Date: 24-Feb-03

Technical report CSM-382
Department of Computer Science, University of Essex, United Kingdom

The Integrated Architecture Environment: A Two-Tier Programming Tool

RESEARCH PROPOSAL

Amnon H. Eden

Planned in collaboration with Rick Kazman, Software Engineering Institute, Pittsburgh, PA

Abstract

We describe a software development environment that supports Two-Tier Programming (TTP): An approach that redefines *programming* as the integration of conventional implementation with design specifications. The TTP tool maintains a comprehensive representation of programs by integrating specifications in two ties:

1. Design specifications are represented as a library of design specifications defined formally in a visual specification language (LePUS);
2. The implementation, which strictly adheres to conventional programming practices.

A *mapping* associates design specifications with their instances and facilitates the coordination of changes between the two tiers.

Index terms. Software design and architecture, object oriented programming, formal specification, design patterns, verification and validation, automated software engineering, visualization.

Related documents

A. H. Eden (2002). "LePUS: A Visual Formalism for Object-Oriented Architectures". *The 6th World Conference on Integrated Design and Process Technology*, Pasadena, California, June 22—28, 2002.

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, 2001.

A. H. Eden, R. Kazman (2003). “Two-Tier Programming.” Under preparation.

1. Two-Tier Programming

Two-tier programming [Eden & Jahnke 02] is an emerging paradigm according to which the very representation of a computer program should integrate design specifications along with the implementation. Thus, a two-tier program constitutes specifications in at least two tiers of specification:

1. Intensional specifications [Eden & Kazman 03a], or design specifications, are formulas which may be instantiated in an unbounded number of contexts ("design models").

2. The implementation is represented in a conventional programming language.

The representation of a traditional program consists of text in a 3rd generation programming language ("source code"), commonly distributed over a number of ASCII files accompanied with "build" information. Serious problems arise from such practice, most notably *architectural drift* [Perry & Wolf 92] and *traceability*, which hamper the continuous evolution of programs.

In contrast, a two-tier program tightly integrates design information into its very representation. Thus, the "compilation" of a TTP (i.e., its static validation) requires also the verification of the design requirements. The mapping between the intensional design specifications and their instances in the source code is supported via a *coordination layer*, which simply consists of a list of pairs in the form: $\langle$`constant`, *variable*$\rangle$.

Architectural consistency

To ensure consistency between the design and the implementation, the TTP approach dictates a step-wise construction process of software along the following lines:

1. **Two-tiers**. The representation of the program is divided into two layers of specifications:

    a. An *intensional tier* consists of intensional formulas which capture design specifications.

    b. The *extensional tier* consists of the implementation, which is source code written in a traditional programming language.

    A *coordination layer* maps each intensional formula with each one of its instances in the source code.

2. **Undisturbed Coding**. Coordination does NOT depend on extending or adding the implementation (e.g., it does not require embedded comments or language extensions.) Our implementation will allow the programmer to freely switch from conventional programming to the architectural specifications at any point in the development.

3. **Quanta of Change**. Changes are made to the program in small units.

4. **Locality**. Each quantum of change is restricted to only one tier of specifications.

5. **Consistency**. Consistency between the representation tiers is restored after each quantum of change, before any other change may take place.

2. Nomenclature

The following definitions are extracted from a number of resources, most prominently: “A Theory of Object-Oriented Design” [Eden 02] and “Architecture, Design, Implementation” [Eden & Kazman 03a]. For the complete, formal definition, please refer to these sources.

design schema: A `LePUS` diagram

design model: A finite structure in mathematical logic which consists of *ground entities* (usually, of types *Class* and *Routine*) and *ground relations*

domain is the set of all entities of a certain property. The symbols $\mathbb{F}$ and $\mathbb{C}$ are used to represent the domains of *ground routines* and *ground classes* respectively. We use the notation $\mathbf{P}(\mathbb{T})$ to represent the (“derived”) domain of sets of entities of type $\mathbb{T}$.

first-order language: A statically-typed OOP language, Java™ in our case.

formula: A design specification written in `LePUS` in our case.

second-order language: A language for writing design specifications, `LePUS` in our case.

3. General Requirements

User The TTP tool is a development environment that is meant to support assist users who are both programmer and designer. More specifically, the *Integrated Architectural Environment* (IAE) should support the following:

1. **Implementation**: Any kind of traditional programming activity, such as editing, compilation and debugging.
    - Auxiliary code comprehension techniques would be welcome, such as `javadoc` or any other static browser.
2. **Design**: Comprehensive support in design specifications, specifically statically-analyzed properties in association with properties expressed by design patterns and architectural styles:
    (a) *Specification* (e.g., of the *State* pattern)
    (b) *Association* (e.g., of the *State* with its instances in the implementation)
3. **Verification** of design properties (e.g., that the designated instances of the *State* pattern satisfy the specification)

Figure 1 illustrates the services the IAE supports. Below we elaborate on each element thereof. Figure 2 depicts a screen shot of the tool at work.

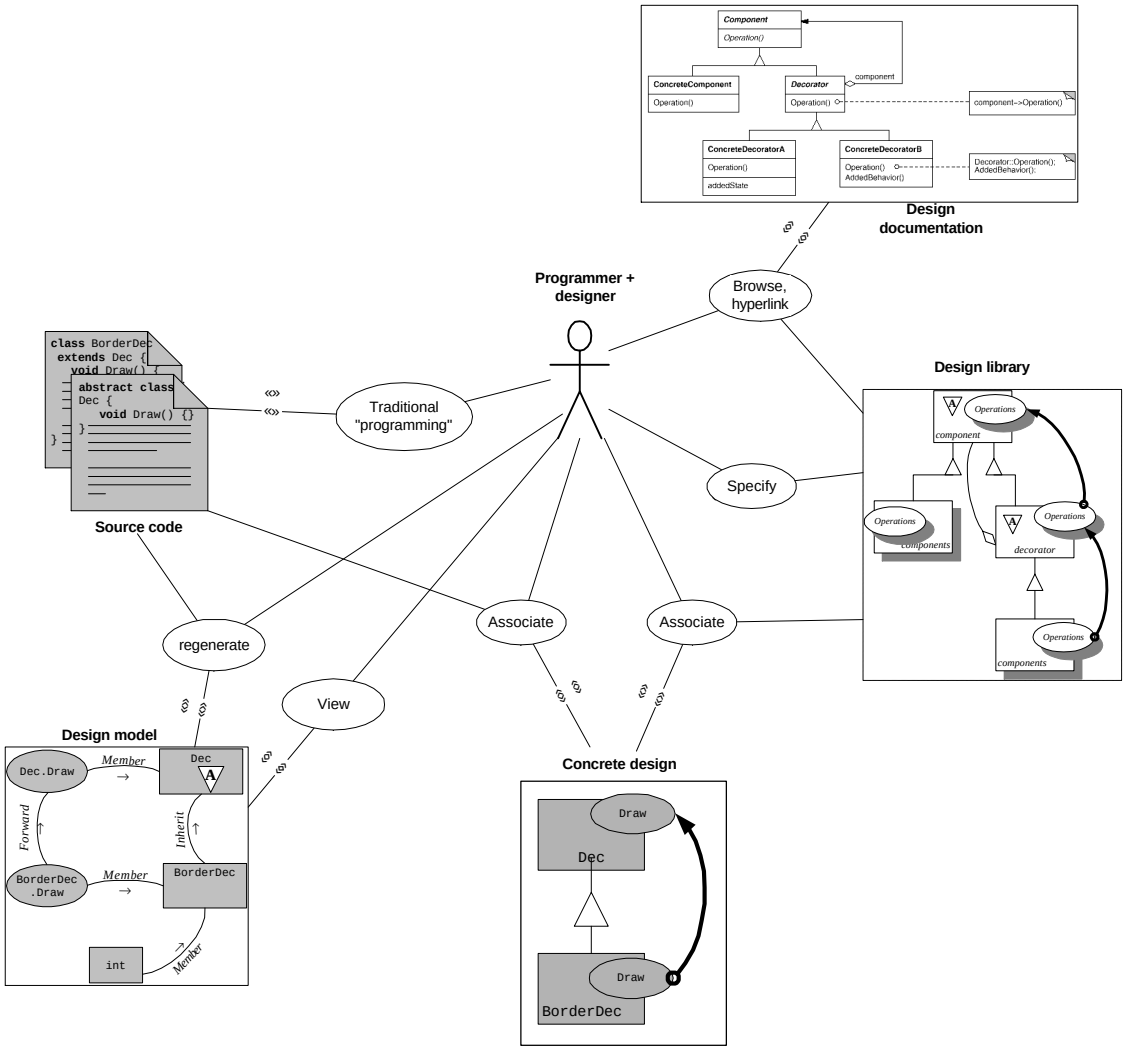

Figure 1. A use-case diagram capturing the user's interactions with different parts of the TTP tool.

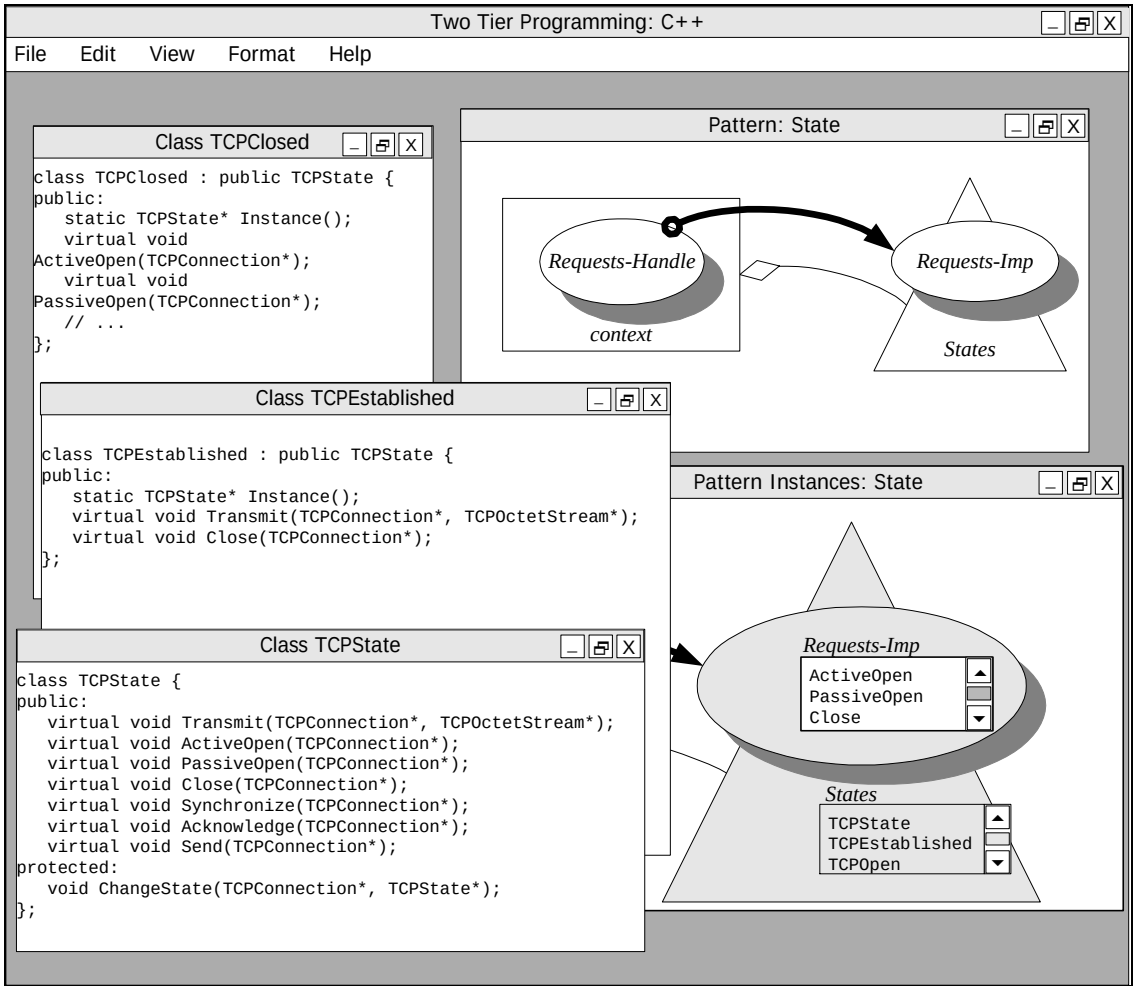

Figure 2. Sample screen shot of the TTP tool

3.1 General

The subject matter of the IAE should be handled via *projects*, which consist a collection of references to the specifications of a specific Java program. A project should consist of references as follows:

- Implementation:
  1. Java source code files (possibly in the form of path expression and a regular expression)
  2. Executable build information (implementation-dependent)
  3. A *design model* for the relevant portion of the source code (Figure 4)
- Design specifications, in either one of the forms:
  1. A library of open formulas (i.e., with free variables)
  2. Closed formulas (i.e., literals are only constant expressions)
- Association relation mapping the design with the implementation ("coordination layer").

At any instance, at most one project can be open.

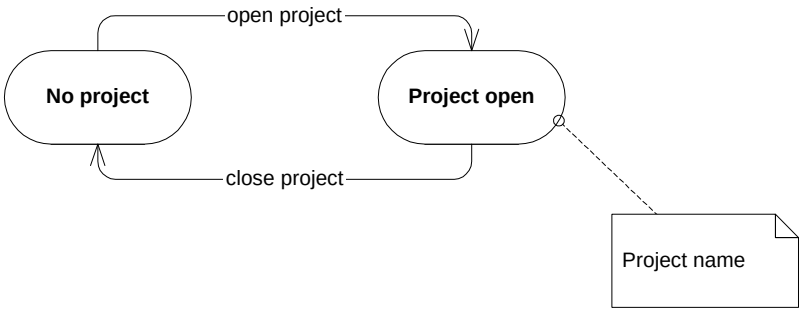

Figure 3. A state diagram for the IAE

3.2 Implementation

Implementation should be specified using traditional `.java` source code files along with the libraries and all the build information needed to create a complete executable (or the byte-code equivalent). No other kind of information must exist in the source code other than what ordinary Java programs contain. In particular, we do not expect comments embedded in the program.

Source code should be accessible for editing, compilation, debugging, and old-fashioned means of browsing (`javadoc`) as frequently as the user may be arbitrarily wish.

The IAE should generate a design model from the given source code and represent it both visually and textually. Figure 4 depicts an example for a visual depiction of a design model.

```
abstract class Dec {
   abstract void Draw();
}

class BorderDec extends Dec {
   void Draw() {
      // do_something();
      Dec.Draw(); //...
   }
   int BorderWidth;
}
```

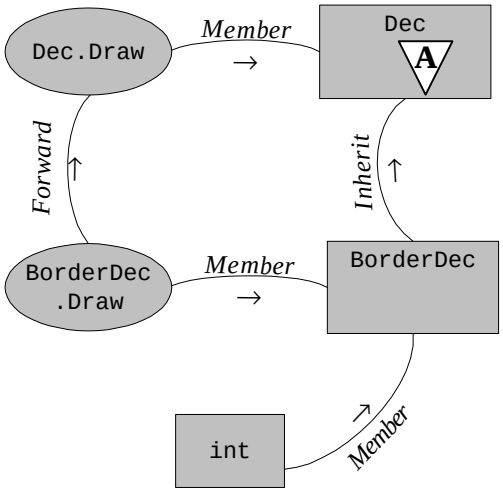

Figure 4. Java™ code excerpt and a visual depiction of its design model

3.3 Design

Design specification should support specification in both symbolic and visual version of LePUS. A specifications may include declarations of variables and constants of a specific domain, and any number of well-formed clauses, which consist of any combination of the language's predefined predicates over relations and operators. Table 1 depicts a well-formed LePUS expression which consists of three declarations, two predicates, and two instances of the selection operator. Figure 5 depicts the visual version of the same expression.

Table 1. A simple **LePUS** expression (symbolic representation).

`Dec`, *component* : ℂ `Draw`, *Operations* : **P**(𝕊)
Inherit(component, `dec`*)* *Forward*$^{\leftrightarrow}$*(Operations*⊗*component,* `Draw`⊗`Dec`*)*

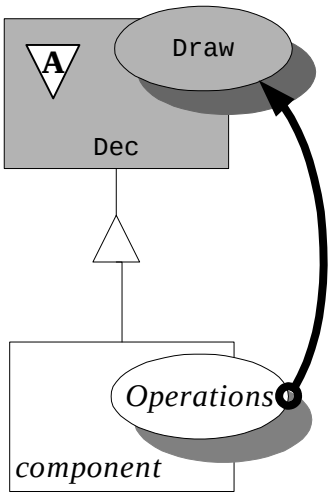

Figure 5. Visual depiction of the **LePUS** expression in Table 1

A LePUS formula is a design specification interpreted as follows:

- Constants (e.g., dec) appearing anywhere represent an entity in the design model by the respective name, if exists; otherwise, *verification* fails.
- An expression with free variables (such as *component*) is interpreted only in the context of a specific *association*, if exists in the coordination layer.

3.4 Association

Ground constants are the only literals that can be directly interpreted. Other then these, we need consider the interpretation of the following two types of literals:

- **Variables**: An association states that a part of the design model should conform to (is an "instance" of) a certain formula.
- **Higher-dimension constants**: The association states which ground entities in the design model are represented as a higher-dimension constant.

Each association provides interpretation to the variables and higher-dimension in a single LePUS expression.

Signature variables are not associated as such; instead, each occurrence of the selection operator with any variable requires an assignment.

Table 2. A sample association assigning interpretations from the design model in Figure 4 to the expression in Table 1.

component, `BorderDec`
Operations⊗*component*, `BorderDec.Draw`
`Draw`⊗`Dec`, `Dec.Draw`

There are further issues to consider with relation to the association of signature and class hierarchy variables.

3.5 Verification

The purpose of a verification is to certify that the associations are valid. For example, that constants have a respective interpretation (in the design model) and that the interpretations of all expressions are true in the design model. If a verification fails, the point of failure (association and formula) should be made clear.

References

A. H. Eden (2002). "LePUS: A Visual Formalism for Object-Oriented Architectures". *The 6th World Conference on Integrated Design and Process Technology*, Pasadena, California, June 22—28, 2002.

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, 2001.

A. H. Eden. "A Theory of Object-Oriented Design." *Information Systems Frontiers*, Vol. 4, No. 4 (Nov.-Dec. 2002) Kluwer Academic Publishers.

A. H. Eden, J. Jahnke (2002). "Coordinating Software Evolution Via Two-Tier Programming." *Coordination 2002, Lecture Notes in Computer Science, Vol. 2315, pp. 149-159*. Arbab F.; Talcott C (Eds.) Berlin, Germany: Springer.

A. H. Eden, R. Kazman (2003a). "Architecture, Design, Implementation." *International Conference on Software Engineering – ICSE*. Portland, OR, May 3-10, 2003.

A. H. Eden, R. Kazman (2003b). “Two-Tier Programming.” Under preparation.

D. E. Perry, A. L. Wolf. "Foundation for the Study of Software Architecture." *ACM SIGSOFT Software Engineering Notes*, Vol. 17, No. 4 (Oct. 1992), pp. 40-52.