Northeastern University - College of Computer Science - Technical Report NU-CCS-00-01

Automated Architectural Transformation: Objects to Components

David H. Lorenz
Northeastern University
College of Computer Science
Boston, MA 02115 USA
lorenz@ccs.neu.edu

John Vlissides
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598 USA
vlis@us.ibm.com

Technical Report NU-CCS-00-01
April 2000

Abstract

Changing an existing software architecture is difficult and painful. The typical result is either a poor adaptation to the target architecture or a substantial debasement of the original. Software generators can simplify the transformation process and relieve the pain for certain architectures. We present a component generator that transforms an object-oriented specification to a component-based implicit-invocation system. The input to the transformation is a set of classes; the output is a package of components. We present this process for a given source specification and target component technology and demonstrate its advantages over conventional approaches.

1 Introduction

Changing the architecture of an existing software system is both difficult and painful. It is difficult because it demands a conceptual shift to a different style, even a different programming paradigm. It is painful because large parts of the system invariably need to be redone. All too often, a good initial architecture is seriously degraded as a result of inexperience with the target paradigm, or the original is kludged into minimal conformance with the target, prompting constant attention to maintain the conformance.

The shift from procedural to object-oriented programming illustrates the problem. Experienced C programmers who take up C++ tend to continue programming in C using C++ syntax. A lot of time and experience must accrue before the programmer thinks naturally in objects. In the interim, an existing procedural library will be wrapped in a class definition rather than rewritten to take advantage of objects. Even with the right skills, legacy software is retained to avoid the cost of redesign—even at the expense of a clean design and an efficient implementation.

This paper focuses on the transition beyond object-oriented to component-based programming [30]. We present a component generator that transforms an object-oriented, explicit-invocation description into a component-based implicit-invocation [29, 11] architecture. The generator implements a set of transformation rules. The input to the generator is a set of classes in Java; its output is a package of components—a `jar` of Java Beans—ready for use.

The ability to transform explicit method calls to implicit event-based method invocations is useful in many ways. Not only can this transformation overcome the architecture mismatch [10] between classes, but it can also resolve incompatibilities among development environments. Transforming existing classes to JavaBeans, for example, allows them to be used with GUI components in the event-based Bean Development Kit (BDK [17]) environment. Even if no GUI elements are used, manipulating erstwhile nonvisual software visually can be a boon to rapid prototyping and testing [25, 26]. A component assembly tool such as the BeanBox [16] becomes a tool for general-purpose visual programming in the small.

Transforming a class hierarchy into a component-based system is a special case of generating compo-

Northeastern University - College of Computer Science - Technical Report NU-CCS-00-01

nents from a concise yet precise description. A code generator produces code quickly and consistently—desirable properties for any architectural transformation. Generation thus simplifies the architectural transition process and eases the pain.

This work is a proof of concept: Under certain conditions, components can be generated from specifications—an intractable task in general. This work achieves this strong result by narrowing the problem's scope. We start with classes as our specification, where *specification* here refers to an imperative description rather than a declarative one, such as pre- and postconditions. Moreover, the transformation preserves the method's (byte)code (i.e., the generated components call the original method). A precise interface description combined with a specification of the desired transformation is sufficient for an entirely automated transformation process.

2 A Simple Example

There are many ways to map classes and methods to components and events. A straightforward mapping would associate each class with a component and each method (message) with an event. While this mapping is natural in modeling object-oriented interactions, it is not always appropriate in component-based programming. Returning a value from a method call, for example, would require additional mechanism, such as a callback event, for synchronization.

Consider the class hierarchy of Figure 1. *Bool* is an interface that both classes True and False implement. For simplicity, the protocol contains only two methods, the `not` operation and the `nand` operation, with the usual short-circuit Boolean semantics.

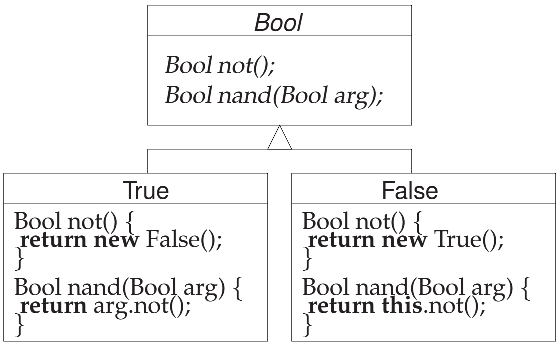

Figure 1: The Boolean class hierarchy

2.1 A Natural But Undesirable Mapping

This mapping turns the methods `not` and `nand` into the events NotEvent and NandEvent, which are kinds of *InvokeEvent* (see Figure 2). Two other events not shown in the figure are NewEvent for constructors and ReturnEvent for returning a value. The classes True and False become the components TrueBean and FalseBean (Figure 3). The incoming arrows NewEvent, NotEvent, and NandEvent in Figure 3 indicate that BoolBean implements the *NewEventListener*, *NotEventListener*, and *NandEventLister* interfaces—that is, TrueBean and FalseBean may receive these events. The outgoing ReturnEvent arrow means that BoolBean subclasses can broadcast the result to potential ReturnEventListeners.

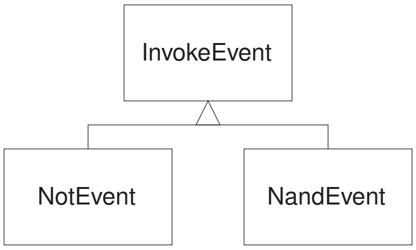

Figure 2: Method-to-Event mapping

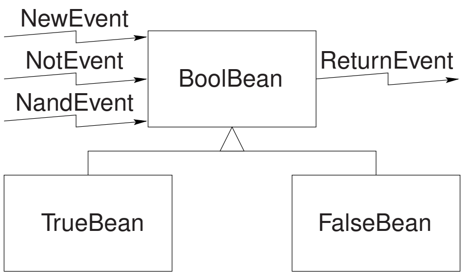

Figure 3: Class-to-Component mapping

The main problem with this mapping is its inflexibility. The TrueBean and FalseBean carry fixed behaviors resembling the objects *true* and *false*, respectively, in SMALLTALK [12]. This is a useful object-oriented design, wherein objects can send messages and serve as return values, but it makes a poor component-based design, rendering simple things like constructing a function with arguments (e.g., $nor(x, y)$) a challenge.

On receipt of a ReturnEvent, moreover, it is difficult for the receiver to tell whether the event resulted from a `not` operation or a `nand` operation. One might consider splitting the ReturnEvent into two

Northeastern University - College of Computer Science - Technical Report NU-CCS-00-01

events, such as NotReturnEvent and NandReturnEvent. But that has the unfortunate side-effect of creating a parallel set of components solely for return values, which complicates maintenance. It also complicates parameter passing, demanding enough settable properties to accommodate the method with the maximum number of arguments. Worst of all, adding an operation (say, `nor`) requires changing BoolBean as shown in Figure 4. And when a component's class changes, every instance of the component must be updated—unless you're willing and able to manage multiple component versions.

Figure 4: Adding a `nor` operation

2.2 An Alternate Mapping

Figures 5 and Figure 6 illustrate a different mapping, where each *method* becomes a component and each *class* becomes an event. Compared to the earlier mapping, the components True, False, Nand, and Not in Figure 6 are much better building blocks for a Boolean expression like that of Figure 8. In this method-to-component mapping, returning a value simply corresponds to firing an event. Adding a new `nor` operation is also simple: simply add another kind of component. The resemblance to hardware components is intentional.

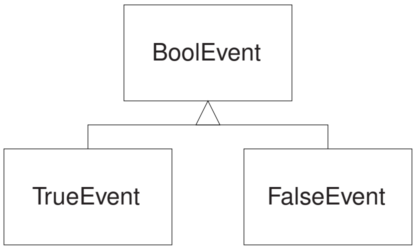

Figure 5: Class-to-Event mapping

Note that the argument of `nand` is supplied in the component Nand by binding its property. Also note that these components are not always convenient enough by themselves. Additional components are desirable, such as the adapters shown in Figure 7 for converting a property to an event and vice versa. A tool that automates this mapping should supply such components as well.

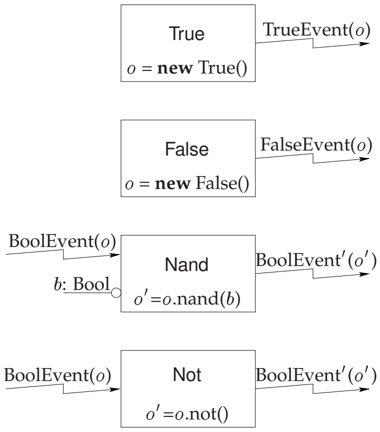

Figure 6: Method-to-Component mapping

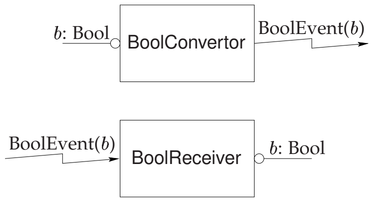

Figure 7: Event-Property adaptors

There a few subtle mapping issues to consider as well. One regards inherited methods: Should each implementation of a method signature yield a new component?

It turns out that the components can be generated from abstract classes or even from interfaces. To see how, examine again the *Bool* interface in Figure 1. When the signature `Bool not()` is introduced, the Not component shown in Figure 6 can be generated even before examining the classes True and False, which implement the *Bool* interface. That's because the component encapsulates the call to `not`. When a concrete object *obj* arrives via a BoolEvent *evt*, the Not component will invoke `evt.obj.not()`. Java's dynamic binding will choose the correct implementation of the specialized `not` method at run-time according to the actual type of *obj*.

Overriding raises another question. Should a method that overrides another yield a new component? Suppose you override the `nor` operation with an alternative implementation, say, one that forgoes short-circuit semantics. Do you need to generate another Nor component?

Recall that the component's polymorphic code makes a new component unnecessary, even if Nor was generated from an interface. But there's still a difficulty: Not cannot work with components of

Northeastern University - College of Computer Science - Technical Report NU-CCS-00-01

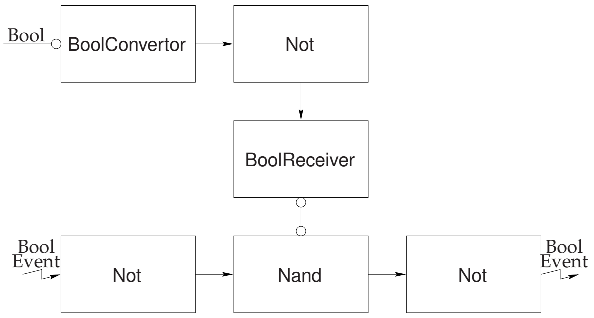

Figure 8: $nor(x, y) = not(nand(not(x), not(y)))$

```
class Class {
    public static Class forName(String className)
        throws ClassNotFoundException;
    public native Class getSuperclass();
    public Object newInstance();
        throws InstantiationException,
            IllegalAccessException;
    ...
}
```

Figure 9: The metaclass `java.lang.Class`

more specialized types without adapters that circumvent Java's strong typing of events. It's all right to connect a component generated from a specialized method to one generated from an interface method. Sending an event the other way requires an adapter, however, because the event types in Java must match, independent of event payload. Fortunately, the adapter's downcast always succeeds, as the original types are compatible.

In summary, it is sufficient to generate components for only declared methods in a class or interface; inherited methods may be ignored. Even if the strong typing of the events poses too great a difficulty, and adapters are undesirable, then generating a new component for overriding methods remains an option (it is equivalent to wrapping the component between two adapters.) Because generation is automatic, the immediate cost is the same, although there may be consequences in the long run—namely, an overabundance of components.

Last but not least, we convert exceptions to events. Exception classes become events just like other classes. If a method can throw an exception, the corresponding component will let exception listeners subscribe to that exception. [1]

3 A More-Complex Example

Consider the metaclass `java.lang.Class`, listed partly in Figure 9. Applying the method-to-class transformation on this class (and other classes in `java.lang.reflect`) would produce a system of components for expressing simple reflective computations. The method `getSuperclass`, for example, would be transformed into a component that receives a ClassEvent [2] (carrying a class object) and then fires a ClassEvent (carrying the corresponding superclass object).

A toolkit of *ReflectionBeans* has proven useful for building applications that compute simple object-oriented metrics or find dependencies among classes [24, 23]:

- Obtaining program information through reflection instead of parsing source code affords the benefit of working with compiled "`.class`" files. The metrics can be applied to third-party components whose source code is unavailable.
- Run-time analysis using reflection rather than analyzing the parsed source code allows simple dynamic checks in lieu of complicated static analysis, as we demonstrate later.
- Assembling components is usually easier than writing reflection code. ReflectionBeans allow quick, graphical construction of simple but meaningful analyses, making it easy to experiment with different metrics.

Checking for various instantiability problems that may cause JavaBeans to fail loading into a development environment illustrates the tradeoffs. Ensuring instantiability of a bean class statically is difficult. Many things can be wrong: the constructor might be private, the superclass's constructor might be private, one of the arguments to the constructor might be private—any one of which will render the class uninstantiatiable. Meanwhile, if a default constructor is missing, the bean cannot be added to the builder. Analyzing the source code statically is an option only if the source code is available, and even a simple analyses like looking for instantiability problems remains a complex task, just because of the inherent complexity of parsing.

Instead of parsing, one can use reflection. Applying reflection to analyze ".class" files and report potential problems is more flexible and less tedious:

[1] Not including *unchecked exceptions*, i.e., `RuntimeException` and `Error` are not detected.

[2] CLASSEVENT, Figure 24 in Appendix A.

Northeastern University - College of Computer Science - Technical Report NU-CCS-00-01

You have the flexibility of analyzing class without their source code. Some problems that are normally flagged by the compiler during compilation (e.g., the compiler checks the visibility of superclass's constructors) may be assumed correct (although verification is always desired). Indeed, a troubleshooting tool called BeanLint [18, 19], takes this approach and uses reflection to to alert about potential problems with JavaBeans classes.

Alternatively, instead of statically checking for all the instantiability problems, one can also use reflection attempting to instantiate the class and catching any exceptions. Moreover, this can be done at component assembly-time by simply using ReflectionBeans components. DIN (depth of inheritance [4, 1, 21, 13]) is a useful metric that illustrates this approach. Computing a hierarchy's DIN involves nothing more than going up the superclass chain and counting the classes. Extending DIN to simultaneously check whether or not an ancestor class can be instantiated is a reliable way to ensure fulfillment of the "*abstract superclass rule*" [14] more accurate than merely counting the classes explicitly marked "abstract."

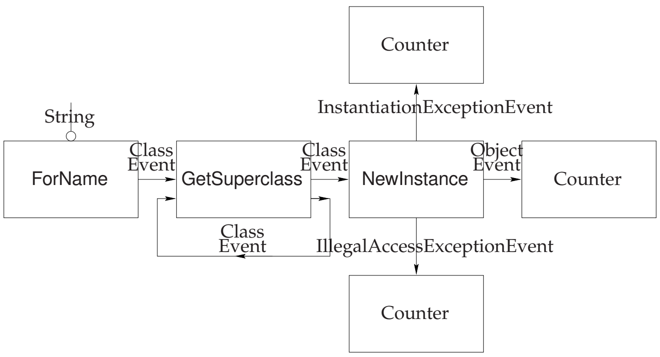

Figure 10: To bean or not to bean?

In Figure 10, we connect ForName [3] to GetSuperclass,[4] and GetSuperclass to NewInstance,[5] repeating the event to GetSuperclass using a feedback loop. Counters are connected to the output and to exception events. The reported total tells you how many superclasses are concrete, and the number of exceptions tells you how many are non-instantiatable and hence abstract *de facto*. *Instantiation* exception is thrown if the class is abstract or an interface. *IllegalAccess* exceptions denote a problem with the design: The class likely needed to be abstract but wasn't declared so. Figure 11 shows the

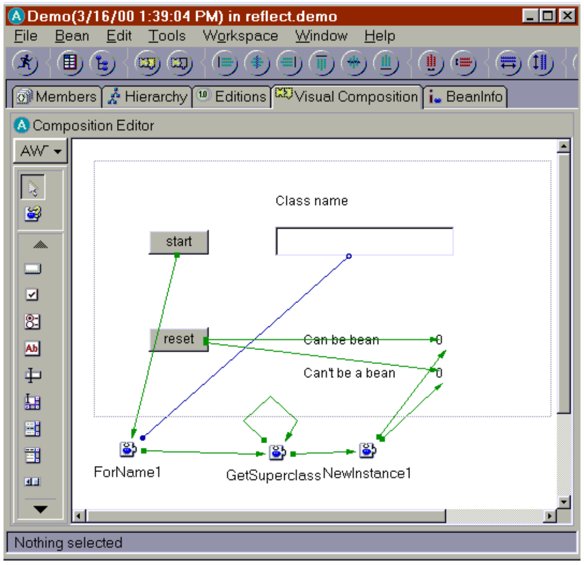

Figure 11: Reflection components in a visual builder

implementation of Figure 10 in a real visual builder (IBM VisualAge [15]) using the components produced by our generator.

ReflectionBeans offer a basis for evaluating the component generator and the translation process it embodies.

1. To demonstrate that the generator works with third party classes, we applied it first to the Java Core Reflection classes and later to the entire Java Development Kit (JDK). The results are presented in Section 5 and Tables 4 and 5. That exercise confirmed that the generator gets by exclusively on compiled code.

2. The ReflectionBeans, which were produced manually, serve as a quality benchmark, establishing whether the generator can produce code comparable to its handcrafted counterpart.

3. To generate equivalent ReflectionBeans, the generator must work across packages—a non-trivial but necessary ability if the generator is to be practical. Two packages are involved (`java.lang` and `java.lang.reflect`), thereby demonstrating that the input can be any set of classes, not necessarily classes in the same package.

4. As the development and implementation of the ReflectionBeans progressed, they became harder to change; inevitably some improvements were left out due to lack of time and pa-

[3] ForName, Figure 25 in Appendix A.
[4] GetSuperclass, Figure 28 in Appendix A.
[5] NewInstance, Figure 29 in Appendix A.

Northeastern University - College of Computer Science - Technical Report NU-CCS-00-01

tience. With automated generation, we can vary design choices more easily and frequently.

5. As an expedient, reflection methods that didn't seem worth implementing, such as `newInstance()`, were left out of ReflectionBeans. The generator, however, does a complete translation, treating all methods fully and equally. Attaining completeness forced us to tackle the hard issues in component translation.

4 The Transformation Process

There are five basic steps in the transformation:

1. Identifying the dimensions of the source architecture—that is, finding the dimensions of its method space.
2. Mapping the source space: classifying methods according to the dimensions identified.
3. Identifying the target space: finding the dimensions of components.
4. Mapping the target space: classifying components.
5. Populating the target space: deciding which method maps to which component.

"Identifying" means discovering the axes of the space using separation of concerns. By "mapping" we mean understanding which combinations of axes are useful—the meaning of an n-tuple point or region of the space. "Populating" refers to the transformation, establishing a function from the mapped source space (domain) to the mapped target space (range).

4.1 Dimensions of Methods

We focus on three dimensions for methods:

- *Method name*. The method's name can reveal intent. We examine each method for the prefixes *get*, *set*, and *is*.

  A classification that relies on naming conventions might be problematic in practice. Conventions may well vary across languages and organizations. Fortunately, naming conventions are institutionalized in Java: JavaBeans rely on naming conventions when introspecting an object. To circumvent naming conventions, mapping can also be guided by examining auxiliary descriptors obtained from an associated BeanInfo class, if it exists. But, there too, the association between a class x and its xBeanInfo class is based on naming conventions.

- *Number of arguments*. We distinguish between methods with no arguments, methods with a single argument, and methods with many arguments.

- *Return value*: We distinguish between methods that do not return a value and those that do.

4.2 Mapping the Method Space

The method space delineates how methods are classified—that is, whether a method is an inspector, mutator, procedure, function, etc. Table 1 shows the classification. '$-$' indicates that the method type (column label) has *none* of an attribute (row label), '$+$' indicates *some*, and 'ϕ' indicates *don't care*.

An *inspector* is a method named *get*Something or *is*Something that returns a value. A *mutator* is a method named *set*Something that returns nothing. If a mutator expects no arguments, then it is an *updator*. If it takes arguments, it is a *setter*. A *function* is a method that returns a value. A *procedure* is a method that does not return a value. If a procedure expects no arguments, it is an *action*; otherwise it is an *instruction.*

A method's position in this space can be pinpointed using reflection. If a method fits into more than one category, the most specific is chosen.

4.3 Dimensions of Components

Before deciding on a transformation (function) from methods to components, we must understand the source (domain) and the target (range). Sections 4.1 and 4.2 discussed mapping the source space; here we discuss the component space. The component we create depends on the target component framework. We consider only the JavaBeans component framework for simplicity, and only a subset of JavaBeans semantics at that.

We focus on five dimensions:

1. Layout—that is, the existence of input/output connectors.
2. Number of externally accessible properties.
3. Type relationship between connectors.
4. Message relationship between events.
5. Immutability of a message-object.

Northeastern University - College of Computer Science - Technical Report NU-CCS-00-01

	inspector		mutator		procedure		function
	getter	tester	updator	setter	action	instruction	
name	get*	is*	set*		ϕ		ϕ
arg	ϕ		$-$	$+$	$-$	$+$	ϕ
return	$+$		$-$		$-$		$+$

Table 1: The method space

4.3.1 Kinds of Layout

The layout of a JavaBeans component is fixed. Like conventional GUI components, only certain events can come in, and certain events go out. Hence a layout must be chosen *before* generating a component. We distinguish between four kinds of layouts: IN, OUT, IN-OUT, and INERT (see Figure 12).

1. An IN component listens to events but never fires them (i.e., it's an event sink).
2. An OUT component fires events but never listens to any (i.e., an event source).
3. An IN-OUT component receives and fires events.
4. A component that neither listens nor fires events is an INERT component. Later we demonstrate the usefulness of INERT components in modeling certain methods despite their sluggishness.

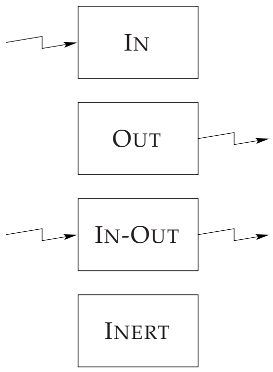

Figure 12: Component layout

4.3.2 Number of Properties

A bean may have properties. Properties are like the sockets of a hardware component (Figure 13). We distinguish two kinds of components.

- π_0: a component with no properties
- π_ϕ: a component with at least one property

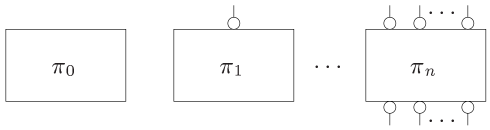

Figure 13: Component properties

4.3.3 Type Relationships Between In and Out Connectors

Events in Java are strongly typed; you can't connect an output of one component to the input of another unless the output event class is a subclass of the input event class.

We distinguish two kinds of IN-OUT components, as shown in Figure 14.

- $\tau_{\neq}$: The output event class is different from the input event class.
  - $\tau_{\in}$: the output event class is part of the system's events.
  - $\tau_{\notin}$: the output event class is external to the system.
- $\tau_{=}$: The output event class is the same as the input event class.

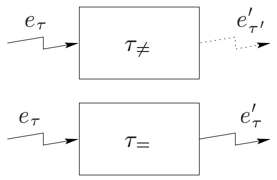

Figure 14: Event types

Northeastern University - College of Computer Science - Technical Report NU-CCS-00-01

4.3.4 Message Relationship Between In and Out Events

Among the components whose out-event type is the same as the in-event type, we further distinguish between those that replace the message and those that do not:

- $\mu_{\neq}$: The message-object shipped in the out-event is different from the message-object received in the in-event (although both are wrapped in an event object of the same event type.)
- $\mu_{=}$: Precisely the same message-object is sent out.

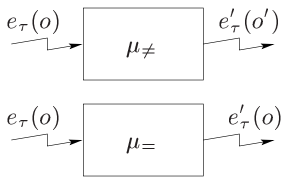

Figure 15: Message-object

4.3.5 State Relationship Between In and Out Message-Objects

We further distinguish between $\mu_{=}$-components that change the message-object's state (but preserves its ID) and those that treat the message-object as immutable.

4.4 Mapping the Component Space

Now that we have identified and chosen the dimensions of components, we can proceed to map the space. Table 2 shows the meaningful combinations and the names we have given them.

TRANSMITTER. A component that is a source of events.

RECEIVER. A sink of events.

TRANSCEIVER. Both a TRANSMITTER and RECEIVER.

TRANSPARENT. A TRANSCEIVER that passes the incoming event through.

TRANSFORMER. A TRANSCEIVER that fires the same type of event as it received but with different contents.

TRANSDUCER. A TRANSCEIVER that fires an event whose type is different than the type of the incoming event.

TRANSLATOR. A TRANSDUCER to an external type, used to connect to other systems.

ASSOCIATOR. Any of the above that also has at least one property.

4.5 Populating the Component Space

Now that the source and target spaces are mapped, we are left with establishing a transformation between the two. We proceed by populating the target space with concepts taken from the source space. Methods are mapped into components, and classes into events.

4.5.1 Transforming Methods into Components

There are four groups of methods in the source space:

1. *Inspector.* By definition, an inspector method has a return type, which is mapped to a TRANSCEIVER (see Procedure DOTRANSCEIVER in Figure 16).
   - If the return type is the same type as the declaring class, then the method maps to a TRANSFORMER. [6]
     - no arguments: a simple TRANSFORMER
     - any number of arguments: an ASSOCIATOR-TRANSFORMER
   - For other return types that correspond to one of the classes in the group, the method maps to a TRANSDUCER.
     - no arguments: a simple TRANSDUCER
     - any number of arguments: an ASSOCIATOR-TRANSDUCER
   - For any other return type, the method maps to a TRANSLATOR.
     - no arguments: a simple TRANSLATOR
     - any number of arguments: an ASSOCIATOR-TRANSLATOR

[6] Observe that for a method that returns `this` (i.e., self), "TRANSPARENT" is a better classification. TRANSPARENT is the one thing in the manual implementation of ReflectionBeans whose reproduction would have required a deeper analysis than we wished to undertake in the generator. Hence the generator misclassified such methods as TRANSFORMERs, but it also creates an abstract TRANSPARENT component class that users can extend to add concrete TRANSPARENT components manually.

Northeastern University - College of Computer Science - Technical Report NU-CCS-00-01

	INERT	IN	OUT	IN-OUT (TRANSCEIVER)			
				$\tau_=$		$\tau_{\neq}$	
				$\mu_=$	$\mu_{\neq}$	$\tau_{\in}$	$\tau_{\notin}$
π_0	INERT	RECEIVER	TRANSMITTER	TRANSPARENT	TRANSFORMER	TRANSDUCER	TRANSLATOR
π_ϕ	(ASSOCIATOR)						

Table 2: The component space

	INERT	IN	OUT	IN-OUT (TRANSCEIVER)			
				$\tau_=$		$\tau_{\neq}$	
				$\mu_=$	$\mu_{\neq}$	$\tau_{\in}$	$\tau_{\notin}$
π_0	**static**	**updator**	**constructor/**	**action**		**getter/**	
π_ϕ	**w/no ret**	**setter**	**static w/ret**	**instruction**		**function**	

Table 3: Populating the space

Procedure DOTRANSCEIVER ($\mathcal{C}$, m, r)

```
switch (r) /* r = returnType(m) */ {
   case r = void.class:
      return
         DOASSOCIATOR(m, class(m)-TRANSPARENT);
   case r = class(m):
      return
         DOASSOCIATOR(m, class(m)-TRANSFORMER);
   case r ∈ C: /* or componentType ∈ C */
      return DOASSOCIATOR(m, class(m)–to–r TRANSDUCER);
   case r ∉ C:
      return DOASSOCIATOR(m, class(m)–to–external TRANSLATOR);
}
```

Figure 16: Procedure DOTRANSCEIVER

We don't care if a type τ is returned, or an array of τ, or even an N-dimensional array. In every case, the components fire τ-events: a single event for a simple return value or multiple events for an array return. [7]

2. *Mutator* (see Procedure DORECEIVER in Figure 18).

   - updator (no arguments): a simple RECEIVER.

   - setter (one or more arguments): an ASSOCIATOR-RECEIVER.

[7] Otherwise, we would have needed to introduce many more new events: "array" event, "array of array" event, and so on.

Procedure DOASSOCIATOR (m, t)

```
switch (n ← #arg(m)) {
   case n = 0:
      return t;
   default:
      return argTypes(m)-ASSOCIATOR t;
}
```

Figure 17: Procedure DOASSOCIATOR

3. *Procedure.*

   - no return value and no arguments: a simple RECEIVER

   - no return value: an ASSOCIATOR-RECEIVER

4. All other functions: can map to any TRANSCEIVER.

Table 3 summarizes the classification. A updator method maps to a RECEIVER, a setter maps to an ASSOCIATOR-RECEIVER, a constructor or static method maps to a TRANSMITTER (see Procedure DOTRANSMITTER in Figure 19), and getters and functions map to either TRANSFORMERS or TRANSDUCERS depending on the type of return value. A static method with no return value maps to an INERT component (see Procedure DOSTATIC in Figure 20).

Algorithm METHOD-TO-COMPONENT, the method classification and component generation algorithm, is shown in Figure 21. The input is a set of classes, $\mathcal{C}$, along with the method m we want

Northeastern University - College of Computer Science - Technical Report NU-CCS-00-01

Package Name	Applet	Awt	Beans	Io	Lang	Math	Net	Rmi	Security	Sql	Text	Util	total	average
inspectors	21	1267	97	44	110	2	84	31	166	235	87	139	2283	190.25
updators	0	2	0	0	0	0	0	0	2	0	0	0	4	0.33
setters	1	379	28	3	19	0	28	1	13	54	61	53	640	53.33
mutators	1	381	28	3	19	0	28	1	15	54	61	53	644	53.67
actions	7	123	7	79	34	0	6	9	5	21	1	46	338	28.17
instructions	9	487	63	198	47	0	6	24	58	66	9	61	1028	85.67
procedures	16	610	70	277	81	0	12	33	63	87	10	107	1366	113.83
functions	0	543	33	207	194	66	25	56	131	151	114	363	1883	156.92
constructors	1	339	40	103	149	10	48	67	72	21	33	121	1004	83.67
static methods	1	114	21	5	202	3	22	49	38	17	40	104	616	51.33
Total number	40	3254	289	639	755	81	219	237	485	565	345	887	7796	649.67

Table 4: Distribution of methods in JDK 1.2

	Package Name	Applet	Awt	Beans	Io	Lang	Math	Net	Rmi	Security	Sql	Text	Util	total	avg
by # of I/O events	transmitters	2	445	54	108	322	13	61	107	106	32	73	180	1503	125.25
	receivers	17	991	98	280	100	0	40	34	78	141	71	160	2010	167.50
	transceivers	21	1810	130	251	304	68	109	87	297	386	201	502	4166	347.17
	inert	0	8	7	0	29	0	9	9	4	6	0	45	117	9.75
by type of I/O	transformers	0	58	1	6	44	38	0	0	0	2	0	7	156	13.00
	transducers	21	1752	129	245	260	30	109	87	297	384	201	495	4010	334.17
by # of properties	associators	21	1810	149	298	427	45	83	114	217	292	173	482	4111	342.58
	non-associators	19	1444	140	341	328	36	136	123	268	273	172	405	3685	307.08
	Total number	40	3254	289	639	755	81	219	237	485	565	345	887	7796	649.7

Table 5: Distribution of components generated from JDK 1.2 packages

Procedure DoReceiver (m)

return DoAssociator(m,$class(m)$-RECEIVER);

Figure 18: Procedure DoReceiver

Procedure DoTransmitter (m)

return
  DoAssociator(m, $class(m)$-TRANSMITTER);

Figure 19: Procedure DoTransmitter

to classify, where $class(m) \in \mathcal{C}$. The output is the component kind generated for that method.

4.5.2 Transforming classes into events

For every class x in $\mathcal{C}$, we generate a dedicated $x$event, and for all other types we use a special ExternalEvent. For example, an *associator* may fire either an ExternalEvent or xEvent where $x \in \mathcal{C}$, depending on its return type. More specifically, for the set of classes given as input, the generator produces, for every class x,

- an event class,

Procedure DoStatic (m,r)

if [r = **`void.class`**] /* $r = returnType(m)$ */
  return DoAssociator(m,INERT);
else
  return DoTransmitter(m);

Figure 20: Procedure DoStatic

- an event-listener interface,
- an event-registration support class, and
- an event-to-property adapter.

5 A Case Study: JDK-Beans

Once the generator was built, we tested it by generating systems of components from arbitrary sets of classes. Tables 4 and 5 summarize the results of applying the generator to the Java JDK 1.2 packages.

These statistics offer some insight into the quality of the transformation. First off, completeness: the classification covers all cases found in the JDK. All methods in all the packages were successfully classified to one of the categories and then transformed

Northeastern University - College of Computer Science - Technical Report NU-CCS-00-01

METHOD-TO-COMPONENT $(\mathcal{C}, m)$
if $[\neg isPublic(m)]$ return;
if $[isConstructor(m)]$ return DOTRANSMITTER(m);
$r \leftarrow returnType(m)$;
while $[isArray(r)]$ $r \leftarrow componentType(r)$;
if $[isStatic(m)]$ return DOSTATIC(m,r);
/* otherwise method is not `static` */
$s \leftarrow name(m)$;
if $[r =$ `void.class`$]$ /* inspector or function */
  return DOTRANSCEIVER($\mathcal{C}$,m);
if $[prefix\ (s) =$ "`set`"$]$ /* mutator */
  return DORECEIVER($\mathcal{C}$,m);
/* else, action or instruction */
return DOTRANSCEIVER($\mathcal{C}$,m);

Figure 21: Algorithm METHOD-TO-COMPONENT

into one component or another. Second, the distribution of the methods and components in each category indicates whether that category is a useful concept. For example, the paucity of *updators* suggests that the distinction between *updator* and *setter* flavors of *mutators* (see Table 1) is arguably unjustified. On the other hand, the distinction between *procedures* that are *actions* and those that are *instructions* seems to be a more useful one.

The distribution of method types in a package also reflects the package's purpose. The distribution of components generated from a package lends further insight. In `java.math`, for example, most of the methods are functions, and none of the generated components are INERT (i.e., there is no notion of environment settings). In `java.awt`, the majority of the methods are inspectors. In `java.rmi` all TRANSCEIVERs are TRANSFORMERS, reflecting the package's utilitarian purposes. The classes (actually they're all interfaces) are physically in the same package but logically need not be (e.g., package visibility is unlikely to be much of an issue.)

5.1 Taxonomy of Components

The process we have described—identifying the dimensions of components, mapping the component space, and populating the space—produced as a byproduct the terminology for a component taxonomy depicted in Figure 23 (contrasted with the object-oriented protocol taxonomy of Figure 22).

5.2 Hierarchy vs. Name Space Pollution

The mapping of methods to components elevates methods to the level of classes. Careful structuring of the generated components is required to resolve name conflicts. For example, two methods with the same name from different classes must map into unique component names.

To effect this without polluting the namespace, the generator creates a package for each category and a sub-package for each sub-category in the taxonomy. Each component is then generated inside the package that reflects the component's classification. This structuring not only resolves name conflicts; it also helps a user navigate and find the right component when building an application.

5.3 Abstract Components vs. Code Duplication

The experience with the JDK packages shows that each component category typically clusters several mapped methods. The taxonomy emphasizes the fact that all the components in a certain category share their layout, certain behavior, etc. To avoid repeating interfaces and code among related components, the generator generates an abstract class or interface for each category, which the rest of the components in that category implement. Hence the taxonomy is used within the generation process for abstraction and reuse. These shared classes will be useful to users who augment the generated components with handcrafted ones, which can reuse the shared functionality.

5.4 The Power of Taxonomy

Finding a good taxonomy is hard, but once found it is very helpful. The traditional taxonomy of methods (operations) in an object-oriented interface (see Figure 22) is helpful in understanding and designing classes. (The taxonomy in Figure 22 corresponds to the method space we have mapped in Table 1.) For example, it is considered good practice [27] to keep inspectors and mutators separate (say, methods *top* and *pop* in a *stack* class), avoiding half-breed inspector-mutators that don't change the object but reveal a reference to the state of the object. A good taxonomy and terminology for components is likely to bring similar benefits to component design.

6 Related Work

The architecture [29] is a vital and integral part of a software system. It governs the kind of software units used, the way they are put together, and their interrelationships. The architecture is a thread weaving the units together. Still, it's occasionally nec-

Northeastern University - College of Computer Science - Technical Report NU-CCS-00-01

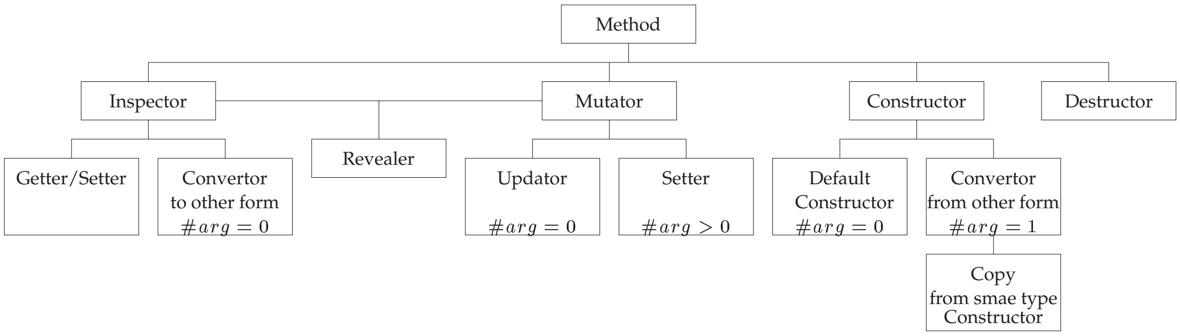

Figure 22: Existing taxonomy of explicit protocol invocation

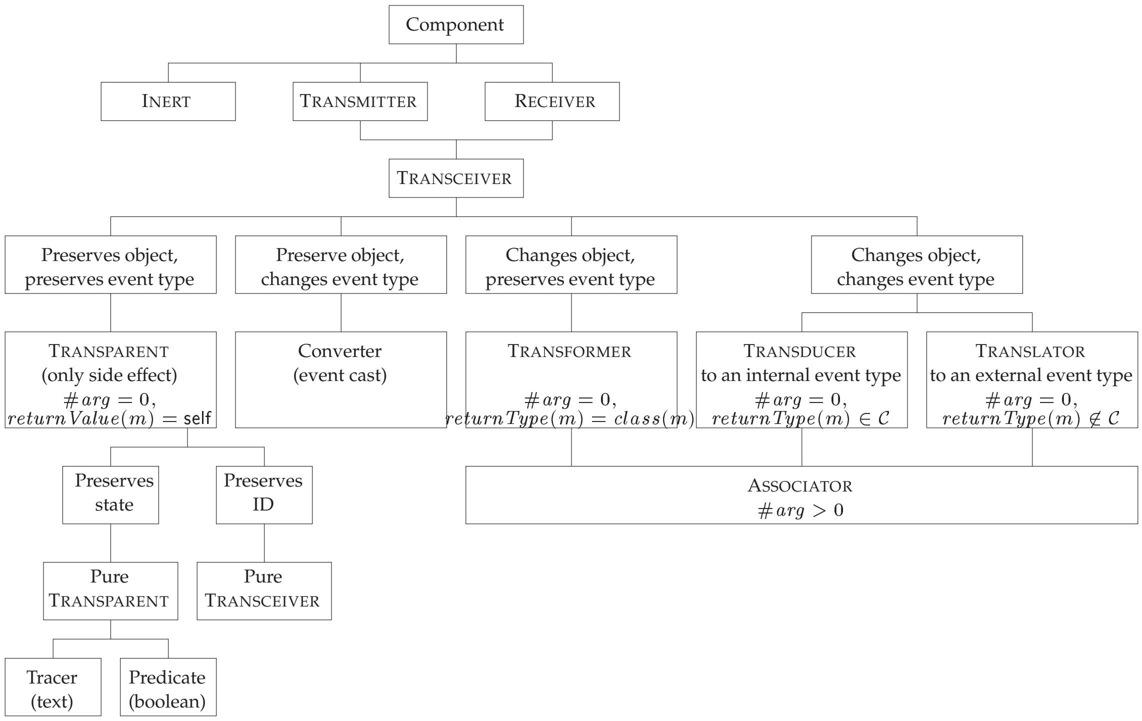

Figure 23: Suggested taxonomy of implicit protocol invocation

Northeastern University - College of Computer Science - Technical Report NU-CCS-00-01

essary to switch architectures despite the repercussions. Unweaving the inseparable is the bread and butter of Aspect-Oriented Programming (AOP) [20].

From an AOP perspective, this work could be considered treating the architecture itself as an aspect. That would permit programming the contents first and settling on the architecture later. AOP per se, however, does not solve the problem this work set out to solve, because AOP is more a conceptual framework than a development tool. Applying it requires rolling your own aspect language and weaver compiler. The contribution of this work is precisely that. The transformation algorithm can be viewed as a domain-specific language—even an architecture description language (ADL)—for describing the target architecture. The generator can be considered a weaver. Early implementations of AOP language tools (e.g., AspectJ [22]) deal only with simple tangling of well-defined and orthogonal aspects, such as the domain model versus its implementation, debugging, and performance monitoring. In contrast, the architecture is so integral to a system's expression in code that it is probably beyond the scope of AOP to call the architecture an aspect.

Subject-Oriented Design (SOD) [5] lets multiple design views of a system exist simultaneously and yet maintainably. Like AOP, SOD is concerned with separation of concerns. But unlike AOP, SOD can have multiple interchangeable views (subjects) of the same underlying software. From a SOD perspective, this work elevates architecture to the subject level, treating explicit and implicit invocation interchangeably.

The transformation process can be interpreted as a Multi-Dimensional Separation of Concerns [31], SOD's successor, where the dimensions are the dimensions of the method and component spaces.

The novelty of this work with respect to generating components [28] lies in its comprehensive approach:

1. We generate a complete system of components rather than an isolated component—unlike integrated development environments (IDE) that offer limited support for JavaBeans or EJB. In these environments, the user declares properties and events through wizards; the system generates the corresponding methods and fields in the component class. In contrast, our approach generates a complete component system, generating a hierarchy of component classes and a hierarchy of event classes, both working in concert.
2. The generator supplies everything you need in a ready-to-use jar of JavaBeans. Documentation, BeanInfo classes, images for icons, a manifest file, a jar file, and a Makefile are generated automatically as well.

Catalysis [9] has introduced new notations for modeling component-based systems. A potential application of our work is to add a front end for processing UML and Catalysis diagrams. The notion of *styles* in Catalysis seems particularly relevant. For example, a diagram using the JavaBeans style can be more easily translated to an actual system [8]. Nothing on this has been published, as far as we know.

7 Conclusion

Implicit invocation is gaining ground as an implementation technique. We have described a code generator that can transform the compiled form of an object-oriented Java program into a "componentized" JavaBeans system. These techniques can aid in transforming existing systems to implicit invocation systems.

7.1 Component-Based Systems

There is rising interest in architectural design and construction of component-based systems beyond ad-hoc reuse of unrelated components [3]. In the current absence of lingual support for component-based programming, and in the absence of good component system-design tools, component-generators can play a leading role. In this respect, the role of component-generators can be compared to the role of compiler-compilers (e.g., YACC [2]) in language design. Component generators can be of help in:

- migrate existing code into components,
- experimenting with large systems of components,
- creating domain-specific as well as general-use components, and
- characterizing generic components.

7.2 Generative Components

This works falls squarely in the area of generative and component-based software engineering [6]. Generating components and systems from smaller

Northeastern University - College of Computer Science - Technical Report NU-CCS-00-01

components or from more abstract representations can help coping with problem such as complexity explosion.

7.3 Specific vs. Generic Components

An obvious limitation of the method-to-component mapping is the proliferation of generated components. Traditionally, components are coarse-grained chunks of software [29, 10] that can be developed with whatever language or infrastructure you like. The transformation and generator does not reflect this view: the object-oriented specification is broken down into a large number of simple components. Whereas the coarse granuality of traditional components admits considerable overhead per component, that same overhead becomes prohibitive at finer granularities.

So while the generator might be useful for toy examples like the Boolean gates or perhaps even large components with small interfaces, it is clearly not practical for most real-world class libraries. There are simply too many methods, giving rise to too many components. The generator can create them easily enough, but managing and maintaining them will be tedious.

Partitioning components into packages mitigates the problem somewhat, as do abstract components. But the real solution will come from using generic components rather than generated ones. Here too, the component taxonomy can come to the rescue—*if* we use it to produce only a few generic components (that is, a highly parameterized component for each point in the space).

One might imagine a totally generic single MethodInvocation component that could be customized to execute any method. This component would have a property of type `java.lang.Method`, and there would also be a method property customizer through which a user customizes the component. This would alleviate the problem of too many components, at least in theory; only one polymorphic component type could cover all cases. It doesn't work so easily in practice, however, because the component layout is inflexible. A generic MethodInvocation would require both input and output events just in case it needs to model a TRANSCEIVER. Then one might mistakenly connect to an inactive input of a TRANSMITTER or the inactive output of a RECEIVER.

Alternatively, one could posit a generic component for *each category* in the taxonomy. A set of generic components (a generic TRANSMITTER, a generic RECEIVER, a genetic TRANSCEIVER, etc) could cover the space. This is indeed possible and has clear advantages: there would be a bigger but still bounded system of components, and there would be just one set of generic MethodInvocation beans for the entire JDK. Of course, in a different component framework, with different constraints, a different set of generic components would be generated.

7.4 Future Work

Breaking down a system in components requires many design decisions. Much has been written about object-oriented design and development, with few practical advances in automating the process. In component-based programming the opposite seems true: A high level of automation exists, but little attention is paid to component-based design. Most JavaBeans books, for example, focus more on what beans are than on how to build software that adheres to the JavaBeans architecture.

When transforming an object-oriented specification to a component-based system, it's not enough to *make the system work*; the design also needs to be adapted to the new architecture. The proposed framework leaves room for such decisions. Once a transformation style has been set, a novice can apply it over and over again. One could create a collection of code-generation styles from which a user can choose a transformation from a component diagram in UML to a system of JavaBeans. Catalysis seems to be going in this direction [8].

The generator can also serve as tool for experts to experiment with alternative component style and designs. Currently, however, the mapping is hardcoded; external configuration of the generator for other mappings is work for the future. Making the generator rule-based would allow one to experiment with different transformations by changing the transformation rules and reapplying them to the system until a satisfactory mapping is found. One can start with generating simple components and migrating gradually to more generic components, thus supporting an evolutionary component development.

The transformation preserves the semantics of each method, but introduces subtle interactions. Ongoing work will address the semantics of the source classes and extending the transformation to UML and other specifications styles.

Northeastern University - College of Computer Science - Technical Report NU-CCS-00-01

8 Acknowledgment

We thank Predrag Petkovic for his helpful comments and suggestions, and for implementing the generator described in this report.

References

[1] G. M. Barnes and B. R. Swim. Inheriting software metrics. *Journal of Object-Oriented Programming*, pages 27–34, November-December 1993.

[2] J. P. Bennett. *Introduction to Compiling Techniques - A First Course Using Ansi C , Lex and Yacc*. McGraw-Hill, 1990.

[3] J. Bosch, C. Szyperski, and W. Weck. Workshop on component oriented programming, ECOOP, 2000. Upcoming event.

[4] S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object-oriented design. In *Proceedings of the 6^{th} Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 197–211, Phoenix, Arizona, USA, Oct.6-11 1991. OOPSLA'91, Acm SIGPLAN Notices 26(11) Nov. 1991.

[5] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Subject-oriented design: Towards improved aligment of requirements, design and code. ACM SIGPLAN Notices 34(10):325-339, Oct 1999.

[6] K. Czarnecki and U. W. Eisenecker. Components and generative programming (invited paper). In *Proceedings of the 7th European Engineering Conference held jointly with the 7th ACM SIGSOFT symposuim on Foundations of software engineering*, pages 2–19, Toulouse France, Sept. 6-10 1999.

[7] S. Demeyer and J. Bosch, editors. *Object-Oriented Technology. ECOOP'98 Workshop Reader*, number 1543 in Lecture Notes in Computer Science. Workshop Proceedings, Brussels, Belguim, Springer Verlag, July 20-24 1998.

[8] D. D'Souza. UML panel in OOPSLA'99.

[9] D. D'Souza and A. Wills. Objects, components, and frameworks with UML: The catalysis approach. Addison Wesley, 1998.

[10] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. IEEE Software, Nov. 1995, pp. 17-26.

[11] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In LNCS 551, 31-44, Springer-Verlag, 1991.

[12] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[13] B. Henderson-Sellers. *Object Oriented Metrics*. Object-Oriented Series. Prentice-Hall, 1996.

[14] W. L. Hürsch. Should superclasses be abstract? In M. Tokoro and R. Pareschi, editors, *Proceedings of the 8^{th} European Conference on Object-Oriented Programming*, number 821 in Lecture Notes in Computer Science, pages 12–31, Bologna, Italy, July 4-8 1994. ECOOP'94, Springer Verlag.

[15] IBM. VisualAge for Java. `http://-www.ibm.com/software/ad/vajava/`.

[16] JavaSoft. BeanBox. `http://-www.javasoft.com/beans/software/-beanbox.html`.

[17] JavaSoft. BDK 1.1, November 1997. `http://-java.sun.com/beans/software/-bdk_download.html`.

[18] M. Johnson. BeanLint: A JavaBeans troubleshooting tool, part 1. *JavaWorld*, 3(12), Dec. 1998. http://www.javaworld.com/-javaworld/jw-12-1998/jw-12-beans.html.

[19] M. Johnson. BeanLint: A JavaBeans troubleshooting tool, part 2. *JavaWorld*, 4(1), Jan. 1999. http://www.javaworld.com/-javaworld/jw-01-1999/jw-01-beans.html.

[20] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of the 11^{th} European Conference on Object-Oriented Programming*, number 1241 in Lecture Notes in Computer Science, pages 220–242, Jyväskylä, Finland, June 9-13 1997. ECOOP'97, Springer Verlag.

[21] H. F. Li and W. K. Cheung. An empirical study of software metrics. *IEEE Transactions on Software Engineering*, 13(6):697–708, June 1987.

Northeastern University - College of Computer Science - Technical Report NU-CCS-00-01

[22] C. V. Lopes and G. Kiczales. Recent developments in AspectJ. In Demeyer and Bosch [7], pages 398–401.

[23] D. Lorenz and J. Vlissides. Unifying the reflection-repository duality. Manuscript in progress.

[24] D. H. Lorenz. Visitor Beans: An aspect-oriented pattern (abstract). In Demeyer and Bosch [7], pages 431–432.

[25] D. H. Lorenz and P. Petkovic. ContextBox: A BeanBox environment for design-time assembly of BeanContext components. Technical Report NU-CCS-99-04, College of Computer Science, Northeastern University, Boston, MA 02115, Nov. 1999.

[26] D. H. Lorenz and P. Petkovic. Design-time assembly of runtime containment components. In *Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems*, Santa Barbara, CA, July 30-Aug. 3 2000. TOOLS 34 USA Conference, IEEE Computer Society. To appear.

[27] B. Meyer. *Object-Oriented Software Construction*. International Series in Computer Science. Prentice-Hall, 1988.

[28] H. Partsch and R. Steinbrueggen. Program transformation systems. ACM Computing Surveys, 15(3), Sep 1983.

[29] M. Shaw and D. Garlan. *Software Architecture, Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[30] C. Szyperski. *Component-Oriented Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1997.

[31] P. Tar, H. Ossher, W. Harrison, and S. Sutton. N degrees of separation: Multidimensional separation of concerns. In ICSE'99.

Northeastern University - College of Computer Science - Technical Report NU-CCS-00-01

A The Code Generated from `java.lang.Class` for the Components Used in Figure 11

Class ClassEvent ()

```
package edu.neu.ccs.beans.reflect.event;

import edu.neu.ccs.beans.external.event.ExternalEvent;

public class ClassEvent extends ExternalEvent {
   public ClassEvent(Object source, int type, java.lang.Class obj, Object data) {
      super(source, type, obj, data);
   }
   public java.lang.Class getEventClass() {
      java.lang.Class EventClass = null;
      try {
         EventClass = (java.lang.Class) getExternal();
      }
      catch (ClassCastException exception) {
         System.err.println(”Exception casting external field to java.lang.Class”);
      }
      return EventClass;
   }
}
```

Figure 24: Class ClassEvent

Northeastern University - College of Computer Science - Technical Report NU-CCS-00-01

Class ForName ()

```
package edu.neu.ccs.beans.reflect.clazz.transmitter.associator;

import edu.neu.ccs.beans.reflect.event.ClassEvent;
import edu.neu.ccs.beans.reflect.event.ClassListener;
import edu.neu.ccs.beans.reflect.event.ClassSupport;
import edu.neu.ccs.beans.util.ExceptionEvent;
import edu.neu.ccs.beans.util.ExceptionListener;
import edu.neu.ccs.beans.util.ExceptionSupport;

public class ForName {
   private transient java.lang.String arg0;
   private ClassSupport ClassSupport = new ClassSupport();
   private ExceptionSupport ExceptionSupport = new ExceptionSupport();
   public void addClassListener(ClassListener newListener) {
      ClassSupport.addEventListener(newListener);
   }
   public void addExceptionListener(ExceptionListener newListener) {
      ExceptionSupport.addEventListener(newListener);
   }
   public void fire() {
      try {
         java.lang.Class returnValue = java.lang.Class.forName(arg0);
         ClassEvent returnEvent = new ClassEvent(this, ClassEvent.NORMAL, returnValue, null);
         fireClassEvent(returnEvent);
      }
      catch (Exception e) {
         fireExceptionEvent( new ExceptionEvent(this, e));
      }
   }
   protected void fireClassEvent(ClassEvent event) {
      ClassSupport.fireClassEvent(event);
   }
   protected void fireExceptionEvent(ExceptionEvent event) {
      ExceptionSupport.fireExceptionEvent(event);
   }
   public void removeClassListener(ClassListener oldListener) {
      ClassSupport.removeEventListener(oldListener);
   }
   public void removeExceptionListener(ExceptionListener oldListener) {
      ExceptionSupport.removeEventListener(oldListener);
   }
   public void setArg0(java.lang.String value) {
      java.lang.String oldValue = arg0;
      arg0 = value;
   }
}
```

Figure 25: Class ForName

Class ClassListener ()

```
package edu.neu.ccs.beans.reflect.event;

public interface ClassListener {
   public void receiveClassEvent(ClassEvent event);
}
```

Figure 26: Class ClassListener

Northeastern University - College of Computer Science - Technical Report NU-CCS-00-01

Class ClassSupport ()

```
package edu.neu.ccs.beans.reflect.event;

import edu.neu.ccs.beans.external.event.ExternalSupport;

public class ClassSupport extends ExternalSupport {
  /**
  * Notify ClassEvent listeners
  */
  public void fireClassEvent(ClassEvent evt) {
    java.util.Vector targets = getListeners();
    for (int i = 0; i < targets.size(); i++) {
      ClassListener target = (ClassListener )targets.elementAt(i);
      if (target != null)
        target.receiveClassEvent(evt);
    }
  }
}
```

Figure 27: Class ClassSupport

Class GetSuperclass ()

```
package edu.neu.ccs.beans.reflect.clazz.transceiver;

import edu.neu.ccs.beans.reflect.event.ClassEvent;

public class GetSuperclass extends ClassTransceiver {
  protected void propagateClassEvent(ClassEvent evt) {
    java.lang.Class returnValue = getEventClass().getSuperclass();
    ClassEvent returnEvent = new ClassEvent(this, ClassEvent.NORMAL, returnValue, null);
    fireClassEvent(returnEvent);
  }
}
```

Figure 28: Class GetSuperclass

Northeastern University - College of Computer Science - Technical Report NU-CCS-00-01

Class NewInstance ()

```
package edu.neu.ccs.beans.reflect.clazz.transformer;

import edu.neu.ccs.beans.reflect.clazz.ClassReceiver;
import edu.neu.ccs.beans.reflect.event.ClassEvent;
import edu.neu.ccs.beans.reflect.event.ClassListener;
import edu.neu.ccs.beans.reflect.event.Classsupport;
import edu.neu.ccs.beans.external.event.ExternalEvent;
import edu.neu.ccs.beans.external.event.ExternalListener;
import edu.neu.ccs.beans.external.event.ExternalSupport;
import edu.neu.ccs.beans.util.ExceptionEvent;
import edu.neu.ccs.beans.util.ExceptionListener;
import edu.neu.ccs.beans.util.ExceptionSupport;

public class NewInstance extends ClassReceiver {
  private ExternalSupport ExternalSupport = new ExternalSupport();

  private ExceptionSupport ExceptionSupport = new ExceptionSupport();

  public void addExceptionListener(ExceptionListener newListener) {
    ExceptionSupport.addEventListener(newListener);
  }
  public void addExternalListener(ExternalListener newListener) {
    ExternalSupport.addEventListener(newListener);
  }
  protected void fireExceptionEvent(ExceptionEvent event) {
    ExceptionSupport.fireExceptionEvent(event);
  }
  protected void fireExternalEvent(ExternalEvent event) {
    ExternalSupport.fireExternalEvent(event);
  }
  protected void propagateClassEvent(ClassEvent evt) {
    try {
      java.lang.Object returnValue = getEventClass().newInstance();
      ExternalEvent returnEvent = new ExternalEvent(this, ExternalEvent.NORMAL, returnValue, null);
      fireExternalEvent(returnEvent);
    }
    catch (Exception e) {
      fireExceptionEvent( new ExceptionEvent(this, e));
    }
  }
  public void removeExceptionListener(ExceptionListener oldListener) {
    ExceptionSupport.removeEventListener(oldListener);
  }
  public void removeExternalListener(ExternalListener oldListener) {
    ExternalSupport.removeEventListener(oldListener);
  }
}
```

Figure 29: Class NewInstance