Embedded Wizard tutorial

Andreas Deininger

andreas@deininger.net

2019-03-16

Contents

1 Introduction **5**

1.1 About Embedded Wizard . . . 5
1.2 Motivation for this tutorial . . . 5
1.3 Scope of the tutorial . . . 6
1.4 Sample application . . . 6
1.5 Target audience, prior knowledge . . . 6
1.6 Prerequisites . . . 7
1.7 Feedback and questions . . . 7

2 Getting started with ’Hello world’ **9**

2.1 Setting up a new project . . . 9
2.2 Structure of the project . . . 9
2.2.1 Units . . . 10
2.2.2 Application class . . . 11
2.2.3 Running the application . . . 16

3 Adding your first GUI component **17**

3.1 Adding the empty component itself . . . 17
3.2 Filling the component . . . 19
3.3 Defining a button action performed on click . . . 24
3.4 Test the component in isolation . . . 26
3.5 Add the component to the application’s root component . . . 28

4 Making your component truly reusable **33**

4.1 Extending your component with properties . . . 33
4.2 Adapt and fix the main application . . . 37
4.3 Adding a second component (weather forecast) . . . 38

5 Device integration: Hello world from our device **41**

5.1 GUI builder and platform packages . . . 41
5.2 Adapting the project structure . . . 41

5.3 Adding a interface device class 43
5.4 Adding another profile 46
5.5 Adding a browser device class variant 48
5.6 Implementing a different behavior for the browser device class variant 50
5.7 Running the application inside a web browser 52
5.8 Finalizing device actions 55
5.8.1 Make the device beep eventually 55
5.8.2 Presenting the weather forecast on the browser device 57

6 Localizing your application **59**
6.1 Multilingual applications: concepts 59
6.2 Adding another language 60
6.3 Adding language strings / constants 61
6.4 Localize the application text strings 63

7 Adding a language switcher **65**
7.1 Adding the resources for the language switcher 65
7.1.1 Adding the flag bitmap resources 65
7.1.2 Building a pushbutton flag component 66
7.1.3 Building the language switcher component 67

1 Introduction

1.1 About Embedded Wizard

Embedded Wizard is a graphical user interface tool developed and distributed by TARA Systems GmbH for creating graphical user interface (GUI) applications mainly for embedded systems.

1.2 Motivation for this tutorial

At work, we used Embedded Wizard GUI builder extensively in a development project. While the Embedded Wizard software proved to be an extremely powerful tool, getting started with this tool was quite challenging for us:

- Embedded Wizard strictly follows and implements an object oriented programming approach. These concepts may not be immediately obvious to a novice user however, who may be overwhelmed by terminology of properties, outlets, class variants, property references and so on.
- Device integration will an essential part of every real world project, this is where newbies are often struggling. This situation is worsened by the fact that Embedded Wizards supports a broad range of target devices, each one being different, which makes it hard to provide universal instructions here.
- Embedded Wizard comes with a comprehensive, excellent documentation which covers all aspects of the system. This documentation is topic oriented and even comes with several small, helpful sample projects. However, it remains a challenge to gain understanding on how to use and properly connect these code snippets in the context of a bigger application.

By time, we get acquainted to Embedded Wizard and the underlying concepts, the learning curve was pretty steep though. At that time, I sorely missed a step-by step tutorial, explaining in detail the development of a reasonably small, but already meaningful application. Since I couldn't find anything like that I took the initiative and started authoring such a tutorial by myself. Hopefully the provided tutorial will be helpful for you and will get you started with Embedded Wizard trouble-free and quickly!

1.3 Scope of the tutorial

This project aims at providing a tutorial for potential users of Embedded Wizard to get them started quickly.

While building a sample application step by step, the tutorial covers the following topics:

- Basic steps when building a GUI application
- Developing and prototyping reusable graphical components
- Device integration
- Localization of your application
- Persistent data storage on the target device

1.4 Sample application

In the course of this tutorial, we are going to develop a small GUI application for execution inside a web browser. A live preview of the sample application is shown in figure_title below. Feel free to click inside and explore the app and its features:

Figure 1.1: Tutorial application, live preview

The tutorial application comprises two components, titled *Bell* and *Weather report*. When pressing the push-button of the *bell* component, a beep will sound. When pressing the push-button of the *weather report* component, you will be redirected to a web site showing a weather report. The application is localized, by clicking on a flag icon in the upper right corner you can select either *English* or German as language of the user interface. When running the app in its own browser window, the chosen language setting will be stored inside the local storage of your web browser and will be remembered on the next reload or startup. This way you can keep your language setting, even when you exit and restart your browser.

1.5 Target audience, prior knowledge

The main audience for the book are professional developers that want leverage Embedded Wizard platform when developing GUi applications for their embedded devices. This tutorial assumes that you have experience with one or more programming languages (C, C++, Java, C#, …) and that you

are familiar with the concepts of object oriented programming. In the first place, this tutorial wants to provide easy to follow step by step instructions on how to build a small, but meaningful sample application. Quite often, the tutorial goes beyond this and tries to reveal the architectural patterns behind the application, specifically pointing out how the patterns are implemented by Embedded wizard. While doing so we assume at least limited familiarity with the patterns of object oriented design, this tutorial does **not** explain things like classes, methods, inheritance, … from ground up.

> **Important**
>
> If you haven't written any code by hand, this tutorial is most likely **not** for you. Don't be deceived: while Embedded Wizard provides a graphical oriented programming approach in the first place, for any meaningful application, you have to write code sooner or later to get the desired results. Don't be scared, though: this isn't too hard, everything will be explained in the course of this tutorial.

1.6 Prerequisites

Download and install Embedded Wizard Free Edition, version 9.10. The free edition has all features of the Professional edition, is is restricted to small projects however. Luckily, the free edition allows to develop and to run the tutorial application without limitation.

1.7 Feedback and questions

If you have any suggestion for improvement or comment concerning this tutorial, feel free to open an issue in the github repository associated with this tutorial.

For general question unrelated to this tutorial, you may make use of the question and answer site for Embedded Wizard users and UI developers.

Let's get started with a simple Hello, world example!

2 Getting started with 'Hello world'

For every language, its a good practice to start by printing out *Hello World!*. So let's start and do this with Embedded Wizard, too!

2.1 Setting up a new project

1. Start up Embedded Wizard Studio
2. From the main menu, select PROJECT > NEW … to start a new project
3. A popup appears:
    - In the template, select the item [Empty project].
    - Specify *EmWiTutorial* as project name.
    - For the *Location* of your project, specify a folder of your choice on your local file system.
4. Once project name and location are set, press the button btn:[Create new project] to bring up the new project.

2.2 Structure of the project

Let's have a look at the structure of the newly created project. As you can see from figure_title, in total 4 items were added to the main area, the so called composer window:

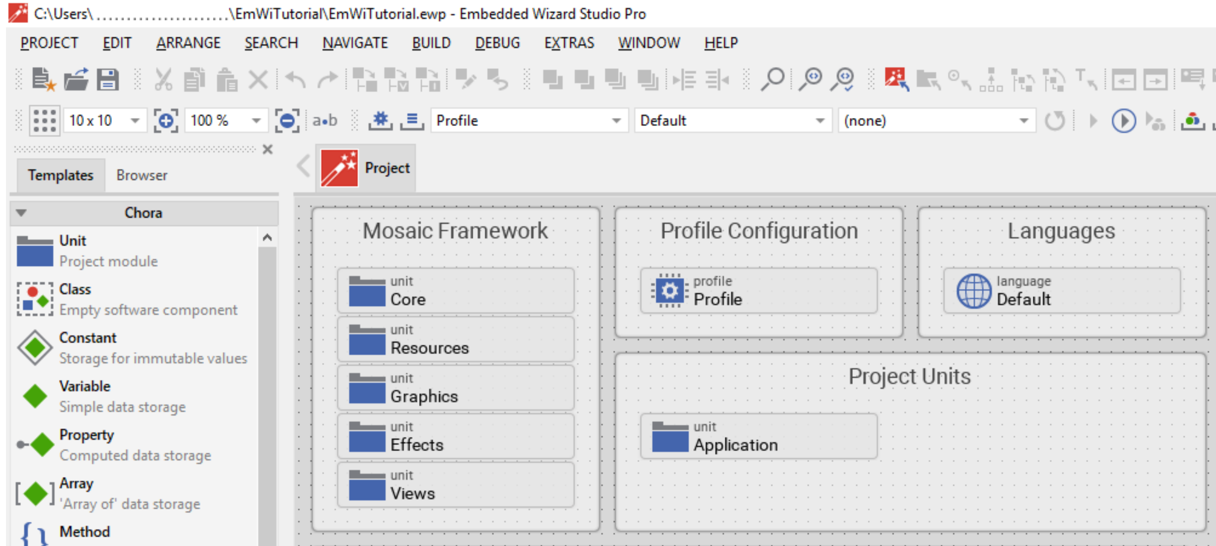


Figure 2.1: Inital project layout

These 4 groups serve totally different purposes:

Mosaic Framework This is embedded wizard's GUI framework used under the hood. You shouldn't be concerned about this right now.

Profile configuration A profile is used to store configuration parameters for your project. By using different profiles, you can customize the project settings to the different target(s) that you ant to use.

Languages The concept of languages is deeply embedded into the system's language *Chora*. If you want to create a multi-lingual app, you can do so by simply adding more languages here.

Project units That's the most interesting part for now, so let's start explaining the concept behind:

2.2.1 Units

Units are a way to structure your project. For now, we will deal with one unit only, the *Application* unit. While we use this unit for now, we will add more units later on. Units are a kind of container for our components used, so have a look now what's inside the *Application* unit:

1. Double click on the icon unit Application, representing the *Application* unit.
2. Inside the composer window, a second tab appears which shows the contents inside the unit which is now opened.

The unit contains one single element only, the *Application* class:

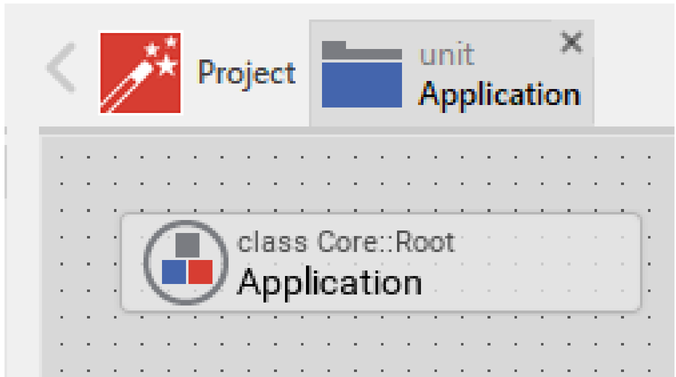


Figure 2.2: Application unit with application class inside

2.2.2 Application class

The *Application* class is the root element, standing at the very top our application. Again let's see what's inside the *Application* class, which is a kind of container for the elements of the class:

1. Double click on the icon

, representing the *Application* class.

2. Inside the composer window, a third tab appears which shows the contents of the class which is now opened:

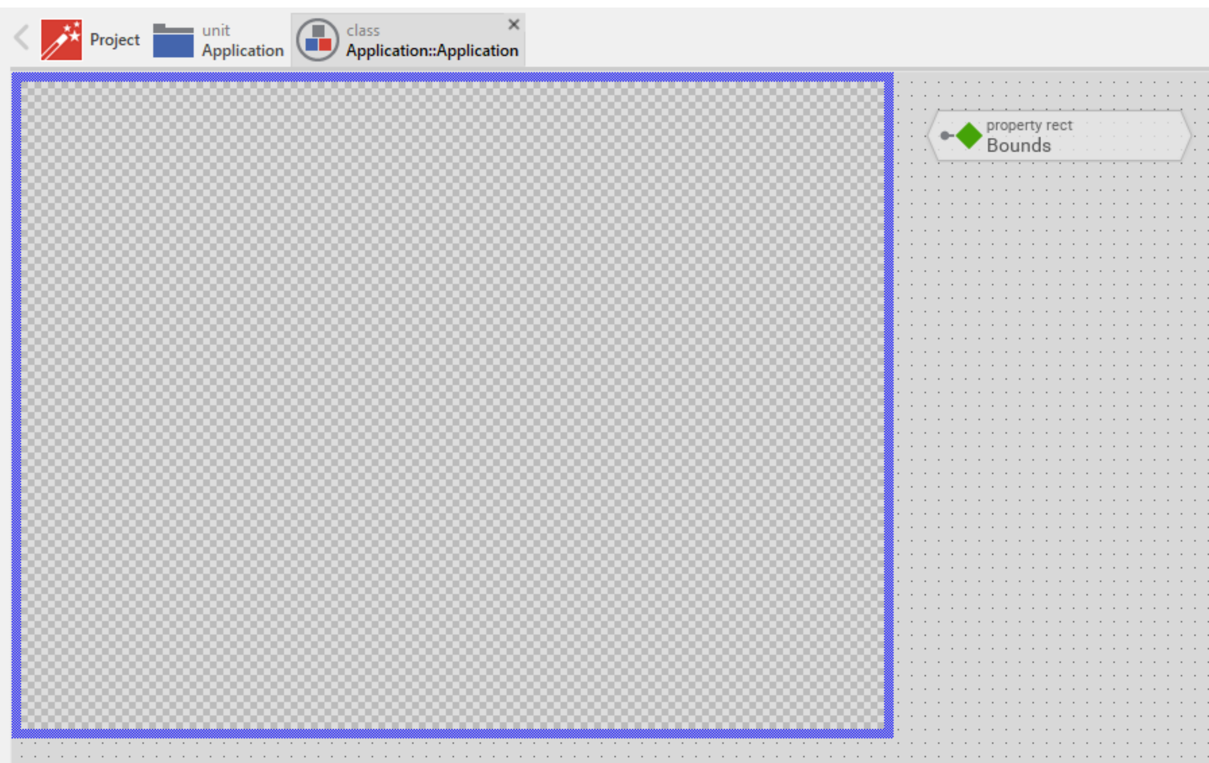


Figure 2.3: Application class with root canvas inside

All you will see here is the transparent root canvas, surrounded by a blue border. That's not much, so let's put some text on the canvas:

- In the gallery templates window, left to the main composer window, make sure that the tab *Templates* is selected.
- In the main area of the templates window, you will find several text item entries. Click on the item *Views*, which will open and show all the *view*-subitems (the items of the templates window follows are arranged in an accordion style layout).
- Identify the item Text, representing a simple text view. Click on the element, drag it over to the root canvas and place it in the middle of the canvas.
- If all went fine, you will see a tiny white text element labelled *Text* in the middle of the canvas.

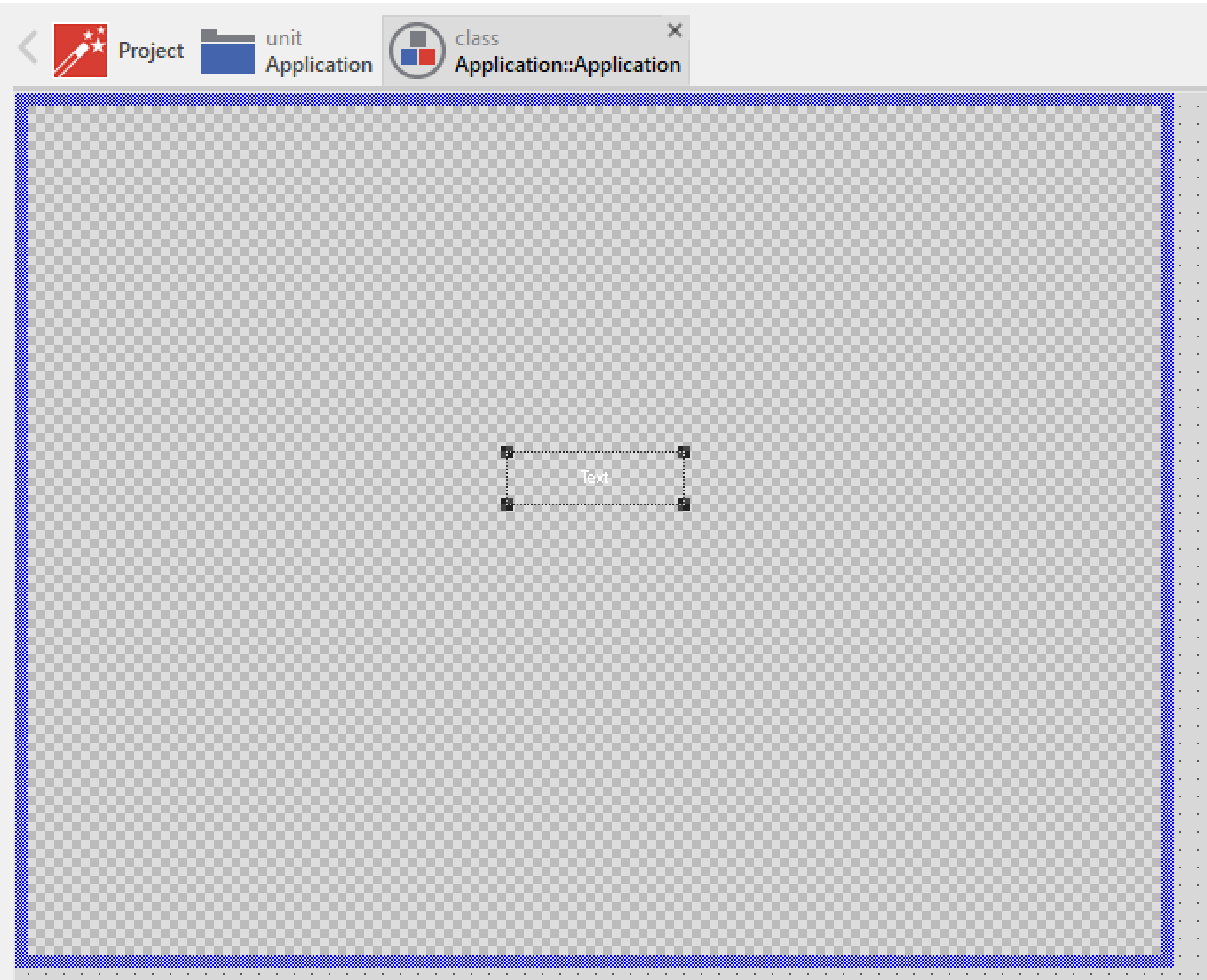


Figure 2.4: Root canvas with inserted text view

So far so good, let's style our text a bit to make it more appealing:

- In the composer window, click on the newly inserted text view to select the element.
- Now have a look at the inspector window right to the main composer window: in the upper *member area* you should see the element named *Text* selected. Also note the attributes and properties area below that shows all properties of the currently selected text view.
- Inside the attributes and properties area, we can adapt the newly inserted text view to our needs:
  - Using the dropdown list, alter the *Font* property of the text element to the value *Resources::FontExtraLarge*.
  - Using the dropdown element, alter the *Color* property of the text element to the value

#000000FF (black, opaque).

- In order to change the display text, alter the *String* property of the text element to the value *"Hello, world!"*.

- Since we do have text overflow in the element now, alter the *AutoSize* property of the text element to the value *true*.

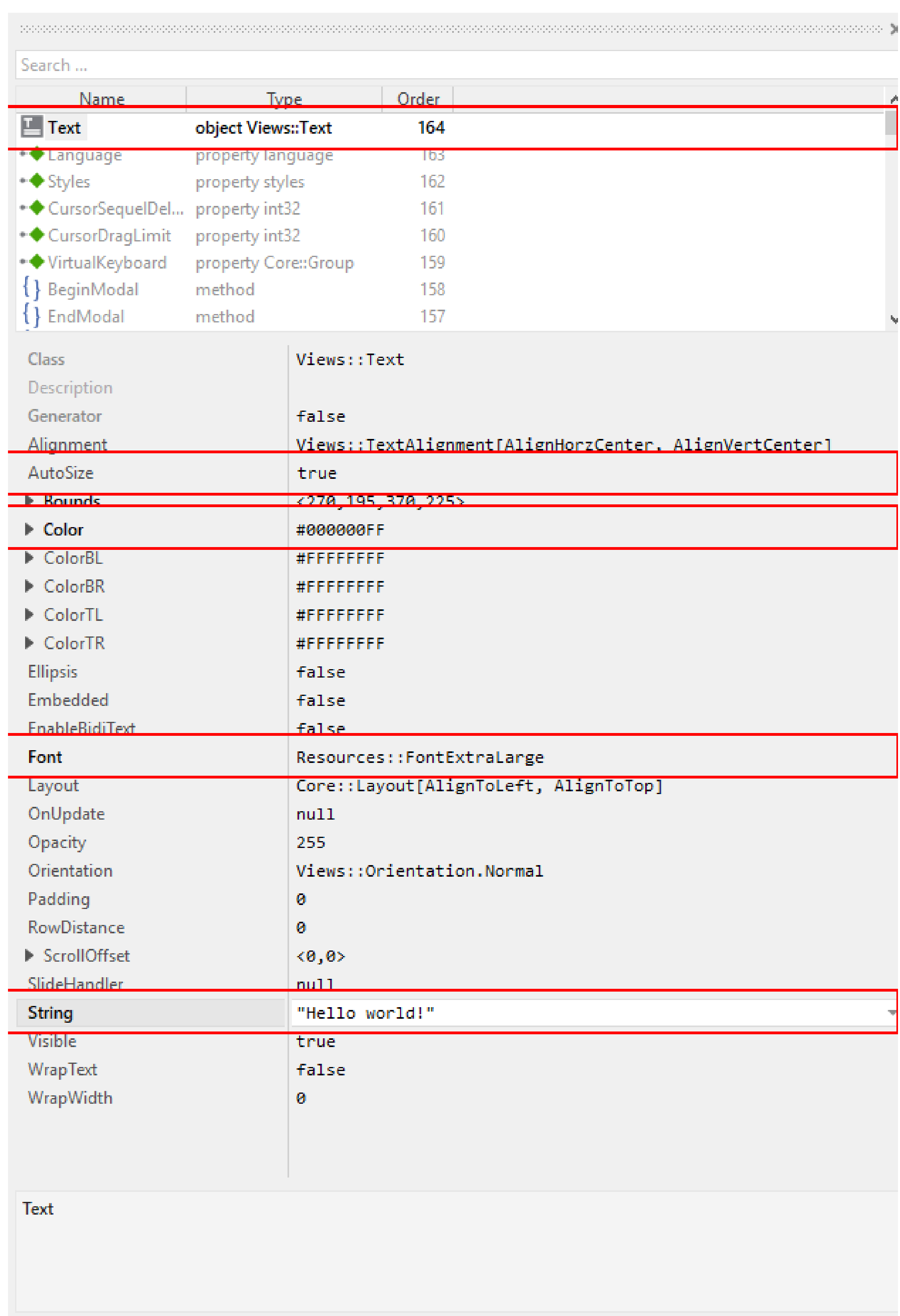


Figure 2.5: Properties area with text view selected

That's it, we do have our message on the screen now!

> **Important**
>
> When typing in the *Hello, world!* text, make sure that the string you typed in is surrounded by double quotes, otherwise an error message will come up.

2.2.3 Running the application

Our *Hello world* application is now ready to run!

There are several ways to launch the app:

- From the main menu, select the menu item BUILD > START PROTOTYPER WITH APPLICATION CLASS, or
- use the keystroke combination [Ctrl+F5], or
- click on the application launch icon in the second row of the toolbar.

Congratulations, you successfully assembled your first application!

Let's move on to the next chapter, there's much more to explore here!

3 Adding your first GUI component

In the last chapter, we successfully built the *Hello world* application. In order to show the text, we placed a text view element directly on the root canvas. While this is a valid approach, it is highly recommended to build reusable components and use these components when assembling your application. This approach has two main benefits:

Testing Using the built-in protoyper, you can test the components in isolation. There is a much higher chance the application works as expected once you assemble it from already tested (sub-)components.

Reusability When using components, it's much easier to follow the DRY principle (**d**on't **r**epeat **y**ourself). You only write a component once and may use it in several places.

So let's start and build our first GUI component: it's a simple graphical unit with a border, a header text, a push button, and a background. Once the button is pressed, some action will occur.

3.1 Adding the empty component itself

- We want to place the component in the *Application* unit, so click on the tab labelled *Application*.

  If this tab is not present yet, click on the *Project* tab and double click on the *Application* unit to invoke the tab. Inside the composer window, you will now see the *Application* root component, the only component currently present.

- Press [Alt+1] to put the focus on the gallery templates window, left to the main composer window. Alternatively, you may mouse click at the tab *Templates* at the very top of the window, to the left.

- Press [C] twice: the first key press opens the ***C**hora* folder, the second key press will then open the invoke folder ***C**omponent templates*, that's what we need. Alternatively, you may mouse click on the header titled *Component templates* inside the main area of the templates window. This will also open the folder and show all available GUI templates.

- Identify the item *Component*, representing an empty GUI component. Click on the element, drag it into the main area and place it underneath the existing *Application* root component.

- We want to use the new component to ring a bell, so we give it a dedicated name: with the component still selected, press [F2] to rename the component. In the inspector window, type in *BellComponent* and press [Enter] once you are done.

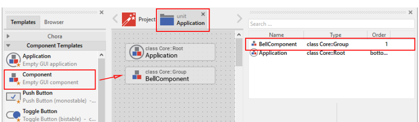
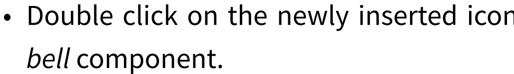


Figure 3.1: Inserting an empty GUI component

Now, we have to open the empty component we just inserted:

- Double click on the newly inserted icon

, representing the *bell* component.



- Inside the composer window, another tab appears which shows the contents of the component class which is now opened:

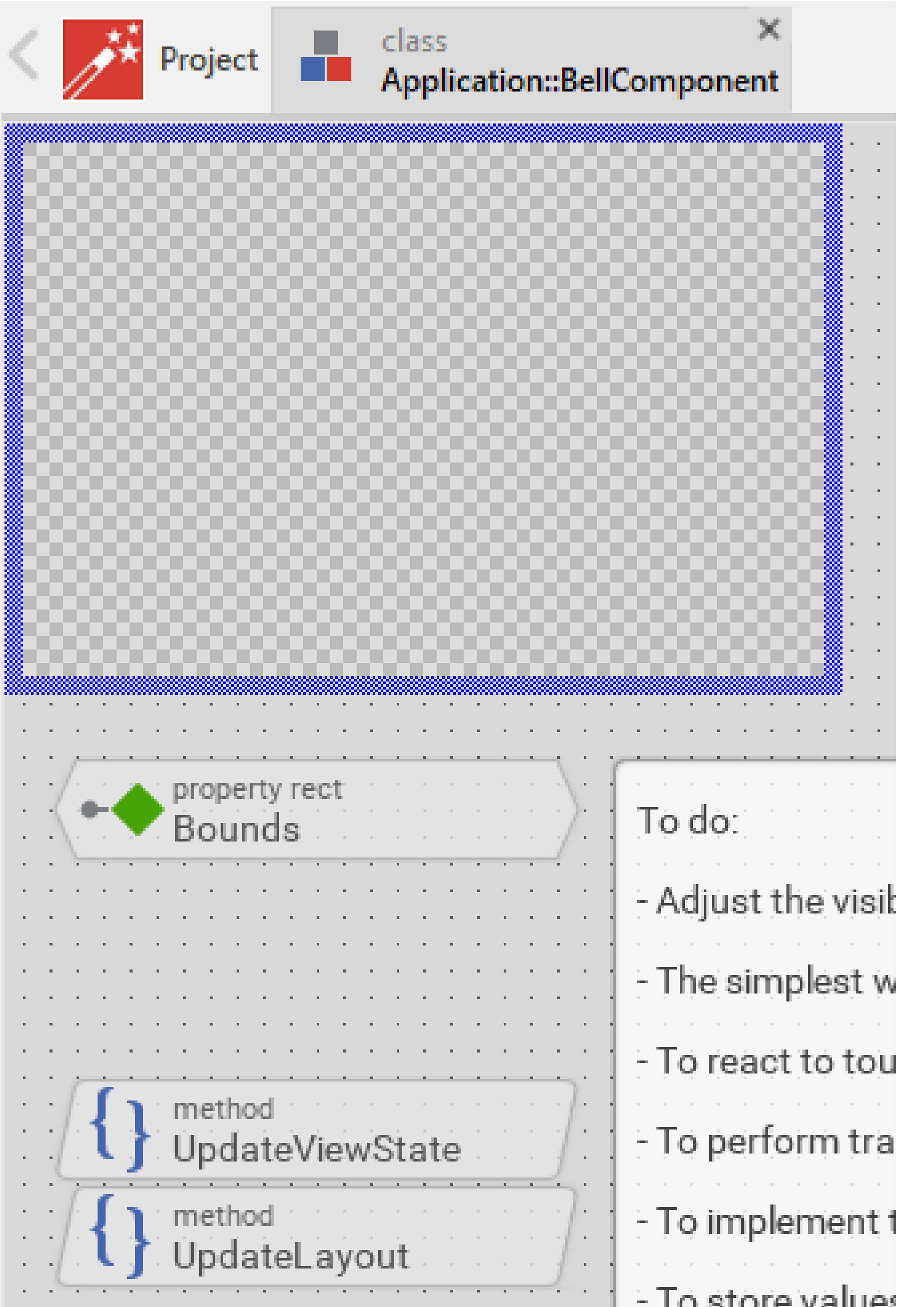


Figure 3.2: Empty GUI component

3.2 Filling the component

Next, we have to fill the empty component:

- First, we do some cleanup: With the [Shift] key pressed, click on the two methods *UpdateView-State* and *UpdateLayout* to select both methods. Press the [Delete] key to delete those methods, we don't need them for now.

- Also, click on the note element that contains a lot of text. Press the [Delete] key to delete this element, too.

- Our component should have a size of 200 × 150 px, so we have to adjust this: click on the property *Bounds* (adorned by a green diamond) to select the element. Now, in the in the upper *member area* of the inspector window right to the main composer window you should see the element named *Bounds* selected. Also note the attributes and properties area below that shows all properties of the currently selected *Bounds* property.

- Inside the attributes and properties area, we can adapt the default bounds values to our needs:

  - Click on the black triangle left to the *Default* element. Multiple lines will show up which hold the values for the origin (*x*, *y*) and the size of the element (***w****idth*, **_h**eight).

  - Alter the *w* instance property of the component to the value *200* to set the default width of the component to 200 px.

  - Alter the *h* instance property of the component to the value *150* to set the default height of the component to 150 px.

  - In order to adapt the size of the component on the screen, click on the *Reload* icon in the second row of the toolbar or press [F7] to reload the class. The blue border of the element will shrink to the new size then.

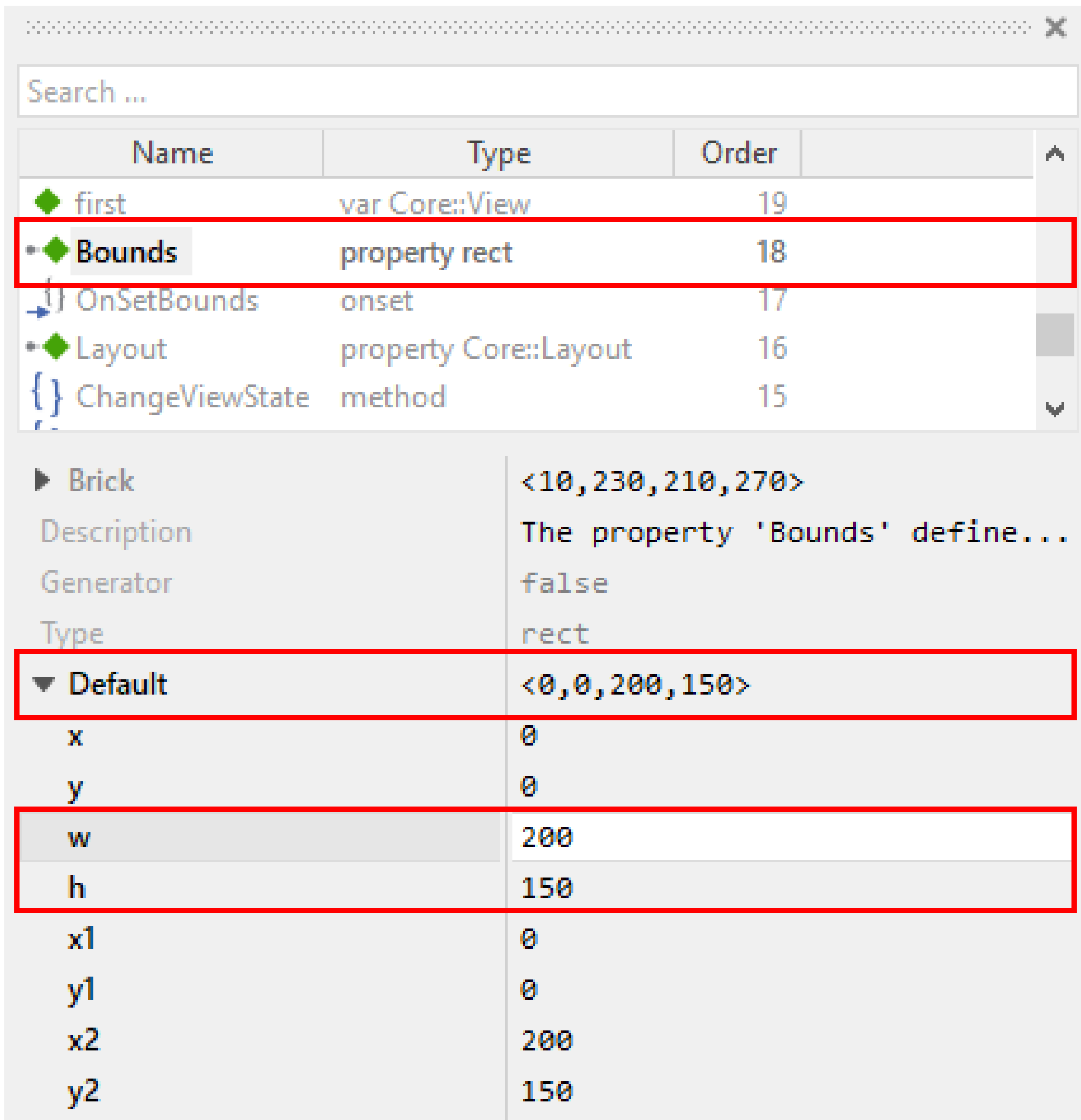


Figure 3.3: Setting the default component size

In a further step, we put all the elements onto the element's canvas:

- With the gallery templates window left to the main composer window focused ([Alt+1]), either click on the folder header *Views* or press key [v], this will open and show all items inside the ***View*** folder.
- Click on the item *Filled Rectangle*, and drag an instance over to the component's canvas. Place the element in the upper left corner of the canvas.

- Press [F2] to rename the component. In the inspector window, type in *Background* and press [Enter] once you are done.
- Adapt the size of the background rectangle to 200 × 150 px. You may do so by either resizing the element with the mouse or by adjusting the property *Bounds* in the in the lower *attributes and properties area* of the inspector window (as described above when setting the default bounds for the component).
- If needed, adjust the color of the background to the value _#FFFFFFFF (white, opaque).

Now we put a border around the component:

- In the gallery templates window, click on the item *Border* and drag an instance over to the component's canvas. Place the element in the upper left corner of the canvas.
- Adapt the size of the border to 200 × 150 px. again, this may be done by either resizing the element with the mouse or by adjusting the property *Bounds* of the component in the lower area of the inspector window.
- Adjust the color of the border to the value *#000000FF* (black, opaque) and set the property *Width* of the border to 1 px.
- You now should see a black border around your component.

Next we add a header text to the component:

- In the *gallery templates* window, click on the item *Text* and drag an instance over to the component's canvas. Place the element centered in the upper area of the component.
- Using [F2] key, rename the name to *HeadingBell*.
- Inside the attributes and properties area, adapt the newly inserted heading text to your needs:
  - Using the dropdown list, alter the *Font* property of the text element to the value *Resources::FontExtraLarge*.
  - Using the dropdown element, alter the *Color* property of the text element to the value *#000000FF* (black, opaque).
  - In order to change the display text, alter the *String* property of the text element to the value *"Bell"*.
  - Since we do have text overflow in the element now, alter the *AutoSize* property of the text element to the value *true*.

Eventually, we add the core element, a push button that will be used to ring the bell:

- In the gallery templates window to the left, either click on the folder header ***W****idgets* or press key [W], this will open and show all items inside the ***w****idgets* folder.

- Click on the *Push Button*, widget and drag an instance over to the component's canvas. Place the element in the lower area of the canvas.

- Press [F2] to rename the component. In the inspector window, type in *PushButtonBell* and press [Enter] once you are done.

- Now customize the appearance of the push button. You may do so by setting the property *Appearance* in the inspector window to *WidgetSet::PushButtonSmall* and by setting the property *Label* to *Ring*.

- You should now see a push button labelled *Ring* in the lower area of the canvas.

- In the search field at the very top of the inspector window, type in *Focus* to look up the property *Focus* of your component. By writing the string *null* into the value input field, set the *Default* value of this property explicitly to *null*. An icon property Core::View Focus *Focus* will appear at the top left corner of the composer window, representing the overridden property. Move this icon to the bottom.

> **Note**
>
> By setting the *Focus* to null, we prevent our component from obtaining the focus. Obtaining the focus changes the component's appearance, which is undesired in our case.

We are finished now with adding elements to our component, and the component should pretty much like shown in figure_title below:

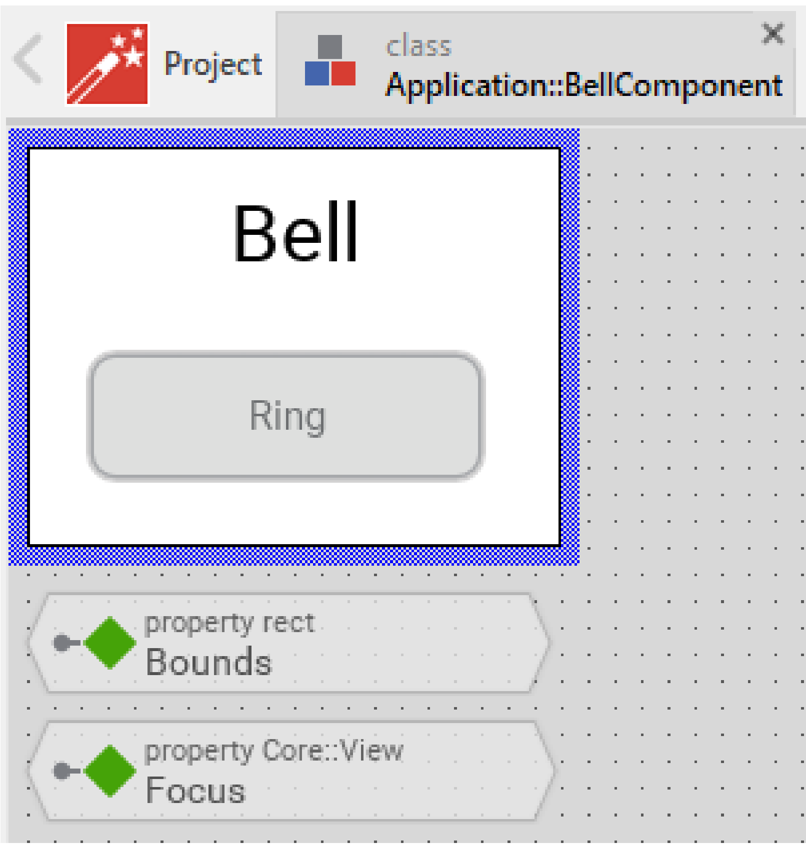


Figure 3.4: Final look of bell component

3.3 Defining a button action performed on click

So far, we successfully added elements the *Bell* component. the only interactive element is the push button, so let's bring life to this component! To do so, we have to add some logic to the component, more specifically some signal handler logic. Embedded Wizard heavily relies on so called slot methods when implementing communication between two objects. Slot methods show the following characteristics:

Code based implementation Every slot method has a method body containing the logic that will be performed once the slot method was called. The programming language used when authoring code inside the method's body is *Chora*:, a relatively unknown, platform independent language which syntax closely resembles C.

Signal based communication between objects In order to invoke a slot method, a signal has to be send to the method. Once the slot method receives the signal the code in the body of the slot method is executed. Since a slot method does not take parameters, signal-based process com-

munication can happen between all kinds of objects, the sender does not have to know about the identity of the receiver object. However, the identity of the sender is passed onto the slot method in the hidden parameter *sender* which can be used inside the body of the slot method.

Inheritance Slot methods are members of class objects. If a class is derived from another class, it inherits all slot methods from this class. As any inherited members, these slot methods can be overridden if needed. You also may call the inherited version ot the slot method by making use of the pseudo method *super()*.

So let's start and build our first slot method to bring life to our push button:

- In the gallery templates window to the left, either click on the folder header ***C**hora* or press key [w] twice, this will open the folder and will present the list of all language elements available in the programming language ***C**hora*.
- To keep our component organized, it's a good idea to place a note group on the canvas first:
  - Click on the element *Annotation Group*, and drag an instance over to the component's canvas. Place the element right beneath the component's canvas.
  - By default, the heading of the note is *This is an annotation*. Change the heading of the note area by changing the property *Caption* in the inspector window to *Slot method(s)*.
- By now we are ready to insert our slot method: Click on the element *Slot Method*, and drag an instance over to the component's canvas. Place the element inside the note rectangle you inserted and adapted in the previous two steps.
- Press [F2] to rename the slot method. In the inspector window, type in *RingTheBellSlot* and press [Enter] once you are done.
- Finally, we have to fill the body of the slot method with some code. To do so, double click on the icon slot RingTheBellSlot representing the slot method *RingTheBellSlot*. In the Code editor, you will now see one single line of Chora code:

```
sender; /* the method is called from the sender object */
```

For now, change this code line to:

```
trace "Sorry, the GUI cannot ring the bell!";
```

The trace is a debugging statement that prints diagnostic output to the log window located in the lower left area of the screen.

We now finished with our slot method now, as soon as a signal will be sent to the method, it will print it's output to the log window. However, we haven't connected our slot method to our push button yet, so let's move on and connect the sender (= push button) with the slot slot method in order to get the push button working!

- To do so, we have to select the push button first. Select it by either clicking on the button object in the composer area or by clicking on the element titled *PushButtonBell*, listed in the upper *member area* of the inspector window to the right.
- With the push button selected, search for the property *OnActivate* in the lower area of the inspector window. The property *OnActivate* refers to a slot method, so as value type in *RingTheBellSlot*. If you want to save typing, click on the small downwards triangle at the right hand side of the value field select the slot method *PushButtonSlot* from the long list offered inside the dropdown area.

You are done with your first component, the layout should look like shown in figure_title below:

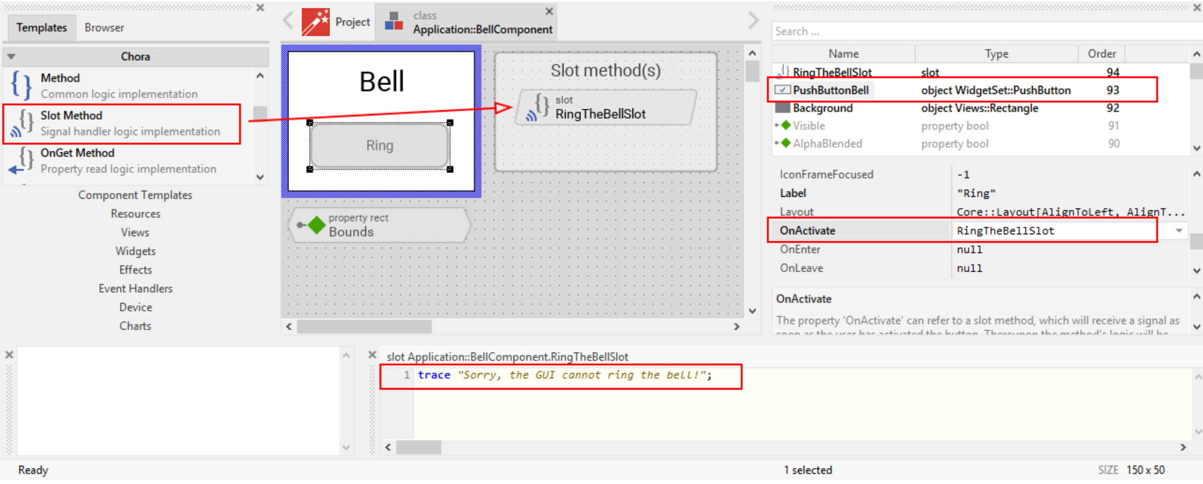


Figure 3.5: Bell component with slot method defined

3.4 Test the component in isolation

Let's go and test our first component! There are several ways to do so:

- From the main menu, select the menu item BUILD > START PROTOTYPER, or
- use the keystroke [F5], or

- click on the launch icon in the second row of the toolbar.

A prototyper window will appear which shows your component and simulate its behaviour: Click on the push button, twice, and two debug messages will appear in the log window:

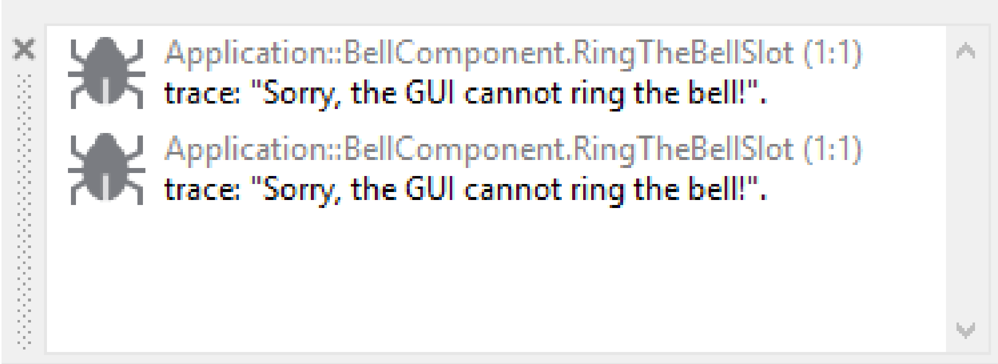


Figure 3.6: Debugging output inside the log window

> **Note**
>
> When we launched the test above, the prototyper acted on a component level, allowing us to test the component in isolation. We do also have the opportunity to launch / prototype the whole application, use [Ctrl+F5] to do so. Have a look at table_title which summarizes the two different prototyping methods.

Table 3.1: Starting the prototyper in different ways

	Prototyping of component	Prototyping of application
Menu	Build > Start prototyper	Build > Start prototyper with application class
Keyboard shortcut	[F5]	[Ctrl+F5]
Toolbar icon		

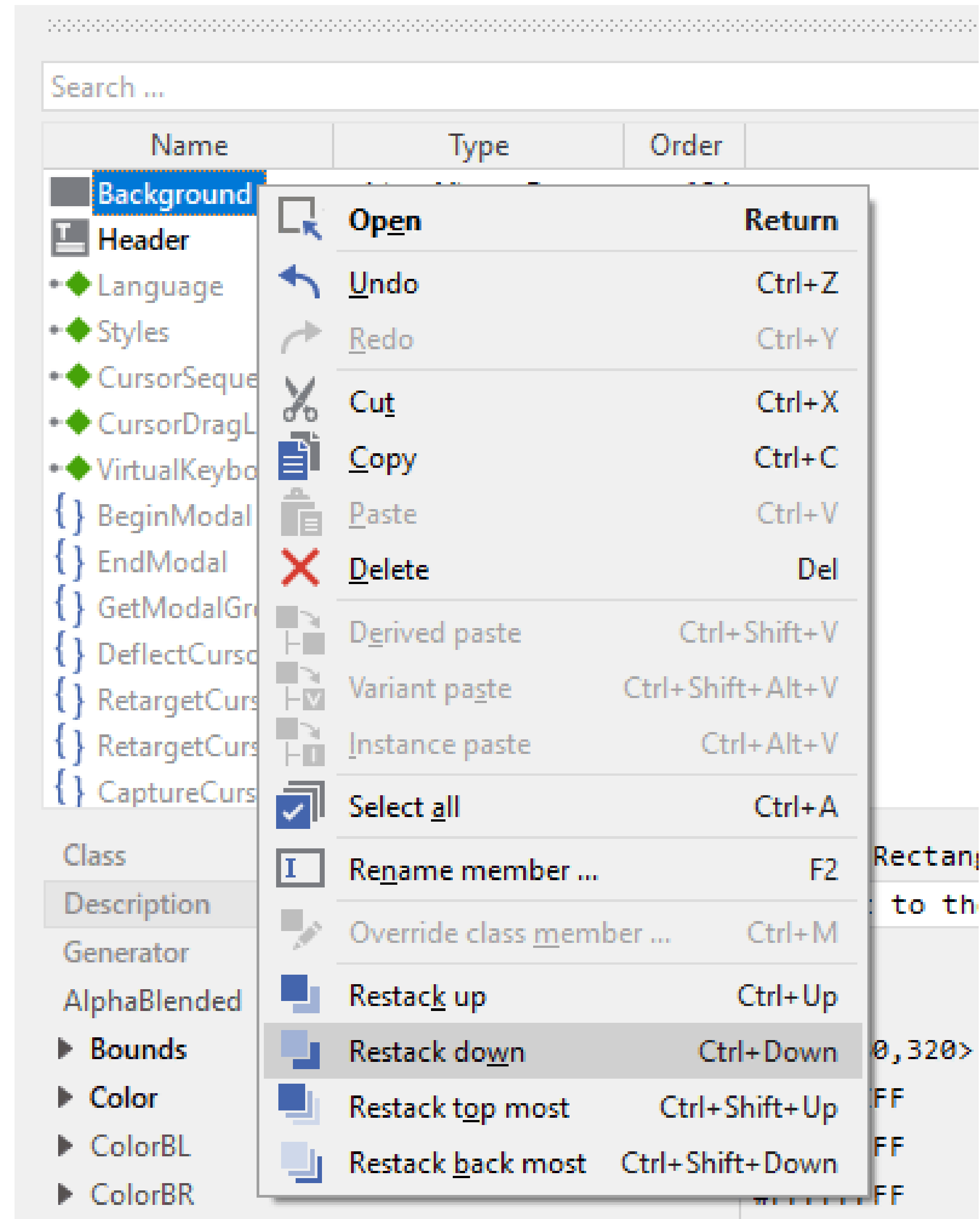
3.5 Add the component to the application's root component

Having first component up and ready is pretty cool, isn't it? Let's move on and integrate the component into the root component, that's what the component is made for!

- Since want to place the component in the *Application* unit, click on the tab labelled *Application*.

  If this tab is not present yet, click on the *Project* tab and double click on the *Application* unit to invoke the tab. Inside the composer window, you should now see the *Application* root component and the *Bell component*, developed by you.
- Rename the root application class to *TutorialApplication* using the [F2] key.
- Double click on the root application class that you just renamed. The root application class will be opened, showing the *Hello world!* text we added in the last chapter.
- Using the inspector window, change the *Hello world!* text to *Tutorial application*.
- Using the *Bounds* property, change the size of the root canvas to 480 × 320 px. If you don't know how to do that, have a look at how we changed the size of the *bell* component above.
- Add a background with the same dimensions of 480 × 320 px to the root canvas. If you don't know how to do that, have a look at how we added a background to the bell component above. Change the color of the background to Gainsborough (*#DCDCDCFF*).

> **Important**
>
> When adding the background onto the canvas, it will be placed in the foreground and will hide your header text. In order to fix that, you have to restack the elements on the canvas.
>
> - Right click on the *Background* element in the inspector window to show its context menu.



- From the context menu, select the menu item *Restack down*.

Tip

When you want to restack an element several levels up or down, select the element and then make use of the shortcuts [Ctrl+Up] or [Ctrl+Down] respectively.

Now we are eventually ready to add our bell component:

- Press [Alt+2] to select the gallery browser window, left to the main composer window. Alternatively, you may mouse click at the second tab *Browser* at the very top of the window.
- The browser's list of classes present is quite long, so we have to narrow down the displayed classes: in the search field immediately below the two tabs, type in *Bell*. While typing have a look at the list and you will notice that the list is getting shorter and shorter. Once you typed in *Bell*, the only class left is the component newly created by you.
- Click on the *Application::BellComponent* class and drag an instance of the class over to the root canvas. Place the component below the header text.

Yeah! You successfully included your component into the main app!

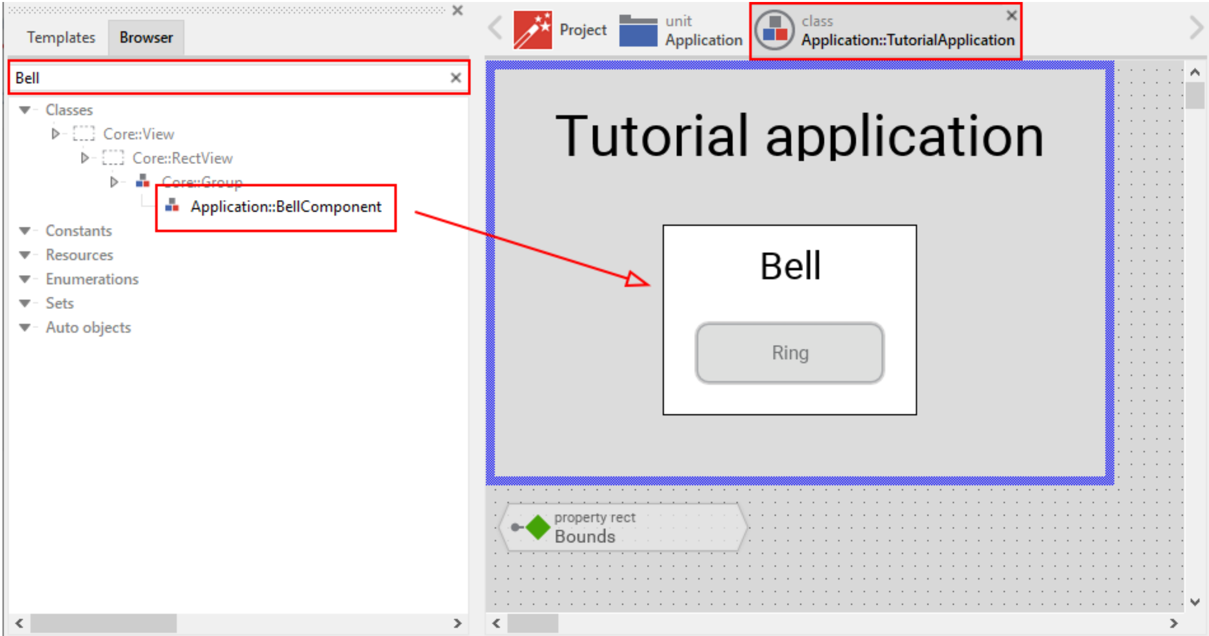


Figure 3.7: Tutorial application with bell component

Let's test it out:

- From the main menu, select the menu item BUILD > START PROTOTYPER with application class], or
- use the keystroke combination [Ctrl+F5], or
- click on application launch icon in the second row of the toolbar.

The application will start up. You will notice that the screen size is larger than the root element we put on it. Let's fix this:

- Click on the *Project* tab (the first tab from the left) and click on the *Profile* item, located inside the note group *Profile configuration*.

- Using the inspector window, change the property *ScreenSize* to *<480,320>*.

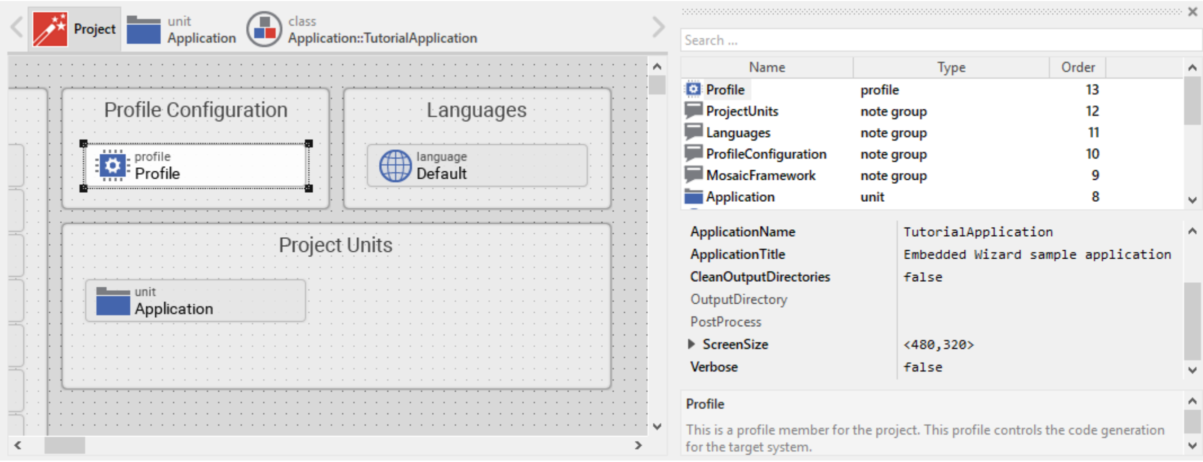


Figure 3.8: Adapting the screen size

Hooray, it we have our first application up and running:

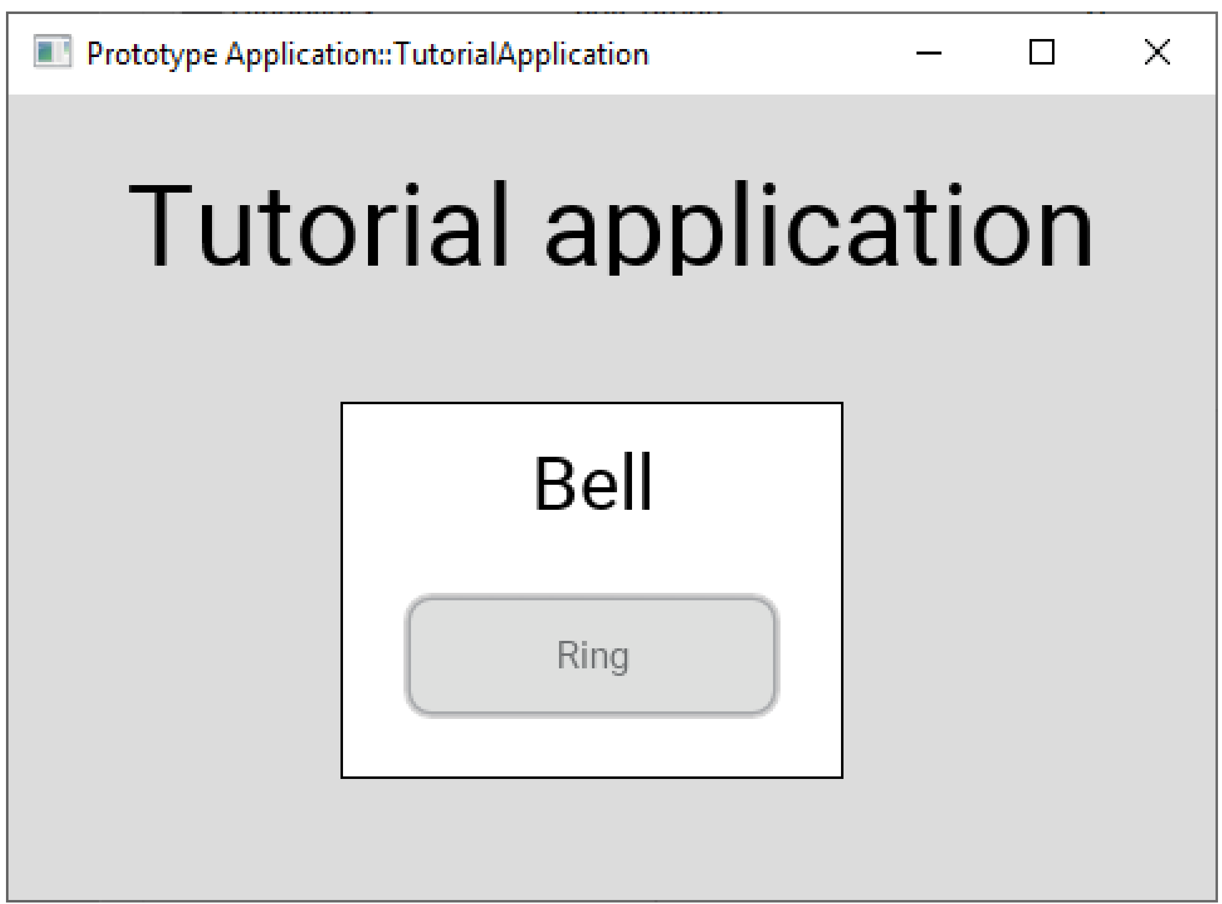


Figure 3.9: First application running

Let's move on to the next chapter, there's still much more to explore!

4 Making your component truly reusable

In the last chapter, we successfully built your first GUI component and put it on to the root canvas. As we saw, this allows you to test your component in isolation, which is a great benefit, especially when working on large scale projects. The second motivation to use components is reusability. Let's think about this a bit more: currently our bell component is tailored to ring a bell. This somehow limits the reuse of the component. Of course you can put two or more instances of the bell component on the root canvas, but it's unlikely that anyone wants to have two ore more buttons on the screen performing exactly the same action: ringing a bell. In a real world application you often have several buttons, which all do perform *different* actions, however. With the components in its current form, implementing this is not possible right now. So let's extend the component and make it truly reusable in the next chapter!

4.1 Extending your component with properties

Let's have a look at our component first:

- Open the the *Application* unit by clicking on the tab labelled *Application*. If this tab is not present yet, click on the *Project* tab and double click on the *Application* unit to invoke the tab.
- We want to refactor the *BellComponent* in order to facilitate reuse of the component. This should be reflected in the name of the component, so go ahead and rename the component to *PushButtonComponent* using the [F2] key.
- Double click on the renamed *PushButtonComponent* element. Its contents will be show in a new tab titled with the class name *Application::PushButtonComponent*.

Let us reflect on the current design of the component a bit: currently there are three items hard-coded, which severely limits the reusability of this component:

- the text shown in the header text view (hard-coded value: *Bell*),
- the text of the label on the push button (hard-coded value: *Ring*) and

- the slot method attached to the property *OnActivate* of the push button (hard-coded value: *RingTheBellSlot*).

In order to allow reuse, we have to extend the component so that the three items listed above can be stored inside the component. That's what properties are made for. If you do have a Java or C# background, you already should be familiar with the concept of properties:

- Properties are a kind of variable where data of an arbitrary Chora data type can be stored (e.g. string, int, slot, …). For each property, you have to specify the data type it can held (including its default value).
- A property represents a more sophisticated variable in that sense that a property does have *OnSet* and *OnGet* methods that are used to get and set the value of the property. Normally, these methods contain boilerplate code that set or gets the internal memory of the property. You are encouraged to add your custom code to these method(s) to tailor them to your needs. We will do some shortly, so hold on!
- You are allowed to attach slot methods as observer to any property we implement. As soon as the value of the property changes, the slot method (= observer) gets notified about the change. This is a core feature of Chora that allows the development of applications following the MVC pattern. We may talk about this later on.
- Properties cannot only store arbitrary data type, they can also store references to any data type. Properties that store references are so called outlet properties. This is an advanced concept, we may talk about that later on.

That's the theory behind property in short, let's start and put theory into practice:

First we have to make a few adjustments to our components in order to reflect the refactoring of the component:

- In the *PushButtonComponent* tab, rename the existing push button *PushButtonBell* to *PushButton* . Change the value of the property *Label* of this component from *"Ring"* to *"Action"*.
- Afterwards, rename the header text element *HeadingBell* to *Header* and change the value of the property *Label* of this component from *"Ring"* to *"Action"*.
- Delete both the existing slot method *RingTheBellSlot* and the corresponding note group *Slot method(s)* using the [Del] key. When using the refactored component, slot methods will be added on the root application level and not at the component level any more.
- Remove the current value *RingTheBellSLot* from the the property *OnActivate* of the PushButton, otherwise you will run in trouble later on. To do so, select the *PushButton* component and right click on its property *OnActivate*. From the context menu select the menu item *Restore default value* ([Ctrl+R]), and the current entry will be replace with the default *null* value.

Let's get our feet wet and add some properties to our component:

- From the gallery templates window ([Alt+1]) with folder ***C**hora* opened (key [C]), drag an instance of the *Annotation Group* to the component's canvas. Place the element right beneath the component's canvas and rename the property *Caption* of the element to *Properties* using the inspector window.
- Now we are ready to insert our properties: Click on the element *Property*, and drag an instance over to the component's canvas. Place the element inside the note rectangle you inserted and adapted in the previous step.

As you can see, a property named *Property* was inserted, together with its *OnSet* and *OnGet* methods:

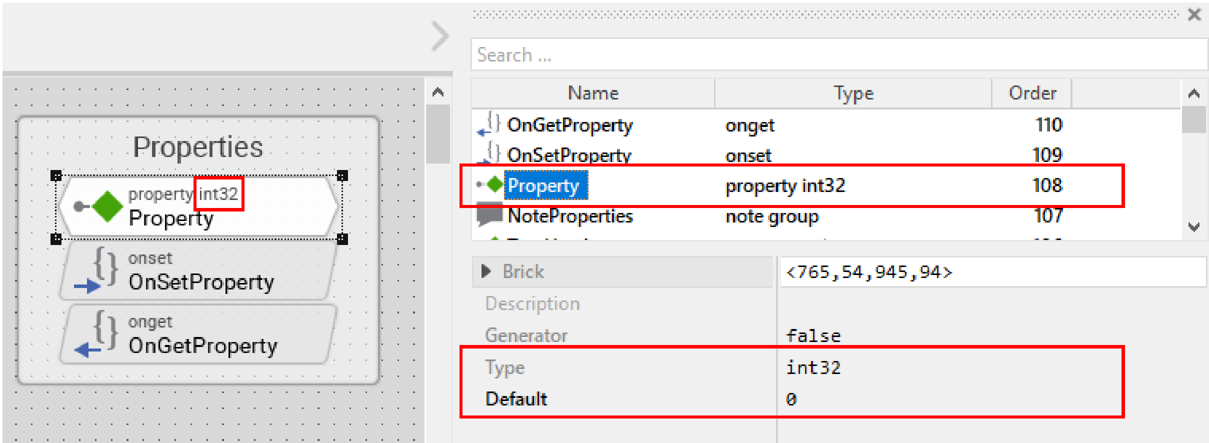


Figure 4.1: Inserting our first property

Let's move on and adapt the property to our needs:

- The property should be used to store the text of the header element of our component, therefore we rename the property to *TextHeader* using the [F2]. Please note that the names of the *OnSet* and *OnGet* methods automatically change to *OnSetTextHeader* and *OnGetTextHeader* respectively.
- Currently, the data type of the property is *int32*, that's not what we want, instead we want to store a string value (= the header text) inside the property. To be able to do so, we change the value of the property *Type* to *string* inside the inspector window.
- The header text of our component should be *Header* by default, so we change the value *Default* to *"Header"* inside the inspector window. Don't forget the surrounding double quotes here or you may run in trouble.

The property is now set up to store the header text value. Currently, when setting the header text property, the change of the property value is not reflected inside the component. To overcome this, we have to add some code to the OnSet method which is called each time a new value is assigned to the property.

- Double click on the method *OnSetTextHeader* of the property. In the code editor window, you will now see some lines of boilerplate Chora code:

```
// The value doesn't change - nothing to do.
if ( pure Property == value )
  return;

// Remember the property's new value.
pure Property = value;

// TO DO:
//
// Now you can handle the alternation of the property.
```

Replace the *TO DO:* section at the bottom with two lines of custom code:

```
// The value doesn't change - nothing to do.
if ( pure Property == value )
  return;

// Remember the property's new value.
pure Property = value;

// change the text of the header
Heading.String = value;
```

Our new line of code assigns the property *String* of the *Heading* element (*Heading.String*) the new value the property was set to, this is immediately reflected in the GUI. That's all we have to do! Now, as soon as the property gets a new value assigned, the header text changes, too.

The first property is ready to go, so add two more properties:

- From the gallery templates window drag another *Property* to the component's canvas.
- Rename the property to *LabelButton* using the [F2] key.

- Change the type of the property to *string*, with a default value *"Label"*.
- In the body of the *OnSetLabelButton* method, replace the *TO DO:* section with the code line `PushButton.Label = value;`.

This way, any change of the property *LabelButton* will immediately change the label text of the button. So far so good. Now we have to take care that not only the label and heading text can be set, but also the action performed once the button is clicked:

- From the gallery templates window drag another *Property* to the component's canvas.
- Rename the property to *ActionButton* using the [F2] key.
- Change the type of the property to *slot*, with a default value *null*.
- In the body of the *OnSetActionButton* method, replace the *TO DO:* section with the code line `PushButton.OnActivate = value;`.

The refactoring of our component is done, it should now look like this:

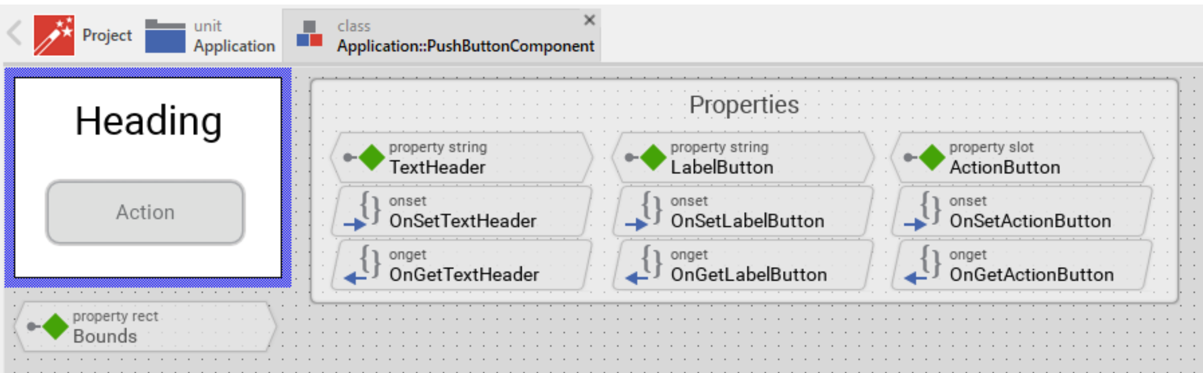


Figure 4.2: Refactored component

4.2 Adapt and fix the main application

Now that refactoring our component is done, we have to make some changes in the main application to make the application work again:

- Bring up the contents of the main application in the tab *Application::TutorialApplication*.
- The refactored pushbutton component now shows *Header* as header text and *Label* as button label. These are the default values of the properties we just introduced to the component. Let's customize the component's properties, that's why we introduced them in our component:

- Using the inspector window, change the property of the push button component *TextHeader* text to *Bell*.
- Using the inspector window, change the property *LabelButton* text to *Ring*.

The appearance of our component now again looks as wanted. When pressing the button, nothing happens yet. Let's fix that, too:

- From the gallery templates window to the left, drag an element *Annotation Group*, over to root canvas. Rename the group to *Slot method(s)*.
- Add a new slot method inside the note rectangle. Rename the slot method to *RingTheBellSlot*.
- Fill the body of the slot method with the code line `trace "Sorry, the GUI cannot ring the bell!";`.
- Using the inspector window, change the property *ActionButton* of the bell push button component to the newly created *RingTheBellSlot*.

That's it! Test the main application in the prototyper ([Ctrl+F5]), and the main app should behave exactly as prior to the refactoring.

4.3 Adding a second component (weather forecast)

If you are asking yourself why we did the refactoring, things are getting clear hopefully as soon as we insert a second instance of the component. The GUI allows ringing the bell of your device already. Imagine your device is able to present the weather forecast to you. Maybe your device has a screen display for that purpose, or it has a speaker to read out the forecast loud. Let's extend the GUI with a second push button component for presenting the weather forecast to you:

- Press [Alt+2] to select the gallery browser window, left to the main composer window. Alternatively, you may mouse click at the second tab *Browser* at the very top of the window.
- In the search field immediately below the two tabs, type in *Push* to shorten the class list.
- Click on the *Application::PushButtonComponent* class and drag a second instance of the class over to the root canvas. Rearrange the two push button components so that both of them fit on the screen.

Yeah! You successfully included a second push button component into the main app. Let's move on and customize the newly inserted component!

- If not already select, select the newly inserted component first:
- Using the inspector window, change the property *TextHeader* of the new push button component to the text string *"Forecast"*.
- Using the inspector window, change the property *LabelButton* of the same component to *"Show"*.
- Add a new slot method inside the note rectangle titled *Slot methods*. Rename the slot method to *ForecastSlot*.
- Fill the body of the slot method with the code line `trace "Sorry, but the GUI cannot tell the weather forecast";`.
- Using the inspector window, change the property *ActionButton* of the new push button component to the newly created *ForecastSlot*.

The extended version of the application with two push button components should now look like in figure_title below:

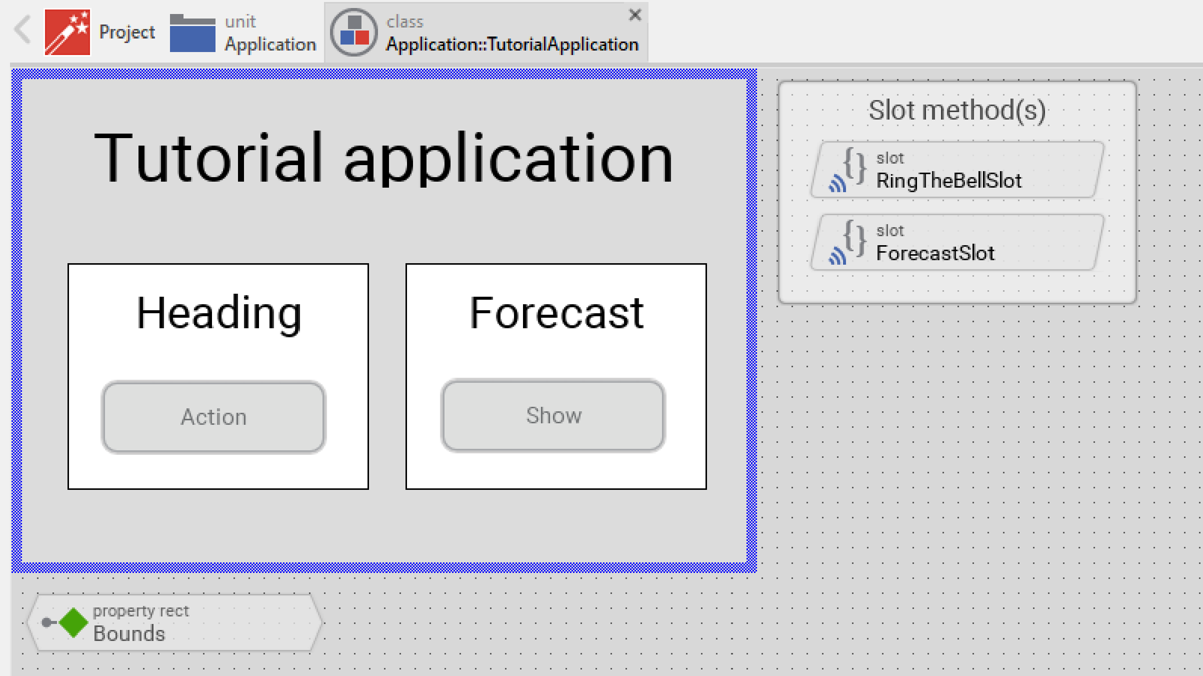


Figure 4.3: Extended tutorial application

Go ahead and test your application! You should see different debugging output depending on the button pressed.

This chapter has come to an end, time to recap: By adding three properties to our component we

managed to create a truly reusable component. Creating reusable components comes has its price, however, this will certainly pay off once your project grows over time.

Let's move on to the next chapter, there's still much more to explore!

5 Device integration: Hello world from our device

In the last chapter, we made our component truly reusable by adding properties to it. Afterwards we completed our GUI which now contains two units, one for ringing a bell and another for showing the weather forecast. Unfortunately, there's no real function behind the push buttons of both units yet. We will address this in the following chapter.

5.1 GUI builder and platform packages

Embedded Wizard is a GUI framework builder that allows you to build GUI platform independent applications that can deployed on many different target systems. This is achieved by using the programming language Chora for all programming tasks related to the GUI. Other than that, Embedded Wizards strictly refrains from any access to the underlying device to ensure platform independence. However, when building embedded devices, the main purpose of the GUI application is to enable and facilitate interaction with the underlying hardware. So how does Embedded Wizard integrate with those devices? That's where the concept of platform packages comes into play. Embedded Wizard offers platform packages for many different hardware platforms (STM, NXP, TI, Raspberry Pi, …). A platform package consists of a code generator, a resource converter, a graphics engine and a runtime environment for the specific platform. You may think of the platform package as a link between the GUI and the hardware. Applications implemented in the programming language Chora can be run on any particular platform if a platform package exists for that platform.

5.2 Adapting the project structure

As explained above, there is a strict separation between the GUI application and the device, represented by one or more platform packages. Let's start and reflect that separation in our application structure, too:

- Click on the *Project* tab (the first tab from the left).

- Identify the note frame that holds the icon unit Application that represents the application unit. Rename the heading text of this frame from *Project Units* to *GUI project*. To do so, change the property *Caption* of the note frame using the inspector window.
- From the gallery templates window to the left, drag an element *Annotation Group* over to root canvas. Rename the group to *Middleware*.
- Drag an element *Unit* over to root canvas, add the new unit inside the note rectangle you inserted in the previous step. Rename the unit to *Device*.
- Drag another *Unit* into the same note rectangle *Middleware*. Rename the unit to *BrowserDevice*.
- Drag a third *Unit* into the note rectangle *Middleware*. Rename the unit to *TargetDevice*.

The structure of your project should now look as shown in figure_title below:

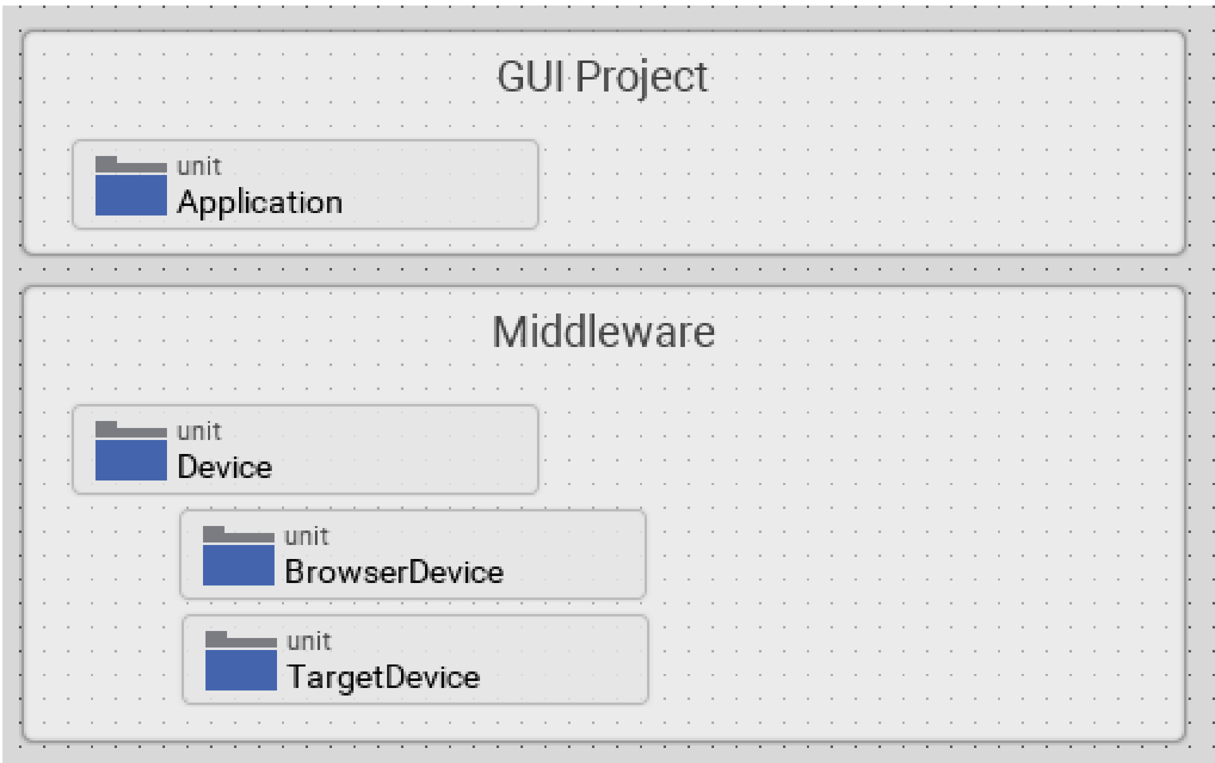


Figure 5.1: Extended project layout

5.3 Adding a interface device class

Let's add content to the newly inserted units! We start with adding a device class interface first:

- Double click on the icon, representing the *Device* unit. This will open the unit in a new tab.
- In the gallery templates window left to the main composer window ([Alt+1]), either click on the folder header ***D****evice* or press key [D], this will open the folder and will present all ***d****evice* subitems.
- Click on the item *Device interface*, and drag an instance over to the component's canvas. Place the element in the upper left corner of the canvas.

By dragging over the *device interface* to the canvas, two new objects were inserted:

- the *DeviceClass* element to the left, which represents the class where we will store device related class members like commands, properties and so on, and
- the *Device* autoobject element associated to the *DeviceClass* element. This autoobject represents the globally available instance of the device class. This autoobject represents Embedded Wizard's implementation of the singleton pattern (if you are familiar with Java, you may think of the *Device* class as a static class). Using the *Device* autoobject, any GUI element has direct access to the device class and its members, which is very handy. We will use this autoobject soon.

Your screen should now look like illustrated in figure_title below:

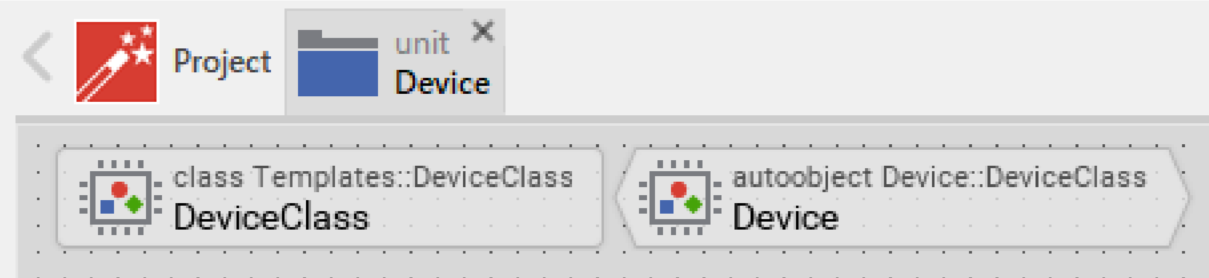


Figure 5.2: Device class interface

Let's go ahead and review and adapt the members of the newly inserted device class interface:

- Double click on the icon

, representing the *Device* autoobject. This will open the *Device* class in a new tab. You will see that the canvas was prepopulated with several class members already.
- Identify the note group that holds the icon representing the *Command* method and rename the caption of this note group from *Example of an interface to perform an operation in the device* to *Command(s)*.
- Rename icon representing the *Command* method to *RingTheBellCommand*.
- Double click on the renamed command. In the code editor window, you will see the method's signature, followed by many lines of template Chora code.
- Have a look at the method signature of the *RingTheBell* command.

Device commands are represented by regular methods. Like in all programming languages, a method can take parameters and can have a return value. Have a look at the signature of the *RingTheBellCommand* method shown at the top of the code editor:

```
method int32 Device::DeviceClass.RingTheBellCommand( arg int32 aParameter1, arg bo
```

As you can see the method currently takes an int32 value as first argument and a boolean value as second argument. Also, the method returns an int32 value. These settings are not what we want, our simple *RingTheBellCommand* method does not need any parameters and won't return anything, so the return type should be *void*. Let's go ahead and adjust the method's signature to our needs:

- In the top title line of the code editor containing the method signature, you can see a small downwards triangle. Click on this triangle to show a frame where the method's return value and parameters are listed in separate lines.
- Right click on the first method parameter *arg int32 aParameter1* to invoke the context menu on this parameter. From this menu select the menu item *Delete* to remove the first parameter.
- Right click on the remaining method parameter *arg bool aParameter2*. From the context menu shown, select *Delete* to remove this parameter, too.
- Right click on the first line that shows the method's name *method int32 RingTheBellCommand*. From the context menu shown, select *Edit* and change the return parameter from *int32* to *void*.

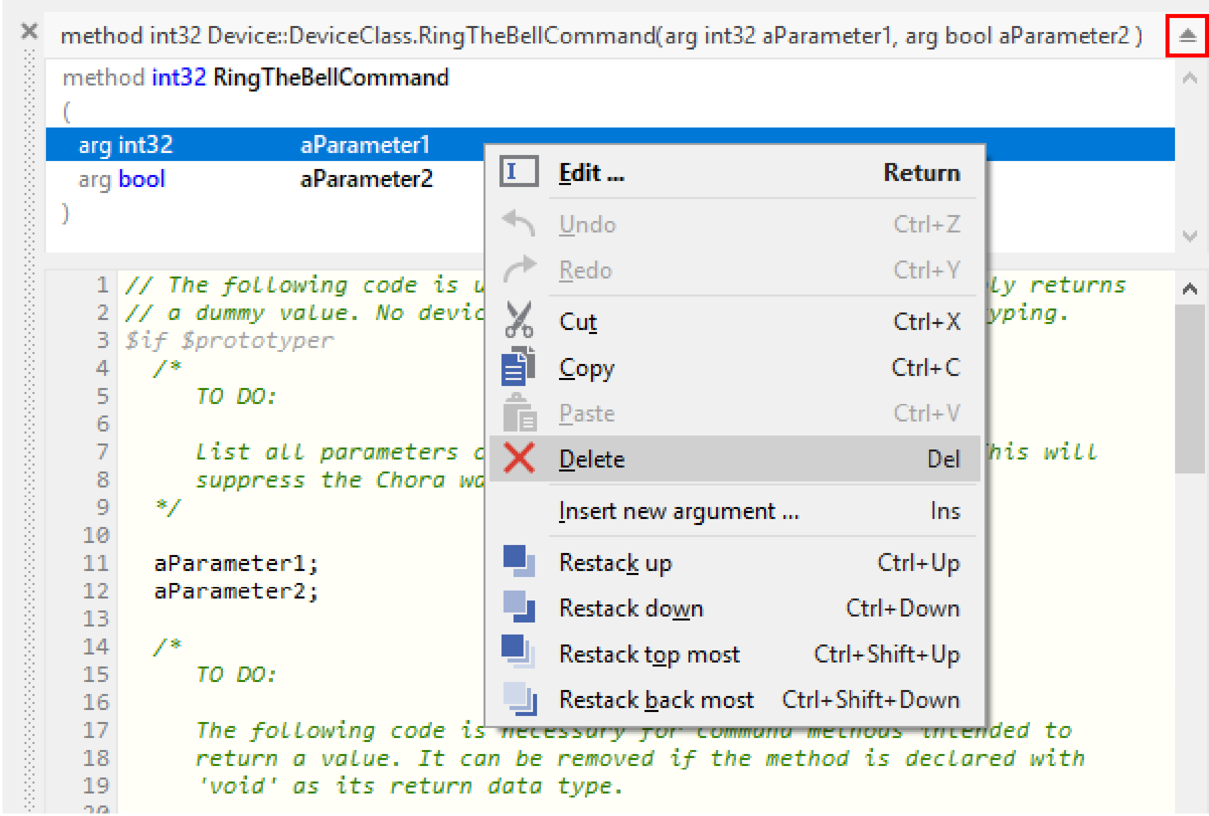


Figure 5.3: Delete methods parameters

Once we adjusted the signature of the method, let us adjust the body of the method, too. Remove all template code and put in one single line:

```
trace "The device class of the GUI is an interface only and cannot run any device comm
```

As said, Embedded Wizard does not have access to the underlying device and therefore cannot advise the device to say *Hello* to us. We have to implement this in a derived class, we will do so shortly.

Since we have set up a device and a command now, let's use it and wire the push button action of the bell to that newly created command:

- Bring up the contents of the main application inside the tab *Application::TutorialApplication*.
- Double click on the icon slot RingTheBellSlot representing the slot method *RingTheBellSlot*. Inside the code editor, you will see the line `trace "Sorry, the GUI cannot ring the bell!";`.

Change this code to

```
Device::Device.RingTheBellCommand();
```

> **Note**
>
> Embedded Wizard code editor ships with integrated code completion, which is very handy and might prevent you from typos when authoring code inside the code editor. To test it out, simply write *Device::* into the editor and should see a list of available completions to the given *Device* unit name you just typed in.

That's how we call a method by code: specify the class name (*Device::DeviceClass*) first, then append the method name (*RingTheBellCommand*), prepended with a dot and terminated with empty parentheses. Now run your code using the prototyper, and you should see a trace message informing you that the GUI cannot run any device commands. Obviously, we are not at the end, so read on!

5.4 Adding another profile

As already explained above, there is a strict separation between the GUI application and the device, represented by one or more platform packages. Two platform packages are included in the Embedded Wizard installer and are available out of the box:

- the *Tara.Win32.xxx* platform package. This is the default platform package that allows you to run the application on your Windows platform. You were using it already when you launched the prototyper to run your application or component (*xxx* stand for one of the available color formats, either Index8 or RGBxxxxx).
- the *Tara.WebGL.RGBA8888* platform. This WebGL/Javascript platform package allows you to run the GUI in any WebGL enabled browser. That's especially handy for this tutorial since you don't need any hardware to follow the instruction given.

> **Note**
>
> Besides the Win32 and the WebGL packages there are many platform packages available to target real hardware (STM, NXP, TI, Raspberry Pi, …). For each of these platform packages, a separate installer exists. You have to obtain and run this installer to make the associated platform packages available inside Embedded Wizard.

If we want to make use of more than one platform package inside our project, we have to have an associated **profile** on the Projects tab for each package you would like to use. So let's add another

profile that allows us to switch between the Win32 package and the WebGL package. We then use the latter package to output *Hello, world!* on the browser device, more specifically on the web console of the browser. The journey goes on ...

- Click on the *Project* tab (the first tab from the left).
- Identify the note frame with the caption *Profile configuration*, it only contains the icon profile Profile representing the default profile. Select this profile and have a look at the inspector window. You will realize that the attribute *PlatformPackage* of the profile has the value *Tara.Win32.RGBA8888* assigned. To reflect this, rename the profile from *Profile* to *Win32* using the [F2] key.
- From the gallery templates window to the left, drag the element *Profile* inside the *Chora* folder over to the canvas and place it underneath the existing profile *Win32* . Rename the profile to *Browser*. Resize the note frame and rearrange the elements on the canvas so that the layout looks nice again.
- Our new profile should be associated with the WebGL platform package, so inside the inspector window, change the value of the attribute *PlatformPackage* from *Tara.Win32.RGBA8888* to *Tara.WebGL.RGBA8888*.
- Inside the inspector window, change the value of the attribute *ScreenSize* to *<480,320>*.
- Also change the value of the attribute *OutputDirectory* to *../WebGL*. This defines the directory where all the code for our website will be stored once we build the project.
- Optionally, you may fill the attributes *ApplicationName* and *ApplicationTitle* with the values *TutorialApplication* or *"Embedded Wizard sample application"*, respectively. For the last value, don't forget the surrounding double quotes here or you may run in trouble.

The *Profile* section of your project should now look as shown in figure_title below:

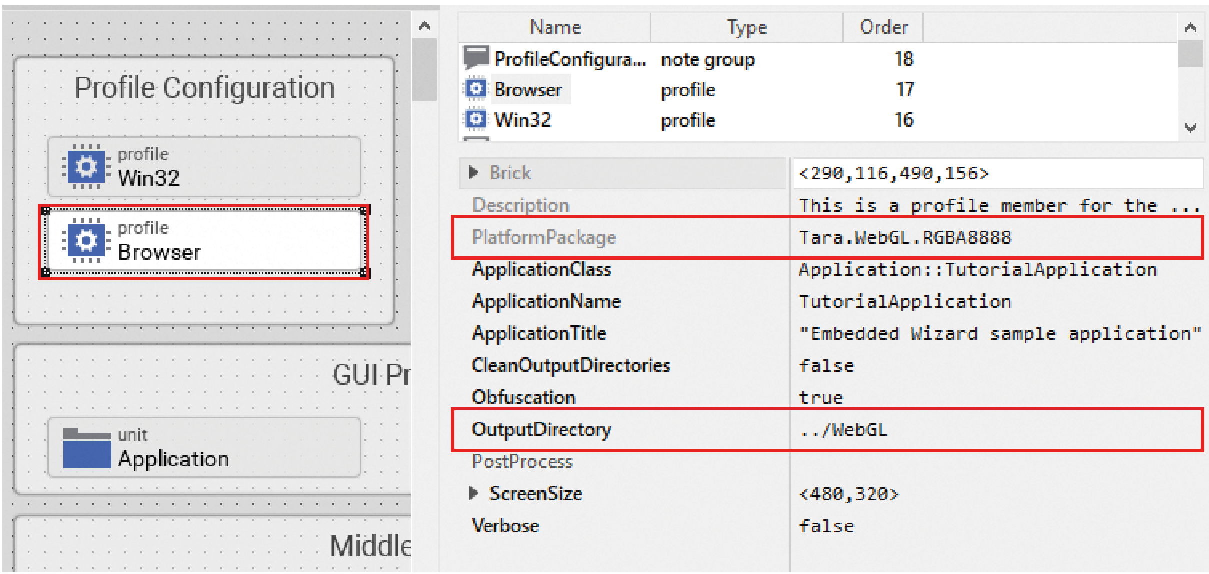
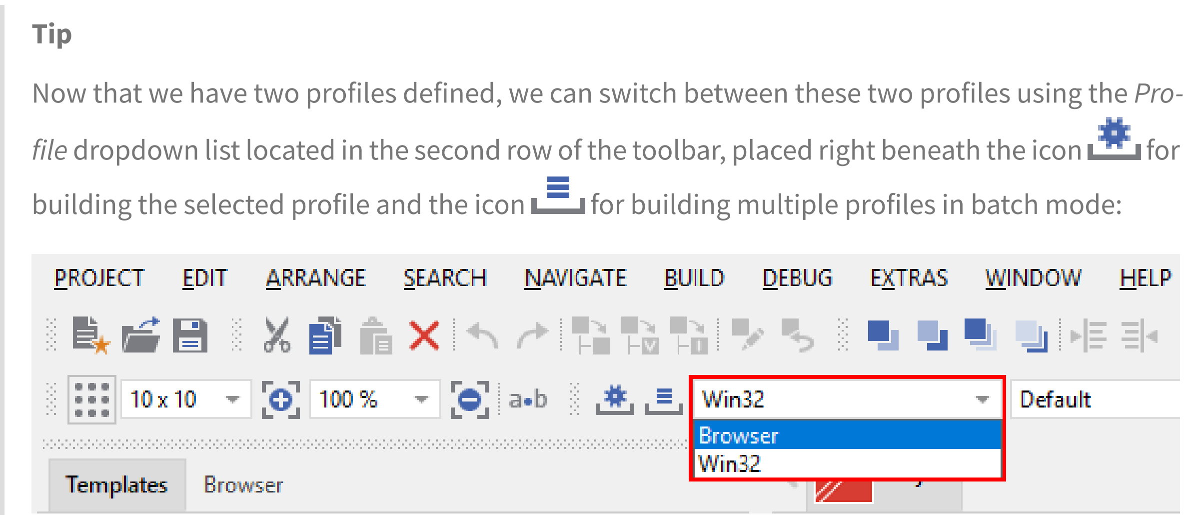


Figure 5.4: Project profiles and their attributes

> **Tip**
>
> Now that we have two profiles defined, we can switch between these two profiles using the *Profile* dropdown list located in the second row of the toolbar, placed right beneath the icon for building the selected profile and the icon for building multiple profiles in batch mode:
>
> 
>
>
> **Figure 5.5:** Dropdown list for switching between different profiles

5.5 Adding a browser device class variant

We already added an interface device class to our project. However, this interface device class is not meant to run any command on the device. The actual execution of the command on the device will happen inside a class variant. Variants are an extremely powerful concept of Embedded wizard, in the example below we use it to manage code execution on different platform packages. Variants are useful

in various other scenarios, you may use them to manage variants of your application for different screen resolutions or if you want to implement a different look and feel for one or more application components. Let's go ahead and add and populate a class variant for the use with browser devices:

- Click on the *Project* tab (the first tab from the left).
- Identify the note frame with the caption *Middleware* which holds three device units, the *Device* unit, the *BrowserDevice* unit and the *TargetDevice* unit.
- Double click on the icon unit BrowserDevice, representing the *BrowserDevice* unit. This will open the empty unit inside a new tab.
- Press [Alt+2] to select the gallery browser window, left to the main composer window. Alternatively, you may mouse click at the second tab *Browser* at the very top of the window.
- In the search field immediately below the two tabs, type in *Device* to shorten the class list.
- Right click on the *Device::DeviceClass* class to invoke the context menu on this class. From this menu select the menu item *Copy* to copy the class to the clipboard.
- Right click on the empty canvas in the main window to invoke a context menu. From this menu select the menu item *Variant paste* to insert a class variant of the device class. Alternatively, you may select the element and drag it over to the canvas while keeping [Ctrl+Shift+Alt] pressed.

  Note the letter *V* in the icon of the newly inserted class which marks the class as a class variant.
- Rename the newly inserted variant class to *DeviceClassBrowser* using the [F2] key.
- In the inspector window, locate the attribute line *VariantCond*. The right hand *value* cell of this attribute line holds a small downwards triangle at the right hand side. Click on this triangle to open the dropdown list populated with all profiles of your project. Deselect all profiles except for the profile *Browser* and click on the lower button labelled with a check mark to confirm your choice. With this setting in place, the class variant is now associated with the *Browser* profile only.

Your screen should now look as shown in figure_title below:

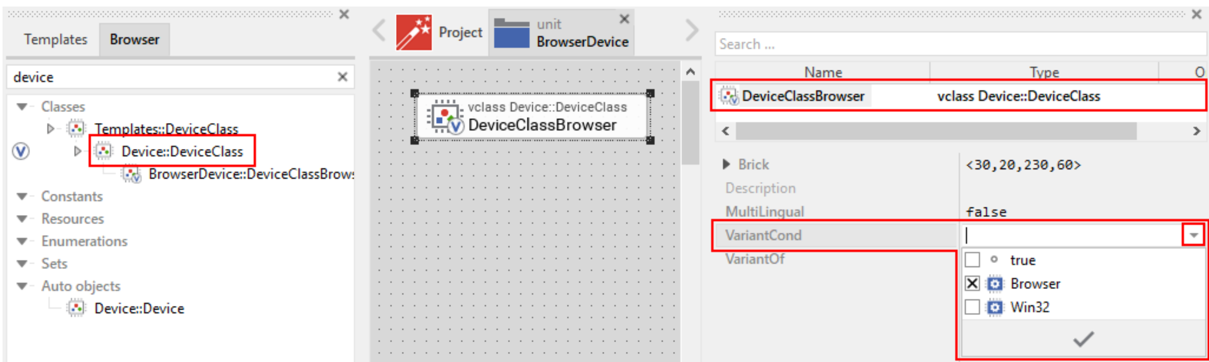


Figure 5.6: Browser device class variant

5.6 Implementing a different behavior for the browser device class variant

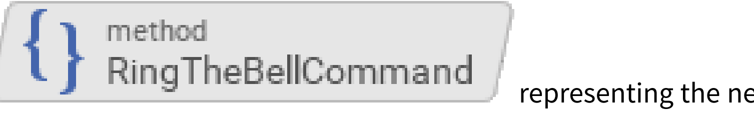
We want to make the newly created class behave differently, so there's still some work to do:

- Double click on the icon vclass Device::DeviceClass DeviceClassBrowser, representing the recently add device class variant. This will open the empty *DeviceClassBrowser* class variant inside a new tab.

- From the gallery templates window to the left, drag an element *Annotation Group* over to root canvas. Rename the group to *Command(s)*.

Have a look at the inspector window and you will see the method *RingTheBellCommand*. This is the command we previously added to the *Device* class. Since the variant class is derived from this class, it has access to all its class members, including the *RingTheBellCommand*. The light grey colour of the method name marks the method as inherited. We now want to implement a different behavior for this command in the variant class, we can do so by overriding the method in the variant class:

- In the inspector window, right click on the method *RingTheBellCommand* to invoke the context menu on the method. From this menu select the menu item *Override class member*. A method element with the same name *RingTheBellCommand* will appear on the canvas.

- We are now able to specify different code in the method body of the newly created method: double click on the icon 

representing the newly in-

serted method *RingTheBellCommand*. Inside the code editor, you will see the line `// TO DO: Write your code here …` . That's great, we can add our custom code here which will be executed only once the browser device class variant was called!

Using the code editor, add the following code inside the method body:

```
trace "Command on browser device was called";

$if (!$prototyper)
  native
  {
    // Javascript code executed inside the browser
    console.log("Hello, world!");
    console.log("We will be able to ring the bell shortly");
  }
$endif
```

Eventually, we are revealing how Embedded Wizard can execute native code on the device: by making use of the *native* statement of the Chora language. Any code inside this statement remains untouched and is passed *as is* to the device. Since we are communicating with browser devices, we have to put JavaScript code inside the *native* statement. More specifically, we make use of the `console.log()` method which outputs arbitrary text to the browser's console.

> **Note**
>
> The construct `$if (!$prototyper) … $endif` around the *native* statement prevents the native code block from being executed once we are using the prototyper for previewing our components or our applications. By adding this statement, we prevent Embedded Wizard from raising a warning that native code will the ignored during prototyping.

The *BrowserDevice* class variant should now look like as depicted in figure_title below:

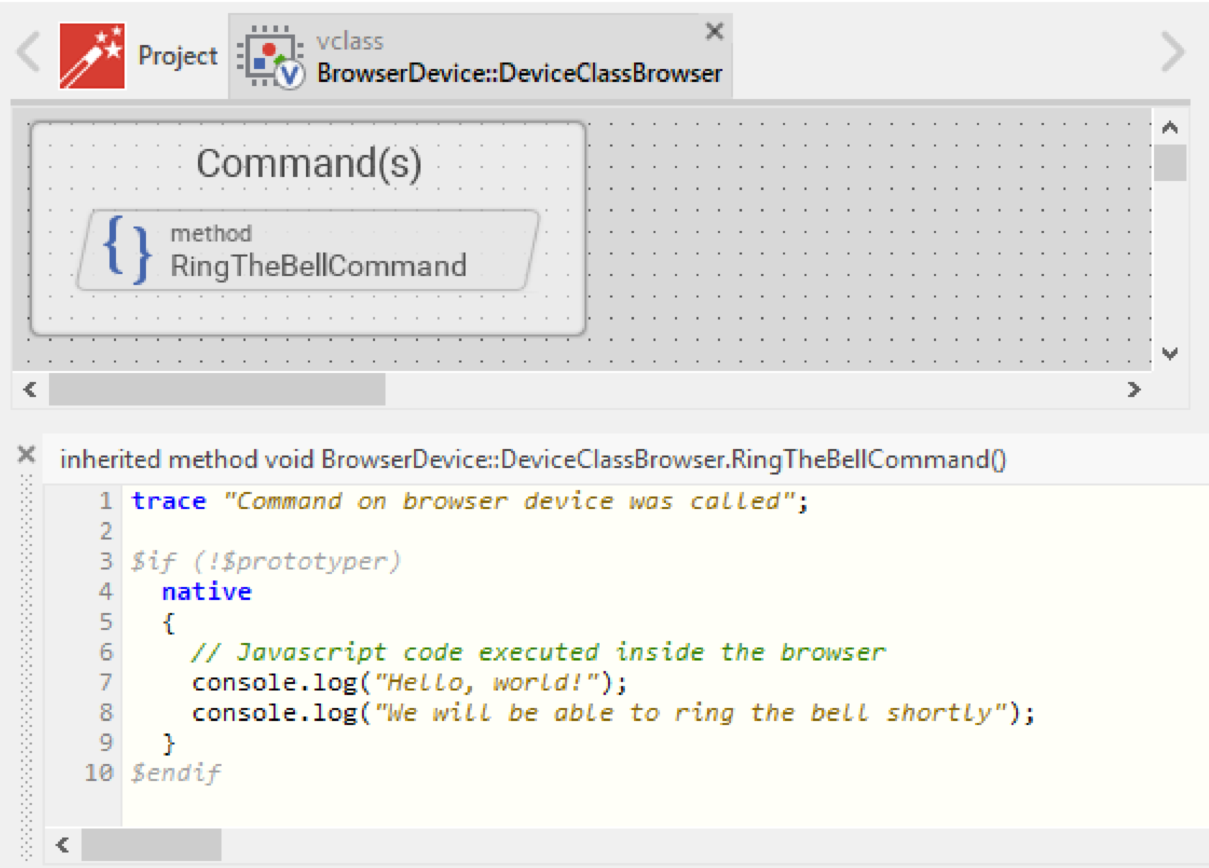


Figure 5.7: Browser device class variant with command added

5.7 Running the application inside a web browser

Hooray, we are now ready to run the application inside a web browser of your choice! To do, so, we have to build the browser specific code first:

- Switch to the *Browser* profile using the dropdown list depicted in figure_title above.
- Click on the the icon for building the application for the selected *Browser* profile. The generated code will be written into the output directory *WebGL* on the root application level. We specified this output directory when creating the *Browser* profile.
- Locate the output directory on your local file system. We contents of this directory should look like as depicted in figure_title.

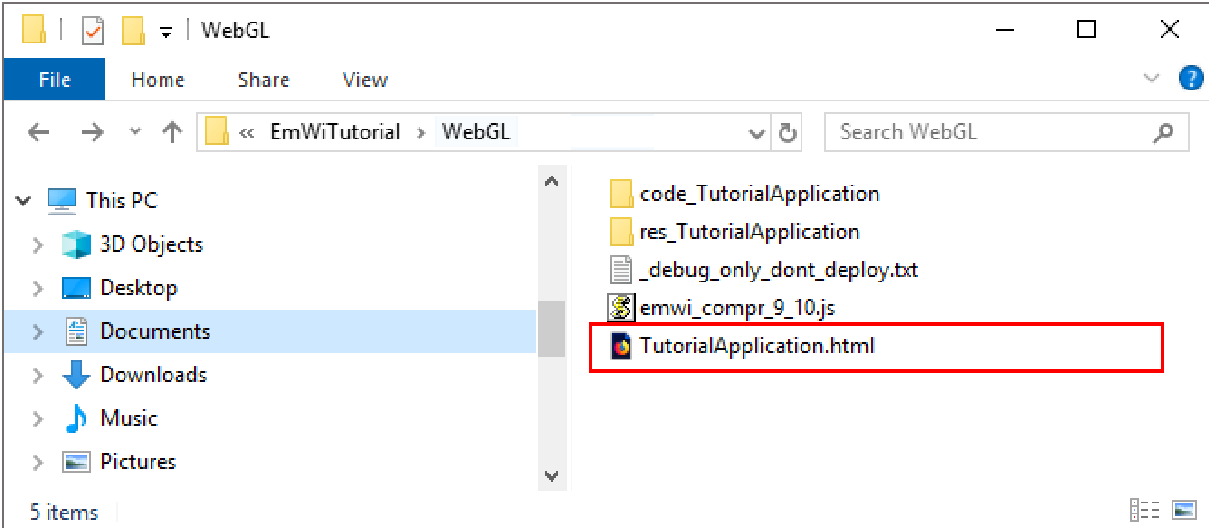


Figure 5.8: Contents of WebGL output folder

- Double click on the file *TutorialApplication.html* inside your *WebGL* output directory. This will open your default web browser with a window that runs your application:

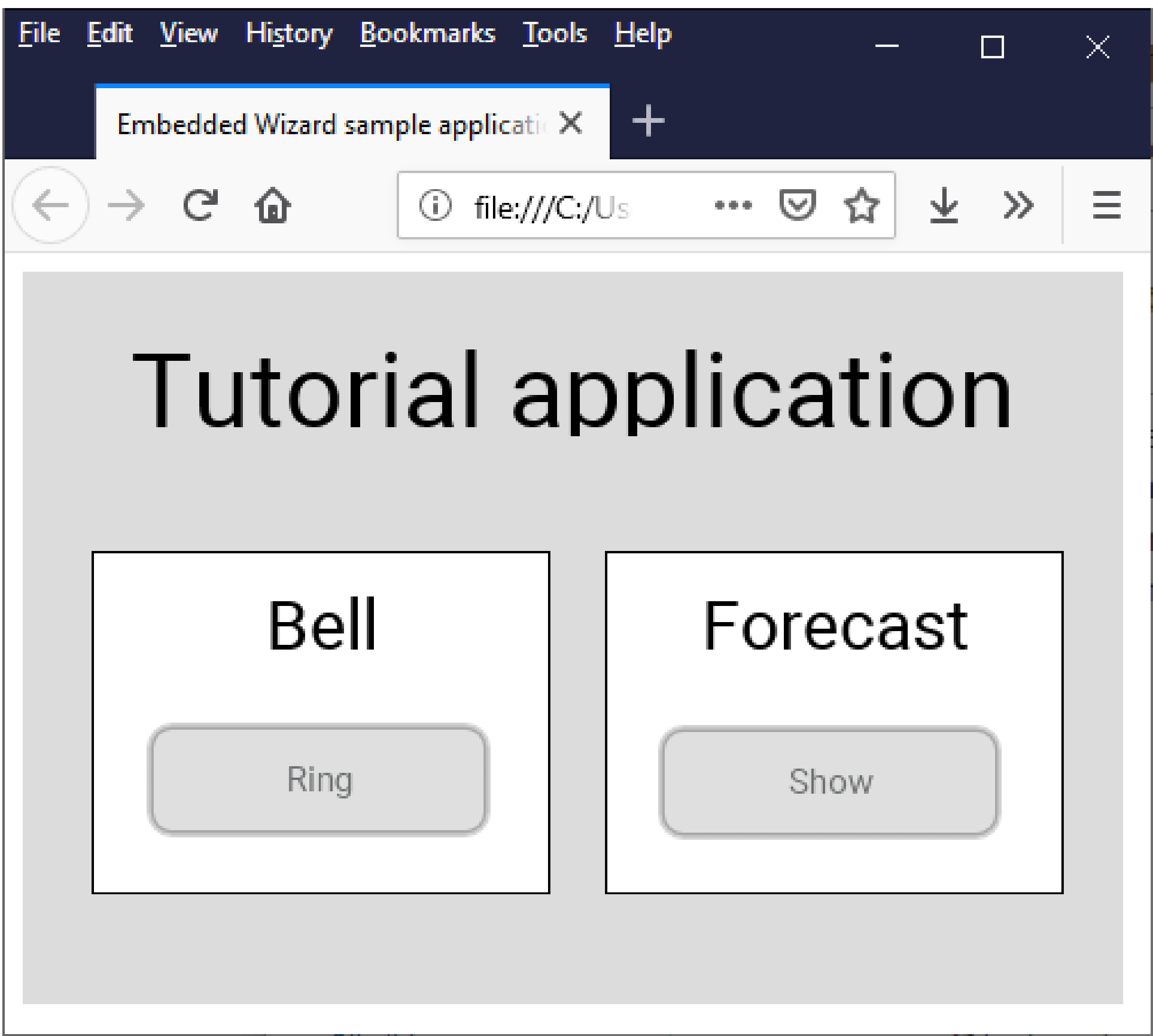


Figure 5.9: Executing the application inside a web browser

> **Important**
>
> Due to security concerns, the *Chrome* browser does not allow to load websites locally. We do **not** recommend the use of this browser for local preview of our application since most likely, you are running into trouble.

- Open the Javascript console of your browser. The way how to achieve that depends on your browser:

  - Firefox: From the menu, select Tools > Web developer > Browser console] or use the keyboard shortcut [Ctrl+Shift+J]

  - Microsoft Edge: Use the keyboard shortcut [F12] to open the Developer Tools, then click on the *Console* tab or press [Ctrl+2] to invoke that tab.

- Inside the application in your browser window, click on the push button labelled *Ring*. From your browser console, you should be greeted with *Hello, world!*:

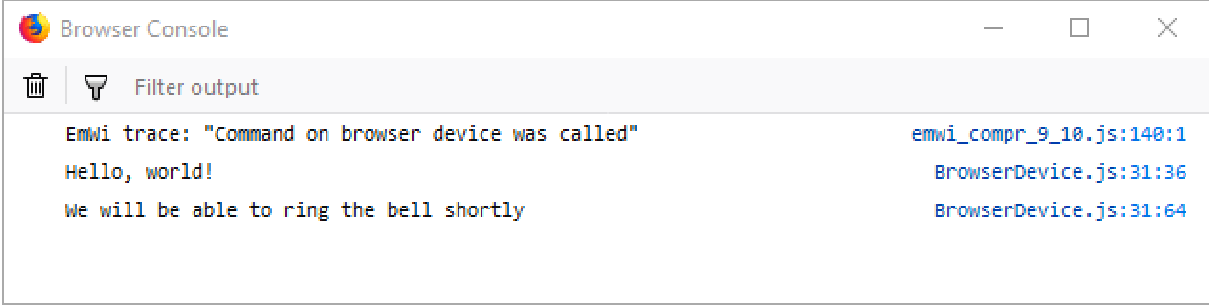


Figure 5.10: *Hello, world!* on the browser console

5.8 Finalizing device actions

5.8.1 Make the device beep eventually

In order to get results quickly, we made use of the `console.log()` inside the *RingTheBell* command. But we certainly can do better here, let's move on and let the bell ring!

- In the body of method *RingTheBellCommand*, remove the two lines with console.log statements and replace them with a single code line with a mere function call `beep();`. The code in the method body should now read:

```
trace "Ring the bell command on browser device was called";

$if (!$prototyper)
  native
  {
    // Javascript code executed inside the browser
    beep();
  }
$endif
```

The function *beep()* does not exist yet, so let's create it. We intentionally move this function out to the unit *BrowserDevice* in order to operate with small, separated code units.

- Select or open the tab with the unit *BrowserDevice*, the unit currently holds the variant class *DeviceClassBrowser* only.
- Click on the *Templates* windows to the left or invoke it by using the keyboard shortcut [Alt+1]. Press key [c] to open the folder *Chora* and show its elements.
- Identify the item *Inline code*, representing a native code snippet. Click on the element, drag it into the main area and place it underneath the existing *DeviceClassBrowser* element.
- Rename the newly inserted *inline code* element to *InlineFunctions* using the [F2] key.

The *BrowserDevice* unit should now look like as depicted in figure_title below:

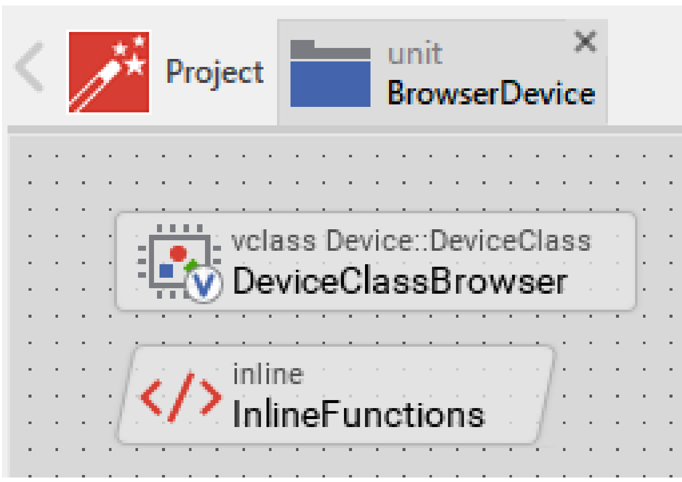


Figure 5.11: Browser device unit

- Double click on the icon representing the inline code element. Inside the code editor, you will see a single code line `// TO DO: Write your code here … .`
- Using the code editor, insert the function *beep()* inside the body of the inline element:

```
// reuse context since browsers limit the number of concurrent audio contexts
var audioContext = new AudioContext();

function beep() {
  var oscillatorNode = audioContext.createOscillator();
  var gainNode = audioContext.createGain();
  oscillatorNode.connect(gainNode);
```

```
  oscillatorNode.frequency.value = 500;
  oscillatorNode.type = "square";
  gainNode.connect(audioContext.destination);
  gainNode.gain.value = 1.5;
  oscillatorNode.start(audioContext.currentTime);
  oscillatorNode.stop(audioContext.currentTime + 0.2);
}
```

Test that your device makes *beep* eventually:

- Make sure the profile *Browser* is selected and rebuild your project using the [F8] key.
- Go to your web browser and issue a page refresh the browser page displaying your application using the [F5] key.
- Click on the push-button labelled *Ring* and your PC should beep eventually, provided, it has a speaker built in.

5.8.2 Presenting the weather forecast on the browser device

We are almost at the end of this long chapter! One task is left, however: we have to teach the browser device to display the weather forecast. That's pretty easy, though:

- Select or open the tab with the variant class *DeviceClassBrowser*, this class currently holds the command method *RingTheBellCommand* only.
- In the inspector window, right click on the method *ShowForecastCommand* to invoke the context menu on the method. From this menu select the menu item *Override class member*. A method element with the same name *ShowForecastCommand* will appear on the canvas. Rearrange elements so that the layout looks nice again if needed.

The final *BrowserDevice* class variant should now look like as depicted in figure_title below:

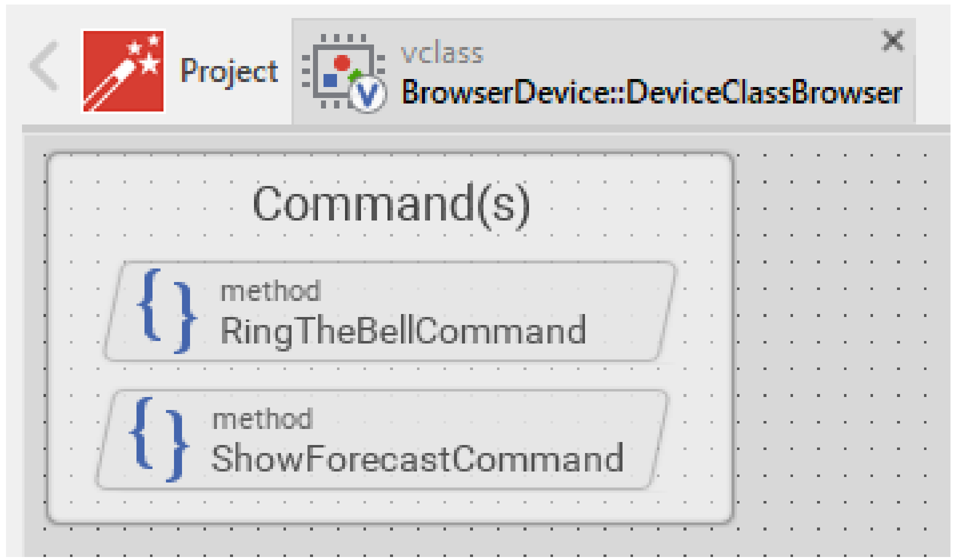


Figure 5.12: Browser device class variant with two commands added

- We are now able to specify different code in the method body of the newly created method: double click on the icon representing the newly inserted method *ShowForecastCommand*. Inside the code editor, you will see the line `// TO DO: Write your code here …` . Again now we can add our custom code here which will be executed only once the browser device class variant was called.

Using the code editor, add the following code inside the method body:

```
trace "Show forecast command on browser device was called";

$if (!$prototyper)
  native
  {
    // Javascript: redirect to weather forecast page
    window.open('https://www.cnn.com/weather', '_self');
  }
$endif
```

We are done! To test the newly added weather forecast functionality, rebuild the project and reload the application page in your web browser. You should now be able to both make your device beep and to view the weather forecast inside your browser. Pretty cool, isn't it?

Let's move on to the next chapter, there's still much more to explore!

6 Localizing your application

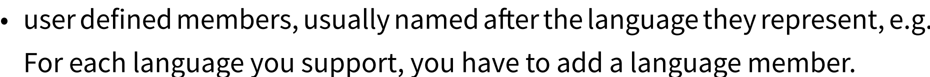
In the last chapter, we explained how to integrate your device, this task will be an essential part of every real world project. This is where newbies are often struggling, so hopefully you could make your way until the very end of this long chapter! In order to provide universal, target independent instructions, we explained the integration with a web browser device which is universally available to all users. So what's next? It's very likely that with your application you are targeting customers worldwide. At this point you are inevitably confronted with the localization of your application. The earlier you get started, the better it is, avoiding cumbersome rework afterwards! Fortunately localization is deeply embedded into *Chora* language used by Embedded Wizard, this makes it really easy to get things done right here.

6.1 Multilingual applications: concepts

As already stated above, the concept of languages is deeply embedded into the *Chora* programming language. In fact, Chora has even a data type *language* that represents a language. Each project contains at least one ore more members of this type. Please note that there are two different kind of language members:

- the member named *Default*, represented by the icon [language Default]. Each project has this member, which represents the so called fallback language exactly once. This *Default* member can neither be renamed nor can it be removed from the project.
- user defined members, usually named after the language they represent, e.g.

. For each language you support, you have to add a language member.

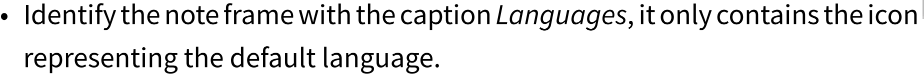


Language switching will be achieved by assigning a language member name to the built in global variable *language*. Don't confuse the built in global variable *language* with the data type *language*, both are named identical!

6.2 Adding another language

If you want to make use of multiple languages inside our project, you have to have an associated **language** on the Projects tab for each language you would like to use. So let's add German as second language to our project. Once done, we are able to switch between the default language (English in our case) and the German language:

- Click on the *Project* tab (the first tab from the left).
- Identify the note frame with the caption *Languages*, it only contains the icon

 representing the default language.

- From the gallery templates window to the left, drag the *Language* element inside the *Chora* folder over to the canvas and place it underneath the existing *Default* language. Rename the language to *German*. Resize the note frame and rearrange the elements on the canvas so that the layout looks nice again.

The *Language* section of your project should now look as shown in figure_title below:

Figure 6.1: Project languages

> **Tip**
>
> Now that we have two languages defined, we can switch between these two languages using the *Language* dropdown list located in the second row of the toolbar, placed between the *Profiles*

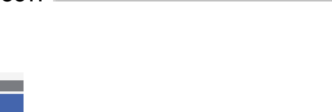
dropdown list and the *Styles* dropdown list:

Win32 Default (none)
Default
German
Project

Figure 6.2: Dropdown list for switching between different languages

6.3 Adding language strings / constants

We just added a second language to our project, but this is not the end: next we have to add language strings to our project that will be used when we want to display text in different languages. We will put all these strings inside a new unit *Strings* so let's go ahead and add the unit and the language strings:

- Click on the *Project* tab (the first tab from the left).
- Identify the note frame with the caption *GUI project*, it only contains the icon 

representing the *Application* unit.
- From the gallery templates window to the left, drag the element *Unit* inside the *Chora* folder over to the canvas and place it beneath the existing *Application* unit. Rename the unit to *Strings*.
- Double click on the newly inserted unit unit Strings. Inside the composer window, a new tab *Strings* appears which shows the currently empty unit.
- From the gallery templates window to the left, drag the element *Annotation Group*, over to root canvas. Rename the group to *Application*.
- From the gallery templates window to the left, drag the element *Constant* inside the *Chora* folder over to the canvas and place it inside the just inserted annotation group . Rename the constant to *HeaderTutorialApplication*.

With the newly inserted language string selected, have a look at the inspector window and identify the element *Values* for the constant, which by default has the value *”Hello World”. Please note the small triangle at the left hand side of the _Values* field. Click on the triangle and two more rows will come up: *Default* and *German*. While the *Default* row holds the fallback value for the string, the row *German* shall be used for the German translation string. Surely you grasped the concept already: the rows represent the language variant of the string constant, there is one row for each project language defined. If we defined a new language Spanish to the project, another row *Spanish* would appear. This concept of language variants makes localizing you application very easy. Let’s move on:

- Since we want to use the newly inserted language string as header text of our application, we change the value *Default* from *”Hello World”* to *”Tutorial application”* and the value *German* to *”Tutorial Anwendung”*. Don’t forget the surrounding double quotes in both cases or you may run in trouble.

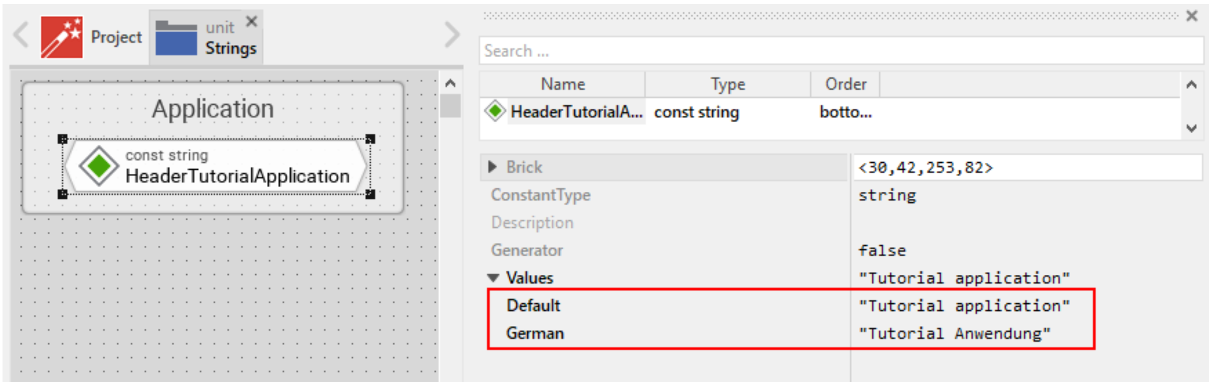


Figure 6.3: Language string and its translations

- Add two new annotation groups with two new language strings inside each group. Consult table_title concerning the details.

Table 6.1: Language string constants (to be inserted)

Name	Default / Fallback value	German translation	Annotation group
HeadingBell	”Bell”	”Klingel”	Bell component
LabelPushButtonBell	”Ring”	”Läuten”	
HeadingForeCast	”Weather report”	”Wetterbericht”	Forecast component
LabelPushButtonForeCast	”Show”	”Aufrufen”	

Once all four language strings were added to the project, the *Strings* unit should look as illustrated in table_title.

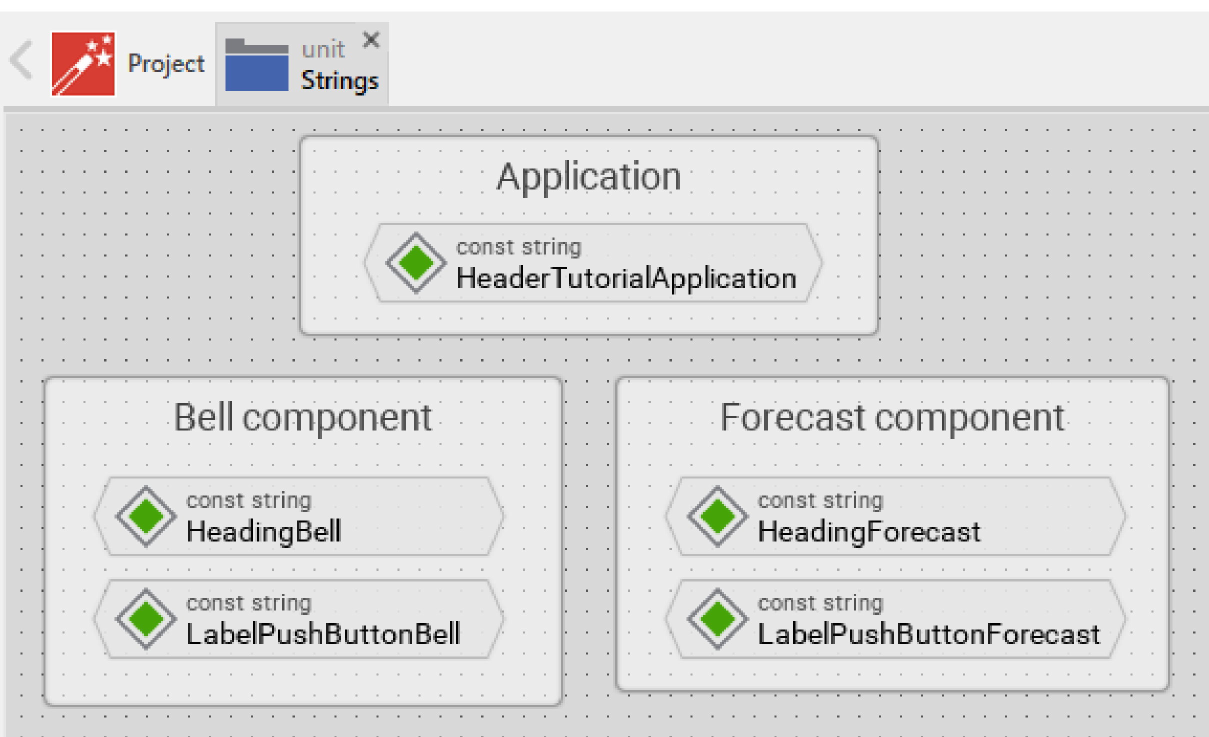


Figure 6.4: Language strings inside the *Strings* unit

6.4 Localize the application text strings

Now that we have the language strings defined, let's make use of them!

- Open the *TutorialApplication* root class. To do so, invoke the *Profile* tab, double click on the icon . In the *Application* unit, click double click on the icon , representing the root application class.
- With the root canvas displayed, click on the header text element of your application. In the inspector window, have a look at the property *String* of this text element. Currently, this property has the hard-coded value *"Tutorial application"* assigned. Delete the hard-coded value and write in *Strings::HeaderTutorialApplication* instead: we now use one of the five localized language strings we added in the previous step. Instead of typing in the value manually, you may

make select this value from the dropdown list that appears once you click on the triangle at the right end of the row.

- Next, we have localize the bell component: inside the inspector window, search for and click on the *BellComponent* or click on the component in the main window. Change the property *TextHeader* of this component to the language string *Strings::HeadingBell*. Also, change the property *LabelButton* of this component to the language string *Strings::LabelPushButtonBell*.
- Finally, we have localize the forecast component: inside the inspector window, search for and click on the *ForecastComponent* or click on the component in the main window. Change the property *TextHeader* of this component to the language string *Strings::HeadingForecast*. Also, change the property *LabelButton* of this component to the language string *Strings::LabelPushButtonForecast*.

That's it, our application uses localized text strings now! Let's test this out: usw the language dropdown (illustrated in figure_title) to change the project language to *German* and all the textual elements used in our application will change to *German* immediately! Isn't that great? For sure, we will need a language switcher now, allowing the user to switch the language. Stay with us, in the next chapter, we will explain how to add such a switch to the application!

7 Adding a language switcher

In the last chapter, we added German as second language to the project and created five localizable language strings. Using these language strings, we replaced all hard-coded text strings in the textual elements of our GUI components. That's great, our application is localized now. However, currently the user does not have the possibility to change the language of the user interface. We will change this in the next lesson by adding a language switcher to the application.

7.1 Adding the resources for the language switcher

Our language switcher will be quite simple: we will place a English and a German flag side by side in the upper right corner of the application window. Once the user clicks on a flag the language will be changed.

First we store the needed image icons inside the project:

- At the project root level, create a new folder *Flags*.
- Download the two flag icons uk.png and de.png and save them in the newly created directory

7.1.1 Adding the flag bitmap resources

Now that we have the two icons available, we will make use of them and create two graphical bitmap resources inside our project. We will put these resources inside a new unit *Icons* so let's go ahead and add the unit first and the two flag bitmap resources afterwards:

- Click on the *Project* tab (the first tab from the left).
- Identify the note frame with the caption *GUI project*, it currently contains the icon

representing the *Application* unit and the icon unit Strings representing the *Strings* unit.

- From the gallery templates window to the left, drag the element *Unit* inside the *Chora* folder over to the canvas and place it underneath the existing *Strings* unit. Rename the unit to *Icons*.
- Double click on the newly inserted unit unit Icons. Inside the composer window, a new tab *Icons* appears which shows the currently empty unit.
- From the gallery templates window to the left, drag the element *Annotation Group*, over to root canvas. Rename the group to *Flags*.
- From the gallery templates window to the left, drag the element *Bitmap Resource* inside the *Resources* folder over to the canvas and place it inside the just inserted annotation group. Rename the constant to *FlagUK*.
- In the inspector window, change the property *FileName* of the newly inserted bitmap resource to *.\Flags\uk.png*. You may either type in the path and filename directly or you can make use of the file chooser that gets invoked one you click at the symbol ... located at the right end of the row.
- From the gallery templates window to the left, drag another *Bitmap Resource* element over to the canvas. Rename the constant to *FlagGermany*.
- Change the property *FileName* of the newly inserted bitmap resource to *.\Flags\de.png*.

7.1.2 Building a pushbutton flag component

With the resources available, we now can build a reusable pushbutton flag component that will be used inside the language switcher component later on:

- We want to place the push button component in the *Application* unit, so we have to invoke this unit first: click on the *Project* tab and double click on the *Application* unit to open the corresponding composer tab. Inside the tab, you should now see the *Application* root component and the *Pushbutton component*, built by you.
- From the gallery templates window to the left, drag the element *Annotation Group*, over to root canvas. Rename the group to *Language selector*.
- ...

7.1.3 Building the language switcher component

To be finished …