Software Design Patterns for TinyOS

David Gay
Intel Research, Berkeley
david.e.gay@intel.com

Phil Levis
University of California at Berkeley
pal@cs.berkeley.edu

David Culler
University of California at Berkeley
culler@cs.berkeley.edu

Abstract

We present design patterns used by software components in the TinyOS sensor network operating system. They differ significantly from traditional software design patterns due to the constraints of sensor networks and 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 optimisations.

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

General Terms Languages

Keywords Design Patterns, Embedded Systems, nesC, TinyOS

1. Introduction

TinyOS [1] 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 which are included as-needed in applications; a significant challenge in TinyOS development is the creation of flexible, reusable components. Additionally, programming abstractions for sensor networks, where TinyOS is the current OS-of-choice, are still under investigation. [2]

Writing solid, reusable software components is hard. Doing so for sensor networks is even harder. Limited resources (e.g., 4kB of RAM) and strict energy budgets (e.g., averages below 1mW) lead developers to write application-specific versions of many services. While specialised 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 re-use [3, 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 [2, Section 2.1]:

- Robustness: once deployed, a sensor network must run unattended for months or years.
- Low resource usage: sensor network nodes include very little RAM, and run off batteries.
- Diverse service implementations: applications should be able to choose between multiple implementations of, e.g., multi-hop 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 [4] — 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: the set of components and their interactions are fixed at compile-time (to promote reliability and efficiency), rather than at run-time, as object-oriented references and instantiation do. 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 five design patterns which show how nesC can be used to build components which address TinyOS's challenges.[1] 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 will include 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 seen above are not radically different from those of other embedded systems. Second, we explore how a few simple features of the nesC language and compiler, particularly parameterised 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

[1] Because of space constraints, we have omitted four more specialised patterns; these can be found on our website [5].

LCTES'05, June 15–17, 2005, Chicago, Illinois, USA.
Copyright © 2005 ACM 1-59593-018-3/05/0006…$5.00.

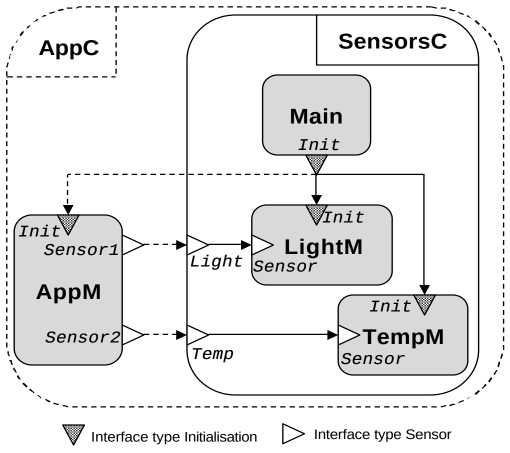


Figure 1. Sample Component Assembly. Solid rectangles are modules, open rectangles are configurations. Triangles pointing into a rectangle are provided interfaces, triangles pointing out are used interfaces. Dotted lines are "wires" added by configuration `AppC`, full lines are "wires" added by configuration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

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 analyse and distill TinyOS programming.

Although prior work has explored object oriented design patterns for embedded and real-time devices [6, 7, 8, 9, 10], they deal with platforms that have orders of magnitude more resources (e.g., a few MB 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 [11]. A SNACK system 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 five TinyOS design patterns, describing their motivation, consequences, and representation in nesC, as well as listing several TinyOS components that use them.[2] 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 [4] 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: all 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). To promote reliability and analysability, nesC does not support dynamic memory allocation or function pointers; all component interactions are specified at compile-time.

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

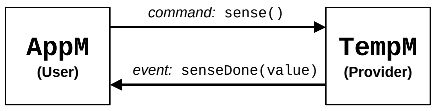


Figure 2. Typical split-phase operation.

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 `Initialise`. Modules are components implemented with C code, while configurations are components implemented by wiring other components together. In the example figure, `Main`, `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 Initialise as Init;
  uses interface Sense as Sensor1;
  uses interface Sense as Sensor2;
}
implementation { ... }
```

declares that `AppM` (from Figure 1) is a module which provides an interface named `Init` and uses two interfaces, named `Sensor1` and `Sensor2`. Each interface has a type, in this case either `Initialise` or `Sense`. A component name denotes a unique, singleton component[3]: 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 Initialise { // component initialisation
  command void init();
}

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

This interaction is bi-directional: *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. Figure 2 shows this relationship for `AppM` and `TempM`. To make the two directions syntactically explicit, nesC events are *signalled* while commands are *called*. In both cases, the actual interaction is a function call.

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:

[3] We discuss in Section 4 how the next version of nesC changes this, and its effect on design patterns.

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

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, `Light` with `LightM`'s `Sensor` interface. Additionally, `SensorsC` *wires* the system's initialisation interface (`Main.Init`) to the initialisation 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 initialised 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.[4]

2.3 Parameterised Interfaces

A parameterised 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 Initialise as Inits[int id];
  uses interface Sense as Sensors[int id];
} ...
```

[4] If a multiply wired function has non-void result, nesC combines the results via a programmer-specified function. [4]

In a module, commands and events of parameterised 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 parameterised interface to another parameterised interface. This equates `Example.Sensors[`i`]` to `ADC[`i`]` for all values of i:

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

2.4 `unique` and `uniqueCount`

In many cases, a programmer wants to use a single element of a parameterised 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")];
```

3. Design Patterns

We present five TinyOS design patterns: two behavioural (relating to component interaction): Dispatcher and Decorator, and three structural (relating to how applications are structured): Service Instance, Placeholder and Facade. A more in-depth presentation of these and other patterns can be found on our website [5]; in particular we do not include any namespace (management of identifiers such as message types) patterns here. We follow the basic format used in *Design Patterns* [3], 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. This diagram follows the same format as Figure 1, with the addition of a folded sub-box for showing source code (a floating folded box represents source code in some other, unnamed, component). *Sample Code* shows an example nesC implementation; *Consequences* describes how the pattern achieves its goals, and notes issues to consider when using it. *Related Patterns* compares to other relevant patterns.

3.1 Behavioural: Dispatcher

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.

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 the application responds to those it cares 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 secondly 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`. But this is very inflexible: any change to the protocols used in an application requires a change in a system component.

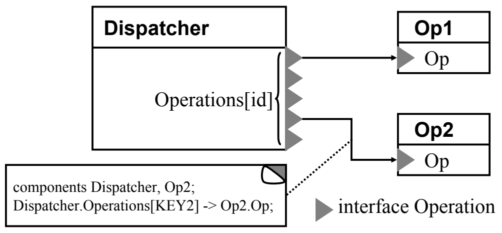
A better approach in TinyOS is to use the Dispatcher pattern. A Dispatcher invokes operations using a parameterised interface, based on a data identifier. In the case of `AMStandard`, the interface is `ReceiveMsg` and the identifier is the active message type field. `AMStandard` is independent of what 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 parameterised `MateBytecode` interface. The instruction set can be easily changed by altering the wiring of the scheduler.

Applicable When:

- A component needs to support an externally customisable 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.

Structure



components Dispatcher, Op2;
Dispatcher.Operations[KEY2] -> Op2.Op;

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];
}
```

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. 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 cause problems.

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 re-used. 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 run-time. An implementor forgetting a run-time 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.

Related Patterns:

- Service Instance: a service instance creates many instances of an implementation of an interface, while a dispatcher selects between different implementations of an interface.
- Placeholder: a placeholder allows an application to select an implementation at compile-time, while a dispatcher allows it to select an implementation at runtime.

3.2 Structural: Service Instance

Intent: Allows multiple users to have separate instances of a particular service, where the instances can collaborate efficiently.

Motivation: Sometimes many components or subsystems need to use a system abstraction, but each user wants a separate instance of that service. We don't 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 reduces CPU load on the system and can allow it to sleep longer, saving energy.

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 requires either sharing state across objects, which is contrary to encapsulation, or state copying, which uses additional RAM.

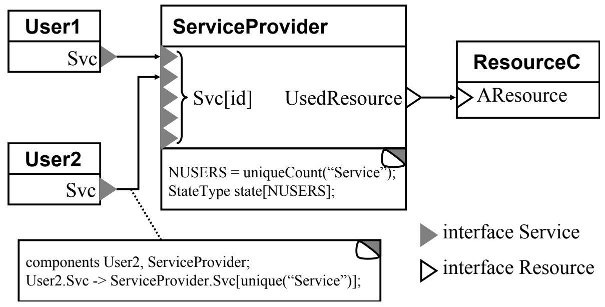
Implementing each timer in a separate module leads to duplicated code and requires inter-module coordination in order to figure out how to set the underlying hardware clock. Just setting it at a fixed rate and maintaining a counter for each Timer is inefficient: timer fidelity requires firing at a high rate, but it's pointless to fire at 1KHz if the next timer is in four 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 parameterised 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 how many instances exist using the `uniqueCount` function and dimension the state array accordingly.

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.

Structure



[5] This restriction will be lifted in the next version of nesC (Section 4.4).

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")];
```

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 run-time failures due to 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 used rarely, 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 run-time (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.

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.

3.3 Structural: Placeholder

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

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.

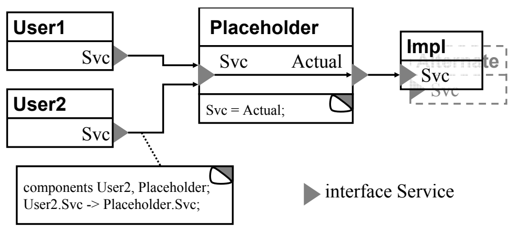
One option is for every implementation to use the same component name, and put them in separate directories. Manipulating the nesC search order allows an application to select which version to use. This approach doesn't scale well: each implementation of each component needs a separate directory. Streamlining 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.

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.

Structure



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;
  ...
}
```

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 optimises away the added level of wiring indirection, so a Placeholder imposes no run-time overhead. The Placeholder supports flexible composition and simplifies use of alternative service implementations.

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.

3.4 Structural: Facade

Intent: Provides a unified access point to a set of inter-related services and interfaces. Simplifies use, inclusion, and composition of the subservices.

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.

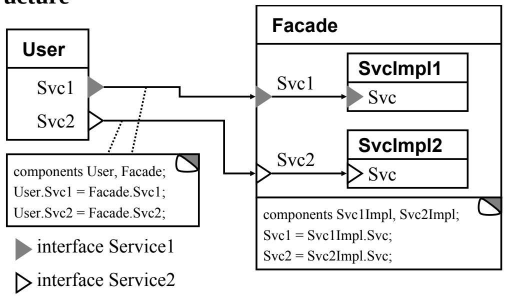
One option would be to put all of the operations in a single, shared interface. This raises two problems. First, the nesC wiring rules mean that a component that wants to use *any* command in the interface has to handle *all* of its events. In the case of a file system, all the operations are split-phase; 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, etc.). 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. Additionally, 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. Additionally, 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. [3] The distinction lies in nesC's static model. An object-oriented Facade instantiates its subcomponents at run-time, 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 run time cost.

Structure



```
components User, Facade;
User.Svc1 = Facade.Svc1;
User.Svc2 = Facade.Svc2;
```

```
components Svc1Impl, Svc2Impl;
Svc1 = Svc1Impl.Svc;
Svc2 = Svc2Impl.Svc;
```

Applicable When:

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

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 non-volatile storage; `Matchbox.nc` 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
}
```

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 support, 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 sub-parts, 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.

Several stable, commonly used abstract boundaries have emerged in TinyOS [2], such as GenericComm (the network stack) and Matchbox (a file system), The presentation of these APIs is almost always a Facade.

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.

3.5 Behavioural: Decorator

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.

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.

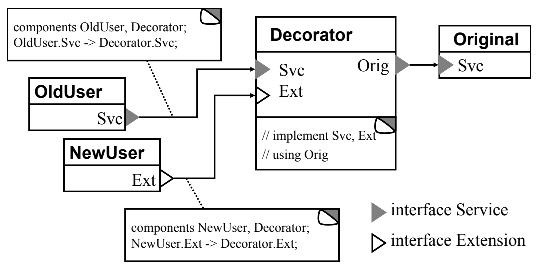
There are two traditional object-oriented approaches to this problem: inheritance, which defines the relationship at compile time through a class hierarchy, and decorators, which define the relationship at run time through encapsulation. [3] As nesC is not an object-oriented language, and has no notion of inheritance, the former option is not possible. Similarly, run-time 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.

Structure



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.

Sample Code: The standard `LogData` interface includes split-phase `erase`, `append` and `sync` operations. `BufferedLog` adds buffering to the `LogData` operations, and, additionally, supports a `FastLogData` interface with a non-split-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();
  }
```

Consequences: Applying a Decorator allows you to extend or modify a component's behaviour though a separate component: the original implementation can remain unchanged. Additionally, 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 3.3) 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.

Decorators are a lightweight but flexible way to extend component functionality. Interpositioning is a common technique in building networking stacks [12], and Decorators enable this style of composition.

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. Finally, we discuss how these patterns have influenced the design of the nesC programming language and how upcoming changes to nesC will address some of the limitations of our current patterns.

4.1 Comparison to object-oriented patterns

The five design patterns described in Section 3 can be separated into classes: Dispatcher and Service Instance are specific to nesC, while Decorator, Facade and Placeholder have analogues in existing pattern [3]. The differences from traditional object-oriented patterns stem from the design principles behind TinyOS [1]. 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 run-time if one allocator misbehaves. As a result, where many object-oriented patterns increase object flexibility and reusability by allowing behaviour 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 TinyOS Facade and Decorator patterns have similar goals and structures to their identically-named object-oriented analogues [3, p.175,p.185]. The Facade assembles a set of existing components and presents them as a single component to simplify use, while the Decorator adds extra functionality to an existing component. 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; additionally, 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). The Decorator is more useful than in an object-oriented context, as it provides a way to define implementation-inheritance hierarchies in a component-based language. However, the use of any given Decorator is limited by the singleton nature of components. Finally, Placeholder has similarities to the Bridge [3, 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.
- A Decorator supports diverse implementations by enabling lightweight component extension.
- Service Instance and Dispatcher increase robustness and lower resource usage by resolving component interactions at compile-time.
- A Dispatcher improves application adaptability by providing a way to easily configure what operations an application supports and how it reacts to its environment.

	inlining + dead-code	dead-code only	unoptimised
Power draw	5.1mW	9.5mW	
Interpreter size	47.1kB	52.5kB	83.8kB

Table 1. Effect of optimisations on code size and power draw in a pattern-intensive program.

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 then discuss how its features combined with the nesC language design supports design patterns.

The nesC compiler generates a single C file containing the executable code of all the modules of the program, where the wirings specified by configurations become direct function calls. The explicit specification of the program's call graph via wiring and parameterised interfaces make it easy to perform two whole program optimisations. First, unreachable functions are eliminated from the output C file. Second, the compiler adds inline directives to all "small" functions (where the size of a function includes the size of all functions it calls) and to all functions called exactly once. We rely on the C compiler which processes the generated C file to perform the actual inlining.[6]

Concise and efficient expression of our patterns is made possible by the following features:

- Inlining makes it possible to break programs into lots of components without a large performance cost (Dispatcher, Placeholder, Facade, Decorator).
- Dead code elimination removes unused functionality, allowing more general components to be designed (Facade).
- Parameterised interfaces allow runtime dispatches (Dispatcher, Service Instance).
- Unique identifiers support compile-time configuration of services, e.g., to identify clients (Service Instance).

To show the importance of nesC's optimisations in real programs, we evaluated the code size and average power draw of a bytecoded interpreter for a Scheme-like language, running on mica2 [13] motes. The interpreter, built with the Maté virtual machine architecture [14], makes heavy use of all these patterns. The mica2 motes have an Atmel ATmega128 microcontroller, running at 8MHz, with 128kB of flash and 4kB of RAM. We ran a simple program that performed a little computation 10 times a second, and sends a radio message with the results every 50s.

Table 1 shows the power draw of this program, and interpreter code size with and without the inlining and dead-code optimisations. Without optimisation, power increases by 86%[7] and code size by 78%. We can thus see that these optimisations, enabled in part by nesC's language features, are essential to performance in programs which make heavy use of these patterns. In contrast, a program that spends most of its time in the radio stack — a monolithic subsystem making little use of patterns — sees "only" a 7% increase (1.22mW to 1.31mW) in power draw when turning off inlining.

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

[6] All our current platforms use gcc. We ensure that it inlines all requested functions by passing it an -finline-limit=100000 option.

[7] Dead-code elimination has no effect on power draw.

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 next version of nesC, 1.2, will introduce 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 behaviour (the keys may no longer be unique).

The upcoming 1.2 version 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;
```

Generic modules make patterns such as Decorator much more reusable, and allow patterns such as Facade to have private components, whose interfaces are only accessible through what a configuration exposes. 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 analogues of many of the structural patterns from the original Design Patterns book [3] can be expressed in nesC, with a "component = class", or "component = object" mapping. Translations of behavioural patterns is harder, reflecting the differences in resources and application domains. The fact that our list contains relatively few behavioural patterns (just Dispatcher and Decorator) 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.

Acknowledgements

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

[1] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "TinyOS: An operating system for wireless sensor networks," in *Ambient Intelligence*. New York, NY: Springer-Verlag, To Appear.

[2] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler, "The Emergence of Networking Abstractions and Techniques in TinyOS," in *First USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, 2004.

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

[4] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," in *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, June 2003.

[5] P. Levis and D. Gay, "Tinyos design patterns," http://www.cs.berkeley.edu/~pal/tinyos-patterns, 2004.

[6] *OOPSLA Workshop Towards Patterns and Pattern Languages for OO Distributed Real-time and Embedded Systems*, 2001.

[7] *OOPSLA Workshop on Patterns in Distributed Real-time and Embedded Systems*, 2002.

[8] *PLOP Workshop on Patterns and Pattern Languages in Distributed Real-time and Embedded Systems*, 2002.

[9] B. P. Douglass, *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Addison-Wesley, 2002.

[10] L. Girod, J. Elson, and A. Cerpa, "Em*: a Software Environment for Developing and Deploying Wireless Sensor Networks," in *Proceedings of the USENIX General Track*, 2004.

[11] B. Greenstein, E. Kohler, and D. Estrin, "A Sensor Network Application Construction Kit (SNACK)," in *Proceedings of the 2nd International Conference on Embedded Sensor Systems (SENSYS'04)*, Nov. 2004, pp. 69–80.

[12] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click modular router," *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 263–297, August 2000.

[13] University of California, Berkeley, "Mica2 schematics," http://webs.cs.berkeley.edu/tos/hardware/design/ORCAD_FILES/MICA2/6310-0306-01ACLEAN.pdf, Mar. 2003.

[14] P. Levis, D. Gay, and D. Culler, "Active Sensor Networks," in *Proceedings of the 2nd USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, May 2005.

[8] The device patterns in EM$\star$ [10] may provide inspiration here.