

# TWO-TIER PROGRAMMING

Technical Report CSM-387, Dept. of Computer Science, University of Essex

A version of this article was submitted to the Journal of Automated Software Engineering – ASE (Amsterdam, The Netherlands: Kluwer Academic Publishers)

Amnon H. Eden

*Dept. of Computer Science,  
University of Essex, United  
Kingdom, and Center for  
Inquiry, Amherst, NY*

eden@acm.org

Rick Kazman

*Software Engineering  
Institute, Pittsburgh, PA and  
University of Hawaii,  
Honolulu, HI*

kazman@sei.cmu.edu

Chris Fox

*Dept. of Computer Science,  
University of Essex,  
United Kingdom*

foxcj@essex.ac.uk

## Abstract

In practice, the development and maintenance (if any) of a system's *design* specifications are carried out in veritable isolation from the development and maintenance of the *implementation*. Design decisions are represented independently from the implementation and its analysis and verification are carried out (if at all) by a separate tool set. The dissociation between the specification layers produces *architectural drift* and *architectural erosion*, leading causes for the incomprehensibility and unmaintainability of aging programs.

We present *two-tier programming*, a conceptual framework for integrated representation of programs in two layers of specification: (1) architecture (“second-order programs”) and (2) traditional implementation (“first-order programs.”) We review two-tier programs, environments and tools, and analyse the two-tier approach to programming.

Keywords: Software design theory, software architecture, architectural erosion/architectural drift, traceability, software maintenance and evolution, program comprehension.

# 1. Introduction

*Software is a child of the latter half of twentieth century – a baby boomer. And like its human counterpart, software has accomplished much while at the same time leaving much to be accomplished.* [Pressman 97]

The notion of a *programming language* has progressively evolved from the mere manifestation of an algorithm in a machine language towards supporting increasingly more abstract constructs, such as *loops*, *procedures*, *modules*, and *objects* [Madsen 00]. The rapid pace of changes since the 1950s has been influenced by frequent technological developments, including those in hardware, development environments, and design methods. More recently, the progress in programming languages have been influenced by software engineering concerns such as “maintainability” and “flexibility”.

The formal notion of *computation* has also undergone significant changes since Turing has introduced his automat [Turing 36]. In addition to RAM and Abstract State Machines, which simulate rudimentary steps of computation, more abstract models of computation were introduced, such as Petri nets [Petri 62], process algebras [Hoare 85], statecharts [Harel 87], and temporal logic [Lamport 94]. Formal specification languages, such as  $\mathbb{Z}$  [Spivey 89] and Larch [Guttag & Horning 93], were designed to support the top-down process of development, which begins with analyzing functional requirements.

*Architectural specification languages* (ADLs) such as WRIGHT [Allen & Garlan 97] and MetaH [Binns et. al 94] were also designed to support a top-down process. Unlike other specification languages, however, these languages were designed to support the analysis of architectural abstractions, which are non-functional requirements. Similarly, formal languages for the support of specification and verification of object-oriented design, such as *Constraint Diagrams* [Lauder & Kent 98], *DisCo* [Mikkonen 98] and LEPUS [Eden 01] were proposed. Nonetheless, the verification of architectural specification has remained an open problem.

## 1.1 Problems with evolving programs

By the “first law of software evolution” [Lehman 80; Lehman 97], *the law of continuing change*, software systems operate in an environment that is in a state of constant flux. There is ample evidence [Boehm 88; Pressman 97; Comer 97] that software development progresses in an open-ended, step-wise process of adaptation to a stream of changes, not unlike *evolution* in a natural species. For example, the manufacturers of popular shrink-wrapped software packages (such as word processors and software development environments) attempt to adapt to new requirements, such as changes in the operating system, better hardware, and releases by the competitors, by distributing a seemingly endless sequence of new releases, each adding a set of new features that may not have been envisioned when the products were conceived.

Despite the progress made since first observing the “software crisis”, research in software engineering has established that the contemporary software industry continues to undergo a “chronic crisis” [Brooks 87; Gibbs 94]. Specifically, programs are commonly diagnosed as too “brittle” and “inflexible” [Pressman 97], thereby aging quickly and becoming too expensive to maintain; that software products become obsolete almost as soon as they are deployed; and that the costs of large projects continue to exceed their initial budget [Sommerville 00]. But the problems are most acute with “aging” programs. In particular, long-lived programs suffer the effects of *architectural drift* and *architectural erosion* as a result of continually attempting to adapt and evolve.

According to Perry and Wolf [92], brittleness and inflexibility are the result of a growing dissociation of architectural specifications from the implementation. Programmers commonly fail to bring the documentation or “design specifications” up to date. As a result, the documentation become progressively inconsistent with the implementation until the design specifications can no longer serve as a reliable model for the program. Each change in the dissociated implementation becomes more difficult to accomplish. Some systems are so difficult to maintain that they win the title “legacy system”, i.e., ten years or older systems “for which current maintenance is very expensive, and one for which integration with current or modern technology or software systems is difficult or impossible” [SEI 95]. The comprehension and modelling of legacy systems require expensive techniques whose purpose is to restore some of the design specifications, such as “reverse engineering” or even “software archeology” [Cunningham et. al 01]. But the source code of programs written in an industrial programming language incorporate only partial information about their design, and a coherent architecture or design model of a program cannot possibly be produced simply by inspecting the implementation (e.g., [Kazman & Carriere 99; Keller et. al 99].)

Another question occurring in the documentation of large programs is *traceability* [Perry & Wolf 92], namely, how to associate each design decision with its respective implementation. Commenting incorporated in the code only help us understand the design decisions related to a given locale in the implementation but not vice versa. Worse, comments embedded in the program (and documentation tools based thereon, such as `Javadoc`) only offer a weak solution, as the verification of informal text requires a laborious manual process.

We conclude that existing software development strategies yield programs that fail to meet the dynamics of requirements as reality dictates and therefore are expected to become even more unsuitable. Thus, a dramatic diversion from contemporary development practice appears to be in place.

## 1.2 Discussion: Solutions

Is it advisable (or even possible) to address the problems of architectural erosion and traceability from the programming language perspective? By such an approach, one may propose to design a more abstract futuristic programming language that directly supports useful design and architectural constructs, not unlike the way Lisp has been introduced to support *functions* and *lists*, and Simula to support the notion of a *class* and *con-*

*catenation* (today's *inheritance*). More specifically, we need “better” programming languages which support the specification of design patterns [Gamma et. al] and architectural styles [Garlan & Shaw 93].

In support of this view, consider for example how Smalltalk supports the *Model-View-Controller* “pattern”: Smalltalk facilitates the distribution of events between *controllers*, *models* and *views* by defining the behaviour of a *model* in the universal superclass `Object`. By the properties of the Smalltalk language, every (<sup>1</sup>) instance (object) inherits these capabilities and can therefore assume the role of a model with relation any number of potential *views*. The communication protocol between models and their views is provided in a similar fashion.

There are, however, indications that this direction is implausible. First, the number of possible design abstractions is very large and probably unbounded (<sup>2</sup>). The proliferation of design patterns since 1993 (see, for instance, [Rising 00]) demonstrates how many of them were deemed sufficiently interesting to be included in the catalogue. Thus, any fixed programming language offering a specific collection of general-purpose abstractions can only offer solutions to a relatively small category of problems (although see [Madsen 00]). The maintenance of all other abstractions (which were not lucky enough to be supported by this programming language) will continue to suffer from the problems of traceability and architectural drift. And even if someone has managed to devise such a programming language which supports the enormous set of all useful architectural abstractions, we believe that design abstractions will continue to find their way into practice for many years to come.

In conclusion, the solution to the problems outlined above does not depend on the programming language in use.

### 1.3 The *two-tier programming* solution

A more likely solution to the problems of architectural drift/erosion and traceability is offered by the *two-tier programming* (TTP) programming paradigm. This approach to programming redefines the very notion of a computer program by assuming an *integrated* representation of design specifications with the implementation.

---

<sup>1</sup> Exceptions to this rule are rare and not of concern to this discussion.

<sup>2</sup> It is easy to show that text-based specification languages define at most an enumerable set of possible specifications.

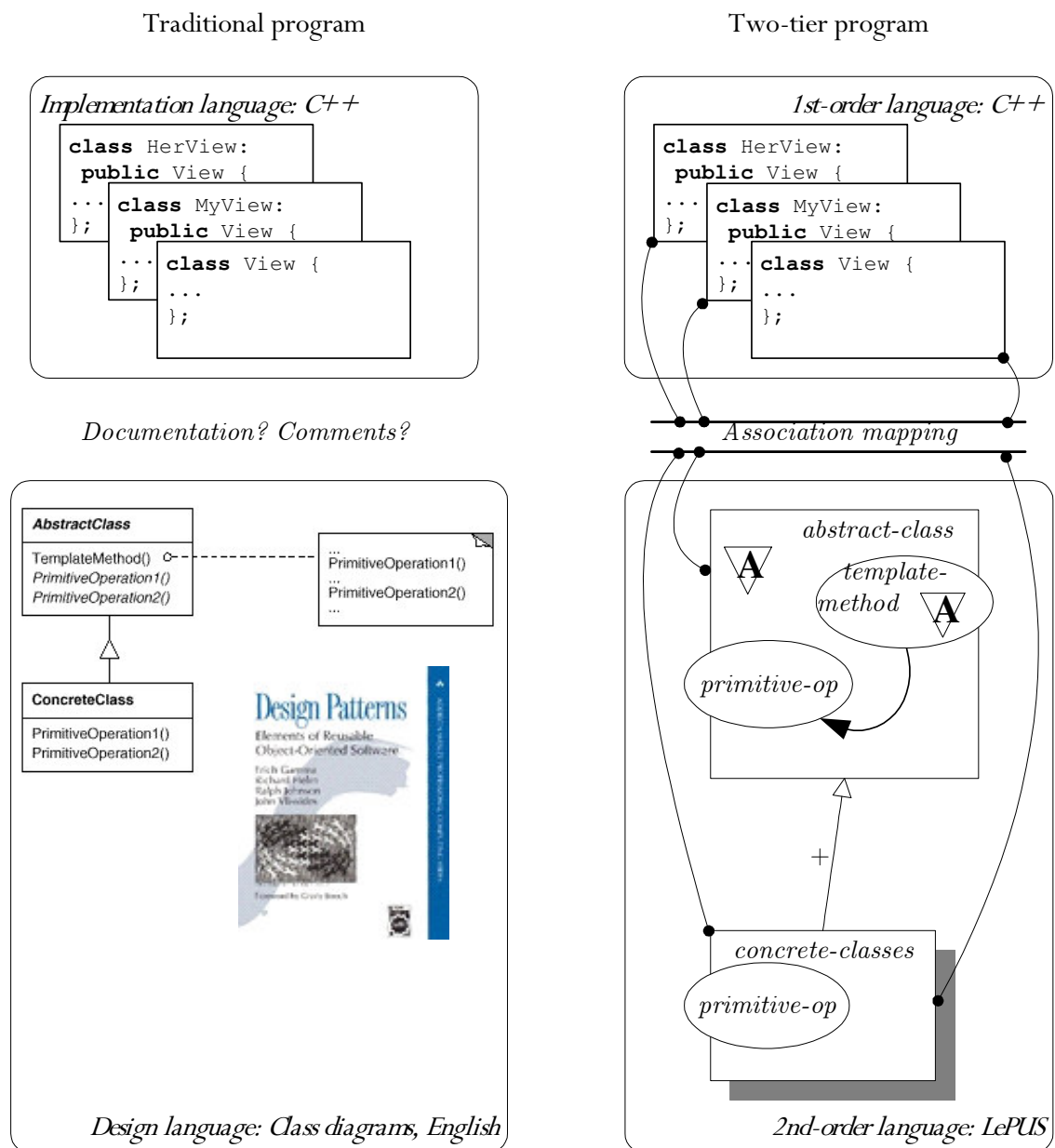


Figure 1. Traditional *vis-à-vis* two-tier program

A two-tier program is defined in two levels of abstraction, each incorporating statements in different languages. Statements in the lower level are made in a traditional programming language (a “first-order” programming language), which delivers a complete and detailed specification of the implementation. The second layer consists of design constraints specified in an appropriate architectural specification language (a “second-order” programming language). Figure 1 compares the configuration of a two-tier program with that of a traditional program (“single-tier”) and illustrates the differences between them.

## 1.4 Intended contributions

This article presents *two-tier programming*, a programming paradigm which addresses problems in program comprehension and maintenance in contemporary practice. In particular,

- ♦ We offer a well-defined ontology for the discussion in two-tier programs.
- ♦ We demonstrate a sample two-tier program and discuss in detail a case of a two-tier programming language.
- ♦ We explain how TTP addresses the problems of *architectural erosion/drift* and provides the ultimate solution to *program comprehension* (specifically, to *traceability*).

## 2. A sample two-tier program

In this section, we illustrate *two-tier programs* by analysing a simple example. This two-tier program will incorporate (a) a representation of the desired design, the “Template Method” design pattern [Gamma et. al 94]; (b) an implementation of the pattern; and (c) a mapping relation which indicates the linkage between the design and the implementation.

To remind the reader, the intent of the Template Method is to “Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.” [*ibid.* p. 175] The OMT diagram in the pattern’s manuscript is depicted in Figure 2.

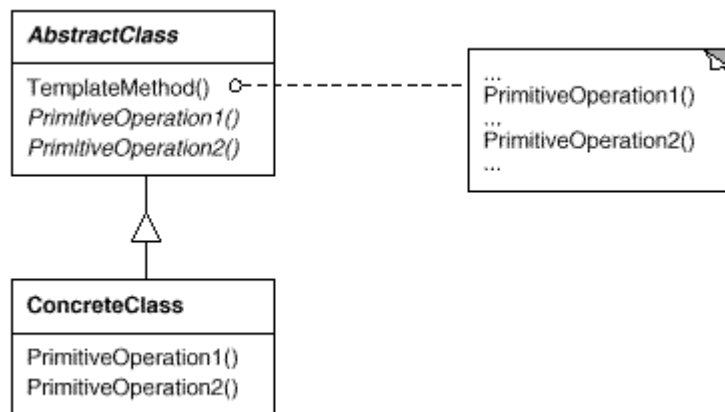


Figure 2. OMT diagram of the Template Method pattern (adapted from [Gamma et. al 94]).

## 2.1 The first-order tier

A rudimentary implementation of the Template Method in C++ is depicted in Table 1. The example depicted expands on the sample implementation given in the same manuscript.

Table 1. A C++ implementation of the Template Method (also adapted from [Gamma et. al [94])

```
struct View {
    void Display () {
        SetFocus(0);
        // do something else
    }
    // Primitive-op is not defined in this class:
    virtual void SetFocus(int loc) = 0;
};

struct MyView: public View {
    virtual void SetFocus(int loc) {
        // define SetFocus in MyView
    }
};

struct HerView: public View {
    virtual void SetFocus(int loc) {
        // define SetFocus in HerView
    }
};
```

## 2.2 The second-order tier

The design of our toy program is specified by the Template Method. The pattern was not defined formally by Gamma et. al [94], but an accurate description is facilitated by sample implementations and class diagrams. The clarity of the informal specification has allowed us to render it formal using LEPUS [Eden 00], a higher-order logic language. Figure 3 depicts the specification of the Template Method (<sup>3</sup>) in the visual version of LEPUS [Eden 02a]. The diagram is followed by same specification in the symbolic version of the language.

---

<sup>3</sup> For the purpose of our discussion, we specify a simplified version of the pattern, which consists of only one “primitive operation”.

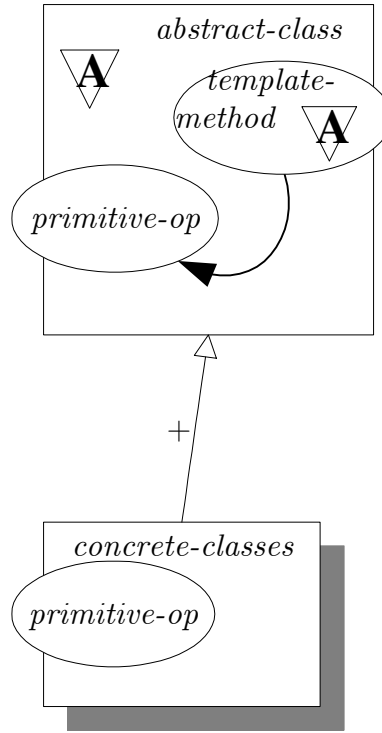


Figure 3. The *Template Method* (simplified version of [Gamma et. al 94]) in visual LEPUS

The symbolic representation of Figure 3 consists of five clauses as follows:

$$abstract-class : \mathbb{C} \quad (1.1)$$

$$concrete-classes : \mathbf{P}(\mathbb{C}) \quad (1.2)$$

$$template-method, primitive-op : \mathbb{S} \quad (1.3)$$

$$Abstract(abstract-class) \quad (1.4)$$

$$Invoke(template-method \otimes abstract-class, primitive-op \otimes abstract-class) \quad (1.5)$$

$$Inherit^+ \rightarrow (concrete-classes, abstract-class) \quad (1.6)$$

Let us explain each clause in the specification of the *Template Method* in expression (1). The first three clauses in the specification declare four variables as follows:

(1.1) declares a variable called *abstract-class* which ranges over entities of type “class”;

(1.2) declares a variable that ranges over entities of type “a set of classes”;

(1.3) declares two variables of type “signature”. A “signature” is an abstraction of a method’s declaration. Methods are declared indirectly by combining a signature and a class declaration. For example, the expression *template-method*⊗*abstract-class* declares a method with the signature *template-method* in class *abstract-class*. Signatures are used in clause (1.5), explained below.

The three clauses that follow impose constraints on the variables declared:

- (1.4) states that class *abstract-class* is *Abstract*;
- (1.5) states that a method with signature *template-method* is defined in class *abstract-class* and that it “invokes” (i.e., its body contains a function call to) a method with signature *primitive-op* which is defined in the same class;
- (1.6) the clause incorporates syntactic sugars defined in LEPUS which state that all classes included in *concrete-classes* ‘inherit’ (possibly indirectly) from *abstract-class*.

## 2.3 The association mapping

Although the C++ program in Table 1 appears to satisfy the Template Method, it is preferable to have an explicit representation for the linkage between the specification and the respective implementations. This explicit representation allows for automated verification and it addresses the traceability problem.

In our example, the mapping associates variables in the specification, the Template Method, with elements of the implementation. This association is depicted in Table 2.

Table 2. The associations between *Template Method* and elements of its implementation

Variables	ground entities
<i>abstract-class</i>	View
<i>concrete-classes</i>	{MyView, HerView}
<i>template-method</i>	Display(void)
<i>primitive-op</i>	SetFocus(int)

## 2.4 Summary

To summarize, the two-tier program illustrated in this section comprises the following:

- ♦ The *1st-order tier* has the C++ implementation of the *Template Method* (Table 1);
- ♦ The *2nd-order tier* has the specification of the *Template Method* (expression (1));
- ♦ The *association mapping* maps the free variables in the design specification into constants in the implementation (Table 2).

### 3. Two-tier programming languages

Elkana [76] uses the term *two-tier thinking* in his discussion in the apparent dichotomy between first- and second-order thinking. According to Elkana, *first-order thinking* designates a reasoning process that is focused on “concrete” thoughts and which is carried out within a specific context, while second order thinking characterises thinking *about* thoughts and about the choice of a first-order context. *Second-order thinking* takes place in a different “tier” of thinking from first-order thinking. Thus, *two-tier thinking* is a thinking process which may take place in either tier, and allows one to use both first- and second-order thinking, and switch from one to the other as appropriate. This terminology has inspired our use of *two-tier programming* and the analogous distinction between first-order and second-order programming.

#### 3.1 1st-order programming languages

Broadly speaking, by first-order programming language we refer to any 3rd-generation, or any general purpose programming language that is sufficiently expressive to support completely detailed implementations. These include familiar programming languages, such as C++, Java™, Pascal and Smalltalk, but they also include certain special-purpose languages. For example, Binns et. al [94] describe an architectural description language (ADL) which uses a domain-specific language, *ControlH*, as the implementation language.

Respectively, an expression  $p$  in a first-order programming language  $L_1$  is the traditional notion of a program. Thus, if  $C++$  is the symbol that represents the set of all well-formed programs in C++, an element of this set  $p \in C++$  is a well-formed program such as the one depicted in Table 1.

#### 3.2 2nd-order programming languages

In contrast with 1st-order programming languages, there is a broad category of languages that can be used to specify constraints on implementations. These languages are considered, in analogy, as second-order programming languages, or metalanguages for implementation languages. Statements in 2nd-order programming languages specify constraints on programs by expressing their desired (or undesired) properties. For example, the expression

$$\forall c \in \mathbb{C} \bullet \text{Inherit}^*(c, \text{Object}) \tag{2}$$

does not define a specific implementation but imposes the constraint that all classes must inherit (possibly indirectly) from class `Object`. Specifically, expression (2) incorporates the second-order variable  $c$  which ranges over  $\mathbb{C}$  the set of all classes, the constant `Object`, and the symbol  $\mathcal{R}^*$  which designates the transitive closure (zero or more) of the relation  $\mathcal{R}$ .

Expressions in specification languages such as *Constraint Diagrams* [Lauder & Kent 98], *DisCo* [Mikkonen 98] and LEPUS [Eden 00; Eden 01] consist of variables that

range over concrete entities of the first-order language, such as classes, routines. In contrast, WRIGHT [Allen & Garlan 97], an architectural description language (ADL), was designed to support statements on the overall configuration of large systems. WRIGHT is an extension to *communicating sequential processes* [Hoare 85] and it can be used to define architectural styles. Consider for example the specification of the *Client-Server* “connector type” in WRIGHT:

$$\begin{aligned}
 \text{connector C-S-connector} &= & (3) \\
 \text{role Client} &= (\text{request!x} \rightarrow \text{result?y} \rightarrow \text{Client}) \sqcap \S \\
 \text{role Server} &= (\text{invoke?x} \rightarrow \text{return!y} \rightarrow \text{Server}) \sqcup \S \\
 \text{glue} &= (\text{Client.request?x} \rightarrow \text{Service.invoke!x} \rightarrow \\
 &\quad \text{Service.return?y} \rightarrow \text{Client.result!y} \rightarrow \text{glue}) \sqcup \S
 \end{aligned}$$

Expression (3) is part of the specification of a simple “client-server” style, consisting of “components” (processes) of types Client and Server, where the connector statement indicates how the specific protocol of their communication.

The design decisions made by the authors of WRIGHT focuses on specifications of behavioural (i.e., dynamic) properties and to the language of process algebra (i.e., *events* and *processes*). Allen and Garlan demonstrate its advantages in reasoning on specifications and even proving useful properties, such as compatibility between systems and deadlock freedom.

Dean and Cordy [95] present yet another kind of a 2nd-order programming language, a visual formalism for the specification of architectural components and their relations. They demonstrate this formalism by the specification of the Pipes and Filters architectural style, depicted in Figure 4:

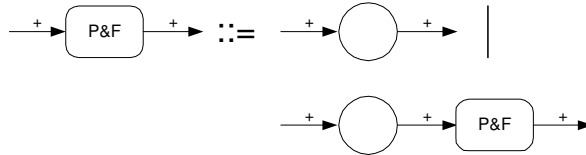


Figure 4. Pipes and Filters (adapted from [Dean & Cordy 95]).

The language of the visual expression in Figure 4 is defined in a BNF-like definition of a context-free grammar. A circle in the visual language represents a “task”, arrows represent streams. The plus sign is the BNF symbol for “one or more.” Thus, Figure 4 reads: A *pipes-and-filters* implementation must be configured such that it consists of a chain of “task” entities such that each task is connected to the next task in the sequence via a “stream” entity.

On the other end of abstraction level we also consider Smalltalk as a second-order specification language, which can be used for the specification of properties of Smalltalk programs. Smalltalk-80 [Goldberg & Robson 89] incorporates a metaclasses hierarchy, which also support metaprogramming tasks (also known as *reflection*). Executing the

following Smalltalk expression, for example, will define a new class named `newClass` with one instance variable:

```
Object subclass: 'newClass' (4)
  instanceVariableNames: 'v'.
```

Smalltalk is distinguished from the other second-order programming languages considered in this section in offering a different level of complexity. Observe that while the 2nd-order expressions demonstrated hitherto were all declarative, a Smalltalk expression has operational semantics.

### 3.3 Two-tier programming languages

We conclude this discussion in formalising the most general notion of a two-tier programming language (TTPL). We base our discussion on the terminology introduced by Guttag, Horning and Wing [82] for the discussion in formal specification of programs. Guttag et. al define *specification language* as follows:

**Definition I.** A specification language is a triple  $\langle L_2, L_1, Sat \rangle$  where  $L_1$  and  $L_2$  are sets and  $Sat \subseteq L_1 \times L_2$  is a relation between them.  $L_2$  is the language's syntactic domain,  $L_1$  is the semantic domain, and  $Sat$  is its *satisfies* relation.

We define a two-tier programming language (TTPL) as a *specification language*. For a given TTPL  $\langle L_2, L_1, Sat \rangle$ , we apply the following terminology:

1.  $L_1$  is a *1st-order programming language*, in which we specify the implementation.
2.  $L_2$  is a *2nd-order programming language*, in which we specify the design.
3.  $Sat$  is a *Satisfies* relation, which indicates for any two programs  $p \in L_1$  and  $P \in L_2$ , whether  $p$  satisfies  $P$ . (<sup>4</sup>)

We further illustrate the notion of a TTPL in Section 5 with the analysis of a specific TTPL.

## 4. Two-tier programs

In this section, we elaborate on the notion of a two-tier program. Given a TTPL  $\lambda = \langle L_2, L_1, Sat \rangle$ , a *two-tier program* consists of two tiers of representations and an association mapping as follows:

---

<sup>4</sup> Guttag et. al imply that the *Satisfies* relation is binary, i.e.,  $(p, P) \in Sat$  means that  $p$  satisfies  $P$ . In Section 5, we demonstrate a ternary *Satisfies* relation. In the general case, we say that *Satisfies* is a ternary relation, such that  $(p, P, c) \in Sat$  iff  $p$  satisfies  $P$  in some context  $c$ , where *context* is relevant as determined by the TTPL.

- ♦ The *1st-order tier* consists of the 1st-order program  $p \in L_1$ . A 1st-order program may only be the implementation (e.g., represented as the source code of a traditional program).
- ♦ The *2nd-order tier* consists of statements in the second-order programming language,  $L_2$ , also the *2nd-order program*. The statements could be general, such as the Template Method design pattern or the Pipes and Filters architectural style, but the design specifications could also be particular to this program (e.g., described using expressions that contain only constants and no variables.)
- ♦ The *association mapping* <sup>(5)</sup> exists to maintain consistency between the two tiers.

Observe that, by this definition, the IBM *Rational Rose*<sup>®</sup> “round-trip engineering” suit and the `Javadoc` documentation utility do not support two-tier representations. The reason is that both tools store some of the design information as part of the implementation, and thereby violate the principle of separating the specification tiers.

The remainder of this subsection is dedicated to a discussion in the *abstract interpretation* of implementations and on the purpose and definition of the association mapping.

## 4.1 Abstract interpretation

As demonstrated by the sample 2nd-order expressions in Section 3, 2nd-order programming languages rarely speak in terms of the implementation. Most modelling formalisms, including Petri nets [Petri 62], process algebras [Hoare 85], statecharts [Harel 87], temporal logic [Lamport 94], and higher-order logic [Barwise 74], support statements in terms of abstractions of implementations (e.g., processes, states, and class hierarchies) rather than in terms of concrete constructs of the implementation language (e.g., classes and variables.)

This should not be surprising: The elements in a design specification are expected to be more abstract than the elements of the implementation. This gap, however, raises the following problem: How can a specification be verified, if not directly by the source-code, or for that matter, by any other representation of the implementation? The solution that TTP offers (the *association mapping*) relies on the formal notion of *abstract interpretation*.

The abstract interpretation <sup>(6)</sup> of a 1st-order program  $p$  is a finite universe of *ground entities* (AKA *atoms*) and first-order relations. This construct is also known as a finite *first-order structure* in mathematical logic [Barwise 77]. The formal definition of a 1st-order structure appears in the Appendix. Formally:

---

<sup>5</sup> Also called *coordination mapping* in [Eden & Janhke 02].

<sup>6</sup> In [Eden & Kazman 03] we use the terms *denotation function* and *design model*.

Definition II. Let  $L_1$  designate a 1st-order programming language,  $\mathcal{M}$  designate the space of all possible finite 1st-order structures (Definition VI), and  $\mathcal{A}$  designate a function such that

$$\mathcal{A} : L_1 \rightarrow \mathcal{M}$$

We say that  $\mathcal{A}$  is an abstract interpretation function.

Consider for example the abstract interpretation of the C++ program in Table 1. A possible such candidate is listed in Table 3.

Table 3. An abstract interpretation for the C++ program in Table 1

<p><u>Atoms:</u></p> <p>View, MyView, HerView of type <i>class</i>; View.Display, View.SetFocus, MyView.SetFocus, HerView.SetFocus of type <i>Method</i></p> <p><u>Relations:</u></p> <p><i>Abstract</i>(View)</p> <p><i>Abstract</i>(View.SetFocus)</p> <p><i>Inherit</i>(MyView, View)</p> <p><i>Inherit</i>(HerView, View)</p> <p><i>Invoke</i>(View.Display, View.SetFocus)</p> <p><i>Member</i>(View.Display, View)</p> <p><i>Member</i>(View.SetFocus, View)</p> <p><i>Member</i>(MyView.SetFocus, MyView)</p> <p><i>Member</i>(HerView.SetFocus, HerView)</p> <p><i>SameSignature</i>(MyView.SetFocus, View.SetFocus)</p> <p><i>SameSignature</i>(HerView.SetFocus, View.SetFocus)</p>
---

The choice of an abstract interpretation and the details of the function's definition highly depend on the nature of the 1st-order and 2nd-order languages in play. Consider for example the 2nd-order program in expression (1) (the specification of the Template Method design pattern LEPUS). It is trivial to prove formally that the 1st-order structure in Table 3 conforms to the specification in expression (1), but it would have been a lot more difficult (if at all possible) to prove that the C++ program in Table 1 conforms to the same expression.

More generally, depending on the abstract interpretation  $\mathcal{A}$ , it can be far easier to prove whether  $\mathcal{A}(p)$  (namely the abstract interpretation of a 1st-order program  $p$ ) conforms to a 2nd-order program  $P$ , than proving that  $p$  conforms to  $P$ . In the following section, we demonstrate how it can be proven.

## 4.2 Association mapping

The purpose of the *association mapping* is to manifest the linkage between the design and implementation, which is at the heart of the TTP solution to the problems discussed in Section 1.1. The mapping supports a two-way association that can be used to answer questions both about design and the implementation, i.e.,

- ♦ Program comprehension: “Which constraints apply to ‘this’ element of the implementation?”
- ♦ Traceability: “Which elements of the implementation must satisfy “this” constraint?”

Table 2 demonstrates a simple association mapping. More formally, given 1st-order program  $p$  and a 2nd-order program  $P$ , an association mapping  $P$  and  $p$  is a total *assignment* function (see Definition VIII in the Appendix), namely, it assigns each free variable of dimension  $k$  in  $P$  with a set of ground entities of dimension  $k$  in  $\mathcal{A}(p)$ . Thus, for example, the association mapping illustrated in Table 2 maps the free variable *abstract class* (expression (1)) to the ground entity `View` (Table 3), and the 1-dimensional variable *concrete-classes* to the 1-dimensional set of ground entities  $\{\text{MyView}, \text{HerView}\}$  (dimensions are defined in Definition VII in the Appendix.)

In the absence of an association mapping, the verification process of expressions such as (1) is a lot more complicated. What can mean a verification of with an expression with free variables mean? Such a query is normally taken as whether  $\mathcal{A}(p)$  “satisfies”  $P$ , written  $P \models \mathcal{A}(p)$ . Formally:

**Definition III.** We say that an abstract interpretation  $\mathcal{A}(p)$  satisfies the 2nd-order program  $P[x_1, \dots, x_n]$  (where  $x_1, \dots, x_n$  are the free variables in  $P$ ), written  $P \models \mathcal{A}(p)$ , iff there is a total assignment function  $\alpha$  (Definition VIII) from  $x_1, \dots, x_n$  to entities in  $\mathcal{A}(p)$  such that  $P[\alpha(x_1), \dots, \alpha(x_n)]$  (namely, the consistent replacement of  $x_i$  with  $\alpha(x_i)$  in  $P$ ) is true in  $\mathcal{A}(p)$ , written  $P[\alpha(x_1), \dots, \alpha(x_n)] \models \mathcal{A}(p)$ .

To illustrate this definition, consider the association mapping illustrated in Table 2, and the consistent replacement of the free variables in expression (1) (our 2nd-order program) with entities in the abstract interpretation, which yields the expression

```
View : C
{MyView, HerView} : P(C)
Display(), SetFocus(int) : S
```

---

```
Abstract(View)
Invoke(View.Display, View.SetFocus)
Inherit+ → ({MyView, HerView}, View)
```

which is evidently true in the structure listed in Table 3.

Clearly, an association mapping greatly simplifies the verification process since it is a total assignment function which provides the very proof that the abstract interpretation of an implementation  $p$  satisfies  $P$ , written as  $P[\alpha(x_1), \dots, \alpha(x_n)] \models \mathcal{A}(p)$ .

The association mapping contributes even in when the implementation seems “obvious”, such as in the example presented in Section 2. Among its benefits, it allows a supporting environment (e.g., an ADL tool) to perform automated verification of design specifications, which improve the reliability of programs and prevent the phenomena of architectural drift and architectural erosion discussed in Section 1.1.

We conclude with the formalization of the notion of a *two-tier program*:

Definition IV. Given a two-tier programming language  $\langle L_2, L_1, Sat \rangle$  and an abstraction interpretation function  $\mathcal{A}$ , the triple  $\langle P, p, \alpha \rangle$  is a two-tier program iff the following conditions apply:

- $P \in L_2$       ( $P$  is a 2nd-order program)
- $p \in L_1$       ( $p$  is a 1st-order program)
- $\alpha$  is a total assignment function (an *association mapping*) from  $P$  to  $p$

If  $P[\alpha(x_1), \dots, \alpha(x_n)] \models \mathcal{A}(p)$  then we say that the two-tier program  $\langle P, p, \alpha \rangle$  is consistent.

### 4.3 Two-tier compilation

As preventing *architectural drift* is pivotal to TTP, the lifecycle of a two-tier program is defined to ensure that changes are only allowed to one tier of specification before consistency is restored. More specifically, at each point in time in its evolution, changes are only allowed to be made in one tier of the specification. Before additional changes can be made, consistency is restored in what we refer to as an extension to the traditional notion of *compilation*.

We define the compilation of a two-tier program  $\pi = \langle P, p, \alpha \rangle$  inductively. Let us assume that in step  $n$  in the evolution of a two-tier program  $\pi = \langle P, p, \alpha \rangle$ ,  $\pi$  is *consistent* (Definition IV), i.e., that  $P[\alpha(x_1), \dots, \alpha(x_n)] \models \mathcal{A}(p)$ . We begin by assuming that our program  $\pi$  is consistent, and at each step in its evolution we only allow changes to be made only in one tier of  $\pi$ . Thus, either one of the following is true about the new program  $\pi'$ :

$$\pi' = \langle P', p, \alpha' \rangle \tag{5.1}$$

$$\pi' = \langle P, p', \alpha' \rangle \tag{5.2}$$

The general form of two-tier compilation follows naturally from this definition. It is either a process of *verification* or that of *justification*, depending whether (5.1) or (5.2) is true (namely, on which tier has been modified):

1. If changes were made to the 2nd-order tier, it is a “top-down” *verification* process whose purpose is to ensure, if  $\pi'$  is inconsistent or if the association mapping  $\alpha$  is obsolete, that  $\alpha$  is updated so as to reflect current and correct associations between the design and the implementation.
2. If changes were made to the 1st-order tier, it is a “bottom-up” *justification* process whose purpose is to ensure that the new implementation still conforms to the design specifications.

There are no general guidelines how consistency should be restored in the general case. Consistency can be restored by applying changes to the association mapping so as to reflect the changes made in either tier, and then proceed with an attempt to prove the consistency of the new program.

## 5. A sample two-tier programming language

To broaden our understanding of two-tier programming, we elaborate in the section on the TTPL of the sample program presented in Section 2.

Central to the definition of the *Satisfies* relation of this TTPL is the abstract interpretation function for C++. Consider for example the C++ program in Table 1 and the abstract interpretation proposed in Table 3. We assume a fixed abstract interpretation function  $\mathcal{A}$  which, as demonstrated by Eden and Hirshfeld [01], abstracts implementation detail from programs in object-oriented programming languages (such as C++, Java, Smalltalk and Eiffel) and facilitates reasoning on their design properties in LEPUS. Thus, given any C++ program  $p$ , we can safely assume that  $\mathcal{A}(p)$  is defined, and that it is a finite structure that consists of a collection of ground entities (*classes* and *methods*) and ground relations among them.

Given a fixed abstract interpretation function  $\mathcal{A}$ , we are free to formulate the following satisfaction relation.

**Definition V.** We say that two-tier program  $\langle P, p, \alpha \rangle$  is in the relation  $\models_{\mathcal{A}}$  iff the two-tier program  $\langle P, p, \alpha \rangle$  is *consistent* (Definition IV).

Although the relation  $\models_{\mathcal{A}}$  is ternary, we may use it as a *Satisfies* relation in the definition of a TTPL. Thus, the triple  $\langle LePUS, C++, \models_{\mathcal{A}} \rangle$  specifies a two-tier programming language, and that the two-tier program illustrated in Section 2 is in fact a program in this two-tier programming language. In particular, we demonstrated in Section 4.2 that this two tier program is consistent, which means that the implementation indeed satisfies the specification.

## 6. Related Work

Although two-tier programs have not hitherto been built, neither are they completely unprecedented. There is a substantial body of background research and many tools that are relevant to the development of such programs.

For example, many *Architecture Description Languages* (ADLs) [Garlan et. al 97; Luckham et. al 95] are designed to combine support for formal modelling with an architectural-based development tool. In some respects way ADLs are the closest relatives to TTP. Most ADLs, however, do not support both bottom-up and top-down propagation of changes, do not completely separate design from implementation, and most are, at best, only loosely couple with programming languages and development environments.

MetaH [Binns e. al 94], for example, is both a formalism that allows intensional specifications and an environment that generates extensional expressions, called ControlH. WRIGHT [Allen & Garlan 97] supports intensional specifications in a formalism that elaborates on *communicating sequential processes* (CSP), allowing users to represent *architectural styles* such as “client-server” and “shared memory”. WRIGHT, however, is intended to support the formulation properties such as *deadlock-freedom* and *liveness*. Thus, the validation of these properties requires dynamic analysis and cannot be verified by a static compilation process.

Similarly, *Rapide* [Kenney 96] was designed to validate the behaviour of concurrent systems and has been demonstrated for distributed transaction processing applications (specifically X/Open). Intensional specifications in this formalism declare behavioural constraints on the implementation (“executable systems” in *Rapide*’s terminology.) The environment is able to carry out both static “verification” process (such as the “validation” process described in Section 4.3), as well as a dynamic “validation” process of formal properties.

Murphy, Notkin and Sullivan [95] describe a system that has some similarities to the goals of TTP in the form of a suite of tools for the analysis of the composition of large software systems. The authors use the following terminology:

- ♦ The *source model* is a representation of the source code terms of atoms, such as *functions* and *classes*, and relations, such as *function calls*, written in the  $\mathbb{Z}$  specification language [Spivey 89].
- ♦ A *high-level model* is also a collection of atoms and relations among them. The authors do not discuss design languages or how the specification of design constraints can be characterised. Instead, the high-level model only differs from the source model in that it manifests the “desired state” of the system, which in turn can be compared with the “actual state” (represented by the source model.)
- ♦ The *declarative map* (roughly equivalent to the *association mapping*) is a morphism that expresses the expected correlations between the models.

The tool suite described by the authors compares the source model with the high-level model and yields a *reflexion model*, which indicates inconsistencies between the specifications and the implementation. This has been used primarily for determining architectural conformance in a post-facto mode—it is not integrated with a development environment and is not particularly well-suited to aid in the real-time process of implementation.

In a similar vein, but more closely related to TTP, the ArchJava system [Aldrich et al, 2002] is expressly designed for connecting architectures to implementations. ArchJava is

actually an extension to Java, which aids in connecting implementation to architecture and requires its own specialized compiler. With ArchJava, the ADL is effectively embedded within the programming language.

## 7. Summary and future directions

Consider relational databases with “schema evolution” [Date 00]: Facing a volatile domain, certain relational databases achieve greater flexibility by integrating both schema and data in the representation. Similarly, two-tier programs facilitate the validation of the design properties by integrating the first- with the second-order representation layers. This approach attempts to address the concerns of software engineering discussed above by redefining the notion of a *computer program* to answer additional requirements:

1. *Formal specification*: Well-defined specifications are preferred over the prolixity and ambiguity of informal documentation techniques.
2. *Traceability* (“top-down”): For each part of the design specifications, indicate the exact locus of its implementation(s) in the implementation.
3. *Comprehensibility* (“bottom-up”): For each expression in the program, indicate which part of the design specifications affect it.
4. *Consistency*: Ensure that the implementation “satisfies” the design specifications and prevent from architectural drift from developing.

We observe that TTPs do not offer an advantage unless changes to a large software-intensive system are made in an incremental, gradual, and stepwise process (“evolution”.) A formal analysis can facilitate the understanding of the extent of changes that justify the use of the TTP paradigm.

Formal analysis by means of the two-tier ontology offered in this paper can advance our understanding if carried out on various combinations of 1st- and 2nd-order programming languages. It may also be interesting to investigate the *Satisfies* relation for TTP in Smalltalk, i.e., when  $L_1=L_2=Smalltalk$ , as well as the relation maintained by mechanisms for associating design with implementations (e.g., as described by Baniassad, Murphy and Schwanninger [03].)

## Acknowledgements

The authors wish to thank Yehuda Elkana for his trust and support. We also wish to thank Mary J. Anna for her inspiration.

## Appendix

- Definition VI. A 1st-order structure (also *model*)  $\mathfrak{m}$  is a pair  $\langle \mathbb{U}_m, \mathbb{R}_m \rangle$ , such that
- $\mathbb{U}_m = \{ a_1, \dots, a_k \}$  is a finite set of *ground entities*, and
  - $\mathbb{R}_m = \{ \mathcal{R}_1, \dots, \mathcal{R}_n \}$  is a set of *ground relations* among the elements of  $\mathbb{U}_m$ .

The set of all 1st-order structures is designated  $\mathcal{M}$ .

- Definition VII. We say that a ground entity  $e$  (a variable  $x$  that ranges over ground entities) is of dimension 0. We say that a finite set of entities (a variable  $y$  that ranges over entities) of dimension  $n-1$  is an entity (a variable) of dimension  $n$ .

- Definition VIII. Let  $p$  designate a 1st-order program,  $P[x_1, \dots, x_n]$  a 2nd-order program with free variables  $x_1, \dots, x_n$ . Let  $\mathcal{A}(p)$  designate the abstract interpretation of  $p$ . A function  $\alpha$  is an assignment to variable  $x_i$  if it  $x_i$  is a free variable of dimension  $k$  in  $P$  and  $\alpha(x_i)$  is a set of dimension  $k$  of ground entities in  $\mathcal{A}(p)$ .

## References

- J. Aldrich, C. Chambers, D. Notkin (2002). "ArchJava: Connecting Software Architecture to Implementation". In: *Proceedings of International Conference for Software Engineering – ICSE 2002*, May 19—25, 2002, Orlando, FL.
- R. Allen, D. Garlan (1997). "A Formal Basis for Architectural Connection." *ACM Transactions on Software Engineering and Methodology*, Vol. 6, No. 3 (July 1997), pp. 213—249.
- E. L. A. Baniassad, G. C. Murphy, C. Schwanninger. "Design Pattern Rationale Graphs: Linking Design to Source." *Proc. of the 25th IEEE International Conference in Software Engineering*, May 6—8, 2003, Portland, OR.
- J. Barwise, ed. (1977). *Handbook of Mathematical Logic*. Amsterdam, The Netherlands: North-Holland Publishing Co.
- P. Binns, M. Englehart, M. Jackson, S. Vestal (1994). "Domain Specific Software Architectures for Guidance, Navigation, and Control." *Technical report*. Honeywell Technology Center, Minneapolis, Minnesota, January 1994.
- B. Boehm (1988). "A Spiral Model for Software development and Enhancement." *IEEE Computer.*, Vol. 21, No. 5 (May 1988), pp. 440—454.
- F. P. Brooks. "No Silver Bullet: Essence and Accidents in Software Engineering." *IEEE Computer*, Vol. 20, No. 4 (Apr. 1987), pp. 10—19.

- E. R. Comer (1997). "Alternative Software Life Cycle Models." In *Software Engineering*, M. Dorfman, R. Thayer (eds.), pp. 404—414. Piscataway, NJ: IEEE Computer Society Press.
- W. Cunningham, A. Hunt, B. Marick, D. Thomas (2002). "Software Archeology : Understanding Large Systems", *Workshop in conjunction with the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications – OOPSLA '01*, October 14—18, 2001, Tampa, Florida, USA.
- C. J. Date (2000). *The Database Relational Model: A Retrospective Review and Analysis*. Reading, MA: Addison Wesley Longman, Inc.
- T. R. Dean, J. R. Cordy (1995). "A Syntactic Theory of Software Architecture". *IEEE Transactions on Software Engineering*, Vol. 21, No. 4, April 1995.
- A. H. Eden (2000). "Precise Specification of Design Patterns and Tool Support in their Application." PhD diss., Department of Computer Science, Tel Aviv University.
- A. H. Eden (2001). "Formal Specification of Object-Oriented Design." *International Conference on Multidisciplinary Design in Engineering CSME-MDE 2001*. Montreal, Canada, November 21—22.
- A. H. Eden (2002a). "LePUS: A Visual Formalism for Object-Oriented Architectures." *The 6th World Conference on Integrated Design and Process Technology*, June 26—30, Pasadena, CA.
- A. H. Eden, Y. Hirshfeld (2001). "Principles in Formal Specification of Object Oriented Architectures." *CASCON 2001*, November 5—8, 2001, Toronto, Canada.
- A. H. Eden, R. Kazman (2003). "Architecture, Design, and Implementation." *International Conference on Software Engineering – ICSE 2003*, May 3—10, 2003, Portland, OR.
- Y. Elkana (1976). "Two-Tier Thinking: Philosophical Realism and Historical Relativism". *IYYUN*, Vol. 27 (1976—77).
- E. Gamma, R. Helm, R. Johnson, J. Vlissides (1994). *Design Patterns: Elements of Reusable Object Oriented Software*. Reading, MA: Addison-Wesley.
- D. Garlan, R. Monroe, D. Wile (1997). "ACME: An Architectural Description Interchange Language." *Proceedings of CASCON'97*, Nov. 1997, Toronto, Canada.
- D. Garlan, M. Shaw (1993). "An Introduction to Software Architecture." In V. Ambriola, G. Tortora, eds., *Advances in Software Engineering and Knowledge Engineering*, Vol. 2, pp. 1—39. New Jersey: World Scientific Publishing Company.
- W. W. Gibbs (1994). "Software's Chronic Crisis." *Scientific American*, Vol. 271, No. 3, pp. 86—95.
- A. Goldberg, D. Robson (1989). *Smalltalk-80: The Language*. Reading, MA: Addison Wesley Longman, Inc.
- J. V. Guttag, J. J. Horning (1993). *Larch: Languages and Tools for Formal Specification*. Berlin: Springer-Verlag.
- J. V. Guttag, J. J. Horning, J. M. Wing (1982). "Some remarks on putting formal specifications to productive use." *Science of Computer Programming*, Vol. 2, No. 1 (Oct. 1982), pp. 53—68.

- D. Harel. "Statecharts: A Visual Formalism for Complex Systems." *Science of Computer Programming*, Vol. 8, No. 3 (June 1987), pp. 231—274.
- C. A. Hoare (1985). *Communicating Sequential Processes*. Upper Saddle River, NJ: Prentice-Hall.
- R. Kazman, S. J. Carriere, "Playing Detective: Reconstructing Software Architecture from Available Evidence", *Automated Software Engineering* Vol. 6 No. 2, April 1999, 107—138.
- R. K. Keller, R. Schauer, S. Robitaille, P. Pagé (1999). "Pattern-based reverse-engineering of design components." *Proceedings of the 1999 International Conference on Software Engineering*, May 16 – 22, 1999.
- J. J. Kenney (1996). "Executable Formal Models of Distributed Transaction Systems based on Event Processing," PhD diss., Department of Electrical Engineering, Stanford University.
- L. Lamport. "The Temporal Logic of Actions." *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 3 (May 1994), pp. 872—923.
- A. Lauder, S. Kent (1998). "Precise Visual Specification of Design Patterns." In: E. Jul (ed.), *Proceedings of the 12th European Conference on Object Oriented Programming, Brussels, Belgium*, Lecture Notes in Computer Science 1445, July 1998. Berlin: Springer-Verlag.
- M. M. Lehman (1980). "Programs, Life Cycles, and Laws of Software Evolution." *Proceedings of the IEEE*, Vol. 68, No. 9 (Sep. 1980), pp. 1060—1076.
- M. M. Lehman (1997). "Laws of software evolution revisited." *EWSPJ'96, Lecture Notes in Computer Science 1149*, pp. 108—124. Berlin, Germany: Springer-Verlag.
- D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, W. Mann (1995). "Specification and Analysis of System Architecture Using Rapide." *IEEE Transactions on Software Engineering*, Special Issue on Software Architecture, Vol. 21, No. 4, pp. 336-355, April 1995.
- O. L. Madsen (2000). "Towards a Unified Programming Language." *ECOOP 2000, Lecture Notes in Computer Science 1850*, pp. 1—26. Berlin, Germany: Springer-Verlag.
- T. Mikkonen (1998). "Formalizing Design Patterns." *Proceedings of the International Conference on Software Engineering*. Kyoto, Japan, April 19 - 25.
- G. C. Murphy, D. Notkin, K. Sullivan (1995). "Software Reflexion Models: Bridging the Gap Between Source and High-Level Models". *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, October 1995, pp. 18-28. New York, NY.
- D. E. Perry, A. L. Wolf (1992). "Foundation for the Study of Software Architecture". *ACM SIGSOFT Software Engineering Notes*, 17 (4), pp. 40—52.
- C. A. Petri (1962). "Communications with Automata." *Technical report RADC-TR-65-377*. Princeton, NJ, Applied Data Research.

- R. S. Pressman (1997). "Software Engineering", Chap. in: M. Dofrman, R. H. Thayer (eds.) *Software Engineering*, pp. 57—74. Los Alamos, CA: IEEE Computer Society Press.
- L. Rising (2000). *The Pattern Almanac 2000*. Reading, MA: Addison Wesley Professional.
- SEI 1995. The Software Engineering Institute *Glossary of Software Engineering Terms*.
- I. Sommerville (2000). *Software Engineering*, 6th Edition. Reading, MA: Addison-Wesley
- J. M. Spivey (1989). *The Z Notation: A Reference Manual*. New Jersey: Prentice Hall.
- A. Turing (1936). "On Computable Numbers, with an Application to the Entscheidungsproblem." *Proceedings of the London Mathematical Society*, Vol. 2, No. 42 (1936-7), pp. 230—236.