Designing Components versus Objects: A Transformational Approach

David H. Lorenz
Northeastern University
College of Computer Science
Boston, MA 02115 USA
+1 617 373 2076
lorenz@ccs.neu.edu

John Vlissides
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598 USA
+1 914 784 7918
vlis@us.ibm.com

ABSTRACT
A good object-oriented design does not necessarily make a good component-based design, and vice versa. What design principles do components introduce? This paper examines component-based programming and how it expands the design space in the context of an event-based component architecture. We present a conceptual model for addressing new design issues these components afford, and we identify fundamental design decisions in this model that are not a concern in conventional object-oriented design. We use JavaBeans-based examples to illustrate concretely how expertise in component-based design, as embodied in a component taxonomy and implementation space, impacts both design and the process of design. The results are not exclusive to JavaBeans—they can apply to any comparable component architecture.

Keywords
Component-based software engineering, component-based design, classification, taxonomy, JavaBeans.

1 INTRODUCTION

Components are "units of independent production, acquisition, and deployment that interact to form a functioning system" [16]. The decoupling of software production and deployment (as exemplified by markets for third-party components) and the centrality of large-scale composition are largely why component-based programming (CBP) lies "beyond object-oriented programming (OOP)." These attributes bring with them many new and largely unexplored issues in software design.

In this paper we characterize the fundamental design decisions in CBP and the design space of component-based design (CBD) beyond object-oriented design (OOD). We illustrate how a good object-oriented (OO) design is not necessarily a good component-based (CB) design, and we identify component design decisions that are not a concern in OOD. We don't try to list all component design decisions and their solutions. Rather, this work underscores the importance of design, particularly in CBP, through a conceptual model for addressing new design issues that CBP affords.

For concreteness, we do this in the context of a specific component architecture, JavaBeans (JB). By grounding our discussion in a working component system—implementation warts and all—we are forced to confront real issues in CBD, lending credibility to the results. While these results are not exclusive to JB, their application to substantially different component models is left for future work.

There are many design decisions to consider as you decompose a system into components. We illustrate the unique characteristics of CBD through three key decisions:

1. *Identifying the components and connectors.* What do components model in the problem domain, and what does their communication model?
2. *Designing the components and connectors.* What are the dimensions of the components (i.e., fan-in, fan-out, event types, etc.)?
3. *Implementing the components and connectors.* How do you trade off space, time, and safety to optimize performance, reusability, maintainability, and cost?

We characterize component design by showing how each of these decisions requires expertise in CBD as embodied in a component taxonomy and implementation space. A series of related JB examples illustrates how this expertise impacts both design and the process of design. The next three sections consider these decisions in the context of implementing implicit method invocation [4] within increasingly complex examples. Section 2 describes component-event parti-

0-7695-1050-7/01 $10.00 © 2001 IEEE

tioning through a single class example, Section 3 describes component-event design through an example involving multiple classes from the same package, and Section 4 describes component-event implementation through a multiple-package, multiple-class example.

The examples address issues in component partitioning, design, and implementation in light of analogous issues in a corresponding object-oriented design. Indeed, we discuss CBD almost exclusively through direct comparisons to OOD, going so far as to consider the canonical problem in CBD to be how to map object-oriented designs to component-based designs.

2 COMPONENT-EVENT PARTITIONING

Consider the task of simulating Boolean logic using JB, which has an event-based, implicit invocation architecture. What logical concept(s) should components and events model? Recall that $nand$ comprises a complete set—that is, $not(x) = nand(x, x)$—and by applying DeMorgan's laws, any boolean function can be formulated solely in terms of $nand$ operations. For example, $or(x, y) = nand(nand(x, x), nand(y, y))$. Even so, implementing not as a separate gate can cut down on wiring: nor, for example, simplifies to $not(nand(not(x), not(y)))$, eliminating the need to wire together $nand$ inputs.

In a conventional OO design, one is likely to define an interface such as *Bool* (Figure 1) for objects of Boolean type. *Bool* contains only two methods, `not` and `nand`. Classes True and False would implement these operations differently.

What should a CB version of this design look like?

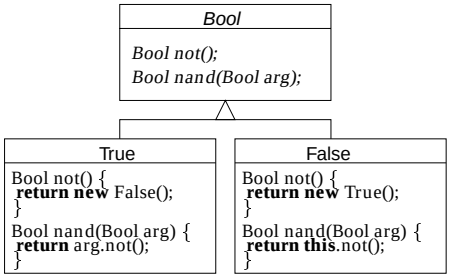

Figure 1: The Boolean class hierarchy

A Naive Component-Based Design

Basic OO principles do not give us much insight into a good CB design, at least not for the *Bool* example. A naive CB design mimics the OO design by mapping methods to events, and objects to components. The resulting component hierarchy, shown in Figure 2, leaves much to be desired.

The components TrueBean and FalseBean in Figure 2 (corresponding to the classes True and False in Figure 1) carry fixed behaviors akin to the objects *true* and *false*, respectively, in Smalltalk [5]. Events NotEvent and NandEvent, which implement the *InvokeEvent* interface shown in Figure 3, correspond to the methods `not` and `nand` in the OO design. Two other events are NewEvent and ReturnEvent for constructing and returning a value, respectively. The incoming arrows NewEvent, NotEvent, and NandEvent in Figure 2 indicate that BoolBean subclasses implement the *NewEventListener*, *NotEventListener*, and *NandEventListener* interfaces. The outgoing ReturnEvent arrow means that instances can broadcast the result to potential *ReturnEventListener*s.

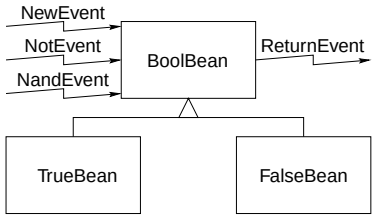

Figure 2: Class-to-component mapping

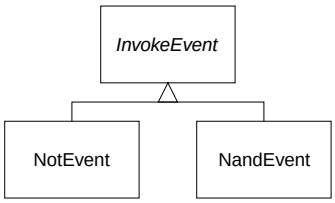

Figure 3: Method-to-event mapping

This decomposition is effective in an OO design, wherein objects can send messages and serve as return values, but it makes a poor CB design, rendering simple things like constructing a function with arguments (e.g., nor) 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 events, such as NotReturnEvent and NandReturnEvent. But that has the detrimental side-effect of creating a parallel set of components solely for return values, which complicates maintenance. The decomposition also complicates parameter passing, demanding enough settable properties to accommodate the method with the maximum number of arguments in the OO design. Adding an operation (say, `nor`) requires changing BoolBean, further complicating maintenance. 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.

An Improved Component-Based Design

Figure 4 illustrates a different CB design, wherein each

method in the OO design becomes a component and each *class* becomes an event. Compared to the earlier design, the resulting components Nand and Not in Figure 4 reflect a dataflow factoring rather than the original's data-oriented factoring. The data (that is, the receiver o and the result of the operation o') travels in the input and output events rather than lying in fixed structures. The components thus become far more convenient building blocks for a JB-based Boolean toolkit.

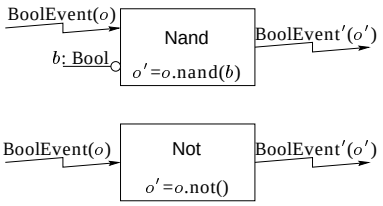

Figure 4: Method-to-component mapping

In this method-to-component design, returning a value simply corresponds to firing an event. Adding a new `nor` operation is also simple: just add another kind of component. For example, Figure 5 shows how a Nor component can be constructed. The resemblance to hardware components is intentional.

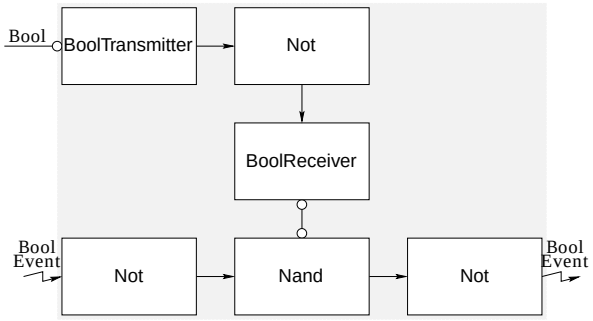

Figure 5: $nor(x, y) = not(nand(not(x), not(y)))$

Note that the receiver of `nand` is supplied within BoolEvent, while the argument of `nand` is supplied to the component Nand by binding its property (denoted by small circles). This is in keeping with CBD because it avoids races and generally simplifies synchronization.

Also note that the components in Figure 4 are not always convenient enough by themselves. Additional components are desirable that have no direct analog in a pure OO design, such as the adapters shown in Figure 6 (and used in Figure 5) for converting a property to an event and vice versa. These adapters make the event-based architecture more flexible. Moreover, they let the *nor* interface mimic *nand*'s—that is, one input is bound as a property and the second is received as an event. The CB design should take such adapters into account as well as the trade-off between using properties versus events as inputs.

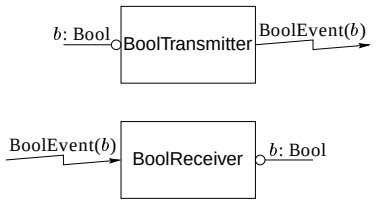

Figure 6: Event-Property adapters

3 COMPONENT DESIGN

Now consider the more complex task of designing a toolkit of REFLECTIONBEANS with which to construct tools that compute simple OO metrics or that find dependencies among classes [11]. Such a toolkit can be helpful in many ways:

- Obtaining program information through reflection instead of parsing source code affords the benefit of working with compiled `.class` files in JAVA. Thus the metrics can be applied to third-party components whose source code is unavailable.
- Reflection allows simple dynamic checks in lieu of complicated static analysis, as we demonstrate later.
- Assembling components in a builder is usually easier than writing reflection code. REFLECTIONBEANS allow quick, graphical construction of a variety of analyses, making it easy to experiment with different metrics.

How would one design such a REFLECTIONBEANS toolkit? We could apply the method-to-component approach of Figure 4 to the JAVA Core Reflection design (rather than just the *Bool* interface). That is, we can produce a system of components for expressing reflective computations just as we produced the Nand and Not components for Boolean expressions. Consider the metaclass `java.lang.Class` in JAVA. Instead of a `getSuperclass` method, we would define a component that receives a ClassEvent (carrying a class object) and then fires a ClassEvent (carrying the corresponding superclass object).

Whereas the *Bool* design involved just one interface, the REFLECTIONBEANS design is the CB equivalent of a package of classes in the OO design. That means a component class exists for every method of every class in `java.lang.reflect`—which raises new issues. Choosing the method-to-component mapping, (i.e., identifying the components and events) is merely the first of many decisions in any nontrivial CB design.

When every method gives rise to a component, the resulting number of component types can become un-

wieldy. As an alternative, a single, parameterized component may serve similar methods (e.g., all inspectors or all mutators), thus reducing the number of different components but also potentially incurring drawbacks, such as reducing reusability (the component becomes monolithic), type safety, or efficiency. Then again, abstract components and subcomponents might also increase efficiency and reusability if, for example, the CB design suffers code duplication across components.

How can one make informed decisions in the design of a component system? Should a single component carry out several duties? Should components share an implementation by sharing subcomponents or by inheriting from more abstract components? How do you structure a potentially large number of components in a way that's easy to navigate and maintain?

Dimensions of Components

The components we create depend on the target component architecture. Despite the preponderance of OO languages, they all adhere closely enough to a common object model that a designer can reason about an OO design largely independent of implementation language. In contrast, the commonalities among component architectures are less apparent and agreed upon, and so the underlying technology must be taken into greater consideration than in OOD.

To reach a more systematic approach to component design in general, we examine JB design in particular, especially the degrees of freedom JB offers. By limiting ourselves to the JB component framework for simplicity—and only a subset of JB static semantics—we can focus on five dimensions:

1. **Bias**—that is, the ability to accept / produce events. The bias of a JB component is fixed. Like conventional GUI components, certain events can come in, and certain events go out. Hence bias must be chosen *statically*.

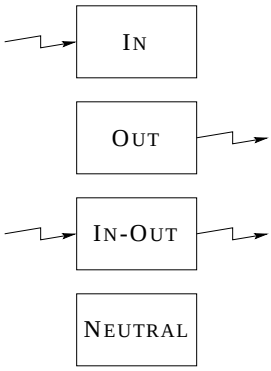

Figure 7: Component bias

We distinguish between four kinds of biases as shown in Figure 7: IN, OUT, IN-OUT, and NEUTRAL. An IN component listens to events but never fires them. An OUT component fires events but never listens to any. An IN-OUT component both receives and fires events. A component that neither listens nor fires events is a NEUTRAL component.

2. **Number of externally accessible properties.** We distinguish components by their number of properties: π_0 denotes a component that has no properties, π_1 denotes one property, and so on (see Figure 8). π_ϕ denotes a component with *at least* one property.

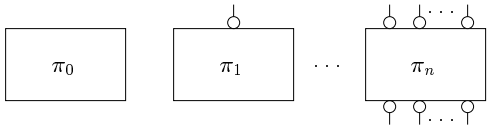

Figure 8: Component properties

3. **Type relationships among connectors**—that is, 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 conforms to the input event class.

   We distinguish two kinds of IN-OUT components, as shown in Figures 9 and 10. $\tau_=$ denotes a component whose output event class is the same as the input event class. $\tau_\neq$ denotes a component whose output event class is different from the input event class. There are two cases of $\tau_\neq$ components. $\tau_\in$ indicates that the output event class is part of the system's events; $\tau_\notin$ says the output event class is external to the system.

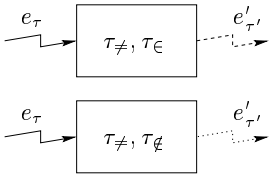

Figure 9: Event types

4. **Payload identity between IN and OUT events.** Among the components whose OUT-event type is the same as the IN-event type, we further distinguish between components that replace the message (or event *payload*) and those that do not. There are two cases of $\tau_=$ components. $\mu_=$ means precisely the same payload is sent out. $\mu_\neq$ indicates that the OUT-event payload is different from the

IN-event payload (although both payloads belong to the same type of event.)

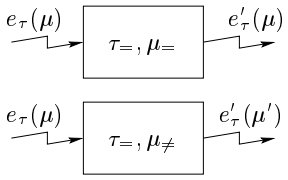

Figure 10: Event payload

5. **Payload mutability**—that is, the relationship between IN and OUT payloads. We further distinguish between $\mu_=$-components that change the payload's state (but preserve its ID) and those that treat the payload as immutable.

Having identified and chosen the component dimensions, we can chart the meaningful combinations and give them names, thereby producing a classification of JB components by dimension.

A TRANSMITTER is an OUT component—a source of events. A RECEIVER is an IN component—a sink of events. A component that is both a TRANSMITTER and RECEIVER is a TRANSCEIVER. Most beans are TRANSCEIVERS. An ISLAND has NEUTRAL bias.

A CONDUIT is a $(\pi_0, \tau_=, \mu_=)$-TRANSCEIVER that passes the incoming event through unchanged. CONDUITS have no functional effect, only side-effects. An example is a tracer, which logs messages for debugging purposes but otherwise has no effect on the computation. A TRANSMUTER, on the other hand, is a $(\pi_0, \tau_=, \mu_=)$-TRANSCEIVER that alters the payload's state but preserves its ID.

A TRANSFORMER is a $(\pi_0, \tau_=, \mu_\neq)$-TRANSCEIVER that fires the same type of event as it received but with different payload. A TRANSDUCER is a $(\pi_0, \tau_\neq, \mu_\neq)$-TRANSCEIVER that fires an event whose type is different than the type of the incoming event—that is, the bean changes the event type, not just the payload. A TRANSLATOR is a TRANSDUCER to an external type, used to connect to other systems. A $(\pi_0, \tau_\neq, \mu_=)$-TRANSCEIVER is a CONVERTER, which changes the event type without changing the payload.

An ASSOCIATOR is a qualifier to any of the above that also has at least one property, i.e., the π_ϕ equivalent. An ASSOCIATOR takes an event whose properties serve as indices or arguments that affect what is sent out.

These classifications give rise to a taxonomy of components that can help designers navigate large component collections. In OOP, there is accepted terminology for common kinds of messages in an object's interface; the resulting taxonomy helps not only in locating a method but also in understanding its intent and the design of the class. Figure 11 depicts the existing taxonomy of features in a typical OO language. For example, an *inspector* never changes the receiver, whereas a *mutator* modifies the object but does not return a value, and a *revealer* is an inspector-mutator whose return type is a reference to an instance variable. Good design rules-of-thumb include separating inspectors from mutators, and avoiding revealers.

Similarly, the classification of methods to components yields a new taxonomy for components. Figure 12 shows an initial terminology for this taxonomy. Corresponding design rules for components are needed too, but they await further research.

Continuing the REFLECTIONBEANS Example
To illustrate the taxonomy, consider the problem of verifying the instantiability of JB.

Statically ensuring instantiability is difficult. Many things can be wrong: the bean class' constructor might be private, the superclass' 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 a builder. Static analysis isn't even an option unless the source code is available, and even a simple analysis for instantiability remains a complex task, if for no other reason than the complexity of parsing.

But one can use reflection instead of parsing. Applying reflection to analyze `.class` files and report potential problems is more flexible and less tedious than parsing: You have the flexibility of analyzing classes without their source code. In fact, BeanLint [8, 9] uses this approach to identify potential problems with JB classes. Some problems that are normally flagged by the compiler—the visibility of superclass constructors, for example—may be assumed correct (although independent verification is always possible).

Another alternative to static checking uses reflection to instantiate the class, catching any exceptions arising from uninstantiability (or any other problem). Indeed, this can be done at component assembly time using REFLECTIONBEANS components. DIN (depth of inheritance [10]) is a useful metric that illustrates the 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 of ensuring fulfillment of the *abstract superclass rule* [7].

In previous work [11], we implemented such a system of REFLECTIONBEANS. In Figure 13, we connect For-

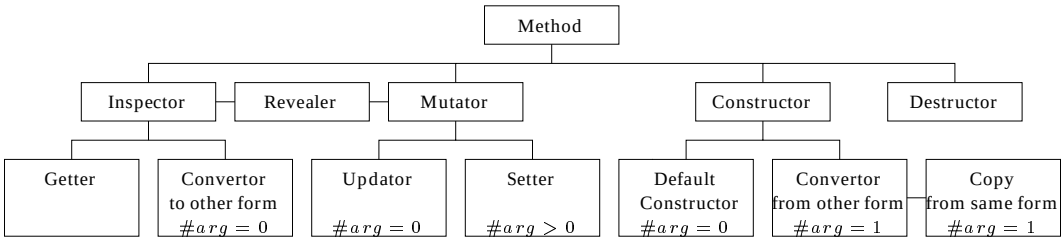

Figure 11: Existing taxonomy of explicit invocation

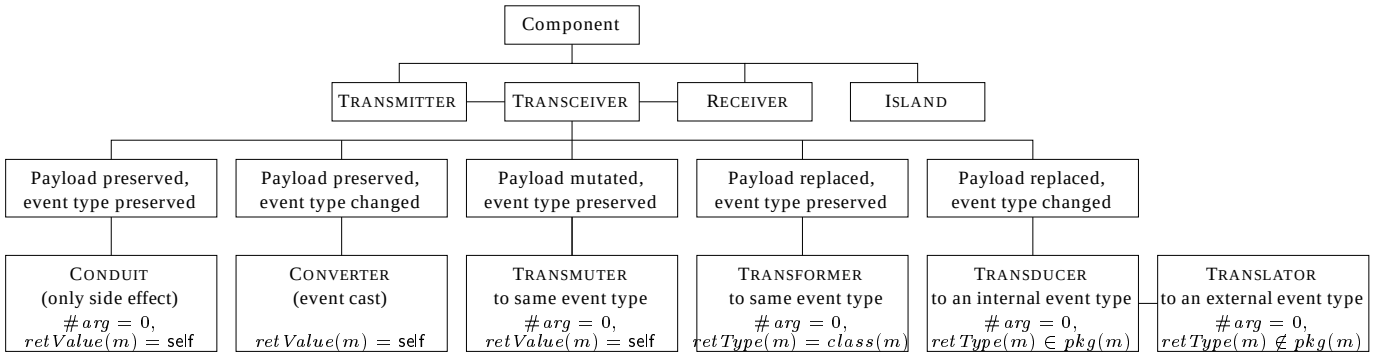

Figure 12: Suggested taxonomy of implicit invocation

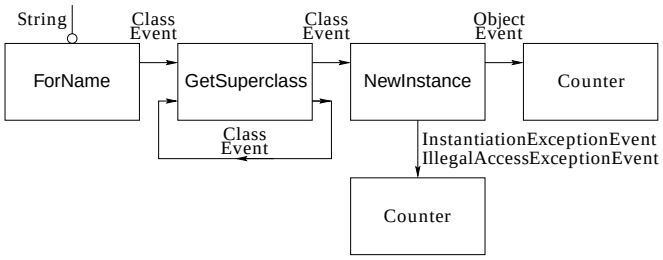

Figure 13: To bean or not to bean?

Name to GetSuperclass, and GetSuperclass to NewInstance, returning 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 superclasses are non-instantiatable and hence abstract de facto.

ForName is an example of a ClassEvent-TRANSMITTER. Generally, static functions and constructors would also be modeled as TRANSMITTERs, and those that return or create a class object would be ClassEvent-TRANSMITTERs. A design that defines support classes (e.g., Transmitter) promotes reuse. Moreover, a customizable ClassEventTransmitter can replace several component types, thereby reducing the overall number of classes.

GetSuperclass is a ClassEvent-TRANSFORMER—that is, a TRANSCEIVER that receives and fires that same kind of event (ClassEvent). NewInstance is a ClassEvent-to-ObjectEvent TRANSDUCER—a TRANSCEIVER whose outgoing event type (ObjectEvent) is different from its incoming event type (ClassEvent).

4 COMPONENT IMPLEMENTATION

In CBP, component compilation is decoupled from component assembly. Such decoupling introduces a new degree of freedom over OOD: A designer must consider not just a component's core functionality but also its behavior at assembly-time. Thus a trade-off emerges between component flexibility and robustness during assembly, prompting new design decisions.

The ideal component is, of course, both flexible and robust. Unfortunately, the two tend toward mutual exclusion. The more specialized the component, the more robust the assembly activity becomes by precluding illegal compositions, but also the more likely it is to preclude some legal ones. Making the component more generic by relaxing static type safety promotes flexibility but brings an increased risk of ill-formed assemblies. A good CBD balances these forces, requiring the component designer to make informed decisions regarding the safety / flexibility trade-off.

So far we have made two design decisions in our examples. The first is that components should model meth-

	Package Name	Applet	Awt	Beans	Io	Lang	Math	Net	Rmi	Security	Sql	Text	Util	total	avg
by # of I/O events	TRANSMITTER	2	445	54	108	322	13	61	107	106	32	73	180	1503	125.25
	RECEIVER	17	991	98	280	100	0	40	34	78	141	71	160	2010	167.50
	TRANSCEIVER	21	1810	130	251	304	68	109	87	297	386	201	502	4166	347.17
	ISLAND	0	8	7	0	29	0	9	9	4	6	0	45	117	9.75
by type of I/O	TRANSFORMER	0	58	1	6	44	38	0	0	0	2	0	7	156	13.00
	TRANSDUCER	21	1752	129	245	260	30	109	87	297	384	201	495	4010	334.17
by # of properties	ASSOCIATOR	21	1810	149	298	427	45	83	114	217	292	173	482	4111	342.58
	non-ASSOCIATOR	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 1: Distribution of components implementing JDK 1.2 packages

ods. The second determined which component category models which method signature, based on the respective taxonomies.[1] Given these decisions, we can concentrate on implementation trade-offs.

Before we do, though, we will expand the scale of application to highlight the consequences of the decisions. In the *Bool* example, we had just a few components; REFLECTIONBEANS had a package-full of components, covering the functionality in the JAVA Core Reflection package. To that we now add all the other JDK packages.

Suppose we wish to create JDKBEANS, letting us treat any method in JAVA as a component. Table 1 gives the distribution of components needed to model JDK 1.2.

There are a few subtleties to consider:

- *Method signature.* Concerning inherited methods: Should each implementation of a method signature yield a new component? Should abstract classes be mapped to components, or just concrete classes? Should interfaces be ignored? Overriding raises another question: Should a method that overrides another yield a new component?
- *Component category.* Should methods with different signatures but belonging to the same taxonomic category (Figure 11) be modeled as a single component from the corresponding category in the CB taxonomy (Figure 12)? Consider inspector methods with different names but matching receiver and return types. Should they be implemented as a single TRANSCEIVER, even though their signatures are different? How can the implementation of a taxonomic category be reused in components across packages?

Four designs points characterize the design possibilities for these components:

[1]The precise mapping of method signatures to component categories appears elsewhere [12].

1. A custom, hard-wired component for every method signature (the *Bool* example).
2. A set of custom, hard-wired components per package for every taxonomic category of component (the REFLECTIONBEANS example repeated for every package).
3. A single, parameterized component per category (the REFLECTIONBEANS example with typing relaxed to allow the same set of components to work across packages).
4. A single, highly parameterized component covering all methods in all packages.

Each of these design points represent a different set of trade-offs and consequences, as summarized in Table 2.

A Custom, Hard-Wired Component Per Method
In this design, every method in every class is transformed to a new component class, just like `nand` and `not` were in the *Bool* example. 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 2. When the signature `Bool not()` is introduced, the Not component shown in Figure 4 can be created without examining the classes True and False. 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*.

Now suppose you override the `not` operation with an alternative implementation. Do you need to generate another Not component?

Recall that the component's polymorphic code makes a new component unnecessary, even if Not originated from an interface. But there's still a difficulty: Not cannot work with components of more specialized types

	Hard-wired (point 1)	Semi-generic (points 2 & 3)	Highly parameterized (point 4)
class granularity	finest	adjustable	coarsest
static safety	full	full for structure; adjustable for data	none
object granularity	fine	fine	fine
run-time overhead	low	adjustable	high

Table 2: The design space for components

without adapters. It's all right to connect a component that models a specialized method to one that models an interface method. But sending an event the other way requires an adapter, because the event types in JAVA must match, independent of payload. Fortunately, the adapter's downcast always succeeds, as the original types are compatible.

This is the most naive transformation from OOD to CBD, and the least maintainable. It is however suitable for a component generator, since maintenance cost is less of an issue if the components are generated automatically. Indeed, we used such a generator to implement the *Bool* and the REFLECTIONBEANS transformations automatically [12], as well as other JDK 1.2 packages (see Table 1). However, there are negative consequences in the long run—namely, an overabundance of components. Over 7000 beans are needed to implement the JAVA core packages plus adapters to circumvent type strictness during assembly. The bright side is that distinct event types preclude ill-formed compositions.

The proliferation of generated components is the main limitation of the naive method-to-component mapping. Traditionally, components are coarse-grained [15], but the transformation does not reflect this view: an entire object-oriented design is decomposed into many simple components. Whereas the coarse granularity of traditional components admits considerable overhead per component, that same overhead becomes prohibitive at finer granularities.

A Set of Custom, Hard-Wired Components Per Package Per Taxonomic Category
This design point follows the REFLECTIONBEANS example but applies it to *all* packages. Each package ends up with its own set of hard-wired TRANSMITTERS, TRANSCEIVERS, etc., for all IN-OUT event combinations. For example, given the Reflection package and the TRANSFORMER category, there will be a ClassEvent-TRANSFORMER, a MethodEvent-TRANSFORMER, and so on. Each component is customized to execute only methods in the corresponding package and classification. For example, the ClassEvent-TRANSFORMER can be set to execute `getSuperclass`.

This approach reduces the number of components without affecting type safety, although it does incur run-time overhead. Whatever else is true of design point 1 is also true here.

A Single, Parameterized Component Per Taxonomic Category
This design point reduces the number of components drastically, to one per taxonomic category. However, because a TRANSFORMER must now play the role of a ClassEvent-TRANSFORMER and a MethodEvent-TRANSFORMER, as well as all other packages' TRANSFORMERS, the flexibility comes at a price in type safety. One can connect, for example, GetSuperclass to GetParameterTypes even though their event types do not match. Other attributes, like bias, are preserved; nothing can be broadcast to ForName, for example, because it is a TRANSMITTER.

A Single, Highly Parameterized Component
This design point represents the opposite extreme from the first: a single, totally generic component is customized to execute *any* method—the all-purpose MethodInvocation component. Its implementation is straightforward. It has a property of type `java.lang.Method`, and it implements a method-property customizer to allow executing any method in JAVA.

This setup eradicates the problem of too many component types, at least in theory: just one polymorphic component type covers all cases. It doesn't work so easily in practice, however. One reason is the static nature of JB bias. MethodInvocation needs both input and output events to cover all possible biases, even if some components are not really IN-OUT. Moreover, one can now connect an output to the (inactive) input of a component that models a constructor, or to the (inactive) output of a component that models a method with no return value.

In sum, this design is statically simplest by class count but least efficient, since methods must be invoked using reflection at run-time. It also offers the least static type safety: all methods are mapped to a single component, and so all classes are mapped to a single event type. Component compositions representing mistyped

method invocations are thus permitted.

Lessons

On the one hand, a single, highly parameterized component constitutes the most maintainable design, but it removes all measures of static type safety. Custom, hard-wired components, on the other hand, provide full static safety, but scalability concerns make them useful only for simple object-oriented designs like the Boolean example or possibly high-function classes with small interfaces. But clearly, the first design point is not practical for most real-world class libraries—there are simply too many methods, giving rise to too many components. A generator can create them easily enough, but managing them will be tedious.

Partitioning components into packages mitigates the problem somewhat, as do abstract components. But the real solution will come from using a family of generic components rather than customized ones or a single, highly parameterized one. The component taxonomy can come to the rescue here—*if* we use it to produce a convenient number of reasonably efficient generic components (that is, a suitably parameterized component for judiciously chosen points in the space).

5 RELATED WORK

Discussing JB-based design in terms of components and connectors reflects the terminology of software architecture [13], particularly architectural styles as characterized by Shaw and Garlan [15]. An architectural style is a set of constraints that defines a family of designs. As such it does not articulate design or implementation trade-offs.

Another tack on CBD comes from the object-oriented methodology community as exemplified by Catalysis [3]. This approach uses two basic constructs, objects and actions, to model components and events. The notion of *styles* in Catalysis is particularly relevant to our work. For example, a specification adhering to the JB style can make automatic implementation feasible [2]. To date, nothing has been published on realizing this possibility.

Meanwhile, many programming environments offer wizards that can generate sets of features within a component. They alleviate the tedium of writing `get`, `set`, and notification methods on bean classes as opposed to creating systems of beans from more abstract representations or to reasoning about their design.

Compared to commercial CBP environments, we have widened our focus to consider the design of complete systems of components. Compared to the architectural community, we have narrowed our focus to design decisions that are close to the implementation level. The mapping to JB forced us to consider implementation issues and their impact on design—and to put them to the test:

- *Component/connector partitioning.* By its nature, software architecture concentrates on abstraction mechanisms. These answer questions regarding whether a single abstraction is sufficient to model both components and connectors, whether two abstractions are equally powerful (e.g., Is "connector" a first-class abstraction? [6]), and so on. In contrast, this work takes the component and connector abstractions as givens and considers their impact on CBD.

- *Component/connector design.* Few taxonomies of components and connectors even mention design decisions and trade-offs. An exception is the work of Mehta, Medvidovic, and Phadke [14]. They too are motivated by the need for better design guidance, but they defer its implications for design decision-making to future work. We feel that design and implementation guidelines make a taxonomy even more useful.

  Still, though we emphasize design choices, the scope here is not as broad as Mehta, et al.'s, given the focus on implicit invocation in the context of JB. Our work falls into the "invocation" dimension of their taxonomy.

- *Component/connector implementation.* Component assembly introduces trade-offs unique to CBD, as we have seen. In JB, the visual builder governs component interactions. The same design and implementation issues apply to components whose interaction is governed by middleware instead of a builder. The taxonomy and the principles behind it transcend the medium of assembly.

6 CONCLUSION

Much has been written about OOD, with few practical advances in automating the process. In CBP the opposite seems true: A high degree of automation exists, but little attention is paid to CBD. Most JB books, for example, focus on what beans are, rather than on how to build software that takes advantage of the JB architecture.

We set out to advance the state of the art in designing and implementing components. To that end, we demonstrated how to adjust and supplement OOD principles to suit CBD. We also devised a taxonomy of design for a particular style of interconnection, namely event-based composition, by considering component classes and events and classifying their roles in real applications. Finally, we characterized a range of implementations of this event-based, dataflow style, taking

into account the assembly activity and the safety, performance, and maintenance trade-offs it engenders.

This work is also a modest attempt to bridge the gap between architectural descriptions and actual development practice. The architectural perspective tends to be too abstract to be practical, while JB tools are too practical to offer the leverage of architectural abstraction. As a result, working programmers have little to guide them in the design of JB systems.

Implicit invocation is gaining ground as an implementation technique. We characterized the explicit invocation taxonomy and proposed a corresponding taxonomy of implicit invocation, which extends the vocabulary of JB. Together, the two taxonomies can aid not only in comprehension of the design space for new designs—using either explicit or implicit invocation—but also in migrating existing systems to implicit invocation [12].

There are at least two logical follow-ups to this work. Documenting design decisions in terms of the proposed taxonomies would promote collection and dissemination of OOD and CBD expertise alike. It would also invite tool and language support for the expertise so obtained, especially as it regards visual builders. We envision a component generator with knobs that let a user choose among design decisions, akin to similar work for code generation from design patterns [1]. Such a tool is sufficiently parameterized to generate any combination of implementation trade-offs on the spectrum from fully static to fully dynamic components. The components that result can thus exhibit just the right combination of flexibility, maintainability, and performance, with a minimum of specification and hackery on the developer's part.

ACKNOWLEDGEMENT We thank Don Batory and the anonymous referees for their valuable comments.

REFERENCES

[1] F. Budinsky, M. Finnie, J. Vlissides, and P. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, 1996.

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

[3] D. F. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Object Technology. Addison-Wesley, 1998.

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

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

[6] L. Howard. Components and connectors are the assembly language of architectural description: compositions deserve first-class status (and better abstraction mechanisms). In *Joint Proceedings of the Second International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints '96) on SIGSOFT '96 Workshops*, pages 44–46, 1996.

[7] 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.

[8] M. Johnson. BeanLint: A JavaBeans troubleshooting tool, part 1. *JavaWorld*, 3(12), Dec. 1998.

[9] M. Johnson. BeanLint: A JavaBeans troubleshooting tool, part 2. *JavaWorld*, 4(1), Jan. 1999.

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

[11] D. H. Lorenz. Visitor Beans: An aspect-oriented pattern. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology. ECOOP'98 Workshop Reader*, number 1543 in Lecture Notes in Computer Science, pages 431–432. AOP Workshop Proceedings, Brussels, Belgium, Springer Verlag, July 20-24 1998.

[12] D. H. Lorenz and J. Vlissides. Automated architectural transformation: Objects to components. Technical Report NU-CCS-00-01, College of Computer Science, Northeastern University, Boston, MA 02115, Apr. 2000.

[13] N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description languages. In *Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT symposium on Software engineering*, pages 60–76, Zurich, Switzerland, Sept. 22-25 1997.

[14] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Proceedings of the 22^{nd} International Conference on Software Engineering*, pages 178–187, Limerick, Ireland, June 4-11 2000.

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

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