AmbulantPlayer - Design Documentation

Version 1.1, 10-Oct-04

This document contains various bits and pieces of ambulant design documentation: a breakdown of the important objects, UML diagrams explaining how they relate to each other, a walkthrough explaining how Ambulant Player plays a SMIL document, an explanation of the source code organization and a list of source code conventions used.
A companion document API documentation explains low-level details of the various objects and other API issues.

Contents

1 Ambulant design, overall **2**
1.1 Design process . . . 2
1.2 Global structure . . . 2
1.3 Replaceable components . . . 2
1.4 run-time system . . . 2
1.5 main loop . . . 3
1.6 Factory pattern . . . 3
1.7 Machine-dependent code . . . 3
1.8 integrating third-party tools . . . 3
1.9 Other languages . . . 4

2 How does the AmbulantPlayer play a SMIL document? **4**
2.1 Introduction . . . 4
2.2 Opening a document . . . 4
2.3 Creating the player . . . 4
2.4 Starting playback . . . 5

3 Ambulant design, main objects **5**
3.1 Introduction . . . 5
3.2 AmbulantPlayer interfaces . . . 6
3.3 AmbulantPlayer Objects . . . 6
3.4 Refcounting protocol . . . 6
3.5 Player Interface . . . 6
3.6 Player implementations . . . 6
3.7 Parser interface . . . 6
3.8 Parser implementations . . . 7
3.9 Datasource interface . . . 7
3.10 Playable interface . . . 7
3.11 Playable implementations . . . 7
3.12 Layout interface . . . 7
3.13 Layout implementation . . . 8
3.14 GUI window interface . . . 8
3.15 Animation interface . . . 8
3.16 Clocks . . . 8
3.17 Event processor . . . 8
3.18 Document . . . 8
3.19 Node . . . 9
3.20 Transitions . . . 9

4 Ambulant source code organisation **9**
4.1 directory organisation . . . 9
4.2 Source code conventions . . . 9

5 Diagrams **11**
5.1 overall structure diagram . . . 11
5.2 dataflow diagram . . . 12
5.3 UML diagram for player . . . 13
5.4 UML diagram for parser . . . 14
5.5 UML diagram for datasource . . . 16
5.6 UML state diagram for datasource . . . 17
5.7 UML diagram for playable . . . 18
5.8 UML state diagram for playable . . . 19
5.9 UML diagram for renderer_playable . . . 20
5.10 UML diagram for layout . . . 21
5.11 UML diagram for region . . . 22
5.12 UML diagram for window . . . 23
5.13 UML diagram for animation . . . 24
5.14 UML diagram for clocks . . . 25
5.15 UML diagram for event processor . . . 26

5.16 UML diagram for document 27
5.17 UML diagram for node 28
5.18 UML diagram for transitions 29

1 Ambulant design, overall

Last updated for Ambulant version 1.1.

1.1 Design process

The design consists of a mixed bag of technologies:

- Text files such as this one for informal descriptions. They are marked up as reStructuredText, a format easily converted to HTML but also readable in source form.
- C++ header files for class definitions and such.
- Image files for UML pictures.

If possible all design documents will carry a notice stating for which version of Ambulant they were last updated, so it is relatively easy to spot outdated (or potentially outdated) documents.

1.2 Global structure

To be able to use the code to create, say, a plugin SMIL renderer for use in a browser we need a global "playback engine" object that has all the others hanging off it (plus factory methods to create them, etc). To support this we have a global object `player` that is the controller of all aspects of playback of a single SMIL document.

In addition we use factories for creating things like renderers, file readers, parsers and windows. These factories are populated by the main program and then used during document playback.

With a structure like this the application itself becomes basically a skeleton embedder: it is responsible for the GUI, handling open/open URL/quit/etc, it creates playback engine objects when needed and has a small number of callbacks for "create window" and such.

In addition, because the main program is responsible for creating all the factories it should be possible to create a different main program that does not actually render anything, but only prints on stdout what should happen at what time, or any other form of symbolic execution.

1.3 Replaceable components

AmbulantPlayer is intended to be a research system, and therefore all components should be easy to replace with interfering with the rest of the system. This allows a researcher to concentrate on one issue, such as network protocols or scheduling algorithms, while the rest of the system is usable as-is.

This replacability is incorporated in the design through two means:

- Clear well-defined APIs between the various parts of the system;
- Factory functions to create most objects.

In the current implementation there are two main objects for which this has not been implemented (the DOM tree and the low-level event scheduler), this will be done at a later stage. All other functionality (media playback, data retrieval, parsing, layout, scheduling) does follow this pattern.

1.4 run-time system

Because we have a C++ implementation we cannot rely on refcounting or garbage collection in the underlying runtime. lib/refcount.h has a simple refcounting implementation that is used for garbage collection.

Ref-counting should be used when it is absolutely needed, it adds an overhead and some complexity since it can not happen automatically. On the other hand there are some cases where it simplifies the code a lot. If I judge from my code and the code of others I have seen, only very few objects need ref counting. First, objects owned by a class, and quite all are, never need to be ref counted. You just delete them. Ref-counting is needed when objects containing references are shared by completely independent components.

The architecture is fairly tightly coupled. The original idea of allowing the high-level scheduler to live on a different machine, precomputing schedules and ending these to a low level scheduler, isn't going to work for SMIL without putting almost all SMIL complexity in the low-level scheduler.

1.5 main loop

The basic architecture is event-driven, with a small number of worker threads picking up events from the event queue. The alternative is to use multiple threads all over, but it seems event-driven is the better choice. An object that wants to use multiple threads can do so more easily on an event infrastructure than the other way around, but these threads are "somebody else's problem", as they are hidden from the rest of the architecture.

At first glance that it appears some objects, such as a renderer, would benefit from a threaded architecture it turns out this isn't really so. The naive threaded implementation:

```
while data = read_data():
    render_data(data)
```

will not work, because many other things can also happen, such as a user-initiated event, or the timeline for the renderer being torn down. So, the naive loop sketched here will become hairy anyway, and look like:

```
while event = wait_for_some_interesting_event():
        switch event:
                case DATA: render_data(data)
                case STOP: close_resources_and_exit()
                ...
```

so we might as will split this out in the architecture.

The event handler architecture needs an elaborate priority scheme, that is expressive enough that the best execution order of things that happen "at the same time" is automatic.

1.6 Factory pattern

There are various factories that follow a common pattern. There is a client interface, called something like `playable_factory`, that can be used to create `playable` objects. If the factory cannot create the object it returns `NULL`.

This interface is implemented by all the providers of objects that have the `playable` interface. For example, the implementation of a video renderer for Cocoa on MacOSX will consist of a `cocoa_video_playable` implementation and a `cocoa_video_playable_factory` implementation.

The core also has a provider interface, usually called `global_playable_factory`. This interface has a method `add_playable_factory` that the playable provider uses to register its `playable_factory`. Then, when a client uses the `global_playable_factory` to create a playable it will iterate over all playable factories until one is found that can create the object.

These `global_playable_factory` objects should be singletons, but in the current implementation this isn't always true. Also, the `global_` naming convention isn't strictly followed for all factories.

1.7 Machine-dependent code

The general way to handle machine-dependency is to create a machine-independent abstract base class, plus machine-dependent subclasses. Then there is a factory function that creates a machine-dependent instance and returns it casted to its machine-independent base class.

With this scheme we can handle machine-dependent extensions to the base class easily: modules using these extensions declare objects of the subclass and call the initializer directly in stead of through the factory function.

The scheme does not work for all objects, however: it breaks if we want to create static copies of the objects. For classes for which this is the case, such as the `critical_region` object, we declare an abstract object `abstract_critical_section` in `lib/abstract_mtsync.h`, subclass that as `PLATFORM::critical_section` in `lib/PLATFORM/PLATFORM_mtsync.h`, conditionally include that in `lib/mtsync.h` and create an empty subclass `critical_section` of it.

1.8 integrating third-party tools

We need to be able to use existing toolkits that take work out of our hands. Think of QuickTime and DirectX, where you basically pass a URL and say "play" and have nothing to worry about anymore. Also, existing URL access libraries (such as the caching infrastructure on windows) and a third party RTSP library need to be used. This also means we don't have to handle firewalls and what more.

An informal dataflow diagram is available showing the various ways in which media bits can go from their source (far away on the net) to the screen.

We would like to be able to re-use existing (Explorer, Netscape) plugins when applicable, but no work on this has been done yet.

1.9 Other languages

It is our intention to eventually allow bridging of Ambulant Player to other languages such as Python or Java. An effort has been made to keep the APIs clean enough to allow this, but no actual work has been done on it yet.

2 How does the AmbulantPlayer play a SMIL document?

Last updated for Ambulant version 1.1.

2.1 Introduction

This section attempts to explain the basic structure of the Ambulant Player by loosely explaining what happens when you run it and play a document.

There is an informal overall structure diagram that tries to put the whole design in one pretty picture and may be worthwhile to keep handy while reading this document.

The main program is platform dependent and GUI-toolkit dependent. The details of this main program are skipped here (and they can actually vary quite a bit for the platforms we support), but at some point after the program has started the GUI is put on the screen, with the usual set of menus for Open, Play, etc.

2.2 Opening a document

When the user selects Open (or Open URL, or double-clicks or drags a document) we need to get the data, parse the document into a DOM tree and create a player to play that DOM tree. In addition, we need to tell the player how it can obtain media data, create windows, and more.

Most player implementations (the Windows player is an exception) have a class with a name like `mainloop` to handle this. Such a mainloop is created per SMIL document. This mainloop object will first create the various factories and populate them:

- A `window_factory` is usually implemented by the main program itself. The player will call this when it needs a window. Usually the first request to create a window will actually return the document window (after resizing to the appropriate size).
- A `global_playable_factory` is created. This is the object the player will use to create renderers for the various media types. The global factory is filled with the various renderers this ambulant player supports. In effect, this is the step where you get to decide how various media are rendered.
- A `datasource_factory` is created and filled with the factory functions that will create datasources for audio, video or other, raw, data such as text. The factory functions that are added to the datasource_factory partially determine how data is retrieved over the net, which protocols and formats are supported and such. Partially, because some media items (audio and video, notably) may be rendered by simply passing the URL to some underlying media infrastructure such as DirectX or QuickTime.

Next the factories are put together in a `factories` struct, and if the architecture supports dynamically loadable plugins we get the `plugin_engine` singleton object and ask it to load the plugins. This will search the plugin directories for dynamic objects with the correct naming convention, load them, and call their `initialize` routine. The `factories` object is passed to the `initialize` routine, so the plugin itself can register any factories it wants.

The next step is to create the DOM tree. One way to do this is to use `read_data_from_url` to read the data from the document, and then pass this data to `document::create_from_string`. This will return a `document` object. This object contains the DOM tree itself (implemented by the `node` object) and some context information (XML namespace information, original URL for resolving relative URLs used in the document, a mapping from XML IDs to node objects).

The final step is to create a `player` object. This is done through `create_smil2_player`, passing the `document`, the `factories` and one final object, `embedder`. This object is again implemented by the main program, and implements a small number of auxiliary functions, such as opening an external webbrowser or opening a new SMIL document.

2.3 Creating the player

When the `smil_player` object is created it gets the `document`, `factories` and `embedder` arguments. It now needs to create its internal data structures to facilitate playback later on:

- A `timer` and `event_processor` are created. The timer is the master clock for the presentation, and the event processor is a runqueue object that is used for low-level scheduling. Whenever the high-level scheduler wants some code to be executed it will add an `event` to the `event_processor`, possibly with a timeout and a priority. The event processor runs in a separate thread, waits for events in it runqueue to become elegible and then runs them. This mechanism is the underlying engine that makes the whole player work, anything that needs to wait doesn't do so inline but uses the event processor to get a callback at a later stage: the scheduler, renderers needing input data, etc.

- A `layout_manager` is created, which will be used to find where media items should be displayed. The smil_layout_manager reads the <`layout`> section of the DOM tree and builds a parallel layout tree (which also contains information on some of the body media nodes, the ones that have layout information themselves) of `region_node` objects. Then this tree of `region_node` objects is converted into a tree of `surface_template` objects. To create toplevel windows the `new_topsurface` method of the window factory is used, and subregions are created using the `new_subsurface` method of their parent `surface_template`. The layout manager also contains mappings to be able to get from a `node` to the corresponding `region_node` to the `surface_template`, and this will be used during playback to play media items in the correct location.

- A `timegraph` is created. This is the internal representation of the <`body`> part of the DOM tree that will be used to play back the document. In addition a `scheduler` is created, which will interpret the data in the `timegraph`.

2.4 Starting playback

When the user selects Play we call the `start` method of the `player` object. This will invoke `start` on the scheduler. This will start playing the root node of the tree. The scheduler will now do all the SMIL 2 magic, whereby events such as the root node being played causes other nodes to become playable, etc.

At some point a media item needs to be rendered. The scheduler calls the `new_playable` method from the `global_playable_factory`. This will pass the DOM `node` to the various factories until one signals it can create a `playable` for the object. In addition, if the playable has a renderer (which is true for most media objects, but not for things like SMIL animations) we also obtain the `surface` on which the media item should be renderered, through the `layout_manager`. We then tell the renderer which surface to use.

Soon afterwards the `start` method of the playable is called to start playback. An average renderer will need to obtain data from some URL. It will do this by creating a `datasource` for the document through the `datasource_factory` object. Every time the renderer wants more data it calls the `start` method of the datasource passing a callback routine. Whever data is available the datasource will schedule a call to the callback routine, through the event processor. When the renderer has enough information to start drawing it will not actually draw immedeately, but it will send a `need_redraw` call to its surface. This will percolate up the surface hierarchy, to the GUI code, and eventually come back down as a `redraw` call all the way to the renderer (assuming it is not obscured by other media items, etc). At this point the bits finally get drawn on the screen.

Whenever anything "interesting" happens in the renderer (the media item stopped playing, the user clicked the mouse, etc) it invokes a corresponding method on its `playable_notification`. This interface is implemented by the scheduler, and these notifications are how the scheduler gets informed that it can start scheduling new things, etc.

3 Ambulant design, main objects

Last updated for Ambulant version 1.1.

3.1 Introduction

This document describes the function of various of the more important objects and interfaces in the Ambulant Player. If you haven't already done so it is probably a good idea to first read the Overall design and walkthrough documents. The first one explains the design principles and some of the choices made, the second one is a brief walkthrough of how the player loads, parses and plays a SMIL document.

The nitty-gritty details of these objects, on a level interesting to developers, are available too, in the API documentation. This document may need to be regenerated, see the README file in the documentation directory if it doesn't seem to exist.

3.2 AmbulantPlayer interfaces

- Refcounting protocol. A lowlevel interface shared by many objects.
- Player Interface. This is the toplevel object.
- Parser Interface. Describes the interfaces to the XML parser.
- Datasource interface. The interface used to get external data into the program.
- Playable interface. This interface makes media items appear on the screen.
- Layout interface. This interface is used to determine where those media items show up.
- GUI window interface. This interface is used to create new windows.
- Animation interface. The interface used for SMIL animation.

3.3 AmbulantPlayer Objects

- Clocks. These advance a virtual time.
- Event processor. This is the mainloop plus the event/callback mechanism.
- Document. The representation of a SMIL document.
- Node. The representation of the DOM tree.
- Transitions. Classes to do visual transitions.
- Timeline. This is a description of another scheduler: the MMS scheduler. This scheduler has a much simpler structure than the SMIL 2.0 scheduler.

You may notice that the core of the player, the SMIL 2.0 scheduler, is not mentioned here. Unfortunately it is not documented yet.

There are also additional low-level objects like `thread` and `critical_section` that are not described here. See the API documentation for details.

3.4 Refcounting protocol

The refcounting protocol is contained in the file `lib/refcount.h`. It needs to be implemented only by objects that are truly shared, i.e. any object whose lifetime is not predetermined by some other object. New instances of refcounted objects are created using the operator new. Any object that needs to share a particular instance calls `add_ref()` against this instance. The creator of the refcounted object and any sharer are responsible to call the `release()` method of the object when they don't need the object any more.

3.5 Player Interface

The player is the top-level object. When it is created you pass a DOM tree, a `factories` structure containing references to the `playable_factory`, `datasource_factory` and `window_factory` and an embedder object used for callbacks to the GUI (on state changes, opening of external documents, etc).

There is a UML diagram for player showing how the player relates to various other objects.

3.6 Player implementations

There are currently two implementations of the player interface: `smil_player` and `mms_player`. The first one is the all-singing-all-dancing SMIL 2.0 player, the second one can play MMS documents, which use a very restricted subset of SMIL 1.0.

A concise walkthrough of how the `smil_player` operates is given in the walkthrough document.

3.7 Parser interface

The XML parser roughly follows a SAX interface. To use it you provide it with objects having the `sax_content_handler` and `sax_error_handler` interfaces. You then feed your document to the parser and it will call back through those interfaces.

There is a UML diagram for parser showing how these classes relates to each other.

3.8 Parser implementations

There are currently two parser implementations, `expat_parser` uses James Clark's expat parser, which is a fast and lean no-frills parser. `xerces_parser` uses the Apache Xerces library, which is at the other extreme of the spectrum: it can do document validation with both DTDs and Schemas and lots of other wonderful things. But this comes at the price of a rather hefty memory footprint.

3.9 Datasource interface

There are actually a couple of these interfaces, but they are similar. Their function is to implement URL retrieval schemes or file I/O and get external data to media handlers and other modules requiring data access.

The general interface is that a datasource is acquired through a `datasource_factory` interface, which passes the URL to the various implementations until one is found that can handle it and returns a `datasource` object. The client then calls the `start` method on this object, passing a callback routine, and the datasource will arrange for the callback to be called as soon as data is available. No new callbacks will be done until a new call to `start()` is made, and the datasource has a buffer that can be limited, so this design allows for flow control over the net, if required.

There are specialised datasource interfaces for audio and video, that can handle extra things like converting audio from mp3 format to linear samples, or demultiplexing an audio/video stream.

There is a UML diagram for datasource showing how these classes relates to each other.

There is also a UML state diagram for datasource showing the state machine that a datasource should adhere to.

3.10 Playable interface

The `playable` interface is implemented by what are usually called a media handlers or media renderers: it is this interface the scheduler uses to make things appear on the screen (or sound out of the speakers, or otherwise do their thing).

Playables are created through `global_playable_factory`, which has references to all playable implementations and asks them in order whether they can handle playback of this specific DOM node, until one matches.

When the playable is started it is provided with a `playable_notification` object (implemented by the scheduler), which is where it can send its status messages (such as `stopped()` when the media is finished, or `clicked()` when the user clicks the mouse over the media item).

Most playables have an accompanying interface, `renderer`, which controls where the media item is rendered (non-rendering items such as SMIL animations are the exception to this rule).

There is a UML diagram for playable showing how these classes relates to each other.

There is also a UML state diagram for playable showing the state machine that a playable should adhere to.

3.11 Playable implementations

While some media handlers implement the playable from scratch (an example is the aforementioned SMIL animation handler) there are a number of convenience classes that implement functionality shared by many media handlers. These are:

- `playable_imp` which handles some bookkeeping having to do with the `playable_notification` and `event_processor`.
- `renderer_playable` which augments that with a `renderer` interface and the bookkeeping required for that.
- `renderer_playable_ds` which adds `datasource` creation and bookkeeping to that.
- `renderer_playable_dsall` which builds on that again and collects all data before requiring further action.

There is a UML diagram for renderer_playable showing how these classes relates to each other.

3.12 Layout interface

The layout manager determines where media items appear, and also governs things like z-ordering, background colors for regions and such.

The central interface is the `surface`, which is the object passed to a `renderer`. Whenever a renderer has something new to show it calls `need_redraw()` on this interface. Whenever it is time to actually redraw something the `surface` calls `redraw()` on the `renderer`.

The layout_manager interface maps DOM nodes to the surface objects on which they should play back.

There are two more auxiliary interfaces that are not strictly necessary but used by the layout implementation for historical reasons: surface_template and surface_factory. These interfaces are used to create subregions and toplevel windows, respectively.

There is a UML diagram for layout showing how these classes relates to each other.

3.13 Layout implementation

The SMIL 2.0 implementation of surface, surface_template and surface_factory are the classes passive_region and passive_root_layout.

There is a UML diagram for region showing how these classes relates to each other.

3.14 GUI window interface

This is the abstract interface used to create new windows and tie them to the layout implementation. The implementation is machine-dependent, obviously, and usually supplied by the hosting application.

In addition there is the gui_events interface which goes the other way: it is exposed by the layout implementation, and used by the machine dependent window implementation to communicate things like redraw requests.

There is a UML diagram for window showing how these classes relates to each other.

3.15 Animation interface

The animation interfaces are animation_destination and animation_notification. The SMIL 2.0 playable uses these interfaces to change parameters and send notification of those changes, respectively.

There is a UML diagram for animation showing how these classes relates to each other.

3.16 Clocks

All clocks adhere to the abstract_timer interface. This interface allows you to get the current time and set the speed of the clock.

There is a companion interface abstract_timer_client (which is actually a base class of abstract_timer) that allows objects to get notification of changes in timer speed.

Currently there are two implementations of the abstract_timer interface: the operating-system specific realtime clock (of which you cannot set the speed) and timer, which implements a new zero-based clock based on another abstract_timer. Its speed is settable with set_speed, but it is tightly synchronized with its parent clock.

Eventually there may be other implementations of timer, such as clocks that are allowed to slip synchronization and other such semantics as required by SMIL.

There is a UML diagram for clocks showing how these classes relates to each other.

3.17 Event processor

The event_processor is the low-level scheduler of the system. It is a priority runqueue with methods to add callbacks, with an optional delay until the callback becomes elegible.

There is a UML diagram for event processor showing how these classes relates to each other.

3.18 Document

The document class contains the DOM tree and some auxiliary data:

- the node_context which can be used to lookup nodes by XML ID and to resolve relative URLs, and
- the nscontext which stores information on the use of XML namespaces.

There is a UML diagram for document showing how the document class relates to various other objects.

3.19 Node

The `node` class represents a node in the DOM tree. Actually, our tree isn't 100% compatible with DOM, but close enough. The `node` objects store the tag, attributes and data pertaining to the XML node. There are basic methods to access the parent, next sibling and first child, to insert or remove nodes into a tree and more.

There are two auxiliary classes that augment the node functionality using only these interfaces:

- `node_navigator` uses the basic `up()/down()/next()` methods of node to build more complex navigation.
- `node_iterator` is an iterator that allows you to iterate over a subtree.

There is a UML diagram for node showing how these classes relates to each other.

3.20 Transitions

The implementation of SMIL transitions is fairly complex, because there are very many transition types and they also need to be implemented efficiently on multiple platforms. The current implementation is fully based on inheritance, a model with delegation would probably result in a cleaner design.

The central object is the `transition_engine`, which also supplies to interface used by clients. It has `begin()`, `end()`, `step()` and `next_step_delay()` methods, which the client (usually a media renderer) uses to control the transition. `next_step_delay()` needs an explanation: it returns the delay until the engine would like to get the next call to `step()` from the renderer.

On top of this central object there is a multiple inheritance graph where one leg is machine dependent, and does the actual bitblit operation to combine two images. The other leg is machine independent, but dependent on the actual transition type, and computes the parameters for the bitblit. These two then come together in a stub class that has all the functionality for a specific transition type on a specific platform.

There is a UML diagram for transitions showing how these classes relates to each other.

4 Ambulant source code organisation

Last updated for Ambulant version 1.1.

4.1 directory organisation

Everything is kept under CVS, on sourceforge. On Unix we use the standard *automake*, *autoconf*, *configure* and *gcc* (version 3.2 or later) toolset to build things. On Windows we use Visual Studio 7. For cross-compilation for Windows CE we use Embedded Visual C++ 3.0. For cross-compilation for the Zaurus Linux handheld we use the Sharp toolset, based on gcc 2.95.

At the toplevel we have a number of subdirectories:

- `Documentation` will eventually contain all documentation, currently `Design`, which is what you are reading now, and `API`, which is low-level documentation of classes, methods, etc.
- `include` has all the C++ header files.
- `src` has all the sources, with the engine built as a library from the `libambulant` subdirectory, and driver programs in `player_macos`, `player_mfc`, etc.
- `third_party_packages` holds the source to third party packages we use, such as James Clark's `expat` XML parser, the IJG libjpeg library and more.
- `Extras` has the two included presentations: Welcome and the DemoPresentation on New York.
- `po` has the files used to create localisations. This uses the GNU gettext library.
- `m4` has extra macros for automake.

4.2 Source code conventions

Here's a somewhat random list of source code conventions that we have decided to use:

- Indent 4 spaces, with the following exceptions:
  - `namespace` doesn't indent at all
  - `public:`, `private:` and such indent two spaces

- Whether opening braces are at end-of-line or beginning of the next line depends on the circumstances and personal taste. Closing braces, when on a line by themselves, must however align with the construct that opened them.
- no camelCase, CamelCase or Capitalization in class or variable names
- Underscores to delimit words
- attribute names start with "m_"
- static attributes start with "s_"
- we have a toplevel namespace ambulant, with a second level of namespaces under that
- semi-private classes go into into a namespace named "detail".
- Template type parameters start with an upper case letter (as in: `template <class A> {}`)
- Header files need to include any header files on which they depend, and they guard against multiple inclusion with a preprocessor construct.
- Header files are all in an "ambulant" directory, and are included by full path, as in:

  ```
  #include "ambulant/net/url.h"
  ```

- Everything goes into namespace "ambulant", with sub-namespaces "lib", "net", etc. Machine-dependent code goes into it's own "unix", "win32", etc subnamespace of those.
- Source files have `using namespace ambulant;` at the top. In addition, there's a using namespace for your own namespace, i.e. `using namespace common;` for source files in common. Normally, there are no other global `using namespace` declarations, i.e. everything outside of your own namespace is used qualified.
- Header files that declare abstract interfaces try to include as few other header files as possible. In other words, if you need a `lib::node *` in an abstract header file it is better not to include `ambulant/lib/node.h`, but instead to add a construct:

  ```
  namespace ambulant {
  namespace lib {
  class node;
  }
  }
  ```

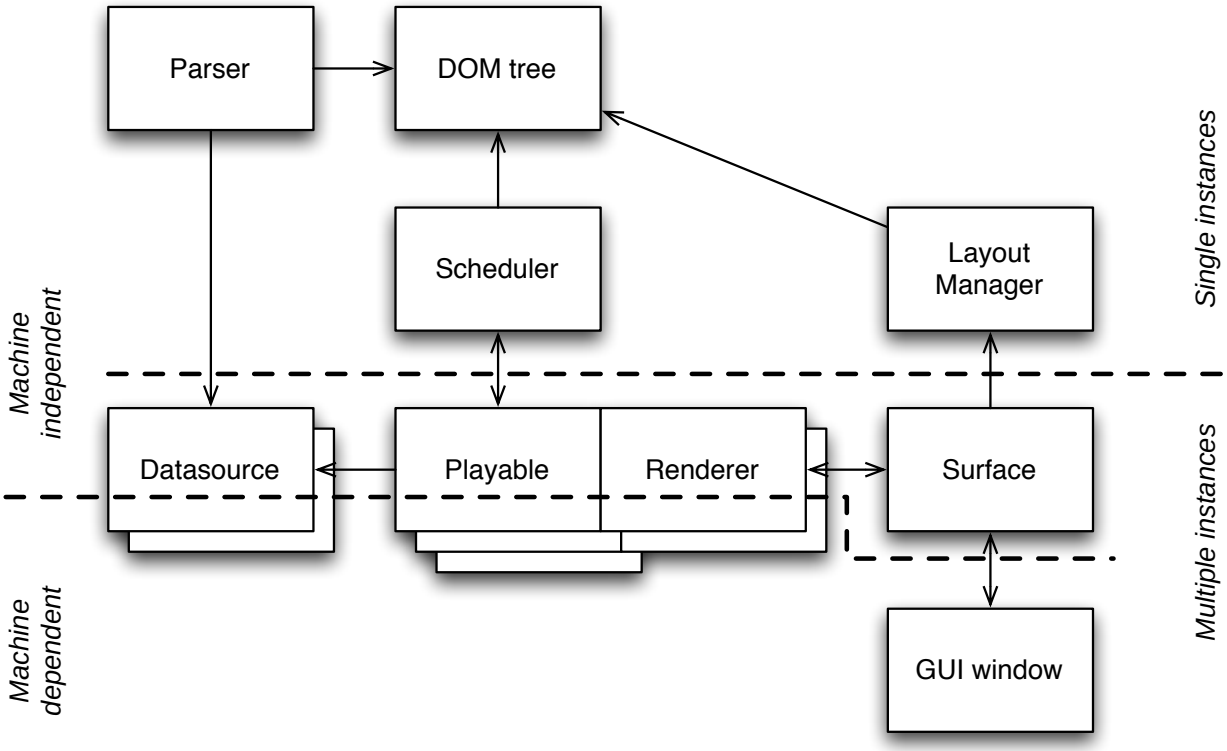
- We need to define which preprocessor defines we are going to switch on for platform-dependent code, so we don't get #ifdef _WIN32 in one place and something completely different in another place. Suggestion:
  - #ifdef _WIN32 to test for Windows
  - something to test for Linux?
  - #ifdef __APPLE__ to test for MacOSX
  - something to test for 386 or other architectures?
  - something to test for Visual Studio versus gcc?
- APIs that could be considered external must be documented inline, in a form compatible with Doxygen. The easiest way to do this is with a triple-slashed comment block. The first line should end in a period and is the short description. Anything after that is the long description:

  ```
  /// Hold user preference information.
  /// This class holds all settings the user can change. It is normally subclassed
  /// in machine-dependent code to override the load() and save() methods.
  class preferences {

          /// Load preferences from disk.
          void load();

          /// Save preferences to disk.
          void save();
  }
  ```

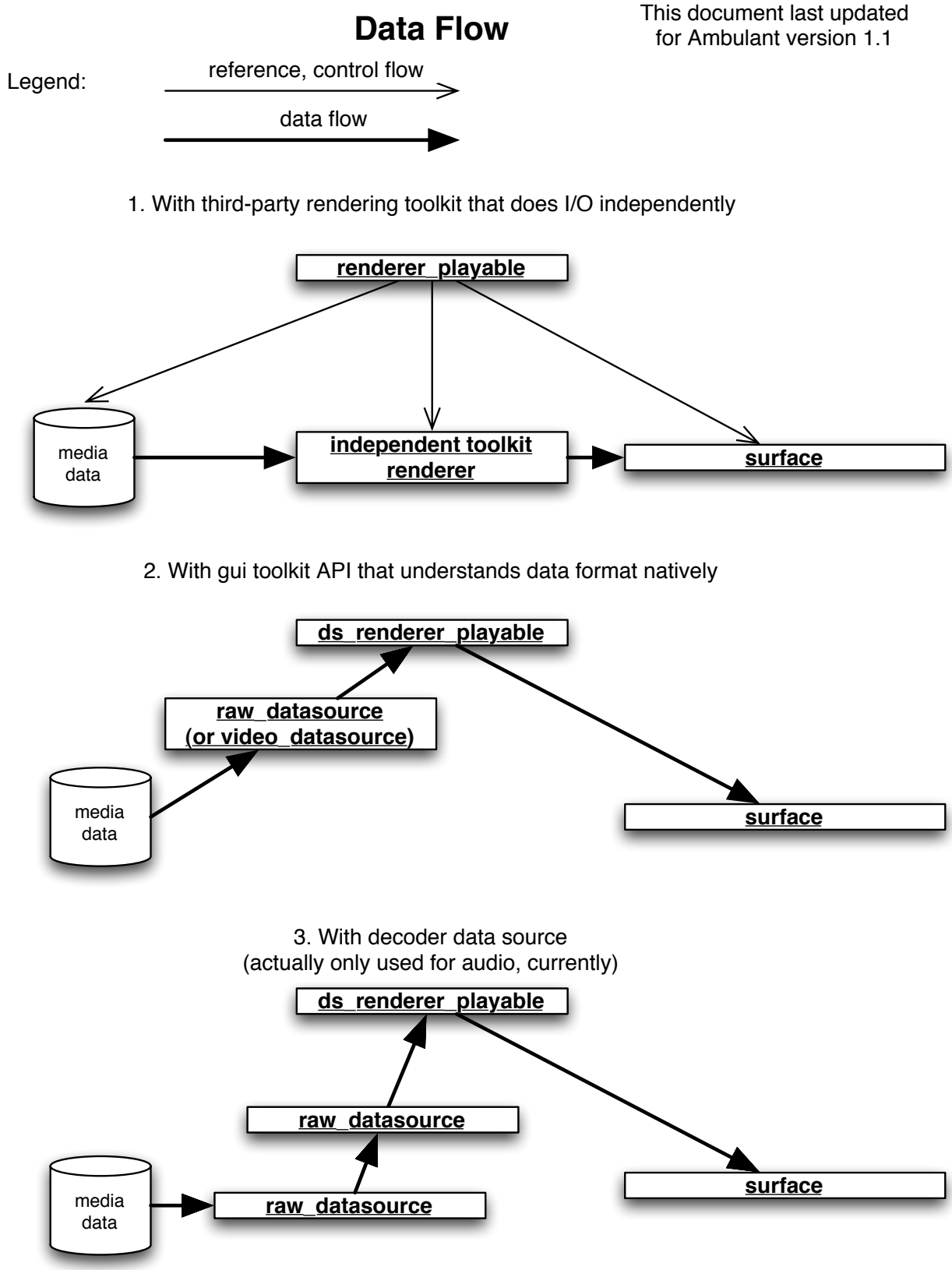
5 Diagrams

This section has UML diagrams (and some non-UML diagrams) that explain the interaction between various parts of Ambulant Player.

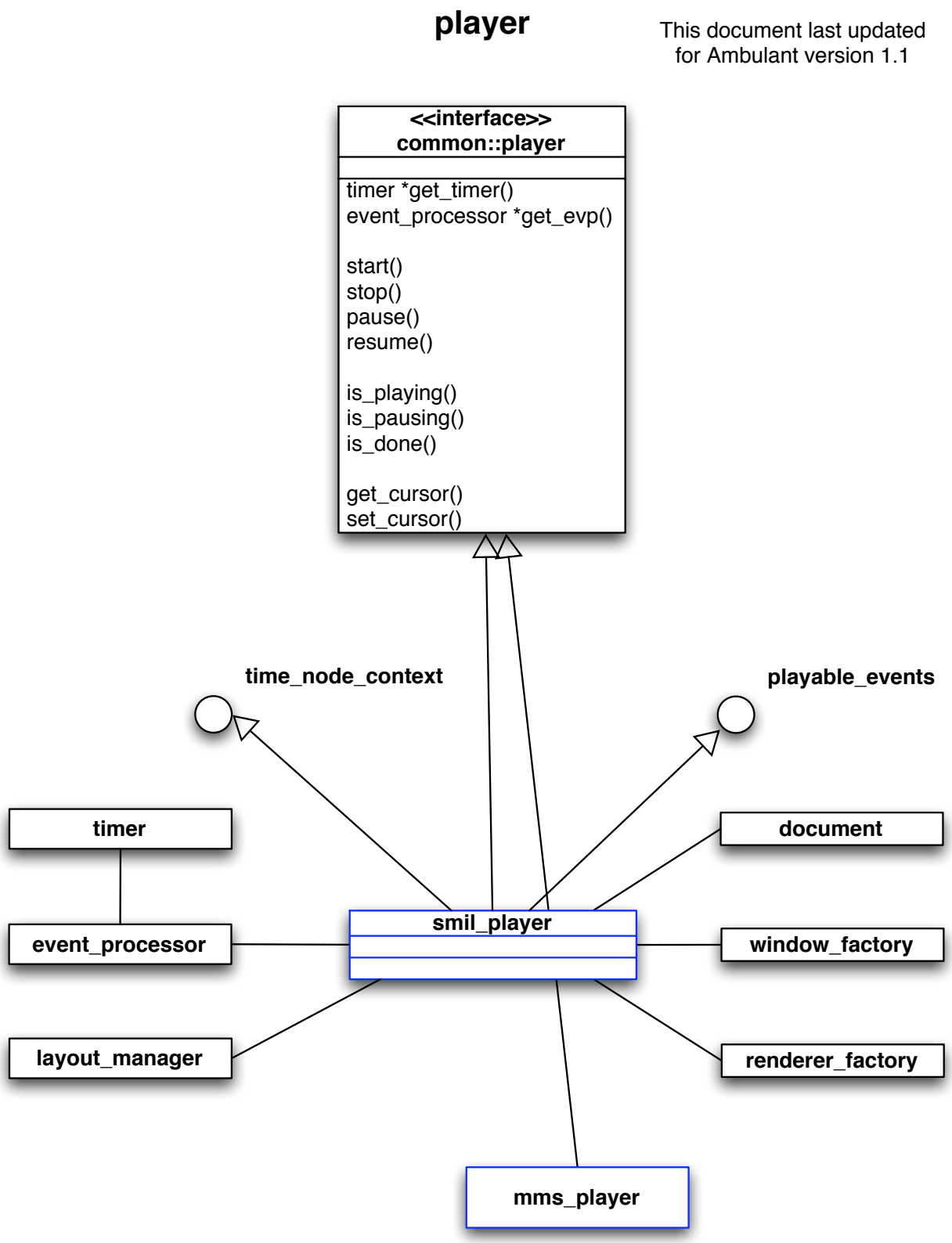
5.1 overall structure diagram



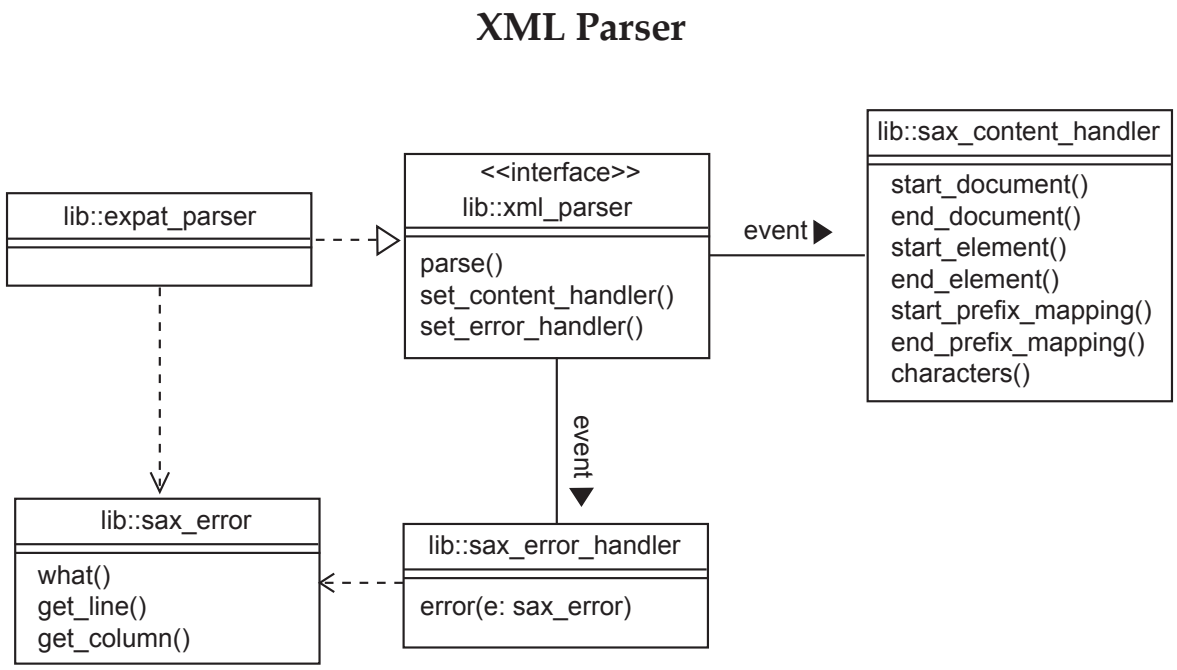
5.2 dataflow diagram



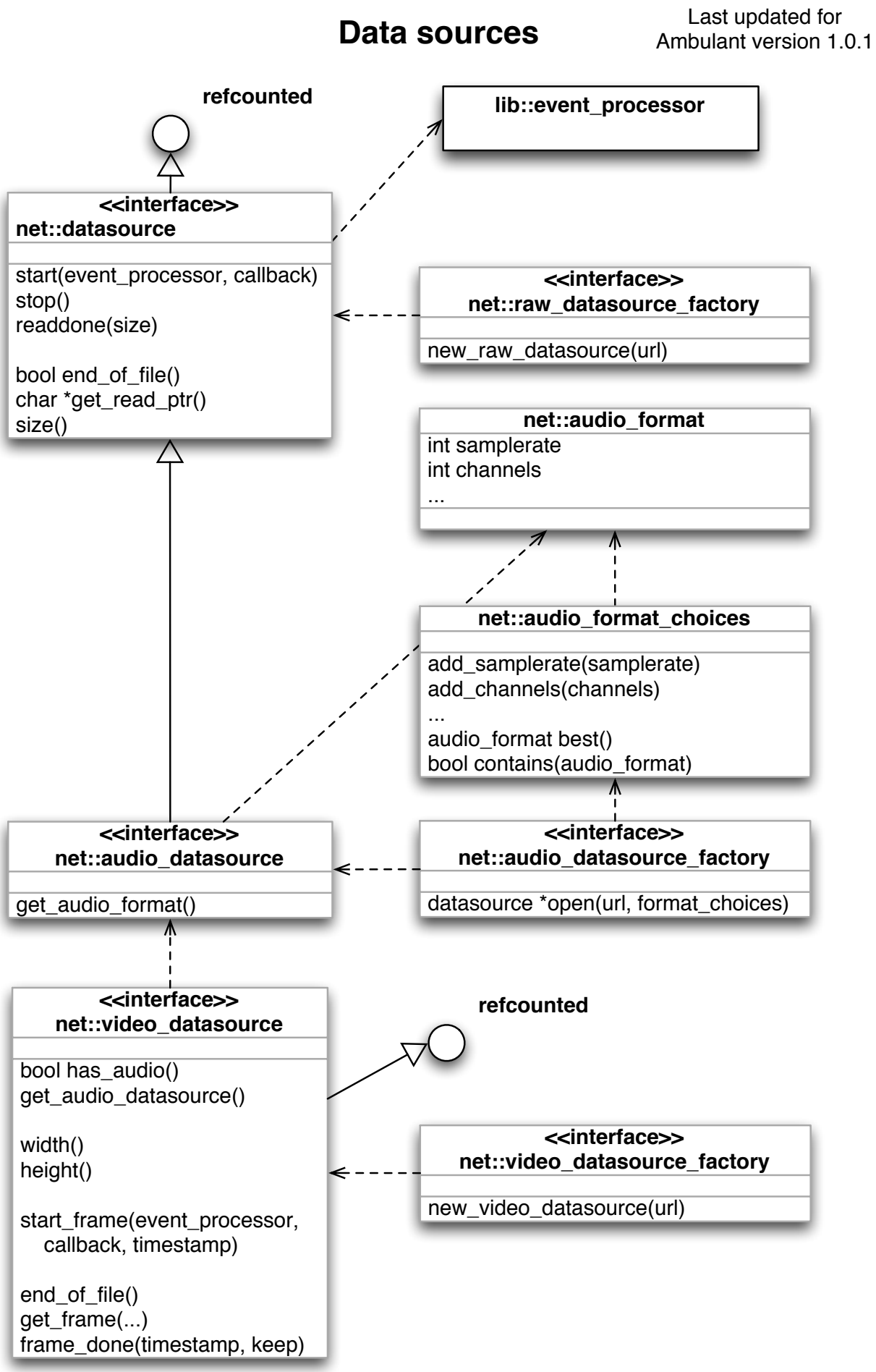
5.3 UML diagram for player



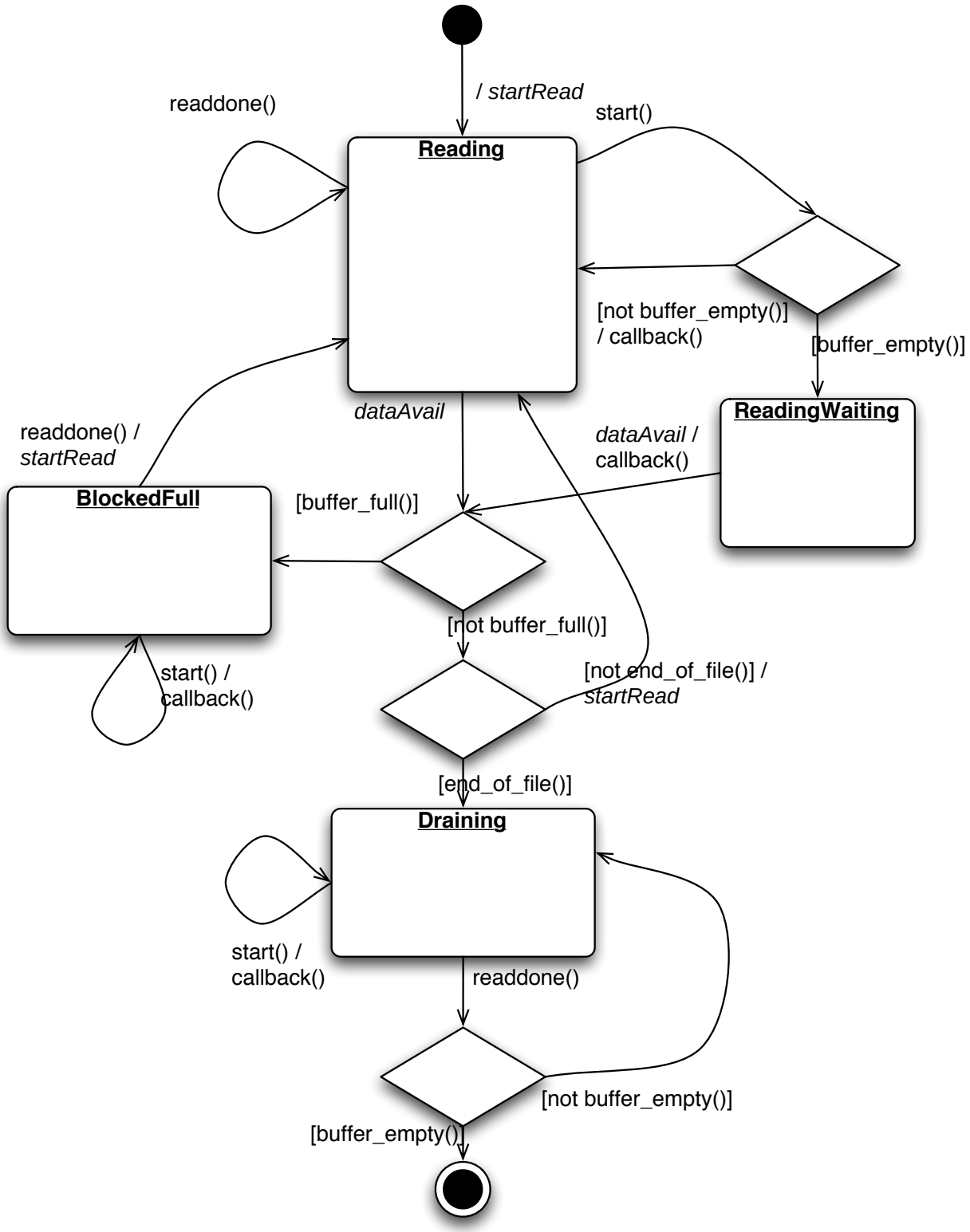
5.4 UML diagram for parser



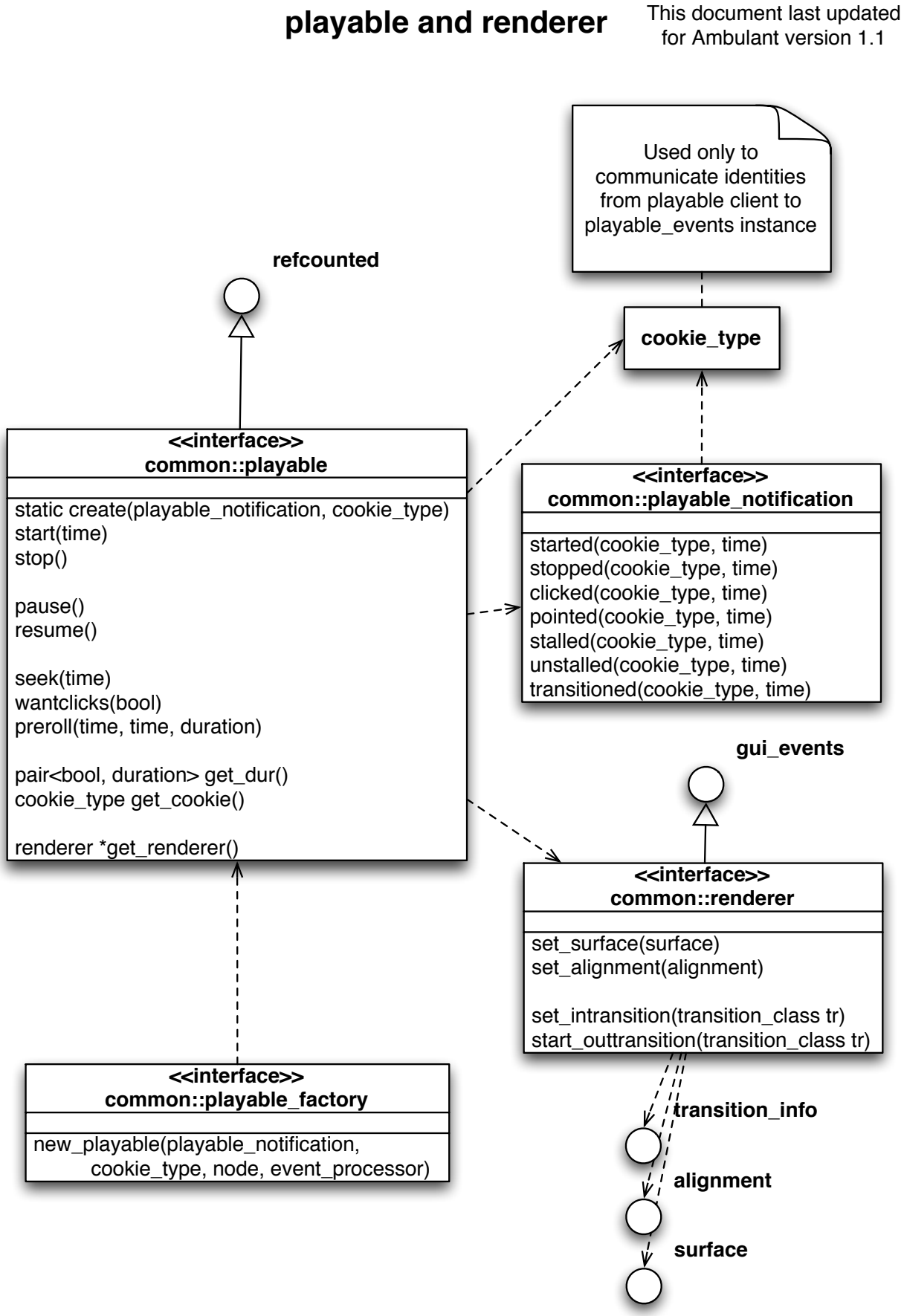
5.5 UML diagram for datasource



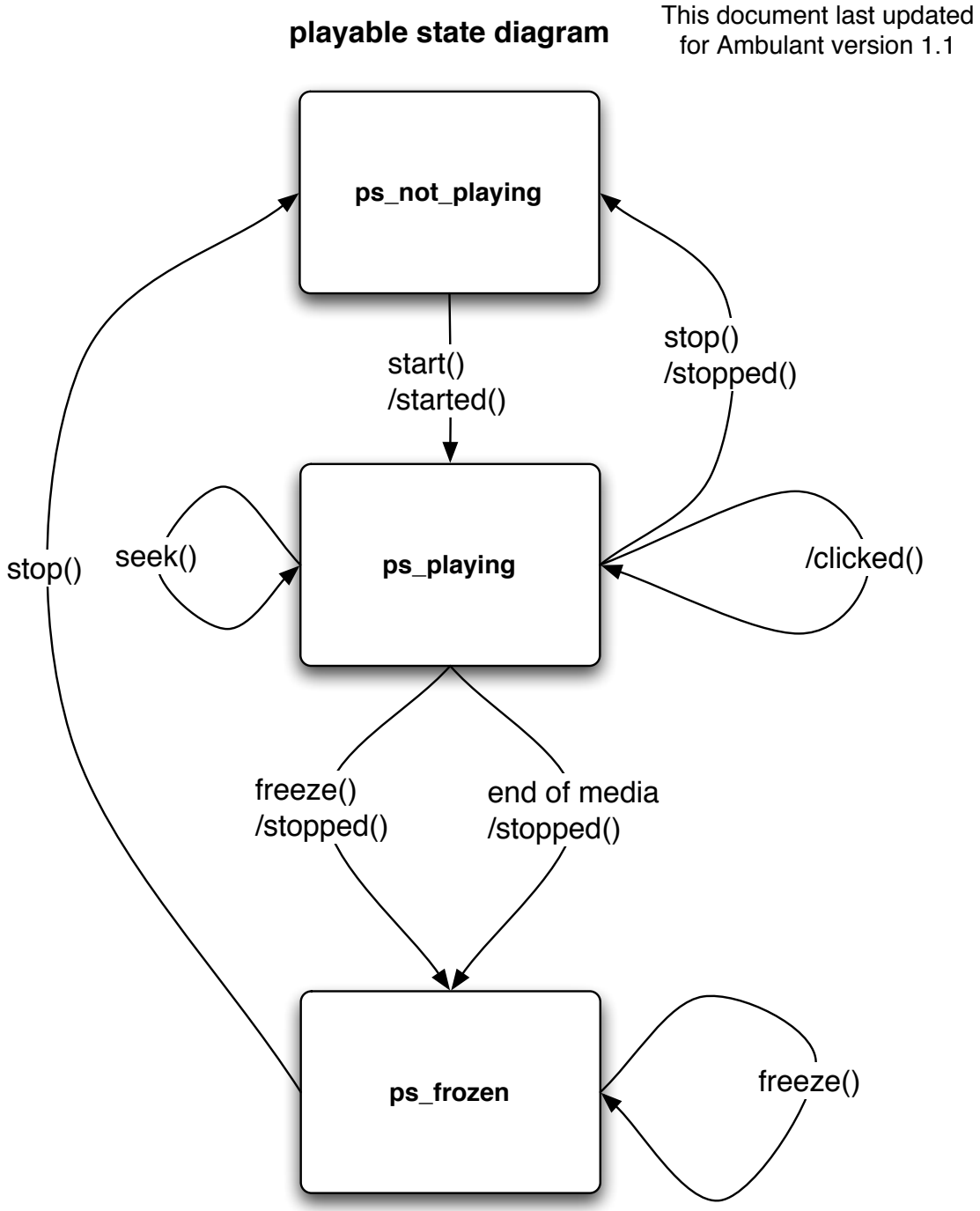
5.6 UML state diagram for datasource



5.7 UML diagram for playable

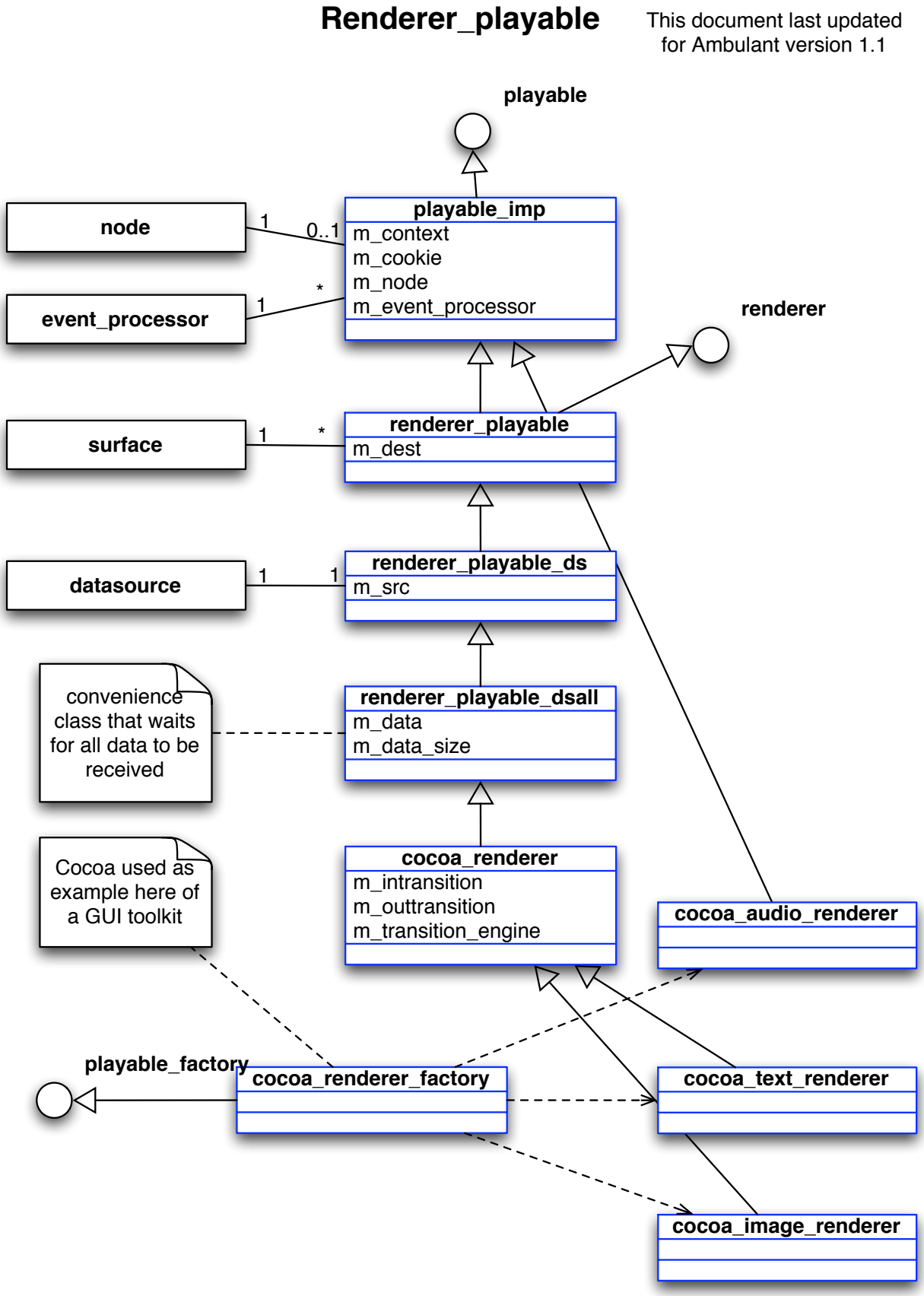


5.8 UML state diagram for playable

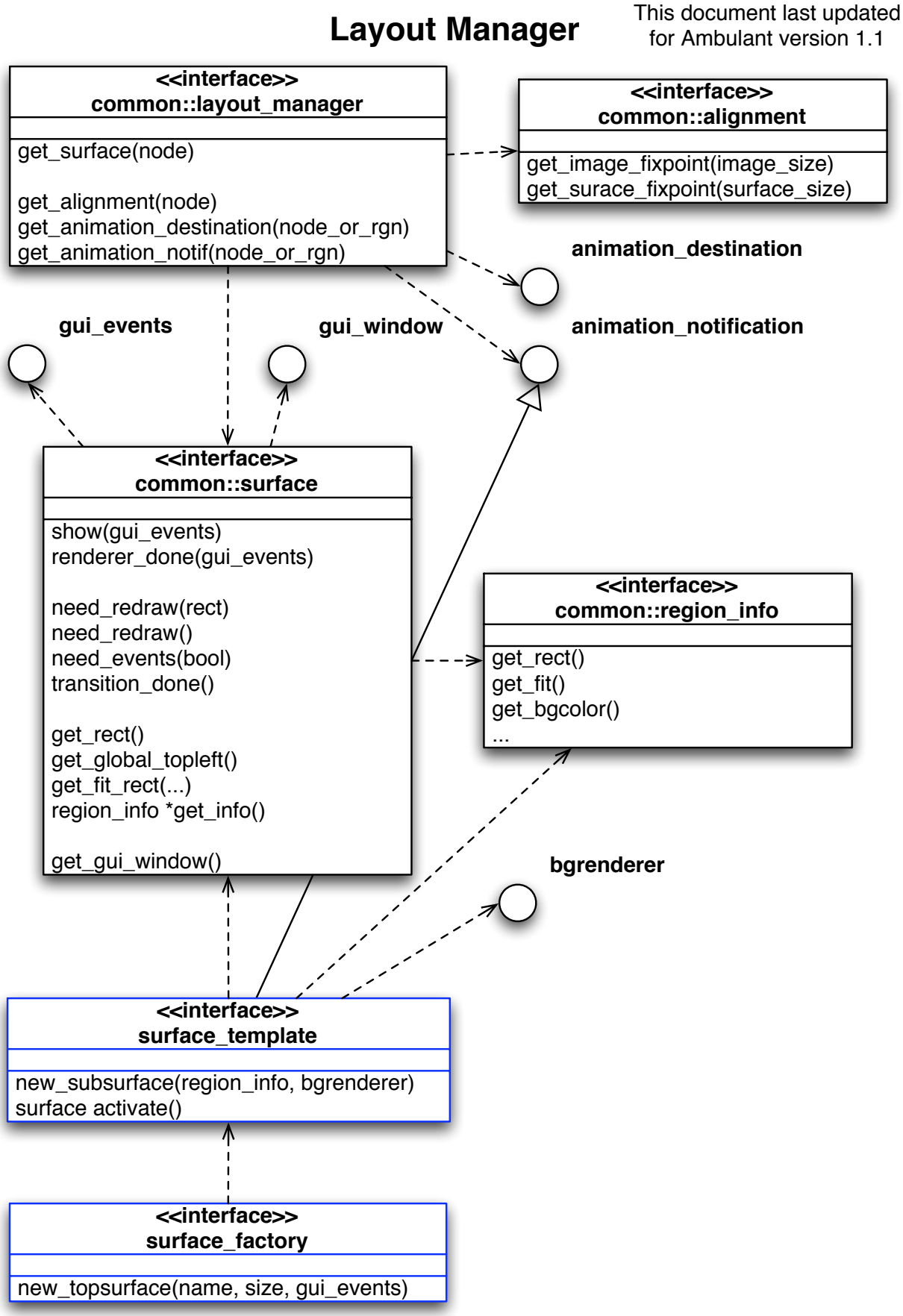


Notes:
1. freeze() **is not implemented, maybe it is not needed?**
2. pause() **and** resume() **behavior is not modeled here**

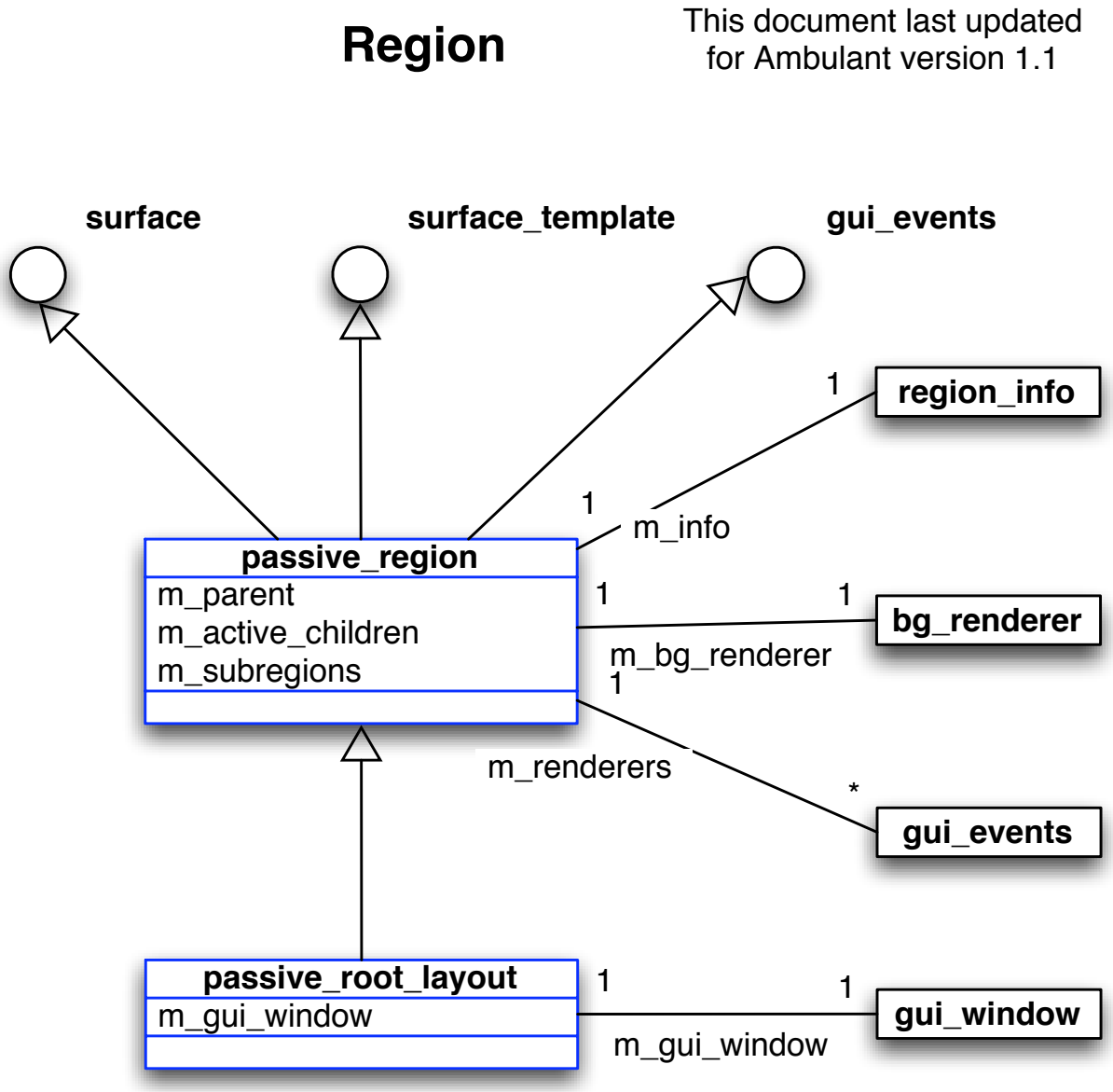
5.9 UML diagram for renderer_playable



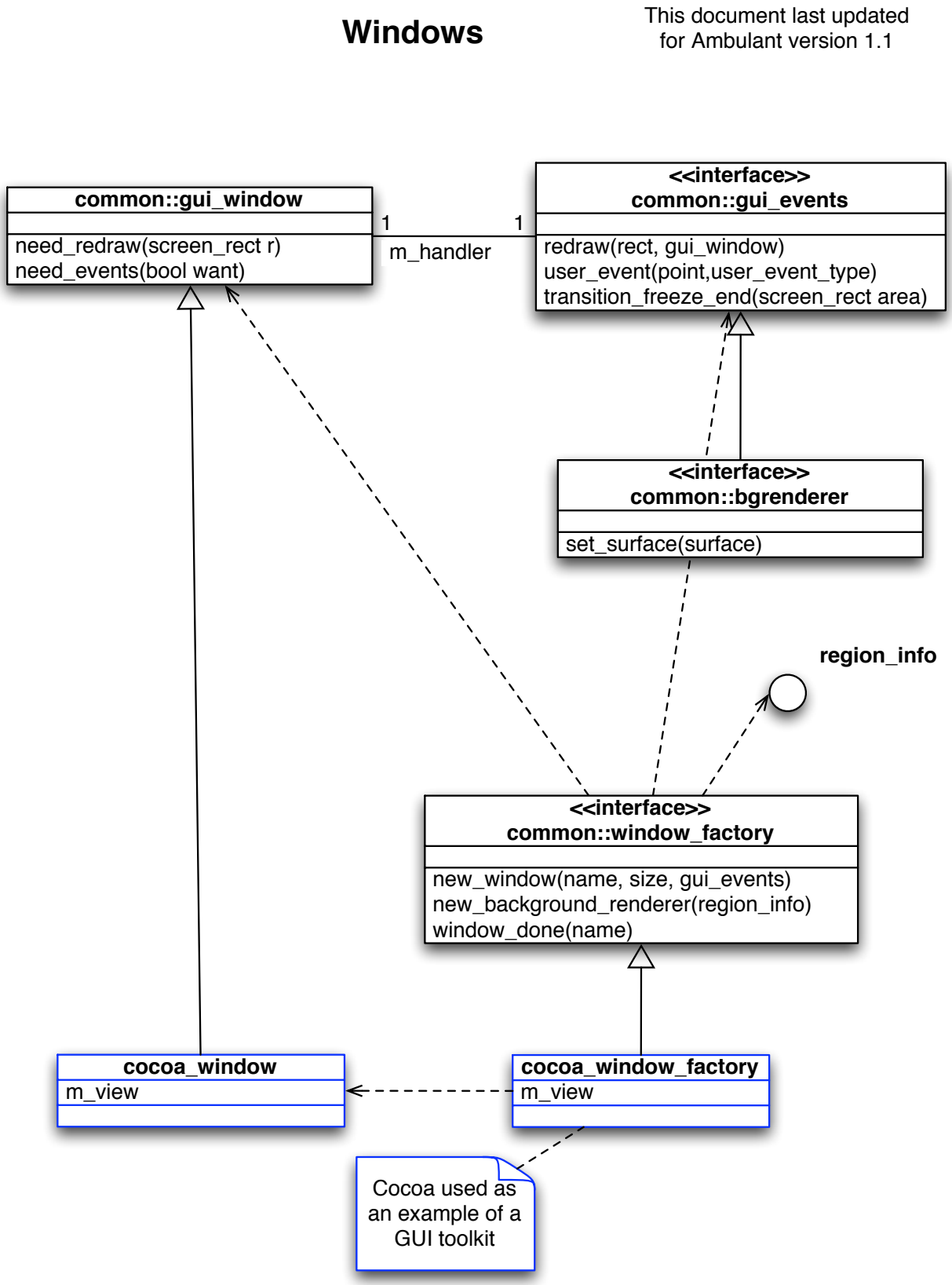
5.10 UML diagram for layout



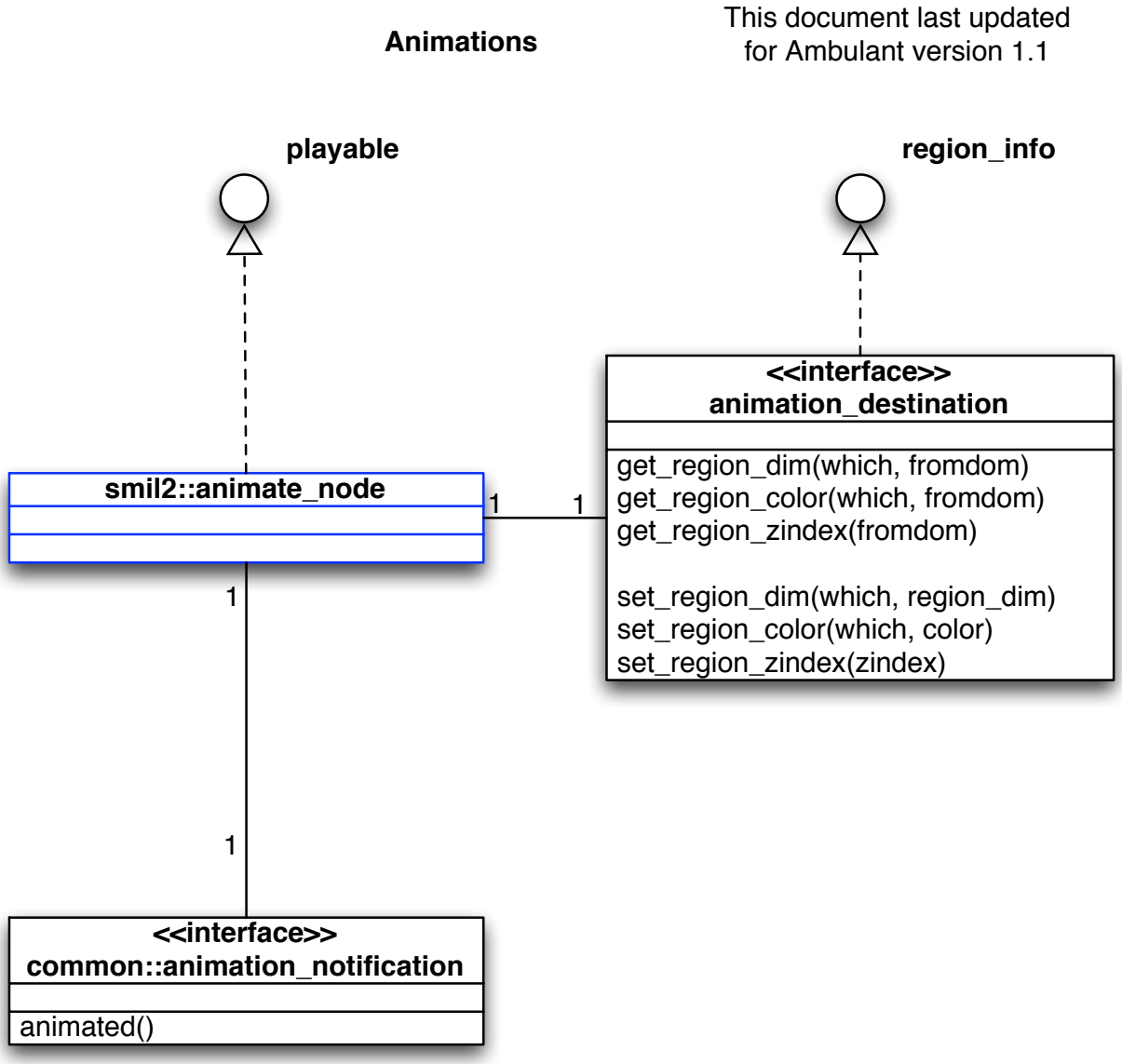
5.11 UML diagram for region



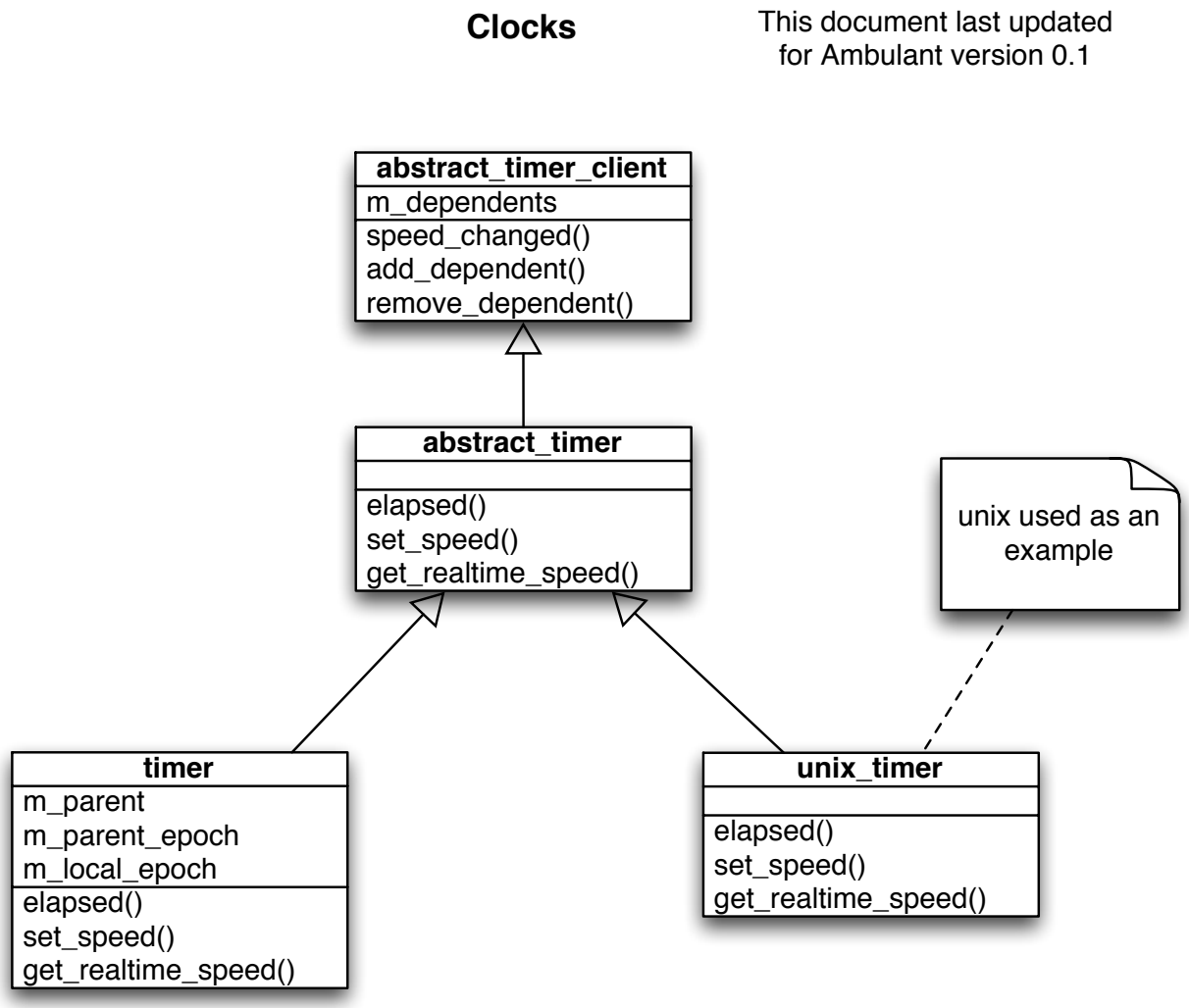
5.12 UML diagram for window



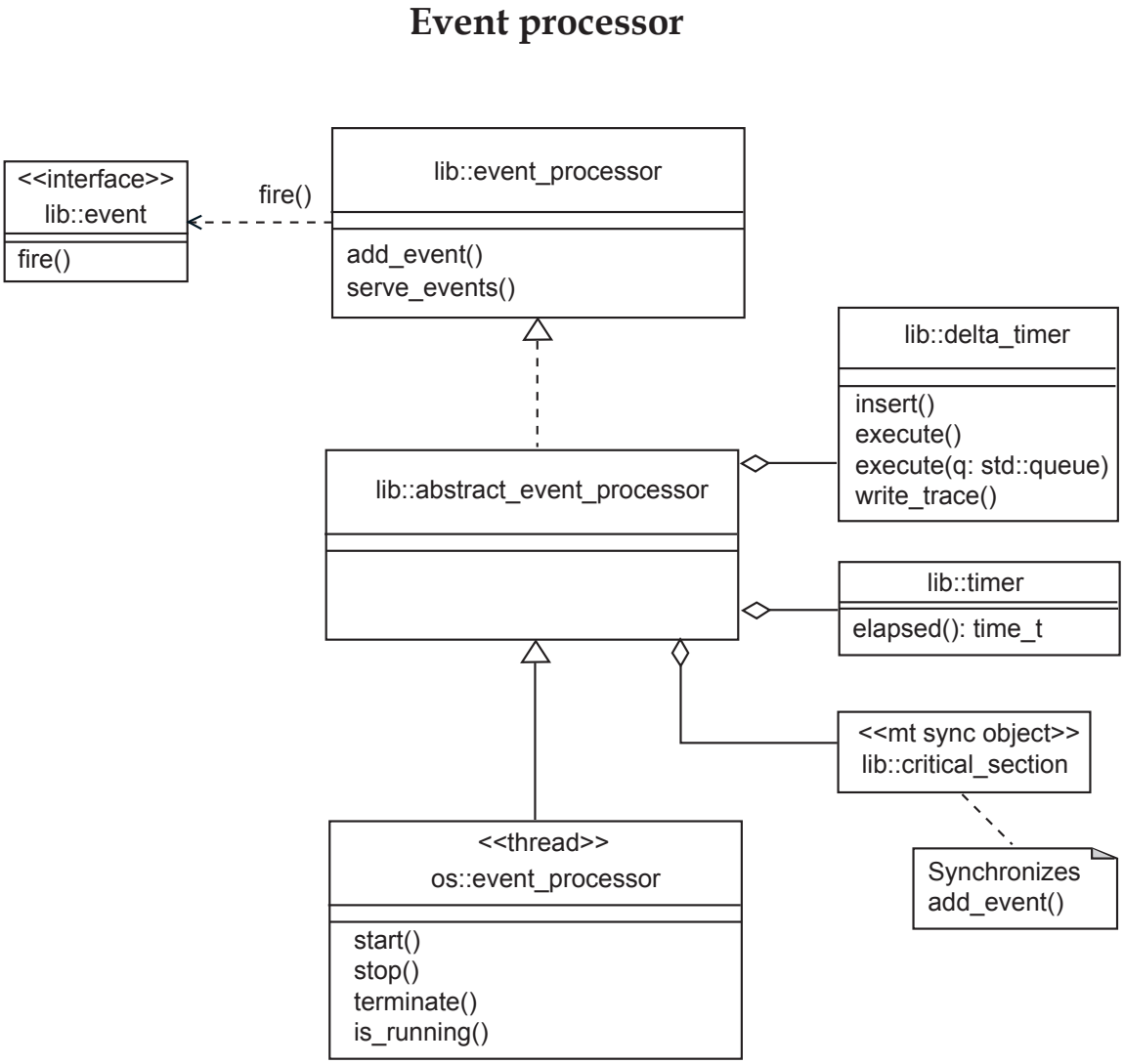
5.13 UML diagram for animation



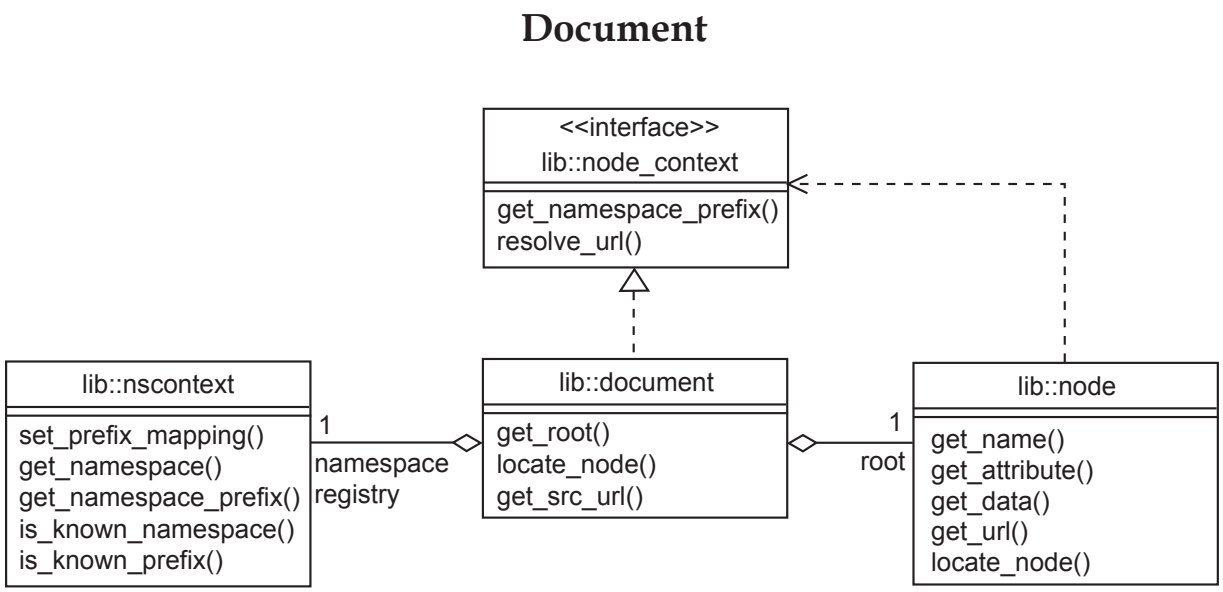
5.14 UML diagram for clocks



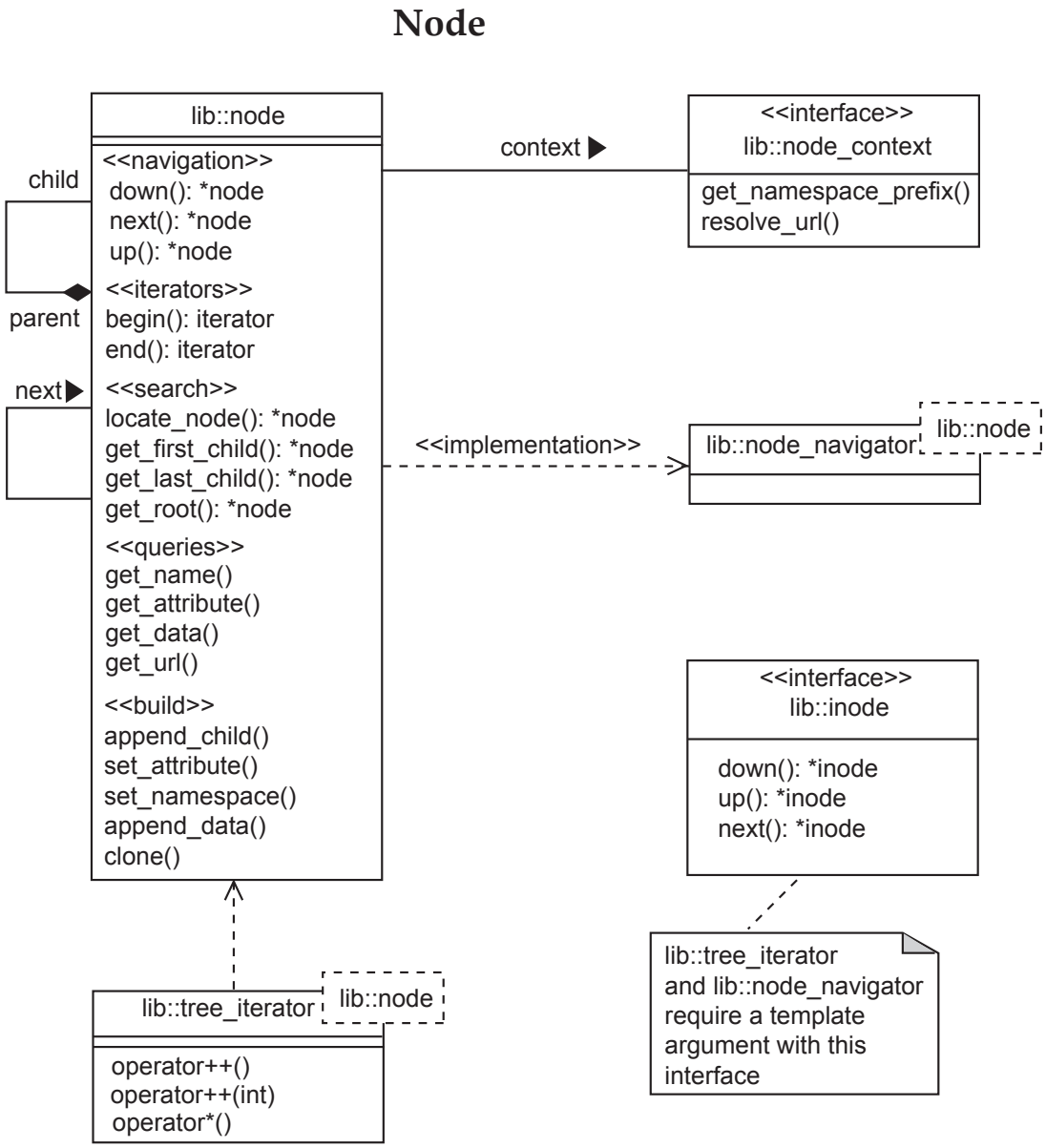
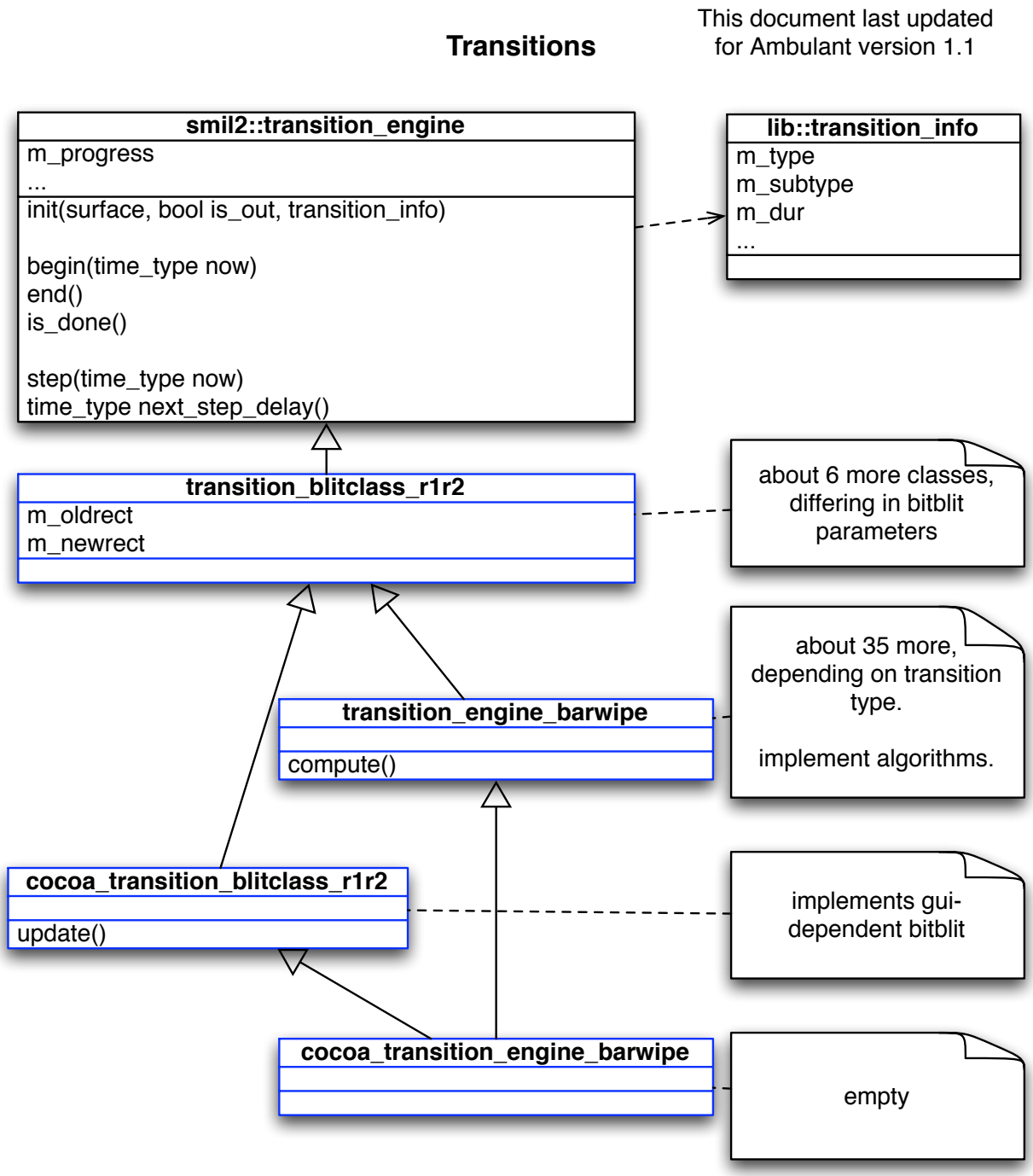
5.15 UML diagram for event processor



5.16 UML diagram for document



5.17 UML diagram for node



5.18 UML diagram for transitions