Gui2 design document

M. de Wever

November 6, 2026

Contents

1 Introduction **4**
- 1.1 Why why why? . . . 4
- 1.2 History . . . 5

2 Overall design **7**
- 2.1 Features . . . 7
- 2.2 The big picture . . . 8
  - 2.2.1 Widget . . . 8
  - 2.2.2 Window . . . 9
  - 2.2.3 Dialogue . . . 9
- 2.3 Directory structure . . . 10

3 Design details **12**
- 3.1 Boot strapping the library . . . 12
- 3.2 Layout algorithm . . . 12
- 3.3 Event handling and dispatching . . . 12
- 3.4 Iterator . . . 13
  - 3.4.1 Design . . . 13
- 3.5 Callbacks . . . 14
- 3.6 Tooltip placement . . . 15
  - 3.6.1 Normal . . . 15
  - 3.6.2 Large . . . 16
  - 3.6.3 Huge . . . 16

4 Creating new widgets and dialogues **17**
- 4.1 Creating the widget . . . 17
- 4.2 Creating the window . . . 21

A Files for creating the widget **23**

B Files for creating the window **31**

Abstract

Late 2007 I started to rewrite the gui engine of Wesnoth, this project is called the gui2 project. A lot of code and documentation has been written since. The documentation is available at:

- https://wiki.wesnoth.org/GUIToolkit The general information regarding the project and documentation regarding the WML used.
- https://devdocs.wesnoth.org The general doxygen information.
- https://devdocs.wesnoth.org/pages.html Contains some links to more gui2 specific information.

This documentation describes a lot of the details and the wiki describes some of the design from the WML side. Doxygen contains some information regarding the design of the C++ side but it misses the overall design. This paper tries to fill that gap, trying to explain more of the overall design and some of the design choices.

Chapter 1

Introduction

This paper describes the general design of the gui2 project. The paper is divided in several chapters:

Chapter 1 Explains why yet another document is added instead of using the doxygen and a short history of the gui2 project.

Chapter 2 Goes into the general design and structure of the gui2 project.

Chapter 3 Dives deeper into certain parts of the design of certain classes and algorithms.

A omission in this paper is the simple story of how to add your own widgets in the C++ code[1]. I have a slightly outdated paper on the subject, but decided not to add it; the paper needs polishing and updates, but I've some design changes planned for the 1.9 development series. These changes will also change the way of adding new widgets, so that paper needs to be rewritten and added after the rewrite is complete.

1.1 Why why why?

Why is this document written in LaTeX instead of doxygen? The reason is simple; during the development of the iterator class I felt I had to write a lot. Writing much documentation in doxygen feels awkward. A lot of time is spend on formatting, the number of subsections is limited (too low in my opinion). Also it's not possible to document why certain functions aren't implemented, a solution would be to make the function private and declared only, but that feels like an ugly hack.

Why is the document needed? In my opinion most manuals written in doxygen are great as reference manuals but fail to explain how to do certain things with the library and fail to explain why certain things are implemented in the way it is. Writing a separate document separated from the code has the disadvantage of getting earlier out of sync when an implementation details differs, but this can be used as an advantage as well. Since it's hard to keep the details in sync, it's a good reason not to dive into these details. This makes a

[1] The WML part is described in the wiki.

natural separation between which part should be in doxygen and which part in this manual.

Why after the above, are some parts of gui2 documented in separate pages in doxygen? This indeed seems a contradiction, but it's not. Remember the first paragraph stating it's annoying to write this kind of documentation in doxygen? These parts are where I realized it's annoying to do it in doxygen and haven't been incorporated in this document[2].

1.2 History

The gui2 project was started late in 2007 to fix several problems I had with the current gui at that time. The problems I had were:

1. When using small resolutions the gui started to look bad, since widgets simply left their container and were drawn outside it.
2. Most of the gui was hard-coded in the C++ code and thus not configurable for normal users.
3. ThemeWML, the part that allows the user to configure the layout, is not well understood by the current developers.

This let to the following design goals:

1. The gui should look well at every possible resolution and automatically fit well.
2. Everything, or at least as much as possible should be configurable by WML.
3. The project needed more documentation as the current ThemeWML, and explain how to create your own guis.

At the time of this writing the project is still work in progress[3] and will still take a while to finish. The initial goals were pretty clear, and still are. On the other hand how to implement certain parts turned out to be less clear.

The library is the first time I designed a gui toolkit and like with all larger projects, you need to learn from mistakes; preferably of others, but you're doomed to make your own as well. The design process is an iterative one, causing some parts of the design to be changed several times. At the time of writing[4] a lot of areas feel stable, but others are still planned to be rewritten. Mainly the handling of scrollbar widgets feels awkward and hacky and are slated for a redesign.

Other parts of the gui are still in the planning stage and not implemented yet, but the project moves along at a steady pace and more features will be added in the future.

[2] These parts are still mentioned in this paper, but they refer to the doxygen documentation.
[3] It was never expected to be a short during project.
[4] Shortly before the 1.8 release.

The history of this document also goes back a long time, it's mostly build from various scraps of notes I have on my system. The plan was to change these notes to doxygen format some day put it in the source. The problem was to find a nice place in the source to do so.

This lead to the fact the notes just stayed on my system and the number of notes accumulated. During travelling to work I started to design some things for the 1.9 development series and for the iterator class I needed a new kind of document. I decided that was the final straw and started to work on writing this document while travelling.

Since you read this document, it's uploaded in the Wesnoth source tree. However, this doesn't mean the document is complete. In fact it means more of the opposite, the draft is uploaded and I can start to work on finishing the document by adding the missing information.

Chapter 2

Overall design

This chapter takes a deeper look into the overall design of the library. Before we can explain how things work, we'll have a look at the features; both implemented and planned for the future. Once we know what the gui should be able to do I'll explain how these goals are achieved. The chapter ends with a short explanation of the directory structure so you'll be able to find the parts of the library.

2.1 Features

Wesnoth runs on a large amount of devices, with an even larger amount of display resolutions, from 320×240 on hand-held devices to 2560×1600 on large 30" screens. The gui should look appealing on all these devices. Therefore there are three resolution groups:

hand-held devices For these devices the code needs to be compiled with **TINY_GUI**. This forces all images to be scaled down by a factor two. The typical resolutions on these devices are 320×240 – 640×480.

PC For the usage on PC's Wesnoth offers the resolutions 800×600 – 2560×1600[1].

Netbooks The first time when netbooks were introduced the normal resolution was 800×480, which is slightly smaller than the minimum PC resolution. Therefore a start option –SMALLGUI was added making minimal modifications to the layout[2].

In order to facilitate this range of resolutions the gui2 code allows several definitions of a window, tuned for a specific resolution. The reason is twofold; first to make a difference between the hand-held devices and PCs, second to allow different views for different resolutions.

The main example of this feature[3] is the attack dialogue. The dialogue has a button to show the damage calculation, which shows a new dialogue with the calculation overview. From the UI point of view I consider that rather ugly and rather have tabs to switch between the views. But when I have a larger screen

[1] And larger once screens that size become available.
[2] Actually this is gui1 only.
[3] Not yet implemented, but one of the main reasons to add this feature.

the dialogue only fills a small part and I need to switch between tabs to see the info, in that case I rather have one dialogue without tabs, which directly shows all information. When the user changes the resolution the dialogue should switch between these two views, depending on the current resolution.

The gui should be able to adapt to the size actually needed, the current gui uses fixed sizes at some places. This leads to problems such as when the screen resolution is reduced widgets end up outside the dialogue or get truncated. The same for some translated texts, where the translation is much longer as the English original. Gui2 solves this problem by dynamically determining the size of a widget and adjust the layout until it fits. This has the disadvantage that dialogues with a lot of dynamic content, resize at random times and changing the layout of the dialogue slightly. Another disadvantage is that when a dialogue doesn't fit Wesnoth terminates. This problem can be fixed by adding scrollbars to every window. When the dialogue doesn't fit the scrollbars are used and the dialogue fits again, might be a bit ugly but at least everything fits again.

It must be possible for WML designers to change the entire gui of Wesnoth with their own version. For example Spacenoth[4] is Wesnoth in a space setting, so the project might want to use a more fitting user interface.

Adding a new gui is a lot of work and can't be done in one fell swoop, so the code needs to support a gradual conversion. Therefore when you select your own theme[5] the engine first searches in the new theme for items[6]. When an item is not found it uses that item from the default theme. Therefore it's mandatory that every item is defined for the default gui, which the engine validates at startup[7].

2.2 The big picture

First we dive into the components that define the gui.

2.2.1 Widget

A widget is the basic user interface element, like a label, a button or a text entry. Every widget has its own behaviour, which sometimes can be influenced by certain settings, but the main behaviour is fixed per widget. Next to behaviour a widget also has a visual representation so the user can see and interact with the widget.

Before the widget is shown to the user it needs to be created first, the creation happens in the window builder class. This class reads the definition of a widget and fills in the blanks et voilà the widget is there. Let's go over this part in more slow-motion.

[4] That project is dead, but that doesn't matter for this example.

[5] Support for this selection and the entire fall-back haven't been added yet. Obviously I want to add support, but it remains low priority until somebody really wants to add their own gui.

[6] A widget or window.

[7] That causes the problem that Wesnoth sometimes refused to start when the source and data are out-of-sync.

The definition The definition determines how a widget looks and some basic properties, these properties are the same for all instances of that widget. For example a button has a minimum size so the decoration can be drawn. It's possible to make multiple definitions of a button. These different definitions, can look different, have different minimum sizes and other properties.

These definitions are written in WML and a small loader class loads and validates the definition. The definition is then added to the list of known widgets of this type.

The definition of the widget depends on the resolution, this is for example used in the button. It has a minimum size depending on the decoration used, in **TINY_GUI** mode the decoration is scaled down, so the minimum size can also be reduced.

The builder The builder is started from the C++ code, started while building a window. This window definition contains a list of widgets to build with more instance specific values. For example a label builder has the text to show to the user as parameter.

These builder "scripts" are also written in WML and loaded by a small loader class that does the validation and builds the needed widget.

Widget The widget itself is written in a larger C++ class and it defines the behaviour and provides various hooks to modify the properties of that widget. These hooks are used by the builder, but can also be modified later by the engine.

Other hooks provide bindings to react to events. The bindings are now rather static, but with the new event handling added late in the 1.8 release series more things are possible. The plan is to enhance this part during the 1.9 release series and deprecate and remove the current interface.

Of course the question "why wait until 1.9" raises. The reason is simple: it was added late in the 1.8 cycle to fix certain issues with the MP lobby. At that time I had no time to convert the rest of the code, since I was working on the MP lobby instead.

2.2.2 Window

A window can mean two things, the window widget and the window definition. In this section the window definition is meant. Already discussed before, but a bit more verbose this time.

The window definition defines a window and which widgets are placed in the window. This window definition also depends on the resolution. This allows the window to look different depending on the resolution. The changes can be small or the window can look completely different.

2.2.3 Dialogue

A dialogue is a pure C++ thing. A dialogue shows a window, but is not a window itself. So what's the difference[8]? A window is a dumb combination of

[8]These are the definitions used in the gui2 code and not the definition of other window toolkits.

widgets created depending on the definitions in a WML file. After the window is created it often needs extra content and reacts to certain events. For example the language dialogue after building has an empty list of languages. This is where the dialogue comes into play. The user asks for the language selection dialogue. The code creates and shows a dialogue. The dialogue code builds a window, then fills the language list with the available languages, selects the current language and then shows it to the user.

In other dialogues the code also needs to wire in event handlers or build other structures. It searches the wanted widget by id, in some cases it doesn't even care what kind of widget is used. In other cases it needs to be of a certain class or "concept". This allows a flexible design and lets the user select the kind of widget used in some cases.

So the dialogue is the sugar between the WML window and an interactive dialogue shown in the game.

2.3 Directory structure

This section describes the directories available. Instead of listing them in alphabetic order, I list them in an order that makes explaining them more natural.

src/gui The general source code directory with all parts used for the gui.

- **dialogs** This directory contains all dialogues used in Wesnoth, in general every file contains one dialogue, the name of the file being the logical name of what the dialogue does. Some dialogues have a helper dialogue, which is sometimes embedded in the same file.
- **widgets** This directory contains all widgets used in the library. Every file contains one widget. Also base classes or concepts of widgets are stored in this directory. During the development it also accumulated some helper files, which don't fit in the aforementioned descriptions - this led to the creation of the auxiliary directory. Some files haven't been moved to this new directory yet.
- **auxiliary** This directory contains auxiliary classes. Some helper parts became so large that they got their own subdirectory.
  - **event** Contains all event handling code, the translation from SDL events to the internal events and their dispatching.
  - **widget_definition** Contains the code to serialize the WML to an internal data structure, needed to define that kind of widget. The names of the files match the widget names. (Most files are rather small, but I prefer small single tasked files over huge files controlling a lot of code[9].
  - **window_builder** Contains the code to create a widget object from the widget definition and the data supplied in the window definition. Again the name of the file matches the name of widget they build. (These files are also mostly small.)

[9] Some might remember I started with a single file, which did exactly that, but the file got too large to maintain efficiently.

data/gui The general data directory with all parts used for the gui. All guis shipped with Wesnoth should be in a sub-directory of this one, with a config file with the name of the directory as main entry point and include the sub-directory. At the time of writing only one gui is shipped, the default one.

default This directory is the bases for the default gui.

macro This directory contains some helper macros, for default font sizes.

widget This directory contains the definitions of widgets. Since there can be multiple definitions of a widget their name is the name of the widget with a suffix. The suffix for the default widgets is DEFAULT, for the others an appropriate name is picked.

window Contains definitions of windows, the name of the file is the name of the dialogue it represents.

Chapter 3

Design details

Now that the big picture regarding the library is known, I dive into parts of the code that can use more explanation. If a part is not explained here, it doesn't mean the code is obvious or simple, lack of time to document it properly is more likely the excuse.

3.1 Boot strapping the library

Before the library can be started the available widgets and windows need to be registered. This registering is done before MAIN() is started. The registering is done by some small static classes that do the registering of the item[1] in its constructor. Since this involves some redundant typing there are REGISTER_XXX macros written.

After all items are registered the library can be started, which is simply done by GUI2::INIT(). This function does all required steps to get the library up and running.

3.2 Layout algorithm

An important part of the gui engine is to properly layout the widgets in the available space. The documentation of that algorithm is written in doxygen[2].

3.3 Event handling and dispatching

The event handling translates the "raw" SDL events to an event structure specific to gui2, effectively decoupling the interface. This also allows adding frameworks to fake events. The documentation of that code is written in doxygen doxygen[3]

[1] Widget or Window.
[2] https://devdocs.wesnoth.org/layout_algorithm.html
[3] https://devdocs.wesnoth.org/event_dispatching.html

3.4 Iterator

The iterator class is written[4] to alleviate certain problems. The scrollbar containers have their own grid and a grid for its content. The implantation makes the looping over all children tricky. This has been solved, but the design of the solution is rather awkward. Obviously fixing the design is the right thing to do, but that breaks the iteration.

By first writing an iteration class the interface for iteration can be kept cleaner and the classes can easily be refactored.

3.4.1 Design

There are two basic kind of iterators in the design, the simple ones that can only travel to themselves and their direct children. This type will be referred to as basic iterator hereafter.

This basic iterator is a superclass for several specific subclasses. The superclass is a concrete class, which can be instantiated. This class acts as a sentinel iterator, signalling the end of a list.

Every widget has its own creation function returning a pointer to a subclass object, allowing the main iterator to keep pointers to basic iterators, which it uses for travelling.

The other kind is called the main iterator. This iterator is of the type which the user normally creates and uses. The class is a template class, where a policy designs how the travelling should go[5].

When the main iterator is created, it's possible to add a predicate to the constructor. The predicate determines what the travelling routine does with a candidate widget.

Now that we know the players in the game, let's look further into the implementation details and the decisions made.

Travelling

A widget can have several "layers", namely:

self The widget itself.

grid Container widget has a grid which is another layer, note that for a grid widget, its grid and self layer are the same.

content Scrollbar containers have a grid, containing their scrollbars and a dummy content spacer. Their real content is stored in a separate internal node, which is used as the content layer.

The travel policy decides in which order these layers are visited, and when a node from a "grid" or "content" is returned, which direction to travel? Travel over them first and then into their children or children first? These decisions are coded in the travelling policy.

[4] The code hasn't been written yet, only designed how it should look. Still I feel the design is rather finished and I can update this paper if details change too much.

[5] For now only one policy is planned, but I can think of more kinds

While travelling, the policy finds a candidate widget to travel to. This widget is offered for evaluation to the predicate, which returns one of the following values:

return The widget referred by the iterator is accepted and the algorithm found its target.

continue The widget is skipped and the next candidate is sought.

break The widget is not allowed and the travelling at this layer stops. This doesn't mean the algorithm gives up. This layer is cancelled but the travelling path might have more options, which are used. At the moment there's no way to tell that the candidate has failed and that the searching should stop altogether, an "exit" result might be added for that case.

Obviously, these names are inspired by the C++ keywords.

Copying

The basic iterators are copyable since their state can easily be copied.

The main iterator can't be copied, it would involve copying the state and copying the main iterator is deemed not to be needed.

Operator++(int) (postfix increment)

The postfix increment operator hasn't been added to either widget type. For the basic iterators, there's the problem that the type iterator returned after iteration might not be the original type. This works properly on the prefix increment operator since it's a virtual function, using the covariant return type.

The postfix version needs to return an object and that would involve slicing the returned object instead of binding it to a reference.

The main iterator has no postfix increment operator since it's not copyable. This of course is a bit like the chicken and the egg problem: since I wanted to prevent the postfix increment operator I made the class not copyable.

For most (standard) iterator classes the overhead of copying isn't too high since those iterators carry little state. This iterator carries a lot of state, so you don't want to copy it. The cost of copying is $O(n)$ where n is the depth in the widget tree the iterator is[6]. So in the same spirit that the standard library doesn't add operator[] for std::list I omitted the postfix increment operator.

3.5 Callbacks

§3.3 describes the generic event handling for the widgets, but in some cases a widget wants to notify other widgets of a state change. Parts of gui2 use simple C-style callbacks for that purpose, but using boost::function makes better replacement. Therefore the code was analysed closer and another problem should be rectified; The callback causes a binding between two objects, but they are

[6]This might depend on the travelling policy, but it won't get cheap

not notified of the deletion of the object leaving a hole for calling a destroyed object.

In order to fix the problem two classes are defined:

tnotifier The class sending the wanted notification.

tnotifiee The class to manage the lifetime of the connection.

The tnotifiee is a small class that holds a pointer to the receiver it's connected to, upon destruction it uses this pointer to deregister itself by the receiver. After that the callback function will no longer be called.

The class should be used as member of a class so it can manage the lifetime of the connection with the tnotifier.

The tnotifier is the main class. When a callback is registered it stores the callbacks in an internal list and updates the pointer in the tnotifiee to itself.

Upon destruction it clears the pointer in all tnotifiees that point to use. This way upon destruction of the tnotifiee it won't try to deregister itself with this destroyed object.

Subclasses of the tnotifier should add an notification function, so the notifier can call all callbacks in the list. The notifier should take care of this calling.

3.6 Tooltip placement

The placement of the GUI2 tooltips are handled by the tooltip window. This window needs to determine where it wants the tooltip to be placed, and determine its optimal size.

§3.6.1 Shows the how normal tooltips look in the game window.

§3.6.2 Shows the how large tooltips look in the game window.

§3.6.3 Shows the how huge tooltips look in the game window.

3.6.1 Normal

This tooltip has a single line of text and the width of the tooltip is below or equal to the wanted width of the tooltip. The wanted width of the tooltip is determined by a size where the line-length of the text is pleasant to read.

Or the tooltip has either a end of line character, causing it to be spread over several lines, or its text is too long for the wanted width, causing it to be wrapped. The text's height will still fit on the screen.

Figure 3.1 gives a visual representation of the placement algorithms used. The algorithms used are:

I. This is the preferred placement of the tooltip. The lower right corner of the tooltip is attached to the top left of the mouse pointer.

II. This is used when the mouse x-position is greater than the width of the tooltip window, but the mouse y-position is smaller than the height of the tooltip window. The top of the tooltip is attached to the top of the screen at the left side of the mouse pointer.

III. This is used when the mouse x-position is smaller than the width of the tooltip window, but the mouse x-position is greater than the height of the tooltip window. The left side of the tooltip is attached to the left side of the screen at the top of the mouse pointer.

IV. This is used when the mouse x-position is smaller than the width of the tooltip window, and the mouse y-position is smaller than the height of the tooltip window. The top of the tooltip is attached to the top of the screen at the right side of the mouse pointer.

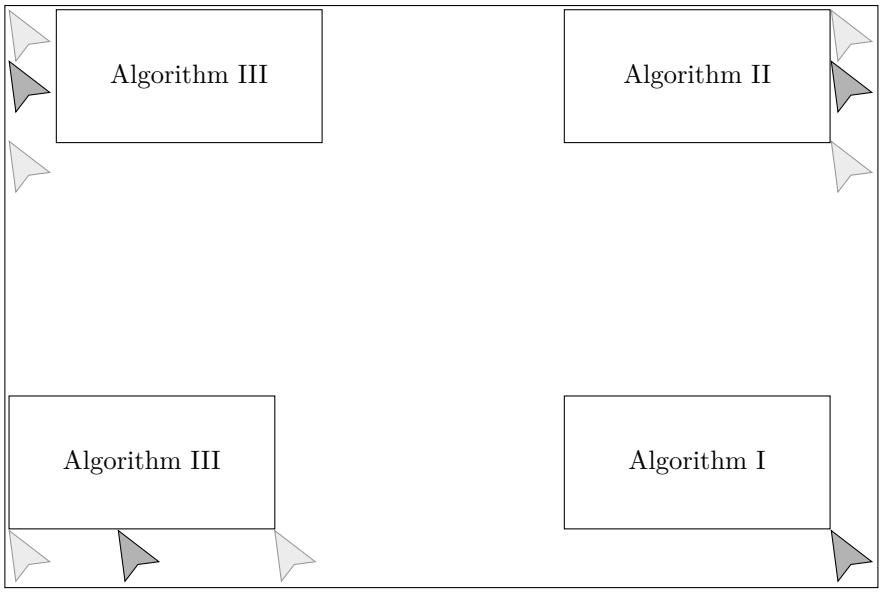


Figure 3.1: Overview of the tooltip placement.

3.6.2 Large

This tooltip needs so much space that its height no longer fits on the screen. In this case the width will be increased. There is no guarantee the tooltip will fit properly; the creator of the tooltip should consider whether or not it contains too much information. Once the width is adjusted the placement algorithm used it the same as for the normal tooltip, see §3.6.1.

3.6.3 Huge

The tooltip will certainly not fit nicely on the screen and no attempt is made to even try to do something sane. Instead it is placed in the top left corner. Unless a very small screen resolution is used the creator of the tooltip really needs to consider reducing the amount of text on the tooltip. There is no guarantee the entire tooltip is visible.

Chapter 4

Creating new widgets and dialogues

This chapter dives into the area of creating your own widgets or dialogues. For this example we use a real widget and a dialogue under development. The widget being developed is the progress bar and the screen the initial load screen.

First we make the widget, then the screen in order to test the widget. It happens often that, when you need to add a new widget, you need it for a new dialogue - so you need to create both for testing. Another option is to add the widget as dummy item to another existing dialogue. When adding a dialogue for which all widgets already exist this doesn't matter and you can directly dive into generating the dialogue.

4.1 Creating the widget

The widget is normally split in 7 different files. Three .cpp/.hpp files and one WML file. Usually I start to create the new files by simply copy pasting the old files and pick a rather small class like timage or something. Then, I start to update the build system files, followed by creating the code for the classes[1].

Now, we start to explain what the various files do. First, I do a general overview of the files after which I dive more into the implementation of each file.

src/gui/auxiliary/widget_definition/progress_bar.* These files contain the definition of the progress bar. This mainly involves the fixed fields for the widget. This widget has no extra fields so the copy pasted version suffices. This file contains the *static* properties for a certain definition of a progress bar.

hpp Listing A.1 contains the sample code.

Line 8 Normally a widget definition inherits from tcontrol_definition which defines the basic mandatory fields for a widget definition

[1] Obviously I already thought about the design of the class before that, but that's of less importance in this tutorial.

and a templated load function for the resolution. Most widgets don't add more members to this class.

Line 13 Normally a resolution definition inherits from tresolution_definition_ which define the basic mandatory fields for a resolution definition. Not all widgets use these members, but most do. Most widgets don't add more members to this class, except most container classes that add a grid builder definition. Another example is are the scrollbars which define some minimum sizes for parts of their control.

cpp Listing A.2 contains the sample code.

Line 1 Every file in the library is inside the "wesnoth-lib" text domain.

Line 9 The constructor used is pretty typical for all widgets.

Line 17 The constructor used is semi-typical. Resolutions that have their own members initialize them in the constructor, but also validate mandatory fields for their existence.
Also the progress bar only has one state, most widgets have more and thus add more states.
And maybe the most important part of this constructor is the wiki comment. This comment is used to generate wiki pages about the widget. These wiki pages are used by content creators to know how to create a proper WML file for the class.

data/gui/default/widget/progress_bar_default.cfg Now that the widget is defined, we can decide how it should look. Which is done in this file. Note that in the formula we use "percentage" which is the percentage of the bar to be filled.

Line 1 Every file in the library is inside the "wesnoth-lib" text domain.

Line 7 Beginning of the progress bar. This part is much better documented in the wiki[2]. Since this part is aimed at WML authors, duplicating that effort here makes no sense so just visit the wiki.

src/gui/auxiliary/window_builder/progress_bar.* These files contain the code to build a widget from the definition described above and a config object as defined in the window. These files contain also the *dynamic* properties of the progress bar. Since the progress bar has no dynamic content this file is also rather short.

hpp Listing A.4 contains the sample code.

Line 10 Normally a window builder inherits from tbuilder_control which defines the basic mandatory fields for a window builder.
The function also declares the build function.
Widget define their own member if they want to have some runtime settings, e.g. the spacer defines its fixed size if needed.

cpp Listing A.5 contains the sample code.

[2] https://wiki.wesnoth.org/GUIWidgetDefinitionWML

Line 1 Every file in the library is inside the "wesnoth-lib" text domain.

Line 13 The constructor used is pretty typical for all widgets.
If the constructor initializes custom members it may also add some validation. This is also true when the widget has a grid, like the listbox or multi_page.

Line 18 This build() is semi-typical, it creates the widget, initializes the default fields and returns the created object.
Widgets with their own members initialize them after initializing the default members.

Line 35 At the end of the file it contains two wiki comment sections: The first defines a macro with a short description of the widget. The second describes the extra fields for the instance of the widget.

src/gui/widgets/progress_bar.* These files contain the interaction the widget has with the outer world and how it reacts to events. Again, the progress bar is rather boring. The interesting part is the setter of the percentage. The value of the percentage used in the drawing routines is set here[3].

hpp Listing A.6 contains the sample code.

Line 8 Most simple widgets derive from tcontrol, which is the base class of the visual widgets. Another common base is tcontainer or tscrollbar_container.

Line 13 The class calls the control constructor with the number of states, which is the number of canvases used. Other then that, it does the initialization of its members. Some constructors do a bit more processing, but where a lot needs to be done it's often deferred to a finalize() function.

Line 21 This section implements or overrides member functions of the base class. Some functions are purely virtual and need to be overridden by others to change the behaviour.[4]

Line 36 This section contains the setters and getters for the class. The list below shows the common convention where T is the type of the setter/getter and "foo" the name of the member without trailing underscore.

setter Signature void set_foo(const T foo); The type T can be by reference if a larger class.

getter Signature T get_foo() const return foo_; The type T is always returned by value.

reference Signature T& foo() return foo_; This version returns the value by reference, and it should also have a const version. These functions or at least the non-const version are often not public.

[3]Other classes have a separate update canvas, but that's because the canvas in those classes is invalidated in several functions.

[4]Of course, adding here a list of functions which are common would be nice. However, the library is still too volatile to do so - no need to create a soon out of date list.

pointer Signature T* foo() return foo_; Almost the same as the reference version and is used for members that are pointers. The text of the reference also applies here.

The setter is normally available. The getter is often omitted when a reference or pointer function are used. Especially it there is a const version of these functions.

Line 48 The state has the list of states available and normally has COUNT as last value, which in turn is used in the constructor to initialize the base class.

cpp Listing A.7 contains the sample code.

Line 1 Every file in the library is inside the "wesnoth-lib" text domain.

Line 11 The common headers used for logging, this makes sure the output of the items always have the same format. Used by LOG_FOO « LOG_HEADER « "foo.\n"; or log_scope(foo, LOG_SCOPE);

Line 16 Registers a widget, this part is really volatile and still has some hoops and rings to jump through. Normally, it works. If not, too bad - try to look through the other widgets to see how to fix it.

Caveat emptor Just when all compiles fine and you suddenly get a runtime error about your new nice class, the builder claims your widget doesn't exist. *Don't panic*, probably you did nothing wrong. Grep TRY in src/gui/auxiliary/window_builder.cpp (at the moment of writing line 96) and read the comment above, duly add your class to the list and be happy.

Line 33 A simple helper to get the human readable name of the class. The first version was inlined in the header, but that seems to be an ODR violation. At least at some point a compiler or linker complained, can't really remember which one.

This completes the writing of the files for the new classes, some files need to be modified to get things up and running. The build system files need the three new .cpp files listed, these are:

- src/Makefile.am
- src/SConstruct
- src/CmakeLists.txt

The next step has become optional now, when most of build systems can glob the files needed for translation, but I still add them manually as well. The file is po/wesnoth-lib/POTFILES.IN. It holds the files for the wesnoth-lib text domain.

The last file to be edited is src/gui/widgets/settings.cpp. Here, a wiki comment for the new class needs to be added, this might change later. The reason is that before automatically registering widgets this file needed more modification, so it was done in one fell swoop.

This concludes the simple progress bar widget. Of course, we haven't tested it yet, since there's no dialogue to test it in. So let's implement a dialogue.

4.2 Creating the window

The window is normally split in 3 files. One .cpp/.hpp file and one WML file. The dialogue used is the unit attack dialogue, at least an initial version. This version barely does what it needs to do, and the in game version will be more polished. This simple version is nice as an example here since it's not too large.

Now, let's explain what the various files do. First, a general overview of the files after which I dive more into the implementation of the file.

src/gui/dialogs/unit_attack.*pp These files contain the code of the dialogue. This code 'binds' the WML code and allows the C++ code to show the dialogue show the right data at the right place. In several cases this also allows the designer of the WML file to decide which widget s/he picks.

hpp Listing B.1 contains the sample code.

Line 10 Defines the class itself, which normally inherits from tdialog. Sometimes a class inherits from another class, but this other class then derives from tdialog; for example the wml_dialogs have a common base and separate classes for left and right.

Line 14 Dialogues are often created and then shown once before being destroyed, therefore the constructor often takes all the parameters needed. Since the show() function shouldn't be overloaded, the only other alternative would be member functions to set the members. Note that when keeping 'references' to widgets these members must be pointers and set to NULL in the constructor, this due to the simple fact that the widgets aren't yet created at this point. Also keep in mind that when the dialogue is shown twice (using the *same* object) the widgets won't be the same. The window owning the widgets is destroyed after being shown and a new one created upon the next call to show().

Line 20 As said before, there are often no setters, since all is set in the contructor. Most members also remain hidden, since the caller has the information. Only members that change are 'exported'. The most common example in a class with a listbox, the last selected item is made public, since this is often used as the choice the user made.

Line 26 This function is a part of the window registration and only needs to be declared in the header, the definition will be provided by a macro.

Line 29 This function is called after the window is created, but before being shown. This is the point to set the members that point to a widget. As said before, every time the dialogue is shown a newly created window is shown as well, so setting the pointers can be unconditional. This is also the place to fill the widgets with the proper content, if needed. E.g. filling the languages in the language selection dialogue.

Line 32 Called after the dialogue is shown, this allows you to update certain statuses. E.g. if the dialogue is closed with the OK button set the selected language to the newly selected language.

You can also update it unconditionally and let the caller test the status.

Line 34 At this point the list of private members start.

cpp Listing B.2 contains the sample code. Note some parts are already described more verbosely in the header and thus not mentioned here.

Line 1 Like written for the widget, every file is inside the 'wesnoth-lib' text domain.

Line 13 Directly after the includes (inside the gui2 namespace) the wiki documentation starts.

Line 40 After the wiki documentation the registration code follows, it defines the window_id function and does some other registration parts.

Line 120 The pre_show is already mentioned in the header file and it the one used here uses several static helper functions to do it's job.

src/gui/dialogs/unit_attack.cfg This file contains the WML that defines the widget. It should honour the restrains set in the wiki comment like line 13. These constrains set the minimum set of needed widgets and their types, and mentions the optional widgets. How the widgets are placed and whether only the optional ones are set is up to the designer of the dialog. The gui toolkit wiki has more information on the subject. Still some parts are worth mentioning.

Line 1 Every file in the library is inside the "wesnoth-lib" text domain.

Line 6 With larger dialogs, I often first design them on paper and then divide the dialog in several sections. These sections often are turned into defines in order to keep the main dialog rather simple. WML tends to be verbose and deeply nested, and the GUI's also tend to have a lot of nested items, these powers combined result in deeply nested structure, where one can easily lose the way. Also note that all local macros are prefixed with _GUI, this is to avoid clashes with other macros. The BIG in the names are just because I want to make a different definition of high resolution screens (this will be the first screen using that feature, but that will be added later).

Line 218 Not much to tell about this part, since it's all documented in the wiki. But do note that due to the defines above, the grid in the window itself is rather simple.

Line 295 The local macros are no longer needed so undefine them, I normally undefine them in the oposite order of the definitions. I'm quite sure that's not actually needed, but it feels the right way™.

Appendix A

Files for creating the widget

This chapter contains the files created in §4.1, these files aren't the real files added, but a slightly modified version; The copyright headers are stripped to avoid taking up useless space. Some extra comments are added to make referencing possible. This also means the files here will get out of date with the real files when fixes are committed, that's also fine since it doesn't change how the files are structured. The label still mentions the original filename, but might over time differ from the listings below.

Listing A.1: src/gui/auxiliary/widget_definition/progress_bar.hpp

```
1  #ifndef GUI_AUXILIARY_WIDGET_DEFINITION_PROGRESS_BAR_HPP_INCLUDED
2  #define GUI_AUXILIARY_WIDGET_DEFINITION_PROGRESS_BAR_HPP_INCLUDED
3
4  #include "gui/auxiliary/widget_definition.hpp"
5
6  namespace gui2 {
7
8  struct tprogress_bar_definition
9      : public tcontrol_definition
10 {
11     explicit tprogress_bar_definition(const config& cfg);
12
13     struct tresolution
14         : public tresolution_definition_
15     {
16         explicit tresolution(const config& cfg);
17     };
18 };
19
20 } // namespace gui2
21
22 #endif
```

Listing A.2: src/gui/auxiliary/widget_definition/progress_bar.cpp

```
1  #define GETTEXT_DOMAIN "wesnoth-lib"
2
3  #include "gui/auxiliary/widget_definition/progress_bar.hpp"
4
5  #include "gui/auxiliary/log.hpp"
6
7  namespace gui2 {
8
9  tprogress_bar_definition::tprogress_bar_definition(const config& cfg)
10     : tcontrol_definition(cfg)
11 {
12     DBG_GUI_P << "Parsing progress bar " << id << '\n';
13
14     load_resolutions<tresolution>(cfg);
15 }
16
17 tprogress_bar_definition::tresolution::tresolution(const config& cfg)
18     : tresolution_definition_(cfg)
19 {
20 /*WIKI
21  * @page = GUIWidgetDefinitionWML
22  * @order = 1_progress_bar
23  *
24  * == Progress bar ==
25  *
26  * @macro = progress_bar_description
27  *
28  * The definition of a progress bar. This object shows the progress of a certain
29  * action, or the value state of a certain item.
30  *
31  * The following states exist:
32  * * state_enabled, the progress bar is enabled.
33  */
34     // Note the order should be the same as the enum tstate is progress_bar.hpp.
35     state.push_back(state_definition(cfg.optional_child("state_enabled")));
36 }
37
38 } // namespace gui2
```

Listing A.3: data/gui/default/widget/progress_bar_default.cfg

```
#textdomain wesnoth-lib
###
### Definition of an progress bar, which has the same height on normal and tiny
### gui.
###

[progress_bar_definition]
    id = "default"
    description = "A progress_bar."

    [resolution]

        min_width = 14
        min_height = 41

        default_width = 54
        default_height = 41

        max_width = 0
        max_height = 41

        [state_enabled]

            [draw]

                # Outer "golden" border.
                [rectangle]
                    x = 0
                    y = 0
                    w = "(width)"
                    h = 41
                    border_thickness = 1
                    border_colour = "188, 176, 136"
                [/rectangle]

                # Inner black border, with gray filling.
                [rectangle]
                    x = 1
                    y = 1
                    w = "(width - 2)"
                    h = 39
                    border_thickness = 1
                    border_colour = "0, 0, 0"
                    fill_colour = "21, 22, 24"
                [/rectangle]

                # 3D effect at upper side.
                [rectangle]
                    x = 2
                    y = 3
                    w = "(((width - 4) * precentage) / 100)"
                    h = 2
                    border_thickness = 0
                    fill_colour = "79, 103, 123"
                [/rectangle]

                # Main bar
                [rectangle]
                    x = 2
                    y = 5
                    w = "(((width - 4) * precentage) / 100)"
```

```
62                      h = 33
63                      border_thickness = 0
64                      fill_colour = "21, 53, 80"
65                  [/rectangle]
66
67                  # 3D effect at lower side.
68                  [rectangle]
69                      x = 2
70                      y = 38
71                      w = "(((width - 4) * precentage) / 100)"
72                      h = 2
73                      border_thickness = 0
74                      fill_colour = "10, 26, 40"
75                  [/rectangle]
76
77              [/draw]
78
79          [/state_enabled]
80
81      [/resolution]
82
83  [/progress_bar_definition]
```

Listing A.4: src/gui/auxiliary/window_builder/progress_bar.hpp

```
#ifndef GUI_AUXILIARY_WINDOW_BUILDER_PROGRESS_BAR_HPP_INCLUDED
#define GUI_AUXILIARY_WINDOW_BUILDER_PROGRESS_BAR_HPP_INCLUDED

#include "gui/auxiliary/window_builder/control.hpp"

namespace gui2 {

namespace implementation {

struct tbuilder_progress_bar
    : public tbuilder_control
{

    tbuilder_progress_bar(const config& cfg);

    twidget* build () const;
};

} // namespace implementation

} // namespace gui2

#endif
```

Listing A.5: src/gui/auxiliary/window_builder/progress_bar.cpp

```
#define GETTEXT_DOMAIN "wesnoth-lib"

#include "gui/auxiliary/window_builder/progress_bar.hpp"

#include "config.hpp"
#include "gui/auxiliary/log.hpp"
#include "gui/widgets/progress_bar.hpp"

namespace gui2 {

namespace implementation {

tbuilder_progress_bar::tbuilder_progress_bar(const config& cfg)
    : tbuilder_control(cfg)
{
}

twidget* tbuilder_progress_bar::build() const
{
    tprogress_bar* widget = new tprogress_bar();

    init_control(widget);

    DBG_GUI_G << "Window builder: placed progress bar '"
            << id << "' with definition '"
            << definition << "'.\n";

    return widget;
}

} // namespace implementation

} // namespace gui2

 /*WIKI_MACRO
```

```
36  * @start_macro = progress_bar_description
37  *
38  *        A progress bar shows the progress of a certain object.
39  * @end_macro
40  */
41
42 /*WIKI
43  * @page = GUIWidgetInstanceWML
44  * @order = 2_progress_bar
45  *
46  * == Image ==
47  *
48  * @macro = progress_bar_description
49  *
50  * A progress bar has no extra fields.
51  */
```

Listing A.6: src/gui/widgets/progress_bar.hpp

```
1  #ifndef GUI_WIDGETS_PROGRESS_BAR_HPP_INCLUDED
2  #define GUI_WIDGETS_PROGRESS_BAR_HPP_INCLUDED
3
4  #include "gui/widgets/control.hpp"
5
6  namespace gui2 {
7
8  class tprogress_bar
9      : public tcontrol
10 {
11 public:
12
13     tprogress_bar()
14         : tcontrol(COUNT)
15         , percentage_(-1)
16     {
17         // Force canvas update
18         set_percentage(0);
19     }
20
21     /***** ***** ***** ***** Inherited ***** ***** ***** *****/
22
23     /** Inherited from tcontrol. */
24     void set_active(const bool /*active*/) {}
25
26     /** Inherited from tcontrol. */
27     bool get_active() const { return true; }
28
29     /** Inherited from tcontrol. */
30     unsigned get_state() const { return ENABLED; }
31
32     /** Inherited from tcontrol. */
33     bool disable_click_dismiss() const { return false; }
34
35
36     /***** ***** ***** setters / getters for members ***** ****** *****/
37
38     void set_percentage(const unsigned percentage);
39     unsigned get_percentage() const { return percentage_; }
40
41 private:
42
43     /**
```

```
44          * Possible states of the widget.
45          *
46          * Note the order of the states must be the same as defined in settings.hpp.
47          */
48         enum tstate { ENABLED, COUNT };
49
50         /** The percentage done. */
51         unsigned percentage_;
52
53         /** Inherited from tcontrol. */
54         const std::string& get_control_type() const;
55 };
56
57 } // namespace gui2
58
59 #endif
```

Listing A.7: src/gui/widgets/progress_bar.cpp

```
1  #define GETTEXT_DOMAIN "wesnoth-lib"
2
3  #include "gui/widgets/progress_bar.hpp"
4
5  #include "gui/auxiliary/widget_definition/progress_bar.hpp"
6  #include "gui/auxiliary/log.hpp"
7  #include "gui/widgets/settings.hpp"
8
9  #include <boost/bind.hpp>
10
11 #define LOG_SCOPE_HEADER get_control_type() + " [" + id() + "] " + __func__
12 #define LOG_HEADER LOG_SCOPE_HEADER + ':'
13
14 namespace gui2 {
15
16 REGISTER_WIDGET(progress_bar)
17
18 void tprogress_bar::set_percentage(const unsigned percentage)
19 {
20     assert(percentage <= 100);
21
22     if(percentage_ != percentage) {
23         percentage_ = percentage;
24
25         foreach(tcanvas& c, canvas()) {
26             c.set_variable("percentage", variant(percentage));
27         }
28
29         set_dirty();
30     }
31 }
32
33 const std::string& tprogress_bar::get_control_type() const
34 {
35     static const std::string type = "progress bar";
36     return type;
37 }
38
39 } // namespace gui2
```

Appendix B

Files for creating the window

This chapter like the previous one contains listings of files, this time for §4.2. About the same is true for these files, except we know the originals will change in the future.

Listing B.1: src/gui/dialogs/unit_attack.hpp

```
1  #ifndef GUI_DIALOGS_UNIT_ATTACK_HPP_INCLUDED
2  #define GUI_DIALOGS_UNIT_ATTACK_HPP_INCLUDED
3
4  #include "actions.hpp"
5  #include "gui/dialogs/dialog.hpp"
6  #include "unit_map.hpp"
7
8  namespace gui2 {
9
10 class tunit_attack
11     : public tdialog
12 {
13 public:
14     tunit_attack(
15                 const unit_map::iterator& attacker_itor
16               , const unit_map::iterator& defender_itor
17               , const std::vector<battle_context>& weapons
18               , const int best_weapon);
19
20     /***** ***** ***** setters / getters for members ***** ****** *****/
21
22     int get_selected_weapon() const { return selected_weapon_; }
23
24 private:
25
26     /** Inherited from tdialog, implemented by REGISTER_WINDOW. */
27     virtual const std::string& window_id() const;
28
29     /** Inherited from tdialog. */
30     void pre_show(CVideo& video, twindow& window);
31
32     /** Inherited from tdialog. */
33     void post_show(twindow& window);
34
35     /** The index of the selected weapon. */
```

```
36      int selected_weapon_;
37
38      /** Iterator pointing to the attacker. */
39      unit_map::iterator attacker_itor_;
40
41      /** Iterator pointing to the defender. */
42      unit_map::iterator defender_itor_;
43
44      /** List of all battle contexts used for getting the weapons. */
45      std::vector<battle_context> weapons_;
46
47      /** The best weapon, aka the one high-lighted. */
48      int best_weapon_;
49  };
50
51  } // namespace gui2
52
53  #endif
```

Listing B.2: src/gui/dialogs/unit_attack.cpp

```
1   #define GETTEXT_DOMAIN "wesnoth-lib"
2
3   #include "gui/dialogs/unit_attack.hpp"
4
5   #include "gui/widgets/image.hpp"
6   #include "gui/widgets/listbox.hpp"
7   #include "gui/widgets/settings.hpp"
8   #include "gui/widgets/window.hpp"
9   #include "unit.hpp"
10
11  namespace gui2 {
12
13    /*WIKI
14    * @page = GUIWindowDefinitionWML
15    * @order = 2_unit_attack
16    *
17    * == Unit attack ==
18    *
19    * This shows the dialog for attacking units.
20    *
21    * @start_table = grid
22    *
23    *      attacker_portrait (image)       Shows the portrait of the attacking unit.
24    *      attacker_icon (image)           Shows the icon of the attacking unit.
25    *      attacker_name (control)         Shows the name of the attacking unit.
26    *
27    *
28    *      defender_portrait (image)       Shows the portrait of the defending unit.
29    *      defender_icon (image)           Shows the icon of the defending unit.
30    *      defender_name (control)         Shows the name of the defending unit.
31    *
32    *
33    *      (weapon_list) (listbox)         The list with weapons to choos from.
34    *      -[attacker_weapon] (control)    The weapon for the attacker to use.
35    *      -[defender_weapon] (control)    The weapon for the defender to use.
36    *
37    * @end_table
38    */
39
40  REGISTER_WINDOW(unit_attack)
41
```

```
42  tunit_attack::tunit_attack(
43            const unit_map::iterator& attacker_itor
44          , const unit_map::iterator& defender_itor
45          , const std::vector<battle_context>& weapons
46          , const int best_weapon)
47      : selected_weapon_(-1)
48      , attacker_itor_(attacker_itor)
49      , defender_itor_(defender_itor)
50      , weapons_(weapons)
51      , best_weapon_(best_weapon)
52  {
53  }
54
55  template<class T>
56  static void set_label(
57            twindow& window
58          , const std::string& id
59          , const std::string& label)
60  {
61      T* widget = find_widget<T>(&window, id, false, false);
62      if(widget) {
63          widget->set_label(label);
64      }
65  }
66
67  static void set_attacker_info(twindow& w, unit& u)
68  {
69      set_label<timage>(w, "attacker_portrait", u.absolute_image());
70      set_label<timage>(w, "attacker_icon", u.absolute_image());
71      set_label<tcontrol>(w, "attacker_name", u.name());
72  }
73
74  static void set_defender_info(twindow& w, unit& u)
75  {
76      set_label<timage>(w, "defender_portrait", u.absolute_image());
77      set_label<timage>(w, "defender_icon", u.absolute_image());
78      set_label<tcontrol>(w, "defender_name", u.name());
79  }
80
81  static void set_weapon_info(twindow& window
82          , const std::vector<battle_context>& weapons
83          , const int best_weapon)
84  {
85      tlistbox& weapon_list =
86              find_widget<tlistbox>(&window, "weapon_list", false);
87      window.keyboard_capture(&weapon_list);
88
89      const config empty;
90      attack_type no_weapon(empty);
91
92      for (size_t i = 0; i < weapons.size(); ++i) {
93          const battle_context::unit_stats& attacker =
94                  weapons[i].get_attacker_stats();
95
96          const battle_context::unit_stats& defender =
97                  weapons[i].get_defender_stats();
98
99          const attack_type& attacker_weapon = attack_type(*attacker.weapon);
100         const attack_type& defender_weapon = attack_type(
101                 defender.weapon ? *defender.weapon : no_weapon);
102
103         std::map<std::string, string_map> data;
```

```
104             string_map item;
105
106             item["label"] = attacker_weapon.name();
107             data.insert(std::make_pair("attacker_weapon", item));
108
109             item["label"] = defender_weapon.name();
110             data.insert(std::make_pair("defender_weapon", item));
111
112             weapon_list.add_row(data);
113
114         }
115
116         assert(best_weapon < static_cast<int>(weapon_list.get_item_count()));
117         weapon_list.select_row(best_weapon);
118     }
119
120     void tunit_attack::pre_show(CVideo& /*video*/, twindow& window)
121     {
122         set_attacker_info(window, *attacker_itor_);
123         set_defender_info(window, *defender_itor_);
124
125         selected_weapon_ = -1;
126         set_weapon_info(window, weapons_, best_weapon_);
127     }
128
129     void tunit_attack::post_show(twindow& window)
130     {
131         if(get_retval() == twindow::OK) {
132             selected_weapon_ = find_widget<tlistbox>(&window, "weapon_list", false)
133                     .get_selected_row();
134         }
135     }
136
137     } // namespace gui2
```

Listing B.3: src/gui/dialogs/unit_attack.cfg

```
1   #textdomain wesnoth-lib
2   ###
3   ### Definition of the window select which unit to attack
4   ###
5
6   #define _GUI_BIG_ATTACKER_PANEL
7   [grid]
8       id = "attacker"
9       linked_group = "unit"
10
11      [row]
12
13          [column]
14              border = "all"
15              border_size = 5
16
17              [image]
18                  id = "attacker_portrait"
19              [/image]
20
21          [/column]
22
23          [column]
24              grow_factor = 1
25              horizontal_grow = "true"
```

```
26
27              border = "all"
28              border_size = 5
29
30              [label]
31                  id = "attacker_name"
32                  definition = "alignment"
33              [/label]
34
35          [/column]
36
37      [/row]
38
39  [/grid]
40  #enddef
41
42  #define _GUI_BIG_DEFENDER_PANEL
43  [grid]
44      id = "defender"
45      linked_group = "unit"
46
47      [row]
48
49          [column]
50              grow_factor = 1
51              horizontal_grow = "true"
52
53              border = "all"
54              border_size = 5
55
56              [label]
57                  id = "defender_name"
58                  definition = "alignment"
59
60                  text_alignment = "right"
61              [/label]
62
63          [/column]
64
65          [column]
66              border = "all"
67              border_size = 5
68
69              [image]
70                  id = "defender_portrait"
71              [/image]
72
73          [/column]
74
75      [/row]
76
77  [/grid]
78  #enddef
79
80  #define _GUI_BIG_UNIT_PANEL
81  [grid]
82
83      [row]
84
85          [column]
86              {_GUI_BIG_ATTACKER_PANEL}
87          [/column]
```

```
88
89          [column]
90              {_GUI_BIG_DEFENDER_PANEL}
91          [/column]
92
93      [/row]
94
95  [/grid]
96  #enddef
97
98  #define _GUI_BIG_WEAPON_PANEL
99  [grid]
100
101     [row]
102
103         [column]
104             horizontal_grow = "true"
105
106             [listbox]
107                 id = "weapon_list"
108                 definition = "default"
109
110                 [list_definition]
111
112                     [row]
113
114                         [column]
115                             vertical_grow = "true"
116                             horizontal_grow = "true"
117
118                             [toggle_panel]
119                                 definition = "default"
120
121                                 return_value_id = "ok"
122                                 [grid]
123
124                                     [row]
125
126                                         [column]
127                                             grow_factor = 1
128                                             horizontal_grow = "true"
129
130                                             border = "all"
131                                             border_size = 5
132
133                                             [label]
134                                                 id = "attacker_weapon"
135                                                 definition = "alignment"
136                                                 linked_group = "weapon"
137                                             [/label]
138
139                                         [/column]
140
141                                         [column]
142                                             grow_factor = 1
143                                             horizontal_grow = "true"
144
145                                             border = "all"
146                                             border_size = 5
147
148                                             [label]
149                                                 id = "defender_weapon"
```

```
                                                    definition = "alignment"
                                                    linked_group = "weapon"

                                                    text_alignment = "right"
                                                [/label]

                                            [/column]

                                        [/row]

                                    [/grid]

                                [/toggle_panel]

                            [/column]

                        [/row]

                    [/list_definition]

                [/listbox]

            [/column]

        [/row]

[/grid]
#enddef

#define _GUI_BUTTON_ROW
[grid]

    [row]

        [column]
            grow_factor = 1

            border = "all"
            border_size = 5
            horizontal_alignment = "right"

            [button]
                id = "ok"
                definition = "default"

                label = _ "Attack"
            [/button]

        [/column]

        [column]
            border = "all"
            border_size = 5

            [button]
                id = "cancel"
                definition = "default"

                label = _ "Cancel"
            [/button]

        [/column]
```

```
212
213     [/row]
214
215 [/grid]
216 #enddef
217
218 [window]
219     id = "unit_attack"
220     description = "Unit attack dialog."
221
222     [resolution]
223         definition = "default"
224
225         automatic_placement = "true"
226         vertical_placement = "center"
227         horizontal_placement = "center"
228
229         # Both unit panels are the same width.
230         [linked_group]
231             id = "unit"
232             fixed_width = "true"
233         [/linked_group]
234
235         # All weapons share the same size, regardless whether attacker or
236         # defender.
237         [linked_group]
238             id = "weapon"
239             fixed_width = "true"
240         [/linked_group]
241
242         [grid]
243
244             [row]
245
246                 [column]
247                     grow_factor = 1
248
249                     border = "all"
250                     border_size = 5
251                     horizontal_alignment = "left"
252                     [label]
253                         definition = "title"
254
255                         label = _ "Attack enemy"
256                     [/label]
257
258                 [/column]
259
260             [/row]
261
262             [row]
263
264                 [column]
265                     horizontal_grow = "true"
266                     {_GUI_BIG_UNIT_PANEL}
267                 [/column]
268
269             [/row]
270
271             [row]
272
273                 [column]
```

```
274                         horizontal_grow = "true"
275                         {_GUI_BIG_WEAPON_PANEL}
276                     [/column]
277
278                 [/row]
279
280                 [row]
281
282                     [column]
283                         horizontal_grow = "true"
284                         {_GUI_BUTTON_ROW}
285                     [/column]
286
287                 [/row]
288
289             [/grid]
290
291         [/resolution]
292
293     [/window]
294
295     #undef _GUI_BUTTON_ROW
296     #undef _GUI_BIG_WEAPON_PANEL
297     #undef _GUI_BIG_UNIT_PANEL
298     #undef _GUI_BIG_DEFENDER_PANEL
299     #undef _GUI_BIG_ATTACKER_PANEL
```