MODEL-BASED DEVELOPMENT - TUTORIAL

Objectives

- To get familiar with the fundamentals of Rational Rhapsody.
  - You start with the simplest example possible.
  - You end with more complex functionality, and a more complex state machine.

At the end of this section …

- At the end of this section, you will be able to:
  - Create a new project
  - Perform some basic modeling using classes, attributes, operations, relations, and state charts
  - Generate and compile code
  - Debug the model by injecting events, setting breakpoints, capturing behavior on sequence diagrams, visualizing the state of objects, and so on

Agenda

- Exercise 1 : Hello World
  - You start with the simplest example possible, just a single object that prints out Hello World.
- Exercise 2 : Count Down
  - Next, you create a simple counter using a simple statechart.
- Exercise 3 : Dishwasher
  - Finally, you create a dishwasher and a more complex statechart.
- Summary

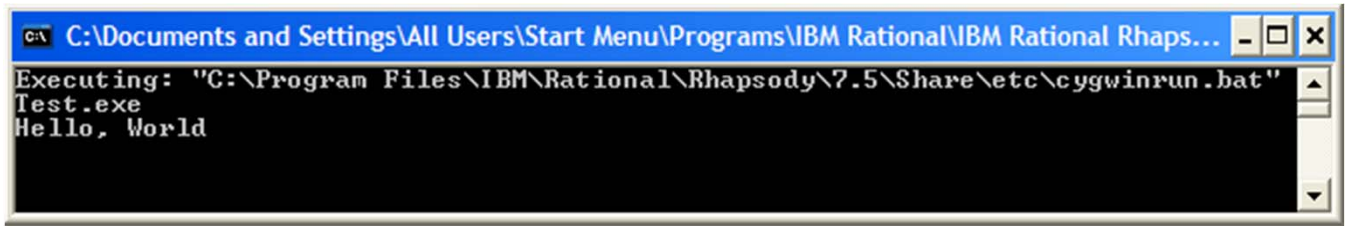
Before you start

- Rational Rhapsody uses C, which is case-sensitive. Most of the errors that are made during this training course are due to entering text incorrectly.
- During this training, you use a naming convention where all classes start with an upper case, and all operations and attributes with a lower case. If two words are concatenated, then the first letter of each word is capitalized, for example, `thisIsAnOperation`, `MyClass`, `anAttribute`.

Where are we?

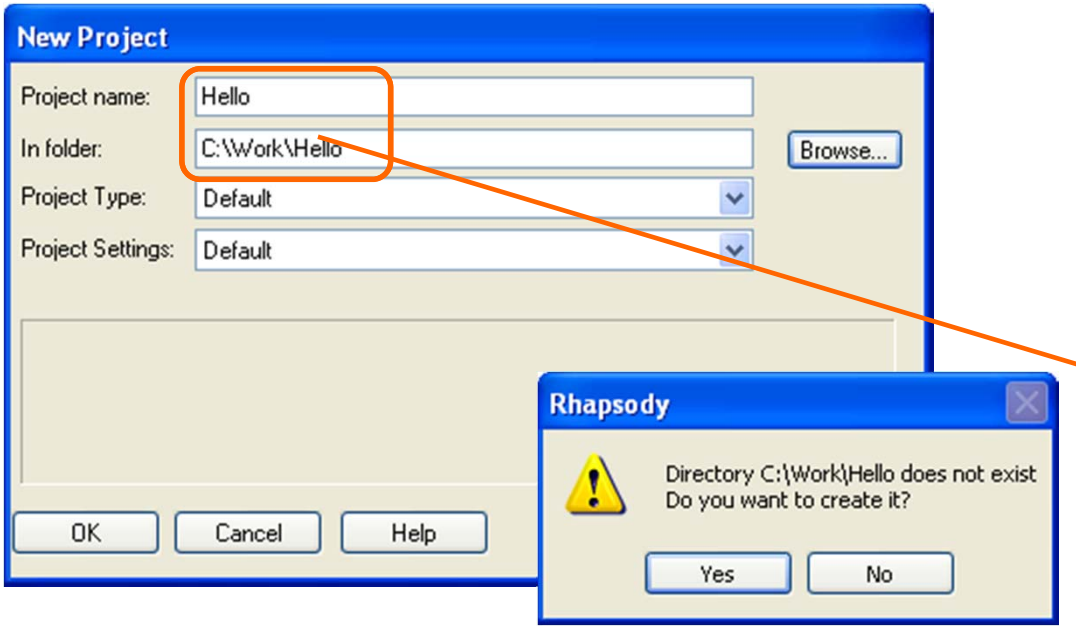
- Exercise 1 : Hello World
  - You start with the simplest example possible, just a single object that prints out Hello World.
- Exercise 2 : Count Down
  - Next, you create a simple counter using a simple statechart.
- Exercise 3 : Dishwasher
  - Finally, you create a dishwasher and a more complex statechart.
- Summary

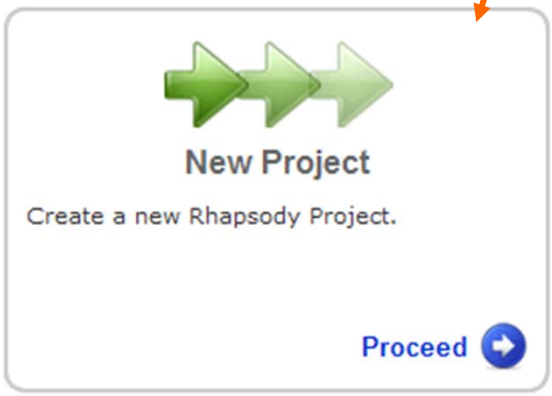
Exercise 1 : Hello World



Creating a project

- Start Rational Rhapsody in C (development edition).
- Use either the (new) icon or select **File > New** or **New Project** to create a new project called *Hello* in a desired working directory.



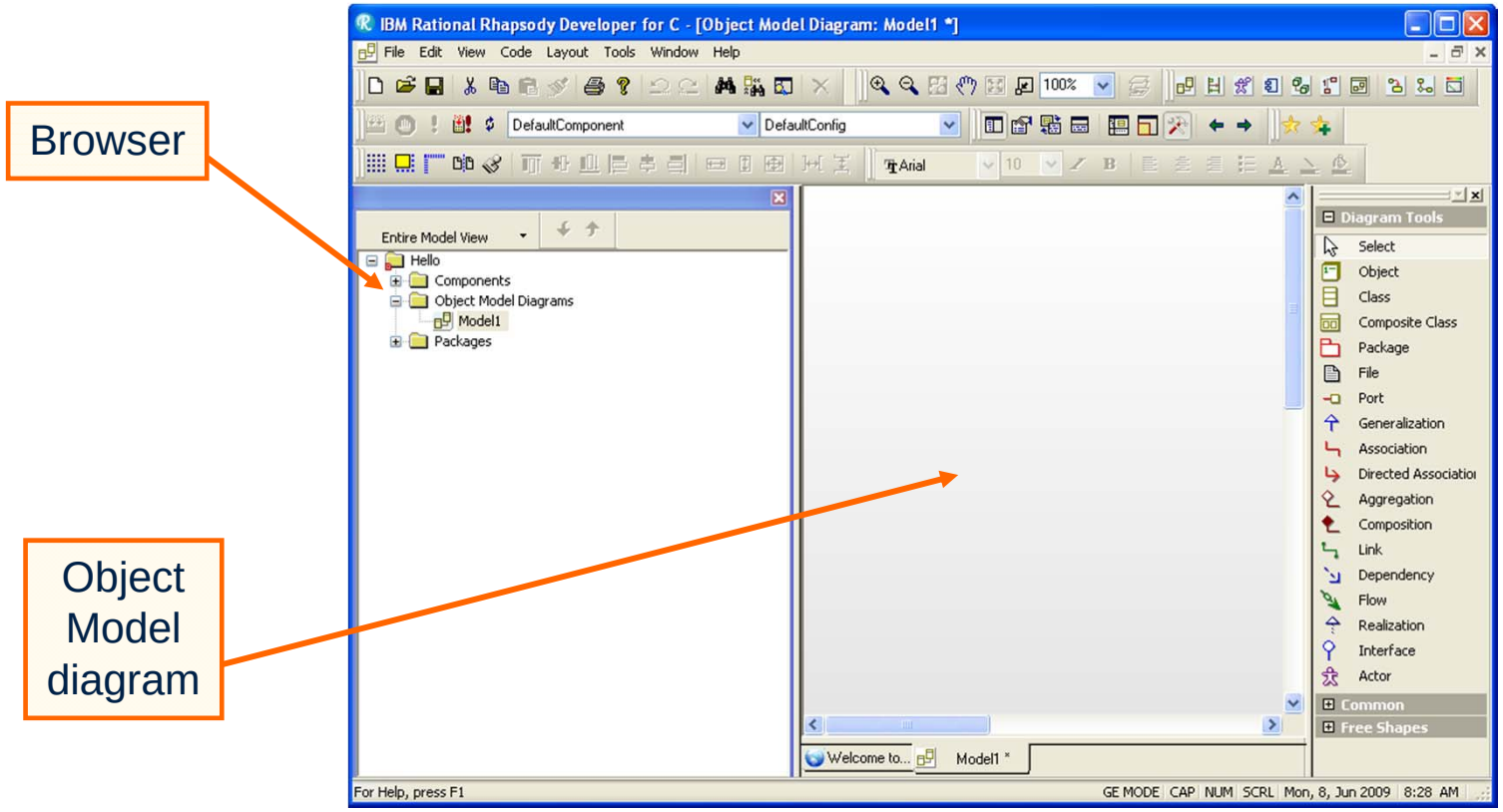


You can choose any working directory you want. Just make sure you create subfolder named Hello.

- Click **OK** then **Yes** to save project.

IBM

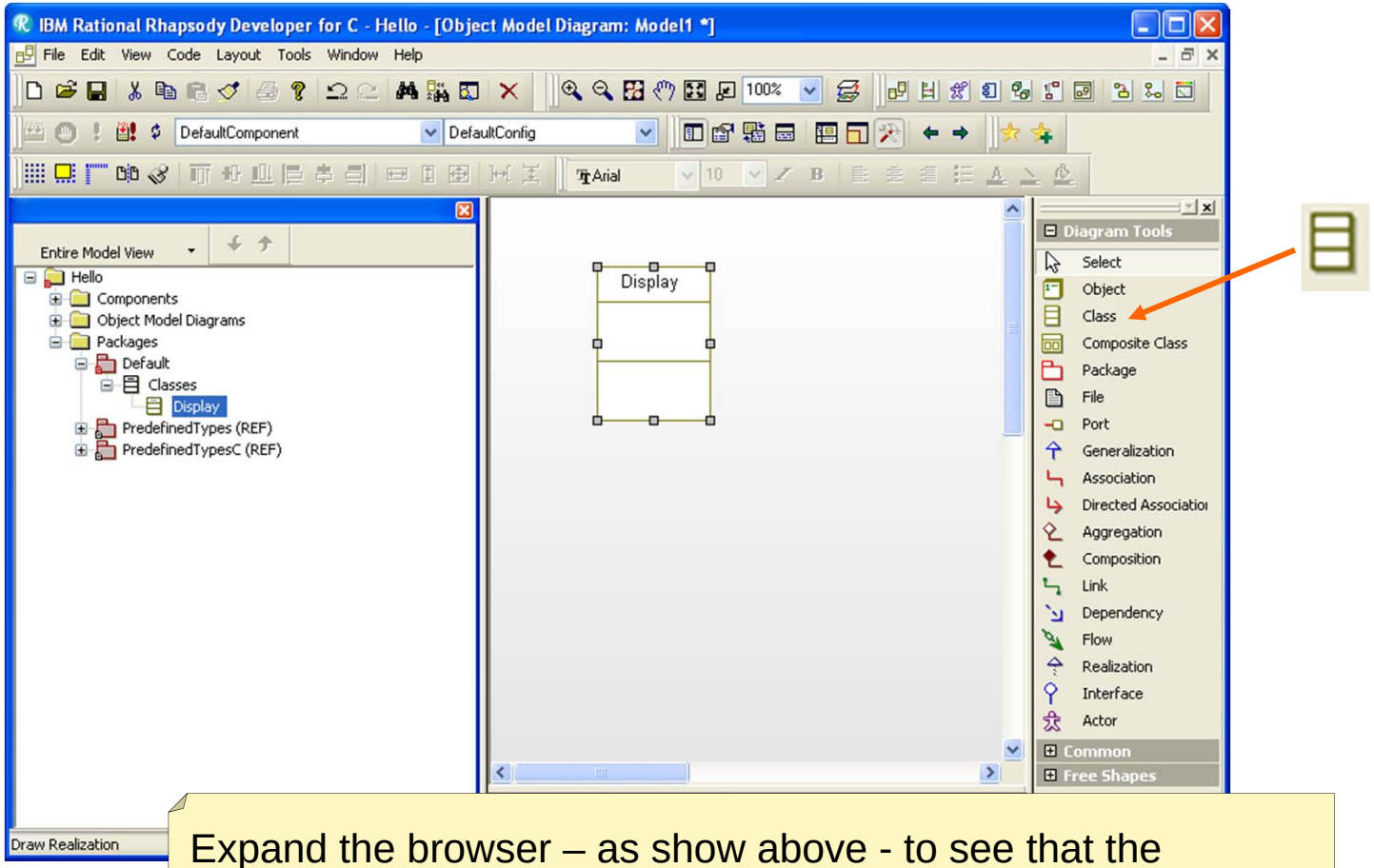
Browser



The browser displays everything that is in the model.
Note that Rational Rhapsody creates an Object Model diagram.

Drawing a class

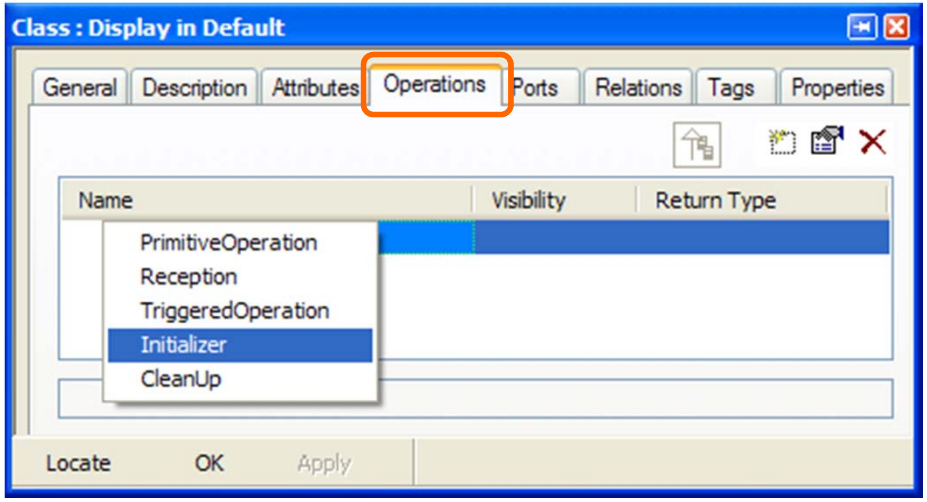
- In this Object Model diagram, use the class icon to draw a class named *Display.*

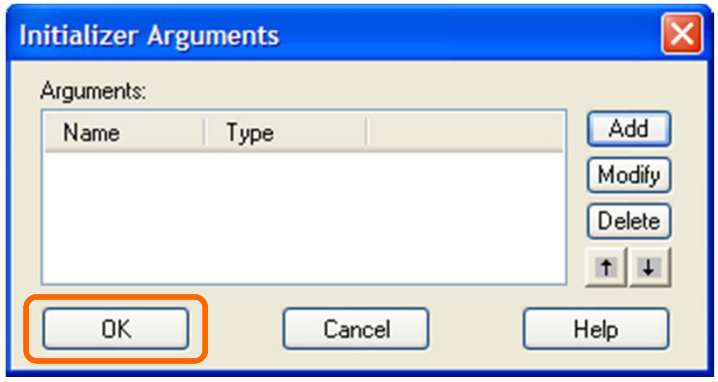


Expand the browser – as show above - to see that the display class also appears in the browser.

Adding an initializer

- The simplest way to add an *initializer* to the class is to double-click on the class to open the features (or right-click and select **Features**).
- Select the **Operations** tab, click **New**, and select **Initializer**.
- You do not need any Initializer arguments, so click **OK**.

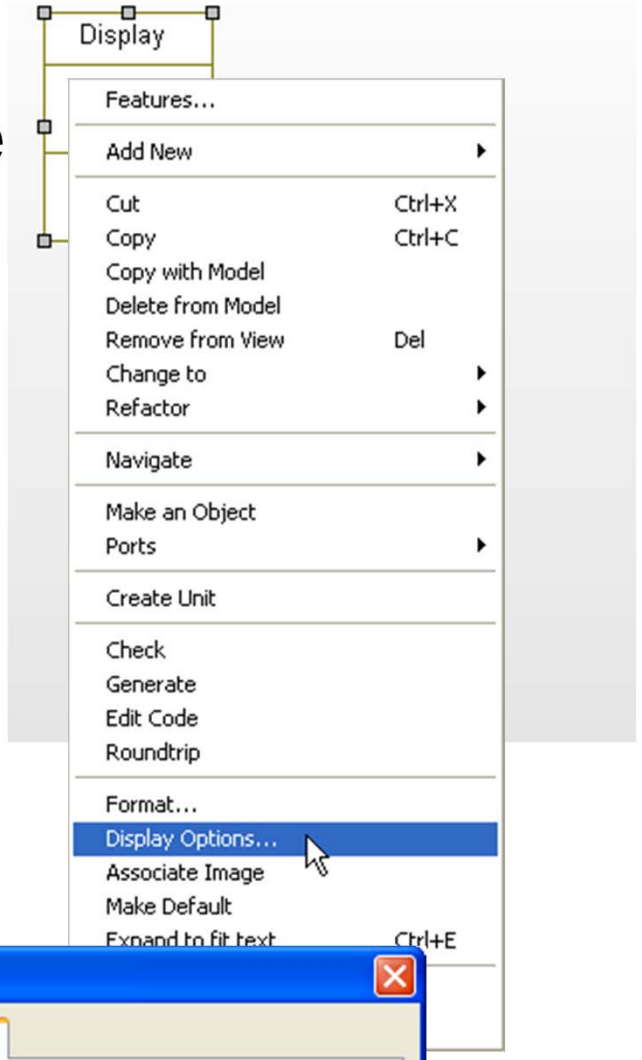


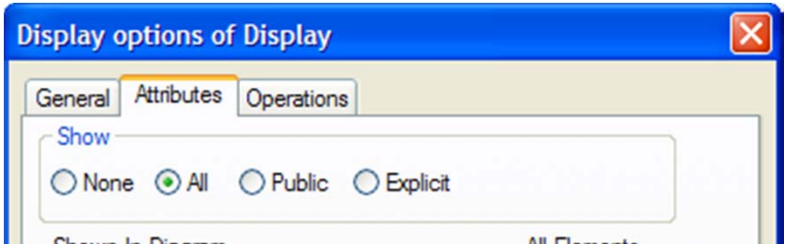


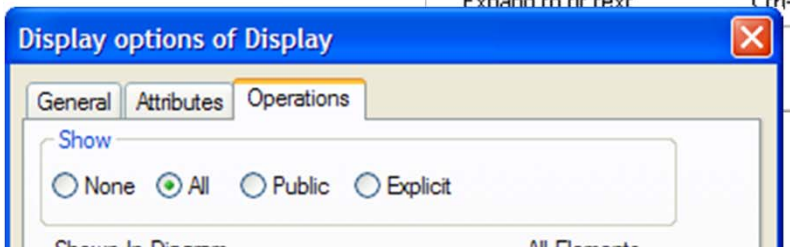
An Initializer is the operation that gets executed when the object is created at runtime. (It is the equivalent of a constructor in C++.)

Display options

- You would expect to see the *Initializer* shown on the class on the Object Model diagram.
- You can control what gets displayed on this view of the class by selecting **Display Options**.
- Right-click the *Display* class, select **Display Options**, and then set the options to show **All** on both the **Attributes** and **Operations** tabs.

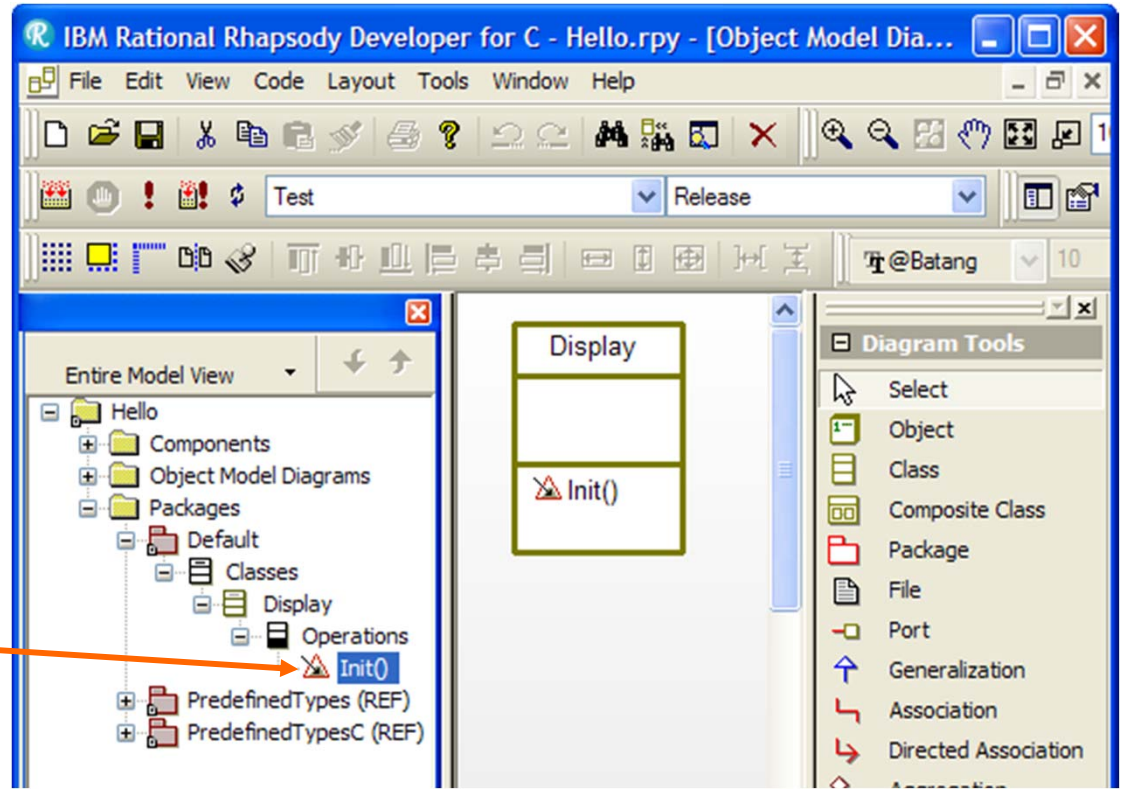






Display initializer

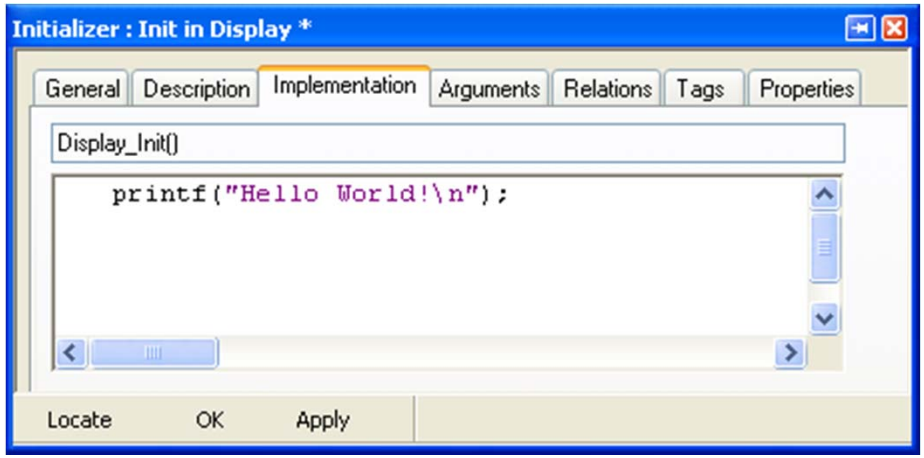
- You should be able to see that the *Initializer* is now shown in both the browser and the OMD (Object Model diagram).



Initializer
`Init()`

Adding an implementation

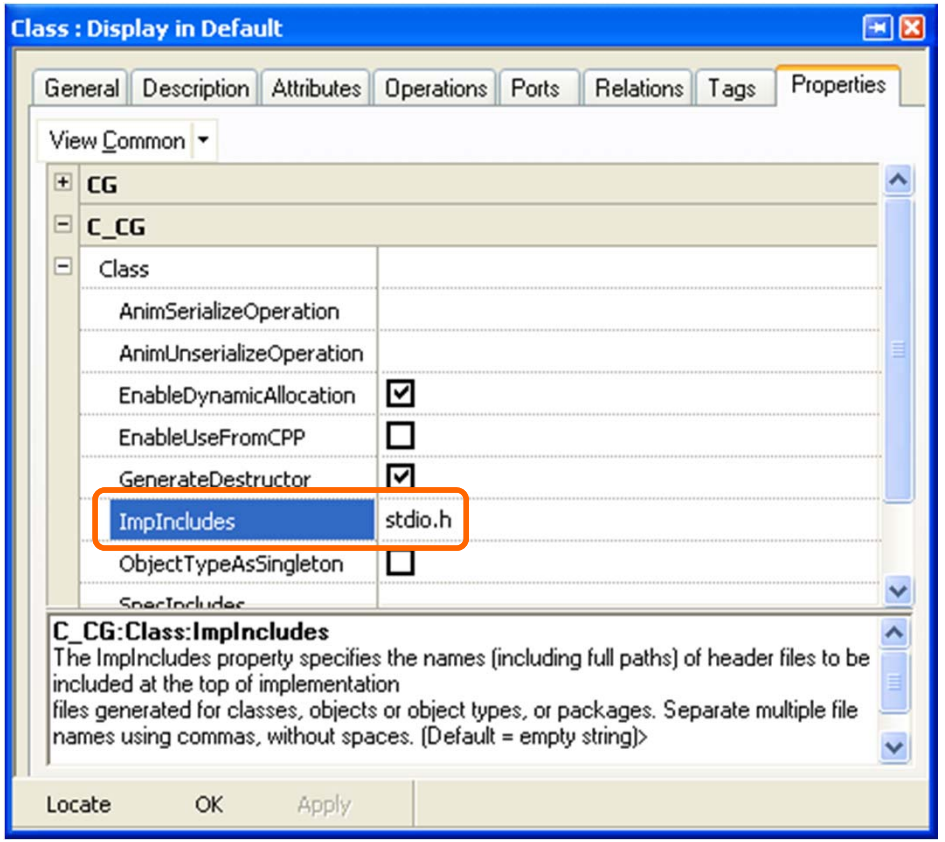
- Select the Display Initializer in the browser and double-click to open the features window.
- Select the **Implementation** tab and enter the following: `printf (“Hello World\n”);`



If you want to close the window by clicking (the upper-right "x"), then make sure that you apply the changes first.

#include stdio.h

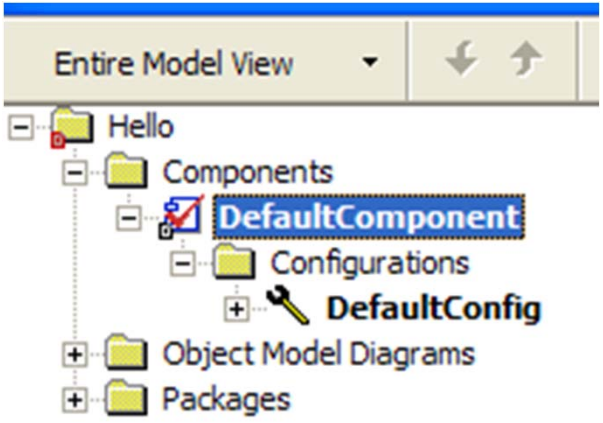
- Because you used `printf()`, you must do an `include` of the stdio.h file in the *Display* class.
- In the browser, select the *Display* class and double-click to bring up the features.
- Select the **Properties** tab (ensure that the Common view is selected) and enter `stdio.h` into the **ImpIncludes** property.

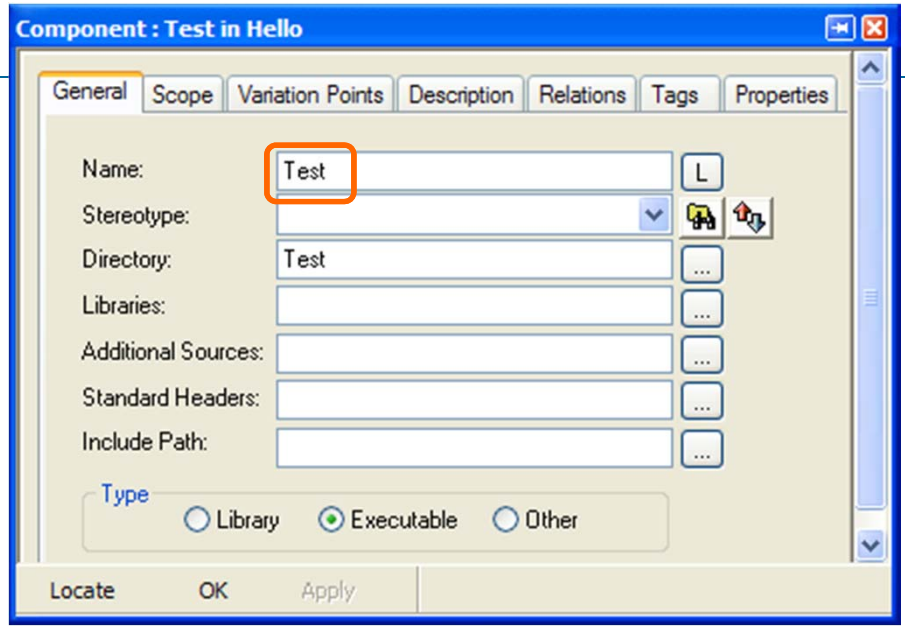


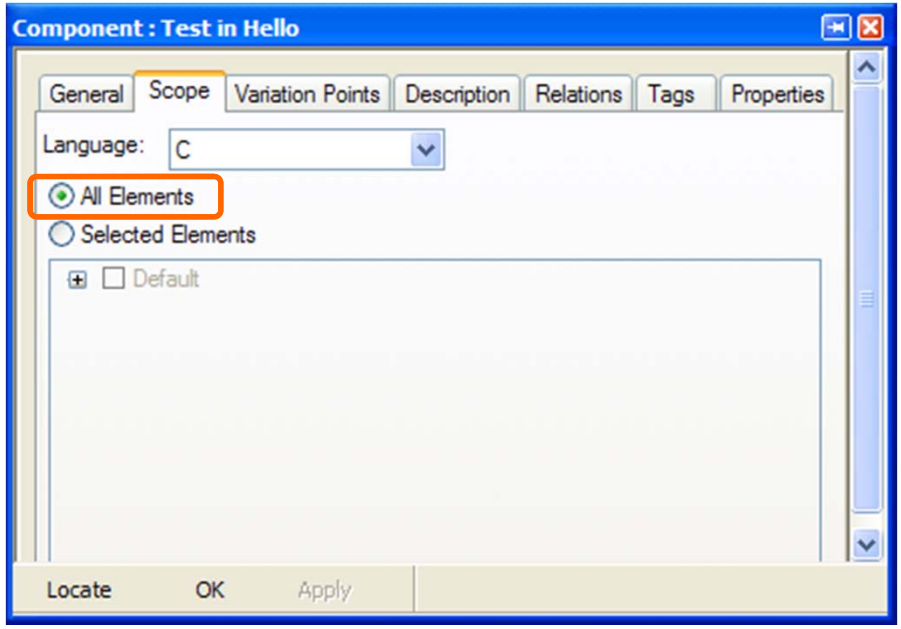
ImpIncludes is an abbreviation for Implementation Includes; C_CG means "C" Code Generation.

Renaming a component

- In order to generate code, you must first create a *component*.
- Expand the components in the browser and rename the existing component called *DefaultComponent* to *Test*.

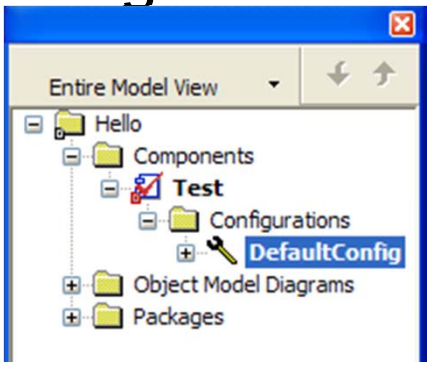


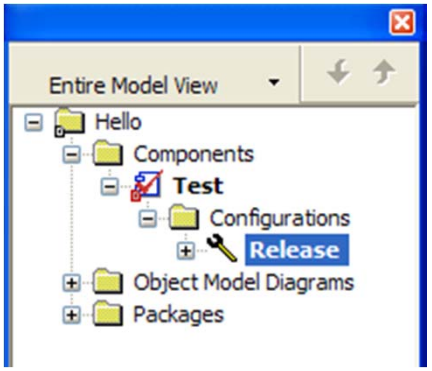


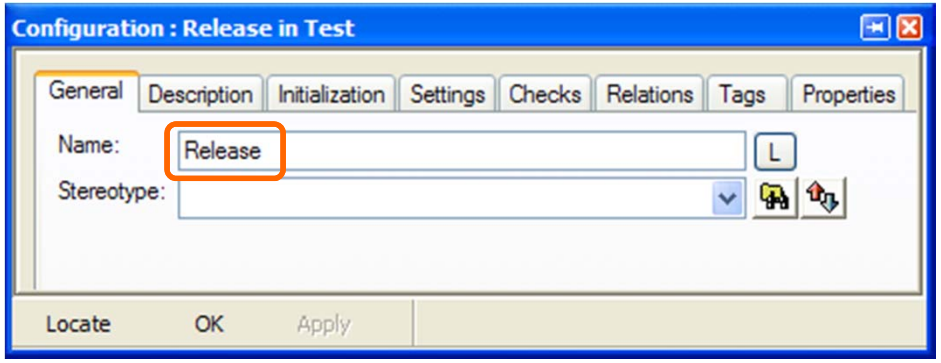


The Test component

- Now expand the *Configurations* tab under the *Test* component and rename the *DefaultConfig* to *Release* using the **General** tab.



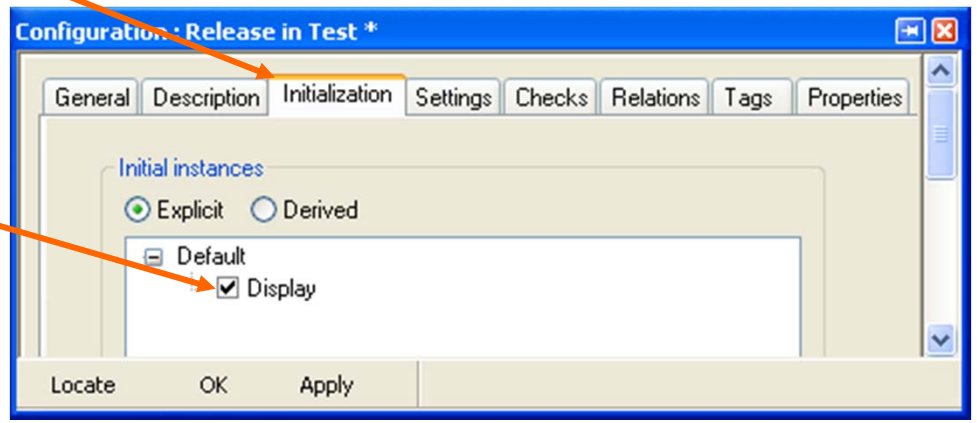




In a *component*, you tell Rational Rhapsody what to compile.
In a *configuration*, you define how to compile in detail.

Initial instance

- double-click the *Release* configuration to bring up the features.
- Select the **Initialization** tab, expand the *Default* package, and select the **Display** class.
- The main now executes and creates an initial instance of the *Display* class.

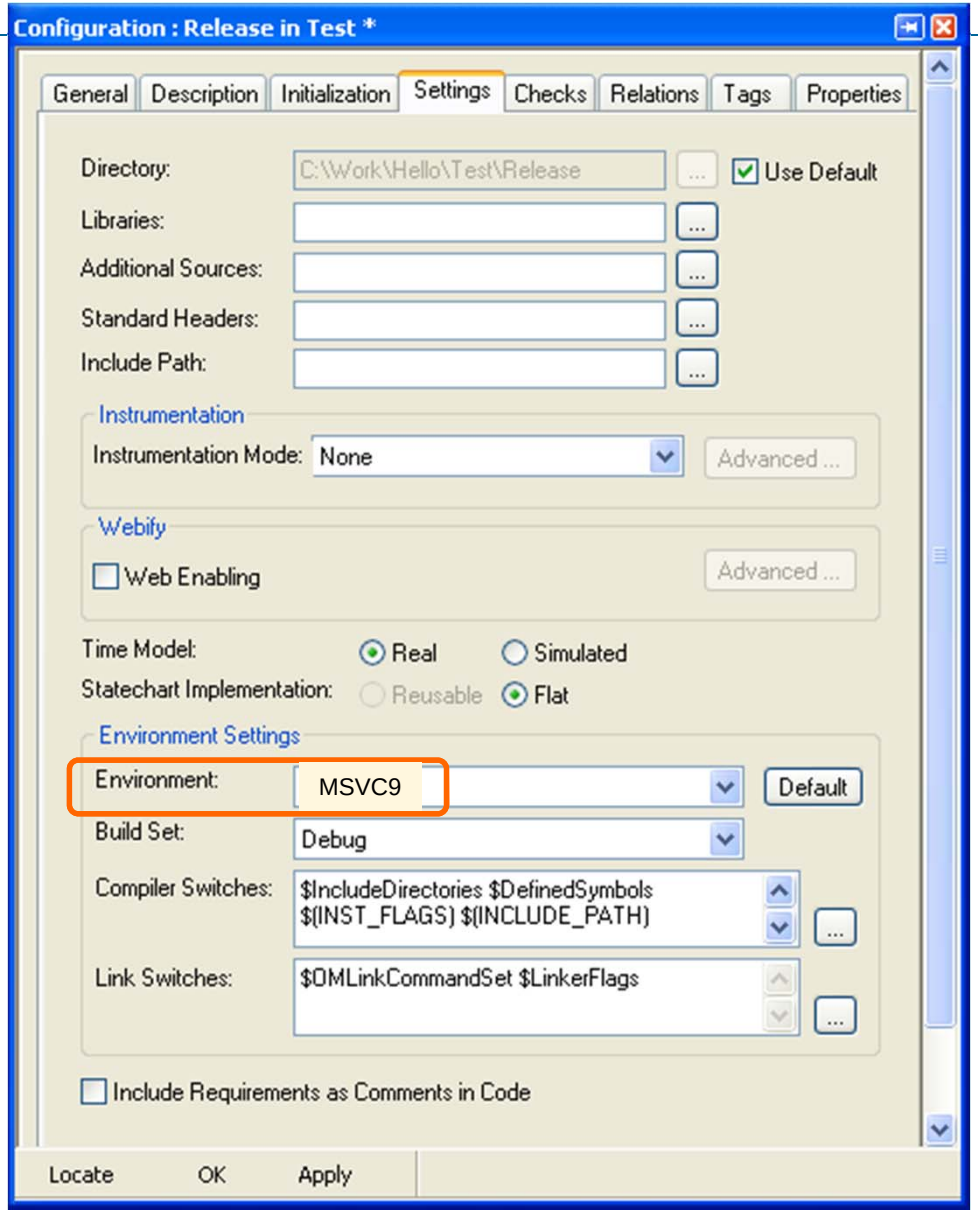


The difference between *explicit* and *derived* is explained later.

Settings

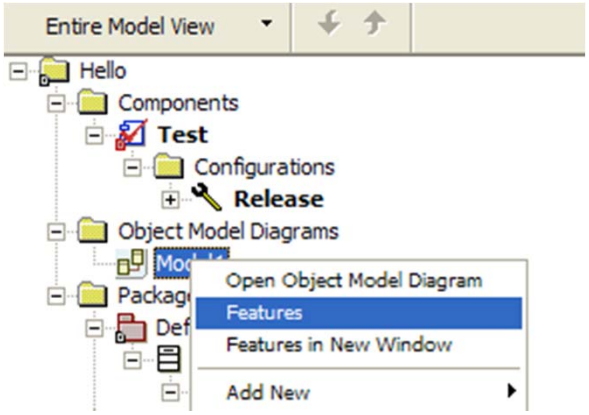
- You need to select an environment so that Rational Rhapsody knows how to create an appropriate Makefile.
- Select the **Settings** tab.
- Select the appropriate **Environment** – in our case - **MSVC9**.
- Click **OK**

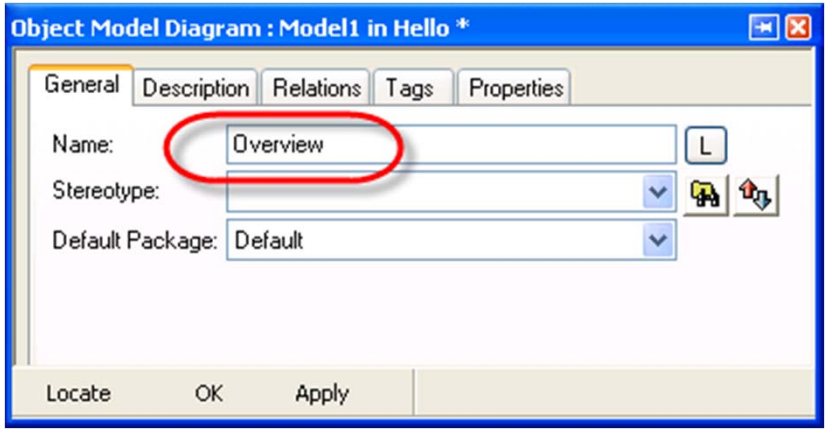
Many other settings are explained later. A Rational Rhapsody component can contain multiple configurations.



Renaming the OMD

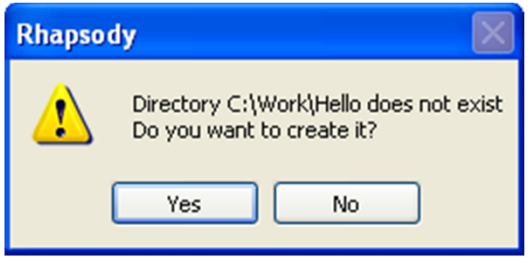
- Expand the Object Model Diagrams in the browser and use the features dialog to rename the diagram from *Model1* to *Overview*.
- **Apply** your change or click **OK** to apply and close the features dialog.





Generating code

- You are now ready to generate code.
- Save the model (click the Disk icon)
- Select the **Generate/Make/Run** icon
- Click **Yes** to create the directory




```
All Checks Terminated Successfully

Checker Done
0  Error(s), 0  Warning(s)

Code generated to directory: C:/Work/Hello/Test/Release
Generating file Display.h
Generating file MainTest.h
Generating file Display.c
Generating file MainTest.c
Generating make file Test.mak

Code Generation Done

0  Error(s), 0  Warning(s), 0  Message(s)
Building ------------  Test.exe   ------------
Executing: ""C:\Program Files\IBM\Rational\Rhapsody\7.5\Share"\etc\cygwinmake.bat Test.mak build "
Setting environment for Cygwin
"make.exe"
Compiling Display.c
Compiling MainTest.c
Linking Test.exe

Build Done
```

Log | Check Model | Build | Configuration Management | Animation

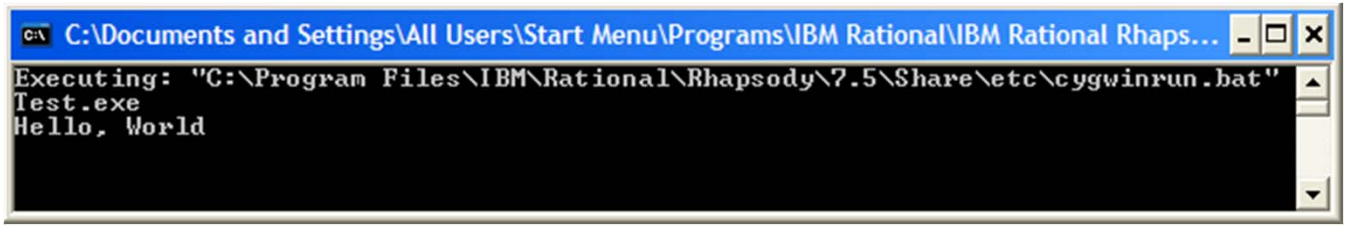
All Build Messages View

Severity	Model Element	Description	File	More
		Building ------------ Test.exe ------------		
		Executing: ""C:\Program Files\IBM\Rational\Rhapsody\7.5\Share"\etc\cygwinmake.bat Test.mak build "		
		Setting environment for Cygwin		
		"make.exe"		
		Compiling Display.c		
		Compiling MainTest.c		
		Linking Test.exe		
		Build Done		

Log | Check Model | **Build** | Configuration Management | Animation

Hello World

- You should see the following:

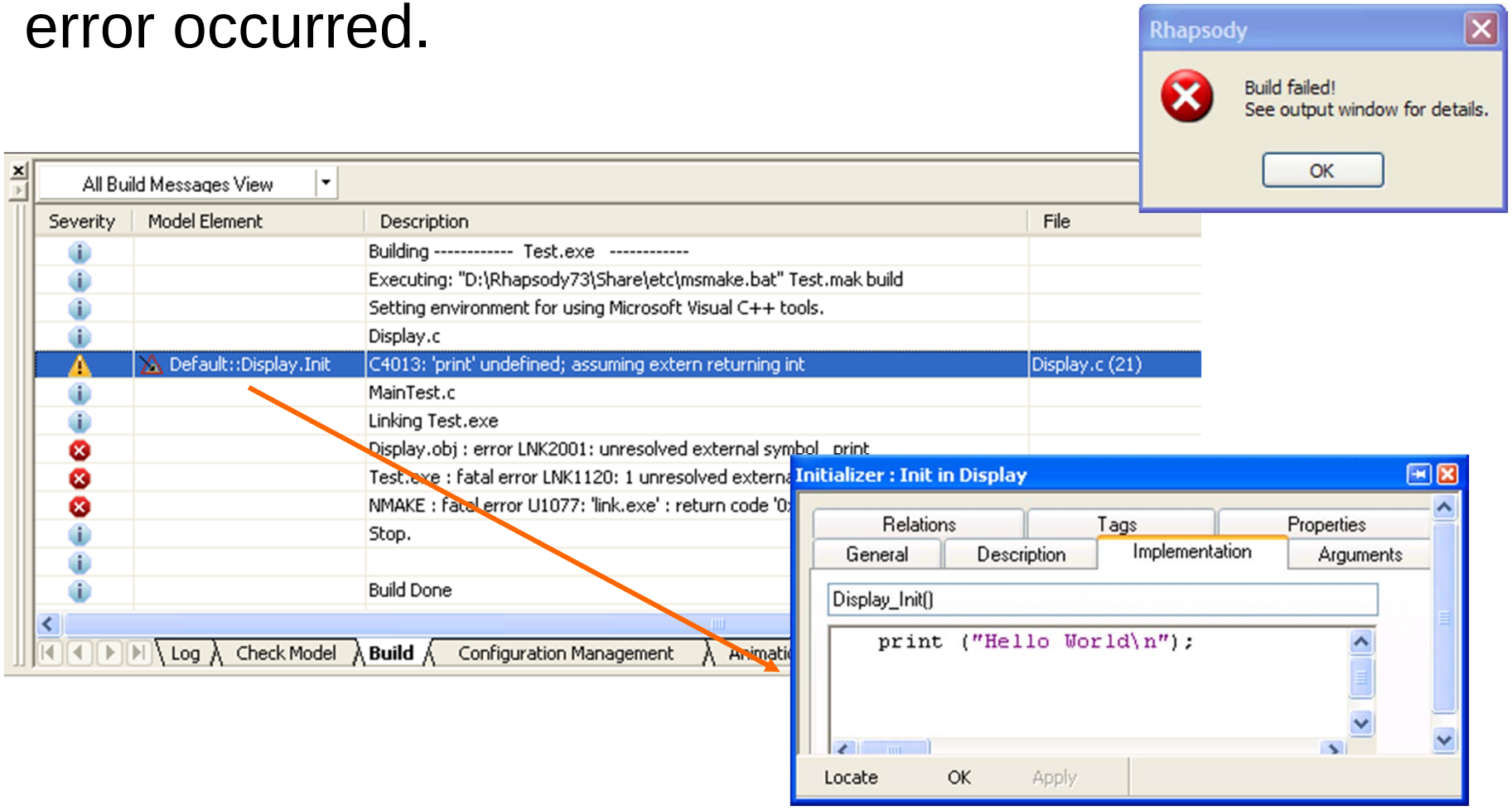


- Before continuing, make sure that you stop the executable. Do this either by closing the console window or by using the icon (white hand on stop sign).

If there was a compilation error during compilation, then simply double-click on the error and Rational Rhapsody indicates where in the model the error occurred.

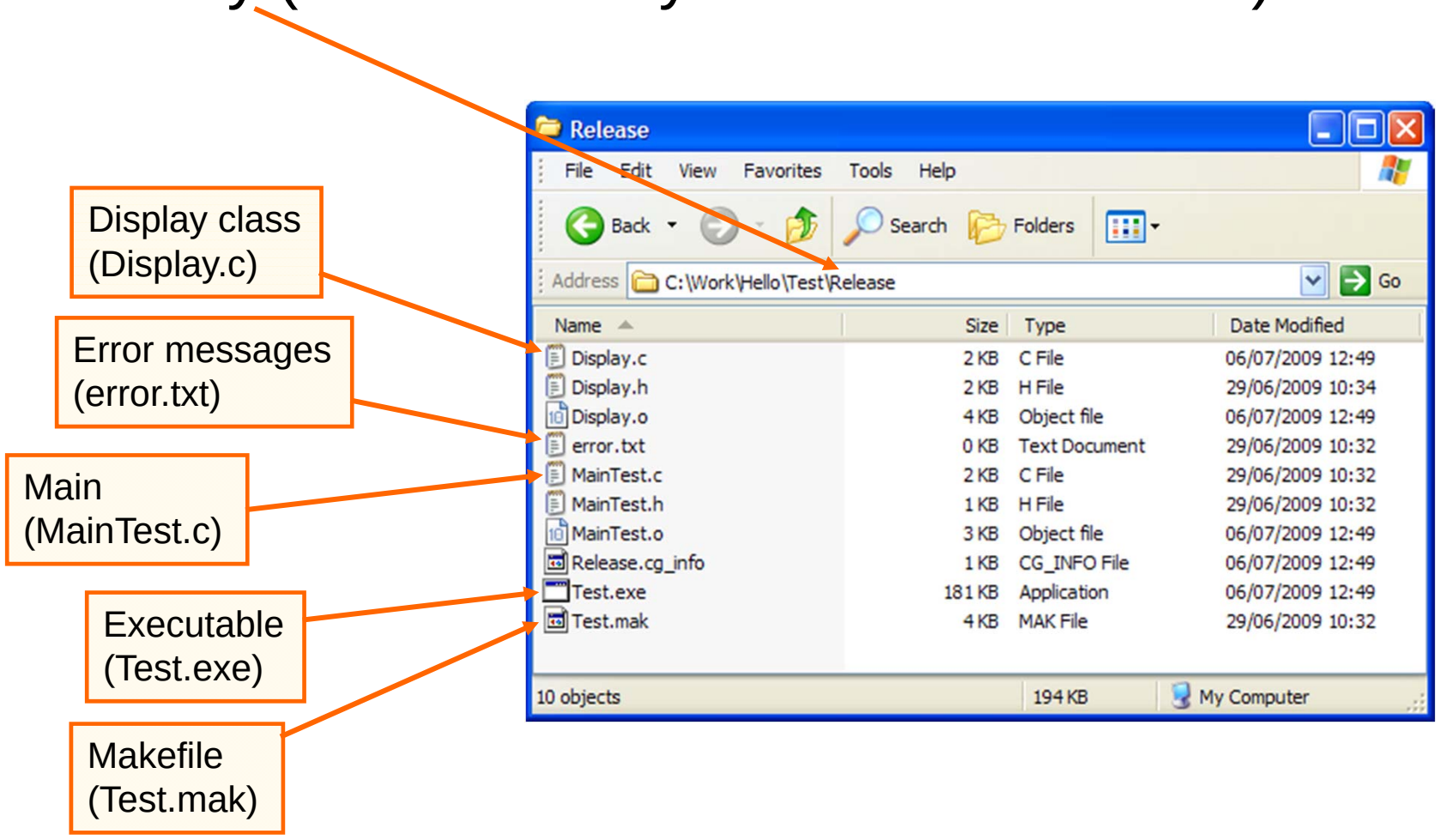
Handling errors

- If there are errors during the compilation, then double-click the relevant line to find out where the error occurred.



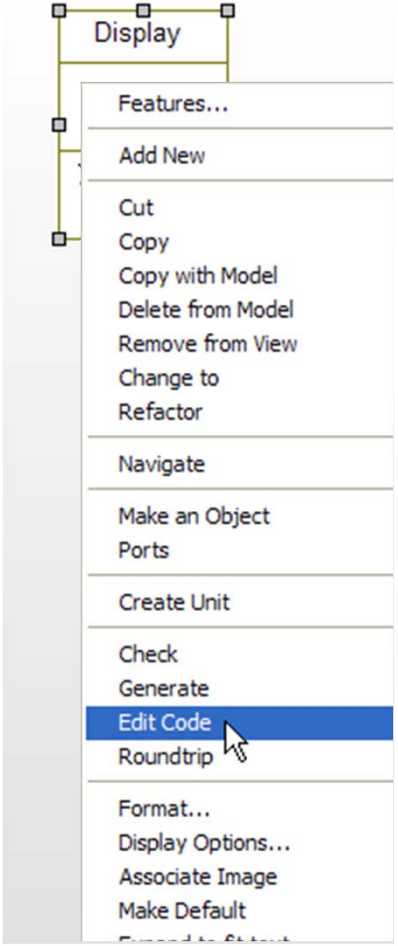
The generated files

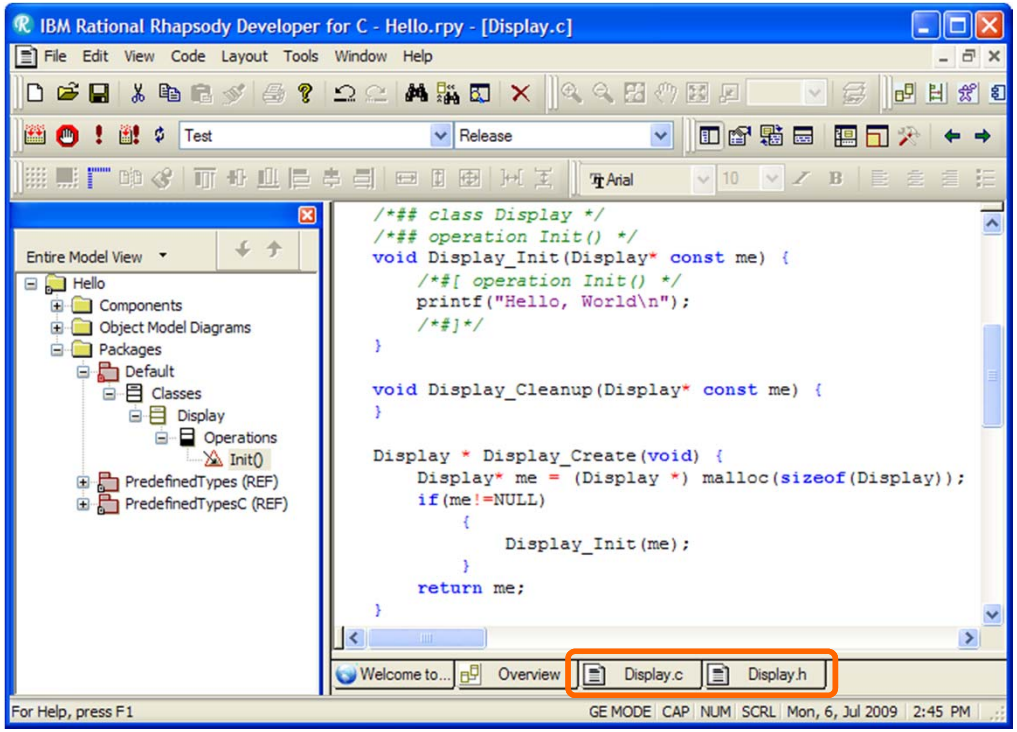
- The generated files are located in the following directory (YourDirectory\Hello\Test\Release):



Editing the code

- You can edit the generated files from within Rational Rhapsody.
- Select the *Display* class in the diagram or in the browser
- Right-click and select **Edit Code**.





Generated code

- You can see that the class Display is a *struct*; however, because you have no attributes or relations, it is empty.
- Any public operation of the `Display` class is prefixed with *Display_* to make it a member operation of the class.
- There are several properties to customize the code generation. These are discussed later.

```
#ifndef Display_H
#define Display_H

/*## auto_generated */
#include <oxf/Ric.h>
/*## package Default */

/*## class Display */
typedef struct Display Display;
struct Display {
    RIC_EMPTY_STRUCT
};

/***    User implicit entries    ***/

/* Constructors and destructors:*/

/*## operation Init() */
void Display_Init(Display* const me);

/*## auto_generated */
void Display_Cleanup(Display* const me);

/*## auto_generated */
Display * Display_Create(void);

/*## auto_generated */
void Display_Destroy(Display* const me);

#endif
/*********************************************
```

Auto generated operations

- **Display_Init** is an Initializer that is used to initialize an object after it has been created. It is the equivalent of a C++ constructor.
- **Display_Cleanup** is used to clean up any allocated memory (and so on) before the object gets destroyed. This is the equivalent of a C++ destructor.
- **Display_Create** is used to dynamically create an instance of the object_type. It is the equivalent of *new Display* in C++.
- **Display_Destroy** is used to delete an instance of the object_type. It is the equivalent of *delete Display* in C++.

> You will see later how to generate instances statically, and also how to avoid generating the Create and Destroy operations.

What is this *me* pointer?

- You could have several instances of a class, so all operations must know which instance they can access. In C++, there is the *this* pointer that gets passed automatically. In C, *this* is a reserved word, so instead, a pointer to the class called *me* is used.

```
/*## class Display */
/*## operation Init() */
void Display_Init(Display* const me) {
    /*#[ operation Init() */
    printf("Hello, World\n");
    /*#]*/
}

void Display_Cleanup(Display* const me) {
}
```

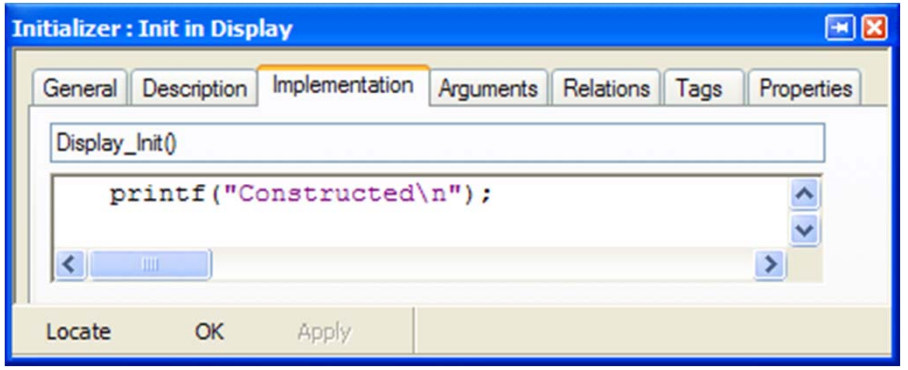
Later, you will look at a way of eliminating the *me* pointer for classes that have only one single instance.

Modifying the code

- You can modify the generated code.
- In the Display.c file, change the implementation to print out *Constructed* instead of *Hello World*.
- Transfer the focus back to another window to roundtrip the modifications back into the model.
- Note that the model has been updated automatically.

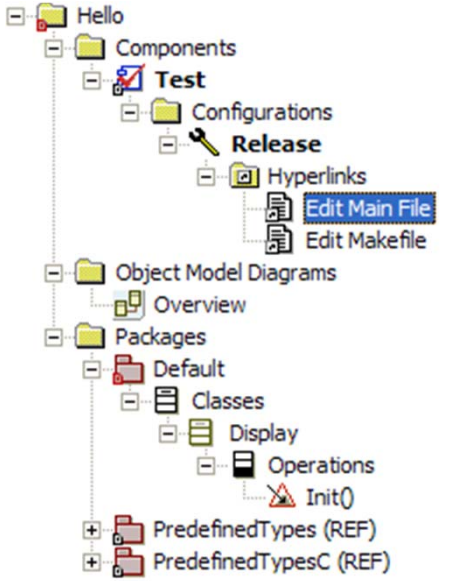
```
/*## class Display */
/*## operation Init() */
void Display_Init(Display* const me) {
    /*#[ operation Init() */
    printf("Constructed\n");
    /*#]*/
}

void Display_Cleanup(Display* const me) {
}
```



Displaying the Main File and Makefile

- The Main File and Makefile can be displayed from within Rational Rhapsody by double-clicking the hyperlinks:



MainTest.c

```
/*## auto_generated */
#include "MainTest.h"
/*## auto_generated */
#include "Display.h"
int main(int argc, char* argv[]) {
    int status = 0;
    if(RiCOXFInit(argc, argv, 6423, "IBM-70E7B177341", 0, 0, RiCTRUE))
        {
            Display * p_Display;
            p_Display = Display_Create();
            /*#[ configuration Test::Release */
            /*#]*/
            RiCOXFStart(FALSE);
            Display_Destroy(p_Display);
            status = 0;
        }
    else
        {
            status = 1;
        }
    return status;
}
```

Project files

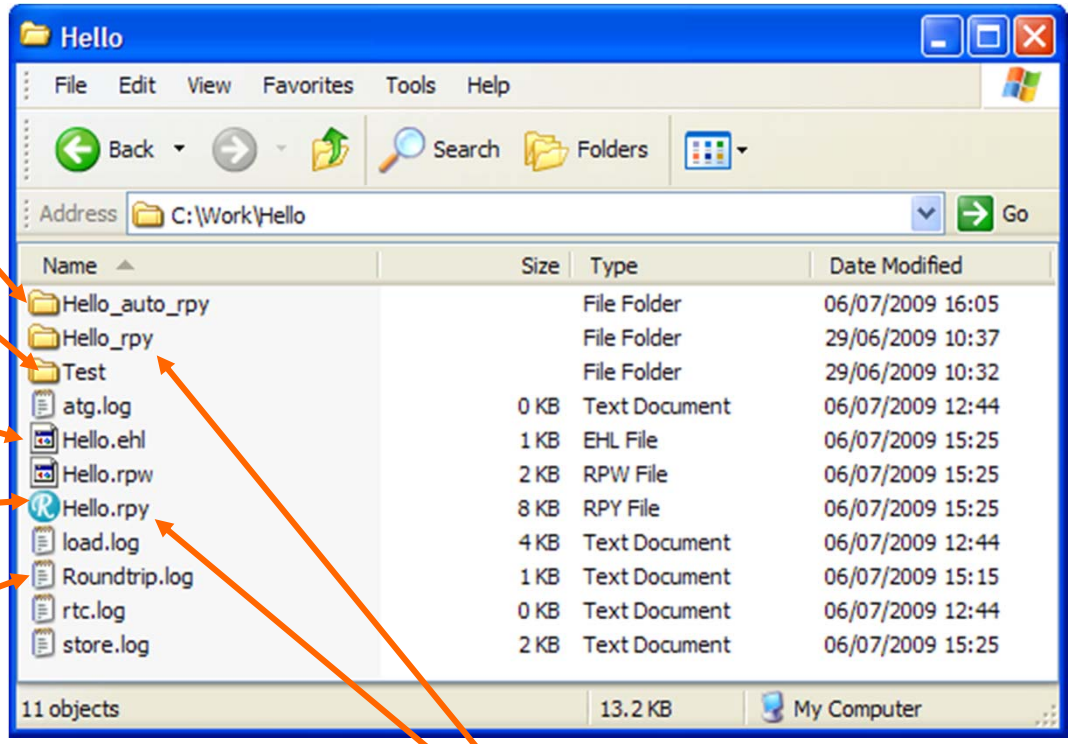
AutoSave
(Hello_auto_rpy folder)

Generated code
(Hello_rpy folder)

Event history list
(Hello.ehl)

Project workspace
(Hello.rpy)

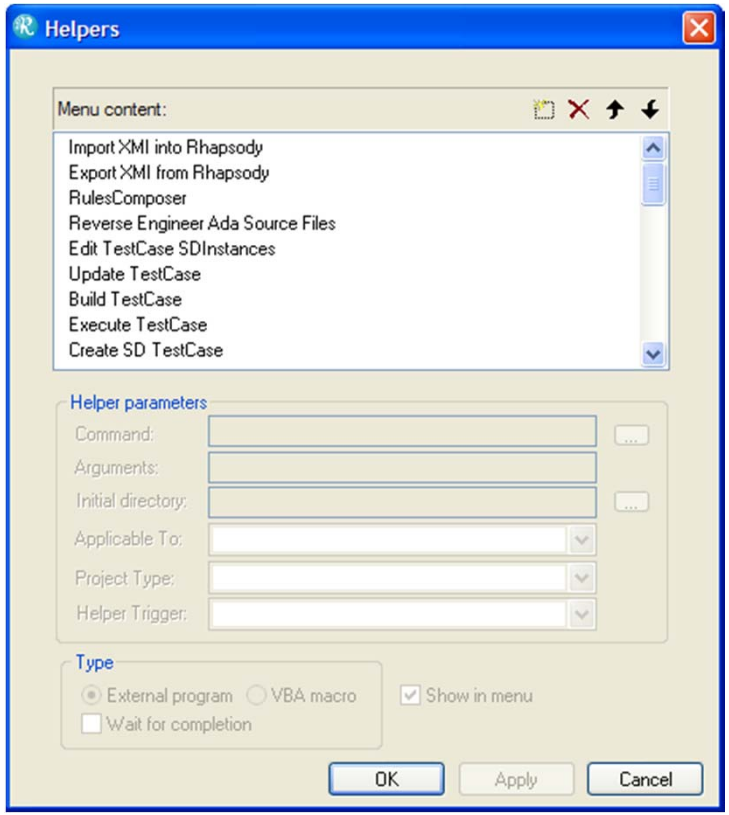
Roundtrip log
(Roundtrip.log)

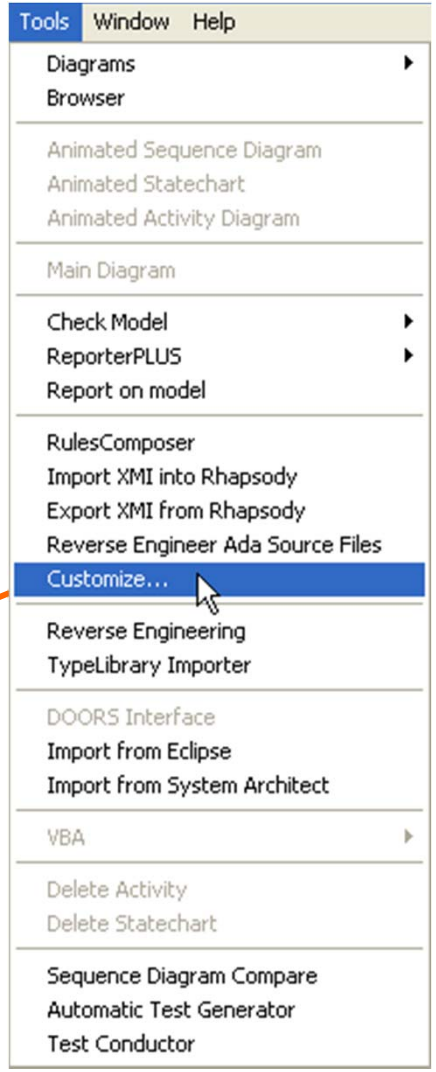


The model
(Hello_rpy folder and Hello.rpy file)

Extended exercise

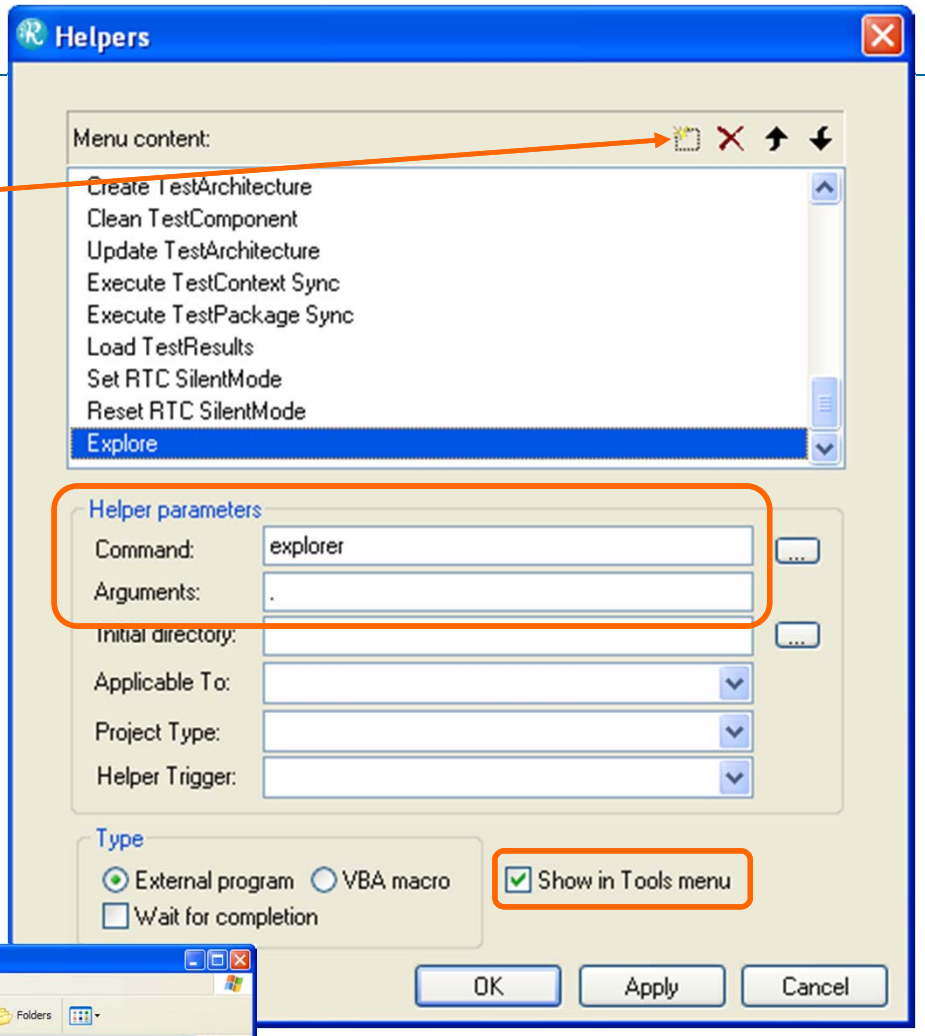
- You can customize Rational Rhapsody to allow quick access to the location of current project.
- Select **Tools > Customize**.

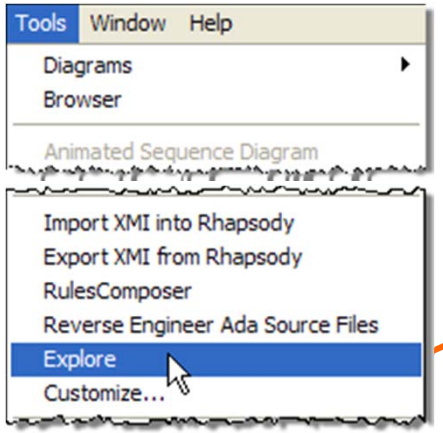


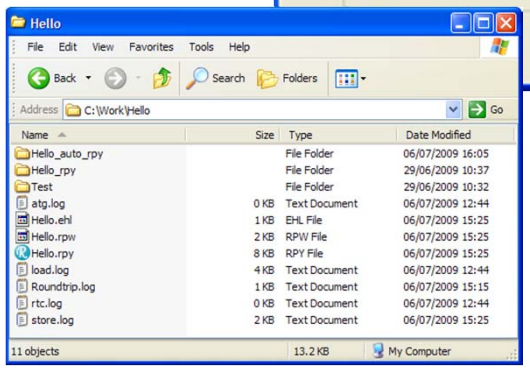


Customize

- Use the New icon to enter a new entry *Explore* to the **Tools** menu.
- Enter `explorer` in the **Command** field.
- Set **Arguments** to **.** (a period).
- Select the option **Show in Tools menu**.
- Click the **OK** button.
- Select **Tools > Explore**.



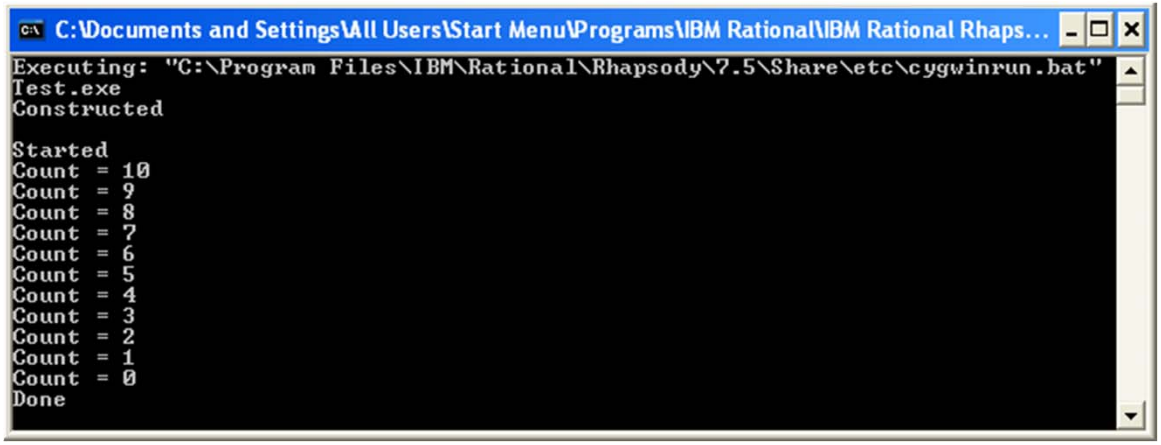




Where are we?

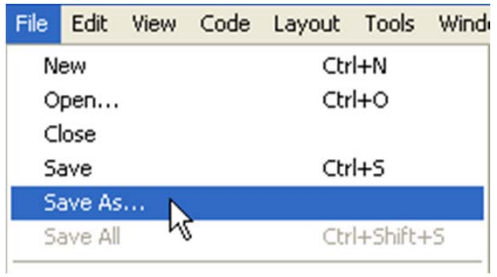
- Exercise 1 : Hello World
  - You start with the simplest example possible, just a single object that prints out Hello World.
- Exercise 2 : Count Down
  - Next, you create a simple counter using a simple statechart.
- Exercise 3 : Dishwasher
  - Finally, you create a dishwasher and a more complex statechart.
- Summary

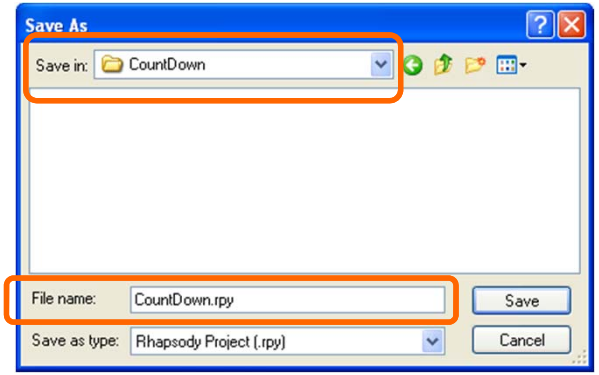
Exercise 2: Count down

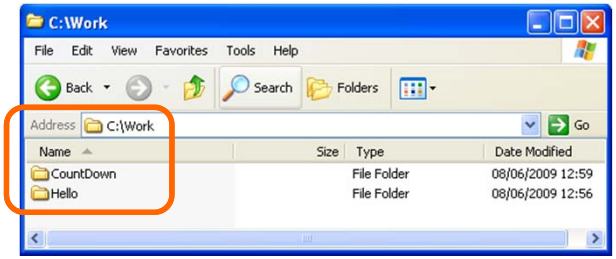


Copying a project

- Select **File > Save As**.
- Select (up a level) to select your working folder.
- Select (new folder) to create a new folder.
- Rename New Folder to CountDown.
- Select the CountDown folder.
- Save the project as CountDown.rpy.
- The new CountDown project is opened in Rational Rhapsody with the previous work-space preserved.



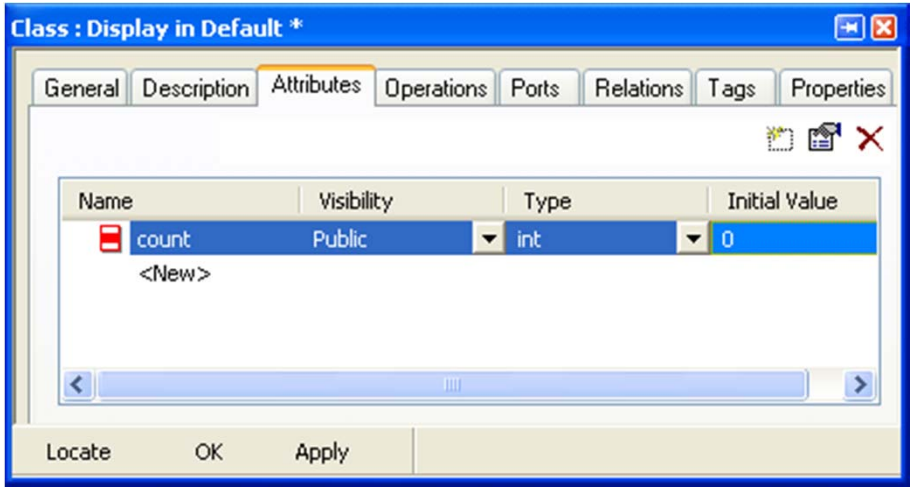




Each time there is an auto-save, Rational Rhapsody only saves what has changed since the last manual save.

Adding an attribute

- To add an attribute, double-click the *Display* class to bring up the features, and then select the **Attributes** tab.
- Click **New** to add an attribute `count` of type `int`.
- Set the initial value to `0`.



Generated code

- The attribute `count` is declared in the `struct` in Display.h:

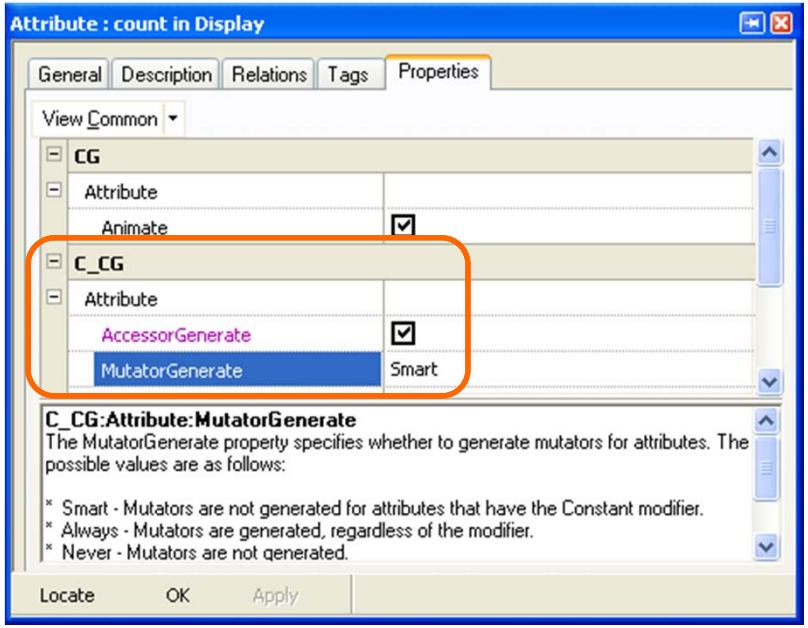
```
/*## class Display */
typedef struct Display Display;
struct Display {
    int count;      /*## attribute count */
};
```

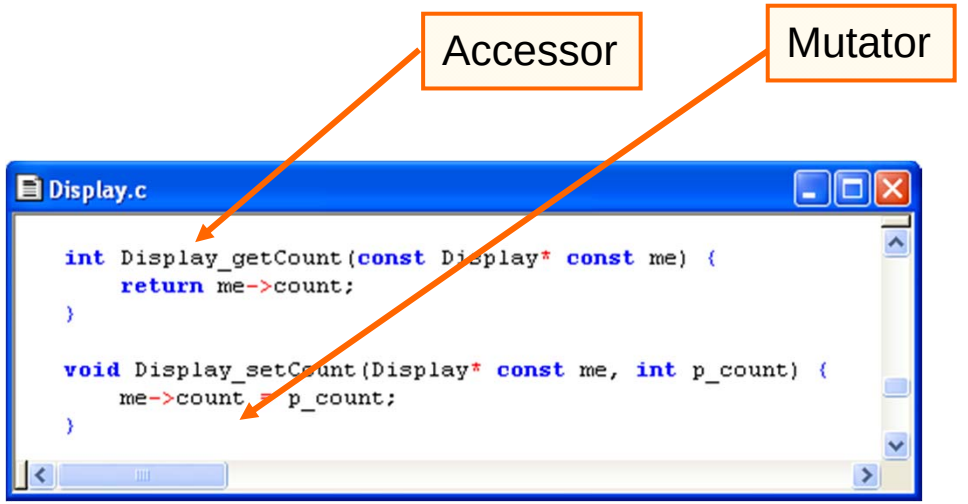
- The attribute `count` gets initialized by the Initializer in Display.c

```
/*## class Display */
/*## operation Init() */
void Display_Init(Display* const me) {
    me->count = 0;
    {
        /*#[ operation Init() */
        printf ("Constructed\n");
        /*#]*/
    }
}
```

Additional code for an attribute

- Rational Rhapsody also generates object oriented (OO)-style accessor and mutator operations to encapsulate attributes, if you enable the corresponding properties of the `count` attribute in the features dialog under the **Properties** tab:



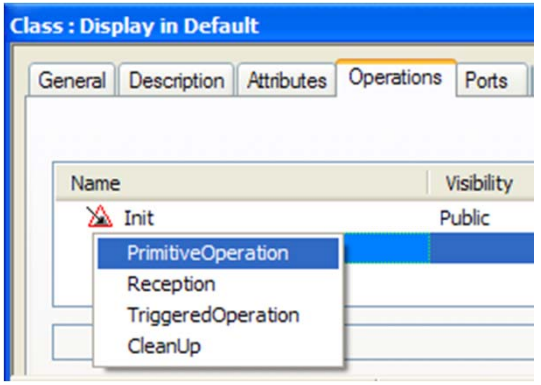


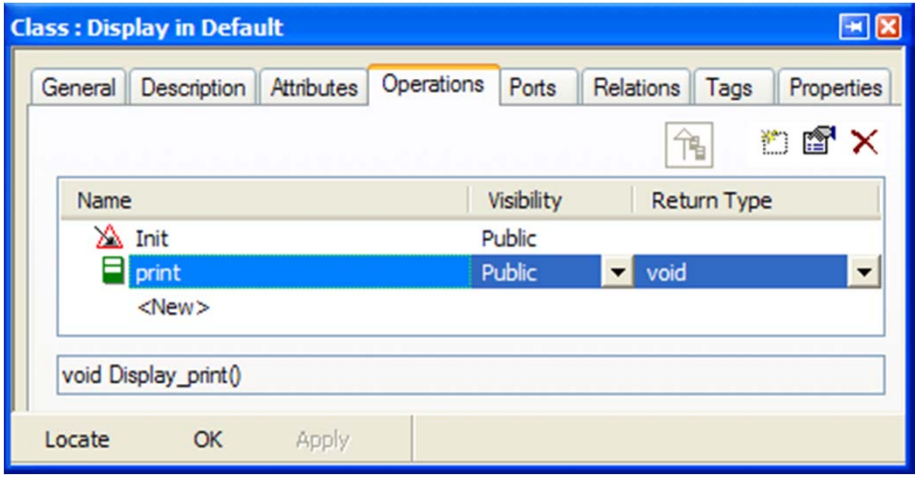
What are accessors and a mutators?

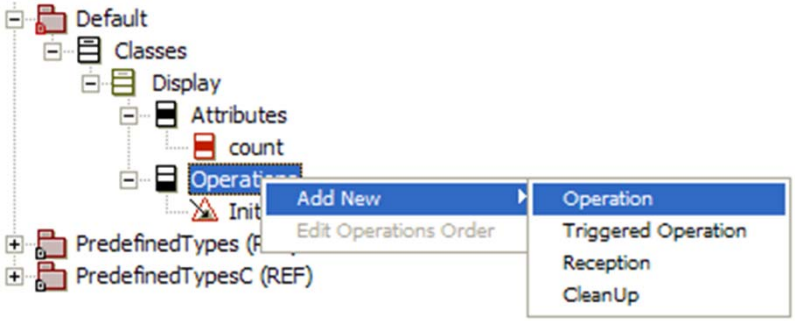
- If one needs access to the attributes, then they should use an accessor, such as `Display_getCount()`, or a mutator, such as `Display_setCount()`.
- This allows the designer of a class the freedom to change the type of an attribute without having to alert all users of the class. The designer needs to modify the accessor and mutator implementation.
- In most cases, attributes do not need accessors or mutators, so by default they are not generated.

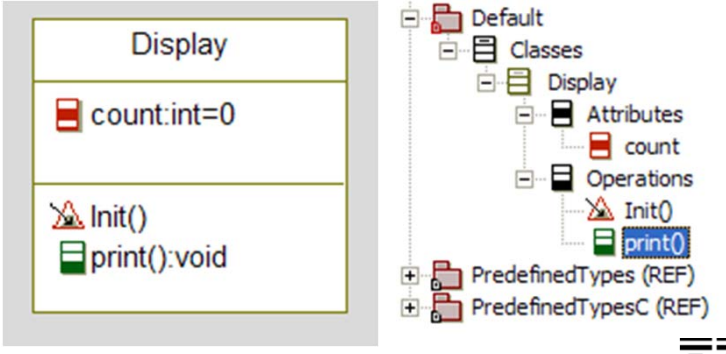
Adding an operation

- Use the features for the `Display` class, select the **Operations** tab, and add a new primitive operation `print`.
- Or, right-click the Display class (or Operations in the browser) and select ***Add New > Operation*** to add `print`.



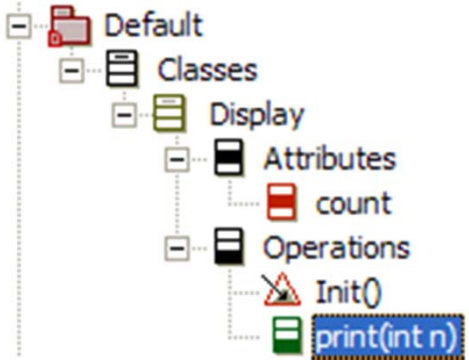


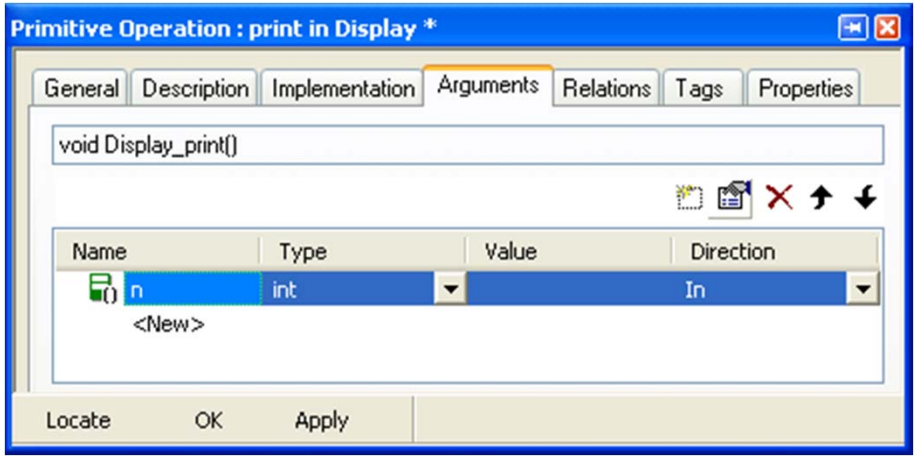


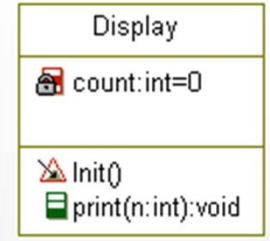


Arguments for operation print()

- Double-click **print** to open the features for the print operation.
- Add an argument `n` of type `int`.



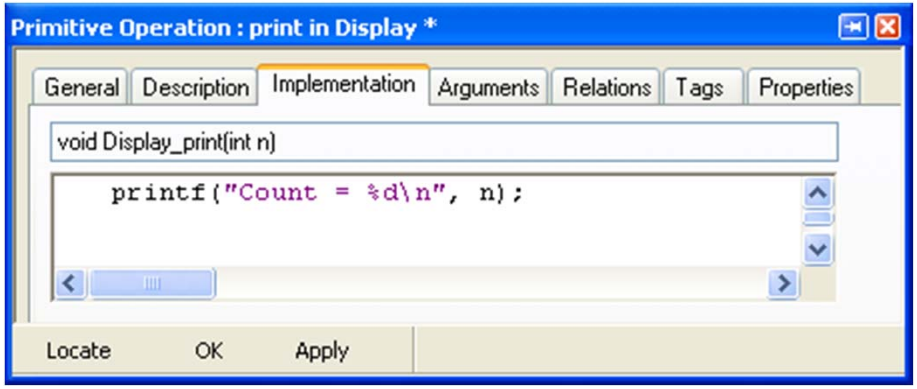




Adding implementation

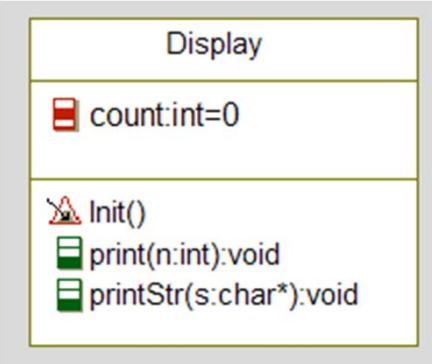
- Select the **Implementation** tab for the `print` operation and add:

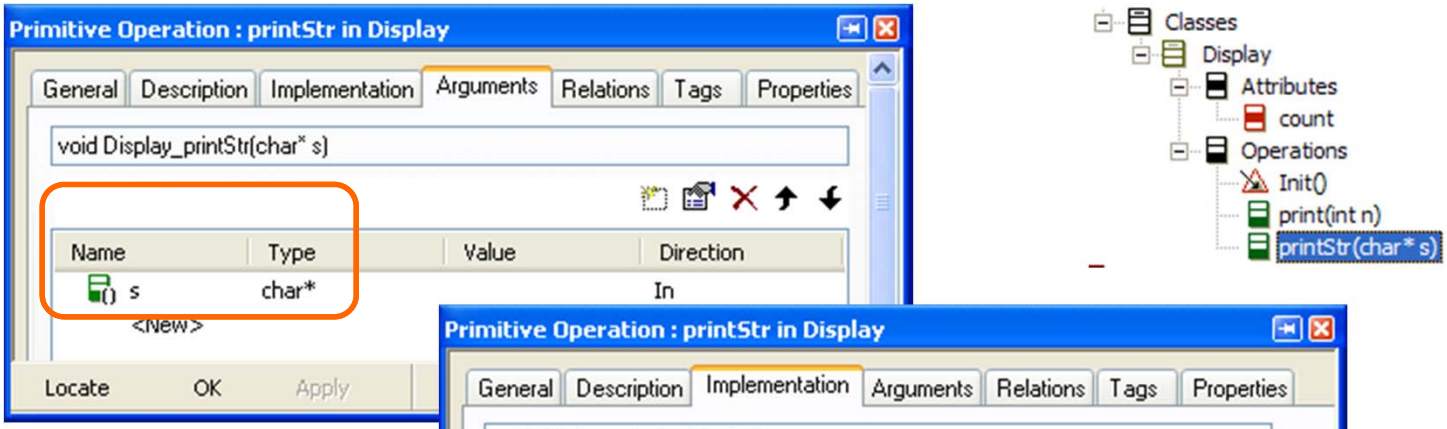
```
printf (“Count = %d\n“, n);
```

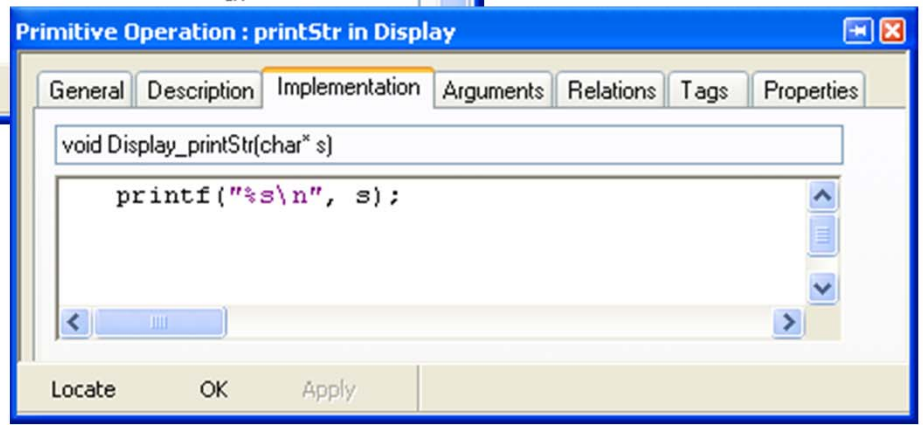


Operation printStr(char *)

- In a similar way, add another operation called `printStr`, this time with an argument `s` of type `char*` and with implementation: `printf ("%s\n",s);`
- Are your operations not visible as above?
  - (1) Right-click on the class, (2) choose "Display Options", (3) on the "Operations" tab, select "Show:All".



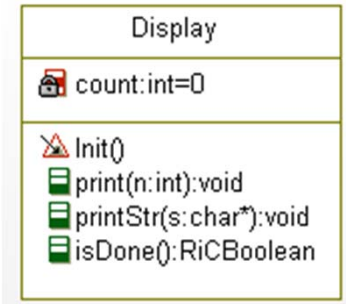


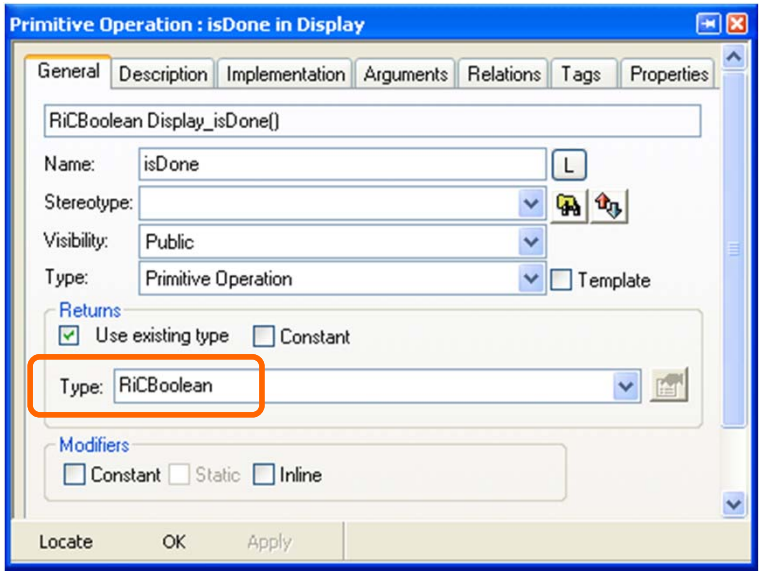


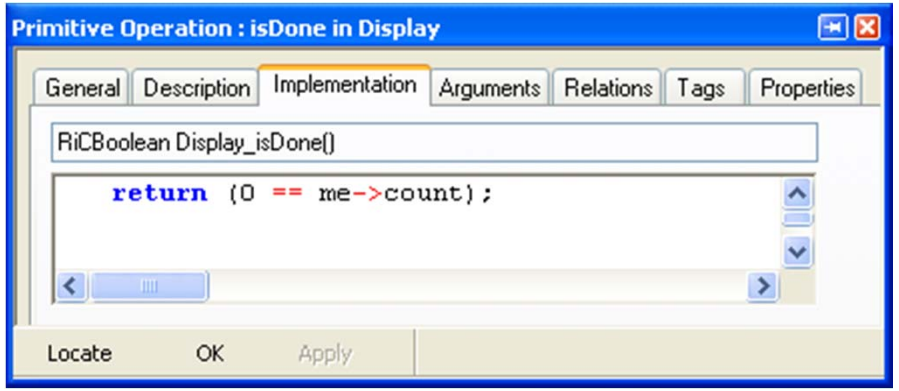
Operation isDone()

- Add another operation called `isDone` that returns a `RiCBoolean` and has the following implementation:

```
return (0 == me->count);
```



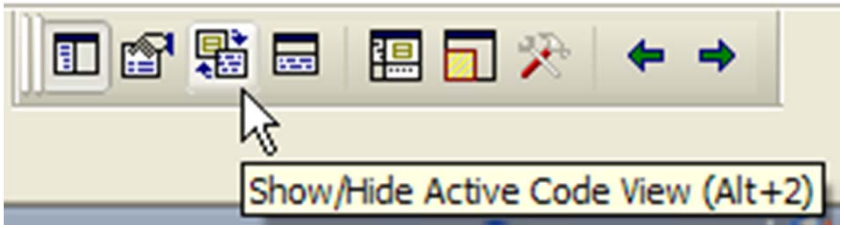




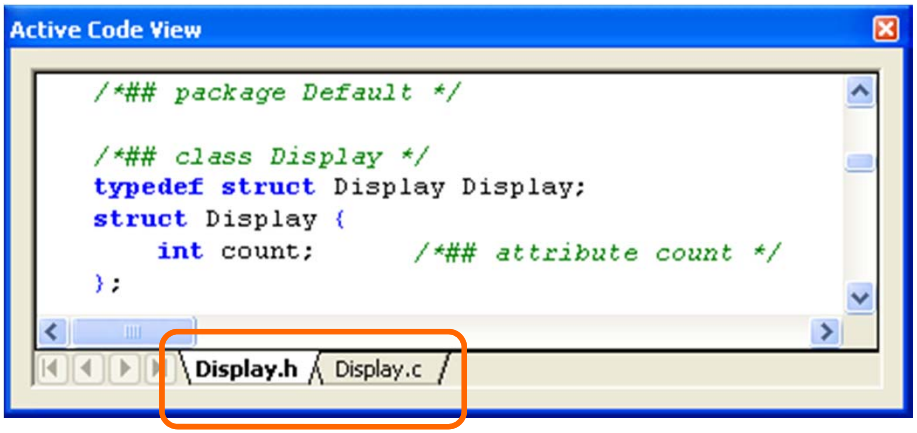
Typing `0==me->count` instead of `me->count==0` enables the compiler to detect the common error of where = is typed instead of ==.
RiCBoolean is defined in rhapsody\share\langC\oxf\RiCTypes.h as an unsigned char.

Active code view

- Click **Active Code View**.



- The active code view is context-sensitive and automatically updates as the model is changed. The window changes to dynamically show the code for any highlighted model element.

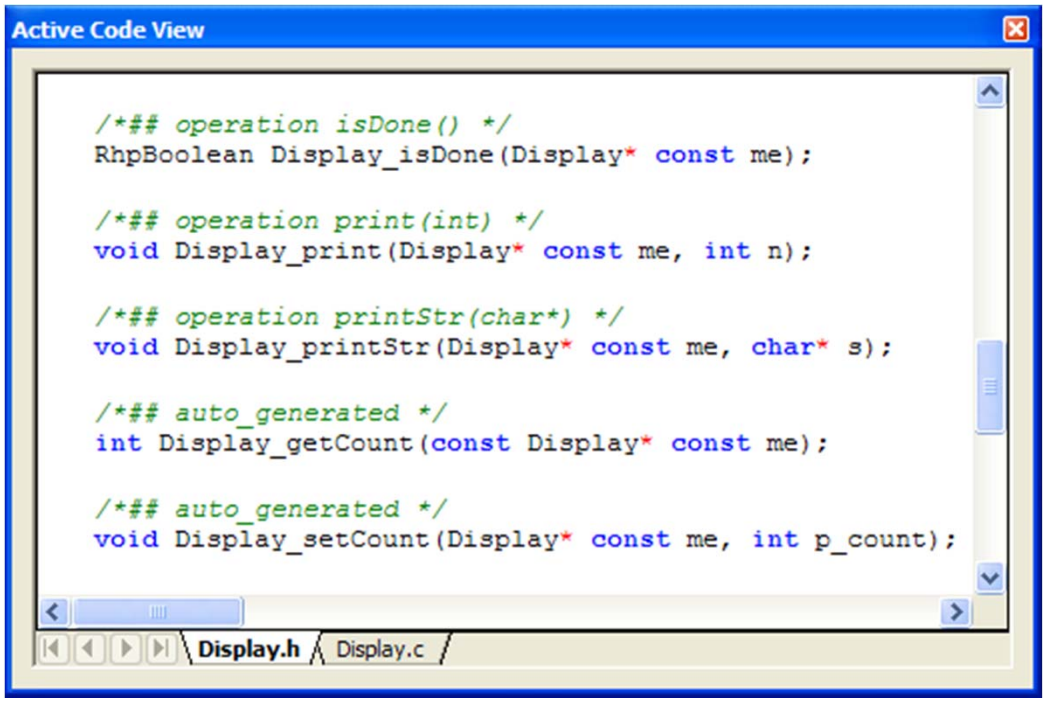


Although leaving the active code view open is useful, it does slow down model manipulation, because the code regenerates anytime any model element gets modified.

IBM

Generated code

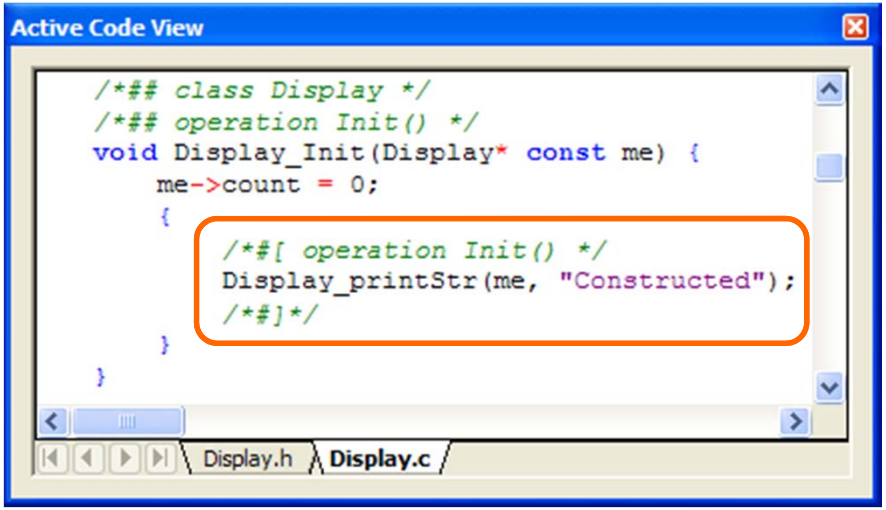
- Because the added operations are public, the name of the generated operations are preceded by the name of the class.
- This avoids potential name clashing with other classes.

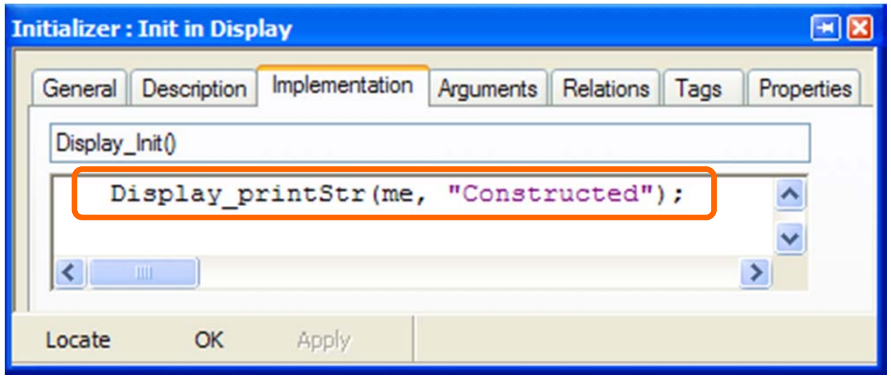


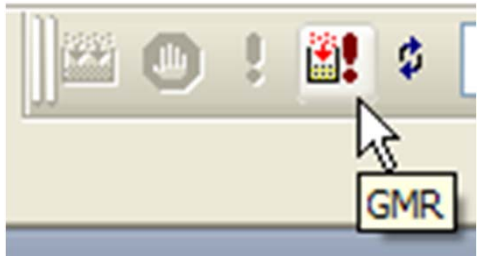
Note that you have to pass the me pointer as the first argument of each operation to specify an instance of the class you access.

Using the print operation

- In the Active Code View, (make sure you have selected the **Display.c** tab), change the code for the *Initializer* to use the `Display_printStr` operation.
- Change the focus to another window, such as the browser, and check that this modification has been automatically round-tripped.
- Save, and then click the Generate/Make/Run icon.

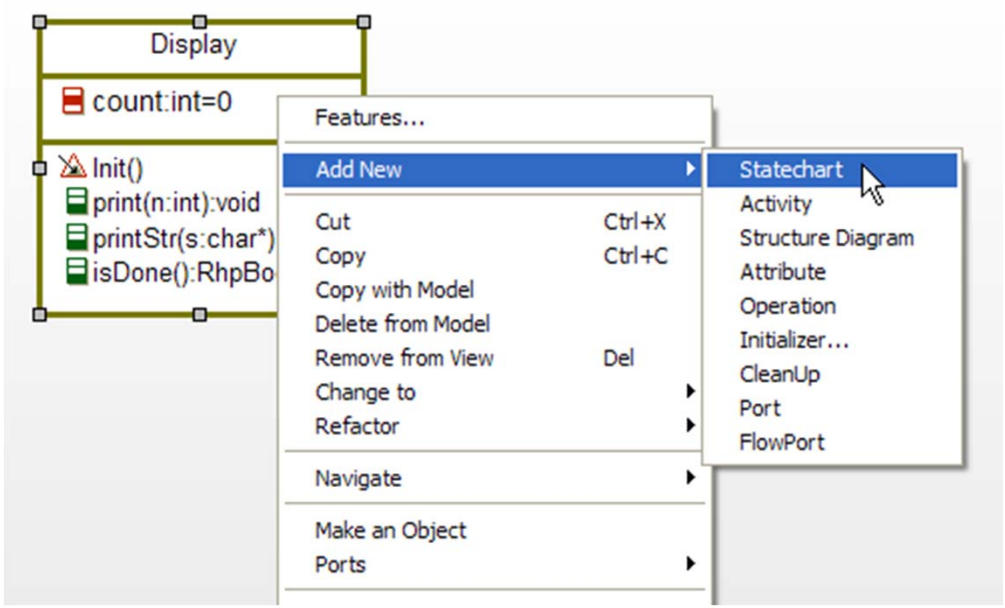






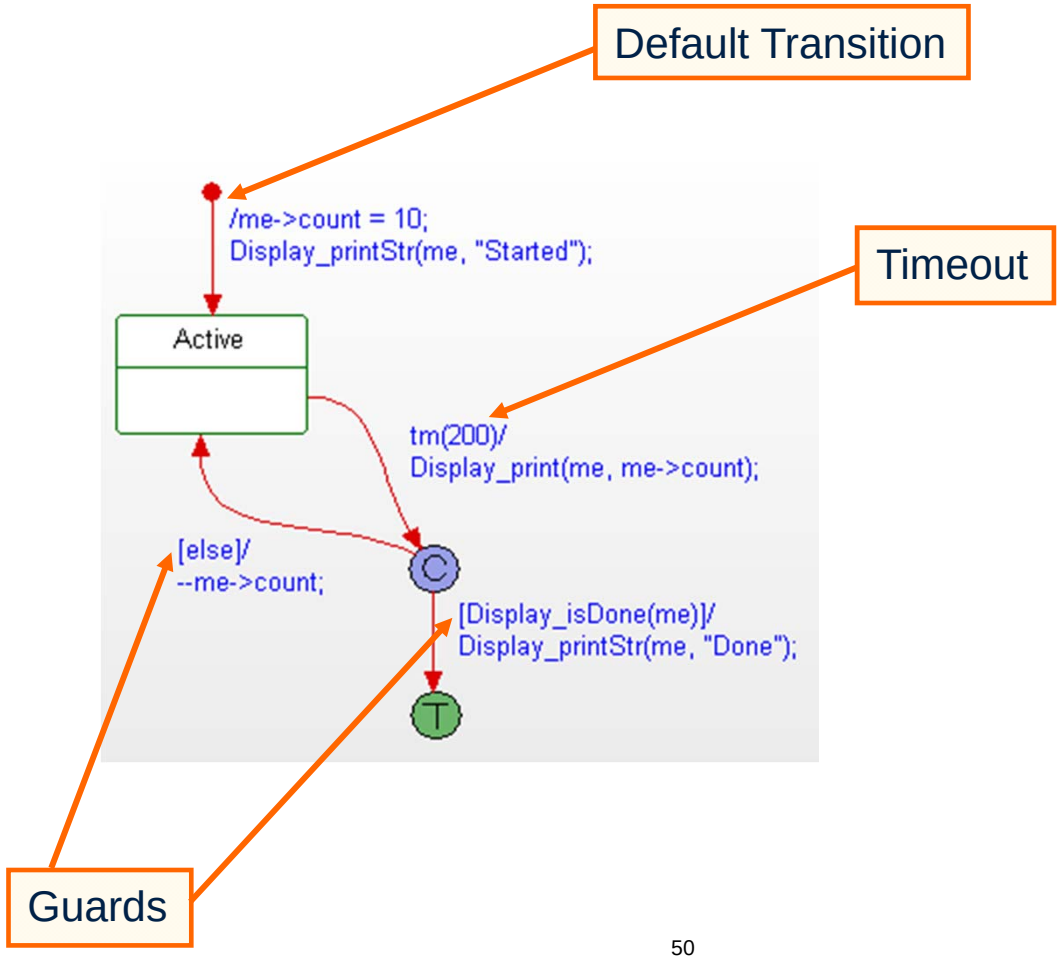
Adding a statechart

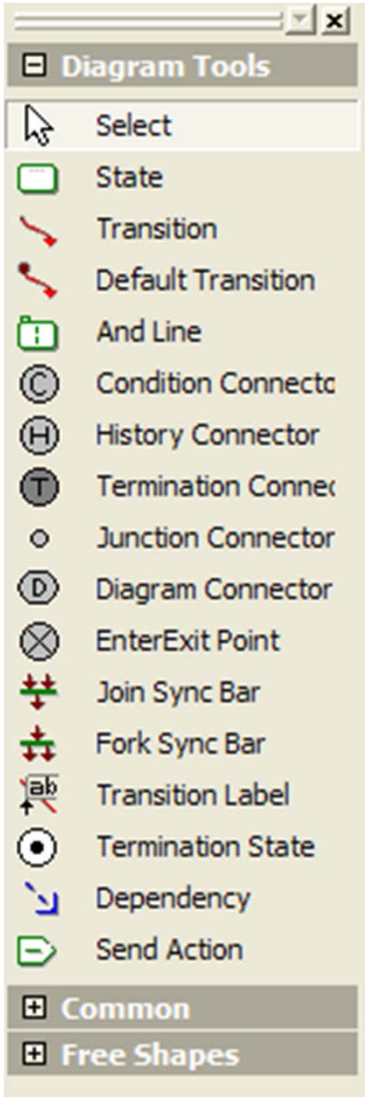
- You would like to get the `Display` class to count down from 10 to 0 in intervals of 200ms.
- In order to do so, you need to give some behavior to the class. You can do this by adding a statechart.
- Right-click the `Display` class and select **New Statechart**.



Simple statechart

- Draw the following statechart
  - Some hints in the comming slides. But see how far you can go by yourself!





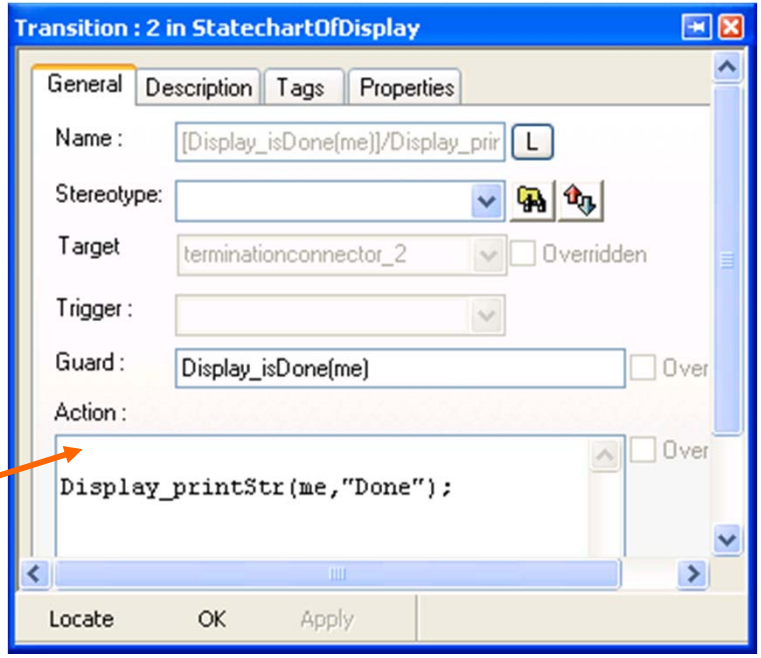
Transitions

- Once a transition has been drawn, there are two ways in which to enter information:
  - In text format, for example:

    ```
    [Display_isDone(me)]/Display_printStr(me,“Done”);
    ```

  - By the features of the transition (activated by double-clicking or right-clicking the transition).

An empty line forces the action to appear on a new line.



Timer mechanism

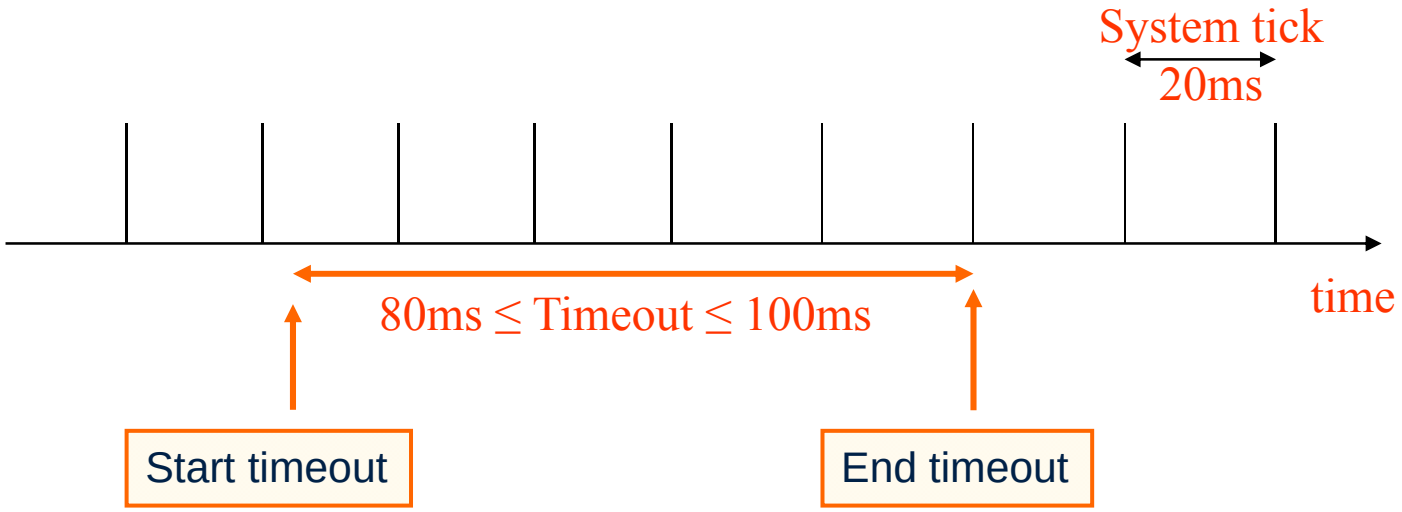
- A timer is provided that you can use within the statecharts.
- `tm(200)` acts as an event that is taken 200ms after the state has been entered.
- On entering into the state, the timer is started.
- On exiting from the state, the timer is stopped.

```
tm(200)/
Display_print(me,me->count);
```

The timer uses the OS Tick and only generates timeouts that are a multiple of ticks.

Timeouts

- If there is a system tick of, say, 20ms and you ask for a timeout of 65ms, then the resulting timeout will actually be between 80ms and 100ms, depending on when the timeout is started relative to the system tick.



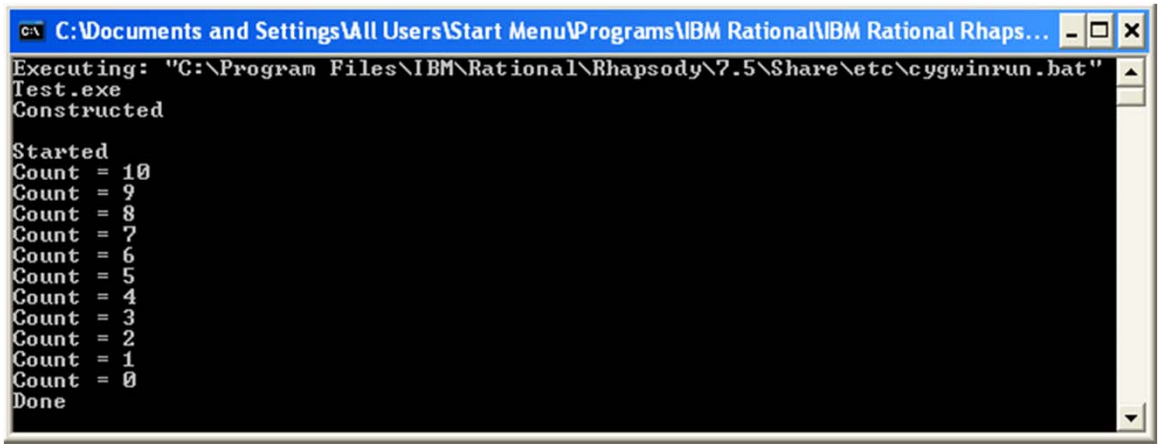
If precise timeouts are required, then it is recommended to use a hardware timer in combination with triggered operations.

Counting down

- Save, and then Generate/Make/Run

Initializer

Default transition



Do not forget to close this window before doing another Generate/Make/Run.

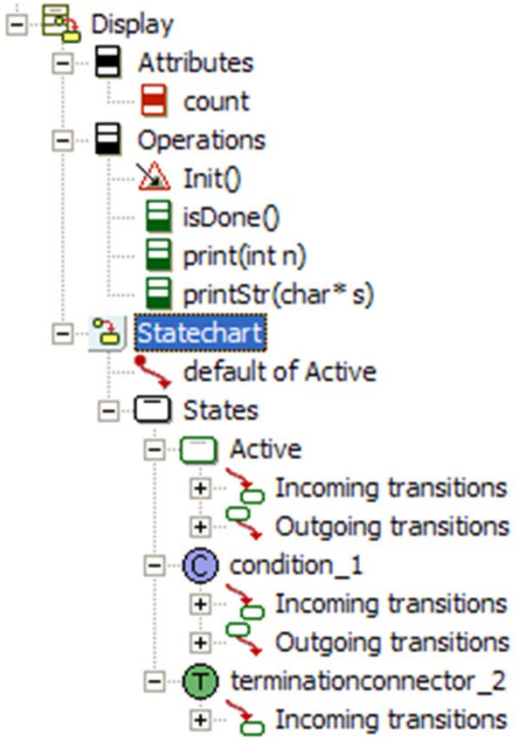
Make sure you understand …

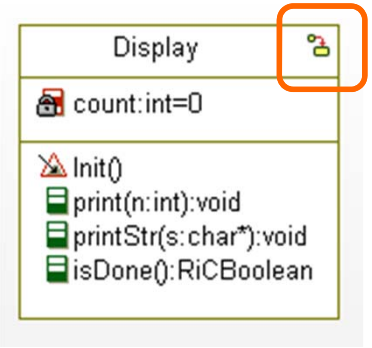
- Now is the time!
  - Do you understand the behaviour of the state machine? Explain it to your partner!
  - Can you related the produced output to the state machine model?
  - Do you understand the relation between the graphical model, and the C code behind it?

Statechart symbol

- Now that the `Display` class is reactive, it has a special symbol in both the:
  - Browser
  - OMD

- Statechart appears in the browser and indicates that the features dialog can be used to access the state / transition details.

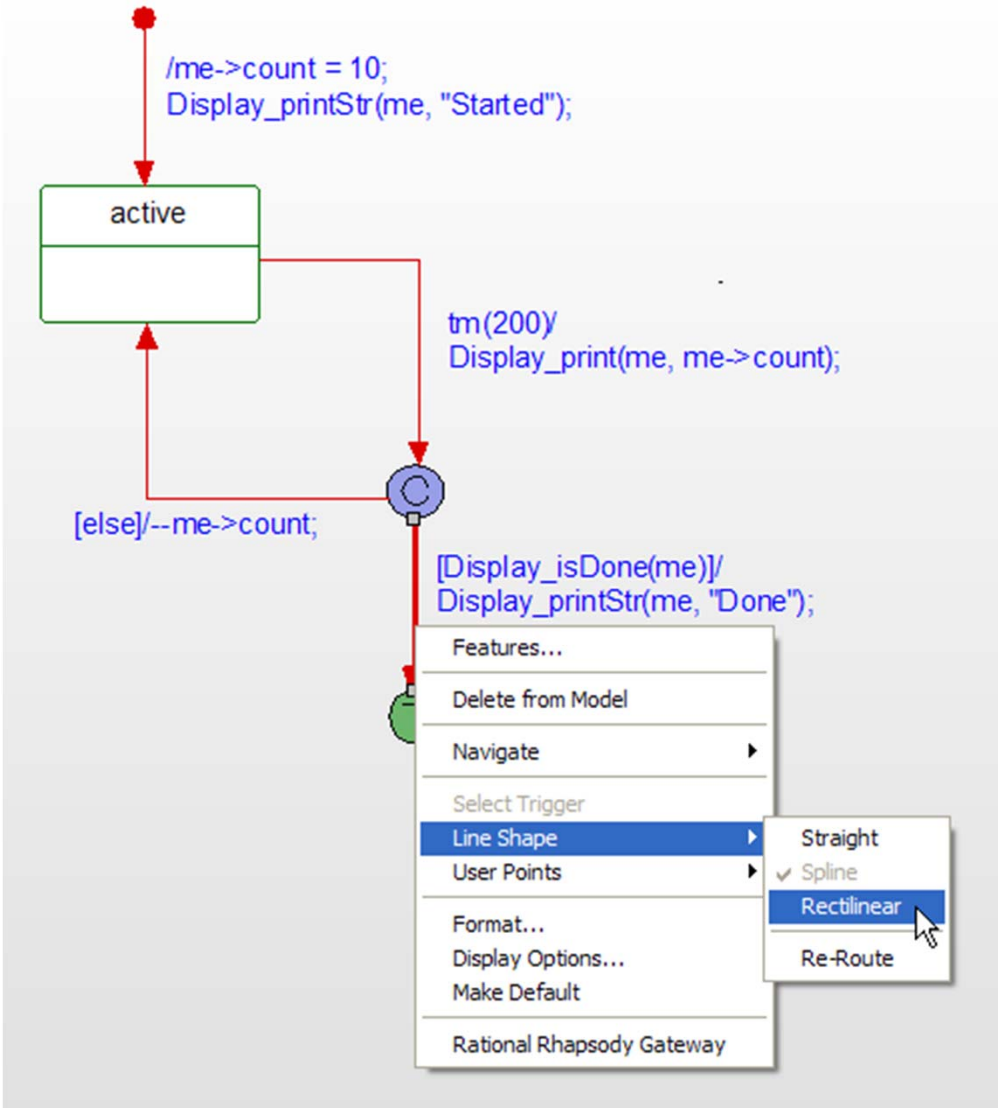




A reactive class is one that reacts to receiving events or timeouts.

Extended exercise

- Experiment with the line shape of transitions.

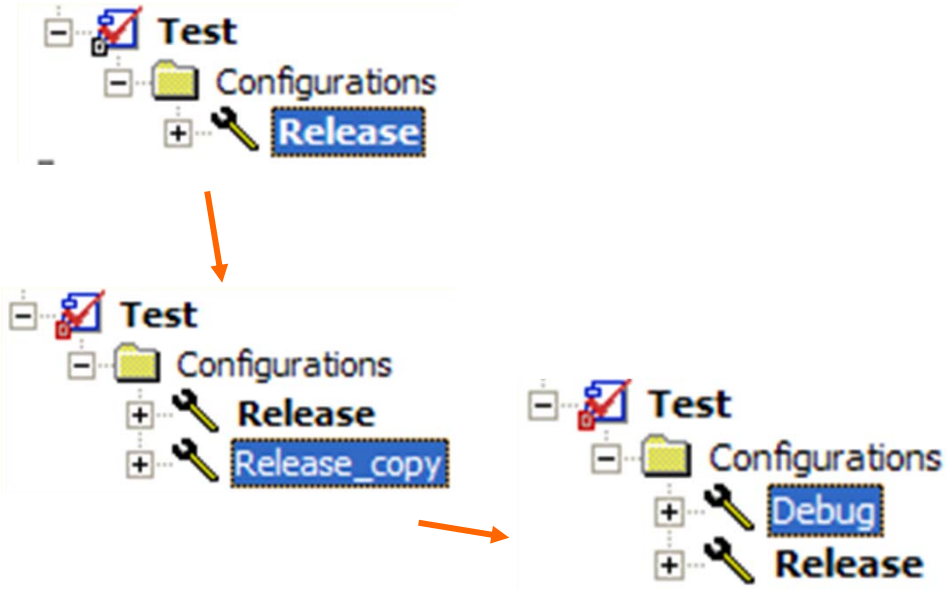


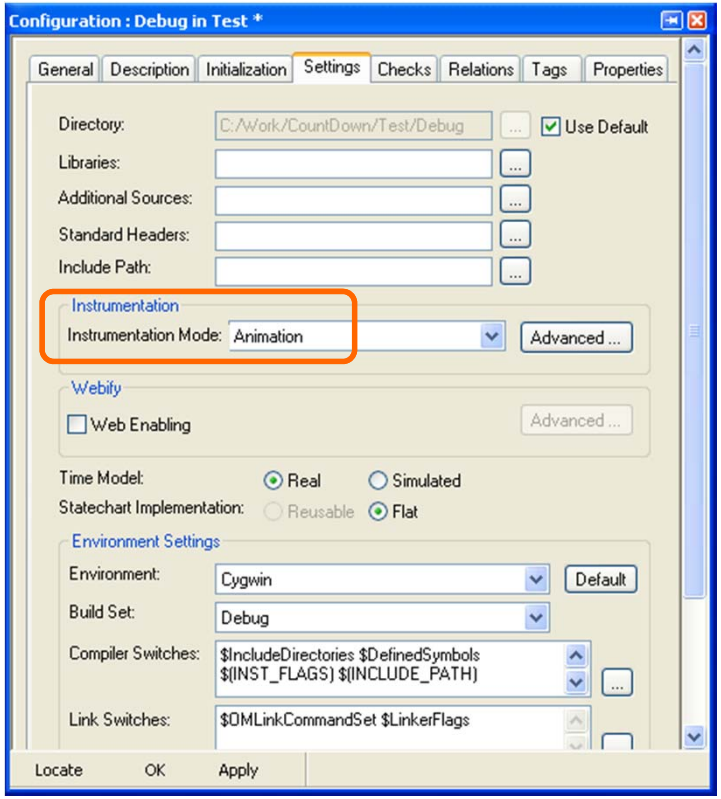
Design level debugging

- Until now, you have generated code and executed it, hoping that it works. However, as the model gets more and more complicated, you need to validate the model.
- From now on, you are going to validate the model by doing design level debugging known as *animation*.

Animation

- Create a new configuration by copying the *Release* configuration. Press Ctrl while dragging *Release* onto the Configurations folder.
- Rename the new configuration *Debug.*
- In the *Debug* Features window, select **Animation** from the **Instrumentation Mode** list.

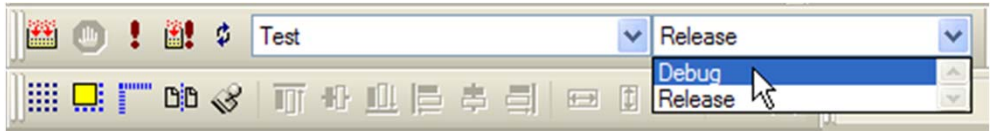




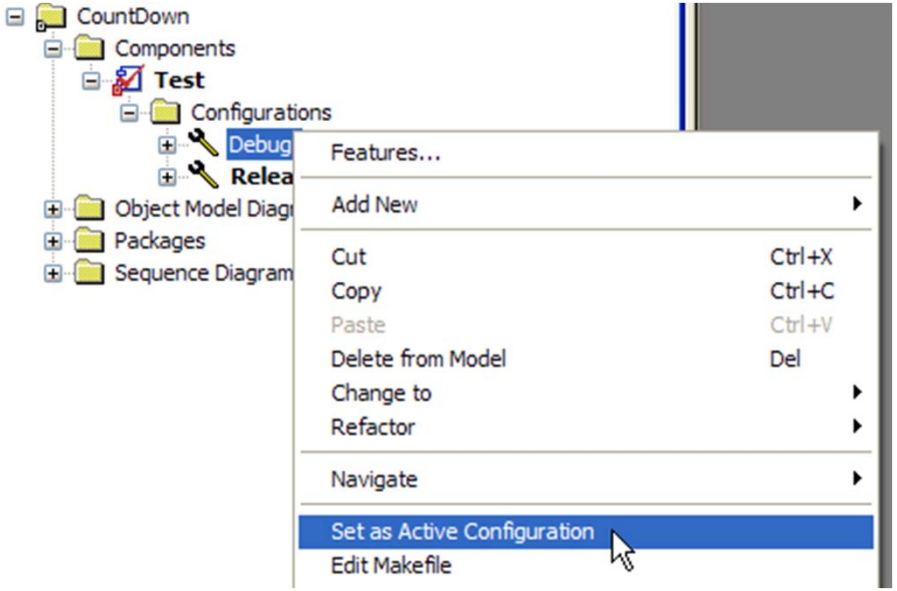
IBM

Multiple configurations

- Now that you have more than one configuration, you must select which one to use. There are two ways to do this:
  - Select the configuration using the **Debug/Release** pull-down list.



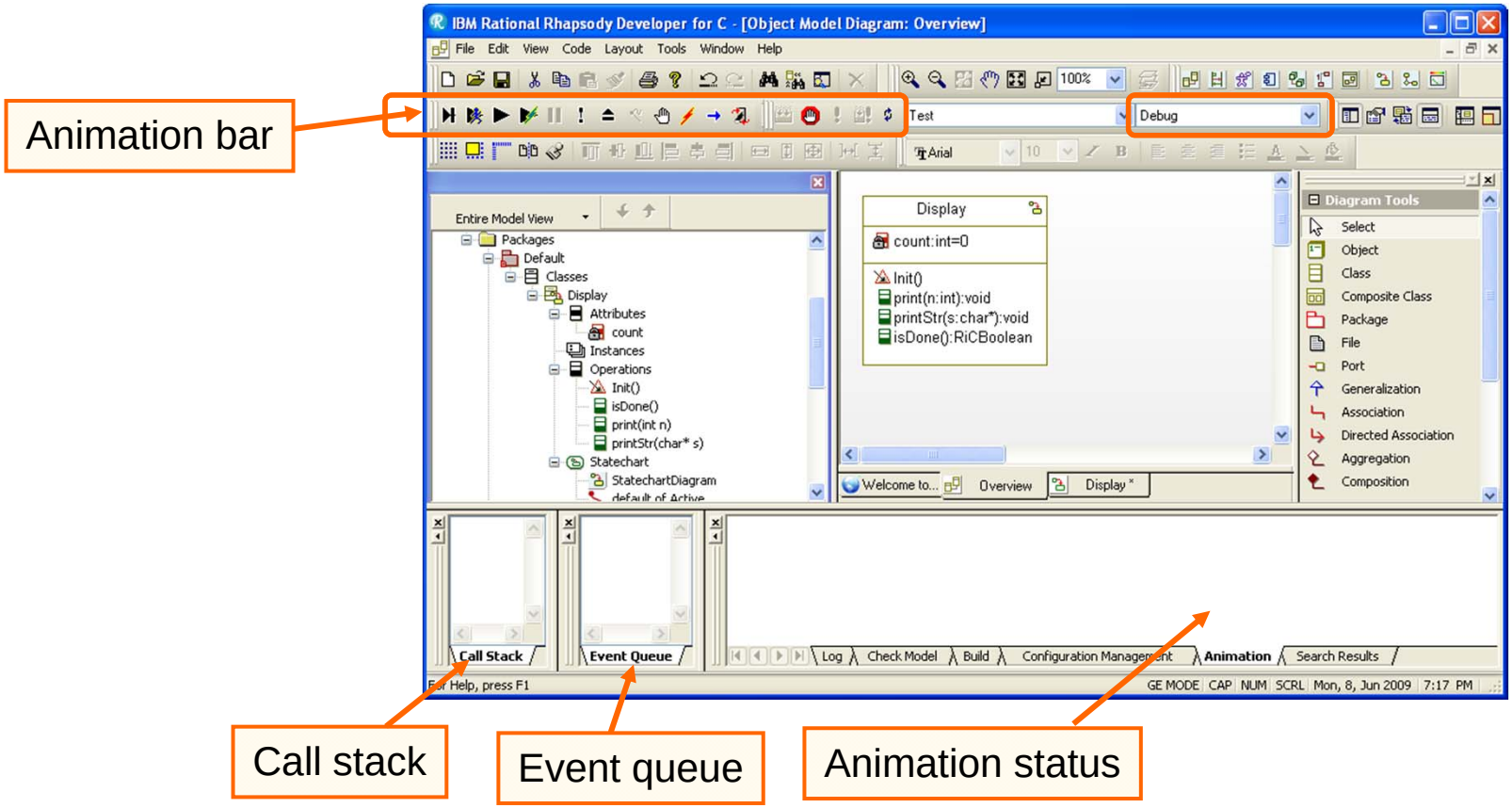
  - Or, right-click the configuration and select **Set as Active Configuration**.



IBM

Animating

- Make sure that the active configuration is **Debug** before clicking **Save**, and then **Generate/Make/Run**.



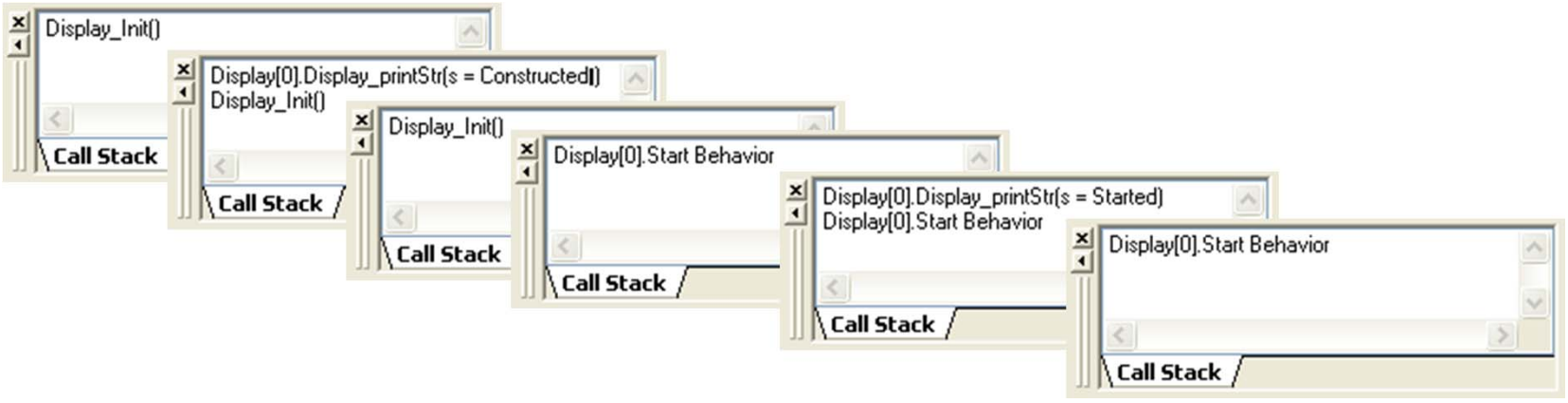
Animation toolbar

- Go Step
- Go
- Go Idle
- Go Event
- Animation Break
- Command Prompt
- Quit Animation
- Threads
- Breakpoints
- Event Generator
- Call Operations
- Watch – Display Continuous Update

When disabled, this indicates a singled-threaded application.

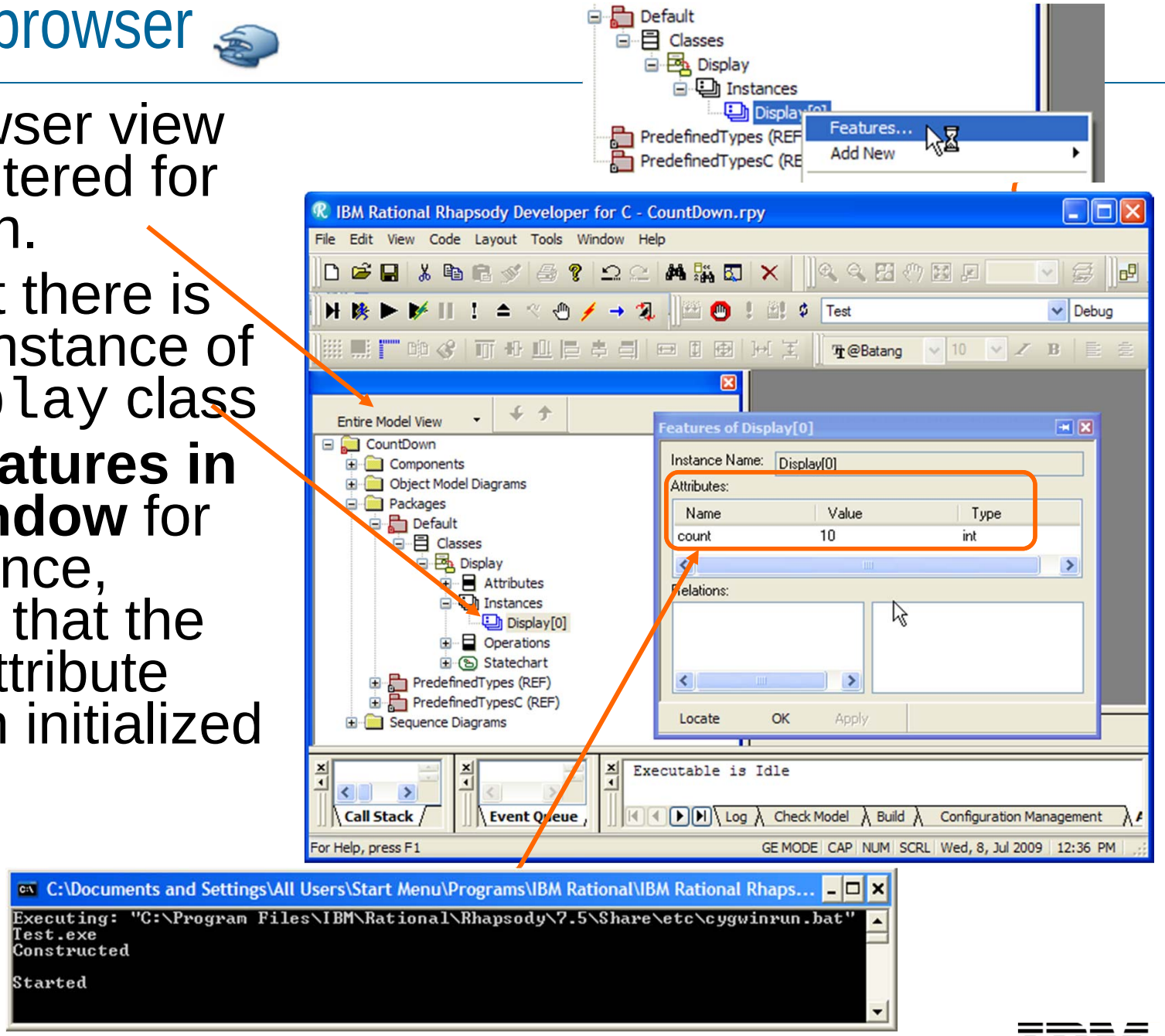
Starting the animation

- Go Step.
- The *Display* initializer appears in the call stack.
- Continue to Go Step until the *Executable is Idle* message appears in the Animation window.



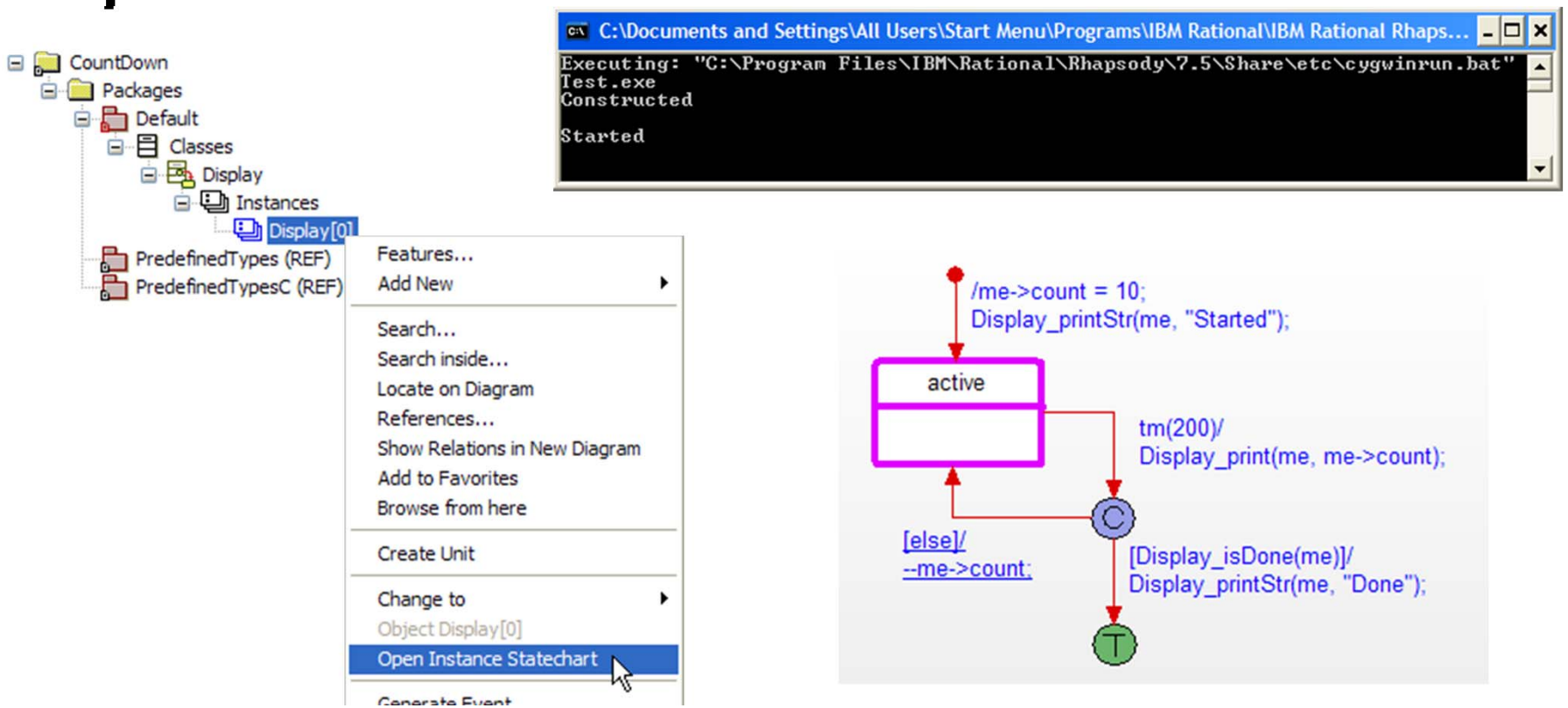
Animated browser

- The browser view can be filtered for animation.
- Note that there is now an instance of the `Display` class
- Open **Features in New Window** for this instance, and note that the `count` attribute has been initialized to be 10.



Animated statechart

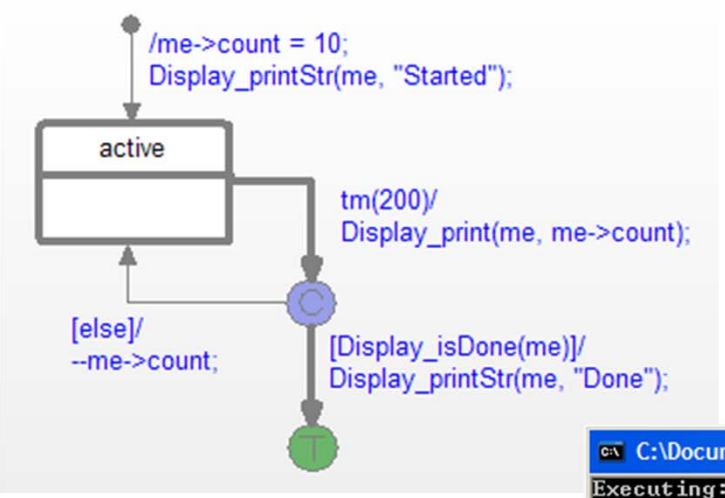
- Right-click the instance and select **Open Instance Statechart**.



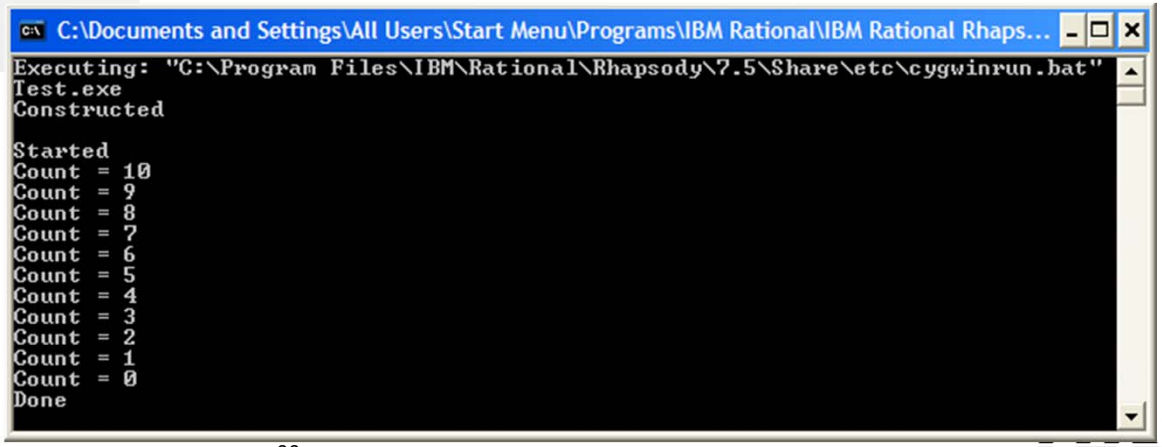
If you do not see a highlighted state, then you might be looking at the statechart of the class rather than the instance statechart. If code is changed and recompiled, close and reopen the instance statechart.

Go Idle / Go

- Repeatedly click Go Idle or Go and watch the animation until the instance is destroyed.
  - At the same time, observe the instance state machine, as well as the output window.

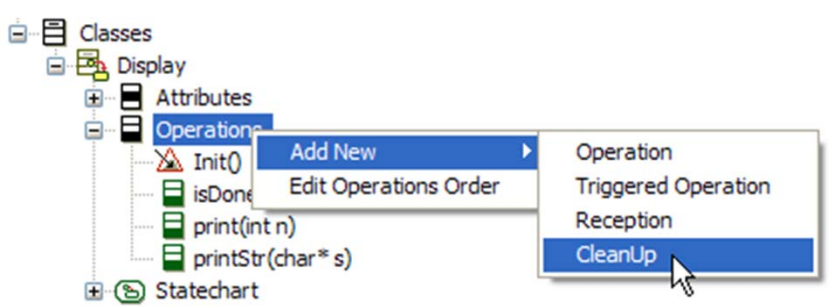


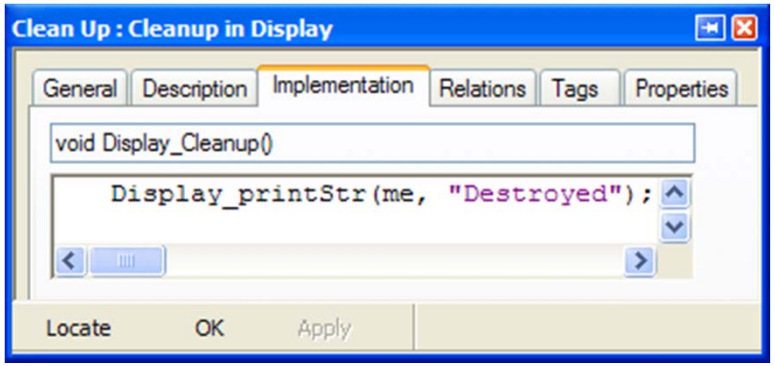
The value of the `count` attribute also changes and indicates that the transition taken in the statechart is highlighted.



Destructor

- Exit the animation
- Add a CleanUp operation to the *Display* class (right-click Opertations and select **Add New > CleanUp**).
  - This operation is executed when the instance is terminated (corresponds to a C++ destructor.).
- Implement CleanUp:
  - `Display_printStr(me, “Destroyed”);`
- **Save**, and then **Generate/Make/Run**.
  - Can you see the effect of this new operation?



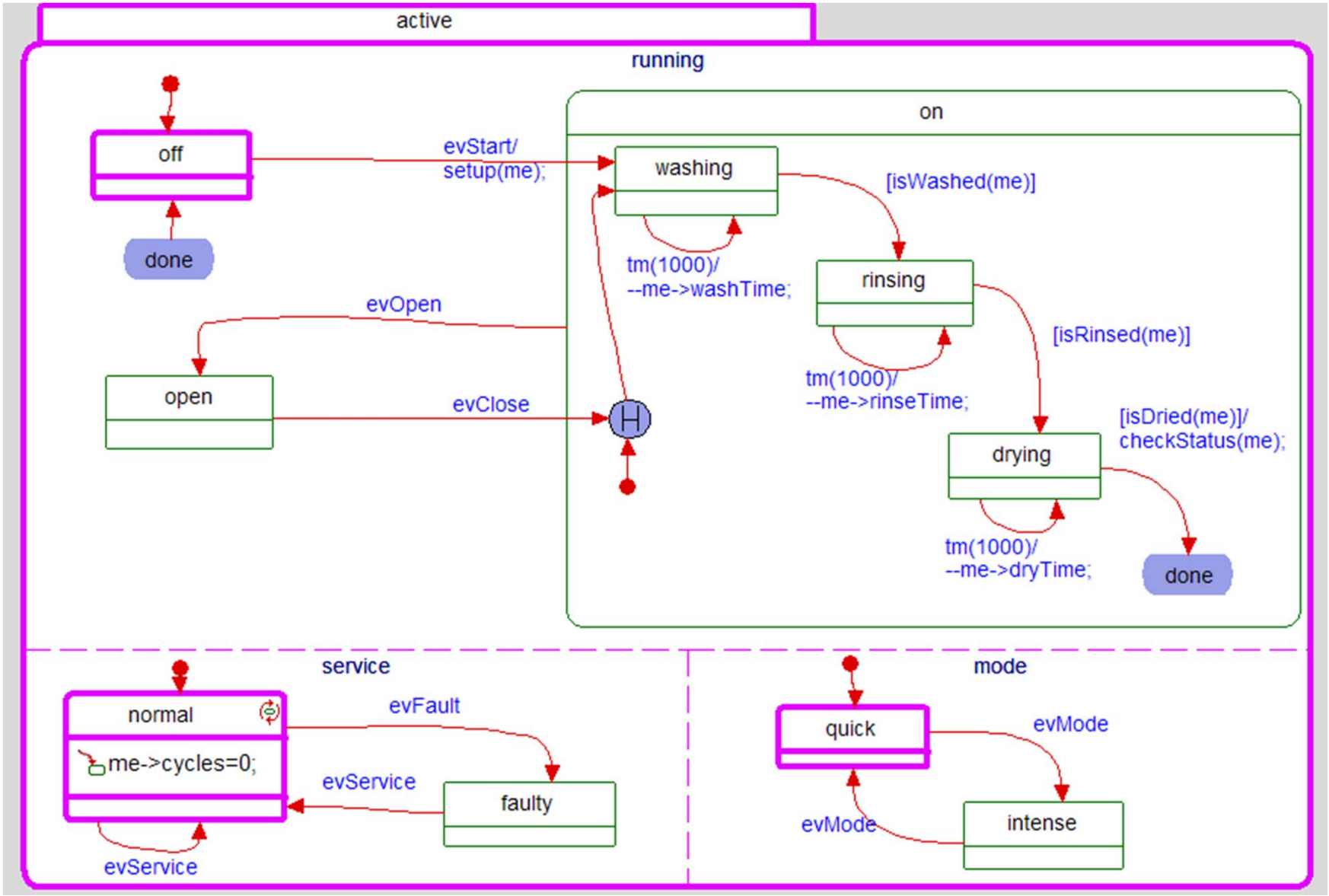


Make sure that you enter the code into the **Implementation** and not the **Description** field.

Where are we?

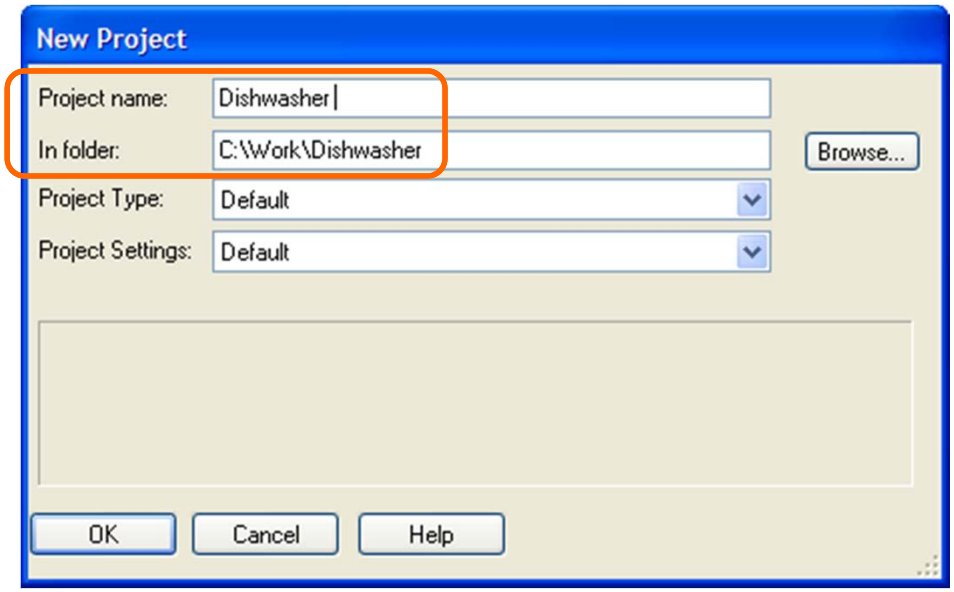
- Exercise 1 : Hello World
  - You start with the simplest example possible, just a single object that prints out Hello World.
- Exercise 2 : Count Down
  - Next, you create a simple counter using a simple statechart.
-  Exercise 3 : Dishwasher
  - Finally, you create a dishwasher and a more complex statechart.
- Summary

Exercise 3: dishwasher



Dishwasher

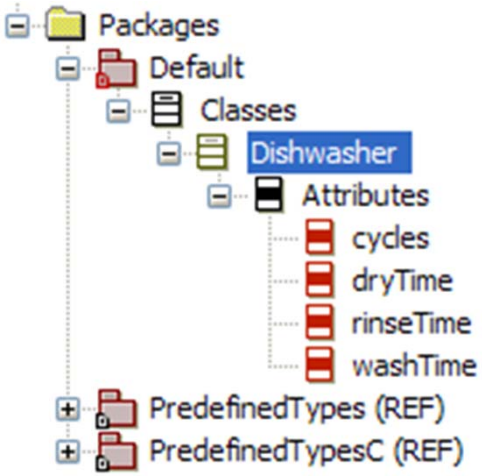
- Create a new project **Dishwasher**, making sure that it is created in its own folder.

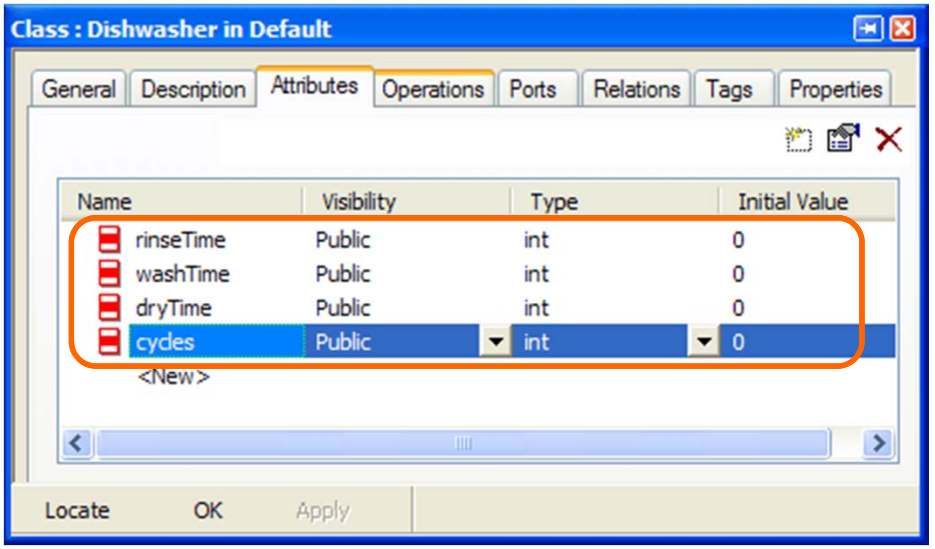


- Draw a single class *Dishwasher.*

Attributes

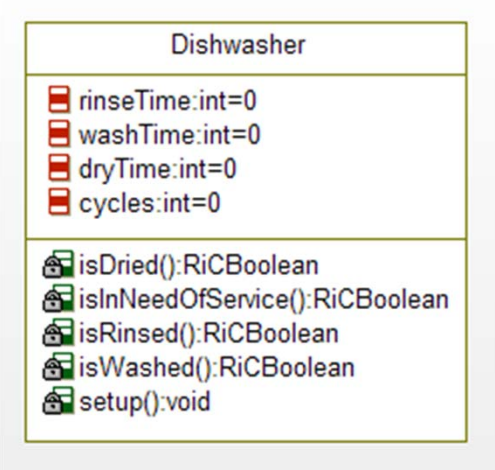
- Add the following attributes, all of which are of type `int` and with initial value of `0`:

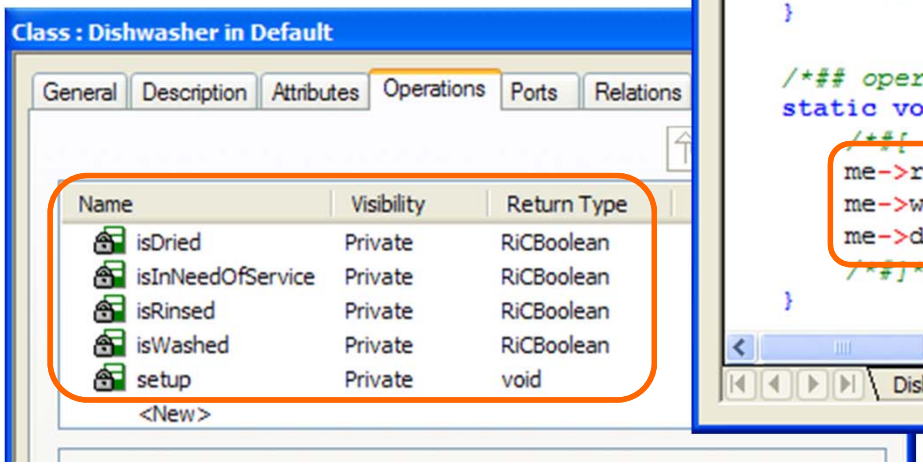


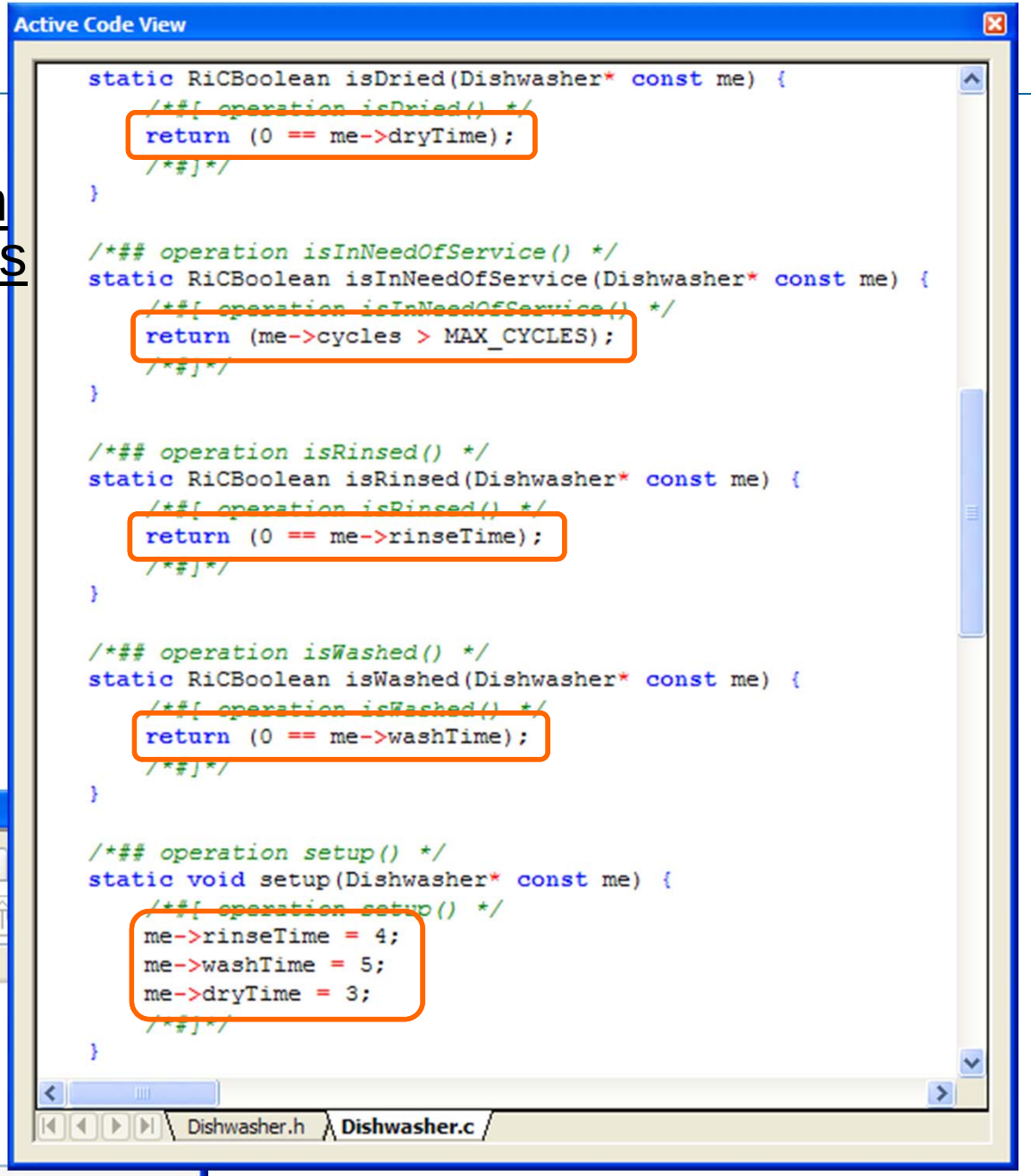


Operations

- Add the following private operations, with appropriate return types and implementations:

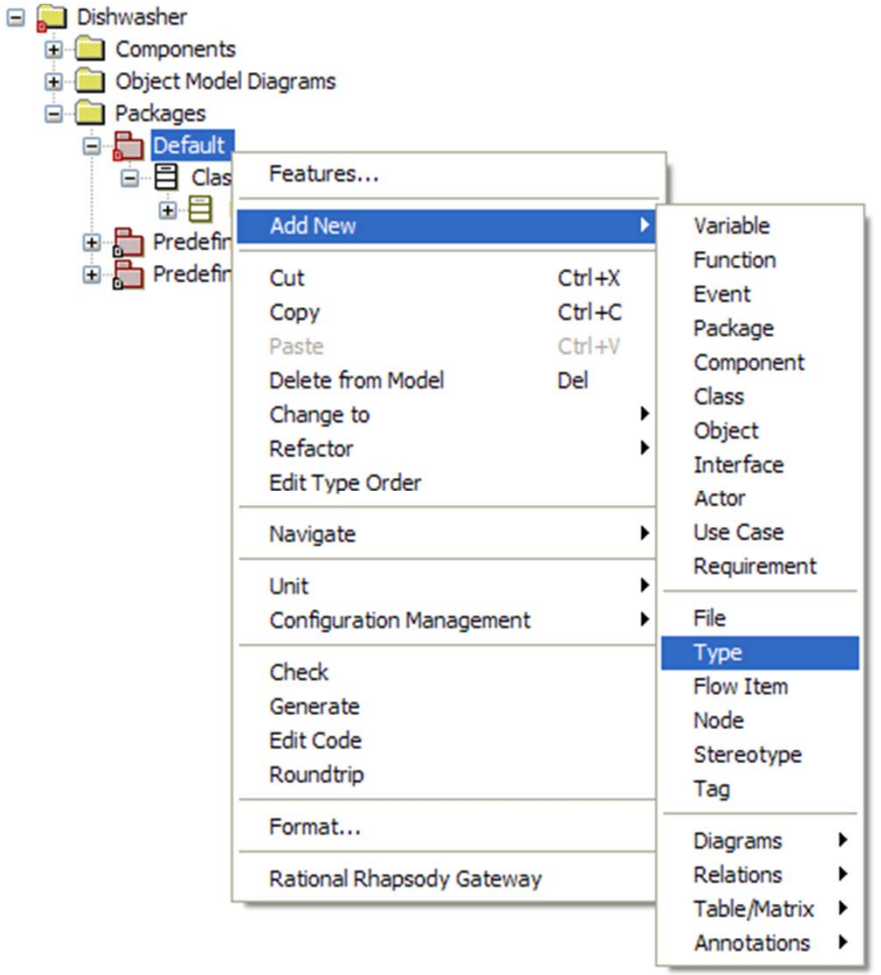






Types (part 1)

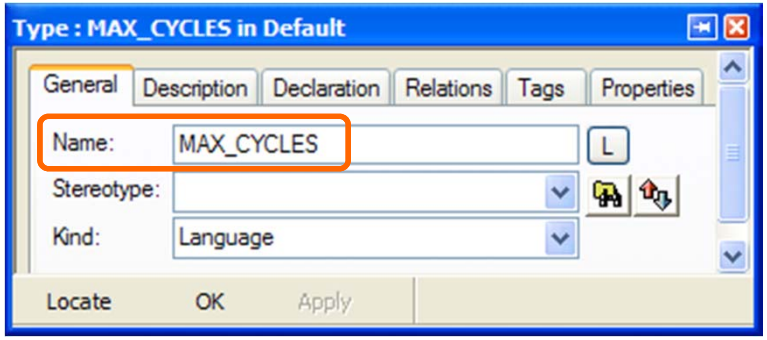
- In the implementation for the operation `isInNeedOfService()`, you used a constant called `MAX_CYCLES`.
- Select the *Default* package, right-click, and select **Add New > Type**.

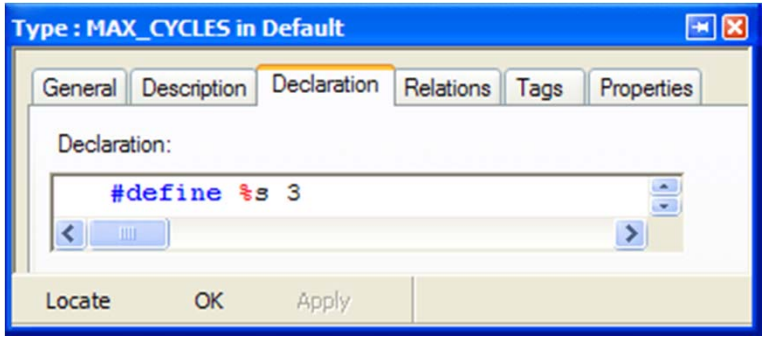


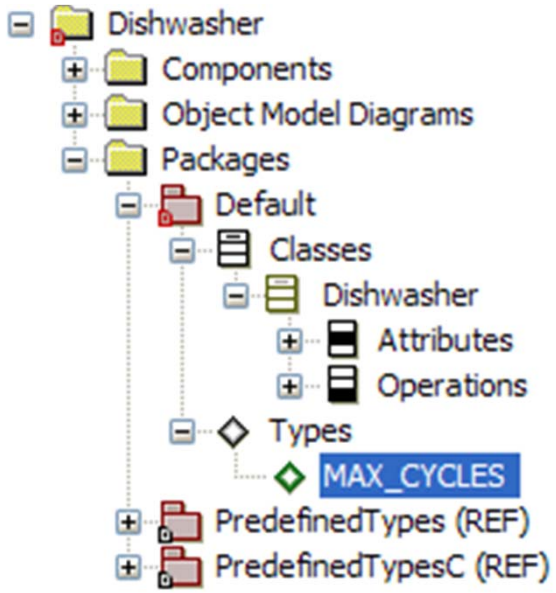
Types (part 2)

- Enter MAX_CYCLES as the **Name** and make this **Declaration**:
  - `#define %s 3`

%s is a shortcut for the name of the type.



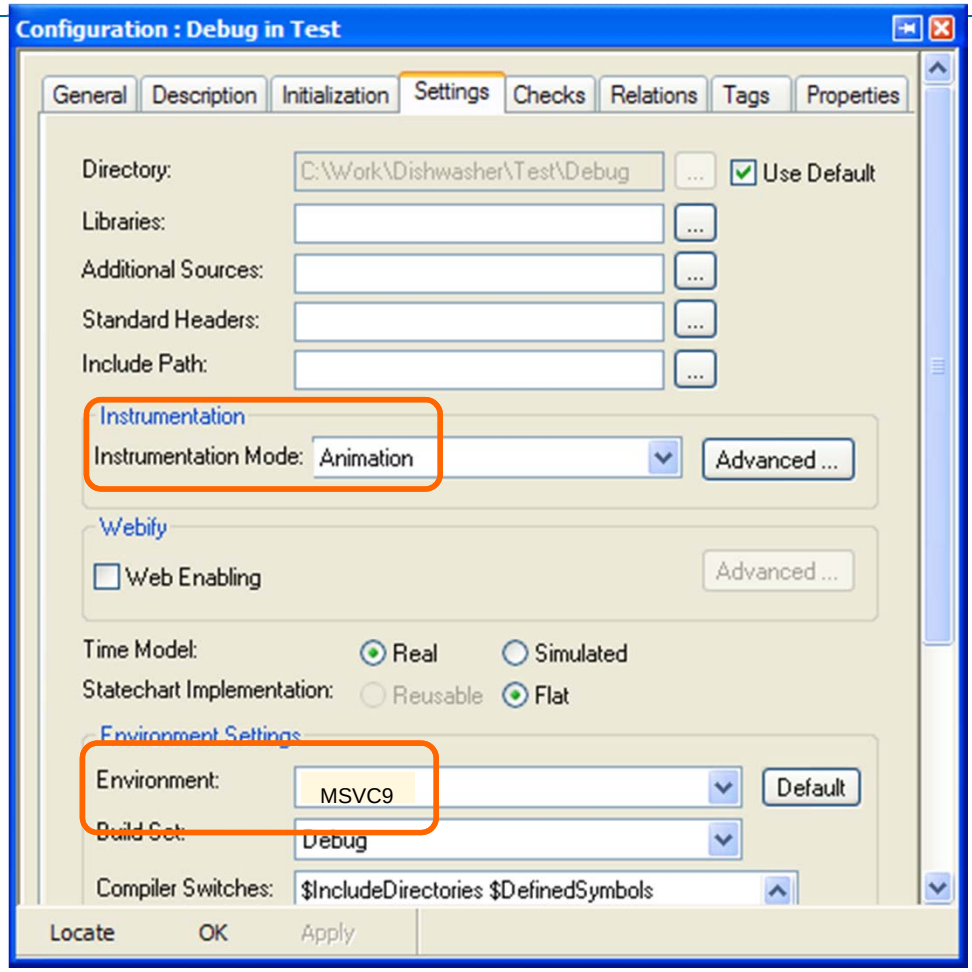


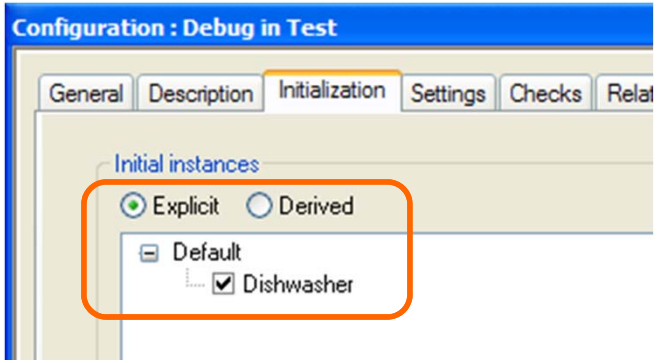


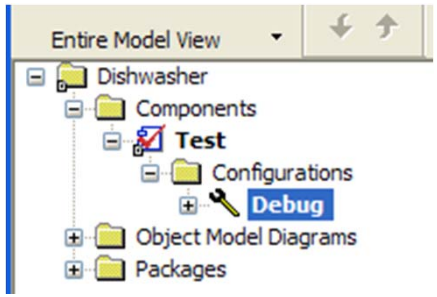
IBM

Creating a component

- As in the previous exercises
  - Rename the *DefaultComponent* to *Test* and the *DefaultConfig* to *Debug.*
  - Select **Animation** for **Instrumentation Mode**
  - Create an initial instance of *Dishwasher.*
  - Select **MSVC9** for the Environment.

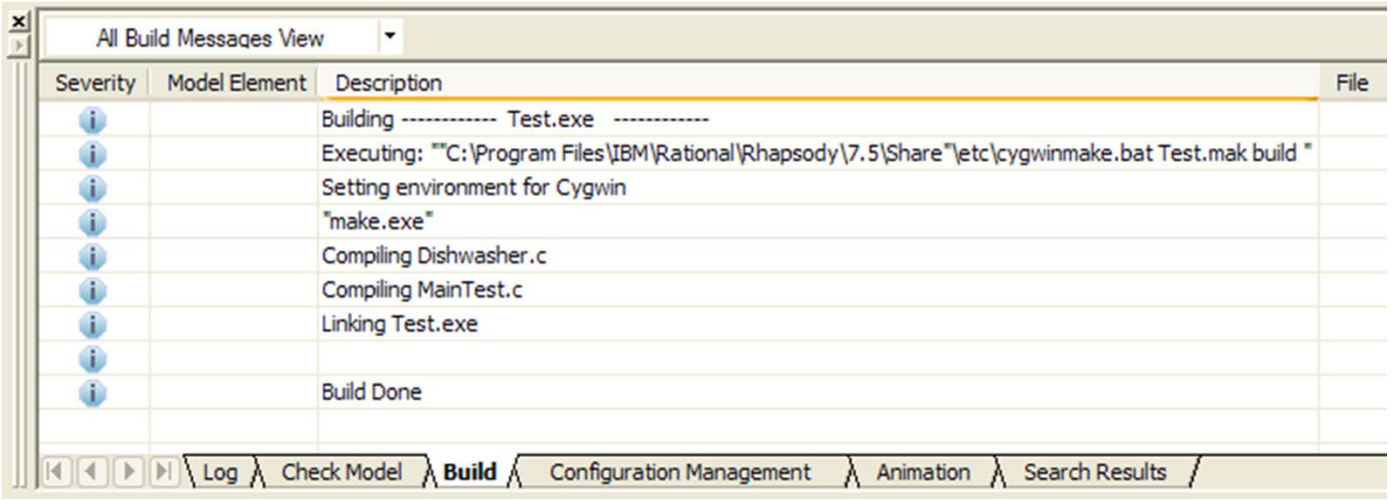






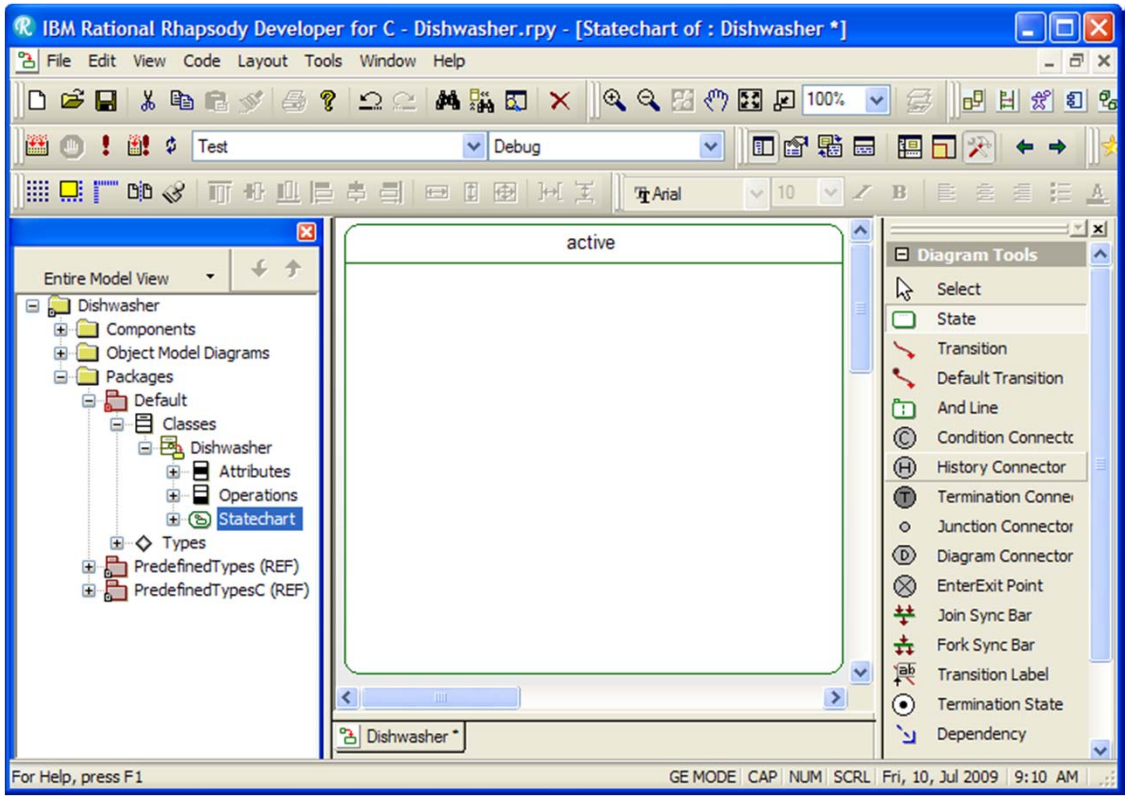
Save / Generate / Make

- Before adding a statechart, make sure that you have not made any errors by doing a build.
  - Make sure you deal with all errors before you proceed
- **Save**, and then **Generate/Make/Run**.



Creating a statechart

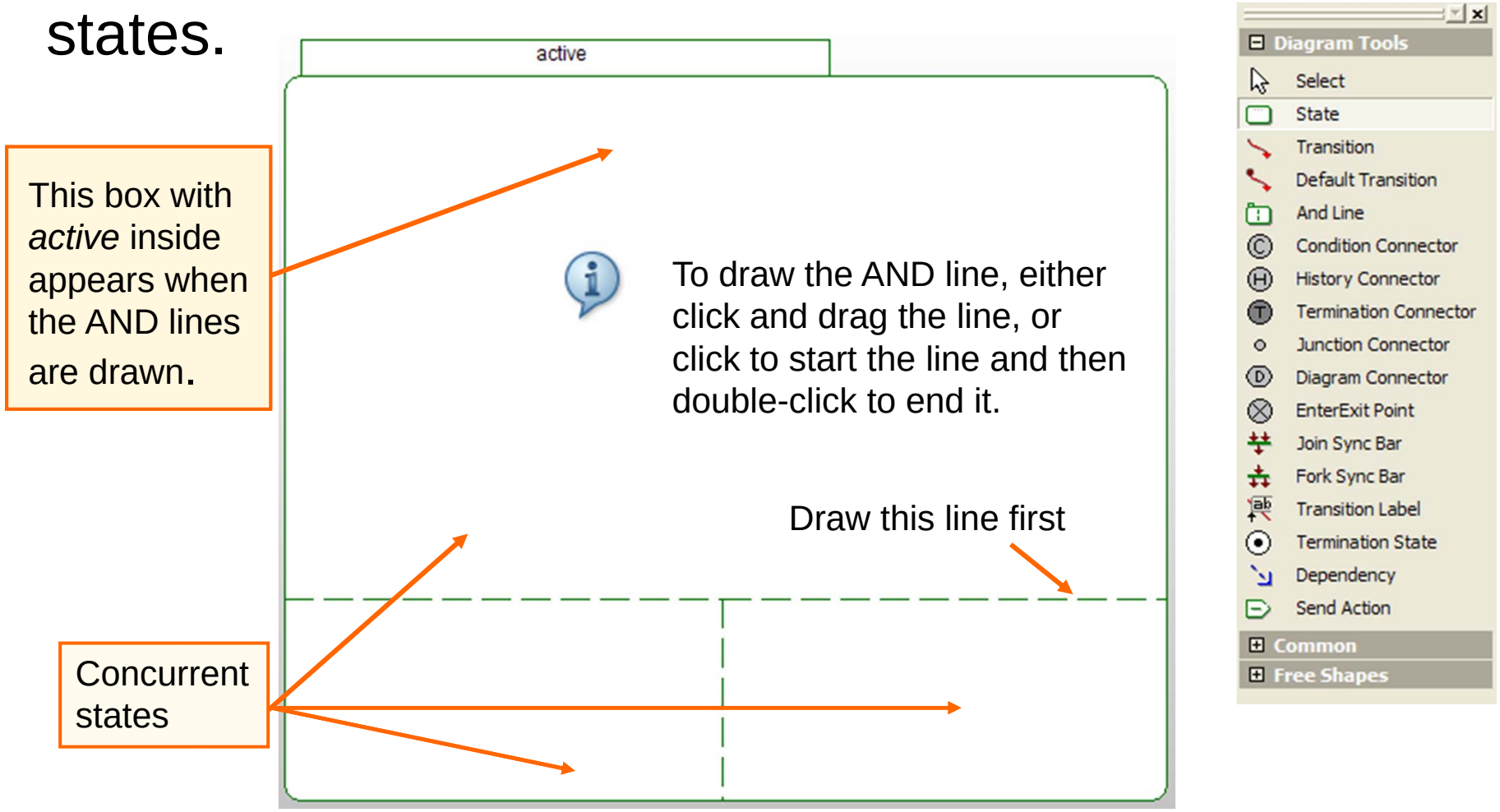
- Add a Statechart to the `Dishwasher` class.
- Draw a single state as large as possible called *active.*



Because you are drawing a complex diagram, it is highly recommended you maximize the Statechart window and close all other windows.

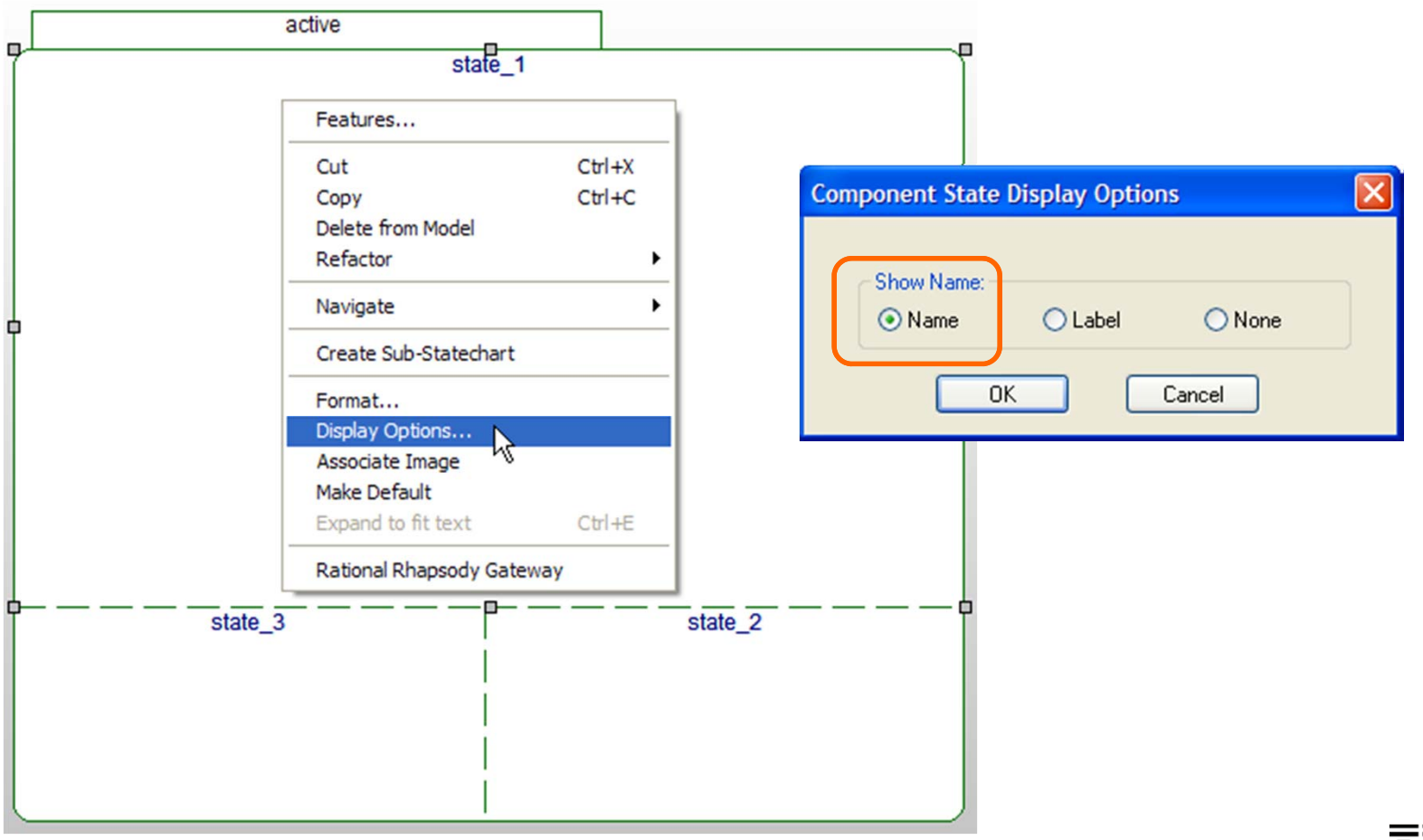
Creating concurrent states

- Use the and-line icon to create concurrent states.



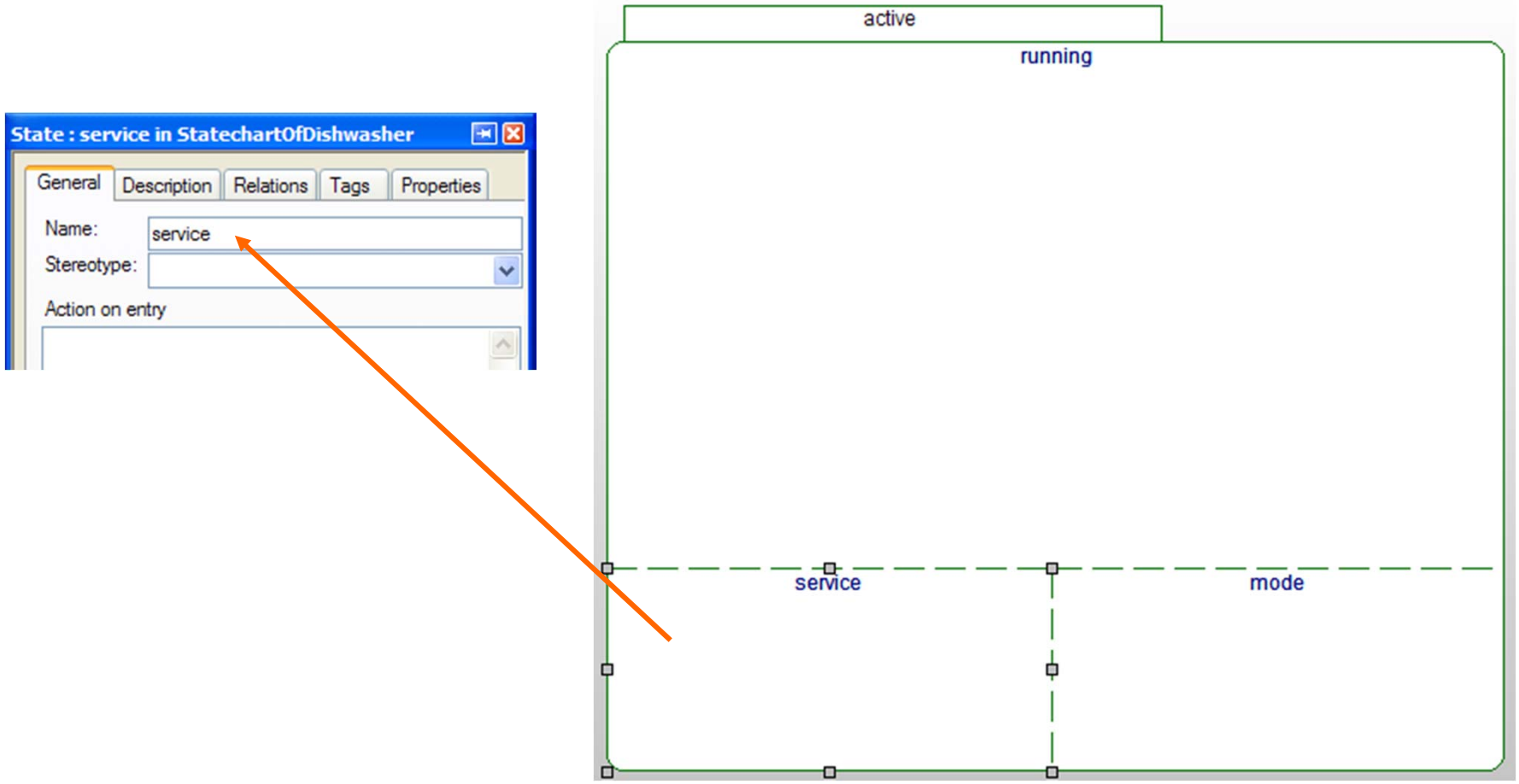
Displaying state names

- Each of the three AND state names can be displayed by selecting each state and selecting **Display Options** to show the **Name**.



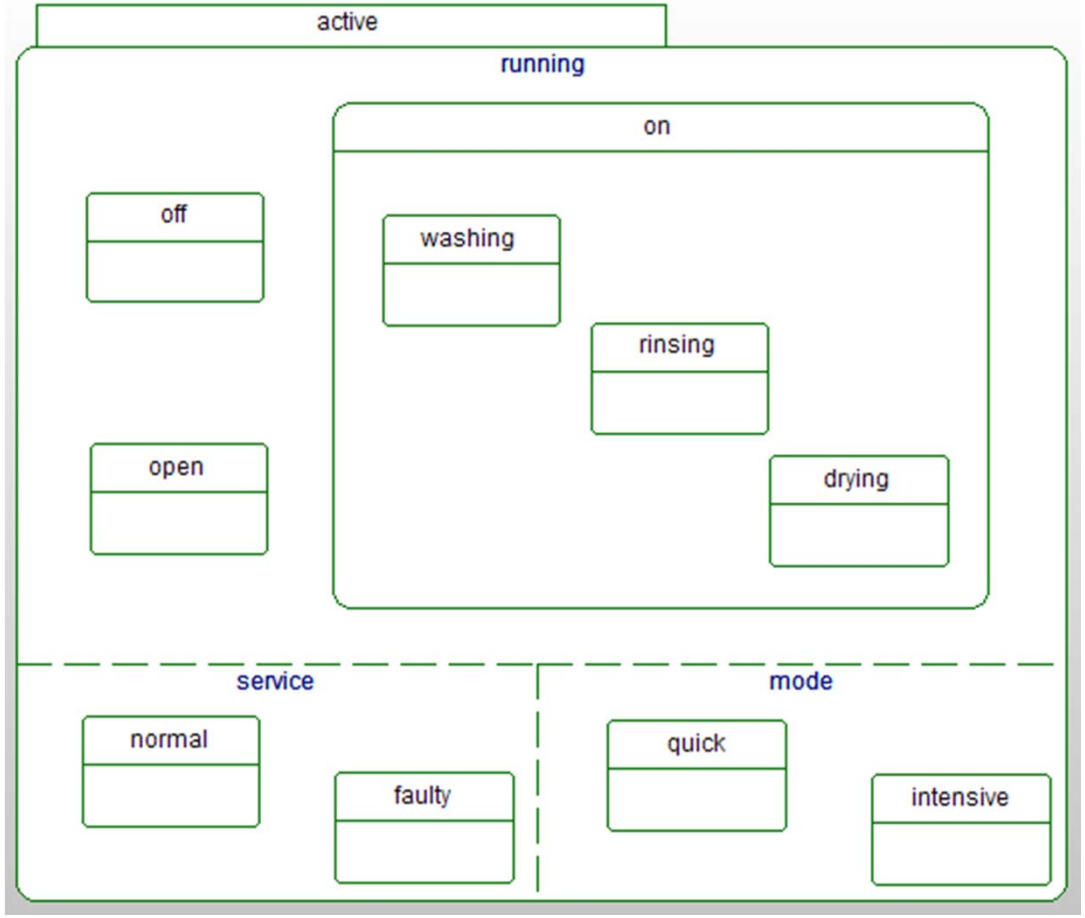
Naming the concurrent states

- With the concurrent states names displayed, they can now be changed using the features to *running*, *service,* and *mode*.



Adding nested states

- Select the **Stamp** mode, then add the following states:

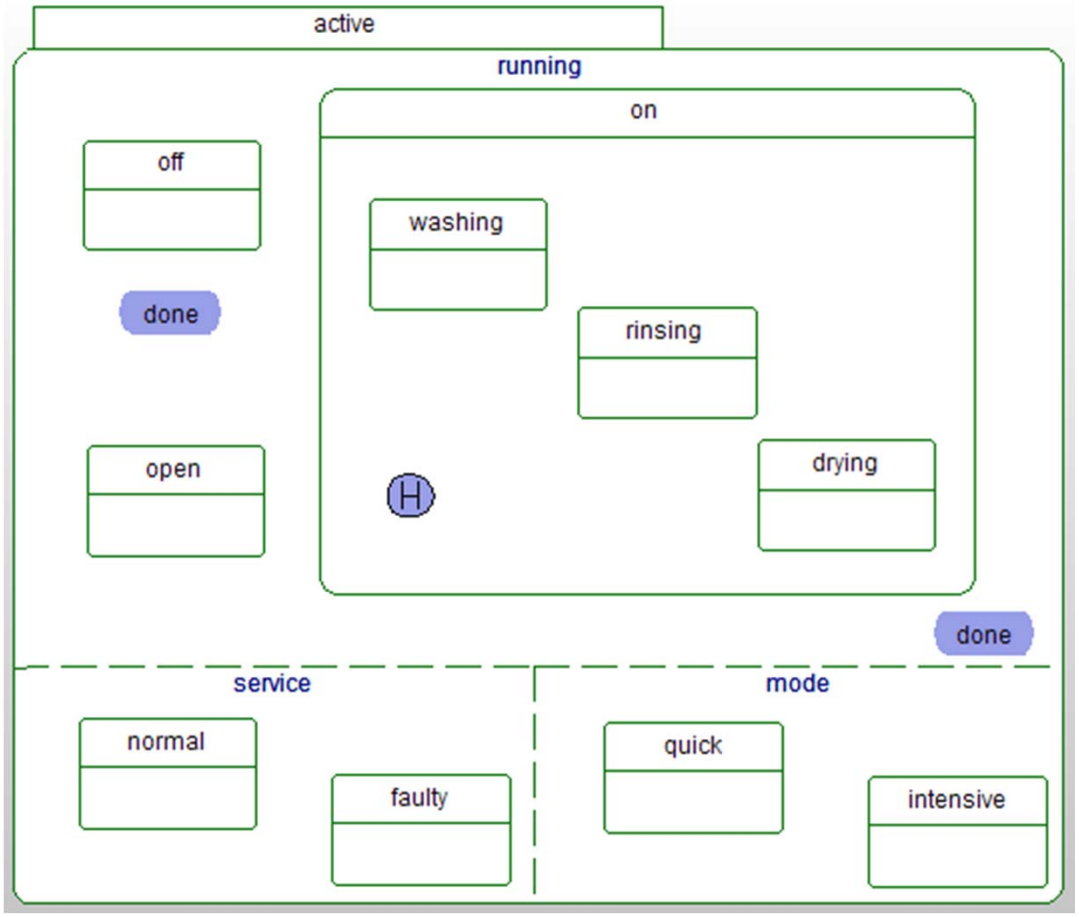


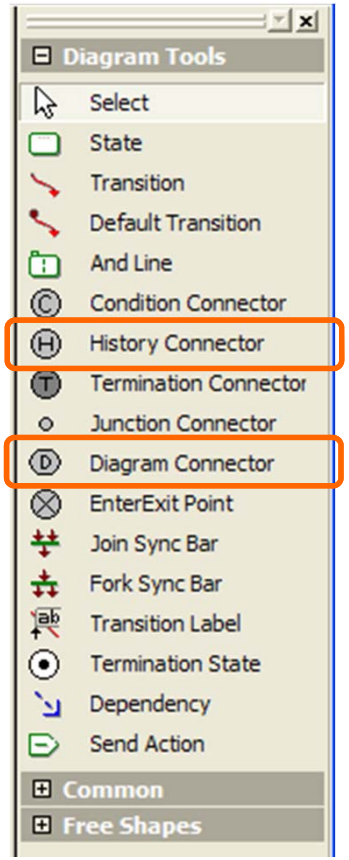
To change the size of an outer state, without changing the inner states, press the **Alt** when changing the size.

IBM

Adding History and Diagram connectors

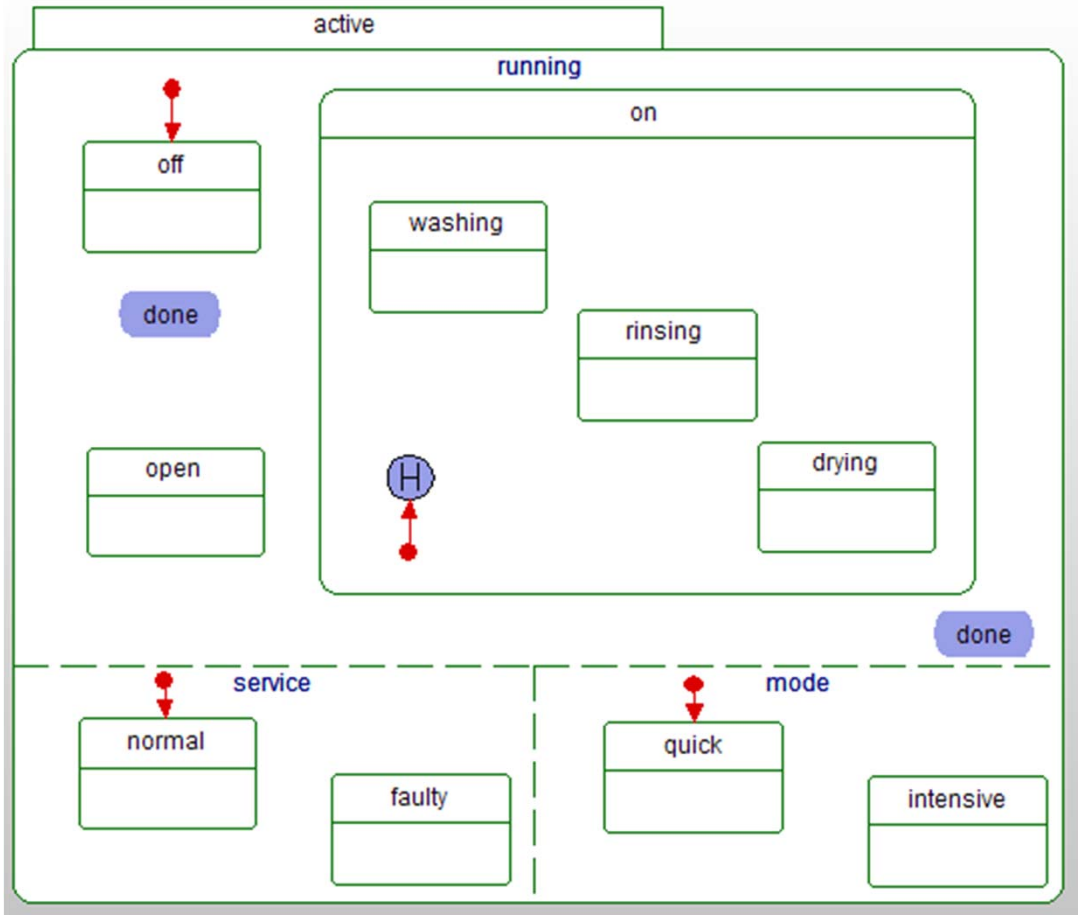
- Add a **History Connector** to the on state.
- Add two **Diagram Connector**s, both named *done.*

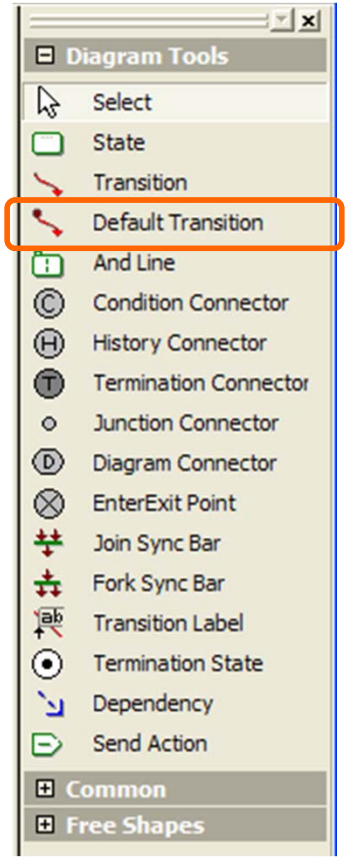




Adding default transitions

- Add the 4 **Default Transitions** below:



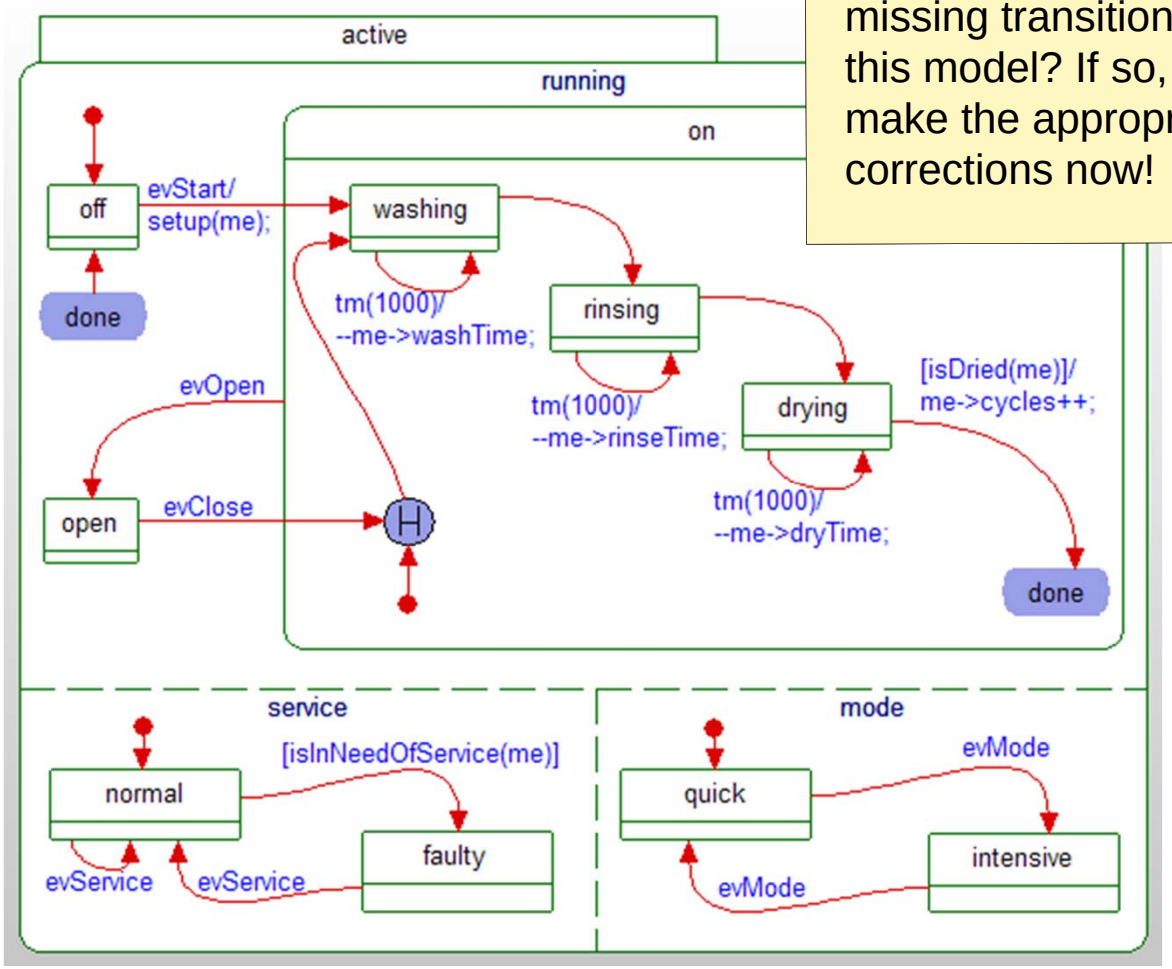


Adding the transitions

- Add transitions and actions:

Can you identify any missing transitions in this model? If so, make the appropriate corrections now!

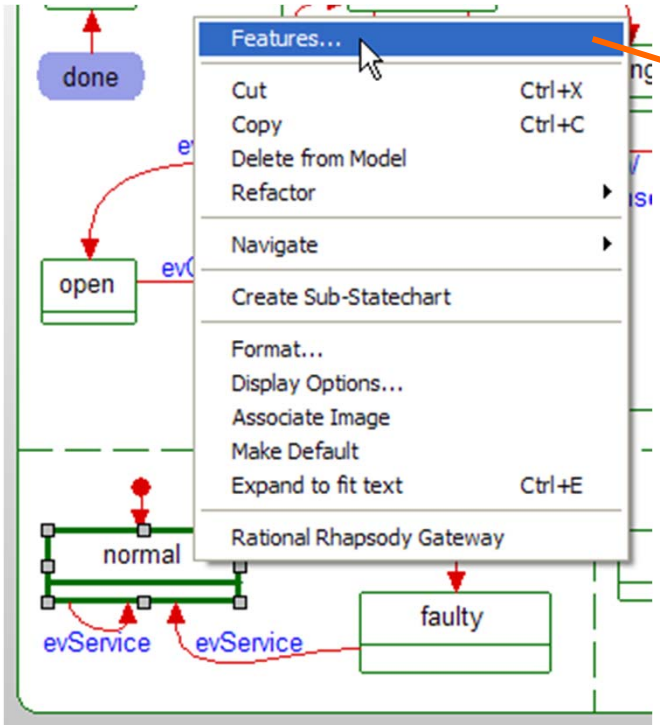
When you need to type a name of an existing operation or attribute, press **Ctrl+Space** to facilitate entering these names.

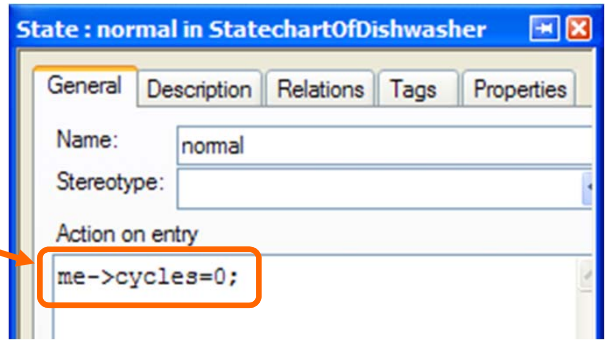


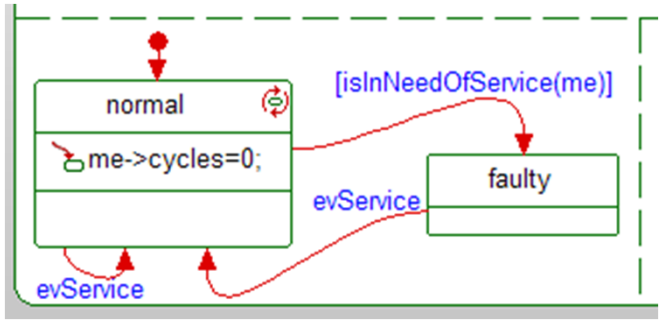
IBM

Action on entry

- In the `normal` state, add an `Action on entry` to set `me->cycles=0;`.



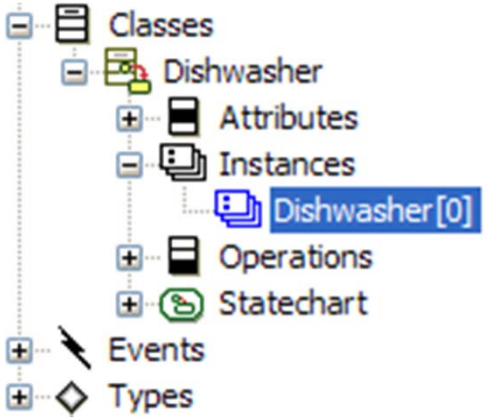


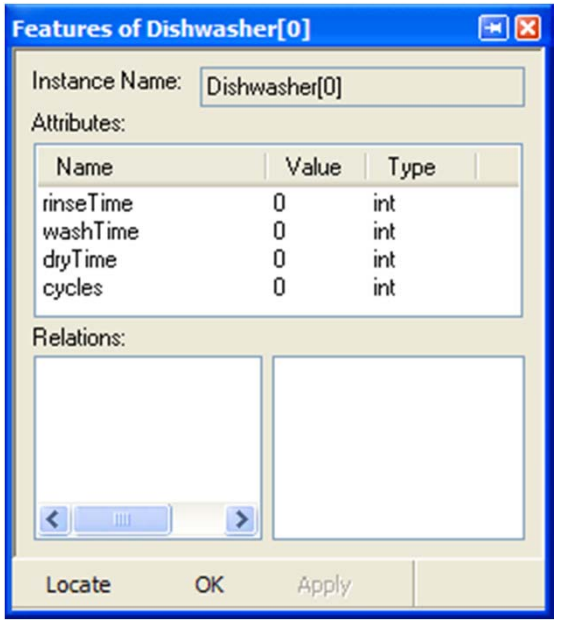


Once an action has been set, the symbol is shown (two red arrows circling an oval icon).

Save / Generate / Make / Run

- **Save**, and then **Generate/Make/Run**.
- Click **Go Idle** to create a *Dishwasher* instance.
- Select **Open Instance Statechart** for the Dishwasher instance created.

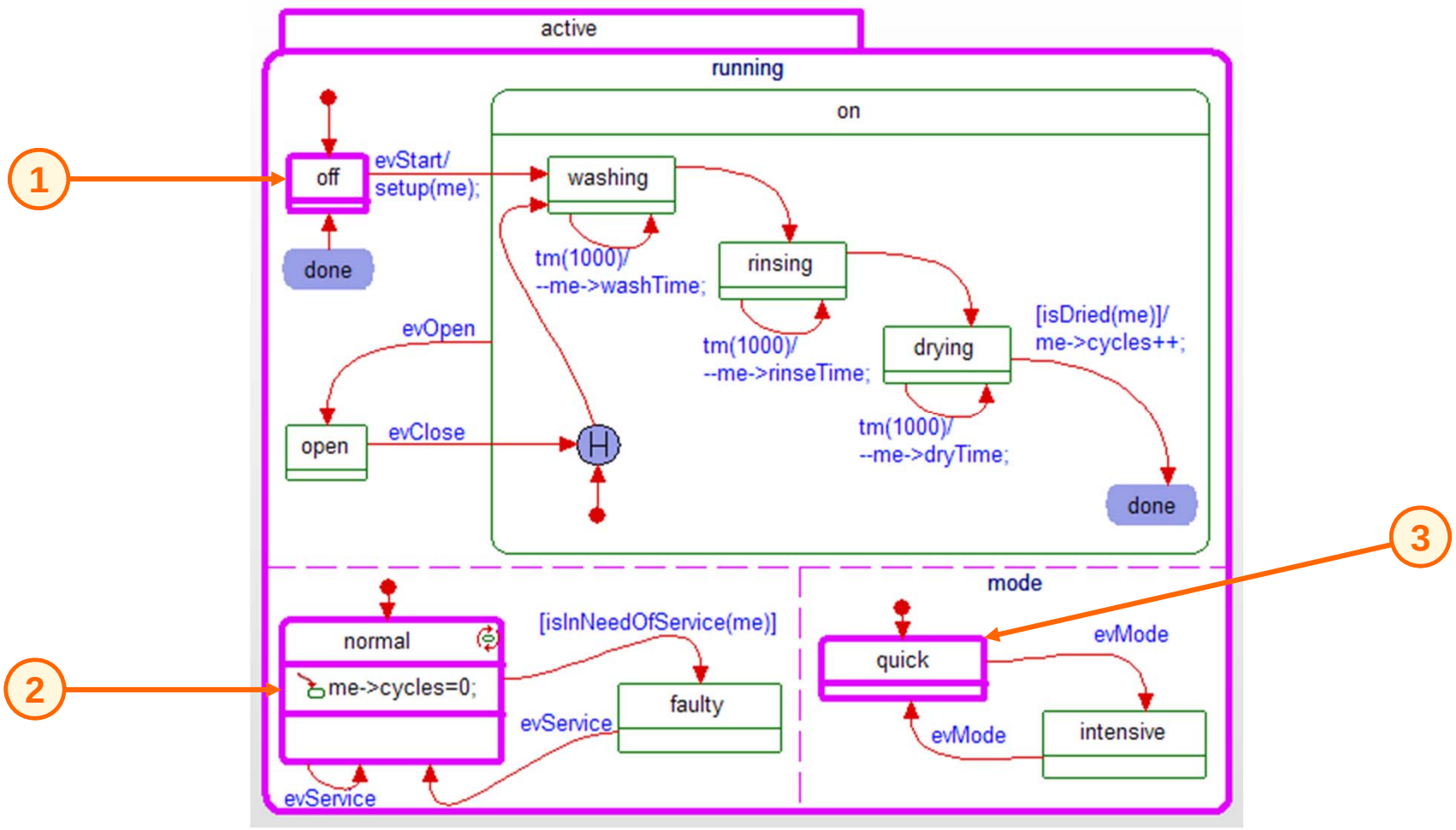




If there is no instance created, then it is possible that an initial instance of *Dishwasher* was not selected in the configuration.

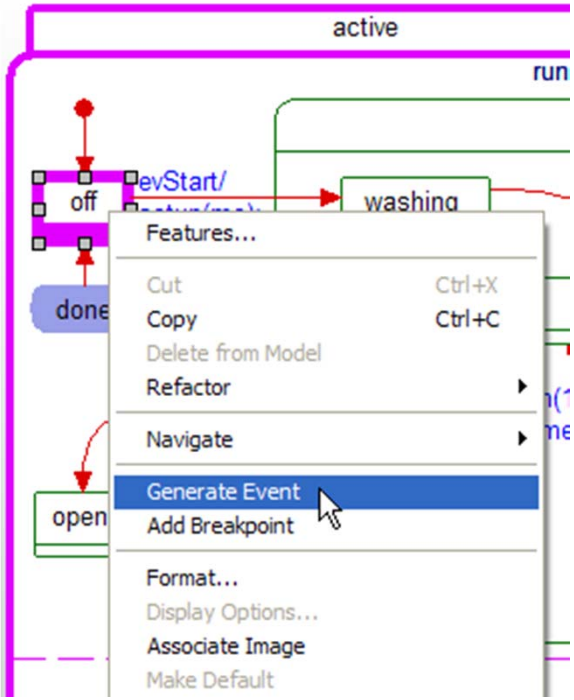
Animated statechart

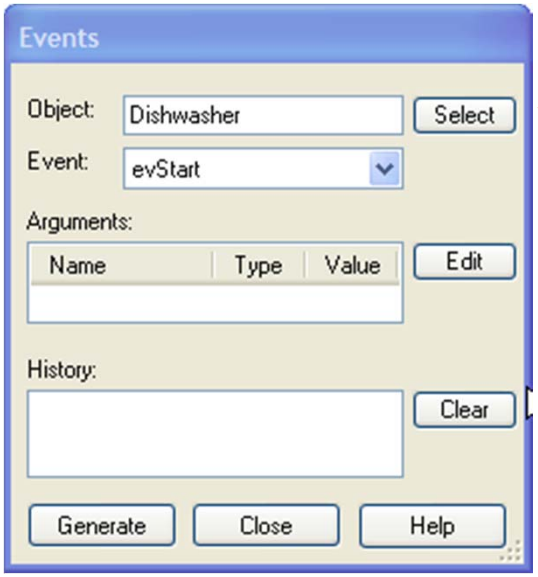
- Check that there are three concurrent animated states:

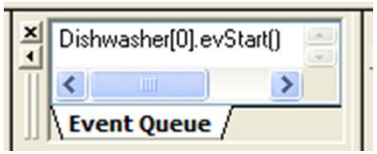


Injecting events

- The *Dishwasher* is in an idle state waiting for some events.
- Generate the event `evStart` by right-clicking anywhere inside the active state and selecting **Generate Event**.
- The event `evStart` appears in the event queue.



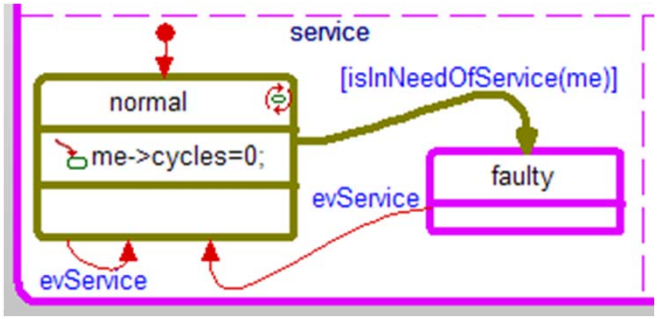




Events can also be generated via the Command prompt (slider) or via the Event Generator (lightning bolt).

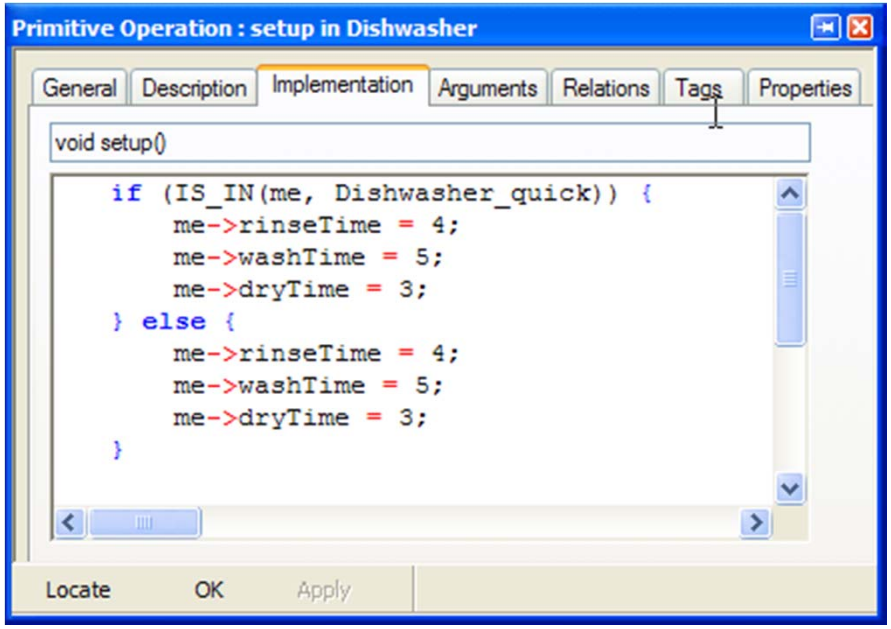
Design level debugging

- Click **Go** and watch the animation.
- Does your *Dishwasher* work as expected?
- What happens if you open the door when it is on, does it remember what state it was in?
- Why does the *Dishwasher* become faulty after four cycles?
- Can you get the *Dishwasher* back to the normal state?



Intense / quick

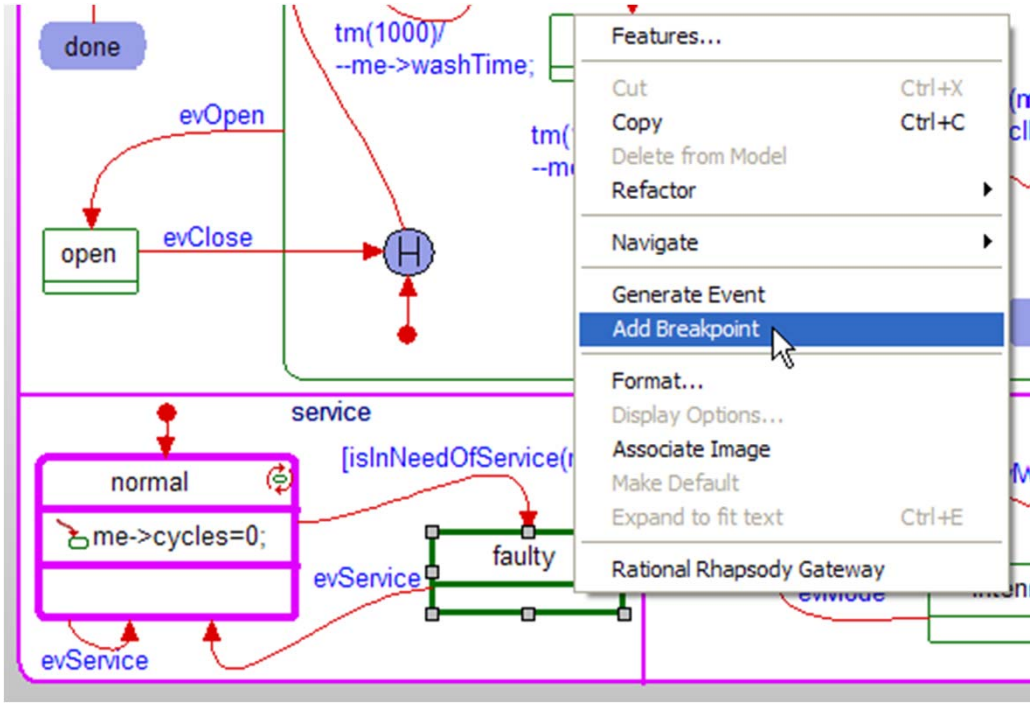
- Modify the `setup` operation so that the timings are different in the quick and intense modes.
- **Save**, and then **Generate/Make/Run**.
- It should now be quicker to get the *Dishwasher* into the `faulty` state.



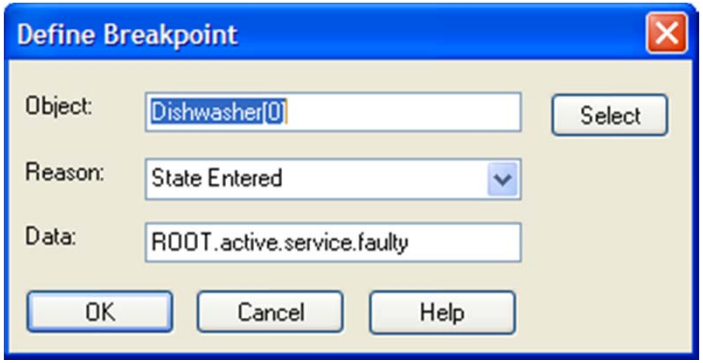
`IS_IN` is a macro that tests to see if the object is in a particular state. (You could use the IS_IN macro also in a guard)

Additional Info: Breakpoints

- Setting breakpoints can be done in a similar way to injecting events by right-clicking a state in the animated statechart.



Breakpoints can be added/removed via the breakpoint icon on the animation toolbar.

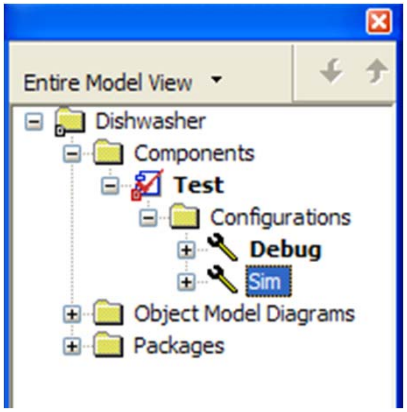


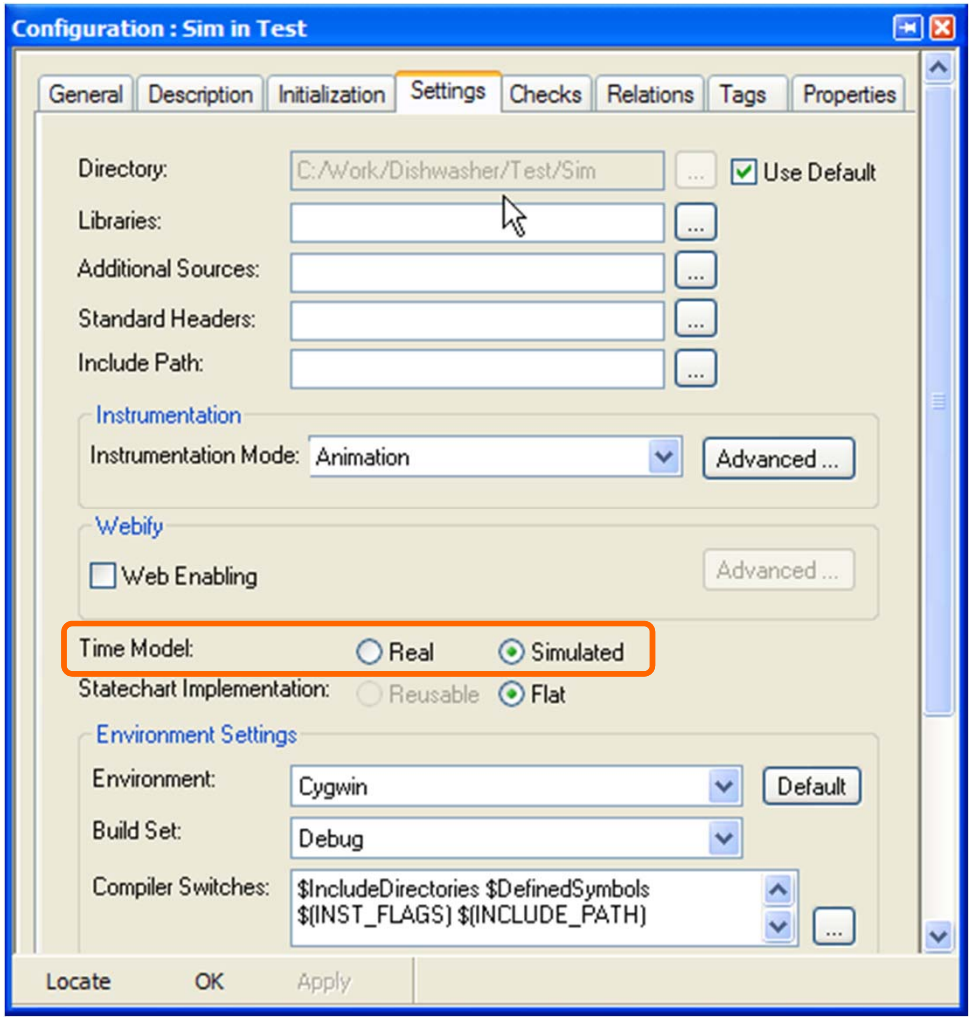
Additional Info: Using the simulated time model

- At the moment, you are using the System tick for all the timeouts, and so a timeout of 1000ms takes 1000ms, which means that all simulations can be long.
- There is an alternative time model that can be used which is referred to as the *simulated* time model. When this time model is used, all of the timeouts are executed in the appropriate order; but rather than waiting, the shortest timeout immediately times out. This means that models such as this one, can be tested much quicker.

Additional Info: Using the simulated time model

- Make a copy of the *Debug* configuration, rename it to *Sim* and set the **Time Model** to **Simulated**.



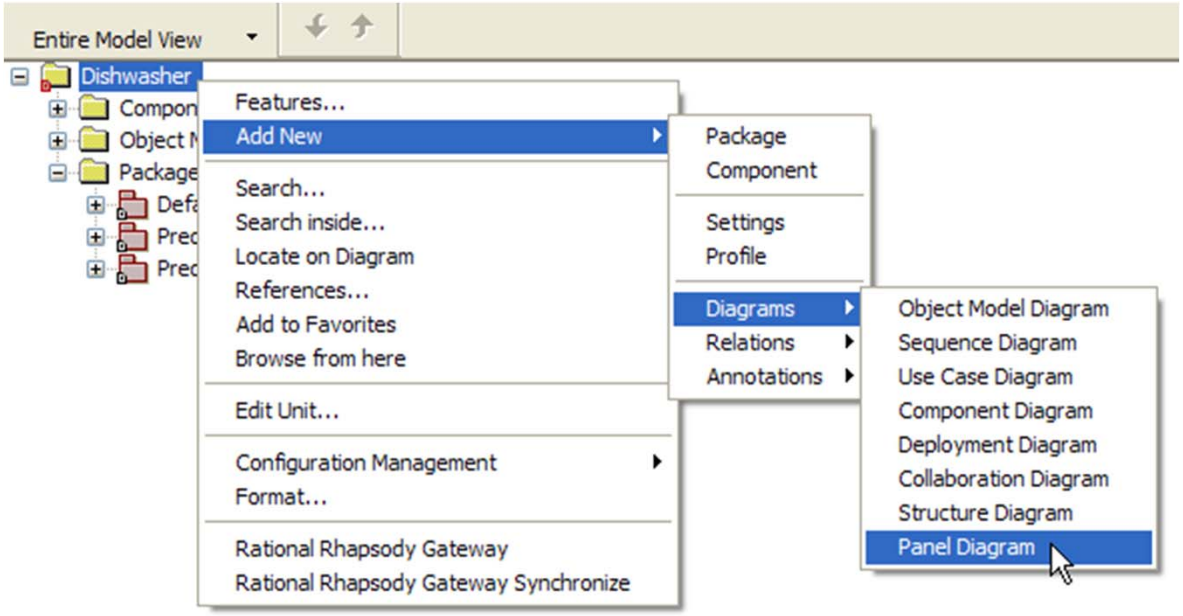


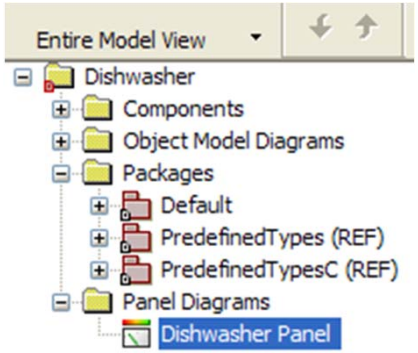
Additional Info: Command prompt

- Events and breakpoints can also be generated through the command prompt.
- For example, you can inject the `evStart` by typing `Dishwasher[0]->GEN(evStart)` in the command window.
- It may be useful to use the command window to invoke scripts.
- GEN is a macro that creates the event before sending it to the object. If there are multiple instances of a class, then you need to explicitly provide the instance. With only one instance, it is not necessary to write `Dishwasher[0]->GEN(evStart)`, because `instance [0]` is the default.

Panel diagram

- One way to test the Dishwasher is to use a panel.
- Add a Panel Diagram called *Dishwasher Panel*. To do so, right-click *Dishwasher* and select **Add New > Diagrams > Panel Diagram**.

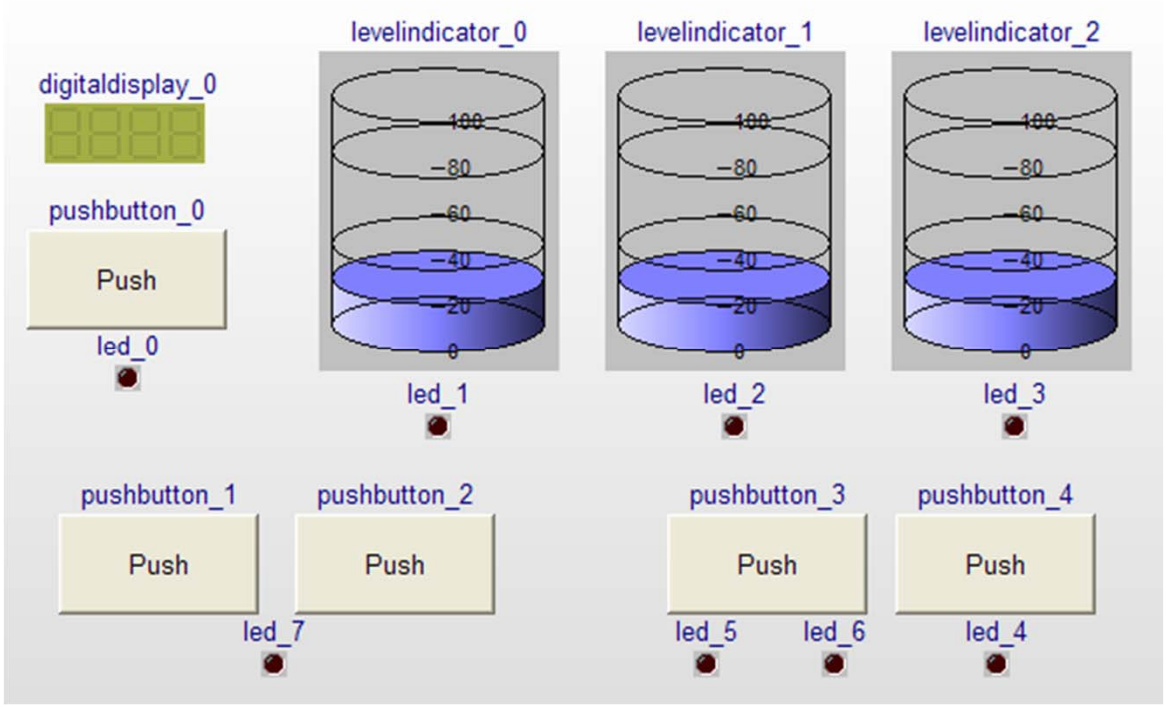


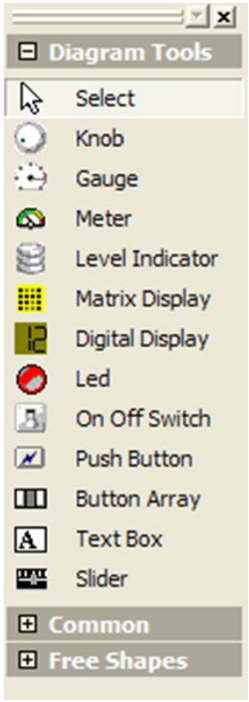


Panel Diagrams can only be used with animation.

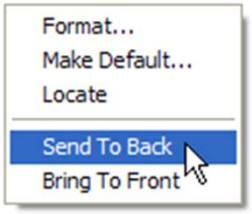
Panel diagram

- Add LEDs, push buttons, level indicators, and a digital display to the panel.

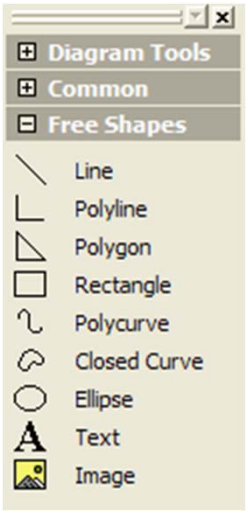


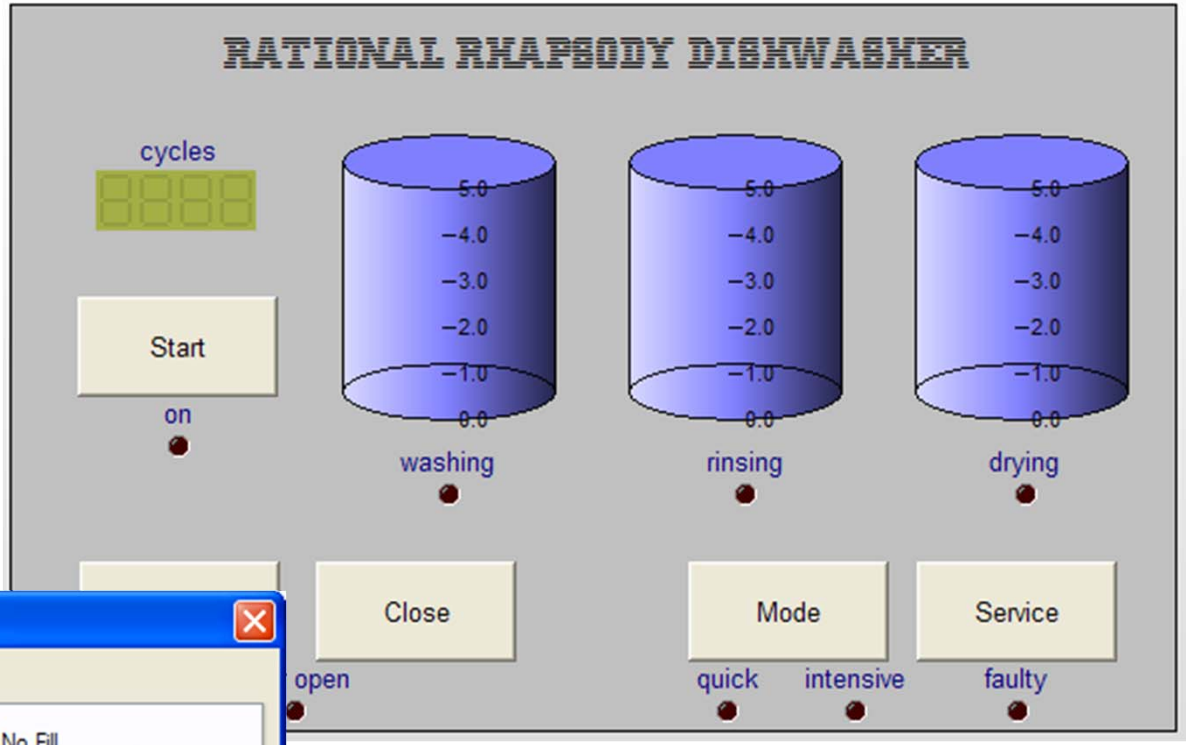


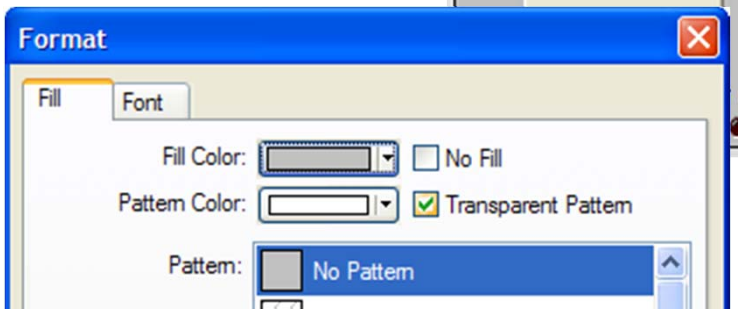
Box



- Draw a box around the panel.
- Right-click and select **Send to Back**.
- Right-click and select **Format…** to change the fill color.
- Add *Rational Rhapsody Dishwasher*, with desired font and size.

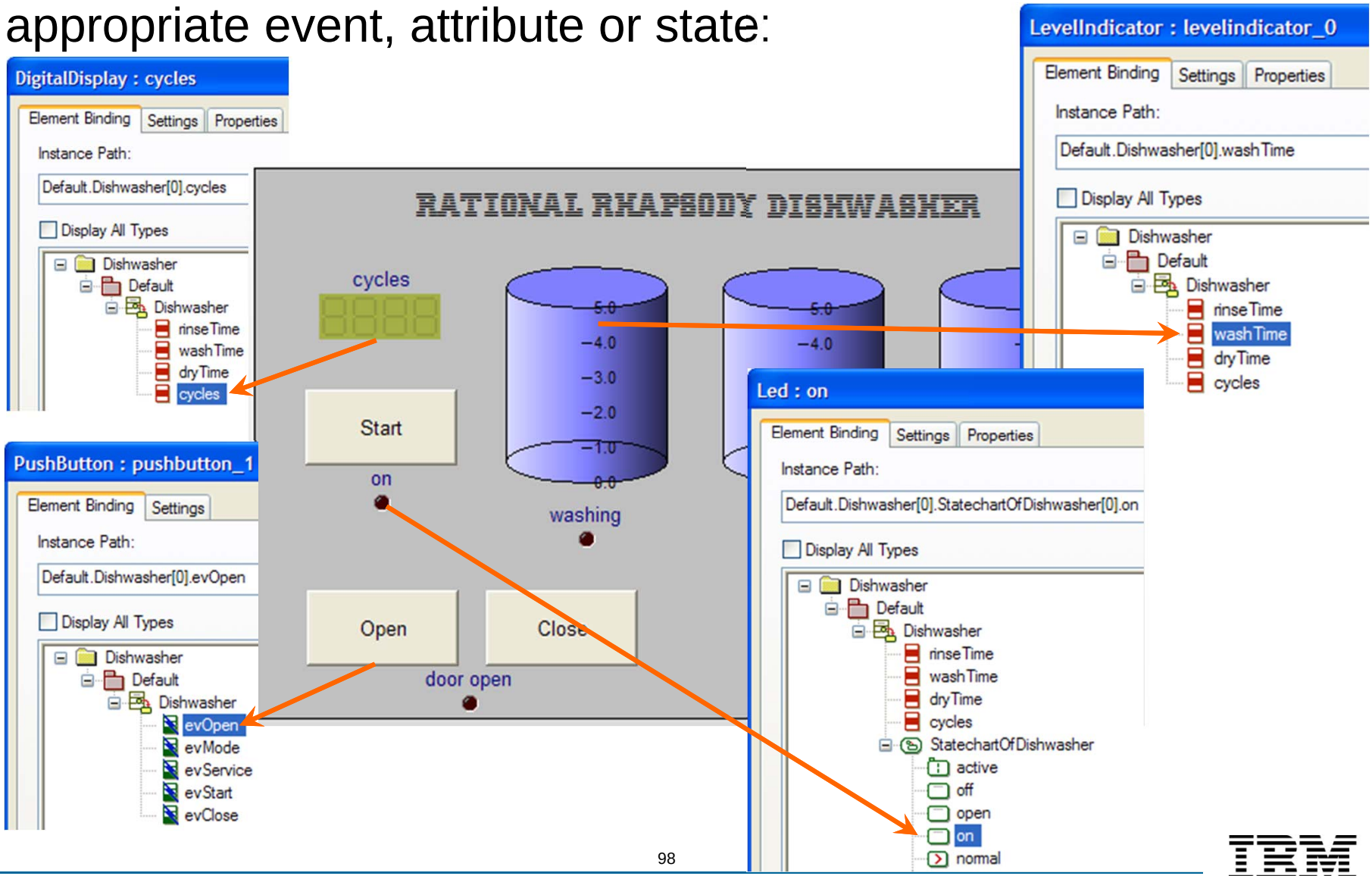






Bind

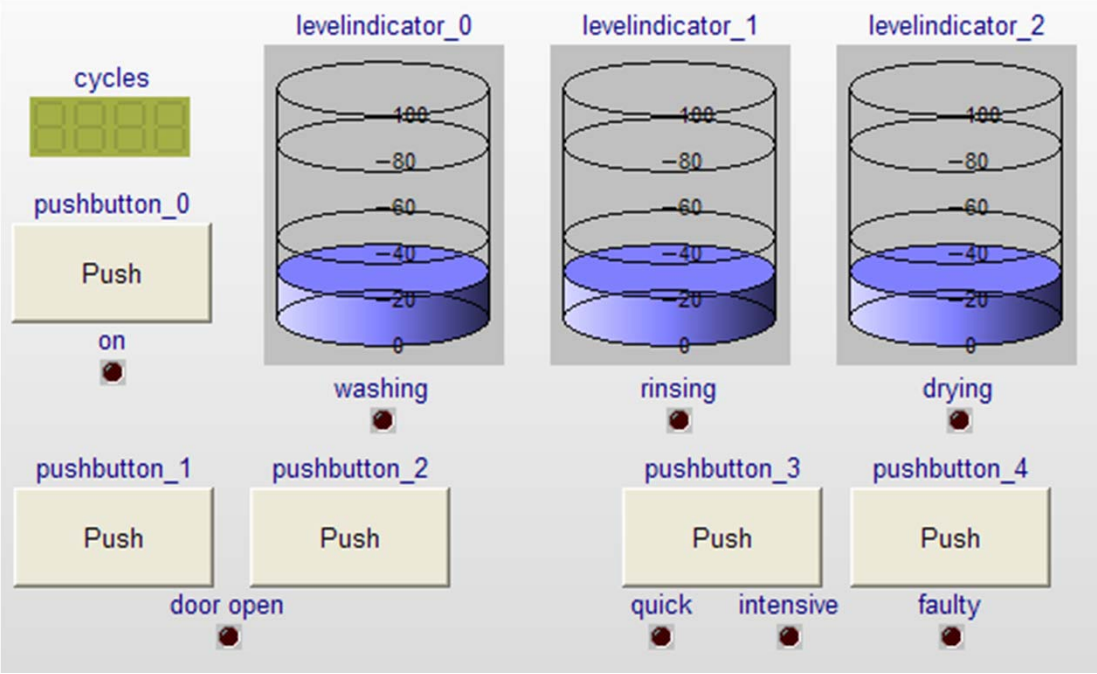
- Double-click each panel element and bind them to the appropriate event, attribute or state:



IBM

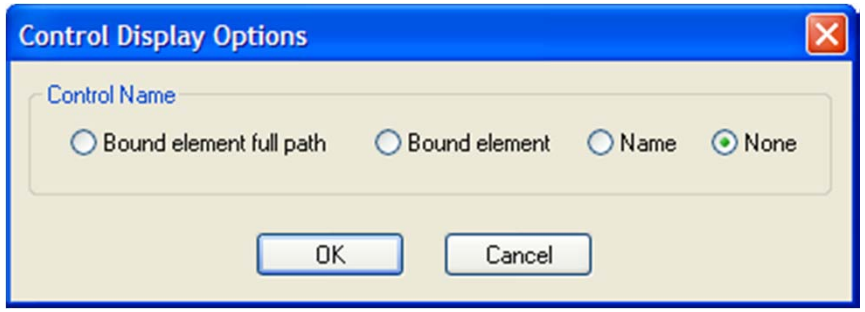
Renaming panel elements

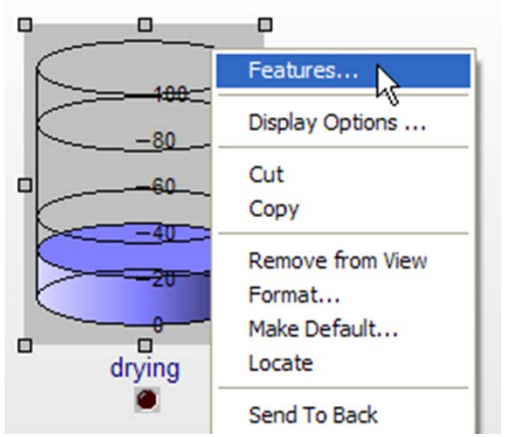
- Double-click on the text of each LED and the digital display, renaming the panel elements as shown below:

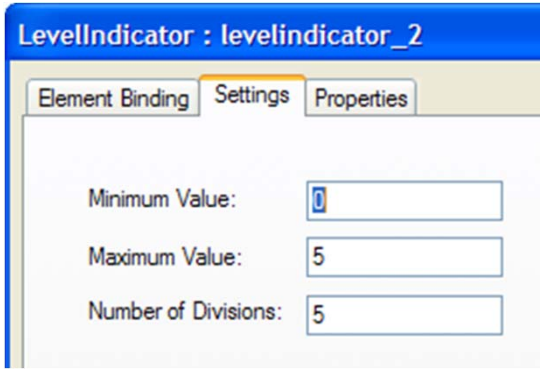


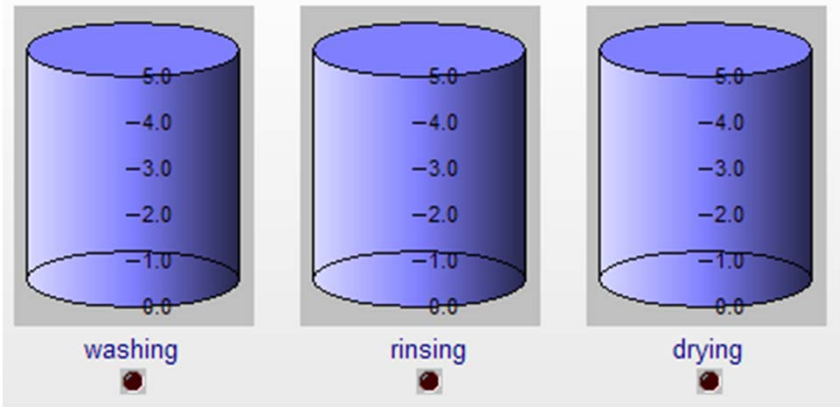
Level indicators

- For each Level Indicator:
  - Use the Display Options to display no name (**None**).
  - Use the **Features** to set the **Maximum Value** to 5.



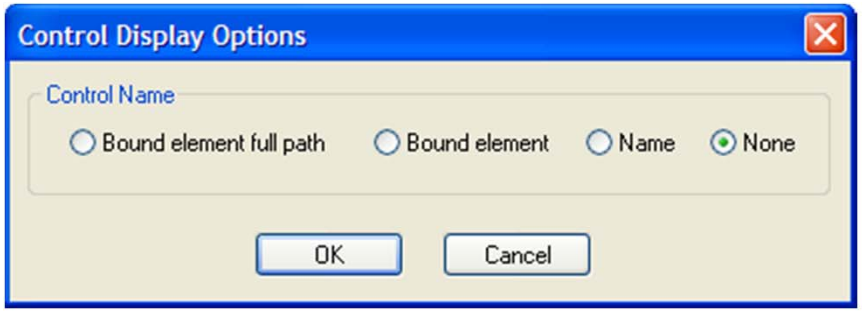


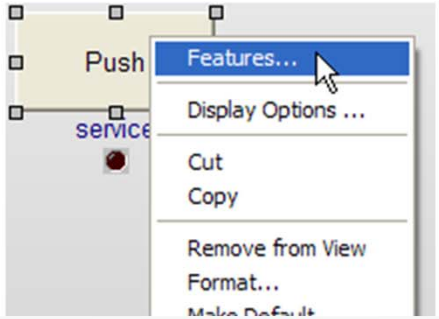


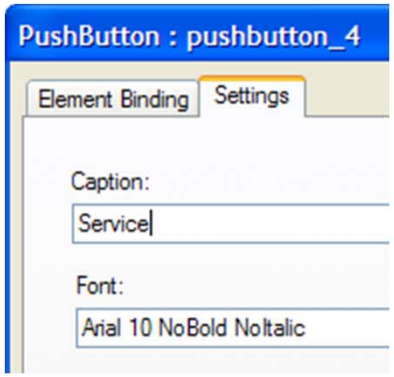


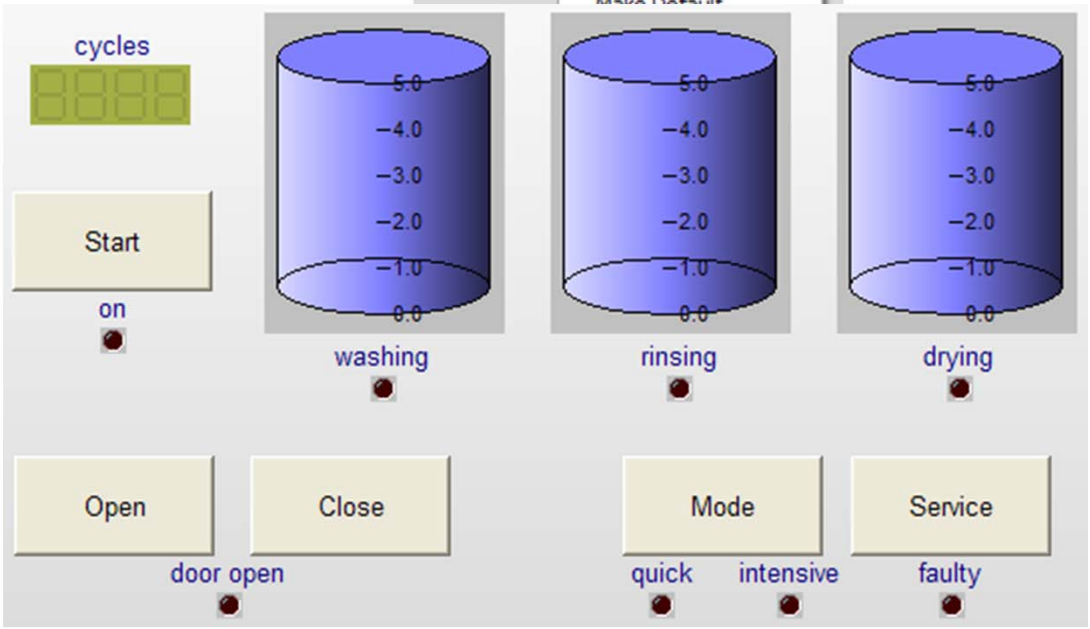
Push buttons

- For each push button:
  - Use the Display Options to display no name (**None**).
  - Select **Features** to set the caption appropriately:



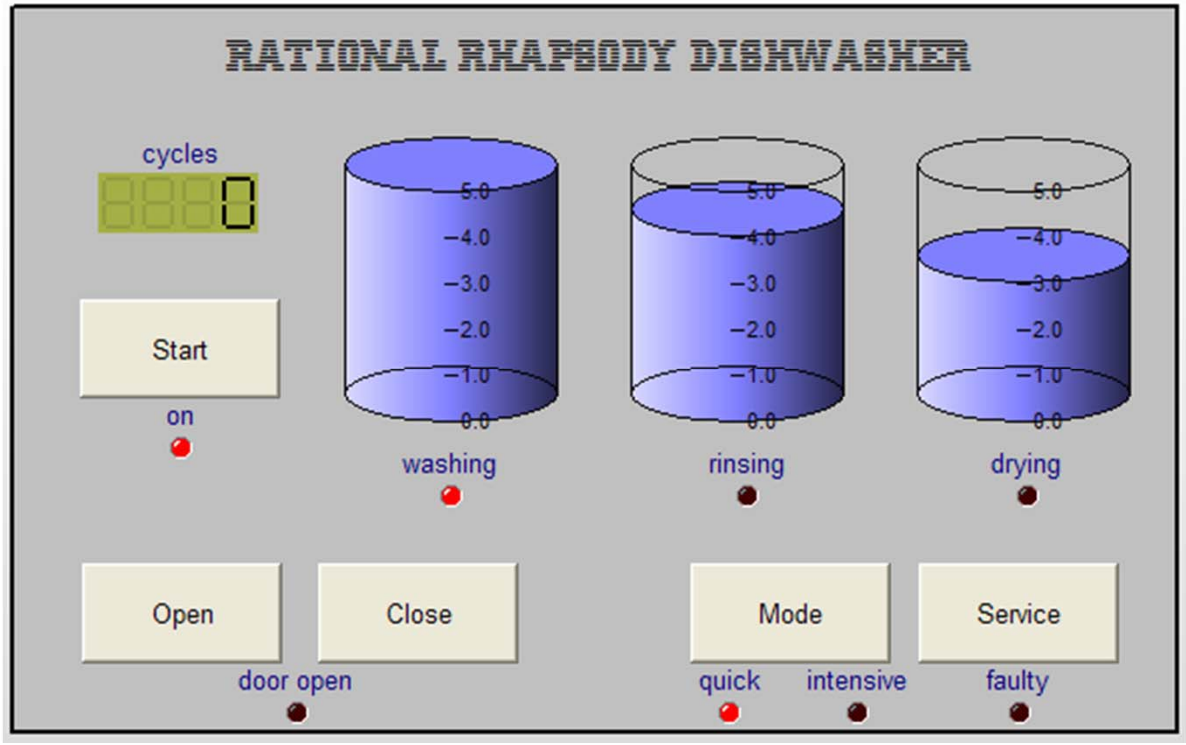






Panel

- There is no need to **Generate** or **Make**, just **Run**.
- Use the panel to drive the dishwasher.



- When using the panel, you should use the Debug configuration.

Where are we?

- Exercise 1 : Hello World
  - You start with the simplest example possible, just a single object that prints out Hello World.
- Exercise 2 : Count Down
  - Next, you create a simple counter using a simple statechart.
- Exercise 3 : Dishwasher
  - Finally, you create a dishwasher and a more complex statechart.

- Summary

Summary

- You should be starting to understand the basics of Rational Rhapsody, and you should now be able to do the following:
  - Create a new project.
  - Do some basic modeling using classes, attributes, operations, relations, and state charts.
  - Generate and compile code.
  - Set properties to customize the tool and the generated code.
  - Edit the code and roundtrip the changes back into the model.
  - Debug the model by injecting events, setting breakpoints, capturing behavior on sequence diagrams, visualizing the state of objects, and so on.