IBM Software Group

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

Basic Rational Rhapsody

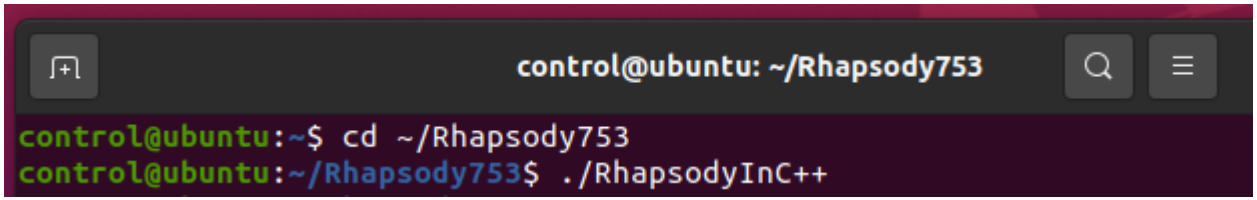
Rational. software

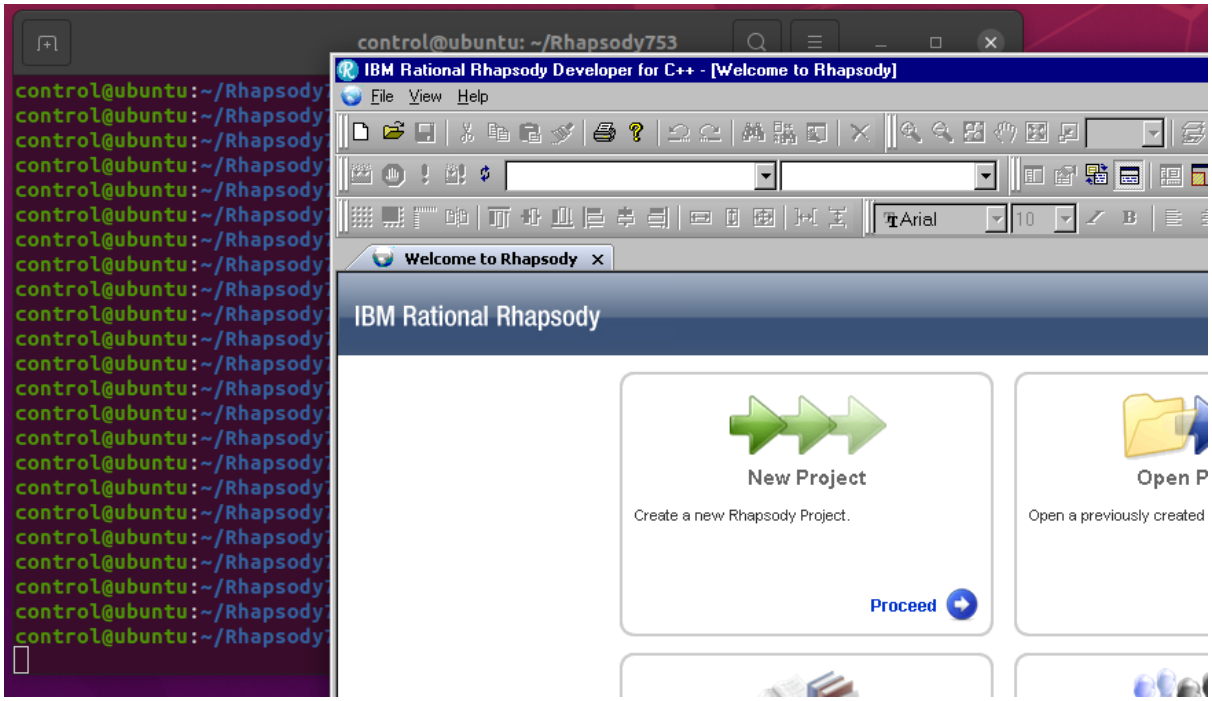
Exercise 1 : Hello World

Start Rhapsody in C++

$ cd ~/Rhapsody753

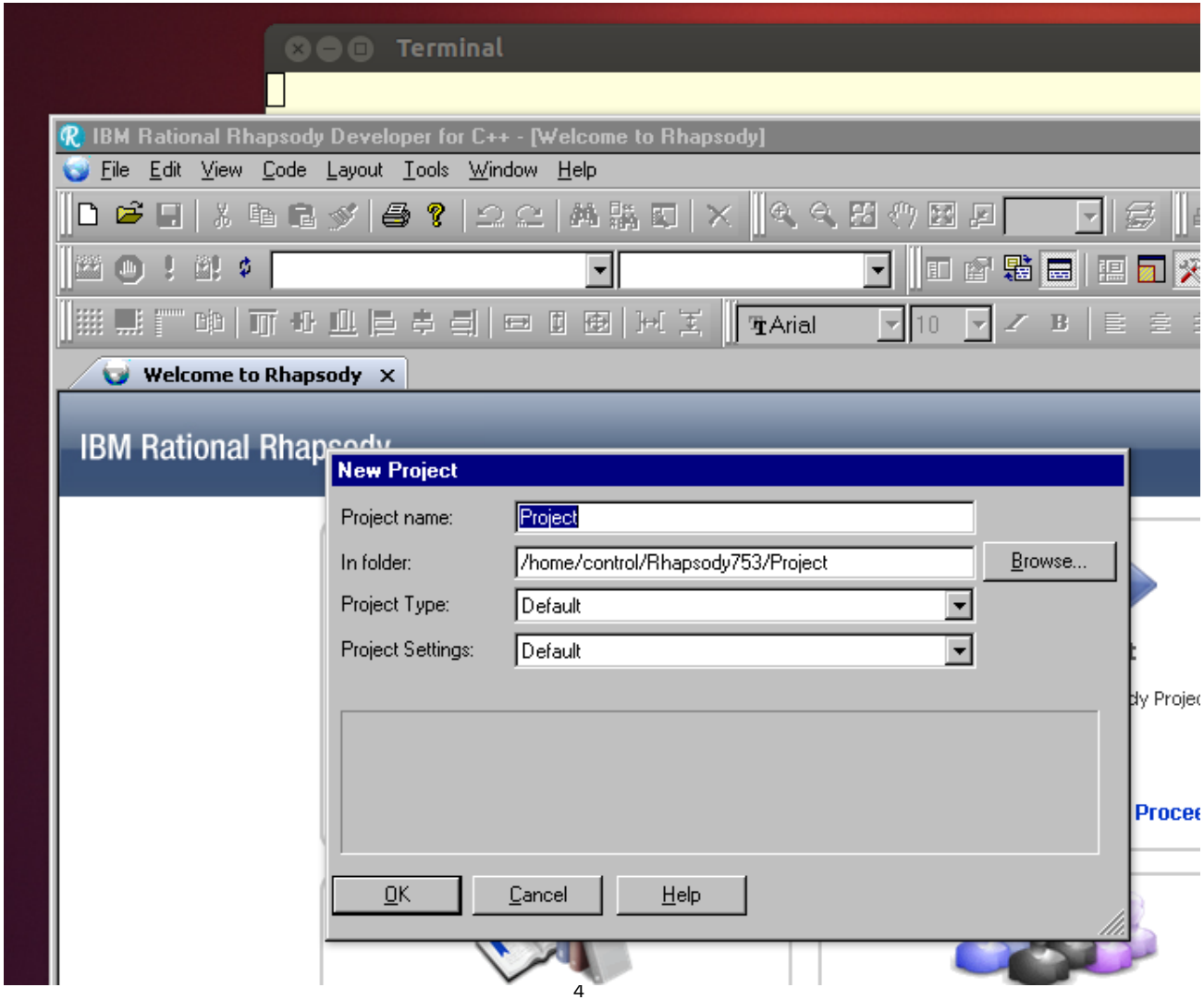
$ ./RhapsodyInC++



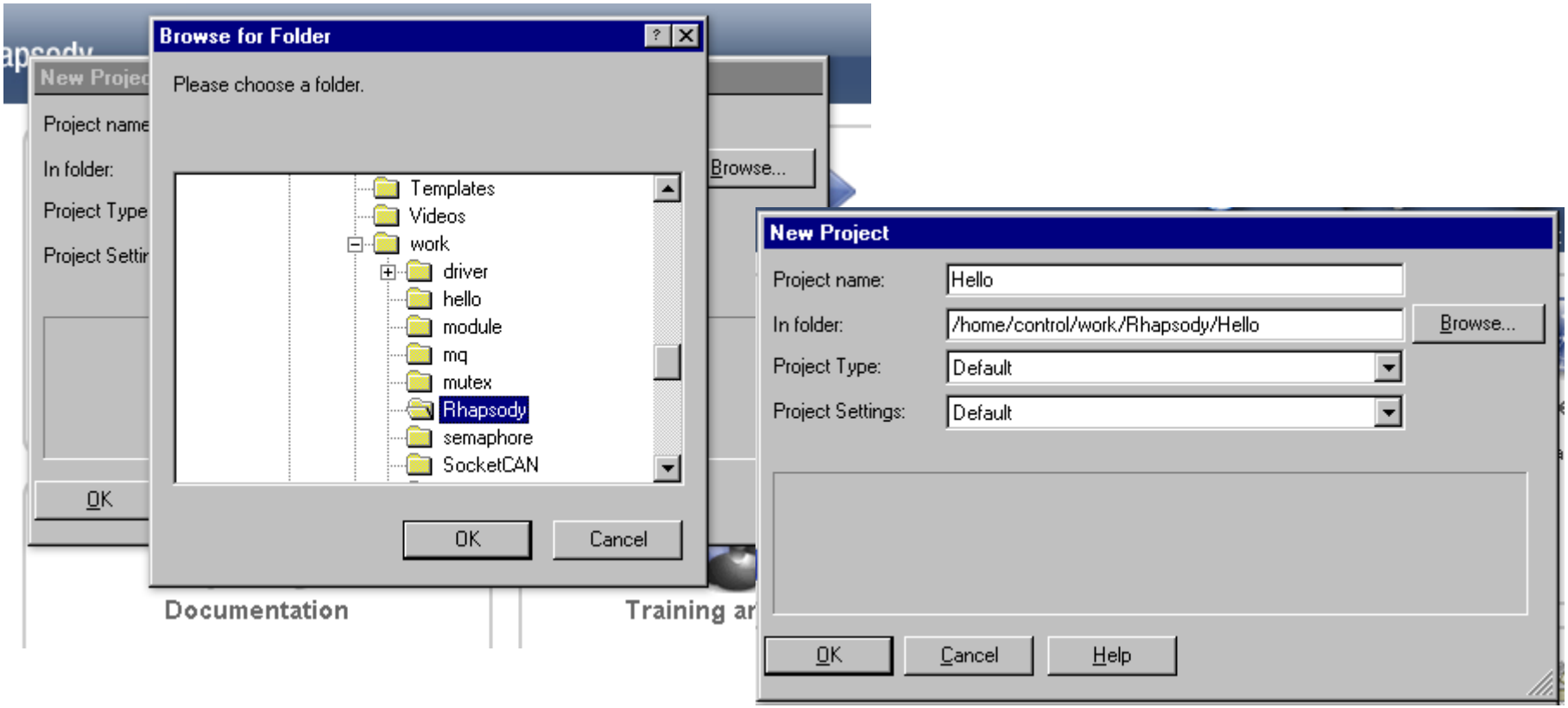


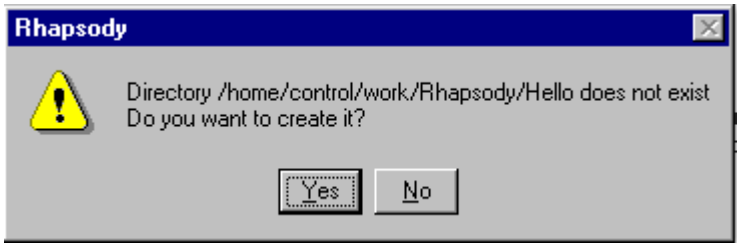
Creating a Project

- New from File menu



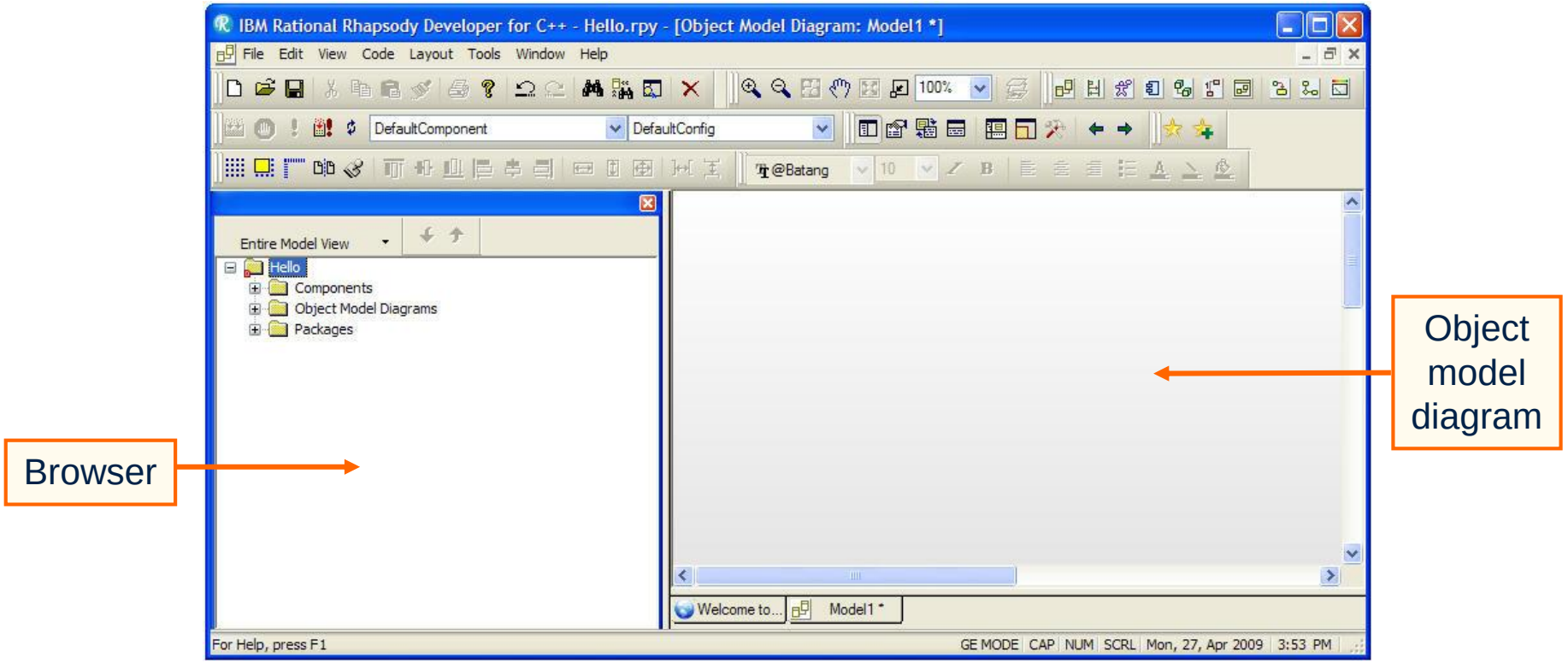
Select the working directory





Browser

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

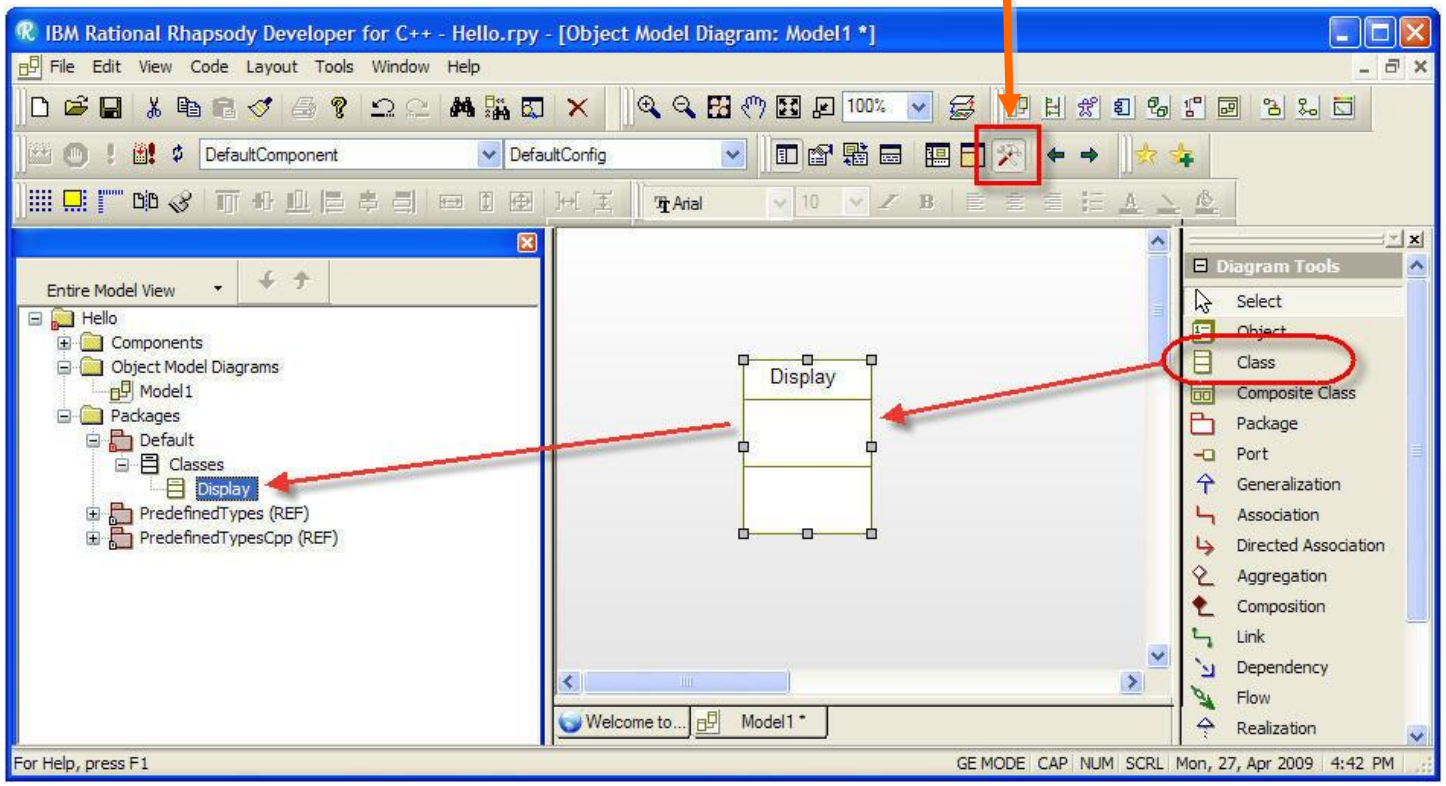


Drawing a class

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

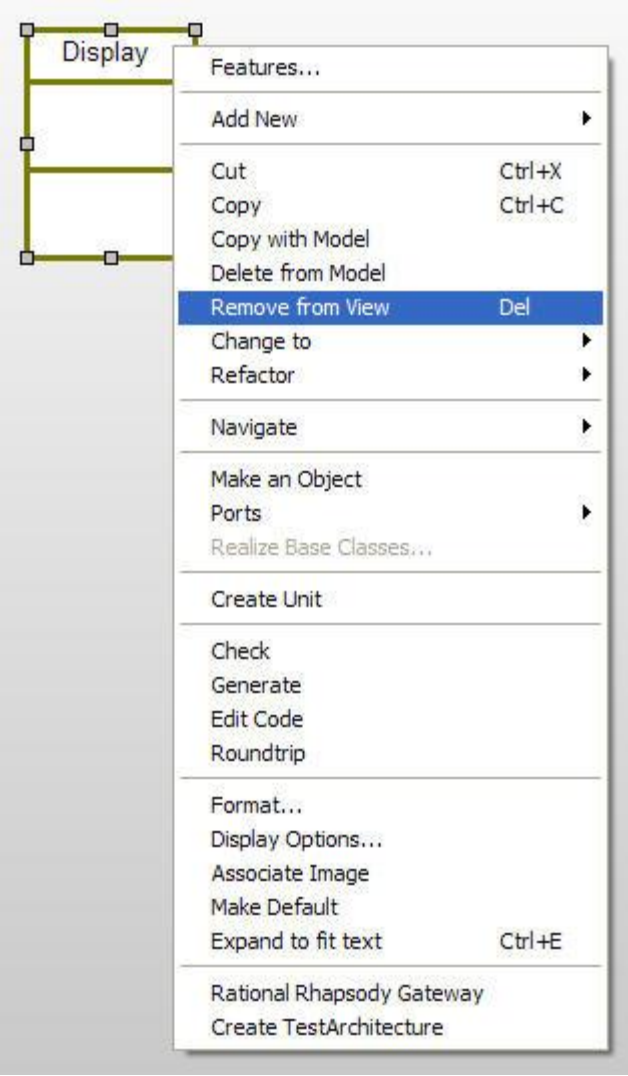
Show/Hide Drawing Toolbar

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



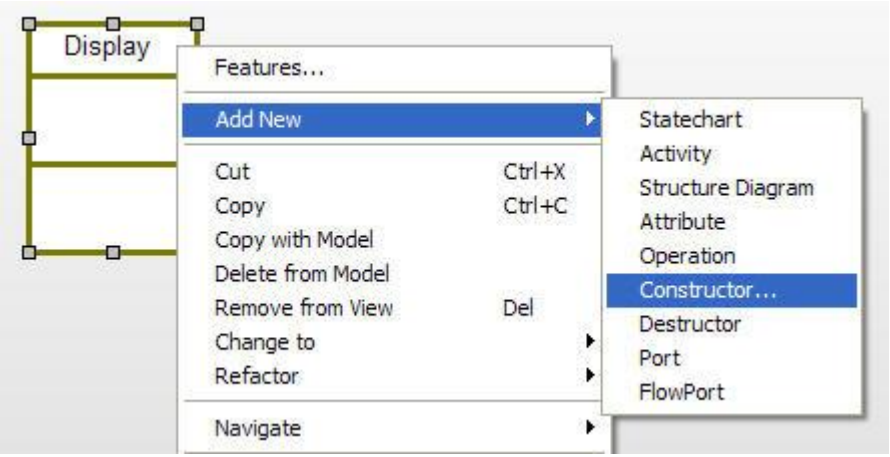
Remove from View / Delete from model

- 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.

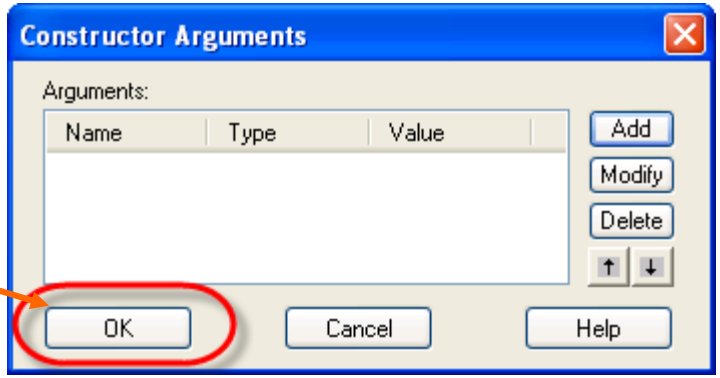


Adding a constructor

- The simplest way to add a constructor is to right-click on the class and choose **Add New > Constructor.**



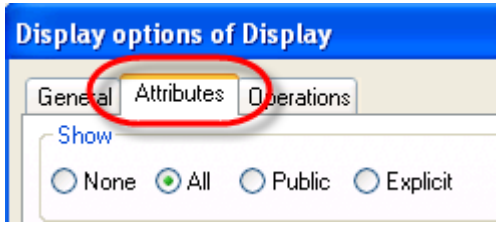
- You do not need any constructor arguments; click **OK**.

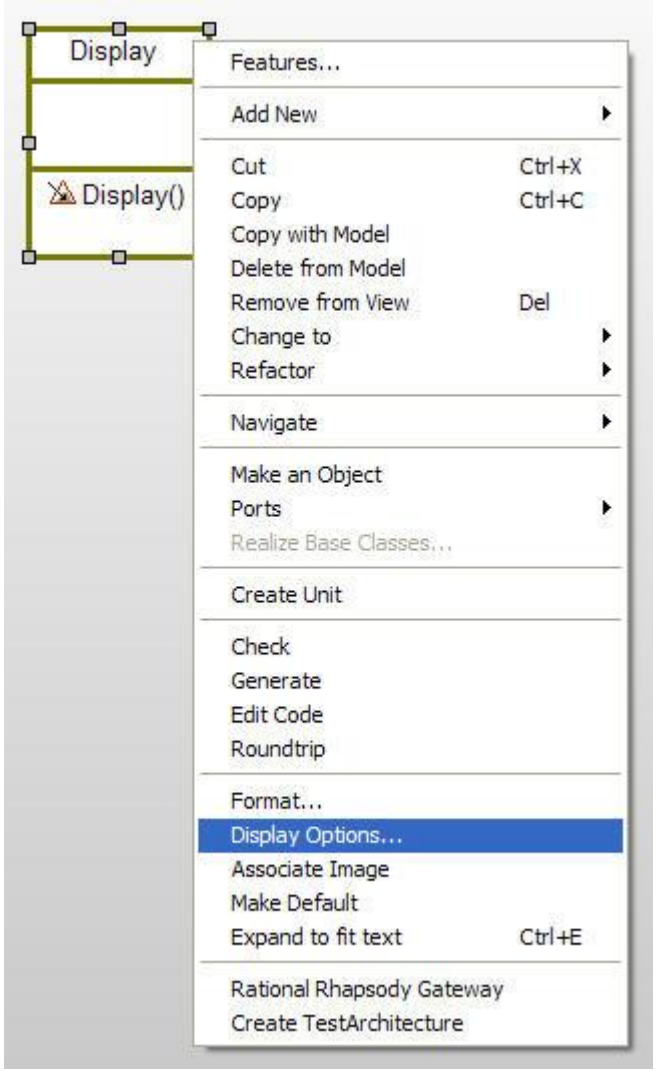


Constructors may also be added through the features**Operations** tab. Click **New** and select **Constructor**.

Display options

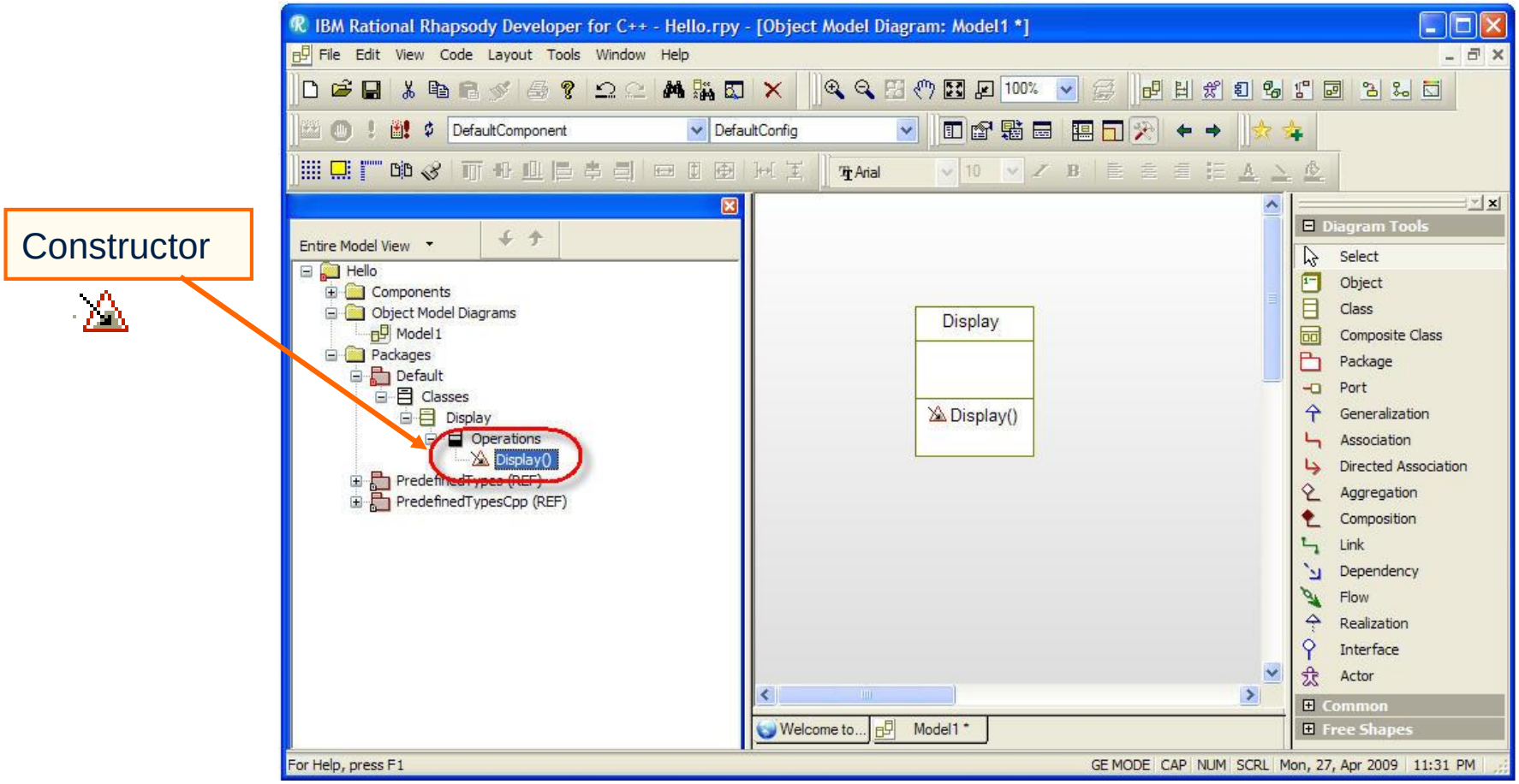
- You would expect to see the constructor shown on the class on the Object Model Diagram.
- You can control what gets displayed on this view of the class by using *Display Options*.
- Right-click **Display** class and select **Display Options**.
  - Set the options to display **All** attributes and **All**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





Display constructor

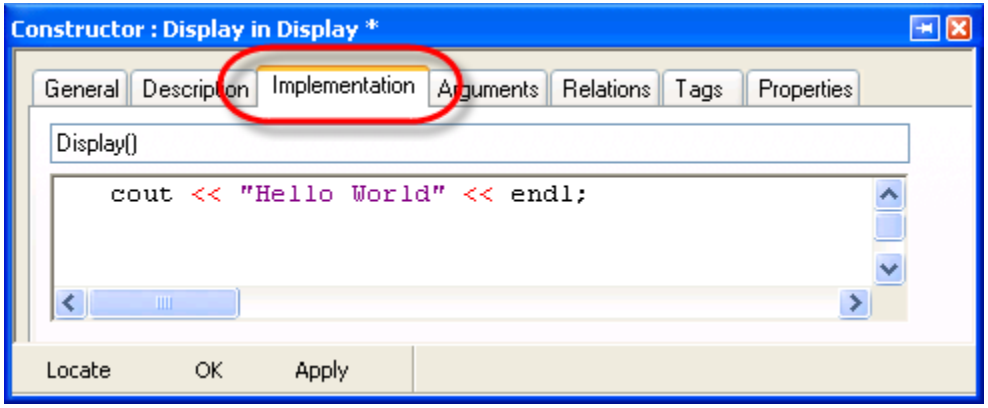
- You should be able to see the constructor is now shown in both the browser and the OMD (object model diagram).



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;
```

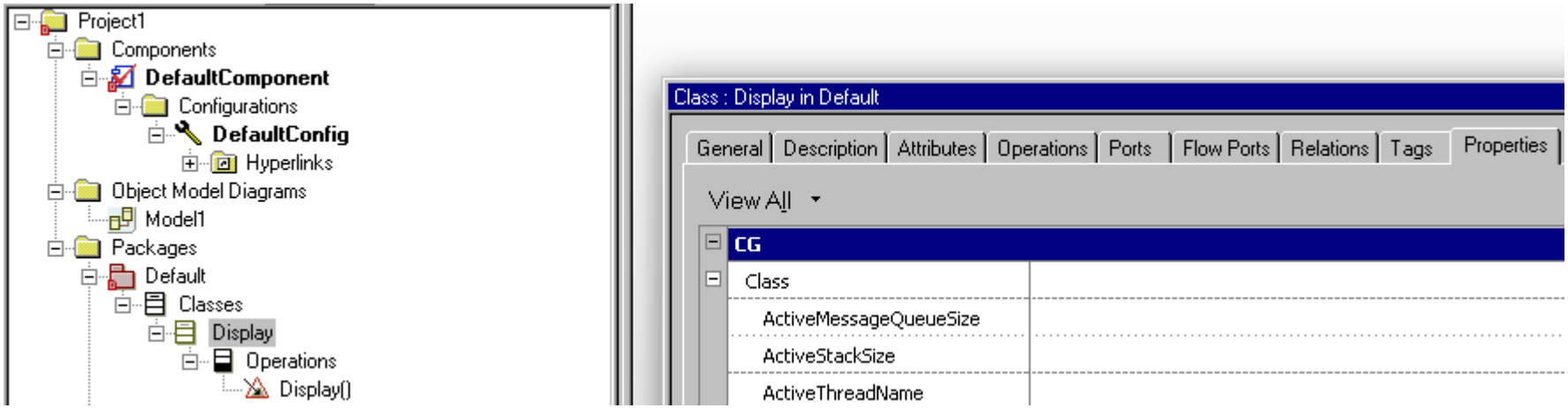


If you are not using Visual C++ 6.0, then you should add the std namespace, for example, **`std::cout << "Hello World" << std::endl;`** Or, set the property *CPP_CG::Class::ImplementationProlog* to `using namespace std;`.

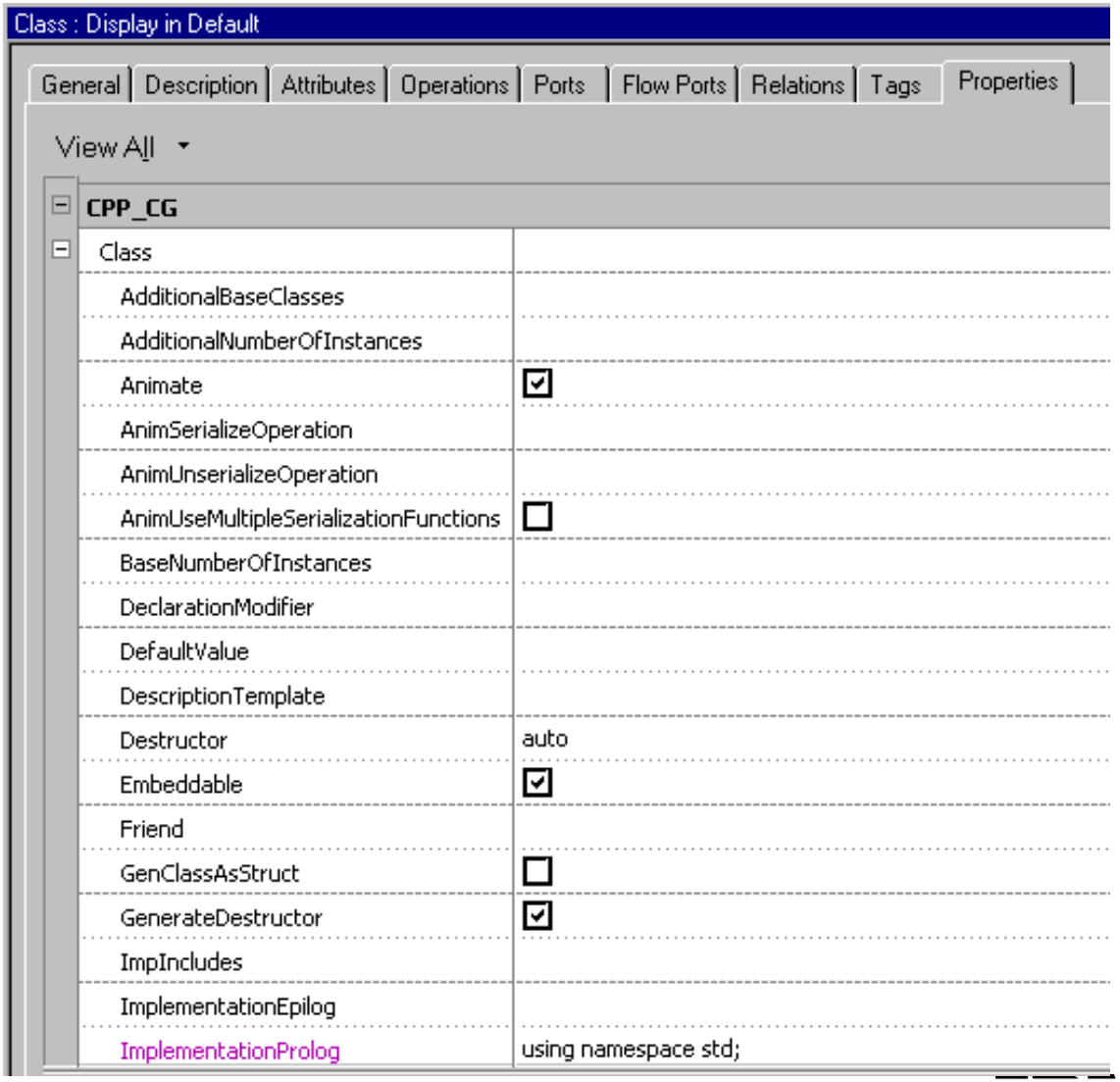
Adding an implementation

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;`.

- Display class를 더블 클릭하고 Properties 탭을 선택한 후 View All로 바꿈



- CPP_CG의 Class 항목에서 ImplementationProlog에 아래와 같이 입력



Adding an implementation

- 앞의 과정이 번거로울 경우 Ubuntu에서는 아래와 같이 입력

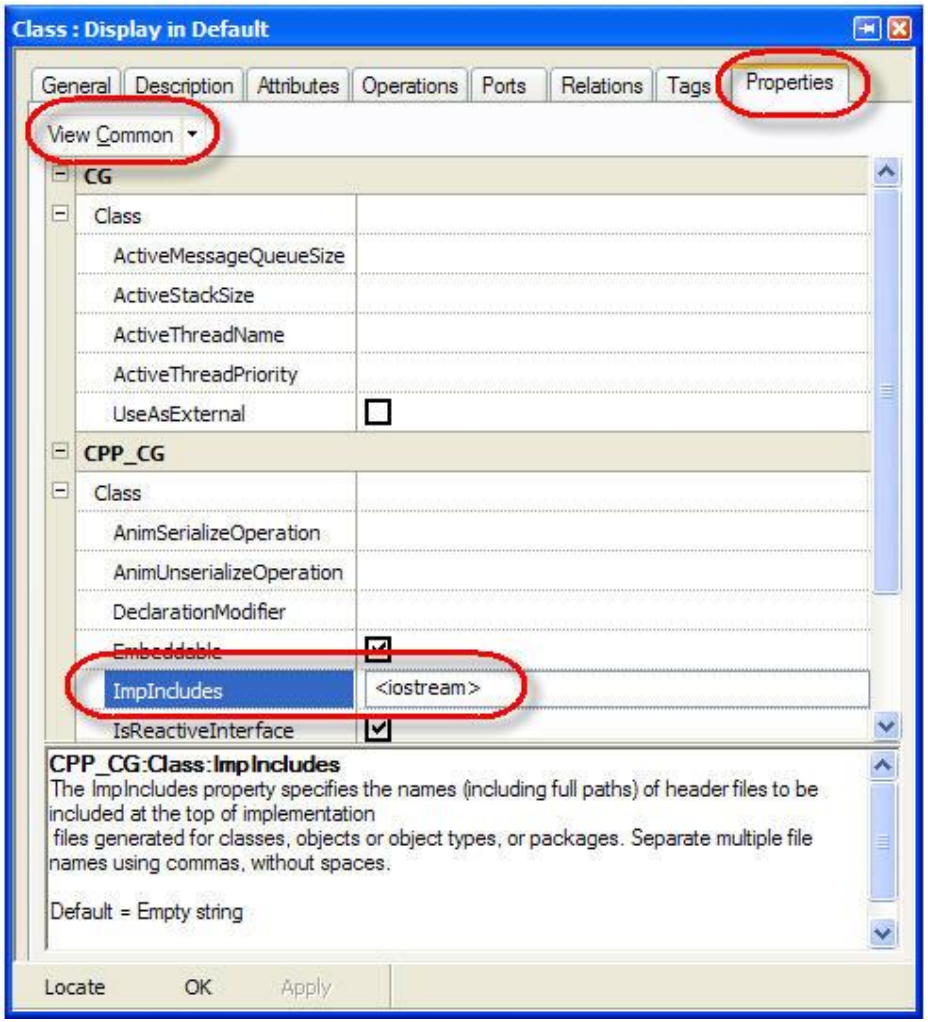
```
std::cout << "Hello World" << std::endl;
```

- 또는 아래와 같이 입력 해도 가능. Ubuntu에서는 #include <stdio.h> 가 없어도 컴파일이 됨

```
printf("Hello World\n\r");
```

#include <iostream>

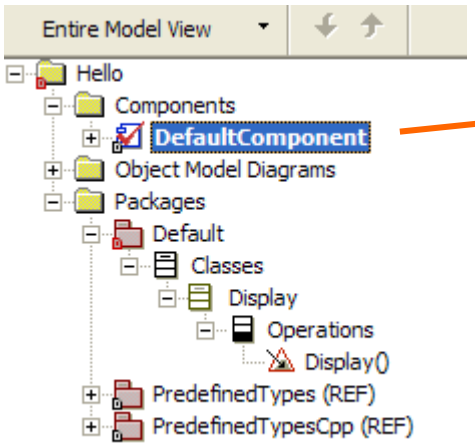
- Since you have used *cout*, you must add an include of the *iostream* header to the Display class.
- In the browser, select the **Display** class and double-click to bring up the features.
  - Select the **Properties** tab
  - Ensure that the Common View is selected
  - Enter <*iostream*> into the "ImpIncludes" property.

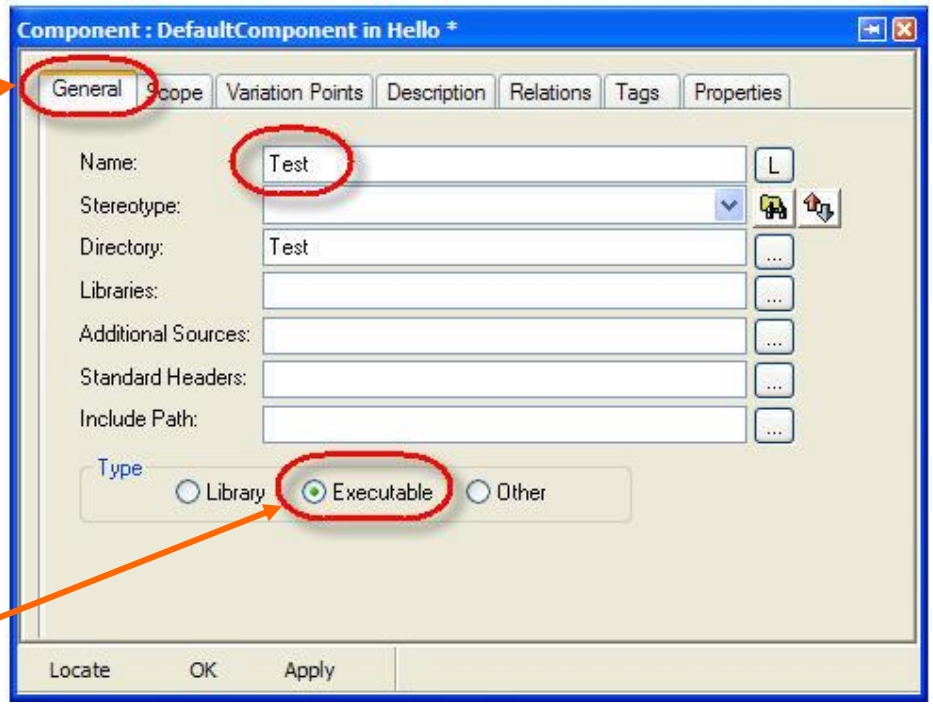


ImpIncludes is an abbreviation for Implementation Includes.

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*.

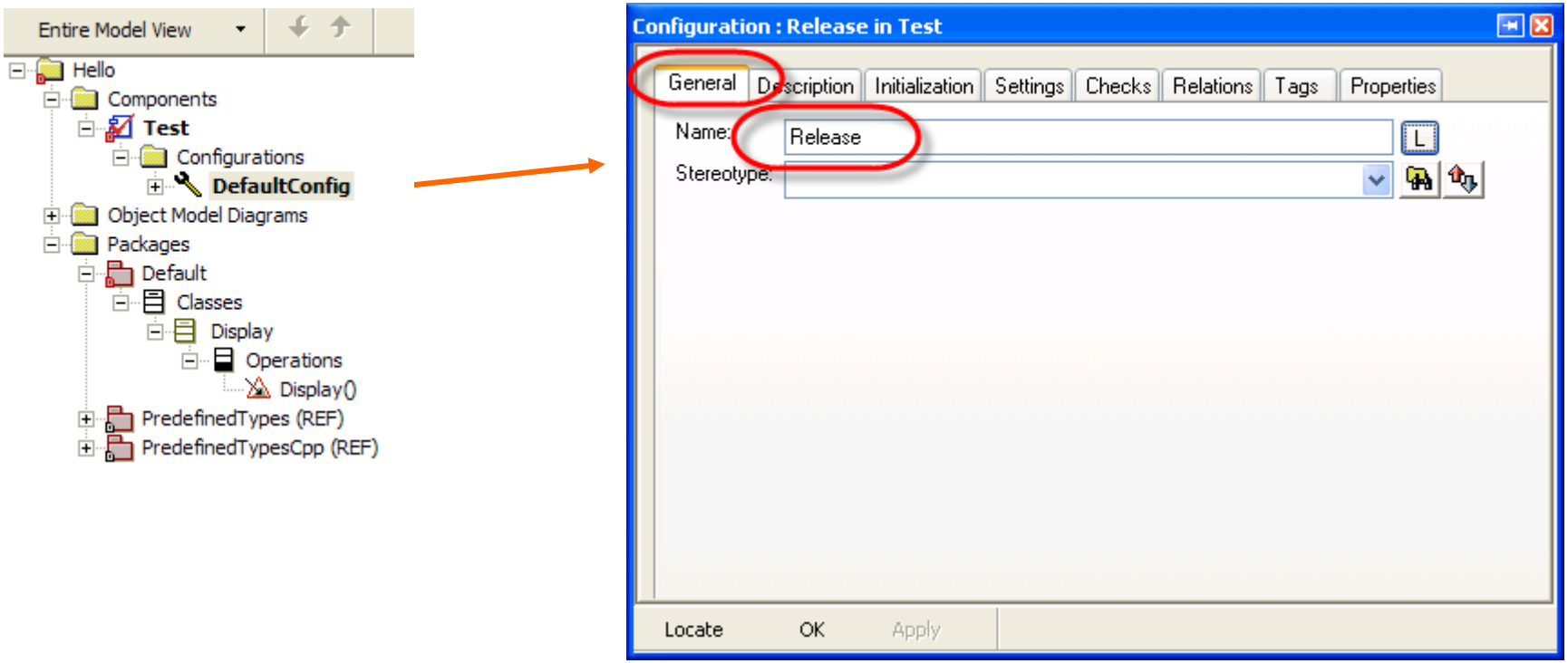




Executable

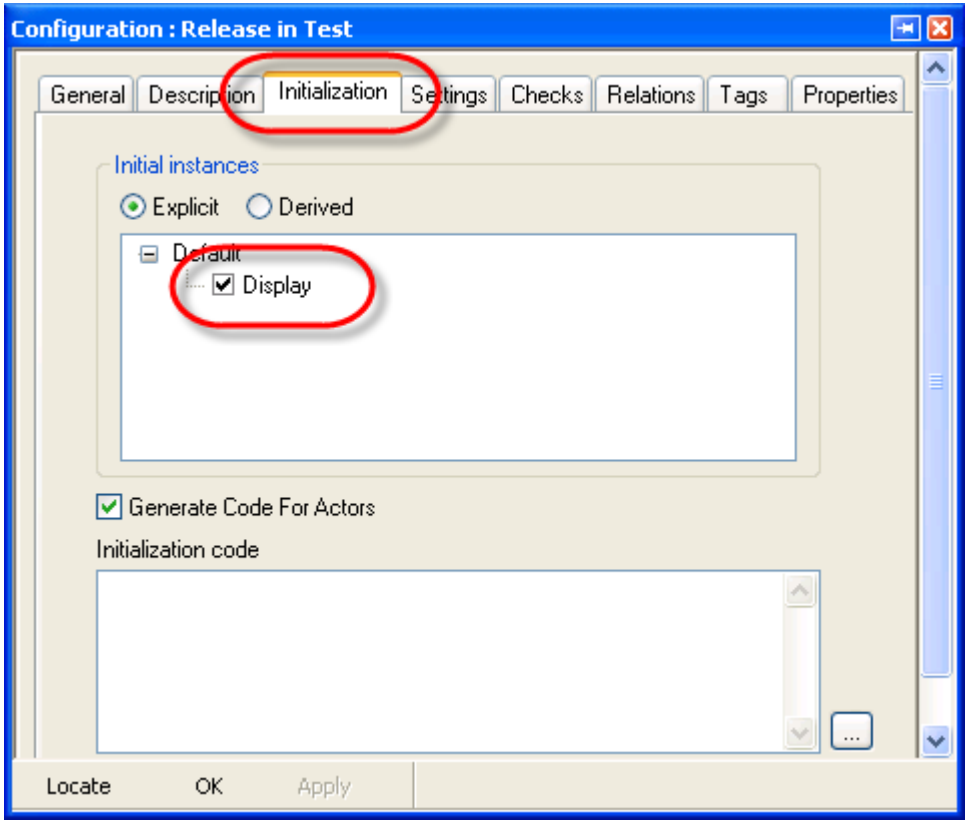
Test component

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



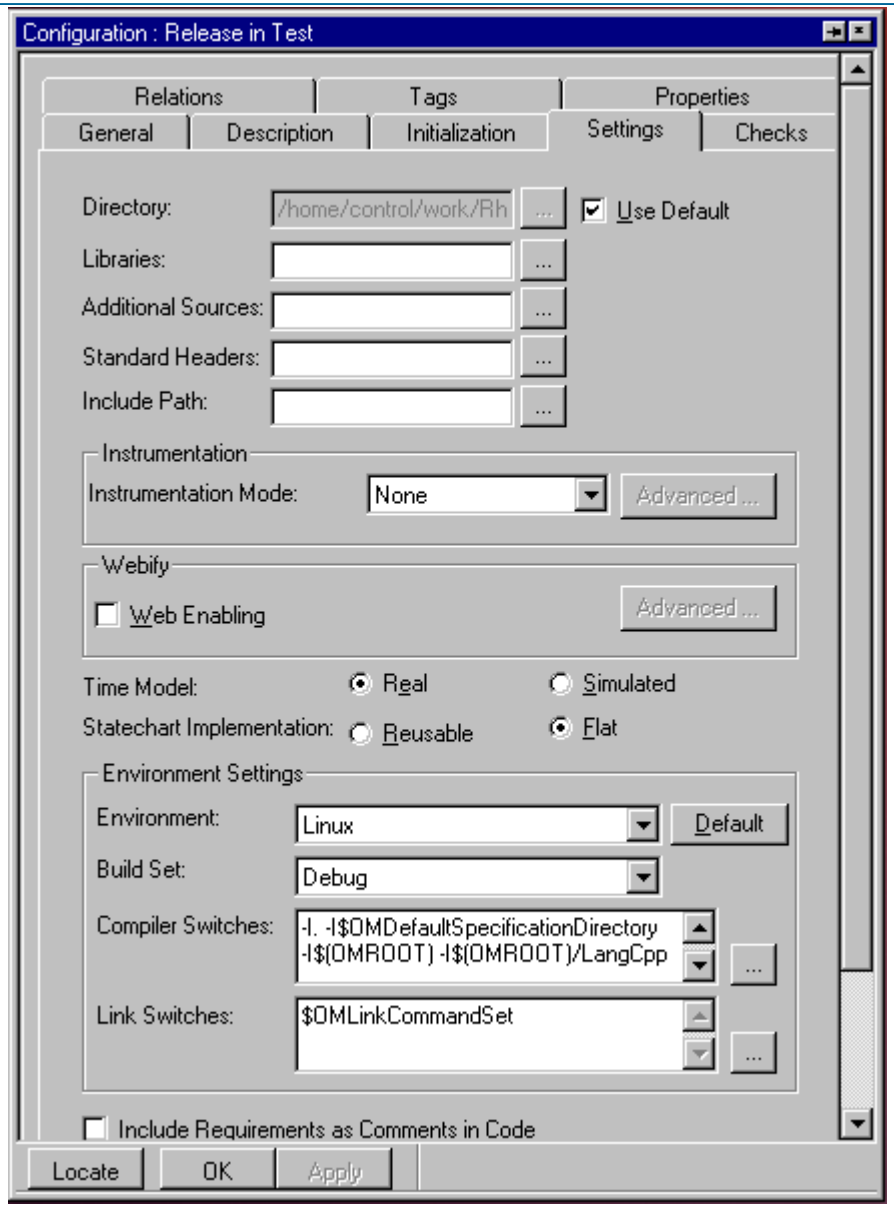
Initial instance

- 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

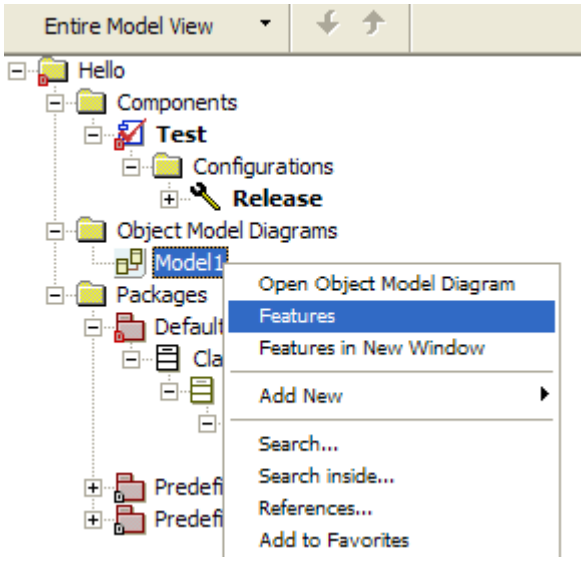
- You need to select an environment so that Rational Rhapsody knows how to create an appropriate Makefile.
- Select the **Settings** tab.
- Select the appropriate environment, for example: Linux.

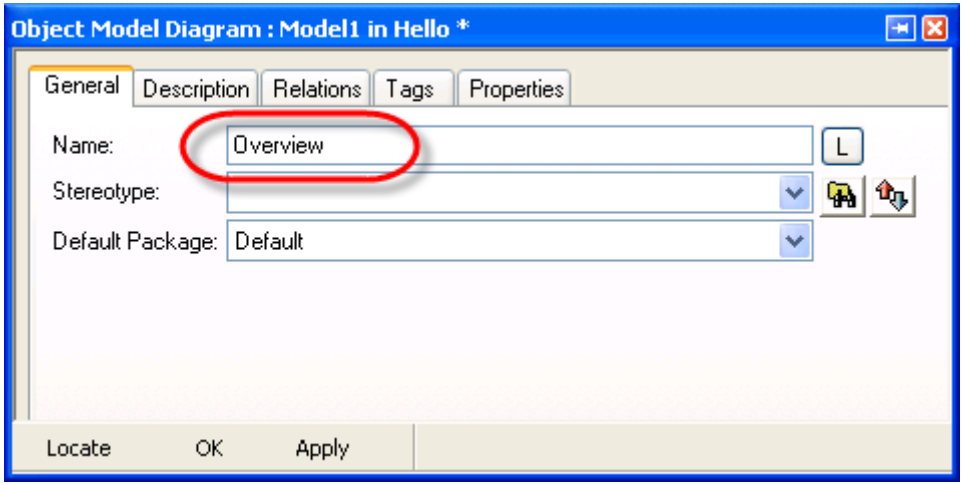


You will learn about the many other settings later.

Renaming the OMD

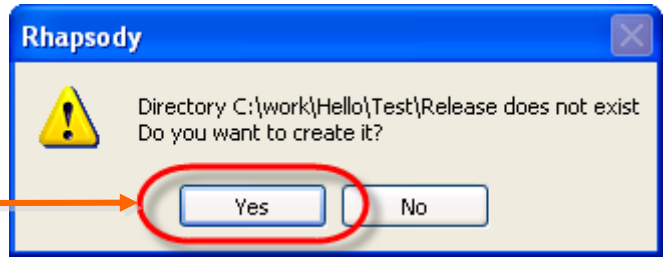
- Expand the Object Model Diagrams in the browser. Right-click the Object Model Diagram **Model1** to invoke the features dialog.
- Rename the diagram from *Model1* to *Overview*.





Generating code

- You are now ready to generate code.
  - Save the model.
  - Select Generate/Make/Run.
  - Click **Yes** to the question:

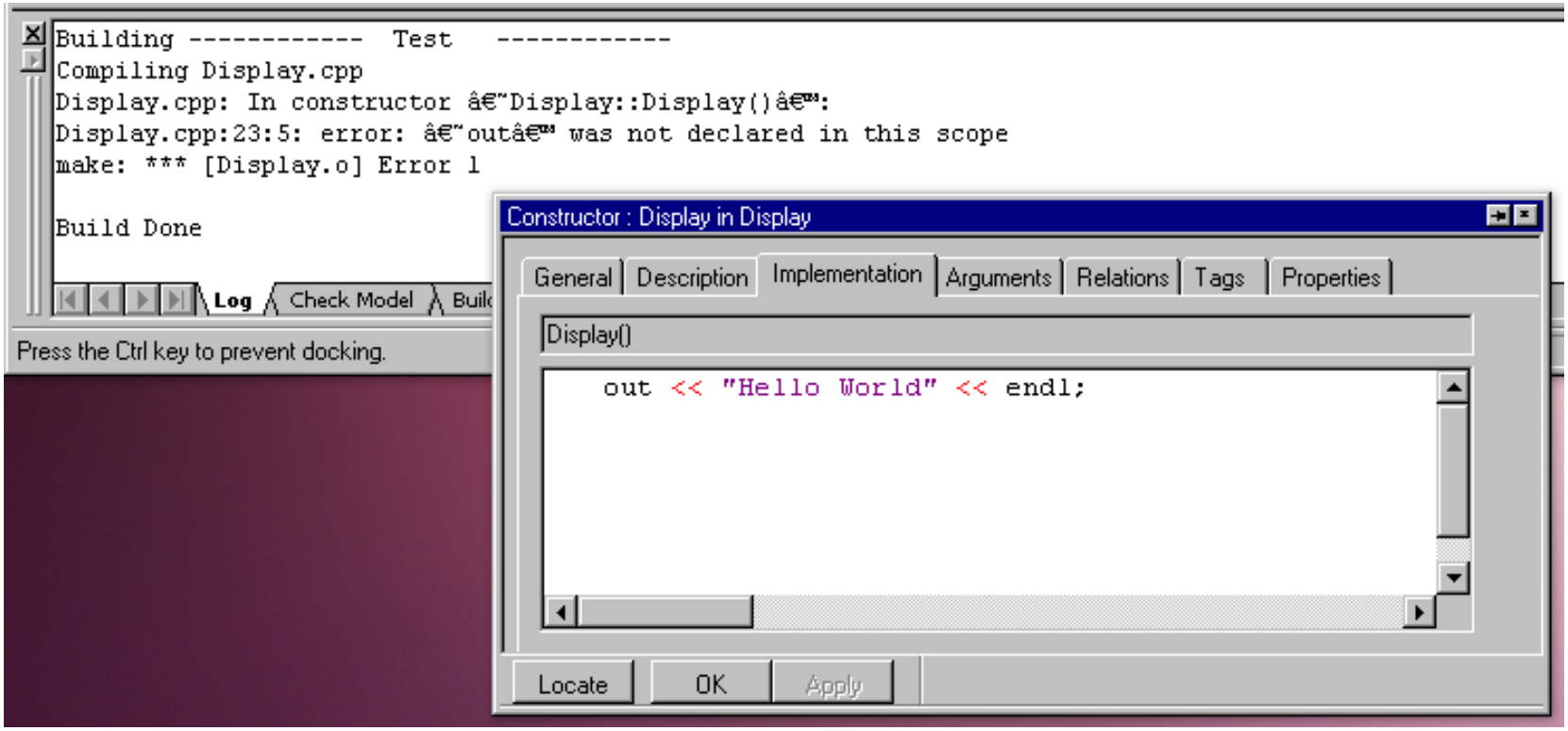



```
All Checks Terminated Successfully

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

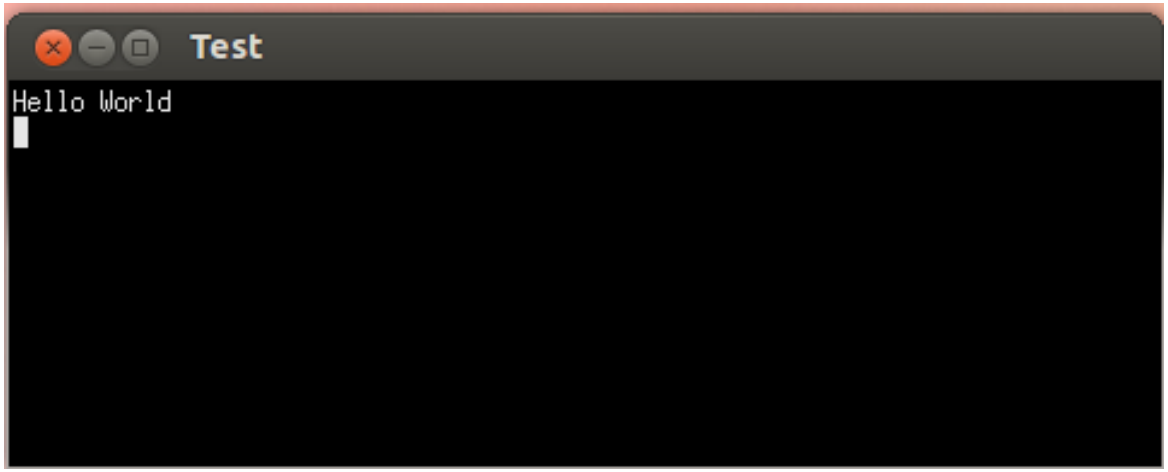
Handling errors

- 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

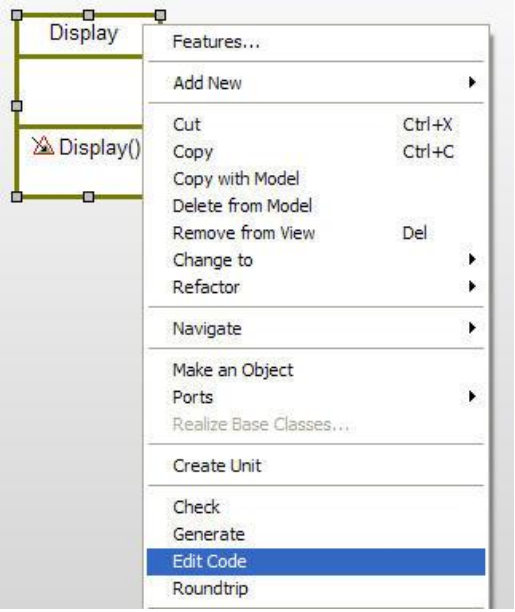
Makefile

Home work Rhapsody Hello Test **Release** Search

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();
};
```

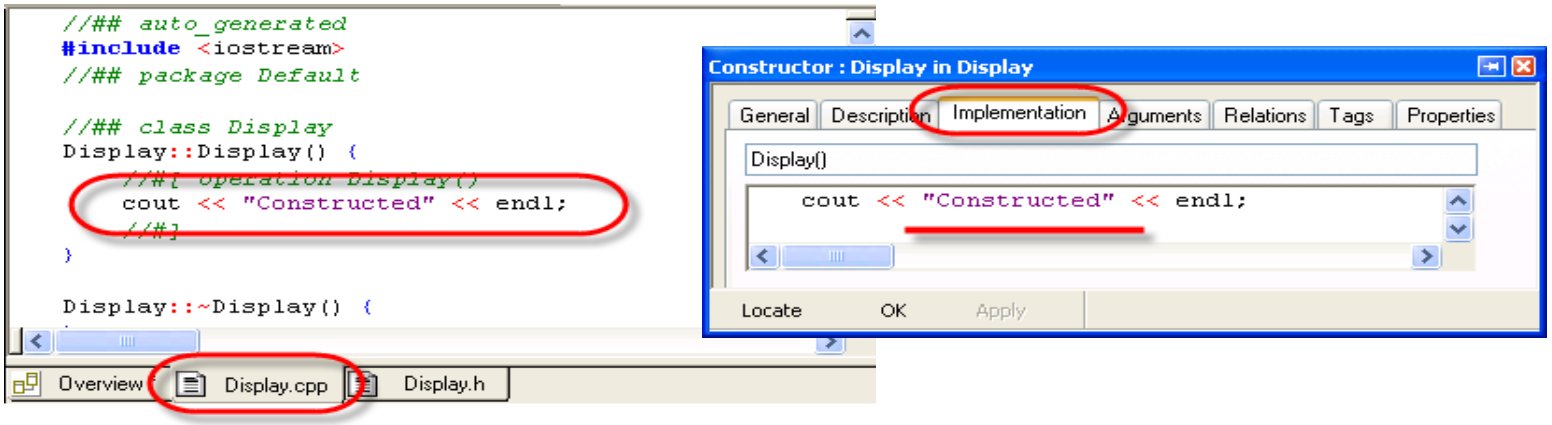
Overview * | Display.cpp | Display.h

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

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

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

Overview | Display.cpp | Display.h

Modifying the code

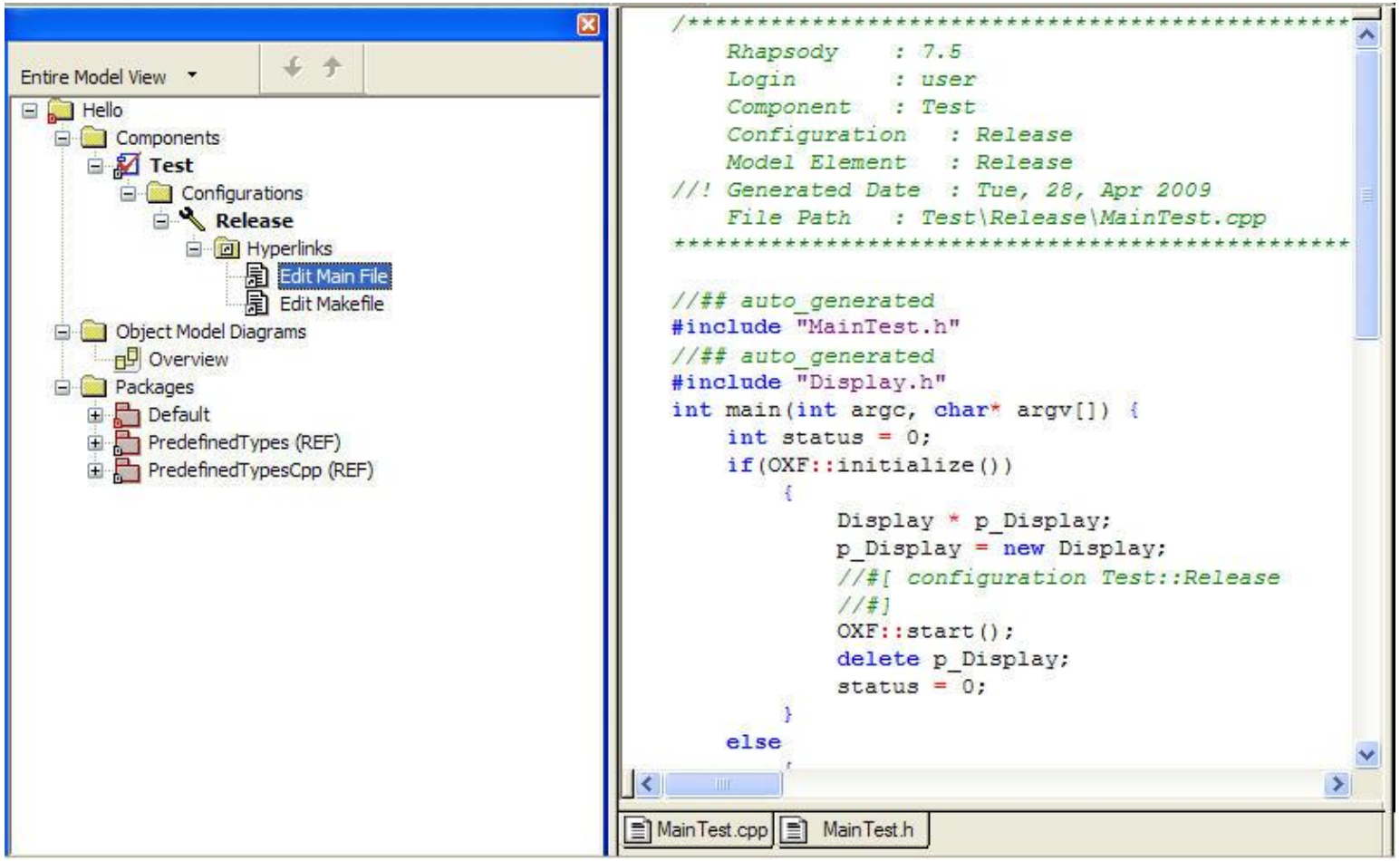
- You can modify the generated code.
- In the Display.cpp file, change the implementation to print out *Constructed* instead of *Hello World*.
- Transfer the focus back to another window to roundtrip the modifications back into the model.
- Note that the model has been updated automatically.



- 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:



Project files

AutoSave

Generated code

Project workspace

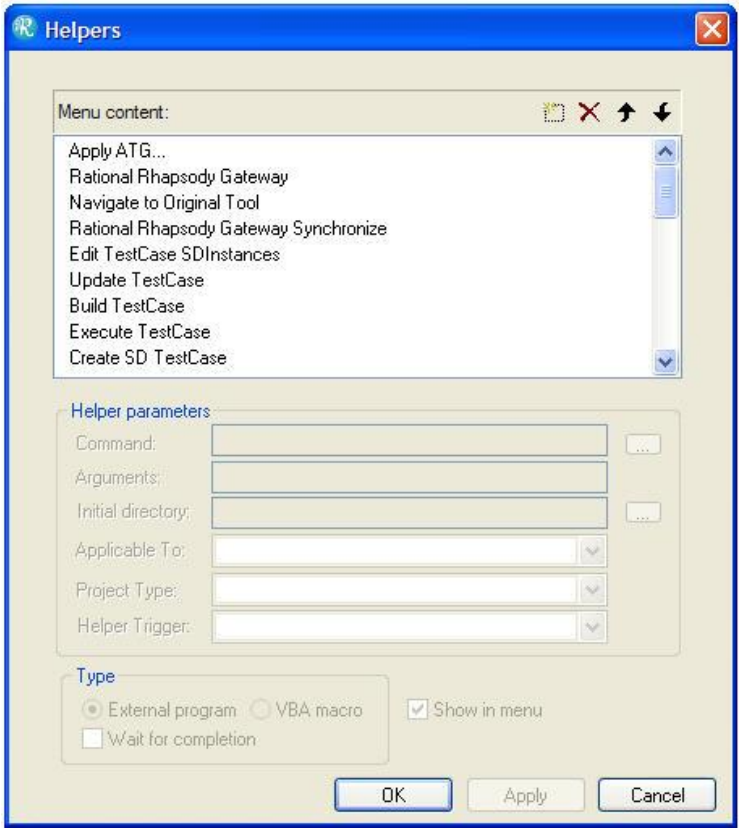
Home work Rhapsody **Hello** Test Release Search

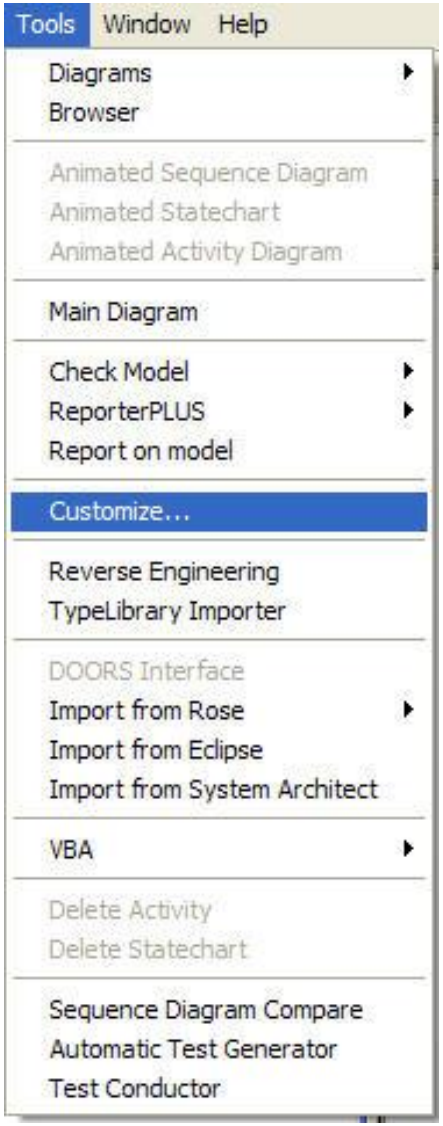
Name	Size	Type	Date Modified
Hello_auto_rpy	0 items	folder	Fri 29 Dec 2017 05:17:50 PM KST
Hello_rpy	4 items	folder	Fri 29 Dec 2017 05:04:05 PM KST
Test	1 item	folder	Fri 29 Dec 2017 05:12:05 PM KST
Hello.ehl	64 bytes	plain text document	Fri 29 Dec 2017 05:07:50 PM KST
Hello.rpw	1.3 kB	plain text document	Fri 29 Dec 2017 05:07:50 PM KST
Hello.rpy	7.5 kB	plain text document	Fri 29 Dec 2017 05:04:05 PM KST
Hello.save	0 bytes	plain text document	Fri 29 Dec 2017 05:07:50 PM KST

The model

Extended exercise

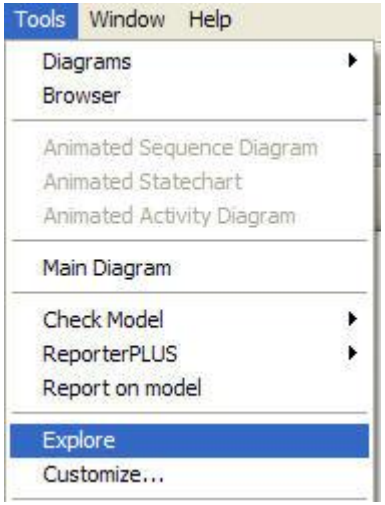
- You can customize Rational Rhapsody to get quick access to the location of the current project.
- Select **Tools > Customize**.

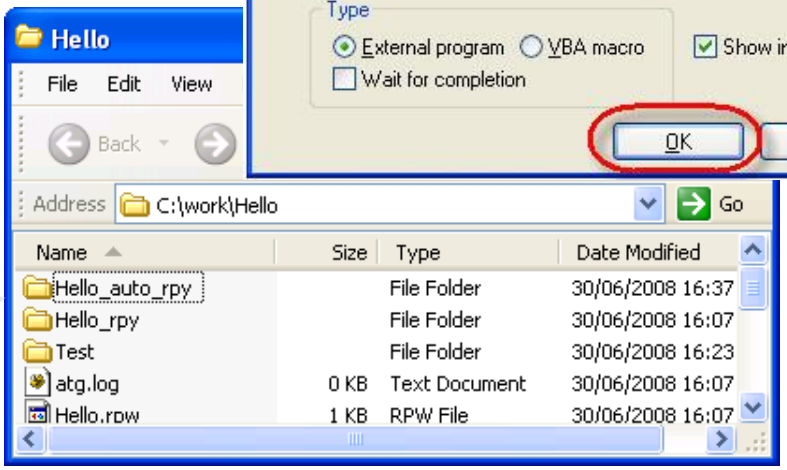


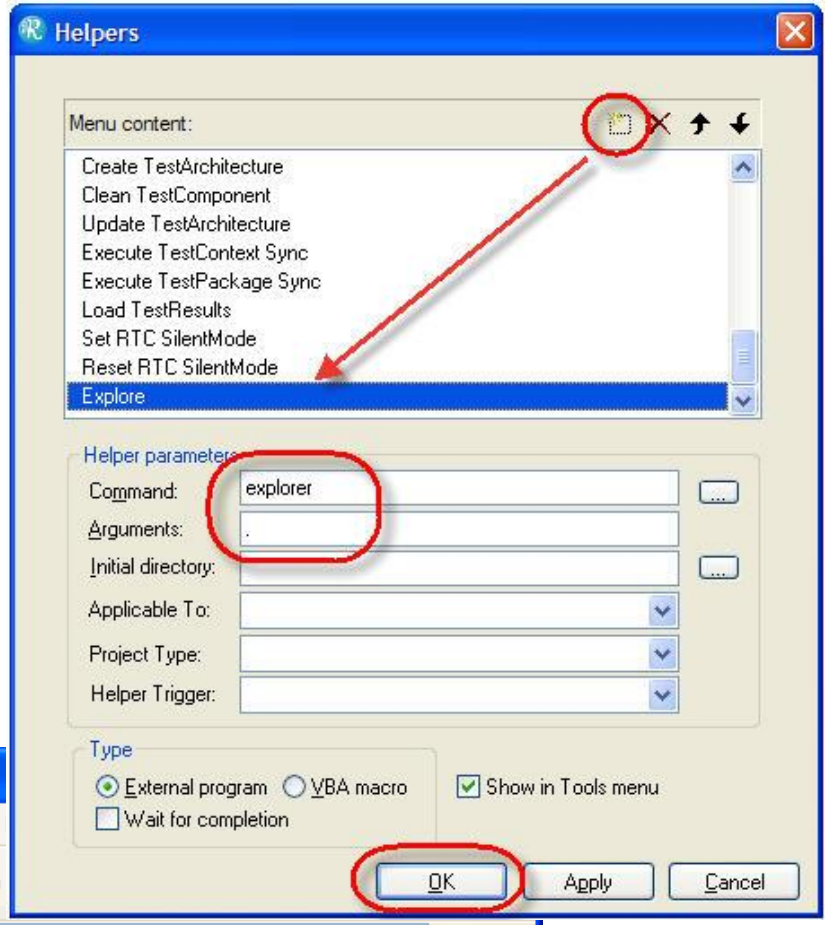


Customize

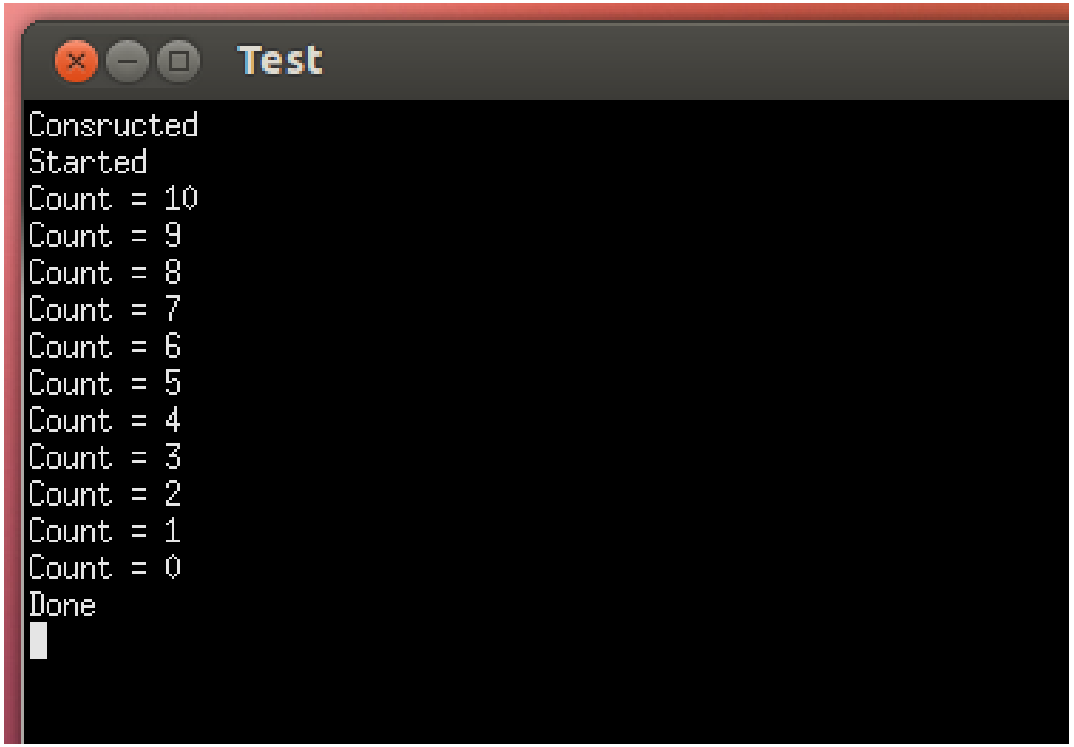
- Click to enter a new entry `Explore` to the **Tools** menu.
- Set the **Command** to `explorer`.
- Set **Arguments** to **.**
- Click **OK**.
- Select **Tools > Explore**.





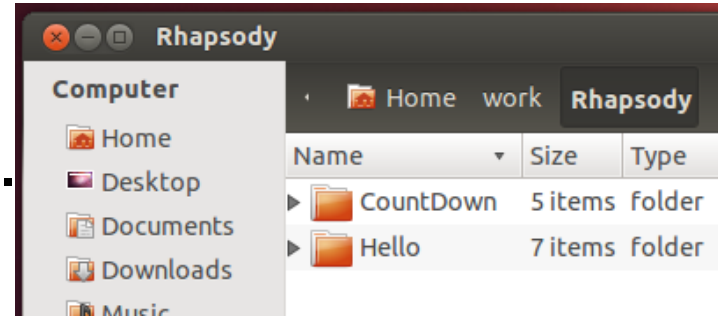


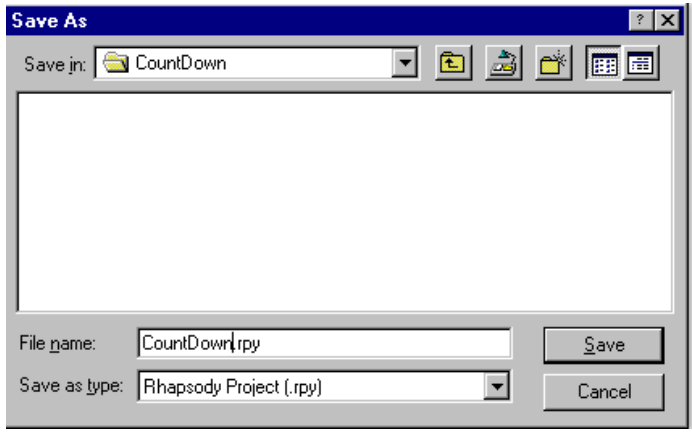
Exercise 2: Count down



Copying a project

- Select **File > Save As**.
- Press to select the upper 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 preserved.

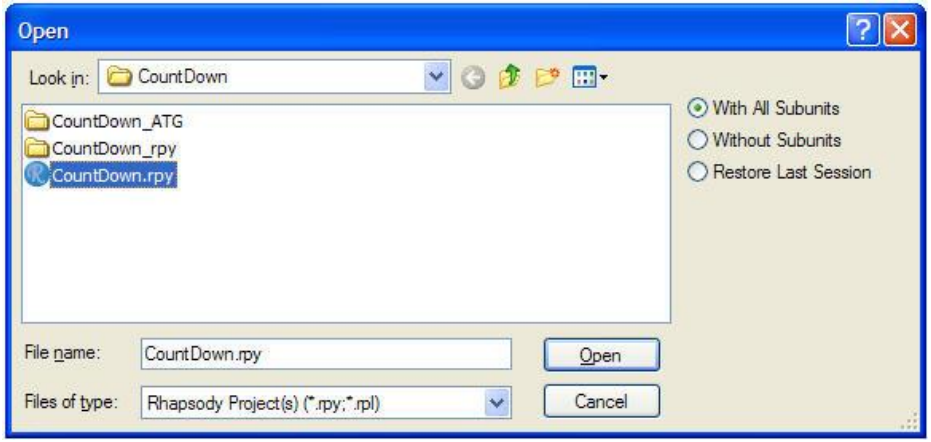


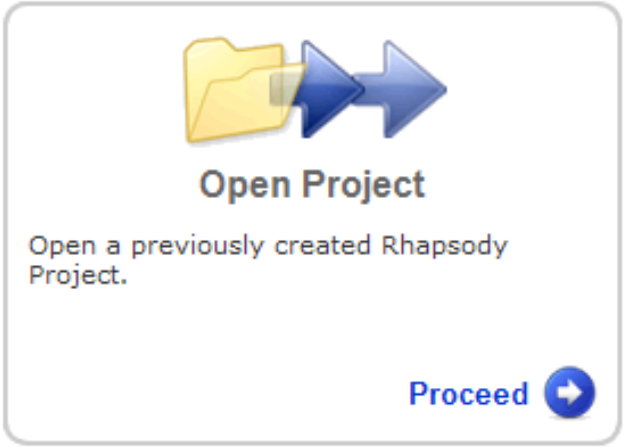


Each time there is an auto-save, Rational Rhapsody only saves just what has changed since the last auto-save.

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.

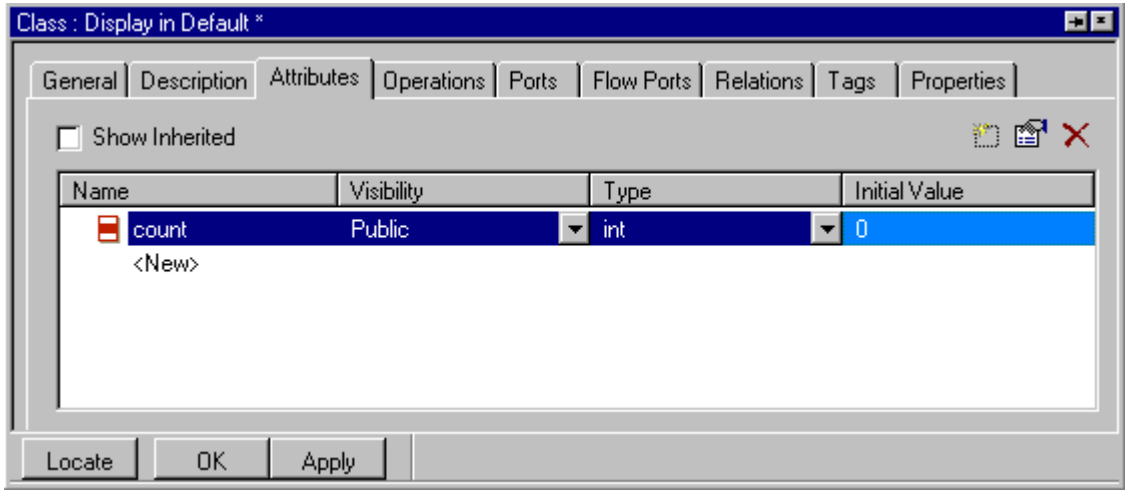


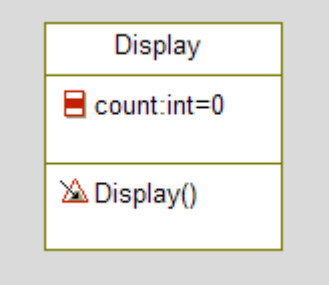


The Rhapsody.ini file determines which Rational Rhapsody (C / C++ / J / Ada) will be opened on double-clicking the .rpy file.

Adding an attribute

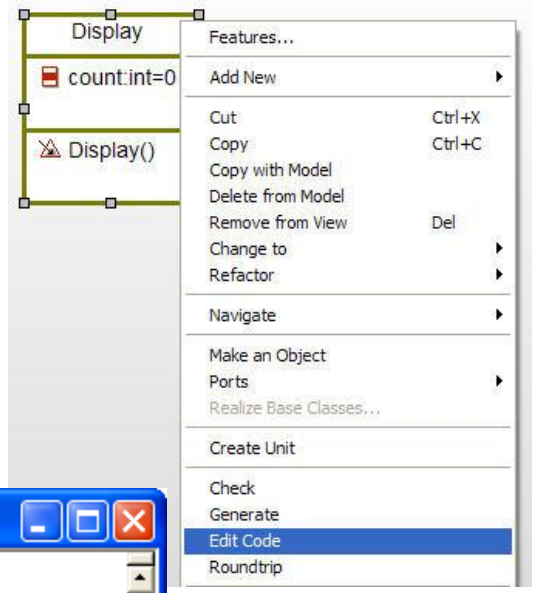
- To add an attribute, double-click on the **Display** class to bring up the features and select the **Attributes** tab.
- Click **New** to add an attribute *count* of type *int.*
- Set the initial value to **0**.





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

};
```

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;
}
```

Initial Value

Mutator

Protected attribute

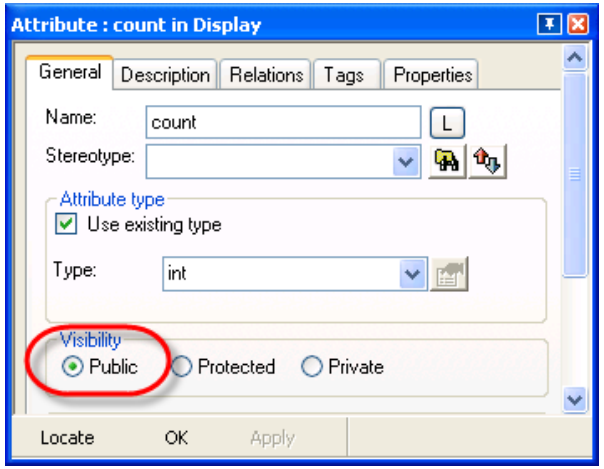
Accessor

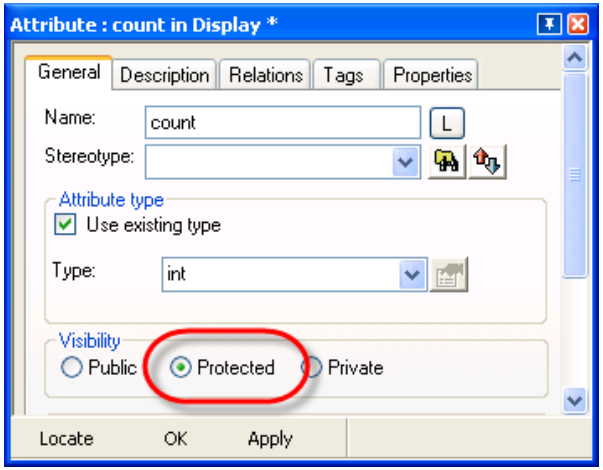
What are accessors and mutators?

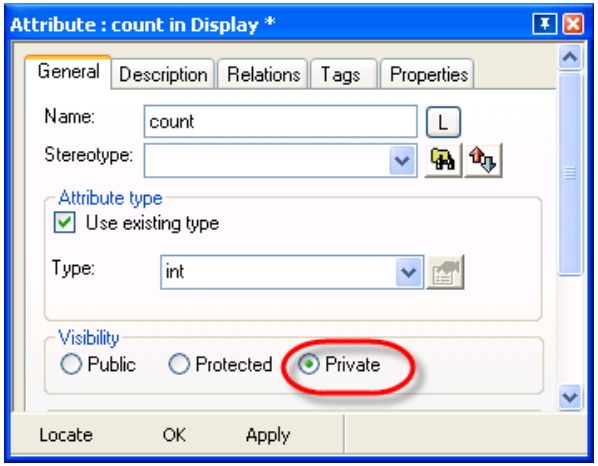
- By default, all attribute data members in Rational Rhapsody are protected.
- If other classes need access to these attributes, then they must use an Accessor, for example, *getCount*() or Mutator, for example, *setCount*().
- This allows the designer of a class, the freedom to change the type of an 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).







count

```
public :

    //## auto_generated
    int getCount() const;

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


////    Attributes    ////
protected :

    int count;      //## attribute count
```

count

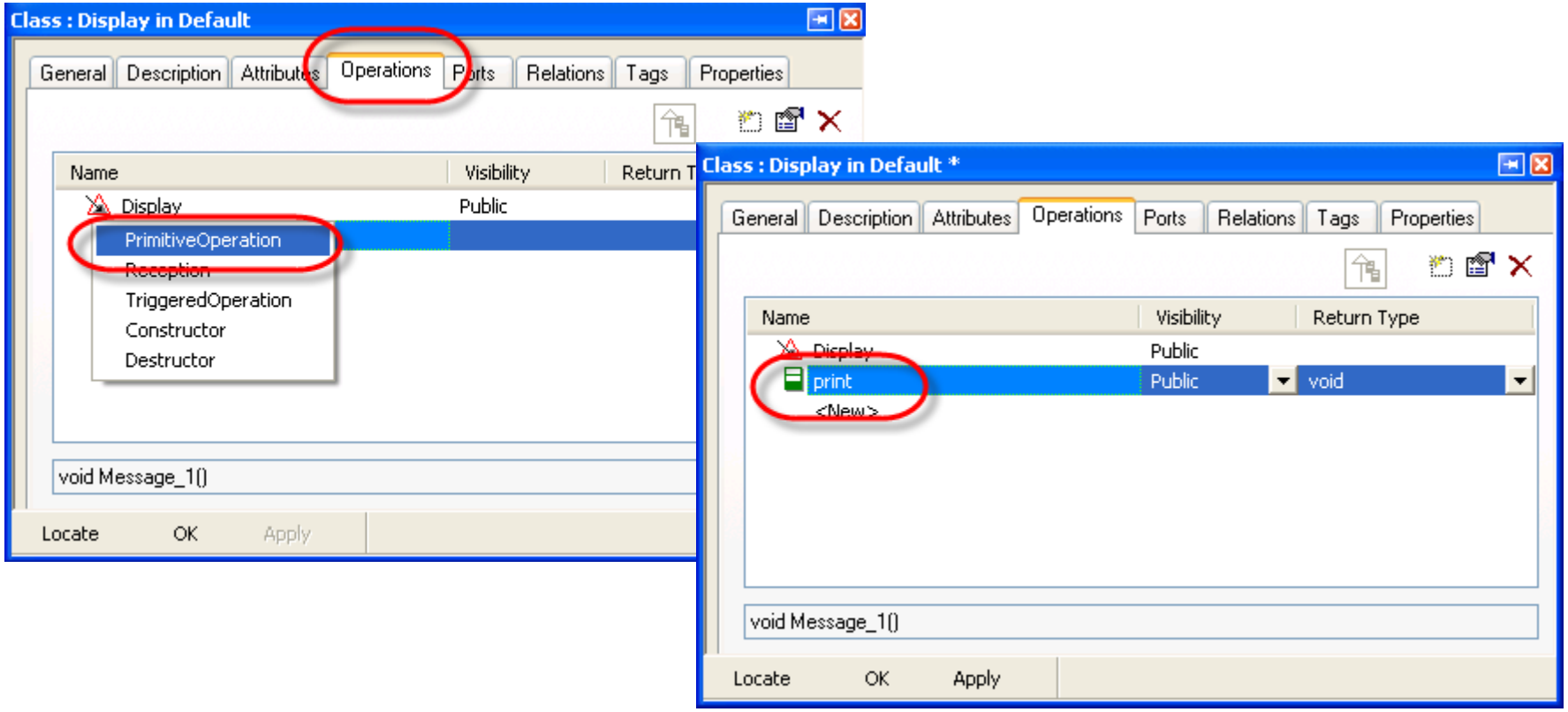
```
protected :

    //## 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
```

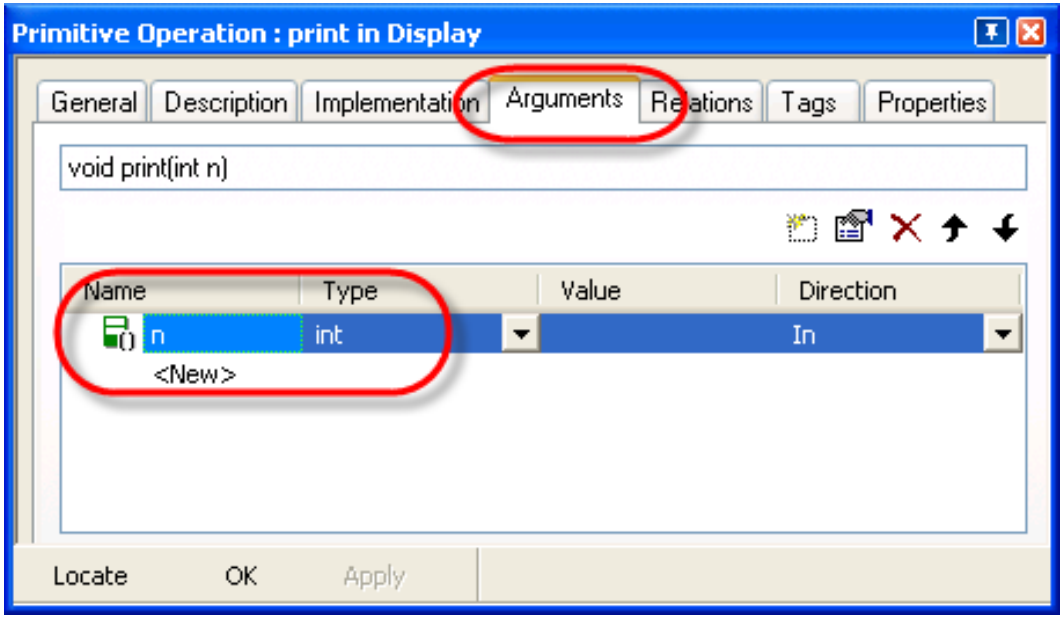
Adding an operation

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



Arguments

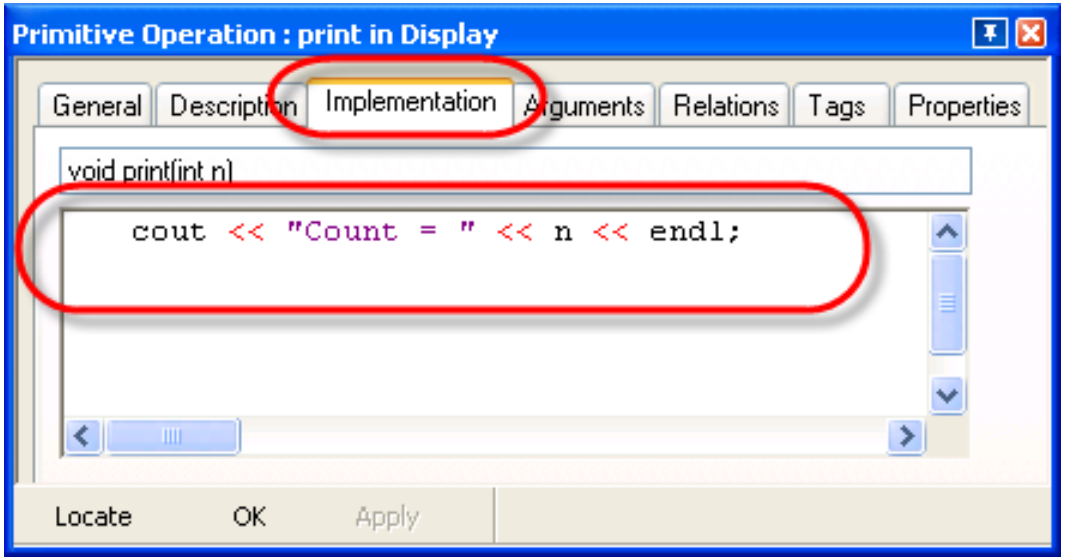
- Double-click **Print** to open the features for the print operation.
- Select the **Arguments** tab.
- Add an argument *n* of type *int.*



Adding implementation

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

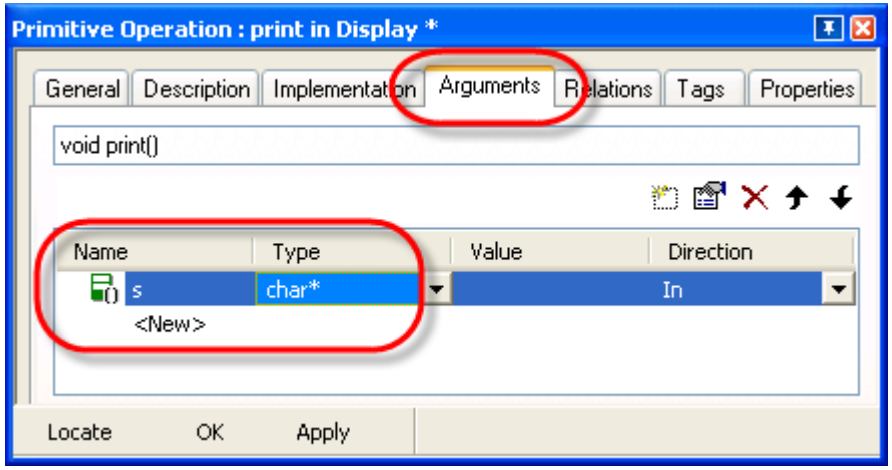
```
cout << "Count = " << n << endl;
```

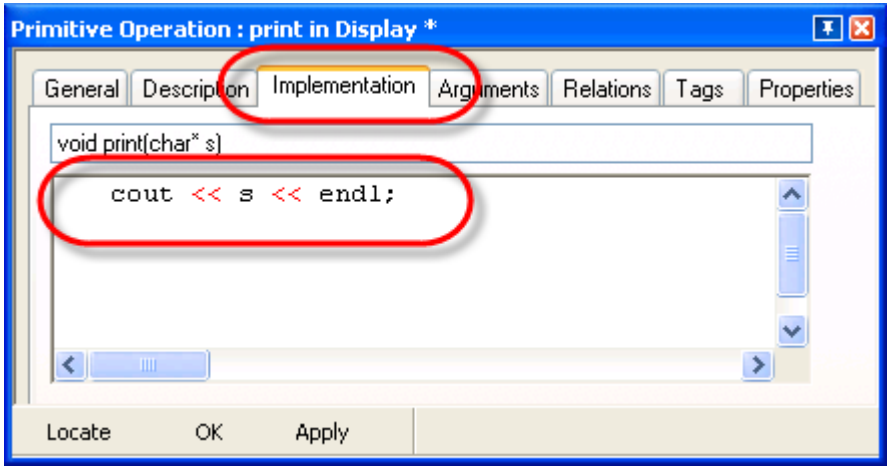


Another print operation

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

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



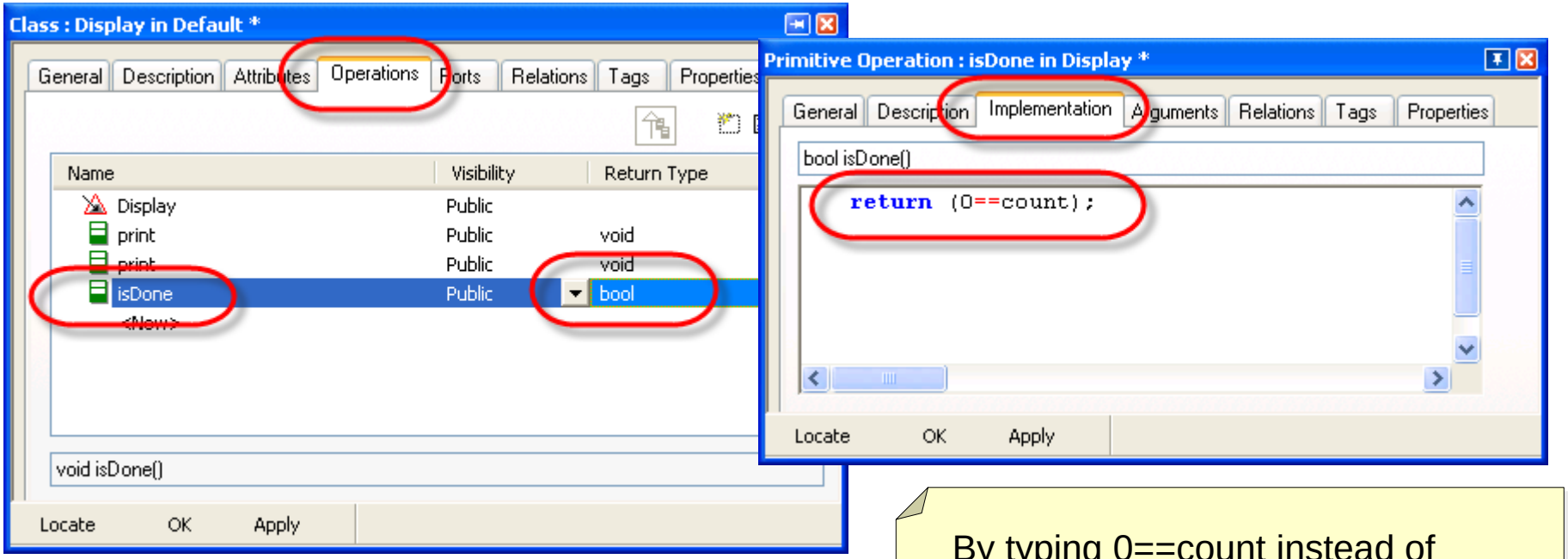


Set the argument type before setting the name. This avoids a conflict where the two print operations have identical signatures.

Operation isDone()

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

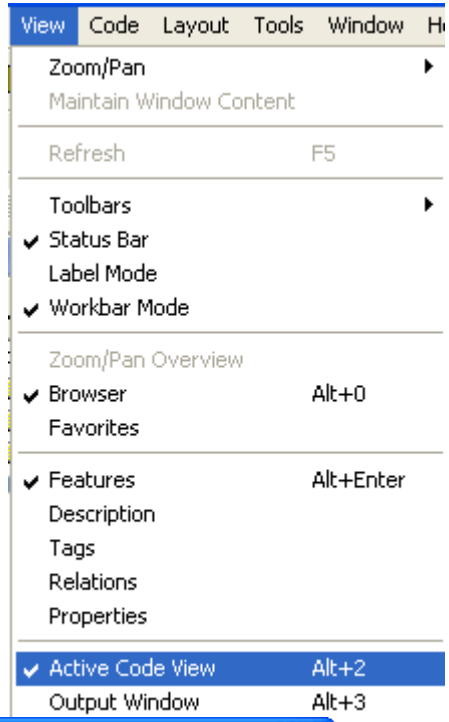
```
return (0==count);
```

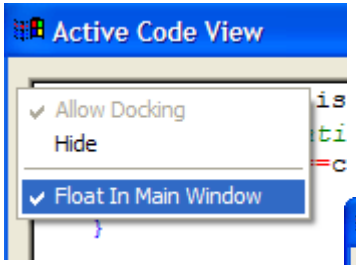


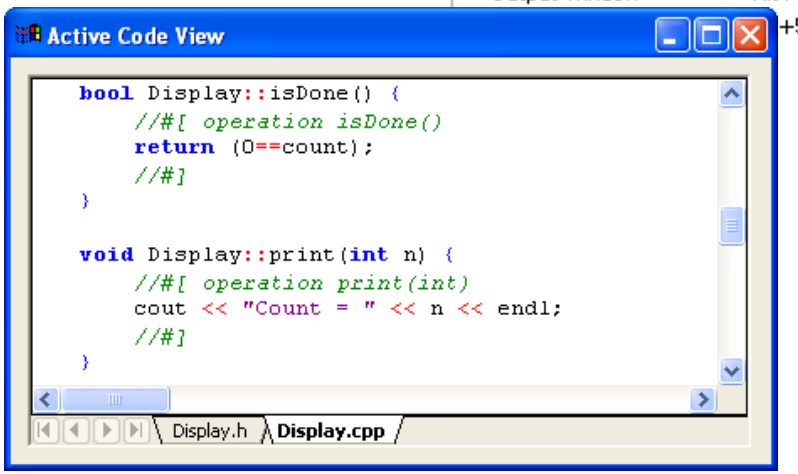
By typing 0==count instead of count==0, enables the compiler to detect the common error of where = is typed instead of ==.

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.



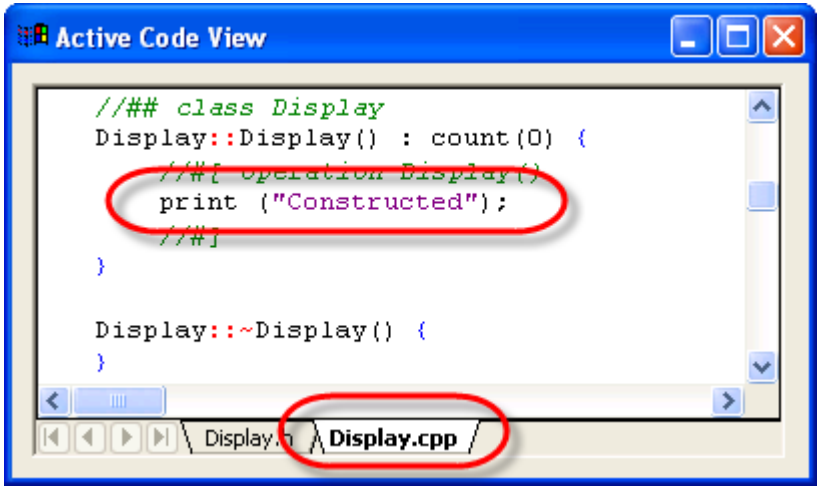




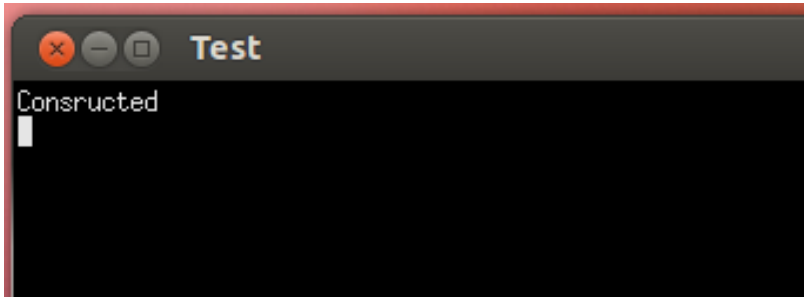
Note that although leaving the active code view always open is useful, it does slow down model manipulation since the code will get regenerated anytime any model element gets modified.

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.



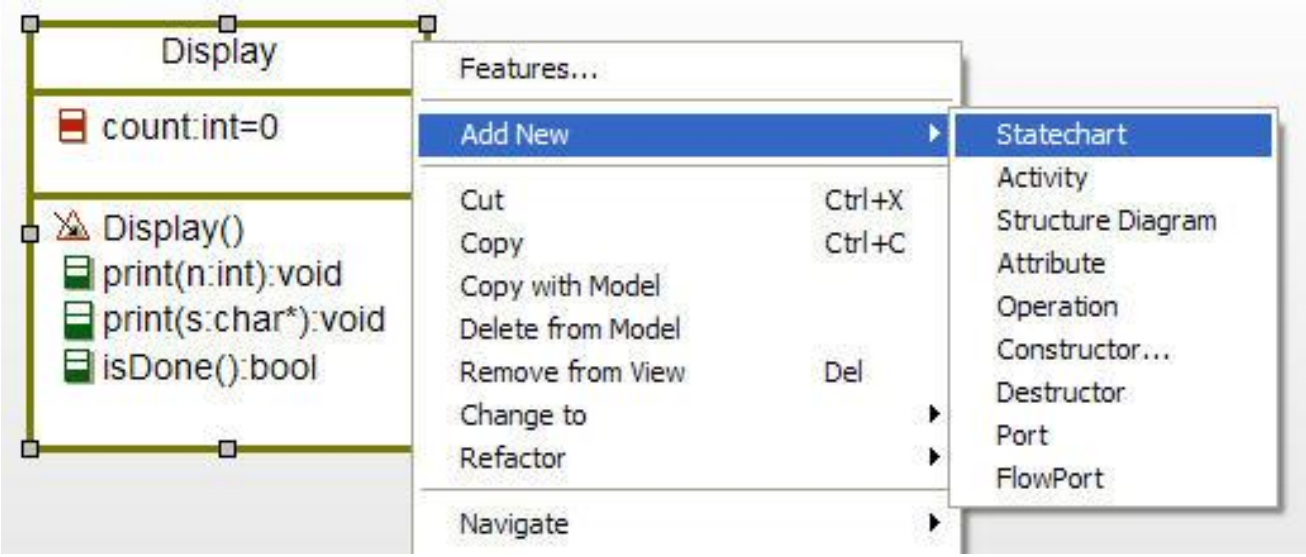
- Change the focus to another window such as the browser and check that this modification has been automatically round-tripped.



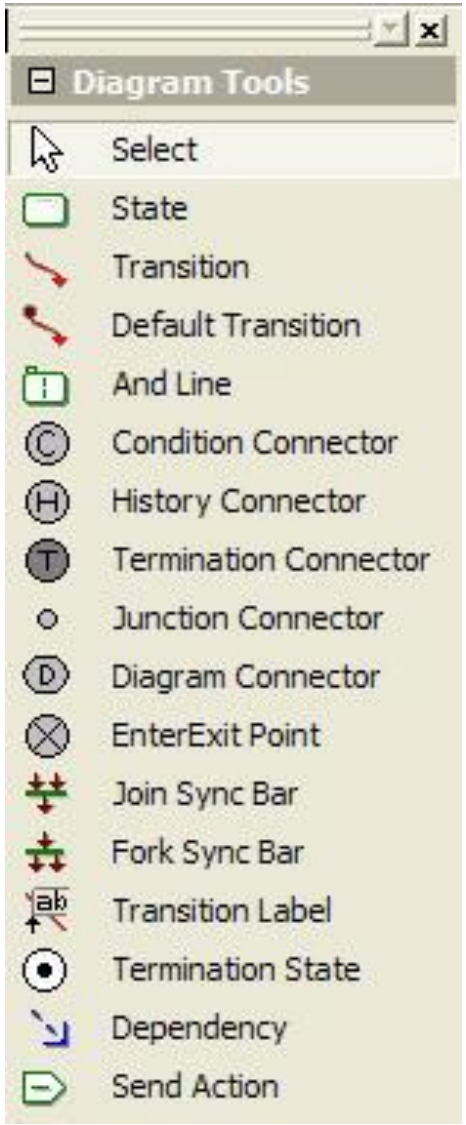
- Save the changes.
- Generate / Make / Run.

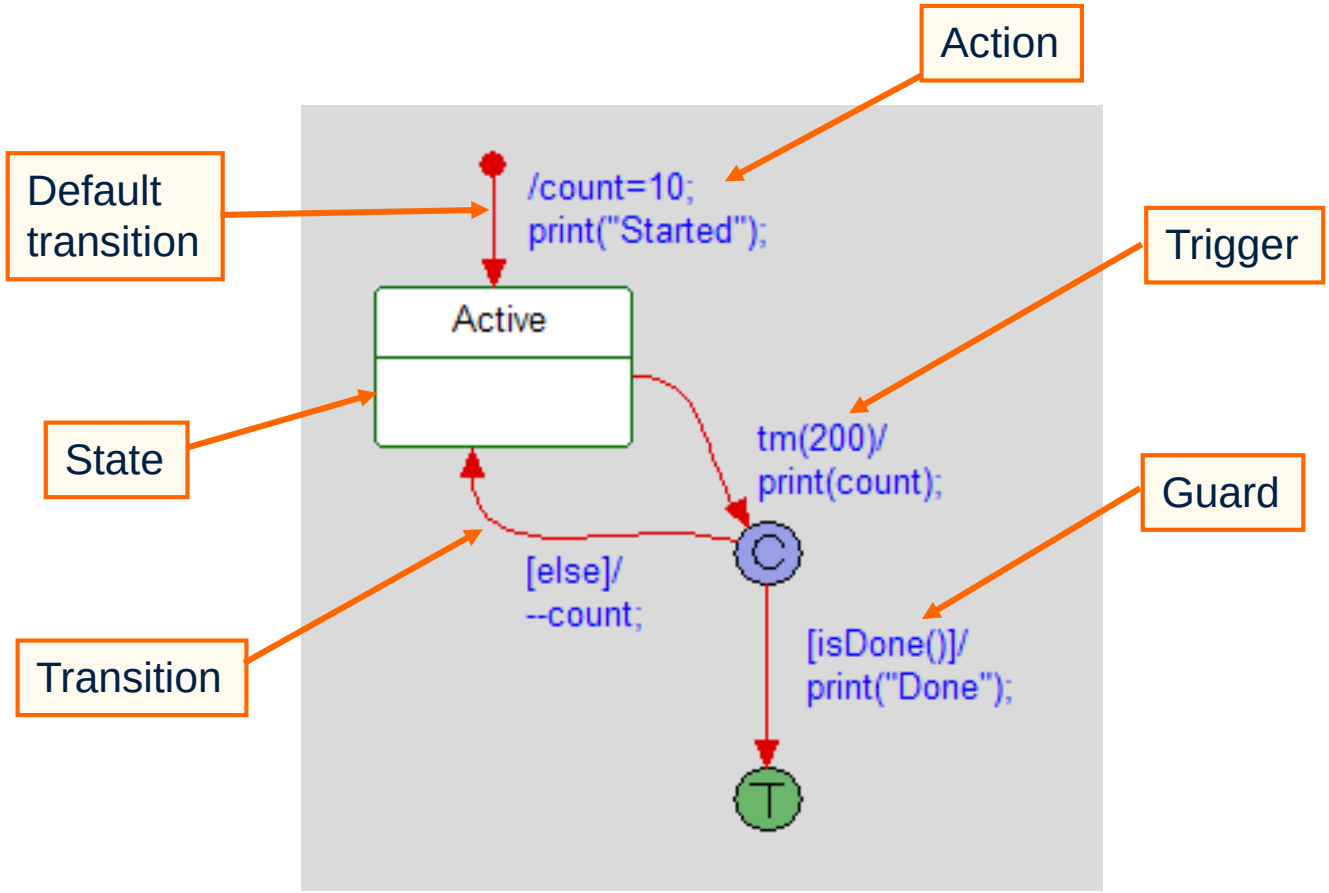
Adding a statechart

- 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**.



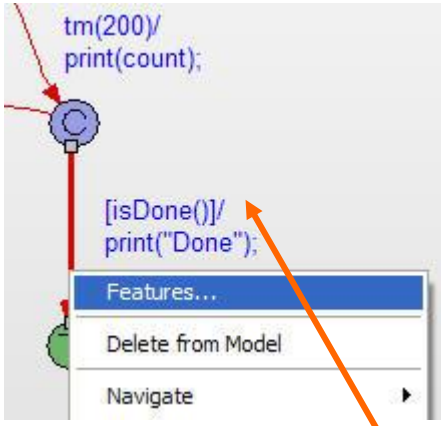
Draw a simple statechart



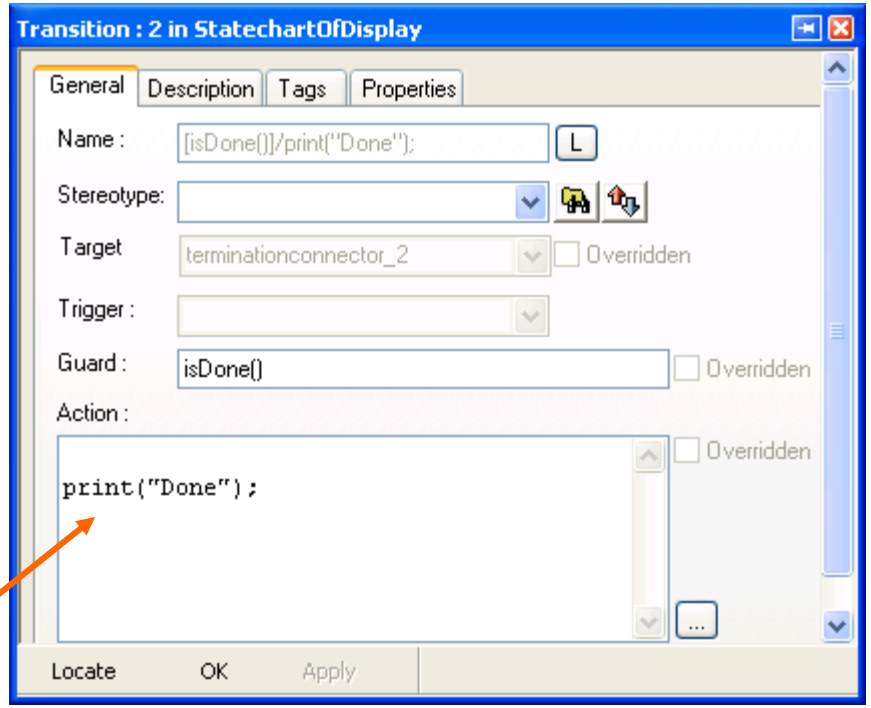


Transitions

- 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

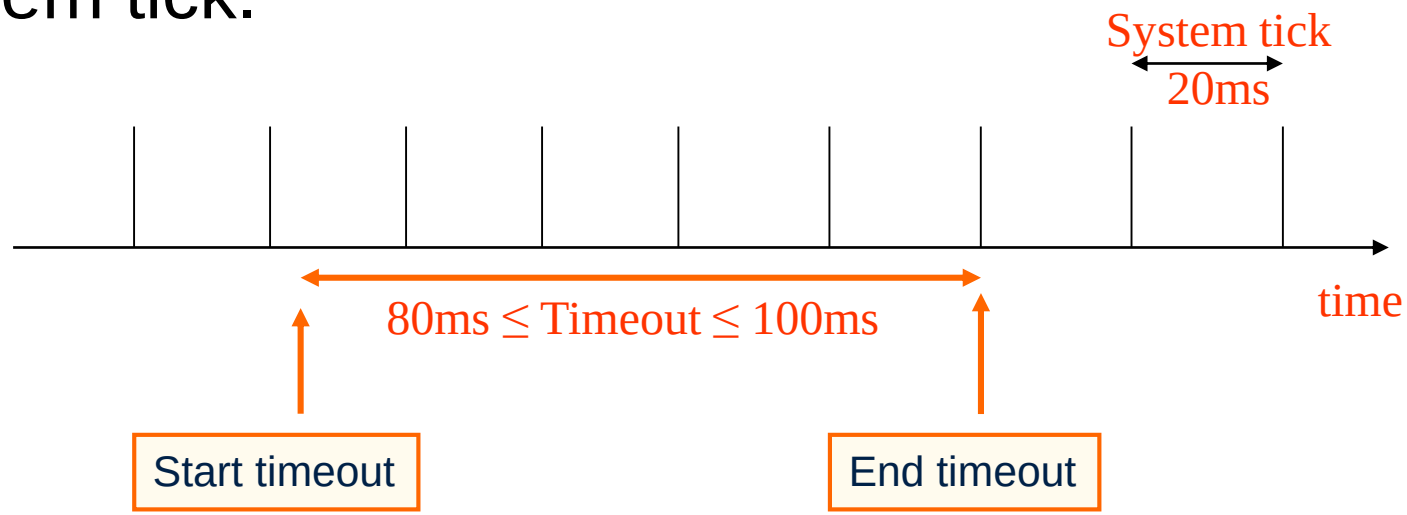
- A timer is provided that can be used within the statecharts.
- tm(200) acts as an event that will be taken 200ms after the state has been entered.
- When entering into the state, the timer will be started.
- When exiting from the state, the timer will be stopped.

```
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.



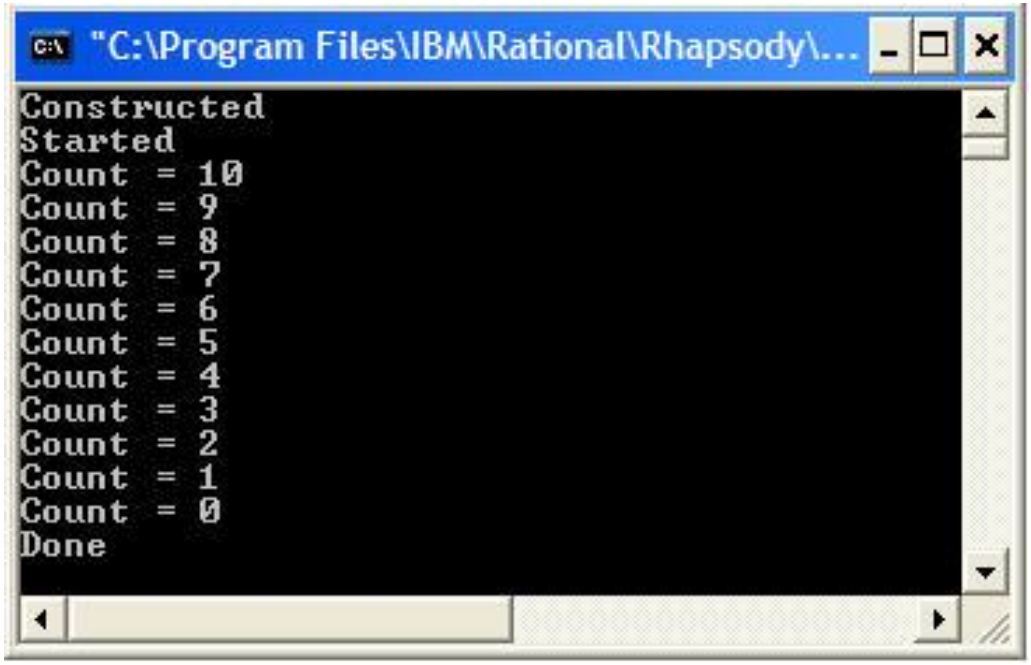
If precise timeouts are required, then it is recommended you use a hardware timer in combination with triggered operations.

Counting down

- Save
- Generate / Make / Run

Constructor

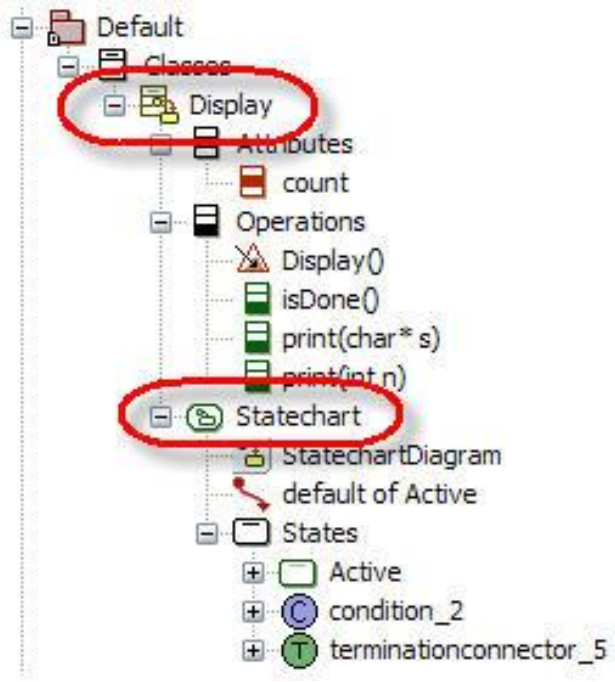
Default Transition

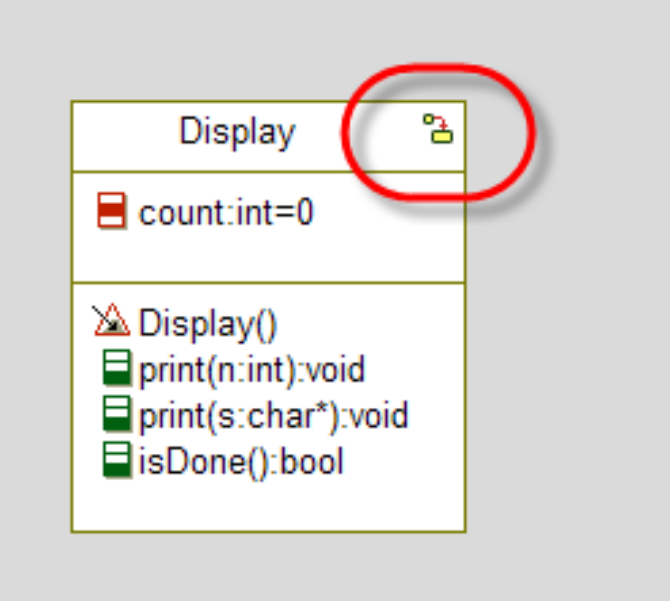


Do NOT forget to close this window, before doing another Generate / Make / Run.

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.

Framework class

Framework includes

Thread on which to wait

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);
```

Display.h | Display.cpp

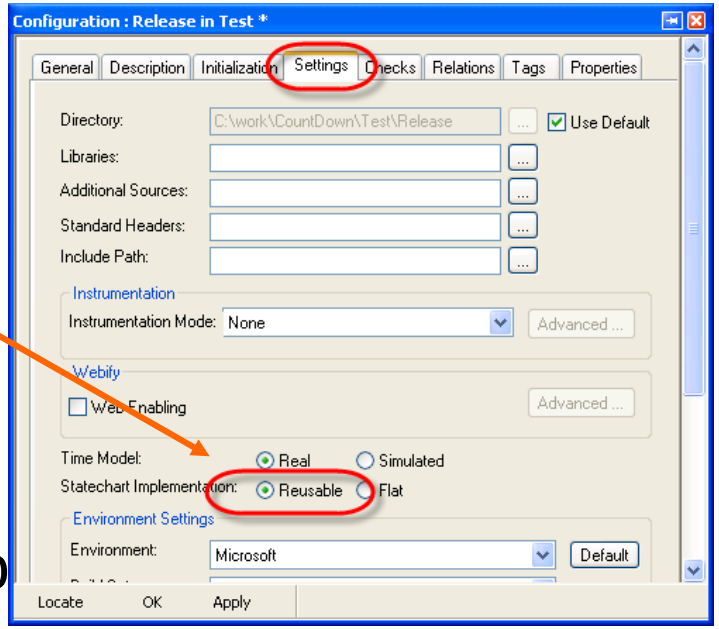
Note that the Display class inherits from OMReactive, which is one of the framework base classes. This is a class that simply waits for timeouts or events. When it receives a timeout or an event, it calls the rootState_processEvent() operation.

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 though OXF operation OMReactive::processEvent() whenever the object receives an event or timeout.

Statechart implementation

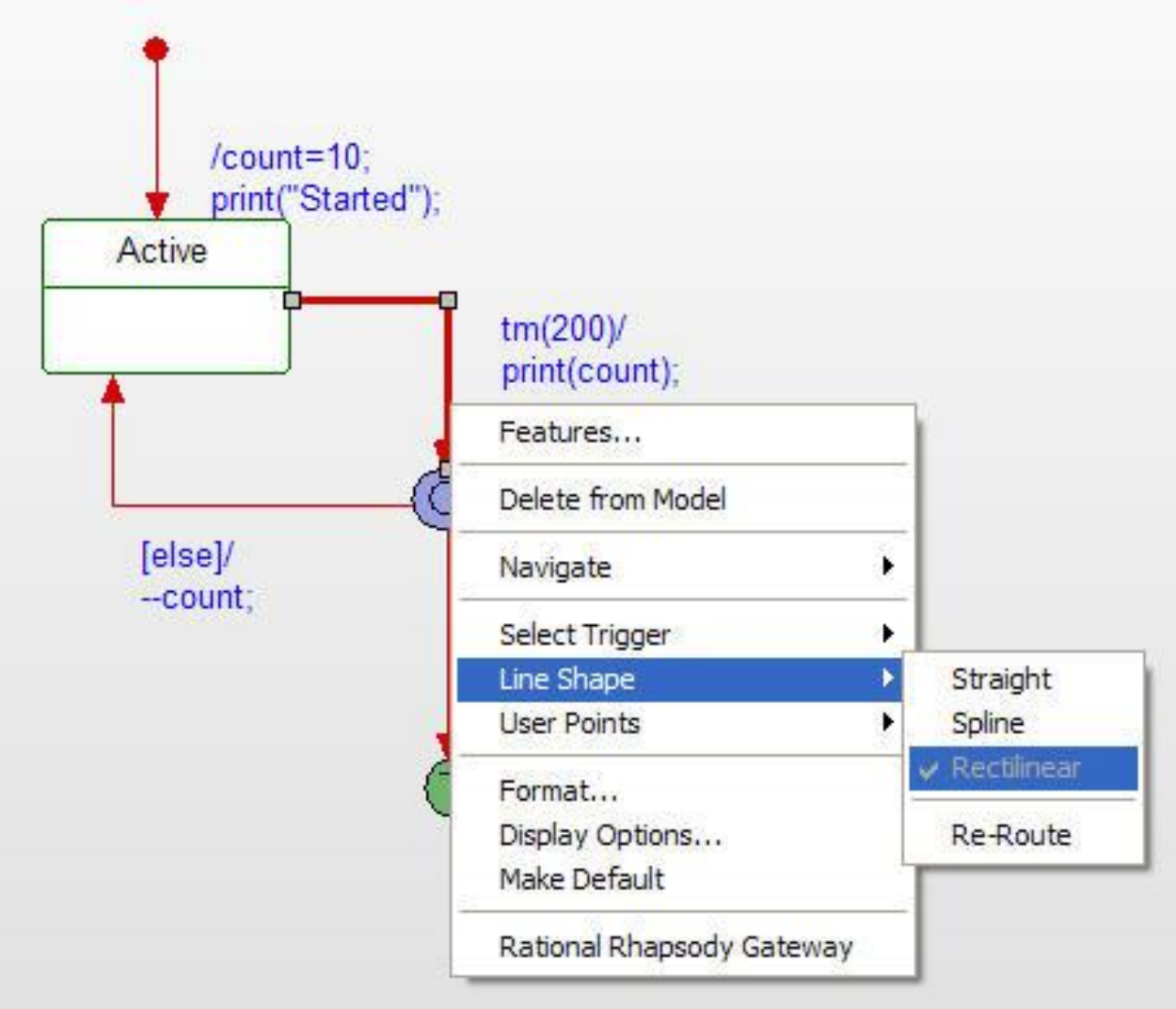
- Change the statechart implementation
  - Select the features for the configuration *Release.*
  - Select the Settings tab and set Statechart Implementation from *Flat* to *Reusable.*
  - Save / Generate / Examine code.



- The Rational Rhapsody framework allo 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.

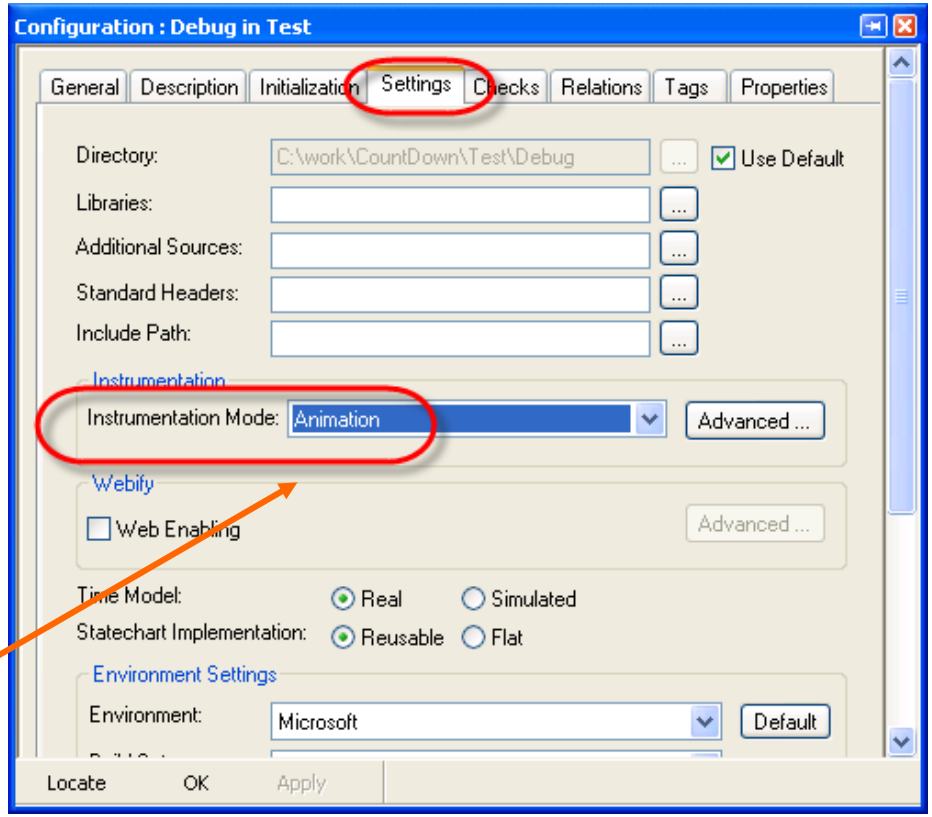


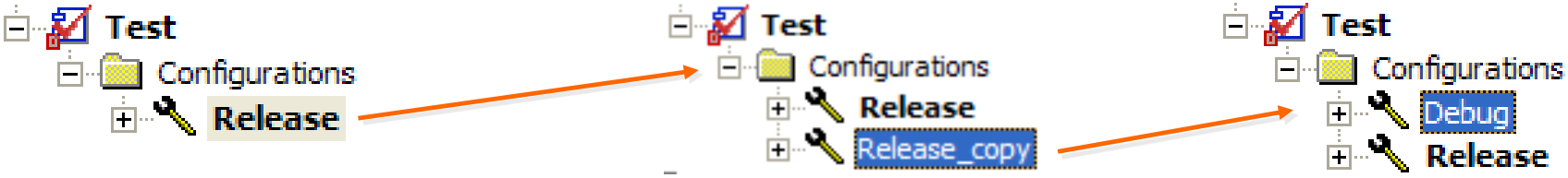
Design level debugging

- 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

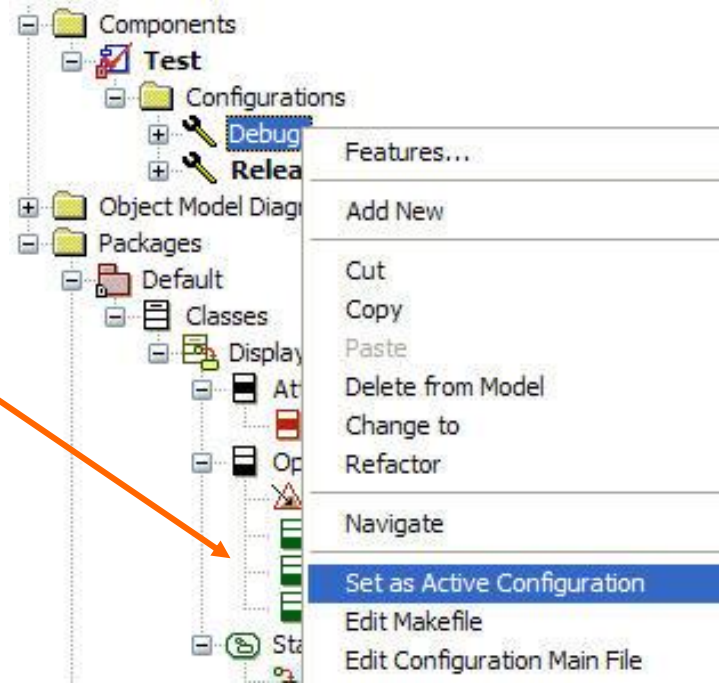
- Create a configuration by copying the *Release* configuration:
  - Press **Ctrl** and drag the *Release* configuration onto the **Configurations** folder.
  - Rename the new configuration *Debug.*
- Set the **Instrumentation Mode** to *Animation.*

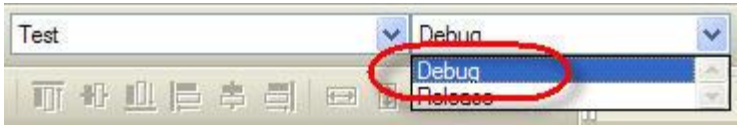




Multiple configurations

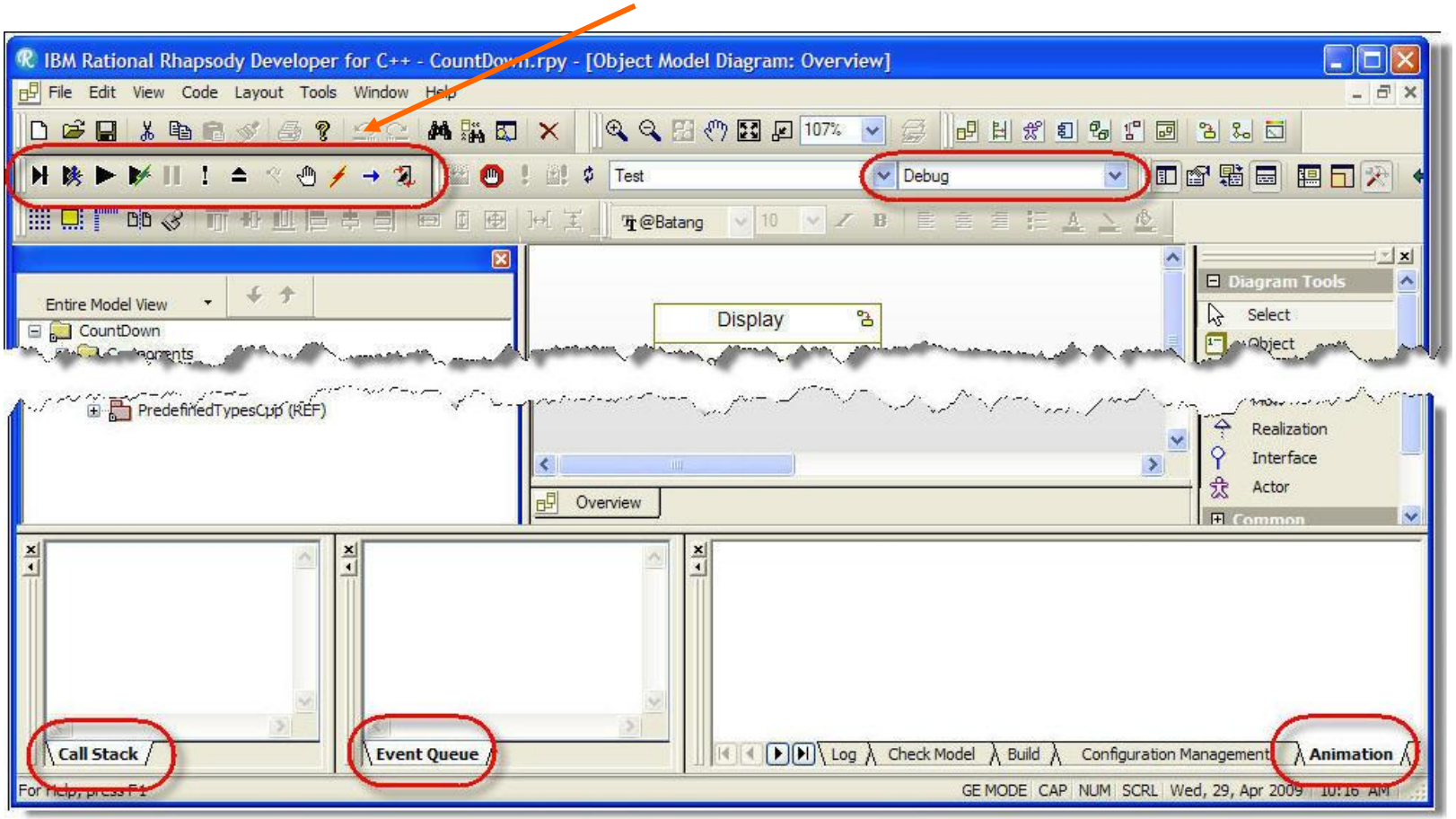
- Now that you have more than one configuration, you must select which one you want to use.
- There are two methods:
  - Right-click the configuration and select **Set as Active Configuration.**
  - Select the configuration using the pull-down box.





Animating

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



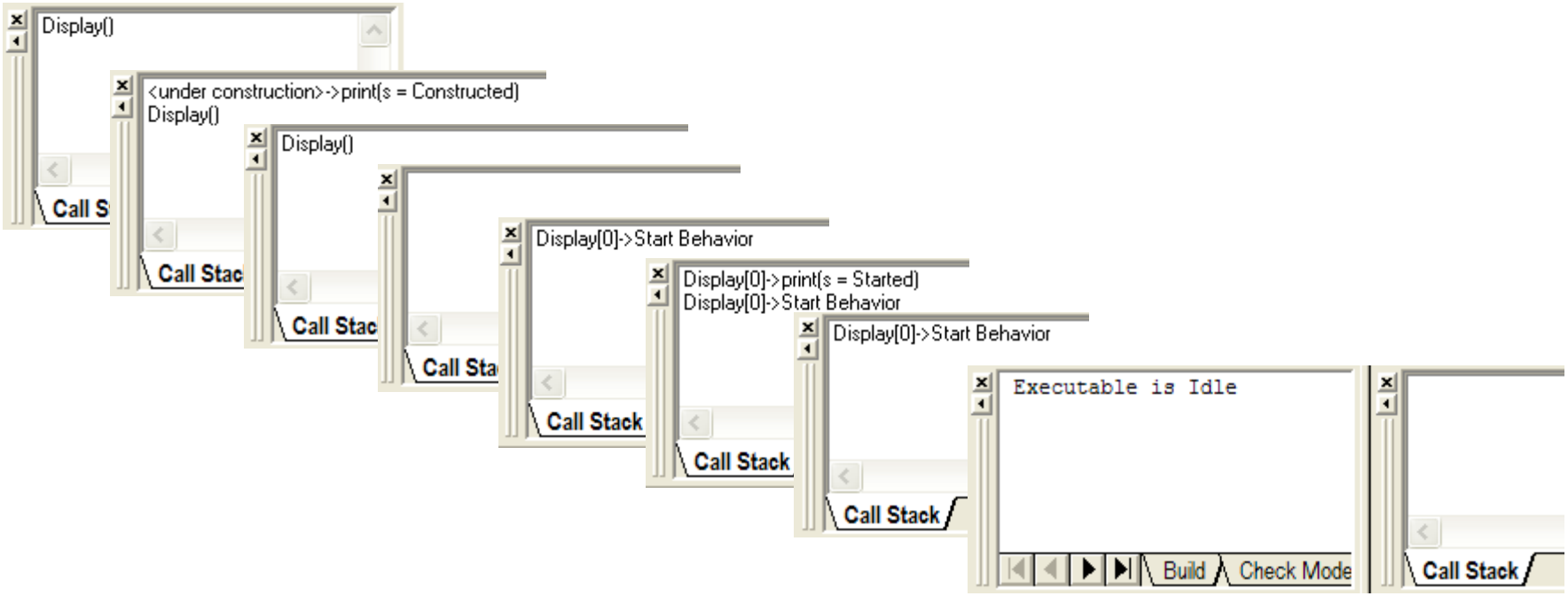
Animation Toolbar

- 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.

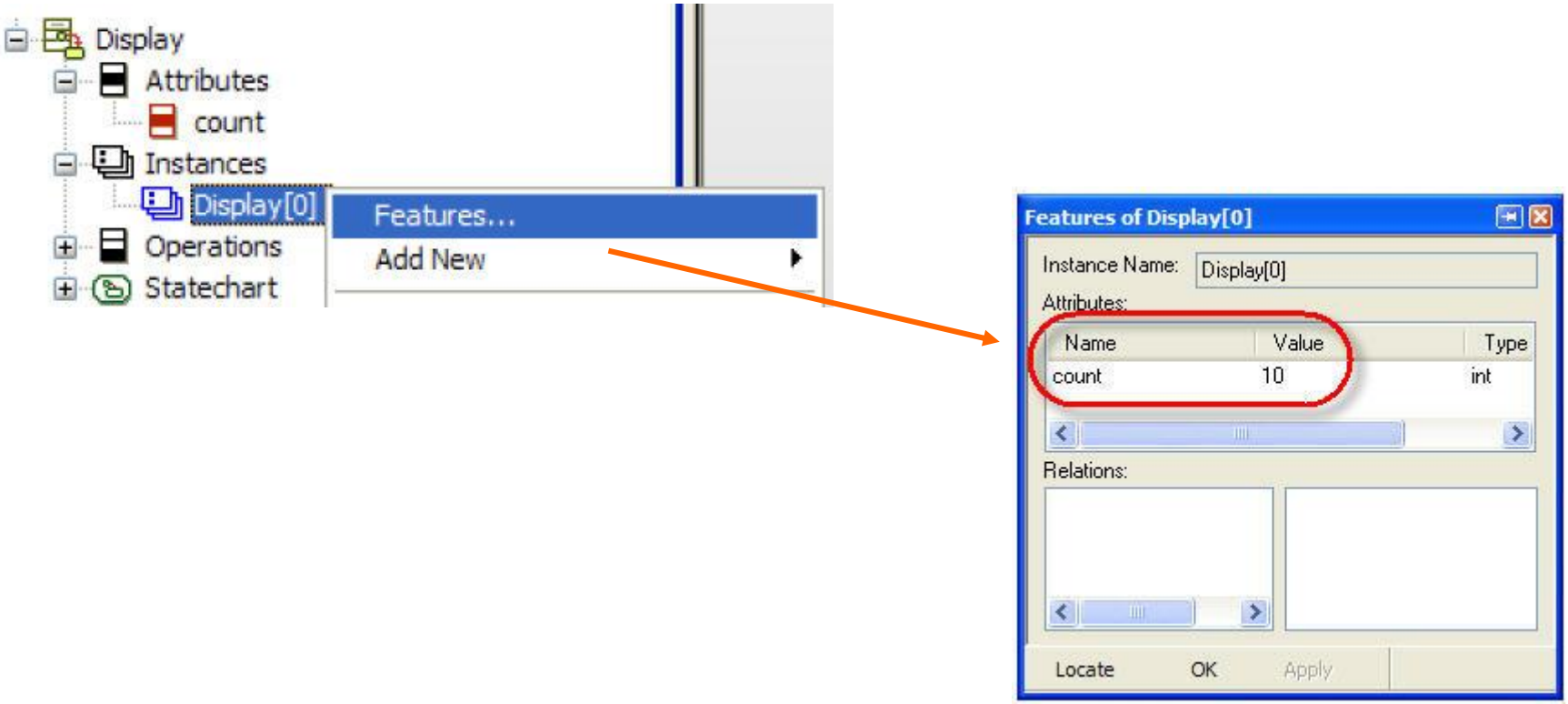
Starting the animation

- 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.



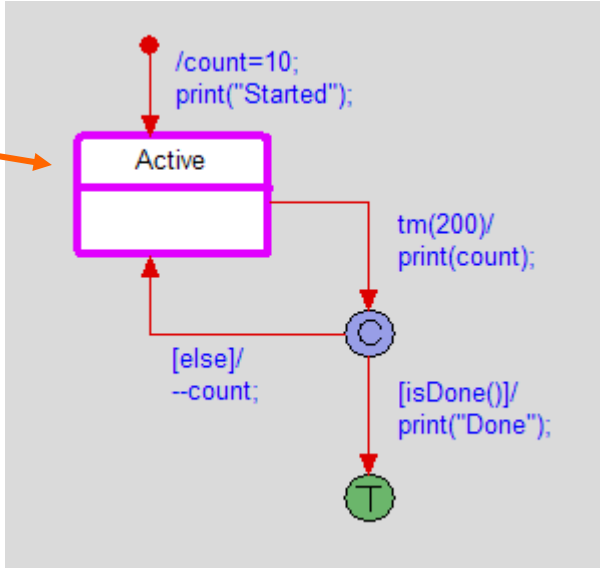
Class Instance

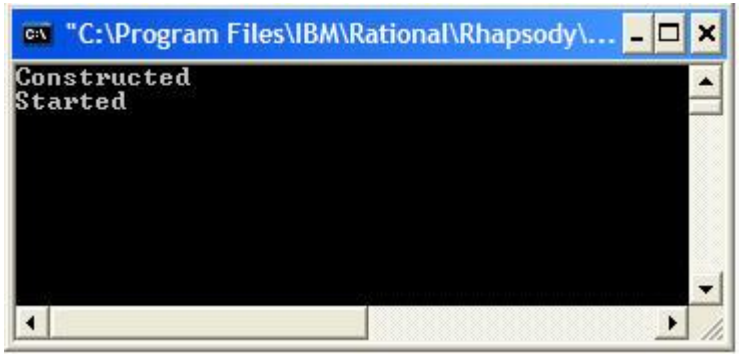
- Browser contains an instance of the *Display* class.
  - Right-click the instance and select **Features.**
  - Note that the attribute *count* has been initialized to 10.



Statechart Instance

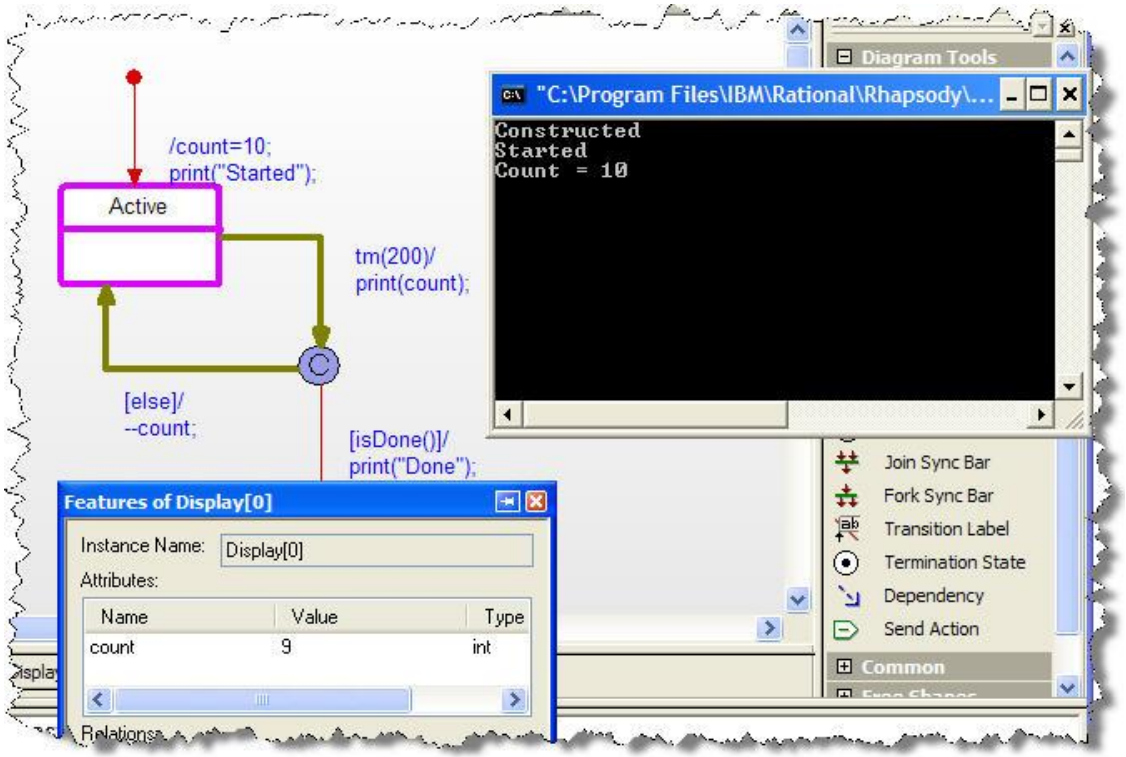
- Right-click the instance and select **Open Instance Statechart.**
  - Highlighted state indicates the current state of the model.
    - If you do not see a highlighted state, you may be looking at the statechart of the class (design) rather than the statechart of the instance (runtime).
  - Default transition has also been triggered.
    - *Started* will have been printed to the display.





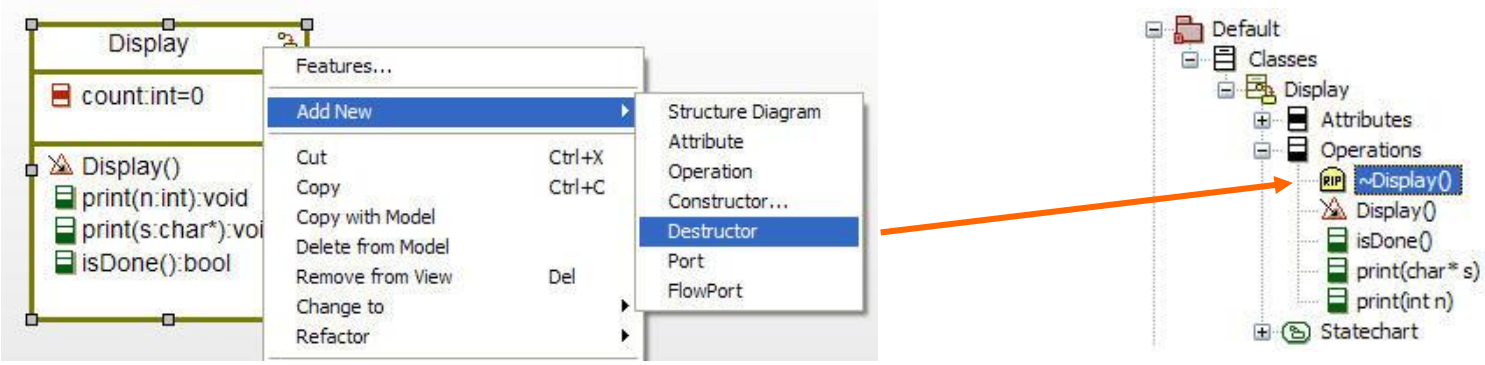
Go Idle / Go

- 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.

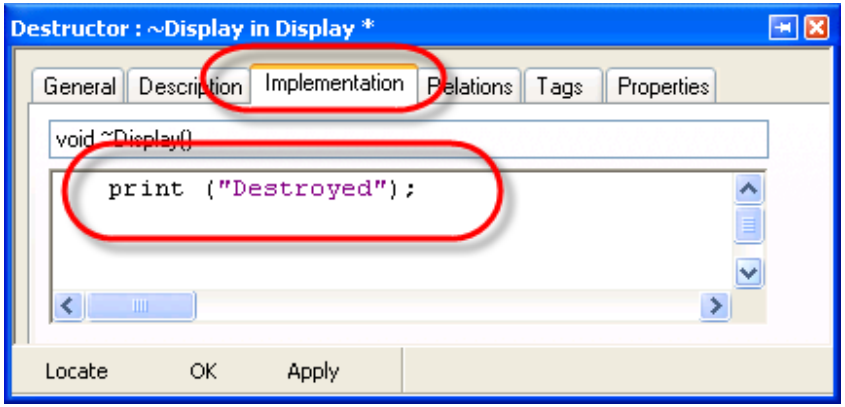


Destructor

- Add a Destructor to the *Display* class.



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

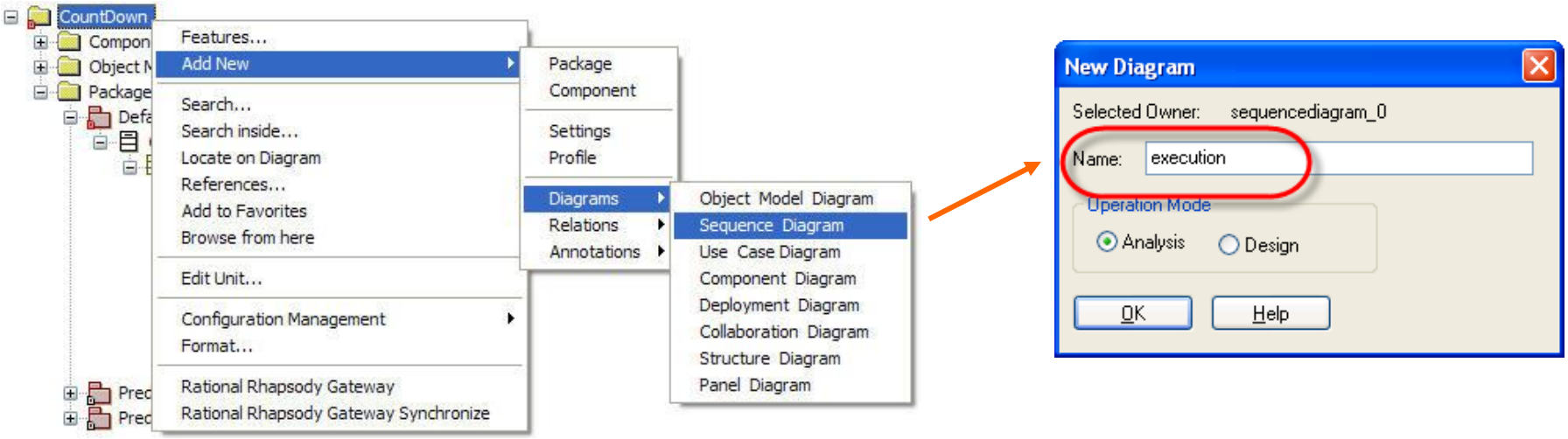


Make sure you enter the code into the Implementation and not the Description field.

- Save then Generate / Make / Run.

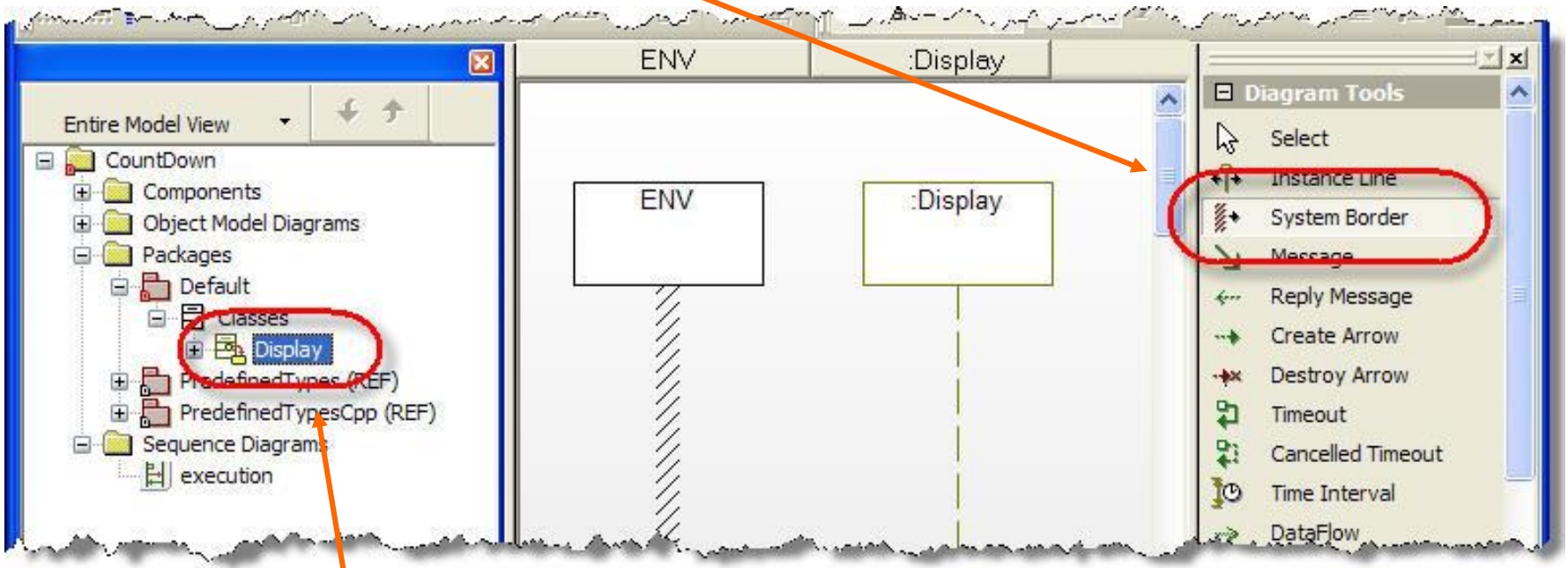
Sequence diagrams

- 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.

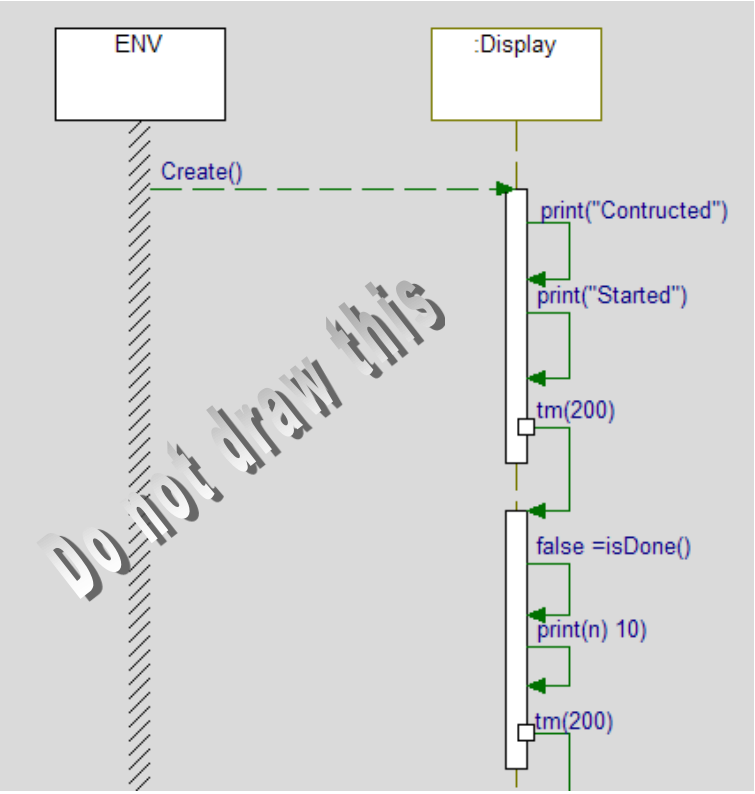


- Drag the *Display* class from the browser onto the sequence diagram.

Drawing a sequence diagram

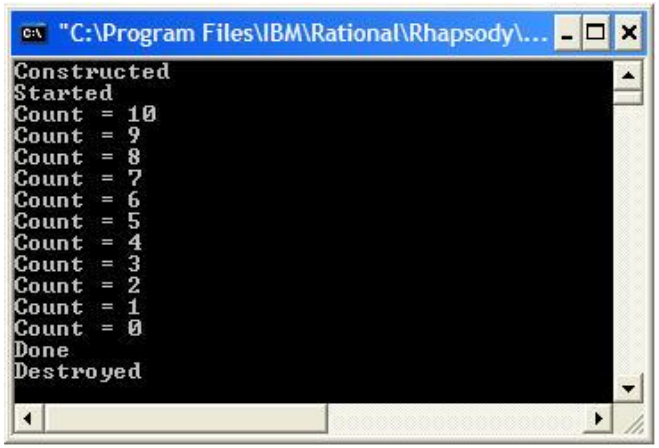
- Normally, you would describe an actual scenario similar to this one here, however in this case, you are more interested in capturing what actually happens.

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.

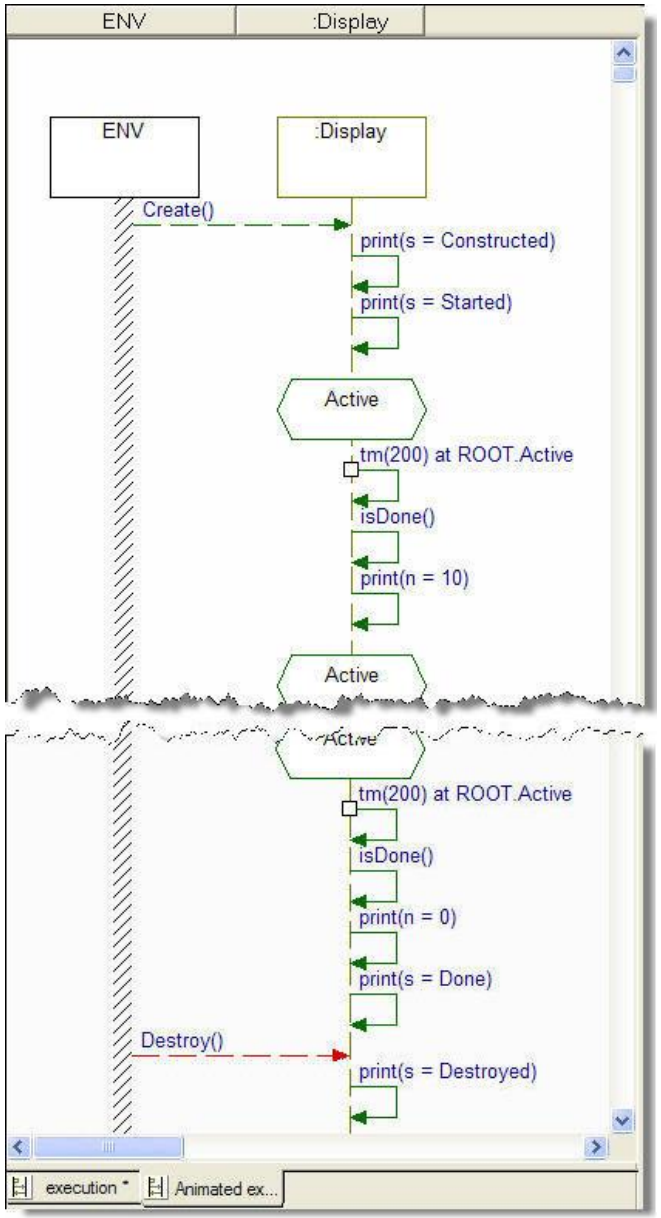


Animated sequence diagrams

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

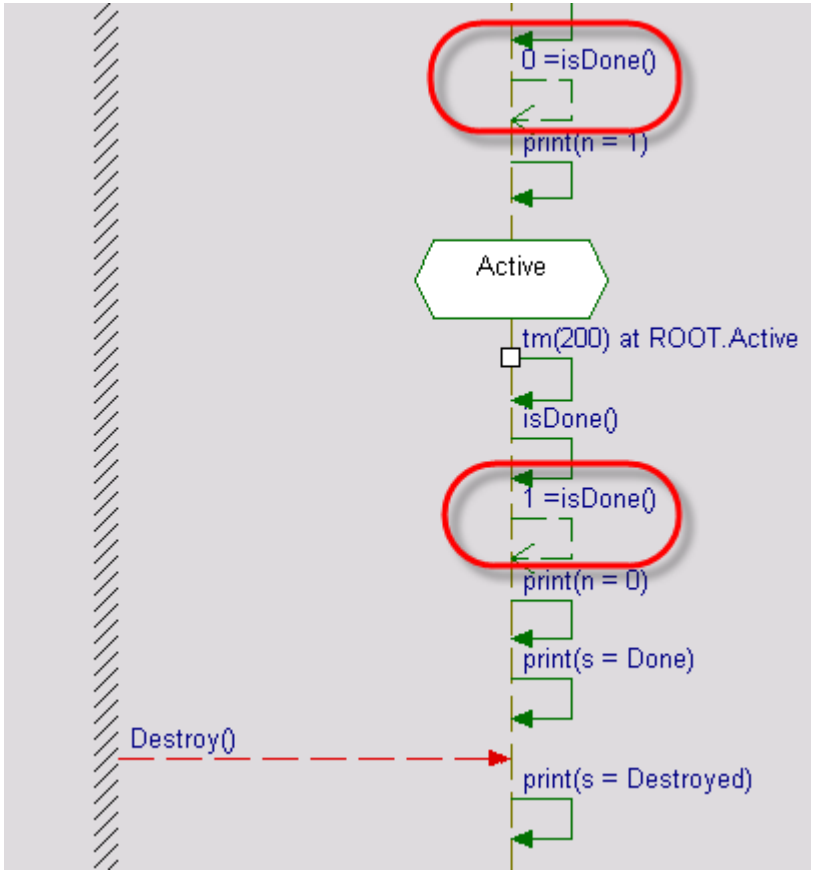


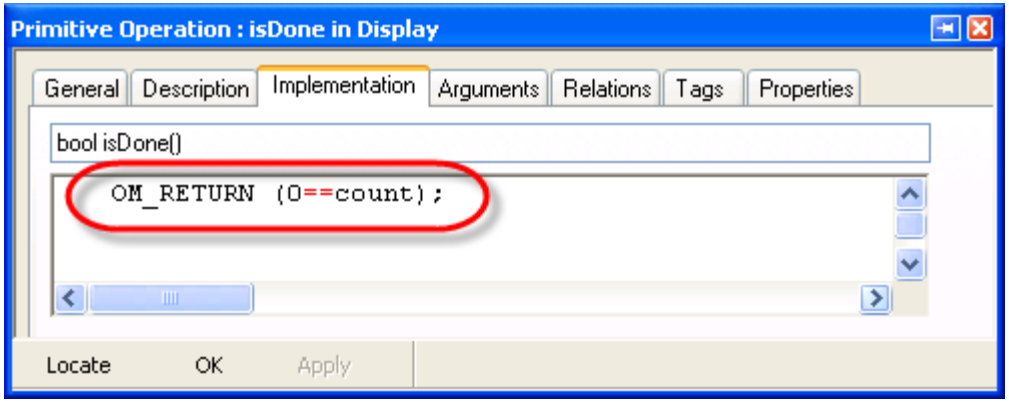
- If a sequence diagram is open, then Rational Rhapsody creates a new animated sequence diagram based on the execution.
  - Note that the animated sequence diagram captures operations, timeouts, and states.



Extended exercise I

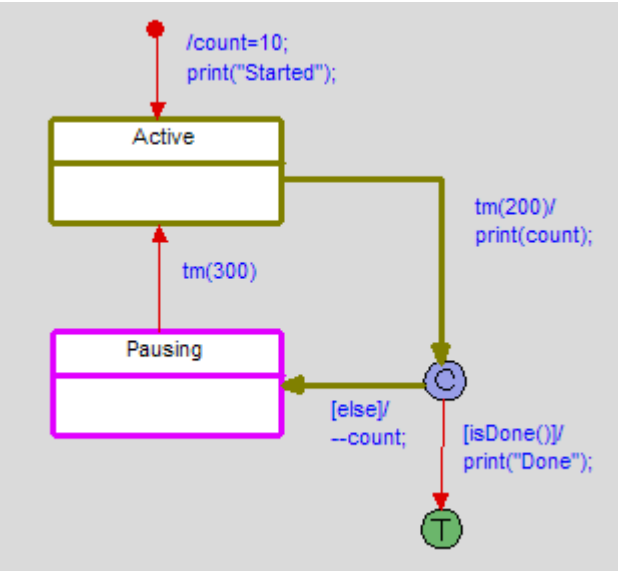
- Rational Rhapsody can display the return value on animated sequence diagrams. To do so, you must use a macro OM_RETURN.
- In the implementation of the operation *isDone()*, replace *return* with *OM_RETURN.*

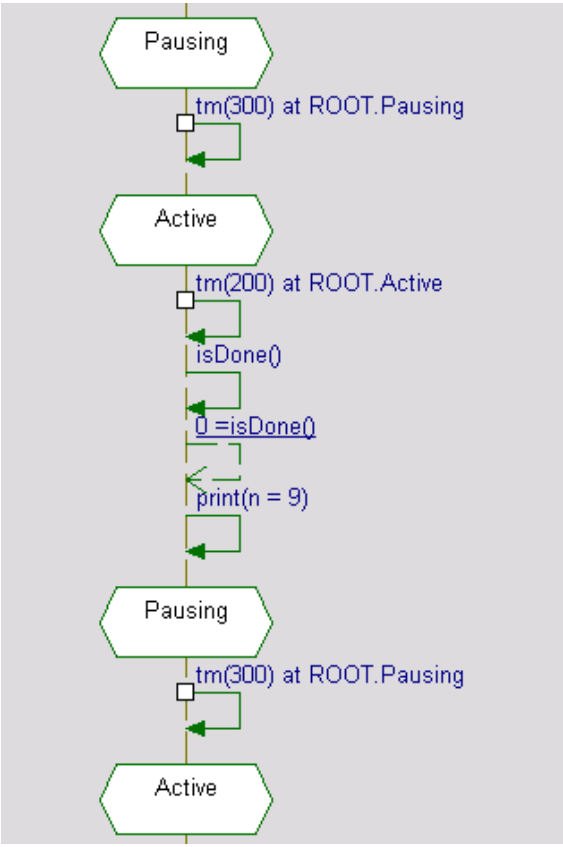


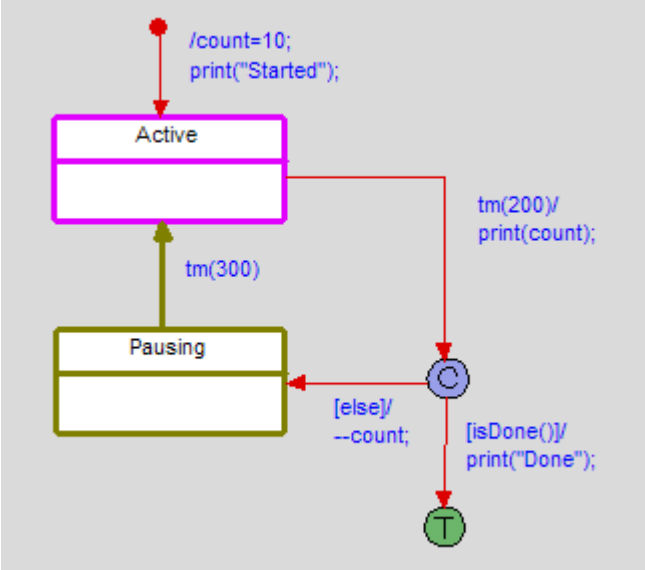


Extended exercise II

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





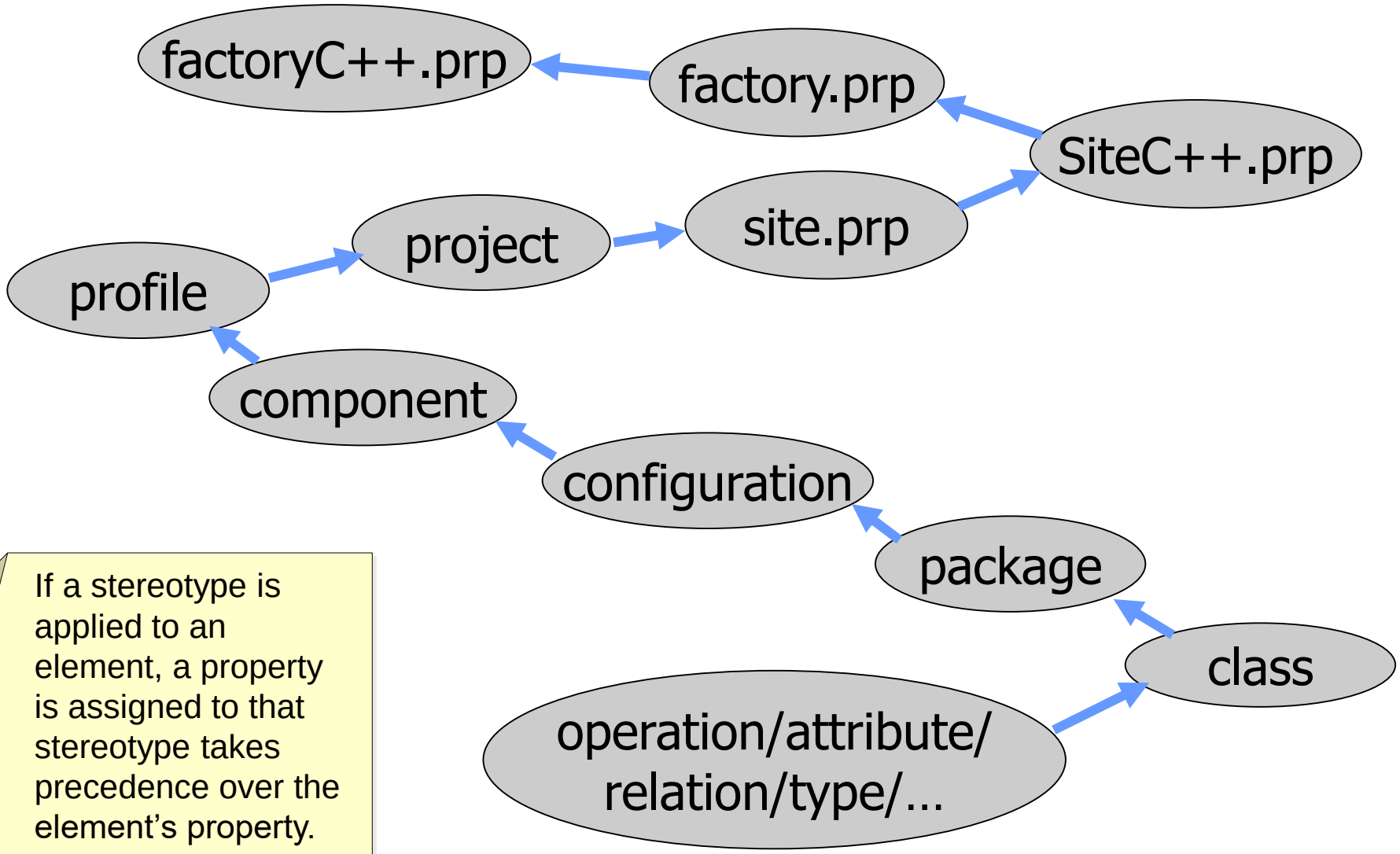


Properties

- 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.

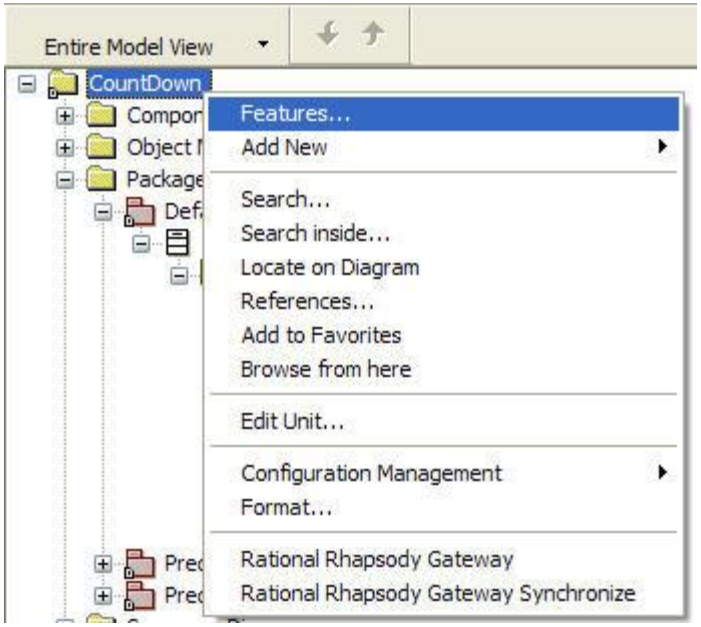
Properties hierarchy

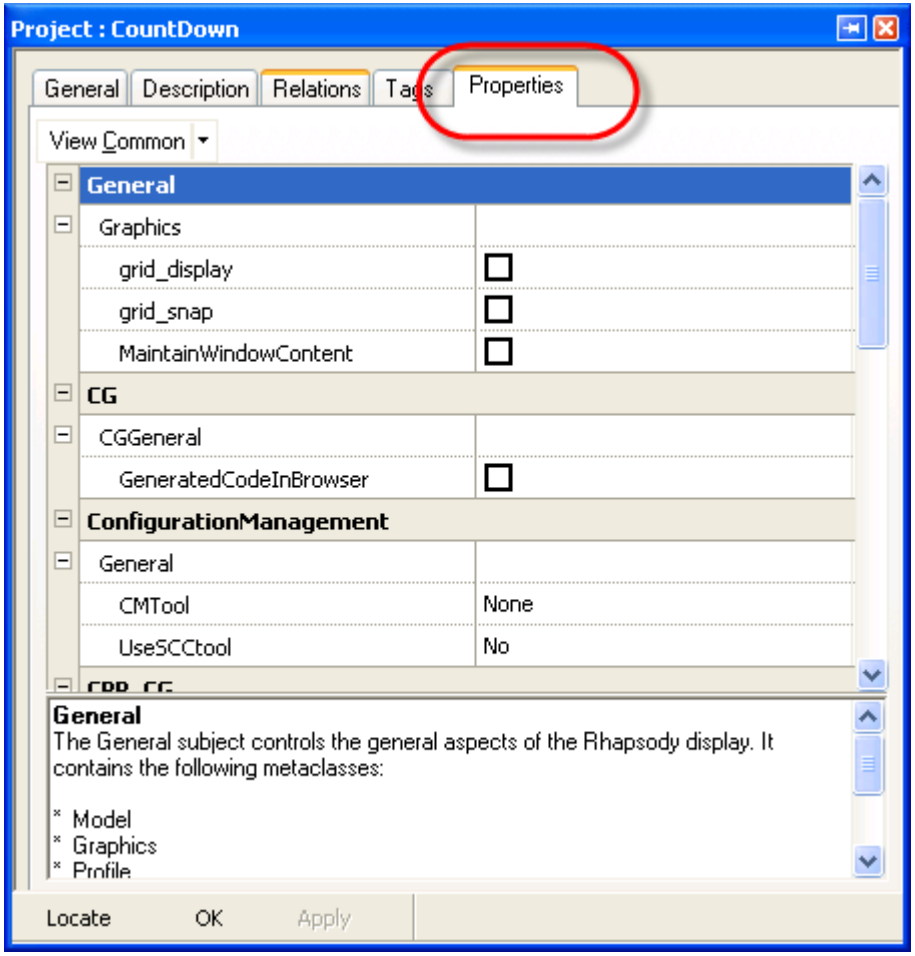


If a stereotype is applied to an element, a property is assigned to that stereotype takes precedence over the element's property.

Project properties

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

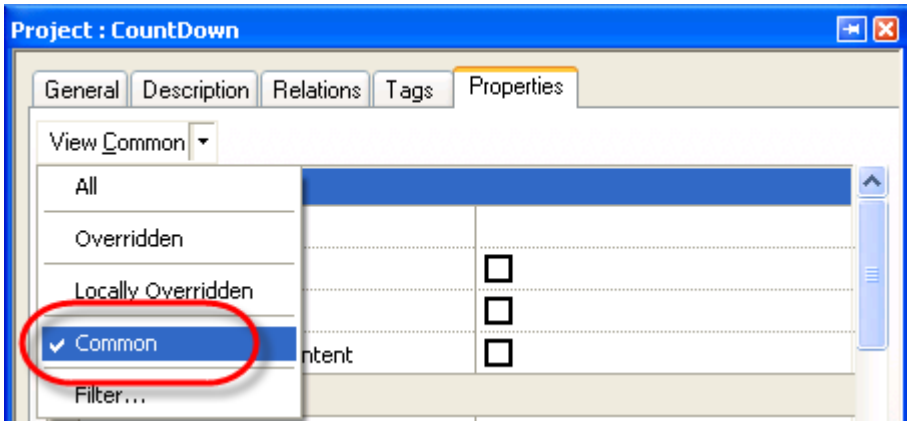




Properties view

- 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.

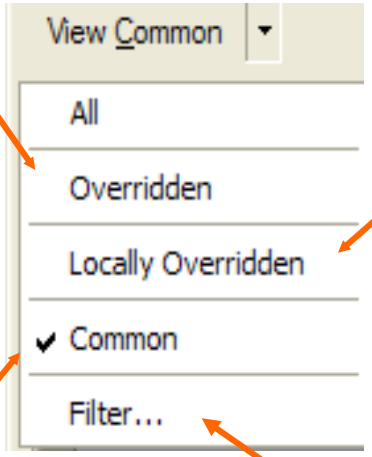


Properties views

- 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

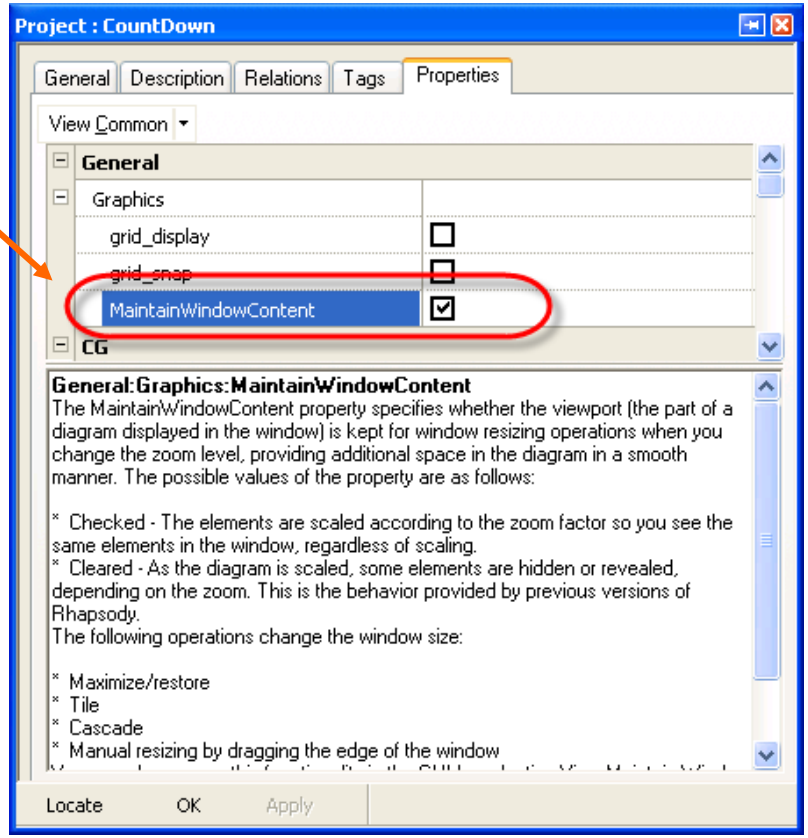
Properties filtered by some keyword

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.

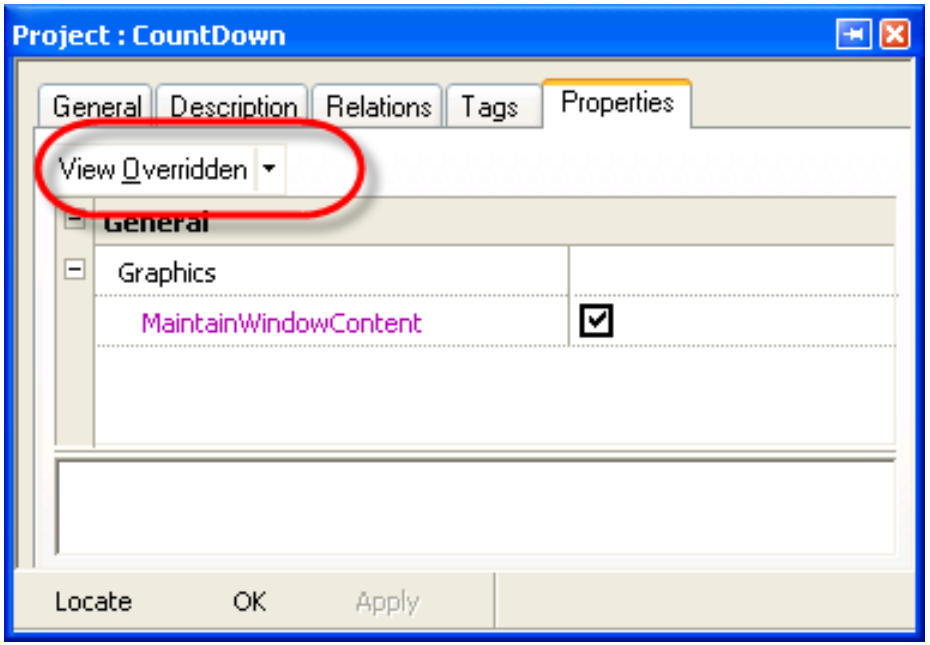
Once a property has been modified it is highlighted. To restore the default, right-click on the property and select **Un-override**.

Note also the description is displayed for the selected property.



Overridden properties

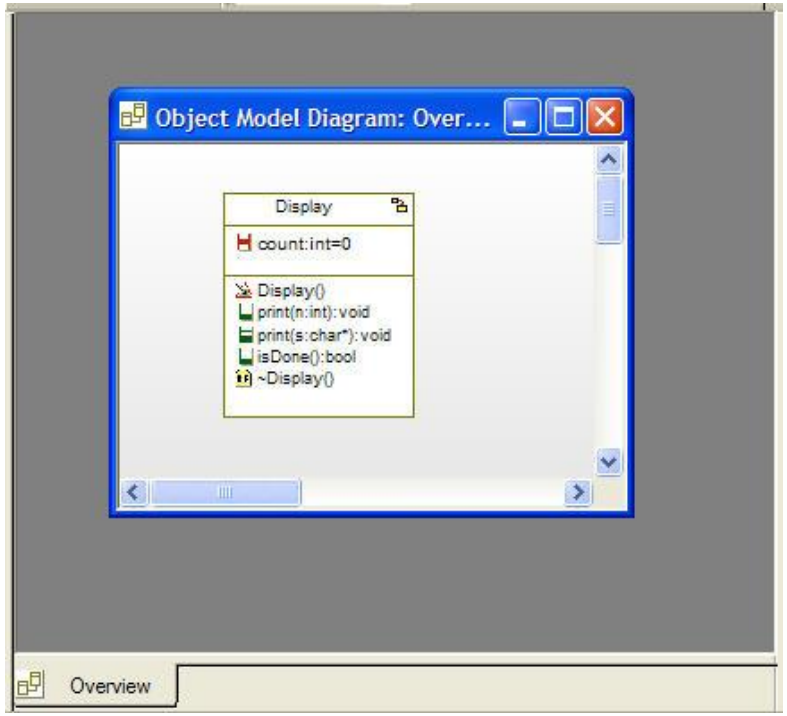
- Select **View Overridden**.

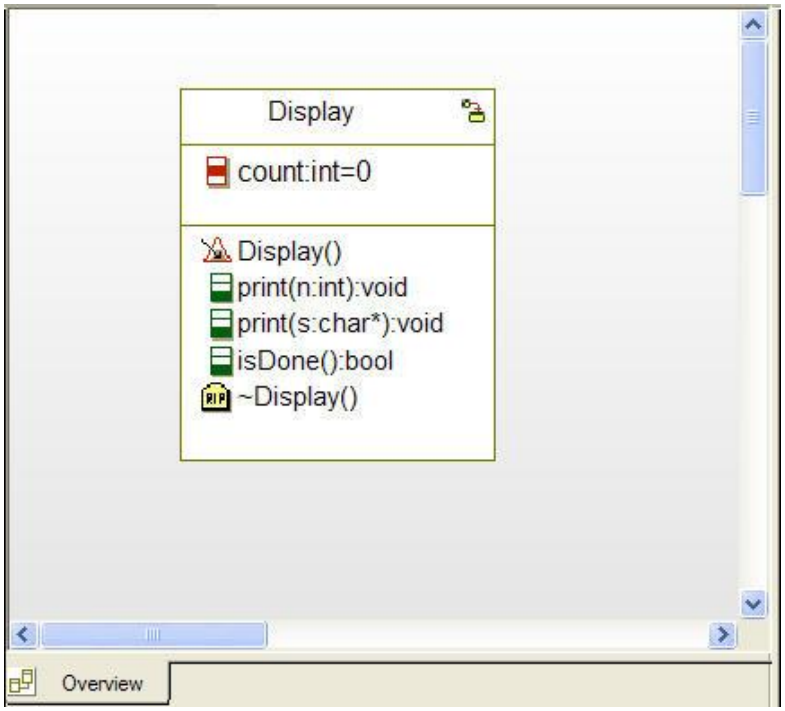


This shows just the properties that have been modified.

General:graphics:MaintainWindowContent

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

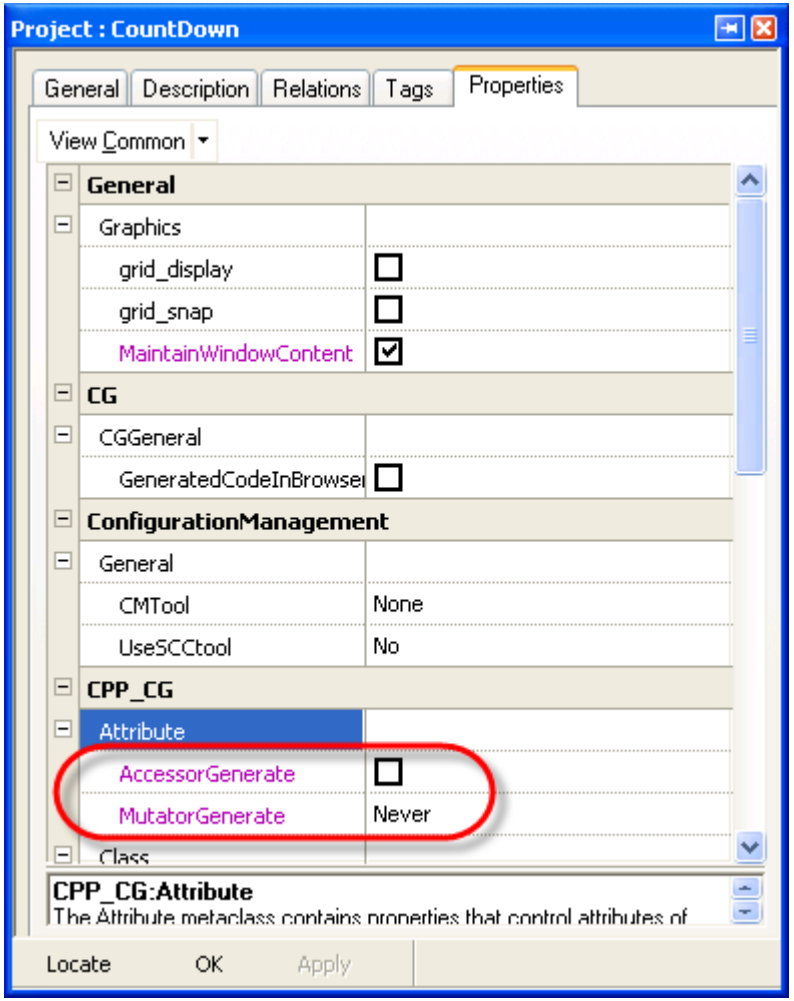




- You need to close any open windows and then reopen them after setting this property.

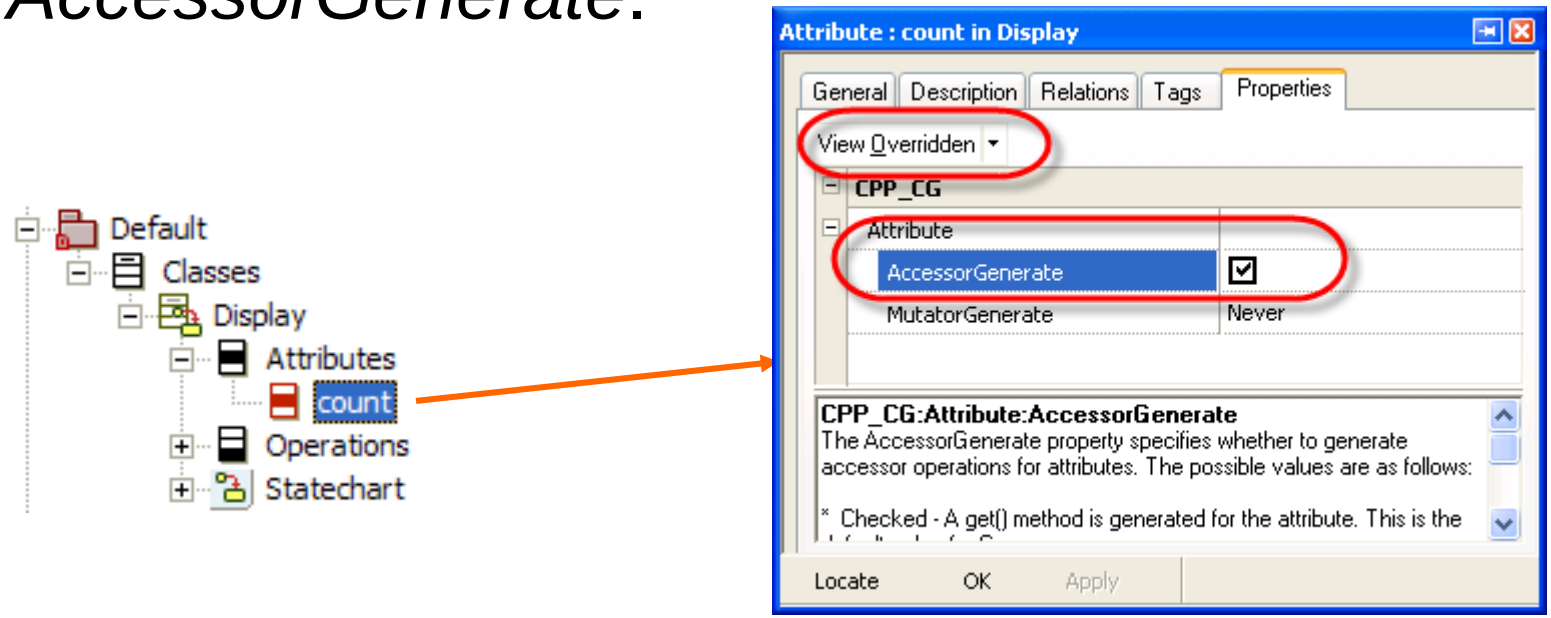
Accessors and mutators

- 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.*



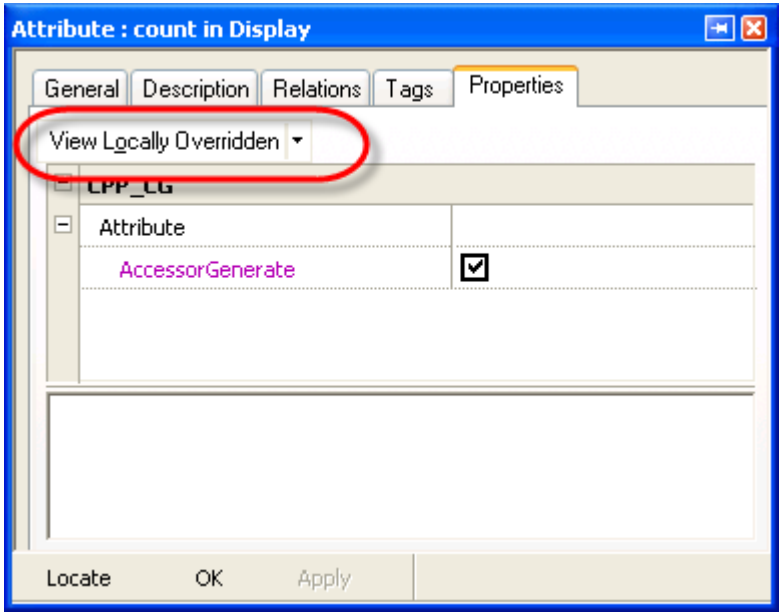
Overridden properties

- 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.

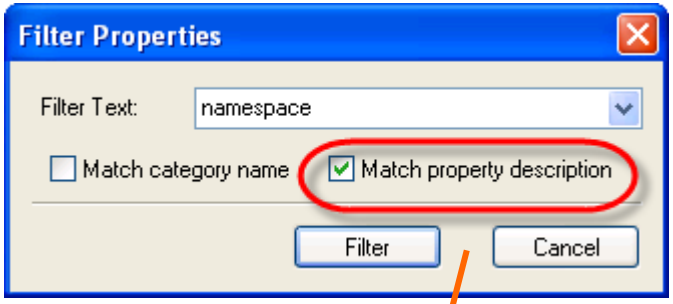


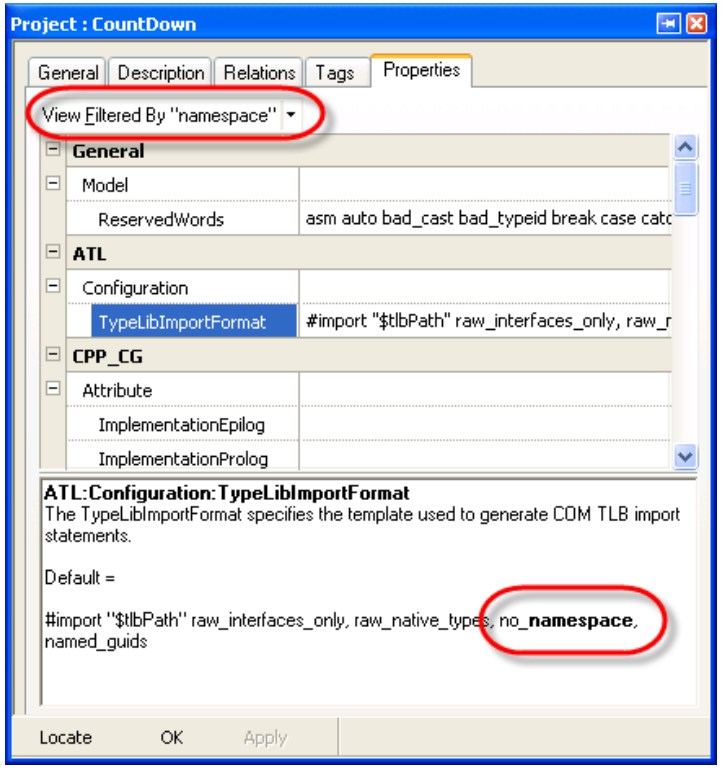

```
int Display::getCount() const {
    return count;
}
```

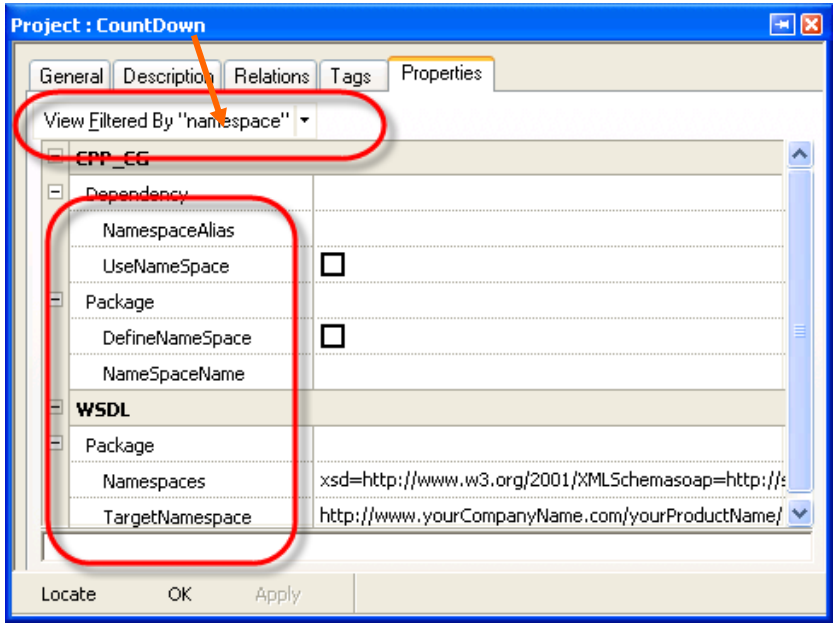
- Generate code and check that there is just an accessor for the attribute *count*.

Property filter

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

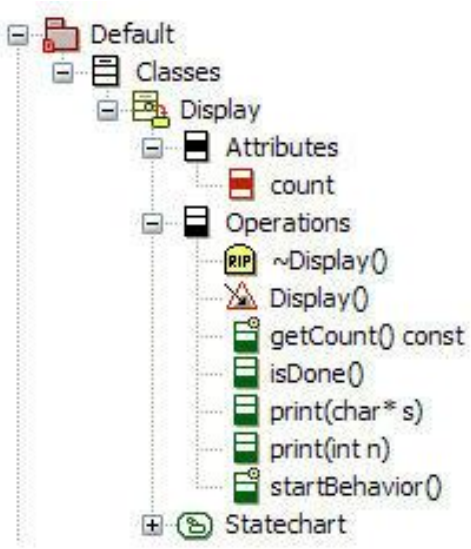


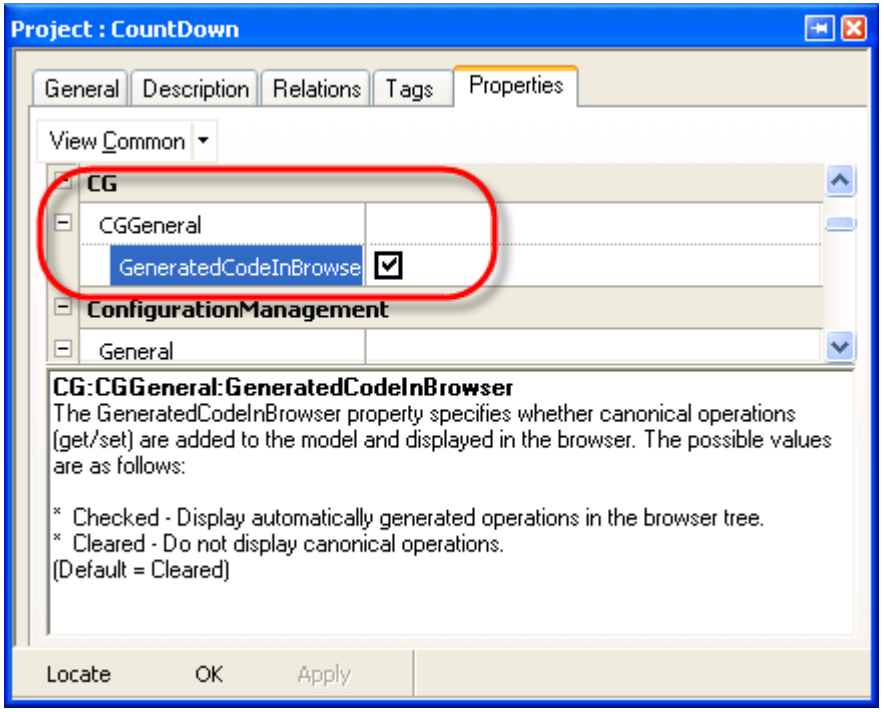


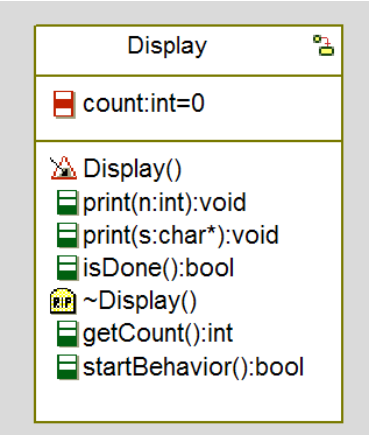


Extended exercise

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

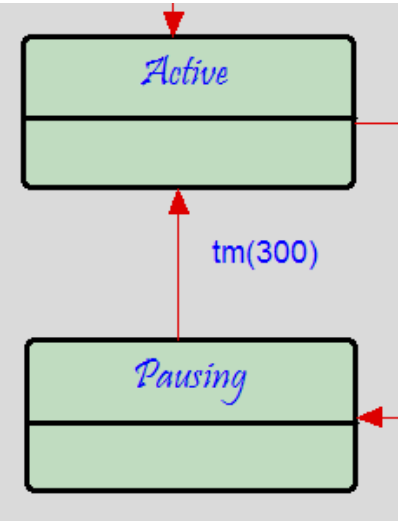


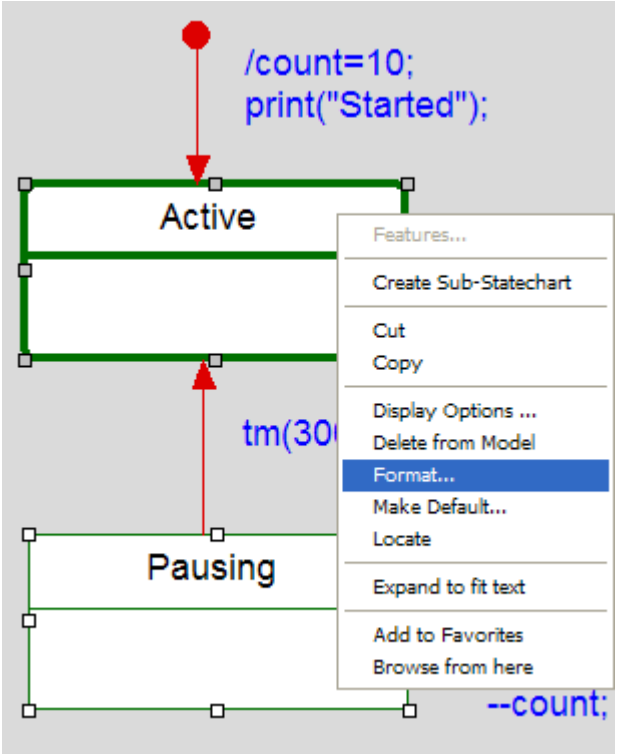


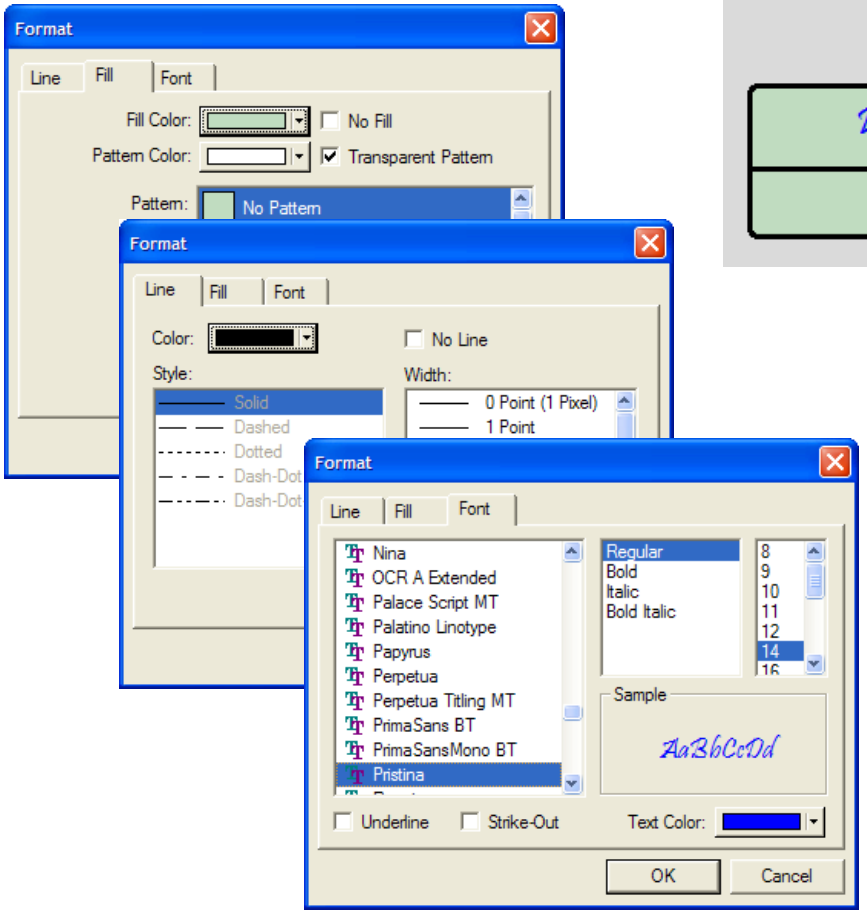


Formatting individual items

- 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

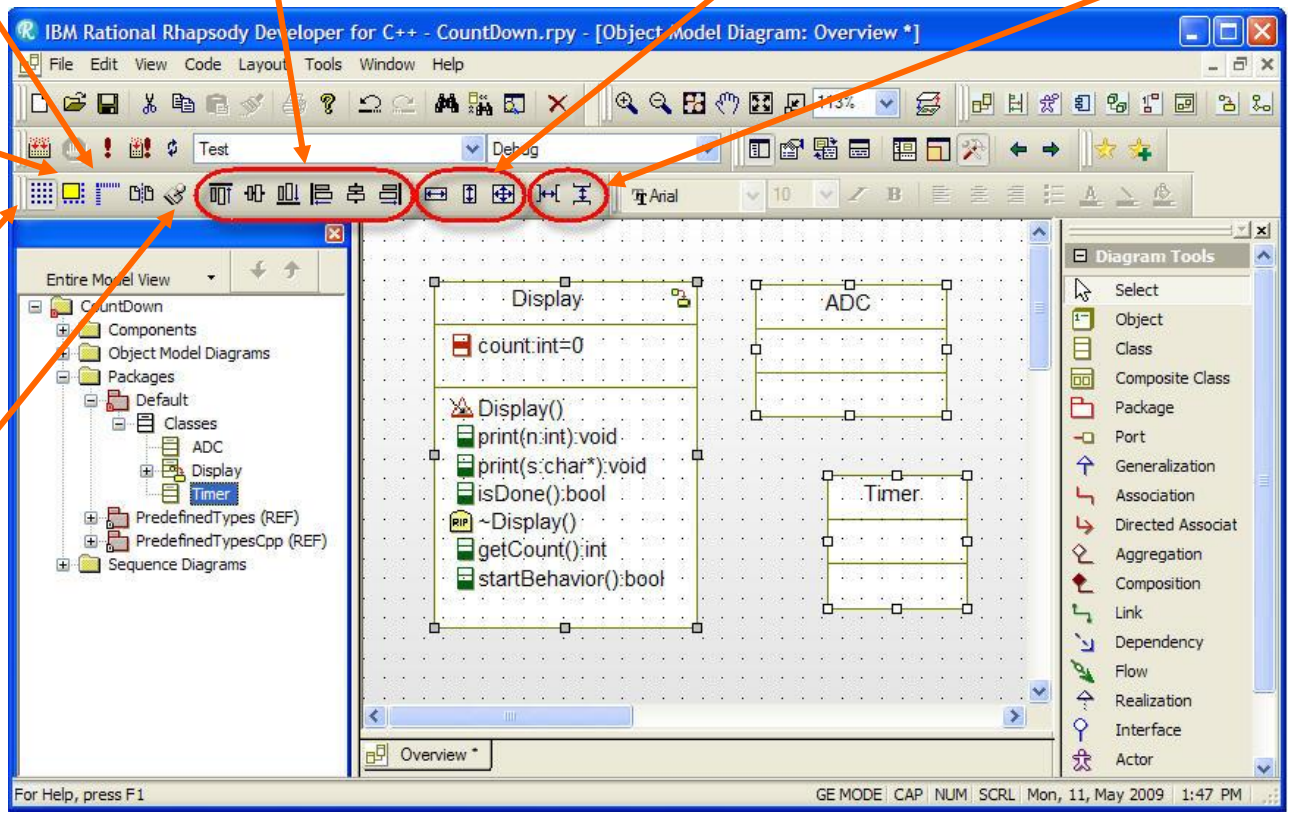
Make item same size

Give items same spacing size

Snap to grid

Turn grid on/off

Stamp mode - use when drawing same items repeatedly

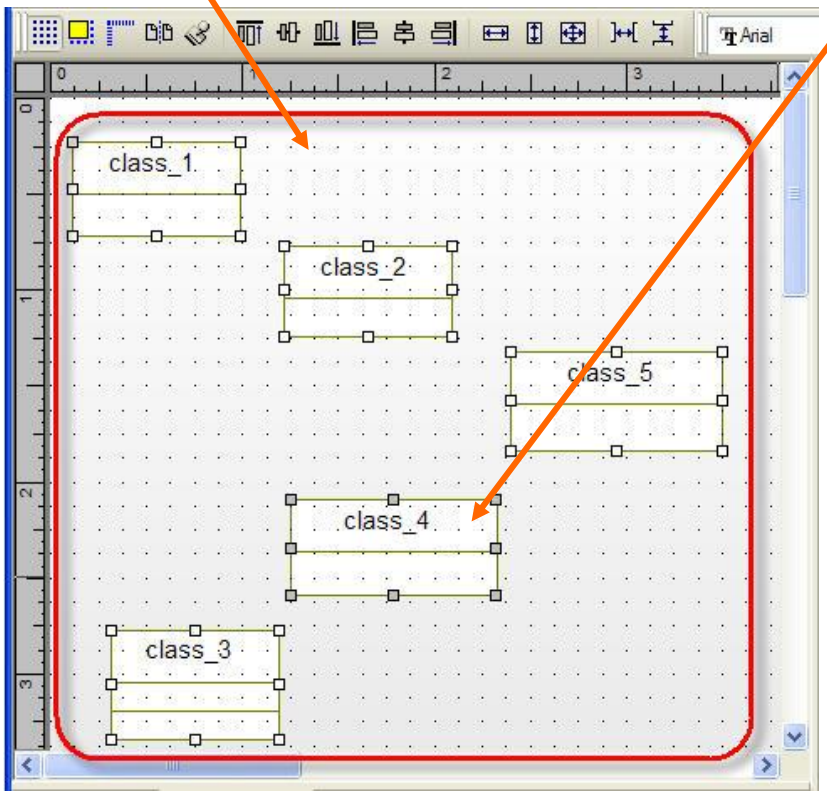


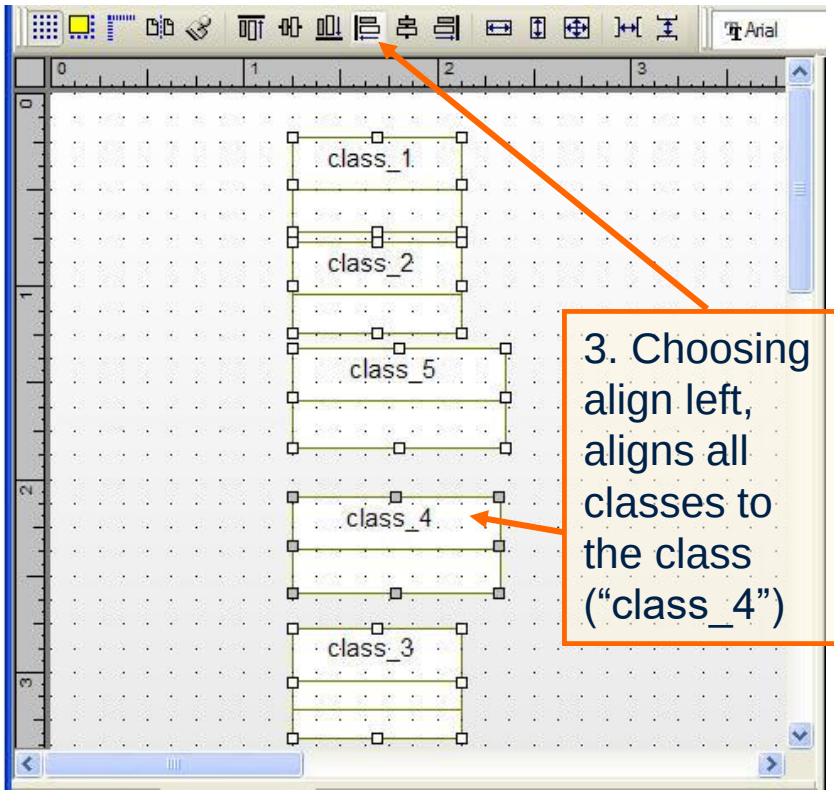
Aligning items to common edge

- 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

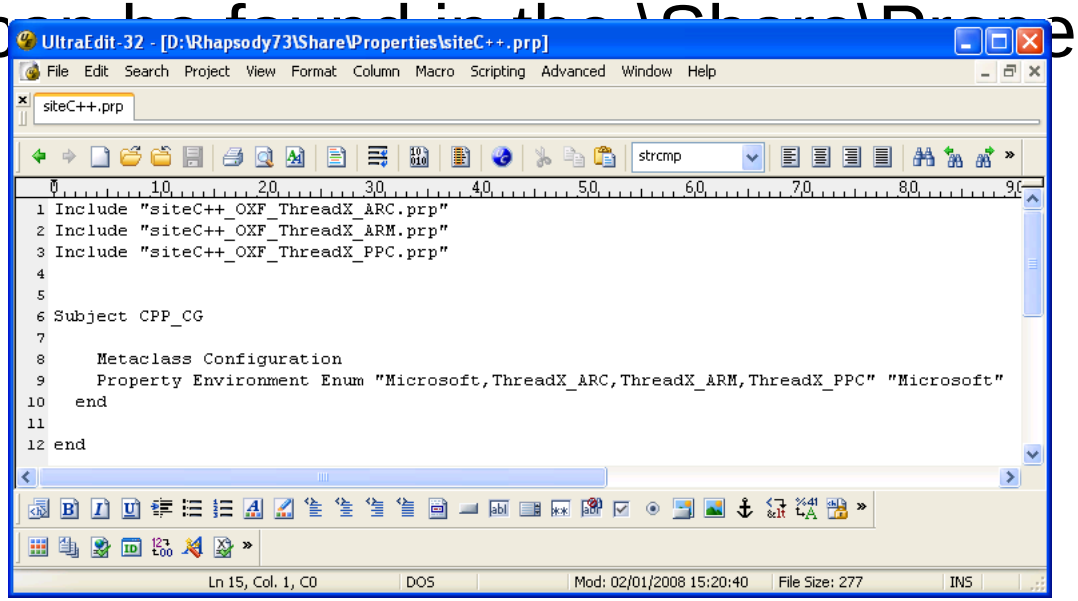




3. Choosing align left, aligns all classes to the class (“class_4”)

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.



Exercise 2A: Count down with LCD Display

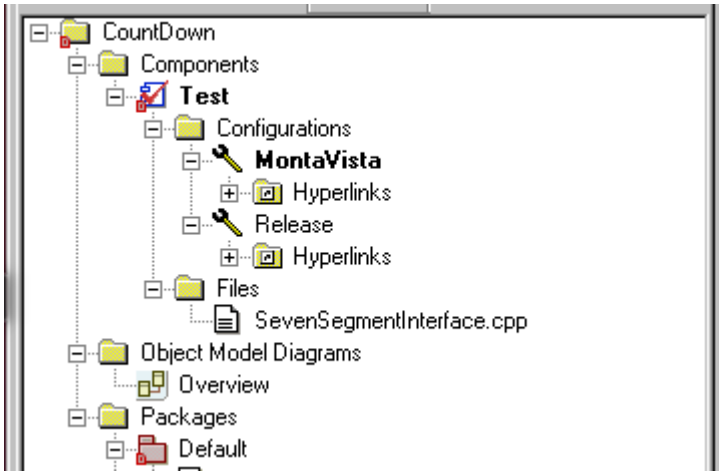
- First, try to run the Count down application on the embedded target as it is.

- Then, include the code accessing LCD display.

- You need to prepare the hardware interface code. Then, combine it with the model code.
- Model code + legacy code

New Configuration

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



Open Feature Window for MontaVista Configuration

- Environment: MontaVista

Configuration : MontaVista in Test

General | Description | Initialization | Settings | Checks | Relations | Tags | Properties

Directory: /home/control/work/Rhapsody/CountDown/Test/MontaVista ... ☑ Use Default

Libraries: ...

Additional Sources: ...

Standard Headers: ...

Include Path: ...

Instrumentation

Instrumentation Mode: None Advanced ...

Webify

☐ Web Enabling Advanced ...

Time Model: ◉ Real ○ Simulated

Statechart Implementation: ○ Reusable ◉ Flat

Environment Settings

Environment: MontaVista Default

Build Set: Debug

Compiler Switches: -I. -I$OMDefaultSpecificationDirectory -I$(OMROOT) -I$(OMROOT)/LangCpp -I$(OMROOT)/LangCpp/oxf $(INST_FLAGS) $(INCLUDE_PATH) $(INST_INCLUDES) -DUSE_IOSTREAM $OMCPPCompileCommandSet -c ...

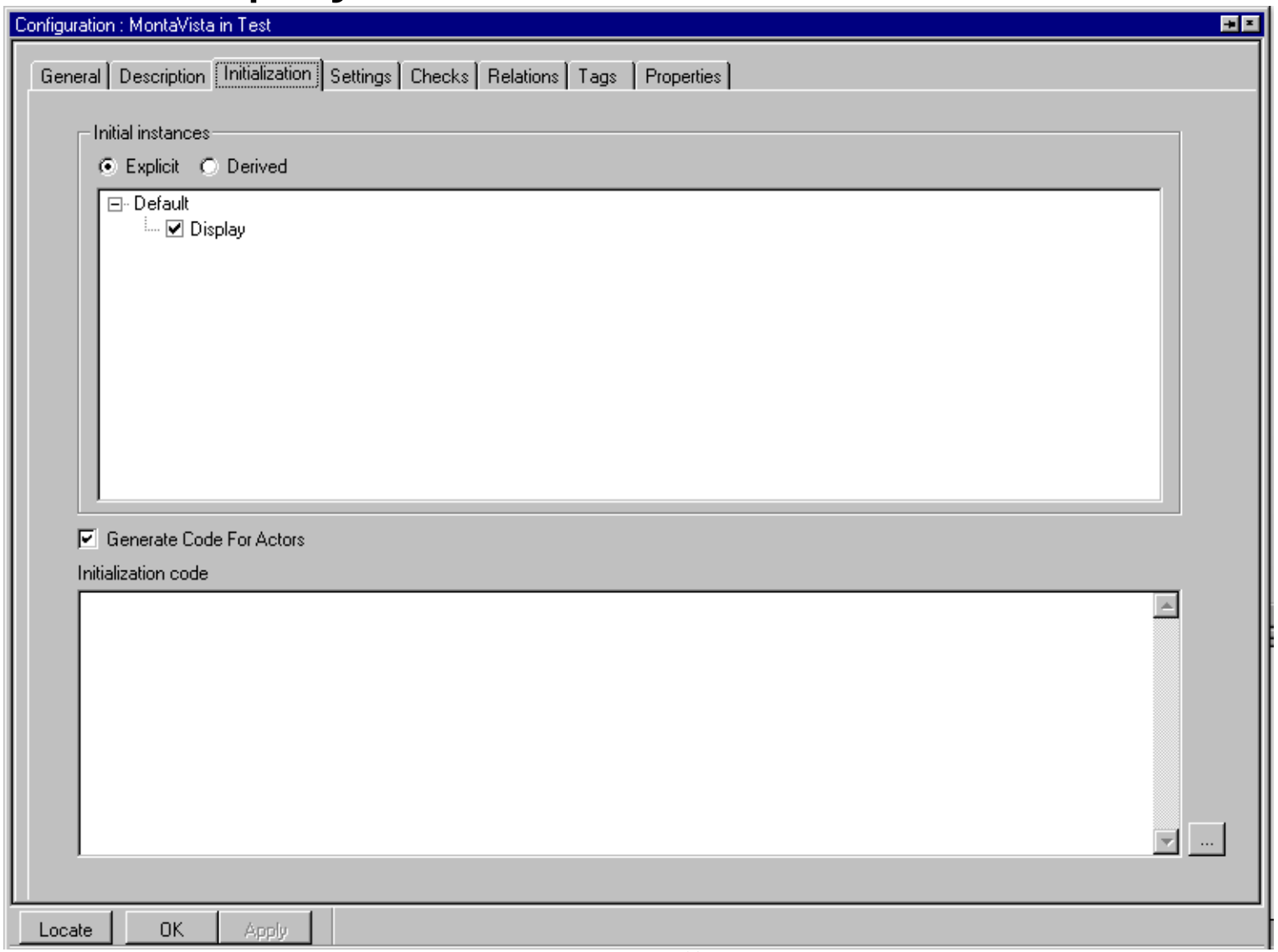
Link Switches: $OMLinkCommandSet ...

☐ Include Requirements as Comments in Code

Locate | OK | Apply

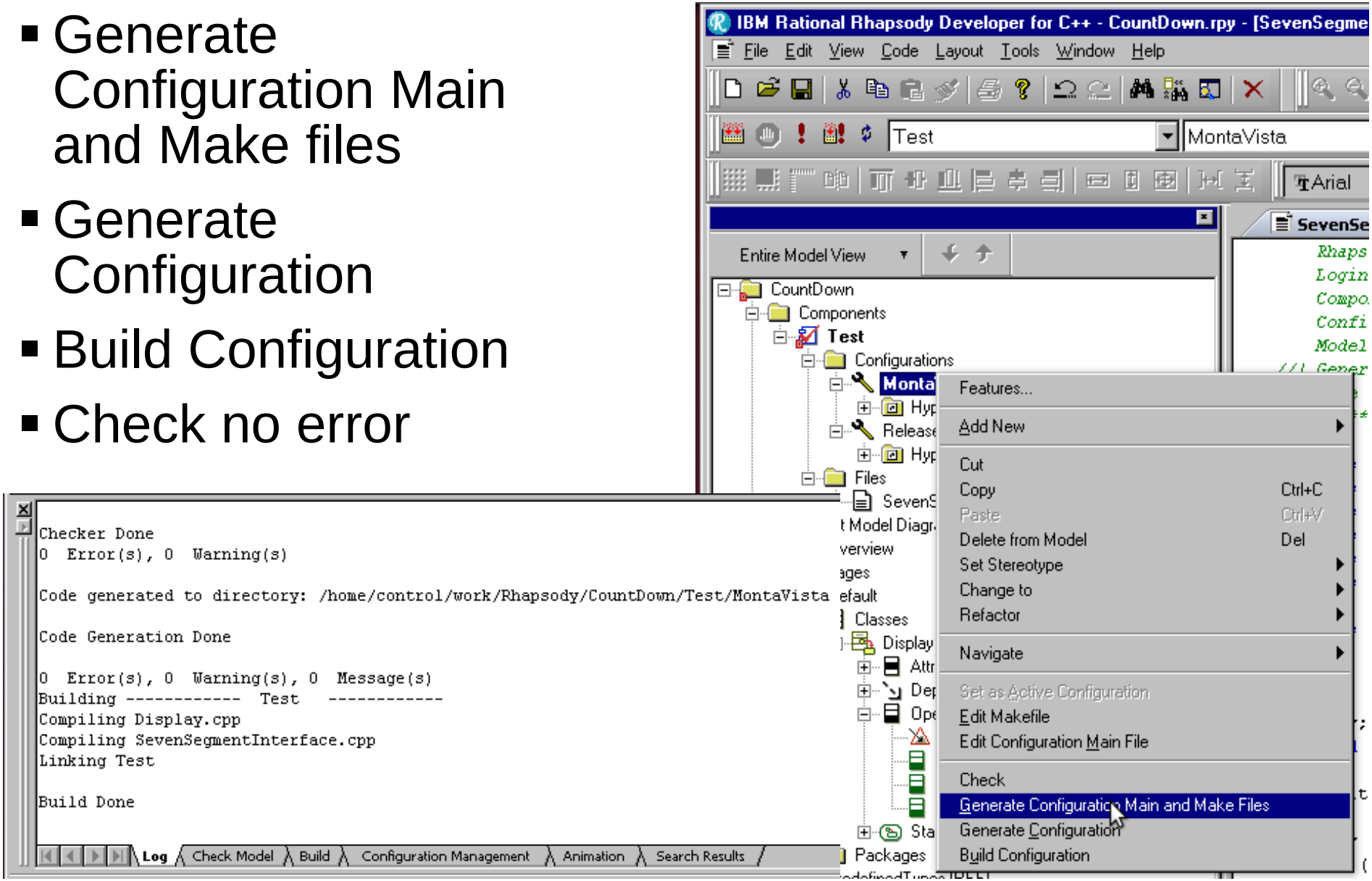
Initial instances

- Check Display

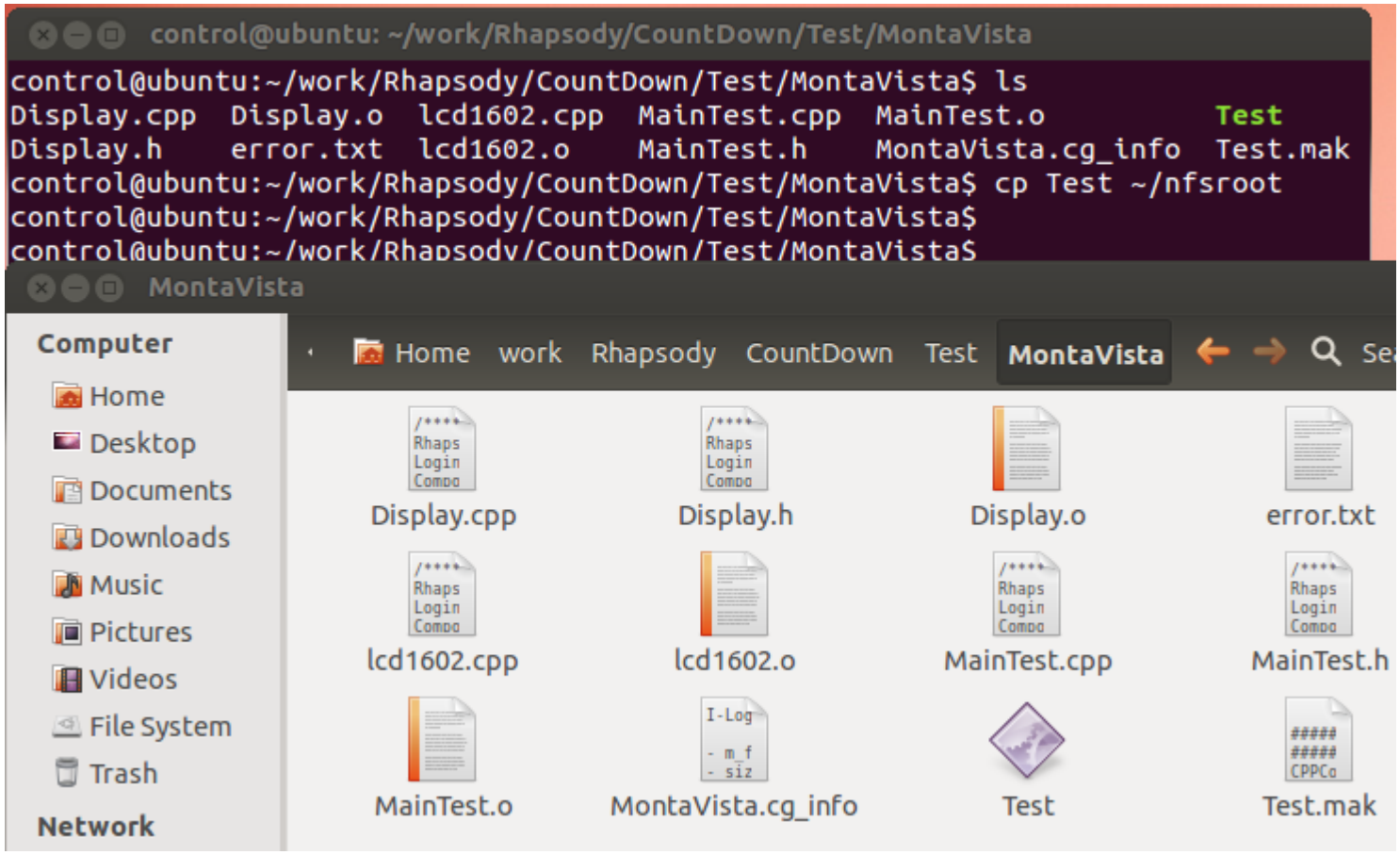


Build

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

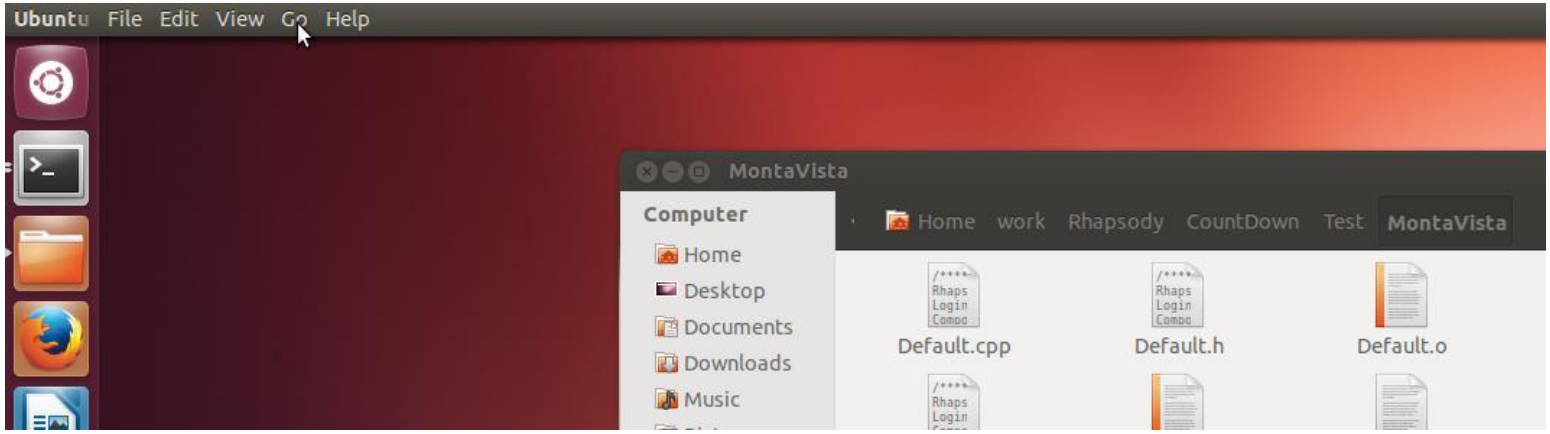


- Copy executable file to ~/nfsroot folder

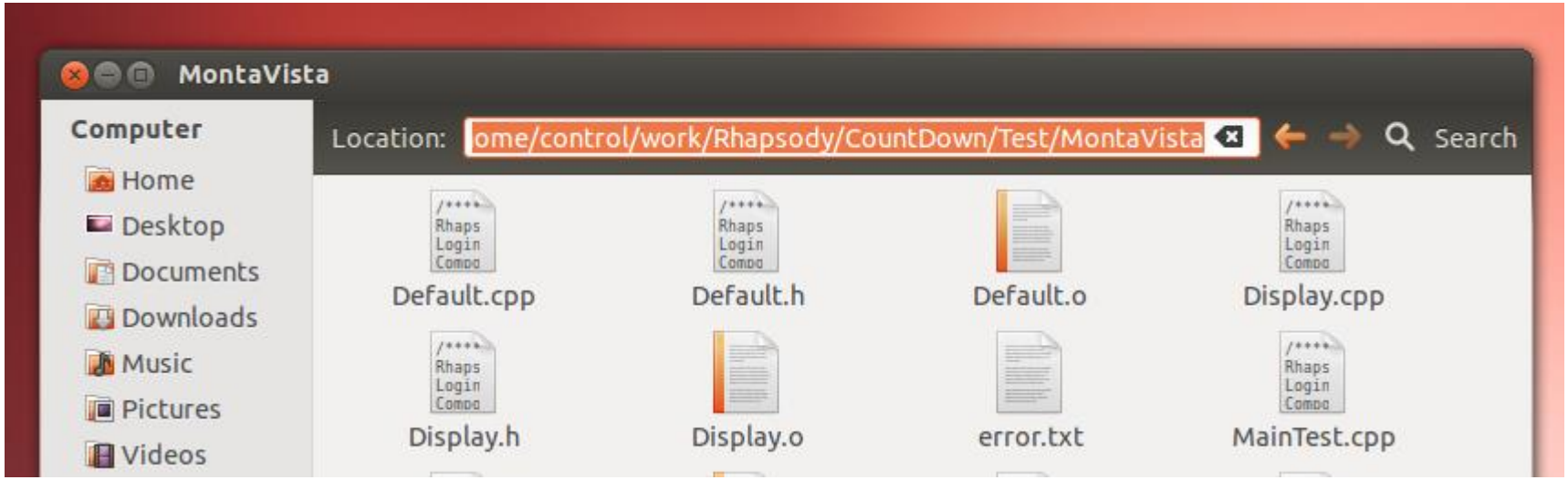


cd command with a long path name

- Move the mouse cursor to the menu bar

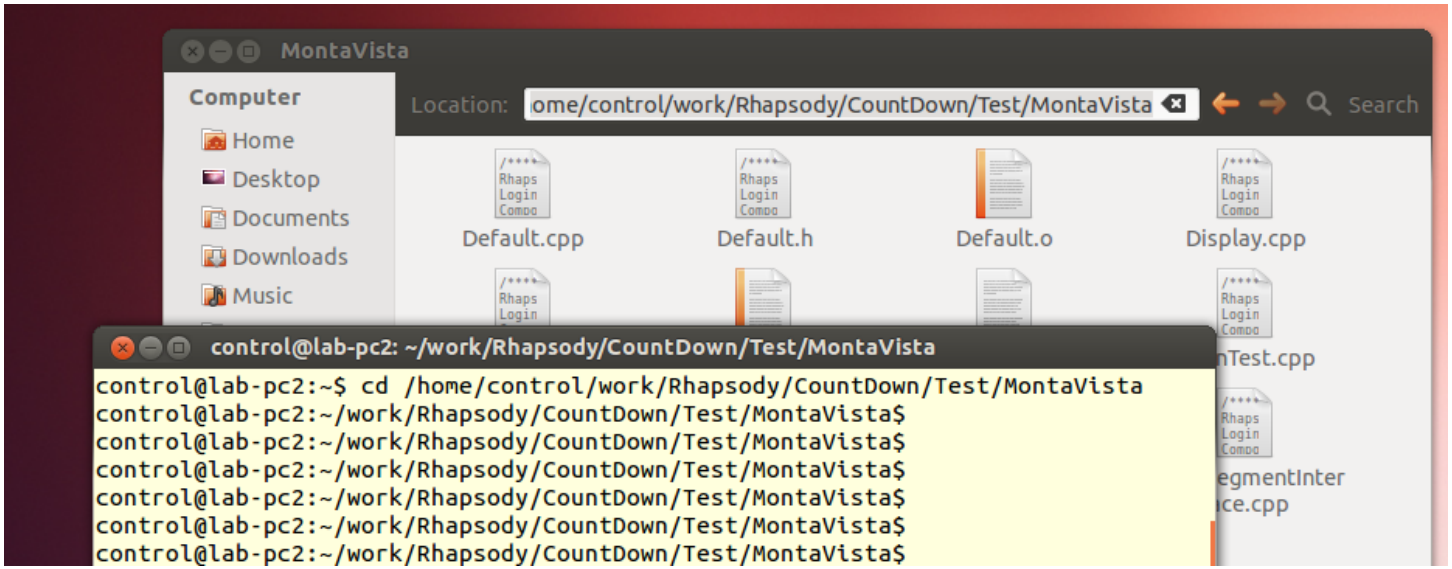


- Select Location from Go menu



cd command with a long path name

- Copy and paste to the Terminal window



Run on the target

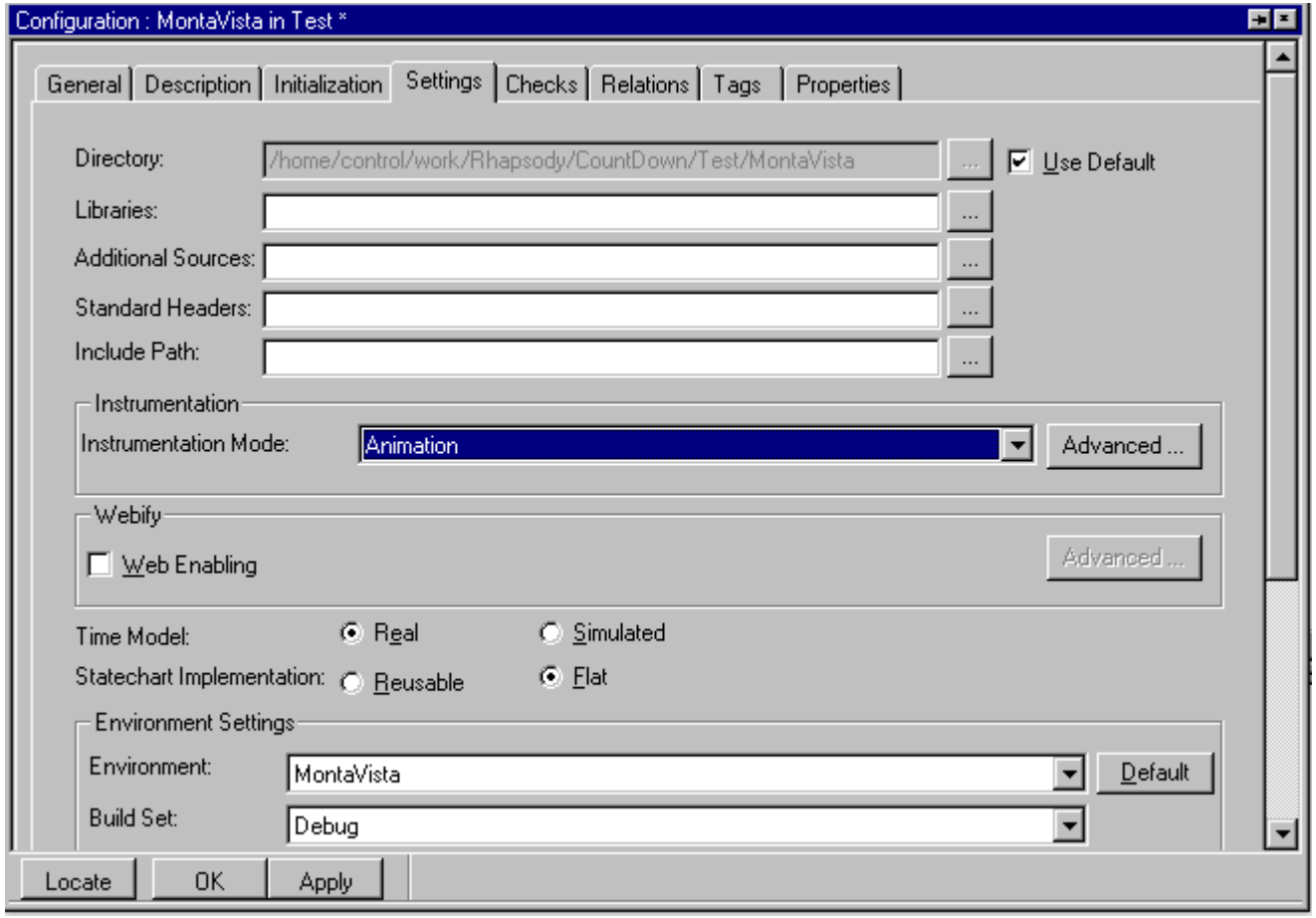
SmarTTY - 192.168.0.121

File Edit View SCP Tools Help

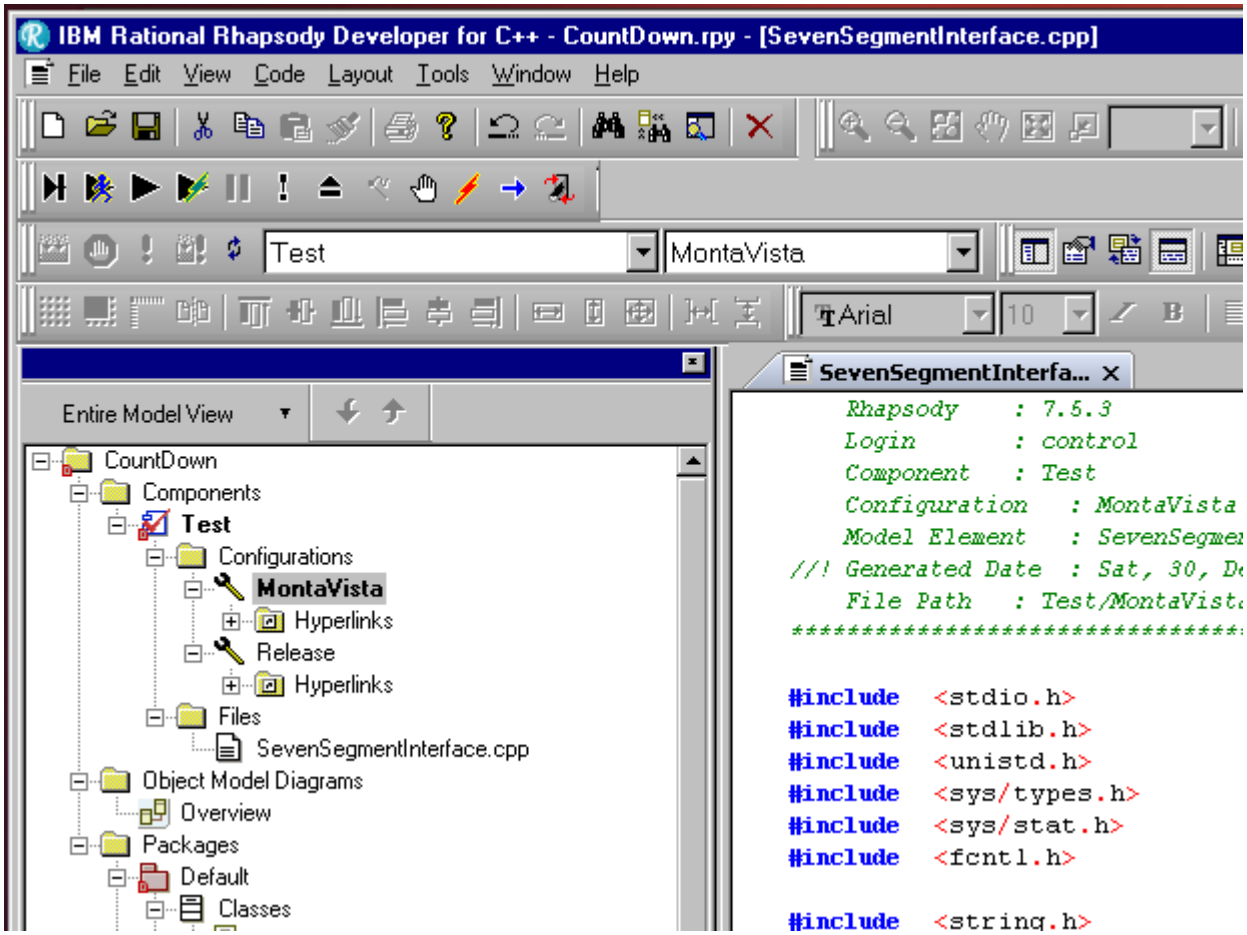
```
pi@raspberrypi:~/mnt $ ls
Test
pi@raspberrypi:~/mnt $ ./Test
Constructed
Started
Count = 10
Count = 9
Count = 8
Count = 7
Count = 6
Count = 5
Count = 4
Count = 3
Count = 2
Count = 1
Count = 0
Done
pi@raspberrypi:~/mnt $
```

Run on the target with animation

- Change to Animation, build, and copy



Run on the target with animation



Copy and modify lcd1602_test.c(1)

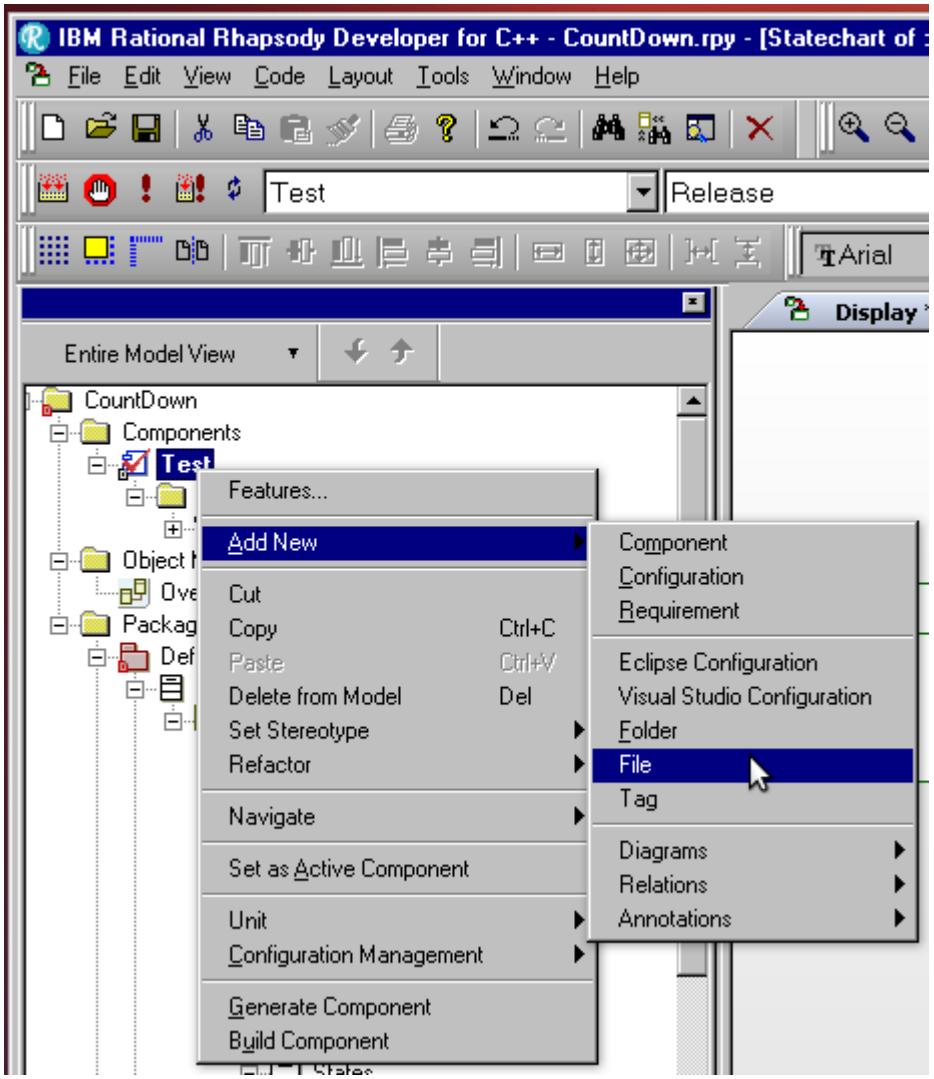
```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
int fd;

void initLCD(void)
{
        fd=open("/dev/lcd1602",O_RDWR);
        if (fd < 0) {
                printf("Device open error : %s\n","/dev/lcd1602");
                exit(1);
        }
}
```

Copy and modify lcd1602_test.c(2)

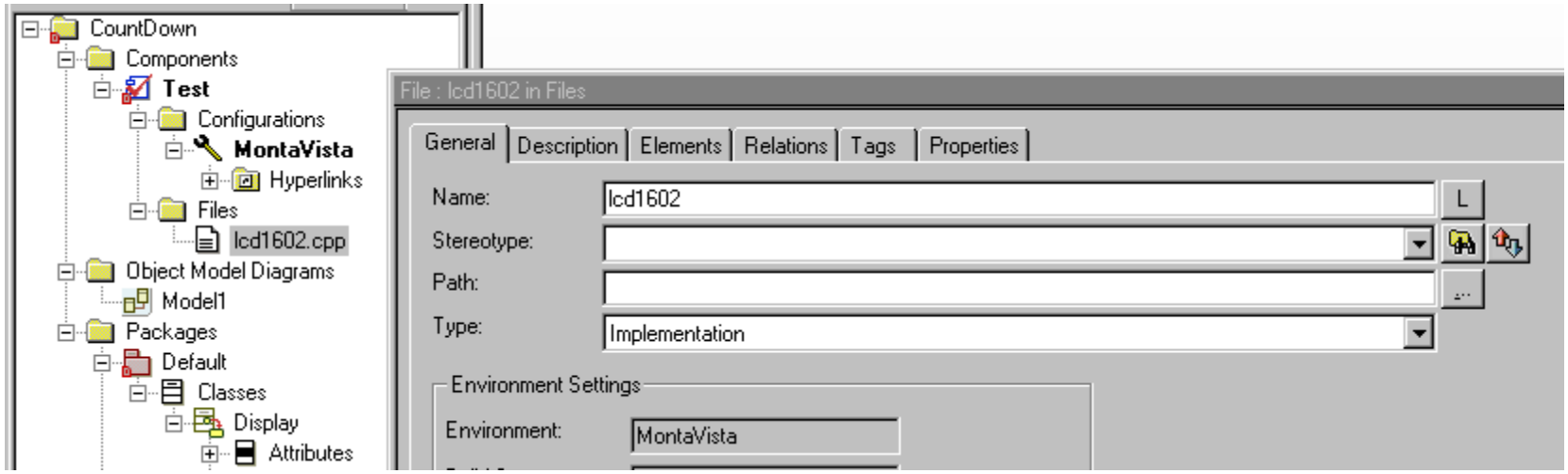
```
void displayLCD(int count)
{
        char wbuf[30];

        wbuf[0] = count / 10 + 0x30;
        wbuf[1] = count % 10 + 0x30;
        for (int i=2;i<15;i++) wbuf[i]=' ';
        wbuf[15] = 0x0;
        write(fd, wbuf, strlen(wbuf));
}
```

Add Hardware Interface Code

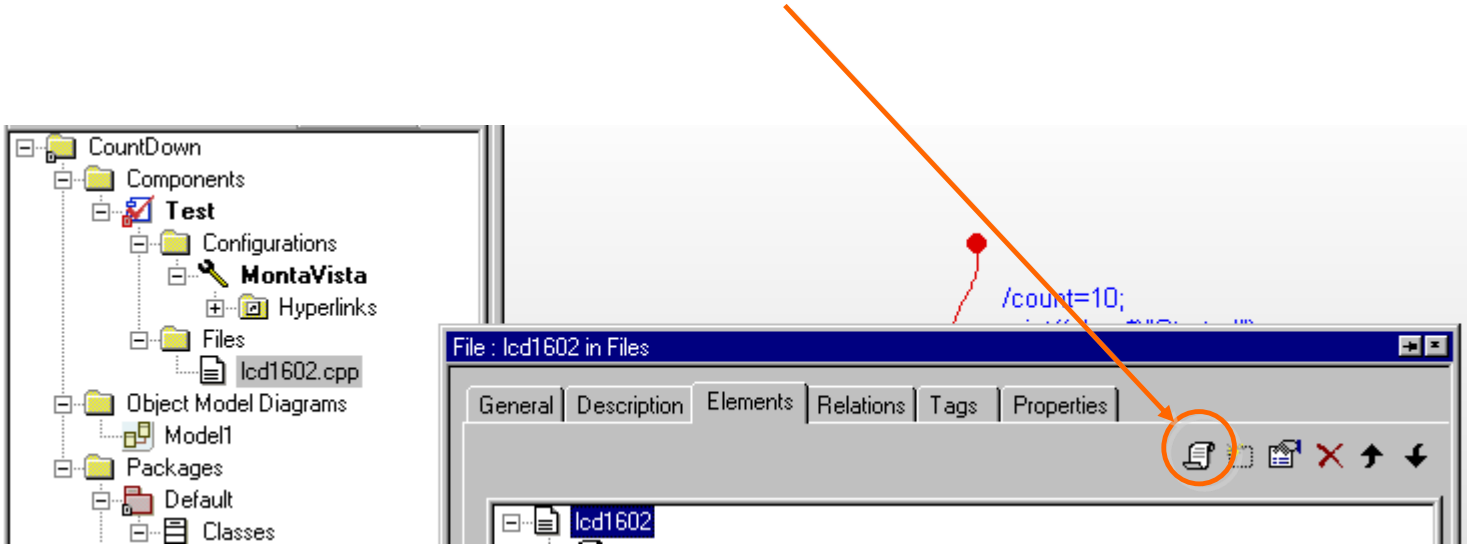
- Right Click Test Component and Select Add New “File”



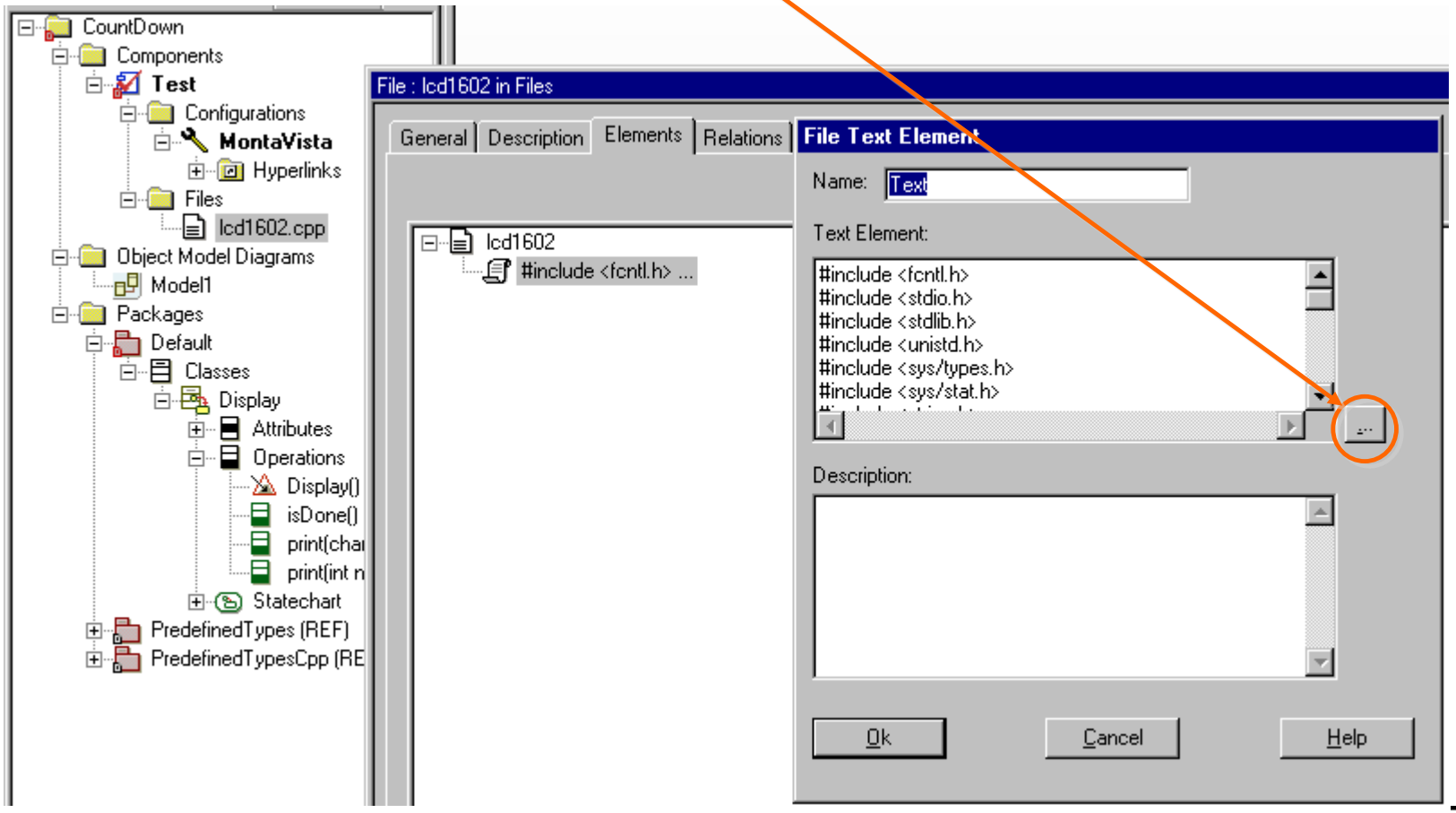
- Double click a new file and change Name to “lcd1602” and Type to Implementation



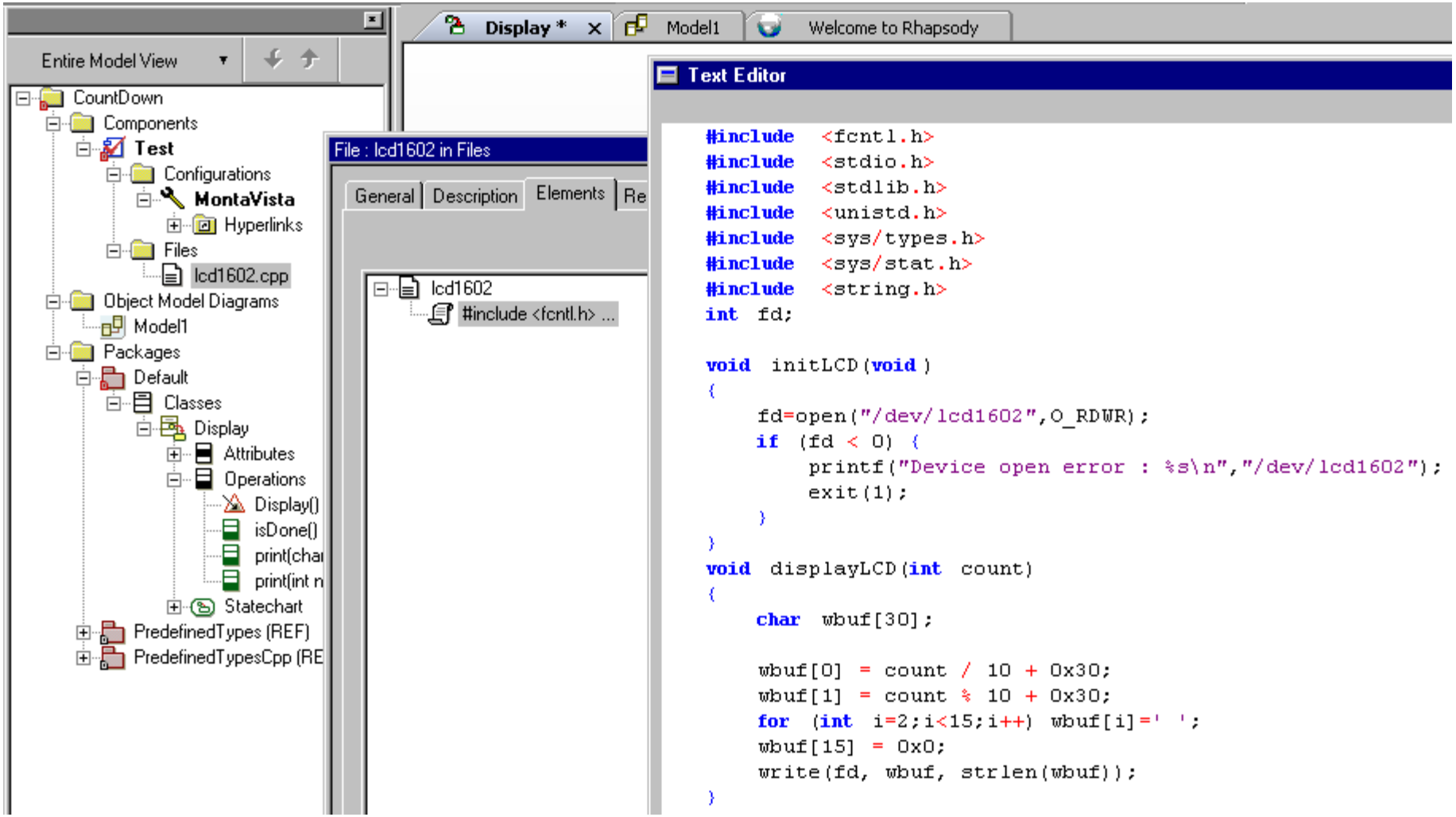
- Double click lcd1602.cpp and select the tab “Elements”.
- Then click New Text Element button



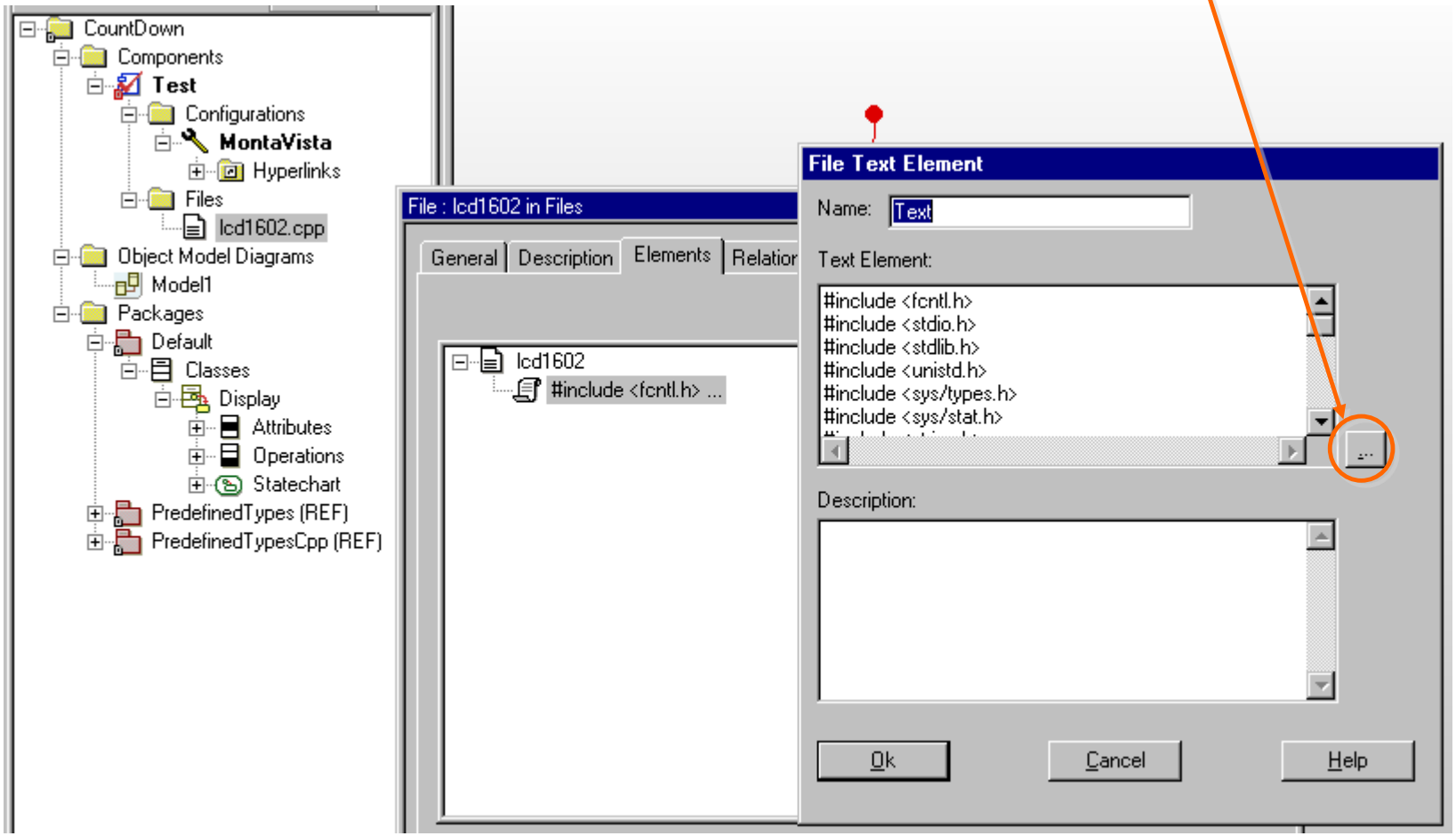
- Open the text editor



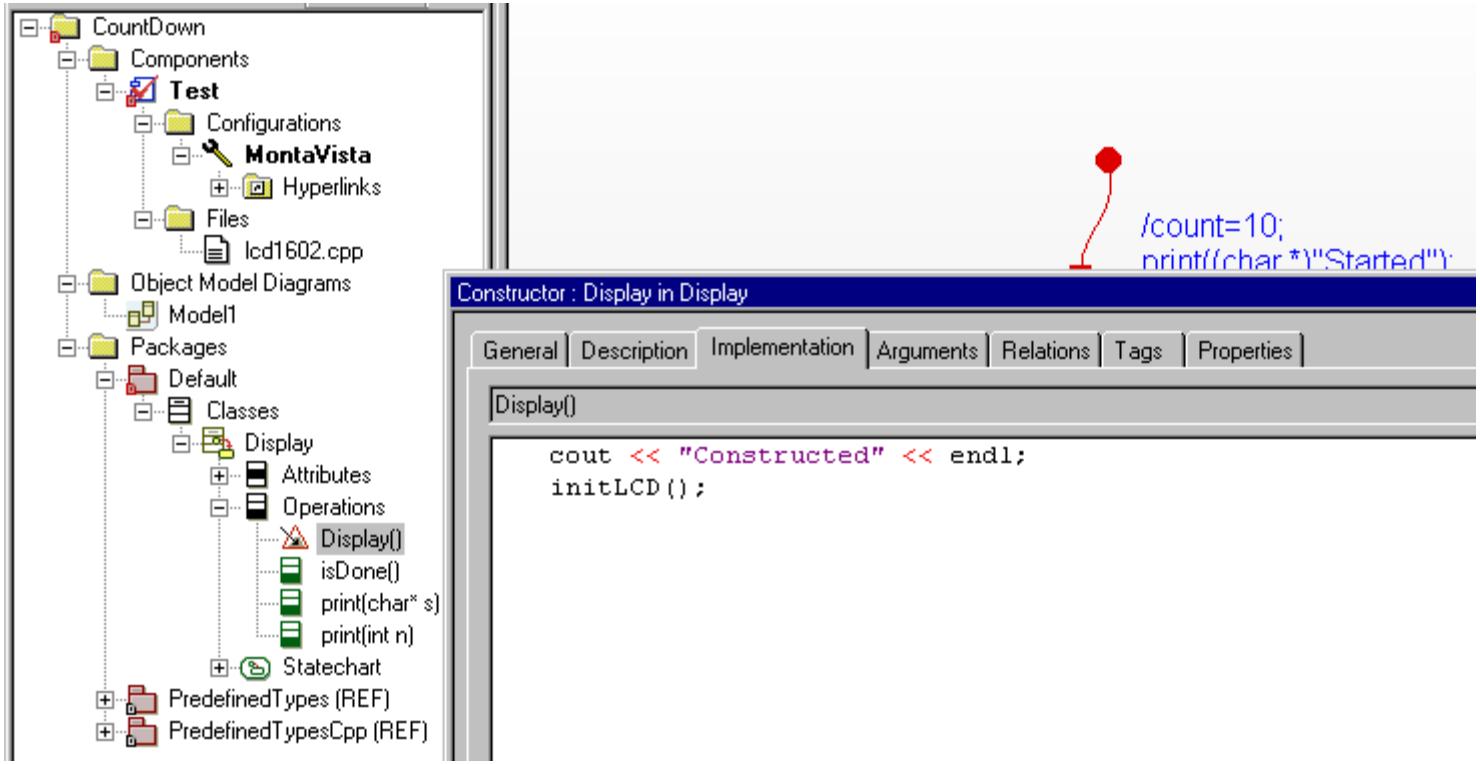
- Paste the code and click OK



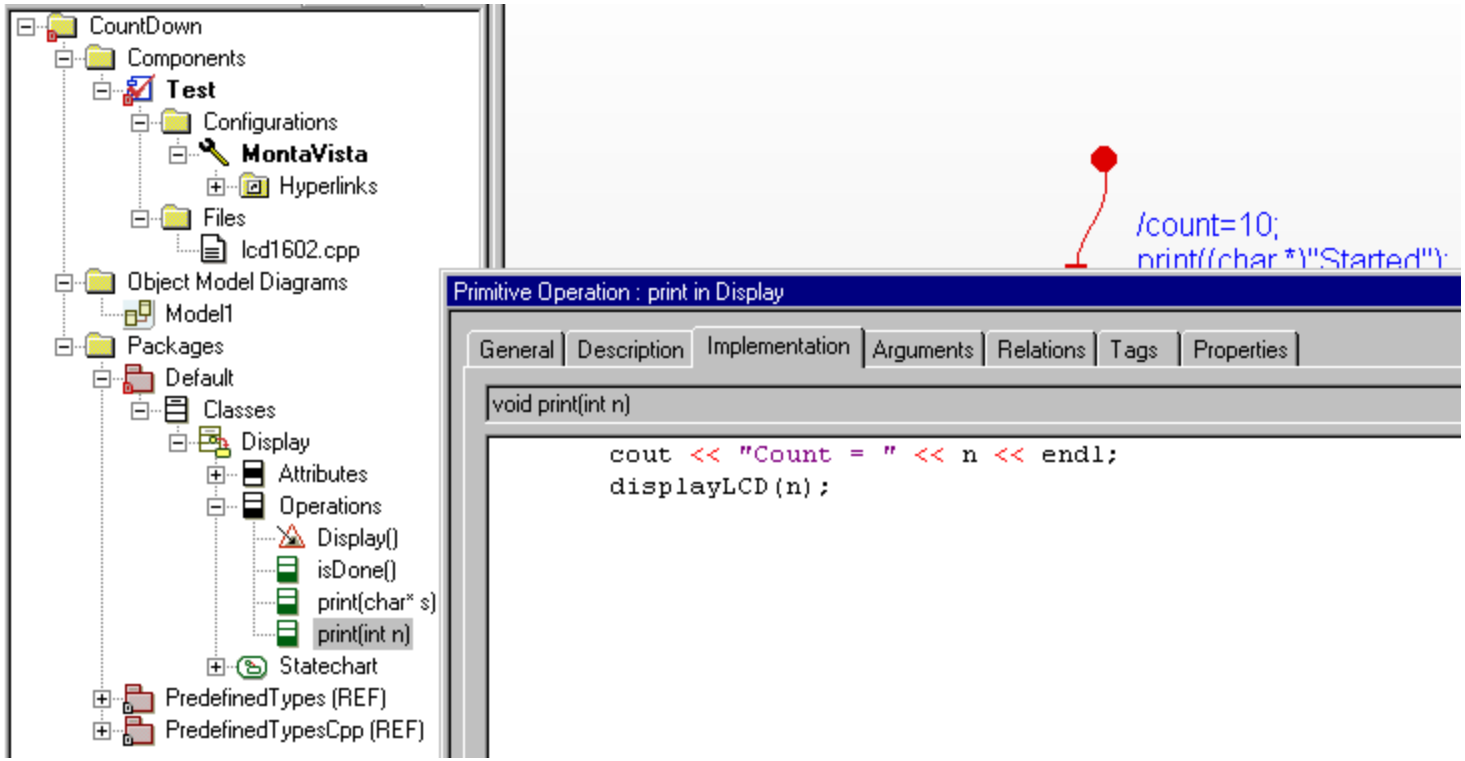
- 이미 입력된 파일을 수정할 경우:Open the text editor



Open the driver

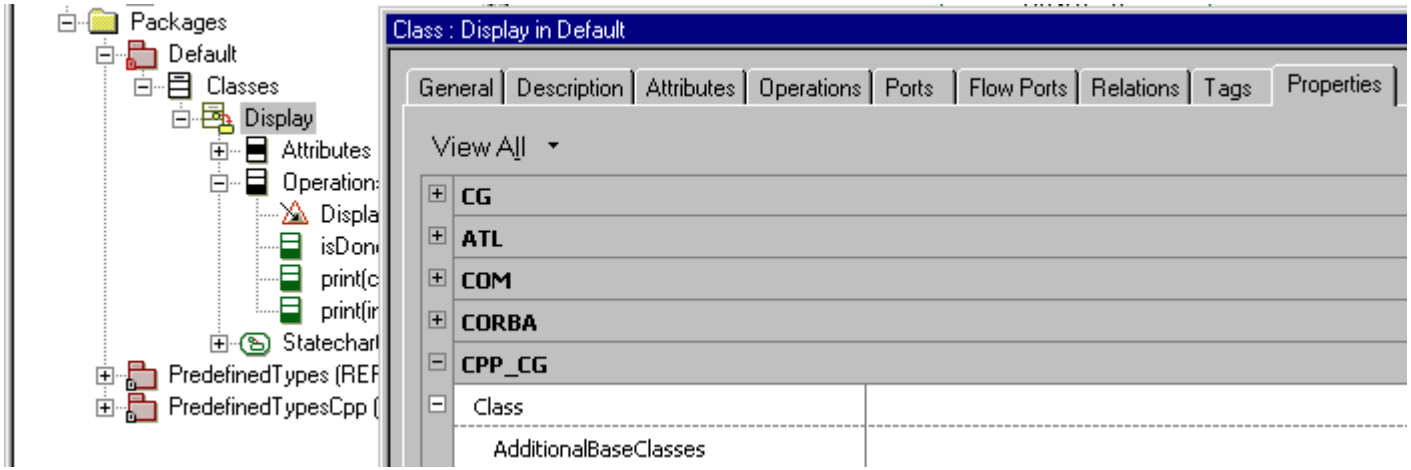


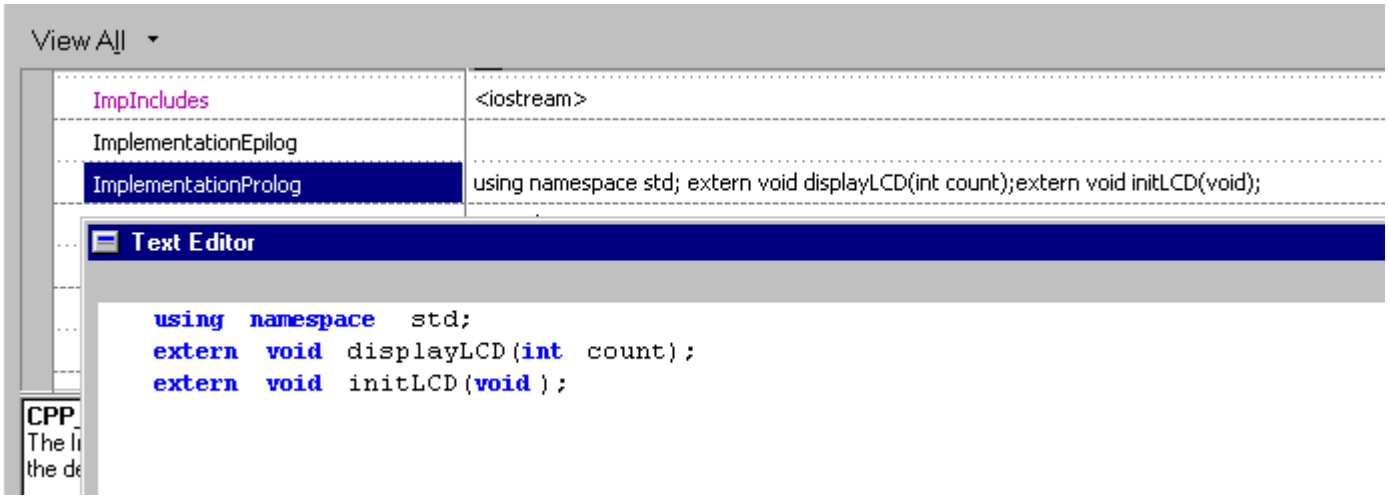
Call Interface Routine



Include extern (function prototype)

- Open Display class Feature window and select Properties





Load the driver and run on the target

SmarTTY - 192.168.0.121

File Edit View SCP Tools Help

File List

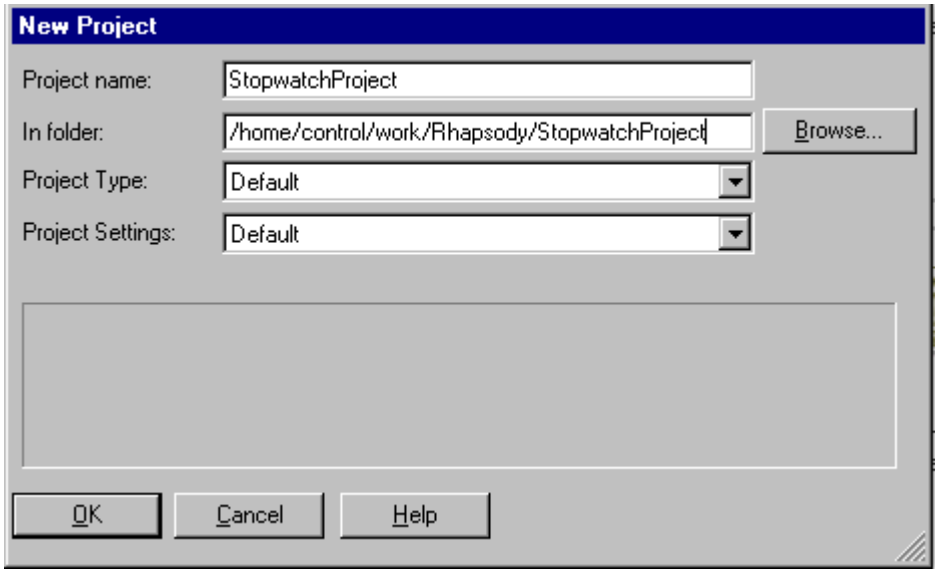
Filter:

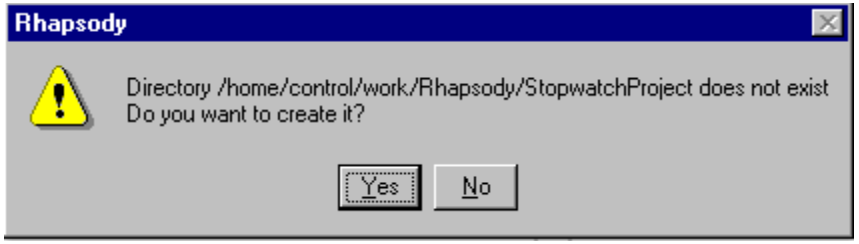
File name	Size
..	<dir>
Test	685K

```
pi@192.168.0.121:~/work/lcd1602driver$ ./load.sh
pi@192.168.0.121:~/work/lcd1602driver$
pi@192.168.0.121:~/work$
pi@192.168.0.121:~$
pi@192.168.0.121:~/mnt$ ./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
pi@192.168.0.121:~/mnt$
```

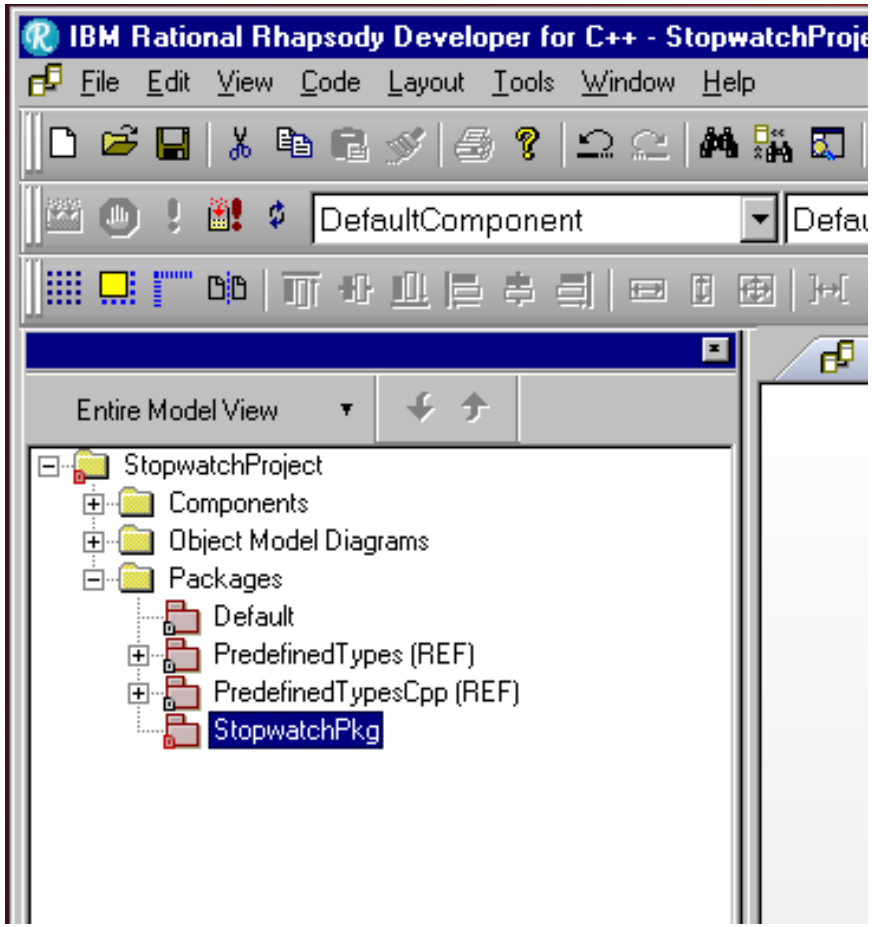
Exercise 3: Stopwatch Project

Create a new project

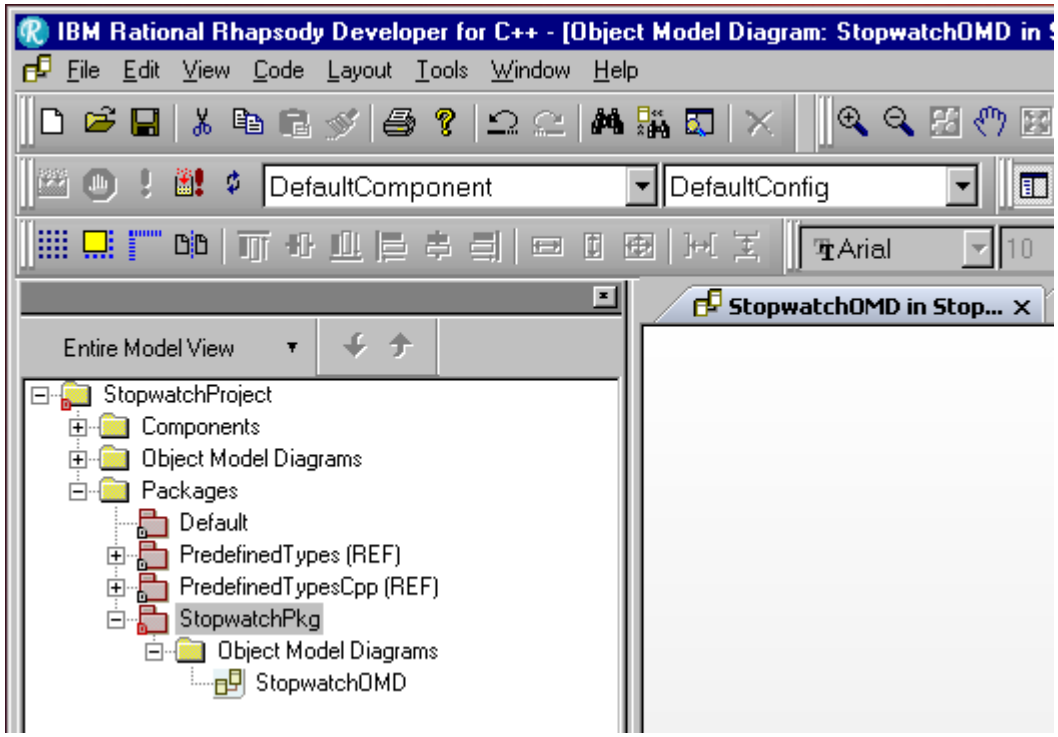




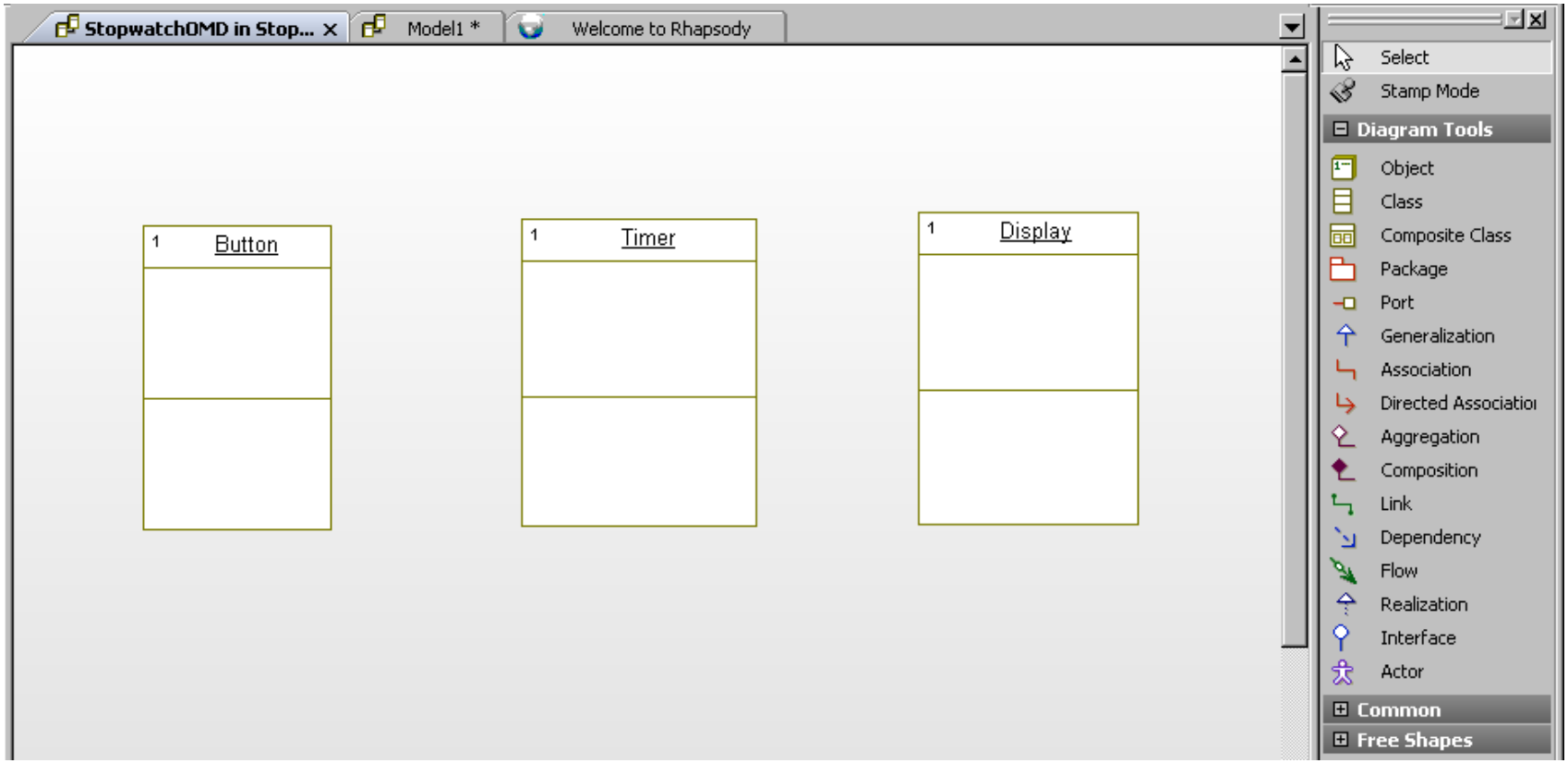
- Right click Packages and Add New Package
- Change the name to StopwatchPkg



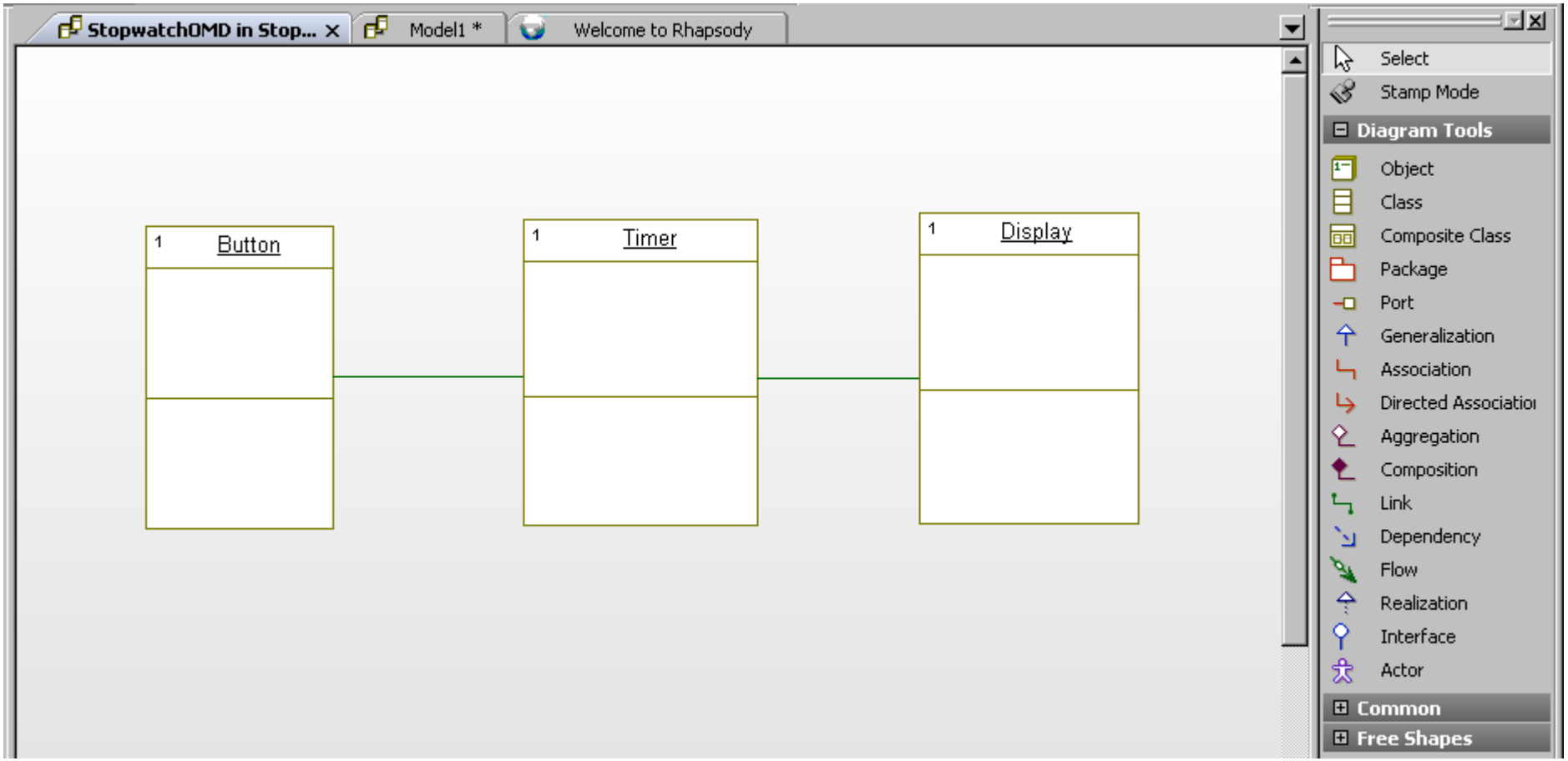
- Right click StopwatchPkg and Add New-Diagrams-Object Model Diagram
- Change the name to StopwatchOMD



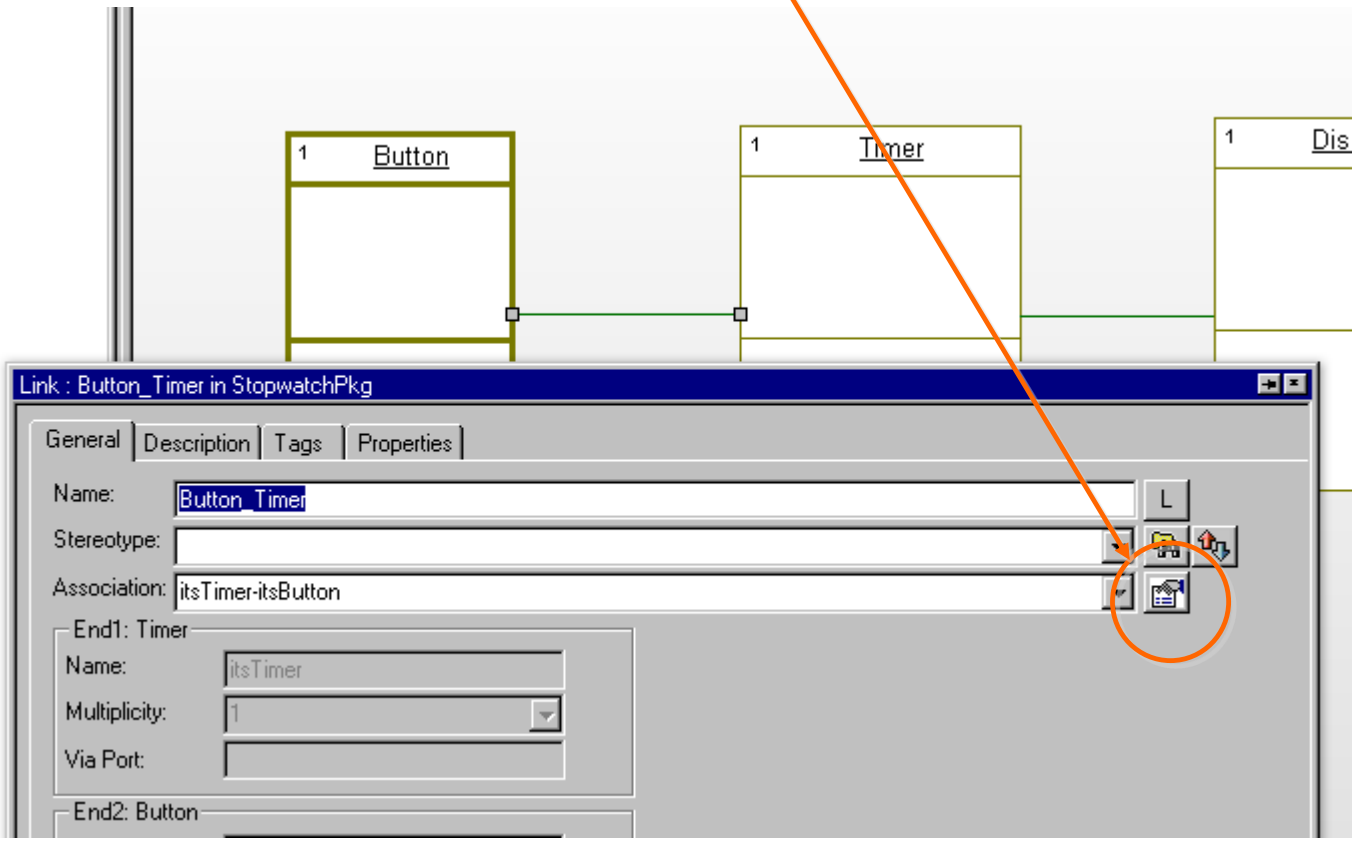
- Select Object in Diagram Tools
- Draw three objects, Button, Timer, Display



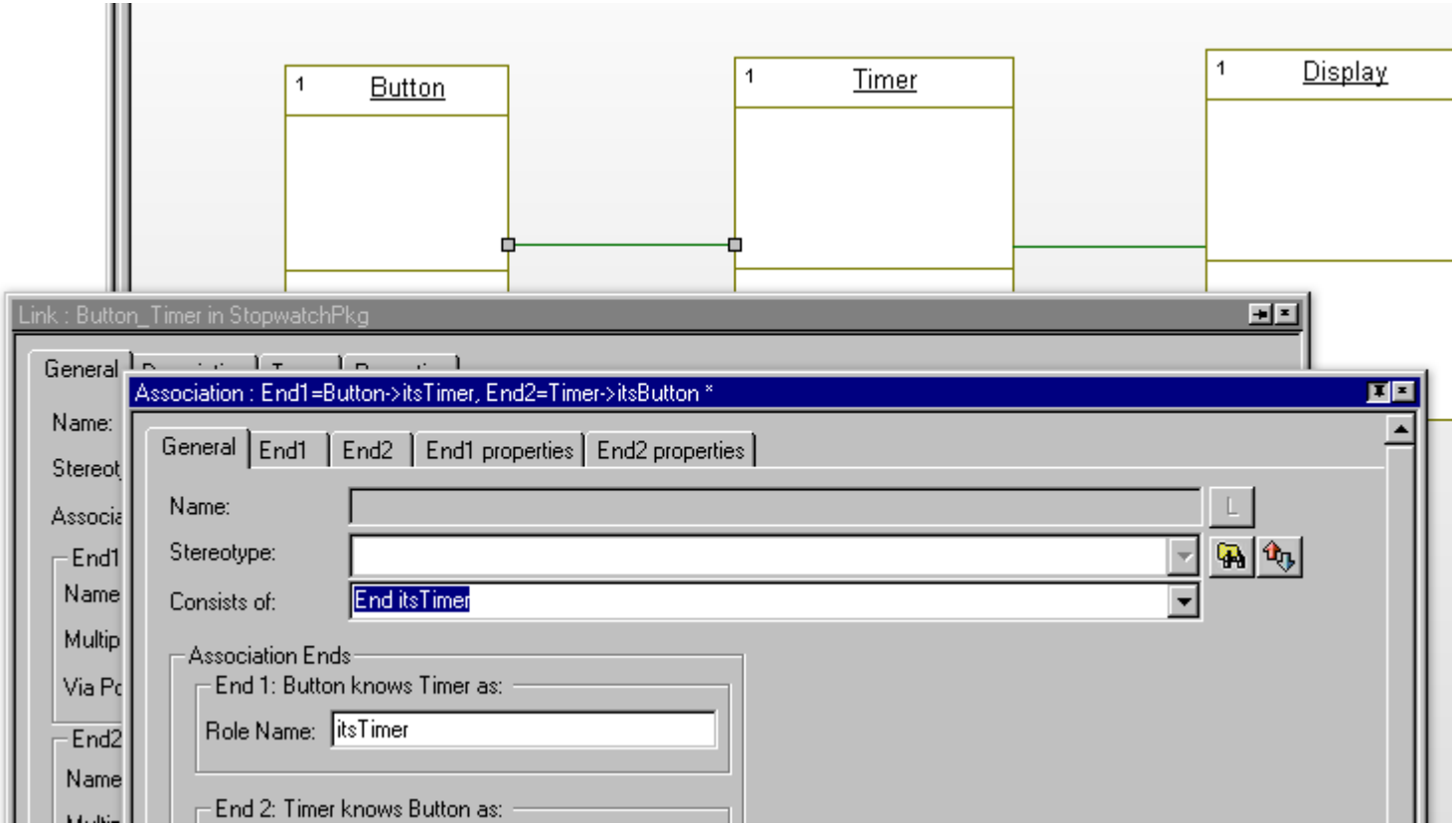
- Select Link in Diagram Tools and draw links



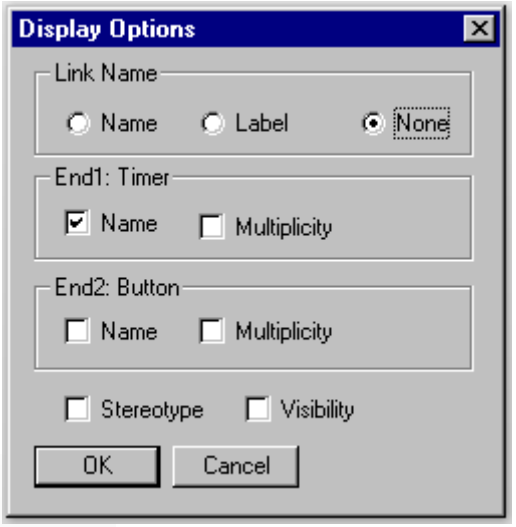
- Double click the link between Button and Timer
- Click Association change button

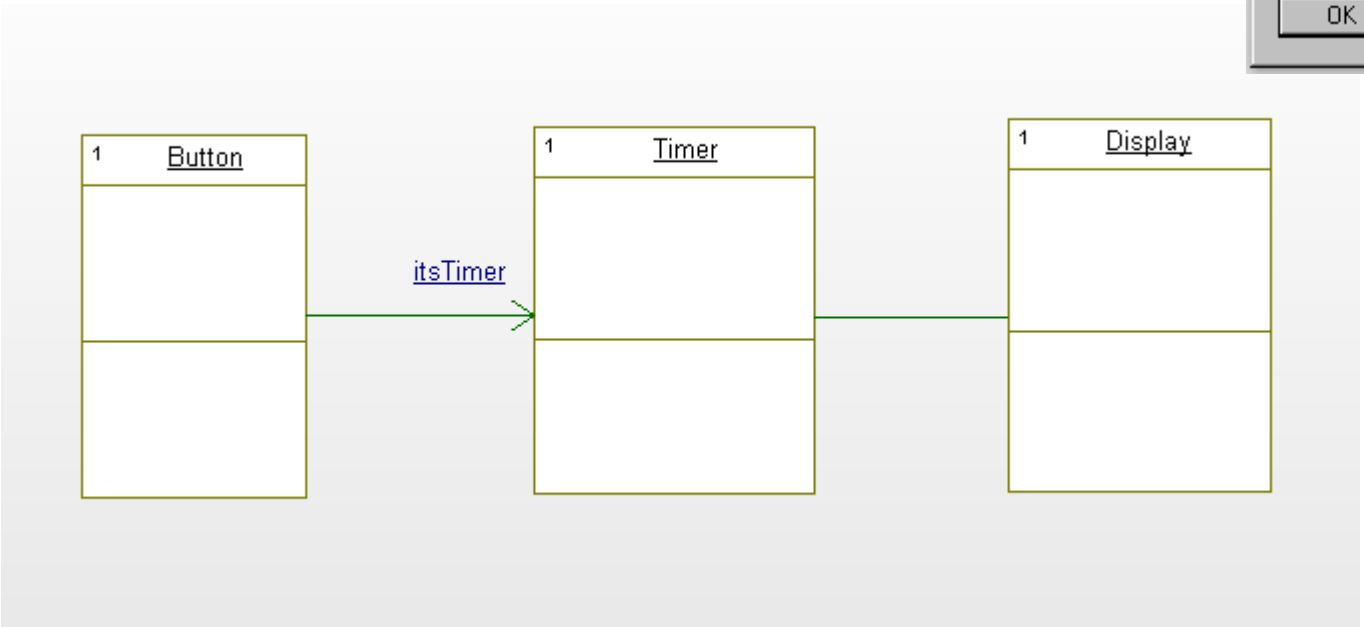


- Change Both Ends to End itsTimer

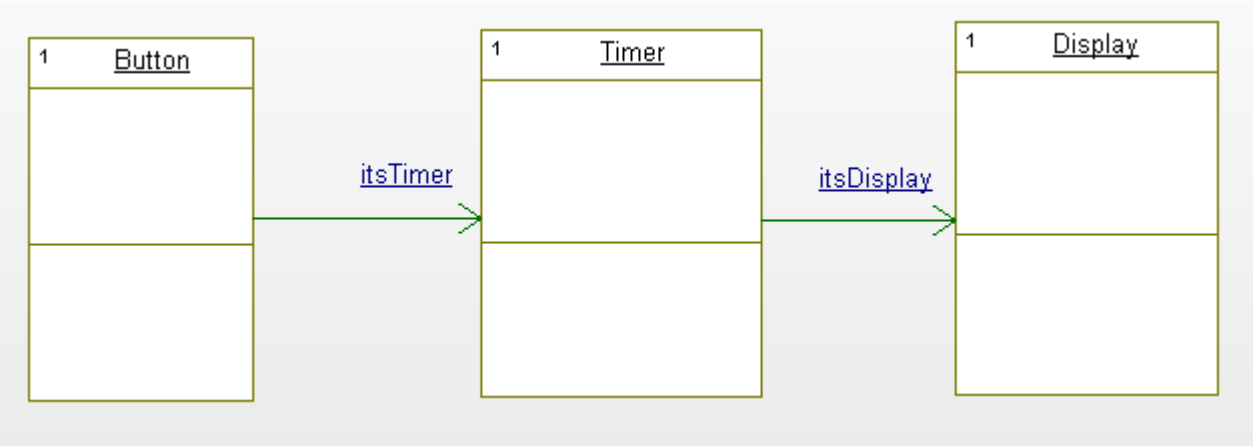


- Right click the link and change Display options

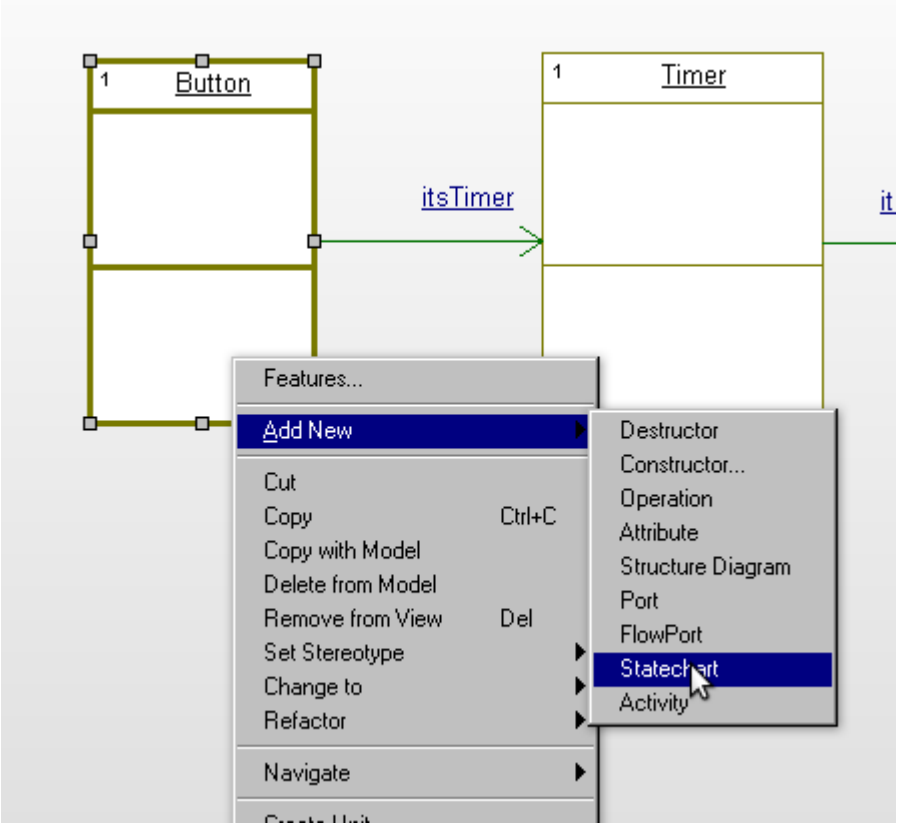




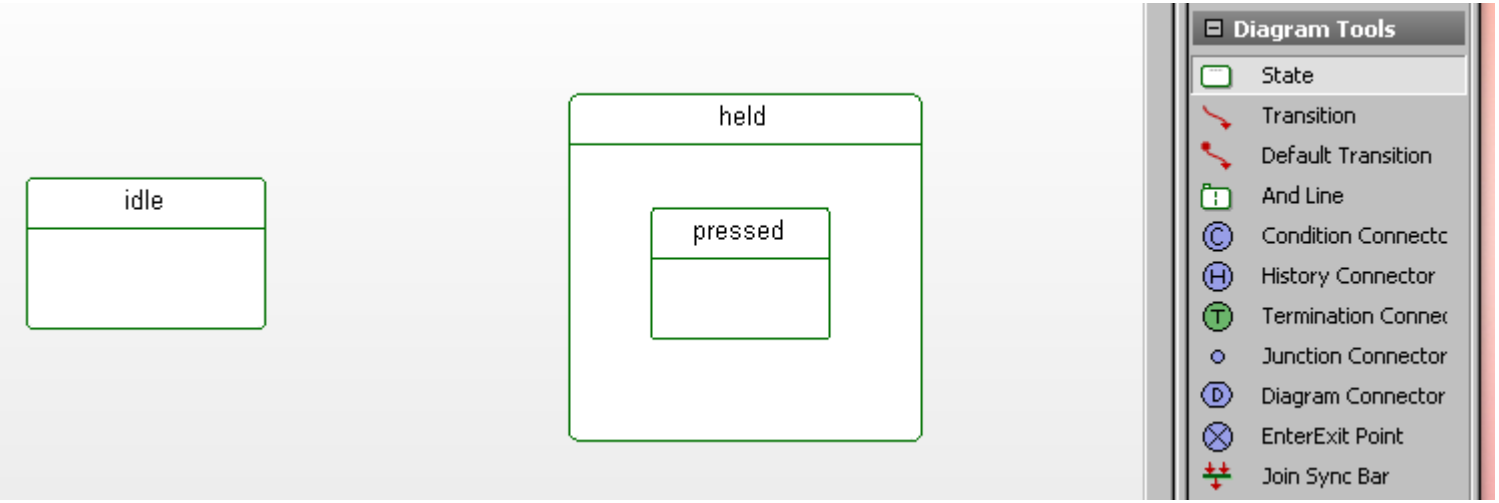
- Repeat the same for the link between Timer and Display



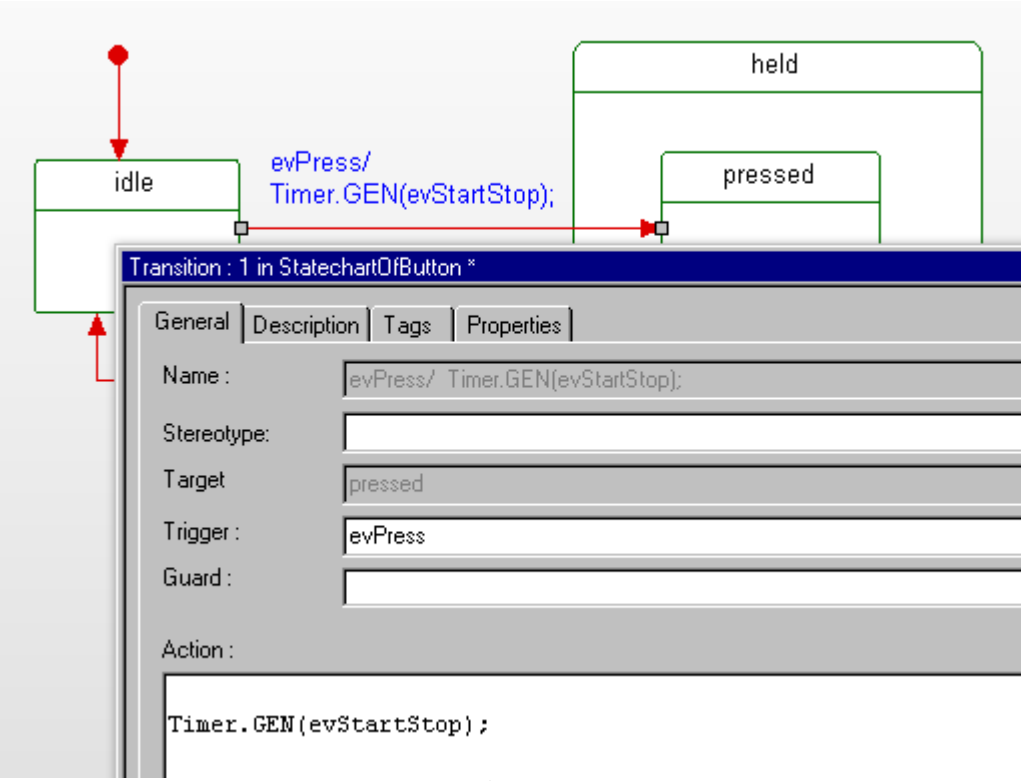
- Select Button object and right click
- Add New-Statechart



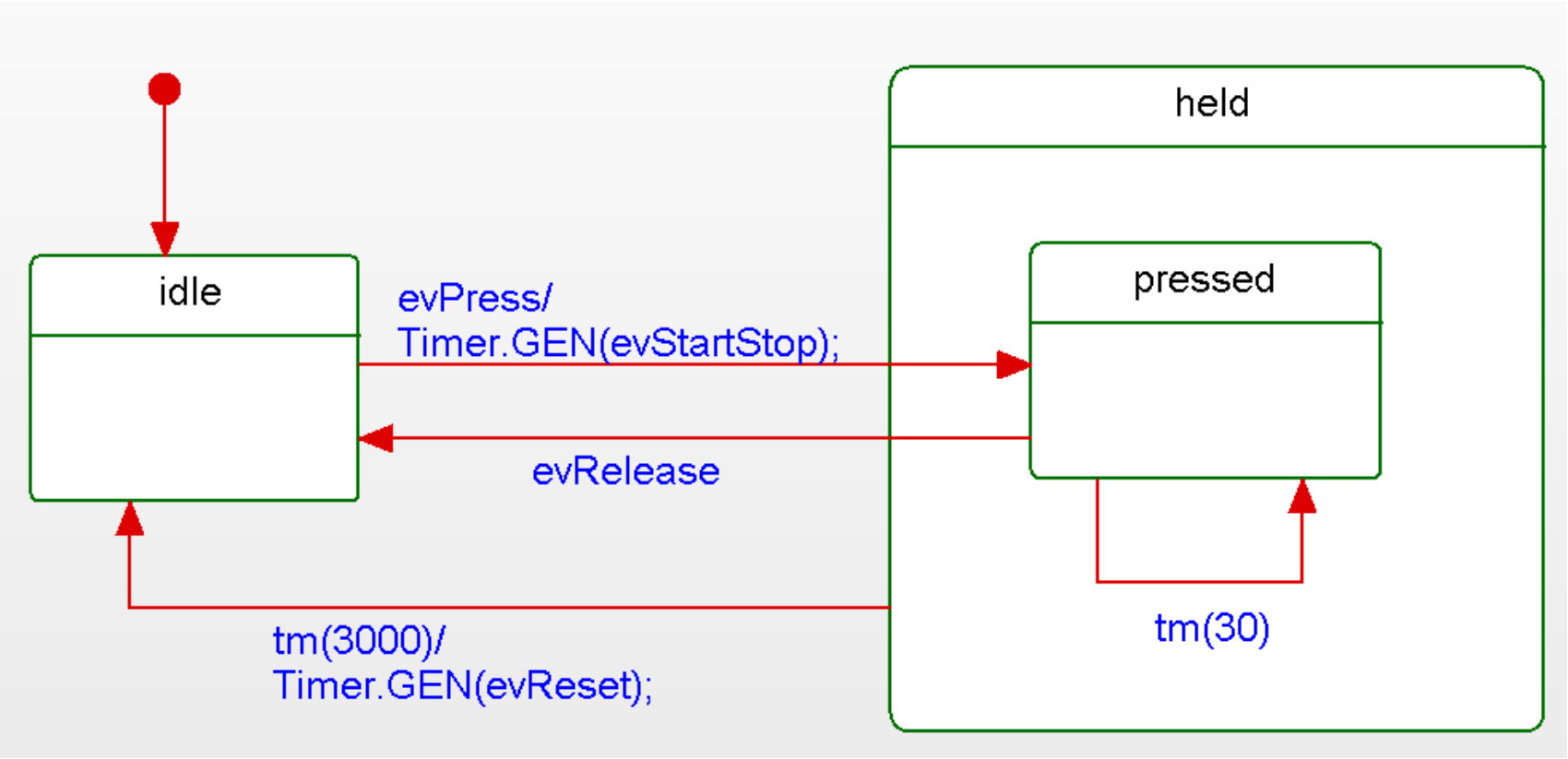
- Select State and draw states.



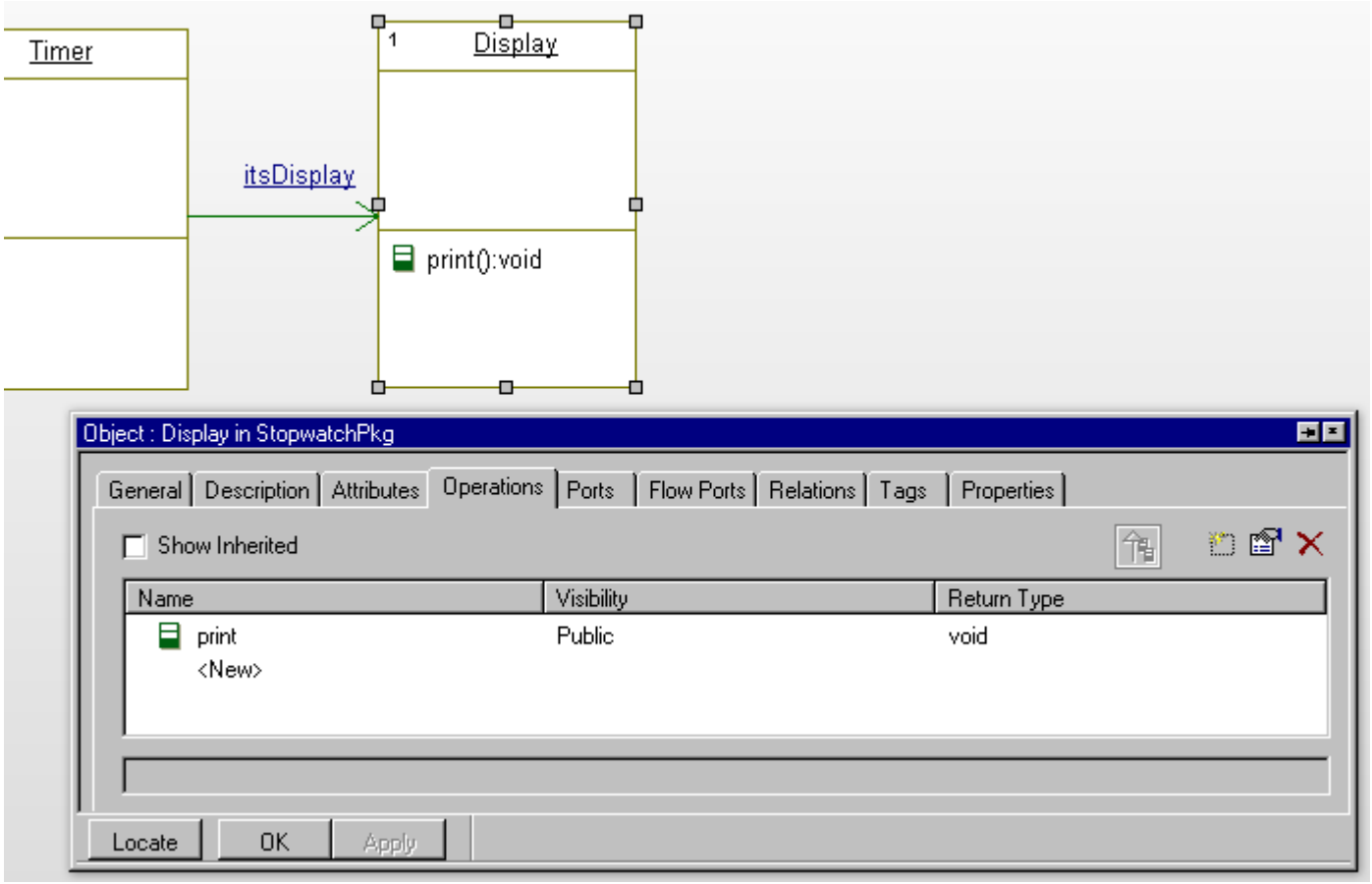
- Select Transition and draw
- Double click the transition
- Type in Trigger: evPress
- Type in Action: Timer.GEN(evStartStop);



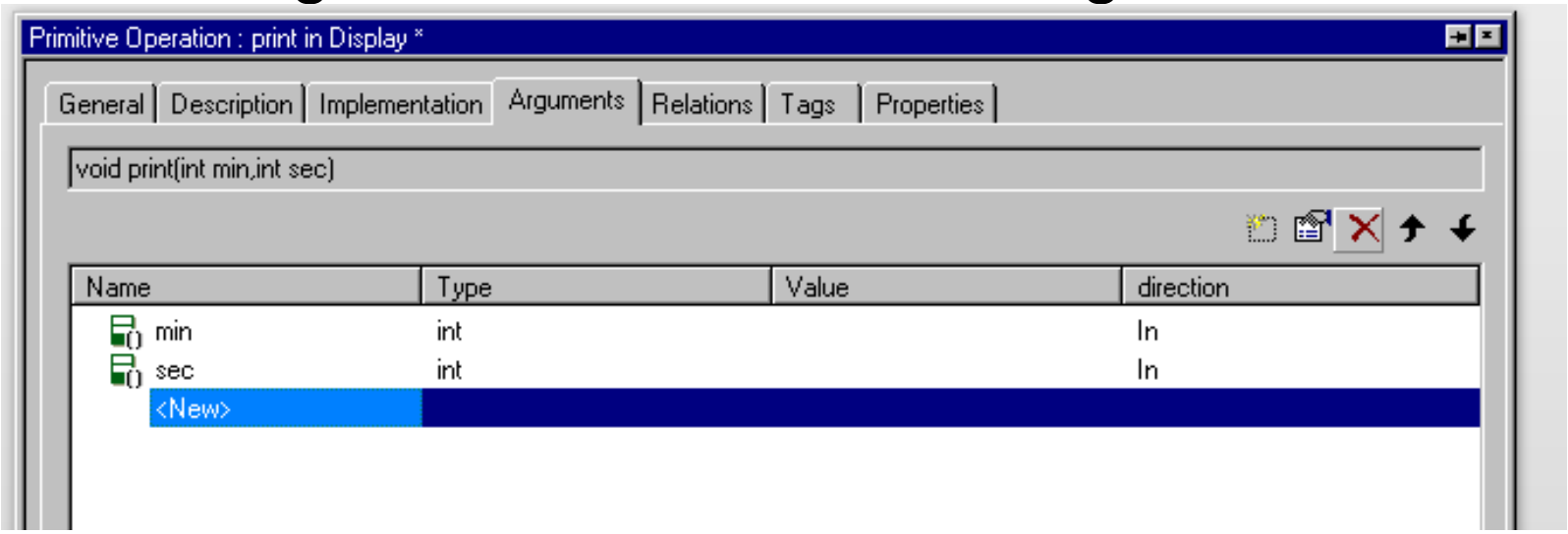
- Complete the statechart



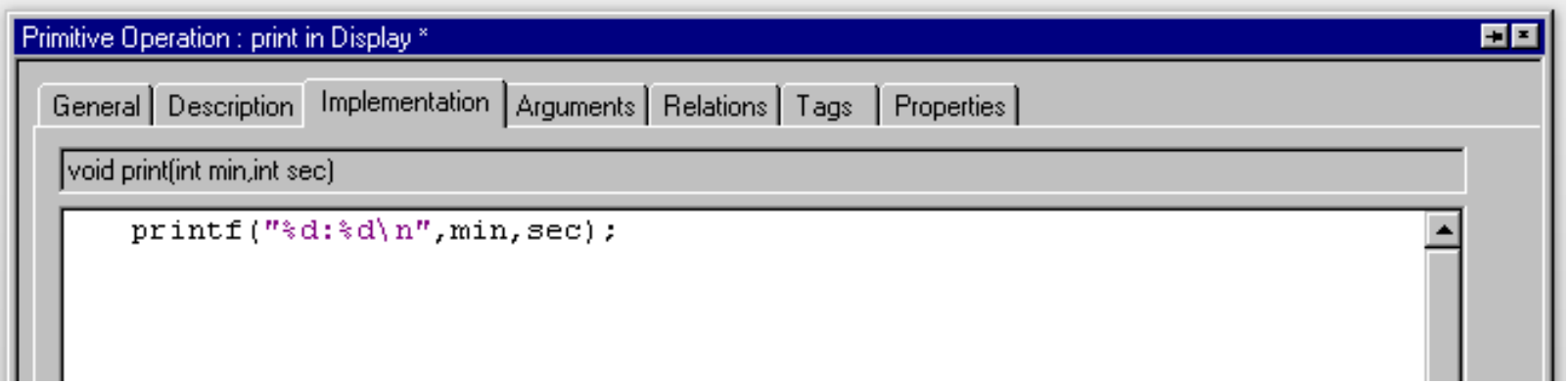
- Doble click Display object and open Feature window
- Select Operations Tab and press New
- Select Primitive Operations and name print



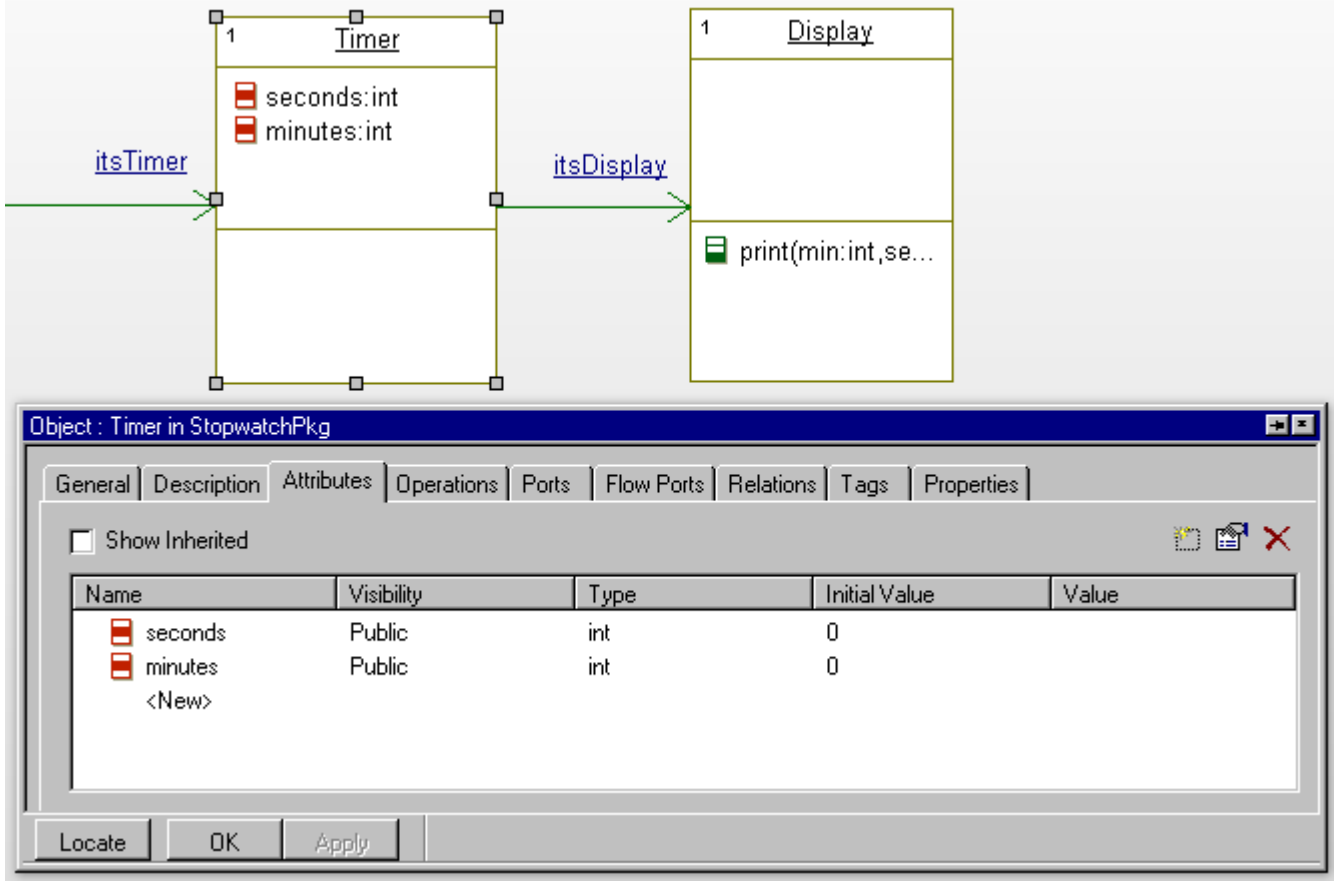
- Double click print operation
- Select Arguments Tab and add arguments: min, sec.



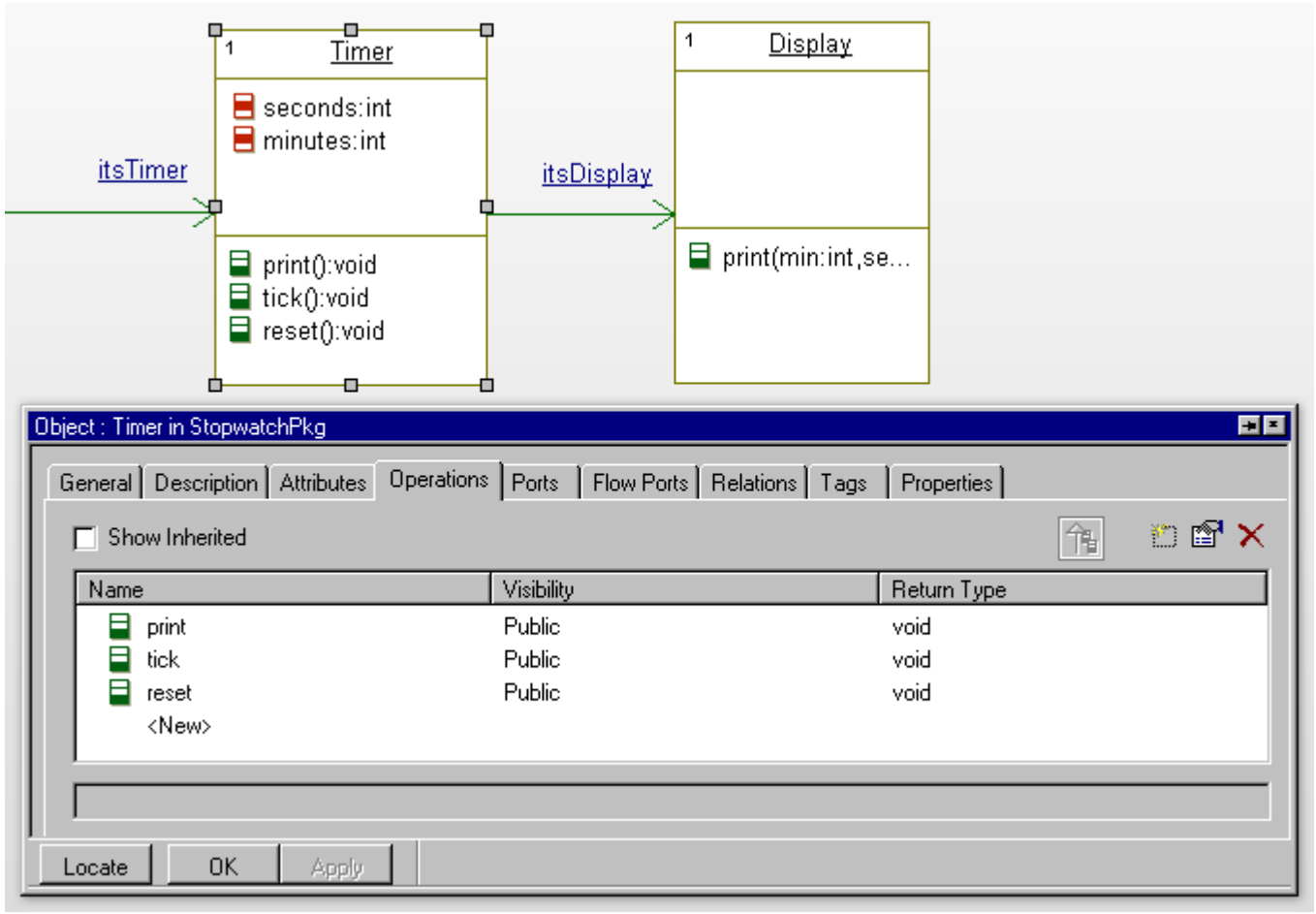
- Select Implementation and type in code



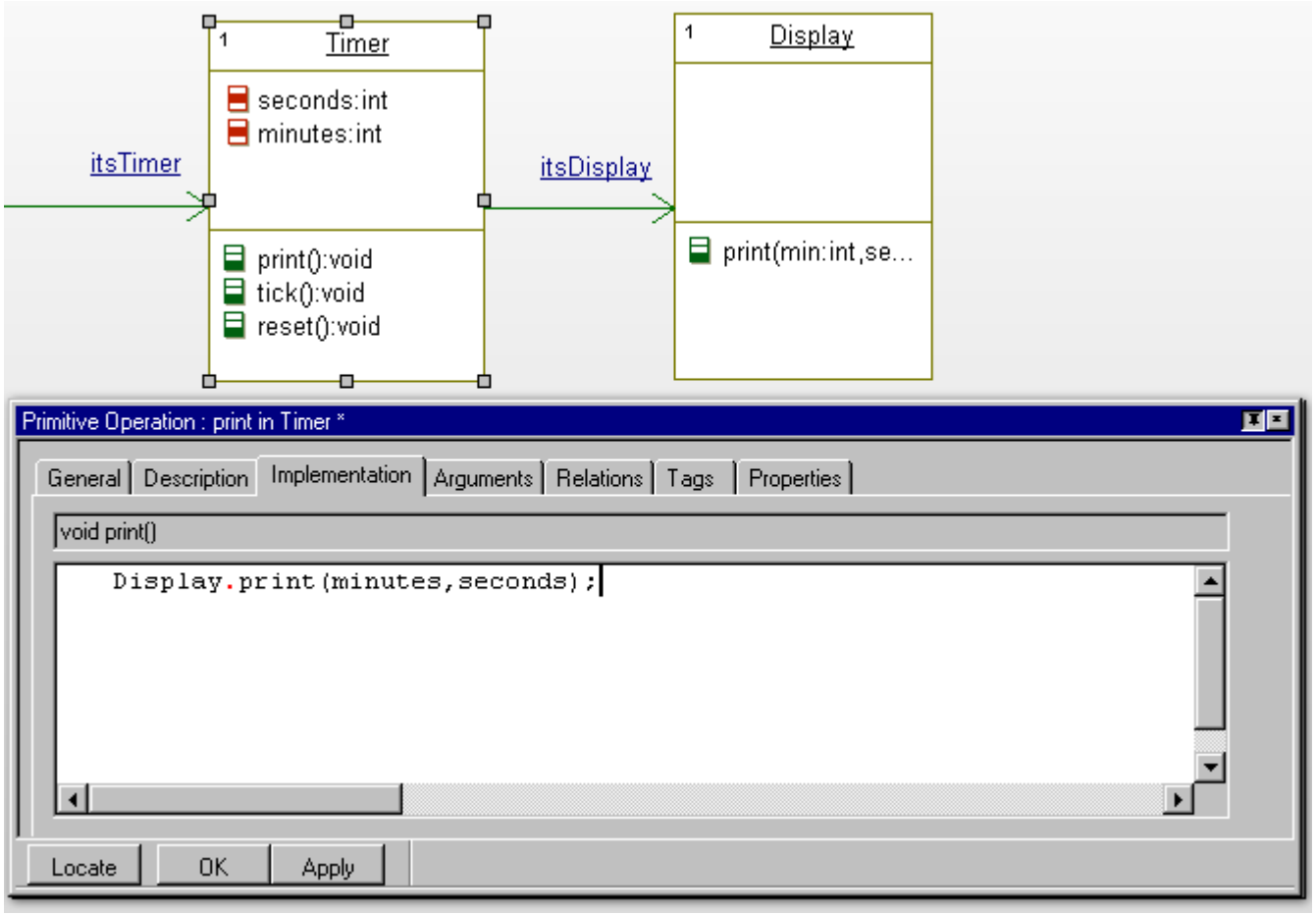
- Double click Timer object
- Select Attributes Tab
- Add attributes(seconds, minutes)



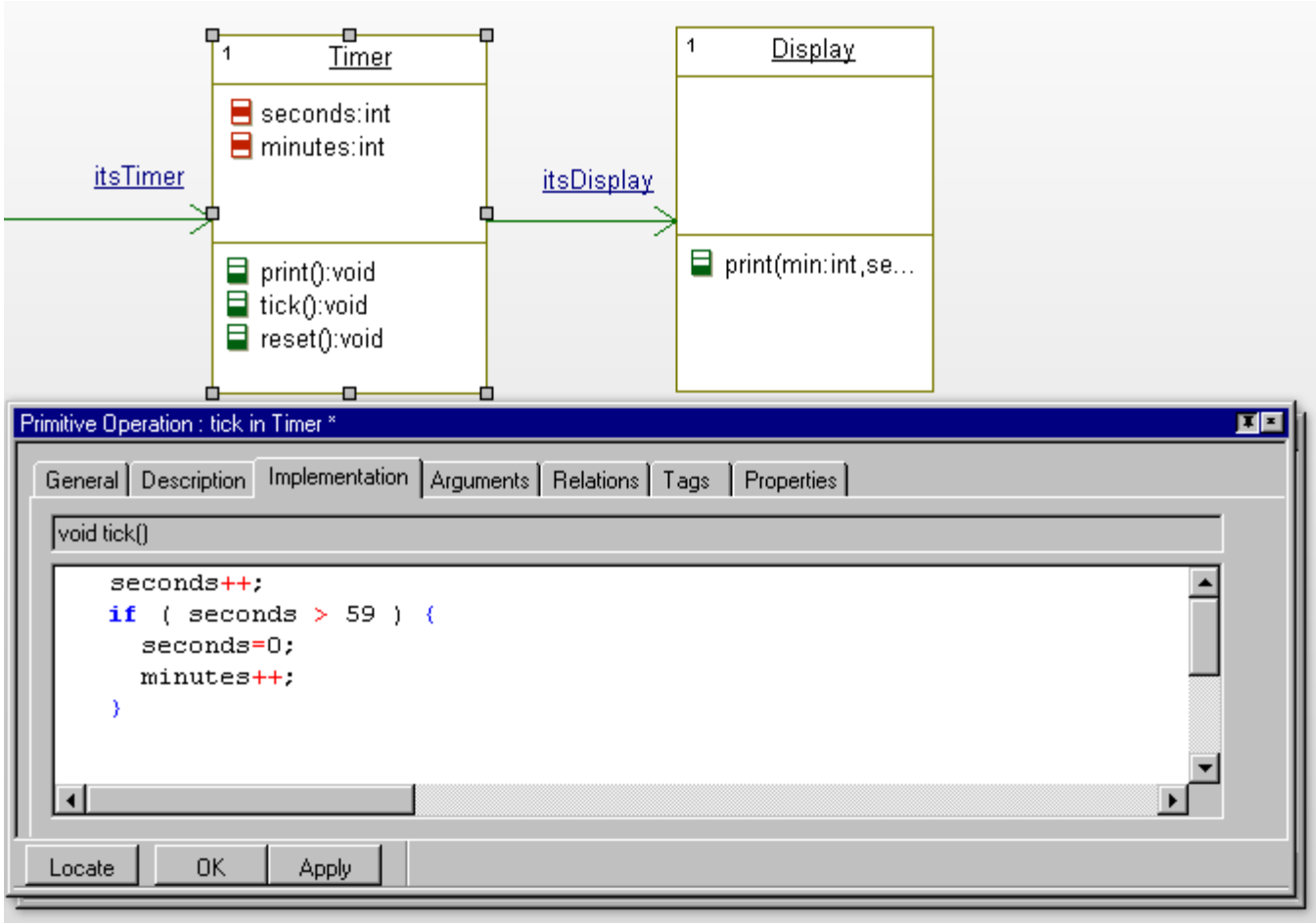
- Add operations(print, tick, reset) in Timer object



- Double click print operation
- Add Implementations

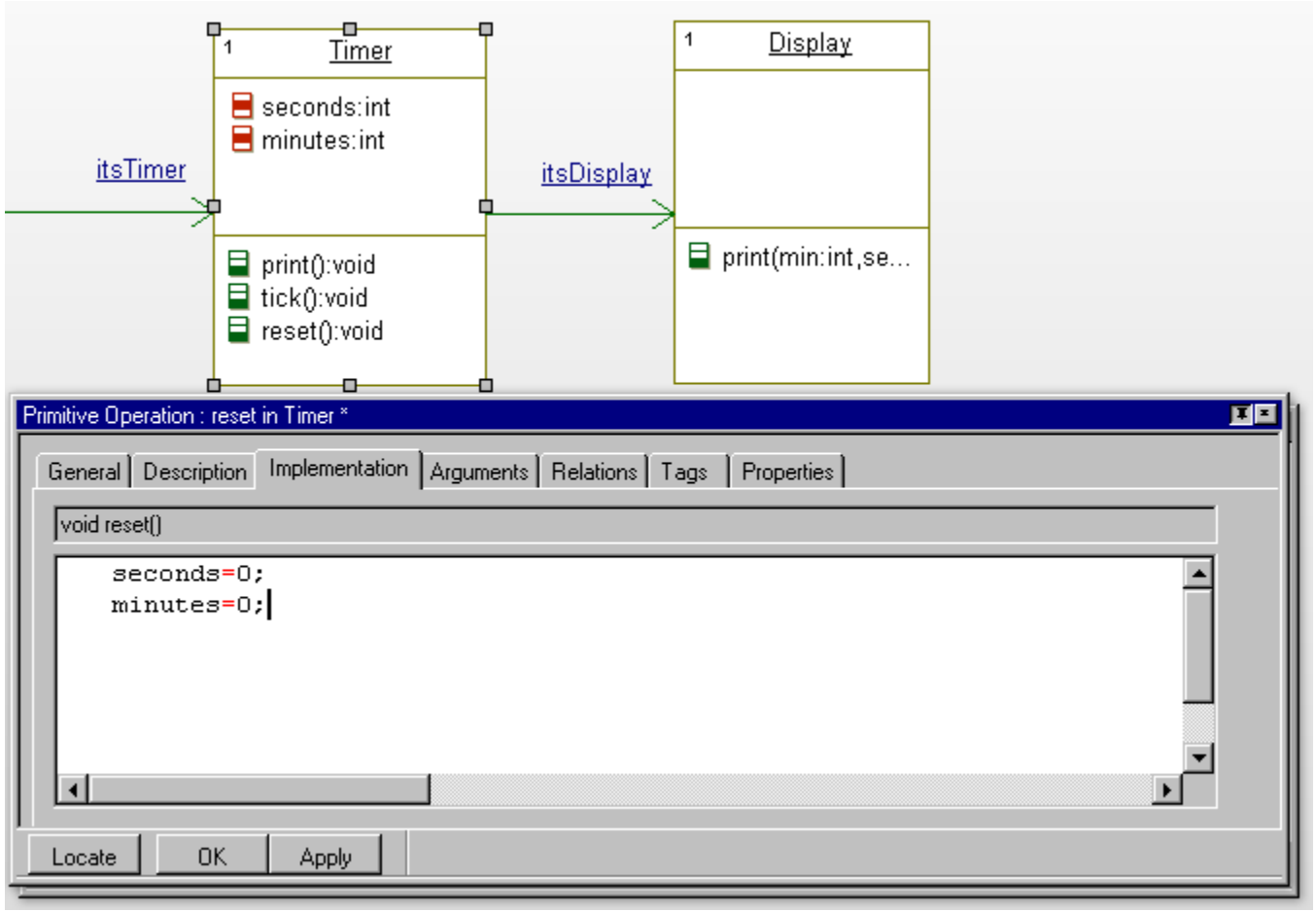


- Double click tick operation
- Add Implementations



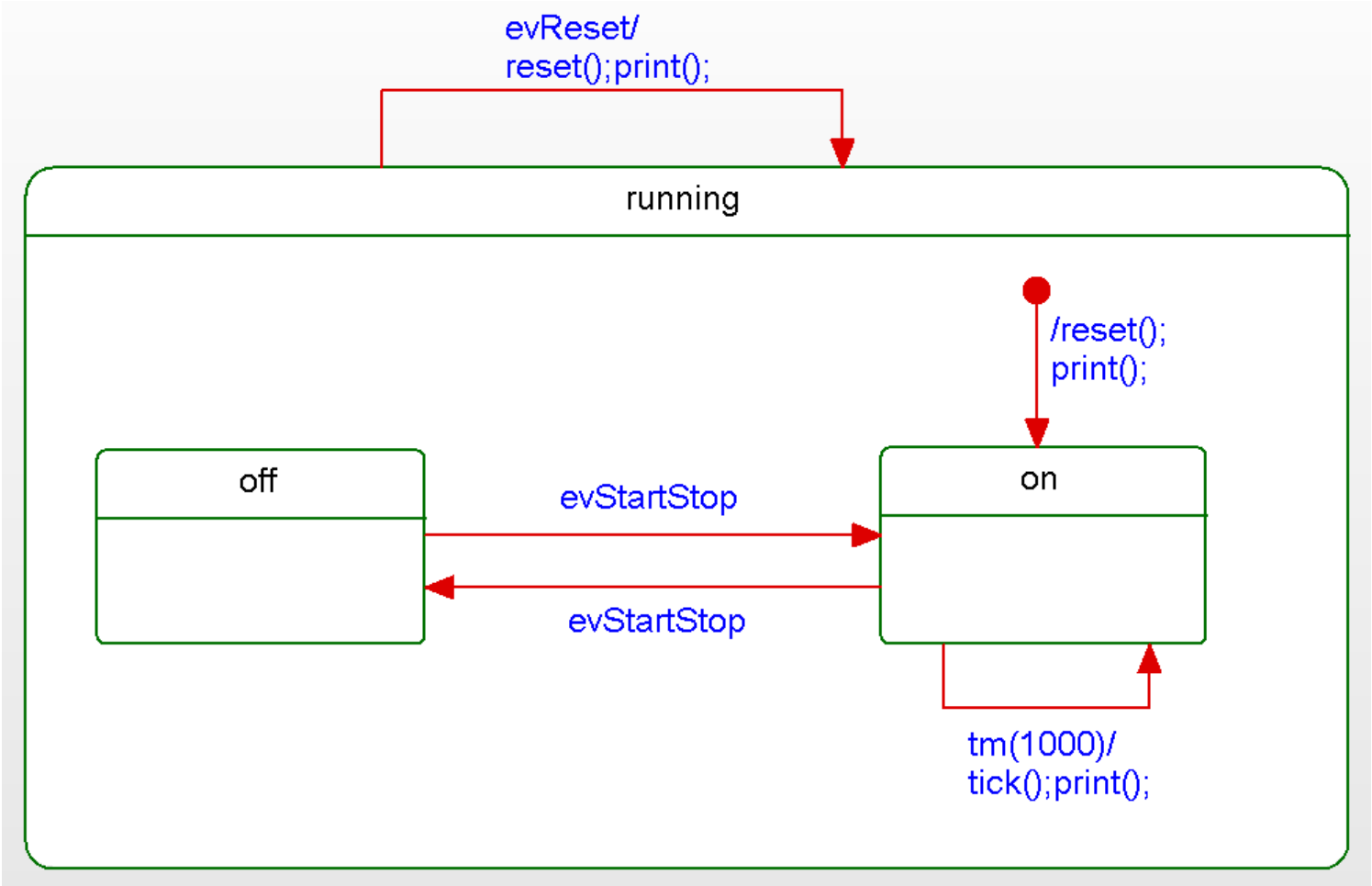
Timer attributes

- Double click reset operation
- Add Implementations



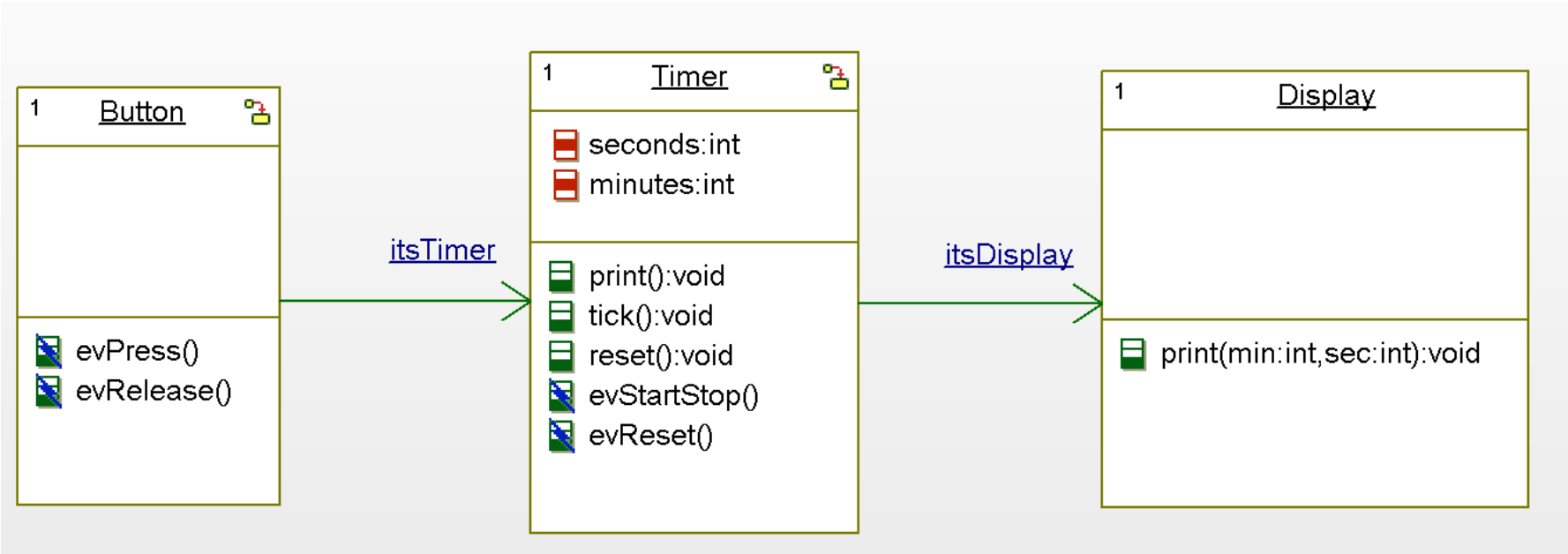
Statechart in Timer

- Right click Timer and Add New-Statechart



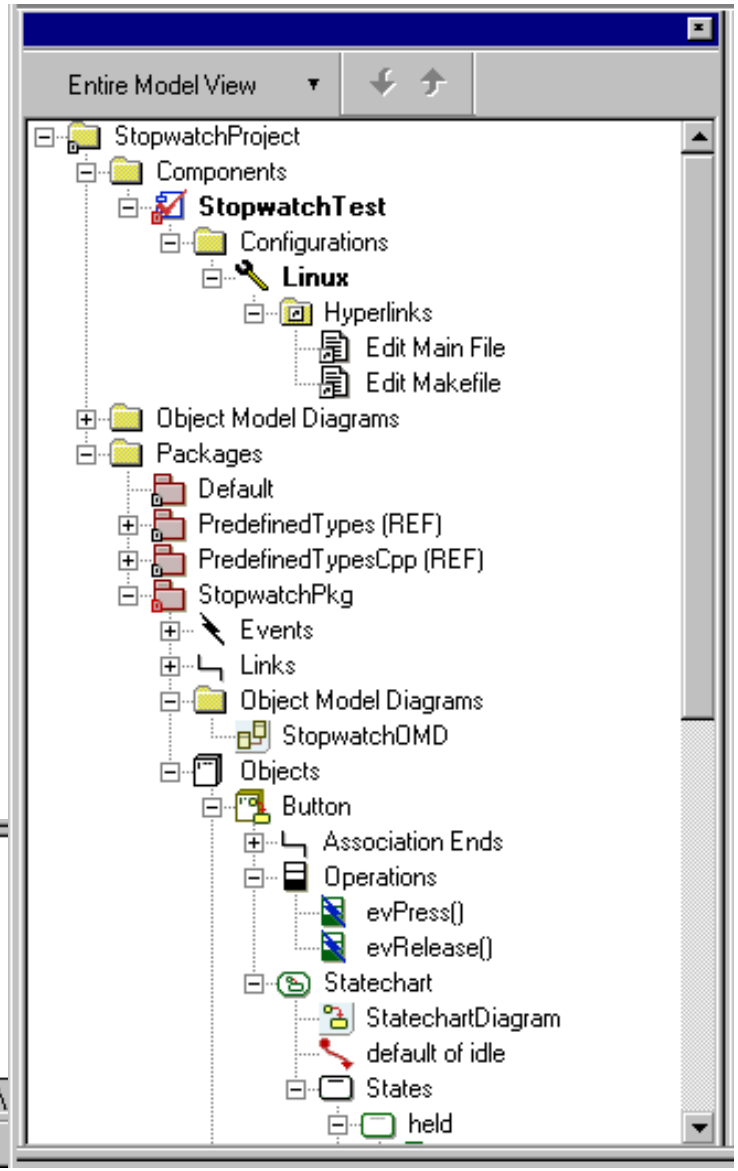
Object Model Diagram

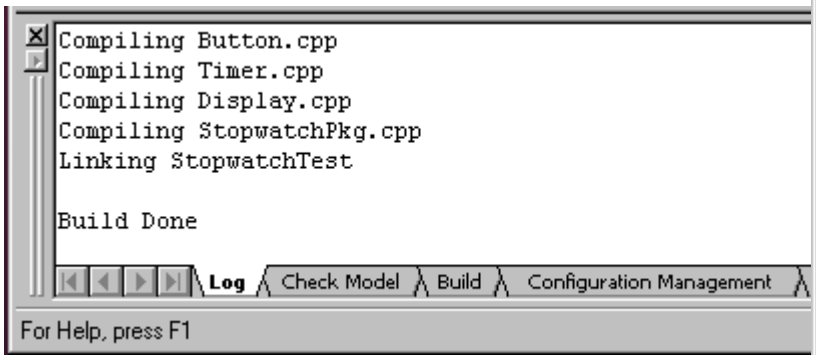
- Stopwatch OMD

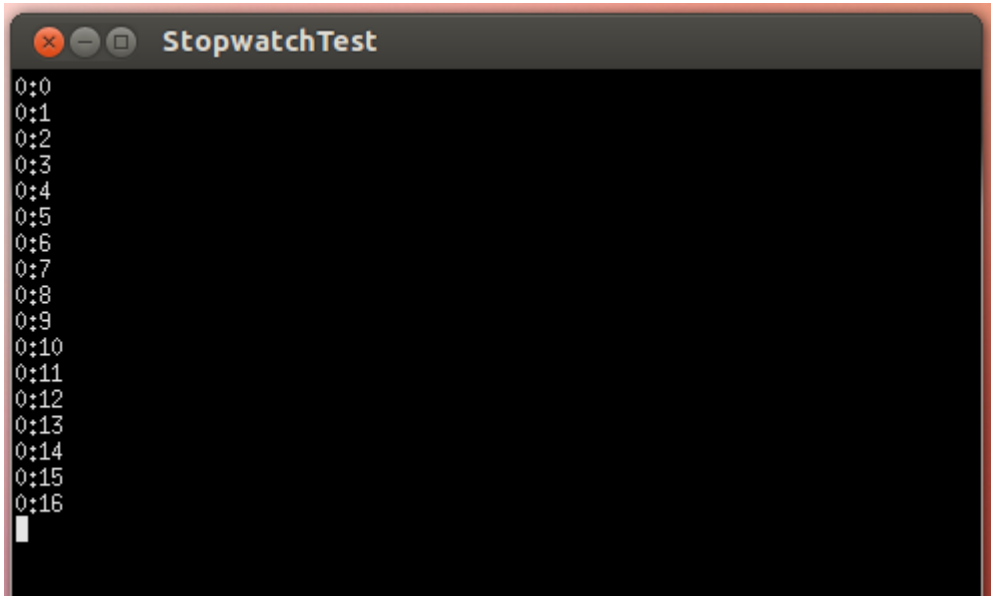


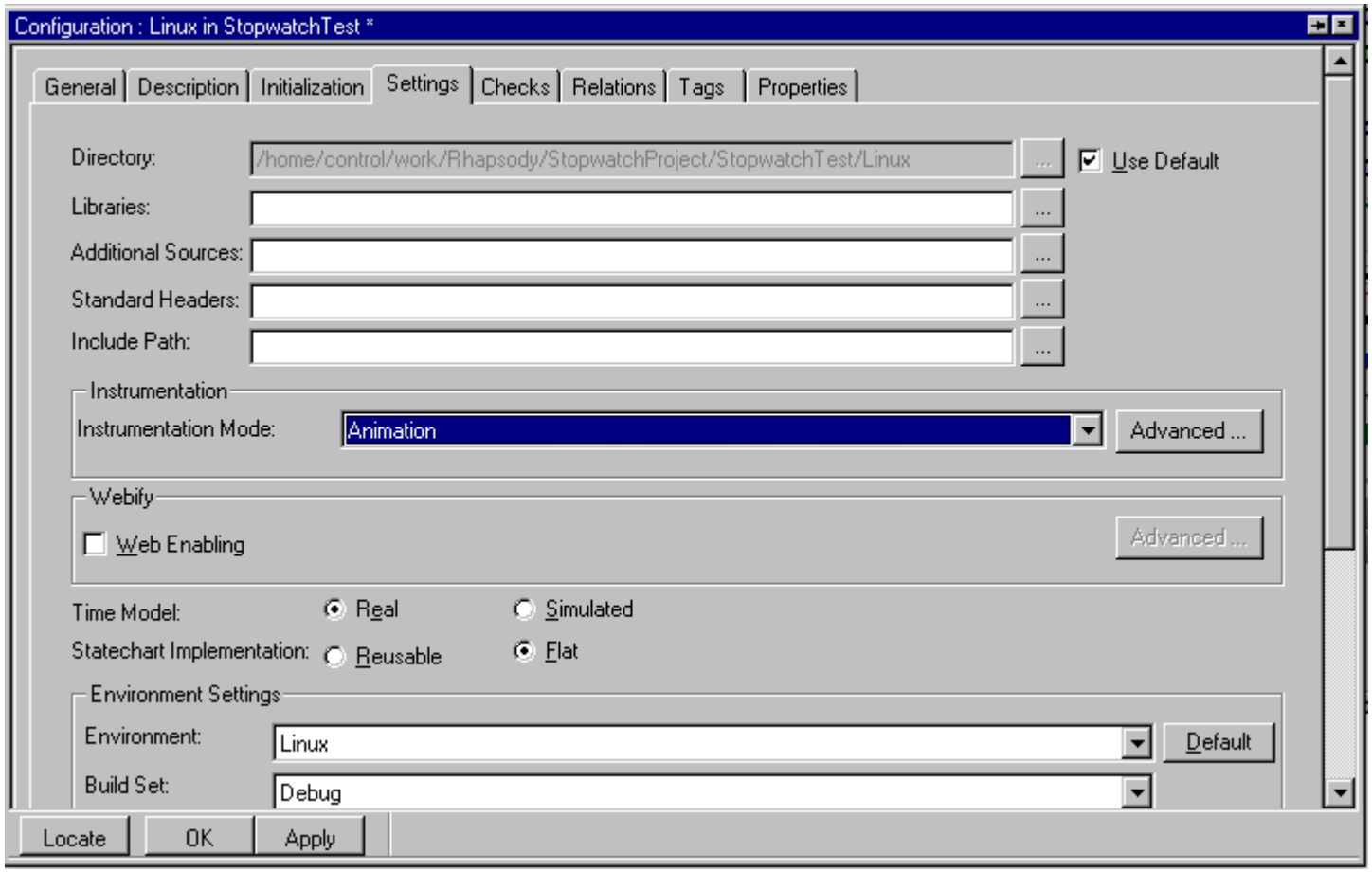
Generate and build

- Change component and configuration names
- Generate and build

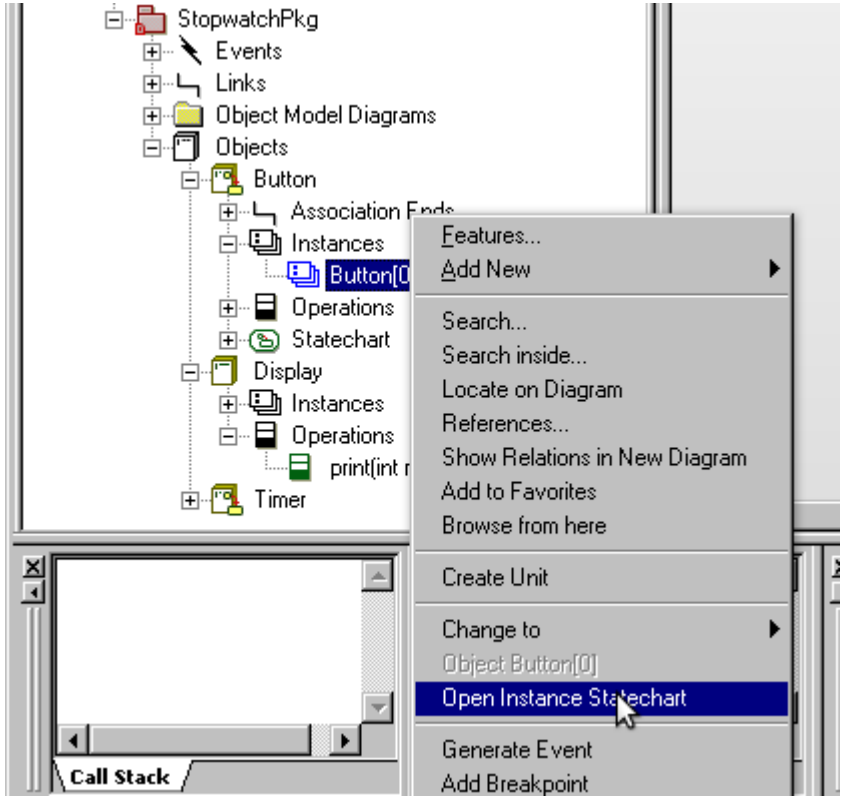




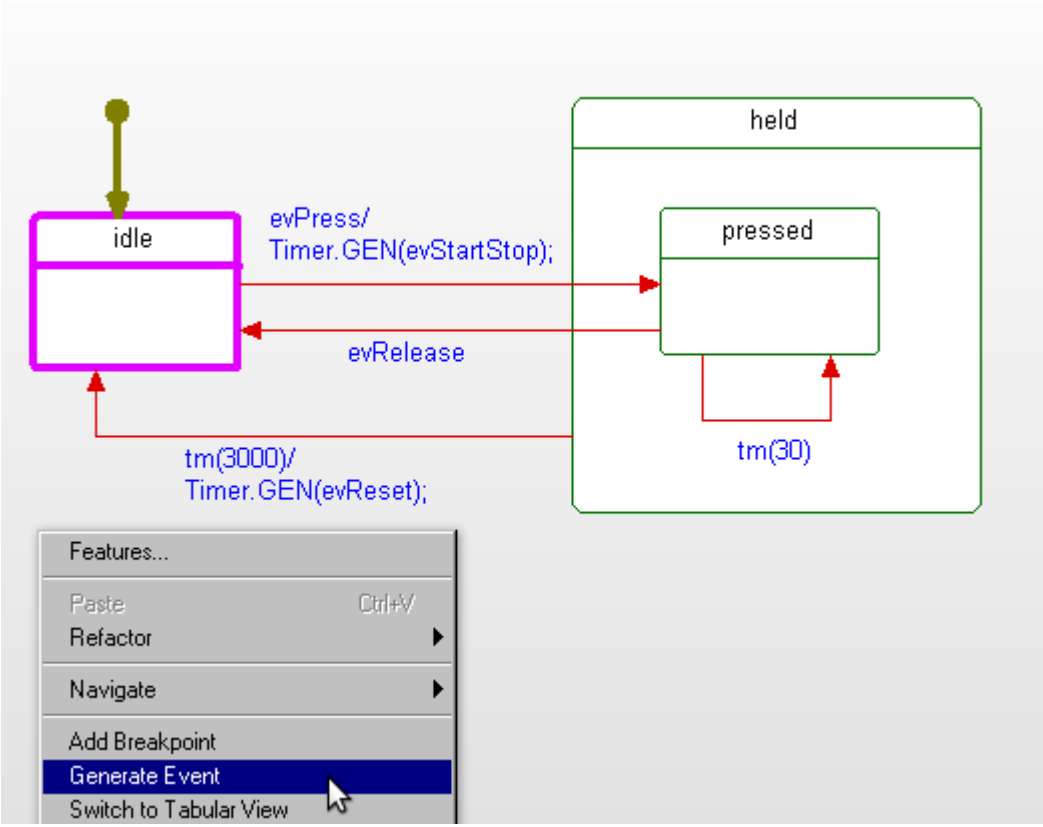

StopwatchTest
0:0
0:1
0:2
0:3
0:4
0:5
0:6
0:7
0:8
0:9
0:10
0:11
0:12
0:13
0:14
0:15
0:16


Configuration : Linux in StopwatchTest *
General
Description
Initialization
Settings
Checks
Relations
Tags
Properties
Directory:
/home/control/work/Rhapsody/StopwatchProject/StopwatchTest/Linux
Use Default
Libraries:
Additional Sources:
Standard Headers:
Include Path:
Instrumentation
Instrumentation Mode:
Animation
Advanced ...
Webify
Web Enabling
Advanced ...
Time Model:
Real
Simulated
Statechart Implementation:
Reusable
Flat
Environment Settings
Environment:
Linux
Default
Build Set:
Debug
Locate
OK
Apply

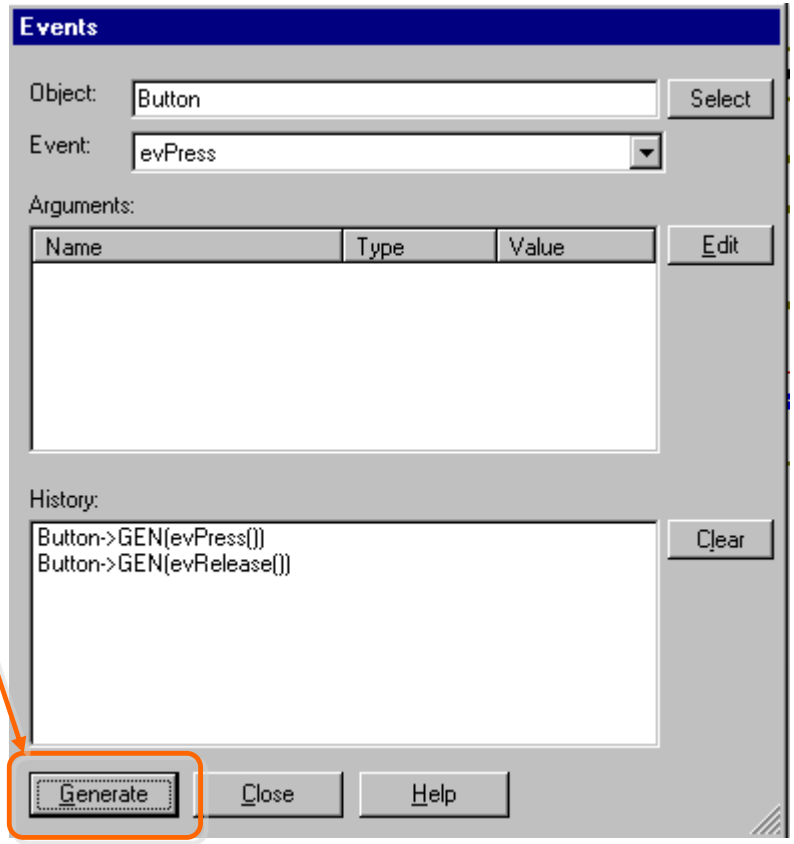

IBM Rational Rhapsody Developer for C++ - StopwatchProject.
File Edit View Code Layout Tools Window Help
StopwatchTest
Linux


StopwatchPkg
Events
Links
Object Model Diagrams
Objects
Button
Association Ends
Instances
Button[0
Operations
Statechart
Display
Instances
Operations
print(int
Timer
Features...
Add New
Search...
Search inside...
Locate on Diagram
References...
Show Relations in New Diagram
Add to Favorites
Browse from here
Create Unit
Change to
Object Button[0]
Open Instance Statechart
Generate Event
Add Breakpoint
Call Stack

- Right click and Generate Event

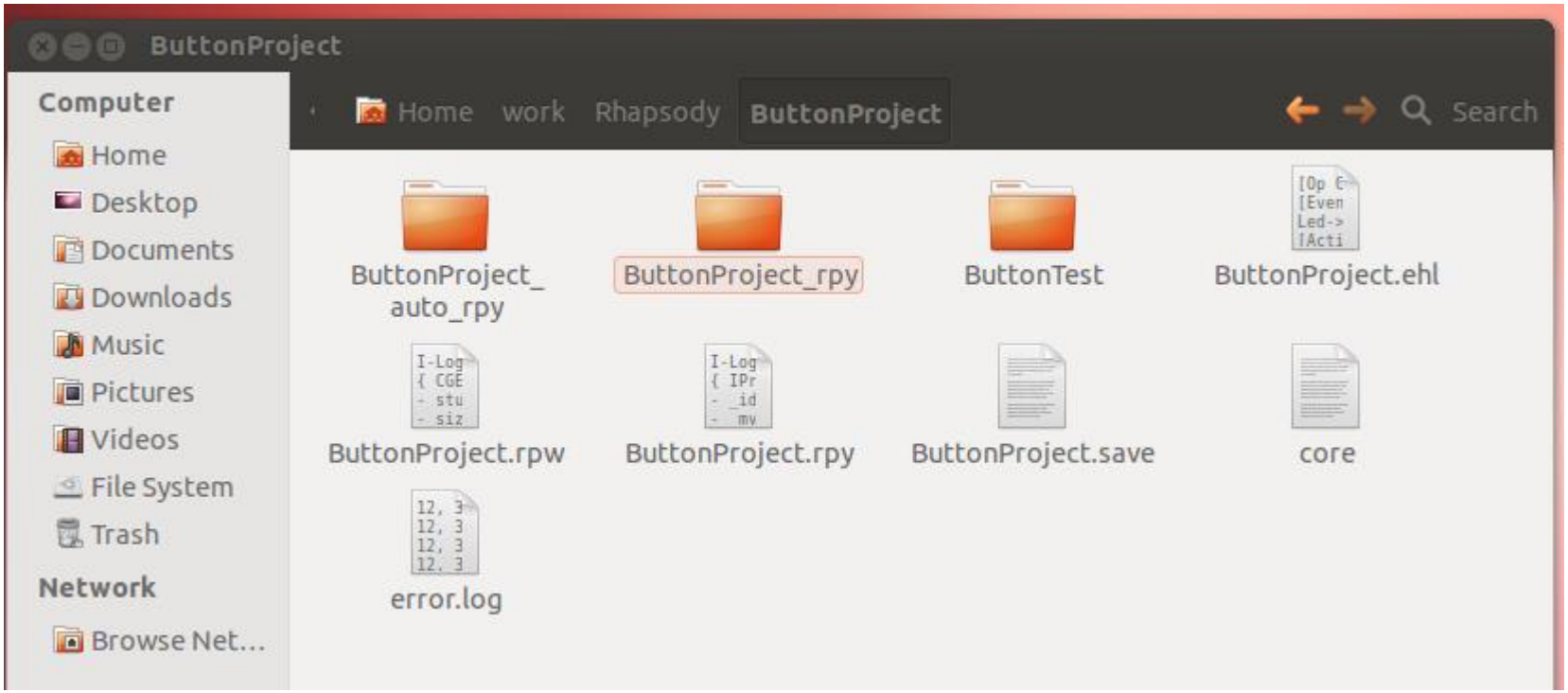


- Generate evPress and evRelease within 3 seconds



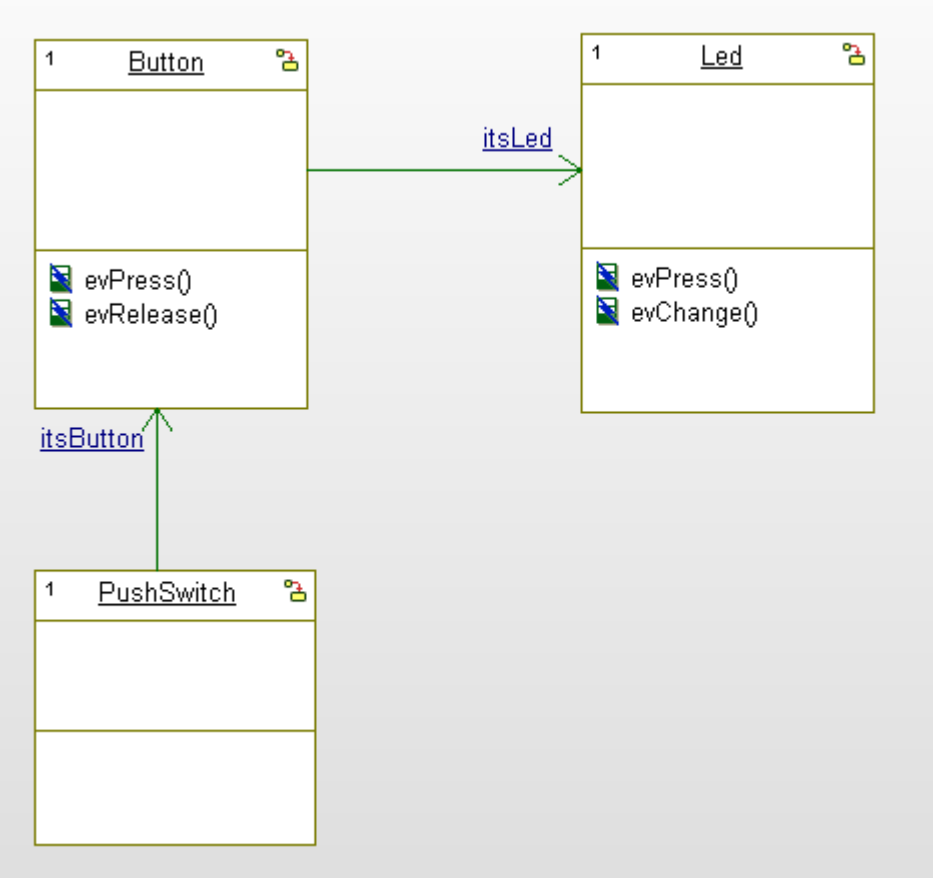
Exercise 4: Button Project

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

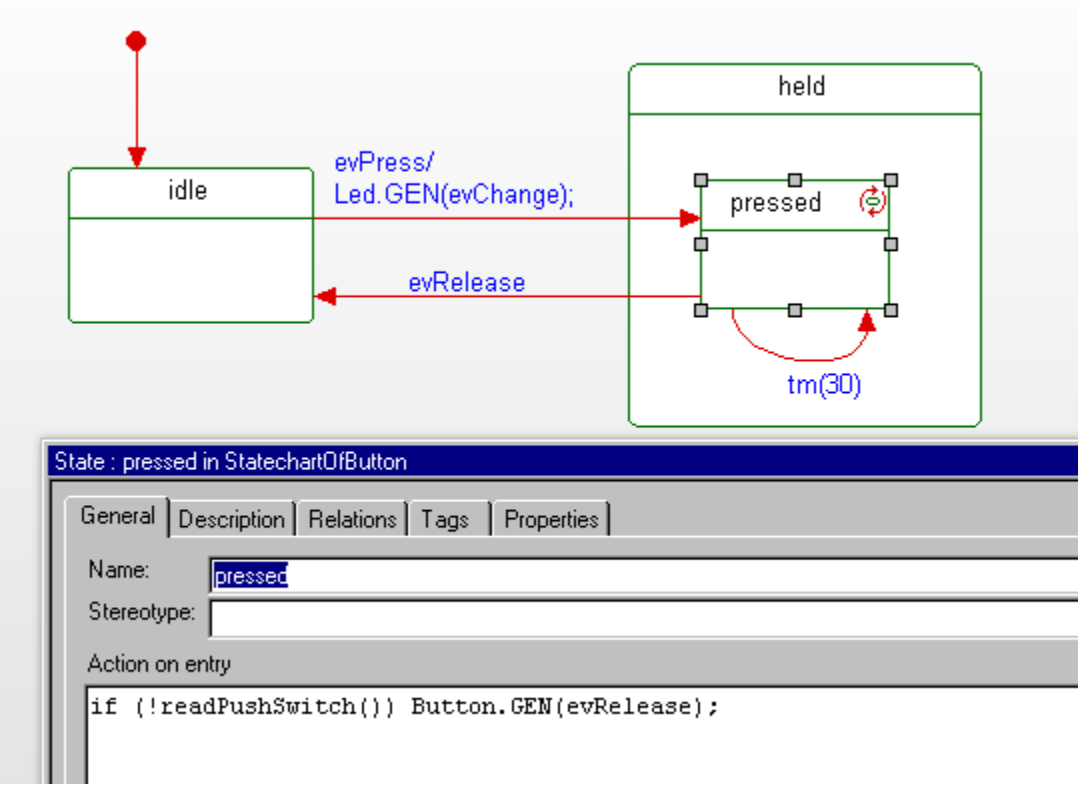


Object Model Diagram

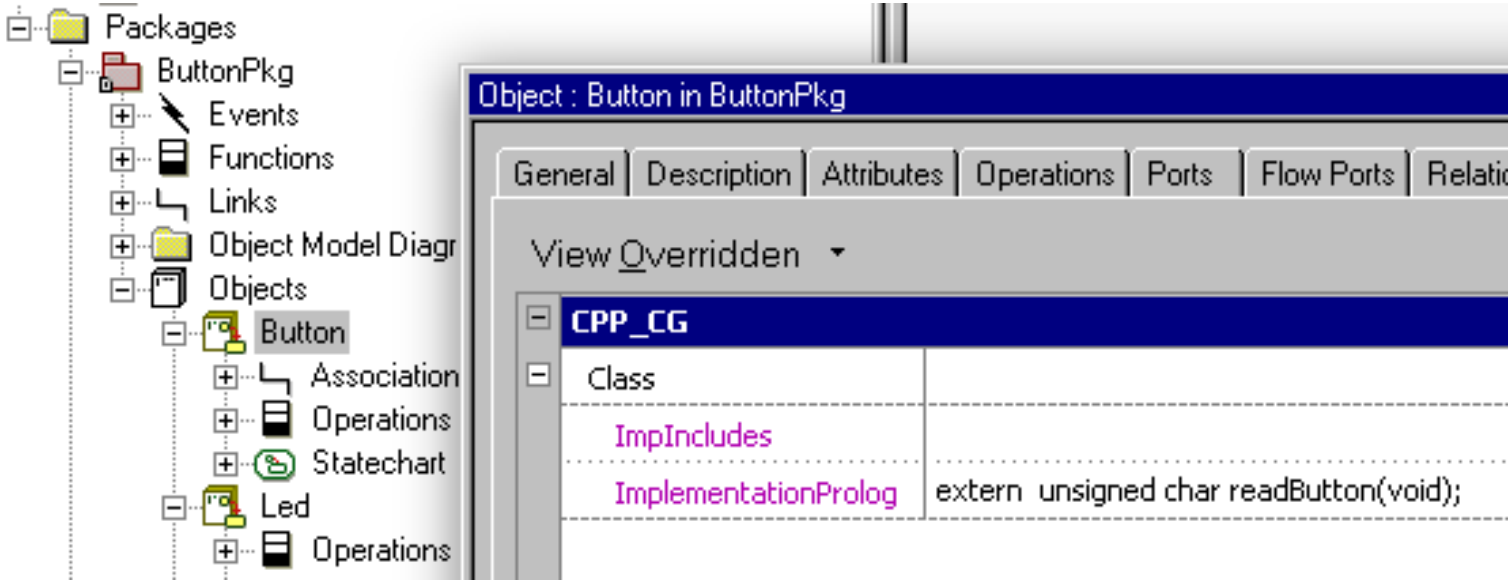
- Object Model Diagram in ButtonProject



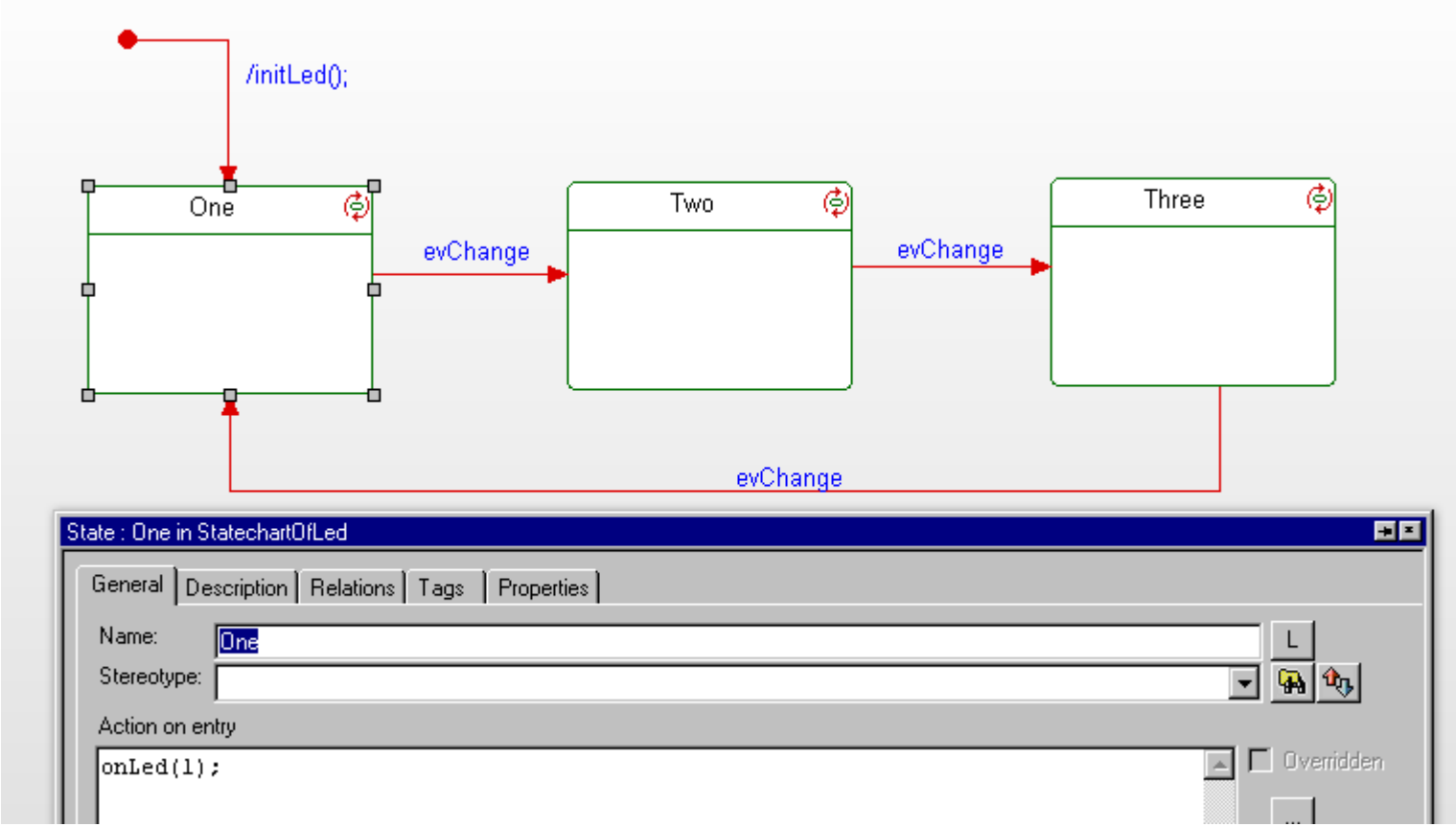
- Statechart in Button



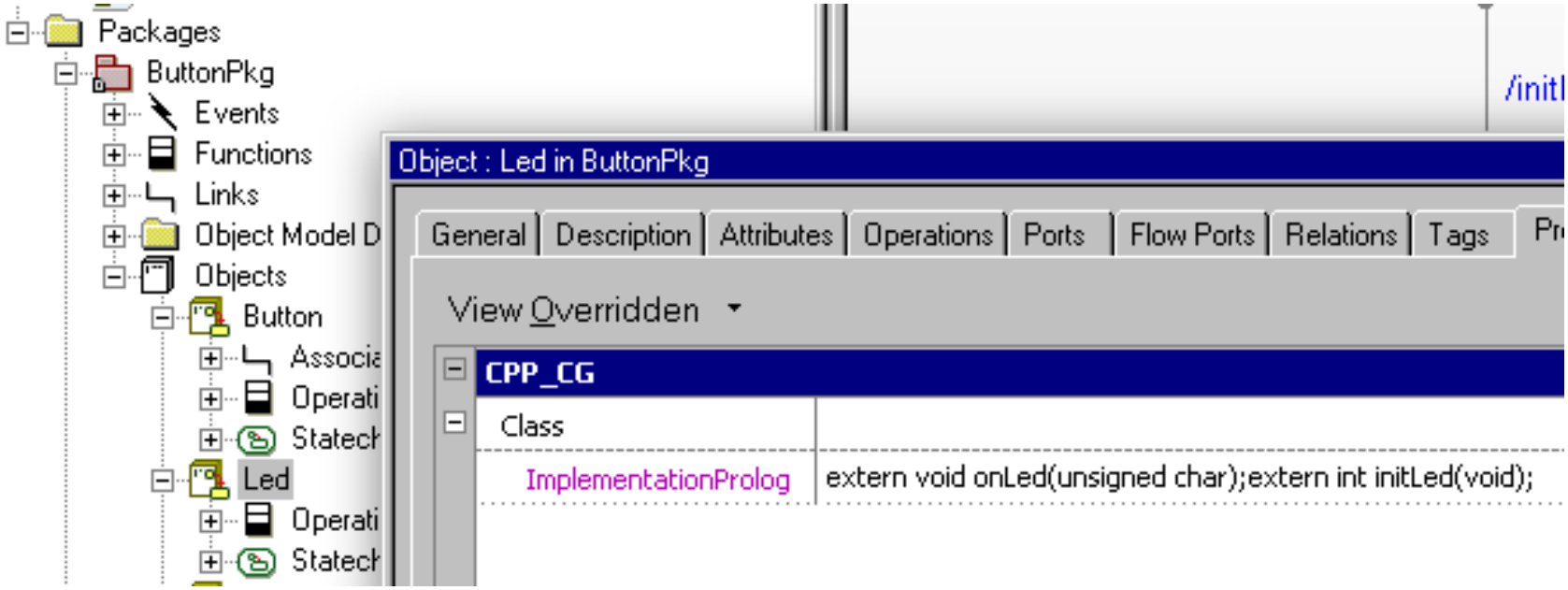
- External function prototypes



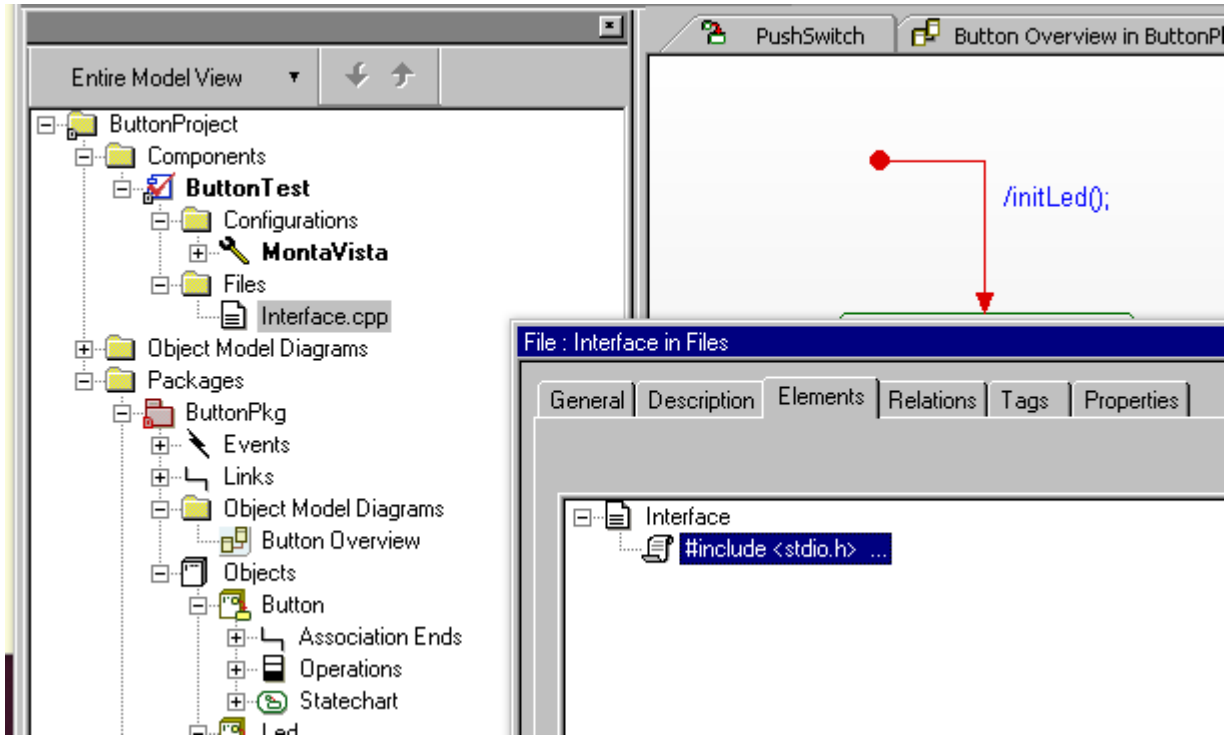
Statechart in Led



- External function prototypes



- Interface.cpp



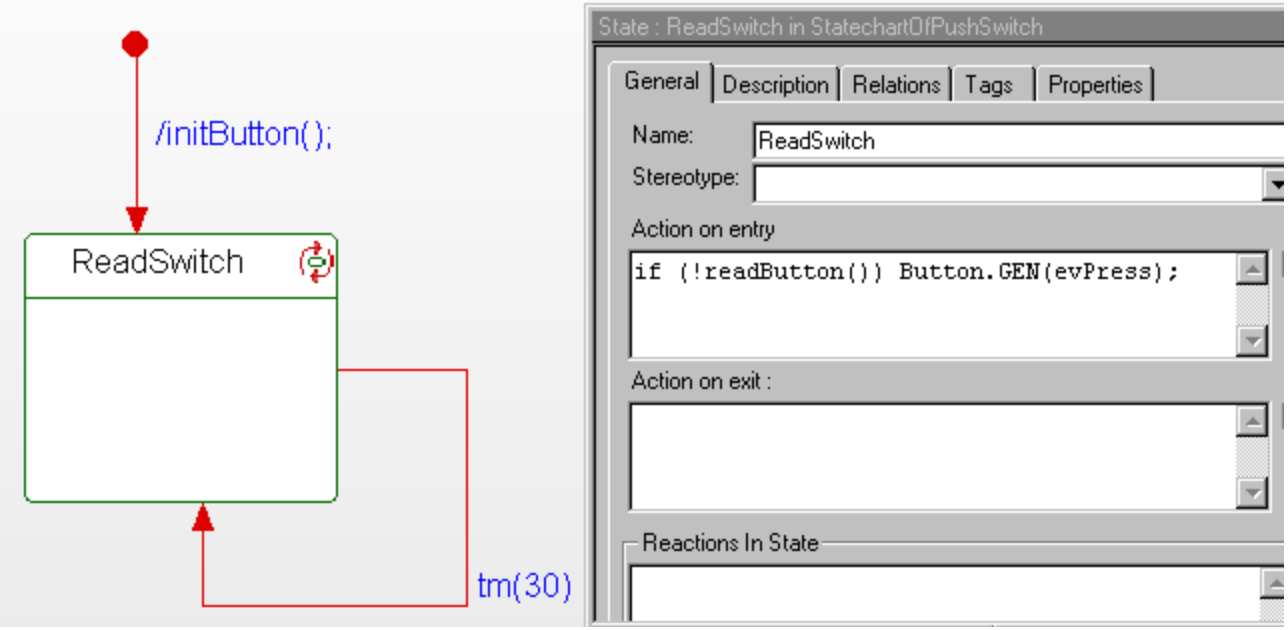
Interface.cpp(1)

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

int fd_button;
int fd_led;

int initButton(void)
{
        fd_button=open("/dev/gpio_button",O_RDWR);
        if (fd_button < 0) {
        printf("Device open error : %s\n","/dev/gpio_button");
        exit(1);
        }
}
int initLed(void)
{
        fd_led=open("/dev/gpio_led",O_RDWR);
        if (fd_led < 0) {
        printf("Device open error : %s\n","/dev/gpio_led");
        exit(1);
        }
}
```

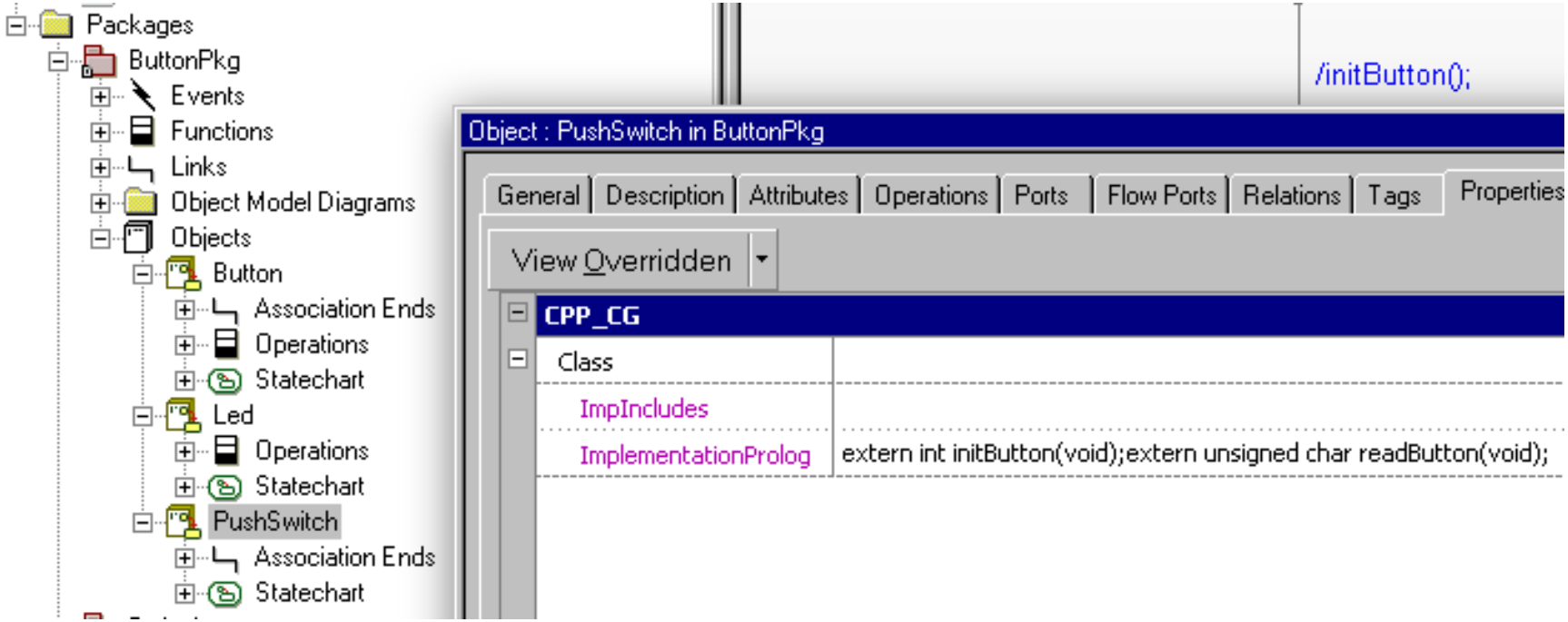
Interface.cpp(2)

```
unsigned char readButton(void)
{
        char buf[30];
        read(fd_button, buf, 1);
    return buf[0];
}
void onLed(unsigned char data)
{
        char wbuf[30];
        wbuf[0]=data;
    write(fd_led,&wbuf,1);
}
/*********************************************************************
        File Path       : ButtonTest/MontaVista/Interface.cpp
*********************************************************************/
```

- Statechart in PushSwitch



- External function prototypes



Exercise 5: Stopwatch with real displays and switches

- Modify the StopwatchProject to use LCD displays and button switch on the target