15,327,303 members

0

articles Q&A forums stuff lounge ?

Search for articles, questions

Watch

State Machines, in C++

HoshiKata

11 Nov 2016 CPOL

Rate me: 3.86/5 (14 votes)

An example state machine framework that uses Doxygen to auto-draw the actual code's behavior.

Introduction

The article tries to show how you can solve the major problem with using statemachines, debugging and testing. Simple statemachines (debouncing a button, managing an SQL transaction) are relatively easy to do in just about any framework. When a statemachine is being used to handle dozens of events and sensors in 10-12 different modes of operation it gets to a point where the individual statemachine components are easy to describe and code but the system as a whole can't quite be pictured. Very often the concurrent behavior of one state vs. the paralell behavior of another statemachine becomes far to complex. Then "fixing" the problem often chases the problem from one state to the next.

For complex statemachines the problem is:

> How do you maintain it / make changes to the code?
>
> How do you prove that the code meets the requirements and was exercised correctly in that scenario?

The solution presented here the author has used on may projects: have the code tell you what it could do, and when run, tell you what it did do, and do it graphically.

Roughly 20 years ago, this kind of solution worked by recording state numbers, and event transitions with timestamps and then automating a graph that described the system. The framework here uses a feature built into Doxygen which allows the code to write (at creation or as a log) text that can describe a statemachine graph. The author has also used SVG, and other sequence diagram from text tools, but felt that for this example ... Doxygen is particularly great, and the graphviz project by AT&T research was groundbreaking.

For embedded systems the code presented here can have the strings stripped out when not testing or debugging to cut down on the overhead, or they can be replaced with 4 letter symbols (that can be logged as 32 bit numbers).

Background

State machines are a good alternative to threads for complex problems because: they split behaviors into groups by state; allow re-entrance at numerous steps that would be less practical with a thread; and they can transition to different and even concurrent sets of behaviors as needed.

In practice software engineering is largely about what is practical in terms of achieving quality goals quickly with maintainable code that can be documented and tested cheaply. Most frameworks in recent years seem to focus on making design, and framework adoption easier by allowing users to draw states and then generate the code automatically.

The author feels that simplifies the easy and not very expensive part of development. There are plenty of tools for sketching state diagrams, and software engineers know how to write code. I use a napkin or scrap paper, then I spend a few minutes to write it, and an initial hour debugging it, then a few days trying to figure out why it doesn't work when combined with this or that corner case. Maintaining, fixing, documenting, and testing software is often more than half the total cost.

The author has done a fair bit of both Desktop and Embedded development. Statemachines are often the best solution when you can't have the overhead of a thread, or enough threads. They are also ideal when you can segment the behavior of different parts of the system as a state. They are generally the preferred way of doing embedded software, but what is the real advantage? A thread always provides a better narrative of how one thread of execution of a procedure needs to work. In embedded systems often each sensor (of dozens) have a different narrative of a procedure to make a reading or setup the sensor. Having that many threads requires too much overhead in embedded software, it takes up memory and requires context swaps. Seperate threads then lead to mutexes because the code can swap out in various ways, and this is not the case in statemachines because they can run multiple thread like operations sequentially without interruption off of the same thread. Where threads excel in software is providing a narrative in the code that shows the writer / maintainer how the code works very explicitly, but that is kind of useless if there really isn't a narrative, but a few dozen actions that get taken in this mode but not that mode based on button presses. In that case state machines can provide a functional decomponsition methodology to the behaviors that simplifies the code.

The point is, both architectural mind sets (threads, states) make sense, and a balanced decision should be made to use (or not) a statemachine framework. When deciding on the framework there are choices.

> Is a thread better (often it is)?
>
> Is behavior event driven, or are values monitored (differently) by state?
>
> Is action taken while in a state, or on transition, or both?

If a statemachine is the best fit, if most of the logic is discrete event based (UI's tend to fit that category) then event handling should be the central theme. If it is process driven then something that allows entering, exiting and doing while in a state is more useful. The framework here provides both methodologies equally because most problems require some mixture. The framework exists as base classes because the author hopes you consider wrapping the framework / modifying it and identifying use scenarios for your problem. Make it your own.

The framework presented here generates diagrams good enough for testing, debugging from the code itself. The framework is very generic, and should be reworked at least slightly for the specific project's requirements but the code is a good starting place.

The framework has the following features:

- States are instances of classes (many instances of the same logic)
- States are configurable
- State machines can run nested and in parallel
- States can inherit behavior from other states
- States can override behavior for entering, exiting, waiting in a given state
- States can react to events, and nesting super states can handle generalized events for children.
- State diagrams can be what the state machine was coded to do (state diagrams)
- States can log the actual transitions (graphically)
- It can also be modified to compile time remove the strings for embedded software.

This covers most common styles of state machines (things that do their work on transition vs. in the state, and state machines that react to events or monitor abstract sets of data). It is meant to be a starting point for rapid modification for a specific product. It leverages the DOT language to generate graphs, a feature of Doxygen[1] a widely used code documentation tool that parses comments and generates HTML documentation with graphs and images similar to Java's Javadoc or C# XML markup.

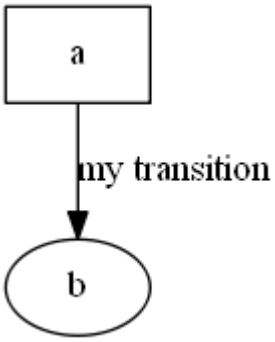
1https://en.wikipedia.org/wiki/DOT_(graph_description_language)

The graphviz tool (dotty) allows Doxygen to generate graphs from text embedded in the code. For example:

Copy Code

```
/**
   digraph mygraphname {
        state_a[label="a" shape=box];
        state_b[label="b"]; // use default shape.
        state_a->state_b[label="my transition"];
   }
*/
```

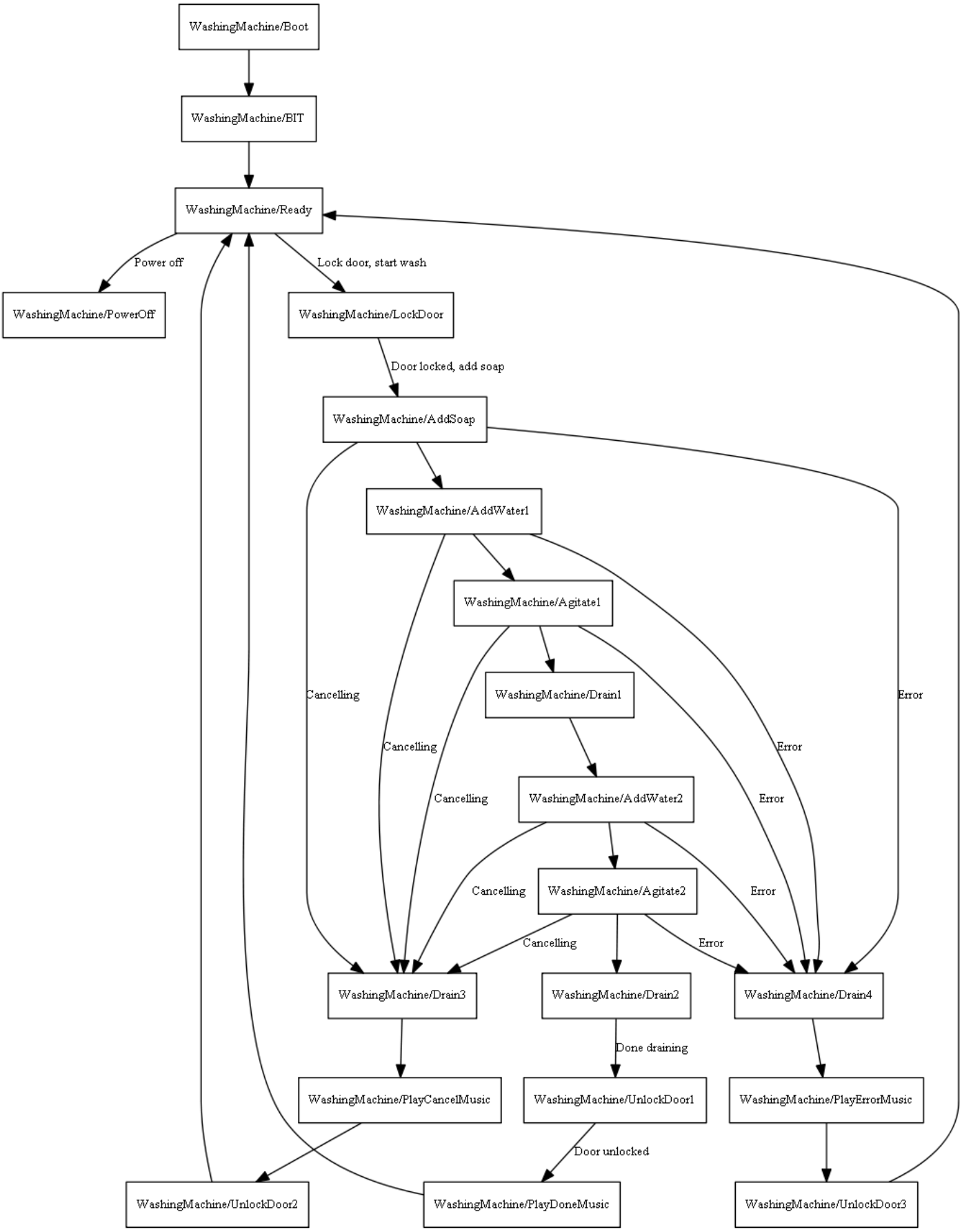
Gets turned into a graph:



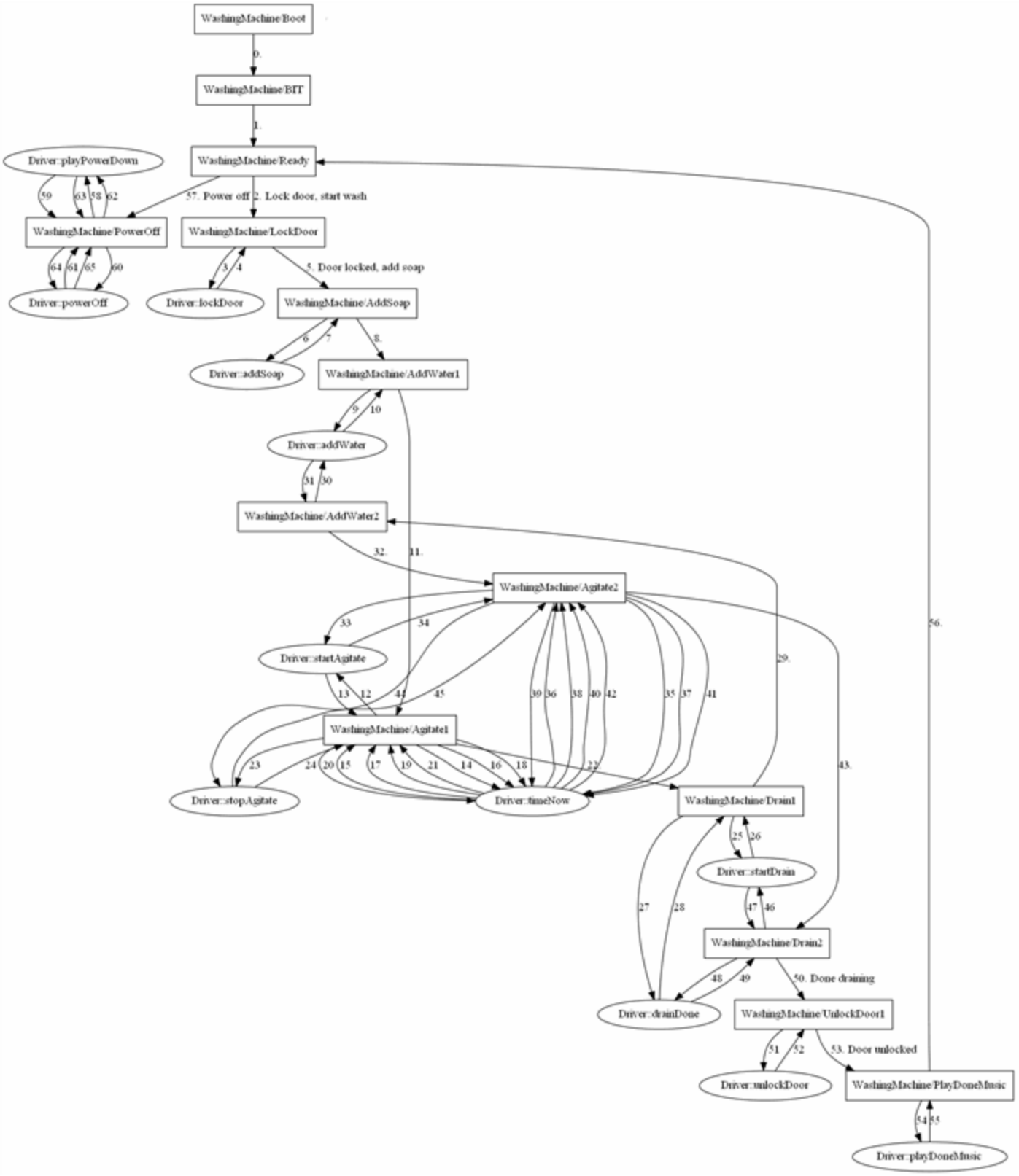
Washing Machine Example

This example describes a simplified washing machine that needs to react to a start button, a cancel button, provide a wash and rinse, and when needed drain and play some sounds.

The following is the generated SM from the washing machine code.



When running it with a few events sent in, a thread of execution test can be done and the results graphed. This map of the states (boxes) and driver calls (ovals).



Using the code

Source code (not diagrams) are parse-able by tools, review-able and something that can be statically analyzed, so while having diagrams is nice, paper is our product. The code must be easy to read, review and write. To do this a few C++ syntax tricks are used.

For the most part, the code is meant to speak for itself. Doxygen of the source code is provided.

A good example of how the code works is how the “drain” state works with the agitate state. There are two instances of drain, one after washing and one after rinsing, so the behavior is defined in the state definition, but the transitions are assigned to the instances. The class defines behavior, but instantiation defines relative order and linkage.

Copy Code

```
// Agitate, wait 10 min to be done, handle cancel and error signals.
DO_STATE(Agitate) { // Macro forms a class w/ a doState() { started.
        // Called while in the state.
        if (gpDriver->timeNow(this) > mTime + TEN_MINUTES_MSEC)
        setState(next());
    }
    void enter() {
        gpDriver->startAgitate(this);
        mTime = gpDriver->timeNow(this);
    }
    void exit() {
        gpDriver->stopAgitate(this);
    void handleSignal(uint32_t signal) {
        if (signal == SIGNAL_CANCEL) setState(cancel());
        if (signal == SIGNAL_ERROR)  setState(error());
    }
}; // Finish declaring the class.
```

It defines a class called *Agitate* that has a doState(), enter(), exit() and handleSignal() calls. On entrance it starts the agitation and notes the time. It then transitions to *next* (what ever that instance assigned as next) when the time expires. The exit function always stops agitation when the state is exited, and there is a signal handler to catch the cancel and error signals and do different target states.

This macro (macros are generally bad) makes the code dramatically smaller and easier to understand by declaring most of the class typing.

Copy Code

```
#define DO_STATE(n) class n: public SequenceState { public: n(const std::string &name,
State *parent=NULL) : SequenceState(name, parent) {} virtual void doState()
```

The instances of these objects are:

Copy Code

```
Agitate * agitate1 = new Agitate("Agitate1", s);
Agitate * agitate2 = new Agitate("Agitate2", s);

...

// Use the instances to get the behavior but assign different
// "next" states for agitate1 and agitate2.

agitate1 << Next(drain1); // Make the "next" state go to the drain1 instance in
agitate1.
```

```
agitate1 << Cancel(drain3, "Cancelling"); // On cancel goto drain3, and logg cancelling.
agitate1 << Error(drain4, "Error");

agitate2 << Next(drain2); // Make the "next" state go to the drain2 instance in
agitate2.
agitate2 << Cancel(drain3, "Cancelling"); // On cancel goto drain3, and logg cancelling.
agitate2 << Error(drain4, "Error");
```

The instances always drain, but these drains then play a different sound (an error sound, a cancel sound or a done sound) before returning to the ready state because structural connection is seperate from the class behavior.

History

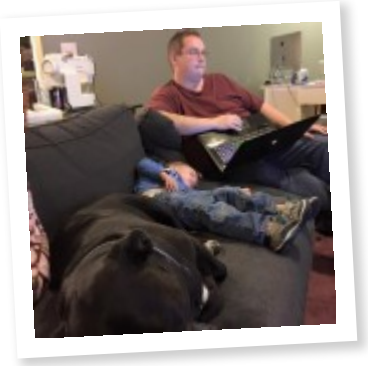
Fixeds some formatting issues, things looked ok in the editor but not in the actual posted HTML.

License

This article, along with any associated source code and files, is licensed under The Code Project Open License (CPOL)

Share

About the Author



HoshiKata

Technical Lead

United States

Watch this Member

Phil is a Principal Software developer focusing on weird yet practical algorithms that run the gamut of embedded and desktop (PID loops, Kalman filters, FFTs, client-server SOAP bindings, ASIC design, communication protocols, game engines, robotics).

In his personal life he is a part time mad scientist, full time dad, and studies small circle jujitsu, plays guitar and piano.

Comments and Discussions

Add a Comment or Question | Email Alerts | Search Comments

First Prev Next

Please don't ...
Stefan_Lang 22-Nov-16 2:06

Re: Please don't ...
Rick York 26-Jan-17 10:45

Re: Please don't ...
HoshiKata 29-Aug-17 12:06

I like this!!! could you provide a demo source example to add to the article.
leestudley 15-Nov-16 18:47

Re: I like this!!! could you provide a demo source example to add to the article.
studleylee 15-Nov-16 22:25

Snipets
Joe Pizzi 14-Nov-16 19:23

Great Article
Mark Kolenski 14-Nov-16 15:18

Refresh 1

General News Suggestion Question Bug Answer Joke Praise Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.

Permalink
Advertise
Privacy
Cookies
Terms of Use

Layout: fixed | fluid

Article Copyright 2016 by HoshiKata
Everything else Copyright © CodeProject, 1999-2022

Web02 2.8.2022.06.07.1