Haywire: Design Contract

Table of contents

Copyright © 2002 Vadim Tkachenko All rights reserved.

1. Requirements

In no specific order:

- The API user must not be concerned with 1-Wire® network path discovery.
- The device state must be completely hidden within a *device container*. If the user wants to retrieve and/or modify the data, so be it. However, thread safety must be observed by the API.
- The device container must have one-to-one correspondence with the actual hardware device (for simple devices), or part of the device (for composite devices), or device set (for multipart devices). Repeated device discovery requests must return the same instance of the device container, with the exception below.
- The device container abstraction must handle the device arrival and departure. As long as an instance of a device container is still referenced, it will be returned as a result of a device discovery request. However, if all the references are released, the device container instance may be dropped (for it makes little sense to keep it).
- The device container instantiation must not happen in any other way than by the adapter, to enforce the condition above (in other words, there must be a `DeviceFactory`).
- The device container must support locking, both explicitly and implicitly. Locking mechanism should be based on a lock object, not per-thread or open/close. It is the responsibility of the API provider to take care of nested locks, should they be necessary.
- The exception hierarchy must properly reflect the abstraction level separation. For example, the error caused by the attempt to call a method on a *device container* whose device has just departed from the network must *not* cause the exception originated in the adapter, but the device container proper.
- Device access must not require anything not related to the device container abstraction, including, but not limited to, speed selection or network path handling.
- The network path doesn't have to be exposed anymore.
- The API must support notions of a `CompositeDevice` (more than one physical 1-Wire® device per one logical device) and `MultipartDevice` (more than one logical device per one physical 1-Wire® device).
- The implementation must be able to handle multiple 1-Wire adapters, and represent a common address space for all attached devices.
- The implementation must also offer a `Bus` abstraction, representing all devices connected to a particular adapter.

Copyright © 2002 Vadim Tkachenko All rights reserved.

2. Unanswered Questions

- The role of DS2409 devices on the network requires further clarification. Sometimes, they act purely as path selectors, sometimes, they can be used as controllers. It is not clear how to separate those roles, and it is not clear whether the couplers should be present in the device enumeration.
- It is not clear how to handle the state of the stateful devices when the disconnects/reconnects happen. One extreme is to require that whenever the device is present on the network, its state must be cached, and whenever the device departs from the network and arrives later, the state must be restored. This would be true, for example, in case of switch containers controlling the hardware. However, it will not be true in case of devices used for authentication.
- It is not clear what to do with the device container class for which is not available. One way (originally used in DalSemi 1-Wire® API) is to provide a generic `OneWireContainer` with complete access to memory banks etc., and the other way is to disallow all the operations altogether, because *it is not clear what kind of device it is*.

3. Original State of Affairs

The 1-Wire® API that existed at the moment of the project inception provided roughly the following architecture (keep in mind that this is a simplified representation distorted by my vivid imagination):

`OneWireAccessProvider`	The basic entity representing the 1-Wire® adapter. Provides complete access to all the network and device control functions.
`DSPortAdapter`	The basic entity representing the 1-Wire® adapter. Provides complete access to all the network and device control functions. On the same level of abstraction, it's quite OK. However, what is *not* OK is that you have to talk to the adapter to do *anything*. The device abstractions clash with the adapter abstraction, and you have to talk to both in order to do just about anything. It might be a good idea to hide the adapter abstraction, and replace it with "1-Wire® network" instead. The adapter abstraction will still provide the functions it does now, and the network abstraction will take care of higher level operations - like arrival/departure notifications, path discovery, device

Copyright © 2002 Vadim Tkachenko All rights reserved.

	enumeration, etc.
`OneWireContainer`	A generic device container. All the concrete container classes extend this class. Again, there's nothing wrong with this *abstraction*. However, there's quite a few of the problems with it in the current *implementation*: • It is difficult to keep track of device containers, given the way the adapter creates them. • The way the device state is handled is inconsistent - sometimes you have to keep the data outside of the container (`readDevice()/writeDevice()`), sometimes it is hidden inside of the container (`TemperatureContainer.setResolution()`). • The arrivals and/or departures are not handled at the device level - instead, the API user gets the generic adapter level exception.

Copyright © 2002 Vadim Tkachenko All rights reserved.