



IBM Software Group

Essentials of IBM® Rational® Rhapsody® v7.5 for Software Engineers (C++)

Basic Rational Rhapsody



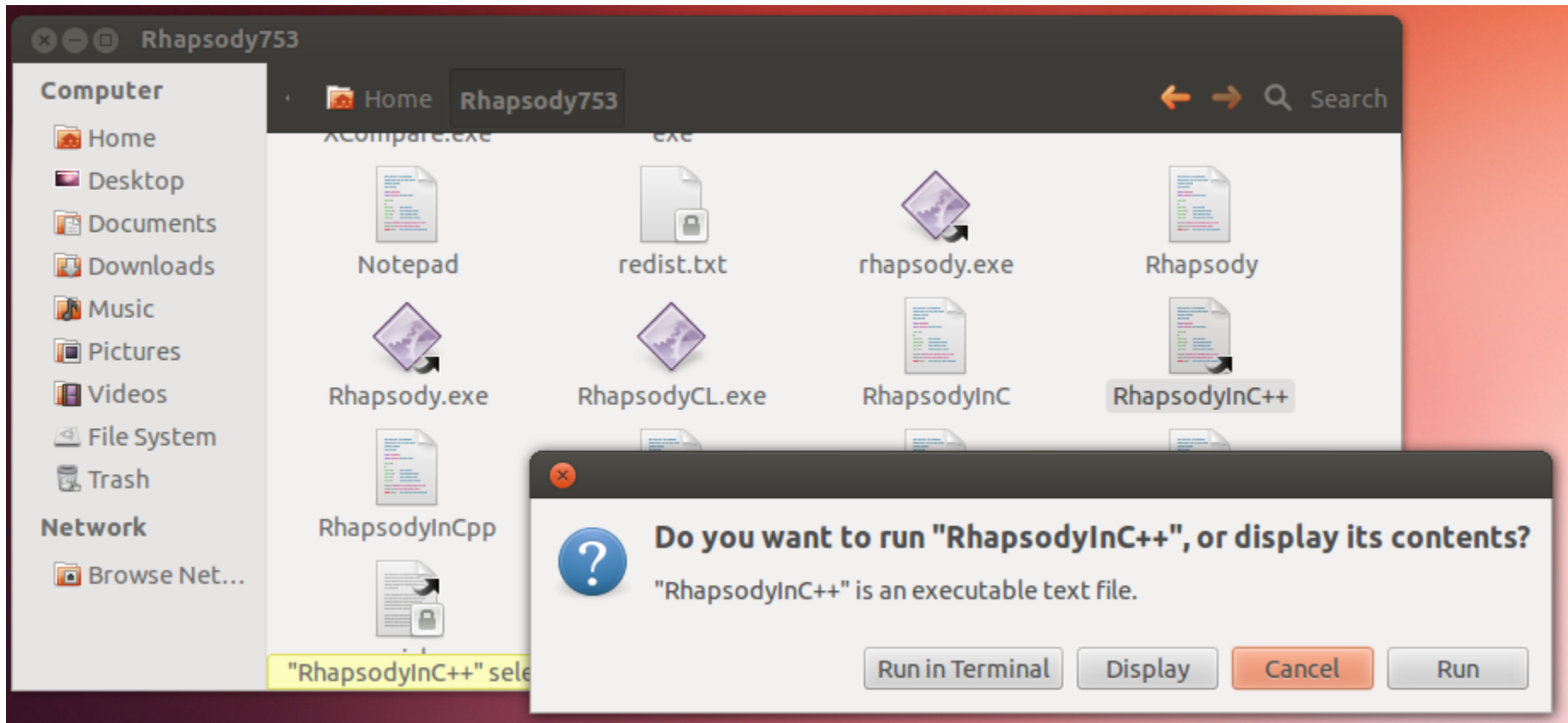
Rational. software

Exercise 1 : Hello World



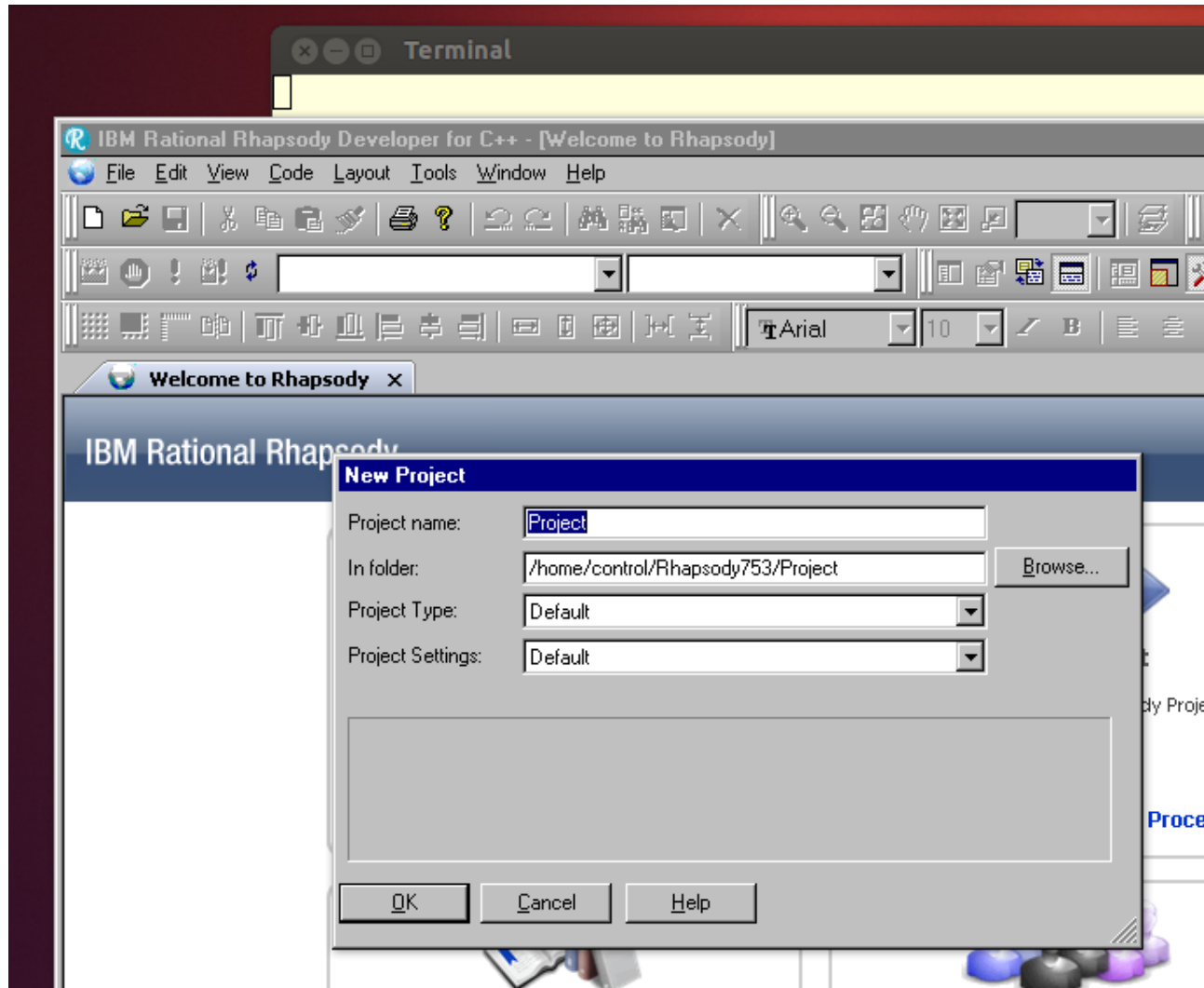
Start Rhapsody in C++

- Double click RhapsodyInC++
- Select **Run in Terminal**

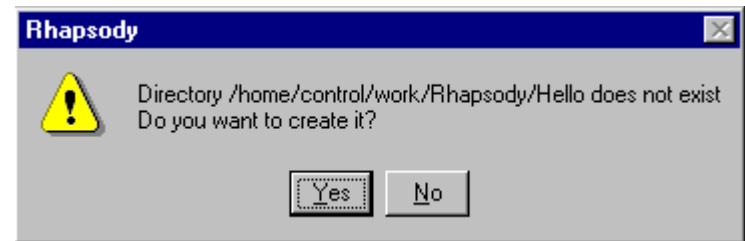
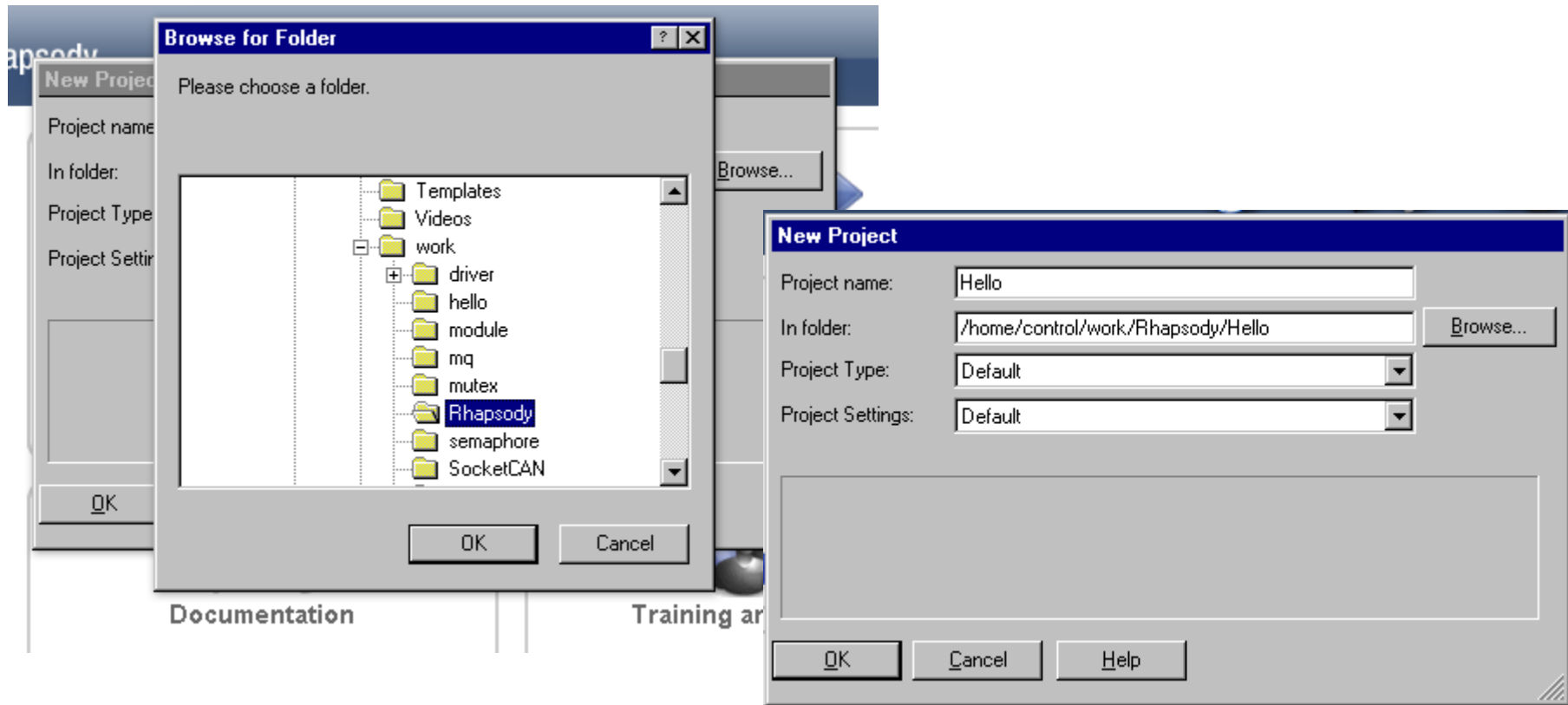


Creating a Project

- New from File menu

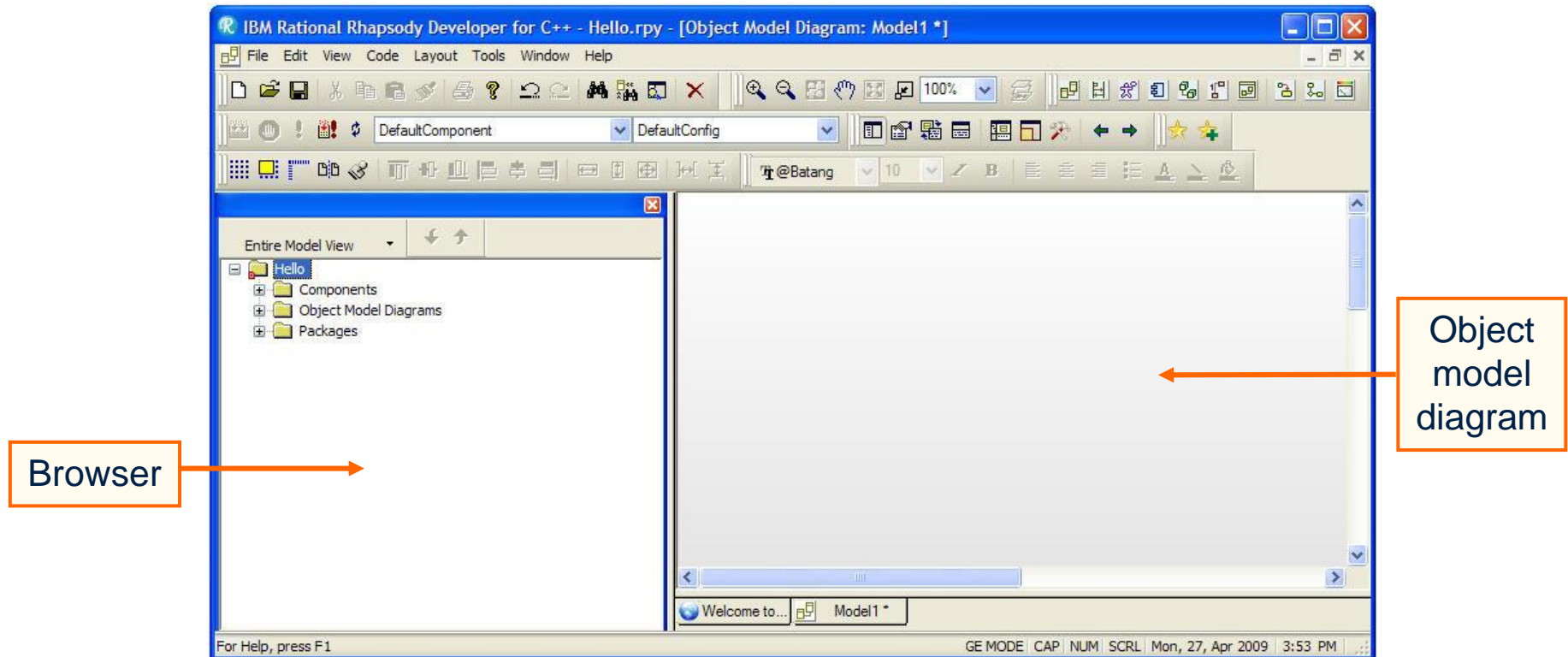


Select the working directory



Browser

- The browser shows you everything that is in the model.
- Note that Rational Rhapsody creates an Object Model Diagram (OMD).

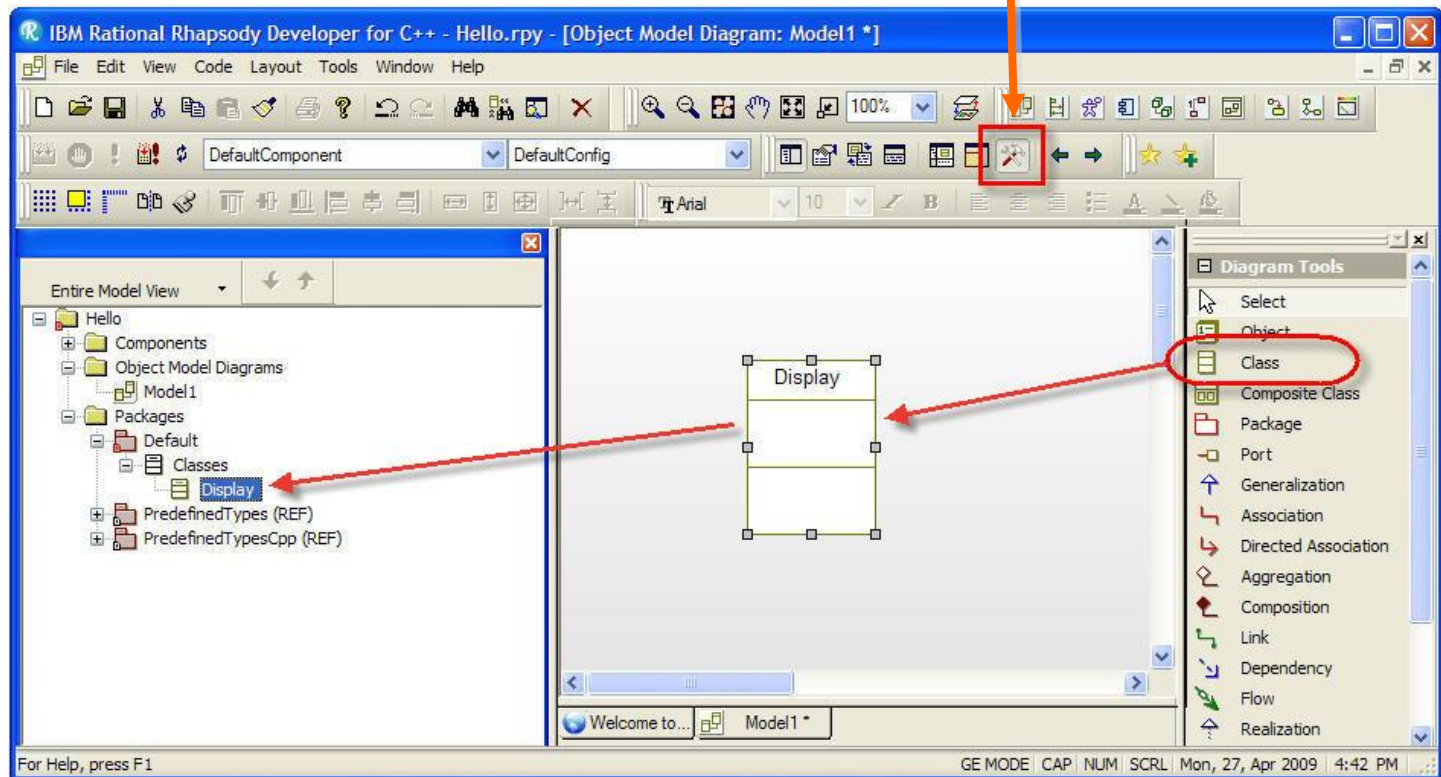


Drawing a class

- In this Object Model Diagram, click the **Class** icon  to draw a class named *Display*.

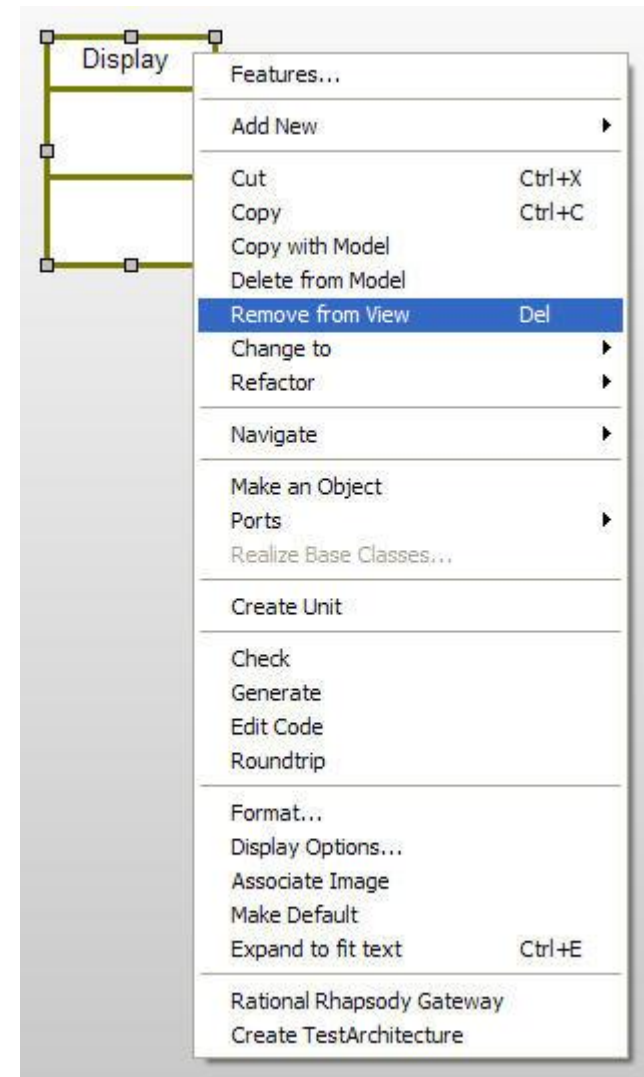
Show/Hide Drawing
Toolbar

Expand the
browser to see
that the class
Display also
appears in the
browser.



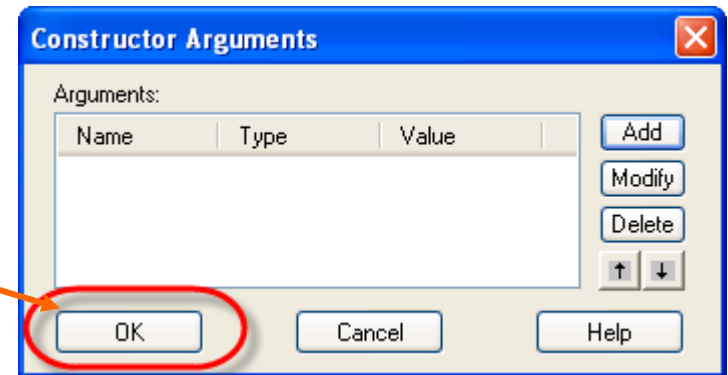
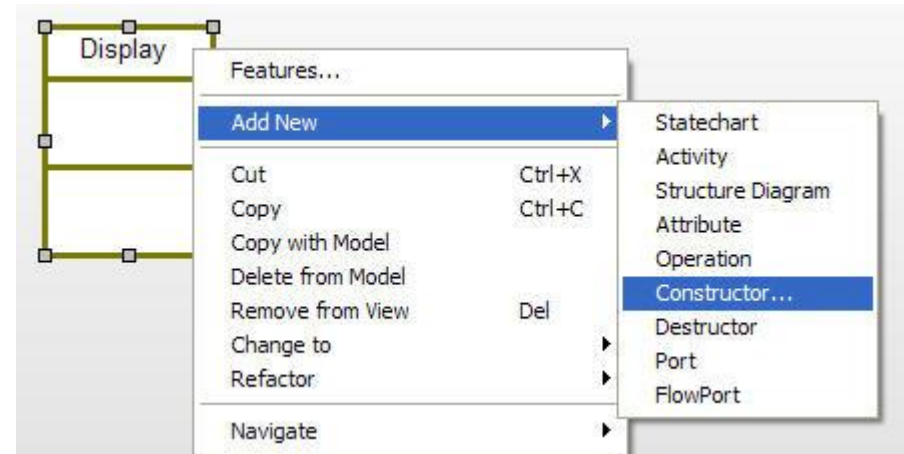
Remove from View / Delete from model

- Two ways of deleting a class
 - ▶ Remove the class from the view (this is what the Delete key does).
 - ▶ Delete the class from the model.
- If you use the delete key or select **Remove from View**, then the class *Display* is just removed from this diagram, but remains in the browser.
- If you select **Delete from Model**, then you must confirm with **Yes** in order to remove the class from the entire model.



Adding a constructor

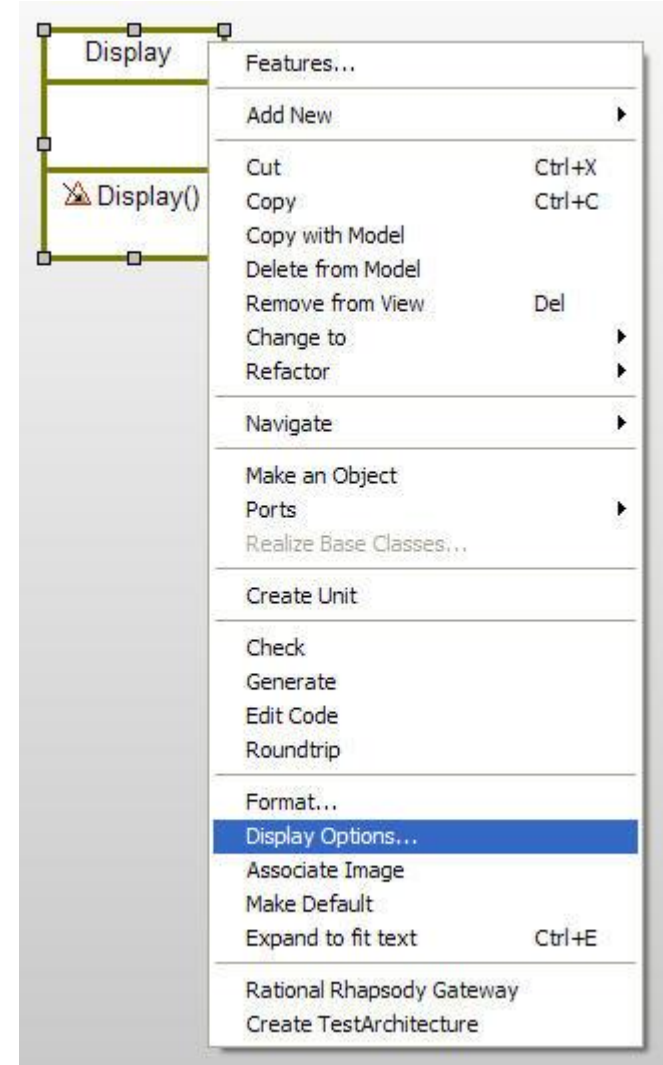
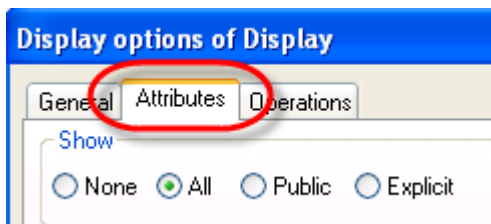
- The simplest way to add a constructor is to right-click on the class and choose **Add New > Constructor**.
- You do not need any constructor arguments; click **OK**.



Constructors may also be added through the features **Operations** tab. Click **New** and select **Constructor**.

Display options

- You would expect to see the constructor shown on the class on the Object Model Diagram.
- You can control what gets displayed on this view of the class by using *Display Options*.
- Right-click **Display** class and select **Display Options**.
 - ▶ Set the options to display **All** attributes and **All** operations.



Display constructor

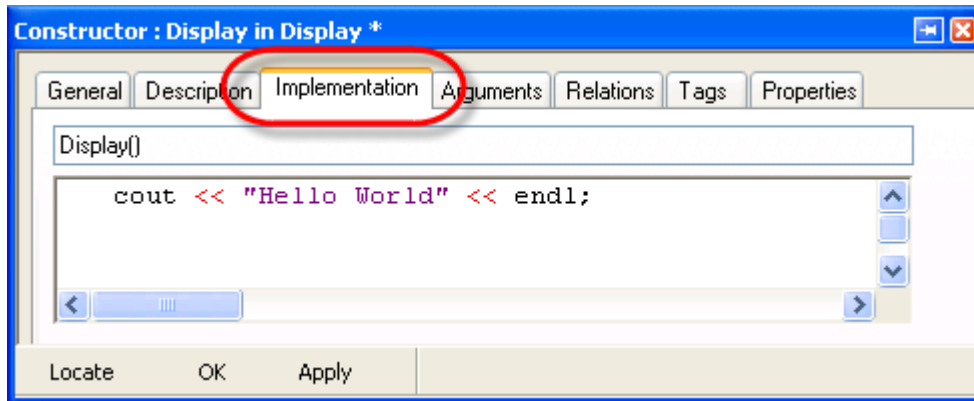
- You should be able to see the constructor is now shown in both the browser and the OMD (object model diagram).

The screenshot shows the IBM Rational Rhapsody Developer for C++ interface. The main window displays an Object Model Diagram (OMD) for a class named 'Display'. The diagram shows the class 'Display' with a constructor 'Display()' indicated by a small triangle icon. The 'Entire Model View' on the left shows the project structure, with the 'Display' class and its 'Operations' folder containing 'Display()' highlighted. A callout box labeled 'Constructor' with a triangle icon points to the 'Display()' entry in the 'Operations' folder. The 'Diagram Tools' palette on the right lists various modeling tools. The status bar at the bottom indicates the current date and time: 'Mon, 27, Apr 2009 11:31 PM'.

Adding an implementation

- Select the **Display** constructor in the browser and double-click to open the features window.
- Select the **Implementation** tab and enter the following:

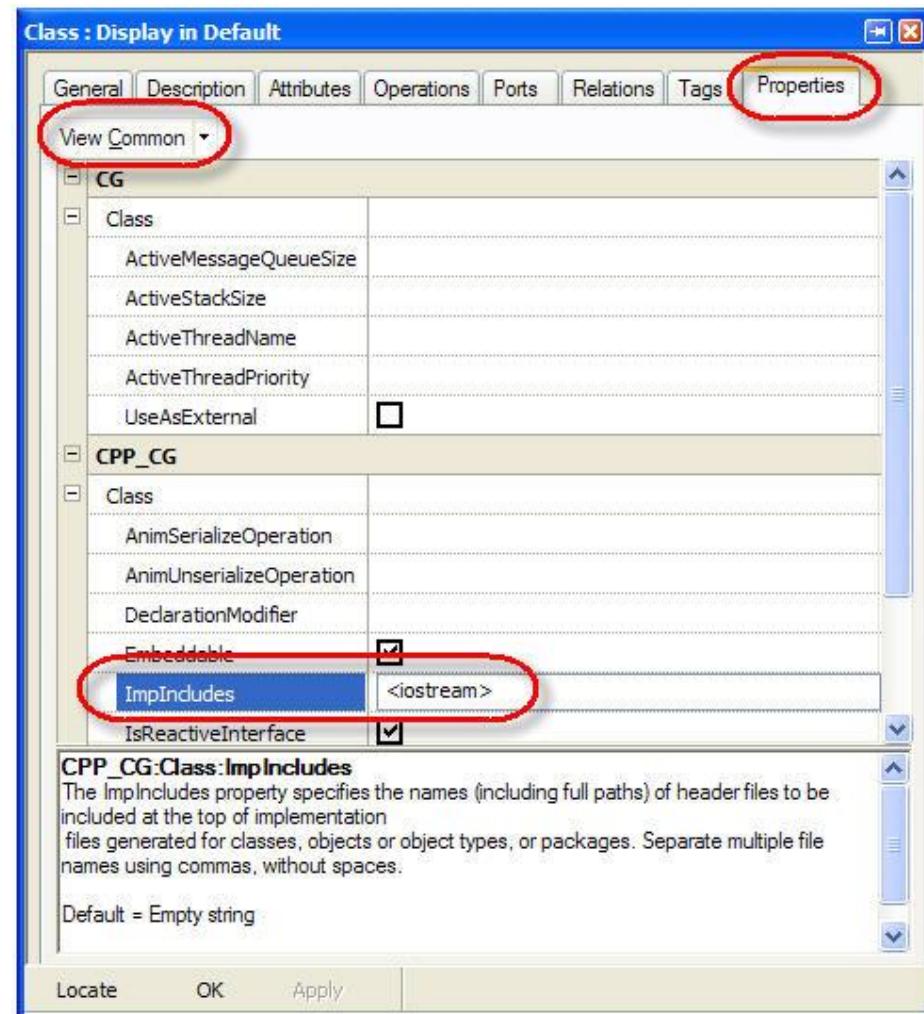
```
cout << "Hello World" << endl;
```



If you are not using Visual C++ 6.0, then you should add the `std` namespace, for example, `std::cout << "Hello World" << std::endl;` Or, set the property `CPP_CG::Class::ImplementationProlog` to `using namespace std;`

#include <iostream>

- Since you have used *cout*, you must add an include of the *iostream* header to 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
 - ▶ Enter *<iostream>* into the “ImplIncludes” property.



ImplIncludes is an abbreviation for Implementation Includes.

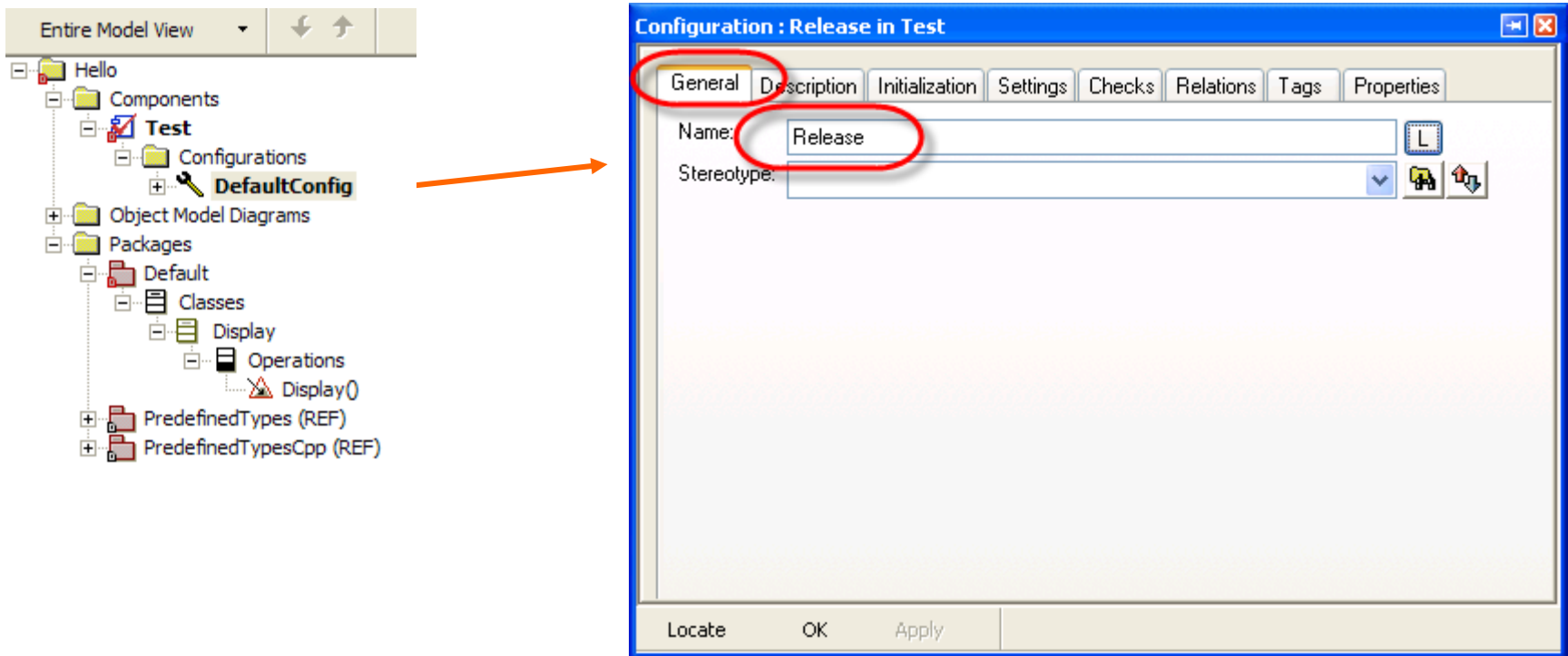
Renaming a component

- 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*. Also name the Directory to *Test*.

The screenshot shows the Eclipse IDE interface. On the left, the 'Entire Model View' browser displays a project named 'Hello' with a sub-component 'DefaultComponent' highlighted. An orange arrow points from this component to the 'Component: DefaultComponent in Hello' dialog box on the right. The dialog box has several tabs: 'General', 'Scope', 'Variation Points', 'Description', 'Relations', 'Tags', and 'Properties'. The 'General' tab is active. In this tab, the 'Name' field is set to 'Test', the 'Directory' field is set to 'Test', and the 'Type' section has the 'Executable' radio button selected. An orange arrow points from a box labeled 'Executable' to the 'Executable' radio button. The dialog box also has 'Locate', 'OK', and 'Apply' buttons at the bottom.

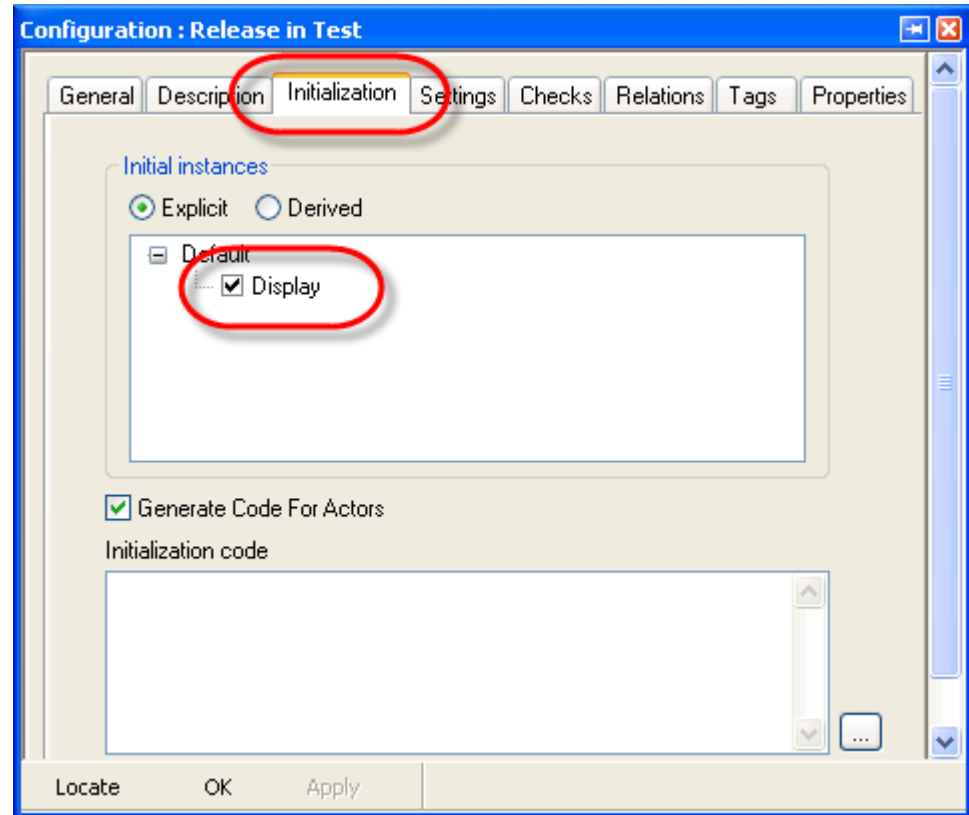
Test component

- Now expand Configurations and rename the *DefaultConfig* to *Release*.



Initial instance

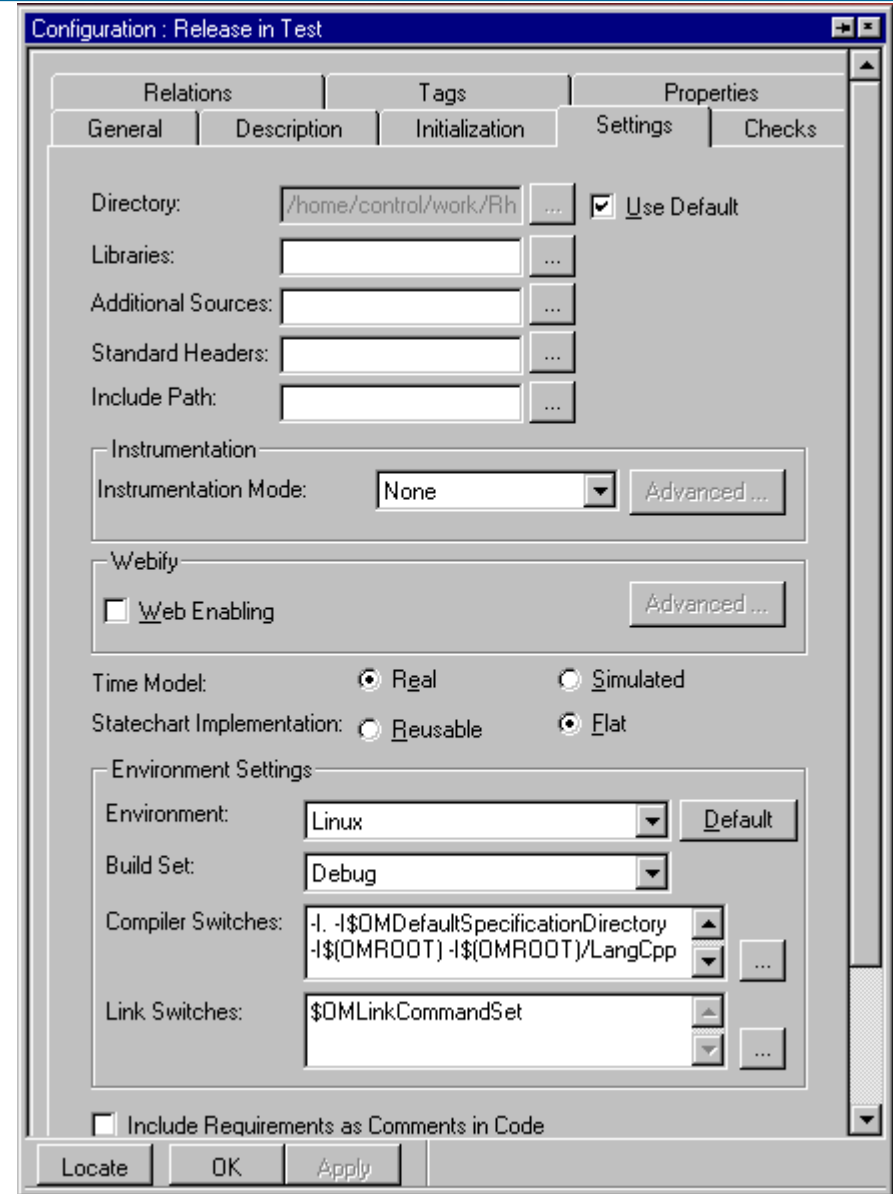
- Select the **Initialization** tab, expand the Default package, and select the **Display** class.
- The main will create an initial instance of the Display class.



Settings

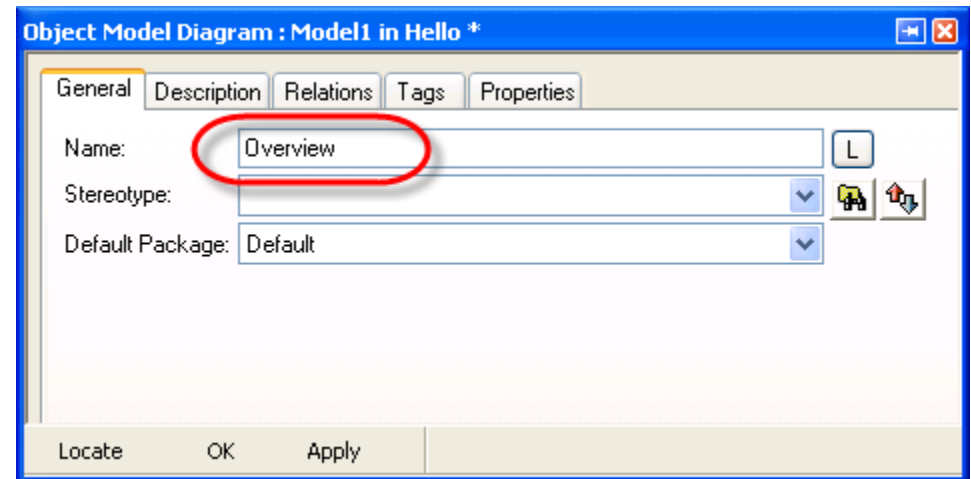
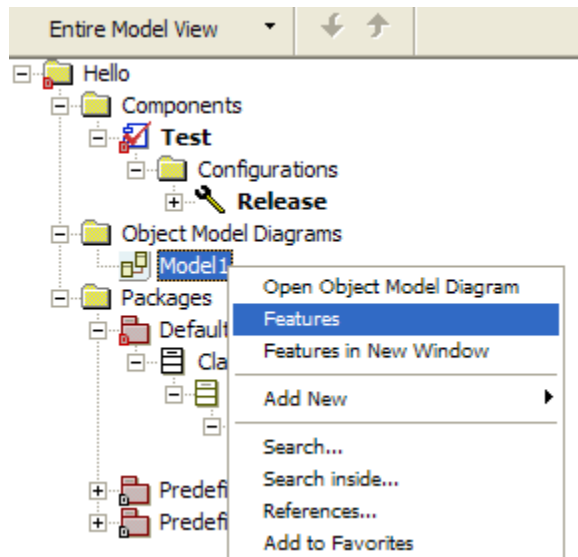
- You need to select an environment so that Rational Rhapsody knows how to create an appropriate Makefile.
- Select the **Settings** tab.
- Select the appropriate environment, for example: Linux.

You will learn about the many other settings later.





Renaming the OMD

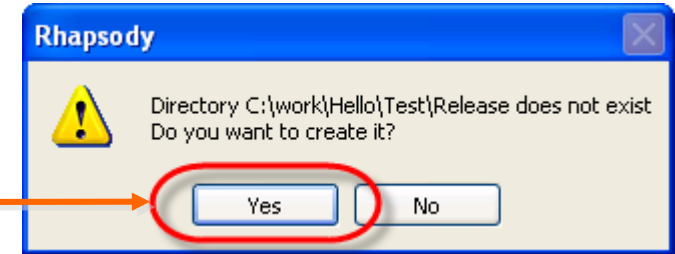
- Expand the Object Model Diagrams in the browser. Right-click the Object Model Diagram **Model1** to invoke the features dialog.
- Rename the diagram from *Model1* to *Overview*.



Generating code

- You are now ready to generate code.

- ▶ Save the model. 
- ▶ Select Generate/Make/Run. 
- ▶ Click **Yes** to the question:



```
All Checks Terminated Successfully

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

Code generated to directory: /home/control/work/Rhapsody/Hello/Test/Release
Generating file Display.h
Generating file Display.cpp
Generating main file MainTest.h
Generating main file MainTest.cpp
Generating make file Test.mak

Code Generation Done

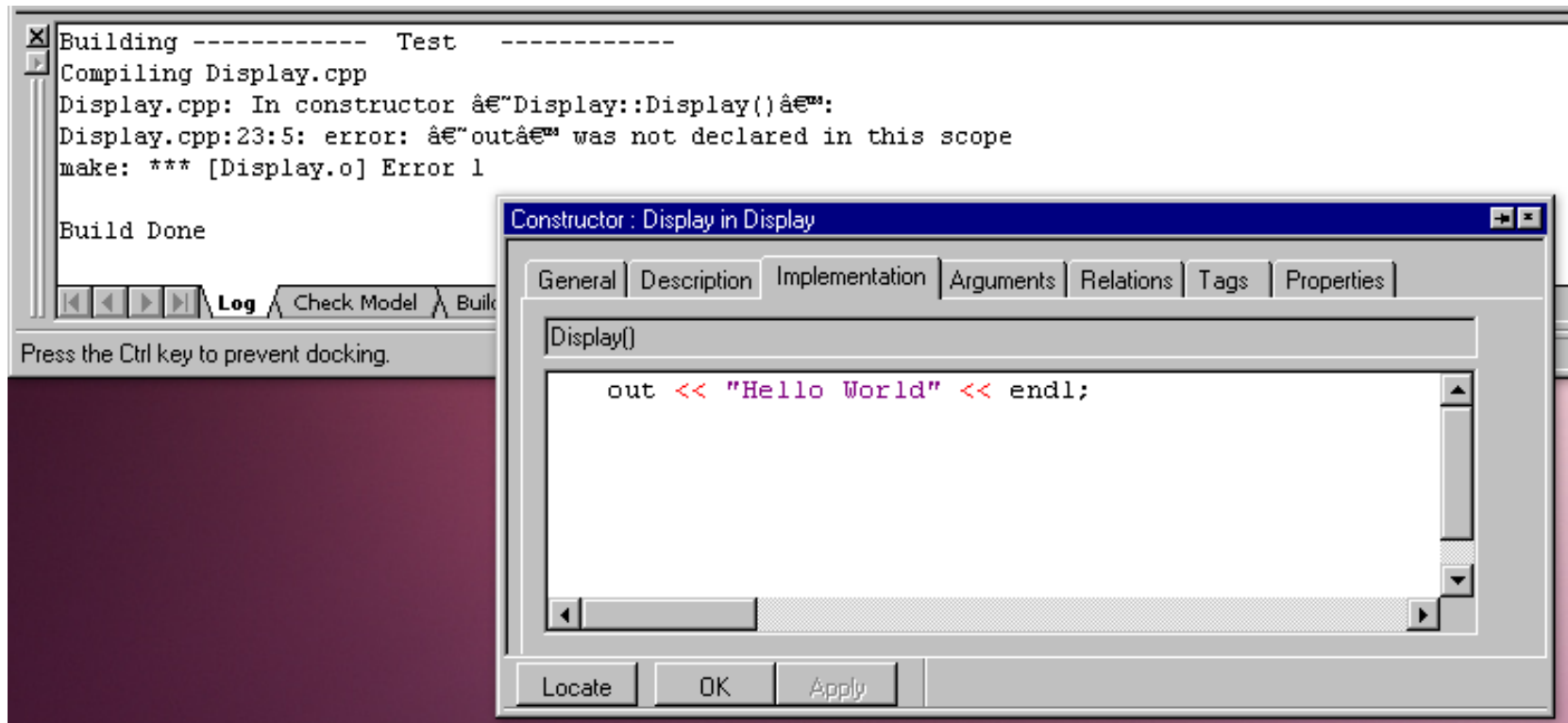
0 Error(s), 0 Warning(s), 0 Message(s)
Building ----- Test -----
Compiling Display.cpp
Linking Test

Build Done
```

Log | Check Model | Build | Configuration Management | Animation

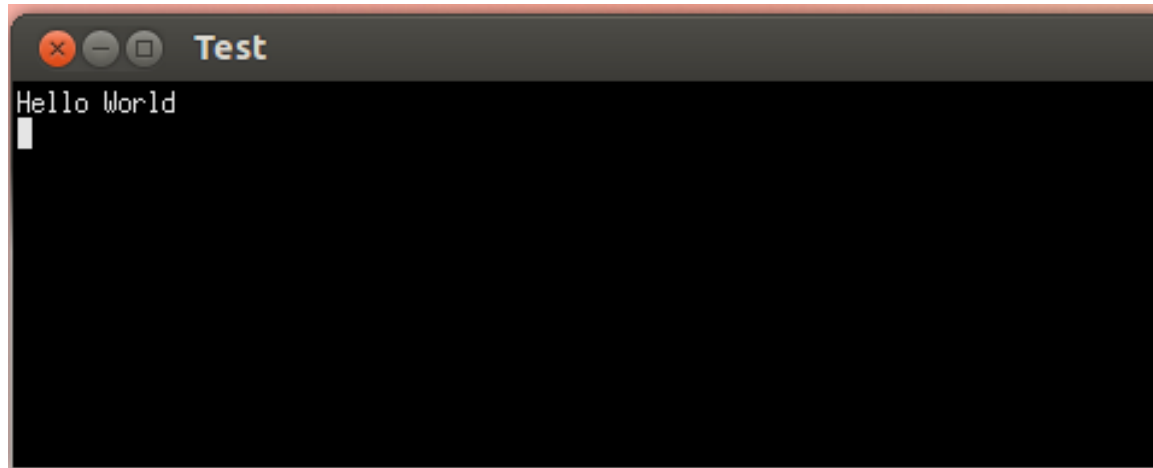
Handling errors


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



Hello World

- You should see the following:



- Before continuing, make sure you stop the executable by one of the following methods:
 - ▶ Closing the console window.
 - ▶ Using the Stop Make / Execution button. 
 - ▶ Ctrl+Break.

Generated files

- The generated files are located in the following directory:

Display class

Main

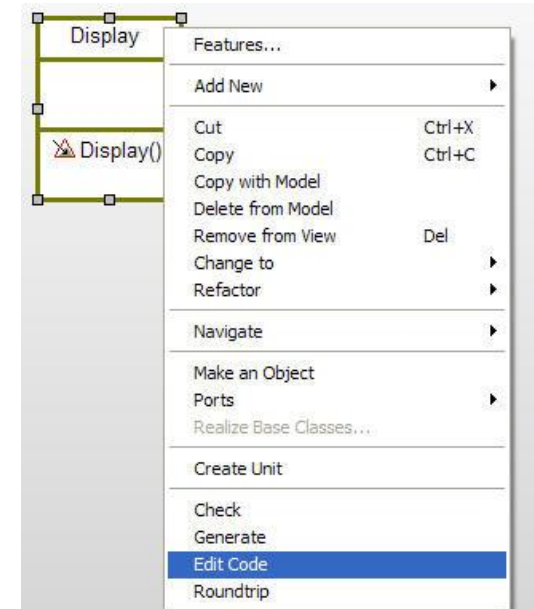
Executable

Makefile

Name	Size	Type	Date Modified
Display.cpp	775 bytes	C++ source code	Fri 29 Dec 2017 05:12:05 PM KST
Display.h	804 bytes	C header	Fri 29 Dec 2017 05:12:05 PM KST
Display.o	27.8 kB	object code	Fri 29 Dec 2017 05:12:05 PM KST
error.txt	0 bytes	plain text document	Fri 29 Dec 2017 05:12:05 PM KST
MainTest.cpp	982 bytes	C++ source code	Fri 29 Dec 2017 05:12:05 PM KST
MainTest.h	594 bytes	C header	Fri 29 Dec 2017 05:12:05 PM KST
MainTest.o	26.1 kB	object code	Fri 29 Dec 2017 05:12:05 PM KST
Release.cg_info	812 bytes	plain text document	Fri 29 Dec 2017 05:12:05 PM KST
Test	733.6 kB	executable	Fri 29 Dec 2017 05:12:06 PM KST
Test.mak	3.7 kB	plain text document	Fri 29 Dec 2017 05:12:05 PM KST

Editing the code

- You can edit the generated files from within Rational Rhapsody.
- Select the **Display** class, right-click, and select **Edit Code**.
- Both the implementation (.cpp) and specification (.h) are shown in tabbed windows.

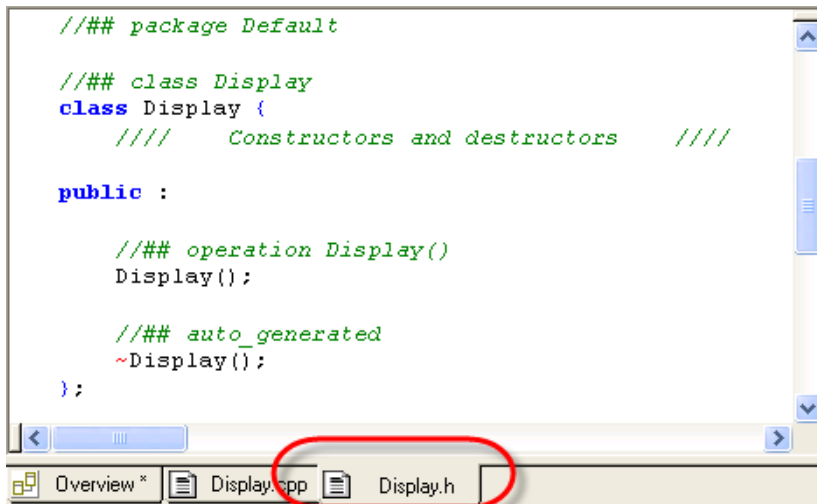


```
/// package Default
/// class Display
class Display {
    /// Constructors and destructors    ///

public :

    /// operation Display()
    Display();

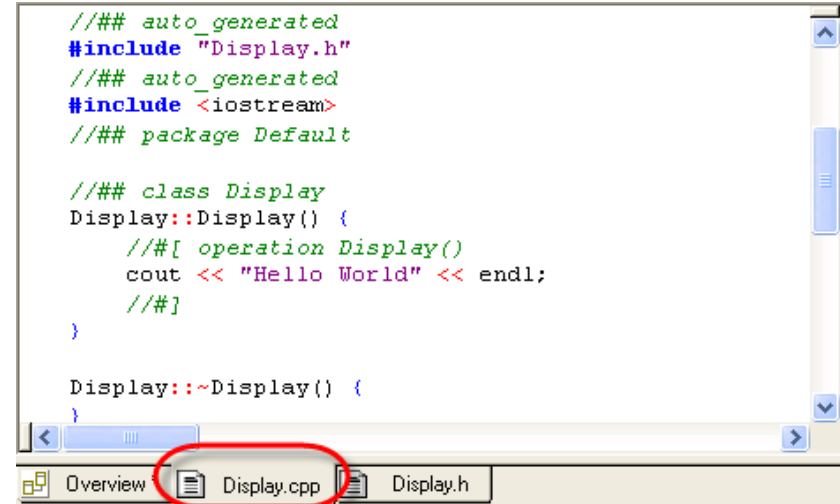
    /// auto_generated
    ~Display();
};
```

A screenshot of the Rational Rhapsody code editor showing the 'Display.h' header file. The code is displayed in a monospaced font with syntax highlighting. The 'Display.h' tab is highlighted in the bottom tab bar with a red circle.

```
/// auto_generated
#include "Display.h"
/// auto_generated
#include <iostream>
/// package Default

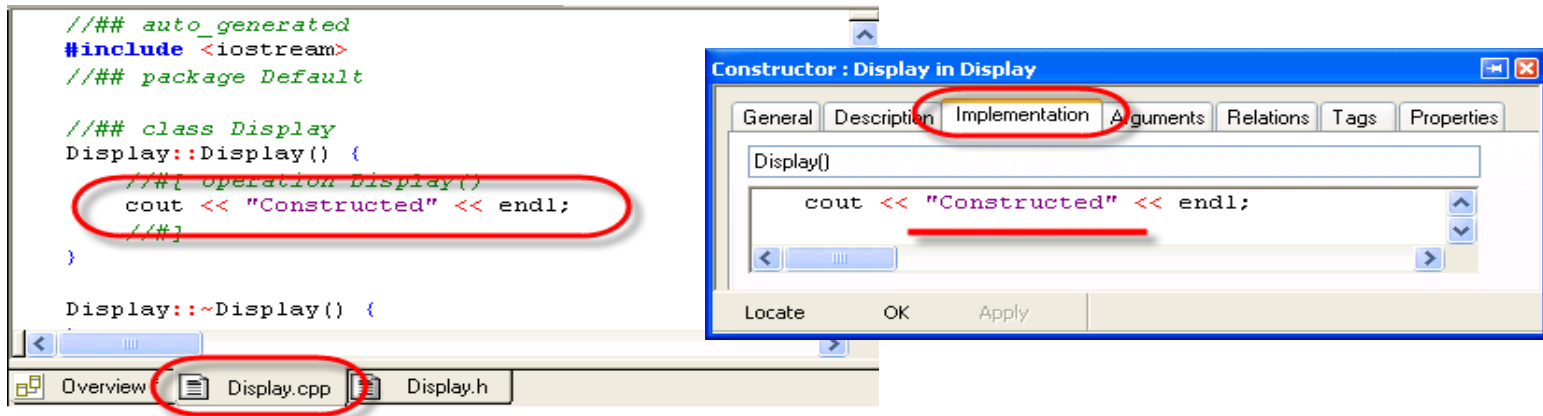
/// class Display
Display::Display() {
    /// operation Display()
    cout << "Hello World" << endl;
    ///
}

Display::~Display() {
}
```

A screenshot of the Rational Rhapsody code editor showing the 'Display.cpp' implementation file. The code is displayed in a monospaced font with syntax highlighting. The 'Display.cpp' tab is highlighted in the bottom tab bar with a red circle.

Modifying the code

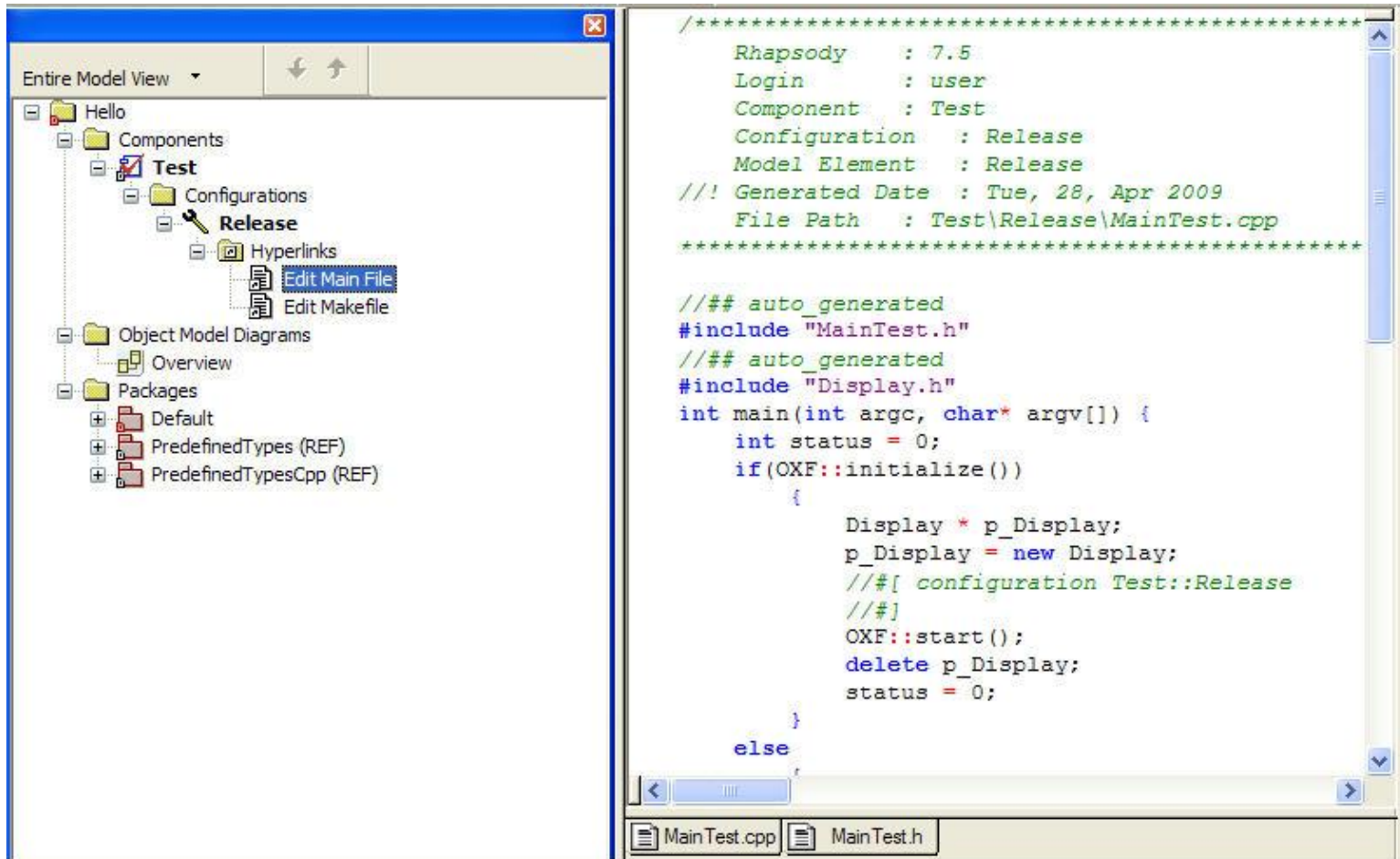
- You can modify the generated code.
- In the Display.cpp file, change the implementation to print out *Constructed* instead of *Hello World*.
- Transfer the focus back to another window to roundtrip the modifications back into the model.
- Note that the model has been updated automatically.



- In general, the roundtripping works very well, but beware not everything can be roundtripped.

Displaying the Main and Make

- The Main and Makefile can be displayed by simply double-clicking the hyperlinks:

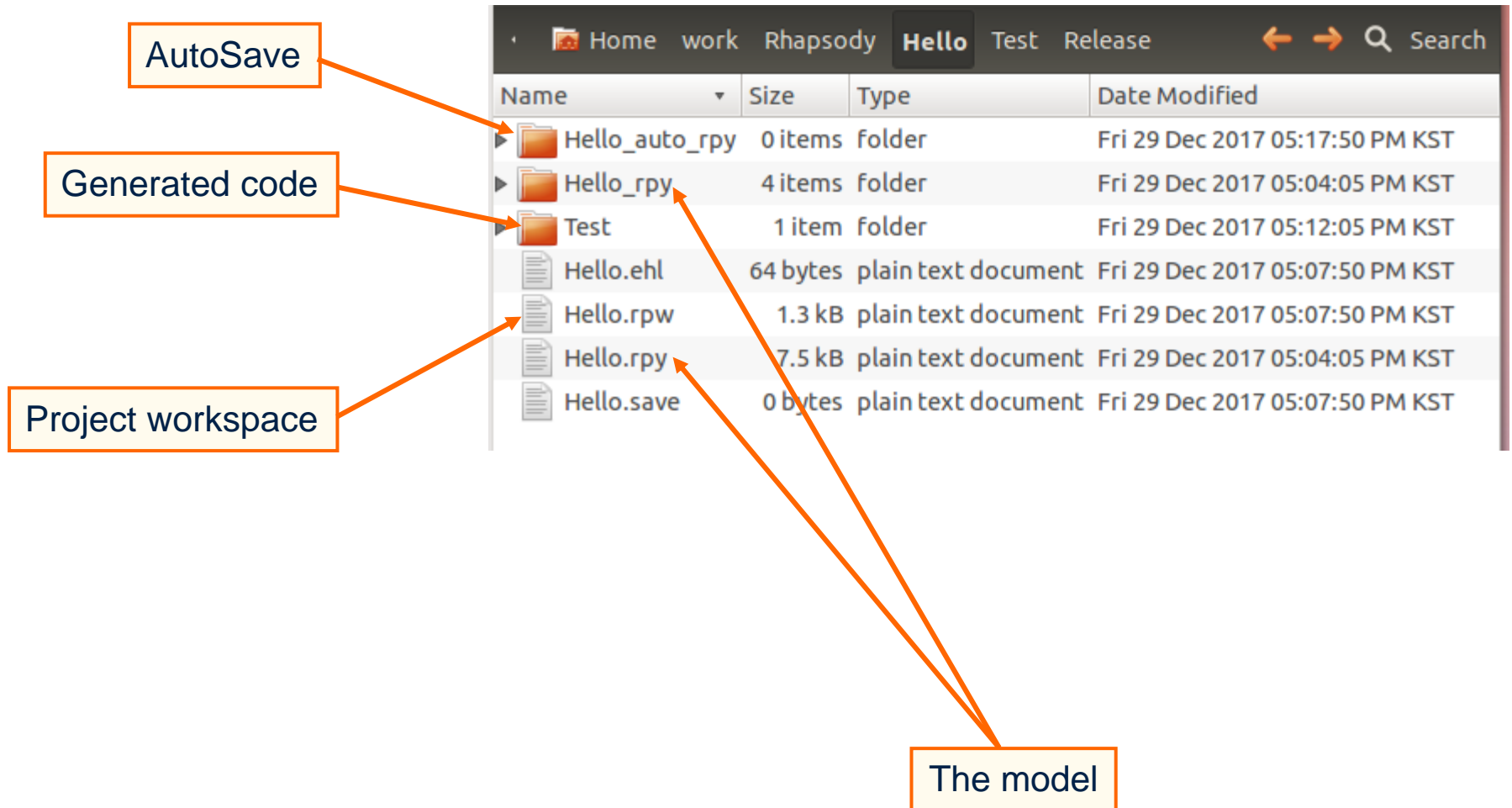


The screenshot displays a software development environment with two main panes. The left pane, titled 'Entire Model View', shows a hierarchical tree structure. Under the 'Hello' folder, there is a 'Components' folder containing a 'Test' folder. Inside 'Test', there is a 'Configurations' folder with a 'Release' configuration. Under 'Release', there is a 'Hyperlinks' folder containing two items: 'Edit Main File' and 'Edit Makefile'. The right pane is a code editor showing the content of 'MainTest.cpp'. The code is as follows:

```
/******  
Rhapsody      : 7.5  
Login         : user  
Component     : Test  
Configuration  : Release  
Model Element : Release  
//! Generated Date : Tue, 28, Apr 2009  
File Path    : Test\Release\MainTest.cpp  
*****  
  
//## auto_generated  
#include "MainTest.h"  
//## auto_generated  
#include "Display.h"  
int main(int argc, char* argv[]) {  
    int status = 0;  
    if(OXF::initialize())  
    {  
        Display * p_Display;  
        p_Display = new Display;  
        //#[ configuration Test::Release  
        //#]  
        OXF::start();  
        delete p_Display;  
        status = 0;  
    }  
    else  
    {  
        status = 1;  
    }  
}
```

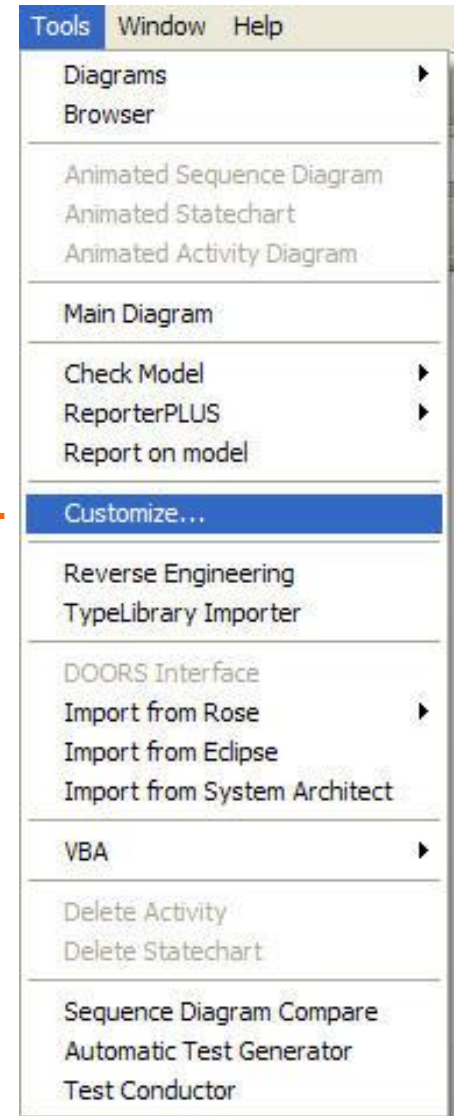
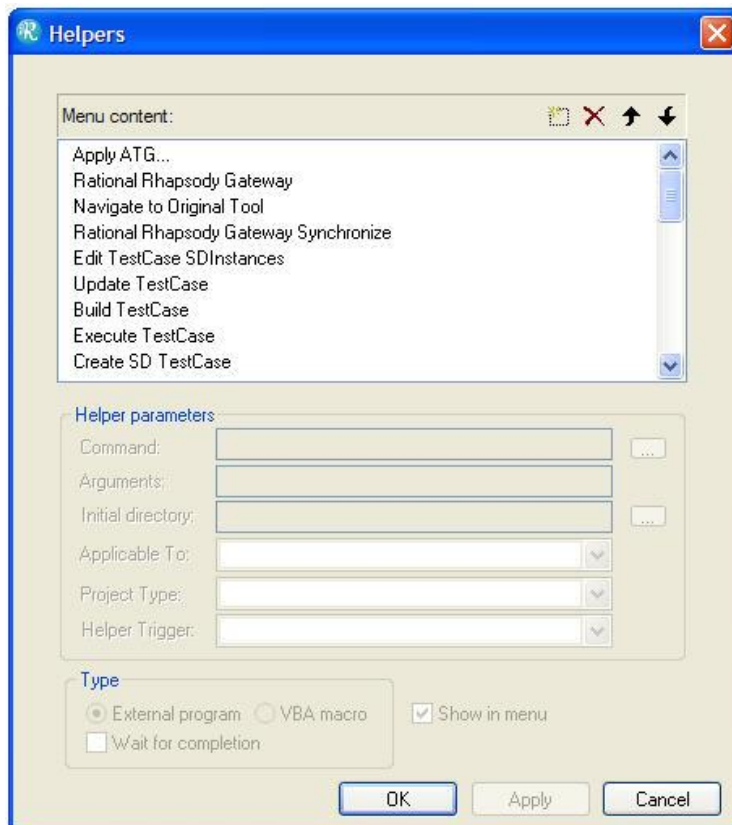
The code editor also shows a tab for 'MainTest.h' at the bottom.

Project files




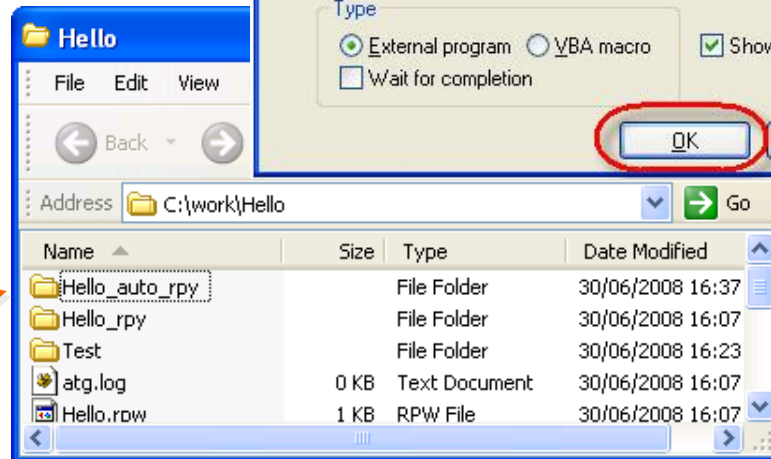
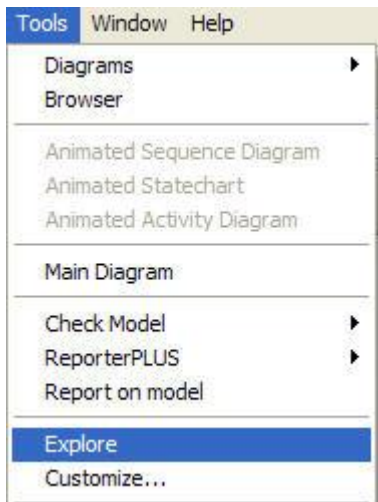
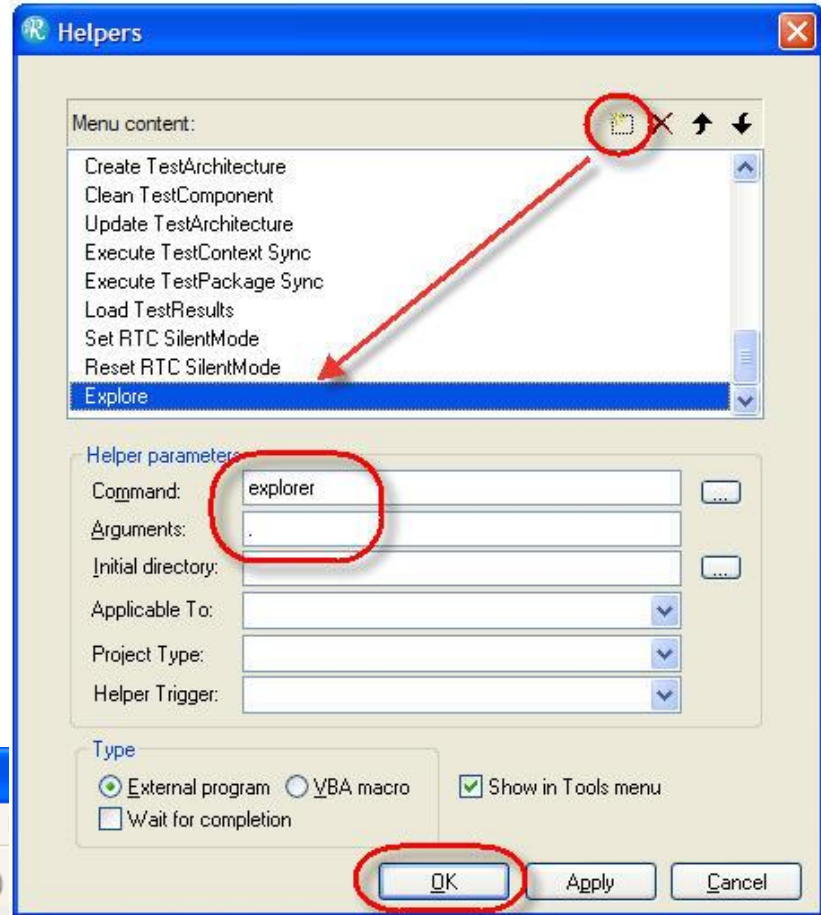
Extended exercise

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



Customize

- Click  to enter a new entry Explore to the **Tools** menu.
- Set the **Command** to explorer.
- Set **Arguments** to .
- Click **OK**.
- Select **Tools > Explore**.





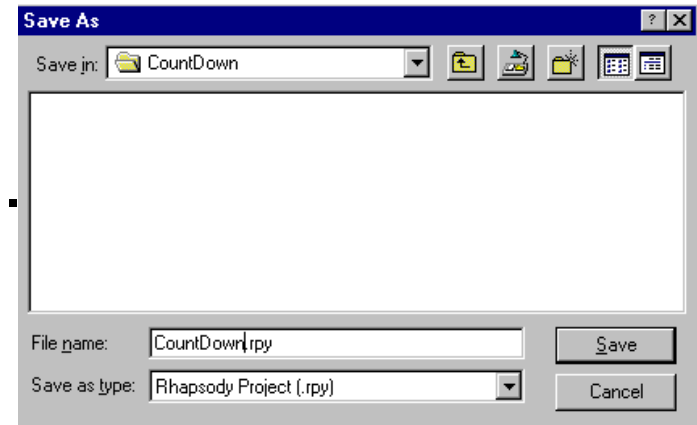
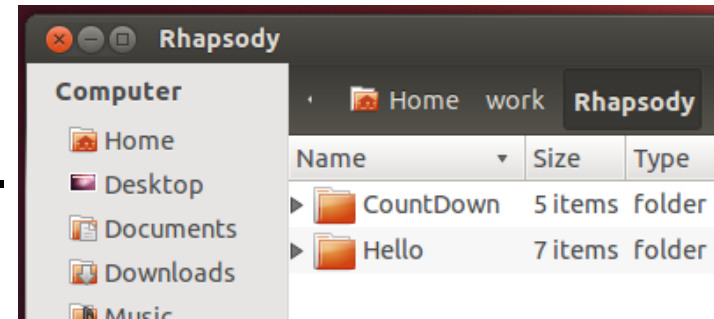
Exercise 2: Count down



```
Test
Constructed
Started
Count = 10
Count = 9
Count = 8
Count = 7
Count = 6
Count = 5
Count = 4
Count = 3
Count = 2
Count = 1
Count = 0
Done
█
```



Copying a project

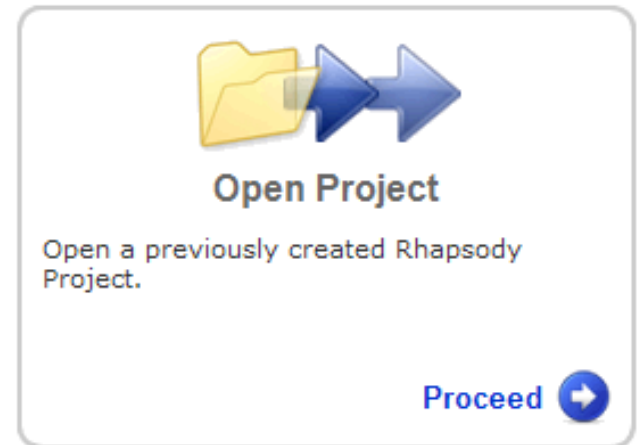
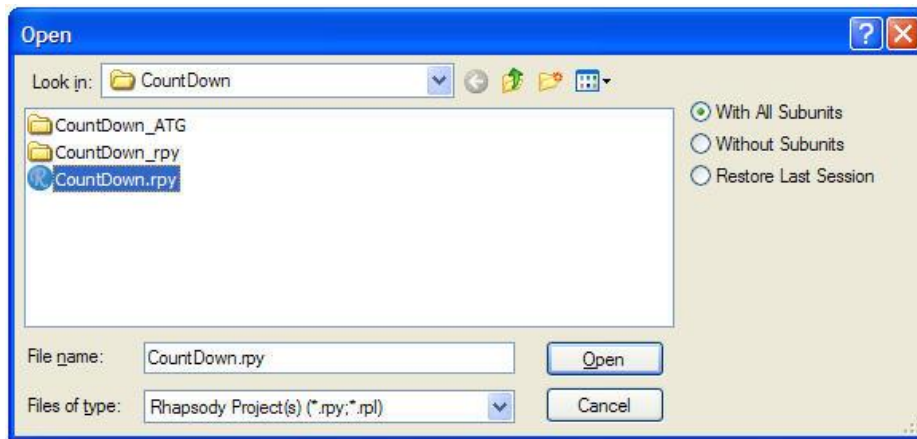
- Select **File > Save As.**
- Press  to select the work folder.
- Press  to create a new folder.
- Rename New Folder to Countdown.
- Select the new folder Countdown.
- Save the project as Countdown.rpy.
- The new Countdown project is opened in Rational Rhapsody with the previous workspace project.



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

Loading a project

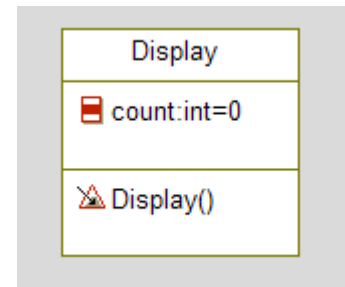
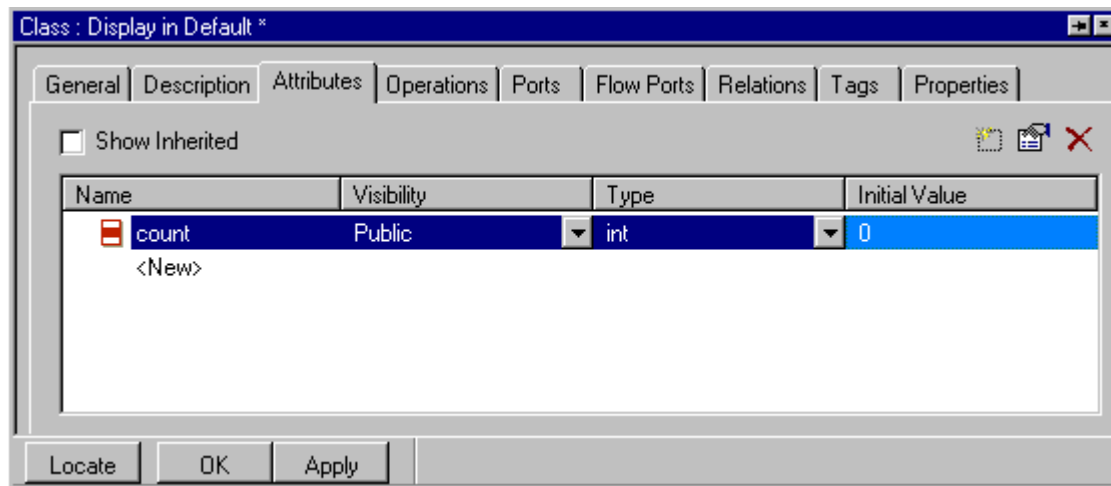
- Choose one of the following ways to open a project:
 - ▶ Start Rational Rhapsody and select **File > Open**.
 - ▶ Or double-click on the  Countdown.rpy file.
 - ▶ Or start Rational Rhapsody and drag the  Countdown.rpy file into Rational Rhapsody.
 - ▶ Or use **Open Project** in the Welcome screen.




The Rhapsody.ini file determines which Rational Rhapsody (C / C++ / J / Ada) will be opened on double-clicking the .rpy file.

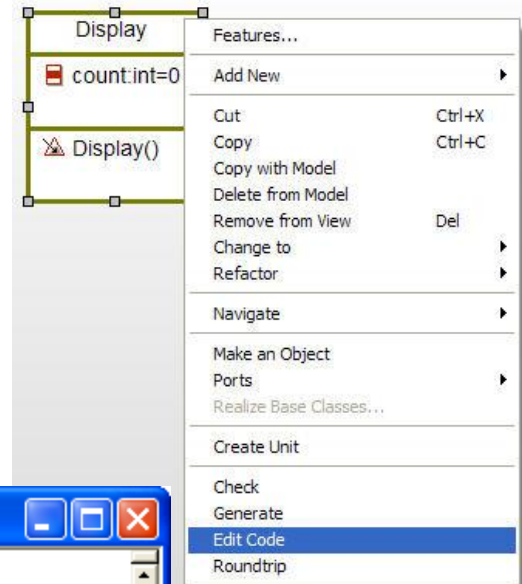
Adding an attribute

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



Generated code for an attribute

- Click **Save**  then edit the code for the *Display* class so you can examine the code.



```
Display.h

public :

    ///# operation Display()
    Display();

    ///# auto_generated
    ~Display();

    ///# auto_generated
    int getCount() const;

    ///# auto_generated
    void setCount(int p_count);

protected :

    int count;    ///# attribute count

};
```

Protected attribute

```
Display.cpp

///# class Display
Display::Display() : count(0) {
    ///# operation Display()
    cout << "Constructed" << endl;
    ///#
}

Display::~~Display() {
}

int Display::getCount() const {
    return count;
}

void Display::setCount(int p_count) {
    count = p_count;
}
```

Accessor

Initial Value

Mutator

What are accessors and mutators?

- By default, all attribute data members in Rational Rhapsody are protected.
- If other classes need access to these attributes, then they must use an Accessor, for example, *getCount()* or Mutator, for example, *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 would just need to modify the accessor and mutator.
- In most cases, attributes do not need accessors or mutators; you will see later how to stop them being generated.

Attribute visibility

- Changing the Visibility in the Attribute features dialog changes the mutator and accessor visibility (not the data member visibility).

Attribute: count in Display

General Description Relations Tags Properties

Name: count

Stereotype:

Attribute type

Use existing type

Type: int

Visibility

Public Protected Private

Locate OK Apply

Attribute: count in Display *

General Description Relations Tags Properties

Name: count

Stereotype:

Attribute type

Use existing type

Type: int

Visibility

Public Protected Private

Locate OK Apply

Attribute: count in Display *

General Description Relations Tags Properties

Name: count

Stereotype:

Attribute type

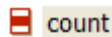
Use existing type

Type: int

Visibility

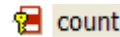
Public Protected Private

Locate OK Apply



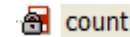
count

```
public :  
  
    ///# auto_generated  
    int getCount() const;  
  
    ///# auto_generated  
    void setCount(int p_count);  
  
//// Attributes ////  
protected :  
  
    int count;    ///# attribute count
```



count

```
protected :  
  
    ///# auto_generated  
    int getCount() const;  
  
    ///# auto_generated  
    void setCount(int p_count);  
  
//// Attributes ////  
protected :  
  
    int count;    ///# attribute count
```

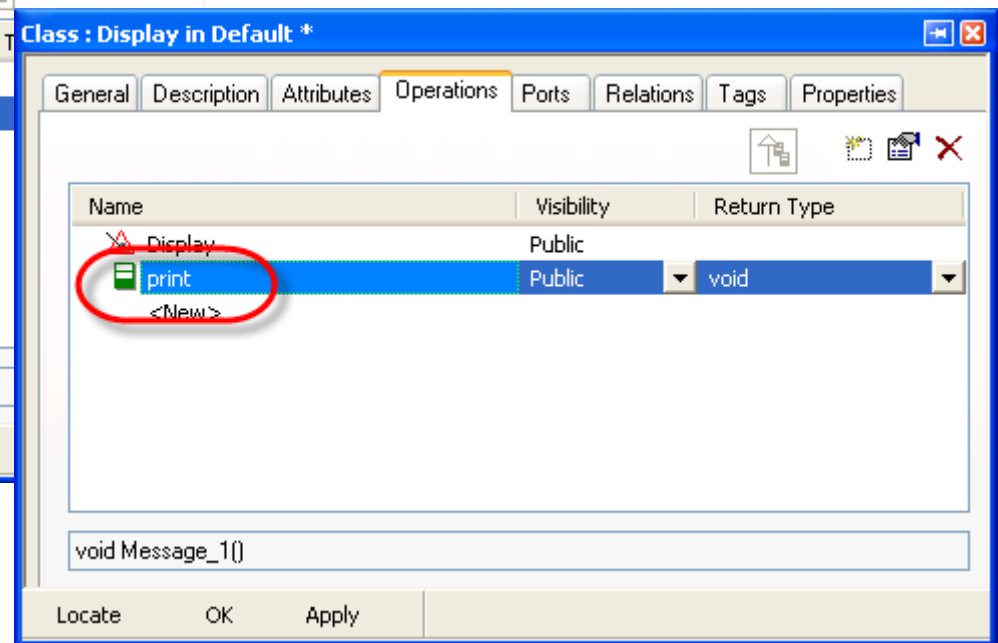
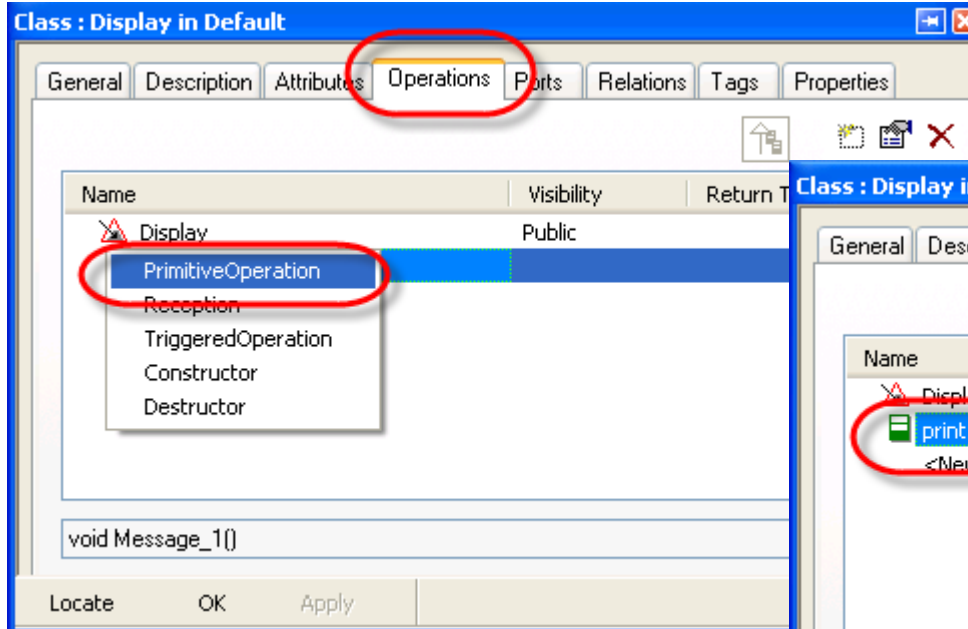


count

```
private :  
  
    ///# auto_generated  
    int getCount() const;  
  
    ///# auto_generated  
    void setCount(int p_count);  
  
//// Attributes ////  
protected :  
  
    int count;    ///# attribute count
```

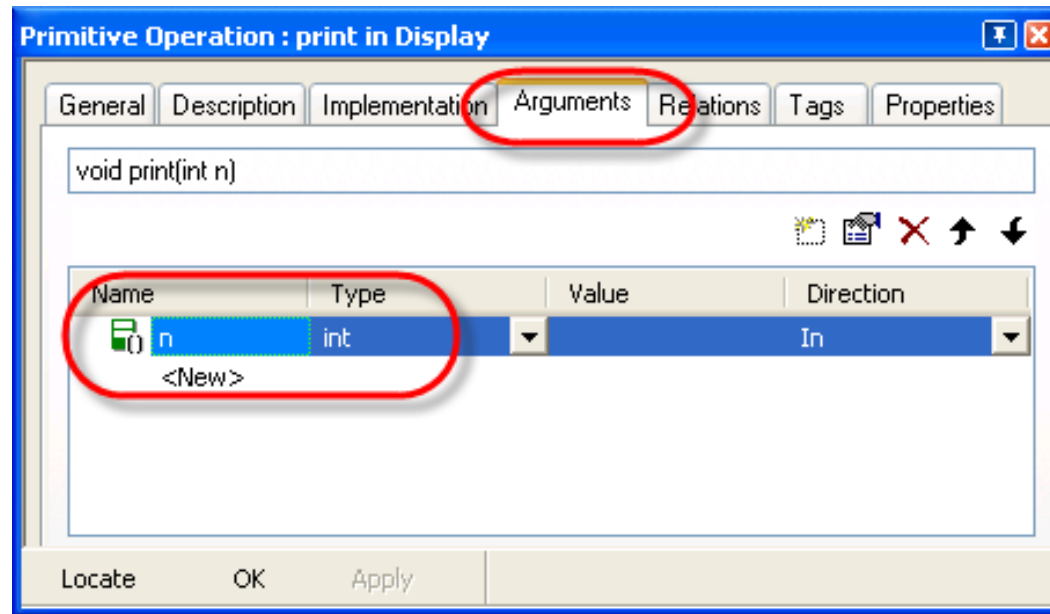
Adding an operation

- Using the features for the *Display* class, select the **Operations** tab > **Primitive Operation**.
- Add a new primitive operation called *print*.



Arguments

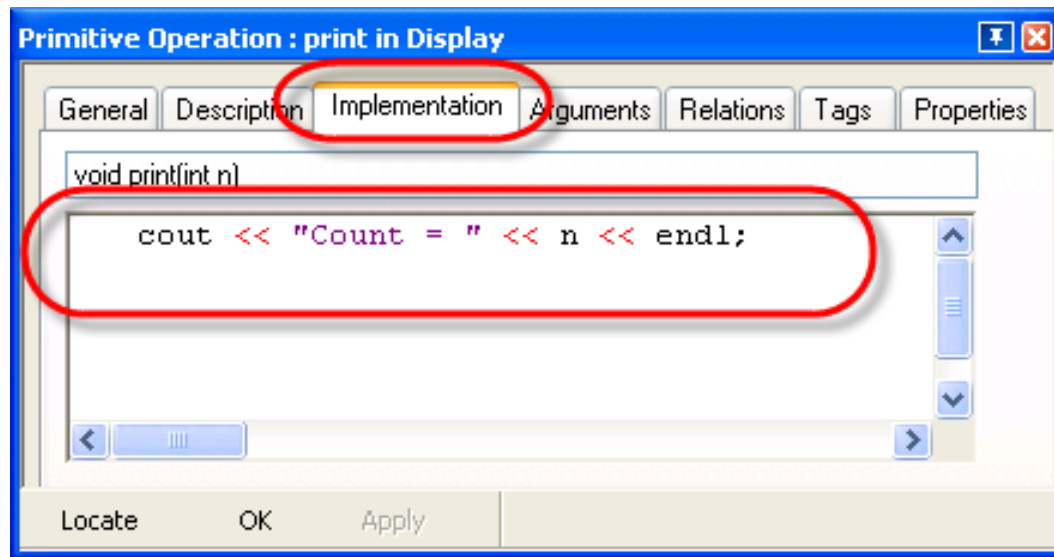
- Double-click **Print** to open the features for the print operation.
- Select the **Arguments** tab.
- Add an argument *n* of type *int*.



Adding implementation

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

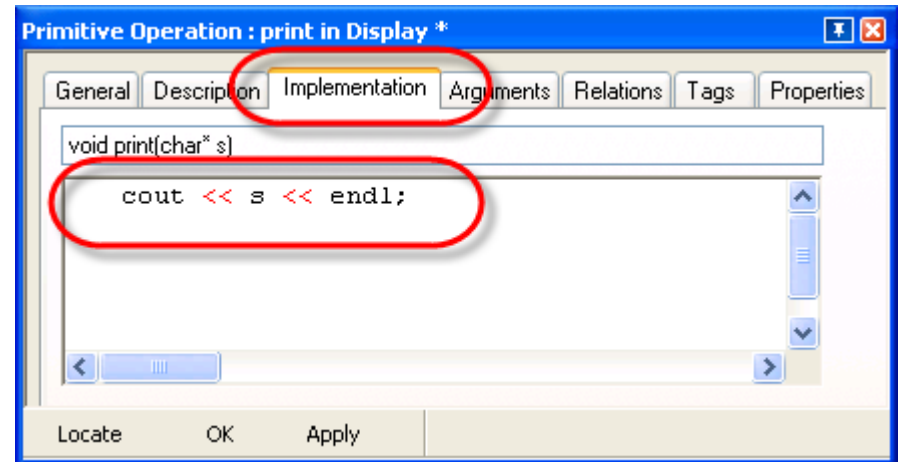
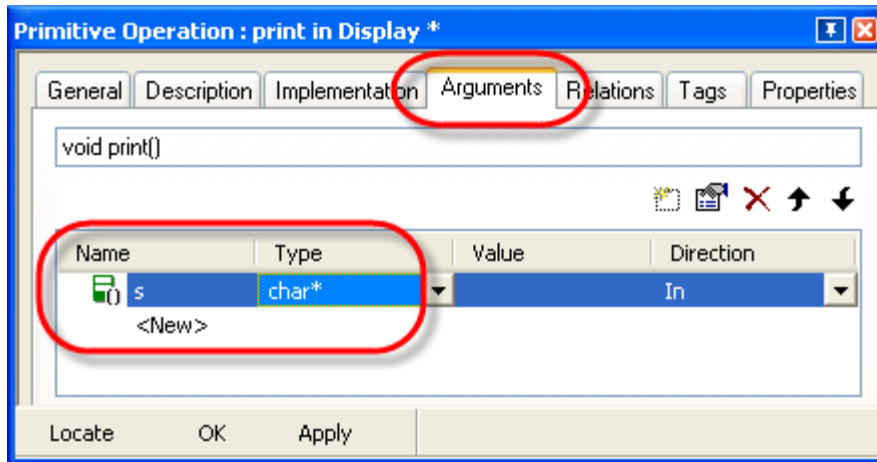
```
cout << "Count = " << n << endl;
```



Another print operation

- In a similar way, add another operation called *print*, this time with an argument *s* of type *char** and with implementation:

```
cout << s << endl;
```



Set the argument type before setting the name. This avoids a conflict where the two print operations have identical signatures.

Operation isDone()

- Add another operation called *isDone* that returns a *bool* and has the following implementation:

```
return (0==count) ;
```

The image shows two overlapping windows from an IDE. The left window, titled "Class : Display in Default *", has the "Operations" tab selected. It displays a table of operations:

Name	Visibility	Return Type
Display	Public	
print	Public	void
print	Public	void
isDone	Public	bool


The "isDone" row is highlighted. Below the table, the signature "void isDone()" is visible. The right window, titled "Primitive Operation : isDone in Display *", has the "Implementation" tab selected. It shows the implementation of the "isDone" operation:

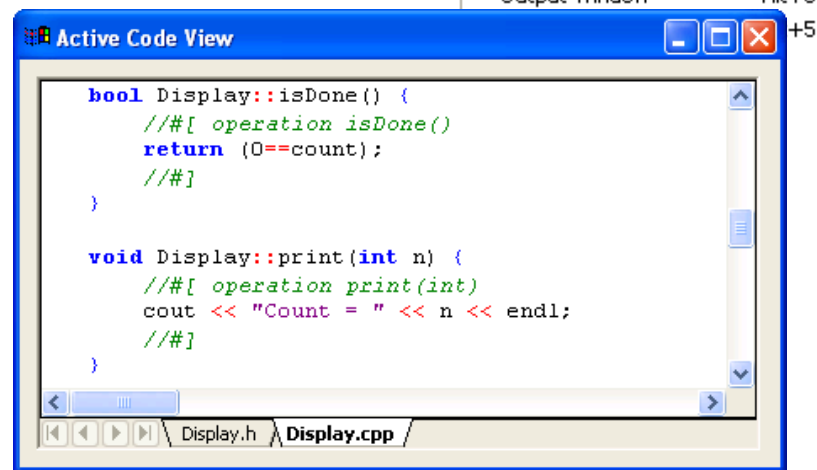
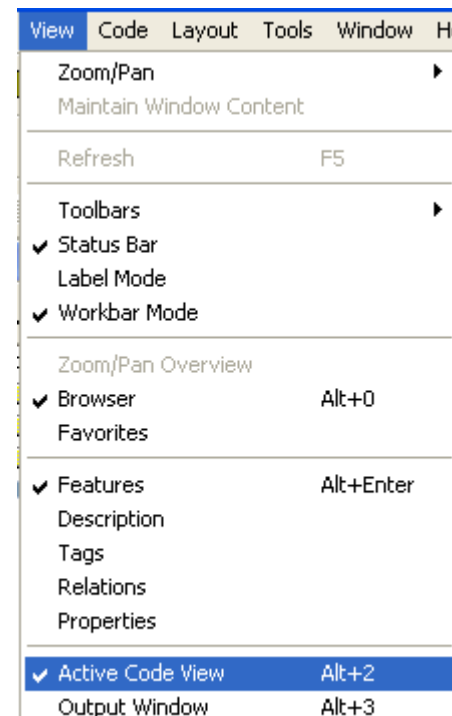
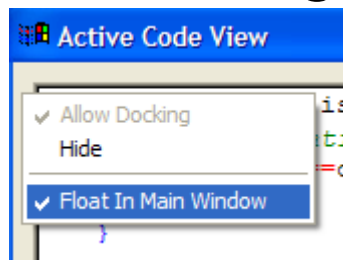
```
bool isDone()  
return (0==count) ;
```

Red circles highlight the "Operations" tab in the left window, the "Implementation" tab in the right window, and the "isDone" row in the table and its implementation code.

By typing `0==count` instead of `count==0`, enables the compiler to detect the common error of where `=` is typed instead of `==`.

Active Code View

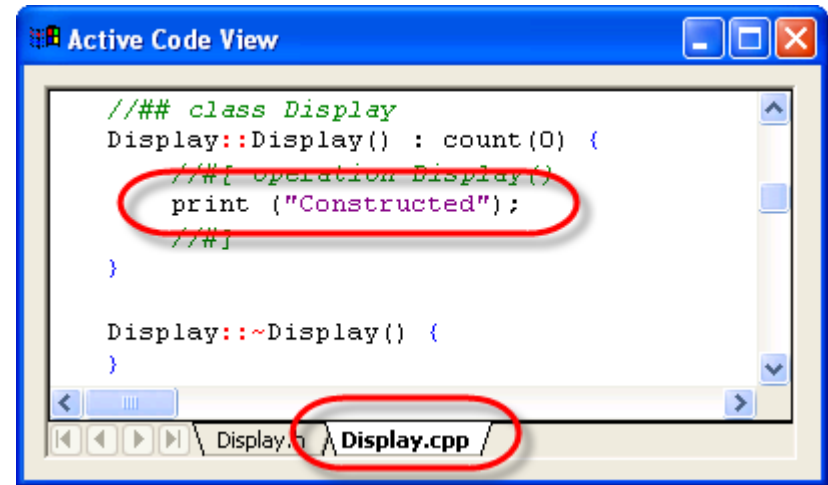
- Select **View > Active Code View**. 
- The active code view is context sensitive and is automatically updated as the model is changed. The window also changes dynamically to show the generated code for the highlighted model element.



Note that although leaving the active code view always open is useful, it does slow down model manipulation since the code will get regenerated anytime any model element gets modified.

Using the print operation

- In the Active Code View, change the code for the *constructor* to use the *print* operation.
 - ▶ Make sure you have selected the Implementation.





```
Active Code View

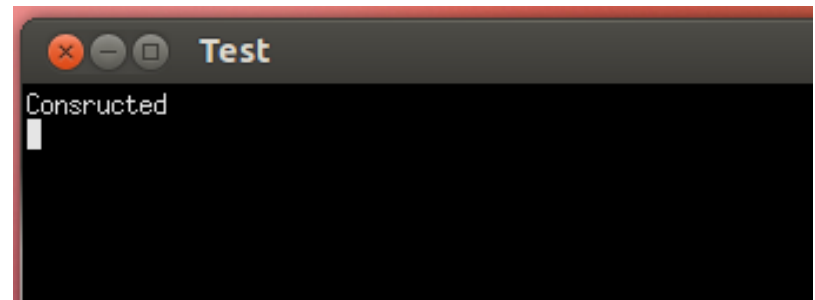
/// class Display
Display::Display() : count(0) {
    /// Operation Display()
    print ("Constructed");
    ///
}

Display::~Display() {
}

Display.cpp
```

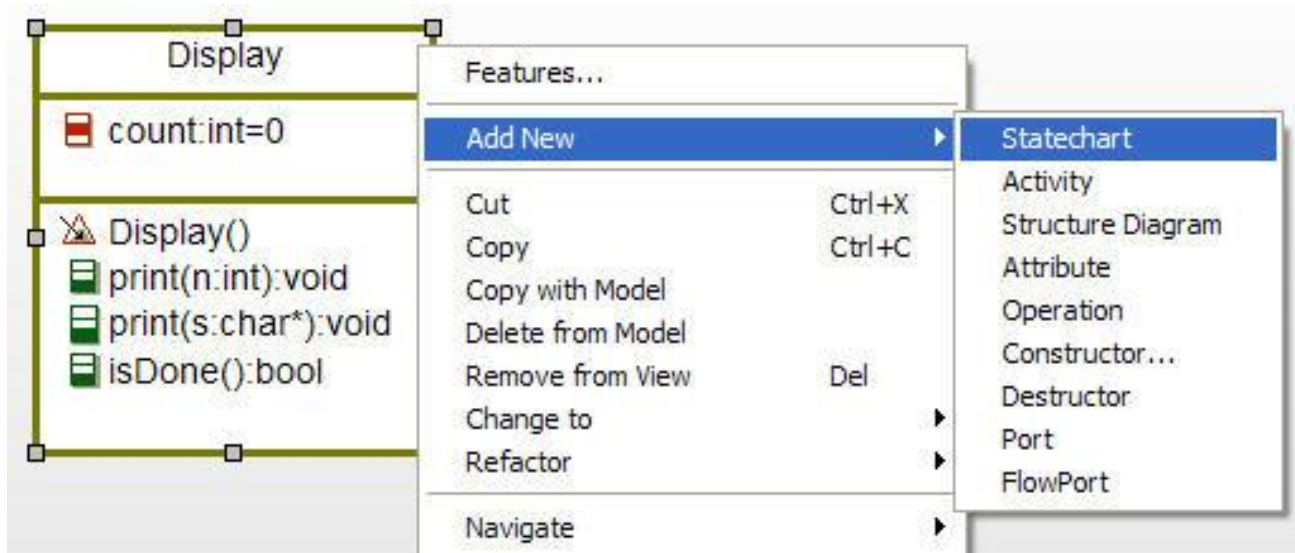
- Change the focus to another window such as the browser and check that this modification has been automatically round-tripped.

- Save the changes. 
- Generate / Make / Run. 

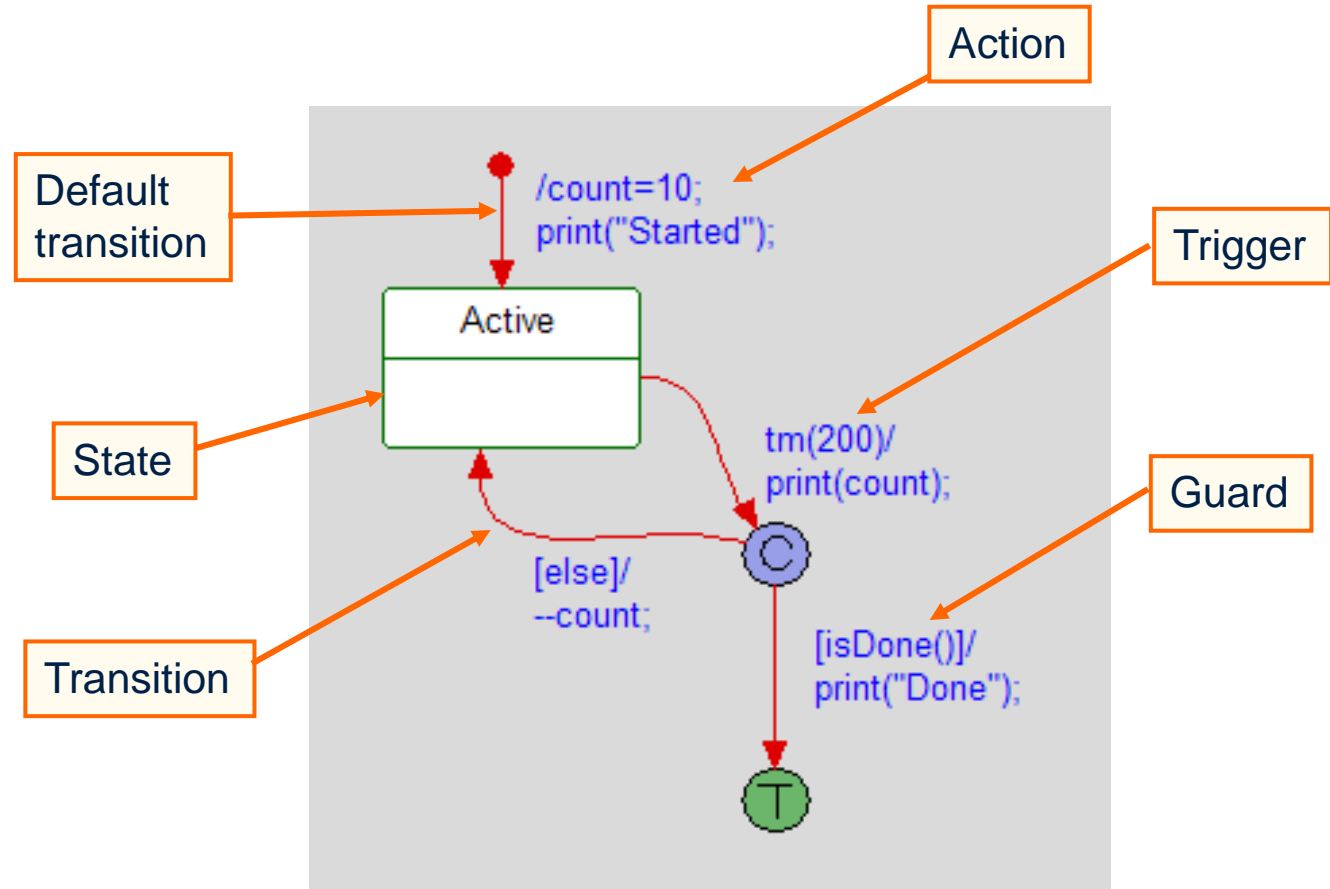
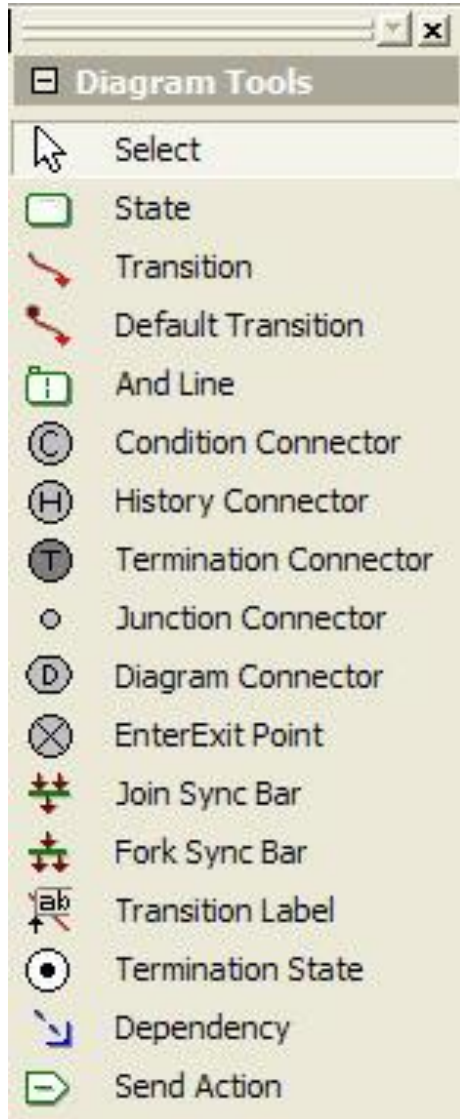


Adding a statechart

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

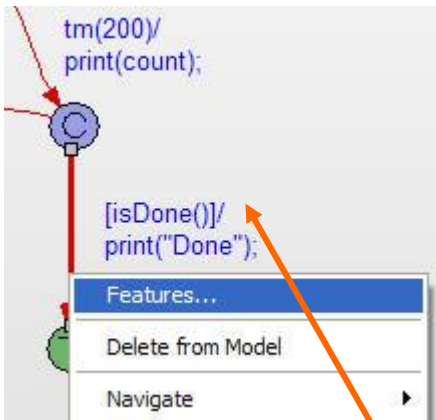


Draw a simple statechart



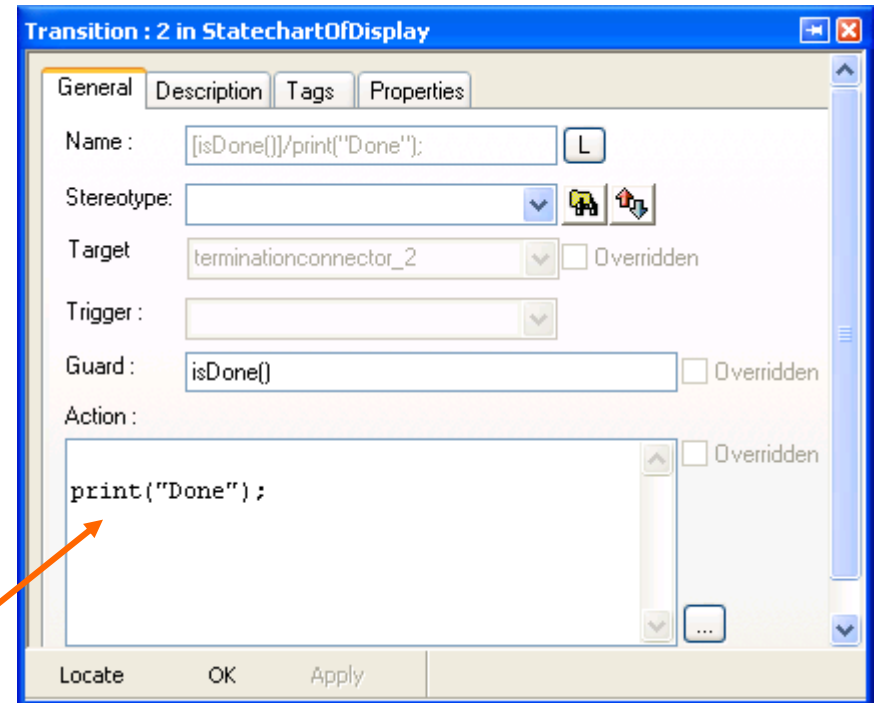
Transitions

- Once a transition has been drawn, there are two ways in which to enter information:
 - In text format - example: `[isDone()]/print("Done");`
 - By the features of the transition (activated by double-clicking or right-clicking on the transition).



Ctrl+Enter closes the entry field.

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



Timer mechanism

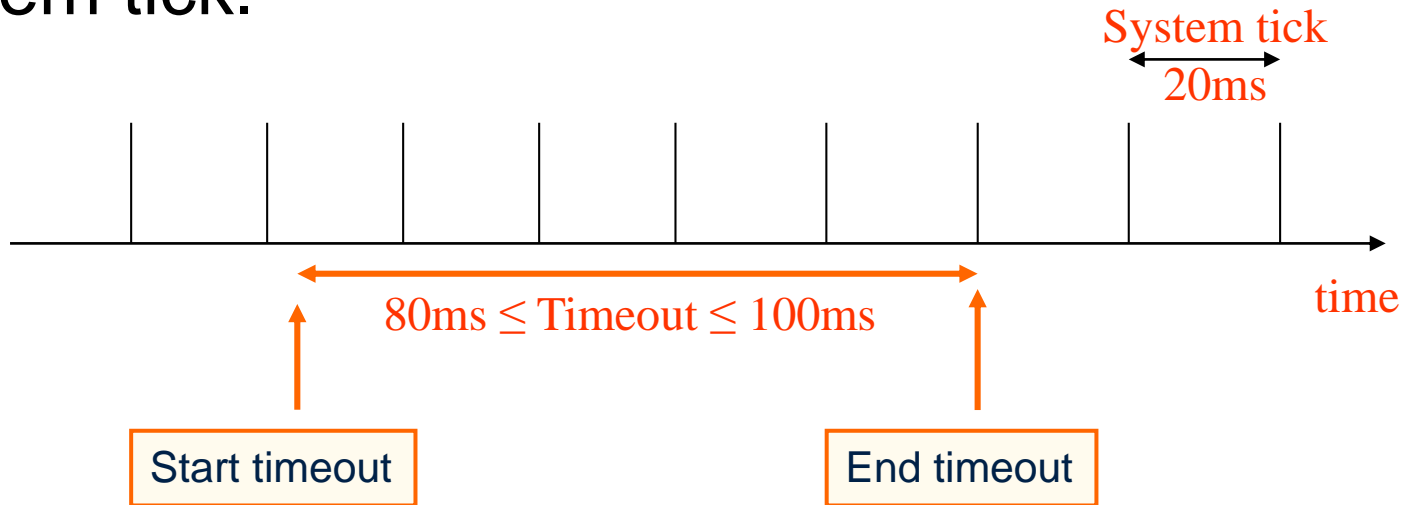
- A timer is provided that can be used within the statecharts.
- `tm(200)` acts as an event that will be taken 200ms after the state has been entered.
- When entering into the state, the timer will be started.
- When exiting from the state, the timer will be stopped.

```
tm(200)/  
print(count);
```

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

Timeouts

- If you have 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.



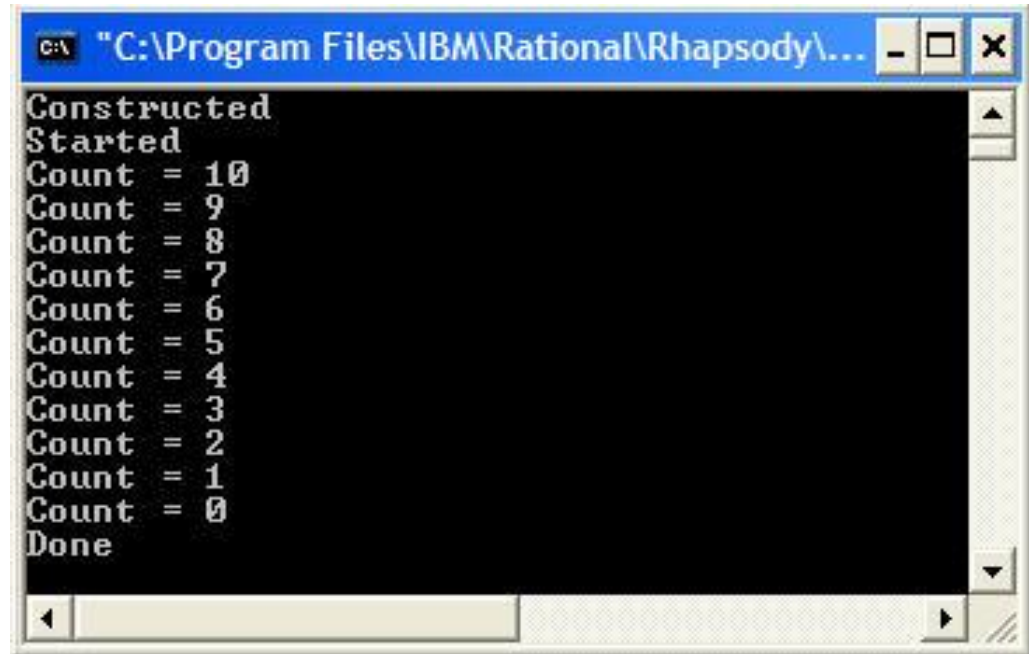
If precise timeouts are required, then it is recommended you use a hardware timer in combination with triggered operations.

Counting down

- Save 
- Generate / Make / Run 

Constructor

Default Transition





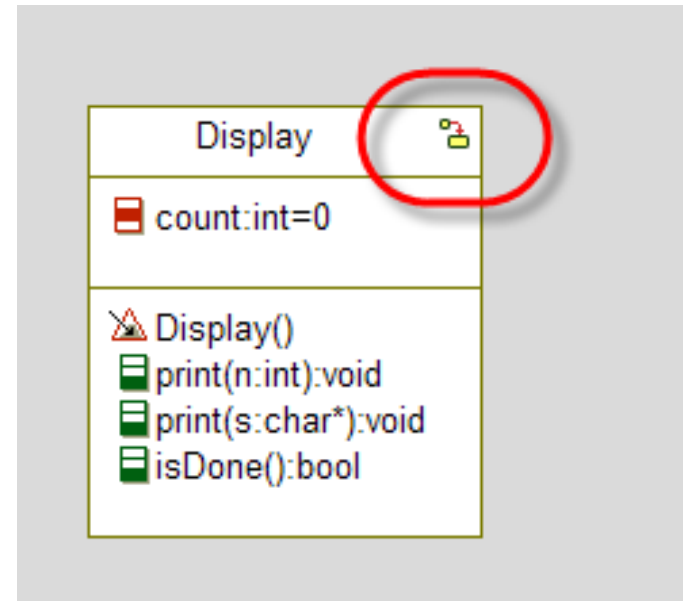
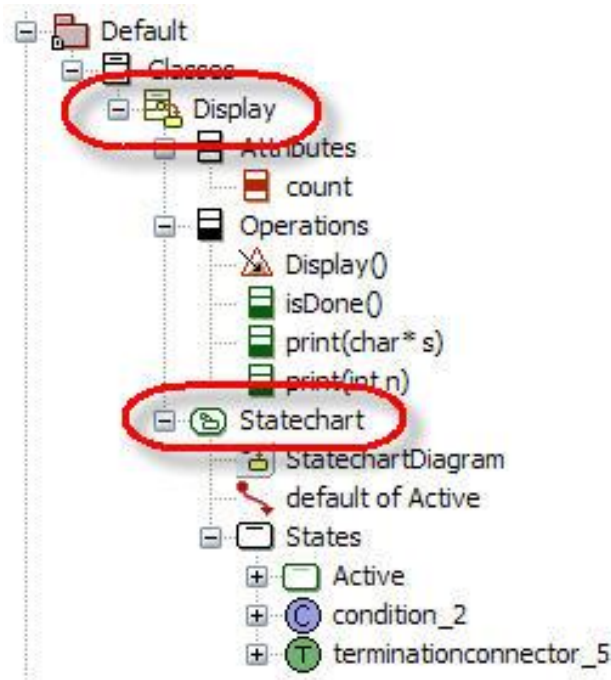
```
C:\Program Files\IBM\Rational\Rhapsody\...  
Constructed  
Started  
Count = 10  
Count = 9  
Count = 8  
Count = 7  
Count = 6  
Count = 5  
Count = 4  
Count = 3  
Count = 2  
Count = 1  
Count = 0  
Done
```



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

Statechart symbol

- Now that the *Display* class is *Reactive*
 - ▶ A reactive class is one that reacts to receiving events or timeouts.
 - ▶ Identified by symbol in the browser  and the OMD. 
- Also note that the Statechart appears in the browser.



Generated code: display.h

- Use the **Active Code View** to examine the generated code for the *Display* class.

```
Active Code View
//## auto_generated
#include <oxf\oxf.h>
//## auto_generated
#include <oxf\omreactive.h>
//## auto_generated
#include <oxf\state.h>
//## auto_generated
#include <oxf\event.h>
//## package Default

//## class Display
class Display : public OMReactive {
    /// Constructors and destructors    ///
public :

    //## operation Display()
    Display(IoxfActive theActiveContext = 0);
```

Framework class

Framework includes

Thread on which to wait

Note that the *Display* class inherits from *OMReactive*, which is one of the framework base classes. This is a class that simply waits for timeouts or events. When it receives a timeout or an event, it calls the `rootState_processEvent()` operation.

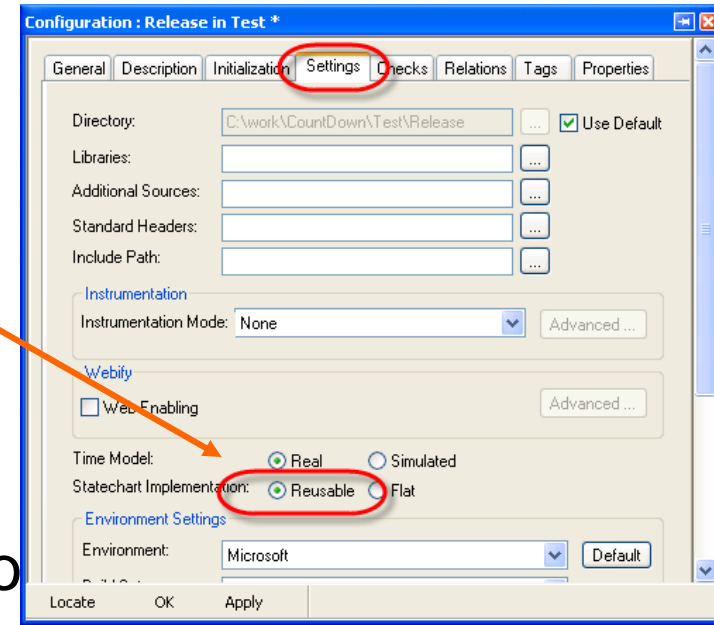
Generated code: display.cpp

- `Display::Display(IOxfActive* theActiveContext)`
 - ▶ The constructor needs to know on which thread to wait.
- `Display::initStatechart()`
 - ▶ Called by the constructor to initialize the attributes used to manage the Statechart.
- `Display::startBehavior()`
 - ▶ Kicks off the behavior of the Statechart, invokes the `rootState_entDef()` via OXF calling `OMReactive::startBehavior()`.
Typically invoked from outside after construction completed.
- `Display::rootState_entDef()`
 - ▶ Called by `OMReactive::startBehavior()` to take the initial default transition.
- `Display::rootState_processEvent()`
 - ▶ Called through OXF operation `OMReactive::processEvent()` whenever the object receives an event or timeout.

Statechart implementation

- Change the statechart implementation

- ▶ Select the features for the configuration *Release*.
- ▶ Select the Settings tab and set Statechart Implementation from *Flat* to *Reusable*.
- ▶ Save / Generate / Examine code.

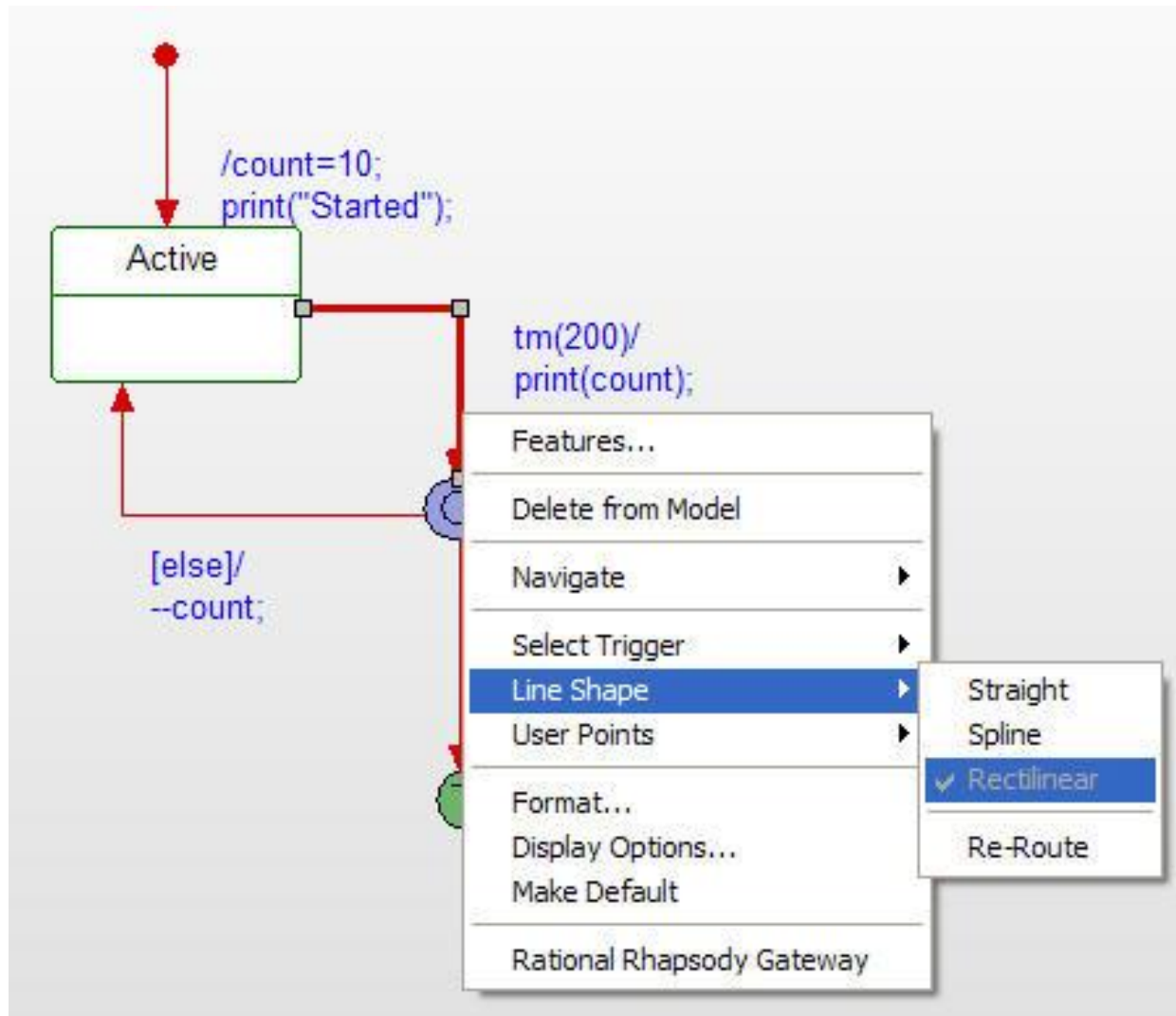


- The Rational Rhapsody framework allows two ways of implementing statecharts:

- ▶ *Reusable* is based on the state design pattern where each state is an object.
 - Results in faster execution and if a lot of statecharts are inherited, can result in smaller code.
- ▶ *Flat* uses a switch statement.
 - Results in less code that is easier to read, but is slower.

Extended exercise

- Experiment with the line shape of transitions.

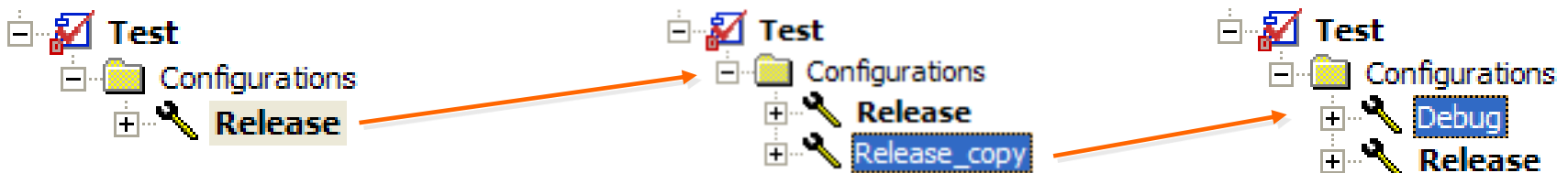
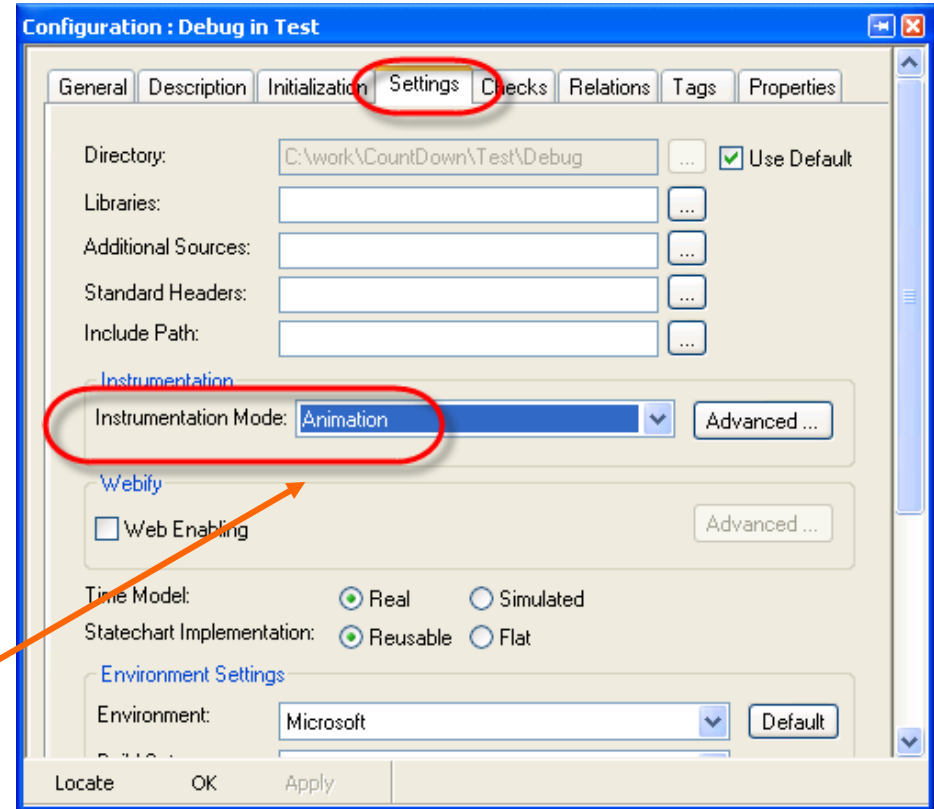


Design level debugging

- Up to 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, this is known as *Animation*.

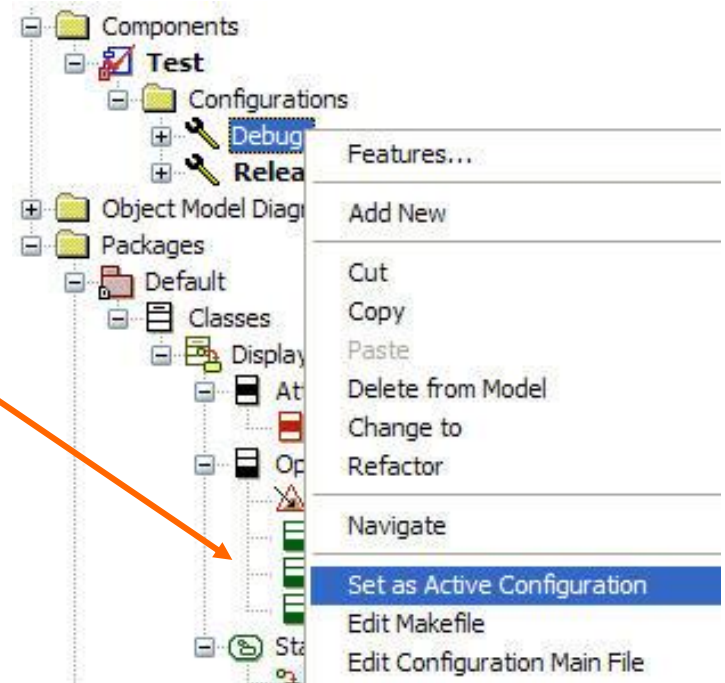
Animation

- Create a configuration by copying the *Release* configuration:
 - ▶ Press **Ctrl** and drag the *Release* configuration onto the **Configurations** folder.
 - ▶ Rename the new configuration *Debug*.
- Set the **Instrumentation Mode** to *Animation*.





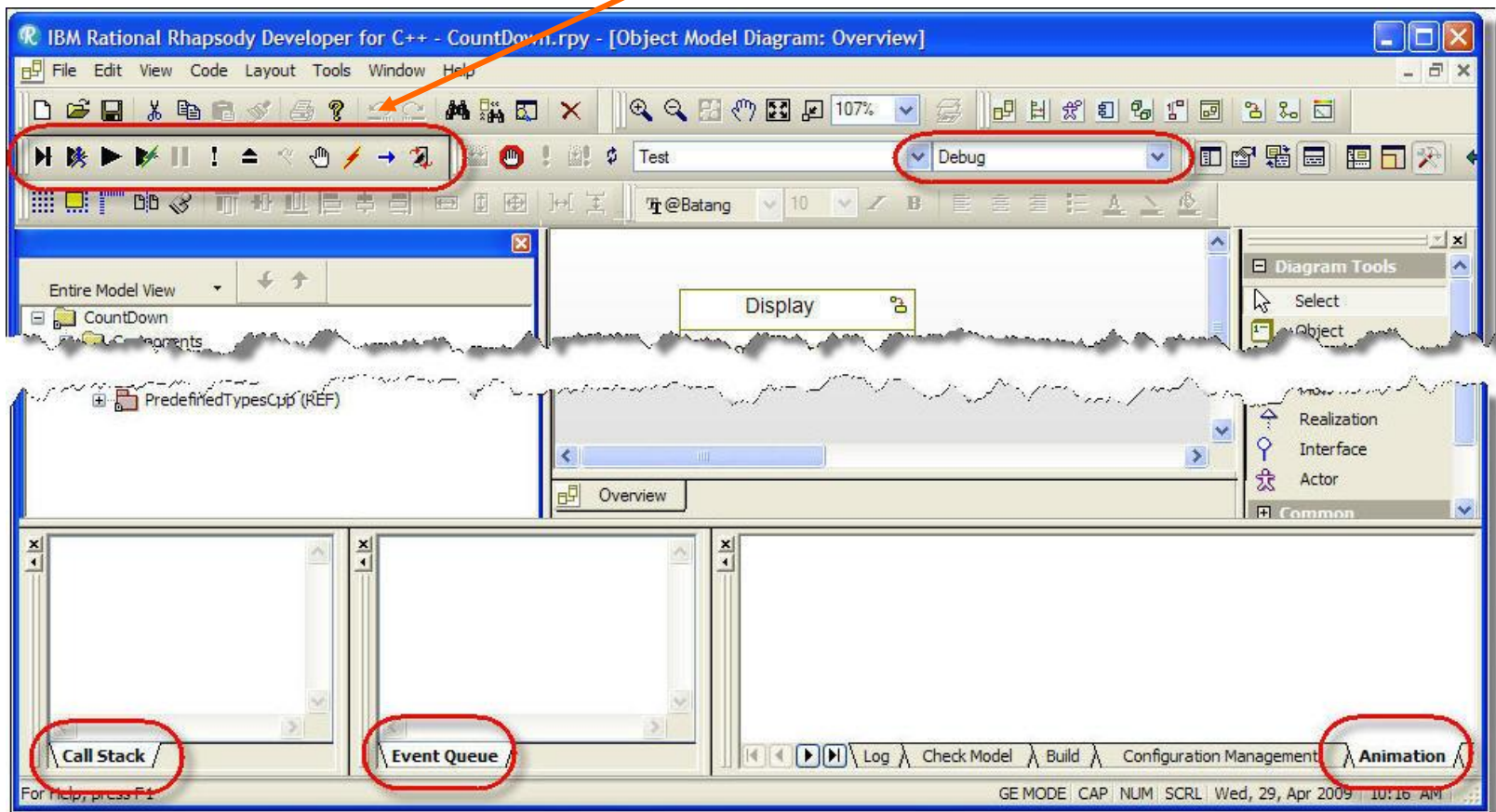
Multiple configurations

- Now that you have more than one configuration, you must select which one you want to use.
- There are two methods:
 - ▶ Right-click the configuration and select **Set as Active Configuration**.
 - ▶ Select the configuration using the pull-down box.



Animating

- Make sure the active configuration is *Debug* before doing Save  then Generate / Make / Run. !
- Run will cause the Animation toolbar to be displayed.





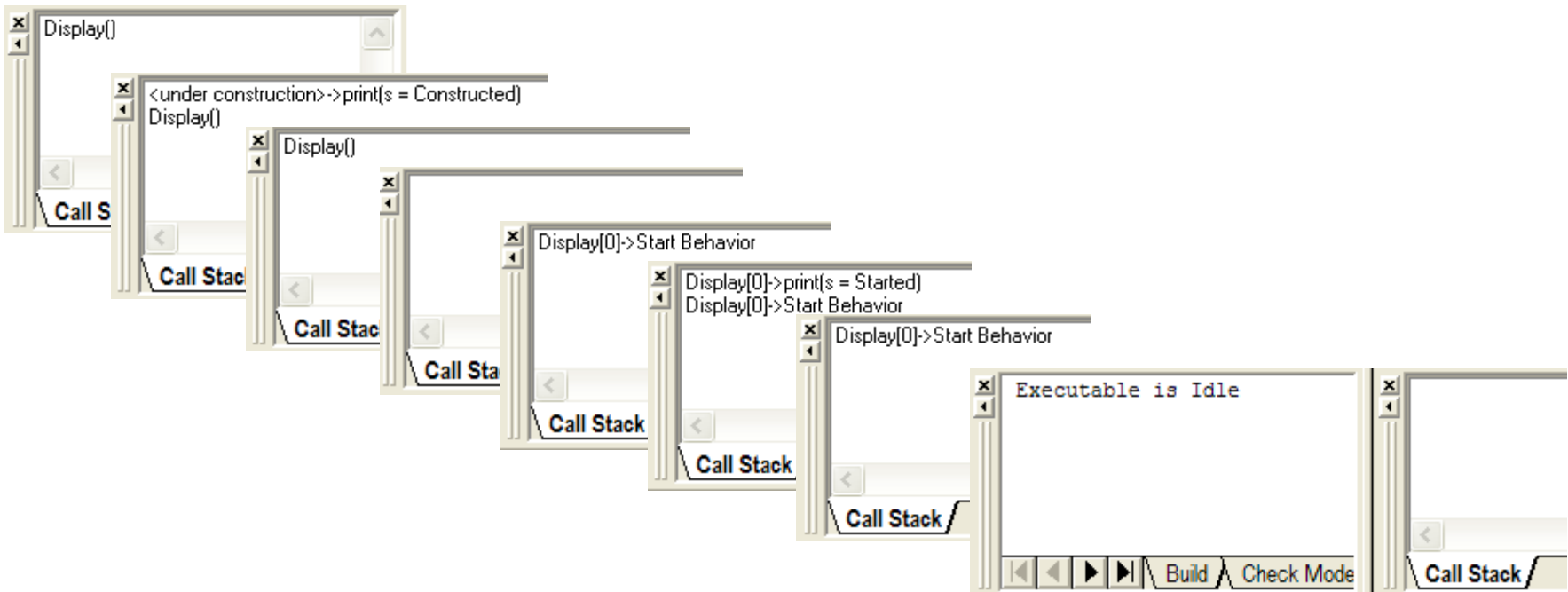
Animation Toolbar

- Automatically appears when an executable model is run and instrumentation is set to **Animation**.
 - ▶ To display or hide during animation session, select **View > Toolbars > Animation**.
- For detailed button information, select **Help > Help Topics** and search on `animation toolbar`.
 - ▶ For example, grayed out (disabled) **Thread** button indicates single-threaded application.



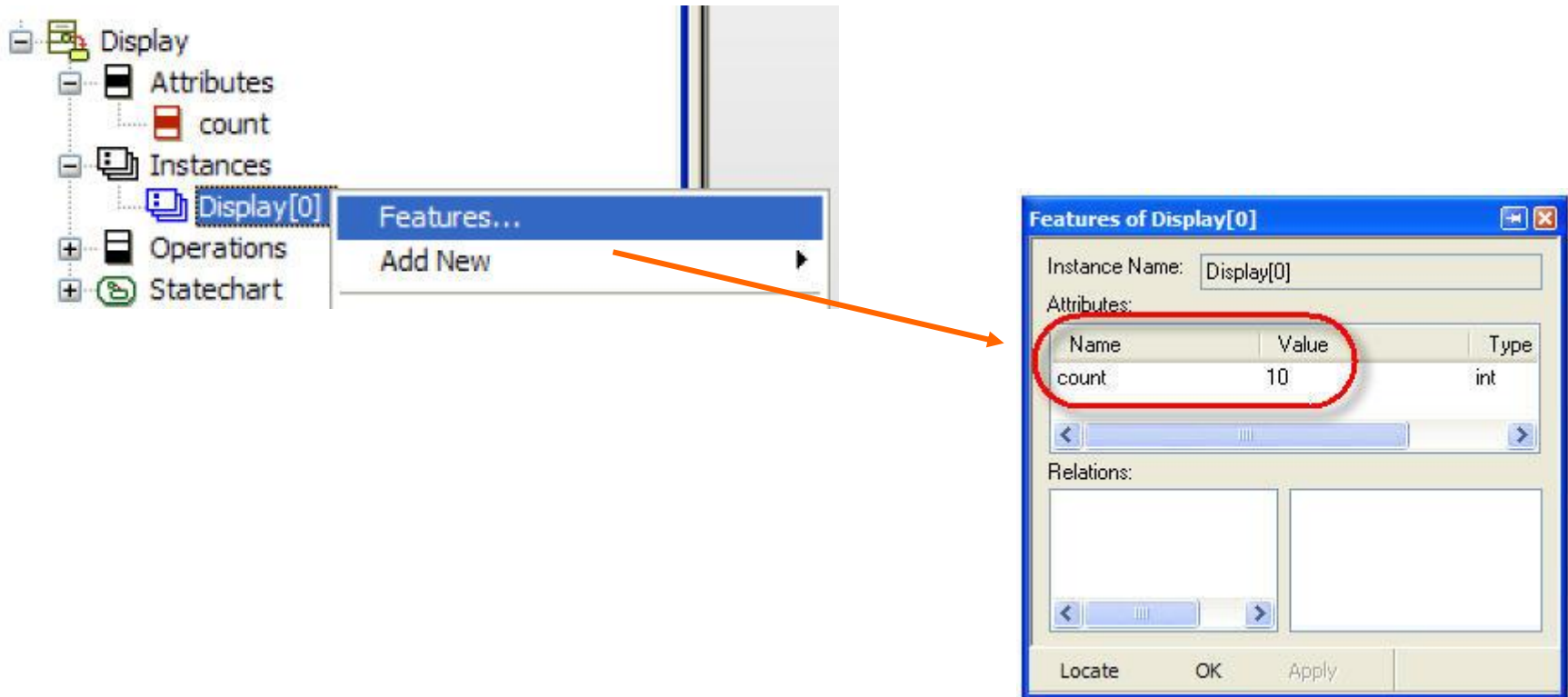
Starting the animation

- Click the Go Step  button.
 - ▶ Note that the *Display()* constructor appears in the Call Stack.
- Continue to Go Step  until the *Executable is Idle* message appears in the Animation window.



Class Instance

- Browser contains an instance of the *Display* class.
 - ▶ Right-click the instance and select **Features**.
 - ▶ Note that the attribute *count* has been initialized to 10.



Statechart Instance

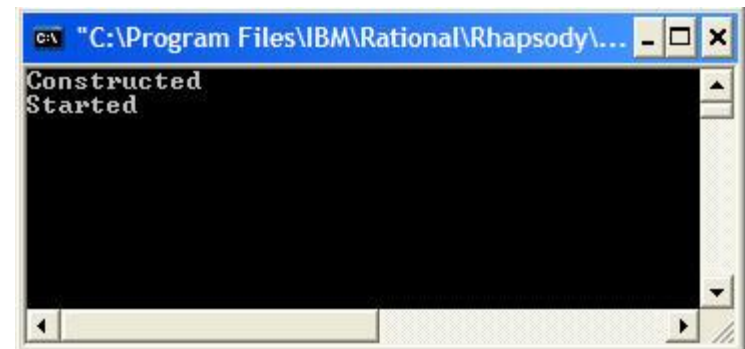
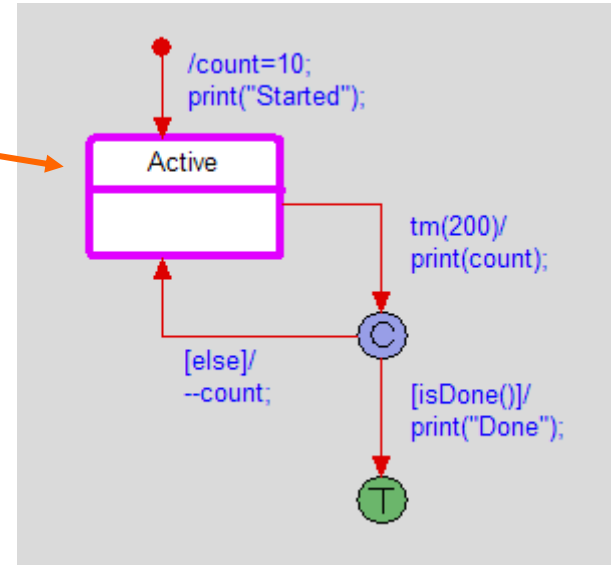
- Right-click the instance and select **Open Instance Statechart**.

- ▶ Highlighted state indicates the current state of the model.




- If you do not see a highlighted state, you may be looking at the statechart of the class (design) rather than the statechart of the instance (runtime).

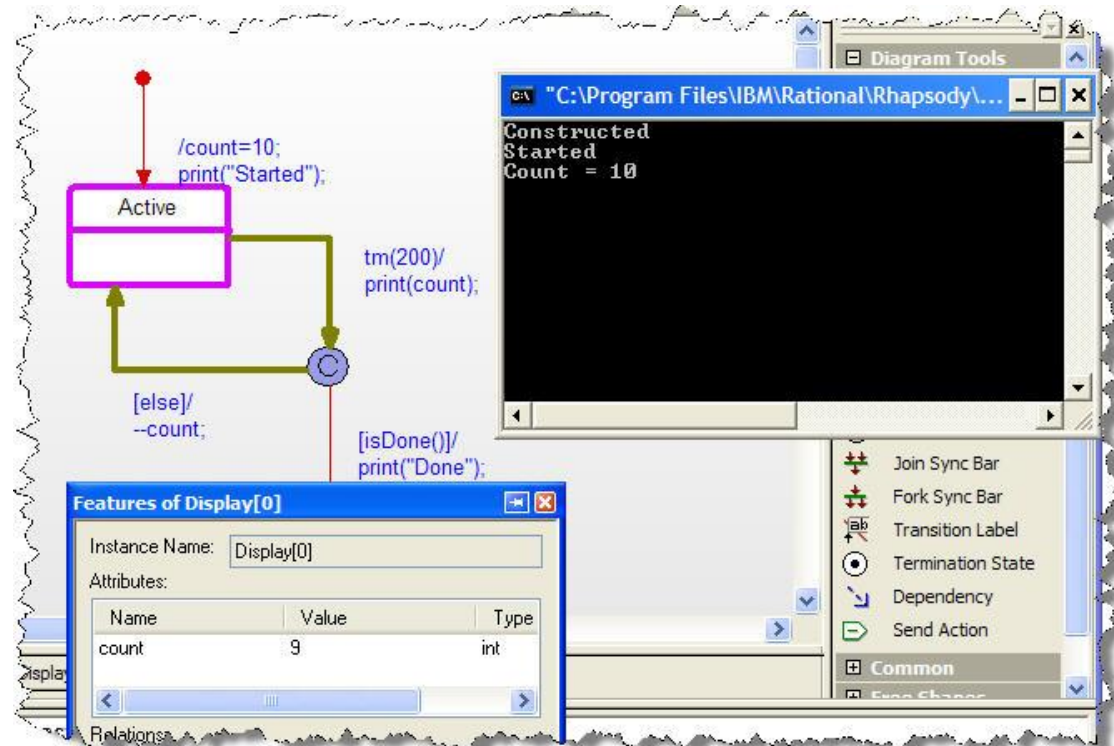
- ▶ Default transition has also been triggered.

- *Started* will have been printed to the display.



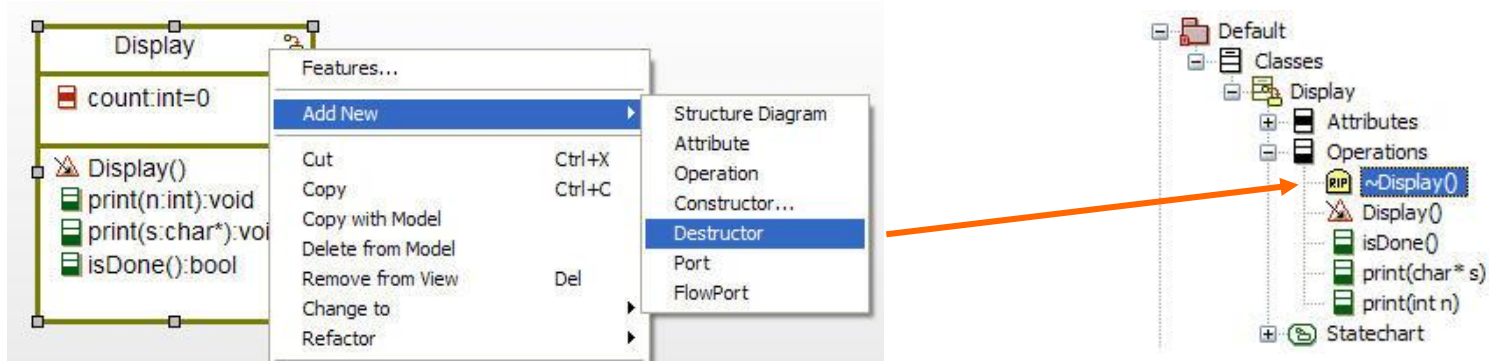
Go Idle / Go

- Click **Go Idle**  to advance to next timeout.
 - ▶ The executed transition chain in statechart is highlighted.
 - Value for count is printed to display.
 - Condition is checked for is done.
 - Not done so value of count is decremented.
- Click **Go**  and watch the animation until the instance is destroyed.
- Exit the animation. 

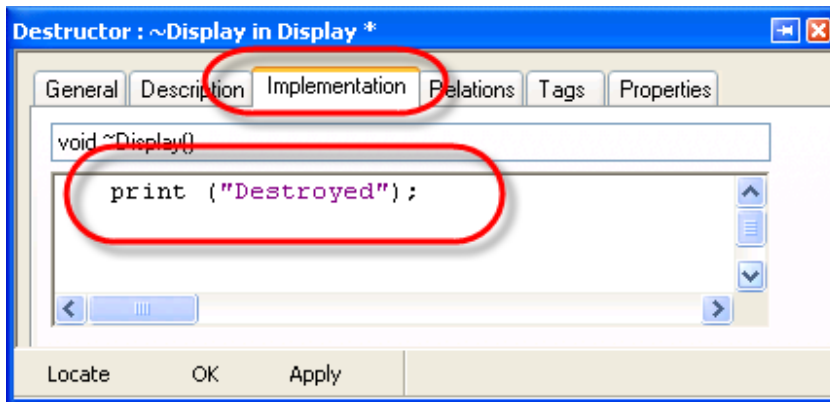


Destructor

- Add a Destructor  to the *Display* class.



- Add implementation `print ("Destroyed") ;`

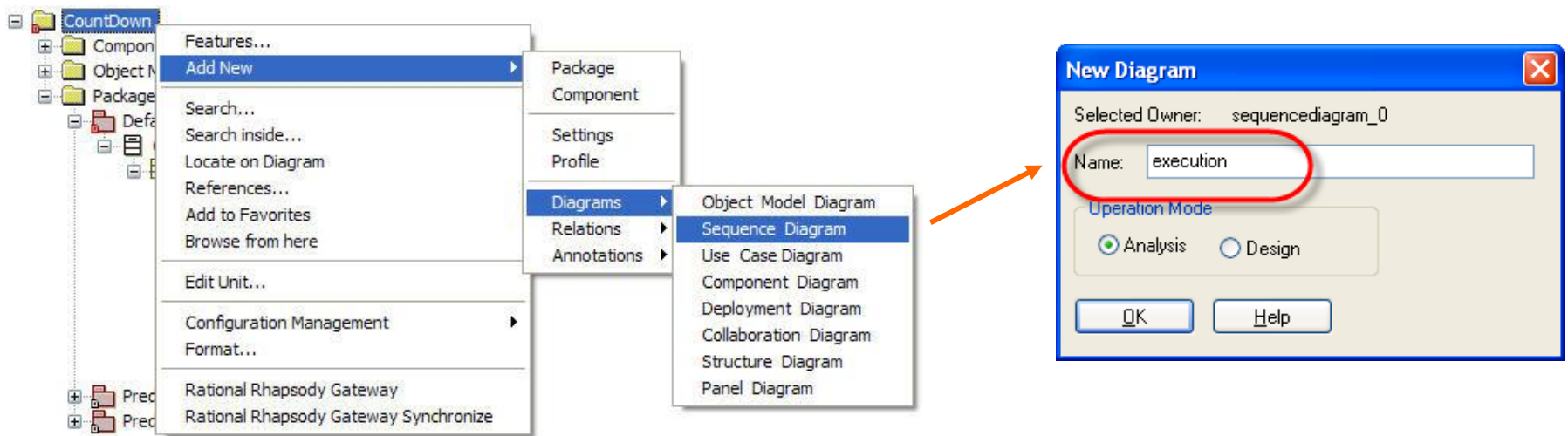


Make sure you enter the code into the Implementation and not the Description field.

- Save  then Generate / Make / Run .

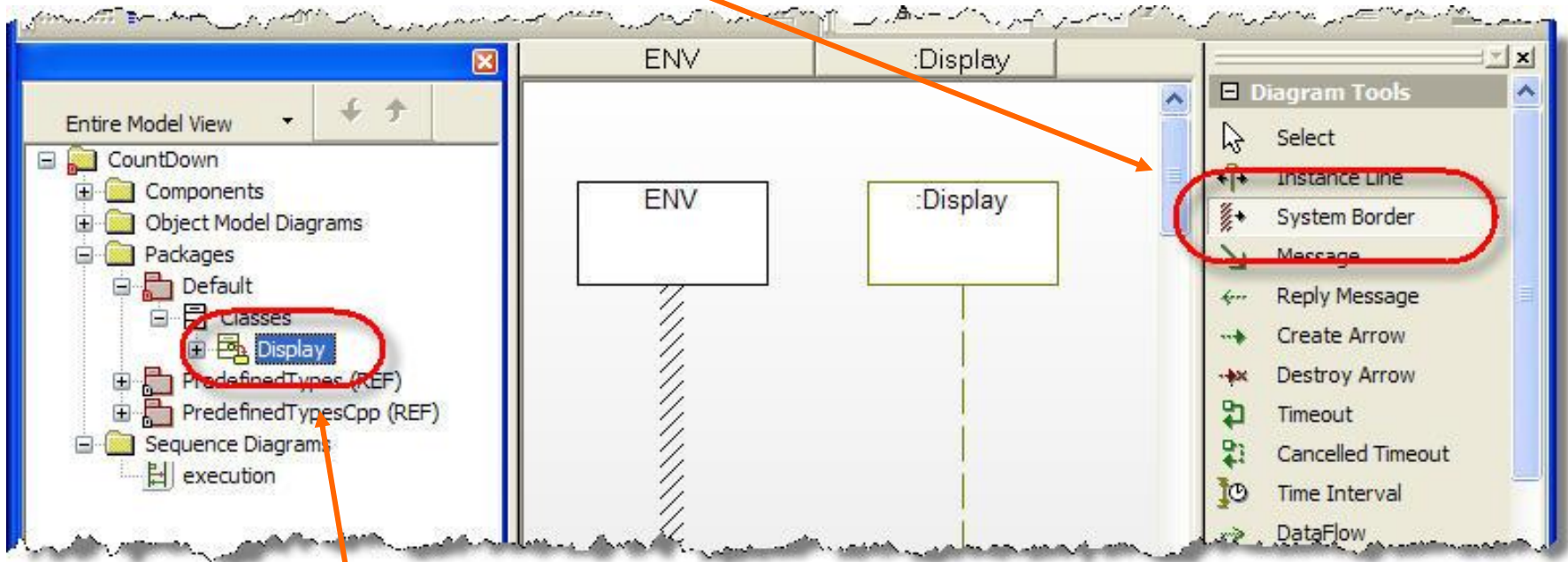
Sequence diagrams

- From the browser, create a new sequence diagram called *execution*.
 - ▶ This sequence diagram will be used to capture what happens in execution.
 - ▶ Operation Mode will be discussed later but for this example, it does not matter if Analysis or Design is selected.



Adding instances

- Add a System Border to the sequence diagram.

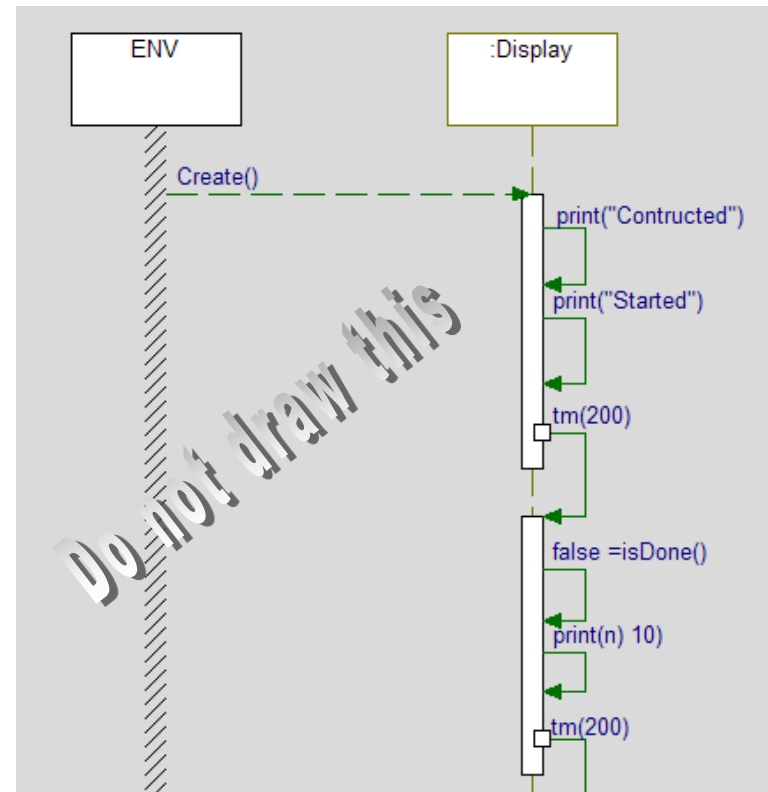


- Drag the *Display* class from the browser onto the sequence diagram.

Drawing a sequence diagram

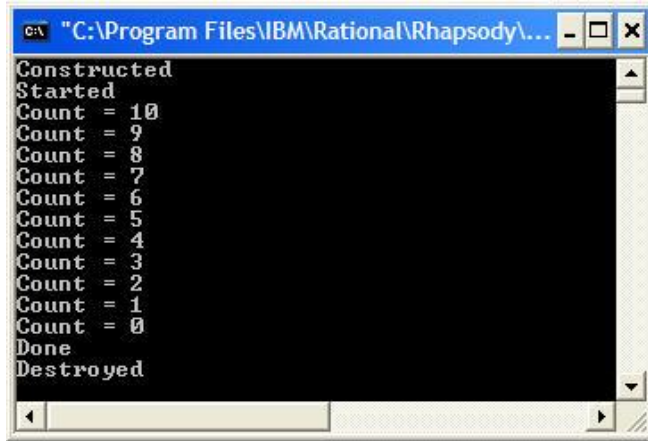
- Normally, you would describe an actual scenario similar to this one here, however in this case, you are more interested in capturing what actually happens.

For the purpose of this training, you only need the system border and the Display instance line. There is no need to add any operations.



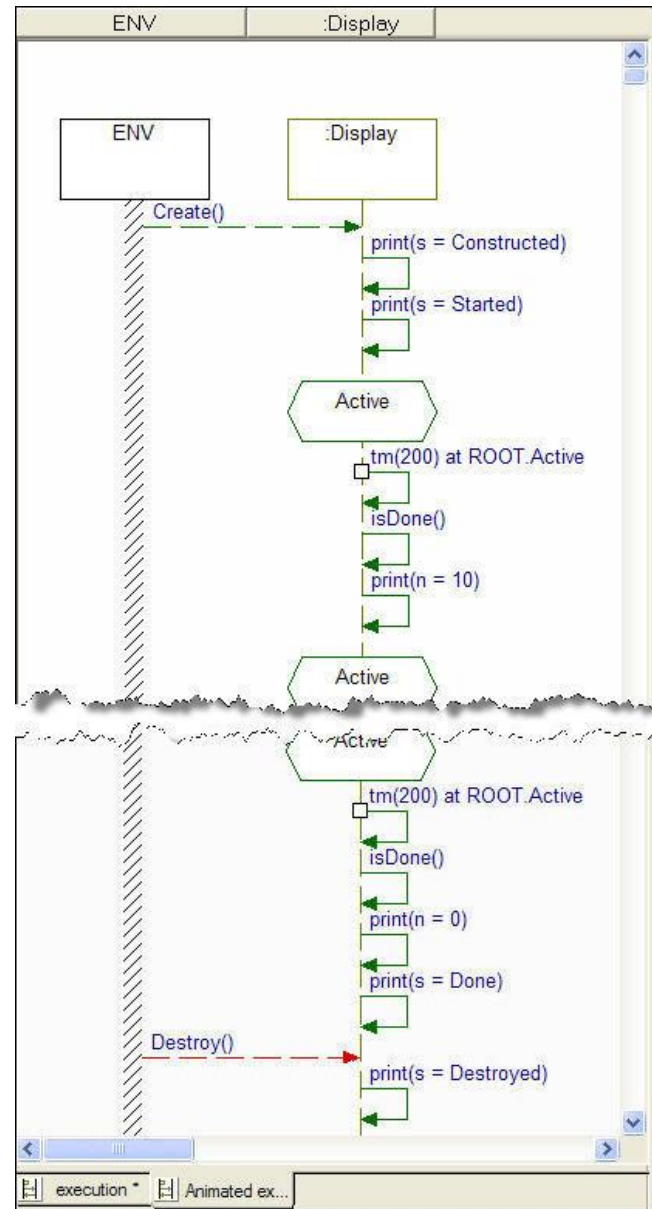
Animated sequence diagrams

- Start the animation and click **Go.**



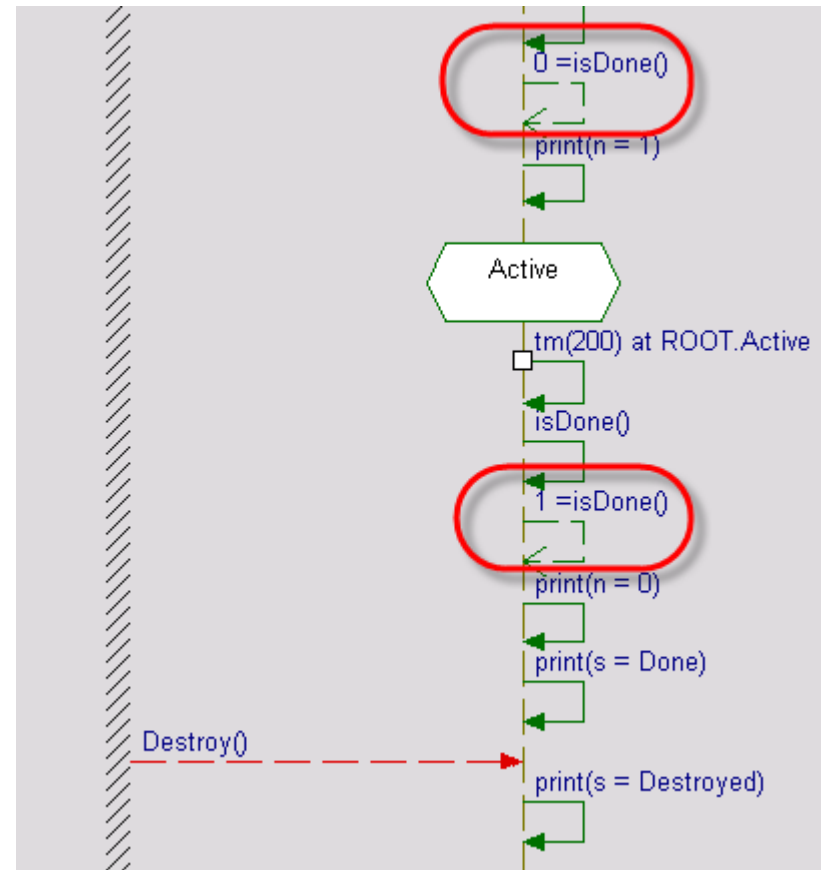
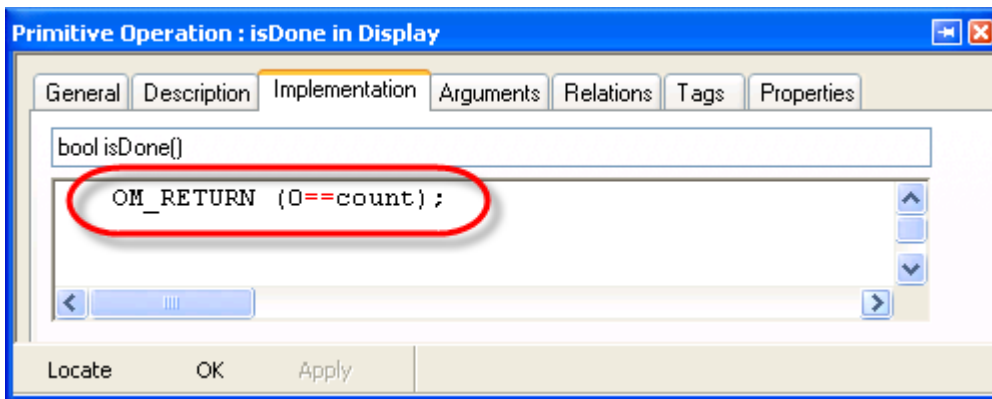
```
Constructd
Started
Count = 10
Count = 9
Count = 8
Count = 7
Count = 6
Count = 5
Count = 4
Count = 3
Count = 2
Count = 1
Count = 0
Done
Destroyed
```

- If a sequence diagram is open, then Rational Rhapsody creates a new animated sequence diagram based on the execution.
 - Note that the animated sequence diagram captures operations, timeouts, and states.



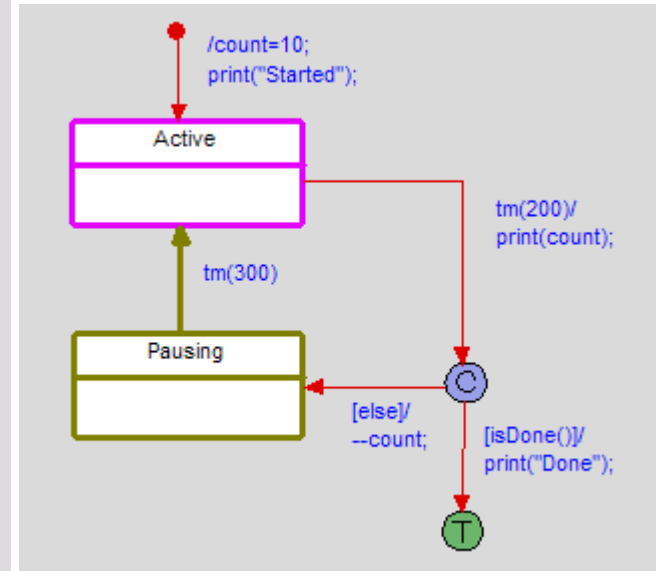
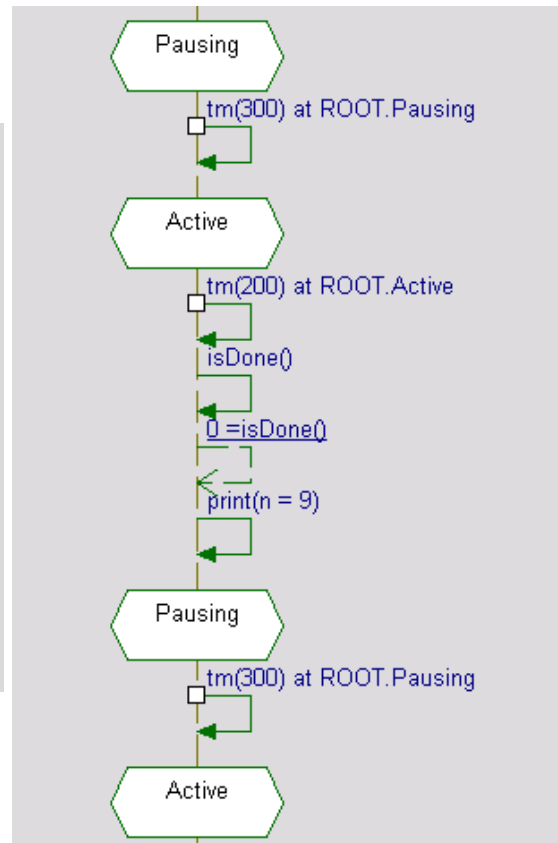
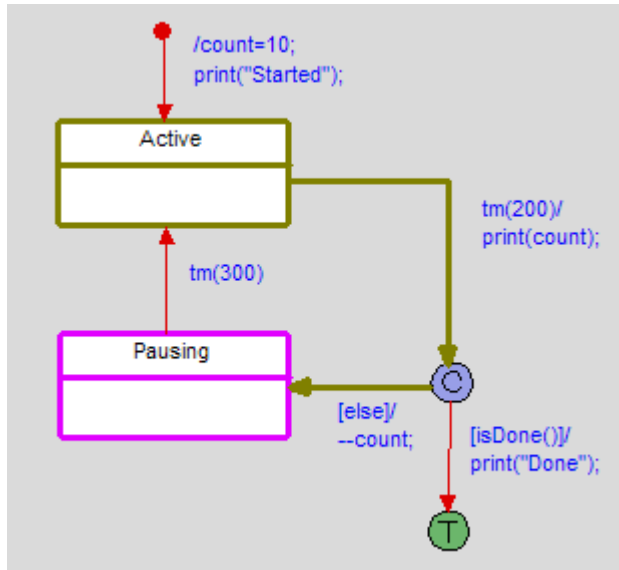
Extended exercise I

- Rational Rhapsody can display the return value on animated sequence diagrams. To do so, you must use a macro `OM_RETURN`.
- In the implementation of the operation `isDone()`, replace `return` with `OM_RETURN`.



Extended exercise II

- Try adding an extra state *pausing*. Then you will see the instance changing states.

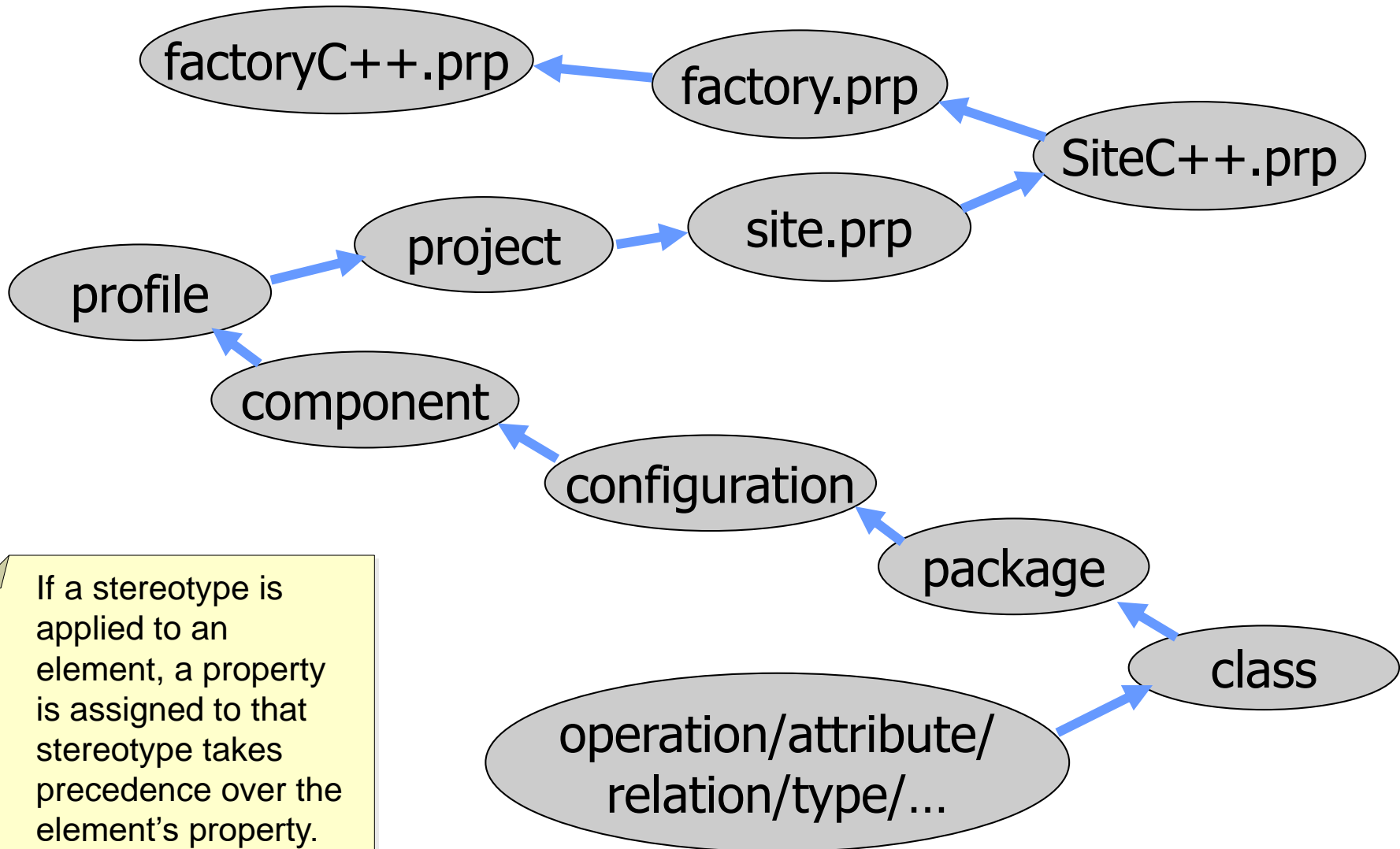


Properties

- There are many properties that allow customization of the tool and the generated code.
- Properties can be set once and for all by modifying the *site.prp* file in the *Rhapsody\7.5\Share\Properties* directory.
- The *factory.prp* and *factoryC++.prp* files contain all the Rational Rhapsody properties.

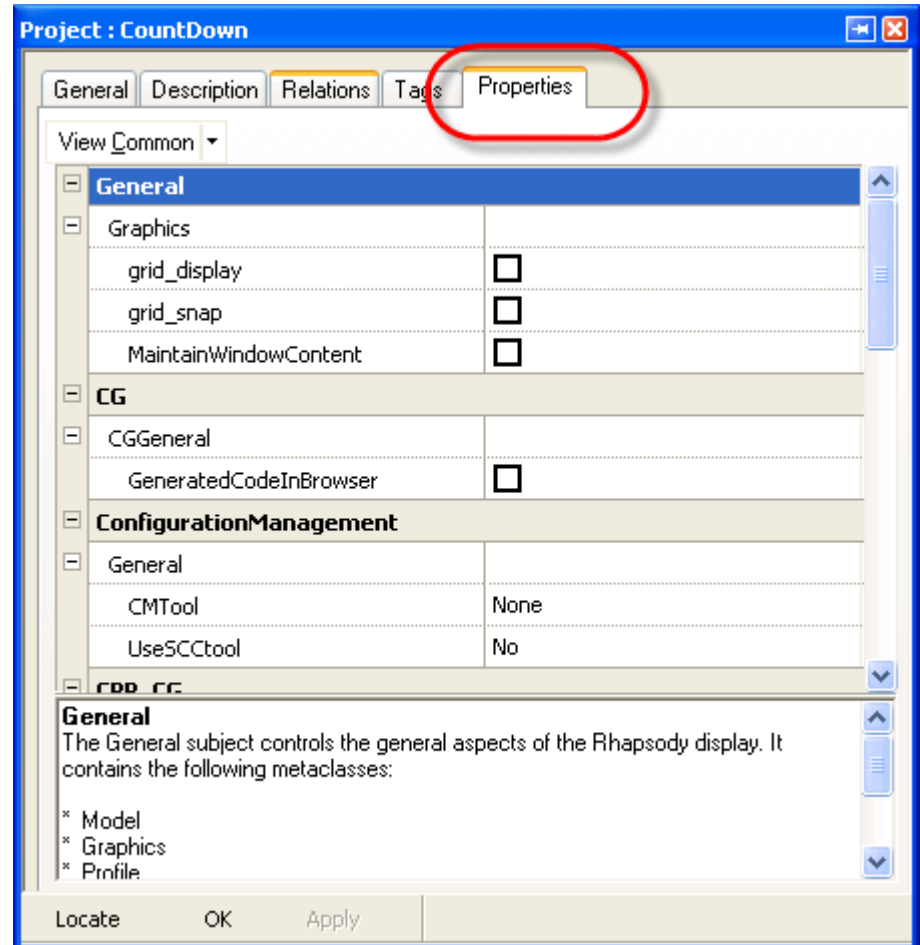
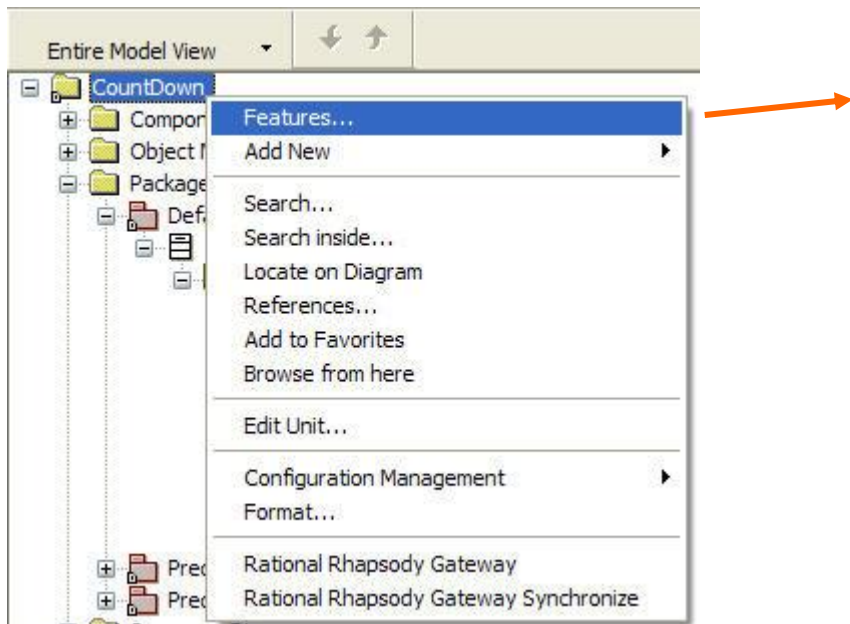
It is recommended you modify the *site.prp* or *siteC++.prp* files rather than the *factory.prp* and *factoryC++.prp* files. To do so, it is easiest to copy and paste from these files into the *site.prp* or *siteC++.prp* file.

Properties hierarchy



Project properties

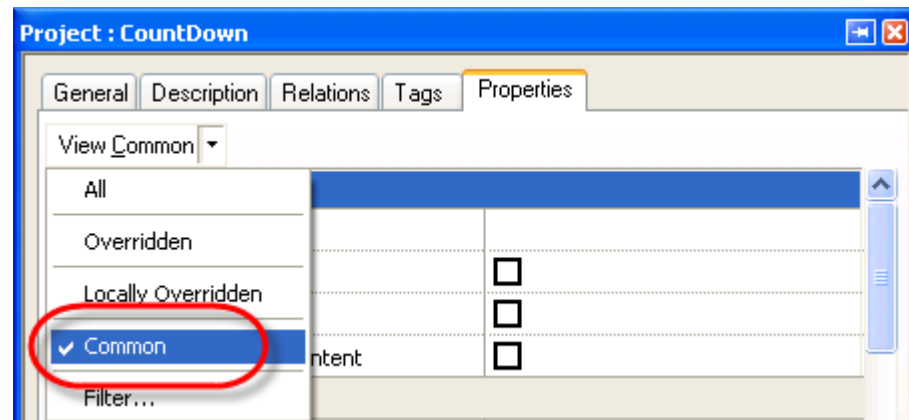
- Bring up the Features for the *CountDown* project and select the **Properties** tab.



Properties view

- There are a very large number of properties which can be used to customize the tool and the generated code.
- In order to facilitate access to these properties, there are several *views* that can be applied to the properties.
- For this training course, you use the most common properties which can be seen using the *Common* view.

It is relatively easy to modify the list of properties that can be seen in the Common view.

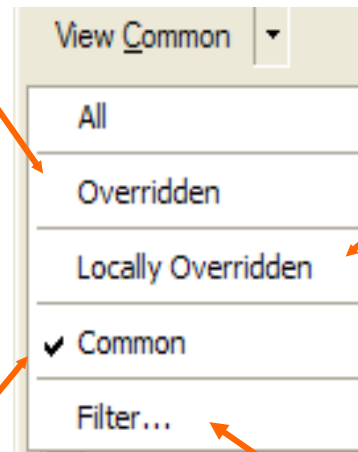


Properties views

- There are several useful views of the Properties:

Properties that have been overridden for the currently selected element & any 'parent' overrides

Only properties that have been overridden for the currently selected element



A user defined list of the most commonly accessed properties

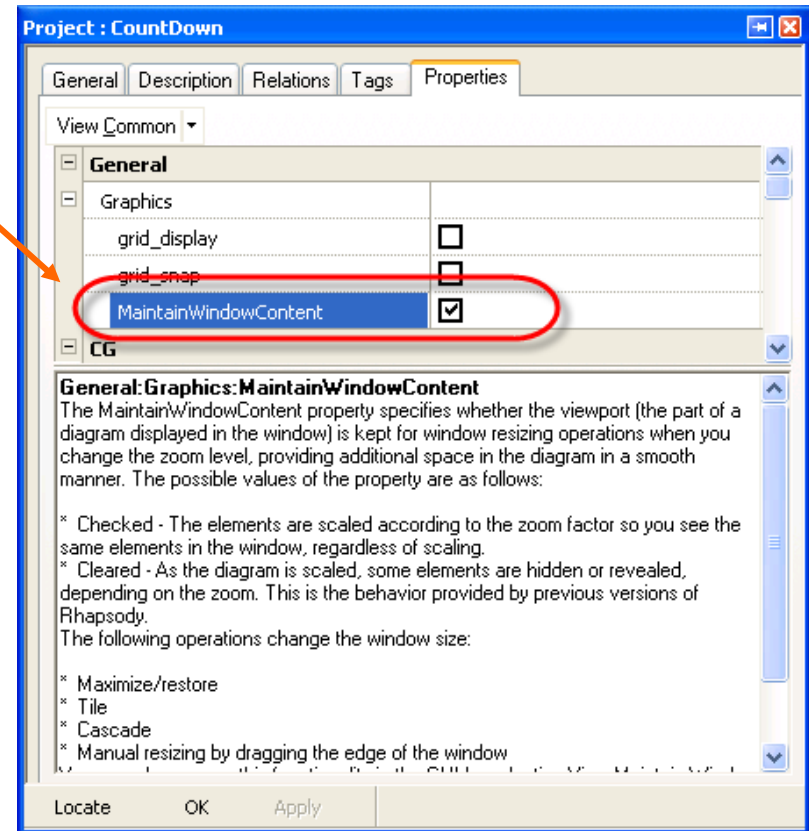
Properties filtered by some keyword

Useful property

- One useful property is *General:Graphics:MaintainWindowContent*.
- Setting this property means that if the size of the window is changed, then the view of the contents changes proportionally.
- Set this property.

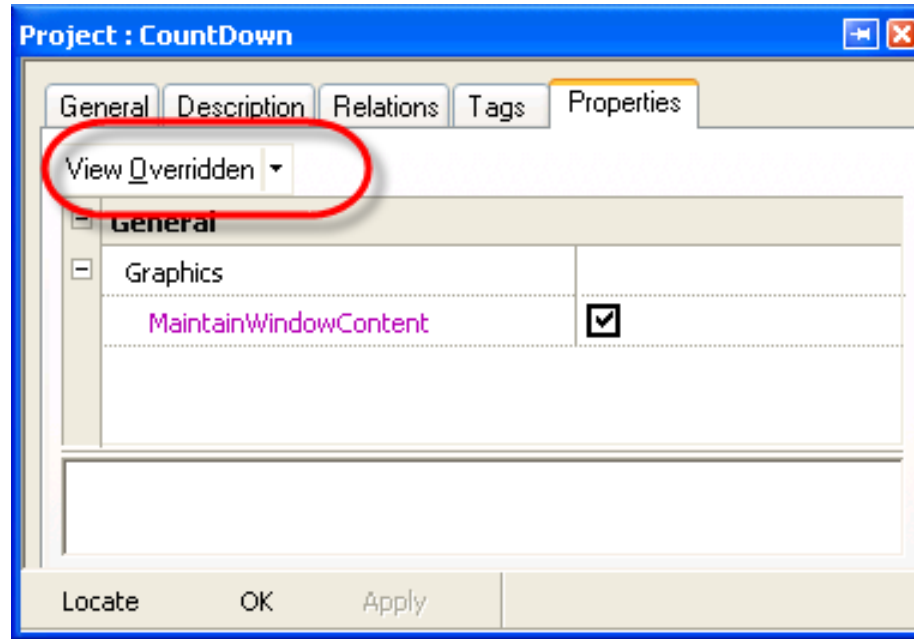
Once a property has been modified it is highlighted. To restore the default, right-click on the property and select **Un-override**.

Note also the description is displayed for the selected property.



Overridden properties

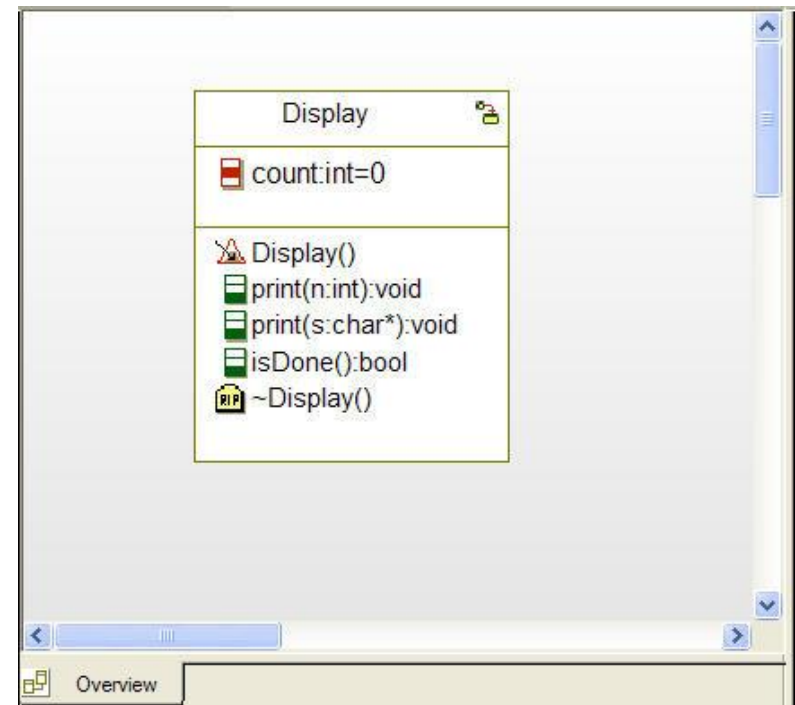
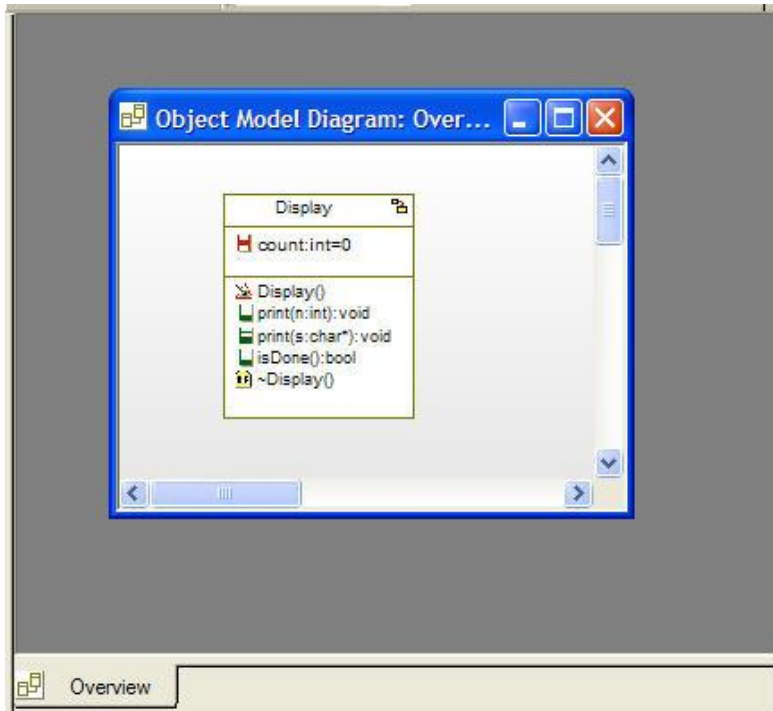
- Select **View Overridden**.



This shows just the properties that have been modified.

General:graphics:MaintainWindowContent

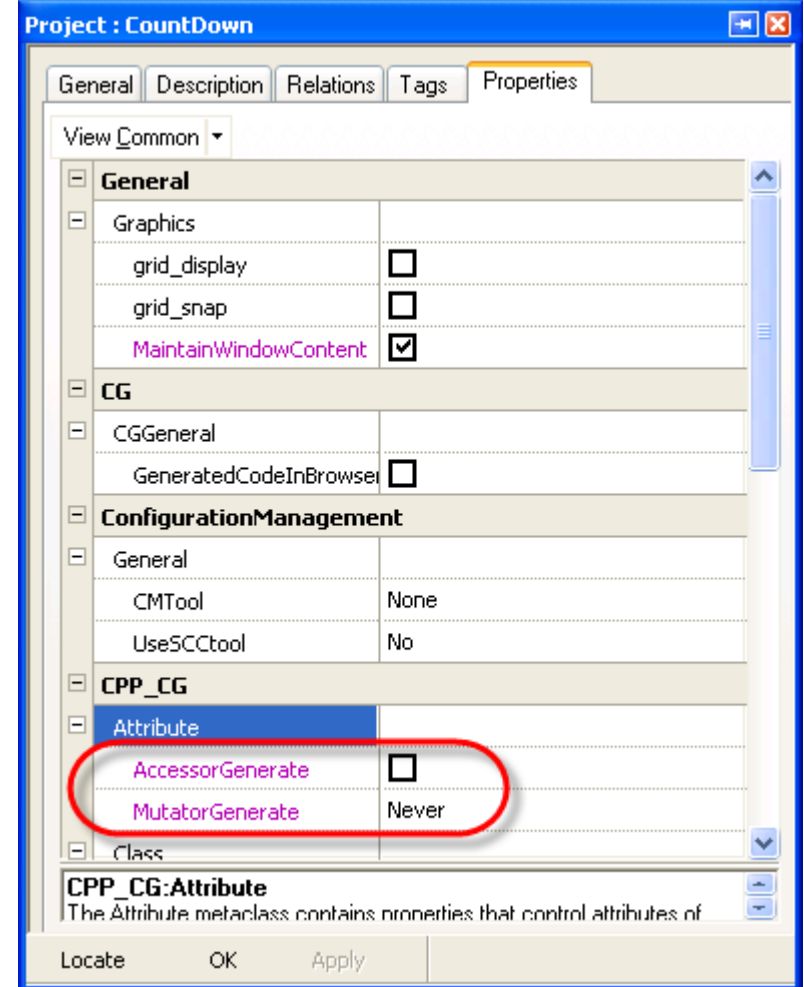
- Once this property has been set, changing the size of a window should keep the same view:



- You need to close any open windows and then reopen them after setting this property.

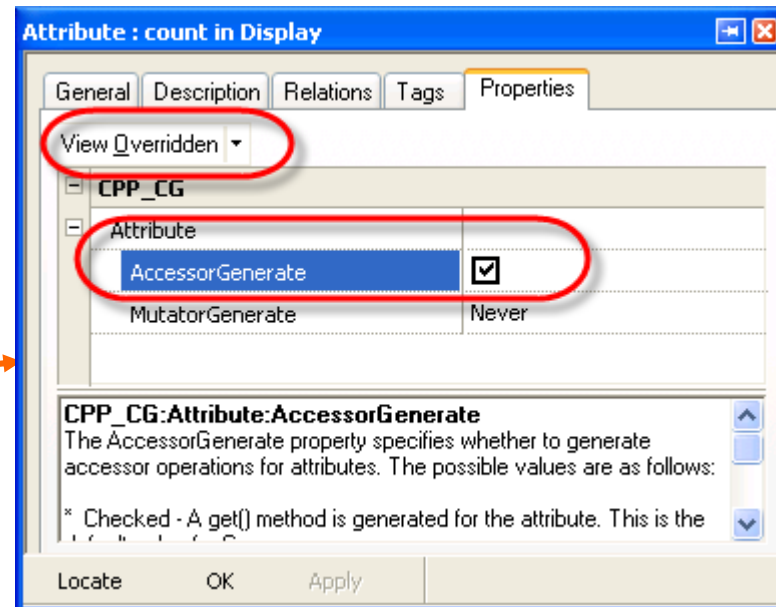
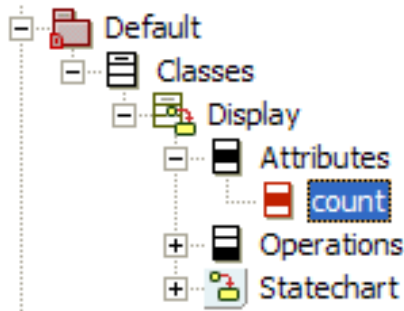
Accessors and mutators

- If *accessors* and *mutators* are not needed for attributes, then properties can be set to stop their generation.
- Set these two properties so that ALL attributes in the project will have neither an *accessor* nor a *mutator*.



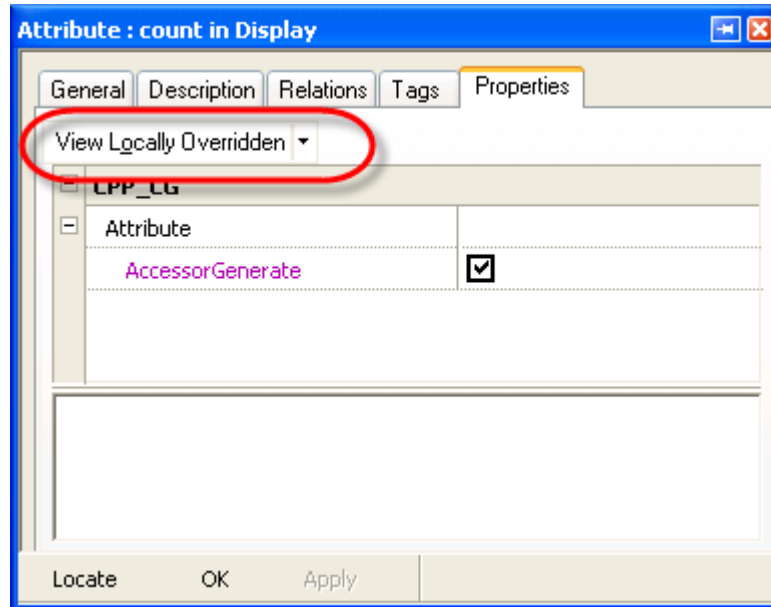
Overridden properties

- For the attribute *count*, you want an accessor.
- Selecting the *overridden* filter shows that the *AccessorGenerate* and *MutatorGenerate* properties have been overridden higher up in the property hierarchy.
- Select the *count* attribute and override the property: *AccessorGenerate*.



Locally overridden properties

- Select the **View Locally Overridden** filter to show that just the *AccessorGenerate* property has been set locally.

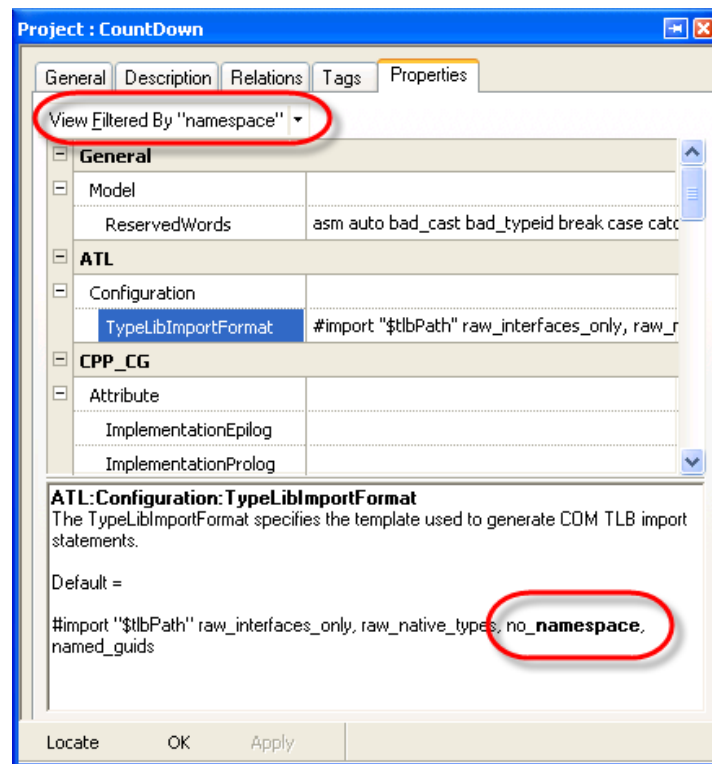
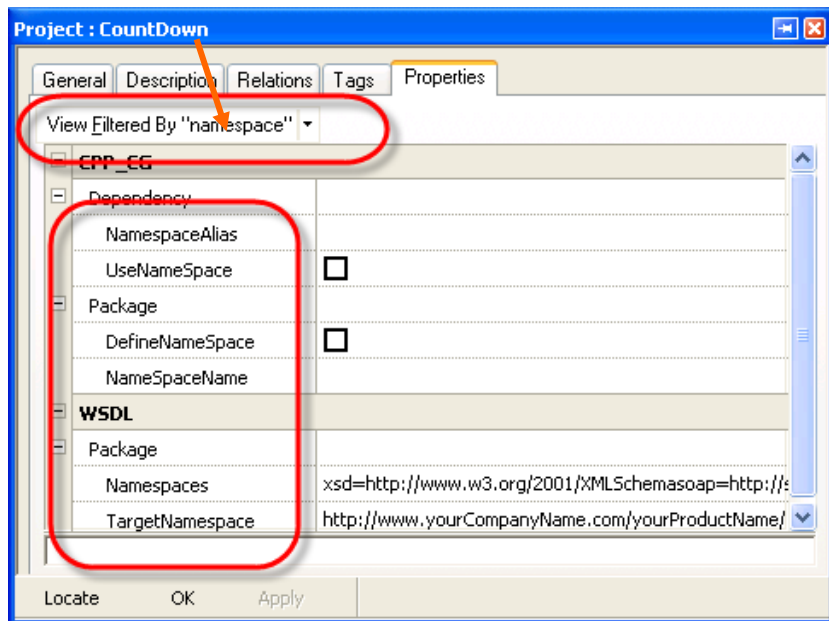
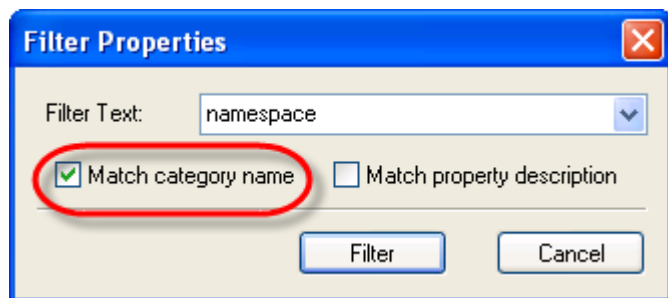
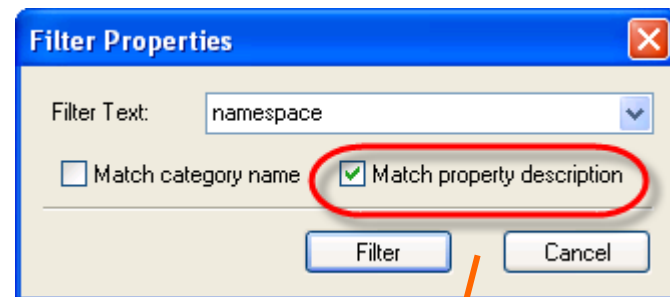


```
int Display::getCount() const {  
    return count;  
}
```

- Generate code and check that there is just an accessor for the attribute *count*.

Property filter

- A customized view of the properties can be created by using the *Filter* view for example:



Extended exercise

- Experiment with some of the properties such as:
CG:CGGeneral:GeneratedCodeInBrowser
- You must regenerate the code after setting this property.

The screenshot shows the UML IDE interface for a project named 'CountDown'. On the left, a class hierarchy tree shows 'Display' with attributes 'count', operations '~Display()', 'Display()', 'getCount() const', 'isDone()', 'print(char* s)', 'print(int n)', and 'startBehavior()', and a statechart. The main window displays the 'Properties' tab for the 'CG:CGGeneral' class. The 'GeneratedCodeInBrowser' property is checked, highlighted with a red circle. Below the table, a description explains that this property specifies whether canonical operations (get/set) are added to the model and displayed in the browser. The possible values are: * Checked - Display automatically generated operations in the browser tree. * Cleared - Do not display canonical operations. (Default = Cleared). On the right, a preview window shows the resulting code for the 'Display' class, including the attribute 'count:int=0', the 'Display()' operation, and the methods 'print(n:int):void', 'print(s:char*):void', 'isDone():bool', '~Display()', 'getCount():int', and 'startBehavior():bool'.

Class	Property	Value
CG	CGGeneral	GeneratedCodeInBrowser <input checked="" type="checkbox"/>
ConfigurationManagement	General	

CG:CGGeneral:GeneratedCodeInBrowser
The GeneratedCodeInBrowser property specifies whether canonical operations (get/set) are added to the model and displayed in the browser. The possible values are as follows:

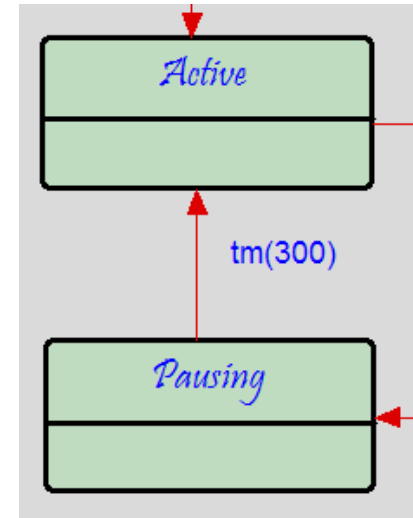
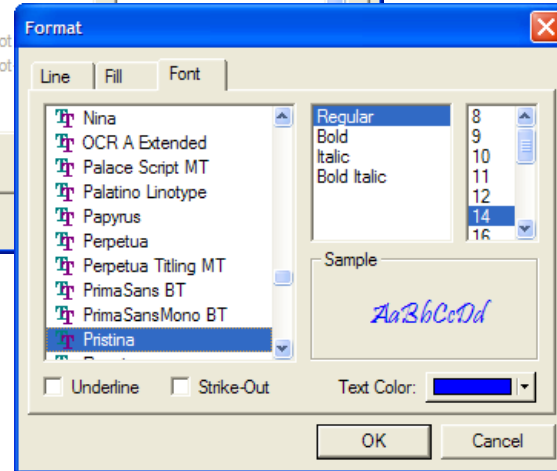
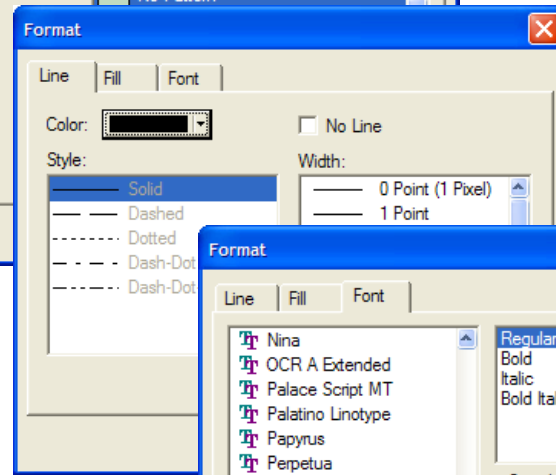
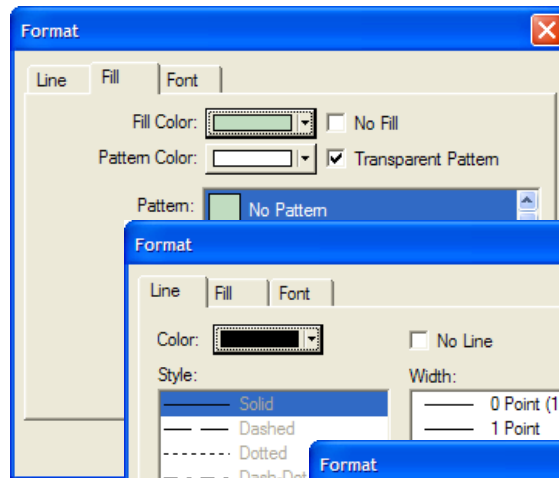
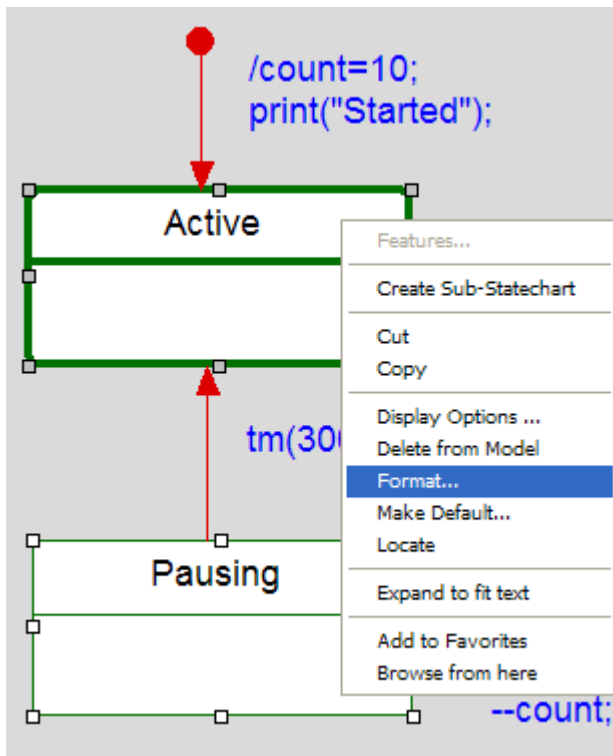
- * Checked - Display automatically generated operations in the browser tree.
- * Cleared - Do not display canonical operations.

(Default = Cleared)

Locate OK Apply

Formatting individual items

- Line Colors, Fill Colors, Fonts, etc of selected element(s) can all be formatted by right clicking and selecting **Format**.



Advanced drawing capabilities

- These advanced drawing capabilities are common to most diagrams:

Show rulers

Align items to common edge

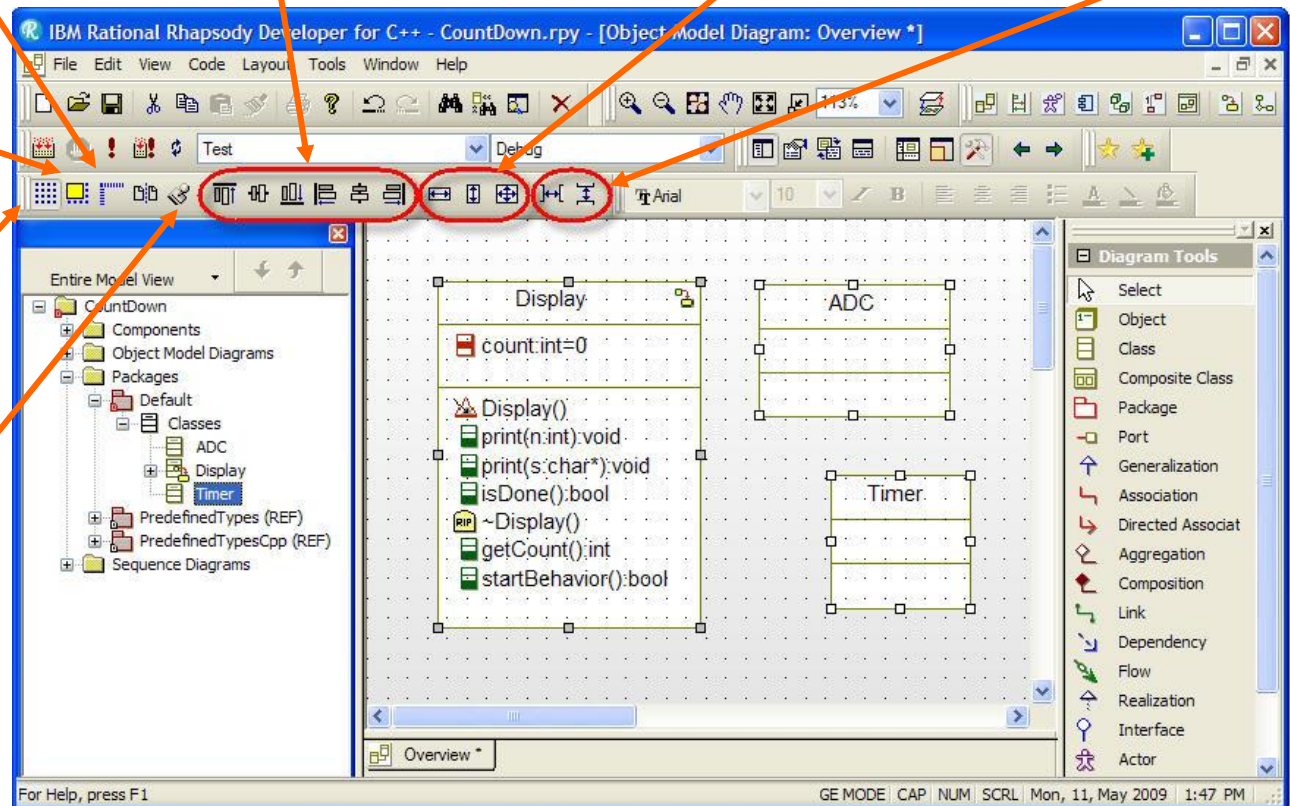
Make item same size

Give items same spacing size

Snap to grid

Turn grid on/off

Stamp mode - use when drawing same items repeatedly



Aligning items to common edge

- A pivot selection mechanism is used for aligning, sizing and spacing:

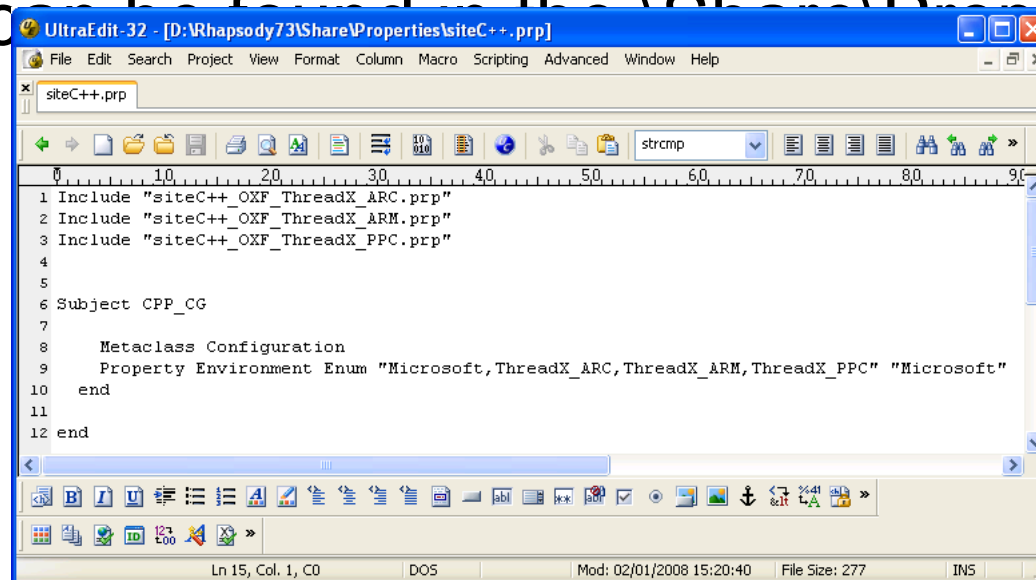
1. Select by area

2. Select the pivot class by holding Ctrl and selecting

3. Choosing align left, aligns all classes to the class ("class_4")

Site.prp / SiteC++.prp

- Adding new environments is done via the file *siteC++.prp*.
- Each organization or team may want to always set certain properties for all of their Rational Rhapsody projects. To do this, set these properties for every Rational Rhapsody project by putting them into the file *site.prp*.
- These files can be found in the \Share\Properties directory.



The screenshot shows a text editor window titled "UltraEdit-32 - [D:\Rhapsody73\Share\Properties\siteC++.prp]". The window displays the following code:

```
1 Include "siteC++_OXF_ThreadX_ARC.prp"
2 Include "siteC++_OXF_ThreadX_ARM.prp"
3 Include "siteC++_OXF_ThreadX_PPC.prp"
4
5
6 Subject CPP_CG
7
8     Metaclass Configuration
9     Property Environment Enum "Microsoft,ThreadX_ARC,ThreadX_ARM,ThreadX_PPC" "Microsoft"
10 end
11
12 end
```

The status bar at the bottom indicates "Ln 15, Col. 1, C0", "DOS", "Mod: 02/01/2008 15:20:40", "File Size: 277", and "INS".

Exercise 2A: Count down with Seven-Segment Display

- It is necessary to access the hardware.
- First prepare a hardware interface code.
- Then combine it with the model code.
- Model code + legacy code



Copy and modify fpga_test_fnd.c(1)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
```

```
#define MAX_DIGIT 4
#define FND_DEVICE "/dev/fpga_fnd"
```

```
int dev;
unsigned char data[4];
```

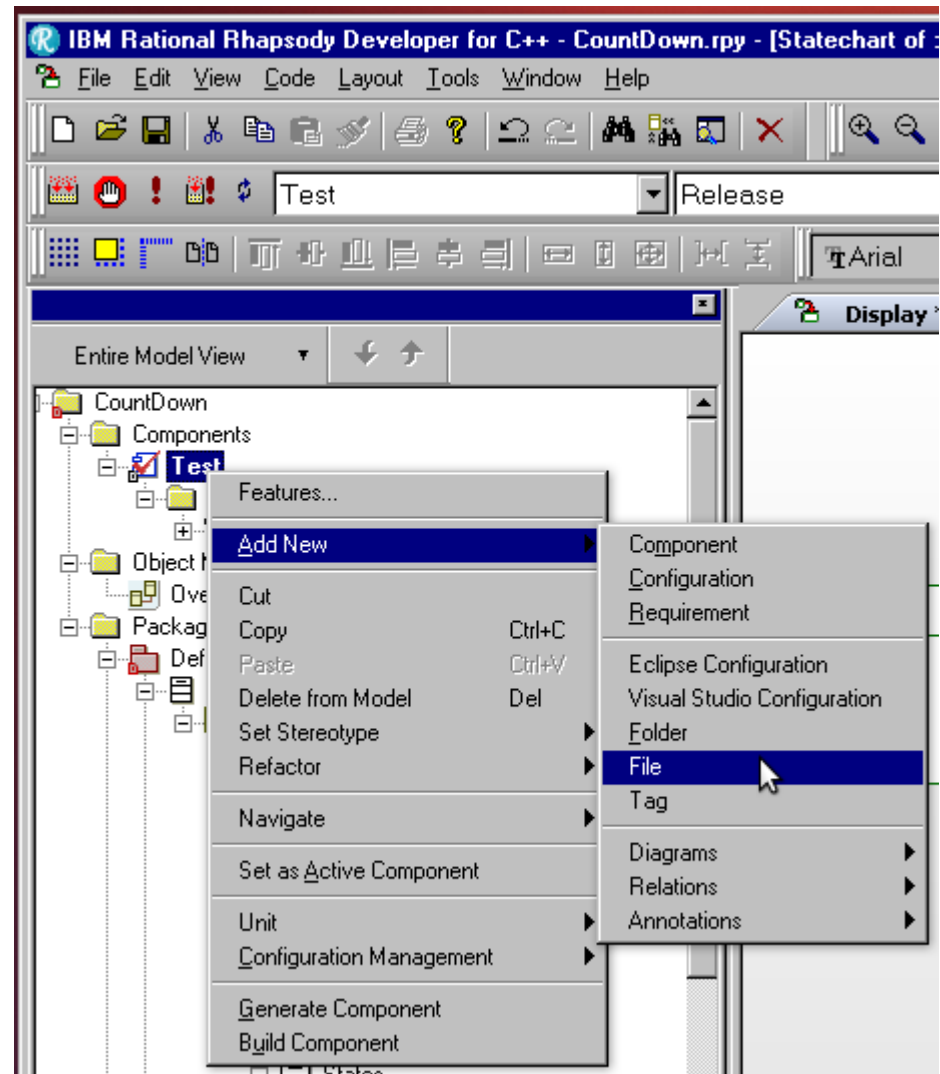

Copy and modify fpga_test_fnd.c(2)

```
int initSevenSegment(void)
{
    dev = open(FND_DEVICE, O_RDWR);
    if (dev<0) {
        printf("Device open error : %s\n",FND_DEVICE);
        exit(1);
    }
}

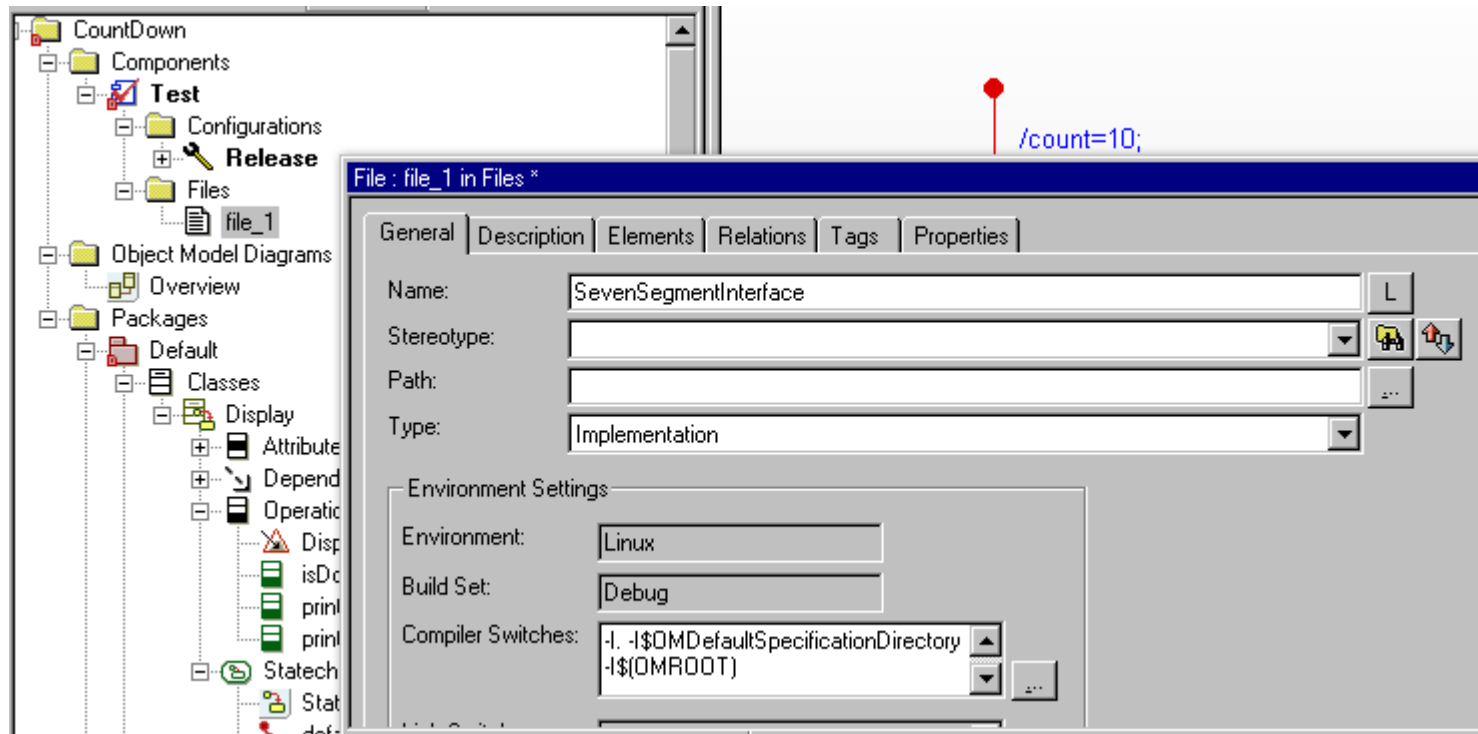
void displaySevenSegment(int value)
{
    data[2]=value/10;
    data[3]=value%10;
    write(dev,&data,4);
}
```

Add Hardware Interface Code

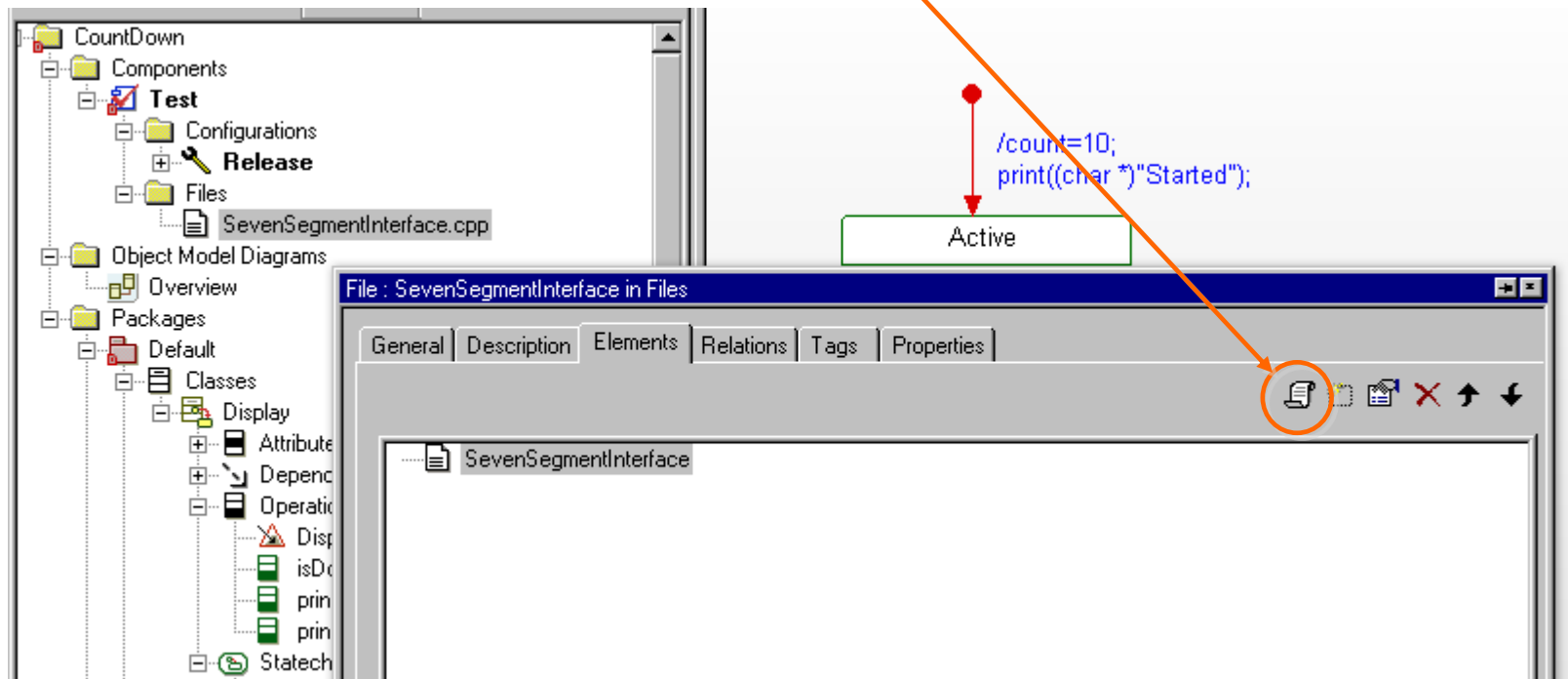
- Right Click Test Component and Select Add New “File”



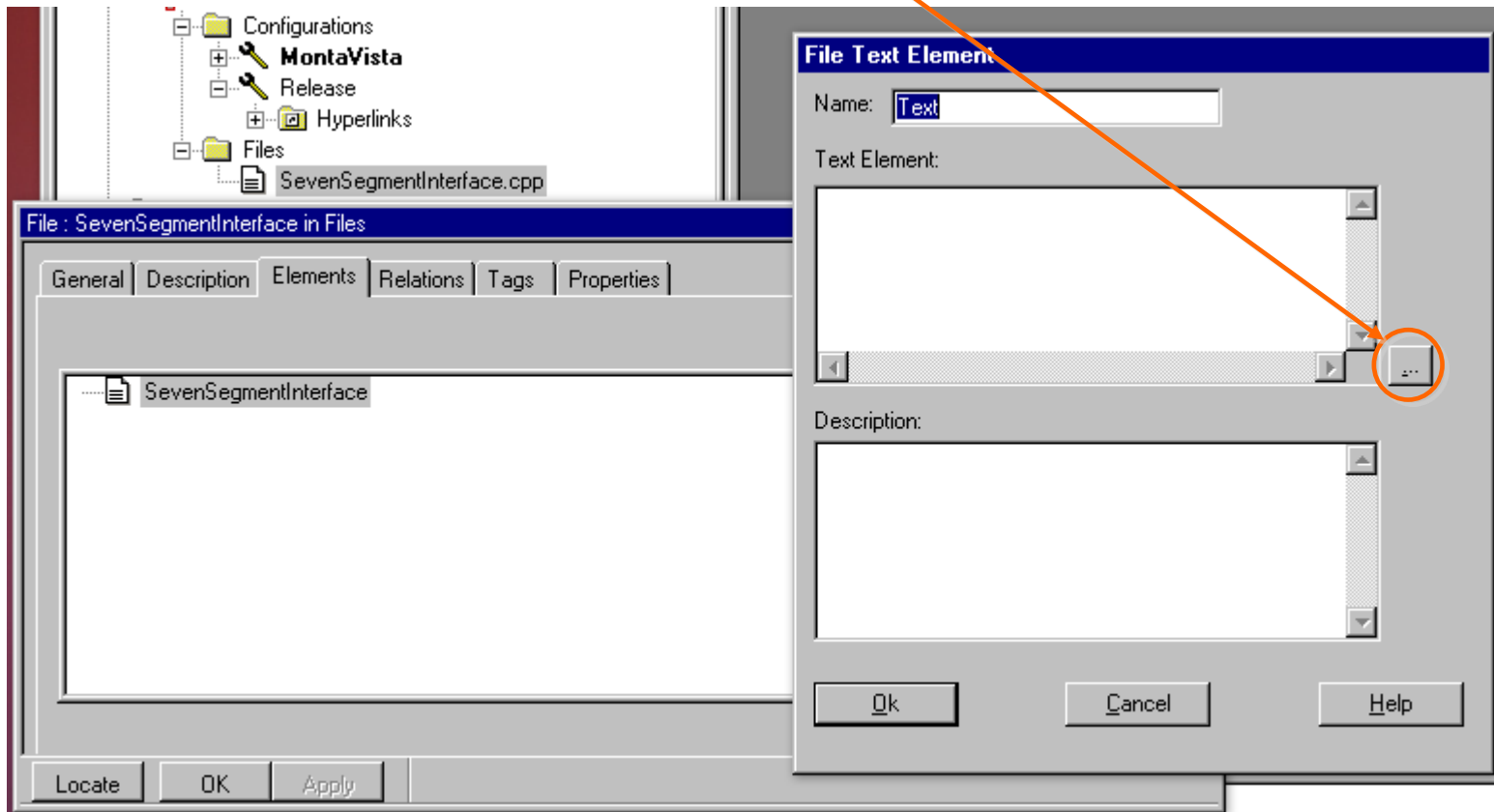
- Double click a new file and change Name to “SevenSegmentInterface” and Type to Implementation



- Double click SevenSegmentInterface.cpp and select the tab “Elements”.
- Then click New Text Element button



- Open the text editor



- Paste the code and click OK

The screenshot shows an IDE interface. On the left, a file explorer displays a project structure with folders like 'CountDown', 'Components', and 'Test'. Under 'Test', there are sub-folders for 'Configurations', 'Files', and 'Hyperlinks'. A file named 'SevenSegmentInterface.cpp' is highlighted in the 'Files' folder. Below the file explorer, a dialog box titled 'File : SevenSegmentInterface in Files' is open, showing a tree view with 'SevenSegmentInterface' selected. At the bottom of this dialog, there are buttons for 'Locate', 'OK', and 'Apply'. A status bar at the bottom of the IDE shows the text: 'Generating main file MainTest.cpp', 'Generating make file Test.mak', and 'Code Generation Done'.

On the right, a 'Text Editor' window is open, displaying the following C++ code:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

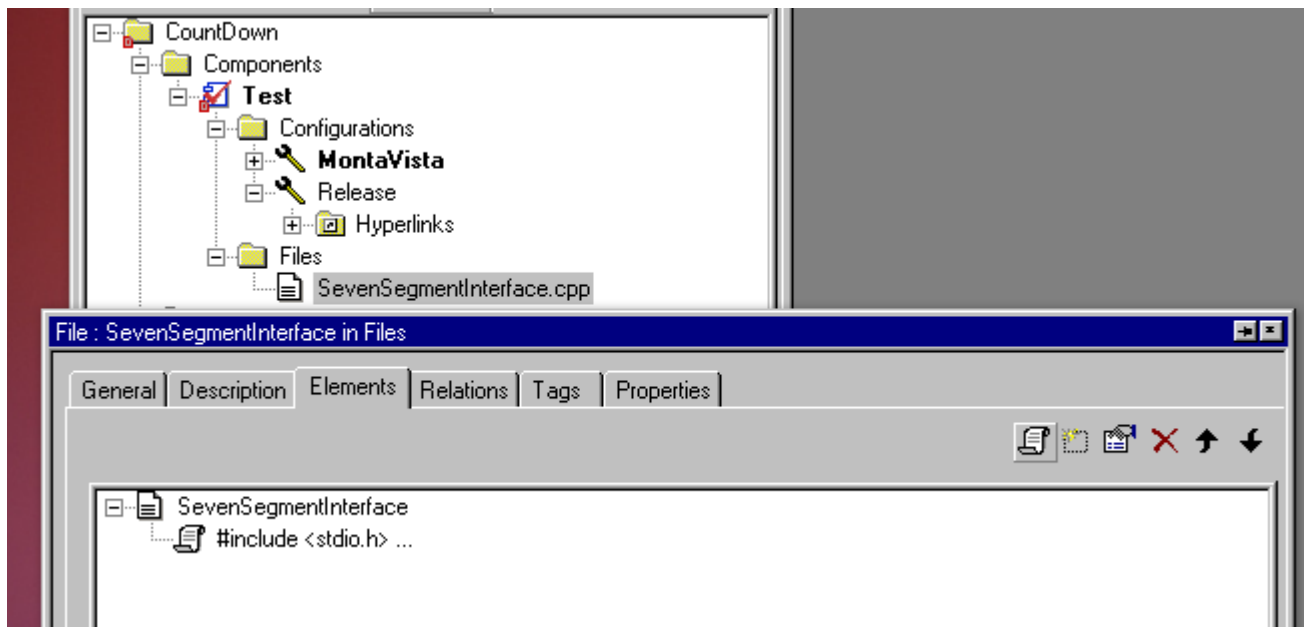
#include <string.h>

#define MAX_DIGIT 4
#define FND_DEVICE "/dev/fpga_fnd"
int dev;
unsigned char data[4];

int initSevenSegment(void)
{
    dev = open(FND_DEVICE, O_RDWR);
    if (dev<0) {
        printf("Device open error : %s\n",FND_DEVICE);
        exit(1);
    }
}

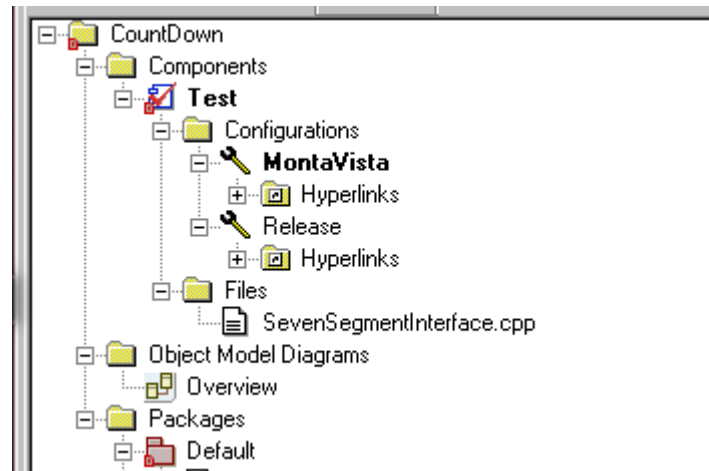
void displaySevenSegment(int value)
{
    data[2]=value/10;
    data[3]=value%10;
```

At the bottom of the text editor, there are buttons for 'OK', 'Cancel', and 'Help'.



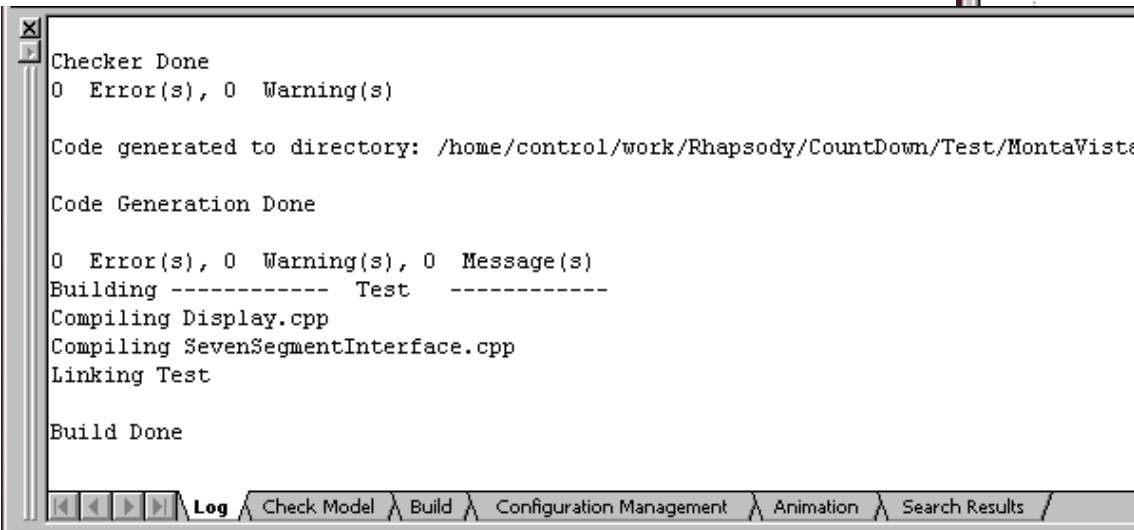
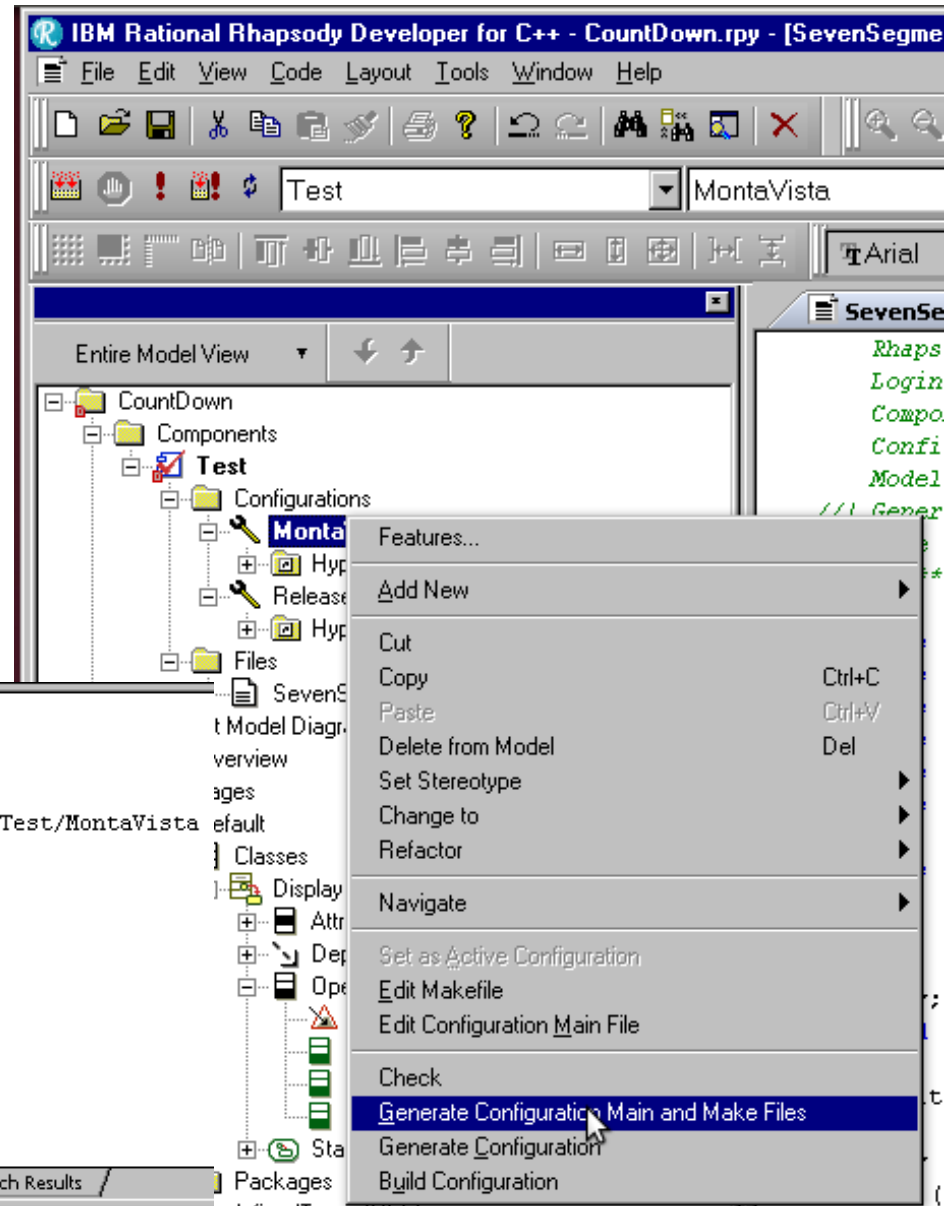
New Configuration

- Right click Configurations and Add New Configurations
- Change the name to MontaVista
- Right click and Set as Active Configurations



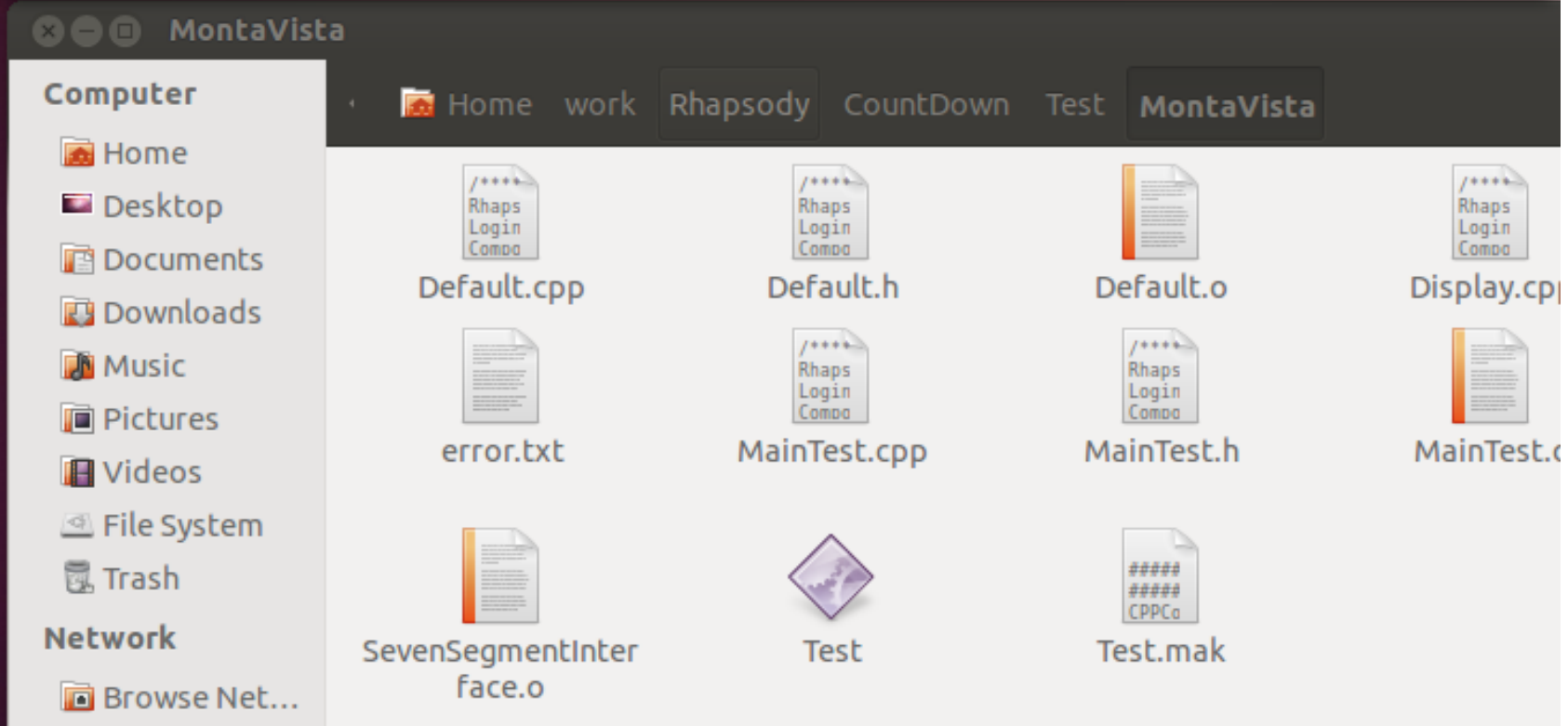
Build

- Generate Configuration Main and Make files
- Generate Configuration
- Build Configuration
- Check no error



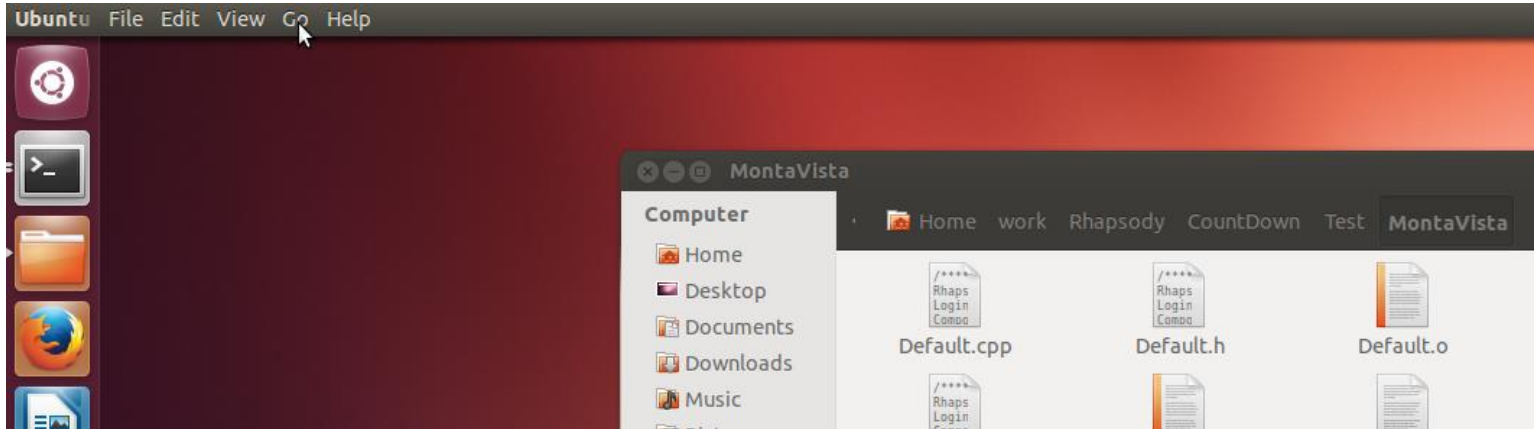
- Copy executable file to /nfsroot folder

```
control@lab-pc2:~/work/Rhapsody/CountDown/Test/MontaVista$ ls
Default.cpp  Display.h      MainTest.h      SevenSegmentInterface.o
Default.h    Display.o      MainTest.o      Test
Default.o    error.txt      MontaVista.cg_info  Test.mak
Display.cpp  MainTest.cpp  SevenSegmentInterface.cpp
control@lab-pc2:~/work/Rhapsody/CountDown/Test/MontaVista$ cp Test /nfsroot
control@lab-pc2:~/work/Rhapsody/CountDown/Test/MontaVista$
```

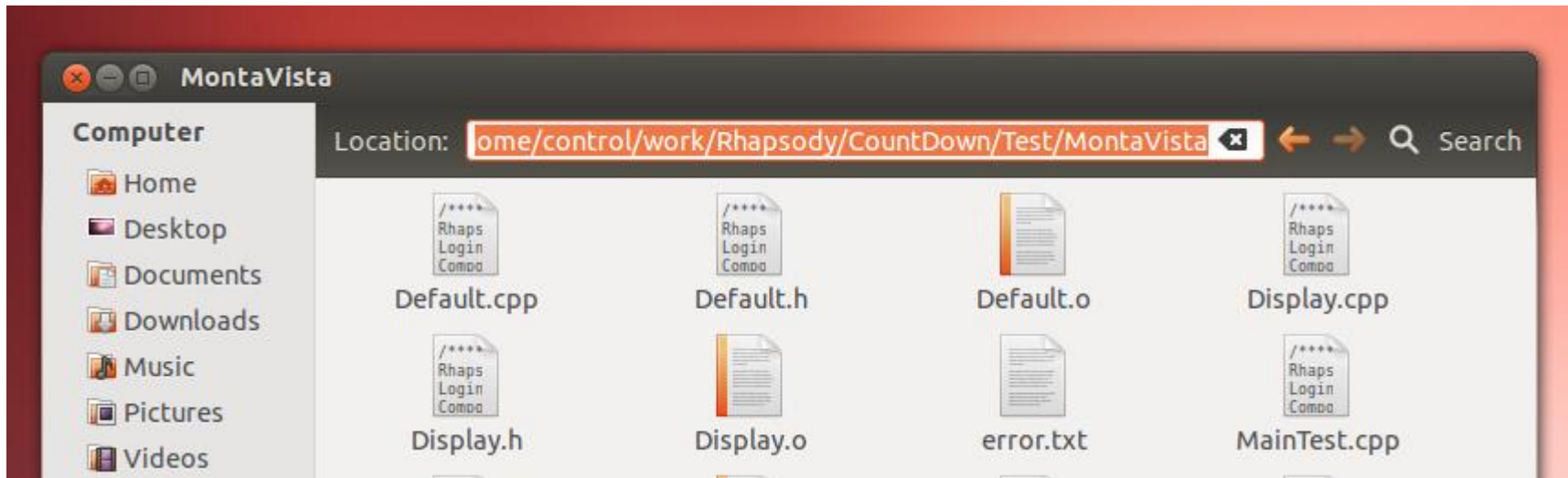


cd command with a long path name

- Move the mouse cursor to the menu bar

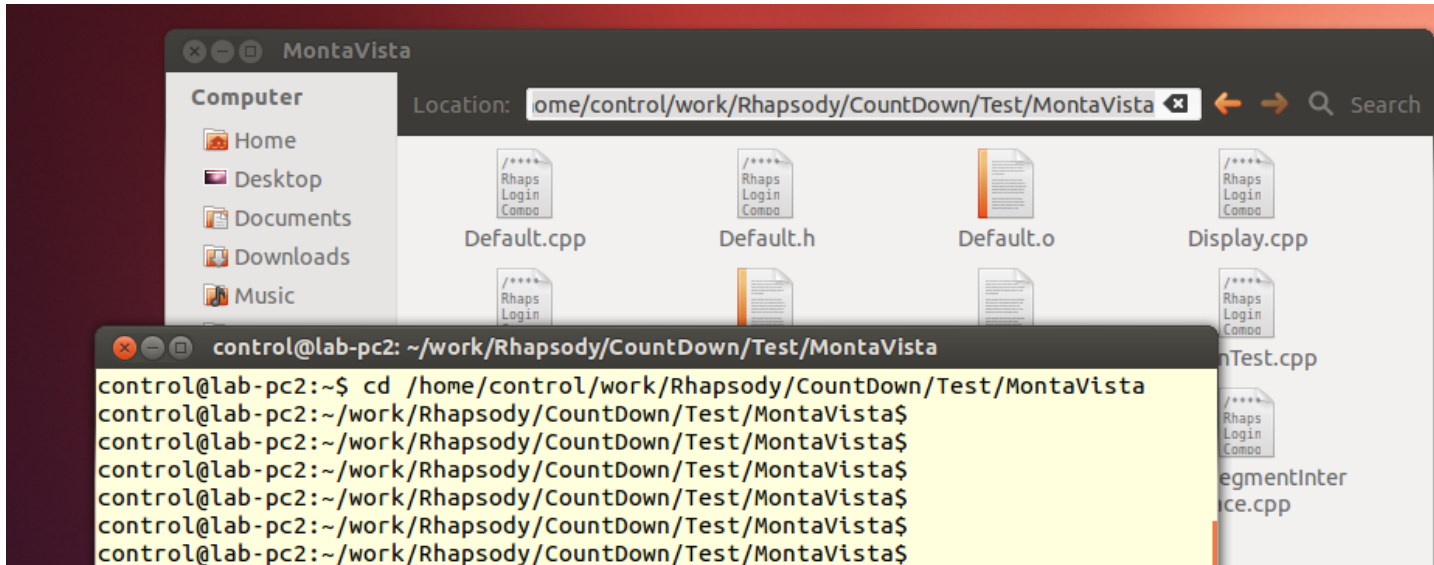


- Select Location from Go menu




cd command with a long path name

- Copy and paste to the Terminal window



Run on the target

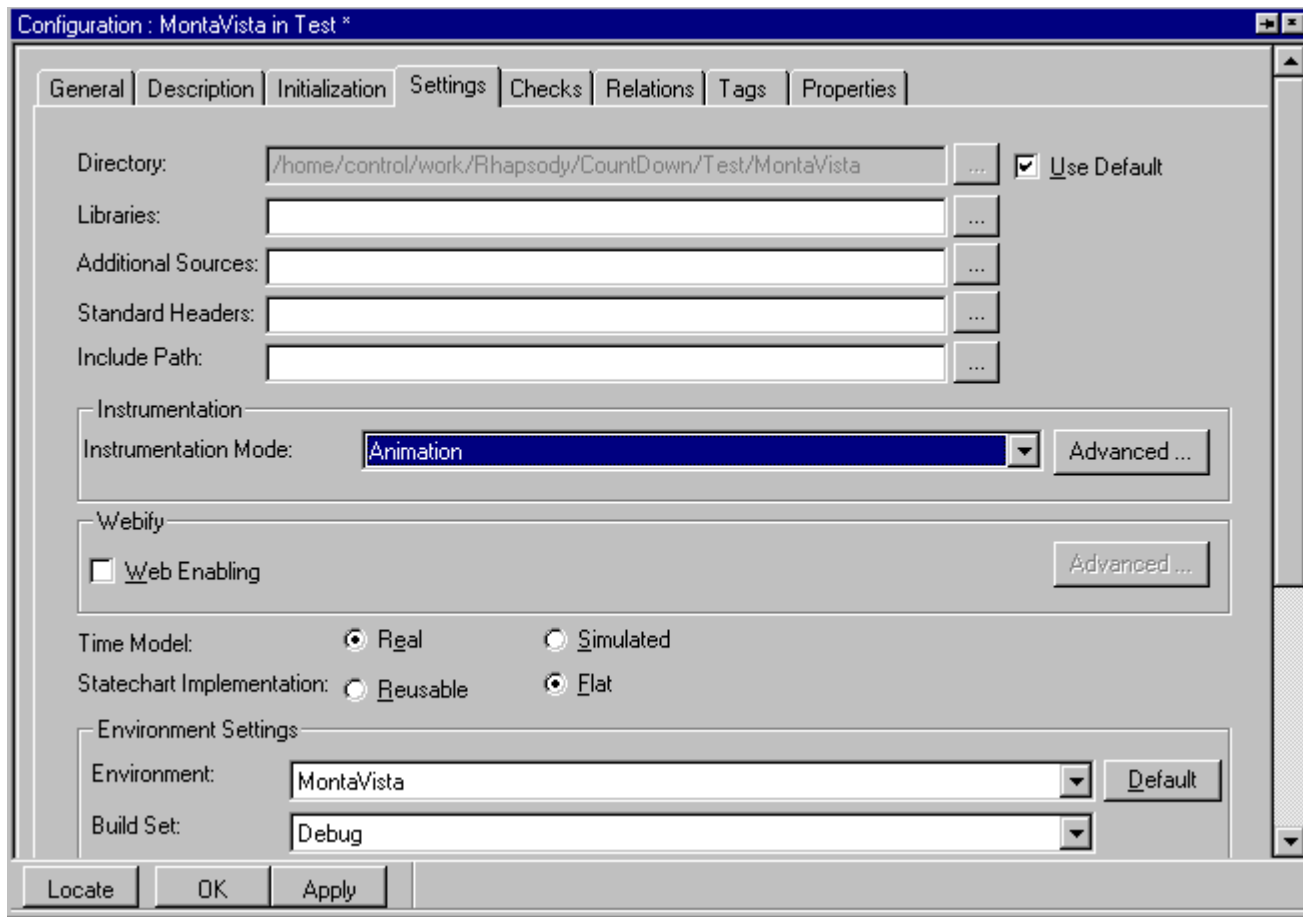


The image shows a terminal window titled "COM3 - Tera Term VT". The window has a menu bar with "File", "Edit", "Setup", "Control", "Window", and "Help". The terminal content shows a user at the root of a device named "ACHRO" navigating to the "/mnt/nfs" directory and running a script named "Test". The script outputs a series of "Count" values from 10 down to 0, followed by "Done".

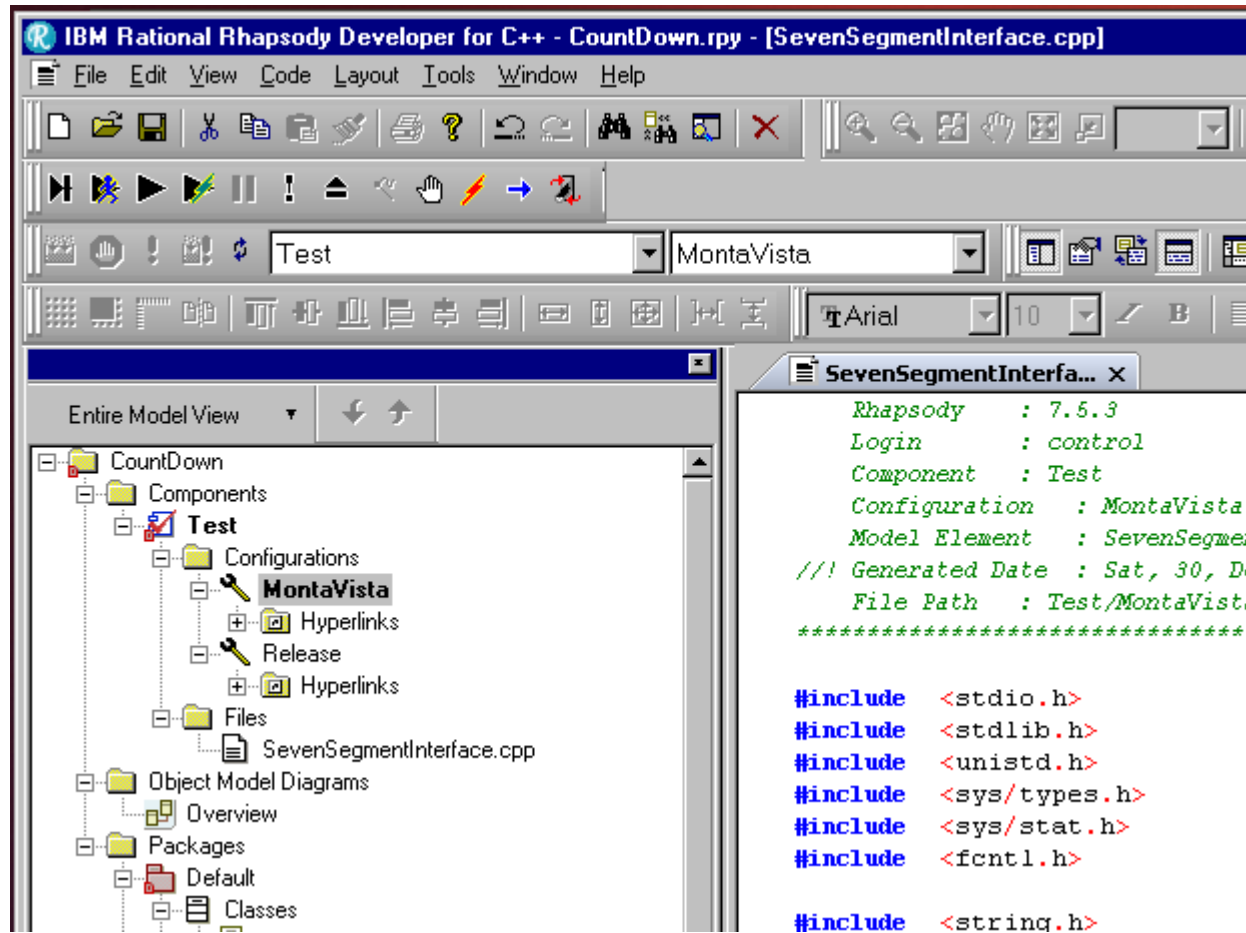
```
COM3 - Tera Term VT
File Edit Setup Control Window Help
[root@ACHRO ~]# cd /mnt/nfs
[root@ACHRO nfs]# ./Test
Consruacted
Started
Count = 10
Count = 9
Count = 8
Count = 7
Count = 6
Count = 5
Count = 4
Count = 3
Count = 2
Count = 1
Count = 0
Done
```

Run on the target with animation

- Change to Animation, build, and copy



Run on the target with animation



The screenshot displays the IBM Rational Rhapsody Developer for C++ interface. The title bar reads "IBM Rational Rhapsody Developer for C++ - Countdown.rpy - [SevenSegmentInterface.cpp]". The menu bar includes File, Edit, View, Code, Layout, Tools, Window, and Help. The toolbar contains various icons for file operations and development tools. The main window is divided into two panes. The left pane, titled "Entire Model View", shows a hierarchical tree structure of the model. The right pane, titled "SevenSegmentInterfa...", displays the generated code for the "MontaVista" configuration.

Entire Model View

- CountDown
 - Components
 - Test**
 - Configurations
 - MontaVista**
 - Hyperlinks
 - Release
 - Hyperlinks
 - Files
 - SevenSegmentInterface.cpp
 - Object Model Diagrams
 - Overview
 - Packages
 - Default
 - Classes

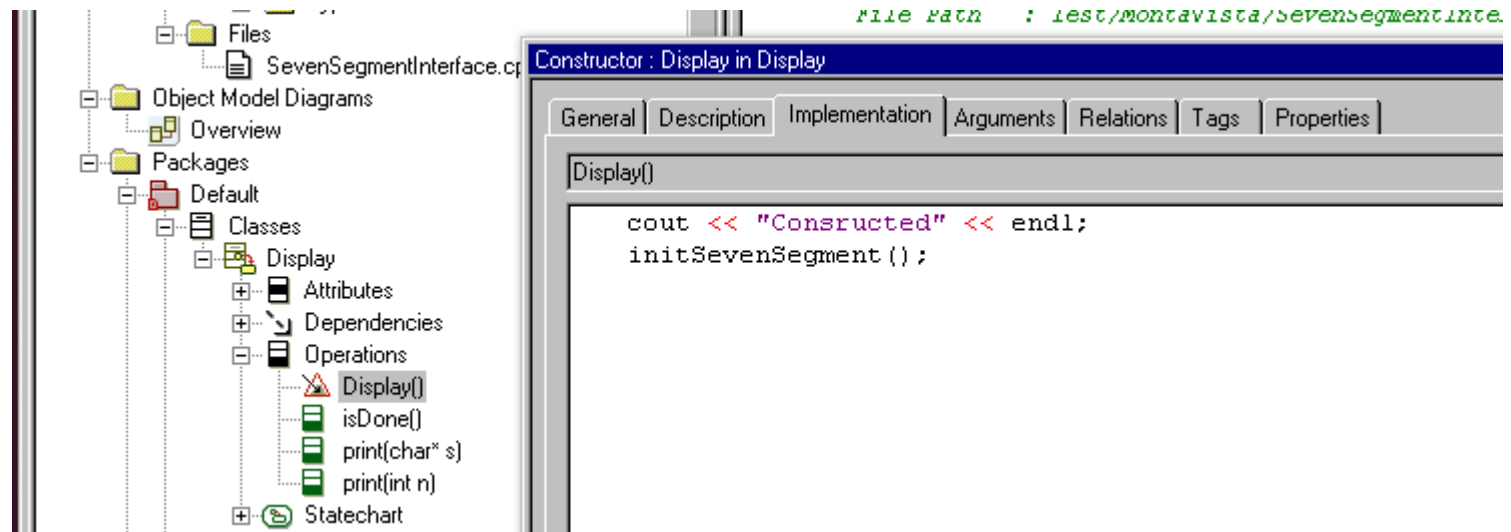
SevenSegmentInterfa...

```
Rhapsody      : 7.5.3
Login         : control
Component     : Test
Configuration : MontaVista
Model Element : SevenSegme
//! Generated Date : Sat, 30, D
File Path    : Test/MontaVist.
*****

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include <string.h>
```

Open the driver



The screenshot displays an IDE interface. On the left, a project tree shows a hierarchy: Files (SevenSegmentInterface.cpp), Object Model Diagrams (Overview), Packages (Default), and Classes (Display). Under the 'Display' class, there are sections for Attributes, Dependencies, and Operations. The 'Display()' constructor is selected and highlighted. On the right, a code editor window titled 'Constructor : Display in Display' shows the implementation of the constructor. The file path is indicated as 'file PATH : test/MONTAVISTA/sevensegmentinte...'. The code in the editor is:

```
Display()
{
    cout << "Construted" << endl;
    initSevenSegment();
}
```


Call Interface Routine

The screenshot shows an IDE interface. On the left is a project tree for 'SevenSegmentInterface.cpp'. The tree is expanded to show the 'Display' class under the 'Default' package. Under 'Display', the 'Operations' folder is expanded, showing several methods: 'Display()', 'isDone()', 'print(char* s)', and 'print(int n)'. The 'print(int n)' method is selected. On the right, a code editor window titled 'Primitive Operation : print in Display *' is open. It has tabs for 'General', 'Description', 'Implementation', 'Arguments', 'Relations', 'Tags', and 'Properties'. The 'Implementation' tab is active, showing the following code:

```
#include <stdio.h>

void print(int n)

    cout << "Count = " << n << endl;
    displaySevenSegment (n);
```

Load the driver and run on the target

```
COM3 - Tera Term VT
File Edit Setup Control Window Help
[root@ACHRO nfs]# ./Test
Construted
Device open error : /dev/fpga_fnd
[root@ACHRO nfs]# ls
SocketCANexample*          fpga_push_switch_driver.ko
StopwatchTest*             fpga_test_fnd*
Test*                       fpga_test_push_switch*
fpga_fnd_driver.ko
[root@ACHRO nfs]# lsmod
Module                      Size Used by
fpga_push_switch_driver    1550  0
[root@ACHRO nfs]# insmod fpga_fnd_driver.ko
[root@ACHRO nfs]# mknod /dev/fpga_fnd c 261 0
[root@ACHRO nfs]# ./Test
Construted
Started
Count = 10
Count = 9
Count = 8
Count = 7
Count = 6
Count = 5
Count = 4
Count = 3
```

Exercise 3: Stopwatch Project



Create a new project

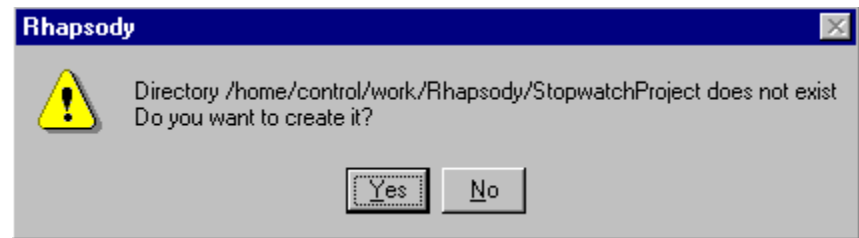
New Project

Project name:

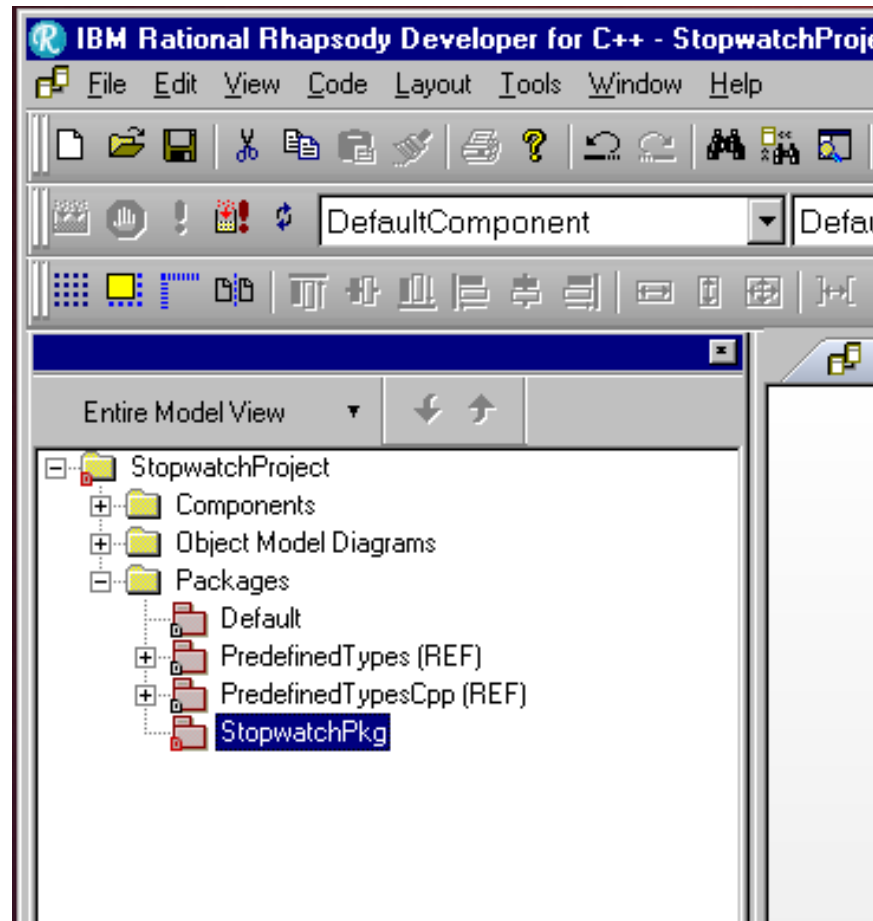
In folder:

Project Type: ▼

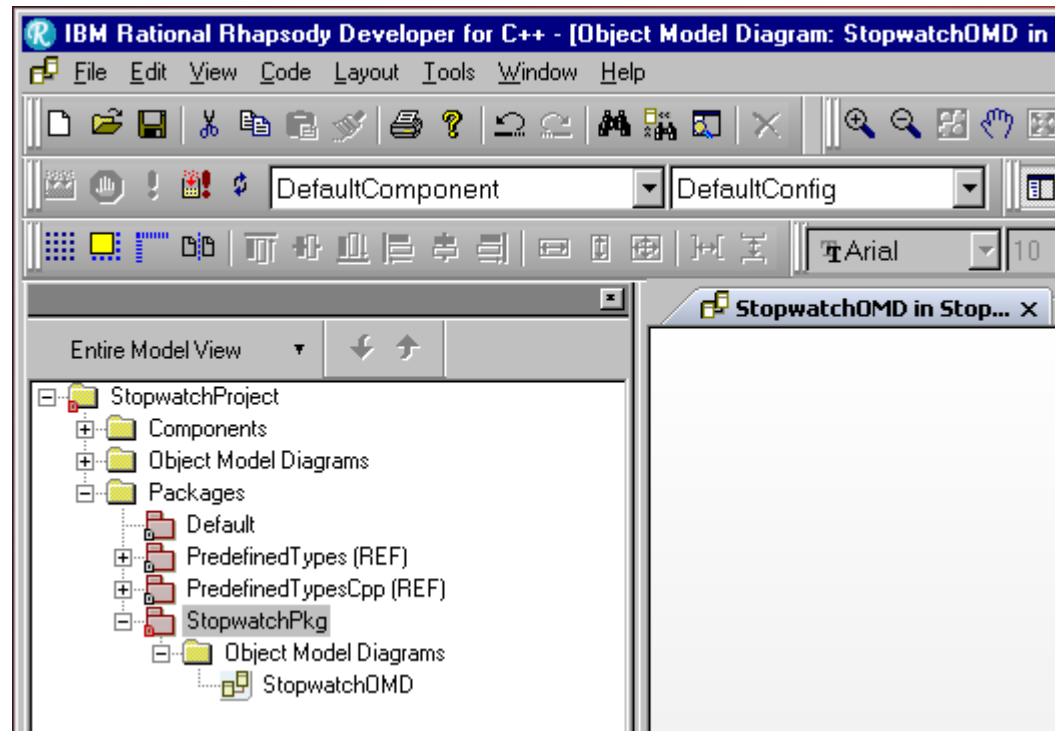
Project Settings: ▼



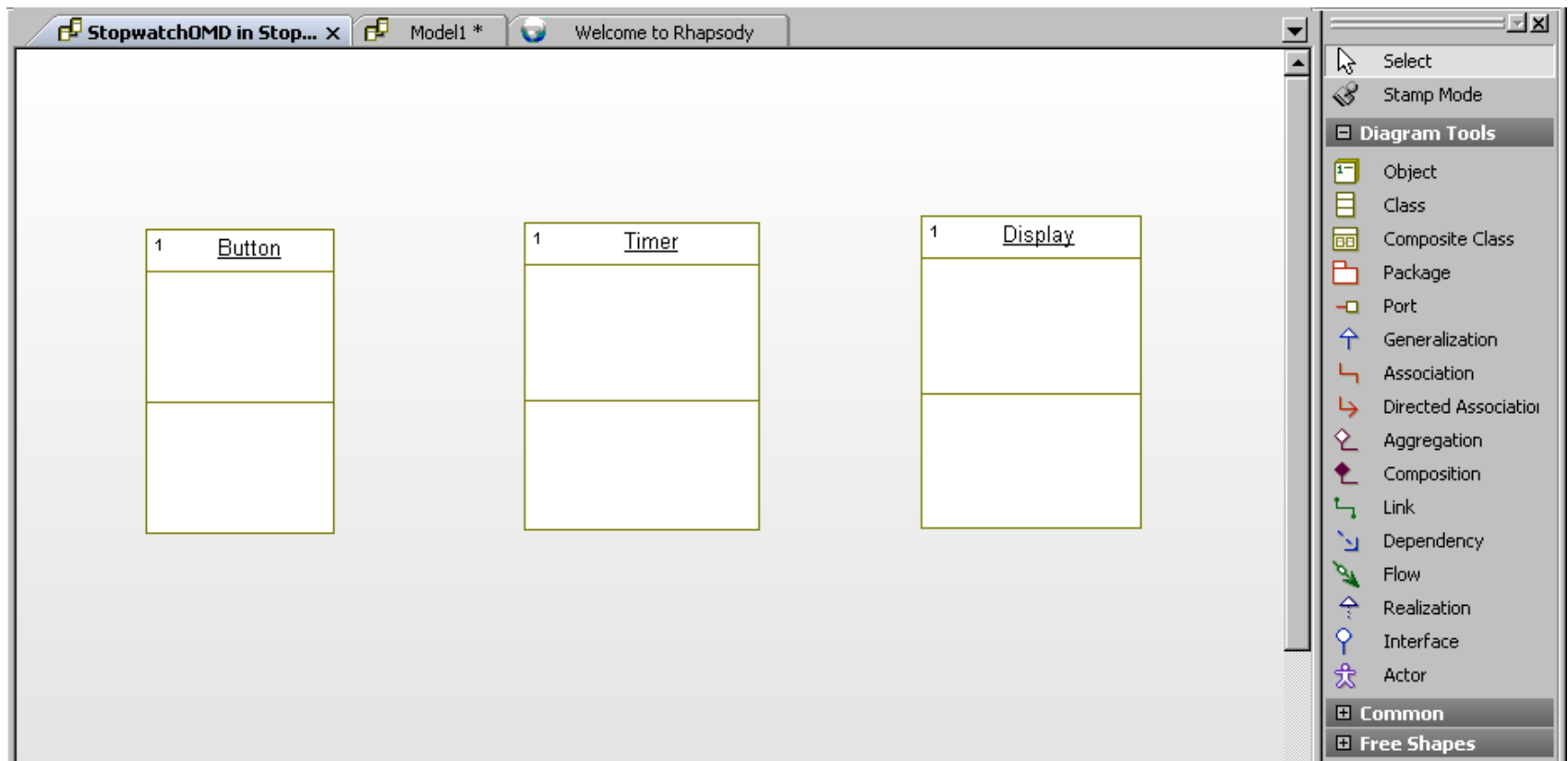
- Right click Packages and Add New Package
- Change the name to StopwatchPkg



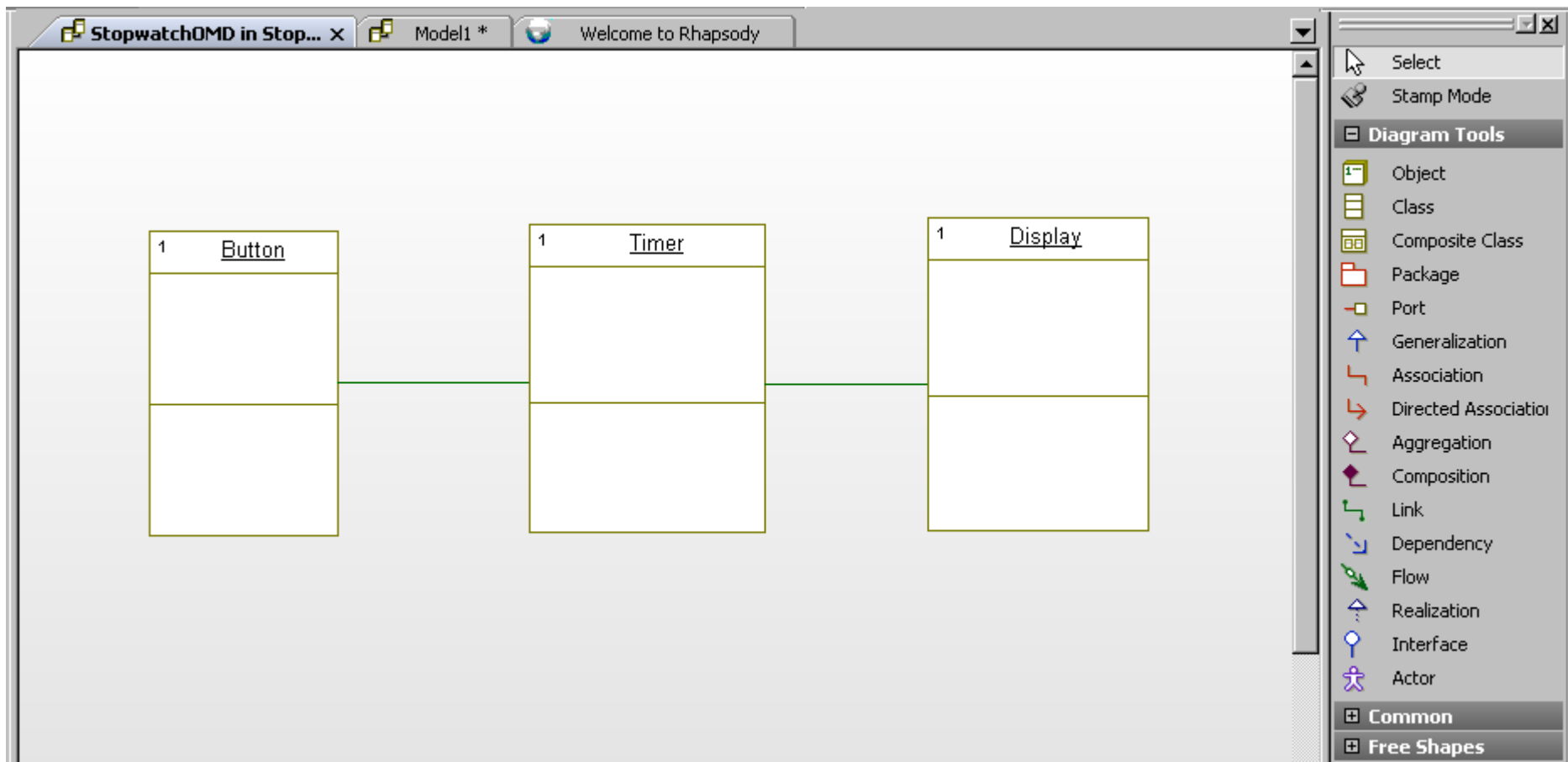
- Right click StopwatchPkg and Add New-Diagrams-Object Model Diagram
- Change the name to StopwatchOMD



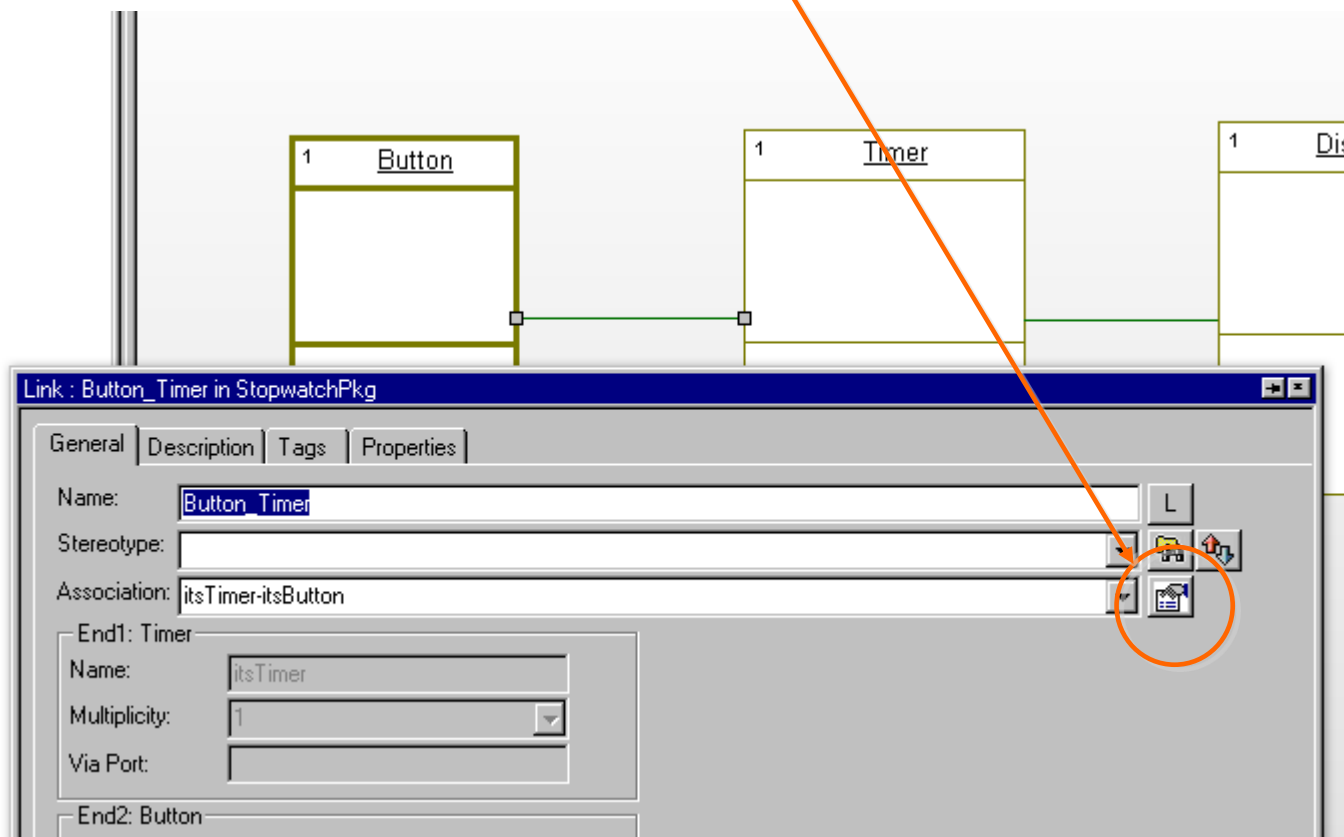
- Select Object in Diagram Tools
- Draw three objects, Button, Timer, Display



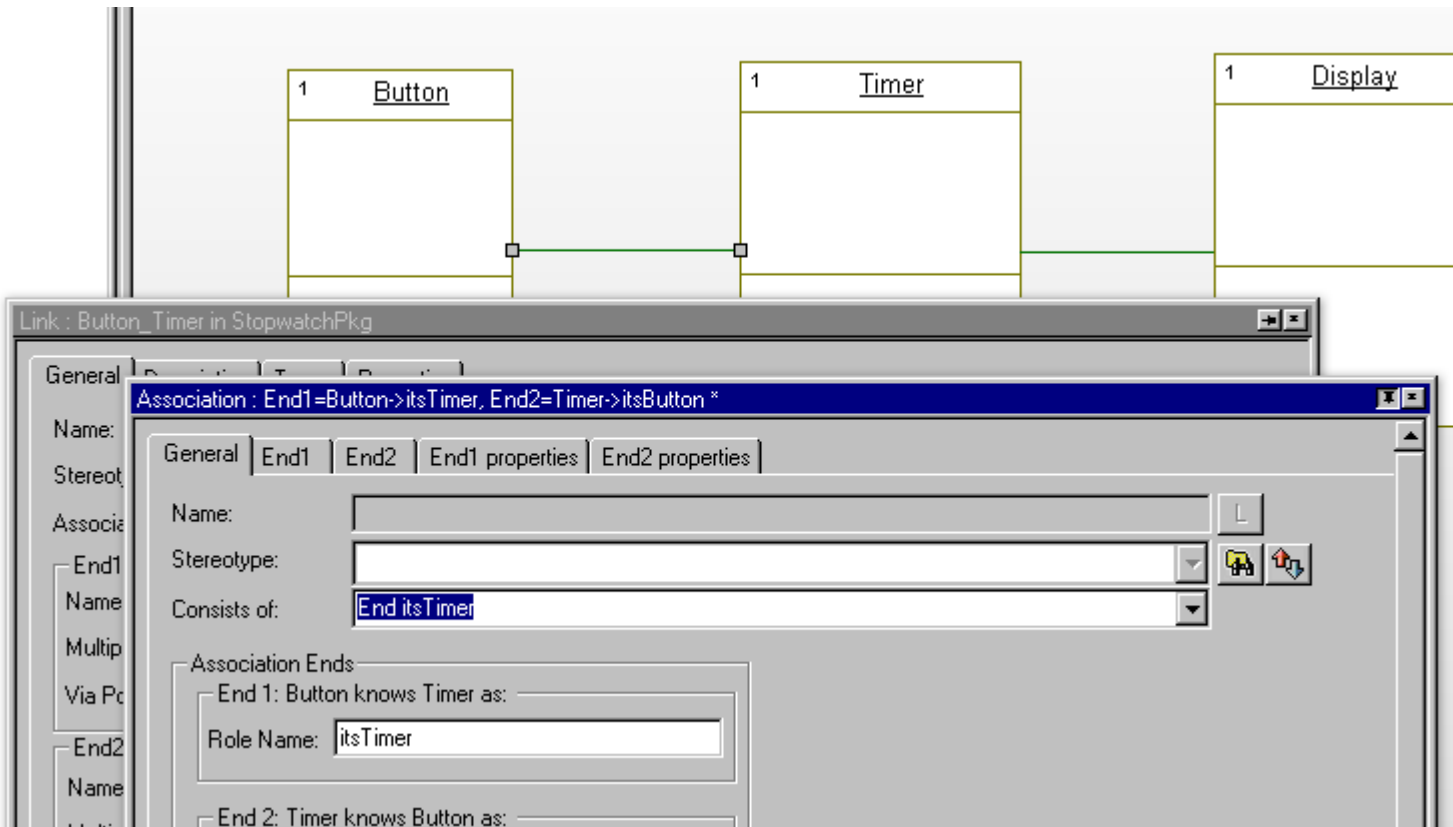
- Select Link in Diagram Tools and draw links



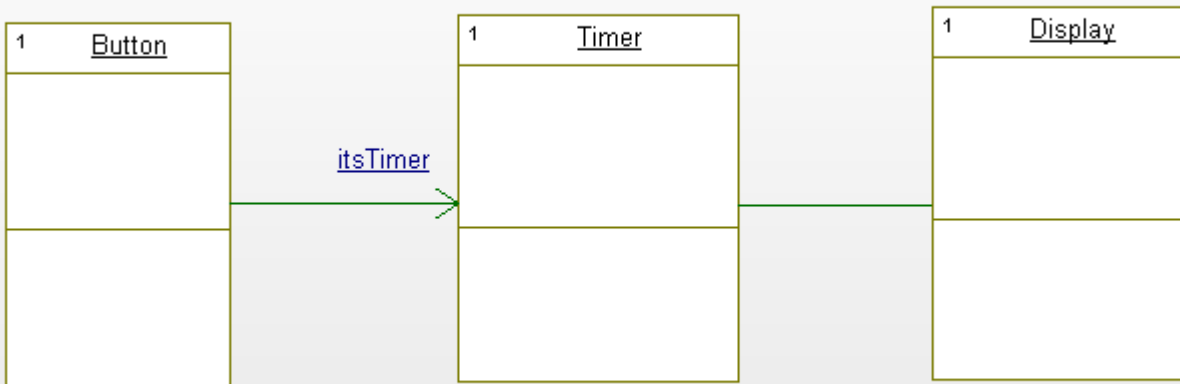
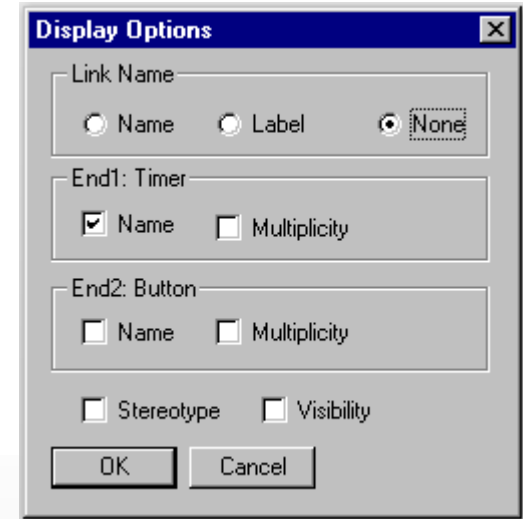
- Double click the link between Button and Timer
- Click Association change button



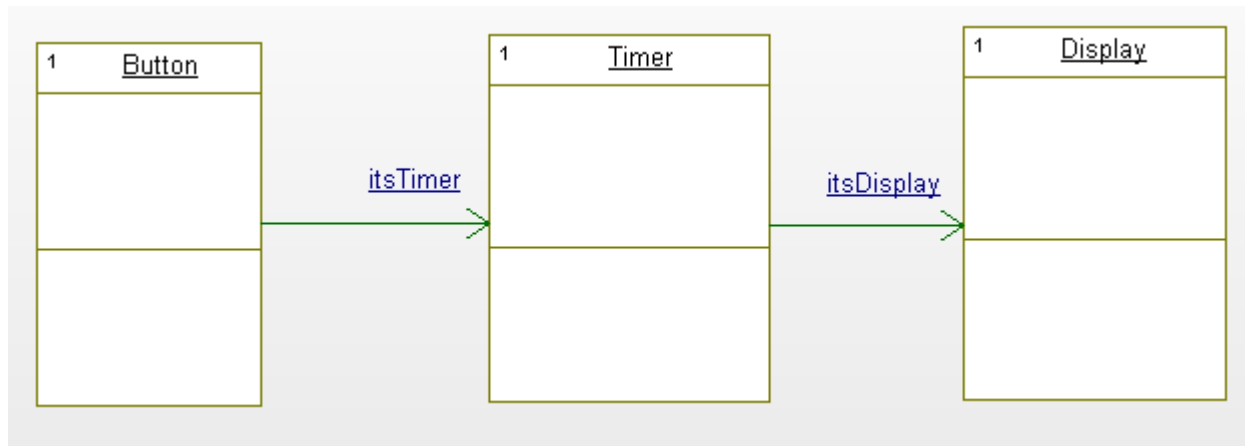
- Change Both Ends to End itsTimer



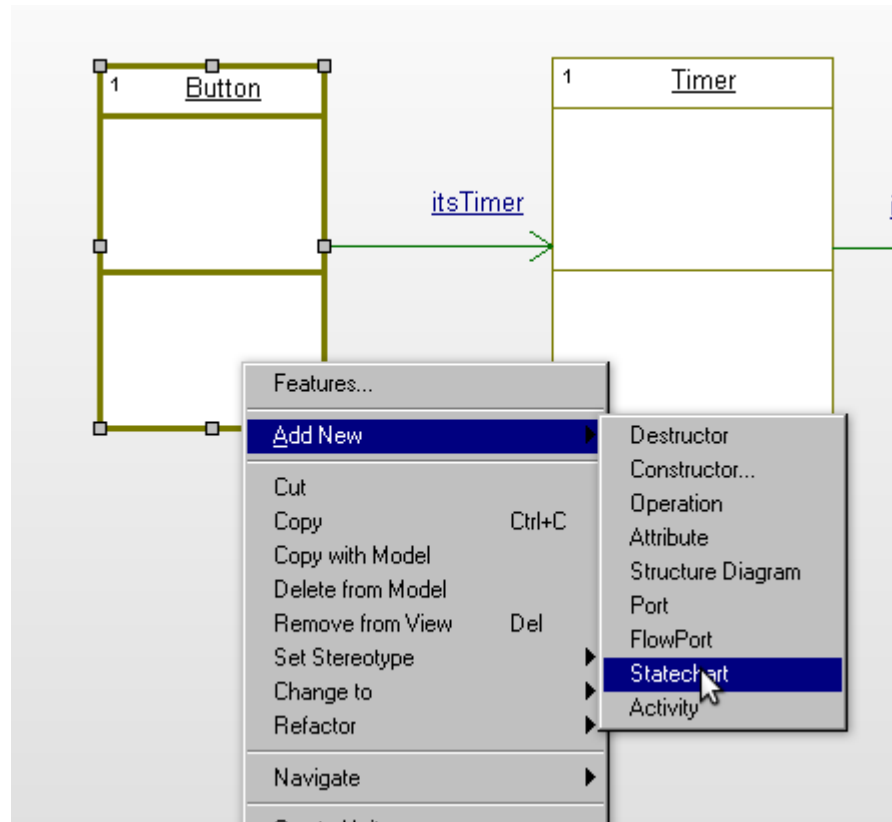
- Right click the link and change Display options



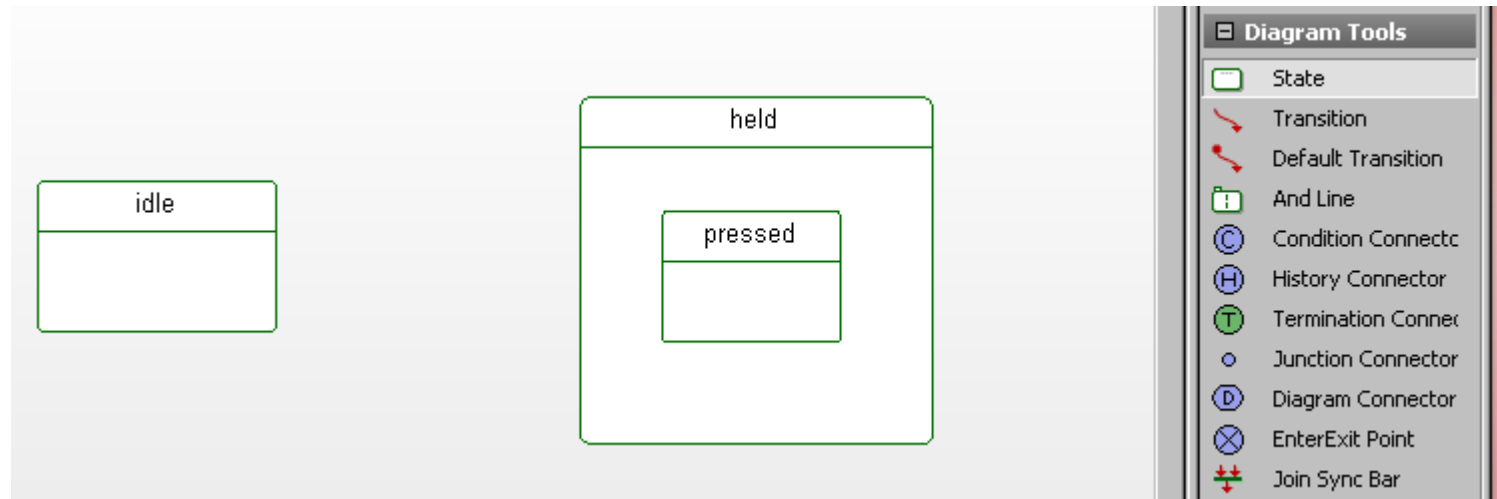
- Repeat the same for the link between Timer and Display



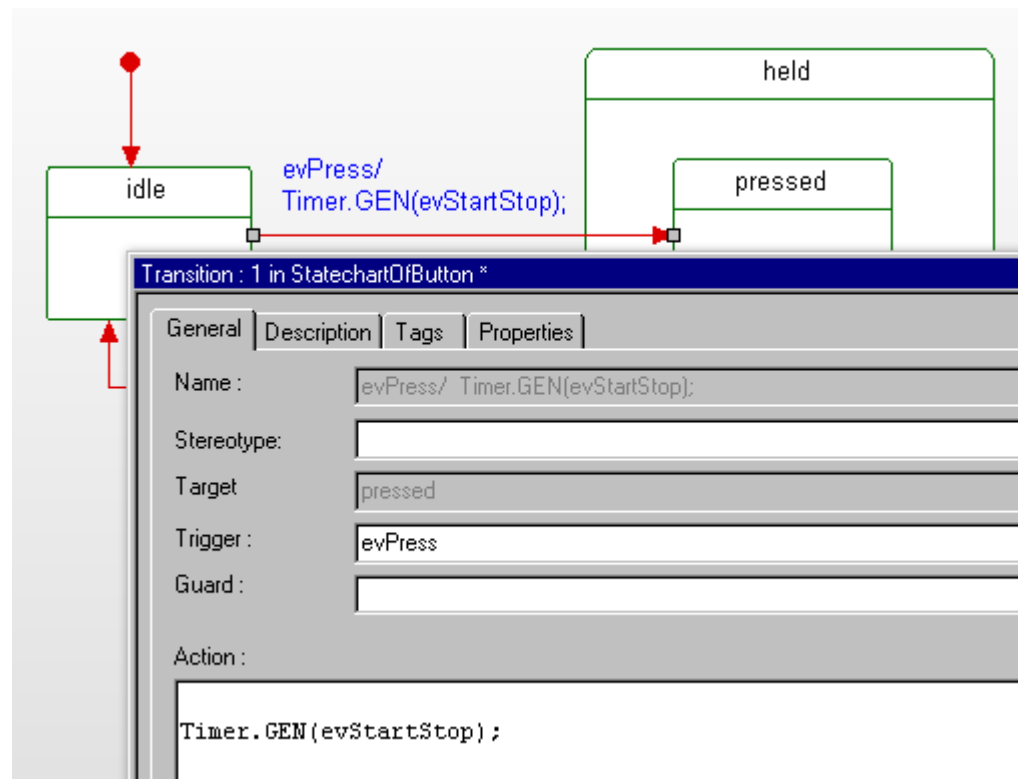
- Select Button object and right click
- Add New-Statechart



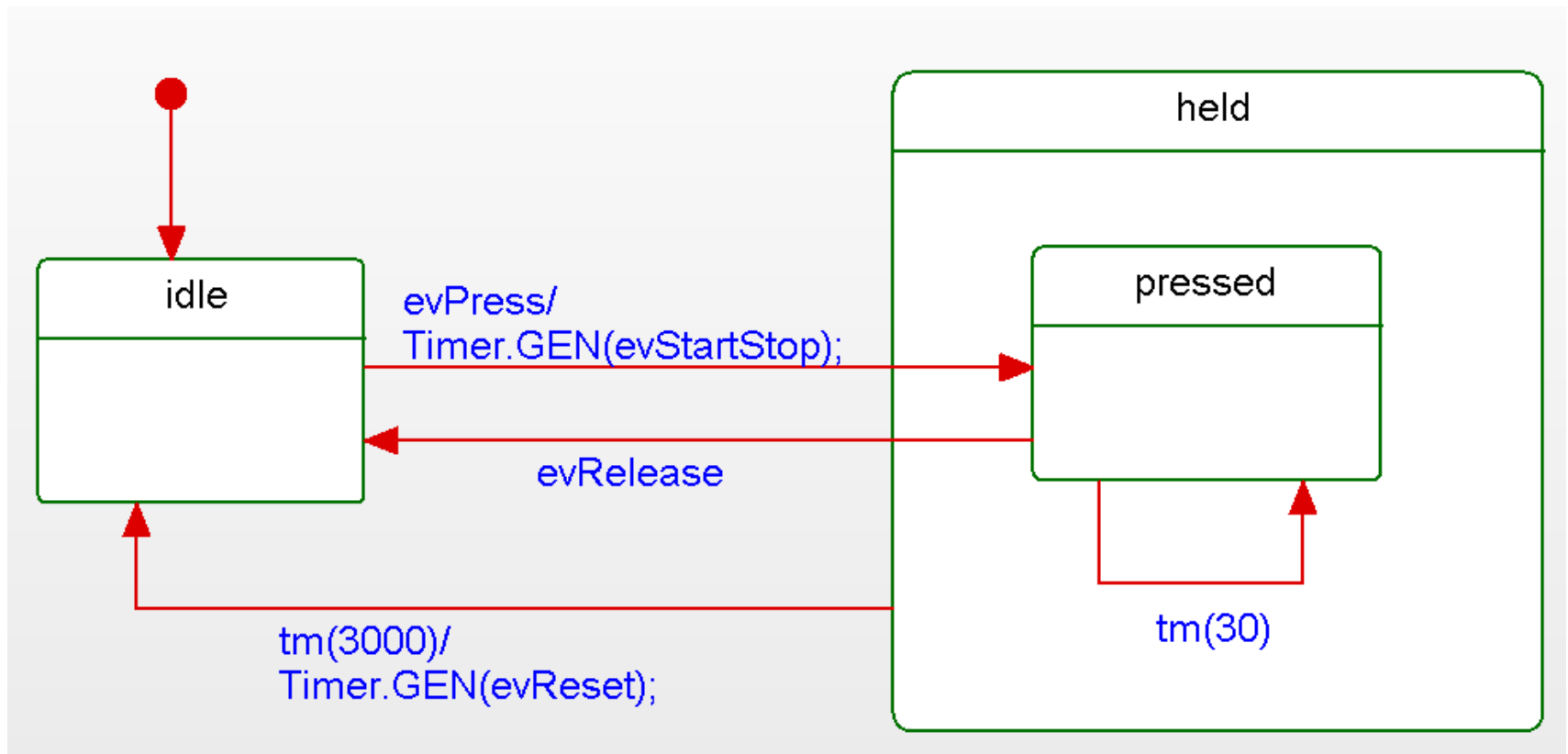
- Select State and draw states.



- Select Transition and draw
- Double click the transition
- Type in Trigger: evPress
- Type in Action: Timer.GEN(evStartStop);



- Complete the statechart



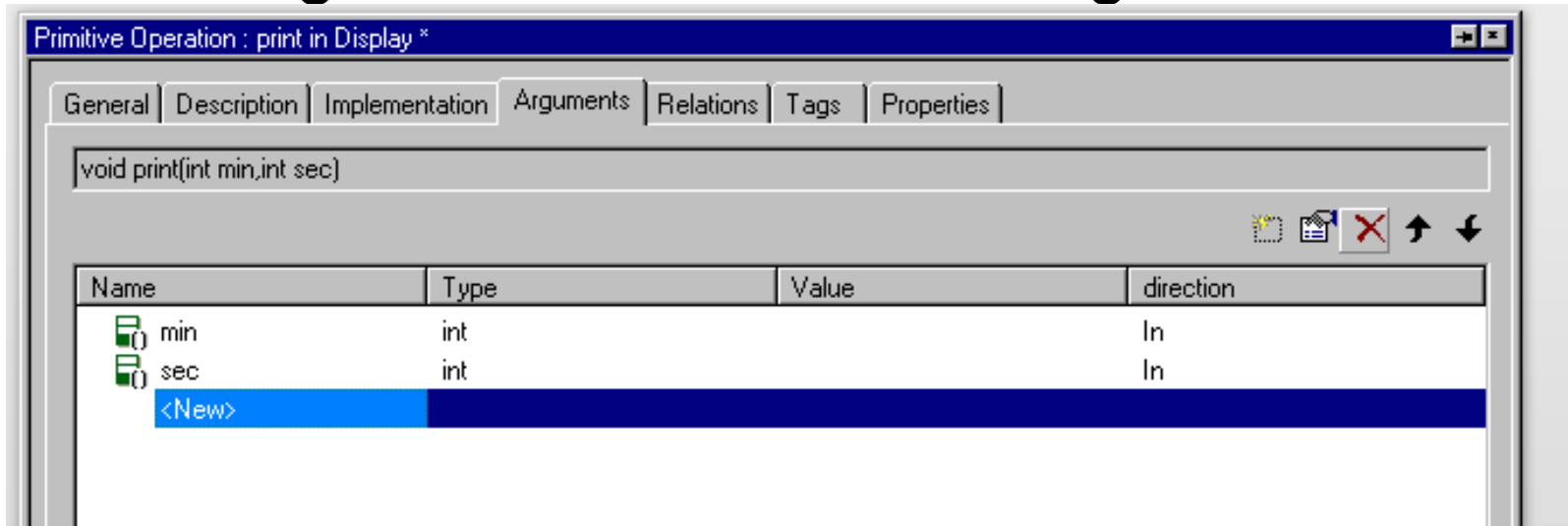
- Double click Display object and open Feature window
- Select Operations Tab and press New
- Select Primitive Operations and name print

The image shows a UML diagram and a Feature window. The UML diagram has two lifelines: 'Timer' and 'Display'. A message arrow labeled 'itsDisplay' points from 'Timer' to 'Display'. The 'Display' lifeline has a box containing 'print():void'. Below the diagram is a Feature window titled 'Object : Display in StopwatchPkg'. The 'Operations' tab is selected. A table lists the operations:

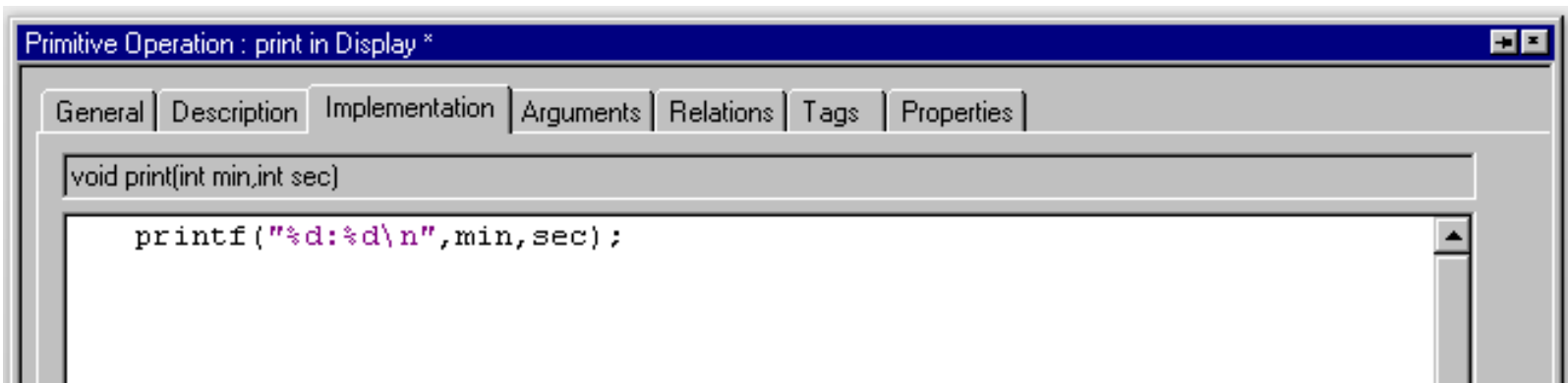
Name	Visibility	Return Type
print <New>	Public	void

Buttons at the bottom of the window include 'Locate', 'OK', and 'Apply'.

- Double click print operation
- Select Arguments Tab and add arguments: min, sec.



- Select Implementation and type in code



- Double click Timer object
- Select Attributes Tab
- Add attributes(seconds, minutes)

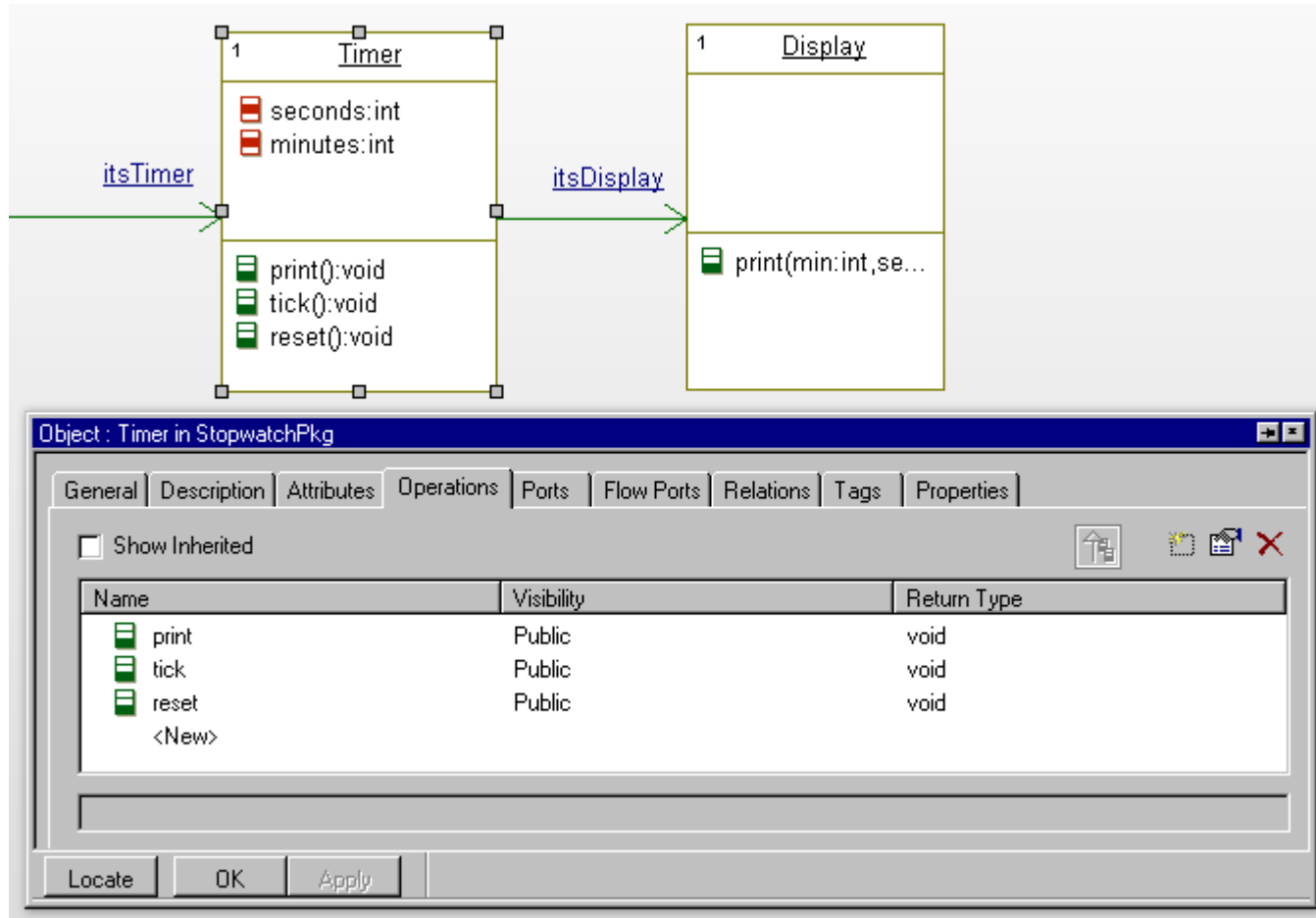
The image shows a UML class diagram and a Properties window for a Timer class. The class diagram features two classes: 'Timer' and 'Display'. The 'Timer' class has two attributes: 'seconds:int' and 'minutes:int'. The 'Display' class has one operation: 'print(min:int,se...)'. A green arrow labeled 'itsTimer' points to the 'Timer' class, and another green arrow labeled 'itsDisplay' points from the 'Timer' class to the 'Display' class.

Below the diagram is a Properties window titled 'Object : Timer in StopwatchPkg'. The 'Attributes' tab is selected. The window shows a table of attributes:

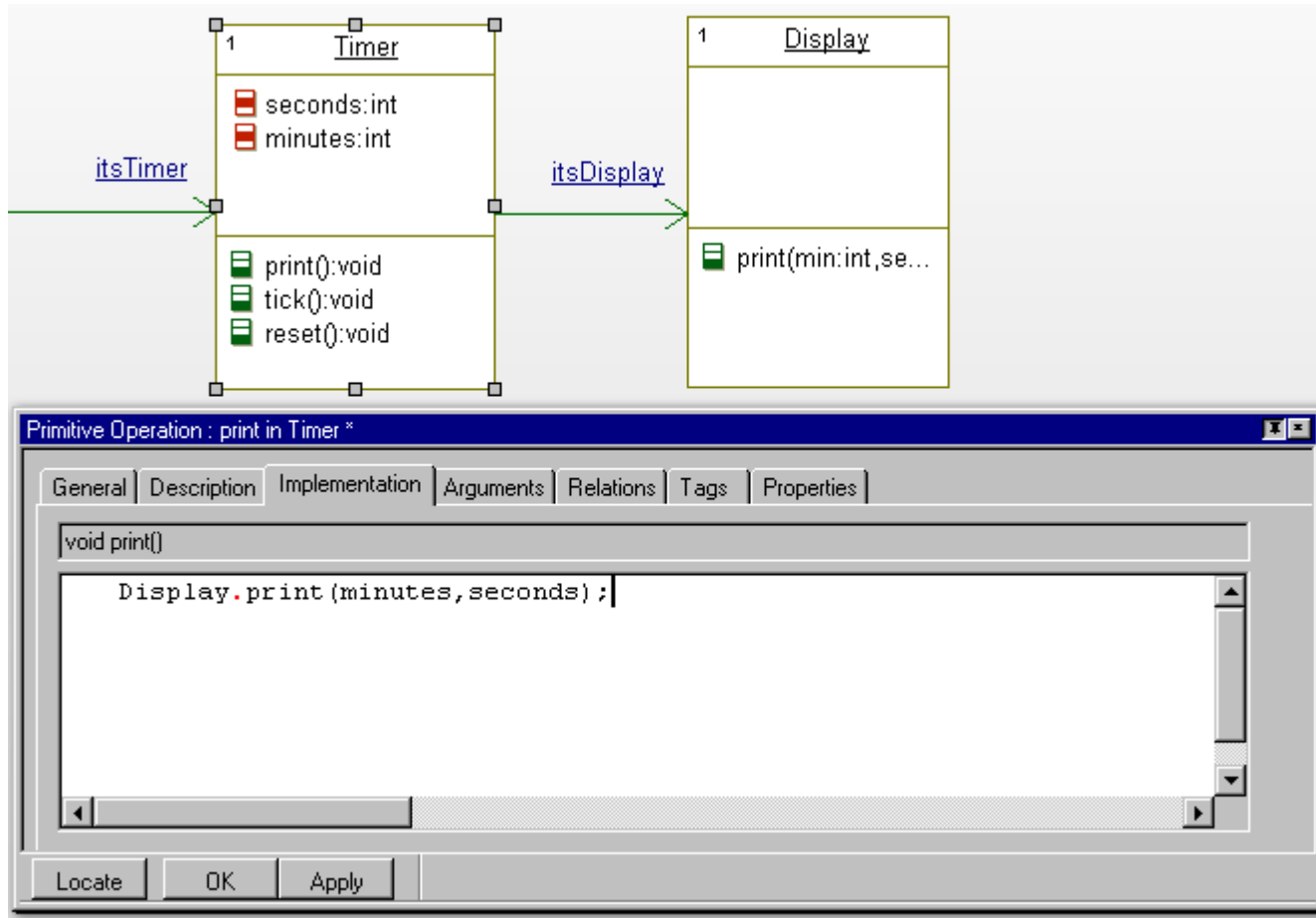
Name	Visibility	Type	Initial Value	Value
seconds	Public	int	0	
minutes	Public	int	0	
<New>				

Buttons at the bottom of the window include 'Locate', 'OK', and 'Apply'.

- Add operations(print, tick, reset) in Timer object



- Double click print operation
- Add Implementations



- Double click tick operation
- Add Implementations

The image shows a UML class diagram and a dialog box for implementing a primitive operation. The class diagram features two classes: **Timer** and **Display**. The **Timer** class has two attributes: `seconds:int` and `minutes:int`, and three operations: `print():void`, `tick():void`, and `reset():void`. The **Display** class has one operation: `print(min:int,se...`. A dependency arrow labeled `itsTimer` points from an external source to the **Timer** class. Another dependency arrow labeled `itsDisplay` points from the **Timer** class to the **Display** class.

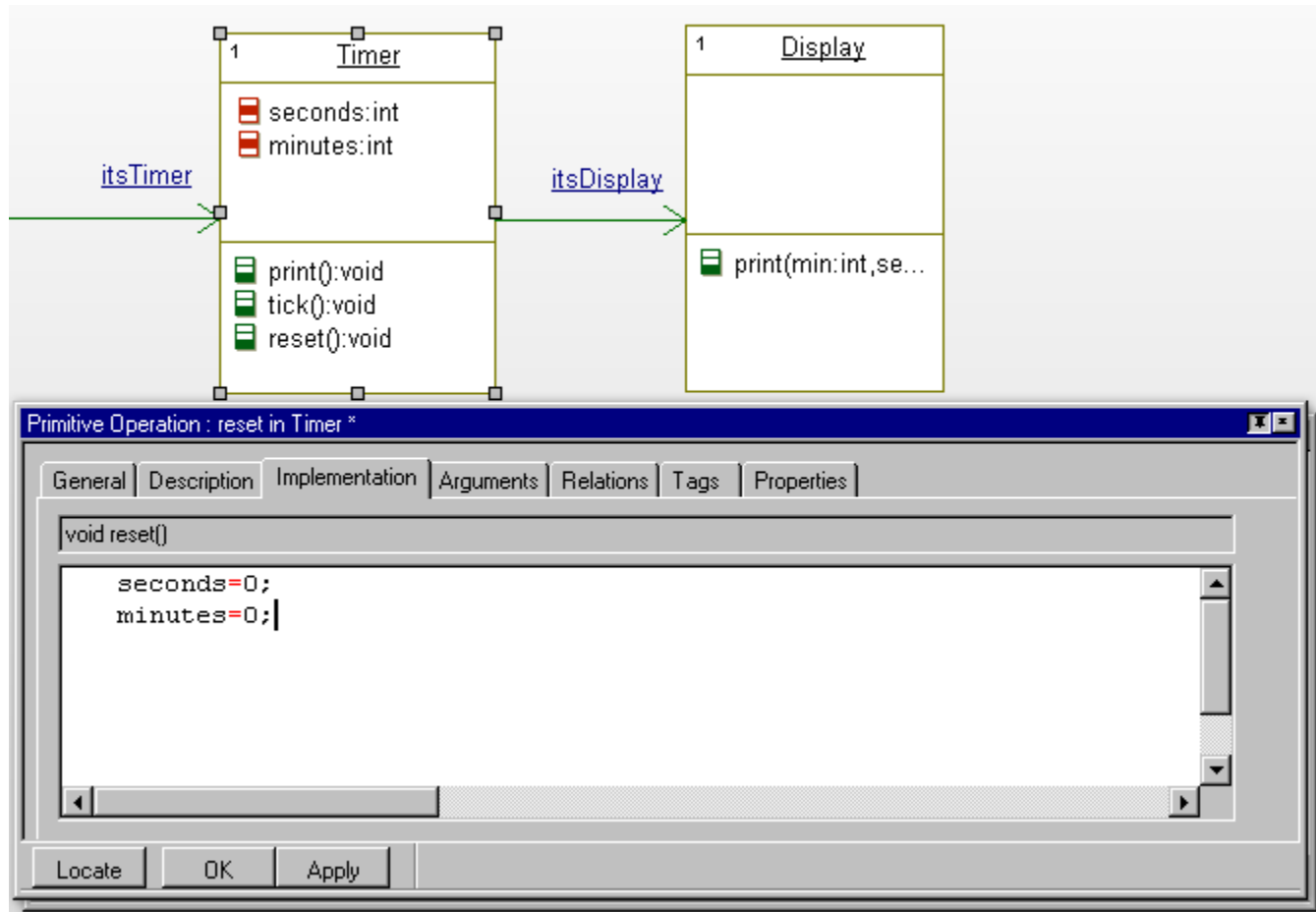
The dialog box, titled "Primitive Operation : tick in Timer *", shows the implementation of the `void tick()` method. The code is as follows:

```
void tick()  
  
seconds++;  
if ( seconds > 59 ) {  
    seconds=0;  
    minutes++;  
}
```

The dialog box includes tabs for "General", "Description", "Implementation", "Arguments", "Relations", "Tags", and "Properties". At the bottom, there are buttons for "Locate", "OK", and "Apply".

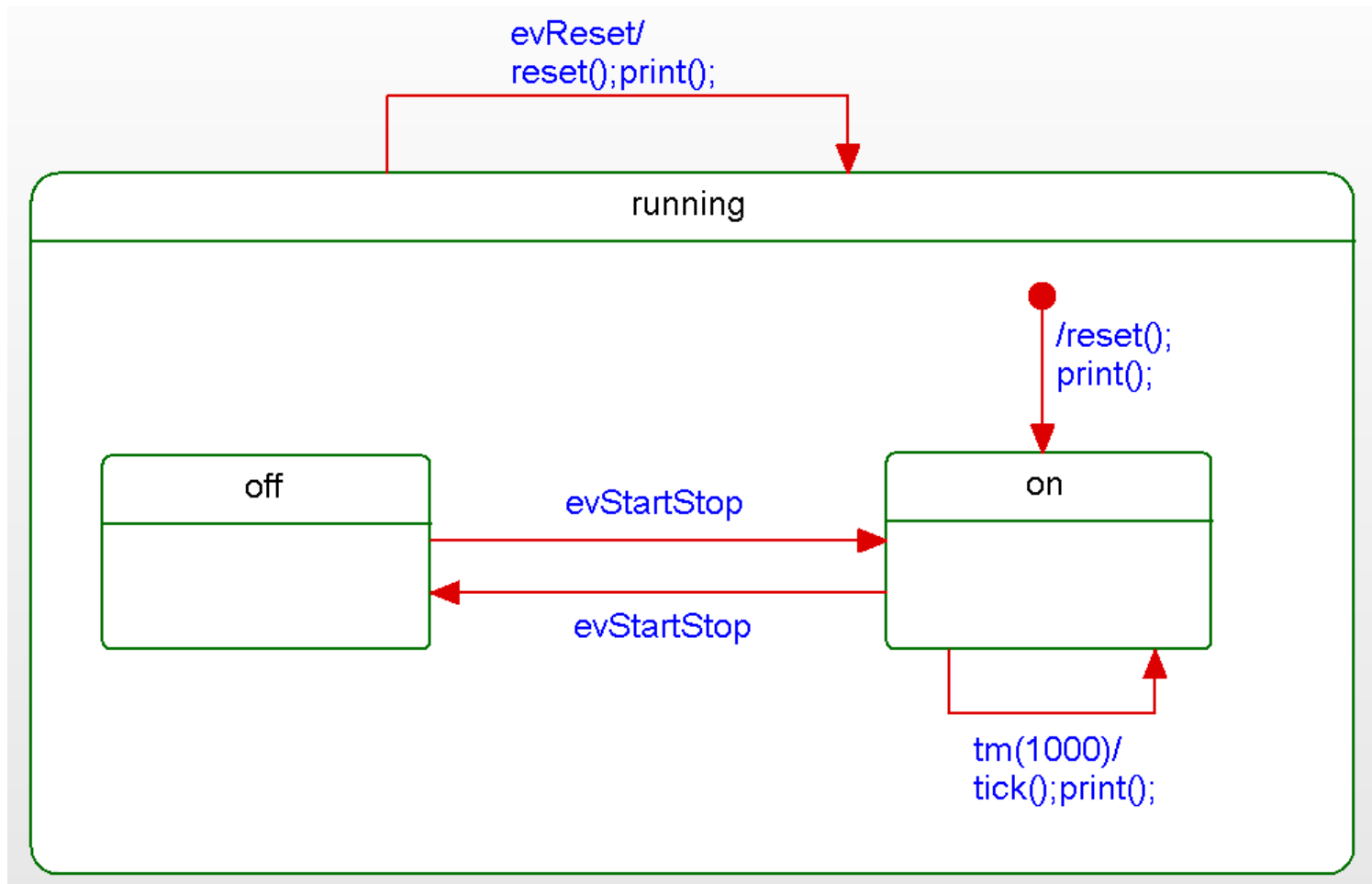
Timer attributes

- Double click reset operation
- Add Implementations



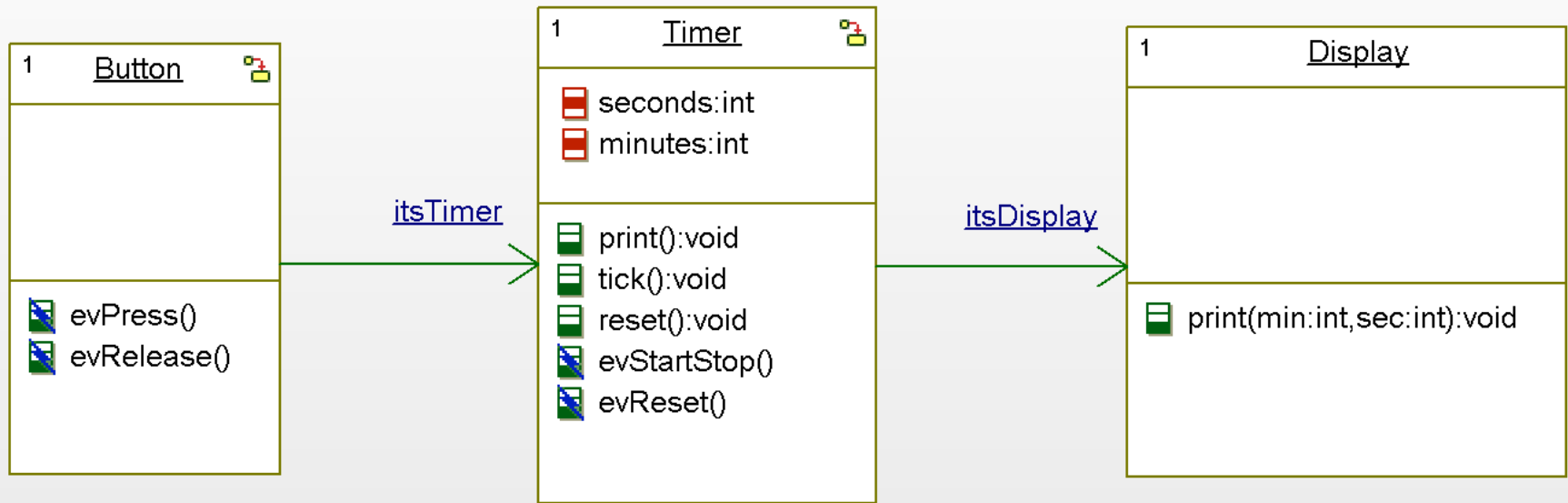
Statechart in Timer

- Right click Timer and Add New-Statechart



Object Model Diagram

■ Stopwatch OMD



Generate and build

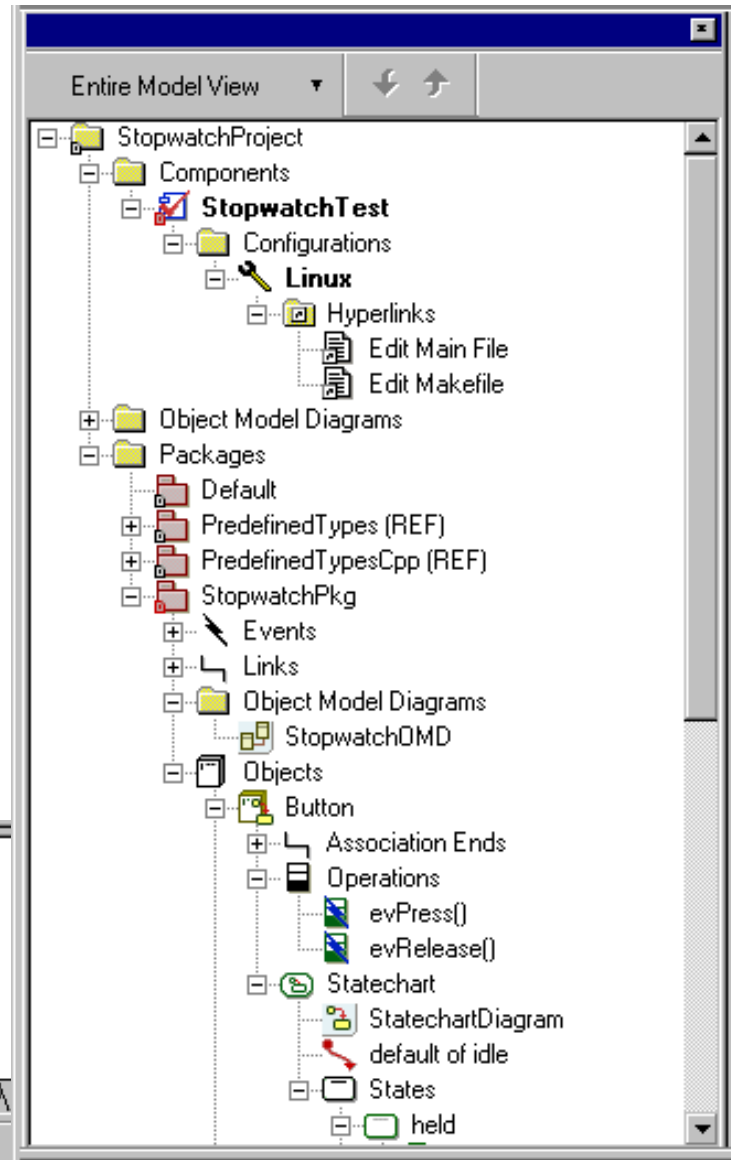
- Change component and configuration names
- Generate and build

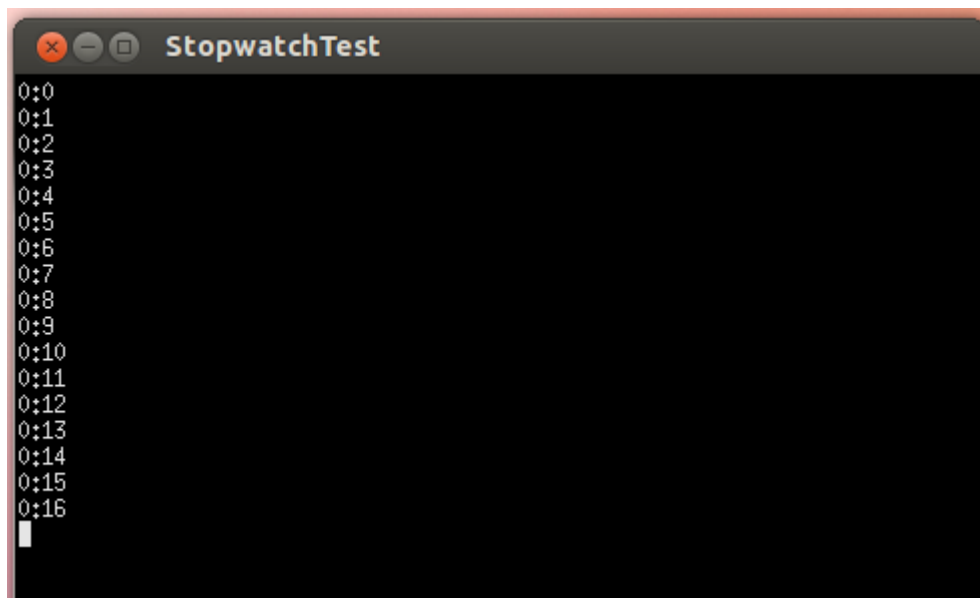
```
Compiling Button.cpp
Compiling Timer.cpp
Compiling Display.cpp
Compiling StopwatchPkg.cpp
Linking StopwatchTest

Build Done
```

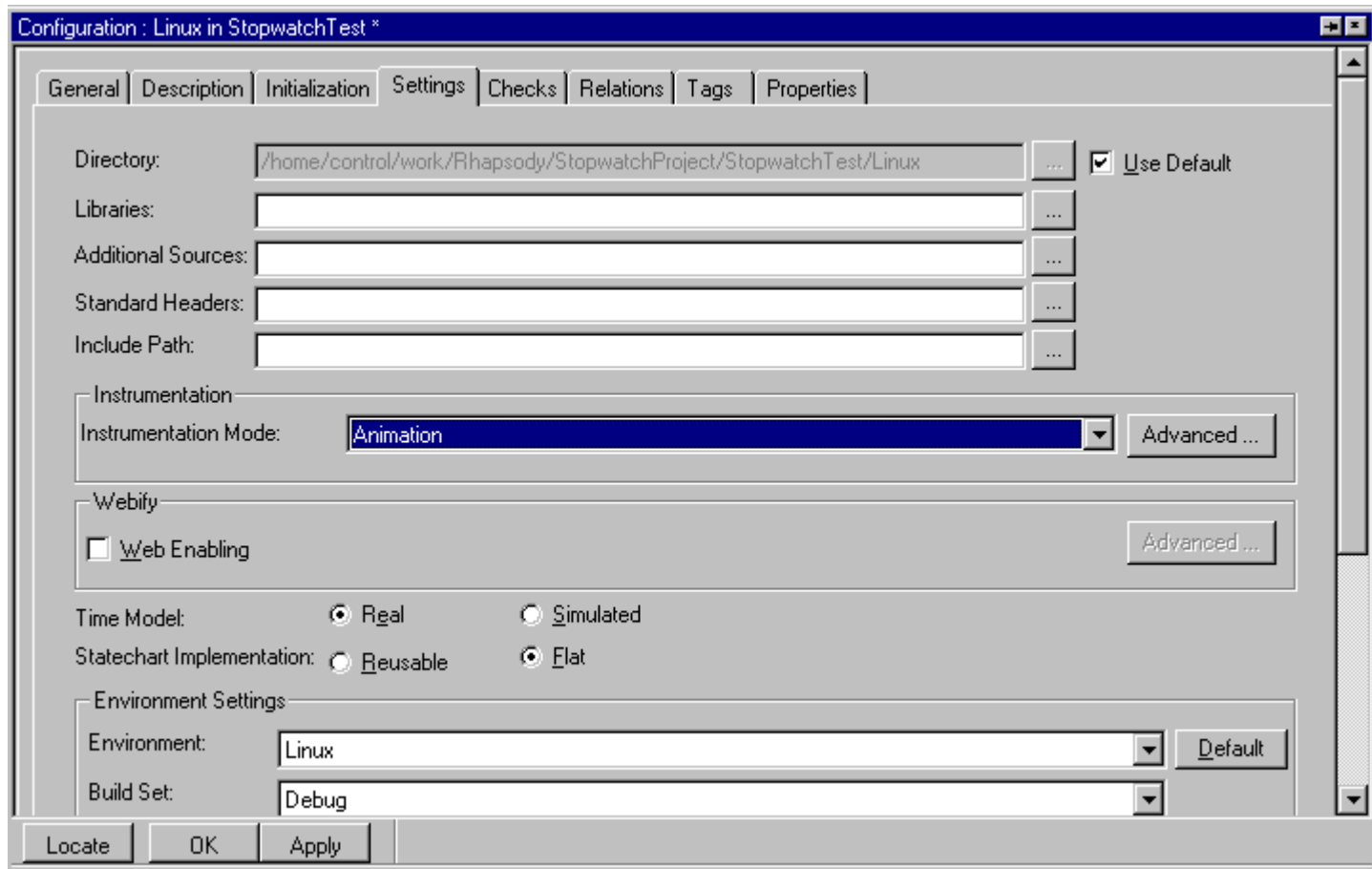
Log Check Model Build Configuration Management

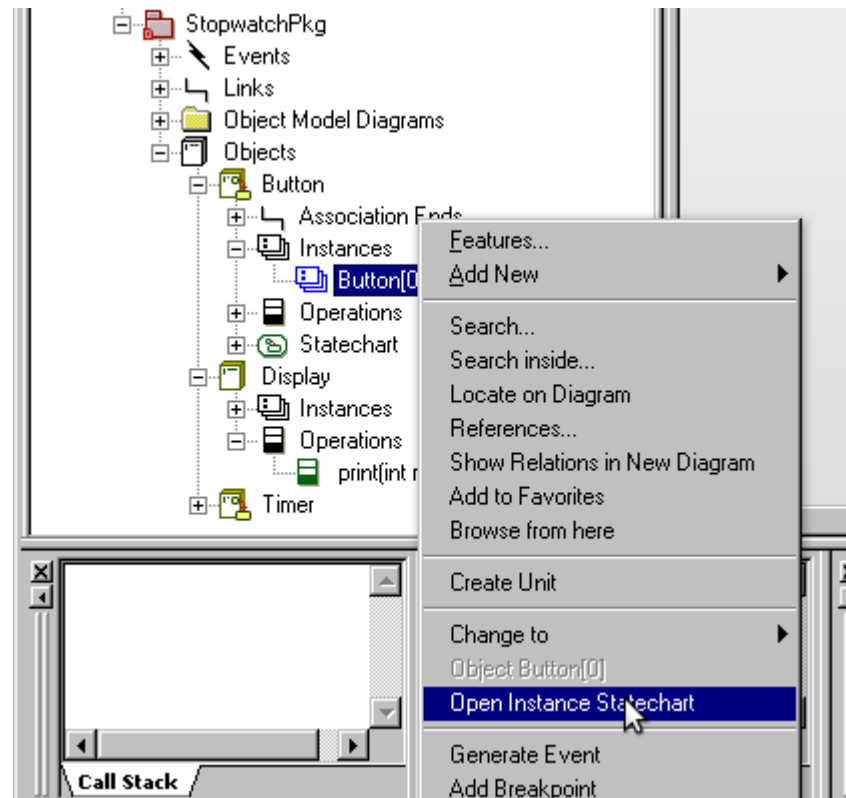
For Help, press F1



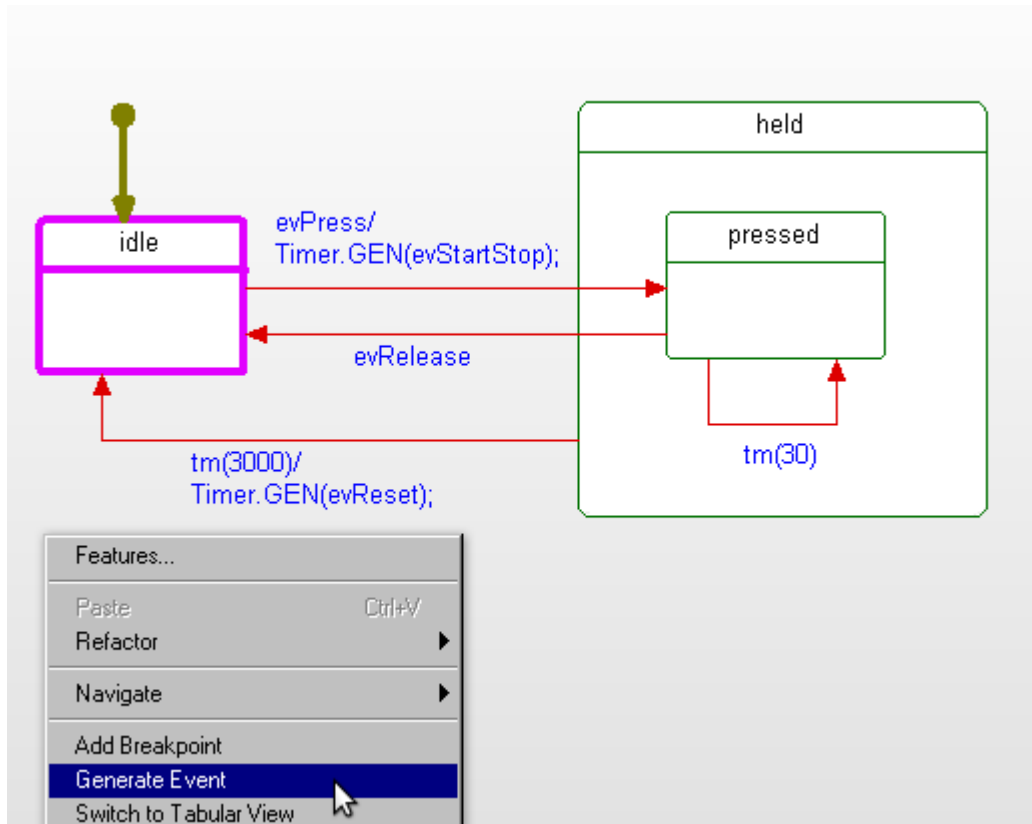


```
StopwatchTest
0:0
0:1
0:2
0:3
0:4
0:5
0:6
0:7
0:8
0:9
0:10
0:11
0:12
0:13
0:14
0:15
0:16
█
```

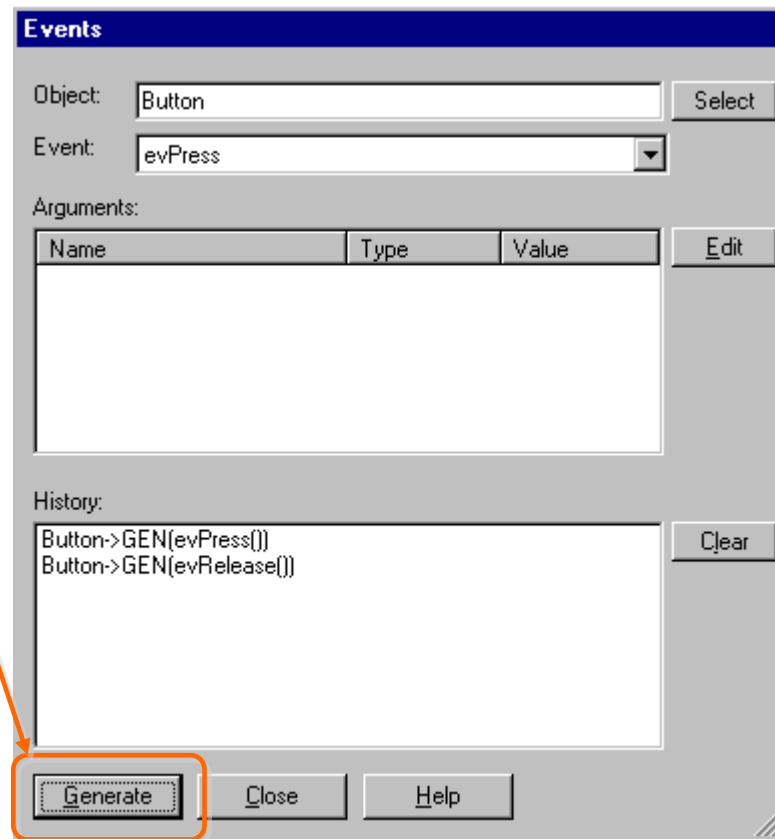




■ Right click and Generate Event

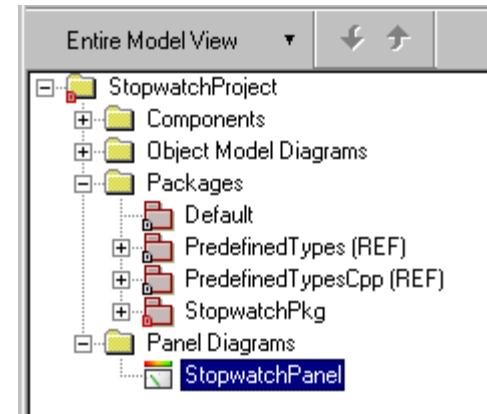
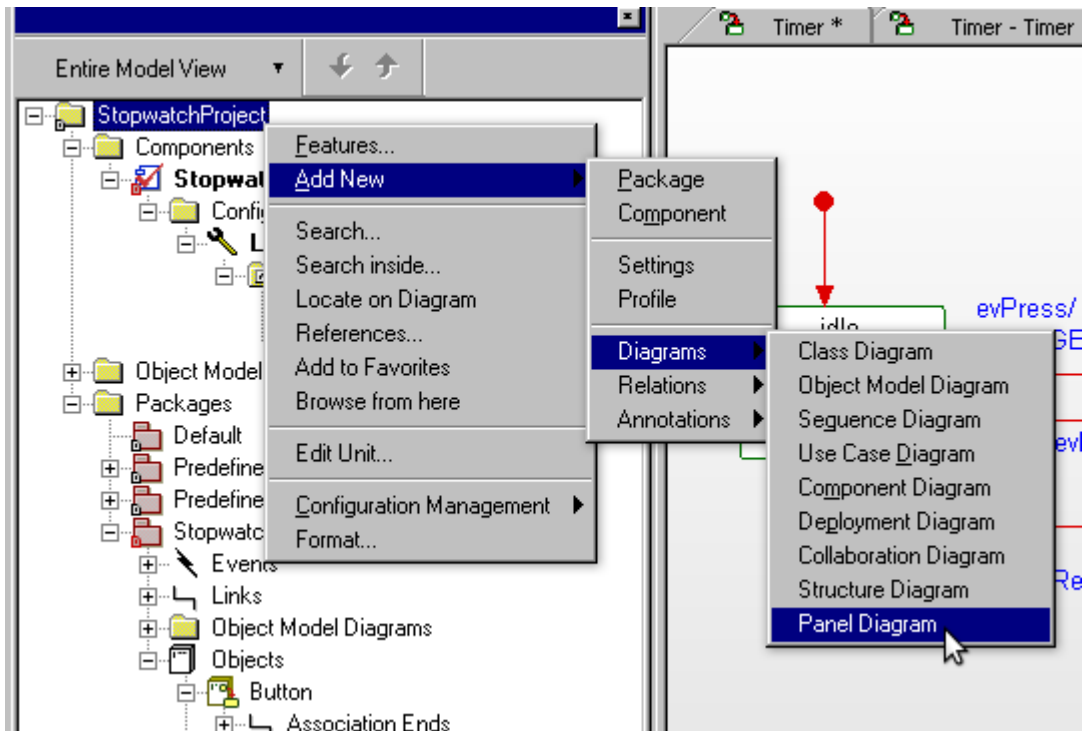


- Generate evPress and evRelease within 3 seconds



Panel Diagram

- One way of testing the model is to use a panel.
- Add a Panel Diagram called Dishwasher Panel.



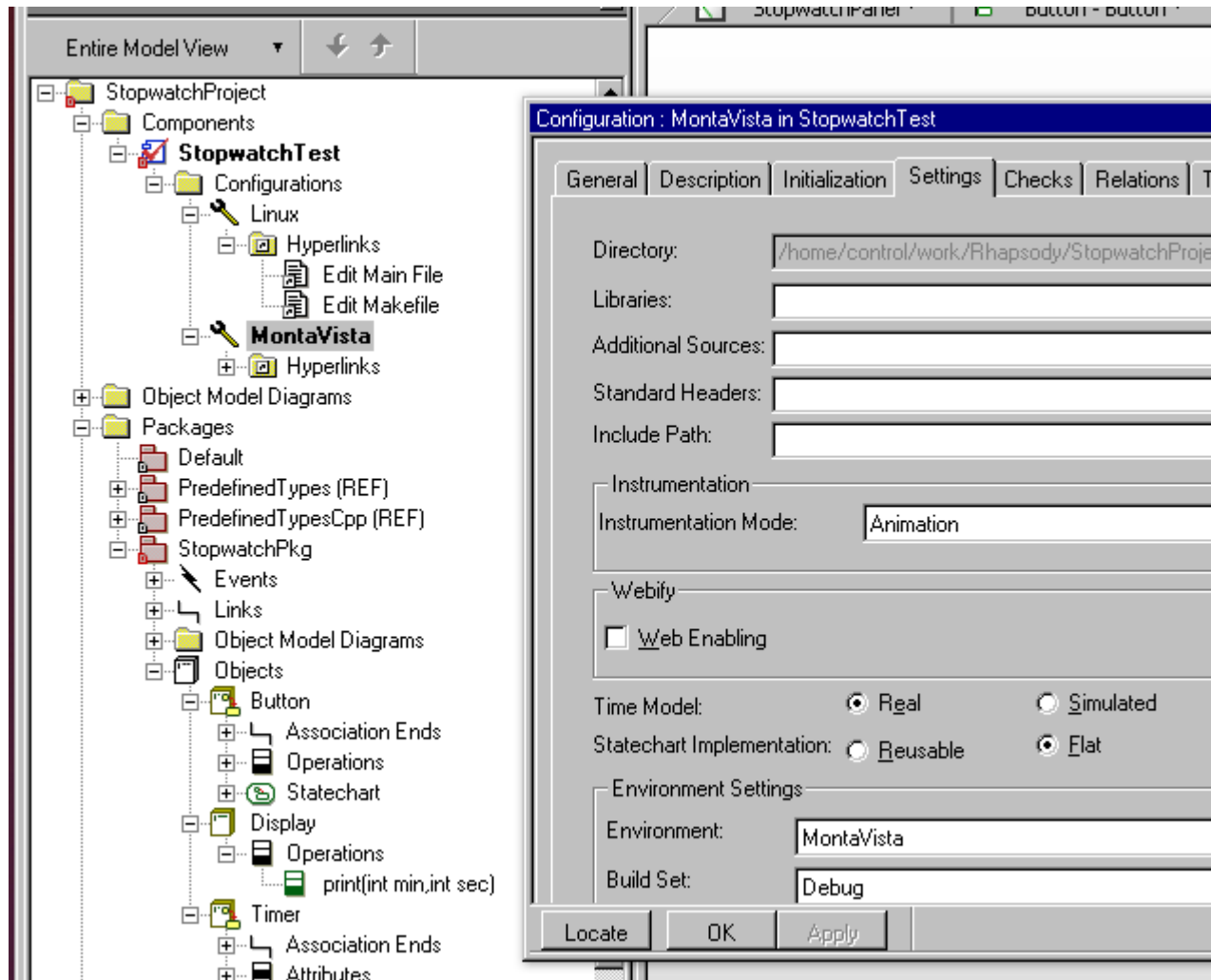
Panel Diagrams can only be used with animation configurations.

Panel Diagram

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

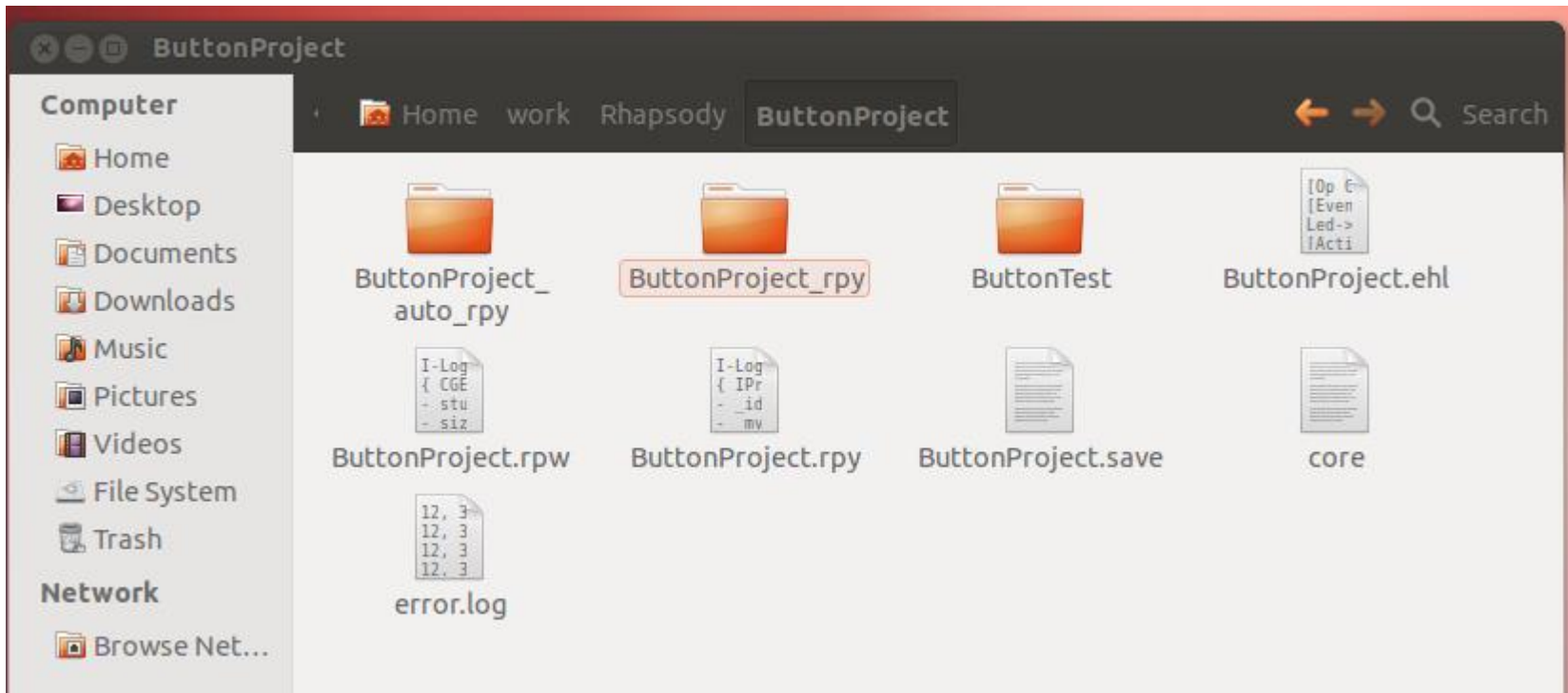
The image displays a software interface for creating a panel diagram. At the top center, a panel diagram is shown with four elements: two digital displays labeled 'digitaldisplay_0' and 'digitaldisplay_1', and two push buttons labeled 'pushbutton_0' and 'pushbutton_1'. The push buttons are labeled 'evPress' and 'evRelease' respectively. To the right of the panel diagram is a 'Diagram Tools' palette containing various UI components: Knob, Gauge, Meter, Level Indicator, Matrix Display, Digital Display, Led, On Off Switch, Push Button, Button Array, Text Box, and Slider. Below the panel diagram are two property windows. The left window is titled 'DigitalDisplay : digitaldisplay_0' and shows the 'Instance Path' as 'StopwatchPkg.Timer.minutes'. The right window is titled 'PushButton : pushbutton_0' and shows the 'Instance Path' as 'StopwatchPkg.Button.evPress'. Both windows have a tree view showing the project structure: StopwatchProject > StopwatchPkg > Timer > seconds/minutes (for the digital display) and StopwatchProject > StopwatchPkg > Button > evPress/evRelease (for the push button).

- Run on the target with or without animation



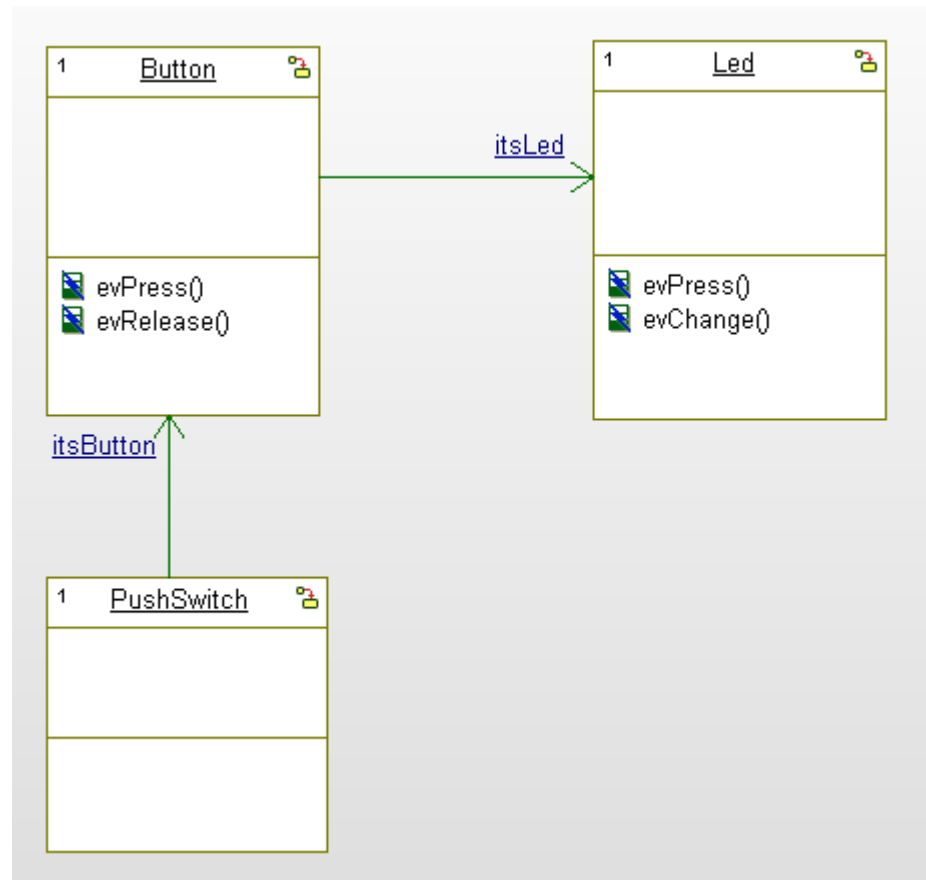
Exercise 4: Button Project

- Run the ButtonProject on the target with or without animation.

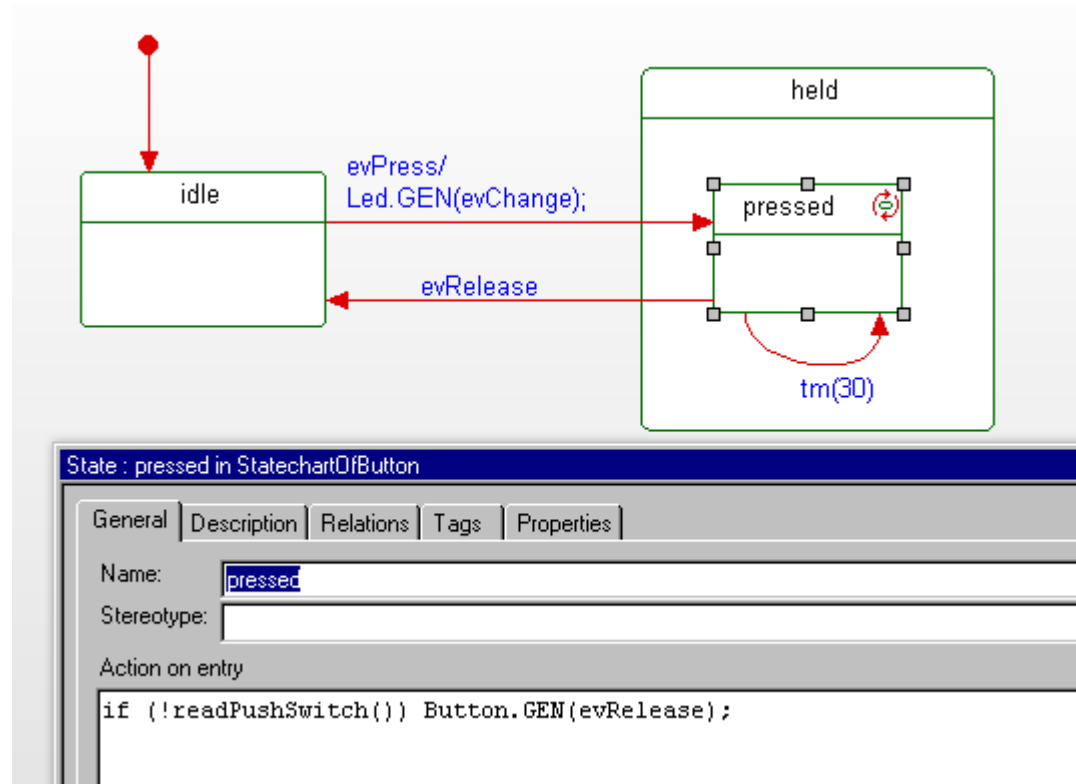


Object Model Diagram

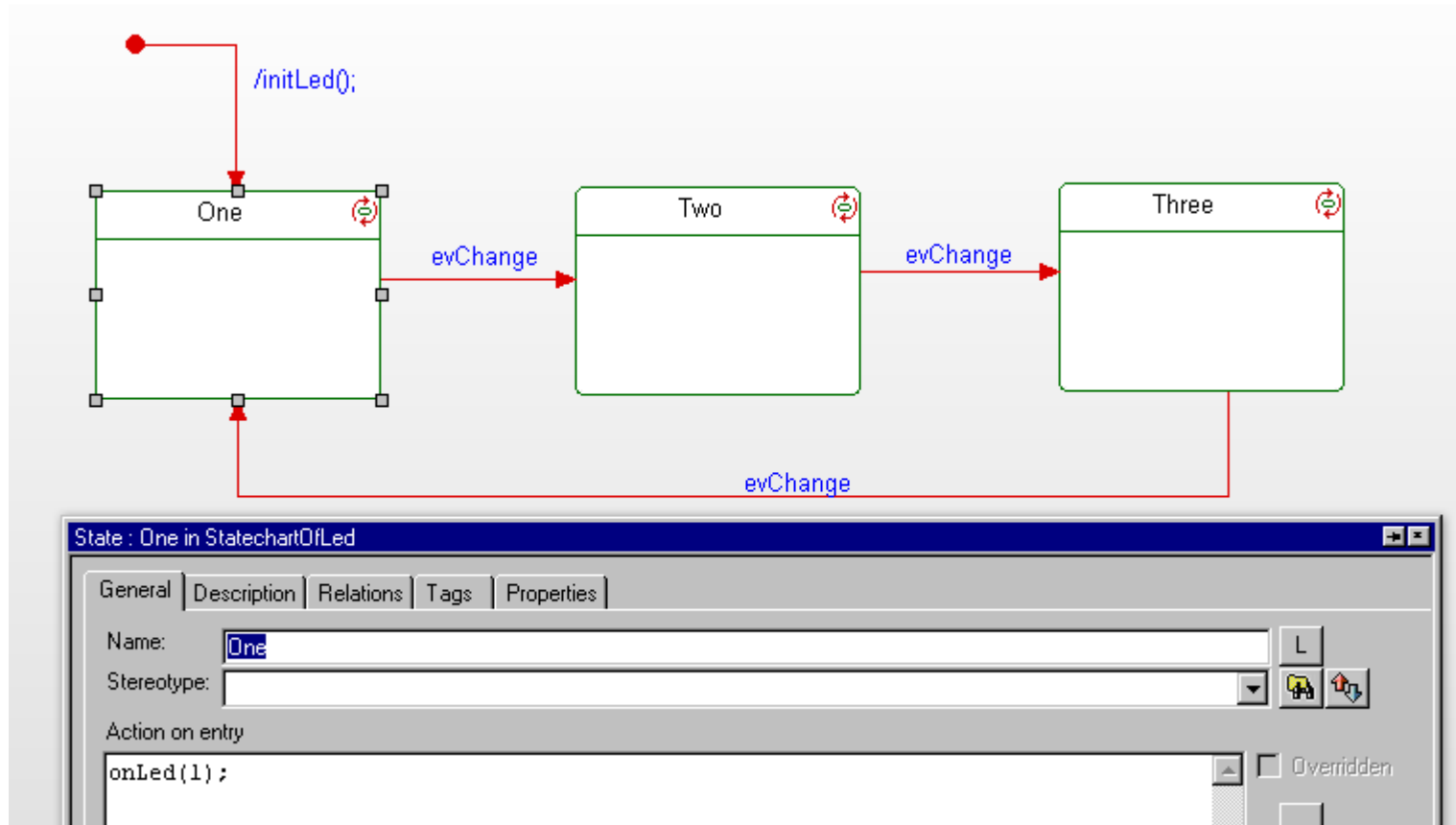
- Object Model Diagram in ButtonProject



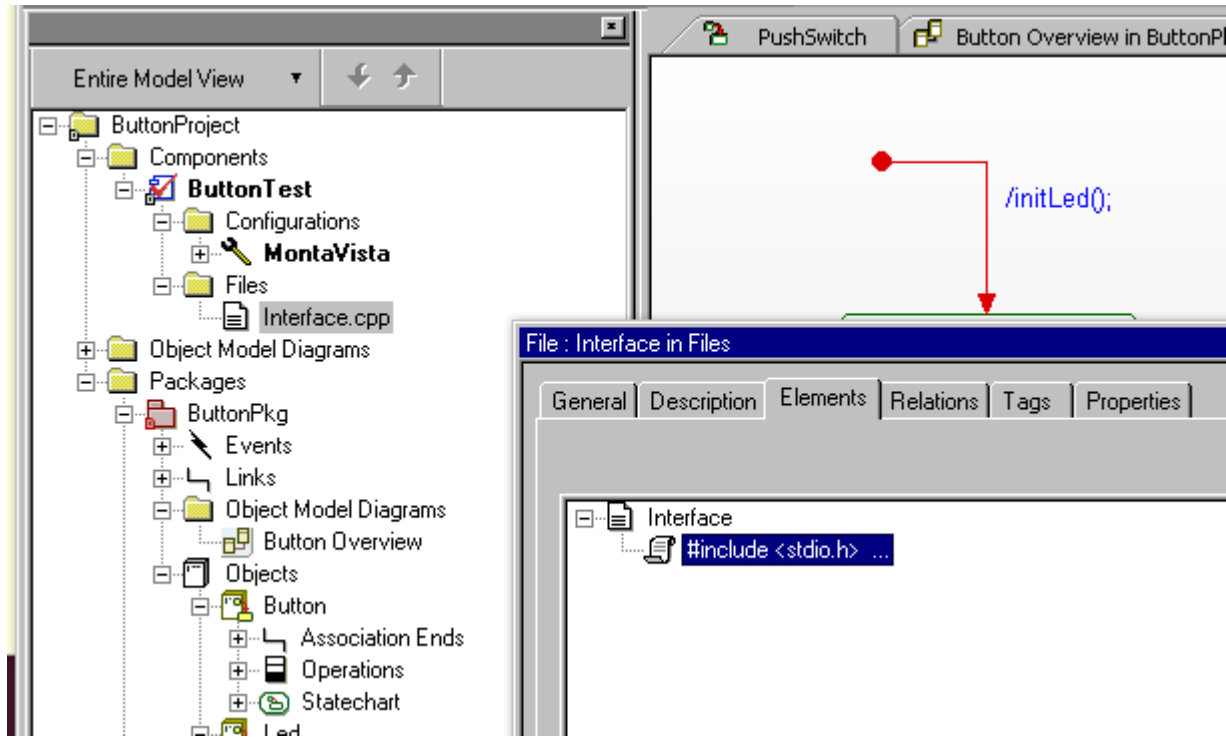
Statechart in Button



Statechart in Led



■ Interface.cpp



Interface.cpp(1)

```
Interface.cpp X
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <signal.h>

#define MAX_BUTTON 9
int dev_switch, dev_led;
int buff_size;
unsigned char push_sw_buff[MAX_BUTTON];

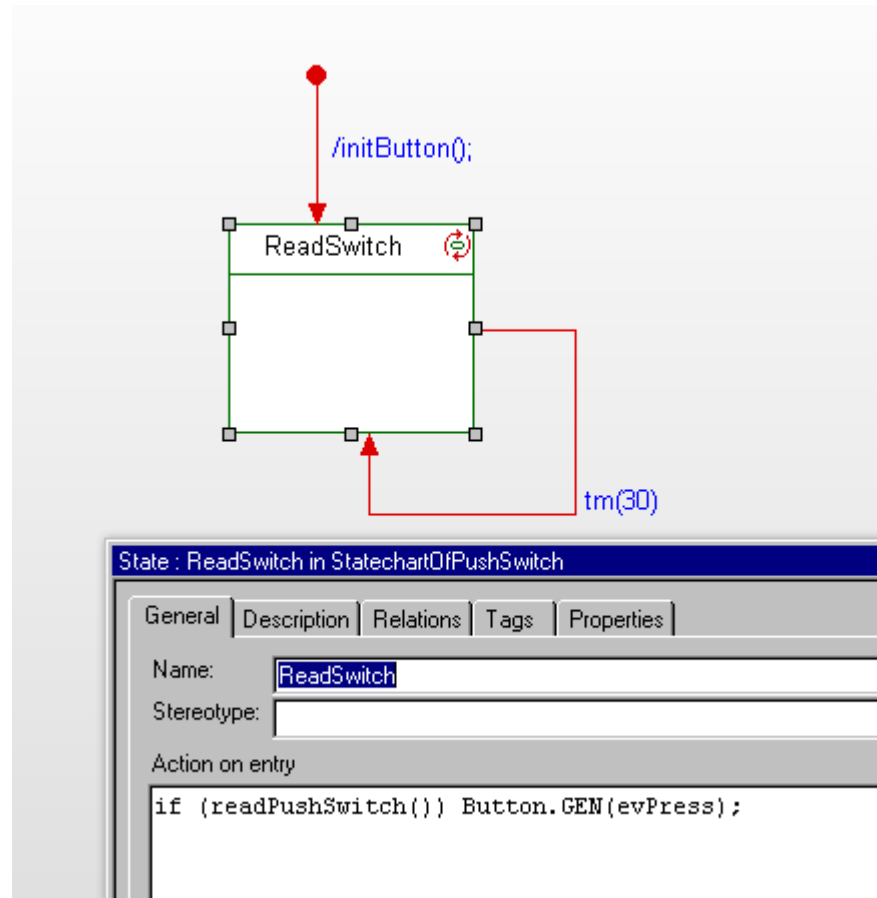
int initButton(void)
{
    dev_switch = open("/dev/fpga_push_switch", O_RDWR);
    if (dev_switch<0){
        printf("Device Open Error\n");
        close(dev_switch);
        return -1;
    }
}

int initLed(void)
{
    dev_led = open("/dev/fpga_led", O_RDWR);
    if (dev_switch<0){
        printf("Device Open Error\n");
        close(dev_led);
        return -1;
    }
    buff_size=sizeof(push_sw_buff);
}
}
```


Interface.cpp(2)

```
unsigned char readPushSwitch(void)
{
    read(dev_switch, &push_sw_buff, buff_size);
    return push_sw_buff[6] | push_sw_buff[7] | push_sw_buff[8];
}
void onLed(int number)
{
    unsigned char data=0;
    if(number==1) data=1;
    if(number==2) data=2;
    if(number==3) data=4;
    write(dev_led,&data,1);
}
/*****
    File Path      : ButtonTest/MontaVista/Interface.cpp
*****/
```

Statechart in PushSwitch



Exercise 5: Stopwatch with real displays and switches

- Modify the StopwatchProject to use seven segment displays and push switches on the target
- Use two digits for minutes, and two digits for seconds
- Use three bottom row push switches.

