Software Design Patterns for TinyOS

DAVID GAY
Intel Research, Berkeley
PHILIP LEVIS
Stanford University
and
DAVID CULLER
University of California, Berkeley

We present design patterns used by software components in the TinyOS sensor network operating system. They differ significantly from traditional software design patterns because of the constraints of sensor networks and to TinyOS's focus on static allocation and whole-program composition. We describe how nesC has evolved to support these design patterns by including a few simple language primitives and optimizations.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*

General Terms: Languages, Design

Additional Key Words and Phrases: Design Patterns, embedded systems, nesC, TinyOS

ACM Reference Format:

Gay, D., Levis, P., and Culler, D. 2007. Software design patterns for TinyOS. ACM Trans. Embedd. Comput. Syst. 6, 4, Article 22 (September 2007), 39 pages. DOI = 10.1145/1274858.1274860 http://doi.acm.org/ 10.1145/1274858.1274860

1. INTRODUCTION

TinyOS [Hill et al. 2000] is an OS for wireless network embedded systems, with an emphasis on reacting to external events and extremely low-power operation. Rather than a monolithic OS, TinyOS is a set of components that are included as-needed in applications. A significant challenge in TinyOS development is designing and implementing flexible, reusable components. Programming abstractions for sensor networks, where TinyOS is the current OS-of-choice, are an area of active research and investigation [Levis et al. 2004]

Authors' addresses: David Gay, Intel Research, Berkeley, California 94720; Philip Levis, Stanford University, Palo Alto, California 94305; David Culler, University of California, Berkeley, California 94720.

Writing solid, reusable software components is hard. Doing so for sensor networks is even harder. Limited resources (e.g., 4 KB of RAM) and strict energy budgets (e.g., averages below 1 mW) lead developers to write application-specific versions of many services. While specialized software solutions enable developers to build efficient systems, they are inherently at odds with reusable software.

Software design patterns are a well-accepted technique to promote code reuse [Gamma et al. 1995, p.1]:

> *These patterns solve specific design problems and make object-oriented designs more flexible, elegant, and ultimately reusable.*

Design patterns identify sets of common and recurring requirements and define a pattern of object interactions that meet these requirements. However, these patterns are not directly applicable to TinyOS programming. Most design patterns focus on the problems faced by large, object-oriented programs; in sensor networks the challenges are quite different. These challenges include [Levis et al. 2004, Section 2.1]:

- Robustness: once deployed, a sensor network must run unattended for months or years.
- Low resource usage: sensor network nodes, colloquially known as *motes*, include very little RAM, and run off batteries.
- Diverse service implementations: applications should be able to choose between multiple implementations of, e.g., multihop routing.
- Hardware evolution: mote hardware is in constant evolution; applications and most system services must be portable across hardware generations.
- Adaptability to application requirements: applications have very different requirements in terms of lifetime, communication, sensing, etc.

nesC [Gay et al. 2003]—TinyOS's implementation language—was designed with these challenges in mind; it is a component-based language with an event-based execution model. nesC components have similarities to objects: they encapsulate state and interact through well-defined interfaces. They also have significant differences: there is no inheritance, no dynamic dispatch, and no dynamic object allocation—the set of components and their interactions are fixed at compile-time rather than at runtime. This promotes reliability and efficiency, but programmers cannot easily apply idioms or patterns from object-oriented languages, and, when they do, the results are rarely effective.

In this paper, we present a preliminary set of eight design patterns, which show how nesC can be used to build components that address TinyOS's challenges. These patterns are based on our experiences designing and writing TinyOS components and applications, and on our examination of code written by others. These patterns have driven, and continue to drive, the development of nesC. For instance, the `uniqueCount` function was introduced in nesC version 1.1 to support the ServiceInstance pattern; nesC version 1.2 (recently released) includes generic components, which simplify expression of some of the patterns presented here (see Section 4).

This paper contributes to embedded system programming in three ways. First, these design patterns provide insight on how programming network embedded systems is structurally different than traditional software and how these different factors motivate software design. We believe that these patterns have applicability beyond the sensor network space: TinyOS's requirements are not radically different from those of many other embedded systems. Second, we explore how a few simple features of the nesC language and compiler, particularly parameterized interfaces, unique identifiers, and inlining, are necessary for concise and efficient expression of these patterns. Finally, this paper helps researchers working with TinyOS write effective programs. The youth of TinyOS precludes us from having a corpus of tens of millions of lines of code and decades of experience, as traditional design pattern researchers do: these patterns are an initial attempt to analyze and distill TinyOS programming.

Although prior work has explored object-oriented design patterns for embedded and real-time devices [PatternsW1 2001; PatternsW2 2002; PatternsW3 2002; Douglass 2002; Girod et al. 2004], they deal with platforms that have orders of magnitude more resources (e.g., a few mega bytes of RAM), and, correspondingly, more traditional programming models, including threads, instantiation, and dynamic allocation.

An alternative approach to building reusable services for sensor networks is offered by SNACK [Greenstein et al. 2004], which is composed of a library of configurable components; a SNACK program is a declarative specification of the components a program needs and their connections. SNACK relies on a compiler to figure out which services should be instantiated (compatible components are shared, e.g., two requests for a timer at the same rate), with what parameters and exactly how they should be connected. Effectively, SNACK aims to make it easy to build an application from an existing set of reusable services; our design patterns show ways of building services so that they are more reusable.

Section 2 provides background on the nesC language. Section 3 presents eight TinyOS design patterns, describing their motivation, consequences, and representation in nesC, as well as listing several TinyOS components that use them.[1] Section 4 discusses the patterns in the light of nesC and TinyOS development, and Section 5 concludes.

2. BACKGROUND

Using a running example of an application component that samples two sensors, we describe the aspects of nesC relevant to the patterns we present in Section 3.

nesC [Gay et al. 2003] is a C-based language with two distinguishing features: a programming model where components interact via interfaces, and an event-based concurrency model with run-to-completion tasks and interrupt handlers. The run-to-completion model precludes blocking calls. Lengthy operations and system services, such as sampling a sensor or sending a packet, are split-phase operations, where a command to start the operation returns immediately and a callback event indicates when the operation completes (Figure 2,

[1]These components can be found in the TinyOS distributions, available from `http://www.tinyos.net`.

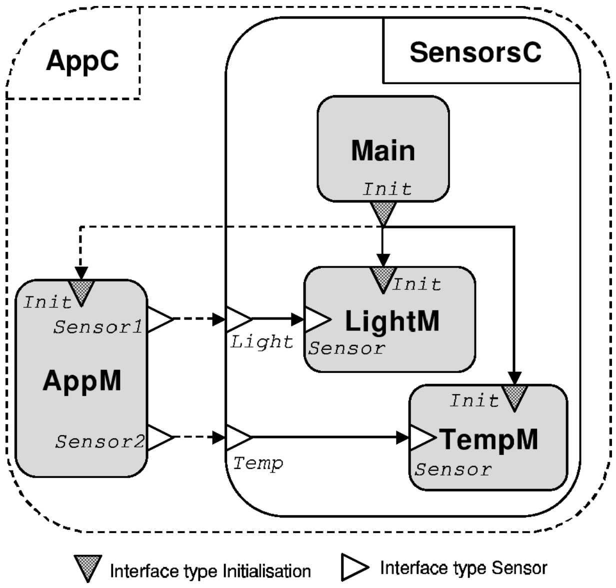


Fig. 1. Sample component assembly. Solid rectangles are modules; open rectangles are configurations. Triangles pointing into a rectangle are provided interfaces; triangles pointing out are used interfaces. Dotted lines are "wires" added by configuration `AppC`; full lines are "wires" added by configuration `SensorsC`. Component names are in bold.

see later). To promote reliability and analyzability, nesC does not support dynamic memory allocation or function pointers; all component interactions are specified and known at compile-time.

2.1 Components and Interfaces

nesC programs are assemblies of components, connected ("wired") via named interfaces that they *provide* or *use*. Figure 1 graphically depicts the assembly of six components connected via interfaces of type `Sense` and `Initialize`. Modules are components implemented with C code, while configurations are components implemented by wiring other components together. In the example figure, `Main` (a "system boot" component), `LightM`, `TempM`, and `AppM` are modules, while `AppC` and `SensorsC` are configurations. The example shows that configuration `AppC` "wires" (i.e., connects) `AppM`'s `Sensor1` interface to `SensorsC`'s `Light` interface, etc.

Modules and configurations have a name, specification, and implementation:

```
module AppM {
  provides interface Initialize as Init;
  uses interface Sense as Sensor1;
  uses interface Sense as Sensor2;
}
implementation { ... }
```

declares that `AppM` (from Figure 1) is a module that provides an interface named `Init` and uses two interfaces, named `Sensor1` and `Sensor2`. Each interface has

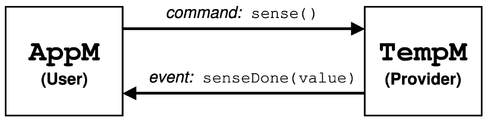


Fig. 2. Typical split-phase operation.

a type, in this case either `Initialize` or `Sense`. A component name denotes a unique, singleton component[2]: references to `Main` in different configurations (see below) all refer to the same component.

An interface type specifies the interaction between a provider component and a user component as a set of named functions:

```
interface Initialize { // component initialization
  command void init();
}

interface Sense { // split-phase sensor read
  command void sense();
  event void senseDone(int value);
}
```

This interaction is bidirectional: *commands* are invocations from the user to the provider, while *events* are from the provider to the user. Interface type `Sense` represents a typical split-phase operation: providers must implement the `sense` command, which represents a request to read a sensor; users must implement the `senseDone` event, which the provider signals when the sensor read completes. To make the two directions syntactically explicit, nesC events are *signaled* while commands are *called*. In both cases, the actual interaction is a function call. Figure 2 shows this relationship for `AppM` and `TempM`.

As a module, `AppM` must provide C implementations of commands in its provided interfaces and of events in its used interfaces. It can call or signal any of its commands or events, and `post` tasks for later execution:

```
module AppM { ... }
implementation {
  int sum = 0;
  task void startSensing() {
    call Sensor1.sense();
  }
  command void Init.init() {
    post startSensing();
  }
  event void Sensor1.senseDone(int val) {
    sum += val;
    call Sensor2.sense();
  }
```

[2]We discuss in Section 4 how version 1.2 of nesC changes this and its effect on design patterns.

```
  event void Sensor2.senseDone(int val) {
    sum += val;
  }
}
```

As this example shows, a command or event f of an interface I is named $I.f$ and is similar to a C function except for the extra syntactic elements such as `command`, `event` and `call`. Modules encapsulate their state: all of their variables (e.g., `sum`) are private. The `Init.init` command posts a task to defer the call to `Sensor1.sense` as the sensor may not yet be initialized (see the discussion of multiple wiring below).

2.2 Configurations

A configuration implements its specification by wiring other components together and *equating* its own interfaces with interfaces of those components. Two components can interact only if some configuration has wired them together:

```
configuration SensorsC {
  provides interface Sense as Light;
  provides interface Sense as Temp;
}
implementation {
  components Main, LightM, TempM;

  Main.Init -> LightM.Init;
  Main.Init -> TempM.Init;

  Light = LightM.Sensor;
  Temp = TempM.Sensor;
}
```

`SensorsC` "assembles" components `LightM` and `TempM` into a single component providing an interface for each sensor: `Temp` is equated to `TempM`'s `Sensor` interface, and `Light` with `LightM`'s `Sensor` interface. In addition, `SensorsC` *wires* the system's initialization interface (`Main.Init`) to the initialization interfaces of `LightM` and `TempM`.

Finally, `AppC`, the configuration for the whole application, wires module `AppM` (which uses two sensors) to `SensorsC` (which provides two sensors), and ensures that `AppM` is initialized by wiring it to `Main.Init`:

```
configuration AppC { }
implementation {
  components Main, AppM, SensorsC;

  Main.Init -> AppM.Init;
  AppM.Sensor1 -> SensorsC.Light;
  AppM.Sensor2 -> SensorsC.Temp;
}
```

In this application, interface `Main.Init` is *multiply wired*. `AppC` connects it to `AppM.Init`, while `SensorsC` connects it to `LightM.Init` and `TempM.Init`. The call `Init.init()` in module `Main` compiles to an invocation of all three `init()` commands,[3] in some unspecified order. Thus, it is possible that `AppM.init` will be called before `LightM.init`. Hence, the need for the deferred execution of the call to `Sensor1.sense` in `AppM`.

2.3 Parameterized Interfaces

A parameterized interface is an interface array. For example, this module has a separate instance of interface `A` for each value of `id`:

```
module Example {
  provides interface Initialize as Inits[int id];
  uses interface Sense as Sensors[int id];
} ...
```

In a module, commands and events of parameterized interfaces have an extra argument:

```
  command void Inits.init[int id1]() {
    call Sensors.sense[id1]();
  }
  event void Sensors.senseDone[int i](int v) {
  }
```

A configuration can wire a single interface by specifying its index:

```
configuration ExampleC {
}
implementation {
  components Main, Example;
  components TempM, LightM;

  Main.Init -> Example.Inits[42];
  Example.Sensors[42] -> TempM.Sensor;
  Example.Sensors[43] -> LightM.Sensor;
}
```

When `Main`'s `Init.init` command is called, `Example`'s `Inits.init` command will be executed with `id = 42`. This will cause `Example` to call `Sensor[42].sense`, which connects to `TempM.sense`.

A configuration can wire or equate a parameterized interface to another parameterized interface. This equates `Example.Sensors[`i`]` to `ADC[`i`]` for all values of i:

```
  provides interface Sense as ADC[int id];
  ...
  Example.Sensors = ADC;
```

[3]If a multiply wired function has nonvoid result, nesC combines the results via a programmer-specified function. [Gay et al. 2003].

2.4 unique and uniqueCount

In many cases, a programmer wants to use a single element of a parameterized interface and does not care which one, as long as no one else uses it. This functionality is supported by nesC's `unique` construction:

```
AppM.Timer1 -> TimerC.Timer[unique("Timer")];
AppM.Timer2 -> TimerC.Timer[unique("Timer")];
```

All uses of `unique` with the same argument string (a constant) return different values, from a contiguous sequence starting at 0. It is also often useful to know the number of different values returned by `unique` (e.g., a service may wish to know how many clients it has). This number is returned by the `uniqueCount` construction:

```
timer_t timers[uniqueCount("Timer")];
```

2.5 Static Programming

The nesC language supports embedded programming by providing a component model, compile-time composition, a concurrency model, and two simple functions, which it resolves at compile time. Individually, each one changes programming methodologies, compiler techniques, and software structure in a small way. Together, however, these features lead to programs that differ greatly from their counterparts written in C or C-like object-oriented languages.

The principal difference is that nesC programs try to push as many decisions to compile-time as possible; we refer to this as a *static programming model*. This model has two main features: static call paths and static allocation. A nesC component preserves flexibility by referring to external functions via interfaces, but configurations bind callers and callees at compile-time, so complete call paths are defined statically. For example, the configuration

```
AppM.Sensor1 -> SensorsC.Light;
```

creates two bindings: the command `AppM.Sensor1.sense` to `SensorsC.Light. sense` and the event `SensorsC.Light.sendDone` to `AppM.Sensor1.senseDone`.

C and C-like object-oriented languages that have a global namespace achieve similar binding flexibility through runtime mechanisms. For example, if a C program does not want to explicitly name its callee, it uses function pointers (e.g., vnodes in the virtual file system [Kleiman 1986]). In C++, Java, and C# the standard technique is to use an interface or class. Static call paths allows the nesC compiler to optimize heavily across call boundaries (see Section 4.3).

The ability to count at compile-time leads to very different allocation strategies by allowing variably-sized allocation at compile-time: entities are "allocated" with `unique`, and space is reserved with `uniqueCount` (Section 2.4). For example, a program that writes four different logging files can, at compile-time, definitively state that it needs to allocate the state for four file handles. Programmers do not have to guess the system's resource usage, potentially wasting space or risking runtime failure. This determinism and efficiency comes by sacrificing flexibility: a nesC program cannot, e.g., readily implement a file server that can handle an unbounded number of clients and open files. However, such requirements are not typical for sensor networks (and similar embedded programming tasks): programs are designed for specific applications, with known worst-case resource requirements. The increased reliability and simpler programming of nesC's static approach almost always outweigh the costs.

2.6 Summary

The basic unit of TinyOS/nesC programming is a component, which provides and uses interfaces. An application is a set of components whose interfaces have been wired together. Because all TinyOS services are components that follow a static programming paradigm, nesC can optimize call paths across components, eliminate dead code and variables, and provide support for efficient state allocation. This approach provides a great deal of structure to embedded programming and can support good software techniques. However, the resulting programs differ from those of other systems languages. These languages have a large number of accumulated idioms and software design patterns. What their analogs in the nesC programming model look like—flexible, reusable, and efficient components—is an interesting question. The next section provides the beginnings of an answer.

3. DESIGN PATTERNS

We present eight TinyOS design patterns: three behavioral (relating to component interaction): Dispatcher, Decorator, and Adapter, three structural (relating to how applications are structured): Service Instance, Placeholder, and Facade, and two namespace (management of identifiers such as message types): Keyset and Keymap. These patterns are also presented on our website [Levis and Gay 2004], which we will update as new patterns are discovered and documented. We follow the basic format used in *Design Patterns* [Gamma et al. 1995], abbreviated to fit in a research paper. Each pattern has an *Intent*, which briefly describes its purpose. A more in-depth *Motivation* follows, providing an example drawn from TinyOS. *Applicable When* provides a succinct list of conditions for use and a component diagram shows the *Structure* of how components in the pattern interact.[4] This diagram follows the same format as Figure 1, with the addition of a folded subbox for showing source code (a floating folded box represents source code in some other, unnamed, component), and is followed by a *Participants* lists, explaining the role of each component. *Sample Code* shows an example nesC implementation and *Known Uses* points to some uses of the pattern in TinyOS. *Consequences* describes how the pattern achieves its goals and notes issues to consider when using it. Finally, *Related Patterns* compares to other relevant patterns.

[4]This diagram is omitted for the Keyset pattern, as it is not concerned with component interactions.

3.1 Behavioral: Dispatcher

3.1.1 *Intent.* Dynamically select between a set of operations based on an identifier. Provides a way to easily extend or modify a system by adding or changing operations.

3.1.2 *Motivation.* At a high level, sensor network applications execute operations in response to environmental input, such as sensor readings or network packets. The operation's details are not important to the component that presents the input. We need to be able to easily extend and modify what inputs an application cares about, as well as the operation associated with an input.

For example, a node can receive many kinds of active messages (packets). Active messages (AM) have an 8-bit type field to distinguish between protocols. A flooding protocol uses one AM type, while an ad-hoc routing protocol uses another. `AMStandard`, the component that signals the arrival of a packet, should not need to know what processing a protocol performs or whether an application supports a protocol. `AMStandard` just delivers packets, and higher level communication services respond to those they care about.

The traditional approach to this problem is to use function pointers or objects, which are dynamically registered as callbacks. In many cases, even though registered at run time, the set of operations is known at compile time. Thus these callbacks can be replaced by a dispatch table compiled into the executable, with two benefits. First, this allows better cross-function analysis and optimization and, second, it conserves RAM, as no pointers or callback structures need to be stored.

Such a dispatch table could be built for the active message example by using a `switch` statement in `AMStandard`. However, this is very inflexible: any change to the protocols used in an application requires a change in a system component.

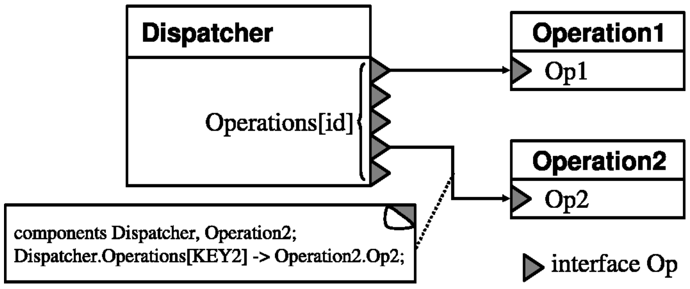
A better approach in TinyOS is to use the Dispatcher pattern. A Dispatcher invokes operations using a parameterized interface, based on a data identifier. In the case of `AMStandard`, the interface is `ReceiveMsg` and the identifier is the active message type field. `AMStandard` is independent of what messages the application handles, or what processing those handlers perform. Adding a new handler requires a single wiring to `AMStandard`. If an application does not wire a receive handler for a certain type, `AMStandard` defaults to a null operation.

Another example of a Dispatcher is the scheduler of the Maté virtual machine. Each instruction is a separate component that provides the `MateBytecode` interface. The scheduler executes a particular bytecode by dispatching to the instruction component using a parameterized `MateBytecode` interface. The instruction set can be easily changed by altering the wiring of the scheduler.

3.1.3 *Applicable When*

- A component needs to support an externally customizable set of operations.
- A primitive integer type can identify which operation to perform.
- The operations can all be implemented in terms of a single interface.

3.1.4 *Structure.*


Dispatcher
Operations[id]
Operation1
Op1
Operation2
Op2
components Dispatcher, Operation2;
Dispatcher.Operations[KEY2] -> Operation2.Op2;
interface Op

3.1.5 *Participants*

- **Dispatcher**: invokes its parameterized interface based on an integer type.
- **Operation**: implements the desired functionality and wires it to the dispatcher.

3.1.6 *Sample Code.*
`AMStandard` is the radio stack component that dispatches received messages:

```
module AMStandard {
  // Dispatcher interface for messages
  uses interface ReceiveMsg as Recv[uint8_t id];
}
implementation {
  TOS_MsgPtr received(TOS_MsgPtr packet) {
    return signal Recv.receive[packet->type](packet);
  }
  ...
}
```

and the `App` configuration registers `AppM` to handle two kinds of messages:

```
configuration App {}
implementation {
  components AppM, AMStandard;
  AppM.ClearIdMsg -> AMStandard.Receive[AM_CLEARIDMSG];
  AppM.SetIdMsg -> AMStandard.Receive[AM_SETIDMSG];
}
```

3.1.7 *Known Uses.* The Active Messages networking layer (`AMStandard`, `AMPromiscuous`) uses a dispatcher for packet reception. It also provides a parameterized packet-sending interface; this ensures that packet types for sends and receives are specified in a uniform manner.

The Maté virtual machine uses a dispatcher to allow easy customization of instruction sets.

The Drip management protocol (described in Section 3.4) uses a Dispatcher to allow per-application configuration of management attributes.

3.1.8 *Consequences.* By leaving operation selection to nesC wirings, the dispatcher's implementation remains independent of what an application supports. However, finding the full set of supported operations can require looking at many files (using `nesdoc`, the nesC documentation tool, can help). Sloppy operation identifier management can lead to dispatch problems. If two operations are wired with the same identifier, then a dispatch will call both, which may lead to resource conflicts, data corruption, or memory leaks from lost pointers. For example, the `ReceiveMsg` interface uses a buffer swap mechanism to pass buffers between the radio stack and network services, in which the higher component passes a new buffer in the return value of the event. If two services are wired to a given `ReceiveMsg` instance, only one of their pointers will be passed and the second will be lost. Wiring in this fashion is a compile-time warning in nesC, but it is still a common bug for novice TinyOS developers.

The current nesC compiler compiles parameterized interface dispatch to a C `switch` statement. Thus, the code size and efficiency of a dispatcher will depend on the identifier space and on the C compiler. Good compilers should compile a compact identifier space to a bounds check, lookup in a table, and jump (slower than a function pointer, but still efficient). Using a large, sparse identifier space is likely to produce relatively large and slow dispatch. A benefit of using a `switch` statement over, e.g., a function pointer, is that the inlining performed by the nesC compiler (Section 4.3) may allow optimization across a dispatch call and between dispatch targets. The inlining also significantly reduces the cost of fine-grained dispatching, as seen in the Maté virtual machine (Section 4.3.3).

The key aspects of the dispatcher pattern are:

- It allows you to easily extend or modify the functionality an application supports: adding an operation requires a single wiring.
- It allows the elements of functionality to be independently implemented and reused. Because each operation is implemented in a component, it can be easily included in many applications. Keeping implementations separate can also simplify testing, as the components will be smaller, simpler, and easier to pinpoint faults in. The nesC compiler will automatically inline small operations, or you can explicitly request inlining; thus this decomposition has no performance cost.
- It requires the individual operations to follow a uniform interface. The dispatcher is usually not well suited to operations that have a wide range of semantics. As all implementations have to meet the same interface, broad semantics leads to the interface being overly general, pushing error checks from compile-time to runtime. An implementor forgetting a runtime parameter check can cause a hard to diagnose system failure.

The compile-time binding of the operation simplifies program analysis and puts dispatch tables in the compiled code, saving RAM. Dispatching provides a simple way to develop programs that execute in reaction to their environment.

3.1.9 *Related Patterns*

- Service Instance: creates many instances of an implementation of an interface, while a dispatcher selects between different implementations of an interface.
- Placeholder: allows an application to select an implementation at compile-time, while a dispatcher allows it to select an implementation at runtime.
- Keyset: the identifiers used to identify a Dispatcher's operation typically form a Global Keyset.

3.2 Structural: Service Instance

3.2.1 *Intent.* Allows multiple users to have separate instances of a particular service, where the instances can collaborate efficiently. The basic mechanism for virtualizing services.

3.2.2 *Motivation.* Sometimes many components or subsystems need to use a system abstraction, but each user wants a separate instance of that service. We do not know how many users there will be until we build a complete application. Each instance requires maintaining some state and the service implementation needs to access all of this state to make decisions.

For example, a wide range of TinyOS components need timers for everything from network timeouts to sensor sampling. Each timer appears independent, but they all operate on top of a single hardware clock. An efficient implementation thus requires knowing the state of all of the timers. If the implementation can easily determine which timer has to fire next, then it can schedule the underlying clock resource to fire as few interrupts as possible to meet this lowest timer's requirement. Firing fewer interrupts allows the CPU to sleep more, saving energy and increasing lifetime.

The traditional object-oriented approach to this problem is to instantiate an object representing the service and use another class to coordinate state. This approach is not applicable in nesC, as we cannot have multiple copies of components,[5] and either requires sharing state across objects, which is contrary to encapsulation, or it requires state copying, which uses additional RAM.

Implementing each timer in a separate module leads to duplicated code and requires intermodule coordination in order to figure out how to set the underlying hardware clock. Just setting it at a fixed rate and maintaining a counter for each Timer is inefficient: timer fidelity requires firing at a high rate, but it wastes energy to fire at 1 kHz if the next timer is in 4 seconds.

The Service Instance pattern provides a solution to these problems. Using this pattern, each user of a service can have its own (virtual) instance, but instances share code and can access each other's state. A component following the Service Instance pattern provides its service in a parameterized interface; each user wires to a unique instance of the interface using `unique`. The underlying component receives the unique identity of each client in each command and can use it to index into a state array. The component can determine at compile-time

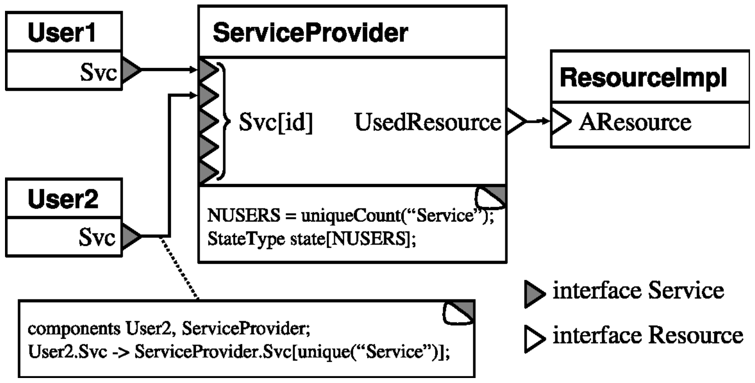
[5]This restriction is lifted in nesC 1.2 (Section 4.4).

how many instances exist using the `uniqueCount` function and dimension the state array accordingly.

3.2.3 *Applicable When*

- A component needs to provide multiple instances of a service, but does not know how many until compile time.
- Each service instance appears to its user to be independent of the others.
- The service implementation needs to be able to easily access the state of every instance.

3.2.4 *Structure.*



3.2.5 *Participants*

- **ServiceProvider**: allocates state for each instance of the service and coordinates underlying resources based on all of the instances.
- **ResourceImpl**: an underlying system resource that ServiceProvider multiplexes and demultiplexes service instances on.

3.2.6 *Sample Code.*

`TimerC` wires `TimerM`, which contains the actual timer logic, to an underlying hardware clock and exports its `Timer` interfaces:

```
configuration TimerC {
  provides interface Timer[uint8_t id];
}
implementation {
  components TimerM, ClockC;

  Timer = TimerM.Timer;
  TimerM.Clock -> ClockC.Clock;
}
```

and `TimerM` uses `uniqueCount` to determine how many timers to allocate and accesses them using unique IDs:

```
module TimerM {
  provides interface Timer[uint8_t clientId];
  uses interface Clock;
}
implementation {
  // per-client state
  timer_t timers[uniqueCount("Timer")];

  command result_t Timer.start[uint8_t clientId](...) {
    if (timers[clientId].busy)
      ...
  }
}
```

Clients wanting a timer wire using `unique`:

```
  C.Timer -> TimerC.Timer[unique("Timer")];
```

3.2.7 *Known Uses.* `TimerC`, as detailed above, uses a service instance pattern to manage various application timers.

The viral code propagation subsystem of the Maté virtual machine uses a service instance to manage version metadata for code capsules. As the virtual machine is customizable, the number of needed capsules is not known until the virtual machine is actually compiled.

In a similar vein, the epidemic dissemination protocol Drip uses the service instance pattern to maintain epidemic state for each disseminated value.

3.2.8 *Consequences.* The key aspects of the Service Instance pattern are:

- It allows many components to request independent instances of a common system service: adding an instance requires a single wiring.
- It controls state allocation, so the amount of RAM used is scaled to exactly the number of instances needed, conserving memory while preventing runtime failures because of many requests exhausting resources.
- It allows a single component to coordinate all of the instances, which enables efficient resource management and coordination.

Because the pattern scales to a variable number of instances, the cost of its operations may scale linearly with the number of users. For example, if setting the underlying clock interrupt rate depends on the timer with the shortest remaining duration, an implementation might determine this by scanning all of the timers, an $O(n)$ operation.

If many users require an instance of a service, but each of those instances are rarely used, then allocating state for each one can be wasteful. The other option is to allocate a smaller amount of state and dynamically allocate it to users as need be. This can conserve RAM, but requires more RAM per real instance (client IDs need to be maintained), imposes a CPU overhead (allocation and deallocation), can fail at runtime (if there are too many simultaneous users), and assumes a reclamation strategy (misuse of which would lead to leaks). This long list of challenges makes the Service Instance an attractive—and more and more commonly used—way to efficiently support application requirements. There are situations, however, when a component internally reuses a single service instance for several purposes: for example, the Maté code propagation component `MVirus` uses a single timer instance for several different timers, which never operate concurrently.

3.2.9 *Related Patterns*

- Dispatcher: a service instance creates many instances of an implementation of an interface, while a dispatcher selects between different implementations of an interface.
- Keyset: a Service Instance's instance identifiers form a Local Keyset.

3.3 Namespace: Keysets

3.3.1 *Intent.* Provide namespaces for referring to protocols, structures, or other entities in a program.

3.3.2 *Motivation.* A typical sensor network program needs namespaces for the various entities it manages, such as protocols, data types, or structure instances. Limited resources mean names are usually stored as small integer keys.

For data types representing internal program structures, each instance must have a unique name, but as they are only relevant to a single mote, the names can be chosen freely. These *local* namespaces are usually dense, for efficiency. The Service Instance pattern (Section 3.2) uses a local namespace to identify instances. In contrast, communication requires a shared, *global* namespace: two motes/applications must agree on an element's name. As a mote may only use a few elements, global namespaces are typically sparse. The Dispatcher pattern (Section 3.1) uses a global namespace to select operations.

The Keyset patterns provide solutions to these problems. Using these patterns, programs can refer to elements using identifiers optimized for their particular use. Components using the Keyset patterns often take advantage of a parameterized interface, in which the parameter is an element in a Keyset. Local Keysets are designed for referring to local data structures (e.g., arrays) and are generated with `unique`; Global Keysets are designed for communication and use global constants.

The bytecodes of the Maté virtual machine form a Global Keyset. The Maté scheduler uses these in conjunction with a Dispatcher to execute individual instructions, each of which is implemented in a separate component. Maté also uses a Local Keyset to identify locks corresponding to resources used by Maté programs. These lock identifiers are allocated with `unique` as the Maté virtual machine can be compiled with varying sets of resources.

The file descriptors of the Matchbox flash filesystem form a Local Keyset.

3.3.3 *Applicable When*

- A program must keep track of a set of elements or data types.
- The set is known and fixed at compile-time.

3.3.4 *Sample Code.* The Maté bytecodes are defined as global constants:

```
typedef enum {
  OP_HALT = 0x0,
  OP_MADD =  0x1,
  OP_MBA3 =  0x2,
  OP_MBF3 =  0xa,
  ...
} MateInstruction;
```

and used by the Maté scheduler to execute individual instructions:

```
module MateEngineM {
  uses interface MateBytecode[uint8_t bytecode];
  ...
}
implementation {
  void computeInstruction(MateContext* context) {
    MateOpcode instr = getOpcode(context);
    context->pc += call MateBytecode.byteLength[instr]();
    call MateBytecode.execute[instr](context);
  }
  ...
}
```

The Maté lock subsystem identifies locks by small integers:

```
module MLocks {
  provides interface MateLocks as Locks;
}
implementation {
  MateLock locks[MATE_LOCK_COUNT];

  command void Locks.lock(MateContext* uint8_t lockNum) {
    locks[lockNum].holder = context;
    context->heldSet[lockNum / 8] |= 1 << (lockNum % 8);
  }
  ...
```

Locks are allocated in components providing shared resources:

```
module OPgetsetvar1M { ... } // a shared variable
implementation {
  typedef enum {
    MATE_LOCK_1_0 = unique("MateLock"),
    MATE_LOCK_1_1 = unique("MateLock"),
  } LockNames;
  ...

module OPbpush1M { ... } // a shared buffer
```

```
implementation {
  typedef enum {
    MATE_BUF_LOCK_1_0 = unique("MateLock"),
    MATE_BUF_LOCK_1_1 = unique("MateLock"),
  } BufLockNames;
  ...
```

and `uniqueCount` is used to find the total number of locks:

```
enum {
  MATE_LOCK_COUNT = uniqueCount("MateLock")
};
```

3.3.5 *Known Uses.* Many components use Local Keysets: they are a fundamental part of the Service Instance pattern. See, for example, the timer service, `TimerC`, or the Matchbox flash file system.

Maté uses a Local Keyset to keep track of Maté shared resource locks (see above).

Active Messages (`AMStandard`) uses a Global Keyset for Active Message types.

The Drip management protocol uses a Global Keyset for referring to configurable variables.

The TinyDB sensor-network-as-database application [Madden et al. 2002] uses a Global Keyset for its attributes; in this case, however, the keyset is composed of strings, which are then mapped to a Local Keyset using a table.

3.3.6 *Consequences.* Keysets allow a component to refer to data items or types through a parameterized interface. In a Local Keyset, `unique` ensures that every element has a unique identifier. Global Keysets can also have unique identifiers, but this requires external namespace management.

As Local Keysets are generated with `unique`, mapping names to keys (e.g., for debugging purposes) is unobvious. The nesC constant generator, `ncg`, can be used to extract this information.

Keysets are rarely used in isolation; they support other patterns such as Dispatcher and Service Instance.

3.3.7 *Related Patterns*

- Keymap: two Keysets are often related, e.g., one Service Instance may be built on top of another, requiring a mapping between two Keysets. The Keymap pattern provides an efficient way of implementing such maps.
- Service Instance: the identifiers used to identify individual services form a Local Keyset.
- Dispatcher: the identifiers used by a dispatcher are typically taken from a Global Keyset.

3.4 Namespace: Keymap

3.4.1 *Intent.* Map keys from one keyset to another. Allows you to translate global, shared names to local, optimized names, or to efficiently subset another keyset.

3.4.2 *Motivation.* Mapping between namespaces is often useful: it allows motes to use a global, sparse namespace for easy cross-application communication and an internal, compact namespace for efficiency.

The Drip management protocol uses the Keyset and Keymap patterns to allow a user to configure parameters at runtime. A component registers a parameter with the `DripC` component with a Global Keyset, so it can be named in an application-independent manner. The user modifies a parameter by sending a key-value pair using an epidemic protocol, which distributes the change to every mote. `DripC` maintains state for each configurable parameter with the Service Instance pattern, using a Local Keyset. A Keymap maps the global key to the local key.

Keymaps are also useful for mapping between two local keysets, when some service, based on the Service Instance pattern, accesses a subset of the resources provided by another service, also based on the Service Instance pattern.

For instance, in TinyOS 2.0, there are three storage abstractions with different interfaces: blocks, logs, and configuration data. Each of these is identified by keys from its own local keyset. However, all three are built upon a common volume abstraction (provided by the StorageManagerC component) used to partition a mote's flash chip into independent areas. The volumes used by an application are identified by a local keyset. Thus it is necessary to map a block's identifier to its corresponding volume identifier.

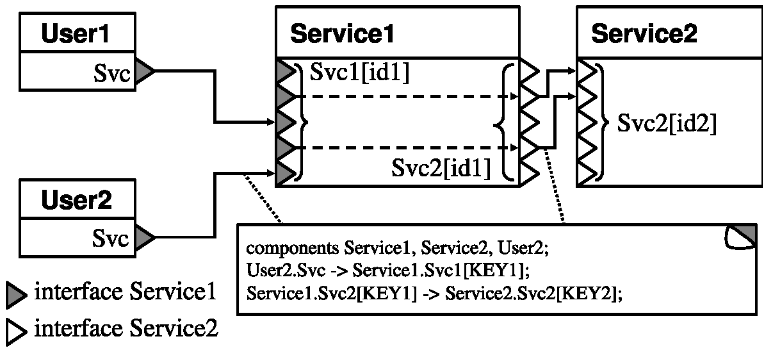
Maps could be implemented using a table and some lookup code. However, this has several problems. If we want to store this table in ROM, then it must be initialized in one place. However, this conflicts with the desire to specify keys in separate components (either with `unique` or with constants). If the table is stored in RAM, then keys can be specified in separate components, but RAM is in very short supply on motes. Finally, keys of Global Keysets are sparse, so the resulting tables would be large and waste space.

Instead, we can use nesC's wiring to build Keymaps. By mapping a parameterized interface indexed with one key to another parameterized interface indexed by a second key, we can have the nesC compiler generate the map at compile-time. In addition, as the map exists as an automatically generated switch statement, it uses no RAM.

3.4.3 *Applicable When*

- An application wants to connect services using different identifier spaces. For instance, the application wants to map global identifiers used for communication (or other purposes) to local identifiers for efficiency. Or, two services are implemented following the Service Instance pattern, and the first service needs an instance of the second service for each of its own instances.
- The identifiers are integer constants.
- The map is known at compile-time.

3.4.4 *Structure.*


User1
Svc
User2
Svc
Service1
Svc1[id1]
Svc2[id1]
Service2
Svc2[id2]
interface Service1
interface Service2
components Service1, Service2, User2;
User2.Svc -> Service1.Svc1[KEY1];
Service1.Svc2[KEY1] -> Service2.Svc2[KEY2];

3.4.5 *Participants*

- **Service1**: service accessed via key 1, dependent on Service2.
- **Service2**: service accessed via key 2.

3.4.6 *Sample Code.* The `DripC` component provides a parameterized interface for components to register configurable values with a Global Keyset:

```
enum { DRIP_GLOBAL = 0x20};
App.Drip -> DripC.Drip[DRIP_GLOBAL];
```

`DripC` uses another component to manage its internal state, `DripStateM`. `DripStateM` uses a Local Keyset for the configurable values (an example of the Service Instance pattern, in Section 3.2), and a Keymap maps between the two:

```
enum { DRIP_LOCAL = unique("DripState")};
DripC.DripState[DRIP_GLOBAL] -> DripStateM.DripState[DRIP_LOCAL];
```

In this example, a user can generate a new value for `App`'s parameter and distribute it based on the `DRIP_GLOBAL` key. `DripC` uses the global key to refer to the value, but `DripStateM` can use a local key to refer to the state it maintains for that value. The wiring compiles down to a simple switch statement that calls `DripStateM` with the proper local key.

The `BlockStorageC` component follows the Service Instance pattern to provide access to blocks and the `StorageManagerC` uses the same pattern to provide access to volumes:

```
configuration BlockStorageC {
  provides interface Block[int blockId];
} ...
configuration StorageManagerC {
  provides interface Volume[int volumeId];
} ...
```

To use a block, you need to allocate unique block and volume identifiers and wire `BlockStorageC` to `StorageManagerC` to form the Keymap:

```
enum {
  MY_BLOCK = unique("Block"),
  MY_VOLUME = unique("Volume")
};

configuration MyBlock {
  provides interface Block;
}
implementation {
  components BlockStorageC, StorageManagerC;

  Block = BlockStorageC.Block[MY_BLOCK];
  BlockStorageC.Volume[MY_BLOCK] -> StorageManagerC.Volume[MY_VOLUME];
}
```

3.4.7 *Known Uses.* The Drip parameter configuration component, described above, uses a Keymap.

The TinyOS 2.0 storage system, also described above, uses a Keymap to map the different storage abstractions to the common volume abstraction.

3.4.8 *Consequences.* A Keymap uses nesC wiring to allow components to transparently map between different keysets. As with Keysets, the Keymap must be fixed at compile-time.

A Keymap translates into a `switch` at compile-time. It thus does not use any RAM; its speed depends on the behavior of the C compiler used to compile nesC's output.

Keymaps only support mapping between integers. If you need, e.g., to map from strings to a Local Keyset, you will need to build your own map.

3.4.9 *Related Patterns*

- Keyset: A Keymap establishes a map from one keyset to another.

3.5 Structural: Placeholder

3.5.1 *Intent.* Easily change which implementation of a service an entire application uses. Prevent inadvertent inclusion of multiple, incompatible implementations.

3.5.2 *Motivation.* Many TinyOS systems and abstractions have several implementations. For example, there are many ad-hoc tree routing protocols (Route, MintRoute, ReliableRoute), but they all expose the same interface, `Send`. The standardized interface allows applications to use any of the implementations without code changes. Simpler abstractions can also have multiple implementations. For example, the `LedsC` component actually turns the LEDs on and off, while the `NoLedsC` component, which provides the same interface, has null operations. During testing, `LedsC` is useful for debugging, but in deployment it is a significant energy cost and usually replaced with `NoLedsC`.

Sometimes, the decision of which implementation to use needs to be uniform across an application. For example, if a network health-monitoring subsystem (`HealthC`) wires to MintRoute, while an application uses ReliableRoute, two routing trees will be built, wasting resources. As every configuration that wires to a service names it, changing the choice of implementation in a large application could require changing many files. Some of these files, such as `HealthC`, are part of the system; an application writer should not have to modify them.

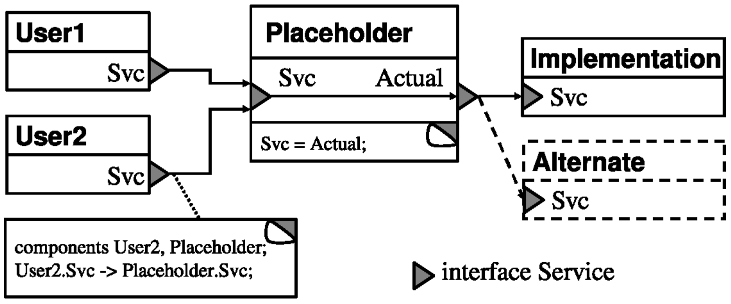
One option is for every implementation to use the same component name and put them in separate directories. Manipulating the nesC search order allows an application to select which version to use. This approach does not scale well: each implementation of each component needs a separate directory. Streamlining this structure by bundling several implementations (e.g., the "safe" versions and the "optimized" ones) in a single directory requires all-or-nothing inclusion. This approach also precludes the possibility of including two implementations, even if they can interoperate.

The Placeholder pattern offers a solution. A placeholder configuration represents the desired service through a level of naming indirection. All components that need to use the service wire to the placeholder. The placeholder itself is just "a pass through" of the service's interfaces. A second configuration (typically provided by the application) wires the placeholder to the selected implementation. This selection can then be changed centrally by editing a single file. As the level of indirection is solely in terms of names—there is no additional code generated—it imposes no CPU overhead.

3.5.3 *Applicable When*

- A component or service has multiple, mutually exclusive implementations.
- Many subsystems and parts of your application need to use this component/ service.
- You need to easily switch between the implementations.

3.5.4 *Structure.*



3.5.5 *Participants*

- **Placeholder**: the component that all other components wire to. It encapsulates the implementation and exports its interfaces with pass-through wiring. It has the same signature as the implementation component.
- **Implementation**: the specific version of the component.

3.5.6 *Sample Code.* Several parts of an application use ad-hoc collection routing to collect and aggregate sensor readings. However, the application design is independent of a particular routing implementation, so that improvements or new algorithms can be easily incorporated. The routing subsystem is represented by a Placeholder, which provides a unified name for the underlying implementation and just exports its interfaces:

```
configuration CollectionRouter {
  provides interface StdControl as SC;
  uses     interface StdControl as ActualSC;
  provides interface SendMsg as Send;
  uses     interface SendMsg as ActualSend;
}
implementation {
  SC = ActualSC;     // Just "forward" the
  Send = ActualSend; // interfaces
}
```

Component using collection routing wire to CollectionRouter:

```
  SensingM.Send -> CollectionRouter.Send;
```

and the application must globally select its routing component by wiring the "Actual" interfaces of the Placeholder to the desired component:

```
configuration AppMain { }
implementation {
  components CollectionRouter, EWMARouter;

  CollectionRouter.ActualSC -> EWMARouter.SC;
  CollectionRouter.ActualSend -> EWMARouter.Send;
  ...
}
```

3.5.7 *Known Uses.* The Maté virtual machine uses Placeholders for all its major abstractions (stacks, type checking, locks, etc). The motlle (a Schemelike language) interpreter implemented in Maté replaces Maté's default stack handler because it uses a different value representation—the replacement stack handler converts between the motlle and Maté value representations.

Hardware abstraction is supported in TinyOS by providing different versions of low-level components for each platform. In this case, implementations are selected by the platform choice rather than the application.

3.5.8 *Consequences.* The key aspects of the Placeholder pattern are:

- Establishes a global name that users of a common service can wire to.
- Allows you to specify the implementation of the service on an application-wide basis.
- Does not require every component to use the Placeholder's implementation.

By adding a level of naming indirection, a Placeholder provides a single point at which you can choose an implementation. Placeholders create a global namespace for implementation-independent users of common system services. As using the Placeholder pattern generally requires every component to wire to the Placeholder instead of a concrete instance, incorporating a Placeholder into an existing application can require modifying many components. However, the nesC compiler optimizes away the added level of wiring indirection, so a Placeholder imposes no runtime overhead. The Placeholder supports flexible composition and simplifies use of alternative service implementations.

3.5.9 *Related Patterns*

- Dispatcher: a Placeholder allows an application to select an implementation at compile-time, while a Dispatcher allows it to select an implementation at runtime.
- Facade: a Placeholder allows easy selection of the implementation of a group of interfaces, while a Facade allows easy use of a group of interfaces. An application may well connect a Placeholder to a Facade.

3.6 Structural: Facade

3.6.1 *Intent.* Provides a unified access point to a set of interrelated services and interfaces. Simplifies use, inclusion, and composition of the subservices.

3.6.2 *Motivation.* Complex system components, such as a filesystem or networking abstraction, are often implemented across many components. Higher-level operations may be based on lower-level ones, and a user needs access to both. Complex functionality may be spread across several components. Although implemented separately, these pieces of functionality are part of a cohesive whole that we want to present as a logical unit.

For example, the Matchbox filing system provides interfaces for reading and writing files, as well as for metadata operations, such as deleting and renaming. Separate modules implement each of the interfaces, depending on common underlying services, such as reading blocks.

One option would be to put all of the operations in a single, shared interface. This raises two problems. First, the nesC wiring rules mean that a component that wants to use *any* command in the interface has to handle *all* of its events. In the case of a file system, all the operations are split-phase; having to handle a half dozen events (`readDone`, `writeDone`, `openDone`, etc.) merely to be able to delete a file is hardly usable. Second, the implementation cannot be easily decomposed into separate components without introducing internal interfaces, as the top-level component will need to call out into the subcomponents. Implementing the entire subsystem as a single huge component is not easy to maintain.

Another option is to export each interface in a separate component (e.g., MatchboxRead, MatchboxWrite, MatchboxRename). This increases wiring complexity, making the abstraction more difficult to use. For a simple open, read and write sequence, the application would have to wire to three different components. In addition, each interface would need a separate configuration to wire it to the subsystems it depends on, increasing clutter in the component namespace. The implementer needs to be careful with these configurations, to prevent inadvertent double-wirings.

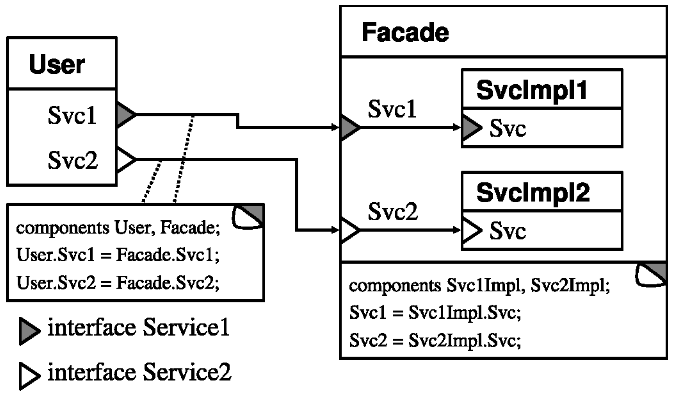
The Facade pattern provides a better solution to this problem. The Facade pattern provides a uniform access point to interfaces provided by many components. A Facade is a nesC configuration that defines a coherent abstraction boundary by exporting the interfaces of several underlying components. In addition, the Facade can wire the underlying components, simplifying dependency resolution.

A nesC Facade has strong resemblances to the object-oriented pattern of the same name [Gamma et al. 1995]. The distinction lies in nesC's static model. An object-oriented Facade instantiates its subcomponents at runtime, storing pointers and resolving operations through another level of call indirection. In contrast, as a nesC Facade is defined through naming (pass through wiring) at compile time, there is no runtime cost.

3.6.3 *Applicable When*

- An abstraction, or series of related abstractions, is implemented across several separate components.
- It is preferable to present the abstraction as a whole rather than in parts.

3.6.4 *Structure.*



3.6.5 *Participants*

- **Facade**: the uniform presentation of a set of related services.
- **SvcImpl**: the separate implementations of each service composing the Facade.

3.6.6 *Sample Code.* The Matchbox filing system uses a Facade to present a uniform filesystem abstraction. File operations are all implemented in different components, but the top-level Matchbox configuration provides them in a single place. Each of these components depends on a wide range of underlying abstractions, such as a block interface to nonvolatile storage; `Matchbox` wires them appropriately, resolving all of the dependencies.

```
configuration Matchbox {
  provides {
    interface FileRead[uint8_t fd];
    interface FileWrite[uint8_t fd];
    interface FileDir;
    interface FileRename;
    interface FileDelete;
  }
}
implementation {
  // File operation implementations
  components Read, Write, Dir, Rename, Delete;

  FileRead = Read.FileRead;
  FileWrite = Write.FileWrite;
  FileDir = Dir.FileDir;
  FileRename = Rename.FileRename;
  FileDelete = Delete.FileDelete;
  // Wiring of operations to sub-services omitted
}
```

3.6.7 *Known Uses.* Several stable, commonly used abstract boundaries have emerged in TinyOS [Levis et al. 2004], such as `GenericComm` (the network stack, combining radio and serial communication) and `Matchbox` (a file system), The presentation of these APIs is almost always a Facade.

3.6.8 *Consequences.* The key aspects of the Facade pattern are:

- Provides an abstraction boundary as a set of interfaces. A user can easily see the set of operations the abstraction supports, and only needs to include a single component to use the whole service.
- Presents the interfaces separately. A user can wire to only the needed parts of the abstraction, but be certain everything underneath is composed correctly.

A Facade is not always without cost. Because the Facade names all of its subparts, they will all be included in the application. While the nesC compiler attempts to remove unreachable code, this analysis is necessarily conservative and may end up keeping much useless code. In particular, unused interrupt handlers are never removed, so all the code reachable from them will be included every time the Facade is used. If you expect applications to only use a very narrow part of an abstraction, then a Facade can be wasteful.

3.6.9 *Related Patterns*

- Placeholder: a placeholder allows easy selection of the implementation of a group of interfaces, while a facade allows easy use of a group of interfaces. An application may well connect a placeholder to a facade.

3.7 Behavioral: Decorator

3.7.1 *Intent.* Enhance or modify a component's capabilities without modifying its implementation. Be able to apply these changes to any component that provides the interface.

3.7.2 *Motivation.* We often need to add extra functionality to an existing component, or to modify the way it works without changing its interfaces. For instance, the standard `ByteEEPROM` component provides a `LogData` interface to log data to a region of flash memory. In some circumstances, we would like to introduce a RAM write buffer on top of the interface. This would reduce the number of actual writes to the EEPROM, conserving energy (writes to EEPROM are expensive) and the lifetime of the medium.

Adding a buffer to the `ByteEEPROM` component forces all logging applications to allocate the buffer. As some application may not able to spare the RAM, this is undesirable. Providing two versions, buffered and unbuffered, replicates code, reducing reuse and increasing the possibility of incomplete bug fixes. It is possible that several implementers of the interface—any component that provides `LogData`—may benefit from the added functionality. Having multiple copies of the buffering version, spread across several services, further replicates code.

There are two traditional object-oriented approaches to this problem: inheritance, which defines the relationship at compile time through a class hierarchy, and decorators [Gamma et al. 1995], which define the relationship at runtime through encapsulation. As nesC is not an object-oriented language, and has no notion of inheritance, the former option is not possible. Similarly, runtime encapsulation is not readily supported by nesC's static component composition model and imposes overhead in terms of pointers and call forwarding. However, we can use nesC's component composition and wiring to provide a compile-time version of the Decorator.

A Decorator component is typically a module that provides and uses the same interface type, such as `LogData`. The provided interface adds functionality on top of the used interface. For example, the `BufferedLog` component sits on top of a `LogData` provider. It implements its additional functionality by aggregating several `BufferedLog` writes into a single `LogData` write.

Using a Decorator can have further benefits. In addition to augmenting existing interfaces, they can introduce new ones that provide alternative abstractions. For example, `BufferedLog` provides a synchronous (not split-phase) `FastLog` interface; a call to `FastLog` writes directly into the buffer.

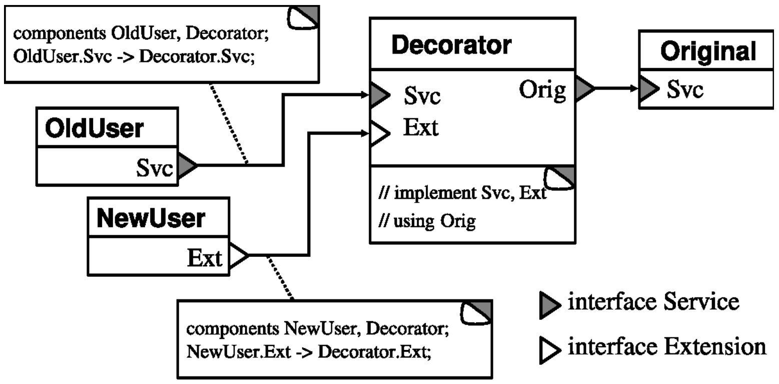
Finally, separating added functionality into a Decorator allows it to apply to any implementation. For example, a packet send queue Decorator can be interposed on top of any networking abstraction that provides the `Send` interface; this allows flexible interpositioning of queues and queueing policies in a networking system.

3.7.3 *Applicable When*

- You wish to extend the functionality of an existing component without changing its implementation, or

- You wish to provide several variants of a component without having to implement each possible combination separately.

3.7.4 *Structure.*



3.7.5 *Participants*

- **Original**: the original service.
- **Decorator**: the extra functionality added to the service.

3.7.6 *Sample Code.* The standard `LogData` interface includes split-phase `erase`, `append` and `sync` operations. `BufferedLog` adds buffering to the `LogData` operations, and, in addition, supports a `FastLogData` interface with a nonsplit-phase `append` operation (for small writes only):

```
module BufferedLog {
  provides interface LogData as Log;
  provides interface FastLogData as FastLog;
  uses interface LogData as UnbufferedLog;
}
implementation {
  uint8_t buffer1[BUFSIZE], buffer2[BUFSIZE];
  uint8_t *buffer;
  command result_t FastLog.append(data, n) {
    if (bufferFull()) {
      call UnbufferedLog.append(buffer, offset);
      // ... switch to other buffer ...
    }
    // ... append to buffer ...
  }
```

The `SendQueue` Decorator introduces a send queue on top of a split-phase `Send` interface:

```
module SendQueue {
  provides interface Send;
  uses interface Send as SubSend;
}
```

```
implementation {
  TOS_MsgPtr queue[QUEUE_SIZE];
  uint8_t head, tail;
  command result_t Send.send(TOS_MsgPtr msg) {
    if (!queueFull()) enqueue(msg);
    if (!subSendBusy()) startSendRequest();
  }
```

3.7.7 *Known Uses.* `BufferedLog` improves split-phase logging interface by buffering small writes.

`CRCFilter` decorates a `ReceiveMsg` interface by filtering packets that did not pass a CRC check: packets that pass are signaled up; those that do not are not.

`QueuedSend` decorates a `SendMsg` interface by enqueuing multiple requests, which it serializes onto an underlying `SendMsg`, providing in-order transmission.

3.7.8 *Consequences.* Applying a Decorator allows you to extend or modify a component's behavior though a separate component: the original implementation can remain unchanged. In addition, the Decorator can be applied to any component that provides the interface.

In most cases, a decorated component should not be used directly, as the Decorator is already handling its events. The Placeholder pattern (Section 3.5) can be used to help ensure this.

Additional interfaces are likely to use the underlying component, creating dependencies between the original and extra interfaces of a Decorator. For instance, in `BufferedLog`, `FastLog` uses `UnbufferedLog`, so concurrent requests to `FastLog` and `Log` are likely to conflict: only one can access the `UnbufferedLog` at once.

Decorating an existing component may consume more resources (code space, power, RAM) than writing a new special-purpose component.

Decorators are a lightweight, but flexible, way to extend component functionality. Interpositioning is a common technique in building networking stacks [Kohler et al. 2000] and Decorators enable this style of composition.

3.7.9 *Related Patterns*

- Adapter: An Adapter presents the existing functionality of a component with a different interface, rather than adding additional functionality and preserving the current interface.

3.8 Behavioral: Adapter

3.8.1 *Intent.* Convert the interface of a component into another interface, without modifying the original implementation. Allow two components with different interfaces to interoperate.

3.8.2 *Motivation.* Sometimes, a piece of functionality offered by a component with one interface needs to be accessed by another component via a different interface. For instance, the TinyDB [Madden et al. 2002] application—which provides a database-like abstraction over a sensor network—accesses the "database attributes" of the sensor network via the `AttrRegister` interface. These attributes represent, among other things, the sensors attached to the sensor network's motes. However, in TinyOS, sensors are accessed via the `ADC` interface.

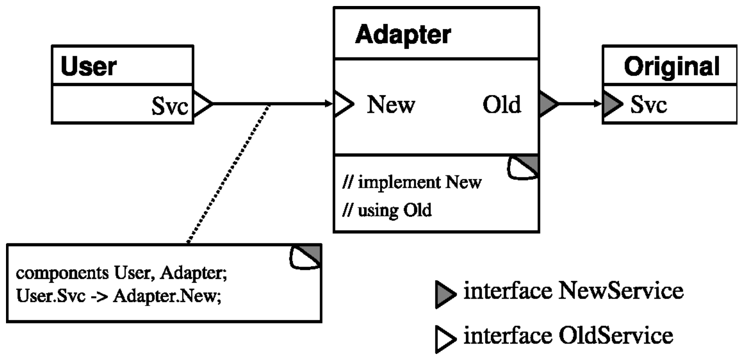
Modifying each sensor to provide an `AttrRegister` as well, or instead of, its current interfaces is not desirable, as `AttrRegister` provides functionality, which is not desirable for all applications (named access to sensors) and does not provide necessary functionality (access to sensors from interrupt handlers). Instead, TinyDB uses Adapter components, which implement the `AttrRegister` interface based on the functionality of the the `ADC` interface provided by sensors.

An Adapter is a component that provides an interface of type A (for instance, `AttrRegister`) and uses an interface of type B (for instance `ADC`), and implements the operations of A in terms of those of B. An Adapter may also need to implement functionality not provided by the B interface, e.g., `AttrRegister` needs to provide a name for the attribute. More generally, an Adapter may provide several interfaces $A_1, \ldots, A_n$ and implement them in terms of several used interfaces $B_1, \ldots, B_m$.

3.8.3 *Applicable When*

- You wish to provide the functionality of an existing component with a different interface.

3.8.4 *Structure.*



3.8.5 *Participants*

- **Original**: the original service.
- **Adapter**: implements the new interface in terms of the functionality offered by the old.

3.8.6 *Sample Code.* The `AttrPhotoM` component adapts the standard `Photo` (light) sensor for use with TinyDB. It gives the attribute a name, implements getting the attribute using the underlying `ADC` interface, and refuses requests to set the attribute:

```
module AttrPhotoM {
  provides interface StdControl;
  provides interface AttrRegister;
  uses interface ADC;
}
implementation {
  void *buffer;
  command void StdControl.init() { // register the attribute's name
    and size signal AttrRegister.registerAttr("light", 2);
  }
  command result_t AttrRegister.getAttr(void *result) {
    buffer = result;
    return call ADC.getData();
  }
  event void ADC.dataReady(uint16_t data) {
    *(uint16_t*)buffer = data;
    signal AttrRegister.getAttrDone(result);
  }
  command result_t AttrRegister.setAttr(void *attrVal) {
    return FAIL; // cannot "set" a sensor
  }
}
```

3.8.7 *Known Uses.* Many TinyDB attributes are implemented using Adapters.

In TinyOS 2.0, hardware resources, such as A/D converters, are presented by a hardware abstraction layer (HAL), which offers high-level, but hardware-specific interfaces and a hardware-independent layer (HIL), which offers high-level, platform-independent interfaces. The HIL layer is typically an Adapter over the HAL layer. For example, see the `AdcP` A/D converter component for the ATmega128.

TinyOS 2.0 has two timing interfaces: `Alarm` signals timer events from an interrupt handler (immediately); `Timer` signals its events in a task (with some delay). The `AlarmToTimerC` component is an Adapter that converts between these interfaces.

3.8.8 *Consequences.* An Adapter allows a component to be reused in circumstances other than initially planned for without changing the original implementation.

In many cases, a component used with an Adapter cannot be used independently in the same application, as the Adapter will already be handling its events. As with the Decorator, the Placeholder pattern (Section 3.5) can help ensure this.

An Adapter can be used to adapt many different implementations of its used interfaces if it does not embody assumptions or behavior specific to a particular adapted component. However, the singleton nature of nesC components means that a particular adapter can only be used once in a given application.

Adding an additional layer to convert between interfaces may increase the application's resource consumption (ROM, RAM, and execution time).

3.8.9 *Related Patterns*

- Decorator: A Decorator adds functionality to an existing component while preserving its original interface. An Adapter presents existing (and possibly additional) functionality via a different interface.

4. DISCUSSION

We compare our design patterns to standard object-oriented patterns and show how they support TinyOS's design goals. We show that our patterns depend fundamentally on features of both the nesC language and compiler. In particular, nesC's optimizations have a major impact on the size and efficiency of pattern-based programs, reducing power use by up to 45% and code size by up to 67%. Finally, we discuss how these patterns have influenced the design of the nesC programming language and how recent changes to nesC address some of the limitations of our current patterns.

4.1 Comparison to Object-Oriented Patterns

The eight design patterns described in Section 3 can be separated into classes: Dispatcher, Service Instance, Keyset, and Keymap are specific to nesC, while Adapter, Decorator, Facade, and Placeholder have analogs in existing patterns [Gamma et al. 1995]. The differences from traditional object-oriented patterns stem from the design principles behind TinyOS [Levis et al. 2005b]. For example, TinyOS generally depends on static composition techniques to provide robust, unattended operation: function pointers or virtual functions can complicate program analysis, while dynamic allocation can fail at runtime if one allocator misbehaves. As a result, where many object-oriented patterns increase object flexibility and reusability by allowing behavior changes at runtime, our patterns require that most such decisions be taken by compile-time.

The nesC-specific patterns represent ways to make nesC's static programming model more practical. Service Instance allow services (e.g., timers, file systems) to have a variable number of clients; it is the standard pattern for a stateful TinyOS service. Dispatcher supports application-configured dispatching (e.g., message reception, user commands). The Keyset pattern both supports these two patterns and allows data structures to be sized according to a particular application's needs. Keymaps are a practical way of building components above Service Instances and of associating state with sparse identifier sets.

The TinyOS Adapter, Facade, and Decorator patterns have similar goals and structures to their identically named object-oriented analogs [Gamma et al. 1995, pp.139, 175, 185, respectively]. The Facade assembles a set of existing components and presents them as a single component to simplify use; the Decorator adds extra functionality to an existing component and the Adapter makes existing functionality available via a different interface. The differences lie in nesC's model of static composition. In the case of the Facade, this means that all of the relationships are bound at compile-time; in addition, nesC provides no way of making the internals of a Facade truly private (the internal components can always be referred to from elsewhere by name). Adapters and Decorators are more important than in an object-oriented context, as they provide a way of defining implementation–inheritance hierarchies in a component-based language. However, the use of any given Adapter or Decorator is limited by the singleton nature of components. Finally, Placeholder has similarities to the Bridge [Gamma et al. 1995, p. 151]: it simplifies implementation switching, but requires that the implementation selection be performed at compile-time.

4.2 Patterns Support TinyOS's Goals

The patterns we have presented, directly support TinyOS's design goals of robustness, low resource usage, supporting hardware evolution, enabling diverse service implementations, and adaptability to application requirements. Specifically,

- A Placeholder supports diverse implementations by simplifying implementation selection and hardware evolution by defining a platform-independent abstraction layer.
- Decorator and Adapter support diverse implementations and hardware evolution by enabling lightweight component extension.
- Service Instance and Dispatcher increase robustness and lower resource usage by resolving component interactions at compile-time.
- Dispatcher and Global Keyset improve application adaptability by providing a way to easily configure what operations an application supports and how it reacts to its environment.
- A Local Keyset increases robustness and lowers resource usage by sizing services to an application's needs.
- A Keymap lowers resource usage by building maps as code rather than in RAM; in addition, it allows Decorators and Adapters to be built for Service Instances.

4.3 Language and Compiler Support for Patterns

To be of practical use, these design patterns must be not only useful for embedded systems programming, but must also be expressible in a sufficiently concise fashion and should not impose significant code space or runtime overhead. We briefly describe the nesC compiler and its inlining and unreachable-code optimizations, and then evaluate how its features combined with the nesC language design supports design patterns.

4.3.1 *nesC compiler.* The nesC compiler generates a single C file containing the executable code of all the modules of the program, and "connection" functions representing the wiring specified by the configurations. Connection functions:

- Exist for each used command and provided event of each module.
- Contain calls to the target(s) of the command or event (as specified by the configurations).
- Combine the results of multiple calls, as specified by the programmer.
- Use a `switch` to implement the dispatch specified by parameterized interfaces.

For instance, the connection function for the `Init.init` command of the `Main` component from Figure 1, Section 2 is:

```
void Main__Init__init() {
  AppM__Init__init();
  TempM__Init__init();
  LightM__Init__init();
}
```

The nesC compiler then performs two optimizations. First, it removes unreachable functions and variables, based on its knowledge of the program's call graph and the program's entry points (boot and interrupt handlers, specified in TinyOS's source code). Second, it aggressively inlines functions across the entire program, as described in the next section. Note that the program's call graph, used by both optimizations, is easily derived from the explicit function calls and the program's wiring. Edges resulting from function pointers (whose use is discouraged in nesC) are not present. The absence of these edges does not affect inlining and is handled in unreachablecode elimination by counting any reference to a function as a call.

Finally, the resulting C code is passed to a target-specific C compiler.

4.3.2 *Inlining in nesC* . Inlining in nesC has three goals: remove the overhead of wiring, remove the overhead of small, separate components, and reduce code size. These goals are achieved through a conservative, whole-program inliner, which takes as input the program's call graph annotated with each function's approximate size. The size is heuristically defined as the number of computational nodes in the function's abstract syntax tree.

Inlining proceeds as follows, until no other functions can be inlined:

- If a function has a single call site, inline it.
- If a function's size is below some threshold (see below), inline it.
- When inlining a function, update the call graph edges appropriately and add the inlined function's size minus one to all callers.

Inlining a function with a single call site normally reduces code size and increases performance. To avoid increasing code size when inlining functions with multiple call sites, we pick a small size threshold for inlining: inlining a small function is likely to reduce (or only slightly increase) code size through elimination of function call overhead and increased optimization opportunities. We chose a threshold of $9 + 2n$, where n is the number of the function's arguments. Functions with more arguments present more optimization opportunities, and, hence, should be inlined more aggressively. The specific constants were selected based on an evaluation of ten TinyOS programs on two platforms (mica2 and telosb). The parameters chosen are the largest (ordered by base size, then per-argument size) that keep code expansion below 5% on all programs.

Actual inlining is left to the C compiler, which processes the generated C file; nesC simply outputs appropriate `inline` directives.[6]

[6]All our current platforms use gcc. We ensure that it inlines all requested functions by passing it an -finline-limit=100,000 option.

4.3.3 *Evaluation.* Concise and efficient expression of our patterns is made possible by the following features:

- The inlining optimization leads to the inlining of most “connection” functions (because they are generally small and/or called only once), significantly reducing the cost of wiring. This makes it possible to break programs into many components without a large performance cost (Dispatcher, Placeholder, Facade, Decorator, Adapter). In addition, inlining will often inline code across component boundaries, further reducing the cost of using many components. For instance, the instructions of a virtual machine split across many components (following the Dispatcher pattern) are compiled into a single function as all instructions have a single call.
- Unreachable code elimination removes unused functionality (typically uncalled commands in OS services), allowing more general components to be designed (Facade).
- Parameterized interfaces allow runtime dispatches (Dispatcher, Service Instance, Keyset, Keymap).
- Unique identifiers support compile-time configuration of services, e.g., to identify clients (Service Instance, Keyset).

To show the importance of nesC’s optimizations in supporting the use of our patterns in real programs, we evaluated the code size and average power draw of five programs:

- Timer: runs three timers, at 25, 50, and 100 ms. There are two versions of this program, for TinyOS 1.1 and 2.0, respectively.
- DataCollection: sample a sensor twice a second, and send a radio message with the results every 10 s. There are two versions of this program, for TinyOS 1.1 and 2.0, respectively.
- VM: a bytecoded interpreter for a Schemelike language, built with the Maté virtual machine architecture [Levis et al. 2005a], running on TinyOS 1.1. The interpreter runs a simple program that performs a little computation ten times a second and sends a radio message with the results every 50 s.

TinyOS 2.0 and the Maté virtual machine make heavy use of all our patterns, while TinyOS 1.1 is much more monolithic. We compiled these five applications for mica2 motes, which have an Atmel ATmega128 microcontroller, running at 8 MHz, with 128 kB of flash and 4 kB of RAM. We measured power draw using an oscilloscope that measured current draw at 100 Hz for 100 s. The output of the nesC compiler was compiled with gcc 3.3.2 (for the TinyOS 1.1 programs) and gcc 3.4.3 (for the TinyOS 2.0 programs)—gcc 3.3.2 gives better performance for TinyOS programs but a bug prevents its use with TinyOS 2.0. Programs are optimized for size (gcc’s `-Os` option).

Table I shows the code size and power draw of these programs with and without the inlining and unreachable-code optimisations.[7] The first observation is that unreachable-code elimination and inlining are important in reducing the

[7]Unreachable-code elimination has no effect on power draw on the mica2 platform.

Table I. Effect of Optimizations on Code Size and Power Draw in Several TinyOS Programs

	Inlining + Unreachable-Code	Unreachable-Code Only	Unoptimized	Improvements (Code/Power)(%)
Timer 1.1	2.0 KB / 0.36 mW	2.7 KB / 0.39 mW	4.1 KB	51 / 8
Timer 2.0	2.6 KB / 0.43 mW	3.8 KB / 0.52 mW	8.0 KB	67 / 17
DataCollection 1.1	12.1 KB / 1.58 mW	14.8 KB / 1.64 mW	17.4 KB	30 / 4
DataCollection 2.0	11.3 KB / 1.90 mW	15.6 KB / 2.12 mW	20.8 KB	46 / 10
VM 1.1	43.5 KB / 4.95 mW	59.5 KB / 8.98 mW	81.8 KB	47 / 45

code size of all nesC programs, giving at least a 33% reduction. However, the effect is larger on programs using patterns (Timer 2.0, DataCollection 2.0, and VM 1.1), with a 46% or more reduction. Second, when comparing similar applications (Timer 1.1 versus 2.0 and DataCollection 1.1 versus 2.0), we see that the power decrease in the pattern-intensive programs is much larger: 17 versus 8% for Timer, and 10 versus 4% for DataCollection. Note that the differences in power consumption between the Timer and DataCollection applications in TinyOS 1.1 and 2.0 reflect, in part at least, differences in functionality: for instance, the TinyOS 2.0 timer maintains the current time, which requires it to handle more interrupts to deal with the overflow of the 8-bit hardware timer. Finally, the virtual machine shows that these optimizations are particularly important in large, pattern-intensive programs: VM's code size is reduced by 47% and its power by 45%.

4.4 nesC, Yesterday and Tomorrow

As experience in using TinyOS has grown, we have introduced features in nesC to make building applications easier. Design patterns have been the motivation for several of these features. For example, the first version of nesC (before TinyOS 1.0) had neither `unique` nor `uniqueCount`. Initial versions of the Timer component coalesced into Service Instance pattern, which led to the inclusion of `unique` and `uniqueCount`. The recently released 1.2 version of nesC introduces the feature of *generic components* to simplify using design patterns.

TinyOS design patterns are limited by the singleton nature of nesC components, leading to a significant amount of code duplication. For example, when wiring to a Service Instance, a programmer must carefully use the same incantation with a particular key for `unique`. If a program needs two copies of, e.g., a data filter Decorator, then two separate components must exist, and their code must be maintained separately. These examples involve replicated code: changing the Service Instance key requires changing every user of the service, and a typo in one instance of the key can lead to buggy behavior (the keys may no longer be unique).

Version 1.2 of nesC addresses this issue with *generic components*, which can be instantiated at compile time with numerical and type parameters. Essentially, component instantiation creates a copy of the code with arguments substituted for the parameters. Configurations (including generic configurations) can instantiate generic components:

```
components new LogBufferer() as LB, ByteEEPROM;
LB.UnbufferedLog -> ByteEEPROM;
```

Generic configurations allow a programmer to capture wiring patterns and represent them once. For example, the key a Service Instance component uses can be written in one place: instead of wiring with `unique`, a user of the service wires to an instance of a generic configuration:

```
  generic configuration TimerSvc() {
    provides interface Timer;
  }
  implementation {
    components TimerC;
    Timer = TimerC.Timer[unique("TimerKey")];
  }
  ....

  components User1, new TimerSvc() as MyTimer;
  User1.Timer -> MyTimer.Timer;
```

They also make the Keymap pattern much more convenient. The actual code for the `BlockStorageC` example from Section 3.4 is:

```
generic configuration BlockStorageC() {
  provides interface Block;
}
implementation {
  components BlockStorageM, StorageManagerC;

  enum {
    BLOCK_ID = unique("Block"),
    VOLUME_ID = unique("Volume")
  };
  Block = BlockStorageM.Block[BLOCK_ID];
  BlockStorageM.Volume[BLOCK_ID] -> StorageManagerC.Volume[VOLUME_ID];
}
```

When the programmer creates a `BlockStorageC` component, all the Keymap wiring is done automatically.

Generic modules make Decorators and Adapters much more reusable through multiple instantiation and component arguments. Generic components allow patterns such as Facade to have private components, whose interfaces are only accessible through what a configuration exposes. Finally, by providing a globally accessible name, a Placeholder provides a way to make a generic component behave like a nesC 1.1 singleton.

5. CONCLUSION

Like their object-oriented brethren, TinyOS design patterns are templates of how functional elements of a software system interact. Flexibility is a common goal, but in TinyOS we must also preserve the efficiency and reliability of nesC's static programming model. Thus, the TinyOS patterns allow most of this flexibility to be resolved at compile-time, through the use of wiring, `unique` and `uniqueCount`.

Our set of TinyOS design patterns is a work in progress. In particular, it is clear that analogs of many of the structural patterns from the original Design Patterns book [Gamma et al. 1995] can be expressed in nesC, with a "component = class," or "component = object" mapping. Translations of behavioral patterns is harder, reflecting the differences in resources and application domains. The fact that our list contains relatively few behavioral patterns (just Dispatcher, Decorator, and Adapter) may reflect the fact that, so far, TinyOS applications have been fairly simple.

Finally, our design patterns are reusable patterns of component composition. TinyOS has many other forms of patterns, such as interface patterns (e.g., split-phase operations, error handling),[8], and data-handling patterns (e.g., data pumps in the network stack). These other sorts of patterns deserve further investigation.

ACKNOWLEDGMENTS

This work was supported, in part, by the Defense Department Advanced Research Projects Agency (grants F33615-01-C-1895 and N6601-99-2-8913), the National Science Foundation (grants No. 0122599 and NSF IIS-033017), California MICRO program, and Intel Corporation. Research infrastructure was provided by the National Science Foundation (grant EIA-9802069).

REFERENCES

DOUGLASS, B. P. 2002. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Addison-Wesley, Reading, MA.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.

GAY, D., LEVIS, P., VON BEHREN, R., WELSH, M., BREWER, E., AND CULLER, D. 2003. The nesC language: A holistic approach to networked embedded systems. In *Proeedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. San Diego, CA. 1–11.

GIROD, L., ELSON, J., CERPA, A., STATHOPOULOS, T., RAMANATHAN, N., AND ESTRIN, D. 2004. EmStar: A software environment for developing and deploying wireless sensor networks. In *Proceedings of the 2004 USENIX Annual Technical Conference*. Boston, MA. 283–296.

GREENSTEIN, B., KOHLER, E., AND ESTRIN, D. 2004. A sensor network application construction kit (SNACK). In *Proceedings of the 2nd International Conference on Embedded Sensor Systems*. Baltimore, MD. 69–80.

HILL, J., SZEWCZYK, R., WOO, A., HOLLAR, S., CULLER, D. E., AND PISTER, K. S. J. 2000. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*. Cambridge, MA. 93–104.

KLEIMAN, S. 1986. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the 1986 USENIX Conference*. Atlanta, GA. 238–247.

KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. 2000. The Click modular router. *ACM Transactions on Computer Systems 18,* 3, 263–297.

LEVIS, P. AND GAY, D. 2004. TinyOS Design Patterns. http://sing.stanford.edu/tinyos/patterns.

LEVIS, P., MADDEN, S., GAY, D., POLASTRE, J., SZEWCZYK, R., WOO, A., BREWER, E., AND CULLER, D. 2004.

[8]The device patterns in EM⋆ [Girod et al. 2004] may provide inspiration here.

The emergence of networking abstractions and techniques in TinyOS. In *Proceedings of the 1st Symposium on Network Systems Design and Implementation*. San Francisco, CA. 1–14.

LEVIS, P., GAY, D., AND CULLER, D. 2005a. Active sensor networks. In *Proceedings of the 2nd Symposium on Network Systems Design and Implementation*. Boston, MA. 343–356.

LEVIS, P., MADDEN, S., POLASTRE, J., SZEWCZYK, R., WHITEHOUSE, K., WOO, A., GAY, D., HILL, J., WELSH, M., BREWER, E., AND CULLER, D. 2005b. TinyOS: An operating system for wireless sensor networks. In *Ambient Intelligence*. Springer-Verlag, New York.

MADDEN, S. R., FRANKLIN, M. J., HELLERSTEIN, J. M., AND HONG, W. 2002. TAG: A tiny aggregation service for ad-hoc sensor networks. In *Proceedings of the 5th Symposium on Operating System Design and Implementation*. Boston, MA. 131–146.

PATTERNSW1 2001. *OOPSLA Workshop Towards Patterns and Pattern Languages for OO Distributed Real-time and Embedded Systems*.

PATTERNSW2 2002. *OOPSLA Workshop on Patterns in Distributed Real-time and Embedded Systems*.

PATTERNSW3 2002. *PLOP Workshop on Patterns and Pattern Languages in Distributed Real-time and Embedded Systems*.

Received October 2005; revised April 2006; accepted June 2006