Table of Contents

Overview 1

Generic Components 2
Glass Integration 2
Window Management 2
Input Device Capabilities 2
Input 2
Platform 3

Generic Linux Port 4
Native Interfaces 4
Implementations 4
Input 5

OMAP Port 6

MX6 Port 7

X11 Port 7

OMAPX11 Port 7

Headless Port 8

Native interfaces 8

Overview

Monocle is a proposed rewrite of Lens, the Glass implementation for systems that do not have an underlying window manager. Unlike Lens, in which most of the work is done in C, Monocle will be almost all Java code. Where required, interactions with C code will be through simple Java-C bindings where the logic is in Java and only the low-level system or library call is in C. The goals of Monocle are:

- Functional and quality parity with Lens on all platforms: Embedded Linux/ARM, Android and headless
- Simplify the process of porting to a new platform.
- Improve maintainabiity and debuggability of the embedded ports. Where possible, simplify the code.
- Pluggability of Linux input device handlers

Like Lens, Monocle assumes it has full control of the screen and does not have to cooperate with other graphical applications. It does not rely on an underlying window system.

Generic Components

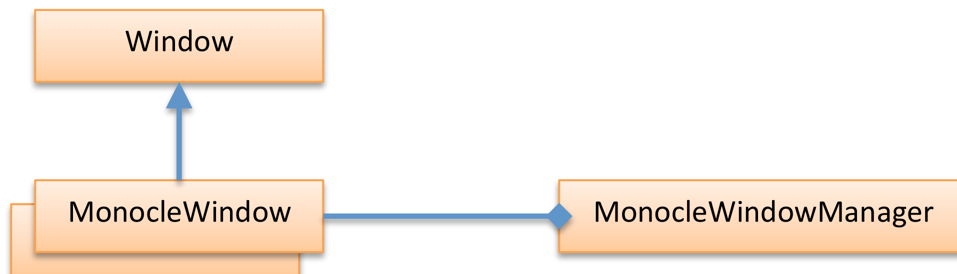
Glass Integration

The classes named Monocle* integrate with class. So MonocleApplication, MonocleWindow and MonocleView extends the Glass classes Application, Window and View.

All code running at this level is on the application thread.

Window Management

Monocle windows do not exist at a native level. Window state is held in MonocleWindow, while MonocleWindowManager maintains the Z-ordering of the window stack and assigns ID numbers to MonocleWindows.



All window management code runs on the application thread.

Input Device Capabilities

An InputDevice represents a single device that can generate input events. An InputDevice can report on its input capabilities. For example, it can declare itself as a multitouch screen or as a 5-way keypad. InputDevices are registered with an InputDeviceRegistry; MonocleApplication listens on changes to the InputDeviceRegistry to get notification on what classes of devices are attached.

Implementations of InputDeviceRegistry are also responsible for making sure input devices are recognized and their events delivered.

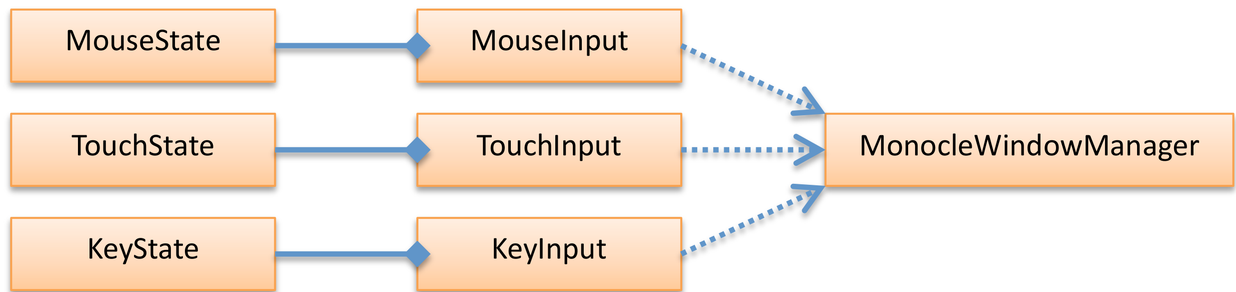
All generic input device code runs on the application thread. Platform-specific InputDeviceRegistry implementations can contain code that runs on other threads.

Input

Three input handler classes process input of different kinds: MouseInput, TouchInput and KeyInput. Each of these maintains its own state in an input state class: MouseState, TouchState or KeyState. When the one of the input handler is notified of a change to the input state it generates events accordingly based on the current state of the window stack and input focus. Low-level input classes do not communicate directly with the window stack.

Each input handler contains a single input state object. When the input handler is notified of an

input state change, it receives a state object containing the new state. The input handler does not store this external state object, but copies its contents into its own records. This helps us to minimize object creation during event processing.



All generic input code runs on the application thread.

Platform

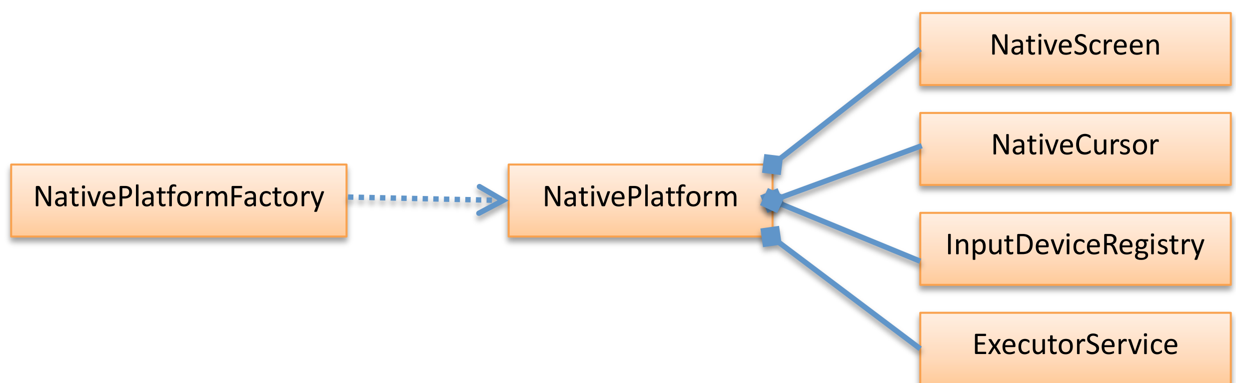
The platform-specific components are: NativePlatform, NativeScreen, NativeCursor and InputDeviceRegistry.

NativePlatform is instantiated by NativePlatformFactory. NativePlatformFactory looks at the system property `monocle.platform` to get an ordered list of factory classes to attempt to use. NativePlatformFactory then instantiates these factory classes, querying each in turn whether it can support the current platform it is running on. When a matching NativePlatformFactory is found, its corresponding NativeFactory will be created.

NativePlatform provides a single-threaded java.util.concurrent.ExecutorService. This ExecutorService is the application thread.

NativeScreen is instantiated by the NativePlatform. NativeScreen reports on the physical characteristics of the screen. It is possible that this class will be used by Prism as well; in this case the class will have to be thread-safe.

NativeCursor is instantiated by the NativePlatform. NativeCursor is responsible for updating the visible cursor state, using a platform-specific hardware cursor where possible. NullCursor is an empty implementation of this that does not display a cursor.



NativePlatform attempts to determine the best platform to use. Use of a specific platform can be forced by setting the system property `monocle.platform` to one or more of the following values in a comma-separated list:

`Linux` – the generic Linux port, using low-level input device nodes for input and no hardware cursor

`OMAP` – for the BeagleBoard xM using EGL/framebuffer with Linux device input and a hardware cursor

`MX6` – for the Freescale i.MX6 using EGL/framebuffer with Linux device input and a hardware cursor

`OMAPX11` – for the BeagleBoard xM using EGL/X11 with Linux device input and a hardware cursor

`X11` – for generic Linux/X11 platforms, using X11 for input and cursor.

`Headless` – for when you want to run with neither input or output

Other platforms that are not implemented yet but should fit into this system include: `Android`, `Headless`, `Dispman` and `MX6`.

Generic Linux Port

Native Interfaces

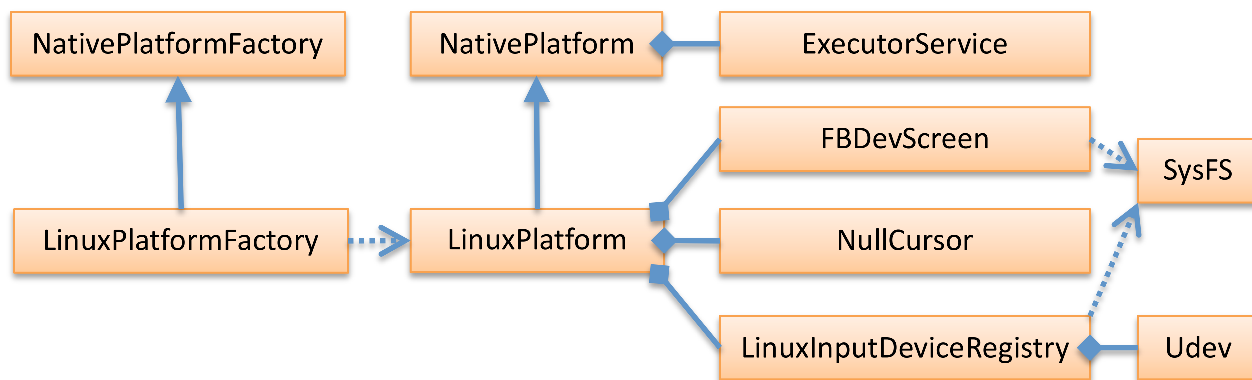
Most of the interaction between the Linux port of Monocle and the OS level is using the pseudo-filesystem under /sys. This is accessed using utility methods in the class SysFS. This is sufficient to read screen and input device characteristics and to request notification on what input devices are attached.

The Udev class is an interface to the Linux udev monitor to get notification when devices are attached and removed from the system. This requires some C code, since Java does not have an API for connecting to Unix domain sockets.

it might be necessary to add another native interface to use ioctl calls to read absolute axis range data for touch screens. This information does not seem to be available in sysfs.

Implementations

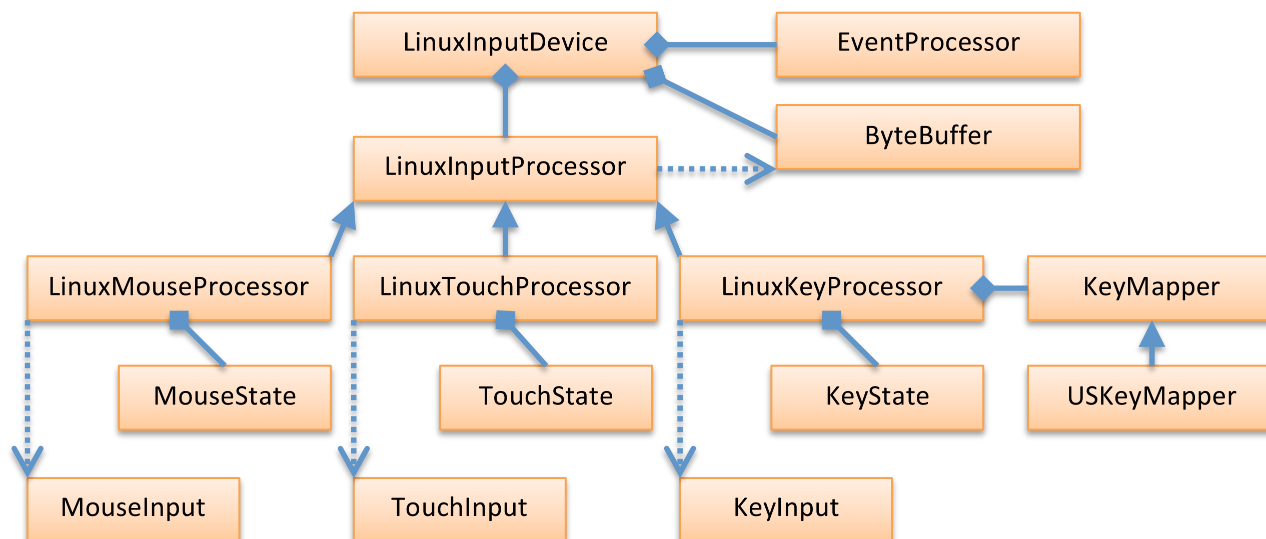
- FBDevScreen reads screen data from the framebuffer device /dev/fb0.
- No cursor implementation is provided for the generic port, since there is no usable standard for hardware cursors on Linux.
- LinuxInputDeviceRegistry receives hot plug notifications of input device addition and removal and sets up input event processors that work with the generic Monocle input handlers.



Input

When an input device is detected by Udev, a LinuxInputDevice instance is created for it. The LinuxInputDevice reads events from the Linux input node into a ByteBuffer. As soon as any data is ready in the ByteBuffer to be processed, a Runnable (actually a singleton EventProcessor) is submitted to the application thread to process this data. "Data ready" means that a complete event has been received, including the terminating EV_SYN SYN_REPORT. If the EventProcessor is already pending execution then it is not resubmitted; the expectation is that every time the EventProcessor runs it will process all pending events.

Each LinuxInputDevice has an implementation of LinuxInputProcessor attached to it. This LinuxInputProcessor is called by the EventProcessor on the application thread. The LinuxInputProcessor iterates over pending input events and notifies input handlers of input state changes.



Note that raw Linux input events are not represented as objects. Instead a single persistent ByteBuffer stores a sequence of event records waiting to be processed. Where possible we avoid object creating during input processing. This will be particularly important in touch and mouse processing where events can be received very quickly.

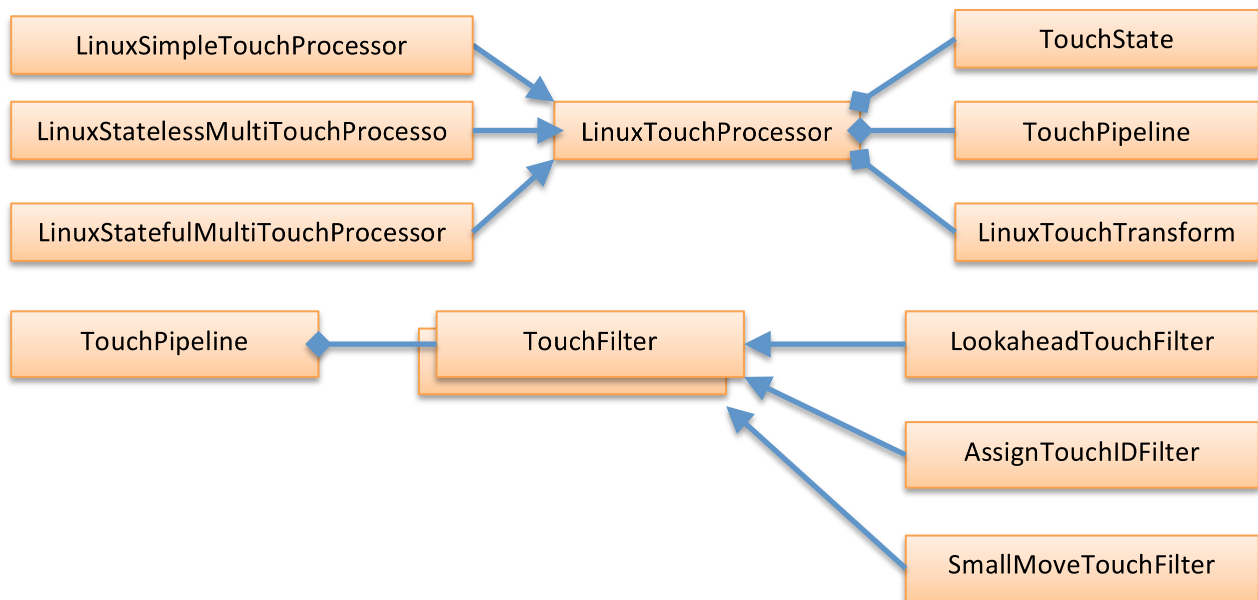
Touch processors do a little more work than other input processors. There are three varieties of Linux touch processors, corresponding to three kinds of input devices:

- LinuxSimpleTouchProcessor – for device drivers that only support a single touch point
- LinuxStatelessMultiTouchProcessor – for device drivers that send all their current state on each event. A stateless driver will send the position of all touch points before every call to EV_SYN, even if some of these points have not changed.
- LinuxStatefulMultiTouchProcessor – for device drivers that send incremental state and can leave out information in events if that information has not changed.

Each touch processor manages a TouchState which it updates according to the events it receives. However the touch processor does not communicate with TouchInput directly; instead it uses a TouchPipeline containing a series of TouchFilters. Available TouchFilters are:

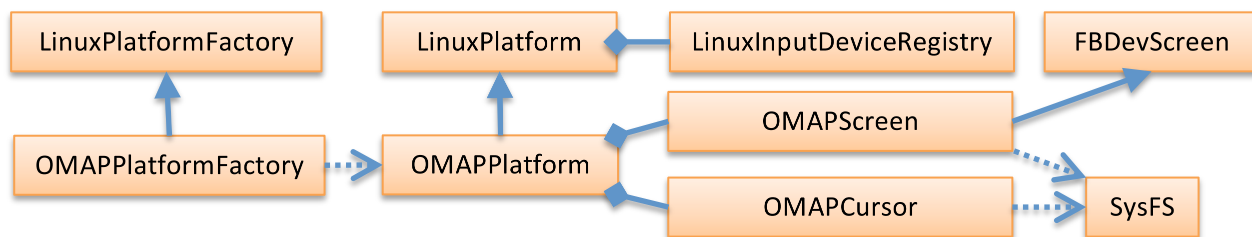
- LookaheadTouchFilter – to compress event sequences by filtering out events in a pulse that differ in their coordinates but not in the number of touch points or their IDs.
- AssignIDTouchFilter – to assign touch IDs to points, for devices that do not assign IDs themselves.
- SmallMoveTouchFilter – to filter out noise in touch events by requiring a minimum move distance between events.

Before applying touch filters, LinuxTouchTransform is used to perform rotate, flip and scale operations on the screen coordinates.



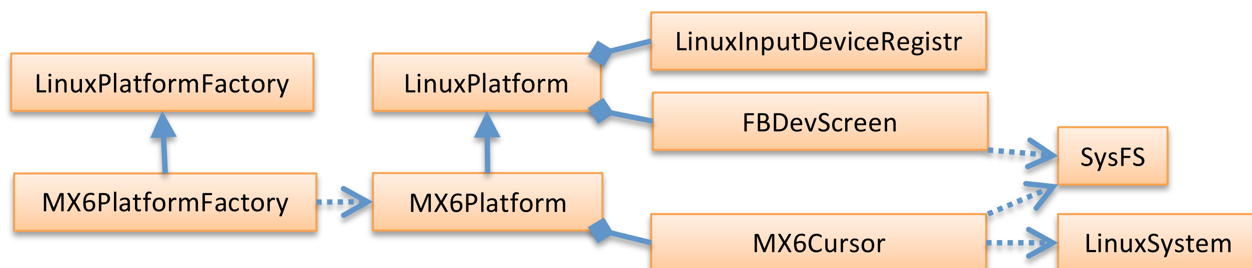
OMAP Port

The OMAP port is a subclass of the generic Linux port that also provides a hardware cursor on OMAP platforms.



MX6 Port

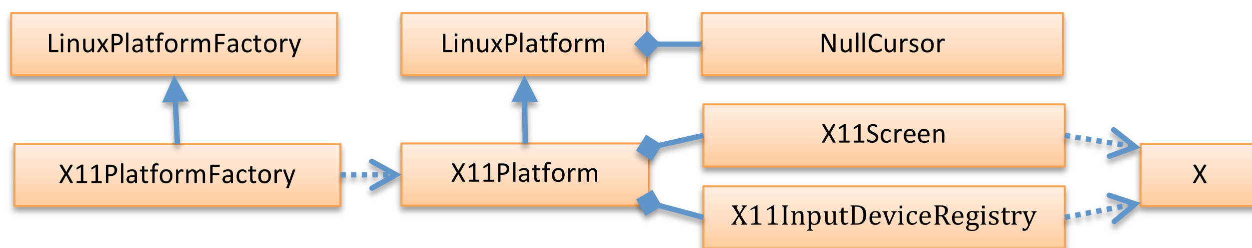
The MX6 port is a subclass of the generic Linux port that also provides a hardware cursor on Freescale i.MX6 platforms. In the future it will also handle the unique EGL platform configuration required for the i.MX6 platform.



Unlike the display overlays used for the hardware cursor on OMAP platforms which can be controlled entirely using sysfs, the i.MX6 cursor needs a number of Linux system calls to operate. The LinuxSystem class provides a thin wrapper to the C APIs required.

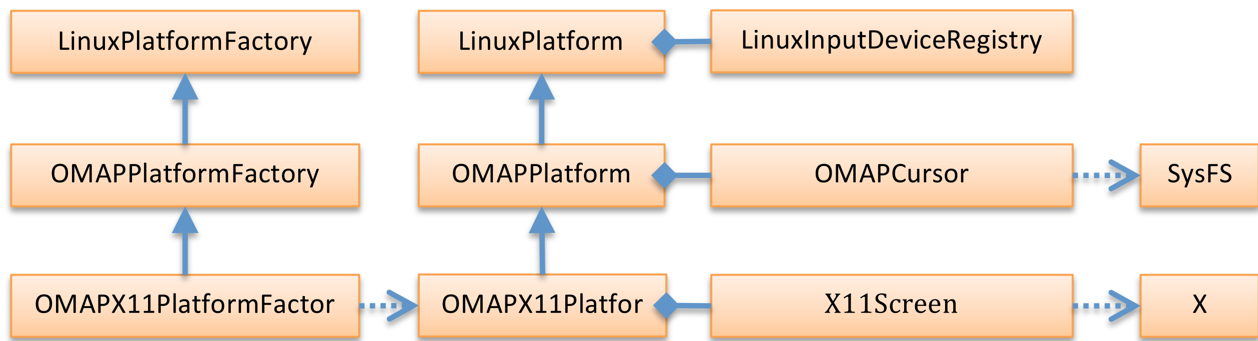
X11 Port

The X11 port is a subclass of the generic Linux port that uses EGL/X11 for rendering instead of EGL/Framebuffer. It also takes its input from X11 events instead of directly from Linux input devices.



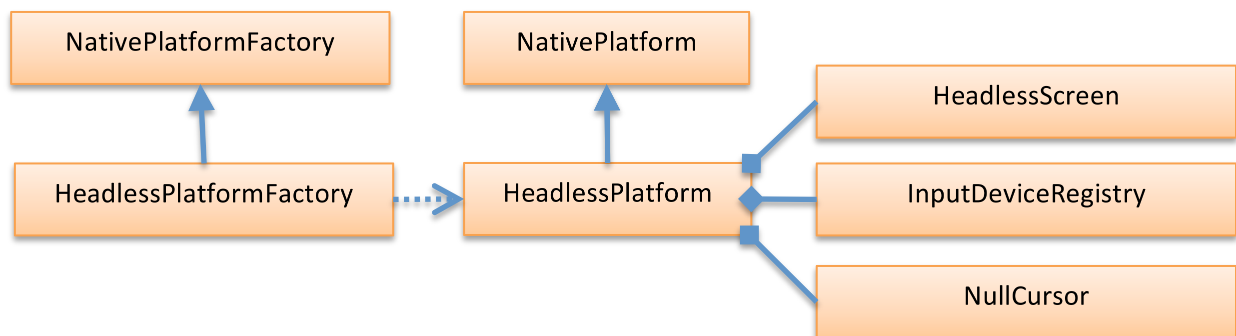
OMAPX11 Port

The OMAPX11 port is a subclass of the OMAP port that uses EGL/X11 for rendering instead of EGL/Framebuffer. Like the OMAP port, it uses Linux input devices for input events and a hardware OMAP cursor.



Headless Port

The headless port does nothing. It is for when you want to run JavaFX with no graphics, input or platform dependencies. Rendering still happens, it just doesn't show up on the screen.



The headless port uses the LinuxInputDeviceRegistry implementation of InputDeviceRegistry. However the headless port does not access any actual Linux devices or any native APIs at all; it uses the Linux input registry in device simulation mode. This allows Linux device input to be simulated even on non-Linux platforms. The tests in tests/system/src/test/java/com/sun/glass/ui/monocle/input make extensive use of this feature.

Native interfaces

There are a few classes in Monocle that use JNI to access native C APIs:

- C.java provides access to C data structures. This can be used to create objects of type C.Structure representing native structures and their pointers. Instantiation of a C.Structure requires the loadLibrary.* permission.
- LinuxSystem.java provides access to Linux APIs. All methods on this class are instance methods; instantiation of a LinuxSystem requires the loadLibrary.* permission. Access to LinuxSystem instances must be tightly controlled.
- X.java has static methods providing access to X11 APIs, for the com.sun.glass.ui.monocle.x11 package only.
- EGL.java provides access to EGL APIs.

Daniel Blaukopf, May 2014