fig. 11 and Fig. 12. Figure 10 employs

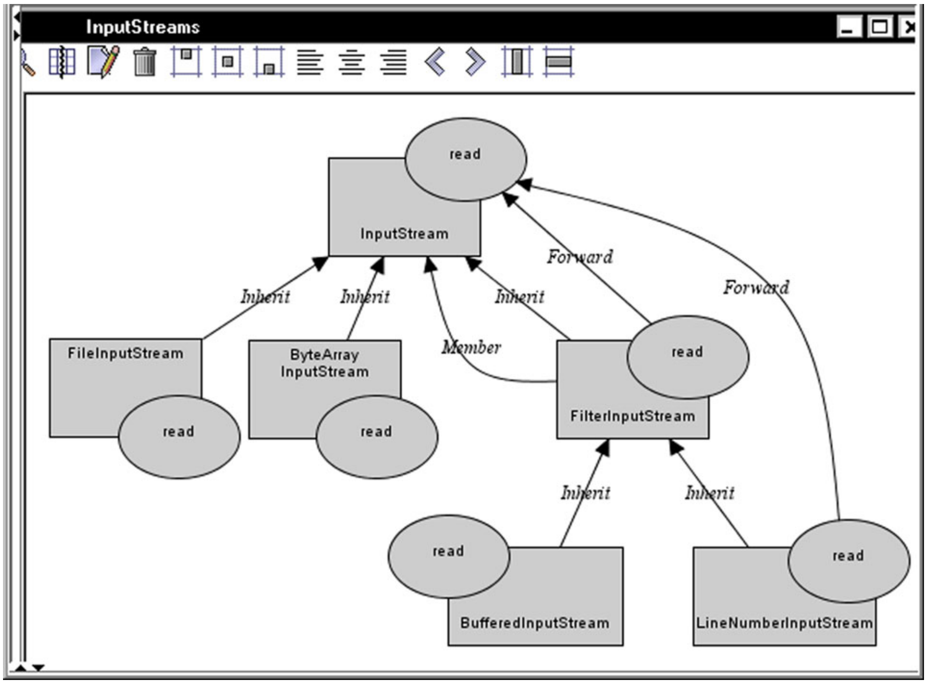


Fig. 9 `InputStream` and its subclasses, for simplicity, signatures other than `read` were abstracted

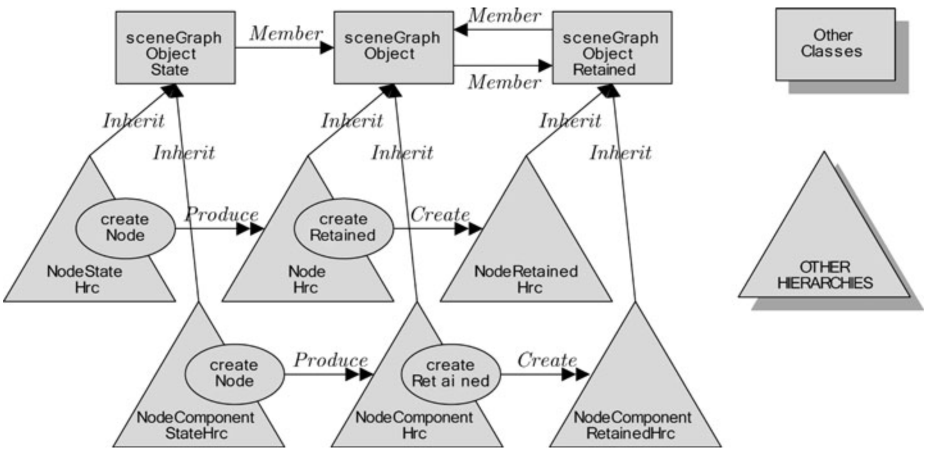


Fig. 10 Java 3D API modeled using a Codechart

several of the abstraction mechanisms to convey information that is not evident in Fig. 11 and Fig. 12. Specifically, it visualizes six central hierarchies and four sets of dynamically-bound methods. It also shows that the three hierarchies 'mirror' each other using the *isomorphic predicate* (double-headed arrows) which represents a 1:1 and onto correlation between them.

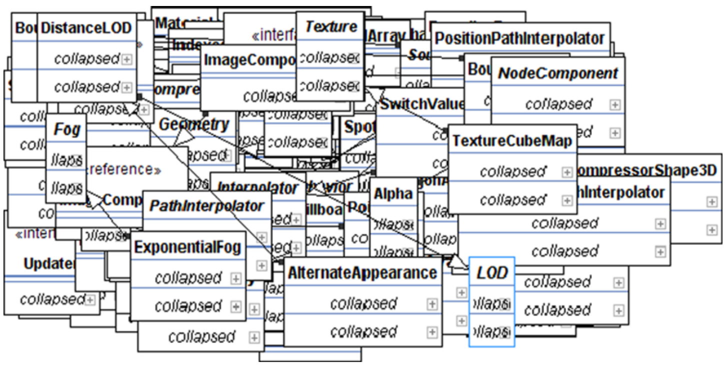
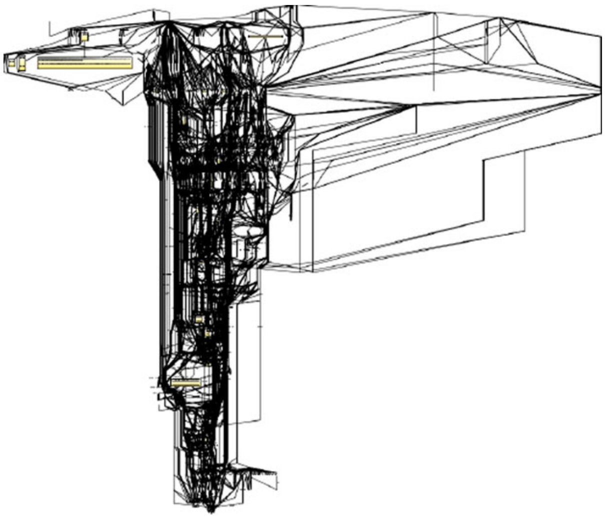


Fig. 11 Java3D class diagram generated by Fujaba [39]

Fig. 12 Java3D class diagram generated by NetBeans 6.1 [39]



5 The conformance question

The need to check whether design decisions were followed arises frequently throughout the software lifecycle. Checking conformance of a program to a Codechart is straightforward. Below we sketch some of the steps in proving (or rather refuting, in this example) the conformance of classes in `java.io` to the Decorator design pattern. The mathematical definitions and a more detailed *design verification* process can be found in [49] and in [5].

The similarity between Fig. 9 and Fig. 6 suggests that the `InputStream` classes may implement the Decorator pattern. Others, [50] for example, have also suggested this. Such conformance claims are difficult to check effectively and nearly impossible to enforce in practice unless some formal method is used to establish the meaning of the specification and of the implementation conclusively.

Our approach to this problem is guided by Guttag, Horning and Wing [18, 51], who define a formal method to consist of the following three elements:

– A *specification* (‘design’) Δ is a statement in a formal language representing [items in] the contract between the designers and the implementers of a program.

– A *specificand* (‘implementation’) p is a statement in a programming language.
– A *Satisfies* relation formalizes the conditions under which p *Satisfies* Δ.

The formal method we employ for Codecharts can be summarized in these terms as follows:

– A *specification* is a Codechart: a visual statement in the language of Codecharts. For example, the Codechart modeling the Decorator design pattern (Fig. 6) represents the set of statements whose truth conditions are unpacked in Table 3.
– A *specificand* is any Java program: a ‘well-formed’ statement in the grammar of the programming language (i.e., a compilable Java source code).
– The *Satisfies* relation is borrowed from model theory, denoted $\vDash$ in the standard manner.

This approach to the Conformance Question takes *design verification* to differ considerably from *program verification* in the tradition of Hoare Logic [52]. Hoare takes every statement in the implementation to be a mathematical expression which “describes with unprecedented precision and in every minutest detail the behavior, intended or unintended, of the computer on which they are executed.” [53] By this approach, verifying that $p \vDash \Delta$ is a process that translates each token in the program’s text into a symbolic formula in Hoare Logic. Furthermore, in the general case, *Satisfies* statements are not fully *Turing-decidable*. This means that program correctness cannot be established by a fully automated tool in the general case.

In contrast, the *design verification* approach to conformance checking allows a weaker notion of the *Satisfies* relation by introducing *semantic abstraction functions*. Thus, *design verification* can be defined as the process that seeks a proof to $\mathcal{A}(p) \vDash \Delta$ which, given *semantic abstraction function* $\mathcal{A}$, does not require an exhaustive formalization of p. Instead, it requires the representation of specific aspects of statements in the source code of p. By this approach, program p is abstracted into a set of facts about the program using a semantic abstraction function $\mathcal{A}$, which replaces the complex representation of the program by a ‘flat’ collection of relevant facts. These abstractions are then semantically analyzed to ensure that the constraints imposed by the specification—e.g., the truth conditions in Table 3—are not violated.

Formally, the semantics of a program is represented as a *finite structure*, a model-theoretic set of *entities* and *relations* (Sect. 2) which can be thought of as a relational database. We define $\mathcal{A}_{Java} : \mathbb{JAVA} \rightarrow \mathfrak{F}^*$, a semantic abstraction function that maps every Java program p (an element of $\mathbb{JAVA}$) into a finite structure $\mathcal{A}_{Java}(p)$ (an element of $\mathfrak{F}^*$). The conformance claim is therefore recast in this framework as claim (IS_1),

$$\mathcal{A}_{Java}(p)(\texttt{java.io}) \vDash \text{Decorator} \qquad (\text{IS}_1)$$

where $\mathcal{A}_{Java}(p)(\texttt{java.io})$ is the semantic abstraction of `java.io` and Decorator is the Codechart in Fig. 6.

Note that Fig. 6 includes variable terms (empty shapes), which stand for the pattern’s participants without referring to any specific implementation. Statement IS_1 can therefore be understood as some general claim that `java.io` implements the Decorator pattern without specifying where exactly. To check the validity of such a claim we must map each variable (participant in the Decorator pattern) to specific classes or methods (or sets thereof) in the program. This is done using a mapping g from variables in Decorator into classes and methods in $\mathcal{A}_{Java}(\texttt{java.io})$, formally known as an *assignment*. For example, assignment g defined in Table 4 maps the variables from Fig. 6 to the corresponding Constants in Fig. 9.

Table 4 Assignment g maps variables in Decorator to `InputStream`

$g(\mathit{component})$	= `FilterInputStream`
$g(\mathit{decorator})$	= `FileInputStream`
$g(\mathit{ConcreteComponents})$	= {`FileInputStream`,`ByteArrayInputStream`}
$g(\mathit{ConcreteDecorators})$	= {`BufferedInputStream`,`LineNumberInputStream`}
$g(\mathit{Ops})$	= {`read`}

The general statement formalized in (IS$_1$), which concerns the implementation of the Decorator anywhere in `java.io`, can therefore be replaced by a claim which specifically states which elements of the program must implement it, formalized in (IS$_2$).

$$\mathcal{A}_{\mathit{Java}}(\texttt{java.io}) \vDash \text{Decorator}\big[g(\mathit{component})/\mathit{component}, \ldots g(\mathit{Ops})/\mathit{Ops}\big] \qquad (\text{IS}_2)$$

Finally, note that the class `InputStream` and its subclasses do not satisfy the Decorator pattern (Fig. 6) because the methods `BufferedInputStream.read()` and `LineNumberInputStream.read()` do not forward the call to the method they override. This is shown formally by noting that claim IS$_1$ is not satisfied under assignment g, because the pairs ⟨`BufferedInputStream.read,FilterInputStream.read`⟩ and ⟨`LineNumberInputStream.read,FilterInputStream.read`⟩ are not in the relation *Forward*. This proves that `InputStream` and its subclasses with the method do not, in fact, conform to the Decorator pattern as represented in Fig. 6.

More generally, a statement such as IS$_1$ is true if and only if there exists some assignment f such that $\mathcal{A}_{\mathit{Java}}(\texttt{java.io})$ *satisfies* the consistent replacement of each variable x in Decorator with $f(x)$. Since the number of structures is finite, the *Satisfies* relation is fully-decidable.

This example demonstrates the nature of the conditions that Codecharts impose and how those are compared against the implementation. It also clearly distinguishes between different kinds of Conformance Questions (Table 1): *verification* vs. *detection*. Detecting instances of a given pattern (pattern mining) is required to determine more general statements, e.g., (IS$_1$) namely whether Decorator is implemented anywhere in `java.io`. Verification, on the other hand, concerns the process required to prove or refute a more specific claim, such as (IS$_2$), which specifies how each one of the *participants* of Decorator is supposedly implemented. Verification tools can be useful to software designers who wish to enforce specific design decisions, whereas detection tools can help the process of reverse engineering.

This example also highlights that the Conformance Question is both easy in the sense of being 'mechanical' (i.e. requiring little insight) and hard in the sense of being tedious and error-prone. Fortunately, the tedious process can be mechanized using an automated design verification tool, as demonstrated in the next section.

6 The automation question

Codecharts were tailored to the requirement of *automated verifiability*. What exactly does this mean and what is entailed by this commitment?

We indicated that the Conformance Question is fully-decidable. More specifically, *design verification* is Turing-complete, terminating within a predetermined (finite) number of steps for any input. In other words, inconsistencies can in principle be detected by a computer program.

Unfortunately, many desirable properties of programs are semi-decidable or undecidable, such as the requirement for a program to not 'crash' or hang forever (enter an infinite loop). The description of design patterns and other design motifs commonly include many undecidable statements. Undecidable properties are useful and important to articulate, in particular in safety-critical software systems where human life or significant assets are at stake. Consequently, most specification and modeling languages do not restrict themselves to decidable properties. However, in the general case, undecidable specifications cannot be verified automatically. In allowing undecidable specifications, notations rule out the possibility of full automation in the general case.

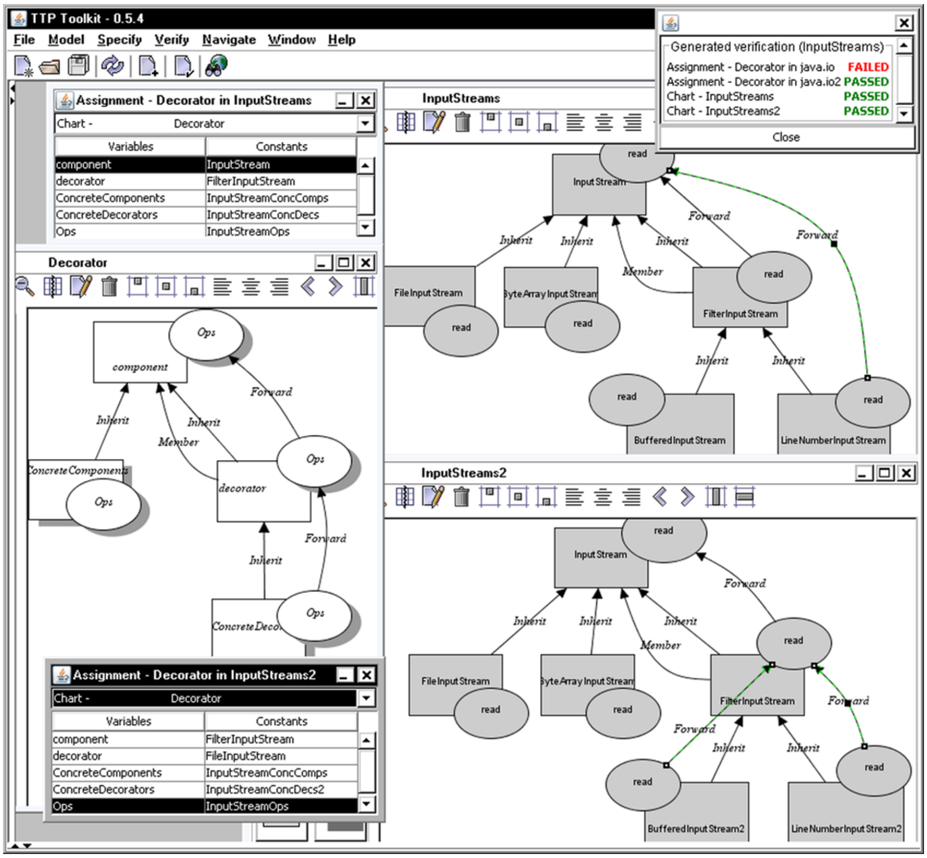
Verification requires a detailed analysis of the syntactic and semantic properties of the program's source code, which takes time, effort and special training. Manual conformance checking of large programs, whose source code spans hundreds of thousands or even millions of lines, is usually too expensive and time consuming, as well as error-prone. Consequently, in absence of tools for automated verification, formal methods have limited [54] use in conformance checking.

The *changeability* (Sect. 1) of software further exacerbates the problem (see for instance [2, 55] and [56]). At any stage in the software lifecycle the software's design and implementation may change. Since even the most subtle changes in the implementation may violate the design, and vice versa, conformance must be re-checked with each change. If conflicts are found then either the design or the implementation must be updated to restore consistency. Tools supporting fully automated conformance checking can reduce the cost of the detection process, and possibly help the team to prevent architectural drift and erosion [57] from developing.

Full-decidability (Sect. 5) implies that the answer to the Conformance Question can be automated *in principle*, but it does not guarantee that it is feasible. Fortunately, the computational complexity of the design verification algorithm of Codecharts is at most squared in the number of the entities and relations in the program and linear in the number of tokens in the Codechart [47]. The feasibility of automated verification for Codecharts and Java was also explored by the Toolkit.

Let us illustrate how the Toolkit supports the process of verification described in the previous sections, namely to create a Codechart specifying the Decorator pattern, to analyze package `java.io`, and to check that it conforms to the Decorator [49]. The process consists of five steps, the results of which are illustrated in Fig. 13.

(1) **Modeling** (or specification) with the Toolkit captures design decisions. It is done simply by creating any number of Codecharts. To support modeling, the Toolkit provides a Codechart editor which offers a palette with the visual tokens of the language and a canvas to which these tokens may be dragged-and-dropped.
(2) **Implementation** (programming) can be carried out using any development environment. Since the Toolkit analyses native source code, it does not interfere with the use of any editor, compiler or any other traditional programming tool and activity.
(3) **Program analysis** is a process of static analysis which creates the *abstract semantics* representation of the program from source code. Programmers can choose to analyze any number of files and directories at the click of a button. The Toolkit will analyze the files in the specified locations and create a relational database storing information about classes, methods, signatures and their membership in unary and binary relations. In the example depicted in Fig. 13 the Toolkit analyzes two programs: one is package



Assignment - Decorator in InputStreams

Chart - Decorator

Variables	Constants
component	InputStream
decorator	FilterInputStream
ConcreteComponents	InputStreamConcComps
ConcreteDecorators	InputStreamConcDecs
Ops	InputStreamOps

Generated verification (InputStreams)

```
Assignment - Decorator in java.io    FAILED
Assignment - Decorator in java.io2   PASSED
Chart - InputStreams                 PASSED
Chart - InputStreams2                PASSED
```

Close

Assignment - Decorator in InputStreams2

Chart - Decorator

Variables	Constants
component	FilterInputStream
decorator	FileInputStream
ConcreteComponents	InputStreamConcComps
ConcreteDecorators	InputStreamConcDecs2
Ops	InputStreamOps

Fig. 13 Screenshot of the Toolkit, showing the result of checking conformance of `java.io` to the Decorator pattern. It shows that the implementation as distributed by Oracle (modeled in Codechart InputSptream) does not conform to the pattern, but that after some minor tweaks (modeled in Codechart InputStream2), it does

`java.io` as distributed by Oracle, the second is a slightly modified version of the same package.

(4) **Assignment** consists of mapping variables in the Codecharts to classes and signatures in the implementation. For example, the top assignment in Fig. 13 maps the variables in the Decorator Codechart to elements in the existing implementation (modeled in Codechart `InputStream`) as defined in Table 4.

(5) **Verification** is the process of checking the conformance of the implementation to all the Codecharts and assignments. The process is invoked by clicking Verify All. If conformance is established, the message "PASSED" is presented next to the respective Codechart, otherwise the message FAILED is shown (Fig. 13, top). A dialogue box (not shown in Fig. 13) explains that the first assignment failed because the pair ⟨`BufferedInputStream.read`,`FilterInputStream.read`⟩ is not in the relation *Forward*. This allows programmers to revise the specification or the implementation to restore conformance.

(6) **Revision** in this example was made to the program, which is made to conform to the Decorator specification. The programmer can now define the method `BufferedIn-`

`putStream2.read()` that forwards the call to the method it overrides, and symmetrically with `LineNumberInputStream.read()`. The revised implementation is modeled in Codechart InputStream2.

(7) **Assignment** no. 2 (bottom left of Fig. 13) maps the variable *ConcreteDecorators* to the class of dimension 1 `InputStreamConcDecs2`, defined in the design model as the set

```
InputStreamConcDecs2
  = {BufferedInputStream2,LineNumberInputStream2}
```

(8) **Verification** of the revised program yields the message "PASSED", as shown next to the second assignment in Fig. 13, showing that the revised program indeed conform to the Decorator Codechart.

6.1 Automating pattern detection

Building a tool automating the process of pattern detection poses an interesting problem that is strictly more challenging than building a pattern verification tool. The reason is because the supporting tool must first search for a suitable set of candidate classes and methods in the implementation before attempting to verify them. Preliminary research into possible solutions to the problem of detection [58] indicates that the complexity of a detection algorithm need not exceed polynomial time in the number of methods in the implementation. An efficient implementation of such an algorithm however has not been fully integrated into the Toolkit at this time.

7 Round-trip engineering

We believe that an iterative, agile process of software evolution can greatly benefit from a *round-trip software engineering* tool. Codecharts were designed to enable tool support in such a process while overcoming the limitations of some such tools. Let us illustrate exactly what round-trip engineering stands for in this context and how it is supported.

Many software engineering tools seek to support the effort of maintaining consistency between design and implementation. Some tools attempt to do so by generating source code from specifications. However, machine-generated code is difficult to maintain manually, and any direct modifications may be lost the next time code is generated.

In contrast with code generation, round-trip software engineering tools and environments maintain design and implementation as separate representations while facilitating the propagation of changes between them. D'Hondt et al. [59] refer to this process as *software co-evolution*. Round-trip engineering tools usually support reverse-engineering by generating visualization from analyzed source code. Significantly, if the output of the reverse engineering tool (Class Diagrams, Statecharts, Codecharts, etc.) can feed directly into the forward engineering tool, and vice versa, then the engineering cycle can be said to be closed, hence *round-trip engineering*. Closing the engineering cycle can ensure that anytime a gap between design and implementation is generated, it can quickly be detected and remedied.

The Toolkit takes the two-tier programming approach to round-trip engineering. This means that any conventional programming technique may be used: we assume that source code can be authored in the native programming language (Java in this example) using any appropriate tool. The toolkit does not generate source code, nor does it extend the implementation language in any way. Separately maintained from the source code is the design specification, which consists of a collection of Codecharts. The Toolkit supports either manual (i.e. modeling) or automatic generation of Codecharts (i.e. visualization), as well as the means for detecting conflicts between the design and implementation (i.e. verification).

Furthermore, round-trip engineering means that Codecharts generated by the Toolkit by program visualization can be edited by the programmer, and the result can in turn be verified against the implementation. For example, each one of the Codecharts in Fig. 7 can be edited using the modeling tool, and in turn can be verified against the implementation as illustrated Sect. 6. It also means that the implementation can always be edited in the traditional way, re-analyzed, and re-verified, and consistency can be maintained by repeated executions of the design verification tool.

8 Pilot study

We sought to examine whether the use of Codecharts and tools supporting it can provide any productivity gains in practical settings. However, the Toolkit we developed is not a commercial product. It is a research prototype that has not (to our knowledge) been tested in an industrial setting. We therefore attempted to measure its benefits by conducting a small pilot study formed of three controlled experiments. These experiments aim to compare the performance of software engineers when using Codecharts and the Toolkit against their performance when using existing practices and common commercial tools in carrying out typical software comprehension and evolution tasks. As the results show, two of these experiments suggest gains both in speed (Sect. 8.1) and accuracy (Sect. 8.2).

In all experiments, participants in the experiment groups used version 0.5.2 of the Toolkit whereas those in the control groups used Sun's NetBeans 6.1, an industry-standard integrated development environment for Java programs, as well as Javadoc files. Participants were mostly graduate computer science students at the University of Essex who had no prior experience with the Toolkit. All participants were paid a fixed amount regardless of the time it took them to complete the tasks.

8.1 Gains in software comprehension

The first experiment tested the hypothesis that using Codecharts generated by the Toolkit's visualization tool increases the speed it took software engineers to demonstrate their understanding of a program by answering questions about it. All participants underwent one hour of training in using the Toolkit, balanced with one hour of training in using NetBeans and Javadoc for same tasks. After training, all participants received Java source code taken from the Standard Java Development Kit (the Java SDK) and were asked to answer some questions about them. Their understanding of the program was assessed by the time it took them to answer these questions accurately. Answers were checked for correctness by the administrators of the experiment; participants were notified of incorrect answers and could not continue until a correct answer was provided or the time allotted for the experiment expired.

The experiment itself was divided into two parts, each part consisting of a different task (of roughly same difficulty). Neither task affected the result of the other.

Table 5 Results of the comprehension experiment

Participant no.	Time in seconds	
	Toolkit	*NetBeans*
1	441	445
3	443	3106
4	595	690
6	420	1140
15	250	4260
18	705	753
22	274	3224
Mean	447	1945
Median	441	1140
Standard Deviation	162	1540
Ratio (Toolkit/NetBeans)	0.23	

The first part of the experiment itself began after all participants were randomly divided into two groups: Participants in the *experiment group* were asked to use the Toolkit in carrying out their task, whereas participants in the *control group* used NetBeans and Javadoc. All participants were handed source code files containing classes `Container` and `Component` from Java SDK. The task was to find four methods in class `Component`, each of which (i) calls another method in `Component` and (ii) returns an instance of class `Container`. The time to complete the task was measured by the administrators.

In the second part of the experiment, participants swapped groups and carried out a second, comparable task using a second set of tools. In other words, a participant who used Codecharts and the Toolkit in the first part of the experiment used NetBeans and Javadoc in the second and vice versa. All participants were handed source code files containing classes `InputStream` and `BufferedInputStream` from the Java SDK. The task was to find two methods in class `BufferedInputStream`, each of which forwards the call to a method in `InputStream`. The time to complete the task was measured by the administrators.

Of the ten original participants in this experiment, three were excluded from our analysis as they did not attend both sessions and therefore did not complete both tasks. The accuracy of the remaining seven participants was within acceptable tolerances (±10 %). In the analysis of our results (Table 5) we observed that those participants who used the Toolkit took 23 % of the time of those participants who used NetBeans/Javadoc. That is, in this experiment those who used the Toolkit were, on average, 77 % quicker to become familiar with the code provided after only one hour of training.

8.2 Gains in software conformance

The second experiment tested the hypothesis that using the Toolkit increases the *accuracy* of judgments that software engineers make about an implementation's conformance to design decisions, in particular to implement a design pattern. As in the experiment described before, participants in this experiment underwent the training relevant to the tasks performed, and randomly divided in two groups.

In the first part of this experiment all participants received four Java files from the Abstract Windowing Toolkit library and a copy of the chapter about the Composite design

Table 6 Results of the conformance experiment

Participant	Answered correctly?	
	Toolkit	*NetBeans*
3	Yes	No
4	Yes	Yes
6	Yes	Yes
8	Yes	No
9	Yes	Yes
13	Yes	No
18	Yes	Yes
Mean of accuracy	100 %	57.14 %
Median of accuracy	100 %	100 %
Ratio (Toolkit/NetBeans)	1.75	

pattern from [45]. Participants were asked to judge whether the implementation conforms to the pattern, where the correct answer was "Yes".

In the second part of this experiment all participants (who now switched groups) received six Java files from the `java.io` library and a copy of the chapter describing the Decorator design pattern from [45]. Participants were asked to judge whether the program conforms to the design pattern, where in this case the correct answer was "No". The data produced in this experiment are presented in Table 6.

Of the original eight participants one was excluded as s/he did not attend both sessions and therefore did not complete both tasks. In the analysis of our results (Table 6) we observed that those participants who used the Toolkit gave the correct answer 1.75 times more often than those who used NetBeans/Javadoc after only one hour of training.

8.3 Threats to validity

Although positive results were obtained from this pilot study there are several considerations regarding how they may be interpreted, such as:

– Our sample was not sufficiently representative of software engineers in industry due to the small number of participants and their limited background (i.e. students). We plan to hold more large-scale experiments in future work with the aim to show similar results in a more representative sample.
– We compared the Toolkit to a single tool out of many possible candidates. We are therefore unable to make claims regarding the Toolkit beyond that comparison. Further investigation against a range of tools is required to see if the Toolkit yields similar results.
– During the second experiment we observed that while the Toolkit helped participants to answer the question correctly it did not speed up their work as in the first experiment. We suggest that this can be attributed to the lack of feedback, showing if their answer is right or wrong, which may have made participants linger in answering. This phenomena, however, requires further study.
– We also observe that the method of measuring accuracy in the second experiment was a Yes/No response, which left little room for analysis. Tasks of the form "identify as many components of program *p* that implement a participant in design pattern *d*" would provide greater insight into the participant's precision and accuracy.

– We omitted from this report a third experiment, which measured the software engineers' performance in conducting a trivial software evolution task. Its results turned out to be inconclusive. Participants in both groups completed their tasks unexpectedly quickly and no significant difference between the tools was recorded. We believe that our experimental design was flawed. We plan to prepare a more carefully designed version of this experiment in the future.

Accounting for the above limitations, the pilot study is nonetheless encouraging. Software comprehension and conformance are key tasks in the design, implementation, evolution, and re-engineering phases of the software engineering lifecycle. Understanding the structure and organization of a software system is a prerequisite to any development, maintenance, or evolution activity. This task becomes more important as the age and size of the code base increases. Being certain that a program conforms to its intended design is invaluable for all stakeholders in a program's development. This results suggest that using Codecharts and the Toolkit could play an important role in helping to make software engineers more productive and accurate in such tasks.

9 Summary

Motivated by the difficulties of software engineering, by practical concerns of continuously evolving software, and by the challenges arising from the gap between theory and practice, we presented the modeling and visualization language of Codecharts. We identified its scope by posing five central questions. To answer the Ontological Question, we demonstrated how Codecharts model the organization of class libraries and design patterns in terms of the basic building-blocks of OO design, such as sets of dynamically-bound methods, hierarchies, and their relationships. We concluded that our minimal ontology limits the scope of Codecharts, but also that it may confer clarity and conceptual elegance on the notation. In response to the Visualization Question we showed Codecharts modeling the Decorator design pattern, and demonstrated how a design recovery tool can create Codecharts that visualize an existing program by analyzing its source code. To answer the Scaling Question we illustrated how Codecharts can visualize programs of arbitrary size and level of abstraction. To answer the Conformance Question we sketched the process of design verification seeking to check whether a Java package implements a design pattern correctly and showed that it does not satisfy the Codechart modeling it. Finally, to answer the Automation Question we illustrated how the Toolkit automates design verification.

We also discussed how Codecharts enable maintaining consistency between design and implementation by facilitating the propagation of changes from design to implementation and vice versa. We concluded with the hypothesis that tools supporting Codecharts can be effective in helping to prevent architectural drift and erosion, by ensuring that the design of a program is current, correct and accurate throughout the process of software co-evolution.

The results of a pilot study suggest that stakeholders in the development and evolution of OO software could benefit from Codecharts in forward, reverse, round-trip and re-engineering tasks without requiring special training or long-term commitment to the notation, programming languages, or tools.

10 Future work

The work described here has a number of natural extensions, such as:

– Gather further empirical evidence of potential productivity gains in an industrial setting.
– Investigate the potential of using automated design verification in a version control system, which can ensure that conformance to specifications is enforced with every revision checked in.
– Examine the potential of using Codecharts and supporting tools in calculating software metrics, and how this might be presented to the user in an elegant and intuitive fashion.

To encourage exploration and improvement of Codecharts, a version of the Toolkit was made available to download from http://ttp.essex.ac.uk under a Creative Commons license [60].

Acknowledgements The authors wish to thank Raymond Turner for his support throughout this research, which was funded in part by the Research Promotion Fund and the Knowledge Transfer Innovation Fund from the University of Essex, and by the EPSRC. We also wish to thank Olumide Iyaniwura, Gu Bo, Maple Tao Liang, Dimitrious Fragkos, Omololu Ayodeji, Xu Yi and Christina Maniati for their contributions.

References

1. Brooks FP (1987) No silver bullet: essence and accidents of software engineering. IEEE Comput Mag 20(4):10–19
2. Lehman MM (1996) Laws of software evolution revisited. In: Proc 5th European workshop software process technology—EWSPT'96, Nancy, France
3. Grant M, Goguen JA (1996) An executable course in the algebraic semantics of imperative programs. In: Dean NC, Hinchey MG (eds) Teaching and learning formal methods. Morgan Kaufmann, San Mateo, pp 161–179
4. Eden AH, Gasparis E, Nicholson J (2007) LePUS3 and Class-Z reference manual. Department of Computer Science, University of Essex, CSM-474, ISSN 1744-8050, Dec
5. Eden AH, Gasparis E, Nicholson J (2007) The 'Gang of Four' companion: formal specification of design patterns in LePUS3 and class-Z. Department of Computer Science, University of Essex, CSM-472, Dec
6. Eden AH, Gasparis E (2009) Three controlled experiments in software engineering with the two-tier programming toolkit: final report. University of Essex, CES-496, ISSN 1744-8050
7. Nicholson J, Gasparis E, Eden AH LePUS3 and class-Z home page [Online]. Available: http://www.lepus.org.uk/. Accessed: 27 Aug 2010
8. Nicholson J, Gasparis E, Eden AH The two-tier programming project website [Online]. Available: http://ttp.essex.ac.uk/. Accessed: 14 Apr 2008
9. Gasparis E, Eden AH, Nicholson J, Kazman R (2008) The design navigator: charting Java programs. In: Tool demonstrations, proc of 30th IEEE int'l conf on software engineering—ICSE 2008, Leipzig, Germany
10. Gasparis E, Nicholson J, Eden AH (2008) LePUS3: an object-oriented design description language. In: Proc 5th int'l conf diagrammatic representation & inference, Herrsching, Germany, vol 5223
11. Gasparis E, Nicholson J, Eden AH, Kazman R (2008) Navigating through the design of object-oriented programs. In: Proc of the 15th working conf on reverse engineering—WCRE, Antwerp, Belgium
12. Eden AH, Nicholson J (2011) Codecharts: roadmaps and blueprints for object-oriented programs. Wiley-Blackwell, New York
13. Medvidovic N, Taylor RN (2000) A classification and comparison framework for software architecture description languages. IEEE Trans Softw Eng 26(1):70–93
14. Taibi T (ed) (2007) Design patterns formalization techniques. IGI Global, Hershey
15. Demeyer S, Ducasse S, Tichelaar S, Tichelaar E (1999) Why unified is not universal: UML shortcomings for coping with round-trip engineering. In: Proc 2nd int'l conf on the unified modeling language, vol 1723, pp 630–645
16. Fowler M (2004) UML distilled: a brief guide to the standard object modeling language, 3rd edn. Addison-Wesley, Boston
17. Harel D (1987) Statecharts: a visual formalism for complex systems. Sci Comput Program 8(3):231–274

18. Guttag JV, Horning JJ, Wing J (1982) Some notes on putting formal specifications to productive use. Sci Comput Program 2(1):53–68
19. Guttag JV, Horning JJ (1993) Larch: languages and tools for formal specification. Springer, New York
20. Pnueli A (1986) Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In: Bakker JW (ed) Current trends in concurrency. Overviews and tutorials. Springer, New York, pp 510–584
21. Jones CB (1990) Systematic software development using VDM, 2nd edn. Prentice Hall International, New York
22. Peterson J (1981) Petri net theory and the modeling of systems. Prentice-Hall, Englewood Cliffs
23. Hoare CAR (1978) Communicating sequential processes. Commun ACM 21(8):666–677
24. Jackson D (2002) Alloy: a lightweight object modeling notation. ACM Trans Softw Eng Methodol 11(2):256–290
25. Abrial J-R (1996) The B-book: assigning programs to meanings. Cambridge University Press, Cambridge
26. Derrick J, Boiten E (2001) Refinement in Z and object-Z: foundations and advanced applications. Springer, Berlin
27. France RB, Kim D-K, Ghosh S, Song E (2004) A UML-based pattern specification technique. IEEE Trans Softw Eng 30(3):193–206
28. Kim D-K (2004) A meta-modeling approach to specifying patterns. PhD Dissertation, Colorado State University, Fort Collins, CO, USA
29. Kent S (1997) Constraint diagrams: visualizing invariants in object-oriented models. In: ACM SIGPLAN notices, New York, NY, USA, pp 327–341
30. Howse J, Molina F, Taylor J, Kent S, Gil J (2001) Spider diagrams: a diagrammatic reasoning system. J Vis Lang Comput 12(3):299–324
31. Object Management Group (2005) UML 2.0 superstructure specification, Aug 2005
32. Tufte E (1997) Visual explanations: images and quantities, evidence and narrative. Graphics Press, Cheshire
33. Maplesden D, Hosking J, Grundy J (2007) A visual language for design pattern modeling and instantiation. In: Taibi T (ed) Design patterns formalization techniques. IGI Global, Hershey
34. Guéhéneuc Y-G, Antoniol G (2008) DeMIMA: a multilayered approach for design pattern identification. IEEE Trans Softw Eng 34(5):667–684
35. Blewitt A, Bundy A, Stark I (2001) Automatic verification of Java design patterns. In: Proceedings of the 16th IEEE international conference on automated software engineering, pp 324–333
36. Gosling J, Joy B, Steele G, Bracha G (2005) The Java language specification, 3rd edn. Addison-Wesley Professional, Reading
37. Smith B (2004) Ontology. In: Floridi L (ed) The Blackwell guide to the philosophy of computing and information. Blackwell Publishers, Malden
38. Nicholson J (2011) On the theoretical foundations of LePUS3 and its application to object-oriented design verification. PhD Dissertation, School of Computer Science and Electronic Engineering, University of Essex
39. Lieberman BA (2006) The art of software modeling, annotated edn. Auerbach Publications, Boca Raton
40. Hoare CAR (1975) Software design: a parable. Softw World 5(9–10):53–56
41. Walsh AE, Gehringer D (2002) Java 3D: API jump-start. Prentice Hall, Upper Saddle River
42. Selman D (2002) Java 3D programming. Manning Publications, Greenwich
43. Cai Y, Ianuzzi D, Wong S (2011) Leveraging design structure matrices in software design education. Presented at the conf on software engineering education and training—CSEET 2011, Honolulu, HI
44. Nicholson J, Eden AH, Gasparis E (2007) Verification of LePUS3/class-Z specifications: sample models and abstract semantics for Java 1.4. Department of Computer Science, University of Essex, Technical Report CSM-471, Dec 2007
45. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison Wesley Longman, Reading
46. Chikofsky EJ, Cross JH (1990) Reverse engineering and design recovery: a taxonomy. IEEE Softw 7(1):13–17
47. Gasparis E (2010) Design navigation: recovering design charts from object-oriented programs. PhD Dissertation, School of Computer Science and Electronic Engineering, University of Essex
48. Shermer M (2005) The Feynman-Tufte principle. Sci Am 292(4):38
49. Nicholson J, Gasparis E, Eden AH, Kazman R (2009) Automated verification of design patterns with LePUS3. In: Proc 1st NASA formal methods symp—NFM 2009, Moffett Field, CA
50. Eckel B (2003) Thinking in Java. Prentice Hall Professional, Upper Saddle River
51. Wing JM (1990) A specifier's introduction to formal methods. Computer 23(9):8–23
52. Hoare CAR (1969) An axiomatic basis for computer programming. Commun ACM 12(10):576–580

53. Mahoney MS (2002) Software as science: science as software. In: Proc int'l conf history of computing: software issues, Paderborn, Germany, pp 25–48
54. Clarke EM, Wing JM (1996) Formal methods: state of the art and future directions. ACM Comput Surv 28(4):626–643
55. Parnas DL (1994) Software aging. In: Proc 16th int'l conf software engineering, pp 279–287
56. Mens T, Demeyer S (2008) Software evolution. Springer, Berlin
57. Perry DE, Wolf AL (1992) Foundations for the study of software architecture. SIGSOFT Softw Eng Notes 17(4):40–52
58. Salazar Saltijeral J (2012) Design pattern detection in Java. MSc dissertation, School of Computer Science and Electronic Engineering, University of Essex
59. D'Hondt T, De Volder K, Mens K, De K, Kim V, Wuyts R (2000) Co-evolution of object-oriented software design and implementation. In: Proc int'l symposium on software architectures and component technology— SACT, Amsterdam, The Netherlands
60. Commons C (2001) Creative commons—attribution-NoDerivs 2.0 UK: England & Wales [Online]. Available: http://creativecommons.org/licenses/by-nd/2.0/uk/. Accessed: 20 Apr 2011