Class Refinement and Interface Refinement in Object-Oriented Programs

Anna Mikhajlova[1] and Emil Sekerinski[2]

[1] Turku Centre for Computer Science, Åbo Akademi University
Lemminkäisenkatu 14A, Turku 20520, Finland
[2] Dept. of Computer Science, Åbo Akademi University
Lemminkäisenkatu 14A, Turku 20520, Finland

Abstract. Constructing new classes from existing ones by inheritance or subclassing is a characteristic feature of object-oriented development. Imposing semantic constraints on subclassing allows us to ensure that the behaviour of superclasses is preserved or refined in their subclasses. This paper defines a class refinement relation which captures these semantic constraints. The class refinement relation is based on algorithmic and data refinement supported by Refinement Calculus. Class refinement is generalized to interface refinement, which takes place when a change in user requirements causes interface changes of classes designed as refinements of other classes. We formalize the interface refinement relation and present rules for refinement of clients of the classes involved in this relation.

1 Introduction

It has been widely recognized that design and development of object-oriented programs is difficult and intricate. The need for formal basis of object-oriented development was identified by many researchers. We demonstrate how formal methods, in particular, Refinement Calculus of Back, Morgan, and Morris [4, 20, 21], can be used for constructing more reliable object-oriented programs.

A characteristic feature of object-oriented program development is a uniform way of structuring all stages of the development by classes. The programming notation of Refinement Calculus is very convenient for describing object-oriented development because it allows us to specify classes at various abstraction levels. The specification language we use is based on monotonic predicate transformers, has class constructs, supports subclassing and subtype polymorphism. Besides usual imperative statements, the language includes specification statements which may appear in method bodies of classes leading to abstract classes. One of the main benefits offered by this language is that all development stages can be described in a uniform way starting with a simple abstract specification and resulting in a concrete program.

We build a logic of object-oriented programs as a conservative extension of (standard) higher-order logic, in the style of [6]. An alternative approach is undertaken by Abadi and Leino in [2]. They develop a logic of object-oriented programs in the style of Hoare, prove its soundness, and discuss completeness issues. Naumann [22] defines the semantics of a simple Oberon-like programming language with similar specification constructs as here, also based on predicate transformers. Sekerinski [24, 25] defines a rich object-oriented programming and specification notation by using a type system with subtyping and type parameters, and also using predicate transformers. In both approaches, subtyping is based on extensions of record types. Here we use sum types instead, as suggested by Back and Butler in [7]. One motivation for moving to sum types is to avoid the complications in the typing and the logic when reasoning about record types: the simple typed lambda calculus as the formal basis is sufficient for our purposes. Another advantage of moving to sum types is that we can directly express whether an object is of exactly a certain type or of one of its subtypes (in the record approach, a type contains all the values of its subtypes). Using summations also allows us to model contravariance and covariance on method parameters in a simple way. Finally, to allow objects of a subclass to have different (private) attributes from those of the superclass, hiding by existential types was used in [24, 25]. It turned out that this leads to complications when reasoning about method calls, which are not present when using the model of sum types. In the latter, objects of a subclass can always have different attributes from those of the superclass.

Constructing new classes from existing classes by inheritance, or subclassing, is one of characteristic features of object-oriented program development. However, when a subclass overrides some methods of its superclass, there are no guarantees that its instances will deliver the same or refined behaviour as the instances of the superclass. We define a class refinement relation and relate the notion of subclassing to this relation. When a class C' is constructed by subclassing from C and class refinement holds between them, then it is guaranteed that any behaviour expected from C will necessarily be delivered by C'.

Class refinement as defined here is based on data refinement [15, 14, 19, 5]. The definition generalizes that of Sekerinski [24] by allowing contravariance and covariance in the method parameters, and by considering constructor methods. Class refinement has also been studied in various extensions of the Z specification languages, e.g. [16, 17], but only between class specifications and not implementations. Other approaches on "behavioural subtyping" of classes [3, 18, 10] also make a distinction between the specification of a class and its implementation. By having specification constructs as part of the (extended) programming language, this distinction becomes unnecessary.

Subclassing requires that parameter types of a method be the same in the subclass and in the superclass or, at most, subject to contravariance and covariance rules, as described in [9, 1]. However, sometimes a change in user requirements causes interface changes of classes designed as refinements of other classes. We formalize the interface refinement relation as a generalization of class refinement, and present rules for refinement of clients of the classes involved in this relation. Interface refinement has also been considered by Broy in [8], but for networks of communicating components rather than for classes.

Paper Outline: In Section 2 we present the required concepts of the Refinement Calculus formalism. In Section 3 we explain our model of objects, classes, subclassing, and subtyping polymorphism. Section 4 defines the class refinement relation. In the following section we generalize class refinement to interface refinement, formalize implicit client refinement, and discuss explicit client refinement. Finally, we conclude with considering applications of our work.

2 Refinement Calculus

A *predicate* over a set of states Σ is a boolean function $p : \Sigma \to Bool$ which assigns a truth value to each state. The set of predicates on Σ is denoted $\mathcal{P}\Sigma$. The *entailment ordering* on predicates is defined by pointwise extension, so that for $p, q : \mathcal{P}\Sigma$,

$$p \subseteq q \;\widehat{=}\; (\forall \sigma : \Sigma \cdot p\ \sigma \Rightarrow q\ \sigma)$$

A *relation* from Σ to Γ is a function $P : \Sigma \to \mathcal{P}\Gamma$ that maps each state σ to a predicate on Γ. We write

$$\Sigma \leftrightarrow \Gamma \;\widehat{=}\; \Sigma \to \mathcal{P}\Gamma$$

to denote a set of all relations from Σ to Γ. This view of relations is isomorphic to viewing them as predicates on the cartesian space $\Sigma \times \Gamma$. The *identity relation* and the *composition* of relations are defined as follows:

$$\begin{aligned} Id\ x\ y \;&\widehat{=}\; x = y \\ (P; Q)\ x\ z \;&\widehat{=}\; (\exists y \cdot P\ x\ y \wedge Q\ y\ z) \end{aligned}$$

A *predicate transformer* is a function $S : \mathcal{P}\Gamma \to \mathcal{P}\Sigma$ from predicates to predicates. We write

$$\Sigma \mapsto \Gamma \;\widehat{=}\; \mathcal{P}\Gamma \to \mathcal{P}\Sigma$$

to denote a set of all predicate transformers from Σ to Γ. Program statements in Refinement Calculus are identified with weakest-precondition monotonic predicate transformers that map a postcondition $q : \mathcal{P}\Gamma$ to the weakest precondition $p : \mathcal{P}\Sigma$ such that the program is guaranteed to terminate in a final state satisfying q whenever the initial state satisfies p. A program statement S need not have identical initial and final state spaces, though if it does, we write $S : \Xi(\Sigma)$ instead of $S : \Sigma \mapsto \Sigma$.

The *refinement ordering* on predicate transformers is defined by pointwise extension, for $S, T : \Sigma \mapsto \Gamma$:

$$S \sqsubseteq T \;\widehat{=}\; (\forall q : \mathcal{P}\Gamma \cdot S\ q \subseteq T\ q)$$

The refinement ordering on predicate transformers models the notion of total-correctness preserving program refinement. For statements S and T, the relation $S \sqsubseteq T$ holds if and only if T satisfies any specification satisfied by S.

The **abort** statement maps each postcondition to the identically false predicate *false*, and the **magic** statement maps each postcondition to the identically true predicate *true*. The **abort** statement is never guaranteed to terminate, while the **magic** statement is *miraculous* since it is always guaranteed to establish any postcondition.

Sequential composition of program statements is modeled by functional composition of predicate transformers. For $S : \Sigma \mapsto \Gamma$, $T : \Gamma \mapsto \Delta$ and $q : \mathcal{P}\Delta$,

$$(S;T)\ q \ \mathrel{\widehat{=}}\ S\ (T\ q)$$

The program statement $\mathbf{skip}_\Sigma$ is modeled by the identity predicate transformer on $\mathcal{P}\Sigma$.

Given a relation $P : \Sigma \leftrightarrow \Gamma$, the *angelic update statement* $\{P\} : \Sigma \mapsto \Gamma$ and the *demonic update statement* $[P] : \Sigma \mapsto \Gamma$ are defined by

$$\begin{aligned}\{P\}\ q\ \sigma &\ \mathrel{\widehat{=}}\ (\exists \gamma : \Gamma \ \cdot\ (P\ \sigma\ \gamma)\ \wedge\ (q\ \gamma)) \\ [\mathrm{P}]\ q\ \sigma &\ \mathrel{\widehat{=}}\ (\forall \gamma : \Gamma \ \cdot\ (P\ \sigma\ \gamma)\ \Rightarrow\ (q\ \gamma))\end{aligned}$$

When started in a state σ, $\{P\}$ angelically chooses a new state γ such that $P\ \sigma\ \gamma$ holds, while $[P]$ demonically chooses a new state γ such that $P\ \sigma\ \gamma$ holds. If no such state exists, then $\{P\}$ aborts, whereas $[P]$ behaves as **magic**, i.e. can establish any postcondition.

Ordinary program constructs may be modeled using the basic predicate transformers and operators presented above. For example, in a state space with two components $(x : T, y : S)$, an assignment statement may be modeled by the demonic update:

$$x := e \ \ \mathrel{\widehat{=}}\ \ [R],\ \text{where } R\ (x,y)(x',y')\ =\ (x' = e) \wedge (y' = y)$$

Our specification language includes specification statements. The *demonic specification statement* is written $[x := x' \cdot b]$, and the *angelic specification statement* is written $\{x := x' \cdot b\}$, where b is a boolean expression relating x and x'. The program variable x is assigned a value x' satisfying b. These statements correspond to the demonic and the angelic updates respectively:

$$x := x' \cdot b \ \ \mathrel{\widehat{=}}\ \ R,\ \text{where } R\ (x,y)(x',y')\ =\ b\ \wedge (y' = y)$$

We also have an *assertion*, written $\{p\}$, where p is a predicate stating a condition on program variables. This assertion behaves as **skip** if p is satisfied and as **abort** otherwise.

Finally, the language supports *local variables*. The construct $|[\ \mathbf{var}\ z\ \bullet\ S\ ]|$ states that the program variable z is local to S:

$$|[\ \mathbf{var}\ z\ \bullet\ S\ ]|\ \ \mathrel{\widehat{=}}\ \ [\mathit{Enter}_z]; S; [\mathit{Exit}_z],\ \text{where}$$

$$\begin{aligned}\mathit{Enter}_z(x,y)(x',y',z') &\ \mathrel{\widehat{=}}\ (x' = x) \wedge (y' = y) \text{ and} \\ \mathit{Exit}_z(x,y,z)(x',y') &\ \mathrel{\widehat{=}}\ (x' = x) \wedge (y' = y)\end{aligned}$$

The semantics of other ordinary program constructs, like multiple assignments, **if**-statements, and **do**-loops, is given, e.g. in [6].

Data refinement is a general technique by which one can change the state space in a refinement. For statements $S : \Xi(\Sigma)$ and $S' : \Xi(\Sigma')$, let $R : \Sigma' \leftrightarrow \Sigma$ be an *abstraction relation* between the state spaces Σ and Σ'. The statement S is said to be data refined by S' via R, denoted $S \ \sqsubseteq_R \ S'$, if

$$\{R\};\ S \ \ \sqsubseteq \ \ S';\ \{R\}$$

Alternative and equivalent characterizations of data refinement using the inverse relation R^{-1}, are then

$$S; [R^{-1}] \ \sqsubseteq \ [R^{-1}]; S' \qquad S \ \sqsubseteq \ [R^{-1}]; S'; \{R\} \qquad \{R\}; S; [R^{-1}] \ \sqsubseteq \ S'$$

These characterizations follow from the fact that $\{R\}$ and $[R^{-1}]$ are each others inverses, in the sense that $\{R\}; [R^{-1}] \ \sqsubseteq \ \mathbf{skip}$ and $\mathbf{skip} \ \sqsubseteq \ [R^{-1}]; \{R\}$.

Refinement Calculus provides laws for transforming more abstract program structures into more concrete ones based on the notion of refinement of predicate transformers presented above. A large collection of algorithmic and data refinement laws is given in [6, 20, 12].

Sum Types and Operators. In our specification language we widely employ sum types for modeling subtyping polymorphism and dynamic binding. The sum or disjoint union of two types Σ and Γ is written $\Sigma + \Gamma$. The types Σ and Γ are called base types of the sum in this case. Associated with the sum types, are the injection relations[3] which map elements of the subsets to elements of the superset summation:

$$\iota_\Sigma : \Sigma \leftrightarrow \Sigma + \Gamma \qquad \iota_\Gamma : \Gamma \leftrightarrow \Sigma + \Gamma$$

and projection relations which relate elements of summation with elements of their subsets:

$$\pi_\Sigma : \Sigma + \Gamma \leftrightarrow \Sigma \qquad \pi_\Gamma : \Sigma + \Gamma \leftrightarrow \Gamma$$

In fact, the projection relation is an inverse of the injection relation for the corresponding subset of the summation.

We define the subtype relation as follows. The type Σ is a subtype of Σ', written $\Sigma <: \Sigma'$, if $\Sigma = \Sigma'$, or $\Sigma <: \Gamma$ or $\Sigma <: \Gamma'$, where $\Gamma + \Gamma' = \Sigma'$. For example, $\Gamma <: \Gamma + \Gamma'$ and, or course, $\Gamma + \Gamma' <: \Gamma + \Gamma'$. If Σ is a subtype of Σ', we can always construct the appropriate injection $\iota_\Sigma : \Sigma \leftrightarrow \Sigma'$ and projection $\pi_\Sigma : \Sigma' \leftrightarrow \Sigma$. The subtype relation is reflexive, transitive and antisymmetric.

A summation operator combines statements by forming the disjoint union of their state spaces. This operator is defined in [7] by extension from the summation of types. For $S_1 : \Sigma_1 \mapsto \Gamma_1$ and $S_2 : \Sigma_2 \mapsto \Gamma_2$, the summation $S_1 + S_2 : \Sigma_1 + \Sigma_2 \ \mapsto \ \Gamma_1 + \Gamma_2$ is a predicate transformer such that the effect of

[3] In fact, the injections are functions rather than relations, but for our purposes it is more convenient to treat them as relations.

executing it in some initial state σ depends on the base type of σ. If $\sigma : \Sigma_1$ then S_1 is executed, while if it is of type Σ_2, then S_2 is executed.

The summation operator was shown to satisfy a number of useful properties. The one of interest to us is that it preserves refinement, allowing us to refine elements of the summation separately:

$$S_1 \sqsubseteq S_1' \ \wedge \ S_2 \sqsubseteq S_2' \ \Rightarrow \ (S_1 + S_2) \sqsubseteq (S_1' + S_2')$$

Product Types and Operators. The cartesian product of two types Σ and Γ is written $\Sigma \times \Gamma$. The product operator combines predicate transformers by forming the cartesian product of their state spaces. For $S_1 : \Sigma_1 \mapsto \Gamma_1$ and $S_2 : \Sigma_2 \mapsto \Gamma_2$, their product $S_1 \times S_2$ is a predicate transformer of type $\Sigma_1 \times \Sigma_2 \mapsto \Gamma_1 \times \Gamma_2$ whose execution has the same effect as simultaneous execution of S_1 and S_2.

In addition to many other useful properties, the product operator preserves refinement:

$$S_1 \sqsubseteq S_1' \ \wedge \ S_2 \sqsubseteq S_2' \ \Rightarrow \ (S_1 \times S_2) \sqsubseteq (S_1' \times S_2')$$

For $S : \Sigma \mapsto \Sigma$ we define lifting to a product predicate transformer of type $\Sigma \times \Gamma \mapsto \Sigma \times \Gamma$ as $S \times \mathbf{skip}_\Gamma$. When lifting is obvious from the context, we will simply write S instead of $S \times \mathbf{skip}_\Gamma$.

A product $P \times Q$ of two relations $P : \Sigma_1 \leftrightarrow \Gamma_1$ and $Q : \Sigma_2 \leftrightarrow \Gamma_2$ is a relation of type $(\Sigma_1 \times \Sigma_2) \leftrightarrow (\Gamma_1 \times \Gamma_2)$ defined by

$$(P \times Q)\ (\sigma_1, \sigma_2)(\gamma_1, \gamma_2) \ \widehat{=} \ (P\ \sigma_1\ \gamma_1) \wedge (Q\ \sigma_2\ \gamma_2)$$

3 Specifying Objects and Classes

Object-oriented systems are characterized by *objects*, which group together data, and operations for manipulating that data. The operations, called *methods*, can be invoked only by sending *messages* to the object. The complete set of messages that the object understands is characterized by the *interface* of the object. The interface represents the signatures of object methods, i.e. the name and the types of input and output parameters. As opposed to the interface, the *object type* is the type of object *attributes*. We consider all attributes as private or hidden, and all methods as public or visible to clients of the object. Accordingly, two objects with the same public part, i.e. the same interface, can differ in their private part, i.e. object types.

We focus on modeling class-based object-oriented languages, which form the mainstream of object-oriented programming. Accordingly, we take a view that objects are instantiated by classes. A class is a pattern used to describe objects with identical behaviour through specifying their interface. Specifically, a class describes what attributes each object will have, the specification for each method, and the way the objects are created. We declare a class as follows:

$$
\begin{array}{l}
C \;=\; \textbf{class} \\
\quad attr_1 : \Sigma_1, \ldots, attr_m : \Sigma_m \\
\quad C\ (p : \Psi) \;=\; S, \\
\quad Meth_1\ (g_1 : \Gamma_1) : \Delta_1 \;=\; T_1, \\
\quad \ldots \\
\quad Meth_n\ (g_n : \Gamma_n) : \Delta_n \;=\; T_n \\
\textbf{end}
\end{array}
$$

Class attributes $(attr_1, \ldots, attr_m)$ abbreviated further on as *attr* have the corresponding types Σ_1 through Σ_m. The type of *attr* is then $\Sigma \;=\; \Sigma_1 \times \ldots \times \Sigma_m$[4]. A *class constructor* is used to instantiate objects and is distinguished by the same name as the class. Due to the fact that the constructor concerns object creation rather than object functionality, it is associated with the class rather than with the specified interface. We take a view that the constructor signature is not part of the interface specified by the class. The statement $S : \Xi(\Sigma \times \Psi)$ representing a body of the constructor initializes the attributes using input $p : \Psi$.

Methods $Meth_1$ through $Meth_n$ specified by bodies $T_1, \ldots, T_n$ operate on the attributes and realize the object functionality. Every statement T_i is, in general, of type $\Xi(\Sigma \times \Gamma_i \times \Delta_i)$, where Σ is the type of class attributes, Γ_i and Δ_i are the types of input and output parameters respectively. A method may be parameterless with both Γ_i and Δ_i the unit type (), have only input or only output parameters. When a method has an output parameter, a special variable $res : \Delta_i$ represents the result and assignment to this variable models returning a value in the output parameter. The signature of every method is part of the specified interface.

The object type specified by a class can always be extracted from the class and we do not need to declare it explicitly. We use $\tau(C)$ to denote the type of objects generated by the class C. Naturally, $\tau(C)$ is just another name for Σ.

Being declared as such, the class C is modeled by a tuple $(K, M_1, \ldots, M_n)$, where

$$
\begin{aligned}
K &= [Enter_{attr}]; S; [Exit_p] \\
M_i &= [Enter_{res}]; T_i; [Exit_{g_i}], \text{ for i = 1,\ldots, n.}
\end{aligned}
$$

Further on we will refer to K as the constructor and to $M_1, \ldots, M_n$ as the methods, unless stated otherwise.

Instantiating a new variable of object type by class C is modeled by invoking the corresponding class constructor:

$$c.C(e) \;\widehat{=}\; [Enter_p]; p := e; K; c := attr; [Exit_{attr}]$$

Naturally, a variable of object type can be local to a block:

$$|[\ \textbf{var}\ c : C(e)\ \bullet\ S\ ]| \;\widehat{=}\; [Enter_c]; c.C(e); S; [Exit_c]$$

[4] We impose a non-recursiveness restriction on Σ so that none of Σ_i is equal to Σ. This restriction allows us to stay within the simple-typed lambda calculus.

Often a class aggregates objects of another class, i.e. some attributes can be of object types. In this case the class declaration states the object types of these attributes, but only the constructor invocation actually introduces new objects into the state space and initializes them.

Invocation of a method $Meth_i\ (g_i : \Gamma_i) : \Delta_i$ on an object c instantiated by class C is modeled as follows:

$$\begin{array}{lcl} d := c.Meth_i\ (g) & \widehat{=} & [Enter_{attr}]; [Enter_{g_i}]; \\ & & attr := c; g_i := g; M_i; c := attr; d := res; \\ & & [Exit_{res}]; [Exit_{attr}] \end{array}$$

As an example of a class specification consider a class of bank accounts. An account should have an owner, and it should be possible to deposit and withdraw money in the currency of choice and check the current balance. We present the specification of the class *Account* in Fig. 1.

$Account\ =$ **class**
  $owner : Name, balance : Currency$

  $Account\ (name : Name, sum : Currency)\ =\ owner := name; balance := sum,$

  $Deposit\ (sum : Currency, from : Name, when : Date)\ =$
    $\{sum > 0\}; balance := balance + sum,$

  $Withdraw\ (sum : Currency, to : Name, when : Date)\ =$
    $\{sum > 0\ \wedge\ sum \leq balance\}; balance := balance - sum,$

  $Owner\ () : Name\ =\ res := owner,$

  $Balance\ () : Currency\ =\ res := balance$
end

Fig. 1. Specification of bank account

Obviously, this specification only demonstrates the most general behaviour of bank accounts. For example, when specifying *Deposit*, we only state that *balance* is increased by *sum* and leave the changes to the other input parameters unspecified. We would like to *subclass* from *Account* more concrete account classes. Let us consider specification of subclasses more closely.

3.1 Subclassing

Subclassing [5] is a mechanism for constructing new classes from existing ones by *inheriting* some or all of their attributes and methods, possibly *overriding* some

[5] We prefer the term *subclassing* to *implementation inheritance* because the latter literally means reuse of existing methods and does not, as such, suggest the possibility of method overriding.

attributes and methods, and adding extra methods. We limit our consideration of class construction to inheritance and overriding. Addition of extra methods is a non-trivial issue because of inconsistencies possibly introduced by extra methods which become apparent in presence of subtype aliasing, and is treated in another study.

We describe a subclass of class C as follows:

$$\begin{array}{l} C' \;=\; \textbf{subclass of } C \\ \quad attr'_1 : \Sigma'_1, \ldots, attr'_p : \Sigma'_p \\ \quad C'\ (p : \Psi) \;=\; S', \\ \quad Meth_1\ (g_1 : \Gamma_1) : \Delta_1 \;=\; T'_1, \\ \quad \ldots \\ \quad Meth_k\ (g_k : \Gamma_k) : \Delta_k \;=\; T'_k \\ \textbf{end} \end{array}$$

Class attributes $attr'_1, \ldots, attr'_p$ have the corresponding types Σ'_1 through Σ'_p. Some of these attributes are inherited from the superclass C, others override attributes of C, and the other ones are new. The class C' has its own class constructor without inheriting the one associated with the superclass. The bodies $T'_1, \ldots, T'_k$ override the corresponding $Meth_1, \ldots, Meth_k$ body definitions defined in C. The bodies of methods named $Meth_{k+1}, \ldots, Meth_n$ are inherited from the superclass C. The class C' is modeled by a tuple $(K', M'_1, \ldots, M'_n)$, where the statements K' and all M'_i are related to $S', T'_1, \ldots, T'_n$ as described above.

We view subclassing as a syntactic relation on classes, since subclasses are distinguished by an appropriate declaration. Syntactic subclassing implies conformance of interfaces, in the sense that a subclass specifies an interface conforming to the one specified by its superclass. In the simple case the interface specified by a subclass is the same as that of the superclass. In the next section we explain how this requirement can be relaxed.

As an example of subclassing consider extending the class *Account* with a list of transactions, where every transaction has a sender, a receiver, an amount of money being transferred, and a date. We specify a record type representing transactions as follows:

$$\begin{array}{l} \textbf{type } Transaction \;=\; \textbf{record} \\ \quad from : Name,\ to : Name,\ amount : Currency,\ date : Date \\ \textbf{end} \end{array}$$

Here $Name$, $Currency$ and $Date$ are simple types. $Date$ is a type of six digit arrays for representing a day, a month, and a year, for example as '251296' for December 25, 1996.

Now we can specify in Fig. 2 a class of bank accounts based on sequences of transactions. Notice that we specify only the overriding methods, *Owner* and *Balance* are inherited from the superclass *Account*.

$$
\begin{array}{l}
AccountPlus \;=\; \textbf{subclass of}\;\; Account \\
\quad owner : Name, balance : Currency, transactions : \;\textbf{seq of}\; Transaction \\
\\
\quad AccountPlus\;(name : Name, sum : Currency) \;= \\
\quad\quad owner := name; balance := sum; transactions := \langle\rangle, \\
\\
\quad Deposit\;(sum : Currency, from : Name, when : Date) \;= \\
\quad\quad \{sum > 0\}; |[\;\textbf{var}\;\; t : Transaction \;\bullet\; t := (from, owner, sum, when); \\
\quad\quad transactions := transactions^\frown\langle t\rangle; balance := balance + sum\;]|, \\
\\
\quad Withdraw\;(sum : Currency, to : Name, when : Date) \;= \\
\quad\quad \{sum > 0 \;\wedge\; sum \leq balance\}; \\
\quad\quad |[\;\textbf{var}\;\; t : Transaction \;\bullet\; t := (owner, to, sum * (-1), when); \\
\quad\quad transactions := transactions^\frown\langle t\rangle; balance := balance - sum\;]| \\
\textbf{end}
\end{array}
$$

Fig. 2. Specification of account based on transactions

3.2 Modeling Subtyping Polymorphism

To model subtyping polymorphism, we allow object types to be sum types. The idea is to group together an object type of a certain class and object types of all its subclasses, to form a polymorphic object type. A variable of such a sum type can be instantiated to any base type of the summation, in other words, to any object instantiated by a class whose object type is the base type of the summation. We will call the object types of only one class *ground* and summations of object types *polymorphic*. Since a ground object type uniquely identifies the class of objects, we can always tell whether a certain object is an instance of a certain class.

A sum of object types, denoted by $\tau(C)^+$ is defined to be such that its base types are $\tau(C)$ and all the object types of subclasses of C. For example, if D is the only subclass of C with the object type $\tau(D)$, then $\tau(C)^+ \;=\; \tau(C) \,+\, \tau(D)$. Naturally, we have that

$$\tau(C) <: \tau(C)^+ \text{ and } \tau(D) <: \tau(C)^+.$$

A variable $c : \tau(C)^+$ can be instantiated by either C or D. The *subsumption* property holds of c, namely, if $c : \tau(C)$ and $\tau(C) <: \tau(C)^+$ then $c : \tau(C)^+$. This property is characteristic of subtype relations, it means that an object of type $\tau(C)$ can be viewed as an object of the supertype $\tau(C)^+$.

Suppose a method $Meth_i$ is specified in both C and D by statements M_i and M_i' respectively. An invocation of $Meth_i$ on an object c of type $\tau(C)^+$ is modeled as follows:

$$c.Meth_i() \;\;\widehat{=}\;\; \left(\begin{array}{l}[Enter_{attr}]; \\ attr := c; M_i; c := attr; \\ [Exit_{attr}]\end{array}\right) + \left(\begin{array}{l}[Enter_{attr'}]; \\ attr' := c; M_i'; c := attr'; \\ [Exit_{attr'}]\end{array}\right)$$

where $attr : \Sigma$ and $attr' : \Sigma'$ are attributes of C and D respectively. Modeling an invocation of a method having input and output parameters is similar to method invocation on a non-polymorphic object.

Being equipped with subtyping polymorphism, we can allow overriding methods in a subclass to be generalized on the type of input parameters or specialized on the type of output parameters. In the first case this type redefinition is *contravariant* and in the second *covariant*[6]. When one interface is the same as the other, except that it can redefine contravariantly input parameter types and covariantly output parameter types, this interface conforms to the original one.

As an example of using polymorphic object types let us consider a client of the classes *Account* and *AccountPlus*, a bank which maintains a sequence of accounts and can transfer money from one account to another. The specification of the class *Bank* is presented in Fig. 3.

$Bank\ =$ **class**
    $accounts:$ **seq of** $\tau(Account)$

    $Transfer\ (from : \tau(Account), to : \tau(Account), s : Currency, d : Date)\ =$
        $\{sum > 0\};$
        $|[$ **var** $sender, receiver : Name\ \bullet$
        $sender := from.Owner(); receiver := to.Owner();$
        $from.Withdraw(s, receiver, d); to.Deposit(s, sender, d)\ ]|$
end

Fig. 3. Specification of bank using accounts

A subclass of *Bank* can redefine the method $Transfer$ with input parameters of types $\tau(Account)^+$ to meet the contravariant constraint. The new bank will be able to work with both *Account* and *AccountPlus* instances in this case, provided that the *accounts* attribute is redefined to be of type **seq of** $\tau(Account)^+$.

4 Class Refinement

When a subclass overrides some methods of its superclass, there are no guarantees that its instances will deliver the same or refined behaviour as the instances of the superclass. Unrestricted method overriding in a subclass can lead to an arbitrary behaviour of its instances. When used in a superclass context, such subclass instances may invalidate their clients. For example, the *Deposit* method of *Account* can be overridden so that the money is, in fact, withdrawn from the account instead of being deposited. Then the owner of the account will actually be at a loss.

Therefore, we would like to ensure that whenever C' is subclassed from C, any behaviour expected from C will necessarily be delivered by C'. For this purpose, we introduce the notion of class refinement between C and C'.

Consider two classes $C = (K, M_1, \ldots, M_n)$ and $C' = (K', M'_1, \ldots, M'_n)$ such that $K : \Psi \ \mapsto\ \Sigma$ and $K' : \Psi' \ \mapsto\ \Sigma'$ are the corresponding class constructors,

[6] For a more extensive explanation of covariance and contravariance see, e.g. [1].

and all $M_i : \Sigma \times \Gamma_i \mapsto \Sigma \times \Delta_i$ and $M'_i : \Sigma' \times \Gamma'_i \mapsto \Sigma' \times \Delta'_i$ are the corresponding methods. The input parameter types of the constructors and the methods are either the same or contravariant, such that $\Psi <: \Psi'$ and $\Gamma_i <: \Gamma'_i$. The output parameter types of the methods are either the same or covariant, $\Delta'_i <: \Delta_i$.

We define the refinement of class constructors K and K' with respect to a relation R as follows:

$$K \sqsubseteq_R K' \quad \widehat{=} \quad \{\pi_\Psi\};\ K \quad \sqsubseteq \quad K';\ \{R\} \tag{1}$$

where $R : \Sigma' \leftrightarrow \Sigma$ is an abstraction relation coercing attribute types of C' to those of C, and π_Ψ is the projection relation coercing Ψ' to Ψ.

The refinement of all corresponding methods M_i and M'_i with respect to the relation R is defined as

$$M_i \sqsubseteq_R M'_i \quad \widehat{=} \quad \{R \times \pi_{\Gamma_i}\};\ M_i \quad \sqsubseteq \quad M'_i;\ \{R \times \iota_{\Delta'_i}\} \tag{2}$$

Here R is as above, $\pi_{\Gamma_i} : \Gamma'_i \leftrightarrow \Gamma_i$ projects the corresponding input parameters, and $\iota_{\Delta'_i} : \Delta'_i \leftrightarrow \Delta_i$ injects the corresponding output parameters. Obviously, when $\Gamma_i = \Gamma'_i$, the projection relation π_{Γ_i} is taken to be the identity relation Id. The same holds when $\Delta_i = \Delta'_i$, namely, $\iota_{\Delta'_i} = Id$.

Now we can define the class refinement relation as follows.

Definition 1 (Class refinement). The class C is refined by the class C', written $C \sqsubseteq C'$, if for some abstraction relation $R : \tau(C') \leftrightarrow \tau(C)$

1. The constructor of C' refines the constructor of C as defined in (1)
2. Every method of C' refines the corresponding method of C as defined in (2).

The class refinement relation shares the properties of statement refinement and is, thus, reflexive and transitive.

Theorem 2. *Let C, C' and C'' be classes. Then the following properties hold:*

1. $C \sqsubseteq C$
2. $C \sqsubseteq C' \ \wedge\ C' \sqsubseteq C'' \quad \Rightarrow \quad C \sqsubseteq C''$

Declaring one class as a subclass of another raises the proof obligation that the class refinement relation holds between these classes. This is a semantic constraint that we impose on subclassing to ensure that behaviour of subclasses conforms to the behaviour of their superclasses and, respectively, that the subclasses can be used in the superclass context.

As an example of class refinement consider the classes *Account* and *AccountPlus*. Since the latter is declared as a subclass of the former, we get a proof obligation $Account \ \sqsubseteq\ AccountPlus$. Under the abstraction relation $R\ (o', b', t')(o, b) \ = \ (o' = o) \wedge (b' = b)$, where o, b correspond to $owner, balance$ of *Account* and o', b', t' correspond to $owner, balance, transactions$ of *AccountPlus*, this proof obligation can be discharged, but we omit the proof for the lack of space.

5 Interface Refinement

Subclassing requires that parameter types of a method be the same in the subclass and in the superclass or, at most, subject to contravariance and covariance rules. However, sometimes, a change in user requirements causes interface changes of classes designed as refinements of other classes.

When the new interface is similar to the old one, we can identify abstraction relations coercing the new method parameters to the old ones. For every pair of corresponding methods we need to find two such relations, for input and output parameters. The rôle of these parameter abstraction relations is crucial for interface refinement of classes and for refinement of their clients. Let us first define the interface refinement relation between classes with respect to these relations.

Consider two classes $C = (K, M_1, \ldots, M_n)$ and $C' = (K', M_1', \ldots, M_n')$ with attribute types Σ and Σ' respectively, such that $K : \Psi \mapsto \Sigma$ and $K' : \Psi' \mapsto \Sigma'$ are the class constructors, and all $M_i : \Sigma \times \Gamma_i \mapsto \Sigma \times \Delta_i$ and $M_i' : \Sigma' \times \Gamma_i' \mapsto \Sigma' \times \Delta_i'$ are the corresponding methods.

Let $R : \Sigma' \leftrightarrow \Sigma$ be an abstraction relation coercing attribute types of C' to those of C, and $I_0 : \Psi' \leftrightarrow \Psi$ an abstraction relation coercing the corresponding input parameter types. We define the refinement of class constructors K and K' through R and I_0 as follows:

$$K \sqsubseteq_{R,I_0} K' \quad = \quad \{I_0\};\ K \quad \sqsubseteq \quad K';\ \{R\} \tag{3}$$

Obviously, (3) is a generalization of (1) with $I_0 = \pi_\Psi$ when the input types are contravariant.

Let $R : \Sigma' \leftrightarrow \Sigma$ be as before, and $I_i : \Gamma_i' \leftrightarrow \Gamma_i$ and $O_i : \Delta_i' \leftrightarrow \Delta_i$ be abstraction relations coercing the corresponding input and output parameter types. We define the refinement of corresponding methods M_i and M_i' through R, I_i and O_i as follows:

$$M_i \sqsubseteq_{R,I_i,O_i} M_i' \quad = \quad \{R \times I_i\};\ M_i \quad \sqsubseteq \quad M_i';\ \{R \times O_i\} \tag{4}$$

Obviously, (4) is a generalization of (2) with $I_i = \pi_{\Gamma_i}$ when the inputs are contravariant, i.e. $\Gamma_i <: \Gamma_i'$, and with $O_i = \iota_{\Delta_i'}$ when the outputs are covariant, i.e. $\Delta_i' <: \Delta_i$.

Definition 3 (Interface refinement). The class C is interface refined by the class C', written $C \sqsubseteq_{I,O} C'$, with respect to parameter abstraction relations $I = (I_0, I_1, \ldots, I_n)$ and $O = (O_1, \ldots, O_n)$ if for some abstraction relation $R : \tau(C') \leftrightarrow \tau(C)$

1. The constructor of C' refines the constructor of C as defined in (3)
2. Every method of C' refines the corresponding method of C as defined in (4).

Being defined as such, interface refinement of classes is a generalization of class refinement. When every I_i and O_i is the identity relation or the projection and injection relations respectively, interface refinement is specialized to class refinement. The interface refinement relation has the basic properties required of a refinement relation, i.e. reflexivity and transitivity.

Theorem 4. *Let C, C' and C'' be classes. Then the following properties hold:*

1. $C \sqsubseteq_{Id,Id} C$
2. $C \sqsubseteq_{I,O} C' \ \wedge\ C' \sqsubseteq_{I',O'} C'' \ \Rightarrow\ C \sqsubseteq_{I';I,O';O} C''$

where the relational compositions $I';I$ and $O';O$ on tuples of relations are taken elementwise.

Proof. The proof of (1) follows directly from reflexivity of statement refinement by taking the abstraction relation R to be Id. To prove (2) we assume that $C \sqsubseteq_{I,O} C'$ and $C' \sqsubseteq_{I',O'} C''$ hold for abstraction relations R and R' respectively. We then show that methods M_i, M_i' and M_i'' of the corresponding classes C, C' and C'' have the property:

$$\begin{aligned}&\{R \times I_i\}; M_i \ \sqsubseteq\ M_i'; \{R \times O_i\} \ \wedge\ \{R' \times I_i'\}; M_i' \ \sqsubseteq\ M_i''; \{R' \times O_i'\} \quad \Rightarrow \\ &\qquad \{(R';R) \times (I_i';I_i)\}; M_i \ \sqsubseteq\ M_i''; \{(R';R) \times (O_i';O_i)\}\end{aligned}$$

The proof of the property is as follows:

$$\begin{aligned}
& \{(R';R) \times (I_i';I_i)\}; M_i \ \sqsubseteq\ M_i''; \{(R';R) \times (O_i';O_i)\} \\
= \ & \text{lemma } (P;P') \times (Q;Q') \ = \ (P \times Q); (P' \times Q') \\
& \{(R' \times I_i'); (R \times I_i)\}; M_i \ \sqsubseteq\ M_i''; \{(R' \times O_i'); (R \times O_i)\} \\
= \ & \text{homomorphism of angelic update statement } \{P\};\{Q\} \ = \ \{P;Q\} \\
& \{R' \times I_i'\}; \{R \times I_i\}; M_i \ \sqsubseteq\ M_i''; \{R' \times O_i'\}; \{R \times O_i\} \\
\Leftarrow \ & \text{assumption } \{R \times I_i\}; M_i \ \sqsubseteq\ M_i'; \{R \times O_i\} \\
& \{R' \times I_i'\}; M_i'; \{R \times O_i\} \ \sqsubseteq\ M_i''; \{R' \times O_i'\}; \{R \times O_i\} \\
\Leftarrow \ & \text{assumption } \{R' \times I_i'\}; M_i' \ \sqsubseteq\ M_i''; \{R' \times O_i'\} \\
& M_i''; \{R' \times O_i'\}; \{R \times O_i\} \ \sqsubseteq\ M_i''; \{R' \times O_i'\}; \{R \times O_i\} \\
= \ & \text{reflexivity of statement refinement} \\
& \mathit{true}
\end{aligned}$$

The proof of the corresponding property for constructors is similar. □

Theorem 2 follows by specializing I and O appropriately.

As an example of interface refinement consider our previous specification of transactions, accounts and banks. Suppose that facing the start of the new century, we'd like to change the type of dates so that it's possible to specify a four-digit year:

$$\textbf{type}\ \ \mathit{NewDate} = \ \textbf{array}\ [1..8]\ \textbf{of}\ \mathit{Digit}$$

Accordingly, we define a new transaction record type $NewTran$ which is the same as $Transaction$ except that the $date$ field is now of type $NewDate$. We construct a new class of accounts using $NewTran$ transactions as shown in Fig. 4. We omit specifications of $Owner$ and $Balance$ methods which are straightforward, and a specification of $Withdraw$, which is similar to that of $Account$ with a local variable of type $NewTran$ rather than $Transaction$.

$NDAccount \ = \ \mathbf{class}$
$\quad owner : Name, balance : Currency, transactions : \ \mathbf{seq\ of}\ NewTran$

$\quad NDAccount\ (name : Name, initSum : Currency) \ =$
$\quad\quad owner := name; balance := initSum; transactions := \langle\rangle,$

$\quad Deposit\ (sum : Currency, from : Name, when : NewDate) \ =$
$\quad\quad \{sum > 0\}; |[\ \mathbf{var}\ \ t : NewTran \ \bullet \ t := (from, owner, sum, when);$
$\quad\quad transactions := transactions ^\frown \langle t\rangle; balance := balance + sum \ ]|,$
$\quad \ldots$
$\mathbf{end}$

Fig. 4. Specification of account based on NewTran

It can be shown that $AccountPlus \sqsubseteq_{I,O} NDAccount$, where $I = (Id \times Id, Id \times Id \times D, Id \times Id \times D, Id, Id)$ and $O = (Id, Id, Id, Id)$. The abstraction relation $D : NewDate \leftrightarrow Date$ is defined so that for constants '1' and '9' of type $Digit$ and for any $d : Date$ and $d' : NewDate$:

$$D(d')(d) \quad = \quad (d'[1..4] = d[1..4]) \wedge (d'[5..6] = \text{`1'`9'}) \wedge (d'[7..8] = d[5..6])$$

Now let us consider how parameter abstraction relations can be used for refinement of clients of the interface refined classes. Interface changes in class methods certainly affect clients of the class. Examining the ways the clients get affected allows us to discover the situations when the clients can benefit from the interface refinement of their server classes but need not be changed in any way. We can also establish conditions under which the clients can be systematically changed to use the refined server classes. For every $OldClass$ and $NewClass$, such that $NewClass$ is designed as a refinement of $OldClass$ but specifies a different interface, we distinguish two ways clients of $OldClass$ can be affected and changed.

5.1 Implicit Client Refinement

This kind of client refinement happens when it is impractical or impossible to redefine clients of $OldClass$, but is, however, desirable that they work with $NewClass$ which may offer a more efficient implementation or improved functionality to new clients, like in our example. We can implicitly refine clients by employing a so-called *forwarding scheme* illustrated in Fig. 5 using the OMT notation [23]. In this diagram the link with a triangle relates a superclass with a subclass with the superclass above. The link with a diamond shows an aggregation relation, i.e. that $Wrapper$ aggregates an instance of $NewClass$ in the attribute $impl$.

The idea behind such kind of forwarding is to introduce a subclass of $OldClass$, $Wrapper$, which aggregates an instance of $NewClass$ and forwards $OldClass$ method calls to $NewClass$ through this instance. This has also been identified as a reoccuring design pattern by Gamma et al. in [11]. Clients of

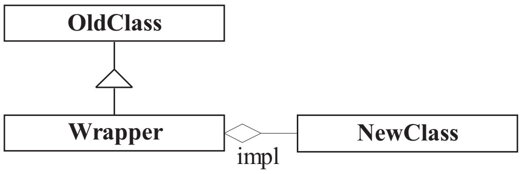

Fig. 5. Illustration of forwarding

$OldClass$ can work with $Wrapper$, which is a subclass of $OldClass$, but have all the benefits of working with $NewClass$ if

$$OldClass \sqsubseteq Wrapper \text{ and } Wrapper \sqsubseteq_{I,O} NewClass$$

Consider again our example. The client *Bank* wants to use *NDAccount* but cannot do so since the latter specifies the interface different from that specified by *AccountPlus*. We can employ the forwarding scheme by introducing in Fig. 6 a new class *AccountWrapper* which aggregates an instance of *NDAccount* and forwards *AccountPlus* method calls to *NDAccount* via this instance. Specifications of $Withdraw$ and *Balance* are straightforward and we omit them for brevity. The function $ToNewDate$ $(old : Date) : NewDate$ converts dates from the old format to the new one. In fact, this function can be modeled by the statement $[D^{-1}]$, where $D : NewDate \leftrightarrow Date$ is as before. Provided that the necessary proof obligations are discharged, clients of *AccountPlus*, such as *Bank*, are implicitly refined to work with *NDAccount* via *AccountWrapper*.

Since wrapper classes are of a very specific form, proof obligations can be considerably simplified. Consider a typical wrapper class as given in Fig. 7. The demonic specification statements transform the input parameters of *OldClass* to the input parameters of $NewClass$ using the corresponding parameter abstraction relations $I_i, i = 0, \ldots, n$. Similarly, the angelic specification statements transform the output parameters of $NewClass$ back to the output parameters

---

$AccountWrapper$ = **subclass of** *AccountPlus*
  $impl : \tau(NDAccount)$

  $AccountWrapper\ (name : Name, initSum : Currency)$ =
    $impl.NDAccount(name, initSum)$,

  $Deposit\ (sum : Currency, from : Name, when : Date)$ =
    $\{sum > 0\}; |[$ **var** $d : NewDate$ $\bullet$
    $d := ToNewDate(when); impl.Deposit(sum, from, d)\ ]|$,

  $Owner\ () : Name\ =\ res := impl.Owner()$,
  $\ldots$
end

Fig. 6. Specification of wrapper class for implicit interface refinement

$$
\begin{array}{l}
Wrapper \ = \ \ \mathbf{subclass\ of}\ OldClass \\
\quad impl : \tau(NewClass) \\
\quad Wrapper\ (p : \Psi)\ =\ |[\ \mathbf{var}\ e : \Psi' \ \bullet\ [e := e' \cdot\ I_0^{-1}\ p\ e']; impl.NewClass(e)\ ]|, \\
\quad Meth_i(g_i : \Gamma_i) : \Delta_i\ = \\
\qquad |[\ \mathbf{var}\ c_i : \Gamma_i', d_i : \Delta_i'\ \bullet\ [c_i := c_i' \cdot\ I_i^{-1}\ g_i\ c_i']; \\
\qquad d_i := impl.Meth_i(c_i); \{res := res' \cdot\ O_i\ d_i\ res'\}\ ]|, \\
\quad \ldots \\
\mathbf{end}
\end{array}
$$

Fig. 7. Schema of wrapper class for implicit interface refinement

of *OldClass*. For the class *Wrapper* with such a structure, we have the following theorem.

Theorem 5. *For parameter abstraction relations* $I = (I_0, I_1, \ldots, I_n)$ *and* $O = (O_1, \ldots, O_n)$ *the following property holds:*

$$OldClass \sqsubseteq_{I,O} NewClass \ \Rightarrow\ OldClass \sqsubseteq Wrapper$$

The form of specification statements gives insight into suitable restrictions when choosing the parameter abstraction relations I_i and O_i. If I_i^{-1} is partial, then the corresponding specification statement can be **magic** and, thus, is not implementable. Hence, I_i has to be surjective, i.e. relate all possible values of the old input parameters to some values of the new input parameters. Likewise, if O_i is non-deterministic (not functional), then the result $res : \Delta_i$ is chosen angelically, and is, therefore, not implementable. Hence O_i must be deterministic (functional), i.e. relate values of the new result parameters $d_i : \Delta_i'$ to at most one value of the old result parameters $res : \Delta_i$.

5.2 Explicit Client Refinement

This kind of client refinement happens quite often in the process of object-oriented development. After *NewClass* has been developed, using *OldClass* may become impractical and undesirable, and therefore, a client *OldClient* of *OldClass* should be explicitly changed to work with *NewClass* instead. We can construct *NewClient* by refinement from *OldClient*. Unfortunately, there are no guarantees that the interface of *NewClient* will conform to that of *OldClient*. Accordingly, we must consider two cases, when *NewClient* is a subclass of *OldClient* and when it is its interface refinement.

When the object type $\tau(OldClass)$ and the types causing the interface change of *OldClass* to *NewClass* are not part of *OldClient* interface, the refinement of *OldClient*, *NewClient*, can be its subclass. In other words, *NewClient* can specify the interface conforming to that of *OldClient*. Naturally, every class using *OldClient* can then use *NewClient* instead and is implicitly refined without respecification.

We feel that there is a strong connection between parameter abstraction relations with respect to which interface refinement is defined and explicit refinement of clients of the refined classes. Investigating how clients can be explicitly refined based on the parameter abstraction relations for the server classes remains the topic of current research.

6 Conclusions

Our approach is suited for documenting, constructing, and verifying different kinds of object-oriented systems because of its uniform way of specifying a program at different abstraction levels and the possibility of stepwise development. We have defined the class refinement relation and the interface refinement relation which allow a developer to construct extensible object-oriented programs from specifications and assure reliability of the final program.

Our model of classes, subclassing, and subtyping polymorphism can be used to reason about the meaning of programs constructed using the separate subclassing and interface inheritance hierarchies, like in Java [13], Sather [26], and some other languages. In that approach interface inheritance is the basis for subtyping polymorphism, whereas subclassing is used only for implementation reuse. By associating a specification class with every interface type, we can reason about the behaviour of objects having this interface. All classes claiming to implement a certain interface must refine its specification class. Subclassing, on the other hand, does not, in general, require establishing class refinement between the superclass and the subclass.

For simplicity we consider only single inheritance, but multiple inheritance does not introduce much complication. With a suitable mechanism for resolving clashes in method names, multiple inheritance has the same semantics as we give for single inheritance. Namely, ensuring that a subclass D preserves behaviour of all its declared superclasses $C_1, \ldots, C_n$ requires proving class refinements $C_1 \sqsubseteq D, \ldots, C_n \sqsubseteq D$ for every corresponding superclass-subclass pair.

Using formal specification and verification is especially important for open systems, such as object-oriented frameworks and component-based systems. Frameworks incorporate a reusable design for a specific class of software and dictate a particular architecture of potential applications. When building an application, the user needs to customize framework classes to specific needs of this application. To do so, he must understand the message flow in the framework and the relationship among the framework classes. The intrinsic feature of open component-based systems is a late integration phase, meaning that components are developed by different manufacturers and then integrated together by their users.

A fine-grained specification can accurately describe the fixed behaviour of classes. In this respect, such a specification is a perfect documentation of a framework or a component, because the user does not have to decipher ambiguous, incomplete, and often outdated verbal descriptions. Neither is it necessary to confront the bulk of source code to gain a complete understanding of the system behaviour. The programming notation we use allows the developer to abstract from implementation details and specify classes with abstract state space and non-deterministic behaviour of methods, expressing only the necessary functionality. Moreover, a certain implementation can be a commercial secret, whereas a concise and complete specification distributed instead of source code enables the user to understand the functionality and protects corporate interests.

Formal verification in the form of establishing a class refinement relation between specifications and their implementations guarantees that any behaviour expected from the specifications will be delivered by the implementations.

It has been acknowledged that frameworks are usually developed using a spiral model that takes feedback from actual use of the framework into account. It can be expected that such development iterations may result in an interface change of some classes. In this case, interface refinement can be used to verify behavioural compatibility of the corresponding classes and the rules for interface refinement of clients can be used to refine the whole framework.

Acknowledgments

We would like to thank Ralph Back for a number of fruitful discussions and Martin Büchi for useful comments.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
2. M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In *Theory and Practice of Software Development: Proceedings / TAPSOFT '97*, volume LNCS 1214, pages 682–696. Springer, April 1997.
3. P. America. Designing an object-oriented programming language with behavioral subtyping. In J.W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop*, volume LNCS 489, pages 60–90, New York, N.Y., 1991. Springer-Verlag.
4. R. J. R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Center Tracts*. Mathematical Centre, Amsterdam, 1980.
5. R. J. R. Back. Changing data representation in the refinement calculus. In *21st Hawaii International Conference on System Sciences*. IEEE, January 1989.
6. R. J. R. Back and J. von Wright. Refinement calculus I: Sequential nondeterministic programs. In W. P. deRoever J. W. deBakker and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, pages 42–66. Springer-Verlag, 1990.
7. R.J.R. Back and M.J. Butler. Exploring summation and product operators in the refinement calculus. In B. Möller, editor, *Mathematics of Program Construction, 1995*, volume LNCS 947. Springer-Verlag, 1995.
8. M. Broy. (Inter-)Action Refinement: The Easy Way. In M. Broy, editor, *Program Design Calculi*, pages 121–158, Berlin Heidelberg, 1993. Springer-Verlag.
9. L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

10. K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering*, pages 258–267, Berlin, Germany, 1996.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.
12. P.H. Gardiner and C.C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87(1):143–162, 1991.
13. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Sun Microsystems, Mountain View, 1996.
14. J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In B. Robinet and R. Wilhelm, editors, *European Symposium on Programming*, volume LNCS 213. Springer-Verlag, 1986.
15. C. A. R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1(4):271–281, 1972.
16. K. Lano and H. Haughton. Reasoning and refinement in object-oriented specification languages. In O. Lehrmann Madsen, editor, *European Conference on Object-Oriented Programming '92*, volume LNCS 615. Springer-Verlag, 1992.
17. K. Lano and H. Haughton. *Object-Oriented Specification Case Studies*. Prentice–Hall, New York, 1994.
18. B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.
19. C. C. Morgan. Data refinement by miracles. *Information Processing Letters*, 26:243–246, 1988.
20. C. C. Morgan. *Programming from Specifications*. Prentice–Hall, 1990.
21. J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.
22. D. A. Naumann. Predicate transformer semantics of an Oberon-like language. In Ernst-R. Olderog, editor, *Programming Concepts, Methods and Calculi*, pages 460–480. International Federation for Information Processing, 1994.
23. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design*. Prentice Hall, Englewood Cliffs, 1991.
24. E. Sekerinski. *Verfeinerung in der Objektorientierten Programmkonstruktion*. Dissertation, Universität Karlsruhe, 1994.
25. E. Sekerinski. A type-theoretic basis for an object-oriented refinement calculus. In S.J. Goldsack and S.J.H. Kent, editors, *Formal Methods and Object Technology*. Springer-Verlag, 1996.
26. C. A. Szyperski, S. Omohundro, and S. Murer. Engineering a programming language – the type and class system of Sather. In *Proceedings, First Intl Conference on Programming Languages and System Architectures*, volume LNCS 782, Zurich, Switzerland, 1994. Springer.