



Your Short Cut to Knowledge

The following is an excerpt from a Short Cut published by one of the Pearson Education imprints.

Short Cuts are short, concise, PDF documents designed specifically for busy technical professionals like you.

We've provided this excerpt to help you review the product before you purchase. Please note, the hyperlinks contained within this excerpt have been deactivated.

**Tap into learning—NOW!**

Visit [www.informit.com/shortcuts](http://www.informit.com/shortcuts) for a complete list of Short Cuts.



**SAMS**

**Cisco Press**

**IBM  
Press™**

**que®**

```
instanceId = instance.InstanceId;
instance.Start();

DisableButtons();
btnGather.Enabled = true;
}
```

The other buttons will raise the appropriate events to transition to the states of the workflow. Now try running the workflow and experiment with the various state transitions.

## Rules-Driven Workflow

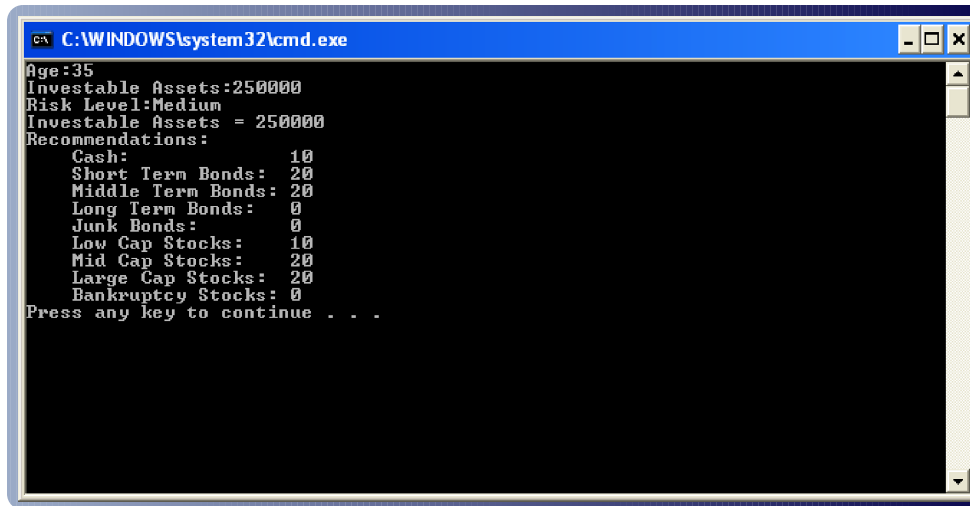
The third workflow pattern that ships with Workflow Foundation is modeled by the PolicyActivity class. The policy activity uses the WF rules engine to incorporate rule based decision making into your workflows. A state or sequential workflow is based on a sequence of operations that evolve over time. Rules allow you to model relationships that are independent of time. You can also view these rules as constraints on the data consumed by your workflow.

The FinancialRecommendations example uses the PolicyActivity to execute a simple set of rules that allocates financial assets based on age, portfolio size, and expressed risk preference (see Figure 22).

## SECTION #2

### Rules-Driven Workflow

**FIGURE 22**  
Output from  
Financial  
Recommendations  
Application



```
C:\WINDOWS\system32\cmd.exe
Age:35
Investable Assets:250000
Risk Level:Medium
Investable Assets = 250000
Recommendations:
  Cash: 10
  Short Term Bonds: 20
  Middle Term Bonds: 20
  Long Term Bonds: 0
  Junk Bonds: 0
  Low Cap Stocks: 10
  Mid Cap Stocks: 20
  Large Cap Stocks: 20
  Bankruptcy Stocks: 0
Press any key to continue . . .
```

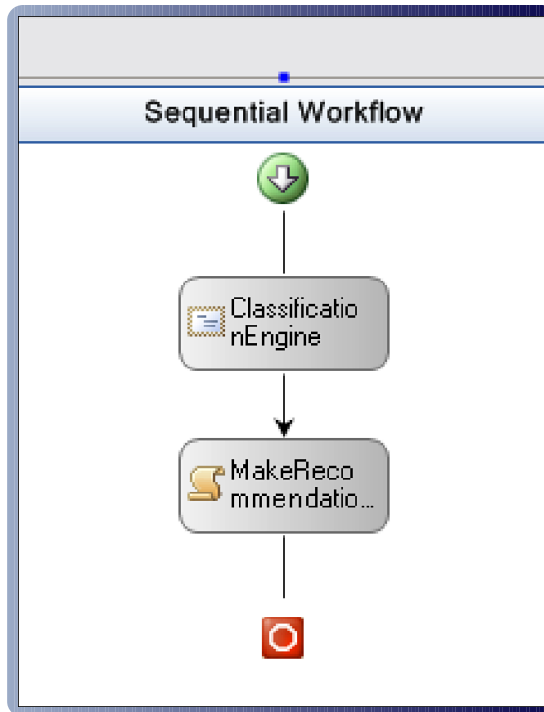
Figure 23 illustrates the simple sequential workflow that consists of a PolicyActivity named ClassificationEngine that uses rules to assign a risk profile to an investor. The CodeActivity MakeRecommendations uses that risk profile to make a suggested asset allocation.

The workflow communicates with its console host through four properties. Age, InvestableAssets, and ExpressedRiskTolerance are input parameters. The InvestableAssets is also an output property since in cases of very low risk tolerance some assets are removed from the investment portfolio. The Recommendation property allows the host to find out what recommendation the workflow made.

## SECTION #2

### Rules-Driven Workflow

**FIGURE 23**  
Financial  
Recommendation  
Engine Workflow



The ClassificationEngine activity assigns a value to the private member variable riskTolerance. The MakeRecommendations CodeActivity uses it to make the actual allocations:

```
private void ExecuteMakeRecommendations (object sender,
EventArgs e)
{
    switch (riskTolerance)
    {
        case RiskToleranceLevel.None:
            recommendation.percentCash = 100;
            break;
        case RiskToleranceLevel.Low:
            recommendation.percentCash = 50;
            recommendation.percentShortTermBonds = 50;
            break;
        case RiskToleranceLevel.Medium:
            recommendation.percentCash = 10;
            recommendation.percentShortTermBonds = 20;
            recommendation.percentMiddleTermBonds = 20;
            recommendation.percentLowCapStocks = 10;
            recommendation.percentMidCapStocks = 20;
            recommendation.percentLargeCapStocks = 20;
            break;
    }
}
```

## SECTION #2

### Rules-Driven Workflow

```
case RiskToleranceLevel.High:
    recommendation.percentCash = 10;
    recommendation.percentMiddleTermBonds = 10;
    recommendation.percentLongTermBonds = 10;
    recommendation.percentLowCapStocks = 30;
    recommendation.percentMidCapStocks = 30;
    recommendation.percentLargeCapStocks = 10;
    break;
case RiskToleranceLevel.Gambler:
    recommendation.percentCash = 5;
    recommendation.percentLongTermBonds = 10;
    recommendation.percentJunkBonds = 20;
    recommendation.percentLowCapStocks = 25;
    recommendation.percentMidCapStocks = 20;
    recommendation.percentLargeCapStocks = 10;
    recommendation.percentBankruptStocks = 10;
    break;
default:
    throw new Exception ("Invalid Risk Tolerance Level");
}
```

Let us now examine how the rules determine the value of riskTolerance.

A rule is analogous to an if / then / else statement in program logic. Depending on the evaluation of a Boolean condition, one or another action might be taken. Rules are combined into a **rule set** which represents a set of rules that are used together to evaluate a given situation. In our case, you need several rules in order to evaluate the risk tolerance of a particular individual.

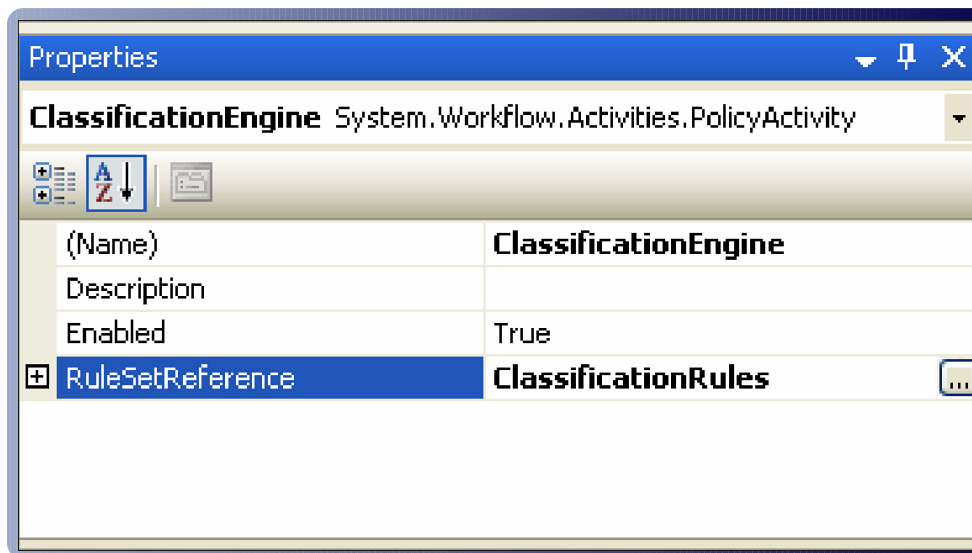
## SECTION #2

### Rules-Driven Workflow

The PolicyActivity has a property called RuleSetReference (Figure 24). This is the name of the particular rule set that this activity is using. In the FinancialRecommendations example, the ClassificationEngine is using the rule set ClassificationRules.

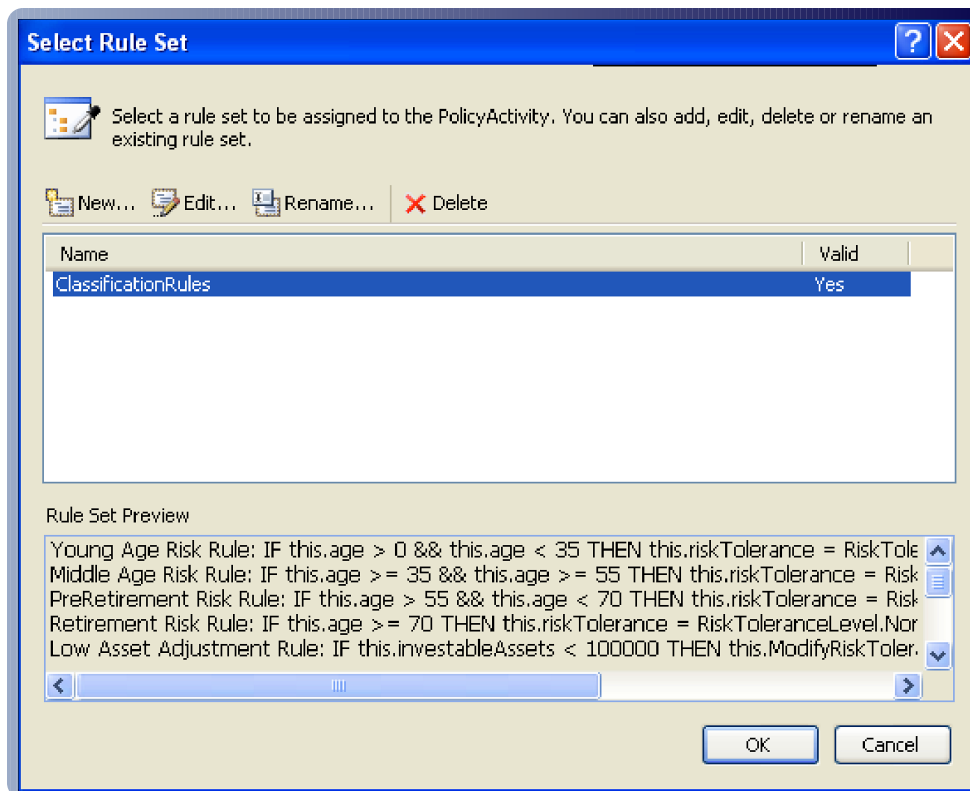
**FIGURE 24**

Property Window  
for PolicyActivity



Although a particular PolicyActivity can only use one rule set at a time, a workflow can have several rule sets associated with it. Pressing the button on property brings up a dialog that manages the rule sets that are associated with the workflow. As Figure 25 illustrates, in our particular case there is only one. It is called ClassificationRules.

**FIGURE 25**  
Rule Set Manager  
Dialog



Clicking on “New...” brings up an empty Rule Set Editor that you can use to populate this new rule set with new rules. Clicking on “Edit...” brings up the Rule Set Editor for the selected rule set. The Rule Set Editor is populated with the existing rules for that set (Figure 26). You can now add, edit, or delete rules within that rule set. Within the Rule Set Manager dialog you can also delete or rename rule sets.

If you have multiple rule sets, you can choose which one a particular PolicyActivity is going to use.

Within the Rule Set Editor you can not only add, edit, or delete rules, but you can also change how the rules interact with each other.

Let us take a look at the Young Age Risk rule which is the first one listed in Figure 26. Each rule has a condition. In this rule the condition is “if the variable age in the workflow is greater than 0 and less than 35”. If this condition is true the risk tolerance level is set to high. There is no “else” condition in this particular rule if the condition is false.

To add a new rule, simply select “Add Rule” in the Rule Set Editor and fill in the condition and the relevant “then” and “else” actions. To delete a rule, just select it and click “Delete” in the Rule Set Editor. Note that Intellisense is available to you in the Rule Set Editor.

Associated with each rule is a priority. Higher numbers are run before lower numbers (a rule with a priority of 2 runs *before* a rule with a priority of 1). Rules with identical priority are run in alphabetic order of their names. In our case, the age risk rules run before the asset adjustment rules. Any rule can be enabled or disabled by checking the Active checkbox associated with it.

The rules in the ClassificationRules rules set fall into several categories. The first group (with Priority of 500) all use the individual’s age to set the riskTolerance variable. The younger, the person, the more risk they can tolerate. The second group (with Priority of 400) adjusts the risk tolerance based on the amount of investable assets an individual has. The less money they have, the less risk they should assume. Individuals with more money can take on greater risks. The third set of rules (with Priority 300), adjusts the risk tolerance based on what the individual says their risk is. If the calculated risk from the evaluation is riskier than their expressed reference, the risk tolerance is adjusted downwards. If their expressed preference is riskier than the calculated risk,

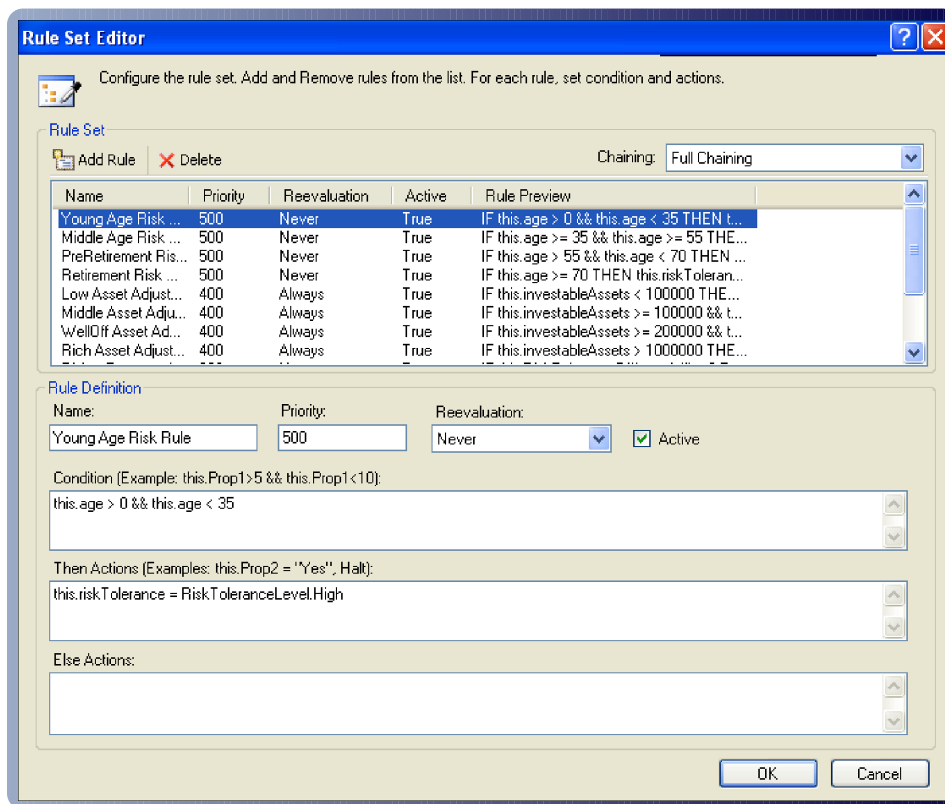


## SECTION #2

### Rules-Driven Workflow

the risk tolerance is adjusted upwards. The final set of rules (with Priority 200) removes some assets from the investment pool for people with low risk tolerances. This money should be put in some savings vehicle instead of being invested. Note that the Cash recommendation here is potentially available for investment. It is not considered as part of an individual's savings for expenses or other emergencies.

**FIGURE 26**  
Rule Set Editor  
for the  
ClassificationRules  
Used to Define  
Risk Tolerance



If you will examine these rules you will see that they are interrelated. If the final set of rules adjusts the amount of investment assets, this affects the results of the second group of rules that adjusted the risk tolerance based on the amount of assets. Should the second group of rules be run again? If the second group of rules is run again, should the resulting change in risk tolerance cause the third set of rules be run again? Depending on the nature of the rules, the results could converge or diverge!

The Chaining and Reevaluation drop down boxes allow you to control how these rules relate to each other. The Chaining drop down applies to the entire rule set. The Reevaluation drop down applies to each rule individually.

The default value for the Chaining drop down is “Full Chaining”. If the Rules Engine detects a change in a value that was used in a rule condition because of the execution of another rule, it will rerun those rules. You can prevent this from happening by setting the reevaluation value for that rule to “Never”. The default is “Always”. Note that the set of age rules has its reevaluation value set to “Never”.

If you set the Chaining value to “Sequence” each rule condition is evaluated only once even if the value used in its evaluation has changed.

The Rules Engine can detect dependencies for variables that are explicitly mentioned in the rules. Notice, however, that some of the rules (such as the rules that modify the risk tolerance) use methods that modify some of the variables used in rule conditions. You use the RuleWrite and RuleRead attributes to indicate to the engine that there are dependencies in the code that the Rules Engine needs to be aware of. There also is a RulesInvoke attribute that tells the engine that the method invokes another method that it needs to inspect the attributes of.

## SECTION #2

### Rules-Driven Workflow

```
[RuleWrite ("riskTolerance")]
private void ModifyRiskTolerance (int increment)
{
    riskTolerance = riskTolerance + increment;

    if (riskTolerance > RiskToleranceLevel.Gambler)
        riskTolerance = RiskToleranceLevel.Gambler;

    if (riskTolerance < RiskToleranceLevel.None)
        riskTolerance = RiskToleranceLevel.None;
}

[RuleRead ("riskTolerance")]
private int RiskToleranceDifferential ()
{
    int diff = expressedRiskTolerance - riskTolerance;
    return diff;
}

[RuleWrite ("investableAssets")]
private void ModifyInvestableAssets (int percent)
{
    investableAssets = (int)((1 - percent * .01) *
        investableAssets);

    return;
}
```

This approach works for code that you write. Suppose you make a call into third party code in an assembly that you have no control over. In that case, in the rule's action you use the "Update" statement in the rule to indicate a dependency:

```
MakeThirdPartyCall ()
Update("this/riskTolerance/")
```

You can also use the Update statement to explicitly control the chaining behavior. Use the "Explicit Update Only" value for the Chaining parameter if this is the behavior that you want.

Another statement you can use inside of an action is "Halt". This will cause the Rules Engine to stop processing any further rules.

Associated with the project is a file called "FinancialRecommendations.rules". Within this XML file is stored the ClassificationRules rule set. Part of this files looks as follows:

```
<Rule Name="Young Age Risk Rule" ReevaluationBehavior="Never"
      Priority="500" Description="{p3:Null}" Active="True">
  <Rule.ThenActions>
    <RuleStatementAction>
      <RuleStatementAction.CodeDomStatement>
        <ns0:CodeAssignStatement LinePragma="{p3:Null}"
          xmlns:ns0="clr-namespace:System.CodeDom;Assembly=System,
          Version=2.0.0.0, Culture=neutral,
          PublicKeyToken=b77a5c561934e089">
          <ns0:CodeAssignStatement.Left>
            <ns0:CodeFieldReferenceExpression
              fieldName="riskTolerance">
```

...

The conditions and actions of the rules are defined in terms of System.CodeDom expressions. Anything that can be expressed using System.CodeDom can be expressed as a rule. In fact, using the Rules Editor you can express only a subset (albeit a useful subset) of the potential rules that can be passed to the Rules Engine.

If you want to see how the Rules Engine processes the rules, you can add a trace statement in the application configuration file:

```
<add name ="System.Workflow.Activities.Rules" value ="Information" />.
```

Tracing was discussed in an earlier section of this article.

At this point you should try to play with various inputs, and see if you can understand how the recommendations were generated based on the rules. If necessary use the trace statements. You can also modify the chaining and reevaluation rules to see how those changes modify the way the Rules Engine operates.

The Rules Engine works with declarative conditions that do not require you to write any code. Therefore, it is much easier to modify the rules than changing a sequence or state machine workflow.

You can incorporate the rules engine in one of your own custom activities. In addition, it is possible to host the rules engine, and the rules editor outside of a workflow. You can also write your own rules editor as well.