The Grocery Store

Contract Evaluation Framework Walk Through

Dave Arnold
Version 1.1
8/3/2007

Contents

Changes from Version 1.0

The following document contains all the information found within the previous version of this document. This document also includes the specification and evaluation of non-functional requirements, the inclusion of a centralized scenario in the store, and some language cleanup. In addition, the concept of contract namespaces and corresponding inclusion has also been added to this document.

Introduction

The following document contains a simple example based on a physical grocery store. The purpose of the example is to illustrate the elicitation of contracts and scenarios for the case study. In addition the example will illustrate the binding and execution tasks performed by the framework. The document is organized as follows. A description of the case study will first be presented. Next, six modular contracts will be derived and their syntax and semantics will be discussed in detail. Once the contracts have been presented and discussed, the contract compilation process will be outlined, followed by the binding steps required to map the elements defined within the contracts to the Implementation Under Test (IUT). Once binding is complete, the static check procedure is presented along with the necessary IUT instrumentation required for the evaluation of scenario execution and the execution of dynamic checks. Following instrumentation the process of scenario execution is discussed. The examination of execution metrics and non-functional requirements is then presented. Finally, the document concludes with a discussion of the contract evaluation report.

Case Study

The case study used within this example is modeled after a physical grocery store. A grocery store was selected because it requires little domain knowledge for the reader. A grocery store can be viewed as a set of customers who each, enter the grocery store, and select one or more items for purchase. Each of these items is placed into their shopping cart. Once the customer has finished selecting the items that he/she wishes to purchase, they proceed to the checkout, where they select an open cash, and then join the queue (possibly empty). Finally the customer purchases their goods and leaves the store.

The following assumptions will be made in order to simplify the example:

- Stores always remove all of their items and cash registers when they are closed
- Each item in the store is unique (i.e. two cans of soup are different items)
- Cashes cannot be closed if they have customers waiting
- Customers cannot leave a queue once they have entered it
- Customers never enter a store unless they are going to buy something

Of course, there are a lot of other aspects within a complete grocery store. However the short description shown above will introduce several scenarios, scenario interactions, and non-functional requirements.

Step 1 – Contract Creation

Initially, we need to specify the contract(s). We will begin by looking at a modular contract design. That is, we will create six separate contracts which will define the grocery store. Each of these six contracts would then be assembled into a single contract project. A contract project consists of three elements:

1. One or more contracts
2. An IUT to evaluate the contracts against
3. A history of bindings which exists between the contract(s) and the IUT

We will now examine the first of the six modular contracts. Following each contract listing a detailed description of the syntax and semantics will be presented. Each contract is expressed using Another Contract Language (ACL). The ACL is a simple high-level contract language. Details of which will be presented in a separate document.

The *Item* Contract

At the heart of all grocery stores are the individual items. The first contract presented here represents a single store item. The contract listing is shown below:

```
Import Core;

Namespace DaveArnold.Examples.GroceryStore
{
	Contract Item
	{
		once Value sku;

		Responsibility new()
		{
			sku = context.SKU;
		}

		Invariant UniqueSKU
		{
			Belief UniqueSKU("All items must have a unique SKU")
			{
				UniqueValue(context.SKU);
			}
		}
```

```
            Responsibility Real Price()
            {
                  Belief NoFreeItems("No item is given away for free")
                  {
                        Post(#"~result~ != 0.00");
                  }
                  Belief NoNegativeItems("No item is sold for a
                                          negative price")
                  {
                        Post(#"~result~ > 0.00");
                  }
            }

            Responsibility String Name()
            {
                  Belief MustHaveName("All items must have a name")
                  {
                        Post(#"~result~ != null");
                  }
            }

            Responsibility Integer SKU()
            {
                  Belief MustHaveSKU("All items must have a sku")
                  {
                        Post(#"~result~ != null");
                  }
                  Belief MustHaveOrigionalSKU("Must have the sku
                                          we started with")
                  {
                        Post(#"~result~ == context.sku");
                  }
            }
      }
}
```

The contract begins with the ***Import*** keyword. The ***Import*** keyword functions in a similar fashion to the #include keyword in C++. ***Import*** is used to include additional static and dynamic checks defined by various contract developers. The ACL will include some pre-packaged static and dynamic checks. Checks are grouped into namespaces which are included into the current contract by specifying the corresponding namespace name following the ***Import*** keyword. The Item contract includes a single namespace "Core." The "Core" namespace will include basic/default static and dynamic checks.

The contract is enclosed within a namespaces, as shown via the ***Namespace*** keyword. Namespaces perform similarly to those found within C++ and C#. The purpose of namespaces is to organize and group contracts. It should be noted, that the contract namespaces which are used to store and group static and dynamic checks are different than the namespaces used to order and group the contracts themselves. If the namespaces overlap a compile-time error will be generated. More

information regarding namespace behavior will be illustrated in future contracts. The Item contract is located within the DaveArnold.Examples.GroceryStore namespace.

Next, the contract proper begins. A contract is denoted by the ***Contract*** keyword, followed by the contract's identifier. Each contract within the same namespace must have a unique identifier. Following the contract's identifier an optional generic parameter list can be specified using the **<** and **>** notation. Multiple generic parameters are separated by commas. The Item contract is not a generic contract and does not contain a generic parameter list. We will examine a generic contract shortly.

The first line of the Item contract defines a contract variable. Contract variables are specified by using the ***Value*** keyword. Contract variables can be used to preserve state information for the type bound to the contract. The ***once*** modifier indicates that the contract variable can only be assigned a single time. All contract variables must be initialized within the ***new*** responsibility. The ***new*** responsibility will be shown shortly. In the case of our Item contract, we will use the "sku" contract variable to record, and then ensure that a given item does not change its SKU during its lifetime.

The next section of the Item contract contains a responsibility. A responsibility can be seen as a functional requirement, or a functional unit. Responsibilities are bound to methods found within the implementation. As such, responsibilities contain a method like signature: optional parameters, and a return value. There are two special types of responsibilities: ***new*** and ***finalize***. The ***new*** responsibility (shown here) is evaluated once a new instance of the type bound to the contract is created and a constructor has been executed. The ***new*** responsibility cannot contain any preconditions. The ***finalize*** responsibility is analogous to a destructor, and is evaluated immediately before a destructor is called and the instance is destroyed. It should be noted that due to garbage collection, the ***finalize*** responsibility may be evaluated at anytime following the release of the instance. The ***finalize*** responsibility cannot contain post-conditions. Both special responsibilities are optional, and are automatically bound to their implementation counterparts. In the case of our Item contract, the ***new*** responsibility is used to store the initial value returned by the SKU responsibility.

The next section of the Item contract denotes an invariant named ***UniqueSKU***. As stated in the literature on design-by-contract, invariants contain one or more checks which are tested at the following times:

- Before the invocation of each instance method
- After the invocation of each instance method
- After the creation of each instance
- Before the destruction of each instance

A contract may have any number of invariant sections, but all of the invariant sections will be merged into a single set of static and dynamic checks which will be tested during the times listed above. In the case of the Item contract, there is a single invariant. The single invariant is nested within a belief. Beliefs allow a set of checks to be viewed as a single check, and when the results of the set of checks are displayed in the contract evaluation report, the belief is used. Beliefs are denoted by the ***Belief*** keyword followed by an identifier. The belief identifier is used to name the belief. Belief identifiers can also be

automatically generated by using the "~" notation. The "~" notation is used to indicate that an automatic belief name should be generated. An example of the "~" notation will be shown in a future contract. Following the belief identifier, a description of the belief is given as a quoted string. Belief descriptions are only used when reporting on the contract evaluation report. The set of checks composing the belief follow surrounded by matching braces.

In the Item contract there is a single check assigned to the UniqueSKU belief. The purpose of the check is to ensure that each item in the grocery store has a unique SKU number. The UniqueValue check would be defined as an extension (and imported via the previous ***Using*** keyword). The extension would insert instrumentation so that the profiler could gather and check each and every instance of the type bound to the Item contract, to ensure that no two items have the same SKU number. The UniqueValue check accepts one parameter, which specifies the field to check. The ***context*** keyword functions similar to the C++ this keyword, and is used to indicate that it is the current instance of the type bound to the Item contract that we are comparing against.

Next the ***Price*** responsibility is defined to have a floating point return value and no parameters. The body of the ***Price*** responsibility contains two beliefs, which are each composed of a single post-condition. Post-conditions are tested upon completion of the method bound to the responsibility. During static analysis, instrumentation is added to the IUT which will test the post-condition, and upon failure, information will be displayed in the contract evaluation report. Details of this process will be presented later. Post-conditions are specified using C# like syntax (via the '#' operator) with a few additional keywords. One of these keywords is the ***~result~*** keyword. The ***~result~*** keyword is a special variable which stores the value returned by the method bound to the responsibility. The functionality of the two post-conditions is straightforward and will not be explained further.

Following the ***Price*** responsibility, a ***Name*** responsibility is defined to ensure that each grocery store item has a name which is not an empty string. Finally, the ***SKU*** responsibility is defined to ensure that there is an actual SKU value, which will be used in the invariant. The ***SKU*** responsibility also ensures that the SKU value does not change. This concludes the Item contract.

The *BoundedContainer* Contract

With the individual items defined, we will require a container which will be used to store multiple items. An example of such storage is when the customers place items into their carts. The BoundedContainer contract listing is shown below:

```
Import Core;

Namespace DaveArnold.Collections
{
	Contract BoundedContainer<Type T, Integer MaxSize>
	{
		Value Integer size;
		Timer item_timer;

		Observability Boolean   IsFull();
		Observability Boolean   IsEmpty();
```

```
Observability T         ItemAt(Integer index);
Observability Boolean   HasItem(T item);
Observability Integer   Size();

Responsibility new()
{
      size = 0;
      Post(#"IsEmpty() == true");
      Post(#"Size() == 0");
      Post(#"context.size == 0");
}

Responsibility finalize()
{
      Pre(#"IsEmpty() == true");
      Pre(#"Size() == 0");
      Pre(#"context.size == 0");
}

Invariant SizeCheck
{
      Inv(#"context.size >= 0");
      Inv(#"context.size == Size()");
      Inv(#"context.size <= MaxSize");
}

Responsibility Add(T item)
{
      Pre(#"IsFull() == false");
      Pre(#"item != null");
      Pre(#"HasItem(item) == false");
      context.size = context.size + 1;
      item_timer.Start(item);

      Post(#"HasItem(item) == true");
}

Responsibility T Remove()
{

      Pre(#"IsEmpty() == false");

      item_timer.Stop(value);
      context.size = context.size - 1;
      Post(#"~result~ != null");
      Post(#"HasItem(~result~) == false");
}

Responsibility Remove(T item)
{
      Pre(#"IsEmpty() == false");
      Pre(#"HasItem(item) == true");

      item_timer.Stop(item);
      context.size = context.size - 1;
      Post(#"HasItem(item) == false");
}
```

```
        Scenario AddAndRemove()
        {
                once Value T x;
                Trigger(Add(x));
                Terminate((x == Remove()) | (Remove(x)));
        }

        Metric Integer TimeInContainer(T item)
        {
                item_timer.Value(item);
        }

        Exports
        {
                Type T
                {
                        not context;
                        not derrived context;
                }
        }
    }
}
```

As with the Item contract, the BoundedContainer contract begins with a single import statement, a namespace definition, followed by the contract declaration. The BoundedContainer contract is located within the DaveArnold.Collections namespace. The contract declaration contains two generic parameters. A generic parameter represents a type or value which can be used to customize the contract. Generic parameters are specified between **<** and **>** brackets, multiple parameters are separated by commas. In the case of the BoundedContainer contract, the first generic parameter is used to assign an element type to the collection. In effect, this creates a strongly typed container. The second generic parameter specifies the maximum size for the container.

Following the contract declaration, a contract variable named "size" is defined to store the actual size of the container. The actual size of the container will be stored separately from the IUT, so that the IUT's Size() method can be verified.

The BoundedContainer contract also includes a timer named "item_timer". Variables which are defined using the ***Timer*** keyword are specialized variables which have predefined methods which can be used for gathering metric information. Timers contain three predefined methods. The first method is ***Start(x)***. ***Start(x)*** is used to initialize and start a new timer linked to the value specified by the parameter "x". If a timer has already been initialized for the given parameter, the existing timer will be reinitialized. The ***Stop(x)*** method stops a timer initialized via the ***Start(x)*** method. If no such timer exists in the system, an error will be shown on the contract evaluation report. Finally, the ***Value(x)*** method is used to report the amount of time (in application ticks) which have elapsed since the timer was started (if the timer is still running), or the number of ticks which have elapsed between the calls to ***Start(x)*** and

Stop(x). The BoundedContainer contract will use the "item_timer" to keep track of the amount of time that each element spends within the collection. Examples showing the use of the timer and corresponding metrics will be discussed shortly.

Five observability methods are then defined. An observability method is bound to a read-only method within the IUT. Observability methods are used to acquire state information about the IUT for evaluation within a contract. The IUT developer is responsible for ensuring that the observability methods required by the contract are implemented within the IUT. Once the method binding for an observability method is completed, static checks will be automatically added to the contract to ensure that the corresponding IUT method is actually read-only. In the case of the BoundedContainer contract the following observability methods are defined:

- ***IsFull()*** – Returns true if the container is full, false otherwise
- ***IsEmpty()*** – Returns true if the container does not contain any elements, false otherwise
- ***ItemAt(Integer index)*** – Returns the item located at the specified index
- ***HasItem(T item)*** – Returns true if the specified item is stored within the container, false otherwise
- ***Size()*** – Returns the number of items currently being store within the container. If no items are being stored, then a value of zero will be returned

The first responsibility to be defined within the BoundedContainer contract is the ***new()*** responsibility. Recall that the contents of the ***new()*** responsibility are evaluated once a new instance of the type bound to the contract has been created, and a constructor execution has been completed. The body of the ***new()*** responsibility initializes our "size" contract state variable to zero, and defines checks to ensure that the observability methods return correct initial values.

Next the ***finalize()*** responsibility is defined. Recall that the contents of the ***finalize()*** responsibility are evaluated when an instance of the type that the contract is bound to is being destroyed by the garbage collector. The body of the ***finalize()*** responsibility ensures that the BoundedContainer has been emptied, and does not contain any elements when it is destroyed.

After the definition of the two special responsibilities, the BoundedContainer contract defines an invariant. The invariant is named ***SizeCheck*** and as the name suggests, ensures two things: first, that the number of elements in the container specified by the contract state variable matches the value returned by the corresponding observability method. Second that the current number of elements being stored by the container is not larger than the maximum size of the container.

The ***Add()*** responsibility is evaluated when a new item is added to the container. The body of the responsibility checks to ensure that the container is in a valid state before adding the new item, and then ensures that the new item was actually added to the container. Remember, that the invariant will ensure that the addition of the new item does not violate the container's size constraints. The ***Add()*** responsibility also uses the previously defined timer to begin recording how long the new item spends in the container.

The two ***Remove()*** responsibilities are evaluated when an item is removed from the container. Depending on the context of the removal a different responsibility is evaluated. The first ***Remove()*** responsibility removes a single item from the container. The second ***Remove()*** responsibility removes the specified item from the container. Each ***Remove()*** responsibility first checks to ensure that there is actually an item in the collection to remove. Then following the removal from the container, the responsibility ensures that the item was actually removed and the number of items remaining in the container has been updated accordingly. Both ***Remove()*** responsibilities contain a call to the ***Stop()*** method defined on the timer, to indicate that the timer tracking how long the item has been in the collection should be stopped. In the case of the first ***Remove()*** responsibility, the ***value*** keyword is used to refer to the value returned by the responsibility. That is, the item being removed from the collection. The ***value*** keyword is equivalent to the ***~result~*** keyword used within the ***#*** (sharp) notation. If the ***value*** keyword is used within a responsibility that does not have a return type, a compile-time error will be generated.

Following the definitions of the responsibilities, the BoundedContainer contract defines a single scenario. The scenario is named ***AddAndRemove()***. The purpose of the scenario is to ensure that each item added to the container is removed from the container before the container is destroyed. The definition of the scenario begins with the declaration of a local variable: “x”. The variable can only be assigned once, due to the use of the ***once*** modifier. The variable will be of type “T”, denoted by the first contract generic parameter. Each separate instance of the scenario will have a unique instance of the local variable. It should be noted that each instance of the contract may have several instances of a given scenario. That is, each instance of the BoundedContainer contract, will have a scenario instance created each time a new item is added to the container. The trigger of the scenario is specified using the ***Trigger*** keyword. In the case the ***AddAndRemove()*** scenario, the trigger is the successful evaluation of the ***Add()*** responsibility. In addition the local variable “x” will be assigned to the item that was added to the container. As previously stated, each time a new item is added to the container via the ***Add()*** responsibility, a new scenario instance will be created. The scenario is active until the scenario termination condition occurs. Such a condition is specified using the ***Terminate*** keyword. In the case of the ***AddAndRemove()*** scenario, there are two responsibilities that are able to remove an item from the container. As such, the or (“|”) operator is used within the body of the termination grammar to specify multiple scenario termination conditions. The first condition occurs when the non-specific ***Remove()*** responsibility is evaluated, and the resultant item removed is the item that was added in the scenario trigger, and stored in the local variable: “x”. The second condition occurs when the specific ***Remove(x)*** responsibility is evaluated, and the item to remove matches the value stored in the local variable: “x”. Any scenario instances which have been triggered, but have not been terminated when the contract instance is destroyed, will be reported as failed in the contract evaluation report.

Following the scenario definition the BoundedContainer contract defines a single metric method. Metric methods are used to report metrics gathered by the contract. The metric defined within the BoundedContainer is named ***TimeInContainer()***. ***TimeInContainer()*** returns an integer value which will be the number of application ticks which have elapsed while the specified item was in the container. This value is attained by using the previously discussed ***Value()*** method defined on the timer

variable type. Usage of the ***TimeInContainer()*** metric will be illustrated shortly. In addition to using timers or contract variables to report a metric value, metric evaluation extensions can be provided by a developer. Metric evaluation extensions allow a developer to create a specialized metric gathering and reporting plug-in. More information on the creation and usage of metric evaluation extensions can be found in a future document. Metrics are not automatically displayed within the contract evaluation report. An example showing how to display a metric value on the contract evaluation report will be shown in a future contract.

The final section in the BoundedContainer contract is an ***Exports*** section. Each contract may have at most one ***Exports*** section. The ***Exports*** section is used to denote binding points which are used within the contract. In the case of the BoundedContainer contract, the item type being stored within the container must be bound to a type found within the IUT. If the type "T" is not defined via an export declaration the "T" symbol will not be recognized by the ACL compiler. Following the export declaration, optional binding rules and restrictions are specified between **{** and **}** braces. There are several types of binding rules, two of which are illustrated in the BoundedContainer contract. The first indicates that the IUT type bound to the item of the container cannot be the same as the type bound to the container itself. The second indicates that the item type also cannot be bound to any type that is derived from the type bound to the container itself. Additional types of binding rules will be seen shortly.

This concludes the definition and explanation of the BoundedContainer contract. The contract is used to represent an un-ordered container of items, where each item is of the same type. The container also keeps track of how long each item is stored within the container. A BoundedContainer will be used to represent the shopping cart, storing items selected by a customer for purchase. We will now look at a contract for a FIFO queue, which will represent the queues of customers at each cash.

The *BoundedQueue* Contract

The BoundedQueue contract will be used to specify a FIFO queue, which will be used to represent customers waiting in line at a specific cash. Each cash will have a separate instance of a queue. The BoundedQueue contract listing is shown below:

```
Import Core;

Namespace DaveArnold.Collections
{
      Contract BoundedQueue<Type T, Integer MaxSize>
                                  extends BoundedContainer<T, MaxSize>
      {
            Value Integer inCount;
            Value Integer outCount;

            Observability Integer BackLocation();
            Observability Integer FrontLocation();
```

```
        Observability T Back()
        {
              ItemAt(BackLocation());
        }

        Observability T Front()
        {
              ItemAt(FrontLocation());
        }

        Responsibiliy new()
        {
              context.inCount = 0;
              context.outCount = 0;
        }

        refine Responsibility Add(T item)
        {
              Post(#"item == Back()");
        }

        refine Responsibility T Remove()
        {
              PreSet(#"~1~ = Front()");

              Post(#"~result~ == ~1~");
        }

        refine Responsibility Remove(T item)
        {
              Belief CannotLeaveQueue("Items must be removed in order")
              {
                    Pre(#"~fail~”);
              }
        }

        refine Scenario AddAndRemove()
        {
              once Value T x;
              once Value Integer index;

              atomic
              {
                    Trigger(Add(x));
                    context.inCount = context.inCount + 1;
                    index = context.inCount;
              }
              atomic
              {
                    Terminate(x == Remove());
                    context.outCount = context.outCount + 1;
                    Post(#"index == context.outCount");
              }
        }
  }

}
```

As with the previous contracts, the BoundedQueue contract begins with an import statement and a namespace declaration. As a bounded queue can be viewed as a specialized bounded container, the BoundedQueue contract will extend and specialize the previously discussed BoundedContainer contract. The extension is accomplished through the use of the ***extends*** keyword. The ***extends*** keyword immediately follows the BoundedQueue contract declaration. Any generic contract parameters required by the base contract, must either be provided literally or by referencing generic contract parameters from the new contract.

The BoundedQueue contract begins with the declaration of two contract variables. These variables will be used to store the number of items which have been put into the queue, and removed from the queue respectively. Notice that the ***once*** modifier is not used. This allows the contract variables to be assigned a value more than a single time.

Following the contract variable declaration, four observability methods are defined. The first two, ***BackLocation()*** and ***FrontLocation()*** are normal observability methods, which will be bound to read-only methods within the IUT that will return the location of the first item and last item in the queue respectively. The third and fourth obervability methods illustrate the concept of an observability method defined via the use of other observability methods, rather than being bound to a method within the IUT. That is, observability methods which define a body are defined by their body (which can only reference other observability methods), rather than being defined by a binding to an actual IUT method. The ***Front()*** and ***Back()*** observability methods use the previously defined ***FrontLocation()*** and ***BackLocation()*** obervability methods, along with the ***ItemAt()*** observability method defined within the BoundedContainer contract, to get the actual item at the front and back of the queue respectively.

The special ***new()*** responsibility is used to specify initial values for the contract variables. The contents of the ***BoundedContainer::new()*** will also be evaluated at the same time as the ***BoundedQueue::new()*** responsibility. As there is no additional functionality required for the ***finalize()*** responsibility, only the functionality found within the ***BoundedContainer::finalize()*** responsibility will be evaluated.

Next, the responsibility ***BoundedContainer::Add()*** is refined. Refinement is specified through the use of the ***refine*** modifier on the responsibility declaration. If such a responsibility does not exist within the base contract with a matching signature, a compile-time error will be generated. In the case of our ***Add()*** responsibility, we are adding an additional post-condition to ensure that the new item was added to the back of the queue.

Analogous to the ***Add()*** responsibility, the two ***Remove()*** responsibilities are also refined. Each version of the ***Remove()*** responsibility is refined in a very different way. The first ***Remove()*** responsibility uses the ***PreSet*** construct to store the item that is located at the front of the queue, before the method bound to the ***Remove()*** responsibility is called. The responsibility then ensures via a post-condition that the item removed from the queue was, in-fact the item at the front of the queue. The second ***Remove()*** responsibility, allows for a specific item to be removed from the queue. As queues generally maintain a fixed order of addition and removal, we do not want items to be removed

from the queue out of order. Thus, the second ***Remove()*** responsibility should not be used. The refined responsibility, defines a belief that all items must be removed from the queue in order, and as such creates a precondition that always fails. That is, if the second ***Remove()*** responsibility is called the contract will instantly fail. Instant failure of pre or post-condition can be accomplished by using the ***~fail~*** keyword. It is recommended that such a failing check is placed within a belief, so that the rationale for the failure is displayed in the contract evaluation report.

Finally, the BoundedQueue contract refines the ***AddAndRemove()*** scenario from the base contract. The scenario begins with the declaration of two scenario variables. The first will store the actual item that is placed into the collection, and the second one will store the index within the queue where the item is stored. This value will be used to ensure that the items in the queue are being stored in FIFO order. The ***atomic*** keyword is used to specify that everything within the ***{*** and ***}*** brackets are evaluated as an atomic action. In the case of the ***AddAndRemove()*** scenario, once the scenario is triggered, the value of the ***inCount*** contract variable is increased by one, and that value is stored within the ***index*** scenario variable. This creates a numerical ordering of each item within the queue. The second atomic section of the scenario specifies the scenario's termination condition. Note that only the first ***Remove()*** responsibility is included within the termination condition, as we do not wish to use the second one. Upon encountering the termination condition for the scenario, the ***outCount*** contract variable is increased by one, and compared against the ***index*** scenario variable in a post-condition. This will ensure that the item was removed from the queue in the correct order.

As there are no new symbols defined within the BoundedQueue contract, an ***Exports*** section is not required. This concludes the BoundedQueue contract. We will now turn our focus to three contracts which explicitly define the case study: Customer, Cash, and Store.

The *Customer* Contract

As the name suggests, the Customer contract represents a single customer within the case study. The Customer contract listing is as follows:

```
Import Core;

Using DaveArnold.Collections;

Namespace DaveArnold.Examples.GroceryStore
{
	Contract Customer
	{
		Value tStore store;
		Value Integer cartSize;

		Structure
		{
			Belief CanStoreItems("The customer needs a way to store
					items in his/her cart")
			{
				HasMemberOfType(tFoodContainer);
			}
		}
```

```
Observability Boolean HasPaid();

Observability Boolean InStore()
{
	context.store == null;
}

Observability Boolean HasFood()
{
	!foodContainer.IsEmpty();
}

Responsibility new()
{
	context.store = null;
	context.cartSize = 0;

	Post(#"InStore() == false");
	Post(#"HasPaid() == false");
	Post(#"HasFood() == false");
}

Responsibility finalize()
{
	Pre(#"InStore() == false");
	Pre(#"HasPaid() == true");
	Pre(#"HasFood() == false");
}

Responsibility EnterStore(tStore s)
{
	Belief ~1("The customer cannot be in more than one store")
	{
		Pre(#"InStore() == false");
	}
	context.store = s;

	Post(#"InStore() == true");
}

Responsibility LeaveStore(tStore s)
{
	Pre(#"InStore() == true");

	Belief ~2("Customer leaves the same store they entered")
	{
		Pre(#"s == context.store");
	}
	context.store = null;

	Post(#"InStore() == false");
}
```

```
Responsibility AddFood(tFoodItem item)
{
	Pre(#"item != null");
	Pre(#"foodContainer.HasItem(item) == false");

	Post(#"foodContainer.HasItem(item) == true");
}

Responsibility RemoveFood()
{
	Pre(#"HasFood() == true");
	context.cartSize = foodContainer.Size();

	Post(#"HasFood() == false");
}

Responsibility Pay()
{
	Pre(#"HasPaid() == false");
	Pre(#"InStore() == true");

	Post(#"HasPaid() == true");
}

Responsibility tCash SelectQueue(tStore s)
{
	Pre(#"s != null");
	Pre(#"s.IsOpen() == true");
	Pre(#"InStore() == true");
	Pre(#"s == context.store");

	Post(#"~result~ != null");
	Post(#"s.HasCash(~result~)");
	Post(#"~result~.IsOpen() == true");
}
```

```
        Scenario BuyItems()
        {
            once Value tStore store;
            exported once Value tCash cash;

            Trigger(new());
            EnterStore(store),
            (AddFood(dontcare))+,
            atomic
            {
                cash = SelectCash(store),
                Belief ValidCash("A valid cash will be selected")
                {
                    Check(#"cash != null");
                }
            },
            cash.AddCustomer(context),
            context == cash.NextCustomer(),
            RemoveFood(),
            Pay(),
            LeaveStore(store);
            Terminate(finalize());
        }

        Metric Integer CartSize()
        {
            context.cartSize;
        }

        Metrics
        {
            ReportAll("The average number of items in a cart is: {0}",
                AvgMetric(CartSize());
        }

        Exports
        {
            Type tFoodItem conforms Item
            {
                Store::tItem;
            }
            Type tCash conforms Cash
            {
                Store::tCash;
            }
            Type tFoodContainer conforms
                            BoundedContainer<tFoodItem, 100>;
            Type tStore conforms Store;
            Field foodContainer tFoodContainer;
        }
    }
}
```

The Customer contract beings with the standard import statement. The contract then references the DaveArnold.Collections namespace via the ***Using*** keyword. Unlike the ***Import*** keyword which imports extension namespaces, the ***Using*** keyword imports contract namespaces. In the case of the Customer contract, the DaveArnold.Collections namespace is imported so the customer's shopping cart can be represented via the BoundedContainer contract. The Customer contract is located within the DaveArnold.Examples.GroceryStore namespace. The Customer contract is standalone and does not extend any other contract, or include any generic contract parameters. The contract begins with the declaration of two contract variables. The first will be used to represent the store that the customer is currently shopping in. If the customer is not located in a store, the variable will be assigned a value of ***null***. The second is used to record the number of items that the customer has in their cart when they checkout. This value will be used to report metrics on the number of items that customers are purchasing.

The Customer contract contains a ***Structure*** section. Each contract may have at most one ***Structure*** section. As the name suggests a ***Structure*** section is used to express structural checks to be evaluated against the IUT. An example of a structural check is illustrated in the Customer contract. The check ensures that the type to which the Customer contract is bound to contains a member of the type bound to the ***tFoodContainer*** symbol. The ***HasMemberOfType()*** check is a static check, which is located within the "Core" namespace. Any number of structural checks could be specified within the ***Structure*** section. Such a check does not constitute a binding. If the member that is being checked is used within the contract, an entry in the contract's ***Exports*** section is required. The ***Exports*** section of the Customer contract contains such a binding.

Following the ***Structure*** section, three observability methods are specified. The first method, ***HasPaid()*** will be used to determine if the customer has paid for his/her items, and will be bound to a read-only method within the IUT. The second observability method, ***InStore()*** will be used to determine if the customer is currently located within a grocery store. ***InStore()*** does not need to be bound to an actual IUT method, as its value can be calculated by checking the previously defined ***store*** contract variable. The final observability method, ***HasFood()*** is used to determine if the customer has food in his/her cart. The value is calculated by checking if the food container is empty. The ***IsEmpty()*** observability method is guaranteed to be part of the food container because of the binding rules, which are specified in the ***Exports*** section. We will discuss the ***Exports*** section shortly.

The special ***new()*** responsibility, provides default values for the ***store*** and ***cartSize*** contract variables. The ***new()*** responsibility also specifies post-conditions to ensure that the customer has been initialized properly by the IUT. Likewise, the special ***finalize()*** responsibility specifies preconditions to ensure that the customer is no longer in the store, and has paid for his/her purchases.

The first normal responsibility for our customer is to get the customer into a store. The ***EnterStore()*** responsibility fulfills this requirement. The responsibility first checks to make sure that the customer is not already in a store, and then assigns our contract variable the value of the store the customer has just entered. That is, the customer will always know which store he/she is in. Finally, a post-condition is specified to ensure that the customer is actually in the store.

Analogous to the ***EnterStore()*** responsibility is the ***LeaveStore()*** responsibility. The ***LeaveStore()*** responsibility first checks to ensure that the customer is currently in a store. Next a check is specified to make sure that the store that the customer is trying to leave, is the same store that they are actually in. The contract variable is updated to indicate that the customer is no longer in the store. Finally, a post-condition is specified to ensure that the customer has actually left the store.

The ***AddFood()*** responsibility is used to add a new food item into the customer's shopping cart. The preconditions ensure that an actual item is specified, and the item is not already in the cart. The post-condition ensures that the food item has actually been put into the customer's cart.

The ***RemoveFood()*** responsibility is used to remove the food items from the customer's cart. The items are then given to the cash for scanning and payment. The ***RemoveFood()*** responsibility contains a precondition to ensure that the customer has at least one food item to remove. The scenario variable ***cartSize*** is assigned the number of elements which are in the collection before the bound IUT method is called. That is, the number of items in the customer's cart is recorded immediately before the cart is emptied. Recall, that the ***cartSize*** scenario variable will store the number of items that he customer has purchased. This value will be used for metric analysis. The post-condition ensures that all of the items have been removed from the customer's shopping cart.

The ***Pay()*** responsibility is used when the customer is ready to pay for his/her items. The preconditions ensure that the customer has not already paid, and that the customer is actually in a store. The post-condition ensures that the customer did actually pay for the items.

The ***SelectQueue()*** responsibility is used when the customer has finished adding items to his/her cart and is ready to select a cash for check out. The responsibility returns the selected cash. The responsibility beings by ensuring that the given store is valid, open, and that the customer is in the given store, and not a different store. Once the queue selection algorithm has been executed by the IUT, the result is checked to ensure that a cash was selected, the selected cash is part of the store, and the cash is open.

Following the responsibility definitions, the main customer scenario is defined. The scenario is named ***BuyItems()*** and contains the complete grammar for the customer's involvement within the grocery store. The scenario begins by defining two scenario variables. The first will contain the value of the store that the customer has entered. The second variable will contain the cash that the customer selects for checkout. Both scenario variables contain the ***once*** modifier. As previously stated, the ***once*** modifier indicates that the scenario variable can only be assigned once. The second scenario variable also contains the ***exported*** modifier. The ***exported*** modifier allows the value of the scenario variable to be referenced from outside the scenario. We will see an example of such usage shortly. The scenario is triggered immediately following the creation of an instance of the type bound to the Customer contract. The scenario then specifies a grammar of responsibilities that must be executed in the order specified by the grammar. The ***BuyItems()*** scenario grammar begins with the execution of the ***EnterStore()*** responsibility. The value of the store used in the call to the ***EnterStore()*** responsibility will be stored in the ***store*** scenario variable. The assignment is performed implicitly because the scenario variable is

referenced before it has been assigned to. In future references, the value passed to the responsibility will be checked against the value stored in the scenario variable. Next, the scenario grammar specifies that one or more food items are to be added to the customer's shopping cart. The ***dontcare*** keyword is used to indicate that we are not interested in the individual food items that are placed within the cart. The next event that the scenario grammar specifies is encompassed within an ***atomic*** block. As previously stated, ***atomic*** blocks specify a set of actions which are to be completed/checked as one atomic unit. In the case of the ***BuyItems()*** scenario grammar, the customer selects a cash register to line up for (join the queue). A check is added to ensure that a valid cash was actually selected. A check is the same as a pre or post-condition except that a check is evaluated at the location it is found within a scenario. Checks can only be used within scenarios, they are not supported within other types of constructs. The scenario grammar also includes an example of a belief nested within the scenario grammar. If the check fails, the ***ValidCash*** belief will be shown as failed. Once a cash is selected for checkout, the grammar indicates that the customer is added to the cash's queue via the ***Cash::AddCustomer()*** responsibility. Recall that the ***context*** keyword is used to indicate the current instance of the customer contract is to be used. The next event in the grammar occurs when the ***Cash::NextCustomer()*** responsibility is executed and the result of the execution is the current customer. Let's elaborate on this statement, the ***BuyItems()*** scenario is suspended after the customer is added to the cash's queue. The scenario continues once the customer is removed from the queue (via ***NextCustomer()***). Other events which occur on the Cash contract are ignored by the scenario. However if an event occurs on the Customer contract which is out of order within the scenario, the scenario fails. That is, only events which occur out of order within the context of the scenario (the Customer contract in this case), cause a scenario to fail. The rationale here is that the Customer contract should know which of its own events should or should not occur, but the Customer should not have to specify events which take place within other contracts. Next, the grammar indicates that the ***RemoveFood()*** responsibility is executed to take the items out of the customer's cart, and to place them onto the cash register for scanning. Once each item is processed, the ***Pay()*** responsibility is executed to pay for the items purchased. Finally, the customer leaves the store via the ***LeaveStore()*** responsibility. The scenario completes upon successful execution of the special ***finalize()*** responsibility.

Within the Customer contract, we have used a contract variable to keep track of the number of items which the customer has placed into his/her cart and purchased. This is an example of using a contract variable to record a metric value. In order for the metric value to be accessible and processed the metric section named ***CartSize()*** is provided. The value returned by the ***CartSize()*** metric is the value stored within the ***cartSize*** contract variable.

Following the metric definition, the Customer contract contains a Metrics section. Each contract may contain at most one Metrics section. A Metrics section is used to call user defined dynamic extensions to process and report, via the contract evaluation report, information gathered while the IUT and corresponding scenarios were being executed. In the case of the Customer contract, the Metrics section specifies a single ***ReportAll*** statement. A ***ReportAll*** statement can only be used within a Metrics section to write directly to the contract evaluation report. ***ReportAll*** statements are similar to the C printf statement. They contain a string to be written to the contract evaluation report, followed by a

variable length argument list. Arguments are replaced within the string as specified by the ***{0}*** notation. The zero-based argument index is specified between matching braces. The single argument in the Customer contract is calculated by calling a dynamic extension named ***AvgMetric()***. The dynamic extension will be supplied within the Core namespace. As the extension name suggests, it is used to calculate average metric values across multiple instances of the same contract. ***AvgMetric()*** takes a single argument which is the number of items which were stored within the customer's cart. Before continuing with Customer contract, let's look at the metric reporting mechanism in more detail. Individual metrics are gathered using various techniques while the IUT is being executed. Metric values are reported via the use of Metric sections. Metric sections exist at the contract instance level. That is, each Customer instance contains a ***CartSize()*** metric which will return a unique value for each customer. Once the execution of the IUT is complete, and the metric values have been calculated, the Metrics block of each contract is executed, if present. The Metrics block is executed a single time. However, the report statements within the Metrics section determine if a given metric should be reported per-instance, or per-contract. For example, in the case of the Customer contract, we only want a single line on the contract evaluation report indicating the average of all of the customer cart sizes, and not a list of each cart size. This is accomplished by using the ***ReportAll*** statement. If we wanted a list of each cart size, the ***Report*** statement would be used. An example of the ***Report*** statement will be illustrated in the next contract. Additional details of metric evaluation and reporting will be illustrated in a future document. Returning to the Customer contract, the ***ReportAll*** statement will display the average number of items placed within all customer shopping carts.

The final section of the Customer contract is the ***Exports*** section. The first export is the ***tFoodItem*** type. The ***conforms*** keyword indicates that the IUT type that is bound to the ***tFoodItem*** type, will automatically have the Item contract applied to it. That is, the Item contract will also be bound to the IUT type that is bound to the ***tFoodItem*** type. The ***tFoodItem*** binding also contains a binding rule. The binding rule states that the IUT type that is bound to the ***tFoodItem*** type is the same type that is bound to the ***tItem*** type specified within the Store contract. The Store contract will be presented shortly. If the binding for the ***Store::tItem*** type has already been performed, the ***Customer::tFoodItem*** binding will be performed automatically. Next, the ***tCash*** type binding specifies that the Cash contract is applied to the corresponding IUT type, and that the ***Store::tCash*** type will be bound to the same IUT type. The *tFoodContainer* type is bound to an IUT type, that will have the BoundedContainer contract bound to it. It should be noted, that the values of the generic contract parameters for the BoundedContainer contract must be specified at this point, as shown above. Missing generic contract parameters will result in a compile-time error. Next, the ***tStore*** type is bound to an IUT type which will have the Store contract bound to it. It should be noted, that cyclic bounding rules are permitted, but not required. Examples of cyclic binding rules will be shown in the last contract. Finally, a field binding is specified to bind the ***foodContainer*** symbol to a field within the customer IUT type that is of the type bound to the ***tFoodContainer*** symbol. As previously stated, the field binding ensures that the check specified within the ***Structure*** section of the contract is preserved. Thus, the ***Structure*** section of the Customer contract is not required.

The ***Exports*** section completes the definition of the Customer contract. We will now examine the Cash contract, which is applied to each cash register.

The *Cash* Contract

The Cash contract represents a single cash register found within a store. Customers use the cash registers to checkout. The Cash contract listing is as follows:

```
Import Core;

Using DaveArnold.Collections;

Namespace DaveArnold.Examples.GroceryStore
{
	Contract Cash
	{
		Set Integer customer_times;
		Value Integer processed_customers;

		Observability Boolean IsOpen();
		Observability Boolean HasCustomers()
		{
			!queue.IsEmpty();
		}

		Responsibility new()
		{
			customer_times.Init();
			processed_customers = 0;
		}

		Responsibility Open()
		{
			Belief ~1("Cannot open a cash that is already open")
			{
				Pre(#"IsOpen() == false");
			}

			Post(#"IsOpen() == true");
		}

		Responsibility Close()
		{
			Belief ~2("Cannot close a cash that is not open")
			{
				Pre(#"IsOpen() == true");
			}
			Belief ~3("Cannot close a cash that has customers")
			{
				Pre(#"HasCustomers() == false");
			}

			Post(#"IsOpen() == false");
		}
```

```
Responsibility AddCustomer(tCustomer customer)
{
	Pre(#"customer != null");
	Pre(#"IsOpen() == true");
	Pre(#"queue.HasItem(customer) == false");
	Pre(#"customer.BuyItems().cash == context");

	Post(#"queue.HasItem(customer) == true");
	Post(#"queue.Back() == customer");
}

Responsibility tCustomer NextCustomer()
{
	Pre(#"IsOpen() == true");
	Pre(#"HasCustomers() == true");
	PreSet(#"~1~ = queue.Front()");

	Post(#"~result~ != null");
	Post(#"~1~ == ~result~");
	Post(#"queue.HasItem(~result~) == false");
}

Responsibility ProcessCustomer(tCustomer customer)
{
	Belief ~4("The customer must be valid")
	{
		Pre(#"customer != null");
		Pre(#"queue.HasItem(customer) == false");

		Belief ~5("The customer hasn't paid yet")
		{
			Pre(#"customer.HasPaid() == false");
		}
	}
	Belief ~6("The customer should have selected this cash")
	{
		Pre(#"customer.BuyItems().cash == context");
	}
	customer.RemoveItems(),
	customer.Pay(),
	atomic
	{
		customer_times.Add(queue.TimeInContainer(customer));
		processed_customers = processed_customers + 1;

		Belief ~6("The customer is done")
		{
			Check(#"customer.HasPaid() == true");
		}
		customer.LeaveStore(dontcare)
	};
}
```

```
Scenario RunCash()
{
	Value Integer count;

	Trigger(new());
	(
		atomic
		{
			count = 0;
			Open()
		},
		(
			atomic
			{
				count = count + 1;
				AddCustomer(dontcare)
			}
			|
			atomic
			{
				count = count - 1;
				ProcessCustomer(NextCustomer())
			}
		)*,
		Close()
	)*;
	Terminate(finalize());

	Belief ~7("All customers have been processed")
	{
		Post(#"count == 0");
	}
}

Metric Set Integer WaitingTimes()
{
	context.customer_times;
}

Metric Integer ProcessedCustomers()
{
	context.processed_customers;
}

Metrics
{
	Report("Avg customer waiting time: {0}",
		AvgMetric(WaitingTimes()));
	Report("Max customer waiting time: {0}",
		MaxMetric(WaitingTimes()));
	Report("Min customer waiting time: {0}",
		MinMetric(WaitingTimes()));
	Report("Number of Customers: {0}", ProcessedCustomers());
}
```

```
            Exports
            {
                  Type tCustomer conforms Customer;
                  Type tQueue conforms BoundedQueue<tCustomer, 100>;
                  Field queue tQueue;
            }
      }
}
```

As with the Customer contract, the Cash contract begins with the standard import statement, DaveArnold.Collections reference, and a namespace declaration to place the contract within the DaveArnold.Examples.GroceryStore namespace. The Cash contract itself begins with the declaration of a contract variable called ***customer_times***. Unlike previous ***Value*** contract variable declarations, the ***customer_times*** contract variable is defined using the ***Set*** keyword. As the name suggests the ***Set*** keyword creates a set of the type specified following the ***Set*** keyword. Contract variables defined using the ***Set*** keyword have the following operations:

- ***Init()*** – Initializes the set and clears any elements which may be in the set.
- ***Add(x)*** – Adds the element x to the set. If x is already in the set, an additional element is added. That is, a set may contain duplicate elements.
- ***Remove(x)*** – Removes the first instance of x found within the set.
- ***RemoveAll(x)*** – Removes all instances of x found within the set.
- ***Contains(x)*** – Returns true if at least one instance if x is stored within the set.
- ***Count(x)*** – Returns the number of instances of x stored within the set.
- ***Count()*** – Returns the size of the set.

The contract will use the ***customer_times*** contract variable to record the amount of time each customer waits in the queue. The ***processed_customers*** contract variable will record the number of customers processed by the cash.

The Cash contract continues with the declaration of two observability methods. The first, ***IsOpen()*** indicates if the cash is open or closed. Only an open cash can be selected by customers. The second observability method ***HasCustomers()*** will indicate if the cash has customers waiting in its queue. The ***IsOpen()*** observability method will be bound to a method within the IUT, where the ***HasCustomers()*** observability method's value will be determined by checking to see if the underlying queue is not empty.

The Cash contract contains a ***new()*** responsibility to initialize the contract variables. Sets are initialized using the ***Init()*** set operation. Each set must be initialized prior to use. Following the ***new()*** responsibility, ***Open()*** and ***Close()*** responsibilities are used to open and close a cash respectively. The ***Open()*** responsibility ensures that the cash is not already open in a precondition and contains a post-condition to ensure that the cash was actually opened upon completion of the bounded IUT method. The ***Close()*** responsibility performs the reverse action, containing a precondition to ensure that the cash

is open, has no customers (cashes cannot be closed if they have customers waiting), and a post-condition to ensure that the bounded IUT method actually closes the cash.

Next, the ***AddCustomer()*** responsibility adds a new customer to the queue leading up to the cash. The responsibility checks to see that a valid customer has been passed in, the cash is open, and that the customer is not already in the queue. In addition, the exported scenario variable ***cash*** from the ***Customer::BuyItems()*** scenario is used to ensure that the cash to which the customer is being added to, is the cash that the customer selected to check out at. The syntax of this precondition should be noted. The scenario ***BuyItems()*** is referenced, and within it, the ***cash*** scenario variable is accessed. If there is currently no active instance of the scenario the precondition will automatically fail. Similarly, in the case where there is more than once instance of the scenario active for the specified customer instance, the precondition will automatically fail, unless each scenario has the same value in their respective ***cash*** variables. Once the bounded IUT method has executed, the ***AddCustomer()*** responsibility checks to see that the customer has been added to the queue, and that the customer is located at the back of the queue.

The ***NextCustomer()*** responsibility is used to get the first customer in the queue. That is, the customer in the front of the line to check out. The responsibility begins with preconditions that ensure that the cash is open and contains customers. The ***PreSet*** construct is used to store the customer that is located at the front of the queue. Once the bounded IUT method has completed execution, the post-conditions ensure that a customer was picked, the customer was the one that was at the front of the queue, before the IUT method executed, and that the customer is no longer in the queue.

The ***ProcessCustomer()*** responsibility shows the use of a different type of responsibility. As with observability methods, responsibilities can either be bound to an actual IUT method, or can be defined in terms of other responsibilities. The ACL compiler detects which type of responsibility is specified and will request bindings only if needed. There is no special syntax to distinguish between the different responsibilities. The non-bounded responsibilities can be seen as a scenario section that does not have a trigger or a terminate condition, but does have a section of scenario grammar in the responsibility body. The ***ProcessCustomer()*** responsibility begins by ensuring that a valid customer has been selected, and that customer has already been removed from the queue. Additional preconditions also check that the customer has yet to pay, and has selected this cash to check out at. Once the preconditions are completed, the grammar specifies that the ***Customer::RemoveItems()*** responsibility is executed followed by the ***Customer::Pay()*** responsibility. Following that the ***atomic*** keyword is used to perform several operations and checks. The first operation, is used to add the amount of time that the customer spent waiting in line (in the queue) to the ***customer_times*** set. The amount of time is acquired by using the ***TimeInContainer()*** metric defined within the BoundedContainer contract. Next, the ***processed_customers*** contract variable is incremented to indicate that another customer has been processed by this cash. A check to ensure that the customer has paid follows, and finally the grammar specifies the ***Customer::LeaveStore()*** responsibility to allow the customer to leave the store. The grammar is terminated with a semicolon (;) to indicate where the responsibility terminates. Post-conditions may be placed after the grammar if needed. The ***ProcessCustomer()*** responsibility does not require any post-conditions.

Following the specification of the responsibilities that compose the Cash contract, the ***RunCash()*** scenario is defined. The ***RunCash()*** scenario specifies the entire life-cycle of a single cash register. The scenario begins by defining a scenario variable named ***count*** that will be used to keep track of the number of customers that are located within the cash's queue. The scenario is triggered upon instantiation of the type bound to the Cash contract. The ***atomic*** keyword is used to set the initial value of the customer counter and to open the cash. The scenario grammar then states that while the cash is open there are two possible events. The first event is that a customer is added to the queue. The ***dontcare*** keyword is used to indicate that the actual customer that is added to the queue is not important form the ***RunCash()*** scenario's point of view. An ***atomic*** block is used to increment the ***count*** variable at the same time as the customer is added to the queue. The second event that can occur is that a customer can be removed from the queue. When a customer is removed from the queue via the ***NextCustomer()*** responsibility, the ***ProcessCustomer()*** responsibility is used to specify the grammar of events that occurs on the selected customer. An ***atomic*** block is also used to decrement the ***count*** variable at the same time that the customer is removed from the queue. The star (*) operator, indicates that any combination of the two events can occur zero or more times. Note that even though the grammar allows the ***NextCustomer()*** responsibility to be executed before a customer has been added to the queue, it is impossible because of the preconditions associated with the ***NextCustomer()*** responsibility. The scenario grammar concludes with the closing of the cash via the ***Close()*** responsibility. The outer star (*) operator allows for the cash to be opened and closed multiple times. The scenario terminates when the cash object is destroyed. Once the scenario has completed, a post-condition is used to ensure that there were no customers left in the queue when the cash closed.

Following the ***RunCash()*** scenario two metrics are defined. The first metric, ***WaitingTimes()*** returns a set of integers which contains all of the waiting times for the customers which were processed by this cash. The ***WaitingTimes()*** metric is implemented by returning the ***customer_times*** contract variable. The second metric, ***ProcessedCustomers()*** is used to return the number of customers which have passed through the cash. ***ProcessedCustomers()*** is implemented by returning the value of the ***processed_customers*** contract variable.

The Cash contract contains a Metrics section. The Metrics section begins by using the ***AvgMetric()***, ***MaxMetric()***, and ***MinMetric()*** dynamic extensions, which are all located within the Core namespace to report the average, maximum, and minimum customer waiting times for each cash via a ***Report*** statement. The Metrics section also reports the number of customers processed by each cash register.

Finally, the Cash contract contains an ***Exports*** section to specify the symbols that need to be bound to types within the IUT. In addition, binding rules are specified. The first export is for the ***tCustomer*** type that must conform to the Customer contract. The second export binds the ***tQueue*** type of a type within the IUT that conforms to the BoundedQueue contract that contains the generic contract parameters which specify that the queue contains customers and that it has a maximum size of 100 customers. The final export is to bind the field within the IUT that represents the actual queue to the ***queue*** symbol for use within the contract.

With the Cash contract specified, the only the specification of the Store contract remains.

The *Store* Contract

The Store contract is the heart of the grocery store system. It represents the entire grocery store, and is the main contract for the system. The Store contract listing is specified as follows:

```
Import Core;

Using DaveArnold.Collections;

Namespace DaveArnold.Examples.GroceryStore
{
	MainContract Store
	{
		Value Integer openCashes;

		Structure
		{
			Belief HasItemsToSell("The store holds items to sell")
			{
				HasMemberOfType(tFoodContainer);
			}
			Belief HasCash("The store has cash registers")
			{
				HasMemberOfType(tCashContainer);
			}
		}

		Observability Integer OpenCashes();
		Observability Boolean IsOpen();

		Observability Boolean HasCash(tCash x)
		{
			cashContainer.HasItem(x);
		}

		Responsibility new()
		{
			context.openCashes = 0;
			Post(#"foodContainer.IsEmpty() == true");
			Post(#"cashContainer.IsEmpty() == true");
			Post(#"IsOpen() == false");
		}

		Responsibility finalize()
		{
			Pre(#"foodContainer.IsEmpty() == true");
			Pre(#"cashContainer.IsEmpty() == true");
			Pre(#"context.openCashes == 0");
			Pre(#"IsOpen() == false");
		}
```

```
Invariant OpenCashNumber()
{
	Inv(#"context.openCashes >= 0");
	Inv(#"context.openCashes == OpenCashes()");
}

Responsibility Open()
{
	Pre(#"foodContainer.IsEmpty() == true");
	Pre(#"cashContainer.IsEmpty() == true");
	Pre(#"context.openCashes == 0");
	Pre(#"IsOpen() == false");

	Post(#"foodContainer.IsEmpty() == false");
	Post(#"cashContainer.IsEmpty() == false");
	Post(#"context.openCashes == 0");
	Post(#"IsOpen() == true");
}

Responsibility Close()
{
	Pre(#"IsOpen() == true");
	Pre(#"foodContainer.IsEmpty() == false");
	Pre(#"cashContainer.IsEmpty() == false");

	Post(#"IsOpen() == false");
	Post(#"foodContainer.IsEmpty() == true");
	Post(#"cashContainer.IsEmpty() == true");
}

Responsibility OpenCash(tCash c)
{
	Pre(#"IsOpen() == true");
	Pre(#"c != null");
	Pre(#"cashContainer.HasItem(c) == true");
	Pre(#"c.IsOpen() == false");

	context.openCashes = context.openCashes + 1;
	Post(#"cashContainer.HasItem(c) == true");
	Post(#"c.IsOpen() == true");
}

Responsibility CloseCash(tCash c)
{
	Pre(#"IsOpen() == true");
	Pre(#"c != null");
	Pre(#"c.IsOpen() == true");
	Pre(#"cashContainer.HasItem(c) == true");

	context.openCashes = context.openCashes - 1;
	Post(#"cashContainer.HasItem(c) == true");
	Post(#"c.IsOpen() == false");
}
```

```
Responsibility AddFood(tFoodItem item)
{
	Pre(#"IsOpen() == true");
	Pre(#"item != null");
	Pre(#"foodContainer.HasItem(item) == false");

	Post(#"foodContainer.HasItem(item) == true");
}

Responsibility RemoveFood(tFoodItem item)
{
	Pre(#"IsOpen() == true");
	Pre(#"item != null");
	Pre(#"foodContainer.HasItem(item) == true");

	Post(#"foodContainer.HasItem(item) == false");
}

Scenario OpenAndCloseCash()
{
	once Value tCash x;
	Trigger(OpenCash(x));
	Terminate(CloseCash(x));
}

Scenario AddAndRemoveFood()
{
	once Value tFoodItem x;
	Trigger(AddFood(x));
	Terminate(RemoveFood(x));
}
```

```
Scenario RunStoreCash()
{
	Value Integer count;

	Trigger(new());
	(
		atomic
		{
			count = 0;
			Open()
		},
		(
			atomic
			{
				count = count + 1;
				OpenCash(dontcare)
			}
			|
			atomic
			{
				count = count - 1;
				CloseCash(dontcare)
			}
		)*,
		Close()
	)*;
	Terminate(finalize());

	Belief CashesAllClosed("No cashes are still open")
	{
		Post(#"count == 0");
	}
}
```

```
Scenario RunStoreFood()
{
	Value Integer count;

	Trigger(new());
	(
		atomic
		{
			count = 0;
			Open()
		},
		(
			atomic
			{
				count = count + 1;
				AddFood(dontcare)
			}
			|
			atomic
			{
				count = count - 1;
				RemoveFood(dontcare)
			}
		)*,
		Close()
	)*;
	Terminate(finalize());

	Belief FoodAllGone("No food is still in the store")
	{
		Post(#"count == 0");
	}
}
```

```
Scenario MainStore()
{
	Trigger(new());
	(
		Open(),
		(AddFood(dontcare))+,
		OpenCash(dontcare),
		(

			OpenCash(dontcare)
			|
			CloseCash(dontcare)
			|
			parallel
			{
				once Value tCash cash;
				once Value tCustomer c;
				c = newInstance tCustomer;
				c.EnterStore(context),
				(c.AddFood(dontcare))+,
				atomic
				{
					cash = c.SelectCash(context),
					Belief ValidCash(
					"A valid cash will be selected”)
					{
						Check(#"cash != null");
					}
				},
				cash.AddCustomer(c),
				c == cash.NextCustomer(),
				c.RemoveFood(),
				c.Pay(),
				c.LeaveStore(context)
			}
		)+,
		(RemoveFood(dontcare))+,
		Close()
	)+;
	Trigger(finalize());
}
```

```
            Exports
            {
                  Type tFoodItem conforms Item
                  {
                        Customer::tFoodItem;
                  }
                  Type tCash conforms Cash
                  {
                        Customer::tCash;
                  }
                  Type tCustomer conforms Customer
                  {
                        Cash::tCustomer;
                  }
                  Type tFoodContainer conforms
                                    BoundedContainer<tFoodItem, 10000>;
                  Type tCashContainer conforms BoundedContainer<tCash, 10>;
                  Field cashContainer tCashContainer;
                  Field foodContainer tFoodContainer;
            }
      }
}
```

The Store contract is defined using the ***MainContract*** keyword instead of the ***Contract*** keyword. ***MainContract*** is used to denote that this contract is the entry point to the contract system. Normally, contracts are not applied or bound until specified in a corresponding ***Exports*** section. However, a contract specified using the ***MainContract*** keyword is automatically bound to a type within the IUT. Each contract project (a collection of one or more contracts), must contain at least one ***MainContract***. Unlike conventional programming languages, a contract project can contain any number of ***MainContract*** declarations. Put another way, regular contracts are not automatically bound to an IUT type, where main contracts are bound to an IUT type automatically. With the exception of binding, main contracts behave exactly the same way and have the same functionality as normal contracts.

The Store contract begins with the declaration of a single contract variable named ***openCashes***. The variable will be used to keep track of the number of open cash registers that the store currently has in operation.

The ***Structure*** section uses the ***HasMemberOfType()*** static check to ensure that the IUT type bound to the store contains a container for storing the food items that are in the store, as well as a container for storing the cash registers that are located within the store. As previously stated, the field bindings located within the ***Exports*** section, ensure that the static checks specified within the ***Structure*** section exist, however if the fields are not needed within the contract, then the static checks should be used. In addition, it would create a more complete contract if the structural requirements were explicitly specified.

Following the ***Structure*** section three observability methods are defined. The first, ***OpenCashes()***, is bound to the IUT and returns the number of open cashes as reported by the IUT. The second observability method ***IsOpen()***, is bound to the IUT and returns true if the store is in an open state, false otherwise. The third observability method ***HasCash()*** is used to determine if the grocery store contains a given cash register. This observability method is implemented by using the result obtained from the ***BoundedContainer::HasItem()*** observability method.

Next, the special ***new()*** responsibility is defined. The responsibility sets the initial value of the ***openCashes*** contract variable, and defines post-conditions to ensure that the store is in a valid initial state and not yet open. The special ***finalize()*** responsibility performs the reverse action and tests to make sure that the store is closed and does not contain any cash registers or food items before it is destroyed. The ***finalize()*** responsibility also ensures that no cash registers were left in an open state.

A single invariant is defined named ***OpenCashNumber()***, the purpose of the invariant is to ensure that the number of open cash registers is never a negative amount and that the number of open cash registers according to the Store contract, matches the value returned by the IUT.

The ***Open()*** responsibility handles the opening of the actual store. The responsibility is defined by a set of preconditions that ensure that the store is not already open, does not have any open cashes, and that the store has no cash registers, or food items within the store. The ***Open()*** responsibility then has a set of post-conditions that ensure that once the store has been opened, it contains some food items, and cash registers, yet none of the cash registers are opened (this is accomplished by a separate responsibility).

The ***Close()*** responsibility handles closing of the store. The responsibility is defined by a set of preconditions that ensure that the store is actually open, and that the store has food items and cashes. That is, the store is in a valid open state. The ***Close()*** responsibility's post-conditions make sure that the store is no longer open, and that it does not contain any food items or cash registers.

The ***OpenCash()*** responsibility is used to open the specified cash register for customer use. The preconditions check to make sure that the store is open, and that the cash register provided is valid, part of the store, and not already open. The ***OpenCash()*** responsibility next increments the ***openCashes*** contract variable, and finally via the post-conditions ensures that the cash is still part of the store and that it has been opened.

The ***CloseCash()*** responsibility is used to close the specified cash register. The preconditions check to ensure that the store is open, and that a valid already open cash register that is part of the store has been provided. The ***CloseCash()*** responsibility then decrements the ***openCashes*** contract variable, and finally the post-conditions ensure that the cash register is still part of the store and that is has actually been opened.

The ***AddFood()*** and ***RemoveFood()*** responsibilities are used to add and remove food items from the store respectively. Each of the responsibilities contain two common preconditions, one to make sure that the store is open, and another to make sure the given food item is valid. The ***AddFood()***

responsibility checks to make sure that the item being added is not already in the store, and then specifies a post-condition to ensure that the item was actually added. The ***RemoveFood()*** responsibility performs the opposite action by ensuring that the item to be removed is currently located within the store, and has a post-condition to ensure that the item has been successfully removed.

The Store contract contains four scenarios, of which two overlap. We will now examine each scenario in detail. The first scenario, ***OpenAndCloseCash()*** is used to ensure that each cash register that is opened, is closed at some point during the execution of the system. Each time a new cash register is opened via the evaluation of the ***OpenCash()*** responsibility a new instance of the scenario is triggered. When the corresponding cash is closed via the evaluation of the ***CloseCash()*** responsibility the scenario is terminated. Any scenarios that are still executing when the application terminates, indicates a cash register that was opened but never closed.

Likewise, the ***AddAndRemoveFood()*** scenario is used to ensure that each item of food put into the store is also removed at some point. The scenario is triggered when a new food item is added to the store via the evaluation of the ***AddFood()*** responsibility. The scenario terminates when that same food item is removed from the store. Any un-terminated scenario indicate food that is either still in the store or has been removed incorrectly (i.e. stolen).

The ***RunStoreCash()*** scenario defines a scenario grammar for the execution of cash openings and closings. The scenario is triggered when a new instance of the store is created. The scenario begins with the opening of the store, followed by a sequence of zero or more cash openings and closings. The scenario grammar completes when the store closes. The scenario contains a post-condition to ensure that the same number of cashes have been opened and closed. The outer star (*) operator allows several openings and closings of the store to occur on one store instance.

The ***RunStoreFood()*** scenario is analogous to the ***RunStoreCash()*** scenario, except that it checks the addition and removal of food items, rather than opening and closings of cashes. Both scenarios have the same triggering events, but because their scenario grammars are different both scenarios will execute concurrently. Additional information on scenario execution will be presented later.

In order to illustrate a centralized scenario, the Store contract contains a scenario called ***MainStore()***. The ***MainStore()*** scenario overlaps with other scenarios which have been already been specified. However the ***MainStore()*** scenario illustrates additional ACL keywords and the flexibility for the specification of scenarios. ***MainStore()*** is triggered when the IUT type bound to the Store contract is instantiated and a constructor has completed executing. The first responsibility the scenario grammar expects is ***Open()***. As the name suggests, ***Open()*** is used to open the grocery store. Once the store has been opened, one or more food items are added to the store via the ***AddFood()*** responsibility. The individual food items are not of interest to the high-level store scenario and thus are referenced via the ***dontcare*** keyword. Once the food items have been added to the store, the grammar specifies that a single cash must be opened. Again the ***dontcare*** keyword is used to indicate that we do not care about the specific cash that has been opened. Next, the scenario grammar specifies that one of three actions can occur. The first is to open a new cash via the ***OpenCash()*** responsibility. The second performs the

reverse operation, that is to close a cash via the ***CloseCash()*** responsibility. The third action which can occur involves the introduction of the ***parallel*** keyword. The ***parallel*** keyword is used to specify that the scenario grammar specified within the brace brackets can be seen as a single responsibility. That is, multiple instances of the same scenario grammar section may be executing in parallel at any given time. A parallel section can also be viewed as a sub-scenario within the main scenario, where there can be any number of sub-scenarios active at any one time. In the case of the ***MainStore()*** scenario, several customers could be in the store buying items. The parallel section begins with the declaration of two scenario variables named ***cash*** and ***c***. The ***cash*** scenario variable will be of the IUT type bound to the ***tCash*** symbol. The ***c*** scenario variable will be of the IUT type bound to the ***tCustomer*** symbol. The first scenario grammar element within the parallel section is called when a new customer is created. The first scenario grammar element within a parallel section, can be viewed as the sub-scenario trigger. The grammar element also introduces the ***newInstance*** keyword. The ***newInstance*** keyword specifies that the scenario grammar continues when a new instance of the IUT type bound to the ***tCustomer*** symbol is created. Next, the customer enters the store and adds one or more food items to his/her cart via the ***Customer::AddFood()*** responsibility. The customer then selects a cash and is added to the end of the queue. The scenario continues, when the customer gets to the front of the queue, and removes his/her food items from the cart via the ***Customer::RemoveFood()*** responsibility. Finally, the customer pays for the items and leaves the store. The ***MainStore()*** scenario concludes with the removal of the food items, and the closing of the store via the ***Close()*** responsibility. The outer plus (+) operator indicates that the store can be opened and closed one or more times during the execution of the IUT.

The Store contract does not define any metrics. There are several metrics which could be reported by the store, such as the number of food items sold, and the number of open cashes. The addition of these metrics would require a dedicated contract variable, and corresponding increment statements within various responsibilities. As the implementation of such metrics does not introduce any additional features of the ACL or contract runtime, they have been omitted.

Finally, the Store contract contains an ***Exports*** section. As previously discussed, the ***Exports*** section is used to specify the bindings required for the contract. The first export line, binds the ***tFoodItem*** symbol to an IUT type which conforms to the previously defined Item contract. In addition, a binding rule is specified to indicate that the ***tFoodItem*** referenced in the Customer contract, is the same as the ***tFoodItem*** referenced here. The second export line, binds the ***tCash*** symbol to an IUT type which conforms to the previously defined Cash contract. A binding rule is also specified to match the ***tCash*** type referenced in the Customer contract. The third export line, binds the ***tCustomer*** symbol to an IUT type which conforms to the previously defined Customer contract. A binding rule is specified to match the ***tCustomer*** type referenced in the Cash contract. Next, the ***tFoodContainer*** and ***tCashContainer*** symbols as bound to IUT types that conform to the BoundedContainer contract using the specified generic contract parameters. Finally, exports to the two internal container fields are specified so that the containers can be referenced within the contract.

With the completion of the Store contract, we have presented a set of six contracts which define the grocery store case study. Such contracts create a testable model of the grocery store case study. With the contracts defined, we will now examine the next steps of the framework.

Step 2 – Contract Compilation

The first step in processing the contracts listed above is to create a contract project. As previously stated, contract projects will be implemented as a project type within Visual Studio 2008, and will consist of the following elements:

- One or more contracts
- An IUT on which to execute the static and dynamic checks and to execute for scenario validation
- A set of bindings to bind the contract to the IUT

The entire contract project is sent to the ACL compiler which tokenizes, and parses the contracts. The compiler flattens any contract inheritance, performs generic parameter substitution, and ensures that all identifiers can be resolved. That is, the ACL compiler ensures that the contract syntax is correct, and all required extensions have been located.

The ACL compiler does not perform binding operations. The previous ACL compiler performed binding operations as needed, during the processing of each contract. In order to incorporate the binding rules and contract conformance, all binding operations are performed once the ACL compiler has checked syntax, and semantics. At this point the contracts are represented by an abstract syntax tree. Each requested binding has been mapped to any other bindings based on the specified binding rules. The next step is to perform the actual bindings.

Step 3 – Bindings

As the name suggests, the types, methods, and fields specified within the ***Exports*** section of a contract are bound to IUT counterparts. Each binding selection is stored within the contract project so that bindings do not need to be specified each and every time the IUT is run against the contract. Of course, any change in either the IUT or contract will require bindings in the affected area to be re-specified. In addition, a tree like view will be present in Visual Studio 2008 to graphically show the bindings between the contract's structure and the structural elements within the IUT. The contract developer is then able to view, edit, and reset the binding information.

For the purposes of this example we will assume that no previous binding history exists. The following chart illustrates the steps required the bind the six contracts that we defined in Step 1. The bindings in the chart below are specified in the order a binding query would be required from the contract developer. For each binding query a structural tree representation of the IUT is presented for a corresponding selection. Depending on the context only a portion of the IUT's structure would be presented to the contract developer. For example, in the case where a field within a given type is requested, only the fields which reside within given type will be displayed.

The binding algorithm begins with the first ***MainContract*** and binds the contract to a type within the IUT. Next bindings for observability methods and responsibilities contained within the contract are preformed. If an observability or responsibility has a parameter or return type, which contains an exported symbol that is not yet bound, binding will be performed for the parameters and return values before the actual observabillity or responsibility has been bound. Finally, all export lines found within the contract's ***Exports*** section are bound.

When binding observability methods, an exact parameter and return type match is required. In the case where the requested observability method does not exist within the IUT, the contract developer is able to specify a literal value for the result of an observability instead of an IUT method. The rationale for this feature is that in some cases the observability method may not be needed from the IUT's point of view. For an example, consider the BoundedContainer<Type, Size> contract. It would be possible to implement a BoundedContainer using an unbounded data structure. As such the BoundedContainer<Type, Size>::IsFull() observability method could be hard-wired to yield a false value. When a hard-wired binding is used, the compiler will issue a warning to notify the contract developer of such hard-wiring.

When binding responsibilities, an actual IUT method must be specified. However to allow for maximum flexibility responsibilities which do not specify a return type may be bound to an IUT method with any return type (including void). Also, when a responsibility specifies a parameter set, that responsibility can be bound to any IUT method which has at least the requested parameters. That is, a parameter map will be created between the IUT method's parameters and the parameters specified by

the responsibility. Any additional parameters specified by the IUT method are simply ignored by the contract.

The following binding chart, illustrates the bindings performed for the six contracts specified in Step 1. It may be easier to follow along with a printed copy of the contracts, while reading the chart.

Contract Symbol	Binding Action
Store	Can be bound to any type within the IUT. This binding is performed because the Store contract is defined using the ***MainContract*** keyword.
Store::OpenCashes()	Can be bound to any method within the type selected for the Store contract symbol. The IUT method must have a return type of Integer (or an implicit cast must exist), and can have any number of parameters. The parameters will be ignored by the contract. The IUT method that is selected, will automatically be instrumented to instruct the profiler to ensure that the IUT method is side-effect free.
Store::IsOpen()	Can be bound to any method within the type selected for the Store contract symbol. The IUT method must have a return type of Boolean (or an implicit cast must exist), and must take zero parameters, or the parameters must have default values (if supported by the IUT's implementation language). The IUT method that is selected will automatically be instrumented to instruct the profiler to ensure that the IUT method is side-effect free.
Store::Open()	Can be bound to any method within the type selected for the Store contract symbol. The IUT method may have any return type (but it is ignored by the contract), and can have any number of parameters. The parameters will be ignored by the contract. The binding cannot be the same as the binding used for Store::IsOpen(), as it would create a cycle when checking the pre and post-conditions.
Store:Close()	Can be bound to any method within the type selected for the Store contract symbol. The IUT method may have any return type (but it is ignored by the contract), and can have any number of parameters. The parameters will be ignored by the contract. The binding cannot be the same as the binding used for Store::IsOpen(), as it would create a cycle when checking the pre and post-conditions.
Store::tCash	The tCash parameter to the Store::OpenCash() responsibility can be bound to any type within the IUT. The selected type will instantly have the Cash contract bound to it.
Cash::IsOpen()	Can be bound to any method within the type selected for the Cash contract symbol. The IUT method must have a return type of Boolean (or an implicit cast must exist), and must take zero parameters, or the parameters must have default values (if supported by the IUT's implementation language). The IUT

	method that is selected will automatically be instrumented to instruct the profiler to ensure that the IUT method is side-effect free.
Cash::Open()	Can be bound to any method within the type selected for the Cash contract symbol. The IUT method may have any return type (but it is ignored by the contract), and can have any number of parameters. The parameters will be ignored by the contract. The binding cannot be the same as the binding used for Cash::IsOpen(), as it would create a cycle when checking the pre and post-conditions.
Cash:Close()	Can be bound to any method within the type selected for the Cash contract symbol. The IUT method may have any return type (but it is ignored by the contract), and can have any number of parameters. The parameters will be ignored by the contract. The binding cannot be the same as the binding used for Cash::IsOpen(), as it would create a cycle when checking the pre and post-conditions.
Cash::tCustomer	The tCustomer parameter to the Cash::AddCustomer() responsibility can be bound to any type within the IUT. The selected type will instantly have the Customer contract bound to it.
Customer::HasPaid()	Can be bound to any method within the type selected for the Customer contract symbol. The IUT method must have a return type of Boolean (or an implicit cast must exist), and must take zero parameters, or the parameters must have default values (if supported by the IUT's implementation language). The IUT method that is selected will automatically be instrumented to instruct the profiler to ensure that the IUT method is side-effect free.
Customer::tStore	The tStore parameter to the Customer::EnterStore() responsibility will be automatically bound to the Store type already selected. This is done, because we are currently in the process of binding the Store contract, so it is that binding that is used here as well.
Customer::EnterStore()	Can be bound to any method within the type selected for the Customer contract symbol. The IUT method may have any return type (but it is ignored by the contract), and must have at least one parameter matching the type bound to the tStore symbol. Any additional parameters will be ignored.
Customer::LeaveStore()	Can be bound to any method within the type selected for the Customer contract symbol. The IUT method may have any return type (but it is ignored by the contract), and must have at least one parameter matching the type bound to the tStore symbol. Any additional parameters will be ignored.
Customer::tFoodItem	The tFoodItem parameter to the Customer::AddFood() responsibility can be bound to any type within the IUT. The selected type will instantly have the Item contract bound to it.
Item::Price()	Can be bound to any method within the type selected for the Item

	contract symbol. The IUT method must have a return type of Real (or one for which an implicit conversion exists to Real). The IUT method can have any number of parameters, but they will be ignored by the contract.
Item::Name()	Can be bound to any method within the type selected for the Item contract symbol. The IUT method must have a return type of String (or one for which an implicit conversion exists to String [char* in C++]). The IUT method can have any number of parameters, but they will be ignored by the contract.
Item::SKU()	Can be bound to any method within the type selected for the Item contract symbol. The IUT method must have a return type of Integer (or one for which an implicit conversion exists to Integer). The IUT method can have any number of parameters, but they will be ignored by the contract.
Customer::AddFood()	Can be bound to any method within the type selected for the Customer contract symbol. The IUT method may have any return type, but it will be ignored by the contract. The IUT method must have at least one parameter which is of the same type that is bound to the tFoodItem symbol. Other parameters will be ignored.
Customer::RemoveFood()	Can be bound to any method within the type selected for the Customer contract symbol. The IUT method may have any return type, but it will be ignored by the contract. The IUT method can have any number of parameters, but they will be ignored by the contract.
Customer::Pay()	Can be bound to any method within the type selected for the Customer contract symbol. The IUT method may have any return type, but it will be ignored by the contract. The IUT method can have any number of parameters, but they will be ignored by the contract.
Customer::tCash	Will be automatically bound to the same IUT type as Store::tCash, as per the binding rule in the Customer contract.
Customer::SelectQueue()	Can be bound to any method within the type selected for the Customer contract symbol. The IUT method must have a return type matching the IUT type bound to the tCash type. The IUT method must have at least one parameter that matches the type bound to the tStore type. Other parameters will be ignored by the contract.
Customer::tFoodItem	Already bound.
Customer::tCash	Already bound.
Customer::tFoodContainer	The tFoodContainer type exported by the Customer contract can be bound to any type within the IUT. The selected type will instantly have the BoundedContainer<tFoodItem, 100> contract bound to it.
BoundedContainer<tFoodItem, 100>::IsFull()	Can be bound to any method within the type selected for the BoundedContainer<tFoodItem, 100> contract symbol. The IUT method must have a return type of Boolean (or an implicit cast

	must exist), and must take zero parameters, or the parameters must have default values (if supported by the IUT's implementation language). The IUT method that is selected will automatically be instrumented to instruct the profiler to ensure that the IUT method is side-effect free.
BoundedContainer<tFoodItem, 100>::IsEmpty()	Can be bound to any method within the type selected for the BoundedContainer<tFoodItem, 100> contract symbol. The IUT method must have a return type of Boolean (or an implicit cast must exist), and must take zero parameters, or the parameters must have default values (if supported by the IUT's implementation language). The IUT method that is selected will automatically be instrumented to instruct the profiler to ensure that the IUT method is side-effect free.
BoundedContainer<tFoodItem, 100>::ItemAt()	Can be bound to any method within the type selected for the BoundedContainer<tFoodItem, 100> contract symbol. The IUT method must have a return type matching the IUT type bound to the tFoodItem symbol, and must take exactly one parameter of type Integer. The IUT method that is selected will automatically be instrumented to instruct the profiler to ensure that the IUT method is side-effect free.
BoundedContainer<tFoodItem, 100>::HasItem()	Can be bound to any method within the type selected for the BoundedContainer<tFoodItem, 100> contract symbol. The IUT method must have a return type of Boolean (or an implicit cast must exist), and must take exactly one parameter whose type must match the IUT type bound the tFoodItem symbol. The IUT method that is selected will automatically be instrumented to instruct the profiler to ensure that the IUT method is side-effect free.
BoundedContainer<tFoodItem, 100>::Size()	Can be bound to any method within the type selected for the BoundedContainer<tFoodItem, 100> contract symbol. The IUT method must have a return type of Integer (or an implicit cast must exist), and must take zero parameters, or the parameters must have default values (if supported by the IUT's implementation language). The IUT method that is selected will automatically be instrumented to instruct the profiler to ensure that the IUT method is side-effect free.
BoundedContainer<tFoodItem, 100>::Add()	Can be bound to any method within the type selected for the BoundedContainer<tFoodItem, 100> contract symbol. The IUT method may have any return type, and must have at least one parameter whose type matches the IUT type bound to the tFoodItem symbol.
BoundedContainer<tFoodItem, 100>::Remove()	Can be bound to any method within the type selected for the BoundedContainer<tFoodItem, 100> contract symbol. The IUT method must have a return type which matches the IUT type bound to the tFoodItem symbol.
BoundedContainer<tFoodItem, 100>::Remove(T)	Can be bound to any method within the type selected for the BoundedContainer<tFoodItem, 100> contract symbol. The IUT

	method may have any return type, and must have at least one parameter whose type matches the IUT type bound to the tFoodItem symbol.
Customer::tStore	Already bound.
Customer::foodContainer	Can be bound to any field within the IUT type selected for the Customer contract, which is of the type bound to the tFoodContainer symbol.
Cash::AddCustomer()	Can be bound to any method within the type selected for the Cash contract symbol. The IUT method may have any return type, and must have at least one parameter whose type matches the IUT type bound to the tCustomer symbol.
Cash::NextCustomer()	Can be bound to any method within the type selected for the Cash contract symbol. The IUT method must have a return type that matches the IUT type bound to the tCustomer symbol. The method can have any number of parameters, but they will be ignored by the contract.
Cash::ProcessCustomer()	Is not bound, because the responsibility contains a scenario grammar.
Cash::tCustomer	Already bound.
Cash::tQueue	The tQueue type exported by the Customer contract can be bound to any type within the IUT. The selected type will instantly have the BoundedQueue<tCustomer, 100> contract bound to it.
BoundedQueue<tCustomer, 100>::IsFull()	Can be bound to any method within the type selected for the BoundedQueue<tCustomer, 100> contract symbol. The IUT method must have a return type of Boolean (or an implicit cast must exist), and must take zero parameters, or the parameters must have default values (if supported by the IUT's implementation language). The IUT method that is selected will automatically be instrumented to instruct the profiler to ensure that the IUT method is side-effect free.
BoundedQueue<tCustomer, 100>::IsEmpty()	Can be bound to any method within the type selected for the BoundedQueue<tCustomer, 100> contract symbol. The IUT method must have a return type of Boolean (or an implicit cast must exist), and must take zero parameters, or the parameters must have default values (if supported by the IUT's implementation language). The IUT method that is selected will automatically be instrumented to instruct the profiler to ensure that the IUT method is side-effect free.
BoundedQueue<tCustomer, 100>::ItemAt()	Can be bound to any method within the type selected for the BoundedQueue<tCustomer, 100> contract symbol. The IUT method must have a return type matching the IUT type bound to the tCustomer symbol, and must take exactly one parameter of type Integer. The IUT method that is selected will automatically be instrumented to instruct the profiler to ensure that the IUT method is side-effect free.
BoundedQueue<tCustomer, 100>::HasItem()	Can be bound to any method within the type selected for the BoundedQueue<tCustomer, 100> contract symbol. The IUT

	method must have a return type of Boolean (or an implicit cast must exist), and must take exactly one parameter whose type must match the IUT type bound the tCustomer symbol. The IUT method that is selected will automatically be instrumented to instruct the profiler to ensure that the IUT method is side-effect free.
BoundedQueue<tCustomer, 100>::Size()	Can be bound to any method within the type selected for the BoundedQueue<tCustomer, 100> contract symbol. The IUT method must have a return type of Integer (or an implicit cast must exist), and must take zero parameters, or the parameters must have default values (if supported by the IUT's implementation language). The IUT method that is selected will automatically be instrumented to instruct the profiler to ensure that the IUT method is side-effect free.
BoundedQueue<tCustomer, 100>::BackLocation()	Can be bound to any method within the type selected for the BoundedQueue<tCustomer, 100> contract symbol. The IUT method must have a return type of Integer (or an implicit cast must exist), and must take zero parameters, or the parameters must have default values (if supported by the IUT's implementation language). The IUT method that is selected will automatically be instrumented to instruct the profiler to ensure that the IUT method is side-effect free.
BoundedQueue<tCustomer, 100>::FrontLocation()	Can be bound to any method within the type selected for the BoundedQueue<tCustomer, 100> contract symbol. The IUT method must have a return type of Integer (or an implicit cast must exist), and must take zero parameters, or the parameters must have default values (if supported by the IUT's implementation language). The IUT method that is selected will automatically be instrumented to instruct the profiler to ensure that the IUT method is side-effect free.
BoundedQueue<tCustomer, 100>::Add()	Can be bound to any method within the type selected for the BoundedQueue<tCustomer, 100> contract symbol. The IUT method may have any return type, and must have at least one parameter whose type matches the IUT type bound to the tCustomer symbol.
BoundedQueue<tCustomer, 100>::Remove()	Can be bound to any method within the type selected for the BoundedQueue<tCustomer, 100> contract symbol. The IUT method must have a return type which matches the IUT type bound to the tCustomer symbol.
BoundedQueue<tCustomer, 100>::Remove(T)	Can be bound to any method within the type selected for the BoundedQueue<tCustomer, 100> contract symbol. The IUT method may have any return type, and must have at least one parameter whose type matches the IUT type bound to the tCustomer symbol.
Cash::queue	Can be bound to any field within the IUT type selected for the Cash contract, which is of the type bound to the tQueue symbol.
Store::OpenCash()	Can be bound to any method within the type selected for the

	Store contract symbol. The IUT method may have any return type, and must have at least one parameter whose type matches the IUT bound to the tCash symbol.
Store::CloseCash()	Can be bound to any method within the type selected for the Store contract symbol. The IUT method may have any return type, and must have at least one parameter whose type matches the IUT bound to the tCash symbol.
Store::tFoodItem	Automatically bound to the same IUT type as the Customer::tFoodItem symbol. This is based on a binding rule found within the Store contract.
Store::AddFood()	Can be bound to any method within the type selected for the Store contract symbol. The IUT method may have any return type, and must have at least one parameter whose type matches the IUT bound to the tFoodItem symbol.
Store::RemoveFood()	Can be bound to any method within the type selected for the Store contract symbol. The IUT method may have any return type, and must have at least one parameter whose type matches the IUT bound to the tFoodItem symbol.
Store::tFoodItem	Already bound.
Store::tCash	Already bound.
Store::tCustomer	Automatically bound to the same IUT type as the Cash::tCustomer symbol. This is based on a binding rule found within the Store contract.
Store::tFoodContainer	The tFoodContainer type exported by the Store contract can be bound to any type within the IUT. The selected type will instantly have the BoundedContainer<tFoodItem, 10000> contract bound to it.
BoundedContainer<tFoodItem, 10000>::IsFull()	Can be bound to any method within the type selected for the BoundedContainer<tFoodItem, 10000> contract symbol. The IUT method must have a return type of Boolean (or an implicit cast must exist), and must take zero parameters, or the parameters must have default values (if supported by the IUT's implementation language). The IUT method that is selected will automatically be instrumented to instruct the profiler to ensure that the IUT method is side-effect free.
BoundedContainer<tFoodItem, 10000>::IsEmpty()	Can be bound to any method within the type selected for the BoundedContainer<tFoodItem, 10000> contract symbol. The IUT method must have a return type of Boolean (or an implicit cast must exist), and must take zero parameters, or the parameters must have default values (if supported by the IUT's implementation language). The IUT method that is selected will automatically be instrumented to instruct the profiler to ensure that the IUT method is side-effect free.
BoundedContainer<tFoodItem, 10000>::ItemAt()	Can be bound to any method within the type selected for the BoundedContainer<tFoodItem, 10000> contract symbol. The IUT method must have a return type matching the IUT type bound to the tFoodItem symbol, and must take exactly one parameter of

	type Integer. The IUT method that is selected will automatically be instrumented to instruct the profiler to ensure that the IUT method is side-effect free.
BoundedContainer<tFoodItem, 10000>::HasItem()	Can be bound to any method within the type selected for the BoundedContainer<tFoodItem, 10000> contract symbol. The IUT method must have a return type of Boolean (or an implicit cast must exist), and must take exactly one parameter whose type must match the IUT type bound the tFoodItem symbol. The IUT method that is selected will automatically be instrumented to instruct the profiler to ensure that the IUT method is side-effect free.
BoundedContainer<tFoodItem, 10000>::Size()	Can be bound to any method within the type selected for the BoundedContainer<tFoodItem, 10000> contract symbol. The IUT method must have a return type of Integer (or an implicit cast must exist), and must take zero parameters, or the parameters must have default values (if supported by the IUT's implementation language). The IUT method that is selected will automatically be instrumented to instruct the profiler to ensure that the IUT method is side-effect free.
BoundedContainer<tFoodItem, 10000>::Add()	Can be bound to any method within the type selected for the BoundedContainer<tFoodItem, 10000> contract symbol. The IUT method may have any return type, and must have at least one parameter whose type matches the IUT type bound to the tFoodItem symbol.
BoundedContainer<tFoodItem, 10000>::Remove()	Can be bound to any method within the type selected for the BoundedContainer<tFoodItem, 10000> contract symbol. The IUT method must have a return type which matches the IUT type bound to the tFoodItem symbol.
BoundedContainer<tFoodItem, 10000>::Remove(T)	Can be bound to any method within the type selected for the BoundedContainer<tFoodItem, 10000> contract symbol. The IUT method may have any return type, and must have at least one parameter whose type matches the IUT type bound to the tFoodItem symbol.
Store::tCashContainer	The tCashContainer type exported by the Store contract can be bound to any type within the IUT. The selected type will instantly have the BoundedContainer<tCash, 10> contract bound to it.
BoundedContainer<tCash, 10>::IsFull()	Can be bound to any method within the type selected for the BoundedContainer<tCash, 10> contract symbol. The IUT method must have a return type of Boolean (or an implicit cast must exist), and must take zero parameters, or the parameters must have default values (if supported by the IUT's implementation language). The IUT method that is selected will automatically be instrumented to instruct the profiler to ensure that the IUT method is side-effect free.
BoundedContainer<tCash, 10>::IsEmpty()	Can be bound to any method within the type selected for the BoundedContainer<tCash, 10> contract symbol. The IUT method must have a return type of Boolean (or an implicit cast must exist),

	and must take zero parameters, or the parameters must have default values (if supported by the IUT's implementation language). The IUT method that is selected will automatically be instrumented to instruct the profiler to ensure that the IUT method is side-effect free.
BoundedContainer<tCash, 10>::ItemAt()	Can be bound to any method within the type selected for the BoundedContainer<tCash, 10> contract symbol. The IUT method must have a return type matching the IUT type bound to the tCash symbol, and must take exactly one parameter of type Integer. The IUT method that is selected will automatically be instrumented to instruct the profiler to ensure that the IUT method is side-effect free.
BoundedContainer<tCash, 10>::HasItem()	Can be bound to any method within the type selected for the BoundedContainer<tCash, 10> contract symbol. The IUT method must have a return type of Boolean (or an implicit cast must exist), and must take exactly one parameter whose type must match the IUT type bound the tCash symbol. The IUT method that is selected will automatically be instrumented to instruct the profiler to ensure that the IUT method is side-effect free.
BoundedContainer<tCash, 10>::Size()	Can be bound to any method within the type selected for the BoundedContainer<tCash, 10> contract symbol. The IUT method must have a return type of Integer (or an implicit cast must exist), and must take zero parameters, or the parameters must have default values (if supported by the IUT's implementation language). The IUT method that is selected will automatically be instrumented to instruct the profiler to ensure that the IUT method is side-effect free.
BoundedContainer<tCash, 10>::Add()	Can be bound to any method within the type selected for the BoundedContainer<tCash, 10> contract symbol. The IUT method may have any return type, and must have at least one parameter whose type matches the IUT type bound to the tCash symbol.
BoundedContainer<tCash, 10>::Remove()	Can be bound to any method within the type selected for the BoundedContainer<tCash, 10000> contract symbol. The IUT method must have a return type which matches the IUT type bound to the tCash symbol.
BoundedContainer<tCash, 10>::Remove(T)	Can be bound to any method within the type selected for the BoundedContainer<tCash, 10> contract symbol. The IUT method may have any return type, and must have at least one parameter whose type matches the IUT type bound to the tCash symbol.
Store::cashContainer	Can be bound to any field within the IUT type selected for the Store contract, which is of the type bound to the tCashContainer symbol.
Store::foodContainer	Can be bound to any field within the IUT type selected for the Store contract, which is of the type bound to the tFoodContainer symbol.

Step 4 – Static Checks

Once binding has been completed, the static checks are executed against the IUT. The IUT is opened using Microsoft's Phoenix system, and a tree structure of the IUT is created internally. The static checks operate against this tree structure. They look for the existence of various structural elements, and patterns.

From the implementation point of view, the execution of static checks is fairly straightforward in that the check executes a query against the model of the IUT to obtain a result. The framework's extension mechanism allows for custom static checks to be designed and implemented.

Step 5 – Instrumentation

Once the static checks have been completed, the IUT is instrumented in several ways in preparation for scenario execution. The following list indicates how the IUT is instrumented:

- Side-Effect Free Methods – IUT methods which have been bound to observability methods are marked so that the profiler can ensure that the state of the system does not change while an observability method is being executed. This is accomplished by taking a snapshot of the system state before and after the method has been executed, and comparing them to see if something has changed.
- Pre/Post/Inv – IUT methods which have been bound to responsibilities which contain design by contract elements are instrumented so that the corresponding preconditions, post-conditions, and invariants are tested. This also includes instrumentation for the ***PreSet*** construct and the saving of the return value for use in post-conditions. The sharp notation (#) used for the specification of the design-by-contract constructs allows for directly executable (after variable/binding substitutions) C# code to be used for the design-by-contract constructs. When a design-by-contract construct fails, a special exception will be thrown. The exception will notify the profiler of the construct failure, and will report any beliefs, the stack trace, and the constraint that failed on the contract evaluation report.
- Scenario Triggers – IUT methods which are bound to responsibilities that are used as a trigger within a scenario are instrumented, so that the profiler is able to create a new internal scenario when the trigger occurs.
- Contract Variables – When a contract variable is used instrumentation is added to the responsibility or observability where the variable is used, so that the profiler can either store or return the requested value as needed.
- Dynamic Checks – Custom dynamic checks are able to instrument the IUT so that the profiler and the check is able to monitor runtime information. Such checks also include the preservation of metric values recorded within the contract.

Once instrumentation is complete, the IUT is executed against the profiler to record execution events and requested metrics.

Step 6 – Scenario Evaluation

As the IUT is being executed, the profiler will record when a scenario triggering event occurs. At this point the profiler will create a new scenario instance within the profiler. That is, scenario evaluation is performed on-the-fly. As the profiler is notified of method calls, events, and other activities, any scenario instances which apply to the instance which received the method call/event will be notified of the event. The scenario instance will then determine if the method call/event matches the scenario grammar. To put this another way, scenario instance objects can be seen as a Windows application, which receives events which occur within their window space. Each scenario instance object will then filter the event to determine if the scenario object should "eat" the event, or if the event has no bearing on the scenario. Such processing involves walking through the scenario grammar, and determining if the scenario termination event occurs. Pre/Post/Checks found within a scenario are translated into specialized methods added to the IUT by the Phoenix system, and are called as needed to execute the check.

Any scenarios which have yet to terminate when the IUT finishes execution are said to fail.

Step 7 – Non-functional Requirements

Non-functional requirements are checked, through the use of metrics and dynamic checks. As illustrated by the contracts described in this document, each contract is able to gather metric information as the IUT and corresponding scenarios are executed. Once the IUT has finished executing each contract's Metrics section is processed. The Metrics section uses the metric methods defined within the contract to get the required values. These values are then passed to specified dynamic checks to interpret and report on the values. That is, dynamic checks are used to evaluate the metrics gathered while the IUT was executed. The result of such evaluation is reported on the contract evaluation report.

Dynamic checks were chosen as the method for the evaluation of non-functional requirements, due to the subjective nature of non-functional requirements. That is, determining if a given metric is "good" or "bad" depends heavily on the domain in which the IUT exists. As such, contract developers can provide specialized dynamic checks to interpret and report on metrics gathered by the contract framework.

Step 8 – The Contract Evaluation Report

The Contract Evaluation Report is displayed once the IUT has finished executing and all metrics have been processed. The report contains a summary of the checks performed, and indicates any scenarios, checks, and design-by-contract elements which have failed to execute. The report also contains the results of metric interpretation performed by the dynamic checks.

The presentation of the contract evaluation report concludes this example.