A Business Rule Extension Framework Proposal

Leo Przybylski

June 1, 2005

Contents

Abstract

To manage *business objects* appropriately, a set of rules – *business rules* – are applied. A business rule is an acceptable use case scenario that describes the intended behavior of a *business object*. One goal of the Kuali API is to offer, extension of it through the creation of *business objects*; likewise, *business rules* may be required to handle additional *business objects*.

The role of a *business object* deterimines the rule(s) that are to be applied. The `TransactionalDocument` is the most interesting *business object* in the scope of this document.

A set of business rules are defined for a `TransactionalDocument`. Base rules packaged with the Kuali Project API may fulfill the needs of some institutions; however, it is not expected that the base rules are adequate for every institution or for all requirements of the application. Some institutions may have a proprietary set of rules to be applied to existing `TransactionalDocument` *business objects* to support new or existing *business rules*.

It has been decided that a framework to support creation of or modification of arbitrary *business rules* without require special maintenance of Kuali project source code is necessary.

1 Introduction

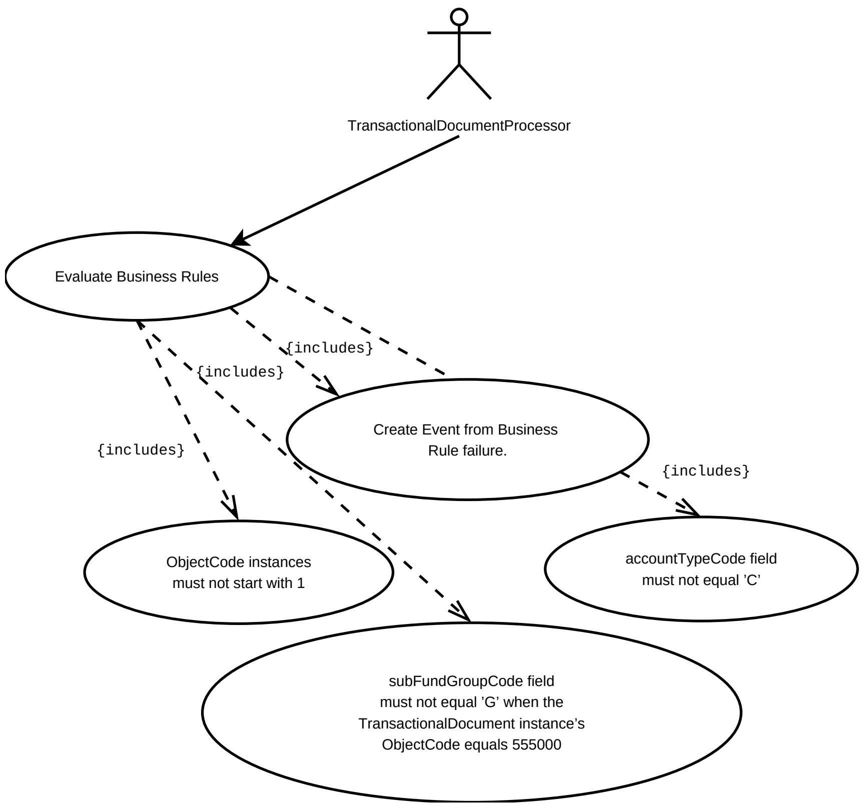
When an institution desires to create third-party business rules for any reason, the available framework is expected to support the creation regardless of the complexity. This flexibility comes from isolating the business rules from the Kuali API save two interfaces (`PolicyFactory` and `Policy`.)

The aforementioned flexibility allows arbitrary business rule definitions to be implemented without the need for proprietary code modifications. To summarize, the beneficial characteristics of the framework are:

- Allow arbitrary class definitions implementing the Policy interface.
- Delegate constructor and method calls, so the framework will automatically instantiate Policy implementations.
- Allow Policy implementations proper access to TransactionalDocument instances.
- Policy implementations can be maintained independently of Kuali API.

> ...a structure such that it is easy to process arbitrary business rules against a document, while making it easy for local implementations (ie, specific schools) to replace the reference implementation's business rules with their own without requiring customization of the reference application. In other words, pain should be absolutely minimal to re-attach local business rules...
> - Andrew Hollamon

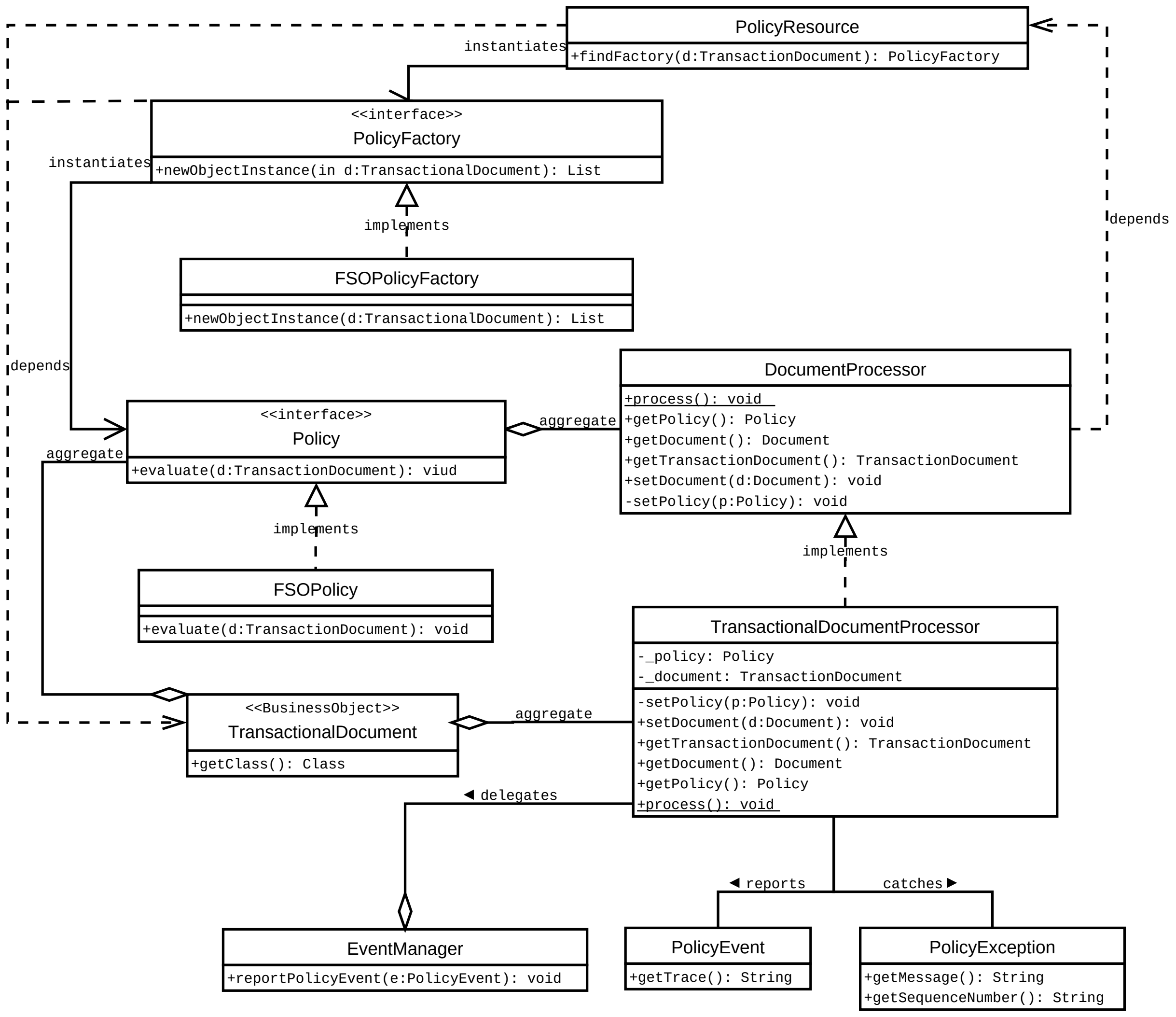
2 Use Cases


TransactionalDocumentProcessor
Evaluate Business Rules
{includes}
{includes}
Create Event from Business Rule failure.
{includes}
{includes}
ObjectCode instances must not start with 1
accountTypeCode field must not equal 'C'
subFundGroupCode field must not equal 'G' when the TransactionalDocument instance's ObjectCode equals 555000

Business rules are acceptable use case scenarios for *business objects*. When *business objects* don't play by the rules, they are following exceptional use case scenarios rather than the rules intended. Above is a diagram describing use cases followed in the examples of this proposal.

3 Framework Overview

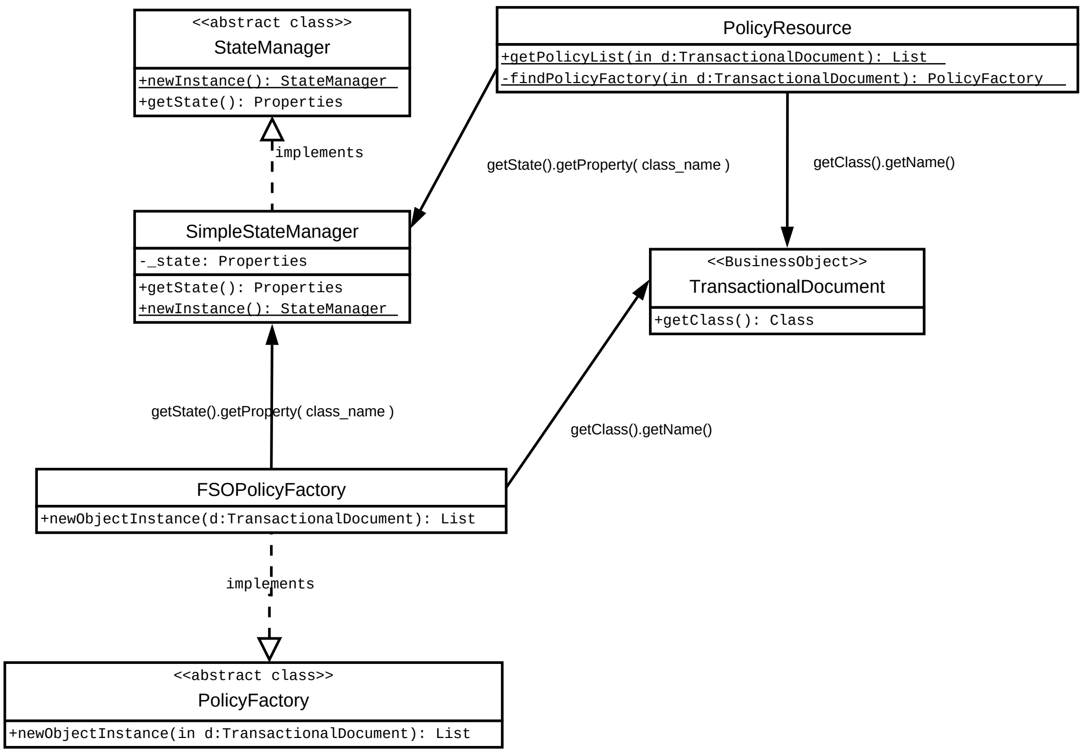
3.1 The Framework Structure



PolicyResource
+findFactory(d:TransactionDocument): PolicyFactory
instantiates
<<interface>>
PolicyFactory
+newObjectInstance(in d:TransactionalDocument): List
instantiates
implements
FSOPolicyFactory
+newObjectInstance(d:TransactionalDocument): List
depends
depends
DocumentProcessor
+process(): void
+getPolicy(): Policy
+getDocument(): Document
+getTransactionDocument(): TransactionDocument
+setDocument(d:Document): void
-setPolicy(p:Policy): void
aggregate
<<interface>>
Policy
+evaluate(d:TransactionDocument): viud
aggregate
implements
FSOPolicy
+evaluate(d:TransactionDocument): void
implements
TransactionalDocumentProcessor
-_policy: Policy
-_document: TransactionDocument
-setPolicy(p:Policy): void
+setDocument(d:Document): void
+getTransactionDocument(): TransactionDocument
+getDocument(): Document
+getPolicy(): Policy
+process(): void
<<BusinessObject>>
TransactionalDocument
+getClass(): Class
aggregate
◀ delegates
◀ reports
catches ▶
EventManager
+reportPolicyEvent(e:PolicyEvent): void
PolicyEvent
+getTrace(): String
PolicyException
+getMessage(): String
+getSequenceNumber(): String

Illustrated above is a high-level view of class relationships used in the extension framework. With each use of the FactoryClass pattern, a new level of flexibility and complexity is added.

For example, mapping a single Policy to a TransactionalDocument may have been sufficient for some applications, but the use of multiple Policy instances adds more flexibility by allowing the reuse of Policy definitions. The cost is that implementation of this kind of flexibility required the use of a PolicyFactory abstraction that allows dynamic creation/discovery of Policy definitions. Also, discovery of the PolicyFactory was handled by mapping it to a TransactionalDocument definition just as the Policy is.



<<abstract class>>
StateManager
+newInstance(): StateManager
+getState(): Properties
PolicyResource
+getPolicyList(in d:TransactionalDocument): List
-findPolicyFactory(in d:TransactionalDocument): PolicyFactory
implements
getState().getProperty(class_name)
getClass().getName()
SimpleStateManager
-_state: Properties
+getState(): Properties
+newInstance(): StateManager
<<BusinessObject>>
TransactionalDocument
+getClass(): Class
getState().getProperty(class_name)
getClass().getName()
FSOPolicyFactory
+newObjectInstance(d:TransactionalDocument): List
implements
<<abstract class>>
PolicyFactory
+newObjectInstance(in d:TransactionalDocument): List

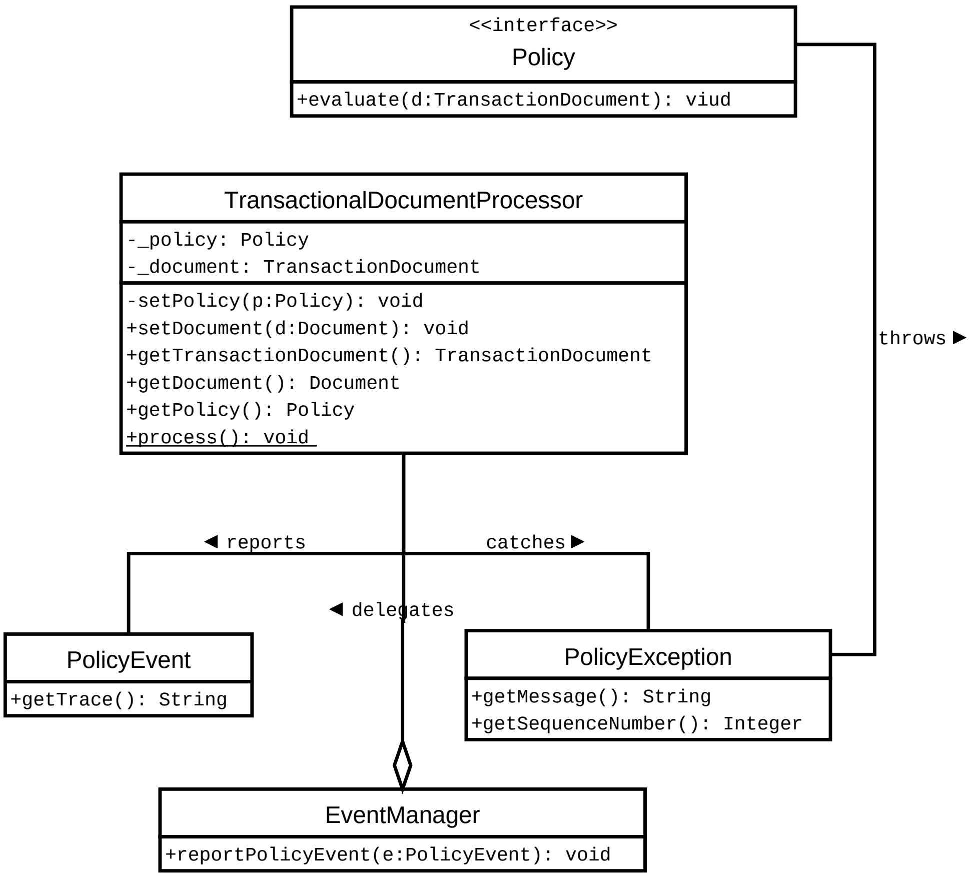
Above is a view that illustrates the relationship between Policy, PolicyFactory, TransactionalDocument, and StateManager, which is used to persistently store the mappings.

The PolicyResource queries the StateManager using the fully qualified class name of the TransactionalDocument. Let's say for example, the name is InternalBillingDocument. The StateManager answers with the class name of the mapped PolicyFactory. PolicyFactory applies the same pattern to discovering the appropriate Policy instances.

When all the Policy definitions have been discovered, then all the business rules can be applied by the TransactionalDocumentProcessor. The functionality of the Policy definition is transparent to the TransactionalDocument and the TransactionalDocumentProcessor instances which allows arbitrary business rules to be applied to whatever TransactionalDocument.

3.1.1 When They Don't Play by the Rules...

A failure to comply with business rules is considered an exceptional use case, and results in an invocation of the Event Sub-System. A discussion on the Event Sub-System in its entirety is beyond the scope of this proposal; however, a general abstraction can be explained.



<<interface>>
Policy
+evaluate(d:TransactionDocument): viud
TransactionalDocumentProcessor
-_policy: Policy
-_document: TransactionDocument
-setPolicy(p:Policy): void
+setDocument(d:Document): void
+getTransactionDocument(): TransactionDocument
+getDocument(): Document
+getPolicy(): Policy
+process(): void
throws ▶
◀ reports
catches ▶
◀ delegates
PolicyEvent
+getTrace(): String
PolicyException
+getMessage(): String
+getSequenceNumber(): Integer
EventManager
+reportPolicyEvent(e:PolicyEvent): void

To invoke the Event Sub-System, an implementation of Policy, or PolicyFactory throws a PolicyException. The PolicyException is part of an Event Sub-System. It is used to create an PolicyEvent which is handled by the Event Sub-System. When the TransactionalDocumentProcessor catches a PolicyException, it assumes they have all are a result of business rule violations and throws an Error back to its caller after reporting the PolicyEvent. What happens after the PolicyEvent is reported is up to the implementation of the system. It may log PolcyEvent instances in a database, a flat file, or just in memory.

Such a system adds further flexibility to the framework for managing business objects by logging and coordinating their behavior with business rules. It is easier to analyze *business rules* for better design and also in the debugging/troubleshooting of the Policy implementation.

The following constrains the Event Sub-System:

- PolicyException must be thrown by evaluate() method of any Policy implementation.
- Is always caught by the calling TransactionalDocumentProcessor instance.
- Provides a sequenceNumber of the AccountingLine that caused the *business rule* evaluation failure.
- EventManager must handle PolicyEvent instances.
- PolicyEvent can only be instantiated from a PolicyException.

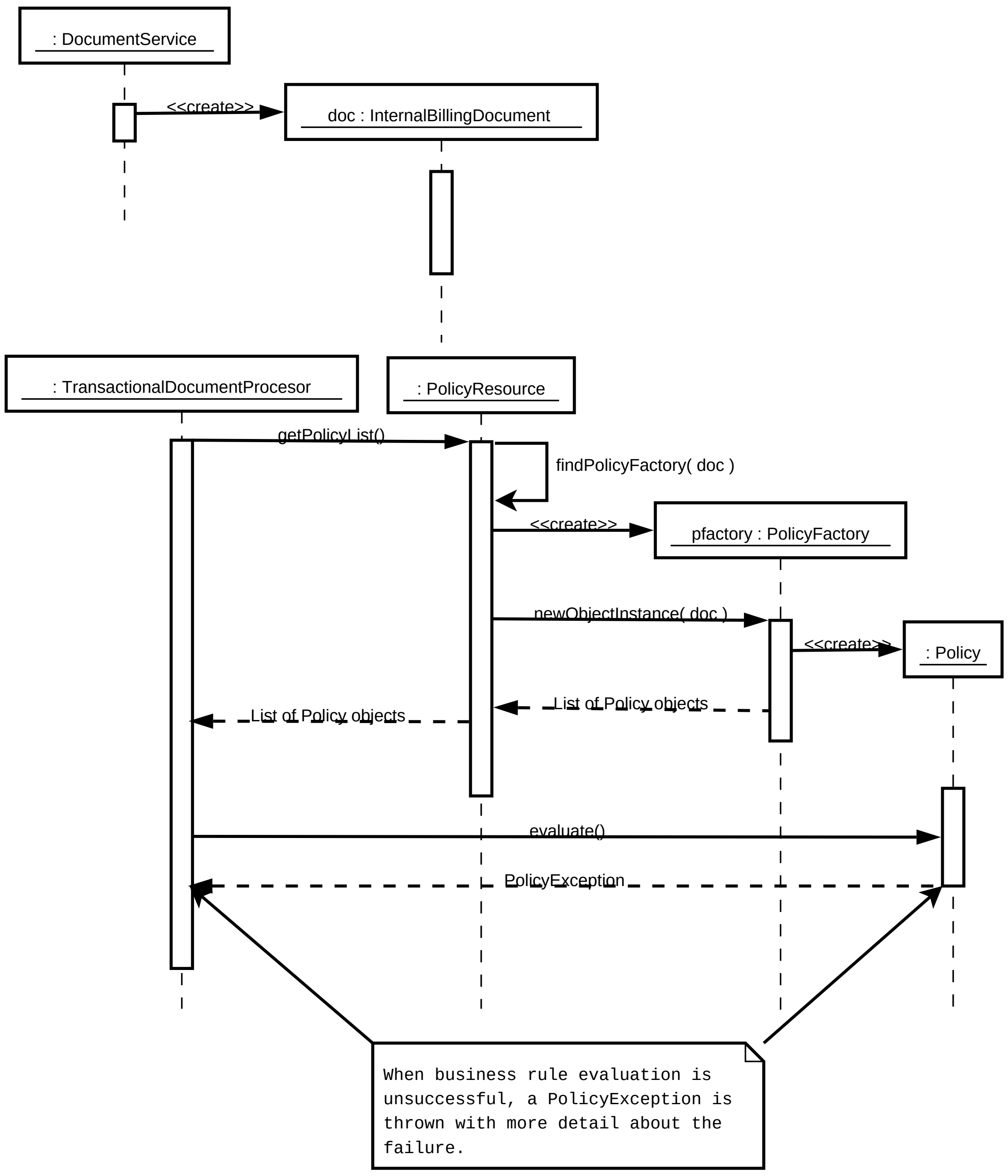
Process flow regarding order of operations and intantiation of objects are explained in greater detail in the following section.

3.1.2 Framework Integration with the Existing API

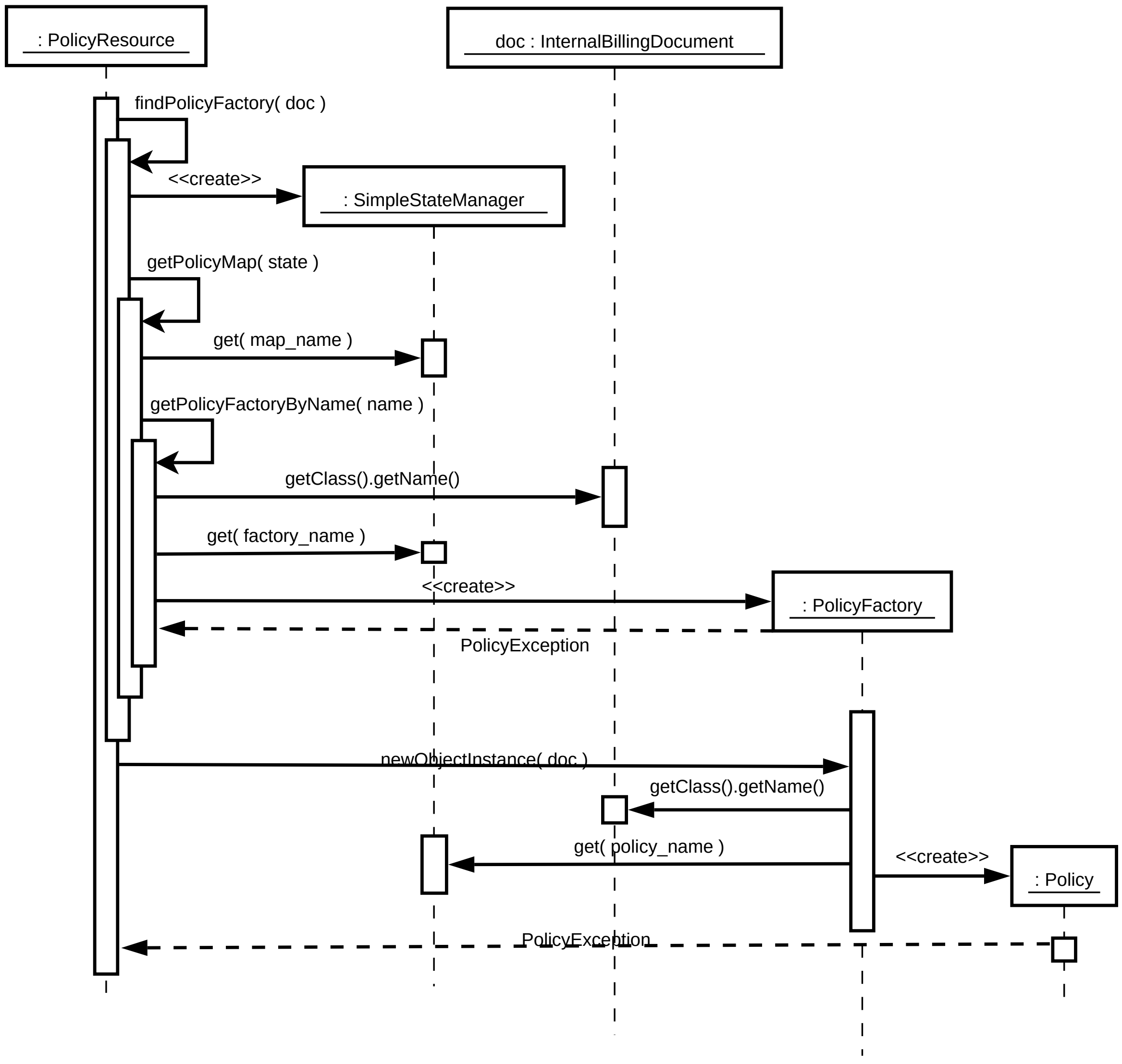
The framework integrates through a single interface in the existing API, the TransactionalDocument interface. The TransactionalDocument interface can be implemented and polymorphically used by the TransactionalDocumentProcessor to discover and evaluate Policy instances relating the TransactionalDocument.

See the section on *Creating the Implementation* for more information.

3.2 Sequence or Process Flow



Above is in illustration of the order of operations in sequence. This is a high-level diagram and omits information about how the mappings are accessed. The process of discovering Policy definitions or PolicyFactory definitions from the mappings is illustrated in the next diagram along with discussion.



Here the aforementioned relationships from the second diagram in the *Framework Overview* are realized in a time-ordered diagram.

4 Creating the Implementation

There are three interfaces part of the framework implementation:

Map is required to provide an implementation that maps a Transactional-Document implementation to a Policy or PolicyFactory implementation.

PolicyFactory is used to create a Policy collection from information provided in the Map.

Policy is the main *business rule* context. It is responsible for evaluating *business rules.*

4.1 Map of Policy Implementations

For the purpose of this proposal, an example Map implementation is provided. The implementation is acquired from the StateManager instance by an instance of PolicyResource (see the section on *Process Flow* for more information.) The StateManager queries a System property kuali.map.class to identify the Map implementation. Observe the following sample source from PolicyResource that describes how the Map is acquired from the StateManager:

```
    return ( Map )Class
            .forName( state.get( MAP_STRING ).toString() )
            .newInstance();
...
...
    private static final String MAP_STRING = "kuali.map.class";
```

Any Map implementation will work (SynchronizedMap, HashMap, etc...) The example provided, makes use of the SimpleStateManager implementation as a Map, but that is not necessary.

4.2 PolicyFactory

PolicyFactory is responsible for building a Collection of the correct Policy instances for the specified TransactionalDocument. PolicyFactory implementations are mapped directly to TransactionalDocumentclass names. The example implementation provided with the proposal obtains the Map implementation from a StateManager instance, and queries the Map implementation using the TransactionalDocument class name for the mapped PolicyFactory. Observe the following sample source from PolicyResource that describes the PolicyFactory class name is queried from the Map:

```
     return policy_map.get( FACTORY_PREFIX
                            + d.getClass().getName() ).toString();
...
...
private static final String FACTORY_PREFIX = "kuali.map.factory.";
```

The mapping uses a prefix of kuali.map.factory. to distinguish itself from Policy mappings.

The example provided with this proposal is org.kuali.bo.FSOPolicyFactory. To create an implementation class definition for the PolicyFactory abstraction, the following was done to create FSOPolicyFactory:

1. Extend the PolicyFactory abstract class.

   ```
   public class FSOPolicyFactory extends PolicyFactory {
   ```

2. Override the getObjectInstance method.

   ```
     public Object getObjectInstance( TransactionalDocument d )
         throws Exception {
         // The purpose behind the factory is flexibility with
         // assigning Policy implementations to a
         // TransactionalDocument.

         // There is already a mapping to the PolicyFactory from
         // the TransactionalDocument, so another mapping isn't
         // necessary. It is possible to just add object
         // instantiations here without losing any flexibility at
         // all. For example, the following is possible:

         List retval = new ArrayList();
   ```

```
        retval.add( new FSOPolicy( d ) );
        // retval.add( new FSOInternalBillingPolicy( d ) );
        // retval.add( new FSOPayrollPolicy( d ) );
        return retval;

        // It is just EXTRA flexibility to reuse the
        // mapping interface.
    }
```

4.3 Policy

A Policy implementation is a class definition of a *business rule*. With each Policy instance that is evaluated by a TransactionalDocumentProcessor, a new *business rule* is evaluated against a TransactionalDocument.

The example provided with this proposal is org.kuali.bo.FSOPolicy. To create an implentation class definition for the Policy interface, the following was done to create FSOPolicy:

1. Implement Policy .

   ```
   public class FSOPolicy implements Policy {
   ```

2. Override the evaluate method (this is probably the most important part of Policy.)

   ```
   public void evaluate() throws Throwable {
       InternalBillingDocument doc =
               ( InternalBillingDocument )getDocument();
       AccountingLine actl = null;
       for( Iterator line_it = doc.sourceAccountingLines();
            line_it.hasNext(); actl = line_it.next() ) {
           if( actl.getObjectCode()
                   .toString().startsWith( "1" ) ) {
               throw new PolicyException
                       ( BAD_OBJECT_CODE_MESSAGE);
           }
   ```

```
                if( actl.getAccount()
                        .getAccountTypeCode().equals( "C" ) ) {
                    throw new PolicyException
                            ( BAD_ACCOUNT_TYPE_MESSAGE );
                }
                if( actl.getAccount().getSubFundGroupCode()
                        .equals( "G" ) &&
                    actl.getObjectCode()
                        .toString().equals( "555000" ) ) {
                    throw new PolicyException
                            ( BAD_GROUP_CODE_MESSAGE );
                }
            }
        }
```

3. Override the getDocument and setDocument methods.

```
        private void setDocument( TransactionalDocument d ) {
            _document = d;
        }

        public TransactionalDocument getDocument() {
            return _document;
        }

        private TransactionalDocument _document
```

5 Invoking the Implementation

TransactionalDocumentProcessor manages the *business rules.* All that is required obtaining a TransactionalDocument for the TransactionalDocumentProcessor to handle.

```
  try {
      new TransactionalDocumentProcessor
```

```
                ( SomeDocumentService.getNewDocument
                        ( "org.kuali.bo.InternalBillingDocument" ) )
                .process();
  } catch( Error e ) {
     // Do something
  }
```

The above creates a new InternalBillingDocument instance and a new TransactionalDocumentProcessor. Finally, it invokes process() method and tries to catch the Error if there is one.