C O D E S H E R P A S

We do the technical heavy-lifting

INTRODUCTION TO THE STONEPATH WORKFLOW MODELING METHODOLOGY

CodeSherpas Inc.

CodeSherpas Inc • 525-K East Market St #190 • Leesburg, VA 20176

(888) 650-7829 • http://www.codesherpas.com

The StonePath Workflow Methodology

Introduction

"We will not tell the delegate from Russia how we manage our export control process in Estonia", the Estonian delegate said at the 1997 Conference on Export Controls in Oxford, UK, and thus began my journey into the large and complex world of modeling workflow. From 1996 through 2007 I worked on an import/export control system developed by the U.S. State Department that was given to former Soviet Union countries to help them control their borders.

There is a lot of stuff sold around the world that can be dangerous in the wrong hands. Riot control gear manufactured in Lithuania being sold to a police force in Latvia is probably a legitimate sale. That same gear going to a tribal leader in Afghanistan might not be. Brewery equipment being shipped from Germany can be used to make beer in one person's hands, or weaponized anthrax in another's. The same equipment that can make carbon fiber golf clubs can be repurposed to make nose cones for warheads. In the international trade community, these are called "dual-use goods", and under many circumstances their import and export is controlled; companies have to apply for licenses to move the material across international borders into 'areas of concern'.

When the Soviet Union dissolved, the newly independent nations found themselves with all kinds of 'interesting' pieces of military hardware and manufacturing facilities, weak economies, and smart people looking for employment. It was in their interests to be able to position themselves on the world market, and it was in the U.S.' interests to make sure they followed international treaties on the sale of this stuff.

Tracker

In 1996, my company was hired to build a system to help track the licensing, approval/denial, and transport of these goods. While we could get cooperation from the involved governments most of the time, there were plenty of roadblocks to sharing information about their internal processes. Writing software when you know what needs to be done is hard enough, but when your end users won't tell you what it needs to do, it becomes almost impossible. There were many interesting technical challenges on that project - from the translation into several Eastern European languages through the output of legally binding paperwork in each country - but the most interesting challenge was trying to model workflow that varied from country to country, when half the time we couldn't know what the official workflow even *was*. This forced us not to solve their particular problems, but to build a generalized solution that could be tailored to their workflow. The system we built is called Tracker; it was refined over an 11 year period and is in use today, protecting the borders of several countries. (More information is available at www.trackernet.org). Key concepts developed under this program have been used in a modernization effort for the U.S' import/export control and in the United Nations Effort on Humanitarian Demining.

Modeling the Impossible

We wrote that tool in Java. When we started, Java was relatively new on the technical landscape. There were no vendors selling workflow solutions, so the work we did was groundbreaking on many levels. In particular, we chose to model the complexity of their workflows as a State Machine. Years later, our customer told us that he "never actually expected the software to succeed, [he] just wanted it as an excuse to get the delegates from the countries

cooperating with each other". A primary reason we succeeded was because of the open-ended nature of state-based workflow modeling.

Today, most vendor workflow solutions are process-based, not state-based. Marketing literature would have you believe that process-based is somehow superior; In truth, both methodologies have their purposes, and they can be used together to model very complex workflow situations. The work we did on Tracker took state-based workflow modeling further than I have seen it taken elsewhere, and into a proprietary Java-based workflow tool called Journeyman. Since then, I have taken these concepts into other environments to solve other workflow-related problems, and this has grown into a comprehensive methodology for workflow modeling called "StonePath". These projects have led to the creation of the StonePath project, a state-based workflow modeling tool for Ruby (open sourced under the MIT license and available on github). While the StonePath reference implementation is in Ruby, the modeling concepts can be useful in just about any modern language.

Basics of StonePath Modeling

SchoolHouse Rock

Almost everything I needed to know to start modeling workflow, I learned from Schoolhouse Rock. I am probably betraying both my age and my nationality with that statement, but SchoolHouse rock was a series of educational cartoon/music videos that aired Saturday mornings in the U.S. from the mid 70's to the early 80's. One of the cartoons taught children how a Bill becomes a Law in the U.S. Government. In that cartoon, a singing and dancing paper scroll (named Bill, of course), describes how he becomes a law. "First, I'm just a good idea that someone tells their congressman", he says... and he goes on to talk about how he is in committee, then voted on, how he might get vetoed, put up for a revote, and so on. We are anthropomorphizing the bill, and by doing so we find out about the different 'states' he can be in and how he moves from state to state. 'Anthropomorphizing' an object - imagining your object as an agent with its own desires and actions as it goes through a workflow - is a great way to start planning your workflow. We will use that technique in an exercise below. (link to the video on youtube)

Your Boss' Boss

In one exercise with a group of business analysts defining a system for managing legal actions, I said, "Imagine that this is a high profile legal action that the New York Times is doing an expose' on. Your Boss' Boss is calling you every hour to find out whats going on, and is expecting an answer in 5 words or less. What are you going to say to her?

"It's in Data Entry."

"Its under investigation now."

"Its on Todd's desk, awaiting legal review"

"We need to verify the investigators findings"

These also point to some of the things we need to think about when defining a state-based workflow, but there is more here than just states... there seem to be actions to be performed, assignments to particular people, and so on.

These two techniques - "Anthropomorphizing the Work Item" and "Your Boss' Boss" - are two important requirement elicitation techniques for modeling a state-based workflow. We will see these techniques used as we learn to model state-based workflows.

Major Components of StonePath Workflow

The WorkItem

The WorkOwner

The Task

The WorkBench

We will discuss each in turn.

A Real-World Example

Lets start with a real-world example; something you probably wouldn't really model in software, but something that we all should have some experience with - a trip to the grocery store.

Tammy needs to do some grocery shopping. She knows she needs to get something for dinner tonight, has a prescription she needs to get filled, and probably needs to get a few other things while she's there. How does this relate to workflow?

First, Tammy is going to do some planning for her trip. She might be the kind of person who goes to the store with a discrete shopping list and a bunch of coupons, or she might be the kind of person that just thinks "I feel like making spaghetti tonight" and wanders around the store picking stuff up. We know she is going to get a prescription filled, and pick up a few other things while she's there.

Second, she actually needs to go to the store and buy her stuff.

Third, she finishes shopping, comes home, puts stuff away, and makes dinner.

Believe it or not, that is a great first-pass at a workflow we can model. It starts out looking like this:

[plan] -> [shop] -> [done]

That is very oversimplified, but once we have a notional "3-5 bubble" diagram, we can start to hang ideas off of it as we elicit business requirements.

What happens in that 'shop' bubble? Well, we know that Tammy has a prescription to fill. Most people would go to the pharmacy, drop it off, and continue to shop while they wait for it... There is no reason to stand by the pharmacy for 15 minutes.

Most shoppers know that you don't put ice cream in the shopping cart until you are almost done shopping - Tammy certainly won't want to put it in while we have 15 minutes to wait for her prescription; it might melt. Taking this a little further, she'd probably plan to get all of our dry goods on her shopping list first, followed by perishable refrigerated things (milk, eggs, veggies), followed by frozen stuff.

what does our 'shop' bubble look like now?

[start filling prescription] -> [get dry goods] -> [get refrigerated goods] -> [get frozen goods]

Wait! What about picking up the prescription? If you noticed that, I'm proud of you - you will have the kind of eye necessary for debugging workflows. Lets save the prescription talk for a little later... after all, that isn't so much a "state" we are in, so much as a "task" the pharmacist is performing and one we need to do when he's done.

Long Running Tasks

At my grocery store there is a deli counter that often has a long line. They recently installed a touchscreen kiosk that lets people place their order, go do their shopping, and come back later for their deli meats and cheeses. This is a lot like the prescription-filling... I can start a long-running task, go deal with the other states in my flow, and come back to it later. so lets rename that first bubble to 'start long-running tasks'. Again, we'll talk more about actual tasks later.

[plan] -> [start long-running tasks] -> [get dry goods] -> [get refrigerated goods] -> [get frozen goods] -> [done]

So there we have it - a simple state-based workflow that models Tammy's shopping trip. It doesn't model every single detail... depending on what we wanted to model, we could easily think of other states to add. But lets leave that for later iterations of refinement. We still have other details to talk about.

The Lie That Teaches You the Truth

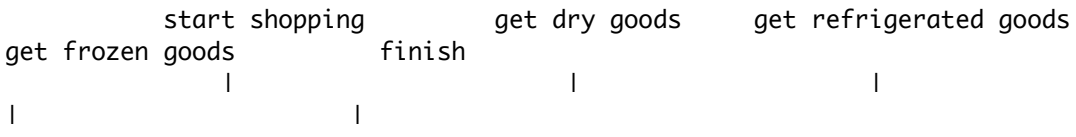
'State machine purists' reading along right now are probably muttering to themselves - 'these aren't states he's modeling!'. And they are right. I have led you down this path because, in dialogs with clients, this is how it typically happens. We talk about states, but in truth, the clients are often more interested in talking about the transitions that put them into states rather than the states themselves. This is an important part of this as a methodology - let the client speak in their language, and slowly bring it around to the model we need. What does this mean to the work we have done?

States vs. Transitions

In the classical computer science concept of a 'State Machine', states are connected by a series of transitions. In order to move from one state to another, there must be a transition connecting them. If there is no transition, there is no way to move from the first state to the second state.

In workflow modeling, we want those transitions to have names. The best names for these transitions are action verbs, and this is why its easy to get the client to talk about transitions over states. Some examples in common workflows are 'approve', 'deny', 'escalate', 'decide', 'perform', 'open', 'close', 'authorize'. One nice side-effect for the design of software systems is that these transition names often appear as menu choices or button names. They work well for that.

States are actually the uninteresting time period in between those actions where things are sitting in people's in box, something is waiting for approval, and so on. We often have to extract those names or derive them from the workflow. Lets remodel our state diagram keeping this in mind:



```
[in_planning] -> [place_orders_phase] -> [dry_goods_phase] ->
[refrigerated_goods_phase] -> [frozen_goods_phase] -> [done]
```

See how those transitions are all actions? Start, get, finish... that is a good test that we are modeling that correctly. While in each state, things are happening to close that state out in preparation of the next transition. For instance, while in the dry_goods_phase, we will be picking things off the shelves in the grocery store and putting them into our cart.

See how the state names can all be used to answer the "Boss' Boss" question? "We are in the frozen goods phase of our shopping trip now". That might not sound much like the way you would normally describe a shopping trip, but if we were modeling an approval process for license applications, we could probably identify states that map back to domain terms.

There are several simple suggestions for good state names:

- start state names with 'positional' terms like 'in', 'on', or 'under', such as 'in process', 'in final review', 'on hold', or 'under investigation'. Or,

- end state names with terms that imply a temporary condition, like 'phase' (like we did above). Or,

- use past tense actions, like "done", "reviewed", or "countersigned".

- Do not use action verbs - these best describe transitions.

- Do not use or imply time or duration in your names, like "3 day waiting period". Time and trigger events are another concept in this modeling methodology.

States often have pretty generic names like 'in-process' and 'pending', or past-tense verbs of the action that brought them to this state, like 'decided' or 'activated'. When possible, a more descriptive name is better, but when in doubt, the descriptive names are probably the names of transitions.

Exactly What is our WorkItem?

In modeling workflows, we model a "Work Item" - our object with state. Exactly what are we modeling here?

In all object-oriented software construction, we often end up modeling concepts. While we might have a tangible thing in some kinds of modeling (like our Bill above - or if we were modeling a photographer's workflow for a photograph), sometimes the thing we model is an intangible. In this example, our intangible workitem being modeled is a "ShoppingTrip".

The WorkOwner

In the StonePath modeling methodology, every WorkItem has one and _exactly_ one WorkOwner. In this example, it is Tammy. If Tammy had instead given her husband Chris the shopping list and said "You're doing the shopping today", then that would be a 'reassignment of the workitem' to Chris. At this point, Tammy technically has no say in how the shopping is done. If Chris decides he wants to pick up some Oreos, that is Chris' call.

So what about the pharmacist and the deli meister? What are they in relation to this workflow? They aren't responsible for the shopping list, but they are playing some role in this whole shopping experience. In order to understand that, we have to model what they are doing.

The Task

In the execution of any workflow, there are 'tasks' that people are going to perform. The pharmacist has a prescription to fill. The Deli meister has a pound of turkey and a half pound of provolone cheese to slice. Tammy is going to pick up some ice cream. These are Tasks in the workflow.

In addition to the states we model, we also need to think about the tasks that fire off while in these states. We can see now in the bubble model, the "start long-running tasks" starts to make some sense... these tasks need to start then, but they don't necessarily need to finish then. They can finish anytime between now and the time Tammy checks out.

Tasks have a life of their own. A task can be:

'pending'... that is, it hasn't started yet.

'in-process', as in 'it has started, but we are still waiting for it to finish'

'completed', as in "you can come pick up your prescription now"

and several other states related to it being cancelled, expiring, and so on.

Hmm... this starts to look like another state machine. Each task has its own state (although a very simple and limited one).

The interesting thing about 'tasks' is that they can also be assigned to people (well technically, 'agents' - we could assign them to other computer programs for processing). Our prescription task is assigned to a pharmacist, our deli order task is assigned to the deli meister, and so on. A task belongs to bothe the workitem that created it and the user (actually 'workbench') that is assigned to handle it.

You might begin to see what the other tasks are for the other bubbles... "Get Oreos", "get Eggs", "get Ice Cream" are the kinds of tasks we will start in the other bubbles.

The WorkBench

The assignment of a task to a person (or agent) is done through a "WorkBench". You might also think of this as an 'in box' for assignments. Workbenches can be private (as in 'all pharmacy orders for narcotics need to go to the senior pharmacist on duty'), or can be shared amongst a group of people (as in any pharmacist on duty can fill a prescription for antibiotics). In simpler workflows, it is common to model the agent *as* the workbench. In more complex workflows, it is useful to have this separation so that tasks can be assigned to groups of people.

WorkItem, WorkOwner, Task, WorkBench. The major ingredients of State-Based Workflow

While there is a lot more in the details of the StonePath Methodology, these four concepts make up the backbone that, once understood, allow developers, business analysts, clients, and users communicate to define the user stories for the development of the system. The rest of StonePath are details that fall out of some of the implications of the interaction of these four elements.

Note how some nice features fall out of this model... If Chris decides to go to the store with Tammy, she can start tasking him: "Go get the lettuce while I go get the eggs". This is modeled as a task assignment to Chris' workbench. I always like when correct behavior I didn't explicitly plan for comes out of a model; it feel like the model is "correct" on some level.

State Guards

As our workflow stands now, technically, it is possible for us to finish our shopping and get to 'done' before our prescription is filled or our deli order has been completed. While we seem to have a good start, there are clearly some problems with this model. What we need is a mechanism of saying "Don't transition into the 'done' state until the long-running tasks are complete". In fact, we might not even want to start shopping for our frozen goods, just in case those tasks take a long time.

This is the concept of a State Guard. In order to build a complete state-based workflow, we need the ability to say things like "Don't enter this state if some other condition isn't true", or even "Don't leave this state unless some condition is true". That is, we want to guard the entry and exit of states with boolean conditions. For now, lets just draw a little lock on our diagram that says this for the frozen food state.

This all seems pretty simple at this point, really... We are modeling things that you might already do without thinking about it - "Don't put the frozen food in my shopping cart if I'm likely to be here a while". We are just modeling it with some explicit concepts.

Parallel and Sequential Activities

In any workflow, state or process based, we need to think about what activities happen sequentially and what activities happen in parallel. In our state-based modeling, these concepts map quite naturally to states and tasks... Our states are sequential - we move from one to the other in a logical, defined order. Our tasks, on the other hand, can happen independently of each other unless we explicitly tie them together (our deli order and prescription can happen independently, as can our dry and refrigerated goods... but all those happen before we get the frozen goods).

This is one of the strengths of the state-based workflow modeling - for me at least, the activities that can happen in parallel map naturally to my thought processes. Defining parallel, independent activities means that our workflows can scale by adding more resources to the problem. That is, if Tammy and Chris brought their daughter to the store too, she'd be able to go get stuff, making the whole trip go faster.

Multiple Paths

Our simple state machine had a linear path through the states. There is no reason why this has to be the case; plenty of state machines have alternate paths where one state can branch into one path or the other, where states can point back to previous states, and so on - the key to understanding how this works is by realizing that, while there are many paths, the WorkItem is still only in one state at any time. We will see some parallel activities shortly.

Long-Running Tasks - A Closer Look

From Tammy's perspective, she gave a task to the Pharmacist - "Fill this prescription".

From the Pharmacist's point of view, this task creates a new WorkItem... That is, he is going to be the Owner of a WorkItem that will have its own flow.

There are tasks to this flow (check customer's insurance, check prescription for drug interactions, fill bottle with pills, bill insurance, notify customer prescription is ready for pickup), and may have a workflow that varies from simple (in-process -> done, to something involving complexity surrounding the distribution of controlled substances).

Likewise, the deli meister may have a workflow for their task. The register clerk may have a workflow associated with cash vs. credit, bagging the groceries, and so on. Each task might simply be something that is 'done', or could spawn a whole sub-slow to the parent flow of "Tammy going shopping". The best part is that the encapsulation, and

communication through tasks and workbenches, keeps each workflow isolated. Tammy's plans for shopping didn't have to change because the pharmacist has new procedures for billing insurance companies. Again, this sounds like common sense coming through in our design - but how often have you seen poorly coupled software that wouldn't allow one thing to happen because of some other, seemingly unrelated thing?

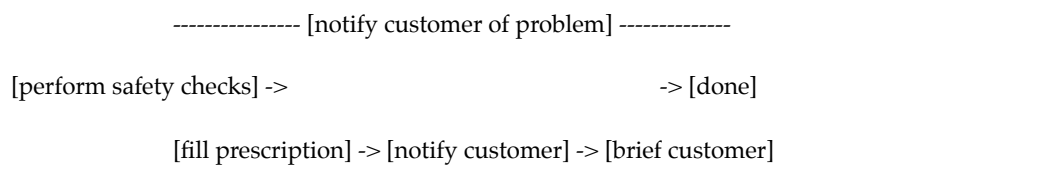
Lets take a closer look at the Pharmacist's workflow for filling a prescription. What does a pharmacist do?

First, he has certain safety checks he's going to perform. He's going to make sure the prescription isn't forged, he's going to check the customer's records to see other prescriptions they have filled to make sure there are no dangerous interactions, he will make sure they haven't filled the prescription recently, and he'll check with the insurance carrier to make sure their prescription card is valid. These can pretty much happen in any order, and could even be given to pharmacy assistants to perform.

Second, assuming everything is ok, he will fill the prescription. Depending on the medicine this might simply be a prepackaged bottle, a liquid that needs to be reconstituted, or pills that need to be dispensed into a pharmacy bottle. He will also print a label for the bottle with handling instructions for the medicine like "take 2 pills twice a day with food". Again, there are two steps here that can happen in any order once we know we are filling the prescription. If there is a problem with the prescription, the pharmacist will call the customer back to handle it.

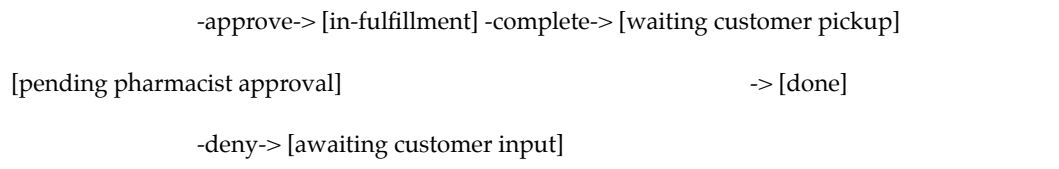
Third, after the prescription has been filled, they will notify the customer. When the customer returns they will sign for the prescription and be given a chance to ask the pharmacist any questions about the medication.

Lets model this workflow:



We have something new in our workflow - we have taken multiple branches. But something isn't quite right yet. Do you see it? Again, we have started by modeling the interesting activities (tasks and transitions), not the states. Look at the names - "perform", "fill", "notify", and so on. Those are clearly action verbs. These don't pass our "Schoolhouse Rock Test". If our prescription were a singing, dancing piece of paper, he'd be more likely to say "I'm waiting for the pharmacists approval", and "I'm waiting for the client to pick me up".

With these naming conventions in mind, lets redraw this workflow:



note that with this refined version, we have a new arrow... from customer input, the flow can go back to "pending pharmacist approval". Perhaps the insurance denied the prescription and Tammy said, "Fine - I'll pay for it myself". at that point, the flow goes back to the pharmacist where it will either be approved, or the next issue is brought to the attention of the customer. The 'pending pharmacist approval' bubble will have many of the tasks we talked about that can happen in any order... check the insurance card, check for dangerous drug interactions, make sure the prescription isn't forged, etc.

Transition Guards

Just as we had guards on the entry and exit of states, we can also guard explicit transitions. This most often occurs when there are transitions that can only be performed by certain roles with authority... for instance, on checkout, a cashier who is not of legal drinking age might not be able to sell alcohol. In this situation, a manager can authorize the sale (the transition from pending sale to sold).

The role of Time in Workflow

When we model workflow, we often have time constraints on some of the things we need to do. Often, people try to model this into the states themselves, like "3-day waiting period" as a state name. Incorporating time into the state name is generally a bad idea; you end up with a proliferation of states for rule variants, like a "2-day waiting period", "5-day waiting period", and so on. If we 'anthropomorphize out workitem', we quickly see that the workitem is the thing that should be managing its time.

In our pharmacist's workflow, we might have a rule that "prescriptions waiting for pickup more than 3 days trigger a phone call to the customer". In order to do this, we don't create new states for "waiting customer pickup after late notification"... we add that rule to the prescription workitem. We add a method named something like "pickup_overdue?" to our object, with logic that says "if in this state for more than 3 days, return true". We then add a method that creates a task to call the client and assigns it to a pharmacist's assistant. We can then have a simple piece of logic:

```
if (prescription.pickup_overdue? && prescription.no_outstanding_notification?)
  prescription.create_overdue_notice_task
end
```

So how and when does that piece of logic get checked?

Periodic Processes

In larger workflow 'engines', there are continually running processes that push things like this along. In this workflow methodology, software agents can be workflow participants as well. In the simplest case, that piece of logic will be put into a periodic task that is automated to run every few hours. In more complex cases, a software agent may actually have it's own workbench and receive task assignments to act on (again, just like a human, waking up every now and then to do some work).

With a periodic task, it is possible to do more conventional 'workflow routing' activities. For instance, you could have a workflow state that isn't meant for 'human eyes' - you have a periodic task that wakes up, gets all the work items in that state, and uses some logic to decide what the next state should be.

If you are interested in using the StonePath gem in a Rails application, I suggest you also look at the Crondonkulous gem (also at http://www.github.com/bokmann) for automating periodic tasks within a Rails application with cron.

Access Control

The final piece of the workflow puzzle is controlled access to information. Lets take, for example, the pharmacist who is delegating out tasks. Perhaps a pharmacist's assistant is going to check for drug interactions, and might brief out the patient when the medication is picked up. Regulations might prevent that assistant from knowing the customer's insurance information, exact medical condition, or home address, but all of this information might be part of the pharmacist's WorkItem for the prescription. At another point in time, the pharmacist's assistant might be doing the data entry and need to be able to not just see, but modify all of this information. How can we control this?

Access Control in StonePath is controlled at the intersection of the WorkItem's state, and the User's Role. Users in a StonePath Workflow have a 'role' they are performing in the workflow - Customer, Pharmacist, Deli Meister, and Pharmacist Assistant are all roles we have discussed so far.

When we define the states a WorkItem can be in, we can control the access any role has to the workitem at that time. We can say, for instance:

```
WorkItem: Prescription
  state: "pending pharmacist action"
    role: Pharmacist
      allow: all
    role: Pharmacist Assistant
      deny: customer address
  state: "in fulfillment"
    role: Pharmacist
      allow: all
    role: Pharmacist Assistant
      deny: all
  state: "waiting customer pickup"
    role: Pharmacist
      allow: all
    role: Pharmacist Assistant
      deny: customer address
      deny: insurance information
```

Of course, the exact details of this language are a little more complex, but this give us the ability to have a fine-grained access control at the intersection of the job responsibilities of people by role and the state of the workitem.

Putting it All Together

The StonePath Reference Implementation is implemented in Ruby, and relies on Scott Baron's excellent "Acts as State Machine" gem for the state and transition definitions (After the release of Rails 3, the reference implementation will transition to using the new ActiveRecord::StateMachine module, to keep dependencies at a minimum). Lets actually look at some source code for how this would work.

The ShoppingTrip class

```
class ShoppingTrip < ActiveRecord::Base
  include StonePath

  stonepath_workitem

  owned_by :Person
  target_of :ShoppingTasks

  aasm_initial_state :plan_trip

  aasm_state :in_planning
  aasm_state :dropoff_orders_phase
  aasm_state :dry_goods_phase
  aasm_state :refrigerated_goods_phase
```

```
  aasm_state :frozen_goods_phase
  aasm_state :done

  aasm_event :start_shopping do
    transitions :to => :drop_off_orders_phase, :from => :in_planning
  end

  aasm_event :get_dry_goods do
    transitions :to => :dry_goods_phase, :from => :drop_off_orders_phase
  end

  aasm_event :get_refrigerated_goods do
    transitions :to => :refrigerated_goods_phase, :from => :dry_goods_phase
  end

  aasm_event :get_frozen_goods do
    transitions :to => :frozen_goods_phase, :from => :refrigerated_goods_phase, :guard
=> !long_running_tasks?
  end

  aasm_event :finish_shopping do
    transitions :to => :done, :from => :dry_goods_phase
  end

  def long_running_tasks?
    # return true if there are outstanding long running tasks, like prescription
filling
  end
end
```

This is real working code for a ShoppingTrip class. On line 1, we include the StonePath library. This brings in everything we need to be able to define the rest of the class, including the state machine library.

On line 4 we declaratively define this class as a stonepath_workitem, which allows it to 'hook into' workflow in several ways.

Line 6 states that this workitem will be owned by instances of the Person class.

Line 7 says that this workitem is the subject of a particular kind of task, shopping_tasks.

We will see how those work in a little bit.

Line 9 defines out first state - the 'plan_trip' state. The rest of this class defines the states and transitions that make up the rest of our state-based workflow. Notice how 'get_frozen_goods' defines a guard? This guard calls the method "long_runing_tasks?", and if that returns true, we will not be able to follow this transition into the 'frozen_goods_phase'

The Person class

```
class Person < ActiveRecord::Base
  include StonePath
```

```
  stonepath_workowner
  workowner_for :ShoppingTrips

  stonepath_workbench
  workbench_for :ShoppingTasks
end
```

Of course, a real Person class is going to contain things like a name, a password, perhaps ldap authentication credentials, and so on... but this is all we need to be able to create a Person object representing Tammy, and have her be an owner for one or many ShoppingTrip workitems. Since we can also give tasks to Tammy and Chris (such as 'get oreos') the Person needs to say that they are a 'workbench_for :ShoppingTasks'. That might seem a little, odd, but we'll talk about that a little more when we talk about the Pharmacist. Note that it would be possible, in more complex workflow scenarios, to be an owner_for more than one kind of workitem and a workbench_for more than one kind of task.

We have referenced a ShoppingTask class twice - once in each model above. Lets define that class now.

The ShoppingTask class

```
class ShoppingTask < ActiveRecord::Base
  include StonePath

  stonepath_task

  task_for :ShoppingTrip
  assigns_to :Person
end
```

Of course, the ShoppingTask will have some data associated with it; perhaps the item to be picked up, or the prescription to be filled.

If you download that sample project, you can now fire up a command-line and begin manipulating these objects:

```
> tammy = Person.create
> trip = ShoppingTrip.create
> trip.owner = tammy
> trip.aasm_current_state
plan_trip
> perscription = trip.shopping_tasks.create("Prescription")
> ordeos = trip.shopping_tasks.create("get Oreos")
> ice_cream = trip.shopping_tasks.create("get Ice Cream")
> trip.start_shopping!
> pharmacist = Person.create
> pharmacist.shopping_tasks << perscription
> # the pharmacist now knows about the persciption and can begin processing it.
> trip.get_dry_goods!
> # go get the oreos
```

```
> trip.get_refrigerated_goods!
```

and so on. We can keep going with this workflow, but we now have the opportunity to discuss two other aspects.

A Person is a Workbench?

While that commonly does occur in workflows, this is really just a shorthand notation that allows us to assign tasks directly to people. You could just as easily model a workbench as an 'inbox' or a 'todo list', and have this make more sense.

```
class GroceryCounter
  include StonePath
  stonepath_workbench
  workbench_for :shoppingTasks
end
```

with this class, we can now say:

```
> deli_counter = GroceryCounter.create
> pharmacy_counter = GroceryCounter.create
```

This seems to make a lot more sense. Now several pharmacists can share responsibilities at the pharmacy_counter.

In order to make this work with the code above, we would need to be able to change the ShoppingTask class slightly:

```
assigns_to :Person, :GroceryCounter
```

As of the 1.0 reference implementation in Ruby though, assignments to multiple workbench types is not supported. This will be available in StonePAth 1.1

Event methods

In our interactive shell above, we can see that every event we defined becomes a method name we can execute. We can find out which ones are available in the current state by asking:

```
> trip.aasm_events_for_current_state
```

As mentioned earlier, it is very common to take the currently available states and assign them to button names or menu items.

In some cases though, it can be inconvenient in our code to have to figure out what our current state is and what the next thing we want the code to do is. In the case of our ShoppingTrip workflow, it is pretty much a linear path, so wouldn't it be convenient if we could just say:

```
> trip.next!
```

and have it transition to the next state in the chain? We can do that by defining the 'next event like so:

```
aasm_event :next do
  transitions :to => :long_running_tasks, :from => :plan_trip
  transitions :to => :dry_goods_phase, :from => :plan_trip
  transitions :to => :refrigerated_goods_phase, :from => :dry_goods_phase
  transitions :to => :frozen_goods_phase, :from => :refrigerated_goods_phase, :guard
=> !long_running_tasks?
  transitions :to => :done, :from => :dry_goods_phase
end
```

Defined this way, the next! method will simply 'do the right thing' depending on the state of the workitem.

Summary

We have covered a lot of concepts and code in this introduction to the StonePath Workflow Methodology. As with any modeling discipline, some things are immediately obvious, but the craft of handling edge cases comes only with experience using the tools. I encourage you to download the sample ShoppingTrip project and play with it yourself.

As a group of concepts, 'workflow' can be baked directly into your domain modeling; you do not need an expensive, monolithic, or resource-intensive 'workflow engine' to handle the concepts most of us will need in our typical workflow needs.