Software Design Patterns for TinyOS

UCB//CSD-04-1350

David Gay
Intel Research Berkeley
2150 Shattuck Avenue
Berkeley, CA 94704
Email: dgay@intel-research.net

Philip Levis
EECS Department
University of California, Berkeley
Berkeley, CA 94720
Email: pal@cs.berkeley.edu

David Culler
EECS Department
University of California, Berkeley
Berkeley, CA 94720
Email: culler@cs.berkeley.edu

***Abstract*— We present design patterns used by software components in the TinyOS operating system. They differ significantly from traditional software design patterns due to TinyOS's focus on static allocation and whole-program composition. We describe how nesC has evolved to support design patterns by including a few simple language primitives.**

I. Introduction

Code re-use is a basic technique of software development. As a program grows in terms of lines of code, its complexity grows non-linearly. Currently, mote-class sensor network programs are still small: TinyDB, one of the largest TinyOS applications, has approximately 4400 nesC statements. A small group of developers can effectively manage programs of this size, writing everything from scratch. As sensor network applications become more intricate and grow in complexity, however, this approach will not be feasible.

Writing solid, reusable software is hard. Doing so for sensor networks is even harder. Limited resources and strict energy budgets 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. TinyOS itself is a library of components; to be useful it must be usable in as many applications as possible.

Software design patterns are a well-accepted technique to promote code re-use [1, 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. Unfortunately, most design patterns focus on the problems faced by large, object-oriented programs: the challenges sensor network programs face are quite different.

In the case of the TinyOS operating system, its programming model can further complicate writing reusable code. In order to support highly concurrent operation with very limited resources, TinyOS depends on split-phase operations, which are a significant departure from C programming with synchronous calls. Although TinyOS components have similarities to objects – they encapsulate state and interact through well-defined interfaces – the differences generally preclude applying object-oriented techniques. To improve program analysis, optimisation, and reliability, TinyOS wiring defines component interactions at compile time [2], rather than at run-time, as object-oriented references and instantiation do. Programmers cannot easily apply idioms or patterns they are comfortable with, and when they do, the results are rarely effective.

In this paper, we present a preliminary set of six design patterns appropriate to TinyOS's component-based model. 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, TinyOS's programming language. 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 IV).

This paper contributes to sensor network research in three ways. First, by presenting a set of TinyOS design patterns, it helps researchers working with TinyOS write effective programs. Second, these design patterns provide insight on how programming sensor networks is structurally different than traditional software, and how different factors motivate software design. Finally, it explores how a few simple language features, particularly parameterised interfaces and unique identifiers, enable scalable, flexible, and efficient software that promotes re-use. 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 analyse and distill TinyOS programming.

Although prior work has explored object oriented design patterns for embedded and real-time devices [3]–[7], 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.

Section II provides background on the nesC language. Section III presents six TinyOS design patterns, describing their motivation, consequences, and representation in nesC, as well as listing several TinyOS components that use them. Section IV discusses the patterns in the light of nesC and TinyOS development, and Section V concludes.

II. 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 III.

nesC [2] is a C-based language with two distinguishing features: a programming model where components interact via interfaces, and a concurrency model based on run-to-completion tasks and interrupt handlers. The concurrency model precludes interfaces from having 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. To promote reliability and analysability, nesC does not support dynamic memory allocation or function pointers and requires that all component interactions be specified at compile-time.

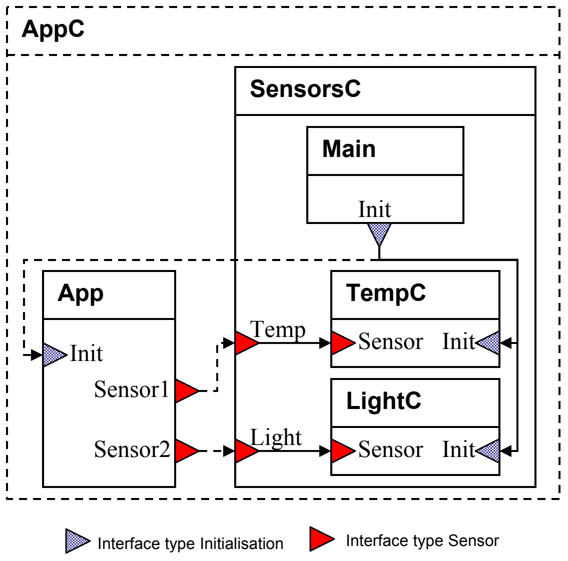


Fig. 1. Sample Component Assembly. Solid rectangles are modules, open rectangles are configurations. Triangles pointing into a rectangle are provided interfaces, triangles pointing out are used interfaces. Dotted lines are "wires" added by configuration `AppC`, full lines are "wires" added by configuration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

A. 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`, `Light`, `Temp`, and `App` are modules while `AppC` and `SensorsC` are configurations. The example shows that configuration `AppC` "wires" (i.e., connects) `App`'s `Sensor1` interface to `SensorsC`'s `Light` interface, etc.

Modules and configurations have a name, specification and implementation:

```
module App {
  provides interface Initialise as Init;
  uses interface Sense as Sensor1;
  uses interface Sense as Sensor2;
}
implementation { ... }
```

declares that `App` (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

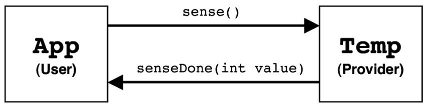


Fig. 2. Typical split-phase operation.

unique, singleton component[1]: 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 {
  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. Figure 2 shows this relationship for `App` and `Temp`. 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.

Interface type `Initialise` represents component initialisation: providers must implement an `init` function. 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.

Following these rules, `App` must implement the `init` command of interface `Init` and the `sensorRead` events of interfaces `Sensor1` and `Sensor2`. `App` is a module, so these functions are implemented as slightly-extended C code:

```
module App { ... }
implementation {
  int sum = 0;
  command void Init.init() {
    call Sensor1.readSensor();
  }
  event void Sensor1.sensorRead(int val) {
    sum += val;
    call Sensor2.readSensor();
  }
  event void Sensor2.sensorRead(int val) {
    sum += val;
  }
}
```

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

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 are private. In App, `sum` variable is used to sum up the values of two sensors connected to the `Sensor1` and `Sensor2` interfaces.

B. 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, LightC, TempC;

  Main.Init -> LightC.Init;
  Main.Init -> TempC.Init;

  Light = LightC.Sensor;
  Temp = TempC.Sensor;
}
```

`SensorsC` “assembles” components `LightC` and `TempC` into a single component providing an interface for each sensor. Interface `Temp` provided by `SensorsC` is *equated* to `TempC`’s `Sensor` interface, `Light` with `LightC`’s `Sensor` interface. Additionally, `SensorsC` *wires* the system’s initialisation interface (`Main.Init`) to the initialisation interfaces of `LightC` and `TempC`.

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

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

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

In this application, interface `Main.Init` is *multiply wired*. `AppC` connects it to `App.Init`, while `SensorsC` connects it to `Light.Init` and `Temp.Init`. The call `Init.init()` in module `Main` compiles to an invocation of all three `init()` commands.[2]

C. Parameterised Interfaces

A parameterised interface is an interface array, where the array indices are wiring parameters. 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];
} ...
```

In a module, interface parameters appear as extra arguments:

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

A configuration can select and wire a single interface by providing arguments to the parameterised interface:

```
configuration ExampleC {
}
implementation {
  components Main, Example;
  components Temp, Light;

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

The system's initialisation interface is connected to `Example`'s `Inits[42]` interface. As a result, at boot time, `Example`'s `Inits.init` command will be called with `id = 42`. This will cause `Example` to call `Sensor[42]`, which connects to the `Temp` component.

A configuration can wire or equate a parameterised interface to another parameterised interface if they have the same parameter types:

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

This equates `Example.Sensors[`i`]` to `ADC[`i`]` for all values of i.

D. `unique`

In many cases, a programmer wants to use a single element of 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:

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

All uses of `unique` with the same argument string (which must be 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]If a multiply wired function has non-void result, nesC combines the results via a programmer-specified function. [2]

III. Design Patterns

We present six TinyOS design patterns: Dispatcher, Service Instance, Keysets, Placeholder, Facade, and Decorator. We follow the basic format used in *Design Patterns* [1], 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. A more in-depth presentation of these and other patterns can be found on our website [8].

Many of the differences between the design patterns we present below and traditional object-oriented patterns stem from the design principles behind TinyOS [9]. 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 OO patterns increase object flexibility and reusability by allowing behaviour changes at runtime, our patterns require that most such decisions be taken by compile-time (e.g., the Dispatcher pattern requires that all possible targets be specified by wirings).

A. *Dispatcher*

Intent: Dynamically select between a set of behaviours based on an identifier. Provides a way to implement and include behaviours independently of the component that uses them.

Motivation: Sometimes our application needs to perform one of a set of operations in response to input. The details of the operation are not important to the invoking component. Additionally, we need to be able to easily extend the set of supported operations. For example, a node can receive many kinds of Active Messages (packets), some of which it must respond to. `AMStandard`, the networking stack component that signals the arrival of a packets should not need to know what processing, if any, an application performs. Other examples include abstractions such as multi-channel ADCs or simple command interpreters.

One way to meet this requirement is with a series of conditionals or a switch statement: ”if the data is of type *T1*, do this; else if it is of type *T2*, do that.” However, this places the implementation of the operations in `AMStandard`.

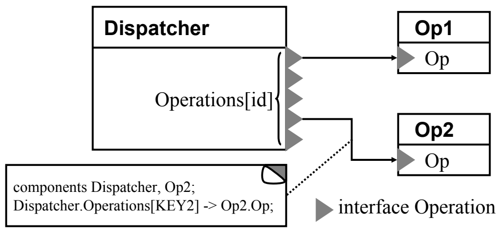
An interface can decouple implementations from operation choice. Instead of embedding the implementation in its conditional statement, `AMStandard` could invoke one of a set of interfaces: ”if the data is of type *T1*, invoke interface *I1*; else if it is of type *T2*, invoke interface *I2*. However, this couples the set of supported operations to the implementation of the network stack: adding a new operation requires modifying `AMStandard`.

A more flexible approach is for `AMStandard` to be a Dispatcher and invoke operations using a parameterised interface, based on the received message's type. This makes the set of supported operations independent of `AMStandard`'s implementation. `AMStandard` doesn't need to know what message types the application processes, or what processing it performs (often, none).

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


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

Sample Code: In this example, the AddXY configuration adds OperationX and OperationY to the set of operations a Dispatcher supports:

```
configuration Dispatcher {
  uses interface Operation as Op[uint8_t id];
}
configuration AddXY {}
implementation {
  components Dispatcher, OperationX, OperationY;
  Dispatcher.Op[OP_X] -> OperationX.Op;
  Dispatcher.Op[OP_Y] -> OperationY.Op;
}
```

Consequences: By leaving operation selection to nesC wirings, the dispatcher component'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 inadvertent fanout on operation calls.

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 to be re-used. Because each operation is implemented in a component, it can be easily re-used across many applications.
- 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.

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

B. Service Instance

Intent: Allows multiple users to have separate instances of a particular service, managing state and enabling instances to collaborate efficiently.

Motivation: Sometimes we need several independent instances of a system service, and don't know precisely how many until we build a complete application. Each instance requires maintaining some state, but the service implementation needs to access the state of every instance. We want to reserve state for all instances at compile time.

For example, TinyOS programs need a wide range of 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 energy-efficient implementation will minimise the interrupt handling rate. This requires knowing when the next timer is due to fire.

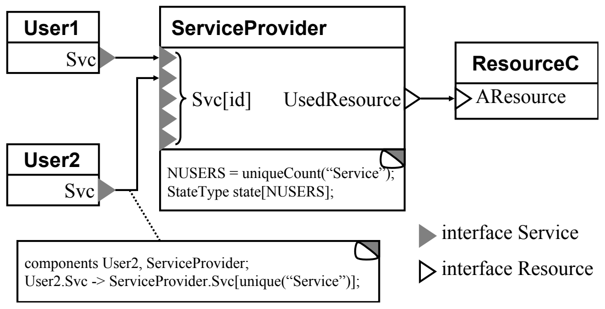
Implementing each timer as a separate component requires redundant code and a lot of component communication to schedule the underlying hardware clock. Allocating a constant number of timers in a single component is neither efficient nor robust. If the system needs fewer, it wastes RAM. If the system needs more, detection of the mismatch only happens at run-time, when the implementation refuses a request.

The Service Instance pattern provides a solution to this problem. 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 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.
- Each service instance appears to its user to be independent of the others.
- The service provider needs to be able to easily access the state of every instance.

Structure



Sample Code: TimerC wires TimerM, the ServiceProvider, to an underlying clock and exports its Timer interfaces. TimerM uses `uniqueCount` to determine how many timers to allocate and maps to them using unique IDs:

```
module TimerM {
  provides interface Timer[uint8_t id];
  uses interface Clock;
}
implementation {
  enum {
    NUM_TIMERS = uniqueCount("Timer")
  }
  timer_t timers[NUM_TIMERS];

  command result_t Timer.start[uint8_t id](...){}
}
```

Clients wanting a timer wire using unique:

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

Consequences: By using `unique` and `uniqueCount`, a ServiceProvider can allocate state according to the number of users, and easily access the state of each user. However, the running time of some operations may depend on the number of users, so execution costs can't be known until an application is compiled.

If many users require an instance, but use them rarely, then allocating state for each one can be wasteful. Another option is to allocate fewer instances and dynamically give them to users. This can conserve total RAM, but requires more RAM per instance (for user IDs), imposes a CPU overhead (for 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.

C. *Keysets*

Intent: Provide namespaces for referring to protocols, structures, or other entities in a program. Allow efficient mapping between different namespaces.

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 III-B) is an example of a local namespace. 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 III-A) is an example of a global namespace. Finally, mapping between namespaces is often useful: this allows motes to communicate, but also control how keys are internally represented.

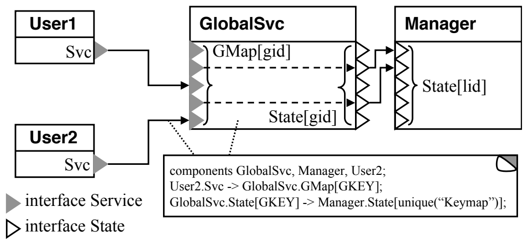
The Keyset patterns provide solutions to these problems. Using these patterns, programs can refer to elements using identifiers optimised for their particular use. Components using the Keyset patterns often take advantage of a parameterised interface, in which the parameter is an element in a Keyset. Local Keysets are designed for referring to local structures and are generated with `unique`; Global Keysets are designed for communication and use global constants; Keymaps can map between different keysets.

The Drip management protocol uses the Keyset patterns to allow a user to configure parameters at run-time. 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.

Applicable When:

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

Structure



```
components GlobalSvc, Manager, User2;
User2.Svc -> GlobalSvc.GMap[GKEY];
GlobalSvc.State[GKEY] -> Manager.State[unique("Keymap")];
```

Sample Code:

The `DripC` component provides a parameterised 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 III-B), 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 Apps'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.

Consequences:

Keysets allow a component to refer to data items or types through a parameterised interface. In a Local Keyset, `unique` ensures that every element has a unique identifier. Global Keysets can also have unique identifiers, but require external namespace management. A Keymap uses nesC wiring to allow components to transparently map between different keysets.

As Local Keysets are generated with `unique`, mapping names to keys (e.g., for debugging purposes) can be difficult. The nesC constant generator, `ncg`, can be useful in this regard.

D. Placeholder

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

Motivation: Often, a service has several variant implementations. For example, there are many ad-hoc collection routing algorithms implemented in TinyOS, but they all expose the same interface. This allows applications to work on top of different routers without internal changes. Another example is a debugging component: during development, you want to log state, but in deployment you want to disable logging to conserve resources.

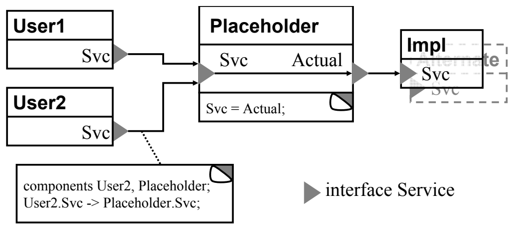
We need the decision of which implementation to use to be system-wide. Otherwise, components in the application might specify different ones, which can conflict or waste resources. However, every configuration that wires to a service names it; changing the implementation in a large application could require changing many files.

The Placeholder pattern offers a solution. A placeholder configuration represents the desired service through a level of 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 by editing a single file. As the level of indirection is solely in terms of names – there is no additional code – 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 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;
    interface SendMsg as Send;
  }
  uses {
    interface StdControl as ActualSC;
    interface SendMsg as ActualSend;
  }
}
implementation {
  SC = ActualSC;
  Send = ActualSend;
}
```

Every component that uses collection routing wires to CollectionRouter:

```
configuration Sensing {  ... }
implementation {
  components SensingM, CollectionRouter;

  SensingM.Send = CollectionRouter.Send;
  ...
}
```

and the application must globally select its routing component:

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

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

Consequences: By adding a level of naming indirection, a Placeholder provides a single point at which you can choose an implementation. 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.

E. Facade

Intent: Provides a unified interface to a set of interrelated services. Simplifies use and inclusion 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, and depend on common underlying services such as reading blocks.

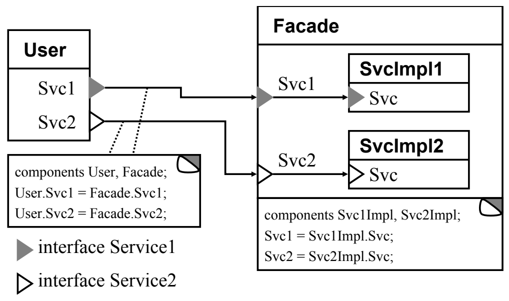
Exporting each interface with a separate component (e.g., MatchboxRead, MatchboxWrite, MatchboxRename, etc.) makes the abstraction more difficult to use. Instead of wiring a single component, an application needs to include several, and wire to each of them. Additionally, each interface would need a 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 uniform access point to interfaces provided by many components. A Facade is a nesC configuration that defines a coherent abstraction boundary by exporting interfaces of several underlying components. Additionally, the Facade can wire its underlying components, simplifying dependency resolution.

Applicable When:

- An abstraction is implemented across several separate components.
- It is preferable to present the abstraction in whole rather than in parts.

Structure



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 FileDelete;
    interface FileDir;
    interface FileRead[uint8_t fd];
    interface FileRename;
    interface FileWrite[uint8_t fd];
  }
}
implementation {
  components Read, Write, Dir, Rename, Delete;
  FileDelete = Delete.FileDelete;
  FileDir = Dir.FileDir;
  FileRead = Read.FileRead;
  FileRename = Rename.FileRename;
  FileWrite = Write.FileWrite;
}
```

Consequences: The Facade 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.

Because the Facade names all of its sub-parts, they will all be included in the nesC component graph. The nesC compiler removes unreachable code, but if any component handles interrupts, then code in the interrupt paths will be included, as will any tasks that those interrupts post. 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 [10]. The presentation of these APIs is almost always a Facade.

F. Decorator

Intent: Modify existing, or add additional responsibilities to a component without modifying its original implementation.

Motivation: We often need to add extra functionality to an existing component, or to modify the way it works without changing its interfaces. A traditional object-oriented approach to this problem is to use inheritance, making the new version a subclass. Component composition is the technique nesC supports to achieve this result.

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 write buffer on top of the interface, to reduce the number of actual writes to the EEPROM.

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. Additionally, it is possible that several implementers of the interface may benefit from the added functionality: having multiple copies, spread across several services, also replicates code.

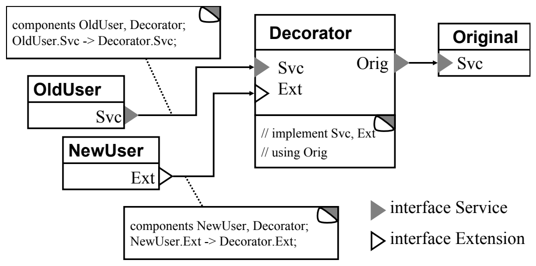
The Decorator pattern provides a better solution. A Decorator component is typically a module that provides and uses the same interface type, such as `LogData`. Its provided interface adds functionality on top of the used interface. For example, the `BufferedLog` component sits on top of a `LogData` provider, and provides the buffering. Additionally, it provides a non-split-phase `FastLog` interface for faster operation.

Using a Decorator has additional benefits. Separating the added functionality from a particular implementation allows it to apply to any implementation. For example, a packet send queue Decorator can be interposed on top of any ad-hoc routing implementation.

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.

Structure



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 {
  char buffer1[BUFSIZE], buffer2[BUFSIZE];
  char *buffer;
  command result_t FastLog.append(data, n) {
     if (buffer_full) {
       call UnbufferedLog.append(buffer, offset);
       // ... switch to other buffer ...
     }
     // ... append to buffer ...
  }
```

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 III-D) can ensure this.

Additional interfaces are likely to use the underlying component, so there will likely be 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 [11], and Decorators enable this style of composition.

IV. Discussion

The six design patterns described in Section III can be separated into classes: Dispatcher, Keyset and Service Instance are specific to nesC, while Decorator, Facade and Placeholder have analogues in existing pattern [1].

The nesC-specific patterns represent ways to make nesC's static programming model – with no dynamic memory allocation, objects or function pointers – more practical by increasing component flexibility. 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). Keysets are a direct result of parameterised interfaces (and are used in support of Dispatcher and Service Instance), while Keymaps enable components with different namespaces to interoperate, so a component can be reused in many circumstances.

The TinyOS Facade and Decorator patterns have similar goals and structures to their identically-named object-oriented analogues [1, 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, the singleton nature of components means that the internals of a Facade cannot be truly private. In the case of the Decorator, its compile-time binding is a way to define inheritance hierarchies, but the singleton nature of components limits the use of any given Decorator. Finally, Placeholder has similarities to the Bridge [1, p.151]: it simplifies implementation switching, but requires that the implementation selection be performed at compile-time. Section IV-B discusses how future nesC 1.2 features extend these patterns and address some of their limitations.

A. *Patterns support TinyOS's goals*

The patterns we have presented directly support TinyOS's design goals [10, Section 2.1]: 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, Keysets and Keymap increase robustness and lower resource usage by binding interactions at compile-time.
- A Dispatcher improves application adaptability by providing a way to easily configure what operations an application supports.

B. *nesC, Yesterday and Tomorrow*

As experience in using has TinyOS 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 next version of nesC, 1.2, will introduce the feature of *generic components* to simplify using design patterns.

TinyOS design patterns tend to depend on users wiring components in a particular way. For example, when wiring to a Service Instance, a programmer must know the key to use as the argument to `unique`. Similarly, in the Keymap pattern, a user must wire to the global keyset as well as introduce a wiring for the global to local mapping. 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). Additionally, the singleton model of TinyOS is inherently opposed to the goal of code re-use. If a program needs two copies of a module, such as a data filter Decorator, then two separate modules must exist, and their code must be maintained separately.

nesC 1.2 introduces generic components, which can be instantiated with numerical and type parameters. The basic model of module instantiation is to create a copy of the code. Configurations can instantiate generic components:

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

Generic modules allow a programmer to use copies of a component in many places while maintaining a single implementation: they are similar to compile-time object instantiation. 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();
User1.Timer -> TimerSvc;
```

Generic modules 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.

V. 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 [1] can be expressed in nesC, with a "component = class", or "component = object" mapping. The relative lack of behavioural patterns (just Dispatcher) in our list 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)[3], 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] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patters: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[2] 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.

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

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

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

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

[7] 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*.

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

[9] 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.

[10] 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.

[11] 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.

[3]The device patterns in EM⋆ [7] may provide inspiration here.