Welcome to this short demonstration on using the code-centric features of the IBM Rational Rhapsody modeling tool. The edition I will be demonstrating is called "Architect for Software" which has a subset of features also available in the Developer Edition of Rhapsody. The primary focus of this Edition is what we call code-centric development.

When using UML modeling with code generation, you can really thing of two extreme viewpoints. Whereas model-centric development emphasizes the ability to generate code from UML abstractions; code-centric emphasizes the ability to reverse engineer existing code without changing its layout; and allowing developers to make changes entirely in their IDE, if they wish.

Rational Rhapsody supports code-centric working using settings to tailor the reverse engineering and round-trip process. These are particularly valuable where you have existing code under development, or if you're initiating a new project based heavily on existing code. I will therefore show how we can import code into IBM Rational Rhapsody.

As part of the import I will get Rhapsody to automatically create diagrams to visualize structure; before making changes. Another feature is the ability to integrate with IDEs such as Eclipse and Visual Studio. In this example we will work with Eclipse and use the Workflow Integration plug-in provided as part of the Rhapsody installation. So let’s start the demo.

We’re going to start with some C++ code in Eclipse. This has been developed and compiled using the C and C++ Development Tools plug-in. The Eclipse CDT essentially provides a rich Integrated Development Environment. You can download it for free from eclipse.org.

You can see here we have a simple application with a number of classes. The main creates a GameController object and then invokes a runGame() method on it. runGame() has a while loop that invokes the draw() and updatePosition() methods on a Ball object. This while loop continues this until the escape key is pressed. If we do a clean build and then run the application from a command window, we can see that the ball bounces around the screen until we press (Esc)ape.

So currently the code has no design documentation. Let’s create a Rhapsody model to capture the design using the internationally recognized UML notation. We'll begin by launching the Rhapsody Architect for Software Edition and creating a new "C++" project called "ConsoleGame" on the desktop. We will also add a user profile to set up a number of default properties like display options.

The project is currently empty, however, we can use Rhapsody's Reverse Engineering dialog to import code and bind it with the model. Reverse engineering is similar to compilation. If our project is in Eclipse We can 'seed' the Reverse Engineering tool by selecting the "Import from eclipse" option. We can then select the project that we want to import.

If we hit Finish and then return to Rhapsody, we can see that Reverse Engineering dialog has selected the C++ code to import. If we look in the advanced dialog we can also see that the reverser has captured a number of include directories and that we've chosen to populate diagrams. Let’s start the import.
We can see from the Log window in Rhapsody that the reverse engineering was successful. We now have classes in the model. The code is now bi-directionally synchronized with Rhapsody. If we right-click and choose "Locate in Eclipse" we can navigate to the related files in Eclipse. Conversely, if we right-click in Eclipse and choose Locate in Rhapsody we can navigate to the element in Rhapsody.

You'll notice here that Rhapsody has captured not only the operation name but also its implementation body. As well as bi-directionally navigating, I can also make changes and have them bi-directionally synchronized. Let's add a comment to the runGame() implementation body in Rhapsody. Notice that When I navigate to eclipse we can see that the change was generated into the code. If I make a change in Eclipse and return to Rhapsody, we can see that the model is updated automatically. This ability to keep the model and code in sync, means that I can make updates in either the model or the code, whichever I feel is the most productive for the change I need to make.

One area where UML is particularly productive is when we need to make structural changes such as adding new classes and relationships. If you recall, one of the reverse engineering options we ticked was to populate diagrams for each package. If we right-click on the GameController class and choose Locate on diagram, we can navigate to the diagram Rhapsody created.

If we double-click on the tab, here, we can switch to Rhapsody's full screen mode and see the relationships between the classes graphically using UML notation. Notice here that the GameController has a unidirectional association to the Ball class. If we choose Locate in Eclipse we can see this represents a pointer to a Ball called itsBall.

If we scroll up we can also see that the Sprite class has a <<Usage>> dependency to NTConsole and this represents a #include in the code. Using the bi-directional navigation we can quickly gain an understanding of the relationship between C++ constructs and their UML notation counterparts.

Finally we can see that the Ball class has a generalization relationship to the Sprite class. Essentially a Ball is a type of Sprite that moves around the screen by virtue of its implementation of the updatePosition operation. Let's make some changes in the model now to extend our design to include a Bat class that is capable of hitting the Ball.

Let's create a new class diagram called Bat Collaboration. We can drag on the Sprite class and add a new class called Bat which is a type of Sprite. We can also say that the GameController wants to invoke operations on a Bat object called itsBat. We want to construct the Bat at a particular position so let's add a non-default constructor with an x, y position and a Color.

From the UML notation we can see that Sprite has a pure virtual method called updatePosition. We can tell this from the Class diagram because its name is written in italics - the UML notation for an abstract operation. The inheriting class must therefore provide a concrete realization for this method. We can use the Realize Base class option in Rhapsody to add it to the class for us.

Let's navigate to Eclipse to add the implementation for these new operations. We can see that Rhapsody has generated a header and cpp file for us so we can add the base Sprite class constructor to the initializer list to create a Sprite at the x, y position with a height of 4, width of 1, and the given color.

In terms of the updatePosition method we want to move the Bat up or down when the 'A' and 'Z' keys are pressed. Notice here that we can use the auto-complete features of the IDE to improve our productivity. We want to decrement the y position if the A key is pressed and increment the y position if the Z key is pressed.
If we locate in Rhapsody we can also see that this implementation has been round-tripped in. Notice also that the GameController has a pointer to a Bat so we need to update its constructor to create a Bat. We also need to update the destructor to delete it. In runGame we can now call the draw() and updatePosition() methods for the Bat. If we build and run now we can see that the Ball and Bat are now displayed and that the Bat can move up and down.

The final change I'm going to make is to get the Ball to bounce off the bat. Fortunately, there is already a line intersection algorithm in my NTConsole class that I can use for this. To make use of this I'm going to add a new method to the GameController class to handle the collision. Rather than make the change in Rhapsody, let's make the change entirely in eclipse.

We will add a new private method called handleCollisionBetween() that handles the collision between a Ball and Bat object. We can then use the Eclipse Refactor menu to implement the method [cutting and pasting in the implementation].

So we now have a new method so let's invoke this in the runGame loop. Once we've done this we can then rebuild and if we run the application we can now see that the Ball bounces off the Bat.

This illustrates how code-centric working allows you to work entirely in the IDE if you like. Some developers may prefer this, while others like working visually. Depending on your IDE some aspects are often best performed in the IDE where you can make use integrated compiler features. Other aspects such as adding new classes and relationships are more productive in UML where you can use diagrams.

Notice if I return to Rhapsody that the model has been kept up to date with minimal manual effort. If I look in rhapsody, you'll notice that the class diagram for the src view has, in fact, been automatically updated with the new Bat class that we added.

This means that I can use Rhapsody to generate design documentation for my project. We can also use Rhapsody and UML to develop artifacts that complement the design such as use case, activity and sequence diagrams. These features help to communicate the design to other people and to ensure that key information is not lost or locked in people's heads.

To summarize, Rhapsody supports code-centric, as well as model-centric workflows. The key thing about code-centric settings, are that they subtly shift the emphasis from code generation to reverse engineering. This enables Rhapsody to document existing code without changing its structure or layout; minimizing the impact of adopting modeling on existing projects, or where layout preservation is critical to success.

As well as working entirely in code, you can still make updates in the model. The important thing is that the design is kept relevant and accurate. This is particularly important in the latter stages of a project when the code is considered king.

Thank you for your time. If you have any further questions please don't hesitate to contact your local IBM team, or Rhapsody point-of-contact.