Appeared in *Proceedings of the 24th International Conference on Software Engineering* (ICSE 2002), pages 208–218, Orlando, FL, May 19–25, 2002.

Static and Dynamic Structure in Design Patterns

Eric Eide
eeide@cs.utah.edu

Alastair Reid
reid@cs.utah.edu

John Regehr
regehr@cs.utah.edu

Jay Lepreau
lepreau@cs.utah.edu

University of Utah, School of Computing
50 South Central Campus Drive, Room 3190
Salt Lake City, Utah 84112–9205
http://www.cs.utah.edu/flux/

ABSTRACT

Design patterns are a valuable mechanism for emphasizing structure, capturing design expertise, and facilitating restructuring of software systems. Patterns are typically applied in the context of an object-oriented language and are implemented so that the pattern participants correspond to object instances that are created and connected at run-time. This paper describes a complementary realization of design patterns, in which many pattern participants correspond to statically instantiated and connected components.

Our approach separates the static parts of the software design from the dynamic parts of the system behavior. This separation makes the software design more amenable to analysis, thus enabling more effective and domain-specific detection of system design errors, prediction of run-time behavior, and more effective optimization. This technique is applicable to imperative, functional, and object-oriented languages: we have extended C, Scheme, and Java with our component model. In this paper, we illustrate our approach in the context of the OSKit, a collection of operating system components written in C.

1. INTRODUCTION

Design patterns allow people to understand computer software in terms of stylized relationships between program entities: a pattern identifies the roles of the participating entities, the responsibilities of each participant, and the reasons for the connections between them. Patterns are valuable during the initial development of a system because they help software architects outline and plan the static and dynamic structure of software before that structure is implemented. Documented patterns are useful for subsequent system maintenance and evolution because they help maintainers understand the software implementation in terms of well-understood, abstract structuring concepts and goals.

This research was largely supported by the Defense Advanced Research Projects Agency, monitored by the Air Force Research Laboratory, under agreements F33615–00–C–1696 and F30602–99–1–0503. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon.

The conventional approach to realizing patterns [13] primarily uses classes and objects to implement participants and uses inheritance and object references to implement relationships between participants. The parts of patterns that are realized by classes and inheritance correspond to static information about the software—information that can be essential for understanding, checking, and optimizing a program. Unfortunately, class structures can disguise the underlying pattern relationships, both by being too specific (to a particular application of a pattern) and by being mixed with unrelated code. In contrast, the parts of patterns realized by run-time objects and references are more dynamic and flexible, but are therefore harder to understand and analyze.

This paper describes a complementary approach to realizing patterns based on separating the static parts of a pattern from the dynamic parts. The *static participants* and relationships in a pattern are realized by component instances and component interconnections that are set at compile- or link-time, while the *dynamic participants* continue to be realized by objects and object references. Expressing static pattern relationships as component interconnections provides more flexibility than the conventional approach while also promoting ease of understanding and analysis.

To illustrate the tradeoffs between these approaches, consider writing a network stack consisting of a TCP layer, an IP layer, an Ethernet layer, and so on. The usual implementation strategy, used in mainstream operating systems, is for the implementation of each layer to directly refer to the layers above and below except in cases where the demand for diversity is well understood (e.g., to support different network interface cards). This approach commits to a particular network stack when the layers are being written, making it hard to change decisions later (e.g., to add low-level packet filtering in order to drop denial-of-service packets as early as possible).

An alternate implementation strategy is to implement the stack according to the *Decorator*[1] pattern with objects: each layer is implemented by an object that invokes methods in objects directly above and below it. The objects at each layer provide a common interface (e.g., methods for making and breaking connections, and for sending and receiving packets), allowing the designer to build a large variety of network stacks. In fact, stacks can be reconfigured at run-time, but that is more flexibility than most users require.

Our design and implementation approach offers a middle ground. Having identified the *Decorator* pattern and having decided that the network stack may need to be reconfigured, but not at run-time, each decorator would be implemented as a component that imports an interface for sending and receiving packets and exports the same interface. The choice of network stack is then statically expressed

[1]Unless otherwise noted, the names of specific patterns refer to those presented in Gamma et al.'s *Design Patterns* catalog [13].

by connecting a particular set of components together.

The basis of our approach is to permit system configuration and realization of design patterns at *compile-* and *link-time* (i.e., before software is deployed) rather than at *init-* and *run-time* (i.e., after it is deployed). Components are defined and connected in a language that is separate from the base language of our software, thus allowing us to separate "configuration concerns" from the implementation of a system's parts. A system can be reconfigured at the level of components, possibly by a non-expert, and can be analyzed to check design rules or optimize the overall system. Our approach helps the programmer identify design trade-offs and strike an appropriate balance between design-time and run-time flexibility.

The contributions of this paper are as follows:

- We describe an approach to realizing patterns that clearly separates the static parts of the design from the dynamic parts, making the system more amenable to optimization and to analyses that detect errors or predict run-time behavior (Section 3).
- We define a systematic method for applying our approach to existing patterns (Section 3.1).
- We show that our approach is applicable to three major programming language paradigms that support the unit component model: imperative languages, exemplified by C [21]; functional languages, exemplified by Scheme [11]; and object-oriented languages, exemplified by Java [17] (Sections 2 and 3). We demonstrate our approach with two examples from the OSKit [12], a set of operating system components written in C (Sections 3.2 and 3.3).
- We evaluate the approach by applying it to each pattern described by Gamma et al. [13] (Section 3.4) and by analyzing its costs and benefits (Section 4).

In summary, although the benefits of separating system architecture from component implementations are well-known, the distinctive features of this paper are that: we show a general approach that can be applied to many patterns and in multiple language paradigms; we consider the static-dynamic decision in the context of design patterns; and we thoroughly evaluate when to apply and when not to apply our approach.

2. THE UNIT MODEL

Our approach to realizing patterns is most readily expressed in terms of *units* [10,11], a component definition and linking model in the spirit of the Modula-3 and Mesa [19] module systems. The unit model emphasizes the notion of components as reusable architectural elements with well-defined interfaces and dependencies. It fits well with the definitions of "component" in the literature [22, p. 34] but differs from other component models that emphasize concerns such as separate compilation and dynamic component assembly. In the unit model, components are compile- or link-time parts of an assembly: i.e., software modules, not run-time objects.

Three separate implementations of the unit model exist: *Knit* [21] for C, *Jiazzi* [17] for Java, and *MzScheme* [11] for Scheme. The implementations differ in details both because of technical differences in the base languages and because of stylistic differences in the way the base languages are used. For the purposes of this paper, we focus on the common features of the three implementations.

2.1 Atomic and Compound Units

An *atomic unit* can be thought of as a module with three parts: (1) a set of *imports* that name the dependencies of the unit, i.e., the definitions that the unit requires; (2) a set of *exports* that name the definitions that are provided by the unit and made available to other units; and (3) an implementation, which must include a definition for each export, and which may use any of the imports as required.

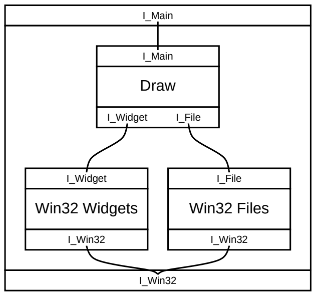


Figure 1: Atomic and compound units

Each import and export is a *port* with a well-defined *interface*. An interface has a name and serves to group related terms, much like an interface or abstract class in an OOP language. The three implementations of the unit model make different choices about what makes up an interface. In Knit, an interface refers to sets of related C types, function prototypes, and variable declarations. In Jiazzi, port interfaces are like Java packages: they describe partial class hierarchies and the public methods and fields of classes. In MzScheme, because Scheme uses run-time typing, interfaces are simply lists of function names.

Definitions that are not exported are inaccessible from outside the unit. The implementation of a unit is usually stored in a file separate from the unit definition, allowing code that was not intended for use as a unit to be packaged up as a unit.

Although all implementations of the unit model use a textual language to define units, in this paper we use a graphical notation to avoid inessential details and to emphasize the underlying structure of our systems. The smaller boxes in Figure 1 represent atomic units. The export interfaces are listed at the top of a unit, the import interfaces are listed at the bottom, and the name of the unit is shown in the center. Consider the topmost unit, called Draw. It has the ability to load, save, and render images, encapsulating the main parts of a simple image viewing program. Draw exports (i.e., implements) one port with interface I_Main and imports two ports: one with interface I_Widget and a second with interface I_File.

Units are instantiated and interconnected in *compound units*. Like atomic units, compound units have a set of imports and a set of exports that define connection points to other units. The implementation of a compound unit consists of a set of unit instances and a set of explicit interconnections between ports on these instances and the imports and exports of the compound unit. The result of composing units is a new unit, which is available for further linking.

Figure 1 as a whole represents a compound unit composed of three other units. In this figure, an instance of Draw is composed with an instance of Win32 Widgets and an instance of Win32 Files. Within a compound unit, connections are defined *explicitly*: this is necessary when there is more that one way to connect the units. Although not shown in this example, a system designer may freely create multiple unit instances from a single unit definition (e.g., two instances of Draw).

2.2 Exploiting Static Configuration

One of the key properties of programming with the unit component model is that component instantiation and interconnection are performed when the program is built instead of when the program is executed. This allows implementations of the unit model to make use of additional resources that may be available at compile- and link-time: powerful analysis and optimization techniques; in the case of embedded systems, orders of magnitude more cycles and memory with which to perform analyses; test cases, test scaffolding, and debugging builds; and finally, freedom from real-world constraints such as real-time deadlines. All three unit implementations check the component composition for type errors. Knit, which implements units for C, provides additional features that exploit the static nature of unit compositions.

Constraint checking. Even if every link in a unit composition is "correct" according to local constraints such as type safety, the system *as a whole* may be incorrect because it does not meet global constraints. For example, [21] describes a design constraint used by operating system designers: "bottom-half code," executed by interrupt handlers, must not invoke "top-half code" that executes in the context of a particular process. The reason is that while top-half code typically blocks when a resource is temporarily unavailable, storing its state in the process's stack, an interrupt handler lacks a process context and therefore must not block. The problem with enforcing this constraint is that units containing bottom-half code (e.g., device drivers) may invoke code from other units that, transitively, invokes a top-half unit. Keeping track of such conditions is difficult, especially when working with low-level systems code that is highly interconnected and not strictly layered. To address this problem, Knit unit definitions can include constraint annotations that describe the properties of imports and exports. Constraints can be declared explicitly (e.g., that imported functions are invoked by bottom-half code) or by description (e.g., that the import properties are set by the exports). At system build-time, Knit propagates unit properties in order to ensure that all constraints are satisfied.

Cross-component inlining. When source is available, Knit inlines function definitions across component boundaries with the help of the C compiler. By eliminating most of the overhead associated with componentization, Knit reduces the need to choose between a clean design and a fast implementation.

2.3 Using Units Without Language Support

The unit model makes it possible for a software architect to design a system from components, describe local and global relationships between components, and reuse components both within and across system designs. These are the features that make it useful to develop and apply units for expressing design patterns. In particular, our unit-based approach to realizing patterns relies on these features of the unit model:

- **Programming to interfaces.** The only connections between components are through well-typed interfaces.
- **Configurable intercomponent connections.** Unit imports describe the "shapes" but not the providers of required services. A system architect links unit instances as part of a system definition, not as part of a component's base (e.g., C or Java) implementation.
- **Static component instantiation and interconnection.** Units are instantiated and linked when the system is built, not when the system is run.
- **Multiple instantiation.** A single unit definition can be used to create multiple unit instances, each of which has a unique identity at system build-time. Each instance can be linked differently.

It is possible to make use of features of the unit component model without support from languages such as Knit, Jiazzi, and MzScheme. However, without support, some benefits of the model may be lost. For instance, a C++ programmer might use template classes to describe units: this can provide optimization benefits but does not help the system designer check constraints of the sort described previously. A C programmer might use the C preprocessor to achieve similar results. In sum, although unit tools can provide important benefits, people who cannot or decide not to use our unit description languages can nevertheless take advantage of our general approach to realizing design patterns.

3. EXPRESSING PATTERNS WITH UNITS

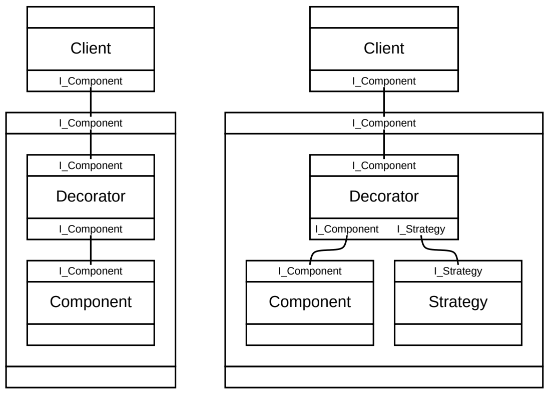
The essence of a design pattern is the set of participants in the pattern and the relationships between those participants. As outlined previously, the conventional approaches to describing and realizing patterns are based on the idioms of object-oriented programming. At design-time, the participants in the pattern correspond to classes. At run-time, the pattern is realized by object instances that are created, initialized, and connected by explicit statements in the program code. This style of implementation allows for a great deal of run-time flexibility, but in some cases it can disguise information about the static properties of a system—information that can be used to check, reason about, or optimize the overall system.

The key idea of this paper is that it is both possible and fruitful to separate static knowledge about a pattern application from dynamic knowledge. In particular, we believe that static information should be "lifted out" of the ordinary source code of the system, and should be represented at the level of unit definitions and connections. The unit model allows a system architect to describe the static properties of a system in a clear manner, and to separate "configuration concerns" from the implementations of the system's parts.

Consider, for example, an application of the *Decorator* pattern: this pattern allows a designer to add additional responsibilities to an entity (e.g., component or object) in a way that is transparent to the clients of that entity. One might apply *Decorator* to protect a non-thread-safe singleton component with a mutual exclusion wrapper (which acquires a lock on entering a component and releases the lock on exit) when using the component in a multi-threaded environment. In an object-oriented setting, this pattern would often be realized by defining three classes: one abstract class to define the component interface, and two derived classes corresponding to the concrete component and decorator. At init-time, the program would create instances of each concrete class and establish the appropriate object connections. While workable, this implementation of the pattern can disguise valuable information about the static properties of this system. First, it hides the fact that there will be only one instance each of the component and decorator. Second and more important, it hides the design constraint that the base component must be accessed only through the decorator: because the realization of the pattern doesn't enforce the constraint, future changes to the program may violate the rule.

To overcome these problems, we would realize the *Decorator* pattern at the level of units, as illustrated in Figure 2(a). We create one unit definition to encapsulate the base component definition; by instantiating this definition exactly once, we make it clear that there will be only one instance in the final program. Furthermore, we annotate the unit definition with the constraint that the implementation is non-thread-safe. We then create a separate unit definition to encapsulate our decorator, and include in the definition a specification that it imports a non-thread-safe interface and exports a thread-safe one. The resulting structure in Figure 2(a) makes it clear that there is one instance of each participant and that there is no access to the base component except through the decorator. Units make the static structure of the system clear, and unit compositions can be checked by tools to enforce design constraints. Of course, unit definitions are reusable between systems (and within a single system): we can include the decorator instances only as needed. If we desire greater reuse, we can apply the *Strategy* pattern to our decorator to separate its wrapping and locking aspects as shown in Figure 2(b). This structure provides greater flexibility while still allowing for cross-component reasoning and optimization when the strategy is statically known.

In sum, our approach to realizing patterns promotes the benefits of static knowledge within patterns by moving such information to



(a) Decorator (b) Strategized Decorator

Figure 2: Units realizing *Decorator* patterns

the level of units. The unit model allows us to describe and separate the static and dynamic properties of a particular pattern application, thus making it possible for us to exploit the features described in Section 2.2. In the sections below we define a method for applying our approach, demonstrate the method in detail on a small example, demonstrate the effect of our method on a large example, and consider how the method applies to each of the patterns in Gamma et al.'s *Design Patterns* catalog [13].

3.1 A Method for Expressing Patterns with Units

In realizing a pattern via units, the software architect's task is to identify the parts of the pattern that correspond to static (compile-time or link-time) knowledge about the pattern and its participants, to "lift" that knowledge out of the implementation code, and then to translate that knowledge into parts of unit definitions and connections. This process is necessarily specific to individual uses of a pattern: each time a pattern is applied, the situation dictates whether certain parts of the pattern correspond to static or dynamic knowledge. In our experience, however, we have found that many patterns are commonly applied in situations that provide significant amounts of static information, and which therefore allow system architects to exploit the features of the unit model.

We have found the following general procedure to be useful in analyzing the application of a pattern and translating that pattern into unit definitions, instances, and linkages. Because patterns are ordinarily described in terms of object-oriented structures (classes, interfaces, and inheritance), we describe our method as a translation from object-oriented concepts to parts of the unit model.

1. Identify the abstract classes/interfaces. Many pattern descriptions contain one or more participating abstract classes that serve to define a common interface for a set of derived classes. The abstract classes therefore serve the same purpose as interfaces in the unit model; the three implementations of the model all allow related operations (and types, if needed) to be grouped together in named interfaces. In Figure 2(a), for example, I_Component corresponds to the abstract component class described in the *Decorator* pattern. The exact translation from abstract class to unit interface depends on whether or not the derived classes are "static participants" in the application of the pattern at hand, as described next.

2. Identify the "static" and "dynamic" participants within the pattern. Within the context of a pattern, it is often the case that some pattern participants will be realized by a small and statically known number of instances. For example, in uses of the *Abstract Factory* pattern (see Section 3.2), there will often be exactly one *Concrete Factory* instance in the final system (within the scope of the pattern). The number of instances does not need to be exactly one: what is important is that the number of instances, their classes, and the inter-instance references are all known statically.

We refer to these kinds of participants as *static participants*, and in the steps below, we realize each of these participants as an individual unit instance—essentially, realizing the participant as a part of our static architecture, rather than as a run-time object. In Figure 2(a), the context of our example says that in this particular use of *Decorator*, both the base component and its decorator are singletons. Thus, they are static participants.

If a pattern participant is not static we refer to it as a *dynamic participant*. In this case, we will translate the participant as a unit that will encapsulate the participant class and will be able to produce instances at run-time. Figure 2(a) has no dynamic participants; later examples will show their use.

3. Define the interfaces for static participants. Following the class hierarchy of the pattern, the software architect defines the unit interfaces to group the operations that will be provided by the static participants. The architect may choose to create one interface per class (i.e., one interface for the new operations provided at each level of the class inheritance hierarchy), or may group the operations at a finer granularity.

Because each instance of a static participant will be implemented by a unique unit instance in the realization of the pattern, the identity of each instance is part of the static system architecture and need not be represented by an object at run-time. Therefore, in the translation from class to unit interface, the methods that constitute the interface to a static participant can be translated as ordinary functions (or as class static methods, in the case of Jiazzi), and data members can be translated as ordinary variables (static members). Any method arguments that represent references to static participants can be dropped from the translated function signatures: these arguments will be replaced by imports to the unit instances (i.e., explicit, unit-level connections between the static participants).

Thus, in the running example of Figure 2(a), the operations in I_Component can be implemented by ordinary functions. Because all of our participants are static, we do not need to represent them as run-time objects.

4. Define the interfaces for dynamic participants. Following the class hierarchy of the pattern, the designer now creates the interfaces for the dynamic participants. As described for the previous step, the designer may choose to create one or several interface definitions per class.

Unlike the static case, each instance of a dynamic participant must be represented by a run-time object (or other entity in a non-OOP language). This means that in translating the participant class to unit interfaces, the designer must include the definition of the type of the run-time objects, as the implementation language requires. With Jiazzi, this is straightforward: Jiazzi unit interfaces contain Java class definitions. In Knit, the interface would include a C type name along with a set of functions, each of which takes an instance of that type as an argument (i.e., the "self" parameter). MzScheme is the simplest: because Scheme uses run-time typing, the unit interface does not need to include the type of the dynamic pattern participants at all.[2]

[2]If the pattern structure relies on implementation inheritance, dynamic method dispatch, or other essentially OOP features, these capabilities must be emulated when translating the pattern to Knit or MzScheme units. In our experience, this is sometimes tedious but generally not too difficult.

Although the interfaces for a dynamic participant must include the class of the participant objects, the unit model allows the designer to avoid hard-coding class inheritance knowledge into the interfaces. By writing our units so that they import the superclasses of each exported class, we can implement our dynamic participant classes in a manner corresponding to *mixins* [8,17]. In other words, we can represent the static inheritance relationships between pattern participants not in the definitions of our units or in the unit interfaces, but in the connections between units.

5. Create a unit definition for each concrete (static or dynamic) participant. With the interfaces now defined, the designer can write the definitions of the units for each participant. The unit definition for a dynamic participant encapsulates the *class* of the dynamic instances; normally, in the context of a single pattern, these kinds of units will be instantiated once. The unit definition for a static participant, on the other hand, encapsulates a single *instance* of the participant. The unit definition for a static participant may be instantiated as many times as needed, each time with a possibly different set of imports, to create all of the needed static participant instances. In either case, the exports of a unit correspond to the services that the participant provides. The imports of a unit correspond to the connections described in the pattern; the imports of each unit instance will be connected to the exports of other unit instances that represent the other (static and dynamic) participants.

Continuing the example of Figure 2(a), the software designer writes definitions for the Component and Decorator units, each encapsulating a single instance of the corresponding participant. The base component has an I_Component export, while the Decorator both imports and exports that interface.

6. Within a compound unit definition, instantiate and connect the participant units. Within a compound unit, the designer describes the instantiation of the pattern as a whole. The implementation of the compound unit specifies how the participant units are to be instantiated and connected to one another. The connections between units follow naturally from the structure of the pattern and its application in the current context. In addition, one must import services that are required by the encapsulated participants.

The above considers just one pattern applied before any code is written. In practice, participants have roles in multiple patterns and patterns are applied during code evolution. These considerations necessitate changes such as omitting the enclosing compound unit, moving some participants outside the compound unit, or choosing to treat a static participant as dynamic (or vice versa) to avoid extensive changes to the implementations of the participants. The system designer may want to make additional changes, such as aggregating groups of interfaces into single interfaces, to reduce the complexity of the unit descriptions.

3.2 Example: Managing Block Devices

We illustrate our approach in the context of a concrete system. The OSKit [12] is a collection of components for building operating systems and standalone systems. The components are almost all written in C, with a few in assembly code. Although the OSKit includes a number of small and modest-sized "from-scratch" components, such as memory and thread management, the majority of its code is taken from elsewhere, including FreeBSD, NetBSD, Linux, and the Mach research operating system. The OSKit consists of over 1,000,000 lines of code, most of which is being independently maintained by the developers of the "donor" systems. At this time, about 40% of the OSKit has been explicitly converted to Knit units. Although the OSKit is written in C, some parts are distinctly object-oriented: a lightweight subset of Microsoft's COM is used in a number of places. The OSKit has been used to build large systems such as operating system kernels and file servers, to implement advanced languages directly on the hardware, and for smaller projects such as embedded systems and bootloaders.

As an initial example, consider the problem of managing block I/O device drivers, which provide low-level access to block-oriented storage media such as disks and tapes. An operating system is generally configured at build-time to include one device driver for each kind of supported block device: e.g., IDE disk, SCSI disk, and floppy disk drive. At run-time, the operating system queries each driver for information (e.g., the type and capabilities of the driver): the driver discovers the physical devices that it manages, and at the request of the OS, creates run-time objects to represent each of these devices. To make it easy to configure OSKit-based systems with different sets of block device drivers, we apply the *Abstract Factory* pattern as illustrated in Figure 3. In OOP terms, we define a common abstract class (BlockDevice) to be supported by all block devices, and we define abstract classes (BlkIO and DriverInfo) for the products that each driver may produce. The actual drivers and products map to concrete classes as shown.

Having identified the pattern at hand, we can now apply the steps of our method to translate the pattern structure into appropriate unit definitions. First (step 1) we identify the abstract classes: clearly, these are BlockDevice, BlkIO, and DriverInfo. Next (step 2): because each device driver can manage multiple physical devices, we need at most one instance of each driver in any system we might build. (We need zero or one, depending on whether or not we choose to support a particular kind of device.) Thus, each of our concrete factories is a static participant. In contrast, since we do not know the number of physical devices that will be present at run-time, each concrete product class is a dynamic participant.

We now define the interfaces for our static participants (step 3). The interface to each concrete factory class is defined by the abstract BlockDevice class: we therefore define a corresponding I_BlockDevice interface. As described in Section 3.1, we translate the BlockDevice methods into ordinary C functions, because we do not need to represent our static participants as run-time objects.

In defining the interfaces for our dynamic participants (step 4), we need to translate the participant's methods in a way that allows us to identify instances at run-time. Because we are using Knit, we translate the BlkIO and DriverInfo classes into unit port interfaces that include C types for the products. In addition, each product method becomes a C function that takes a run-time instance.

Next (step 5) we create the unit definitions for each of our concrete participants. This is a straightforward mapping from the pattern structure: the exports of each unit are determined by the provided interfaces (i.e., the participants' classes), and the imports are determined by the connections in the pattern structure.

Finally, we create a compound unit in which we instantiate the units that we need, and connect those instances according to the pattern (step 6). For example, to create a system with just IDE support, we would define the unit instances and links shown in Figure 4. The unit definitions that we created in steps 1–5 are reusable for many systems, but the structure of the final unit composition in step 6 is often specific to a particular system configuration.

Our method describes the process of creating appropriate unit definitions, but it does not address the problem of unit implementation: i.e., the source code. We have found, however, that appropriate implementation is often straightforward. In the example above, the OSKit units are implemented by existing OS device drivers with little or no modification. Most changes, if needed at all, can be implemented by *Adapter* units that wrap the existing code. Furthermore, the device-specific code can be isolated in the units that define our products. This means that we can write one unit defini-

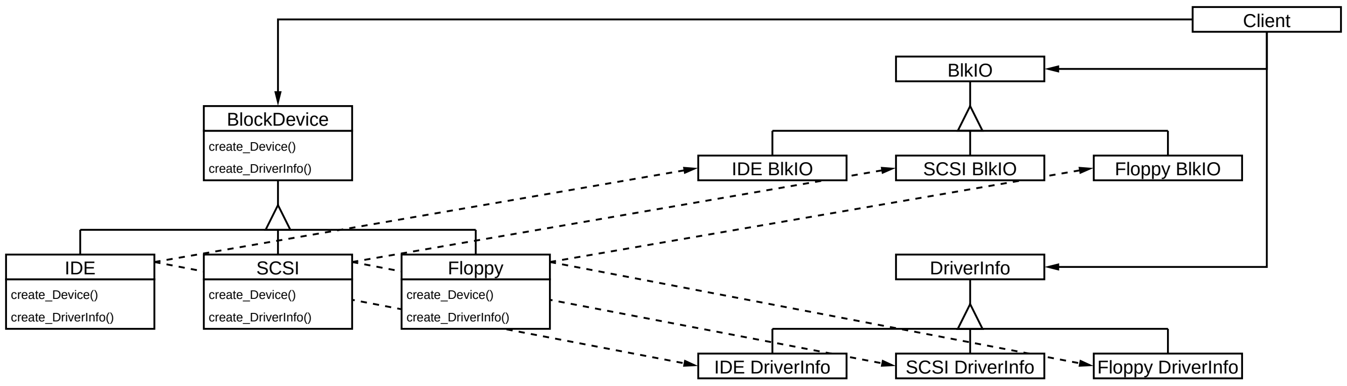


Figure 3: Using the *Abstract Factory* pattern to manage block devices in OSKit-based systems

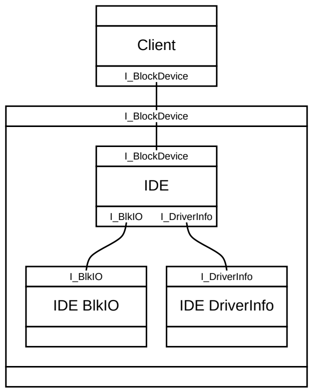


Figure 4: Result of applying our method to Figure 3

tion for our factory instead of one each for IDE, SCSI, and Floppy. Each instance of this factory imports the units that define a related family of products. Knit's constraint system can be used to statically ensure that the system designer does not accidentally connect a mismatched set of products.

3.3 Example: OSKit Filesystems

Having illustrated the method in detail in the previous section, we now show the result of applying the method to a more complex example. Figure 5 shows one possible configuration of a filesystem in the OSKit. The primary parts of the system are: Main, an application that reads and writes files; FS Namespace, which implements filepaths (like `/usr/bin/latex`) on top of the more primitive file and directory abstraction; Ext2FS, a filesystem from the Linux kernel distribution; and Linux IDE, a Linux device driver for IDE disks. The other units in the system connect these primary parts according to the *Abstract Factory*, *Adapter*, *Decorator*, *Strategy*, *Command*, and *Singleton* patterns. All participants in these patterns are currently implemented as described with one exception (*Command*) described below.

Abstract Factory. Figure 5 contains two abstract factories: the Linux IDE and OSEnv/x86 units. (In both cases, only the enclosing compound unit is shown.) The OSKit defines an interface (called the "OS Environment Interface") for all components to use when manipulating interrupts, setting timers, allocating memory, and so on. This interface abstracts the more obtrusive details of the underlying platform. In Figure 5, this interface is implemented by OSEnv/x86 for the Intel x86 hardware but we could have chosen OSEnv/StrongARM for the StrongARM architecture or OSEnv/Linux to run as a user-mode Linux program. (The latter choice would necessitate a different choice of device driver.) It is appropriate to fix on a particular platform at this stage because moving to another would require the system to be rebuilt.

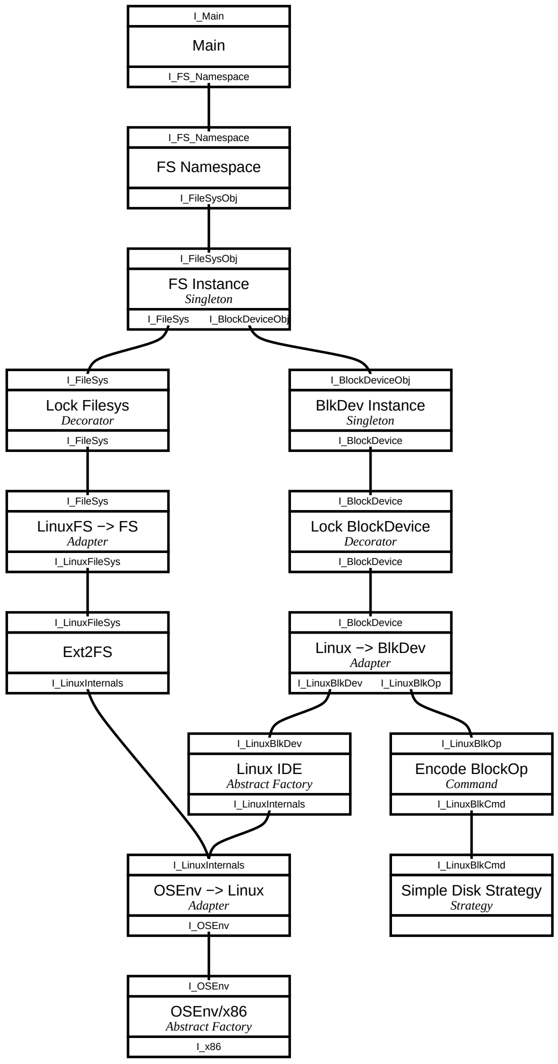


Figure 5: A possible configuration of an OSKit filesystem

Adapter. The hybrid nature of the OSKit gives rise to many adapters. The OSEnv→Linux adapter implements Linux internal interfaces in terms of the OSKit-standard I_OSEnv, allowing us to include Linux-derived units in the system. The LinuxFS→FS and Linux→BlkDev adapters implement standard OSKit interfaces for filesystems and block devices using the internal Linux interfaces for these things. Being able to use Linux-derived units is extremely useful for OSKit systems: instead of writing and maintaining new filesystems and device drivers, the OSKit exploits the hard work of the Linux community. The OSKit uses this approach to provide 30 Ethernet drivers, 23 SCSI drivers, and 11 filesystems.

An interesting part of the LinuxFS→FS and Linux→BlkDev adapters is that they have both static and dynamic aspects. The static aspect adapts the static interfaces of the participants: those used for initialization, finalization, and mounting a filesystem on a disk partition. The dynamic aspect adapts the interfaces of dynamic participants, wrapping Linux block device objects as OSKit block device objects, Linux filesystem objects as OSKit filesystem objects, and Linux file and directory objects as their OSKit equivalents. This illustrates how our approach complements the conventional approach: our units make it apparent which decisions are static (e.g., the decision to use Linux components with OSKit components) and which are dynamic (e.g., how many files will be opened, which files will be opened).

Decorator. If this is a multi-threaded system, we must take care to acquire and release locks when accessing the filesystem and device driver objects. The decorators Lock Filesys and Lock BlockDevice acquire locks when entering the decorated objects and release locks when leaving.

It would be a serious error to omit one of these lock decorators (leading to race conditions) or to insert it in the wrong place (leading to deadlock), so we use the constraint system to check that they are placed correctly. This may seem like overkill in such a simple configuration, but the reader will appreciate that this is just one of many rules that must be enforced and that we have omitted many units that would appear in a complete system. The complete system—including units for bootstrapping, console I/O, memory allocation, threads and locks, etc.—consists of over 100 unit instances.

Strategy. Disk drivers can optimize disk operations by coalescing reads and writes on adjacent blocks and can optimize disk seeks by reordering read and write requests. The series of actual requests issued to the disk is determined by a strategy unit. In Figure 5, we have selected the Simple Disk Strategy unit (which queues requests in the order they are issued) but we could have chosen a strategy that coalesces disk operations or reorders requests using an elevator algorithm. (The elevator strategy is not yet implemented.)

Command. The Simple Disk Strategy unit manipulates a list of outstanding requests, and these requests are parts of a *Command* pattern. The participants in this pattern are currently integrated within the implementation of the Simple Disk Strategy unit, but could be separated as shown in Figure 5 into a separate unit Encode BlockOp which provides a separate function for each kind of request (e.g., read or write). This unit would construct request objects and pass them to Simple Disk Strategy, which would process the requests.

Singleton. In this system, we made a design decision to have a single device and a single filesystem instance. One could imagine using a device driver implementation that supports just one instance of that device type or a filesystem implementation that supports just one instance of that filesystem type. But this is not what Linux components do. Most Linux device drivers and filesystems are written to support multiple instances of a device or filesystem. To overcome this mismatch, we use the BlkDev Instance and FS Instance units that each create and manage a single instance of the corresponding dynamic objects. These units are effectively adapters, making dynamic pattern participants appear as if they were static. This mismatch is common in reuse and maintenance scenarios: the cost of making changes influences the choice of design. Our approach to patterns addresses such real-world concerns.

3.4 Discussion

The previous sections demonstrate our approach to utilizing design patterns in the context of two example systems. In both examples we had a mix of static and dynamic participants: the static participants were realized by unit instances corresponding to "object instances" while the remaining dynamic participants were realized by units that create the pattern participant objects at run-time. In both examples we were able to lift a great deal of static knowledge to the unit level, but the exact amount depended on the patterns and their application to the particular design problems at hand.

In general, the static and dynamic parts of many patterns will vary from situation to situation. However, *in common use*, most pattern structures contain many participants and connections that are in fact static: these parts can be fruitfully lifted out of the participants' source implementations and then managed at the level of units. To test this claim, we analyzed the structures of all of the patterns described in Gamma et al.'s *Design Patterns* catalog [13]. For each, we considered common uses of the pattern in component-based systems software such as that built with the OSKit. We then applied our method to translate the pattern structures into appropriate units and unit parts.

Table 1 summarizes the results of our analysis. Each row of the table shows the translation of the participants within a single pattern. Overall, the table shows that most participants frequently correspond to static, design-time information and are therefore realizable within our unit model as design-time entities. (These are the columns under the "Design-Time/Static Participants" heading.) Abstract classes map naturally to unit interfaces. Participants that are singletons within the context of a pattern map naturally to unit instances that implement these participants. In some cases, a participant both defines an interface and represents a concrete instance, as indicated in the table. For example, in the *Facade* pattern, the *Facade* entity has both interface and implementation roles. In some cases, the designer may choose to implement a particular participant in more than one way: for instance, the designer may choose to implement a *Client* participant as a unit instance, or as a set of ports that allow the client to be connected at a later stage of the overall design. In other cases, the appropriate implementation of one participant depends on the characteristics of another: in the *Decorator* pattern, for example, the appropriate realizations of *Decorator* and *Concrete Decorator* differ according to the "singleton-ness" of the *Concrete Component*. Where the common translation or use varies, we have indicated this with italics, and we list the participant in multiple categories.

Because the OSKit is such a large body of code, largely derived from systems not explicitly organized around patterns, it is difficult to identify all uses of a particular pattern and so it is hard to determine the ratio of static uses to dynamic uses. With that caveat, we have found static instances of all three categories of patterns (creational, structural, and behavioral) in OSKit-based systems, but most examples are either creational or structural. Our admittedly incomplete study failed to find static examples of some patterns including *Flyweight*, *Chain of Responsibility*, and *Visitor*.

In summary, Table 1 shows that our approach to realizing patterns is applicable to many patterns: most have common applications in which many or all of the participants represent static system design knowledge that can be utilized by tools for design rule checking, code generation, and system optimization. This applies

	Design-Time/Static Participants			**Dynamic Participants**
Pattern	**Realized By Unit Interface (Method Steps 1, 3, 4)**	**Realized By Unit(s) Impl'ing the Instance(s) (Method Steps 2, 5)**	**Realized By Port(s) On Unit Instances (Method Step 5)**	**Realized By Unit Defining the Class (Method Steps 2, 5)**
Abstract Factory	Abstract Factory Abstract Product	Concrete Factory *Client*	*Client*	Concrete Product
Builder	Builder	Concrete Builder Director		Product
Factory Method	Product Creator	Concrete Creator		Concrete Product
Prototype	Prototype	*Client*	*Client*	Concrete Prototype
Singleton		Singleton		
Adapter (class)	Target	*Client*	*Client*	Adaptee Adapter
Adapter (object)	Target	*Client* Adaptee Adapter	*Client*	
Bridge	Abstraction (intfc.) Refined Abstraction (intfc.) Implementor	Abstraction (impl.) Refined Abstraction (impl.) Concrete Implementor		
Composite	Component	*Client*	*Client*	Leaf Composite
Decorator	Component	*Concrete Component* *Concrete Decorator*	*Decorator*	*Concrete Component* *Decorator* *Concrete Decorator*
Facade	Facade (intfc.)	Facade (impl.) subsystem classes		
Flyweight	Flyweight	Flyweight Factory *Client*	*Client*	Concrete Flyweight Unshared Conc. Flyweight
Proxy	Subject	Proxy Real Subject		
Chain of Resp.	Handler	Concrete Handler *Client*	*Client*	
Command	Command	*Client* Invoker Receiver	*Client*	Concrete Command
Interpreter	Abstract Expression	Context *Client*	*Client*	Terminal Expression Nonterminal Expression
Iterator	Iterator Aggregate	*Concrete Aggregate*		Concrete Iterator *Concrete Aggregate*
Mediator	Mediator	Concrete Mediator colleague classes		
Memento		Originator *Caretaker*	*Caretaker*	Memento
Observer	Subject (intfc.) Observer	Subject (impl.) Concrete Subject *Concrete Observer*	*Concrete Observer*	
State	Context (intfc.) State	Context (impl.) Concrete State		
Strategy	Strategy	Concrete Strategy Context		
Template Method		Abstract Class Concrete Class		
Visitor	Visitor Element	*Concrete Visitor* *Object Structure*		*Concrete Visitor* Concrete Element *Object Structure*

Table 1: Summary of how the participants within the *Design Patterns* catalog [13] can be realized within the unit model, for common situations in the design of component-based, C language systems software. Participants are classified according to their common and primary realizations; certain uses of patterns will dictate different realizations. Where common use varies, participants are italicized and are listed in all applicable categories. Some participants have both interface and implementation roles as shown. Participants that map to unit instances usually also require interface definitions to describe their ports.

even when a participant is dynamic and is realized by a unit that produces objects at run-time. In these cases, we can use the unit model to define our run-time classes/types in terms of mixins, thus increasing the potential reuse of our unit definitions and implementations.

4. ANALYSIS

The key feature of our approach is that we express static pattern relationships in a component *configuration* language instead of expressing those relationships in the component *implementation* language. In this section, we detail the benefits and costs of this separation of concerns.

4.1 Benefits of Our Approach

Our technique for realizing patterns has three main consequences. First, because static pattern information is located in single place (our compound units) and because component interconnections are fully resolved at build-time, it is possible for tools to perform a more thorough analysis of the software architecture than in the conventional approach to realizing patterns. Second, because the unit language has a single purpose—to express components, their instantiations, and their interconnections—it is possible to provide features in the language that make this task easier. Third, because the task of pattern composition is moved out of the implementations of the participants, those implementations can be simpler and are less likely to be hard-wired to function only in fixed pattern roles. These three consequences lead to benefits in the areas of error detection, performance, and ease of understanding and reuse, which we explore in the following sections.

4.1.1 Checking Architectural Constraints

In the conventional approach to realizing design patterns, it can be difficult to enforce static system design constraints: the rules are encoded “implicitly” in the implementation, making them difficult for people to find and for tools to enforce in the face of future system evolution. Our approach to realizing patterns has the following advantages over the conventional method.

The constraint checker detects global, high-level errors. The constraint checker within the Knit unit compiler can detect “global” errors that involve many parts of a system, whereas a conventional language type system is restricted to detecting relatively local errors. Such global constraints often deal with high-level system composition issues—e.g., ensuring that domain-specific properties hold across many interconnected components—whereas conventional type systems and tools are restricted to detecting relatively low-level and general types of errors such as uncaught exceptions [1], dereferenced null pointers [7], and race conditions [9].

Constraints express domain-specific design rules. A software architect is often interested in detecting domain-specific errors. For example, recent versions of RTLinux [24] permit normal (user-level) application code to be called from a hard real-time kernel. Without going into detail, an essential requirement of such applications is that they never invoke a system call while running in real-time mode. We have used Knit’s constraint system to check this constraint for RTLinux applications: i.e., to verify, at compile-time, that there are no paths from an application’s real-time signal handler into the Linux kernel.

Design errors are separated from implementation errors. In particular, this reduces the level of expertise required in order to use (or reuse) a component correctly, inside or outside of a pattern.

The constraint checker need not deal with the base implementation language. Our constraint checker deals only with the unit specification language, not with the source code of the components. Because the unit language is simple, the constraint checker is simple and precise. Further, it would be easy to extend with more powerful and perhaps more pattern-specific reasoning methods in the future. In contrast, to detect design errors in a conventionally realized design pattern, a tool would need to deal with all the complexities of the base implementation language: loops, recursion, exceptions, typecasts, virtual functions, pointers, and so on. Such a tool is therefore difficult to create—greatly raising the barrier to developing domain-specific analyses—and is often imprecise.

Many architecture description languages can provide the advantages described above: like our tools, they achieve this by separating the description of the architecture from the implementation of the components, and by being domain-specific instead of general-purpose. Bringing these features to bear on the realization of design patterns is one of the strengths of our tools and approach.

4.1.2 Performance Optimization

Another strength of our approach is that static pattern knowledge is readily available for system optimization. The conventional approach to realizing patterns uses language features that introduce indirections to achieve greater flexibility. These indirections—principally indirect function calls—impose a performance penalty that can often be avoided in our approach.

Static implementation enables many optimizations. When component instances are connected statically, indirect function calls are often turned into direct calls. This affords the compiler the opportunity to inline function calls, thus eliminating overhead and exposing additional and often more significant opportunities for optimization, especially those that specialize a function for a particular context. In addition, highly optimizing compilers, or compilers aided by a source transformation that Knit can perform, are able to inline functions across module boundaries. In previous work [21], we used Knit to implement a network router made of very small components. (Each packet traversed 10–20 components while being forwarded.) Applying cross-component inlining eliminated the cost of many function calls but, more significantly, enabled the C compiler to apply all of its intra-procedural optimizations. The overall effect of this optimization was to reduce the execution time of the routing components by 35%.

Static implementation makes performance less sensitive to code changes. To eliminate virtual function calls, a compiler requires a global (or near global) analysis of the program being optimized. These analyses are necessarily affected by subtle features of how the program is expressed: a consequence is that any change to that program could potentially change the analysis result and therefore change whether or not the optimization can be applied. In a performance-sensitive situation (e.g., in real-time code), loss of an optimization may affect program correctness. By making static knowledge explicit, our approach to patterns helps to reduce the complexity of the resulting system, thus promoting compile-time analysis and making “global” performance less sensitive to local code changes.

4.1.3 Ease of Understanding and Code Reuse

In the conventional approach to realizing design patterns, one takes into account the role of each participant when implementing the participant—or, if the pattern is applied after implementation, one modifies the participant to reflect their roles in the pattern. In our approach, because units do not contain direct references to other participants, units often need no modification in order to be used in a particular role in a pattern. Avoiding even small changes to the participants leads to significant benefits.

The approach is usable when code cannot be changed. The implementation of a participant may be unchangeable if the code has multiple users with different needs, if the source code is not available, or if the code is being actively maintained by a separate organization. For instance, the developers of the OSKit cannot practically afford to change the Linux components that they incorporate: they must deal with the code as it is written.

A participant can be used in multiple patterns. Separating a participant's role from its implementation is beneficial when the implementation can be "reused" to serve in many different roles, perhaps concurrently in several different patterns. The unit model allows a programmer to separate a participant's primary implementation from any code needed to adapt that implementation to a particular pattern role: by creating a unit composition, a programmer can "weave" code at the imports and exports of a participant unit instance.

Code is not obfuscated with indirections. The conventional realization of a design pattern often achieves flexibility by introducing additional levels of indirection that are apparent in the implementations of the participants. This indirection can obscure the primary purpose of the code. For example, before applying the unit model to the OSKit, we relied on objects, factories, and registries to enable reconfiguration. Over time, much OSKit code came to look like the following:

```
clientos = registry->lookup(registry, clios_iid);
fsn      = clientos->create_fsnamespace(filesys);
file     = fsn->lookup_path("/usr/bin/latex");
```

The code was often further complicated to deal with run-time errors. In any particular system, however, the values of `clientos` and `fsn` were fixed in the system configuration, and knowable at compile-time. After applying our approach, such code could often be simplified to just:

```
file = lookup_path("/usr/bin/latex");
```

making it clear that the selection of `lookup_path`'s implementation is a static, not dynamic, system property.

4.2 Costs of Our Approach

Our approach to realizing design patterns is not appropriate for all situations and design problems. The following paragraphs summarize the costs and potential problems of our approach.

Our approach only specifies the static parts of patterns. The main goal of our approach is to use an external component language to specify the static aspects of system architecture. It is inappropriate (and often infeasible) to use our approach to specify fundamentally dynamic elements of software architecture.

Our approach commits code to being static or dynamic. One can imagine that having carefully used our approach (with its emphasis on static participants and relationships) to realize a pattern, a change of requirements might turn a relationship from static to dynamic, requiring that the pattern be re-implemented using the conventional object-oriented approach (with its emphasis on dynamic participants and relationships). This is a problem: while it is easy to use a dynamic system in a static situation, it is not so easy to use a static system in a dynamic way. Therefore, when using our approach, one should design systems in such a way that expected changes in the system requirements are unlikely to require changing the static and dynamic natures of pattern participants—but we recognize that this is not always possible. An automated implementation of our approach (perhaps based on partial evaluation [6]) could partly solve this problem by transforming dynamic code into static code, although this would not promote the benefit of easier writing and understanding of code through eliminated indirections.

Our approach requires support for the unit component model. To fully benefit from our approach, one needs language support in the form of an advanced module or component system and, ideally, a constraint checking system. This implies several costs: one must switch to using new tools, learn the component definition and linking language, learn to use the constraint checking language, and convert existing codebases to use the component language. This can be a significant undertaking. As described in Section 2, however, it is possible to use existing tools and techniques to achieve some (but not all) of the benefits of the unit component model.

Our approach can obscure the differences between patterns. When one looks at the unit diagrams of participants and relationships, it is clear that sometimes, different patterns look the same when realized in our approach. However, this observation is also true of the conventional approach to patterns: many patterns are realized in similar ways but differ significantly in their purpose.

5. RELATED WORK

Gamma et al.'s *Design Patterns* book [13] triggered a flurry of papers on implementing patterns in object-oriented languages. Here, we consider representatives of particular styles of implementation. Bosch [3] describes a language LayOM for constructing C++ classes by adding a number of layers to a simple class. By using layers corresponding to particular patterns, Bosch solves the *traceability* problem—that it is hard to find and identify patterns in one's code—and enables pattern implementations to be reused. However, because the layers form part of the class description, the role of each pattern participant is hardwired and the participants cannot be used in other patterns without being modified. Bosch makes no mention of static analysis, detecting design errors, or optimization. Marcos et al. [16] describe an approach that is closer to ours: the code that implements participants is clearly separated from the code that defines their roles in patterns. The difference is that their approach is based on run-time reflection within a metaprogramming system (CLOS), and so they do not support static analysis or optimization. Tatsubori and Chiba [23] describe a similar approach to that of Marcos et al., except that it uses OpenJava's compile-time metaprogramming features. Like Marcos et al., they separate roles from participants and, because they use compile-time metaprogramming, it should be possible to perform static analysis. However, OpenJava does not provide anything like Knit's unit constraint system.

Krishnamurthi et al. [15] describe an approach to pattern implementation based on McMicMac, an advanced macro system for Scheme. Their approach is like that of Tatsubori and Chiba: patterns are expanded statically (enabling optimization) and the application of patterns is not separated from the definitions of the participants. Unlike OpenJava, McMicMac provides source-correlation and expansion-tracking facilities that allow errors to be reported in terms of the code that users wrote instead of its expansion, but there is no overall framework for detecting global design errors.

Baumgartner et al. [2] discuss the influence of language features on the implementation of design patterns. Like us, they note that Gamma et al.'s pattern descriptions [13] would be very different in a language that directly supports abstract interfaces and a module mechanism separate from class hierarchies. Baumgartner also lists a number of other useful features including mixins and multimethods. MultiJava [5] adds some of these features to Java, enabling them to cleanly support the *Visitor* pattern and to describe "open classes." Our colleagues' paper on Jiazzi [17] shows how the open class pattern can be realized with units. Bruce et al. [4] describe virtual types and show how they apply to the *Observer* pattern. All of these papers describe language features that address problems in implementing patterns in object-oriented languages, but their focus is on the technology, not the approach enabled by that technology.

At the other end of the spectrum, there are component programming models, module interconnection languages (MILS) [20], and architecture description languages (ADLS) [18]. Our implementations of the unit model lie at the intersection of these three approaches. Units are like COM or CORBA components except that units play a more static role in software design; units are like MILs in that each implementation of the unit model supports just one kind of unit interconnection; and units are like ADLs in that units support static reasoning about system design.

Module interconnection languages are perhaps closest in purpose to the unit model. The best example we know of using a MIL in the way this paper suggests is FoxNet [14], a network stack that exploits ML's powerful module language. However, although FoxNet clearly uses a number of patterns, there is no explicit statement of this fact and consequently no discussion of implementing a broad range of patterns using a MIL.

Architecture description languages provide a similar but higher-level view of the system architecture to MILs. This higher-level view is the key difference. ADLs describe software designs in terms of architectural features, which may include patterns. ADLs may also provide implementations of these features: the details of implementation need not concern the user. In contrast, this paper is all about those implementation issues: we describe a method that ADL implementors could apply when adding new patterns to the set provided by their ADL. That said, ADLs provide more expressive languages for describing design rules, specifying components, and reasoning about system design than is currently supported by the unit model. We plan to incorporate more high-level ADL features into our unit languages in the future.

6. CONCLUSION

Design patterns can be realized in many ways: although they are often described in object-oriented terms, a pattern need not always be realized in an OOP language nor always with objects and interconnections created at run-time. In this paper we have presented a complementary realization of design patterns, in which patterns are statically specified in terms of the unit model of components. While this approach is not applicable to all software architectures, it can yield benefits when applied to static systems, and to static aspects of dynamic systems. These benefits include verification of architectural constraints on component compositions, and increased opportunities for optimization between components.

7. ACKNOWLEDGMENTS

We thank Mike Hibler for his expert help with the OSKit, and Sean McDirmid for many fruitful discussions about the unit model and design patterns. Matthew Flatt, Jason Baker, David Coppit, and Joe Loyall provided valuable comments on drafts of this paper. We also thank the anonymous reviewers for their many helpful suggestions.

8. REFERENCES

[1] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, second edition, 1998.

[2] G. Baumgartner, K. Läufer, and V. F. Russo. On the interaction of object-oriented design patterns and programming languages. Technical Report CSD–TR–96–020, Department of Computer Sciences, Purdue University, 1996.

[3] J. Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 11(2):18–32, 1998.

[4] K. B. Bruce and J. C. Vanderwaart. Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. In *Electronic Notes in Theoretical Computer Science*, volume 20. Elsevier Science Publishers, 2000.

[5] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proc. of the 2000 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, pages 130–145, Minneapolis, MN, Oct. 2000.

[6] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, E.-N. Volanschi, J. Lawall, and J. Noyé. Partial evaluation for software engineering. *ACM Computing Surveys*, 30(3es), Sept. 1998.

[7] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation*, pages 1–16, San Diego, CA, Oct. 2000.

[8] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proc. of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, pages 94–104, Baltimore, MD, Sept. 1998.

[9] C. Flanagan and S. N. Fruend. Type-based race detection for Java. In *Proc. of the ACM SIGPLAN '00 Conf. on Programming Language Design and Implementation (PLDI)*, Vancouver, Canada, June 2000.

[10] M. Flatt. *Programming Languages for Component Software*. PhD thesis, Rice University, June 1999.

[11] M. Flatt and M. Felleisen. Units: Cool units for HOT languages. In *Proc. of the ACM SIGPLAN '98 Conf. on Programming Language Design and Implementation (PLDI)*, pages 236–248, Montreal, Canada, June 1998.

[12] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for OS and language research. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, pages 38–51, St. Malo, France, Oct. 1997.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[14] B. Harper, E. Cooper, and P. Lee. The Fox project: Advanced development of systems software. Computer Science Department Technical Report 91–187, Carnegie Mellon University, 1991.

[15] S. Krishnamurthi, Y.-D. Erlich, and M. Felleisen. Expressing structural properties as language constructs. In *Programming Languages and Systems (Proc. of the Eighth European Symp. on Programming, ESOP '99)*, volume 1576 of *Lecture Notes in Computer Science*, pages 258–272. Springer-Verlag, Mar. 1999.

[16] C. Marcos, M. Campo, and A. Pirotte. Reifying design patterns as metalevel constructs. *Electronic Journal of SADIO*, 2(1):17–29, 1999.

[17] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proc. of the 2001 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*, pages 211–222, Tampa, FL, Oct. 2001.

[18] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, Jan. 2000.

[19] J. G. Mitchell, W. Mayberry, and R. Sweet. *Mesa Language Manual*, 1979.

[20] R. Prieto-Diaz and J. M. Neighbors. Module interconnection languages. *Journal of Systems and Software*, 6(4), Nov. 1986.

[21] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation*, pages 347–360, San Diego, CA, Oct. 2000.

[22] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1999.

[23] M. Tatsubori and S. Chiba. Programming support of design patterns with compile-time reflection. In *Proc. of the OOPSLA '98 Workshop on Reflective Programming in C++ and Java*, pages 56–60, Vancouver, Canada, Oct. 1998.

[24] V. Yodaiken. The RTLinux manifesto. In *Proc. of the Fifth Linux Expo*, Raleigh, NC, Mar. 1999.