عنوان: **طراحی گردش کاری با استفاده از State machines - قسمت دوم**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۱۰/۱۱ ۲۳:۱۰
آدرس: www.dotnettips.info
گروه‌ها: WorkFlow Foundation, Workflow, State machine

معرفی کتابخانه stateless به عنوان جایگزین سبک وزنی برای Windows workflow foundation

کتابخانه سورس باز Stateless ، برای طراحی و پیاده سازی «ماشین‌های حالت گردش کاری مانند» تهیه شده و مزایای زیر را نسبت به Windows workflow foundation دارا است:

- جمعا 30 کیلوبایت است!
- تمام اجزای آن سورس باز است.
- دارای API روان و ساده‌ای است.
- امکان تبدیل UML state diagrams، به نمونه معادل Stateless بسیار ساده و سریع است.
- به دلیل code first بودن، کار کردن با آن برای برنامه نویس‌ها ساده‌تر بوده و افزودن یا تغییر اجزای آن با کدنویسی به سادگی میسر است.

دریافت کتابخانه Stateless از Google code و یا از NuGet

پیاده سازی مثال کلید برق با Stateless

در ادامه همان مثال ساده کلید برق قسمت قبل را با Stateless پیاده سازی خواهیم کرد:

```
using System;
using Stateless;

namespace StatelessTests
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                string on = "On", off = "Off";
                var space = ' ';

                var onOffSwitch = new StateMachine<string, char>(initialState: off);

                onOffSwitch.Configure(state: off).Permit(trigger: space, destinationState: on);
                onOffSwitch.Configure(state: on).Permit(trigger: space, destinationState: off);

                Console.WriteLine("Press <space> to toggle the switch. Any other key will raise an
error.");

                while (true)
                {
                    Console.WriteLine("Switch is in state: " + onOffSwitch.State);
                    var pressed = Console.ReadKey(true).KeyChar;
                    onOffSwitch.Fire(trigger: pressed);
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine("Exception: " + ex.Message);
                Console.WriteLine("Press any key to continue...");
                Console.ReadKey(true);
            }
        }
    }
}
```

کار با ایجاد یک وهله از ماشین حالت (new StateMachine) آغاز می‌شود. حالت آغازین آن (initialState) مطابق مثال قسمت قبل، مساوی off است.

امضای کلاس StateMachine را در ذیل مشاهده می‌کنید؛ جهت توضیح آرگومان‌های جنریک string و char معرفی شده در مثال:

```
public class StateMachine<TState, TTrigger>
```

که اولی بیانگر نوع حالات قابل تعریف است و دومی نوع رویداد قابل دریافت را مشخص می‌کند.
برای مثال در اینجا حالات روشن و خاموش، با رشته‌های on و off مشخص شده‌اند و رویداد قابل قبول دریافتی، کاراکتر فاصله است.
سپس نیاز است این ماشین حالت را برای معرفی رویدادهایی (trigger در اینجا) که سبب تغییر حالت آن می‌شوند، تنظیم کنیم.
اینکار توسط متدهای Configure و Permit انجام خواهد شد. متد Configure، یکی از حالات از پیش تعیین شده را جهت تنظیم، مشخص می‌کند و سپس در متد Permit تعیین خواهیم کرد که بر اساس رخدادی مشخص (برای مثال در اینجا فشرده شدن کلید space) وضعیت حالت جاری، به وضعیت جدیدی (destinationState) منتقل شود.
نهایتا این ماشین حالت در یک حلقه بی‌نهایت مشغول به کار خواهد شد. برای نمونه یک Thread پس زمینه (BackgroundWorker) نیز می‌تواند همین کار را در برنامه‌های ویندوزی انجام دهد.

یک نکته

علاوه بر روش‌های یاد شده‌ی تشخیص الگوی ماشین حالت که در قسمت قبل بررسی شدند، مورد refactoring انبوهی از if و elseها و یا switchهای بسیار طولانی را نیز می‌توان به این لیست افزود.

استفاده از Stateless Designer برای تولید کدهای ماشین حالت

کتابخانه Stateless دارای یک طراح و Code generator بصری سورس باز است که آن‌را به شکل افزونه‌ای برای VS.NET می‌توانید در سایت Codeplex دریافت کنید . این طراح از کتابخانه GLEE برای رسم گراف استفاده می‌کند.

کار مقدماتی با آن به نحو زیر است:
الف) فایل StatelessDesignerPackage.vsix را از سایت کدپلکس دریافت و نصب کنید. البته نگارش فعلی آن فقط با VS 2012 سازگار است.
ب) ارجاعی را به اسمبلی stateless به پروژه خود اضافه نمائید (به یک پروژه جدید یا از پیش موجود).
ج) از منوی پروژه، گزینه Add new item را انتخاب کرده و سپس در صفحه ظاهر شده، گزینه جدید Stateless state machine را انتخاب و به پروژه اضافه نمائید.
کار با این طراح، با ادیت XML آن شروع می‌شود. برای مثال گردش کاری ارسال و تائید یک مطلب جدید را در بلاگی فرضی، به نحو زیر وارد نمائید:

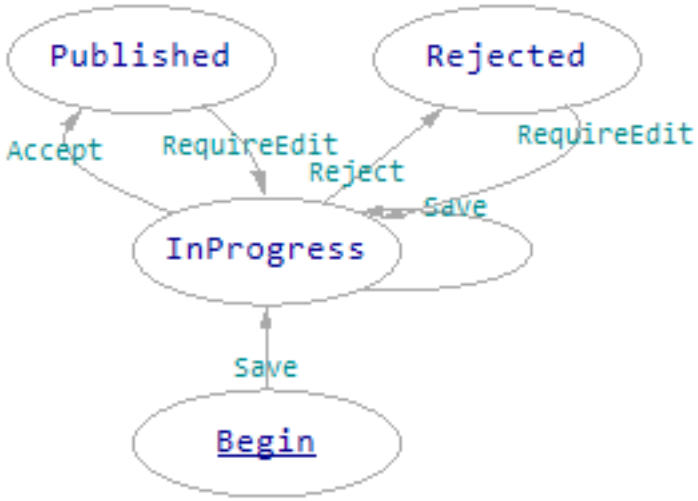
```
<statemachine xmlns="http://statelessdesigner.codeplex.com/Schema">
  <settings>
    <itemname>BlogPostStateMachine</itemname>
    <namespace>StatelessTests</namespace>
    <class>public</class>
  </settings>
  <triggers>
    <trigger>Save</trigger>
    <trigger>RequireEdit</trigger>
    <trigger>Accept</trigger>
    <trigger>Reject</trigger>
  </triggers>
  <states>
    <state start="yes">Begin</state>
    <state>InProgress</state>
    <state>Published</state>
    <state>Rejected</state>
  </states>
  <transitions>
    <transition trigger="Save" from="Begin" to="InProgress" />

    <transition trigger="Accept" from="InProgress" to="Published" />
    <transition trigger="Reject" from="InProgress" to="Rejected" />

    <transition trigger="Save" from="InProgress" to="InProgress" />
```

```
    <transition trigger="RequireEdit" from="Published" to="InProgress" />
    <transition trigger="RequireEdit" from="Rejected" to="InProgress" />
  </transitions>
</statemachine>
```

حاصل آن گراف زیر خواهد بود:



به علاوه کدهای زیر که به صورت خودکار تولید شده‌اند:

```
using Stateless;

namespace StatelessTests
{
  public class BlogPostStateMachine
  {
    public delegate void UnhandledTriggerDelegate(State state, Trigger trigger);
    public delegate void EntryExitDelegate();
    public delegate bool GuardClauseDelegate();

    public enum Trigger
    {
      Save,
      RequireEdit,
      Accept,
      Reject,
    }

    public enum State
    {
      Begin,
      InProgress,
      Published,
      Rejected,
    }

    private readonly StateMachine<State, Trigger> stateMachine = null;

    public EntryExitDelegate OnBeginEntry = null;
    public EntryExitDelegate OnBeginExit = null;
    public EntryExitDelegate OnInProgressEntry = null;
    public EntryExitDelegate OnInProgressExit = null;
    public EntryExitDelegate OnPublishedEntry = null;
    public EntryExitDelegate OnPublishedExit = null;
    public EntryExitDelegate OnRejectedEntry = null;
    public EntryExitDelegate OnRejectedExit = null;
    public GuardClauseDelegate GuardClauseFromBeginToInProgressUsingTriggerSave = null;
```

```
    public GuardClauseDelegate GuardClauseFromInProgressToPublishedUsingTriggerAccept = null;
    public GuardClauseDelegate GuardClauseFromInProgressToRejectedUsingTriggerReject = null;
    public GuardClauseDelegate GuardClauseFromInProgressToInProgressUsingTriggerSave = null;
    public GuardClauseDelegate GuardClauseFromPublishedToInProgressUsingTriggerRequireEdit = null;
    public GuardClauseDelegate GuardClauseFromRejectedToInProgressUsingTriggerRequireEdit = null;
    public UnhandledTriggerDelegate OnUnhandledTrigger = null;

    public BlogPost()
    {
      stateMachine = new StateMachine<State, Trigger>(State.Begin);
      stateMachine.Configure(State.Begin)
        .OnEntry(() => { if (OnBeginEntry != null) OnBeginEntry(); })
        .OnExit(() => { if (OnBeginExit != null) OnBeginExit(); })
        .PermitIf(Trigger.Save, State.InProgress , () => { if
(GuardClauseFromBeginToInProgressUsingTriggerSave != null) return
GuardClauseFromBeginToInProgressUsingTriggerSave(); return true; } )
      ;
      stateMachine.Configure(State.InProgress)
        .OnEntry(() => { if (OnInProgressEntry != null) OnInProgressEntry(); })
        .OnExit(() => { if (OnInProgressExit != null) OnInProgressExit(); })
        .PermitIf(Trigger.Accept, State.Published , () => { if
(GuardClauseFromInProgressToPublishedUsingTriggerAccept != null) return
GuardClauseFromInProgressToPublishedUsingTriggerAccept(); return true; } )
        .PermitIf(Trigger.Reject, State.Rejected , () => { if
(GuardClauseFromInProgressToRejectedUsingTriggerReject != null) return
GuardClauseFromInProgressToRejectedUsingTriggerReject(); return true; } )
        .PermitReentryIf(Trigger.Save , () => { if
(GuardClauseFromInProgressToInProgressUsingTriggerSave != null) return
GuardClauseFromInProgressToInProgressUsingTriggerSave(); return true; } )
      ;
      stateMachine.Configure(State.Published)
        .OnEntry(() => { if (OnPublishedEntry != null) OnPublishedEntry(); })
        .OnExit(() => { if (OnPublishedExit != null) OnPublishedExit(); })
        .PermitIf(Trigger.RequireEdit, State.InProgress , () => { if
(GuardClauseFromPublishedToInProgressUsingTriggerRequireEdit != null) return
GuardClauseFromPublishedToInProgressUsingTriggerRequireEdit(); return true; } )
      ;
      stateMachine.Configure(State.Rejected)
        .OnEntry(() => { if (OnRejectedEntry != null) OnRejectedEntry(); })
        .OnExit(() => { if (OnRejectedExit != null) OnRejectedExit(); })
        .PermitIf(Trigger.RequireEdit, State.InProgress , () => { if
(GuardClauseFromRejectedToInProgressUsingTriggerRequireEdit != null) return
GuardClauseFromRejectedToInProgressUsingTriggerRequireEdit(); return true; } )
      ;
      stateMachine.OnUnhandledTrigger((state, trigger) => { if (OnUnhandledTrigger != null)
OnUnhandledTrigger(state, trigger); });
    }

    public bool TryFireTrigger(Trigger trigger)
    {
      if (!stateMachine.CanFire(trigger))
      {
        return false;
      }
      stateMachine.Fire(trigger);
      return true;
    }

    public State GetState
    {
      get
      {
        return stateMachine.State;
      }
    }
  }
}
```

توضیحات:

ماشین حالت فوق دارای چهار حالت شروع، در حال بررسی، منتشر شده و رد شده است. معمول است که این چهار حالت را به شکل یک enum معرفی کنند که در کدهای تولیدی فوق نیز به همین نحو عمل گردیده و public enum State معرف چهار حالت ذکر شده است. همچنین رویدادهای ذخیره، نیاز به ویرایش، ویرایش، تائید و رد نیز توسط public enum Trigger معرفی شده‌اند.
در قسمت Transitions، بر اساس یک رویداد (Trigger در اینجا)، انتقال از یک حالت به حالتی دیگر را سبب خواهیم شد.
تعاریف اصلی تنظیمات ماشین حالت، در سازنده کلاس BlogPostStateMachine انجام شده است. این تعاریف نیز بسیار ساده

هستند. به ازای هر حالت، یک Configure داریم. در متدهای OnEntry و OnExit هر حالت، یک سری callback function فراخوانی خواهند شد. برای مثال در حالت Rejected یا Approved می‌توان ایمیلی را به ارسال کننده مطلب جهت یادآوری وضعیت رخ داده، ارسال نمود.
متدهای PermitIf سبب انتقال شرطی، به حالتی دیگر خواهند شد. برای مثال رد یا تائید یک مطلب نیاز به دسترسی مدیریتی خواهد داشت. این نوع موارد را توسط delgateهای Guard ایی که برای مدیریت شرطها ایجاد کرده است، می‌توان تنظیم کرد.
PermitReentryIf سبب بازگشت مجدد به همان حالت می‌گردد. برای مثال ویرایش و ذخیره یک مطلب در حال انتشار، سبب تائید یا رد آن نخواهد شد؛ صرفا عملیات ذخیره صورت گرفته و ماشین حالت مجددا در همان مرحله باقی خواهد ماند.

نحوه استفاده از ماشین حالت تولیدی:
همانطور که عنوان شد، حداقل استفاده از ماشین‌های حالت، refactoing انبوهی از if و elseها است که در حالت مدیریت یک چنین گردش‌های کاری باید تدارک دید.

```
namespace StatelessTests
{
    public class BlogPostManager
    {
        private BlogPostStateMachine _stateMachine;
        public BlogPostManager()
        {
            configureWorkflow();
        }

        private void configureWorkflow()
        {
            _stateMachine = new BlogPostStateMachine();

            _stateMachine.GuardClauseFromBeginToInProgressUsingTriggerSave = () => { return
UserCanPost; };
            _stateMachine.OnBeginExit = () => { /* save data + save state + send an email to admin */
};

            _stateMachine.GuardClauseFromInProgressToPublishedUsingTriggerAccept = () => { return
UserIsAdmin; };
            _stateMachine.GuardClauseFromInProgressToRejectedUsingTriggerReject = () => { return
UserIsAdmin; };
            _stateMachine.GuardClauseFromInProgressToInProgressUsingTriggerSave = () => { return
UserHasEditRights; };
            _stateMachine.OnInProgressExit = () => { /* save data + save state + send an email to user
*/ };

            _stateMachine.OnPublishedExit = () => { /* save data + save state + send an email to admin
*/ };
            _stateMachine.GuardClauseFromPublishedToInProgressUsingTriggerRequireEdit = () => { return
UserHasEditRights; };

            _stateMachine.OnRejectedExit = () => { /* save data + save state + send an email to admin
*/ };
            _stateMachine.GuardClauseFromRejectedToInProgressUsingTriggerRequireEdit = () => { return
UserHasEditRights; };
        }

        public bool UserIsAdmin
        {
            get
            {
                return true; // TODO: Evaluate if user is an admin.
            }
        }

        public bool UserCanPost
        {
            get
            {
                return true; // TODO: Evaluate if user is authenticated
            }
        }

        public bool UserHasEditRights
        {
            get
            {
                return true; // TODO: Evaluate if user is owner or admin
            }
```

```
        }

        // User actions
        public void Save() { _stateMachine.TryFireTrigger(BlogPostStateMachine.Trigger.Save); }
        public void RequireEdit() {
_stateMachine.TryFireTrigger(BlogPostStateMachine.Trigger.RequireEdit); }

        // Admin actions
        public void Accept() { _stateMachine.TryFireTrigger(BlogPostStateMachine.Trigger.Accept); }
        public void Reject() { _stateMachine.TryFireTrigger(BlogPostStateMachine.Trigger.Reject); }
    }
}
```

در کلاس فوق، نحوه استفاده از ماشین حالت تولیدی را مشاهده می‌کنید. در delegateهای Guard، سطوح دسترسی انجام عملیات بررسی خواهند شد. برای مثال، از بانک اطلاعاتی بر اساس اطلاعات کاربر جاری وارد شده به سیستم اخذ می‌گردند. در متدهای Exit هر مرحله، کارهای ذخیره سازی اطلاعات در بانک اطلاعاتی، ذخیره سازی حالت (مثلا در یک فیلد که بعدا قابل بازیابی باشد) صورت می‌گیرد و در صورت نیاز ایمیلی به اشخاص مختلف ارسال خواهد شد.
برای به حرکت درآوردن این ماشین، نیاز به یک سری اکشن متد نیز می‌باشد. تعدادی از این موارد را در انتهای کلاس فوق ملاحظه می‌کنید. کد نویسی آن‌ها در حد فراخوانی متد TryFireTrigger ماشین حالت است.

یک نکته:
ماشین حالت تولیدی به صورت پیش فرض در حالت State.Begin قرار دارد. می‌توان این مورد را از بانک اطلاعاتی خواند و سپس مقدار دهی نمود تا با هر بار وهله سازی ماشین حالت دقیقا مشخص باشد که در چه مرحله‌ای قرار داریم و TryFireTrigger بتواند بر این اساس تصمیم‌گیری کند که آیا مجاز است عملیاتی را انجام دهد یا خیر.

نظرات خوانندگان

نویسنده: برنامه نویس
تاریخ: ۱۳۹۱/۱۰/۱۴ ۱۰:۲۵

استفاده از یک Dictionary از نوع string و Action، چه مشکلی نسبت به این روش داره؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۱۰/۱۴ ۱۰:۴۶

مشکلی نداره. شما در هر زمانی می‌تونید دست به اختراع مجدد چرخ بزنید. با یک Dictionary از نوع string و Action فقط قسمت حالات و رویدادها رو طراحی کردید. مابقی قسمت‌ها مانند انتقال‌ها رو هم که اضافه کنید می‌شود کتابخانه Stateless.

نویسنده: امیر هاشم زاده
تاریخ: ۱۳۹۳/۰۶/۱۶ ۲۳:۲۶

طبق کد زیر:

```
_stateMachine.OnPublishedExit = () => { /* save data + save state + send an email to admin */ };
_stateMachine.GuardClauseFromPublishedToInProgressUsingTriggerRequireEdit = () => { return
UserHasEditRights };
```

آیا درست متوجه شدم که: بازهم ابتدا سطح دسترسی بررسی می‌شود و سپس عملیات ذخیره سازی صورت می‌پذیرد؟!

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۶/۱۶ ۲۳:۴۲

این‌ها فقط یک سری تنظیم اولیه سیستم هستند. مهم نیست ترتیب معرفی آن‌ها به چه صورتی است. اجرای آن‌ها اینجا انجام نمی‌شود.

نویسنده: امیر هاشم زاده
تاریخ: ۱۳۹۳/۰۶/۱۷ ۰:۱۶

ظاهرا در کلاس BlogPostStateMachine، متد سازنده آن به اشتباه BlogPost درج شده است.
برای تغییر حالت و مقدار دهی آن از بانک اطلاعاتی، باید کد کلاس BlogPostStateMachine را مانند زیر تغییر دهیم:

```
public BlogPostStateMachine(State state)
    {
      stateMachine = new StateMachine<State, Trigger>(state);
```