Conditional constructors: A CaesarJ extension for better reusability of constructors

Student research project, winter term 2011/12. Supervised by Jan Sinschek.

Marko Martin

MarkoMartin@gmx.net

Technische Universität Darmstadt

Software Technology Group

Abstract

In many object-oriented languages, constructors are a weak point in the sense of bad reusability and high rigidity. Particularly, in Java, constructors have a strong dependence on the superclass; yet, they are not capable of inheriting constructors. With mixins such as implemented with CaesarJ for Java, problems resulting from those weaknesses become even worse as the superclass of a mixin is not statically known. This document presents *conditional constructors* as a countermeasure to the constructor problems in Java and CaesarJ. They support a powerful form of constructor inheritance which also allows for extending or adapting existing superclass constructors. A formal calculus for matching *super* calls in conditional constructors against formal superclass constructors is presented as well as use cases that show the usefulness of conditional constructors.

1. Introduction

In this document, we present a novel extension of the Java programming language which supports better reusability of constructors. In default Java, reusability is limited, especially, due to the lack of constructor inheritance. As a consequence, constructors from superclasses have to be "copied" down into a subclass if they are to be reused which is often the case. This introduces a high degree of rigidity because changes to constructors in such a hierarchy require all subclasses to adopt the changed constructor. We argue that constructors should not be bound statically to superclass constructors at compile time so that superclass constructor changes do not require subclasses to be changed. This work has emerged from an analysis of constructors in existing languages and how they cope with multiple inheritance in a previous seminar paper [4].

As introductory example, consider the classes in Figure 1. *Shape* offers two constructors for the same purpose, namely for initializing its attributes *x* and *y*. *Rectangle* introduces the two new attributes *width* and *height*. Initialization of these can be done in two different ways, too: either by providing separate values for them or by providing a *Dimension* object from which width and height can be extracted. As the developer aimed at keeping the facility to also initialize the position in the two different ways defined by *Shape*, she had to write constructors for each of the four possibilities. This can, in real-world scenarios with more constructor variants, quickly lead to an explosion of the number of constructors, or – what is probably more likely in practice – to vanishing of some possible combinations of constructor parameters as developers might grow tired of writing down all the possible combinations and concentrate on the cases which they consider to be sufficient in usual cases. However, more often than not, inusual cases turn out to occur quite frequently so that the desired combination of constructor parameters is not available. Our approach will tackle this problem by providing a means to partially inheriting constructors from superclasses, intuitively spoken.

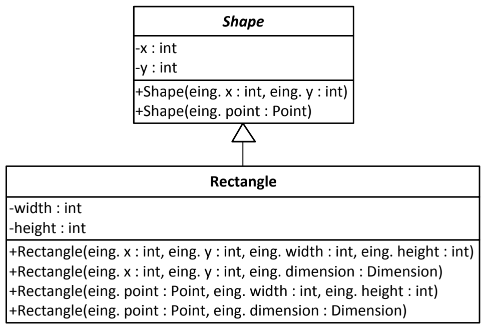
Shape
-x : int
-y : int
+Shape(eing. x : int, eing. y : int)
+Shape(eing. point : Point)
Rectangle
-width : int
-height : int
+Rectangle(eing. x : int, eing. y : int, eing. width : int, eing. height : int)
+Rectangle(eing. x : int, eing. y : int, eing. dimension : Dimension)
+Rectangle(eing. point : Point, eing. width : int, eing. height : int)
+Rectangle(eing. point : Point, eing. dimension : Dimension)

Figure 1. Shape example demonstrating the explosion of number of constructors

Our approach is based on CaesarJ, a mixin extension of Java [1]. Considering mixins, loose coupling of constructors reveals even more advantages since a mixin can now invoke the constructor of a superclass it does not know at runtime.

The paper is organized as follows: First, we present our approach in detail in Section 2. This includes a description of its principles in Section 2.1, a definition of possible arguments in super constructor calls in Section 2.2, a description of the constructor matching concept in Section 2.3, the definition of a conditional constructor in contrast to a default

Java constructor in Section 2.4, and the description of the process of runtime constructor generation in Section 2.5. In Section 3, we provide a formal basis of constructor matching, i.e., the matching from calls to superclass constructors to formal constructors of the superclass. We then present some use cases of our approach in Section 4 in order to proof its usefulness and applicability. Finally, information on the implementation is given in Section 5 which particularly comprises a description of the bytecode format which is produced.

2. The conditional constructors approach

2.1 Principles

Two design principles are the basis of the implementation of conditional constructors:

1. Initialization responsibility. Each class, and particularly each mixin, is responsible for calling a constructor of its superclass or the class it has been applied to, respectively. Imagine we have a mixin *ColoredShape* which introduces color attributes of a shape and we apply it to *Rectangle* from Figure 1. Then, according to this principle, the instantiated mixin has to call *Rectangle*'s constructor when the mixin constructor is called. An alternative design would be that the class which results from this mixin application, say *ColoredRectangle*, is responsible for calling both the constructor of the mixin *ColoredShape* and of *Rectangle*. This design, however, does not allow *ColoredShape* to have influence on the values which are passed to the *Rectangle* constructor which can be desirable as we will see in the use cases.

2. Abstraction from superclass constructors. A class can, in its constructor, call a constructor of a superclass without knowing the actual constructor and without even knowing the actual superclass. This principle is a necessary consequence of the first principle because, in order to have mixins calling the constructor of the class they are applied to, mixin constructors must abstract from a concrete superclass constructor since the concrete class to which a mixin is applied is unknown when the mixin is designed. We will provide syntactic and semantic means for implementing this abstraction in the further sections.

2.2 Argument patterns

With conditional constructors, the *super* call is not longer statically bound to a superclass constructor as it is known from standard Java. Also, for a sufficient high degree of abstraction from a concrete superclass constructor, further patterns are necessary when matching super calls against superclass constructors. In this section, we introduce the three types of patterns which can be used within super calls.

1. Java expressions. This is the type of arguments which is known from standard Java and which is also available with conditional constructors: A super call contains a certain number of expressions as arguments which have a certain

#	Arguments	Parameters	Matching	Types
1	`5, 0`	`int x, int y`	$\emptyset$	1
2	`5, 0`	`int x`	--	1
3	`x : 5, 0`	`int x, int y`	$\emptyset$	1, 2
4	`x : 5, 0`	`int a, int b`	--	1, 2
5	`p*`	`int x, int y`	`p*/(x, y)`	3
6	`p*, 0`	`int x, int y`	`p*/(x)`	1, 3
7	`p*, 0l`	`int x, int y`	--	1, 3

Table 1. Examples for constructor matching. The first two columns denote the input of the matching procedure: The *Arguments* column contains the arguments of the super call. The *Parameters* column contains the formal parameters the super call is matched against. The column *Matching* denotes the output: If the super call arguments and the formal parameters match, it contains a mapping from template arguments to formal parameters; if they do not, it contains "–". The last column enumerates the pattern types which are used in the respective argument list.

type each. This list of types is then matched against the formal parameter lists of the superclass constructors.

2. Named expressions. Argument expressions in super calls can be provided with names such that they only match a parameter of a superclass constructor if this formal parameter has the same name.

3. Template arguments. Template arguments are constituted by an identifier and match an arbitrary list of formal parameters. If such an argument is used in a super call, the very same template argument must appear in the signature of the constructor the super call belongs to. This means that template arguments are used for propagating constructor arguments from a superclass constructor to a subclass constructor.

The syntax for argument patterns conforms to the following rules:

- As with standard Java, multiple arguments in a pattern are separated with commas.
- The names of named arguments (type 2) are given in the form *<name> : <expression>*.
- Identifiers of template arguments are succeeded by an asterisk (*).

2.3 Constructor matching

The procedure of matching super calls against formal superclass constructors has the super call argument patterns and the parameter list of the formal superclass constructor as input and produces a matching as output if it exists. This matching is a function which maps template arguments to the formal parameters covered.

Table 1 gives examples for matching super calls against formal parameters of a superclass constructor. Rows 1 and 2 denote examples which are equivalent to method parameter matching in standard Java. Rows 3 and 4 give examples

	Java constructor	**Conditional constructor**
Parameter definition	identifier and type	identifier and type
		template parameter
Argument for constructor call	expression	expression
		named expression
		template argument
Remaining constructor instructions	no differences	

Table 2. Differences between default Java constructors and conditional constructors

with a named argument. In row 4, matching with the given formal constructor parameters fails because the first parameter has the name "a" and not "x" as required. Rows 5 to 7 contain template arguments: An argument pattern of the form *<name>** as in row 5 generally matches every possible constructor and the produced matching assigns the whole parameter list to the template argument *<name>**. The pattern in row 6 also matches because it requires the last parameter to have type int which is true. As the last parameter is set with an expression, *p** is mapped to the parameter *x* only. In contrast to row 6, row 7 requires the last parameter to have type long which is not true. As long is not casted to int automatically in Java, matching fails.

2.4 Conditional constructor

Like a default Java constructor, a conditional constructor is a special kind of method; however, there are differences as denoted in Table 2: As defined in Section 2.2, a conditional constructor does not only support expressions as arguments, but also named expressions and template arguments. Moreover, in the parameter definition of a conditional constructor, not only a typed identifier is allowed, but also a *template parameter* which is the counterpart of the template argument and allows for propagating formal parameters of the called constructor matched by a template argument into the parameter list of the conditional constructor. Therefore, template parameters appearing in the parameter list must also appear in the argument list of the super constructor call and vice versa.

As an example, consider the following conditional constructor which is contained in a class C:

```
public ? C(p*) {
    super(4, p*);
}
```

This constructor uses a template argument *p** which is propagated into the parameter list.

2.5 Constructor generation

Conditional constructors are not compatible with default Java since they may contain additional constructs as pointed out in the previous subsection. In contrast to conditional constructors, we call constructors which can be executed by a Java virtual machine *concrete*. In our approach, conditional constructors are converted to concrete constructors at runtime. We call this process *constructor generation*:

When loading a class C with superclass C'^{1}, the following algorithm is executed for every conditional constructor Con contained in that class and every – necessarily concrete – constructor Con' of C':

1. Perform constructor matching of Con against Con' as described in Section 2.3. If matching fails, continue with the next pair (Con, Con'). Otherwise, the obtained matching σ maps template arguments contained in the super constructor call of Con to a list of formal parameters which is a succesional sublist of the parameters of Con'.
2. Add a concrete constructor $CCon$ to C which is obtained by modifying Con as follows:
   (a) Replace each template parameter $p*$ of Con by the formal parameter list $\sigma\,(p*)$.
   (b) For each template argument $p*$ of Con, add instructions which load the arguments provided for the parameters $\sigma\,(p*)$.
   (c) Bind the super constructor call to Con'.

Finally, all conditional constructors are removed from C.

Obviously, this algorithm allows conditional constructors to match no superclass constructor, i.e., a conditional constructor is just removed without replacing it by a concrete constructor if there is no superclass constructor which matches the super constructor call. Furthermore, one conditional constructor can also match more than one superclass constructor which will result in multiple constructors being generated from one conditional constructor. Note that this can only be the case if the conditional constructor uses at least one template argument; otherwise, default Java type matching is applied which ensures that at most one constructor is matched.

3. Formalization of constructor matching

In Table 1, we presented some examples of constructor matching in order to provide an intuition about how super call arguments are matched against a formal superclass constructor. In this section, we now formalize this matching using a matching calculus which enables a computational decision of the question whether a matching exists and, if so, how the matching looks like. (Remember from Section 2 that a matching is defined as a function.)

[1] At runtime, the superclass of any class is known for sure. If it has not been loaded yet, it is loaded first which includes the constructor generation process. This ensures that the superclass only has concrete constructors.

3.1 Definitions

We first define the formal elements which will be the basis of the matching calculus:

The set of possible expressions of the target language (which is Java in our implementation) is denoted by E.

The set of valid variable labels is denoted by Lab.

The set of types is denoted by T.

The set of possible single argument patterns is defined as follows:

$$\begin{aligned} P := &\{(1,e) \mid e \in E\} \cup \\ &\{(2,p) \mid p \in Lab \times E\} \cup \\ &\{(3,l) \mid l \in Lab\} \end{aligned}$$

The number which is the first element of each pair in P indicates the pattern type: Type 1 are expressions, type 2 are named expressions, and type 3 are template arguments with the specified label.

The mapping of expressions to their most specific type is denoted by function $\tau : E \rightarrow T$.

The subtype relation is denoted by $\leq \subseteq T \times T$. It is reflexive, asymmetric, and transitive.

Furthermore, we define the union on partial functions f, g : $A \rightharpoonup B$ as follows:

$$\mathrm{f} \cup \mathrm{g} : A \rightharpoonup B : x \mapsto \begin{cases} \mathrm{f}(x) & \text{if } x \in \mathcal{D}(\mathrm{f}) \\ \mathrm{g}(x) & \text{if } x \in \mathcal{D}(\mathrm{g}) \text{ and } x \notin \mathcal{D}(\mathrm{f}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$\mathcal{D}(\mathrm{f})$ denotes the set of values for which the partial function f is defined.

3.2 Notations

For use in the following subsections, we introduce the following notations in order to facilitate reading the calculus rules:

- For $(1,e) \in P$, we write only e.
- For $(2,(l,e)) \in P$, we write $l : e$.
- For $(3,l) \in P$, we write l^*.
- For $(l,t) \in Lab \times T$, we write $l : t$.
- For $((p_1, ..., p_n), (l_1 : t_1, ..., l_k : t_k)) \in P^* \times (Lab \times T)^*$, we write $p_1, ..., p_n \doteq l_1 : t_1, ..., l_k : t_k$.
- For the empty partial function f with $\mathcal{D}(\mathrm{f}) = \emptyset$, we write $\emptyset$.
- For the partial function f with $\mathcal{D}(\mathrm{f}) = \{x_1, ..., x_n\}$ and values $v_i = \mathrm{f}(x_i)$, we write $\{x_1/v_1, ..., x_n/v_n\}$.
- For the empty tuple, we write $\emptyset$.

3.3 Constructor matching calculus

The constructor matching calculus consists of two components: 1. the calculus language and 2. the calculus rules. The rules eventually describe which words of the calculus language are derivable.

3.3.1 Calculus language

We define the calculus language as

$$\mathcal{L} := \left(P^* \times (Lab \times T)^*\right) \times (Lab \rightharpoonup Lab^*)$$

Intuitively, an element

$$((p_1, ..., p_n \doteq l_1 : t_1, ..., l_k : t_k), \sigma) \in \mathcal{L}$$

has the meaning that the list $p_1, ..., p_n$ of argument patterns matches the formal constructor with the parameters $l_1 : t_1, ..., l_k : t_k$ with a matching function $\sigma \in Lab \rightharpoonup Lab^*$ which provides a mapping from labels of template arguments in $p_1, ..., p_n$ to the matched labels in $l_1, ..., l_k$. We provide some examples for the calculus later.

We denote the subset of words in $\mathcal{L}$ which can be derived with the calculus with $\bar{\mathcal{L}}$.

3.3.2 Calculus rules

The calculus rules are provided in the following form:

$$<\text{name}> \frac{<\text{pre}_1> ... <\text{pre}_n>}{<\text{conclusion}>} <\text{side}>$$

The preconditions $<\text{pre}_i>$ as well as the conclusion must be elements of the calculus language $\mathcal{L}$ whereas the side conditions are independent of the calculus language.

The following rules are defined for the constructor matching calculus:

$$\text{type} \frac{}{(e \doteq l : t), \emptyset} \tau(e) \leq t \tag{1}$$

$$\text{name} \frac{(e \doteq l : t), \sigma}{(l : e \doteq l : t), \sigma} \tag{2}$$

$$\text{empty} \frac{}{(\emptyset \doteq \emptyset), \emptyset} \tag{3}$$

$$\text{template} \frac{}{(l* \doteq l_1 : t_1, ..., l_n : t_n), \{l * / (l_1, ..., l_n)\}} (*) \tag{4}$$

$$\text{compos} \frac{\begin{matrix}(\quad p_1 \quad \doteq \quad l_1 : t_1, \quad ..., l_k : t_k), \sigma_1 \\ (p_2, ..., p_m \doteq l_{k+1} : t_{k+1}, ..., l_n : t_n), \sigma_2\end{matrix}}{(p_1, ..., p_m \doteq l_1 : t_1, ..., l_n : t_n), \sigma_1 \cup \sigma_2} \begin{matrix} m \geq 2 \\ (*) \\ (**) \end{matrix} \tag{5}$$

$$\begin{aligned} (*) &= \forall i, j \in \{1, ..., n\} : i \neq j \Rightarrow l_i \neq l_j \\ (**) &= \forall i \in \{k+1, ..., n\} : \forall \sigma_1', \sigma_2' \in Lab \rightharpoonup Lab^* : \\ &\left(\begin{matrix} ((p_1 \doteq \quad l_1 : t_1, \quad ..., l_i : t_i), \sigma_1') \notin \bar{\mathcal{L}} \vee \\ ((p_2 \doteq l_{i+1} : t_{i+1}, ..., l_n : t_n), \sigma_2') \notin \bar{\mathcal{L}} \end{matrix} \right) \end{aligned}$$

The rule *type* (1) describes the usual method parameter matching as known from most object-oriented languages: Expression e can be used for a parameter with type t if the type of e is t itself or a subtype. *name* (2) extends this for named arguments which only match a parameter with the same name. *empty* (3) defines the empty argument list to match the empty parameter list. The rule *template* (4) expresses that a single template argument matches an arbitrary list of formal parameters as long as the labels of these parameters are mutually different. *compos* (5) allow lists of arguments to match lists of formal parameters whose labels must be disjunct again. The side condition $(**)$ ensures that the first argument pattern always matches the longest possible sequence of formal parameters (cf. the example in the following subsection and Figure 2).

3.4 Applying the constructor matching calculus to the constructor matching problem

Now that we have defined the constructor matching calculus, how can we apply it to the constructor matching problem? Remember that, in Section 2, we defined the matching procedure with its inputs and outputs: It takes the argument patterns and the formal parameters against which the patterns should be matched; it produces a matching function if one exists.

First, note that we can express an instance of the constructor matching problem, i.e., the input of the matching procedure, as an expression $\mathcal{E} = p_1, ..., p_m \doteq l_1 : t_1, ..., l_n : t_n \in P^* \times (Lab \times T)^*$. Further, we need the following two results for the constructor matching calculus:

THEOREM 1 (Rule uniqueness). *For each instance $\mathcal{E}$ of the constructor matching problem, there is exactly one calculus rule which might produce the calculus word $(\mathcal{E}, \sigma)$ with some matching function σ. The instances $\mathcal{E}_1, ..., \mathcal{E}_n$ of constructor matching problems in the preconditions $(\mathcal{E}_1, \sigma_1)$, ..., $(\mathcal{E}_n, \sigma_n)$ of the only applicable rule are uniquely determined by the instance $\mathcal{E}$.*

Proof (sketch). Let $\mathcal{E} = p_1, ..., p_m \doteq l_1 : t_1, ..., l_n$. There are three cases to distinguish for the value of m: 1. $m = 0$: Then, the only possible rule is *empty* due to the empty argument list. 2. $m = 1$: Then, for every argument pattern type of p_1, there is exactly one possible rule (*type* for an expression, *name* for a named expression, and *template* for a template argument). 3. $m > 1$: Then, the only possible rule is *compos* which is the only one producing a list of argument patterns. The preconditions of rules *type*, *name*, *empty*, and *template* are trivially determined uniquely by $\mathcal{E}$. The rule *compos* ensures this uniqueness with its side condition $(**)$.

THEOREM 2 (Matching uniqueness). *If, for an instance $\mathcal{E}$ of the constructor matching problem, the calculus word $(\mathcal{E}, \sigma)$ is derivable for a matching function σ, then σ is unique, i.e., there is no $\sigma' \neq \sigma$ such that $(\mathcal{E}, \sigma')$ is derivable.*

Proof (sketch). From Theorem 1, we know that, if $(\mathcal{E}, \sigma)$ has been derived, the rules which have been applied in this derivation are uniquely determined. The rule applications at the leaves of the derivation must have empty preconditions: either *type*, *empty*, or *template*. The matching functions produced by these rules are uniquely determined by the produced instance of the pattern matching problem. For rules *name* and *compos*, the produced matching function is uniquely determined by the matching functions in the precondition. Hence, with induction over rule application, we can argue that the resulting matching function is uniquely determined.

These theorems give us the result that the matching function σ is uniquely determined by a constructor matching problem instance $\mathcal{E}$ and they also give a hint on how to obtain σ from $\mathcal{E}$ using the constructor matching calculus algorithmically: We can build a derivation tree for $\mathcal{E}$, ignoring the matching functions in the calculus words. (If we get stuck, i.e., there is no applicable rule, there is no matching.) Afterwards, from the leaves of the derivation tree to its root, we can construct the matching functions by applying the calculus rules.

Figure 2 gives an example for the application of the calculus to the following instance of the constructor matching problem:

$$p*, 4, q*, x : 5, r* \doteq a : \text{char}, b : \text{int}, c : \text{int}, x : \text{int}, s : \text{String}$$

As a derivation exists, a matching is possible, namely with the matching function in the root of the derivation tree:

$$\{p*/(a,b), q*/\emptyset, r*/(s)\}$$

Note that one might think of an alternative derivation tree yielding the matching function $\{p*/(a), q*/(c), r*/(s)\}$ and assigning the argument 4 to the formal parameter b. However, this would contradict the side condition $(**)$ of *compos* in the very last derivation step because $p*$ must match the longest possible sequence of formal parameters which is, in this case, a and b.

4. Use cases

Use cases of conditional constructors can basically be divided into two parts: One part addresses problems which can appear with default Java classes such as the constructor explosion problem mentioned in the introduction. The other part regards to mixins and the problems arising from the fact that a mixin developer does not know the superclass the mixin will be applied to and, therefore, she does not know the constructors available in the superclass.

$$(*) = \text{compos}\ \dfrac{\text{type}\ \dfrac{}{(4 \doteq c : \text{int}),\emptyset} \quad \text{compos}\ \dfrac{\text{template}\ \dfrac{}{\begin{matrix}(q* \doteq \emptyset),\\ \{q*/\emptyset\}\end{matrix}} \quad \text{compos}\ \dfrac{\text{name}\ \dfrac{\text{type}\ \dfrac{}{(5 \doteq x : \text{int}),\emptyset}}{(x : 5 \doteq x : \text{int}),\emptyset} \quad \text{template}\ \dfrac{}{\begin{matrix}(r* \doteq s : \text{String}),\\ \{r*/(s)\}\end{matrix}}}{(x : 5, r* \doteq x : \text{int}, s : \text{String}), \{r*/(s)\}}}{(q*, x : 5, r* \doteq x : \text{int}, s : \text{String}), \{q*/\emptyset, r*/(s)\}}}{(4, q*, x : 5, r* \doteq c : \text{int}, x : \text{int}, s : \text{String}), \{q*/\emptyset, r*/(s)\}}$$

$$\text{compos}\ \dfrac{\text{template}\ \dfrac{}{(p* \doteq a : \text{char}, b : \text{int}), \{p*/(a,b)\}} \quad \text{compos}\ \dfrac{(*)}{(4, q*, x : 5, r* \doteq c : \text{int}, x : \text{int}, s : \text{String}), \{q*/\emptyset, r*/(s)\}}}{(p*, 4, q*, x : 5, r* \doteq a : \text{char}, b : \text{int}, c : \text{int}, x : \text{int}, s : \text{String}), \{p*/(a,b), q*/\emptyset, r*/(s)\}}$$

Figure 2. Example for the application of the constructor matching calculus

4.1 Use cases without mixins

4.1.1 The constructor explosion problem

Consider the example from Figure 1 again which has been explained in the introduction. The goal is to inherit the constructors from *Shape* in *Rectangle*, but extend them with additional parameters. With conditional constructors, one can define the *Rectangle* constructors as follows:

```
public ? Rectangle(p*, int width,
        int height) {
    super(p*);
    this.width = width;
    this.height = height;
}
public ? Rectangle(p*, Dimension dim) {
    this(p*, dim.width, dim.height);
}
```

The template parameter *p** enables inheriting the parameters from *Shape*. With this solution, we only need to write two *Rectangle* constructors, but four concrete constructors will be generated when loading the class, namely the four shown in Figure 1. In regard to the example code, note that argument patterns can also be applied to **this** constructor calls.[2]

Another advantage of this solution is that, if the constructors of *Rectangle* change or if constructors are added to or removed from *Rectangle*, the *Shape* class must not be adapted and will automatically adopt these constructor changes during runtime.

4.1.2 Cascading constructor inheritance

Consider a hierarchy as presented in Figure 3: The class *ExtendMe* is assumed to be part of a library. It is the base class for subclasses *Subclass_1*, ..., *Subclass_n*. All of them have a constructor which takes an argument *param* and forwards it to the superclass in the hierarchy so that this argument is finally received by the constructor of *ExtendMe*. The class *Client* instantiates the class *Subclass_n* and thereby passes an appropriate argument to its constructor.

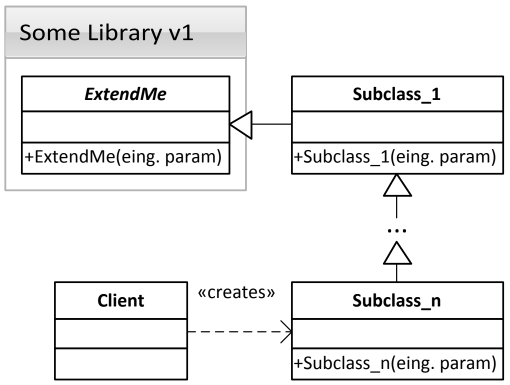

Figure 3. Example for the cascading constructor inheritance use case

Now imagine a new version of the library to be published and integrated into the hierarchy. This new version adds a parameter *param2* to the constructor of *ExtendMe* which means that the old constructor taking only one argument is not available any more. As a consequence, *Subclass_1* must change its super constructor call in order to pass two arguments. Further imagine that it is desirable that the new parameter is set by the client when instantiating *Subclass_n*, i.e., the new argument shall be forwarded through the hierarchy as the old one was before. Then, using default Java constructors, the constructor of each of the n subclasses must be changed in order to accept and forward the new argument to its respective superclass, and finally, the client has to be changed, too, of course. This yields changes to a total number of $n + 1$ classes which is an indicator of rigidity.

Conditional constructors can help with tackling this problem of rigidity: For that purpose, conditional constructors

[2] For avoiding infinite recursion, a conditional constructor with a **this** constructor call may only call constructors which are defined syntactically before the current constructor.

must be employed from the early beginning of design so that the constructor of *Subclass_i* looks as follows:

```
public ? Subclass_i(p*) {
    super(p*);
}
```

This is a quite trivial application of a conditional constructor which simply forwards all received arguments to the superclass constructor. Of course, if necessary, single classes may also add certain parameters; the important thing is to use a template argument for the parameters of the *ExtendMe* constructor. If this scheme has been applied and then the *ExtendMe* constructor changes as described before, no changes are required to *Subclass_1*, ..., *Subclass_n*; only the client must be adapted in order to pass a new appropriate argument to the constructor of *Subclass_n* which cannot be avoided. Hence, conditional constructors reduce the number of classes to be changed in this scenario from $n+1$ to 1. Furthermore, this one class to be changed does not have to be the client; it can also be one of the subclasses which can be changed so that it passes another argument to the superclass constructor.

However, the great advantage of this solution, namely less rigidity, is also the greatest disadvantage as it introduces a slightly higher degree of fragility which is related to the fragile base class problem [5]: According to this problem, changes to a superclass might affect subclasses in an undesired way. With respect to constructors, this is only partially true in default Java: When a superclass constructor changes, only its direct subclasses may cause an error if they rely on the constructor. With conditional constructors, indirect subclasses can be affected by this kind of change, too. Moreover, also clients of indirect subclasses can be affected because they might now depend on constructors of indirect superclasses of instantiated classes. However, this problem must be seen in the context that reliance on indirect superclasses of used classes already exists in default Java, namely for public methods, and could only be avoided for constructors by forbidding constructor inheritance.

4.2 Use cases with mixins

4.2.1 Propagating parameters of the superclass

This use case addresses the case that a new class C' is generated by applying a mixin M to a superclass C and the constructors of C should be available in the new class C'. As a concrete example, we introduce the mixin DrawLogging:

```
public cclass DrawLogging extends Shape {
    public ? DrawLogging(l*) {
        super(l*);
    }
    // logging functionality
}
```

The purpose of the mixin is to perform some logging operations whenever the *Shape* it is applied to is being drawn. (Assume *Shape* to provide an abstract method *draw(...)* or similar.) When we assume that *DrawLogging* shall be applied to a class *Rectangle* such as presented in Figure 1, then it is desirable that the generated class provides exactly the constructors which are provided by *Rectangle*. With conditional constructors, this can be achieved in two steps: 1. The constructors of the superclass of *DrawLogging* are propagated into the mixin class itself. This is realized with the template argument *l** in the above listing. 2. The constructors of the mixin class can then be propagated into the generated class in the very same fashion:

```
public cclass LoggedRectangle extends
        DrawLogging &[3] Rectangle {
    public ? LoggedRectangle(l*) {
        super(l*);
    }
}
```

This will ensure that *LoggedRectangle* has exactly the same constructors as Rectangle. Of course, neither the mixin class *DrawLogging* nor the product class *LoggedRectangle* has to be recompiled when the class *Rectangle* changes.

For completeness, it should be mentioned that mixins such as *DrawLogging* can not only propagate the parameters of a superclass, but also extend the parameter list by adding further parameters. For example, one could think of a *DrawLogging* constructor which requires an output file argument:

```
public ? DrawLogging(l*, File output) {
    super(l*);
    // do something with output
}
```

4.2.2 Influencing the initialization of the unknown superclass

A mixin might wish to influence the initialization of the class it is applied to. *Influencing the initialization* is typically done by passing certain arguments to the constructor of the initialized class; however, a mixin does not know the concrete superclass it is applied to.

As an example, consider a mixin *OriginShape* which always wants to set the *x* and *y* parameters of the *Shape* superclass to 0. With conditional constructors, we can formulate the constructor for this mixin as follows:

```
public ? OriginShape(l*) {
    super(0, 0, l*);
}
```

This constructor will set the first two parameters to 0 and propagate the remaining parameters of the superclass. E.g., applied to the class *Rectangle* from Figure 1, it will match two *Rectangle* constructors, namely those which take *x* and

[3] *A & B* is a CaesarJ construct which makes B become a superclass of A.

y parameters, and yield one constructor for each of them, the first one taking *width* and *height* parameters, and the second one taking a *dimension* parameter.

Note, however, that this constructor is based on the implicit assumption that each constructor of a subclass of *Shape* takes the *x* and *y* coordinates of the shape as its first two parameters. What happens if the developer of *Rectangle* decides to write a constructor taking the parameters *width*, *height*, *x*, *y*, i.e., with *x* and *y* not being the first parameters? In this case, the constructor of *OriginShape* will set *width* and *height* to 0 while propagating *x* and *y* which is clearly not the desired behavior.

As a possible solution, conditional constructors allow to make the assumption explicit by providing names for the instantiated parameters:

```
public ? OriginShape(l*) {
  super(x : 0, y : 0, l*);
}
```

With this modification, *OriginShape* is, in fact, only applicable to classes which have at least one constructor where *x* and *y* are the first two parameters. Moreover, one can reformulate the constructor in order to weaken the assumption so that it does not rely on *x* and *y* being the first parameters. For that purpose, another template argument is added:

```
public ? OriginShape(l*, m*) {
  super(l*, x : 0, y : 0, m*);
}
```

The only noteworthy remaining assumption of this constructor is that parameters of *Shape* are not renamed in subclasses, i.e., *x* and *y* parameters of a *Shape* subclass constructor will always refer to the *x* and *y* parameters in the *Shape* constructor. Yet, this seems to be a reasonable assumption, but we would suggest this, as any kind of assumption, to be mentioned in the documentation of the *OriginShape* mixin.

5. Implementation

The implementation of conditional constructors is based on the CaesarJ [1] implementation in version 0.9.0[4] which itself leverages the Java-based JastAdd compiler [3], a compiler system which allows for modular definition of language constructs. It supports both static type checking and Java bytecode generation. For execution of the specialized bytecode format, we applied the ASM bytecode toolkit [2] in version 4.0.

The general tooling workflow of conditional constructors is as follows:

1. JastAdd: Type checking of the given source code files
2. JastAdd: Java bytecode generation from the given source code files
3. ASM: Loading classes using a specialized Java classloader

The following subsections go through these steps in detail.

5.1 Type checking

During type checking, the existence of constructors matters in two places of the abstract syntax tree generated by JastAdd: 1. with class instance expressions (**new** C (...)) and 2. with super calls within constructors.

For class instance expressions, our implementation statically checks if an appropriate constructor exists in the instantiated class. For this purpose, the constructor generation process as described in Section 2 is executed for conditional constructors contained in that class in order to obtain all concrete constructors. One of these constructors or one of the non-conditional constructors contained in the class must match the class instance expression. If this is not the case, a compiler error is produced in order to ensure that the class instantiation will succeed at runtime.

Super calls within constructors are handled differently depending on whether contained in a conditional or a non-conditional constructor: Super calls of non-conditional constructors are always checked for whether they reference an existing non-conditional constructor in the superclass. If this is not the case, a compiler error is produced. In contrast, super calls of conditional constructors are not checked immediately. Instead, availability of non-conditional constructors is checked during the checks for class instance expressions as mentioned in the previous paragraph. The constructor generation procedure which is included in these checks ensures that the super calls of concrete constructors generated from conditional constructors always reference an existing constructor of the superclass.

Nevertheless, runtime errors regarding constructors cannot be completely ruled out with the above checks: The concrete constructors which are generated from conditional constructors actually depend on the constructors which are available in superclasses in the particular hierarchy. Thus, if constructors in a hierarchy change, the available constructors in a subclass may change. We have pointed out this problem in Section 4.1.2. Yet, note that runtime errors can, in a similar way, also occur in default Java programs with respect to inherited methods which also depend on changes in superclasses.

5.2 Bytecode generation

The generated bytecode mostly equals the bytecode as it would be produced by a default Java compiler[5]. Additionally, CaesarJ adds a class attribute with information about

[4] `http://www.caesarj.org/index.php/Caesar/CVS`

[5] `http://docs.oracle.com/javase/specs/jvms/se5.0/html/VMSpecTOC.doc.html`

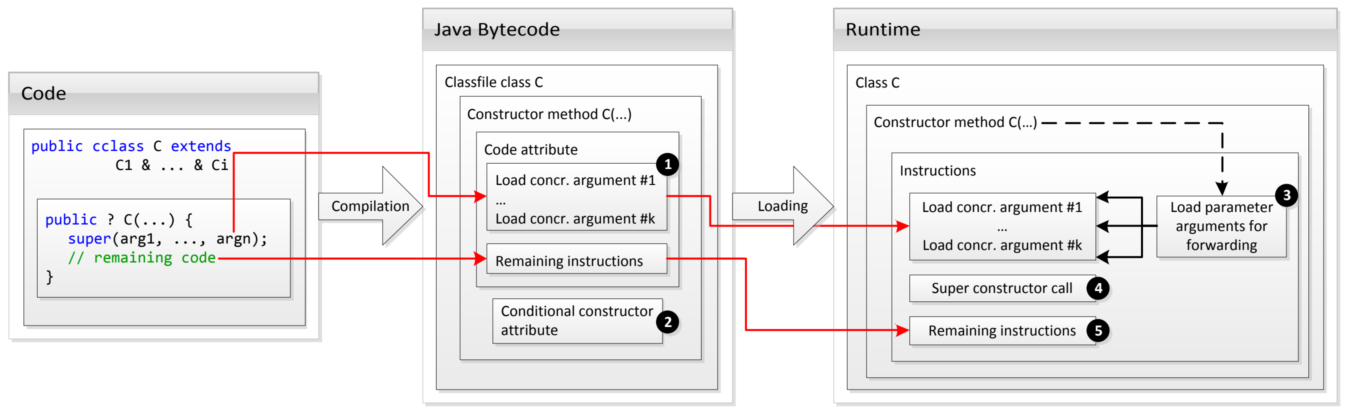

Figure 4. Process of compiling and loading conditional constructors

multiple superclasses. The left part of Figure 4 drafts the process of bytecode generation for conditional constructors: They are defined as normal constructor methods like in default Java. The differences are as follows:

- The method signature does not declare any method parameters.
- The method code does not contain a super call at all. In standard bytecode, this is located between the instructions for loading the arguments and the remaining constructor instructions.
- However, the super call is prepared by loading the expression arguments (cf. Section 2) of the super call onto the stack (1). Names of named argument expressions are ignored here as well as template arguments. Therefore, the super call in the code contains n arguments, but, in the code, only k ($k \leq n$) arguments are loaded, namely the expression arguments.
- A special conditional constructor attribute (2) is provided which defines the parameters of the conditional constructor, including identifiers of parameters and template parameters, and the full list of the super call arguments, including names of named expressions and template arguments.

The compiled classfile is saved with the suffix ".cjclass" in contrast to the normal ".class" suffix. This was inherited from the CaesarJ implementation and allows for distinguishing normal from CJ classfiles during runtime.

5.3 Classloading

CaesarJ classfiles must be processed in a specialized manner as CaesarJ classes can have multiple superclasses and, with our extension, they can have conditional constructors, two features which cannot be handled by the default classloader of Java. During this process, CaesarJ performs a linearization of class hierarchies with multiple inheritance. As a consequence, each class is loaded once for every occurrence of that class in a class hierarchy. For example, if a class A is instantiated without non-trivial superclass in one place and is instantiated as part of a hierarchy in which it is subclass of a class B, then A is loaded twice so that the correct direct superclass can be set by the CaesarJ classloader in each case.

In our implementation, we leveraged this classloading procedure for generating conditional constructors at runtime. The procedure of generating a conditional constructor from bytecode is proposed in the right part of Figure 4: The conditional constructor attribute contained in the bytecode (2) gives all information that is needed to perform constructor matching against the superclass constructors. At this point, the constructor generation procedure as introduced in Section 2 is executed. For instantiating a concrete constructor from the conditional constructor, the correct constructor signature is created using the full parameter list contained in the conditional constructor attribute and according to the matching which maps template parameters to the matched parameters of the superclass constructor. Instructions are then created which load the arguments to be forwarded to the superclass constructor (3). These instructions are interleaved with the instructions for loading expression arguments which are already contained in the bytecode (1). Instructions for the invocation of the matched superclass constructor are inserted (4). The remaining instructions of the constructor are just retained from the bytecode (5).

Note that a conditional constructor can match multiple superclass constructors. This means that a conditional constructor is actually duplicated at runtime and provided with an appropriate super call for each matching superclass constructor.

5.4 Source code structure

The source code of the implementation is available at github[6] in branch *mixin-constructors*. The directory structure is as follows:

java: JastAdd source files which represent the syntax and semantic of the Java programming language

[6] `https://github.com/tud-stg-lang/caesar-jastadd`

caesar/mixincomp: JastAdd source files for the mixin composition extension of CaesarJ

caesar/condconstructors: JastAdd source files for the extension of conditional constructors

build.xml: Ant build file which applies JastAdd to the JastAdd source files and generates Java source files from them which represent the abstract syntax tree of the complete language including Java, CaesarJ extensions, and conditional constructors

caesar/src: Java source files of the CaesarJ compiler, which leverages the generated JastAdd files, and the CaesarJ classloader which has been adapted for conditional constructors

test/src: Java source files for executing tests that are contained in *test/suites*

test/suites: tests in a special XML format for testing Java, CaesarJ, and conditional constructor features

tools: libraries

doc: this document, the paper *Constructors for Multiple Inheritance* [4] which is the origin of this work, and further CaesarJ documentation

Acknowledgments

Thanks to Vaidas Gasiunas for some of the basic ideas of conditional constructors.

References

[1] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An overview of caesarj. In *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer, 2006. URL `http://dx.doi.org/10.1007/11687061_5`.

[2] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[3] T. Ekman and G. Hedin. The jastadd system – modular extensible compiler construction. *Sci. Comput. Program.*, 69(1-3):14–26, December 2007. ISSN 0167-6423. doi: 10.1016/j.scico.2007.02.003. URL `http://dx.doi.org/10.1016/j.scico.2007.02.003`.

[4] M. Martin. Constructors for multiple inheritance. Seminar on *Implementaiton of Programming Languages*, summer term 2011, Software Technology Group, Technische Universität Darmstadt, 2011.

[5] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, ECCOP '98, pages 355–382, London, UK, UK, 1998. Springer-Verlag. ISBN 3-540-64737-6. URL `http://dl.acm.org/citation.cfm?id=646155.679700`.