A Higher Abstraction Level Using First-Class Inheritance Relations

Marko van Dooren and Eric Steegmans

Department of Computer Science, K.U.Leuven
Marko.vanDooren@cs.kuleuven.be
Eric.Steegmans@cs.kuleuven.be

Abstract. Although classes are a fundamental concept in object-oriented programming, a class itself cannot be built using general purpose classes as building blocks in a practical manner. High-level concepts like associations, bounded values, graph structures, and infrastructure for event mechanisms which form the foundation of a class cannot be reused conveniently as components for classes. As a result, they are implemented over and over again.

We raise the abstraction level of the language with a code inheritance relation for reusing general purpose classes as components for other classes. Features like mass renaming, first-class relations, high-level dependencies, component parameters, and indirect inheritance ensure that maximal reuse can be achieved with minimal effort.

A case study shows a reduction of the code between 21% and 36%, while the closest competitor only reduces the size between 3% and 12%.

1 Introduction

Although increasing the reusability of software is one of the main goals of object-oriented software development, an important group of software elements still cannot be reused in a practical manner. These elements are implemented over and over again, resulting in massive code duplication and all its related problems.

A class often consists of application specific functionality written on top of general purpose characteristics like associations, values lying within bounds, lockable values, graph structures, and infrastructure for event listeners. Most of them are well-known high-level concepts which are easy to use during the design phase. But during the implementation phase, these concepts are transformed into low-level code because current reuse mechanisms cannot cope with such reuse in a convenient manner.

Most reuse mechanisms [4,6,7,9,42,32,35,39] differ little from a regular inheritance relation with subtyping and code inheritance. But the requirements for building a class from components differ in important areas from those for creating a subtype. Reusing a class as a building block for another class requires activities such as removing unwanted methods, wiring method dependencies, and especially renaming methods. But for creating subtypes, the first activity is forbidden, the second one is not required, and the third one is required only infrequently. In addition, methods of different building blocks are usually separated even if they have the same definition, while they are usually merged in case of a multiple/repeated subtyping relation.

E. Ernst (Ed.): ECOOP 2007, LNCS 4609, pp. 425–449, 2007.

Reuse mechanisms that focus on composition [26,36] create only shallow compositions; the composition is just the sum of the parts. But a class is more than the sum of its components; it adds application specific code and gives the components an application specific meaning; it creates an abstract data type.

In this paper, we present an inheritance mechanism with two relations. The subtyping relation is used for traditional subtyping inheritance. The component relation allows general purpose characteristics to be encapsulated in classes and be reused conveniently as configurable building blocks for other classes. We analyze the requirements necessary to realize this kind of reuse, and then introduce the required new features. We introduce renaming parameters for mass renaming, and make the inheritance relation first-class for accessing hidden functionality, treating components as separate objects, and resolving method dependencies using high-level component connections. We evaluate the mechanism in a case study, where it is compared to existing approaches. We also created a formal type system and proved the type soundness of the mechanism, but due to space constraints, the formalization is not presented in this paper.

In Section 2, we analyze the requirements for the reuse mechanism, and discuss existing mechanisms. In Section 3, we present the *component relation*, which is used for code inheritance. In Section 4, we present the impact on the subtyping relation. We evaluate the inheritance mechanism in Section 5 with an example and a case study. We discuss related work and future work in Sections 6 and 7, and conclude in Section 8.

2 Requirements Analysis

In this section, we analyze which features are required in order to conveniently reuse general purpose classes as a building block for other class. We use a simple banking application to illustrate the requirements.

In this paper, we illustrate the features of the inheritance mechanism mostly with components for modeling associations, which use a simple protocol to keep the association consistent. The proposed inheritance mechanism, however, can reuse general abstract data types – which can use arbitrarily complex protocols – as components.

Figure 1 illustrates the application. It contains classes for persons, bank accounts, and bank cards. The rectangles inside a class represent its characteristics. For example, an account has a balance, which is a number that lies between the credit limit and an upper bound. In addition, it has a unidirectional association with its account number, and a bidirectional association with its owner. The associations for the parents and children of a person form a graph offering different traversal strategies. Dependencies between characteristics are represented by dashed arrows. For example, the `owner` and `accounts` components need each other's methods to keep the association consistent.

Figure 2 shows a Java implementation of class `BankAccount`. More advanced functionality like sending events, and constraints on the associations is not shown.

The problem with the implementation is that it consists entirely of functionality that has already been implemented millions of times before. Associations and constrained values are common characteristics, and although the exact names of the methods and the used types may differ, the behavior is always the same.

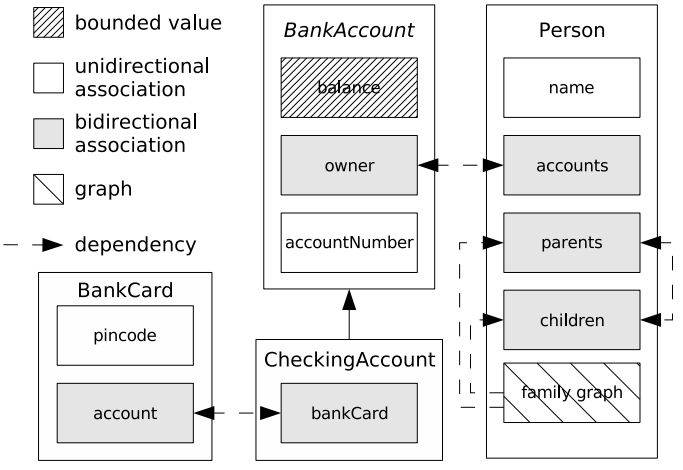

Fig. 1. High-level design of an application

```
class BankAccount {
  public BankAccount(int number) {
    this.creditLimit = -1000;
    this.upperLimit = 1000000;
    this.accountNumber = number;
  }
  private Person owner;
  public Person getOwner() {
    return owner;
  }
  public void setOwner(Person owner) {
    if(this.owner != owner) {
      registerOwner(owner);
      if(owner != null)
        owner.registerAccount(this);
    }
  }
  protected void registerOwner(Person owner) {
    if (this.owner != null)
      this.owner.unregisterAccount();
    this.owner = owner;
  }
  protected void unregisterOwner() {
    owner = null;
  }
  private final int accountNumber;
  public int getAccountNumber() {
    return accountNumber;
  }
  private long balance;
  private long upperLimit;
  private long creditLimit;

  public long getBalance() {
    return balance;
  }
  public void deposit(long amt) {
    if((amt > 0) &&
      (balance<=Long.MAX_VALUE-amt)
      &&(balance + amt <= upperLimit))
            balance += amt;
  }
  public void withdraw(long amt) {
    if((amt > 0) &&
      (balance>=Long.MIN_VALUE+amt)
      &&(balance - amt >= creditLimit))
            balance -= amt;
  }
  public long getUpperLimit() {
    return upperLimit;
  }
  public long getCreditLimit() {
    return creditLimit;
  }
}
```

Fig. 2. The Java version of `BankAccount`

2.1 Requirements

The goal is to construct a reuse mechanism that allows high-level concepts to be encapsulated and reused to build a class. The reusable entity is called a *component*. The mechanism must minimize the effort required to reuse a component, and maximize the reusability of its functionality.

The requirements[1] are illustrated using the example from Figures 1 and 2. They are grouped to increase readability, but some features can be placed in multiple groups. We omit features supported by all mechanisms, such as parameterized types.

Mandatory Features. The following features are mandatory for building a class by reusing components.

1. **ADT Components:** A component must contribute to the abstract data type of the reusing class, which rules out a simple *has-a* relation. Otherwise, the composition is too difficult to use. For example, the methods of `Person` would be spread over several objects at different depths depending on the nesting of the components, and have names that are almost meaningless in the context of the application.
2. **Multiple Reuse:** A class must be able to reuse code from more than one component. For example, class `BankAccount` has three general characteristics.
3. **Repeated Reuse:** Because a class can reuse multiple components of the same kind, it must be able to reuse a component more than once. For example, class `Person` has three bidirectional associations.
4. **Renaming:** Renaming is required to solve name conflicts caused by repeated reuse, give the reused methods a meaningful name in the context of the reusing class, and merge features. Name conflicts will occur because components can be reused more than once by a single class, as for example in class `Person`.

Expressivity Features. These features reduce the amount of work needed to reuse a component. The impact of a feature is shown using big $\mathcal{O}$ notation as $activity : \mathcal{O}_{without} \rightarrow \mathcal{O}_{with}$. It shows the amount of work required for an activity without and with that feature when reusing a component. Note that the activities are not independent of each other. $\mathcal{M}$ is the number of methods in the component, $\mathcal{F}$ the number of fields. $\mathcal{M}_s$ and $\mathcal{F}_s$ are the number of methods and fields exported in the interface of the reusing class, $\mathcal{M}_{ns}$ and $\mathcal{F}_{ns}$ the number of non-exported methods and fields. The required work of some features is explained further on in this paper, and is denoted with '...' for now. Note that $\mathcal{F}_s + \mathcal{F}_{ns} = \mathcal{F}, \mathcal{M}_s + \mathcal{M}_{ns} = \mathcal{M}$, and usually $\mathcal{F}_s \leqslant \mathcal{F} \ll \mathcal{M}_s < \mathcal{M}$.

5. **State Reuse:** *Declaring fields:* $\mathcal{O}(\mathcal{F}) \rightarrow \mathcal{O}(1)$ Reusing the state of a component prevents a lot of duplication. For example, the state of an association is almost always a simple reference. It makes no sense to force a developer to separately provide that state every time he uses an association component.
6. **Interface Reuse:** *Constructing interface:* $\mathcal{O}(\mathcal{M}_s + \mathcal{F}_s) \rightarrow \mathcal{O}(1)$ Reusing the component interface prevents duplication of its signatures. Aside from the exact method names and types, which can be configured using renaming and type parameters, the signatures in the reusing class are the same as those in the component interface.

[1] Many of the requirements are presented in related work under slightly different names.

7. **Selective Interface Reuse:** *Resolving conflicts:* $\mathcal{O}(\mathcal{M} + \mathcal{F}) \rightarrow \mathcal{O}(\mathcal{M}_s + \mathcal{F}_s)$ A developer will usually expose only a part of the component interface based on the intended use of the reusing class. Exposing its entire interface makes the reusing class harder to understand if the component has a lot of functionality. In addition, it can cause a large amount of name conflicts that must be solved even if the involved methods and fields are not relevant in the context of the reusing class.
8. **Powerful Selection:** *Selecting exported methods/fields:* $\mathcal{O}(\mathcal{M}_s + \mathcal{F}_s) \rightarrow \mathcal{O}(\ldots)$ Being able to select which methods and fields are exported in the interface of the reusing class is not enough. If hiding or selecting is done individually for each method, it requires too much work.
9. **Default Separation:** *Separating components:* $\mathcal{O}(\mathcal{M}_{ns} + \mathcal{F}_{ns}) \rightarrow \mathcal{O}(1)$ By default, components – and thus their methods and instance variables – must be separated, since that is how they are typically used. For example, the methods and fields of the association components of `Person` must be kept separate. Separating all methods manually is error-prone and requires separation of non-selected methods.
10. **Mass Renaming:** *Renaming:* $\mathcal{O}(\mathcal{M}_s + \mathcal{F}_s) \rightarrow \mathcal{O}(\ldots)$ Many components have patterns in the names of their methods. For example, the methods for associations are typically named `getX`, `setX`, `isValidX`, and so on. If such a pattern can be exploited, all of its occurrences can be replaced with a single declaration.
11. **High-level Dependencies:** *Resolving method dependencies:* $\mathcal{O}(\mathcal{D}_{\mathcal{M}}) \rightarrow \mathcal{O}(\ldots)$ Some components depend on methods of other components. For example, a component for bidirectional associations needs the method of the other end of the association to maintain consistency, but it does not know their final names. Resolving these dependencies individually is tedious and error-prone. In addition, if additional dependencies are added between two components, all classes that reuse them must add additional wiring code. By directly connecting entire components to each other, all dependencies between them are resolved at once, and additional dependencies require no additional wiring code. In the formula, $\mathcal{D}_{\mathcal{M}}$ is the number of method dependencies of the reused component.

Completeness Features. The following features increase the amount of functionality of a component that can be reused.

12. **Reuse of Hidden Functionality:** Methods that are not exposed in the interface of the reusing class – to prevent conflicts and interface bloat – may still be valuable to clients. They should still be reusable, unless the developer explicitly forbids clients to access them. Examples are advanced iteration methods for associations.
13. **Reuse of Component Type:** If an object cannot somehow be used as if it were of the type of one of its components, certain methods cannot be reused. For example, class `BoundedValue` has a method to transfer the remaining value to another `BoundedValue`. If that method cannot be used to transfer the remaining money from one bank account to another, it must be duplicated even though `BankAccount` offers all required methods and fields. But if the bounded value component of `BankAccount` can be used as a real `BoundedValue`, the transfer method can be reused.

Methodological Features. The following features prevent errors and confusion.

14. **Reuse Without Subtyping:** Mandatory subtyping causes confusion in case of repeated reuse, and it does not make sense for most components. For example, class `BankAccount` is no bi/uni-directional association, or a bounded value. Similarly, class `Person` is not three times a bidirectional association.
15. **No Surprises:** The mechanism must never automatically resolve a name conflict unless one of the candidates overrides all others. Otherwise, methods are overridden based only on the form of their signature, causing unexpected behavior at run-time [37]. A good reuse mechanism exposes such errors, instead of hiding them.

Applicability Features. The last set of features concerns the applicability of the reuse mechanism. They allow the reuse of a component even if it was not anticipated.

16. **No Separate Concept:** If a developer needs to reuse a class as a component, he must be allowed to do so, even if such reuse was not anticipated. In addition, it must be possible to instantiate non-abstract components. For example, there is no reason to complicate the creation of an object that represents a bounded value. If components and classes are the same, they are not limited to a single kind of reuse.
17. **Override State:** If the state of a component is not appropriate for the reusing class, e.g. because it can be computed, it must be possible to override the state. Otherwise, that class cannot reuse the component.
18. **Merge State:** The state of components can overlap in the context of the reusing class. But if the overlapping parts cannot be merged, the components cannot be reused. For example, if a class has two values lying within the same limits, and there is no specific component offering such behavior, it must be possible to use two `BoundedValue` components and merge their upper and lower limits.

2.2 Existing Reuse Mechanisms

Figure 3 shows the features that are supported by different reuse mechanisms. For languages with a separate code inheritance relation, we used that relation in the table. For the other languages, the standard inheritance relation is used. The mechanisms are discussed in more detail in the related work in Section 6.

For delegation, the major problem is that the interface of a component cannot be reused. Every method must be redefined in the reusing class to invoke the corresponding method on the delegatee. The case study shows that this is a big disadvantage. Whether or not state can be overridden or merged depends on the programming language.

The inheritance techniques – with or without subtyping – have poor support for the required features, and very poor support for the expressivity and completeness features. Only two mechanisms support the minimal requirements, and certain important expressivity and completeness features are not supported by any of them. In the columns of features that save of lot of work, there is a big gaping hole.

Our inheritance mechanism supports all the features, and makes the implementation of the *entire* application as big as the traditional implementation of `BankAccount`.

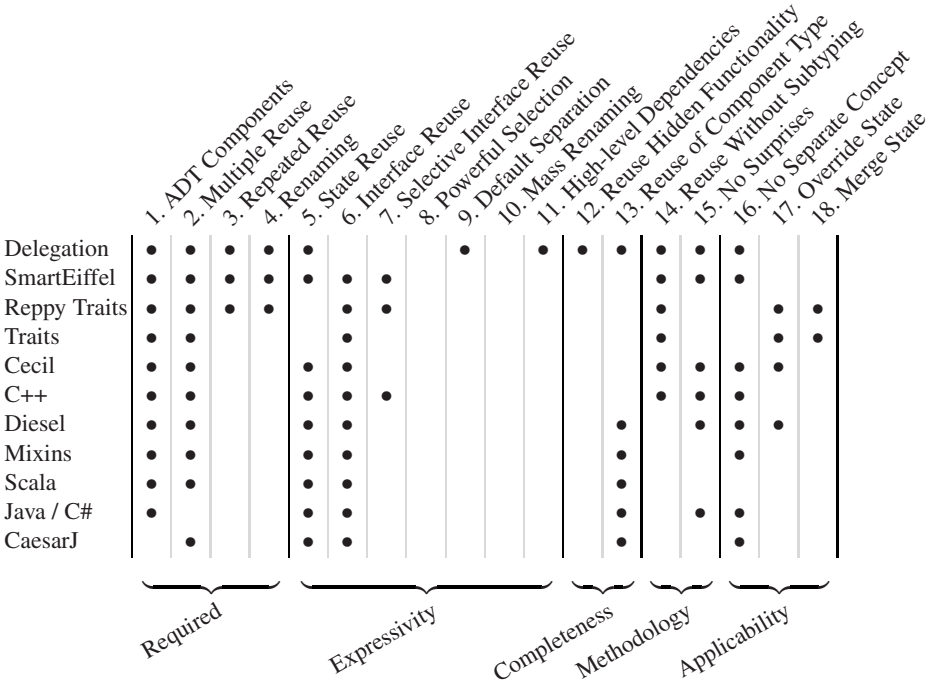

Fig. 3. Feature matrix for different code reuse mechanisms

3 The Component Relation

The component relation is a code inheritance relation for easily reusing existing components in a new class. To simplify the customization of general components for use in a class, the relation offers a number of new features which are explained further on in this section. We introduce renaming parameters for mass renaming the methods of the component. We then turn the component relation into a first-class relation. The relation can be given a name, which can be used to access non-selected functionality, use components as separate objects, and resolve dependencies on a high level. These features allow programmers to work easily with components on a high level of abstraction instead of implementing them with low-level code.

Using the component relation, the banking application of Figure 1 can be implemented by using a component relation for each component. This is illustrated in Figure 4 for the class of bank accounts. The component relations state that the class of bank accounts has a component named `owner` that behaves like a bidirectional association with multiplicity `1`, a component named `balance` that behaves like a bounded value, and a component named `accountNumber` that behaves like a unidirectional association. The assignments are used for renaming, and in this case rename many methods at once by assigning values to renaming parameters. The `owner` component is connected to the component at the other end of the bidirectional association by

```
component BidiAssociation-1-Side<Account,Person> owner (accounts) [X=Owner]
component BoundedValue<long> balance [X=Balance,LOW=LowerLimit,HI=UpperLimit];
component UniAssociation-1<int> accountNumber
          [X=AccountNumber, export private {setAccountNumber}]
```

Fig. 4. The component relations of `BankAccount`

ComponentClause:
  AccessMod? **`component`** *Type Config*?
Config:
  Name? *CompParams*? *ConfigBlock*?
Name:
  AccessMod? *Identifier*
CompParams:
  "("*Identifier* (, *Identifier*)* ")"

ConfigBlock:
  "[" *ConfigClause* (, *ConfigClause*)* "]"
ConfigClause:
  Identifier = *Identifier*?
  `override` "{" *IdentifierList* "}"
  `undefine` "{" *IdentifierList* "}"
  `export` *AccessMod* "{" *IdentifierList* "}"
  `direct` "{" *IdentifierList* "}"
  `indirect` "{" *IdentifierList* "}"

Fig. 5. Grammar for component relations

passing the name of the other component (`accounts`) to the relation. Finally, the setter method for the account number is made `private`.

Figure 5 shows the syntax of the component relation. It consists of the keyword `component` followed by the name of the inherited class, including any generic parameters. There can optionally be a name, component parameters, and a configuration block. The access modifier of the relation determines if the type of the component is visible to the client, which provides valuable information about its behavior. The access modifier of the name determines if he can use the name of a visible component relation to access it as a separate object or resolve dependencies. The configuration block is similar to that of Eiffel. The assignment is used for renaming which is further explained in Section 3.2, `override` if a feature[2] is overridden, `undefine` to undefine a feature in case features are merged, and `export` for changing the visibility of a feature. The inheritance name, component parameters, and `direct` and `indirect` clauses are further discussed in Section 3.3.

3.1 General Semantics

If class `A` has a component relation with class `B`, `A` inherits the features of `B`, but not its type. For example, the class of bank accounts inherits all features of a bounded value, but a bank account is no bounded value. Despite the absence of subtyping, however, both methods and instance variables[3] must conform to all features they override, because the methods of the inherited class expect them to behave according to their original signatures and contracts.

If a feature is inherited via different inheritance paths, a choice must be made to decide if the feature is inherited once, or multiple times. The default policy for features

[2] The features of a class are its instance variables and methods.

[3] Instance variables are properties that can be overridden and merged.

inherited via a component relation is duplication because, generally, the components do not overlap. This means that if a feature is inherited via a component relation and again via another inheritance relation, there is a conflict, even if the definitions are the same. This conflict must be resolved explicitly, e.g. via merging or renaming. To avoid an explosion of the number of renaming clauses, we introduce *renaming parameters* in Section 3.2 and *indirect inheritance* in Section 3.3.

As in SmartEiffel, binding of features in inherited methods is done within the inheritance relation through which they are inherited. This is required to allow separation of the components. For example, `CheckingAccount` inherits the getter method of `BidiAssociation-1-Side` twice: once for the association with the owner, and once for the association with the bank card. Both getters must of course use the instance variable of their own component.

3.2 Renaming Parameters

Without intervention, using duplication as the default for the component relation would force a developer to explicitly rename almost every method of the component. The case study in Section 5 shows that renaming is a significant problem. We introduce a *lightweight macro system* to minimize the effort of renaming features.

The names in the features of a component often exhibit patterns. For example, the names of the methods of the N side of an association are `getX`, `addX`, `removeX`, `replaceX`, `containsX`, and so on. To avoid these patterns from getting lost in the implementation, we introduce *renaming parameters*. A renaming parameter can be written in the names of non-private features, and allows an inheriting class to rename all features that use the parameter with a single renaming declaration.

A renaming parameter is a parameter of a class and is written between square brackets. It can be given a default value; otherwise its name serves as the default value. The parameter can be used in feature names by writing its name between % characters. An inheriting class can assign a value to the parameter in the configuration block of the inheritance relation. The value of the parameter can be any string that is valid for all feature names containing the parameter – which are all visible to the inheriting class.

Figure 6 illustrates the use of renaming parameters. Parameter `X` is used as the name of the other end of the association and is initialized to the empty string. Parameter `XS` represents the plural of `X` and by default equals the value of `X` appended with an '`s`'. For the `children` component of `Person` both parameters are assigned because the default value of `XS` is not appropriate.

We can now determine the amount of work required for renaming. $\mathcal{P}_s$ is the number of renaming parameters in the selected features. $\mathcal{M}_{s,np}$ and $\mathcal{F}_{s,np}$ are the number of selected methods and fields without renaming parameters. The impact of renaming parameters is $\mathcal{O}(\mathcal{M}_s + \mathcal{F}_s) \rightarrow \mathcal{O}(\mathcal{P}_s + \mathcal{M}_{s,np} + \mathcal{F}_{s,np})$.

More details about renaming parameters can be found in the technical report [46].

3.3 First-Class Component Relations

In this section, we introduce first-class component relations to solve a number of problems. We use them to connect components without resolving every individual depen-

```
class BidiAssociation-N-Side <FROM,TO> ... [X=,XS=%X%s] {
  Set<TO> get %XS% {...}
  void add %X% (TO x) {...}
  void remove %X% (TO x) {...}
  void replace %X% (TO x, TO y) {...}
  ...
}

class Person
  component BidiAssociation-N-Side<Person, BankAccount> ... [X=Account]
  component BidiAssociation-N-Side<Person, Person> ... [X=Parent]
  component BidiAssociation-N-Side<Person, Person> ... [X=Child,XS=Children]
{...}
```

Fig. 6. Using renaming parameters

```
class BankAccount
  component BidiAssociation-1-Side<BankAccount, Person> owner ...
  ...
class Person
  component BidiAssociation-N-Side<Person, BankAccount> accounts ...
  ...
```

Fig. 7. First-class component relations

dency, to access functionality that is not exposed in the interface of the reusing class, and to use components as if they were separate objects.

A component relation can have a name, which typically represents the role of the component in the reusing class. Figure 7 illustrates this for classes `BankAccount` and `Person`. The association components of `BankAccount` and `Person` are named `owner` and `accounts`.

Direct and Indirect Inheritance. As presented in Section 2, selective reuse of the interface of a component is required for two reasons.

First, it prevents interface bloat in the reusing class. Take for example the association components. In order to maximize code reuse, it is best to put many features in the association classes. Examples include applying some action to all referenced elements, a universal and an existential quantifier, accumulation, and validation. But for an inheriting class, this means that either its interface gets bloated, or its developer must do a lot of work to hide the functionality, preventing reuse.

Second, because not all method and field names use renaming parameters, there are still many name conflicts. For example, features like `equals` and `hashCode` in the top-level class cause conflicts in every component relation. But if these features are not interesting in the inheriting class, which is usually the case, the developer should not have to resolve their conflicts.

To solve both problems, we make a distinction between *directly* and *indirectly* inherited features. A directly inherited feature is present in the interface of the inheriting class, while an indirectly inherited feature is not, and thus cannot cause a conflict.

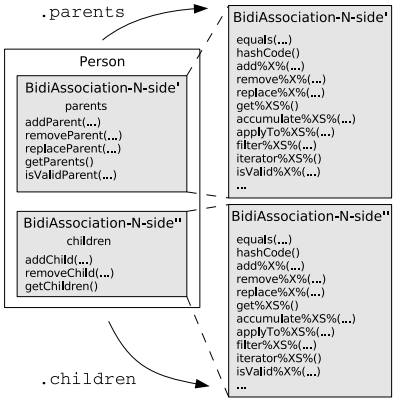

Fig. 8. Indirect Inheritance

```
Person sandra = . . . ;
Person bruno = . . . ;
Person kato = . . . ;

// two identical method calls
sandra .children.add (kato);
sandra .addChild (kato);

bruno.addChild(kato);
kato.parents.applyTo( . . . );
```

Fig. 9. Using indirectly inherited features

An indirectly inherited feature, however, *can still be accessed* if the component relation has been given a name. The feature can then be invoked as `myObject.inheritanceName.feature` using its original name. It is *as if* the component is an object referenced by a field in the reusing class. This way, the client resolves the conflict by using the name of the component relation. It is, of course, the responsibility of the programmer to give the reusing class a meaningful interface. Using inheritance names to access features must not be the standard way of using a class.

Figures 8 and 9 illustrate this for class `Person`. For the `children` component, only the `add`, `remove`, and `get` methods are inherited directly. The `parents` component additionally inherits the `replace` and `isValid` methods. The other methods must be invoked indirectly via the name of the inheritance relation. Note that the invocations of `children.add` and `addChild` in Figure 9 are identical even if the method has been overridden in `Person`.

The inheriting class must specify which features are inherited directly. This is done in the configuration block either by including them with a `direct` declaration, or by renaming or overriding them. All other features are inherited indirectly.

To facilitate selecting directly inherited features, the features of a class can be put in groups as in Eiffel, Smalltalk, and C#. This way, inheriting classes can directly inherit an entire group of methods with little effort. For example, the basic functionality of a class can be put in one group while more advanced functionality can be put in others. To select which features or groups are inherited directly, the programmer can use `direct` and `indirect` declarations in the configuration block of the component relation. A feature is inherited directly if it is listed in a `direct` declaration, and indirectly if it is listed in an `indirect` declaration. If a feature is not listed in such a clause, it is inherited directly if they are part of a group that is listed in a `direct` declaration, and indirectly if its are part of a group that is listed in an `indirect` declaration. Every component relation implicitly has a `direct` declaration for the group named `default`. This is illustrated in Figure 10. The mechanism can be made more flexible, but that is not in the scope of this paper.

```
class BidiAssociation-N-Side<FROM,TO> ... [X,XS=%X%s]
  boolean equals(Object other) {...}
  int hashCode() {...}
  group default {
      Set<TO> get%XS% {...}
      void add%X%(TO x) {...}
      void remove%X%(TO x) {...}
      void replace%X%(TO x, TO y) {...}
  }
  group iteration {
      filter%XS%(...) {...}
      applyTo%XS%(Command<TO>) {...}
      ...
  }
}

class Person
  component BidiAssociation-N-Side<Person,Person> children (parents)
              [X=Child,XS=Children, indirect{replaceChild} ]
  component BidiAssociation-N-Side<Person,Person> parents (children)
              [X=Parent, direct{isValidParent} ]
  ...
```

Fig. 10. Selecting directly inherited features

The impact of indirect inheritance is $\mathcal{O}(\mathcal{M}_s + \mathcal{F}_s) \rightarrow \mathcal{O}(\mathcal{G}_s + \mathcal{M}_{s,ng} + \mathcal{M}_{ns,g} + \mathcal{F}_{s,ng} + \mathcal{F}_{ns,g})$ with $\mathcal{G}_s$ the number of selected groups, $\mathcal{M}_{s,ng}$ and $\mathcal{F}_{s,ng}$ the selected methods and fields not in such a group, and $\mathcal{M}_{ns,g}$ and $\mathcal{F}_{ns,g}$ the unwanted methods and fields in the selected groups.

Component References. Using indirect inheritance, the features of a component can be accessed *as if* the component were an object referenced by an immutable[4] instance variable. To allow even more reuse, we allow the name of a component relation to be *actually* used as a reference to the subobject representing that component, similar to casts in C++ [39]. Because we already require conformance between the actual component and the inherited class, type-safety is not endangered.

Component references make it possible to reuse methods of which a formal parameters has a type that is used as a component. For example, the class representing bounded values has methods to compare it with another bounded value, to transfer the remaining value to another bound value. Another example is the `equals` method for an association, which takes a similar association as its argument, to verify if two association reference the same elements. Without component references, these features cannot be reused if the class is used as a component for another class because there is no subtyping relation between the reusing class and the component.

Figure 11 shows how such methods can be reused using component references. The methods cannot take a `Person` or `BankAccount` as an argument, but by using the names of the component relations, the components can be passed to the method.

[4] Only the reference itself is immutable, the referenced object can still be modified.

```
boolean eq = sandra.children.equals( bruno.children );
yourAccount.balance.transferRemainingValueTo( myAccount.balance );
```

Fig. 11. Using component references

Being able to use component references has an influence on how `this` is treated in the context of a component relation. When a class `A` is reused through a component relation by class `B`, its `this` reference acts as if it were substituted by `this.inheritanceName`. Otherwise, `this` would have type `A` in the context of type `B`, which is not type-safe since `B` is not a subtype of `A`.

Consequently, a component cannot use the `this` reference to obtain a reference to the object of the reusing class because that is a reference to the subobject for that component. For example, the components for bidirectional associations need an object of type `FROM` – a generic parameter – to pass it to the other end of the association. Various techniques can be used to obtain that reference, e.g. storing it explicitly in a field, self types as used in Eiffel, or a variant of the self types in Scala. More details on techniques to obtain a reference to the object of the reusing class can be found in the technical report [46].

Dependency Resolution. Some components depend on methods of other components. Examples are the methods to set up and break down bidirectional associations, as shown in Figure 12. The `setOwner` method of `BankAccount` must know which `register` method to invoke on the `Person` to keep the association consistent. Similar dependencies exist for the other methods. Because `Person` has multiple associations, these dependencies cannot be resolved automatically. The developer of `BankAccount` must connect these methods to the appropriate methods in `Person`. With existing inheritance mechanisms, this must be done with wiring code for each individual method dependency.

To resolve these dependencies more elegantly, we use the names of the component relations. Figure 13 illustrates the approach. The `owner` component of `BankAccount` and the `accounts` component of `Person` are connected by resolving a single high-level dependency on each side.

To specify high-level dependencies, a class can declare formal *component parameters* . They are declared after the generic parameters of a class between parentheses, and have the form `T → C cparam`. In this declaration, `T → C` is a constraint on the

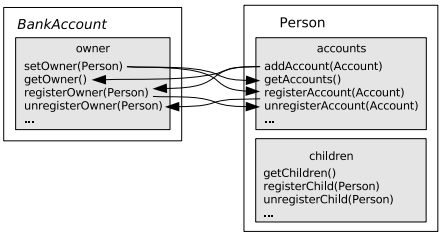

Fig. 12. Low-level dependencies

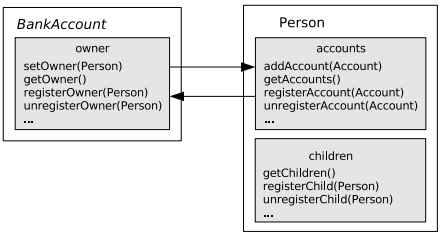

Fig. 13. High-level dependencies

```
class BidiAssociation-1-Side<FROM,TO> (TO → BidiAssociation<TO,FROM> otherEnd)
      subtype BidiAssociation<FROM,TO> (otherEnd){

  private TO other;
  public void setX(TO other) {
    ...
    other@otherEnd.register (expression for the object on this side of the association);
    ...
  }
  protected void register(TO other) {...}
  ...
}
```

Fig. 14. Component parameters

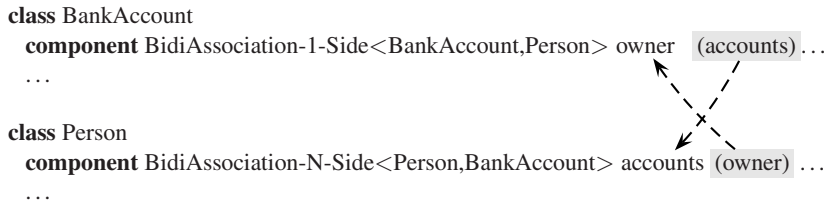

Fig. 15. Implementing high-level dependency resolution

component relation passed through the parameter. `T` is the type *containing* the relation, and `C` is the target type *of* the relation. Finally, `cparam` is the name of the parameter.

Figure 14 illustrates the declaration of a component parameters. The formal parameter expects the name of a relation that *a)* is a relation of the class at the other side of the association (`TO`), and *b)* is a `BidiAssociation` representing an association in the opposite direction (from `TO` to `FROM`). Figure 15 illustrates how two association components are connected to each other. If we substitute the generic parameters, we see that component relation `owner` requires the name of a component relation with type `BidiAssociation<Person,BankAccount>` that is contained in `Person`. Since the `accounts` component of `Person` satisfies these constraints, we can connect the `owner` component to the `accounts` component. Similarly, the `owner` component satisfies the constraints of the `accounts` component. Consequently, the `owner` component of `BankAccount` can be connected to the `accounts` component of `Person`, and vice versa.

A component parameter can be used to invoke features of the actual component passed through the parameter on objects of the type containing the component. Method invocations and field accesses are performed using the following expressions: `expr@cparam.m(args)` and `expr@cparam.f`. If `cparam` has `T` → `C` as constraint, `expr` must be of type `T`, and `m` or `f` must be applicable to type `C`. In the context of a component relation where actual component parameter `aparam` is used, `cparam` is replaced with `aparam`. As a result, method `aparam.m(args)` or field access

`aparam.f` are invoked on the result of `expr`. Note that any renaming or overriding of these features in the run-time type of `expr` is taken into account. We use a symbol different from the dot to emphasize the difference with a regular invocation. In addition, this avoids confusion about the meaning of `expr.cparam` if a feature with name `cparam` is added to `T`.

The `setX` method in Figure 14 shows how the component parameter is used to invoke methods. The invocation of `register` is applied to the `otherEnd` component of `other`. The method that will be invoked, is the `register` method of the actual component relation passed through the parameter, which may be overridden or renamed in the actual class `TO`. In the example of Figure 15, the `setOwner` method inherited by `BankAccount` will invoke the `registerAccount` method inherited by `Person`.

This approach has a number of advantages. First, it saves a lot of work by replacing the individual dependencies with a smaller number of high-level dependencies. The impact is $\mathcal{O}(\mathcal{D}_{\mathcal{M}}) \rightarrow \mathcal{O}(\mathcal{D}_{\mathcal{C}})$ with $\mathcal{D}_{\mathcal{C}}$ the number of component dependencies and $\mathcal{D}_{\mathcal{C}} \leqslant \mathcal{D}_{\mathcal{M}}$. Second, it ensures that the required methods are provided by a single component and not by methods of different components, which is crucial in this example. Third, if additional dependencies are added between two types of components, the reusing classes need no modifications. For example, we can add an `isSibling` method to `BidiAssociation` to check if some object is its sibling. This method would invoke the `contains` method on the other end of the association, introducing another dependency. Inheriting classes, however, do not need to be modified.

Visibility. By default, component relations are public because they are typically used for the characteristics of a class. A public client can see their name, type, and configuration. If a programmer knows the behavior of class `C`, he also knows the behavior of a component of type `C`. But if the relation is not visible, he must study the contracts of the inherited features again in order to understand their behavior. If the component relation is used for traditional code inheritance, for example to implement a `Stack` using an `Array`, it should be hidden from the client.

4 The Subtyping Relation

In this section, we briefly explain the most important differences with the subtyping relation of SmartEiffel. The details can be found in the technical report [46].

Figure 16 shows the syntax of the subtyping relation. It consists of the keyword `subtype` followed by the name of the super type, including any generic parameters. There can optionally be a name for the relation, and a configuration block. The component parameters are used to transfer compent parameters to the superclass, similar to generic parameters. To ensure consistency, component parameters passed to the same class via different subtyping relations must be identical. This is similar to the rule for generic parameters in Java and SmartEiffel.

Because the subtyping relation is no longer used for pure code reuse, it can be simplified. Duplication is forbidden since it is inappropriate for subtyping, avoiding confusion for diamond inheritance. In addition, the rule-of-dominance (as in C++ [39]) is used to avoid needless `undefine` clauses to select a version of a method when there is a single most specific version.

SubtypeClause:
`subtype` *Type Identifier*? *CompParams*? *ConfigBlock*?

Fig. 16. Grammar for the subtyping relation

4.1 Overriding and Merging Components

For the same reasons why overriding and merging of state is required to ensure that a component can always be reused, as discussed in Section 2, it must also be possible to override and merge component relations. Similar to overriding and merging methods, either an overriding component must be defined, or an existing one must be selected.

In both cases, the overriding or selected component must satisfy two rules. First, standard subtyping conformance is required. The overriding component must not only be a subtype of the target class of the component relation, but also of all overridden components. Second, conformance of the component interface is required. This means that every feature that is inherited directly in an overridden component relation must be inherited directly in the overriding component relation. In addition, corresponding features must be given the same name.

4.2 Reducing Hierarchy Dependencies

In [35], it is argued that `super` calls in languages with multiple inheritance increase the dependency of code on the class hierarchy. In such languages, multiple methods with the same name can be inherited by a class, so in order to disambiguate super calls to such methods, they must be qualified with the name of the direct super class containing the method that must be invoked. Examples of languages using this approach are C++, Cecil, Eiffel, and SmartEiffel. This problem does not occur with inheritance mechanisms that linearize the class hierarchy, or in the prototype-based language Self [8], where super calls can be directed to a named parent slot.

These dependencies can be removed by also giving a name to a subtyping relation. It is possible to qualify a super call using the name of that inheritance relation instead of the name of the super class. Consequently, the call remains valid if the actual super class for that relation is changed, as long as an appropriate method is available in the new super class. The name of a subtyping relation is private since only the inheriting class can invoke super calls.

Technically, reuse variables in Timor also reduce this dependency, but in their paper [19], the authors do not present this insight.

5 Evaluation

In this section, we evaluate the complexity and the effectiveness of the proposed inheritance mechanism.

5.1 Complexity

Even though our inheritance mechanism introduces a number of new features, it is still easy to use, and it reduces overall complexity. Programmers already deal with

high-level characteristics, dependencies, and name patterns, but they must encode them using complicated low-level code instead of writing simple high-level code. The features introduced in this paper allow programmers to easily reuse, configure, and connect components to build a class. With the creation of a graphical editor, this can even be done by simply drawing a diagram similar to Figure 1. Components can be dropped on classes, configured by filling in the type and name patterns, and connected to each other.

As an additional advantage, the subtyping inheritance relation can be simplified as it is no longer used just for code reuse. More specifically, forbidding duplication prevents confusion in case of diamond inheritance, and the rule-of-dominance resolves unnecessary conflicts that must otherwise be resolved by the programmer.

5.2 The Banking Application

Figure 17 shows the *entire* implementation of Figure 1. The names of the association classes are abbreviated for reasons of space. The implementation is done almost completely by configuring existing components. Only the constructors are actually implemented. This is an important result, because it means that this implementation can be done by drawing a class diagram, and filling in the parameters. Although the example does not contain any application specific behavior, it illustrates what can be achieved with our approach. A realistic case study is presented further on.

In addition, the high-level concepts of the diagram cannot get lost because they are directly present in the code. In current CASE tools, such concepts can get lost because they are translated into low-level code, leading to synchronization problems.

5.3 Case Study

We compared our inheritance mechanism with manual delegation, and the inheritance mechanisms of Java, SmartEiffel, and Reppy traits, which support repeated inheritance [32], by comparing their impact on the size of an application. We used *Jnome* [43], our metamodel for Java, and *Chameleon* [43], our framework for metamodels of programming languages. Together they contain 9763 lines of Java code.We must note that the reduction in code size is *not* the same as the reduction in complexity. Renaming clauses and manual delegation are much simpler than the reused methods.

We modified the Java programs using our inheritance mechanism[5], and then *calculated* the size for the other techniques based on the overhead of renaming, dependency resolution, encapsulation of state, and manual delegation for each technique. Note that *only the inheritance relations and wiring code* differ for the participating mechanisms. All other code is identical, so all effects are due to differences in the reuse mechanisms.

To study the impact of the size and the nature of extensions of the components, we repeated the experiment for two kinds of extensions. In the first extension, all associations send events when they are modified. This extension is application independent because managing the listeners and invoking `notify` is always the same. In the second

[5] We must note that the resulting code does not currently compile because our compiler is not yet complete.

```
class BankAccount
  component BoundedValue<long> balance
                [Value=Balance, Lower=Credit, increaseBalance=deposit,
                 decreaseBalance=withdraw,
                 export private {setUpperLimit,setLowerLimit,setBalance}]
  component Bidi-1-Side<BankAccount,Person> owner (accounts) [X=Owner]
  component Uni-1<int> accountNumber
                [X=AccountNumber, export private {setAccountNumber}]
{
  public BankAccount(int accountID) {
     balance.super(0,-1000,1000000);
     accountNumber.super(accountID);
  }
}

class CheckingAccount
  subtype BankAccount
  component Bidi-1-Side<CheckingAccount, BankCard> bankCard (account) [X=BankCard]
{
  public BankAccount(int number) {
     super(number);
  }
}

class Person
  component Bidi-N-Side<Person,BankAccount> accounts (owner) [X=Accounts]
  component Bidi-N-Side<Person,Person> parents (children) [X=Parents]
  component Bidi-N-Side<Person,Person> children (parents) [X=Children]
  component Uni-1<String> [X=Name]
  component Graph<Person> family (parents,children)
{
  public Person(String name, Person mother, Person father) {
     setName(name);
     addParent(mother);
     addParent(father);
  }
}

class BankCard
  component Bidi-1-Side<BankCard,CheckingAccount> account (bankCard) [X=Account]
  component Uni-1<int> [X=PinCode]
{}
```

Fig. 17. Implementation of the banking application of Figure 1

extension, which builds on the first one, the associations also check the validity of the elements. For this extension, the validity condition is application specific and must be overridden, while other supporting code can be reused.

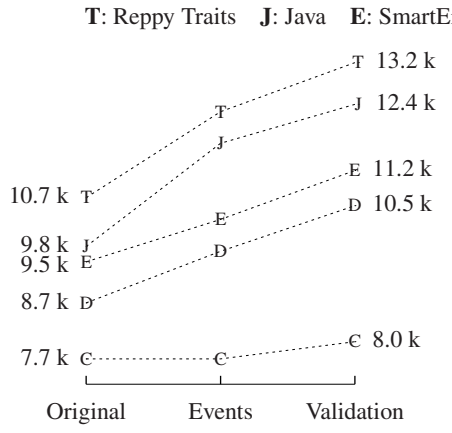

Fig. 18. Lines of Code

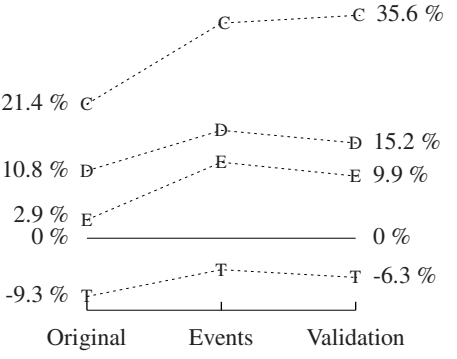

Fig. 19. Reduction Compared to Java

Figure 18 shows the code size for the different techniques and code bases. Figure 19 shows the reduction in size compared to Java. Almost all of the reduction is obtained in the domain model, which takes up 70% of the software. The other 30% consists of input and output algorithms.

Both figures clearly show that our inheritance mechanism results in a much bigger reduction than the other mechanisms. The difference is caused by the additional overhead mentioned above. Manual delegation and code inheritance in SmartEiffel reduce the size much less than our mechanism, but are still a big improvement over the Java version. Using Reppy traits, however, the code size even increases. The additional getter and setter methods – traits cannot contain state – cause so much additional overhead that the application becomes even bigger than the original Java application.

An important result is the impact of adding functionality that is not overridden in the application. Adding support for sending events requires *no modification* of the version using our inheritance mechanism. The renaming parameters, component parameters, and indirect inheritance avoid the need for additional code if all methods and variables added to the `default` group contain existing renaming parameters. With the other techniques, code must be added to the applications for renaming clauses, manual delegations, dependency methods, or state encapsulation. The more functionality is offered by the component, the more modifications are required by other techniques. This is a very important practical result. It shows that the developer of a component can usually add functionality without breaking client code if it is not added to the `default` group or if it uses renaming parameters. In addition, he can now provide lots of functionality without putting a huge burden on his clients.

Another important result shows up if validation is added to the associations, and specific validation rules are implemented in the applications. The version using our inheritance mechanism is the only one in which less code must be added than in the Java version, as shown by the gradients in the right part of Figure 19. This means that it is still beneficial to reuse small components, or small parts of bigger components, using our inheritance mechanism. Using the other techniques, the additional overhead makes reuse unattractive in these scenarios.

6 Related Work

In [28], Odersky and Zenger identify three scalable component abstractions for removing hard references from components to increase their reusability: *abstract type members*, *selftype annotations*, and *modular mixin composition*. Abstract type members and selftypes specify the required services of a component, and mixins perform the composition. But while these abstractions are scalable with respect to the size of the components, they are not scalable in the way components are used. The problem is that both selftypes, and mixins as used in Scala, prohibit any composition involving multiple components of the same kind, or components containing features with the same name. Despite the claim that these abstractions can lift an arbitrary assembly of static program parts to a component system, they already fail for our simple example application, which is little more than an assembly of four kinds of static program parts. The authors argue that nesting of classes is essential because otherwise, the amount of wiring would become substantial. This contradicts our findings. In this paper, we built an application using components without using nested classes. So while nested classes provide certain benefits, they are not a requirement for component composition.

In [44], we introduced anchored exception declarations to remove hard references from the exceptional specification of a component. They allow the exceptional specification of a method to be declared relative to other methods. This increases both the adaptability and reusability of code using checked exceptions. In the context of this paper, they allow a component to specify its exceptional behavior in terms of the exceptional behavior of its dependencies. As a result, the compiler can determine more precisely which exceptions can be thrown for a particular composition.

In [3], Bierman and Wren present a language construct for first-class full-blown relationships. A similar construct is advocated by Rumbaugh in [33]. With our inheritance mechanism, it can be replaced by a library of classes. In this paper, we used relationships without attributes, but classes for full-blown relationships can be built on top of them. An example implementation is given in the technical report [46]. Another language construct that can be replaced by a class are C# events [13].

In [31], Pearce and Noble provide support for relationships using AspectJ [20]. The authors offer a library of relationship aspects, similar to our association components, which are inserted into the application using a point-cut for each relationship in the model. Support for static relationships – relationships that are part of the participating classes – is limited because name conflict for the introduced methods and fields cannot be resolved. As such, AspectJ cannot be used to create and reuse abstract data type components. An advantage of the approach is that components can be added externally to existing classes, but using a form of higher order hierarchies as in [29,30,14,27,28], this can also be achieved with an object-oriented approach.

In the 1997 version of Eiffel [25], the inheritance relation is used both for subtyping and code inheritance. It is possible to duplicate features when inheriting more than once from the same class, which is confusing for subtyping. The resulting diamond problem for repeated inheritance is often considered to make the language more difficult [4,34]. In addition, a subclass can use covariant argument types for a method, or even remove features, which makes a whole program analysis required to ensure type safety.

In SmartEiffel 2.2 [9] and the new Eiffel specification [42], the inheritance mechanism has been extended with non-conforming inheritance. In SmartEiffel 2.2, duplication of features and narrowing their visibility is no longer permitted. Using covariant argument types, however, remains possible. SmartEiffel ensures type-safety by type-checking the code of an inserted class in the context of the inheriting class, but this violates the modularity principle. Because sharing is the default policy for the *insert* relation, accidental merging of components is possible.

Timor [19] and Sather [40] separate types and classes, and the relations between them. Types can inherit from multiple other types. Classes can *include* other classes for code inheritance, and they can implement types. Timor further uses named subtyping relations [19] to support repeated inheritance. We think it is very confusing for an object to be 1.9 times a `CassettePlayer`, as in their example. They also use the inheritance names to disambiguate conflicting names, but for reusing components this approach is not practical. A severe problem with their mechanism is that name conflicts are automatically resolved by removing direct access to the involved methods. As a result, adding a subtyping relation, or even adding a method to an inherited type can break existing clients without even a warning because conflicts can be introduced. Timor also has support for *reuse variables*. Features of the classes referenced by such variables are inherited if they are needed for the types implemented by the class. If they are not inherited, however, they are not available to clients since they are not part of the types via which the class can be used. The mechanism can be seen as delegation-by-value. Reuse variables also reduce the dependency of the implementation of a class on its hierarchy, but the authors do not present this insight.

Traits [35] not only use a separate relation for code inheritance, but also a separate concept – a *trait* – for a set of methods that can be reused via code inheritance. Unlike traits, we do not have a separate concept to represent a component, it is just a class. If the component relation could only be used with special building blocks, unanticipated reuse would be impossible. On top of that, programmers must deal with an extra concept which is just a degenerate abstract class. Another motivation for our choice is the possibility to instantiate components. We see no reason to forbid a programmer to create an object that represents a bounded value. In addition, classification of characteristics is necessary. To reuse almost any kind of association, it is necessary to create a hierarchy of association classes. The relation between classes capturing choices like mutability and arity, and the class `Association` is a subtyping relation, not just a code inheritance relation. Methods inherited via traits automatically override methods inherited from classes although there is no relation between them. This form of structural subtyping can lead to bugs that are hard to find. In addition, dependencies of traits must be resolved individually, and repeated trait-inheritance is not possible. As such, traits allow far less code reuse than our inheritance mechanism.

In [32], Reppy and Turon present trait-based metaprogramming. They add renaming and hiding to traits to allow using a trait more than once in a class. Similar to SmartEiffel, name conflicts and dependencies must be resolved one at a time. But because traits cannot contain state, the overhead is larger than in SmartEiffel.

Languages like CLOS [12], most mixin-based [4] languages like Scala [28], and many others use linearized multiple inheritance. The linearization of the class

hierarchy, however, complicates its use [37,8,35]. It is not possible to determine the meaning of a single inheritance relation of a class without looking at the others because some of its methods may be overridden by methods of other classes that happen to have the same name. This makes it easy for methods to be overridden by accident [37]. Repeated inheritance, which is required for composition of classes is impossible in these languages. The abstract super class of a mixin, however, allows for reusable refinements, which cannot easily be created using our approach.

Cecil [6] supports multiple inheritance. Repeated inheritance, however, is forbidden, and name conflicts result in compilation errors. The language uses properties for instance variables, making it possible to override them. Subtyping and code inheritance relations can be used both separately or combined.

In Self [8], inheritance relations are given a priority. For relations with identical priorities, name conflicts result in an error. For relations with different priorities, conflicts are resolved automatically by inheriting the feature of the relation with the highest priority. The *Sender Path Tiebreaker Rule* resolves additional conflicts by giving priority to methods within the same inheritance path in case of ambiguities. Renaming is not supported. Directed resends do not increase the dependency between the implementation and the inheritance hierarchy because they are sent to named slots, which is very similar to using named inheritance relations.

C++ [39] has limited support for repeated inheritance. A class cannot inherit from the same base class more than once, making it unsuitable for building classes from components. In addition, it has no support for renaming, forcing clients to resolve name conflicts. The language supports separation of subtyping and code inheritance through public and private inheritance.

Some design patterns can benefit from the component relation, but most cannot. Patterns that require the introduction of certain methods benefit from using a component for each of the participants. Examples are *composite*, *singleton*, *observer*, and *memento*. Frequently used *template method* patterns, such as patterns for caching and locking, can also be captured in a component. The *visitor* and *iterator* patterns benefit from the use of association or relationship components which provide navigation methods. The *state* and *adapter* patterns cannot currently benefit from our approach because the component would have to be interchangeable at run-time as in Darwin and Lava [21].

A split object [2] consists of a collection of *pieces*. Pieces represent particular viewpoints or roles of the split object, and are organized in a delegation hierarchy. Unlike the split object, however, pieces have no identity. Invoking methods is done by selecting a viewpoint to send the message to. The main difference with our approach is that component relations are used to build an abstract data type, whereas pieces are used to model different viewpoints on an object. This difference in purpose results in additional technical differences. The hierarchies of both approaches have an opposite order with respect to overriding. For pieces, the leaves are the most specific parts, whereas for component relations, the root – the composition – is the most specific part. In addition, features in pieces cannot be merged, whereas features inherited through different component relations can be merged. Finally, pieces are added dynamically, whereas component relations are declared statically.

7 Future Work

An important task is to finish our compiler and create a library of reusable components. These include, but are not limited to, a hierarchy of association classes allowing choices like multiplicity, value or reference semantics, mutability, constraints. With these associations, graphs can be built to reuse any iteration over an object structure.

The error handling strategy of a class is fixed at this moment. For example, class `BoundedValue` must choose how to deal with invalid input: use preconditions, throw exceptions, or provide a default behavior. That means that to provide all choices to an application developer, we need three versions of the same characteristic. It would be more interesting to have a single version that provides a number of strategies for dealing with errors, and allowing the application developer to choose one.

8 Conclusion

We have shown that current object-oriented programming languages do not offer the abstraction level required to use general purpose classes as building blocks for other classes in a practical manner. This prevents a developer from reusing high-level concepts like associations, bounded values, and graphs.

We showed which features are required to encapsulate and reuse such concepts, categorized them, and showed how current reuse mechanisms support them. We then integrated those features in a new inheritance mechanism.

Our inheritance mechanism is the first to make this kind of reuse practical. By using renaming parameters and making component relations first-class citizens, we eliminate the problems encountered with existing mechanisms. They allow a programmer to easily exploit name patterns, connect components, provide both a simple class interface and lots of functionality, and use components as if they were separate objects. Together, these improvements raise the abstraction level of the programming language, since it is no longer required to create a new language construct or write lots of low-level code to reuse a high-level characteristic. Of course, the component relation can also be used for traditional code inheritance as used in traits, SmartEiffel, and Cecil.

The case study confirms that our inheritance mechanism yields much better results (21% to 36% reduction) than other inheritance mechanisms (3% to 12% reduction), and delegation (11% to 17% reduction). It also shows that our inheritance mechanism is more robust with respect to extensions of components. In addition, it is still beneficial to reuse small components, or small parts of big components with our inheritance mechanism, contrary to the other techniques.

Acknowledgments

We especially thank Adriaan Moors, Jan Smans, and Tom Schrijvers for their advice. We thank the anonymous reviewers for their insightful comments. We thank Dominique Colnet, Guillem Marpons, and Frederic Merizen of the SmartEiffel team for their valuable feedback. We also want to thank Bart Jacobs, Jeroen Boydens, Sven De Labey, Kurt Schelfthout, and Koen Vanderkimpen, who provided many insightful comments.

References

1. America, P.: Inheritance and subtyping in a parallel object-oriented language. In: Proceedings of ECOOP '87, pp. 234–242. Springer, Heidelberg (1987)
2. Bardou, D., Dony, C.: Split objects: a disciplined use of delegation within objects. In: Proceedings of OOPSLA '96, pp. 122–137. ACM Press, New York (1996)
3. Bierman, G.M., Wren, A.: First-class relationships in an object-oriented language. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 262–286. Springer, Heidelberg (2005)
4. Bracha, G., Cook, W.: Mixin-based inheritance. In: Proceedings of OOPSLA/ECOOP '90, pp. 303–311 (1990)
5. Bracha, G., Lindstrom, G.: Modularity meets inheritance. In: Proceedings of the IEEE International Conference on Computer Languages, pp. 282–290. IEEE Computer Society Press, Los Alamitos (1992)
6. Chambers, C.: The Cecil language specification and rationale: Version 3.2 (2004)
7. Chambers, C.: The Diesel language specification and rationale: Version 0.2 (2006)
8. Chambers, C., Ungar, D., Chang, B.-W., Hölzle, U.: Parents are shared parts of objects: inheritance and encapsulation in SELF. Lisp Symb. Comput. 4(3), 207–222 (1991)
9. Colnet, D., Marpons, G., Merizen, F.: Reconciling subtyping and code reuse in object-oriented languages: Using inherit and insert in SmartEiffel, the GNU Eiffel compiler. In: Morisio, M. (ed.) ICSR 2006. LNCS, vol. 4039, Springer, Heidelberg (2006)
10. Colnet, D., Ribet, P., Adrian, C., Merizen, F., Marpons, G.: SmartEiffel 2.2 (2005), `http://smarteiffel.loria.fr`
11. Cook, W.R., Hill, W., Canning, P.S.: Inheritance is not subtyping. In: Proceedings of POPL '90, pp. 125–135 (1990)
12. DeMichiel, L.G., Gabriel, R.P.: The common lisp object system: An overview. In: Bézivin, J., Hullot, J.-M., Lieberman, H., Cointe, P. (eds.) ECOOP 1987. LNCS, vol. 276, pp. 151–170. Springer, Heidelberg (1987)
13. ECMA Technical Committee 39 (TC39) Task Group 2 (TG2). C# Language Specification. ECMA, 2 edn. (December 2002)
14. Ernst, E.: Higher-order hierarchies. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, pp. 303–329. Springer, Heidelberg (2003)
15. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc. (1995)
16. Goldberg, D.S., Findler, R.B., Flatt, M.: Super and inner: together at last! In: Proceedings of OOPSLA '04, pp. 116–129 (2004)
17. Gosling, J., et al.: The Java Language Specification, 2nd edn. Addison-Wesley Longman Publishing Co., Inc. (2000)
18. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. 23(3), 396–450 (2001)
19. Keedy, J.L., Heinlein, C., Menger, G.: Inheriting multiple and repeated parts in Timor. Journal of Object Technology 3(10), 99–120 (2004)
20. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
21. Kniesel, G.: Type-safe delegation for run-time component adaptation. In: Guerraoui, R. (ed.) ECOOP 1999. LNCS, vol. 1628, pp. 351–366. Springer, Heidelberg (1999)
22. Knudsen, J.L.: Name collision in multiple classification hierarchies. In: Gjessing, S., Nygaard, K. (eds.) ECOOP 1988. LNCS, vol. 322, pp. 93–109. Springer, Heidelberg (1988)
23. Leino, K.R.M.: Data groups: specifying the modification of extended state. In: Proceedings of OOPSLA '98, pp. 144–153 (1998)

24. Liskov, B., Wing, J.M.: A new definition of the subtype relation. In: Nierstrasz, O. (ed.) ECOOP 1993. LNCS, vol. 707, pp. 118–141. Springer, Heidelberg (1993)
25. Meyer, B.: Object-oriented software construction, 2nd edn. Prentice-Hall, Inc., Englewood Cliffs (1997)
26. Mezini, M., Ostermann, K.: Integrating independent components with on-demand remodularization. In: Proceedings of OOPSLA '02, pp. 52–67 (2002)
27. Nystrom, N., Chong, S., Myers, A.C.: Scalable extensibility via nested inheritance. In: Proceedings of OOPSLA '04, pp. 99–115 (2004)
28. Odersky, M., Zenger, M.: Scalable component abstractions. In: Proceedings of OOPSLA '05, pp. 41–57 (2005)
29. Ossher, H., Harrison, W.: Combination of inheritance hierarchies. In: Proceedings of OOPSLA '92, pp. 25–40 (1992)
30. Ostermann, K.: Dynamically composable collaborations with delegation layers. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 89–110. Springer, Heidelberg (2002)
31. Pearce, D.J., Noble, J.: Relationship aspects. In: Proceedings of AOSD '06, pp. 75–86. ACM Press, New York (2006)
32. Reppy, J., Turon, A.: A foundation for trait-based metaprogramming. In: International Workshops on Foundations of Object-Oriented Languages (2006)
33. Rumbaugh, J.: Relations as semantic constructs in an object-oriented language. In: Proceedings of OOPSLA '87, pp. 466–481 (1987)
34. Sakkinen, M.: Disciplined inheritance. In: ECOOP, pp. 39–56 (1989)
35. Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.: Traits: Composable units of behavior. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, pp. 248–274. Springer, Heidelberg (2003)
36. Seco, J.C., Caires, L.: A basic model of typed components. In: Bertino, E. (ed.) ECOOP 2000. LNCS, vol. 1850, pp. 108–128. Springer, Heidelberg (2000)
37. Snyder, A.: Encapsulation and inheritance in object-oriented programming languages. In: Proceedings of OOPSLA '86, pp. 38–45 (1986)
38. Stata, R., Guttag, J.V.: Modular reasoning in the presence of subclassing. In: Proceedings of OOPSLA '95, pp. 200–214 (1995)
39. Stroustrup, B.: The C++ programming language, 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1991)
40. Szypersky, C., Omohundro, S., Murer, S.: Engineering a programming language: The type and class system of Sather. Technical Report TR-93-064, Berkeley, CA (1993)
41. Taivalsaari, A.: On the notion of inheritance. ACM Comput. Surv. 28(3), 438–479 (1996)
42. Technical Group 4 of Technical Committee 39. ECMA-367 Standard: Eiffel Analysis, Design and Programming Language. ECMA International (2005)
43. van Dooren, M., Smeets, N.: Jnome (2006), `http://www.cs.kuleuven.be/~marko/jnome/`
44. van Dooren, M., Steegmans, E.: Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations. In: Proceedings of OOPSLA '05, pp. 455–471 (2005)
45. van Dooren, M., Steegmans, E.: Language constructs for improving reusability in object-oriented software. In: Companion to proceedings of OOPSLA '05, pp. 118–119 (2005)
46. van Dooren, M., Steegmans, E.: Abstract data type components. Technical Report CW 439, K.U.Leuven (March 2006), `http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW439.pdf`