The need for Hardware/Software CoDesign Patterns - a toolbox for the new Digital Designer

By Senior Consultant Carsten Siggaard, Danish Technological Institute
May 30. 2008

Abstract

Between hardware and software the level of abstraction has always differed a lot, this is not a surprise, hardware is an abstraction for software and the topics which is considered important by either a hardware developer or a software developer are quite different, making the communication between software and hardware developers (or even departments) difficult. In this report the software concept of design patterns is expanded into hardware, and the need for a new kind of design pattern, The Digital Design Pattern, is announced.

Contents

Background

In many software intensive systems, the concept of design patterns is well known, and has been in wide-spread use since 1995 (and before), where the famous “Gang of 4” were the first to present the concept in the book “Design Patterns” [5]. The design patterns presented in this book were not scoped towards real-time systems, not to speak of hardware. At the same time Grady Booch published his book “Object Solutions” [1], Grady Booch gave good input to the process, but only for legacy systems. Similarly the ROPES process described in “Real Time Design Patterns” [4] is dedicated to embedded systems, and the book does also address Real-Time requirements so the ROPES process is much more suited to embedded systems. In general none of these processes and profiles is targeted towards hardware, but the design patterns described by these processes can serve as inspiration for the hardware designers, a good example is the book “Real Time Design Patterns” [4].

At the same time a hardware evolution has taken place; processors, DSP’s, FPGA’s and ASIC’s have increased their performance and in size. This evolution in hardware has caused great opportunities for the developers but also challenges; the complexity has increased from one single transistor to millions of transistors. The designers can easily loose the overview of the details in the design. More recent hardware platforms such as FPGA’s and ASIC’s do have functionalities which are either re-programmable or reconfigurable. Both the increased level of integration and the re-programmability make it hard to determine which functionality is located in hardware, and which functionality is located in software.

Putting it all together one way to face the challenge is to define a set of patterns, not only classical real-time patterns but also hardware real-time patterns which are divided into at least two levels of abstraction: Architectural Patterns and Design Patterns.

What is a design pattern

The classical concept of design patterns was invented by the building architect Christopher Alexander, and originates from building construction - but the concept is not limited to building construction only, it is also capable of covering any concept of design and of course also software design.
A design pattern is a generalized recipe of describing solutions to one (or more) problem(s) described in a general manner. The pattern can be augmented using more specific explanatory descriptions.
The classical structure of a design pattern consists of the 4 “essential elements” according to GOF [5]:
The Pattern Name is a shortcut for the following properties best describing the pattern. The name gives us a “handle” so when we refer to the pattern we implicitly give a lot of additional information.

The Problem (or Purpose [4]) is a description of the problem, could be in the form of an example design problem seen previously, or in the form of general terms. Furthermore [4] extends this element with QoS requirements, the pattern tries to optimize.

The Solution A solution to the problem; only so specific that is needed to solve the problem, otherwise the details might get too specific for all implementation means such as different programming languages.

The Consequences Which might be the most important element; because applying a design pattern on a problem might not only give a solution to the problem it might in fact give a wrong solution. This could happen if a design solution does not react well to changes in requirements - for example major changes in performance requirements and/or increased functionality.

According to Bruce Douglass a Real-Time design pattern can improve one or a few of the following qualities of service [4]:

- Performance
  - Worst Case
  - Average Case
- Predictability
- Schedulability
- Throughput
  - Average
  - Sustained
  - Burst
- Reliability
  - With Respect to Errors
  - With Respect to failures
- Safety
- Reusability
- Distributability
- Portability
- Maintainability
- Scalability
- Complexity
- Resource Usage - for example, memory
- Energy consumption
- Recurring cost - for example, hardware
- Development effort and cost

It is interesting to note that the hardware is NOT a point of optimization. But all in all Bruce Douglass addresses many of the problems that we want to solve in hardware as well as in software.

Three Disciplines when using Design Patterns

There are 3 disciplines which outline way you will use the design patterns. This paper is not a description in pattern usage, so this section will only outline the disciplines briefly.

Pattern Hatching

The “Pattern Hatching” discipline is the process of identifying a possible and sufficient pattern to solve the problem. The following description originates from [4]:

1. Familiarize yourself with patterns: Read “The Book”.
2. Apply linear thinking Rank the nature of your problem according to the scope and the QoS.
3. Apply pattern Matching: Try to identify your problem with the problems stated in the description of the patterns in “The Book”.
4. A Miracle Occurs: A potential solution is found.
5. Evaluate the Solution Analyze and then evaluate the solution, if the solution is good go to next step, otherwise go back to either step 3 or step 2.
6. Instantiate the pattern: Instantiate the pattern into your model, see section 2.1.3
7. **The solution:** Test that the pattern meets the functional requirements, and test that the required QoS requirements are met.

Pattern Mining

Pattern mining is the process of identifying the patterns in your designs. To become a pattern the solution must have been used in several projects (at least 3). This discipline will not be elaborated here.

Pattern Instantiation

The implementation of patterns may involve changes of your domain model; however keep in mind that the purpose of many patterns is to decouple the domain model from the collaboration of objects of the pattern, e.g. the visitor and iterator patterns described in GOF. Furthermore some patterns are made to serve as an architectural foundation of your applications, e.g. the observer pattern, which will be described in the following, is intended to be generalized super classes which you can augment with the functionality of your specialized classes.
An example of augmented functionality due to inheritance is if you want to have a base class for graphic objects that you can expand with circles, polygons or other shapes. The base class is then the graphical object which augments with the specialized behavior of the concrete object. Another example is a meter with different readouts such as a clock with a digital and an analog interface.

Levels of Abstraction

Originally Alistair Cockburn introduced the levels of abstraction to determine the level of detail of the use-cases that were written to describe a system. It is our claim that the level of abstraction (or even different levels in the same document) does not only apply to use cases, but they do also determine the quality of the entire system from the writing of the first use-case into the implementation of the different use cases.
Most Engineers are comfortable with a fine grained detailed description, because this gives a significantly well-determined description of a part of, or an entire system.

The ability of removing the details, which either are determined in advance or are left as a degree of freedom for the implementer is a virtue of the architect, but we claim also as a virtue for the new digital designer.

Clamp or low sub-function level small details that make up a sub-function goal
Fish or sub-function level; small tasks that by themselves may not mean much, but stitched together allow one to read a function level goal.
Sea or function level: Tasks which are reasonable to expect to complete in a single sitting.
Kite or summary level: Long term goals that we use various functional level goals to achieve.
Cloud or high summary level: Very high level ongoing goals that may never be completely achieved but that will use summary level goals to drive towards.

As it can be seen from the description above, one can actually determine the author of a design pattern from the level of abstraction.

The Classical Design Pattern – Observer

To demonstrate a design pattern the observer pattern is chosen, because it is a standard design pattern which is both to understand, and very applicable in many designs, e.g. classical legacy systems, real-time systems and small embedded systems such as switches.

Name

Observer

With the following aliases:

Dependents, Model View Controller (MVC), publisher-Subscriber [3], publish, Delegation Event Model in Java [7].

Problem

In a distributed or partitioned system the consistency of data over either the entire system or part of the system is crucial to its operation. The classical description in GOF [5] is a spreadsheet where a change in the contents of a cell in the spreadsheet must be distributed to a set of graphs. Secondly the representation of an object is decoupled from its model [6], this makes it possible for the different representations to react in their own way to changes in the subject (model).

Solution

The classical observer pattern is depicted in figure 4.1. The figure is a static class diagram; hence it does only depict the logical relationships between the classes.
The participants are [5]:

Subject - Should be an abstract class

  - Keeps track of the observers, there can be any number of observers.
  - Provides two functions (*Attach()* and *Detach()*) so any observer can register itself.

Observer - Should be an abstract class
  - Provides the function *Update()* which can be called from the subject to notify all attached observers.

ConcreteSubject - Is a concrete class
  - Stores the state of interest to Concrete Observer Objects.
  - Sends a notification to its observers when its state changes by means of the notify function call.

ConcreteObserver - Is a concrete class
  - Maintains a reference to a ConcreteSubject object, in C++ a subject pointer.
  - Stores a state which should stay consistent with the same state within the subject(s).
  - Implements the Observer update interface (which is a virtual function) on the Observer superclass, so that the Subject can update the Observer.

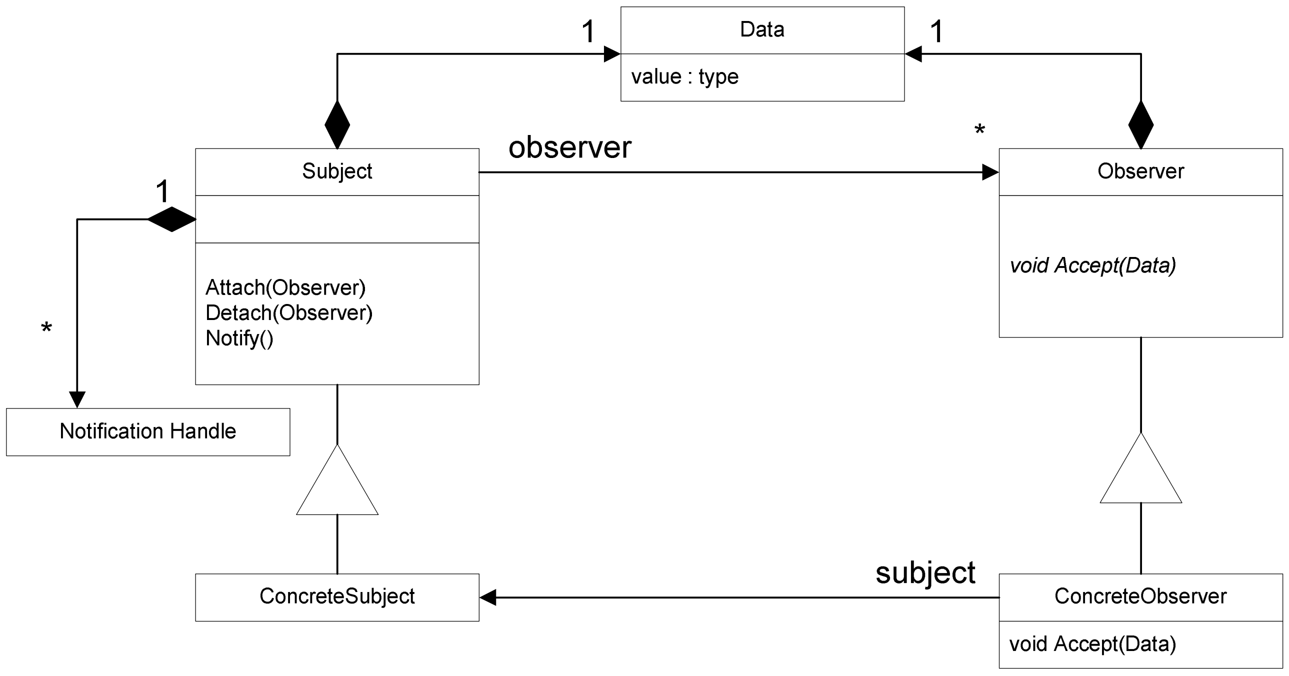

Figure 1: The GOF Observer Pattern.

The operation of the Observer pattern is depicted by means of the sequence diagram below (Figure 2). The attaching and detaching of the concrete observers is not shown.

First **aConcreteObserver** updates the state by calling the *subject->SetState()* method of **aConcreteSubject**, calling this function is not limited to originate from **aConcreteObserver**, it can be done from any object.
aConcreteSubject calls the *Notify()* method of the Subject superclass.
Subject calls the *Update()* function of each Concrete Observer Class.
When **aConcreteOberserver** is notified by means of the update state it calls the *GetState()* method of the **aConcreteSubject** object (if needed).

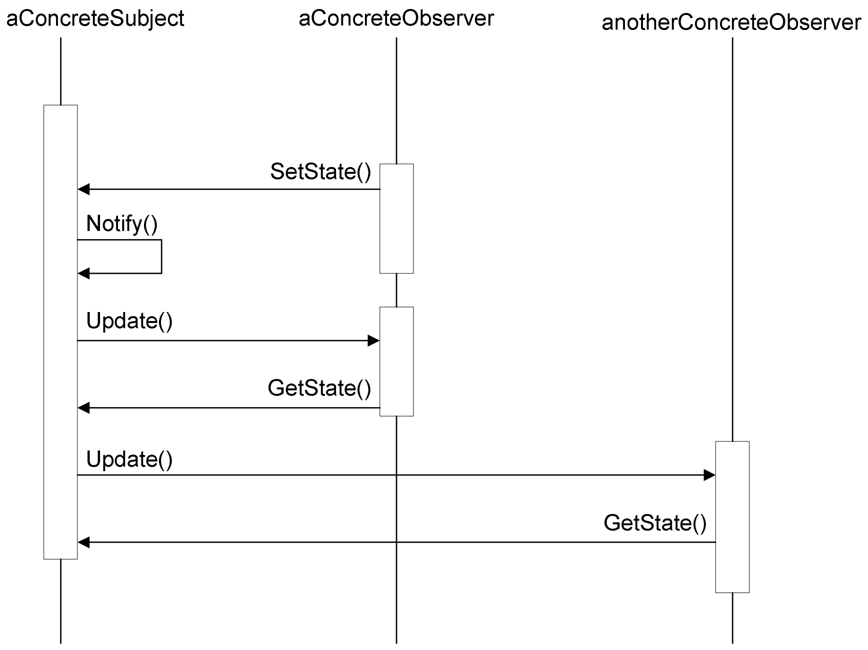

Figure 2: Sequence diagram of the GOF Observer pattern.

Consequences

1. The pattern creates plug-ins for the concrete subject observers of any application where this pattern is usable. As a consequence the same code for the subject and the observer are usable for different applications such as network components, instruments with different view of the same model (meters with both local and remote displays).
2. Fine support for broadcast messages, i.e. a subject does not need to know anything about the concrete observers; it is up to the observers to react on the notification.
3. This pattern will put out a large number of updates, event to the concrete subjects who do not care; hence the load scales with the number of updates and the number of concrete subjects[1].

The RT Design Pattern – Observer

The classical observer pattern does not address any real time aspects; it is covered in the book Real Time Design patterns [4].

Pattern Name

The name of the pattern is Observer and/or model View Controller MVC.

Problem

Again this pattern must solve the problem of distributing knowledge of the state of a data value or an event up to any number of Observers (clients). The distribution is not limited to be done when

1. This does actually depend upon the specific implementation

the value changes, but the distribution can also be done regularly or within some specified time frame. This is different from the pattern described in GOF [5].

Solution

Structure

The structure of the RT observer pattern differs from the original GOF pattern by the fact that it is more elaborated in the way the state (or Data) and the notification handle is described. These are minor improvements which makes the RT observer pattern more general. The naming convention from GOF is kept in the description here, it differs slightly from the one used to describe the RT observer pattern in [4]. One more interesting detail is that the RT observer pattern in [4] looks like it is prepared for the visitor pattern, so that the observers can be updated by the *accept()* method - a bit confusing. Remember that the visitor pattern is created and implements a pull-version. The pattern described here is a push-version.
The *getState()* and *setState()* methods are removed in Figure 3, updating the Subject is done by another method not depicted here.

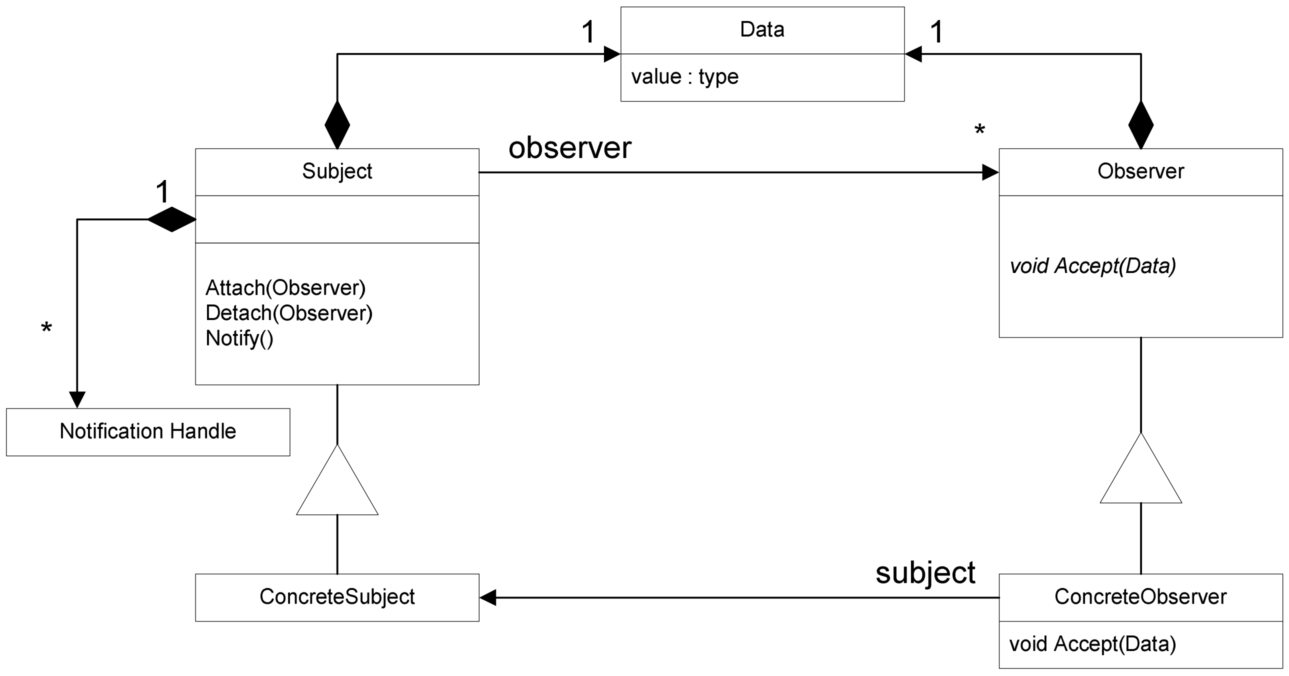

Figure 3: The RT Observer Pattern.

Collaboration

The interoperation of the objects of the pattern is depicted in Figure 4. This sequence diagram depicts the push version of this design pattern. The picture contains the following objects: Two Concrete Observers (**aConcreteObserver** and **anotherConcreteObserver**), aConcreteObject and a **callbackList**.

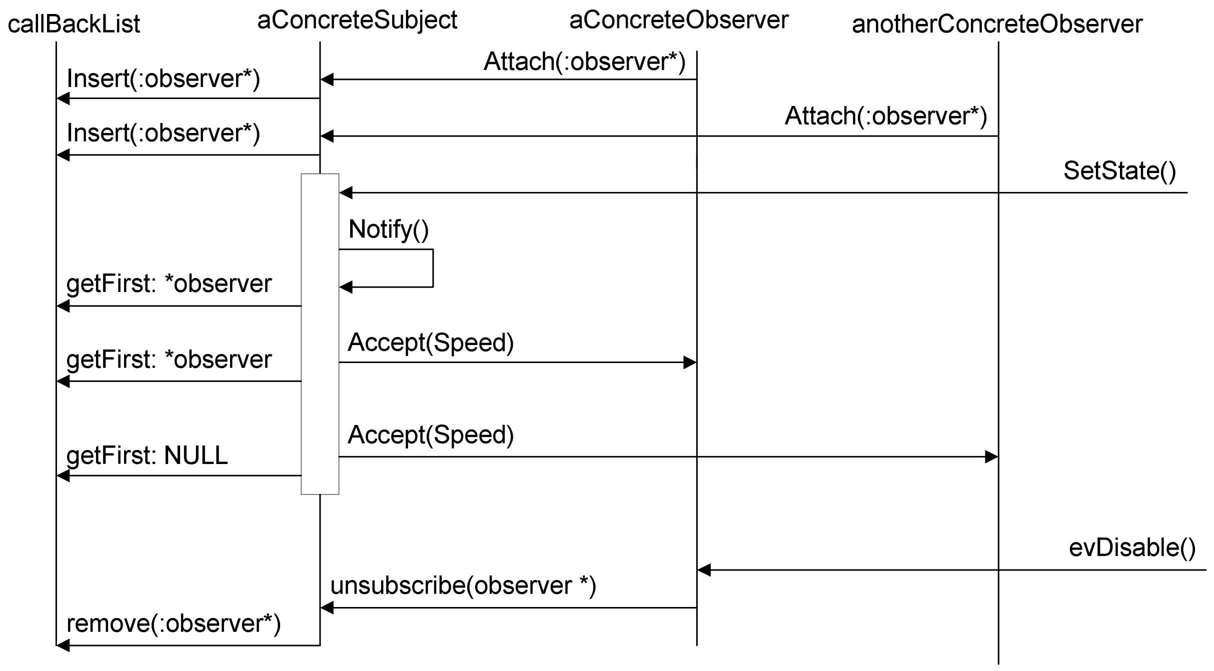

Figure 4: Sequence diagram describing the observer pattern.

The operation is much like the GOF pattern, however instead of update() the accept() method is used, and the callBackList is suspected to use an iterator pattern.

Consequences

The consequences of applying the RT-pattern are in addition to the classical GOF patterns:

The number of subscribers can change dynamically at run-time; this is (of course) also the case for the classic observer pattern.
The process of updating the clients can be centralized to the abstract subject; this simplifies both the implementation of the updating process, and furthermore the clients are simplified because no scheduling policy needs to be implemented.

The RT Design Pattern - Shared Bus - Pull Version

The RT Observer pattern can be extended with a communication channel between the participants which results in the Shared Bus (or Data bus pattern).
It is important to note that this pattern is a software pattern, i.e. this pattern says nothing about the distribution of the objects.

Pattern Name

Data bus,

Also known as
Shared bus (pull version).

Problem

The problem that this pattern solves is much the same as for the observer pattern; if there is a number (which might not be known at design time) of observers whose detailed behavior might also not be known at design time. The behavior of the specific observers might not be known at design time; neither might details of the data distributed within the entire system.

Solution

The solution is to have a well-defined means of communication - a channel - which transports the data without any knowledge of the semantics of the data built into the system. The description in [4] does a lot about the description of the metadata.
Also the pattern is described as a proxy pattern, which is not directly depicted in Figure 5, however note that the Data Bus is a proxy for the Subject, which implementation details are not seen from the client. The metadata is interesting in the case if the bus class does not need to be aware of the concrete type of date it provides, yet still has to have some support for different types – compare this with two hardware solutions where one solution only can provide a bitfield which must be interpreted and a solution where there are fixed fields which cannot be changed.

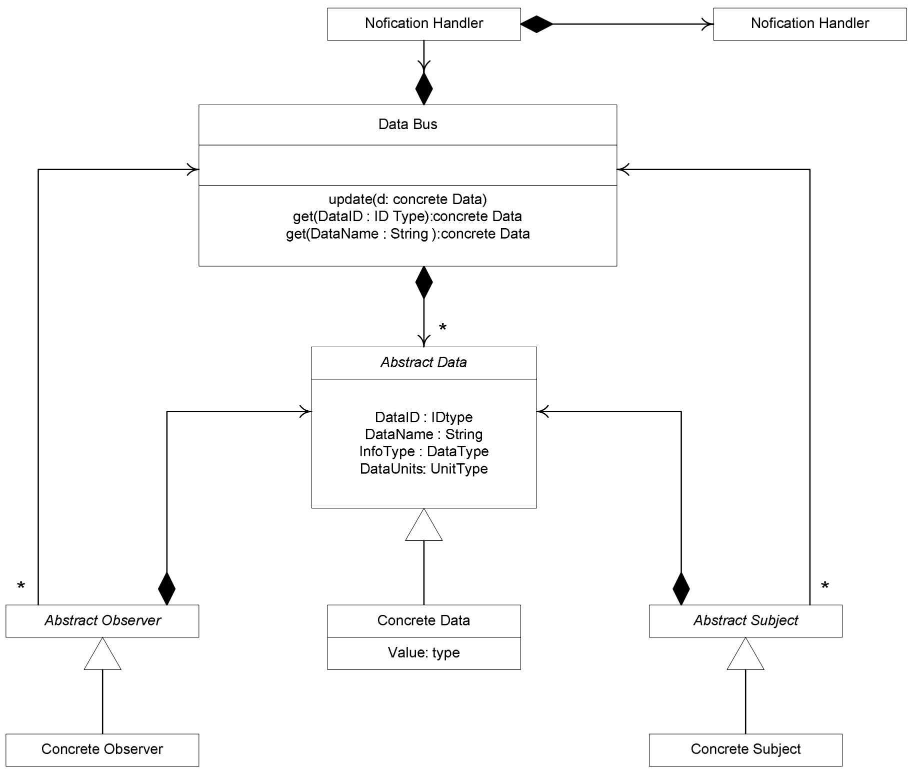

Figure 5: Class diagram depicting the structure of the Data Bus Patten (Pull Version).

Consequences

The (pull) data bus pattern is quite simple to implement, but it suffers from the same problem the Observer pattern have: Updating of data over the system is implemented in the observers, which implies that the amount of traffic on the bus (remember that this is a software pattern) scales not only with the number of observers, but also with the update rate implicitly built into the observers:

$$load = \sum_{i=1}^{n} t_i$$

And if all the update rates are the same then:

$$load = n * \ t_{max}$$

Where t is the update rate, which is not necessarily the update rate of the data in the subject, this rate might be much slower or faster, resulting in redundant traffic on the data bus, or loss of information.

The RT Design Pattern - Shared Bus - Push Version

Name

Data bus.

Also known as:

Shared bus.

This is the push version.

Problem

One wants to implement a system in which a number of clients can subscribe for some information from a subject. The information gets distributed as needed which means either using a time interval or when needed.

Solution

The push version of the data bus design pattern is depicted in
Figure 6. The push version of the data bus version differs from the pull version in the same way as the two versions of the observer pattern. In the push case one observer can subscribe to data on a distinct **dataID** or distinct **dataname** from the bus. It is interesting to note that this version also is capable of handling data using the name of the data - this is not necessary for the pattern to work, but makes to more flexible and independent of the specific kind of data used within the system.

Consequences

This pattern solves many of the same problems as the pull version solves; in addition updating the information in the observers is controlled by the data bus, so the update rate can be controlled from the data bus class.
However, if a subject pushes data out which are quite complex to a subscriber which have no use for it; it can result in a large (unnecessary) load.

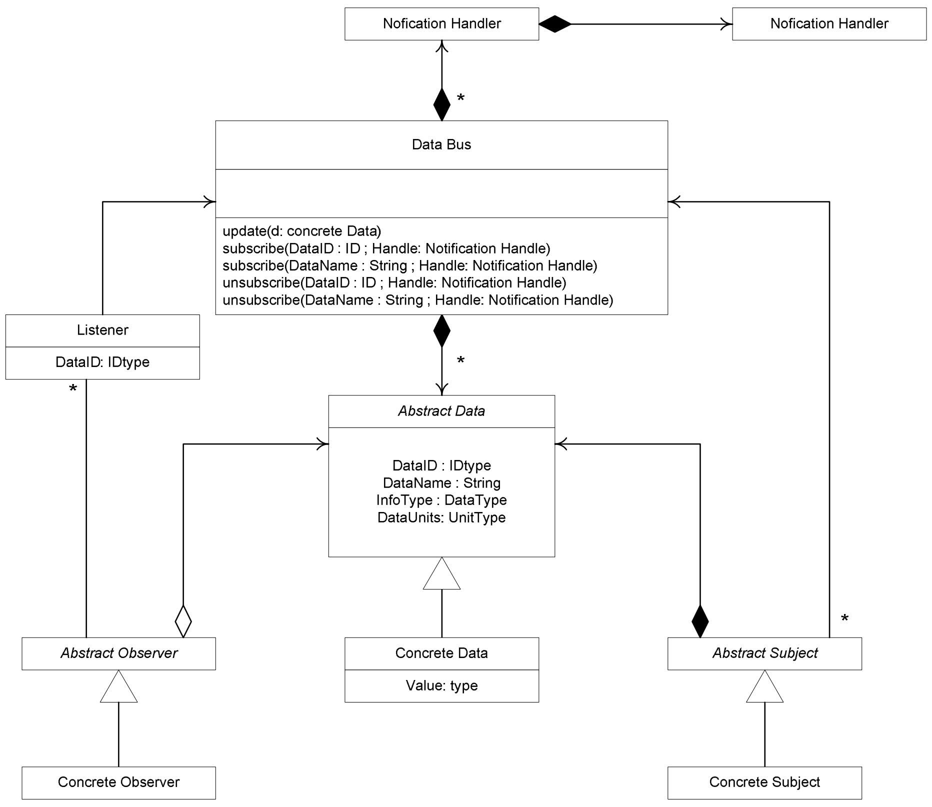

Figure 6: Class diagram depicting the structure of the Data Bus Pattern (Push Version).

A view of a Hardware Observer Pattern – Data Driven

8.1 Name

Hardware Observer or shared bus, is also known as SoC interconnect and Area Network.

8.2 Problem

If a system is required to be highly configurable and extensible both at run-time and as a semi-static system, this pattern can provide a means to solve this problem. The pattern is highly abstract, but a solution using hardware oriented UML notation will be described.

8.3 Solution

As in the RT databus pattern the Hardware Observer Pattern is a derivative of the observer pattern, but there are differences: this case is a data driven version where the observer attaches themselves, not to a specific subject but to the subjects' data. This is called a categorized push/pull model [2]. In this case, if you notify the observers to events created by changes in data, you have created a reactor-like pattern [8].
The reactor pattern is suitable when you use this pattern for a distributed Area Network.
The details of the internals of the Concrete Hardware observer is depicted in figure 8.2. The idea is to avoid putting too much load on the processing resources of the Concrete Hardware Observer, which is accomplished by using the reactor pattern. Note that if the communication bus is a broadcast bus, there must be some kind of filtering of the incoming messages in the Concrete Hardware Observer - and the extensible way to do this is to use registration of handlers.

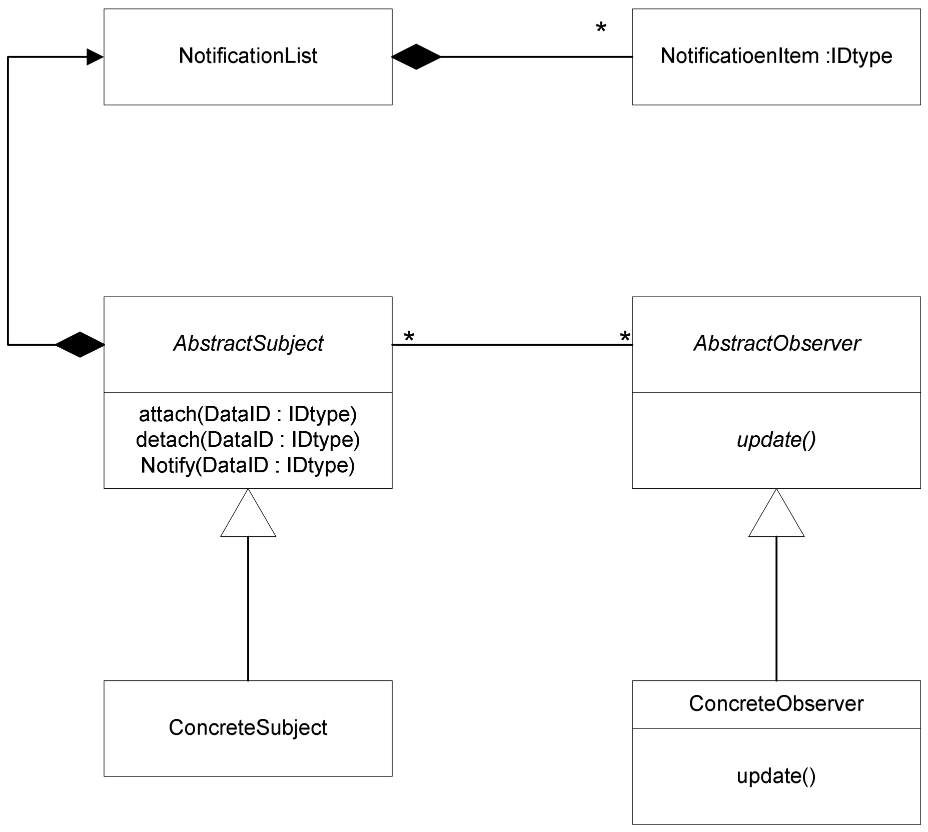

Figure 7: Class diagram depicting the structure of the Hardware Observer Pattern.

For newcomers (to UML), especially hardware guys, deployment diagrams are usually the best way of getting to understand a system - furthermore in this diagram the elaboration into hardware and software can easily be described using traditional UML. A deployment diagram depicting a sample instantiation of the hardware observer pattern is depicted in Figure 8.

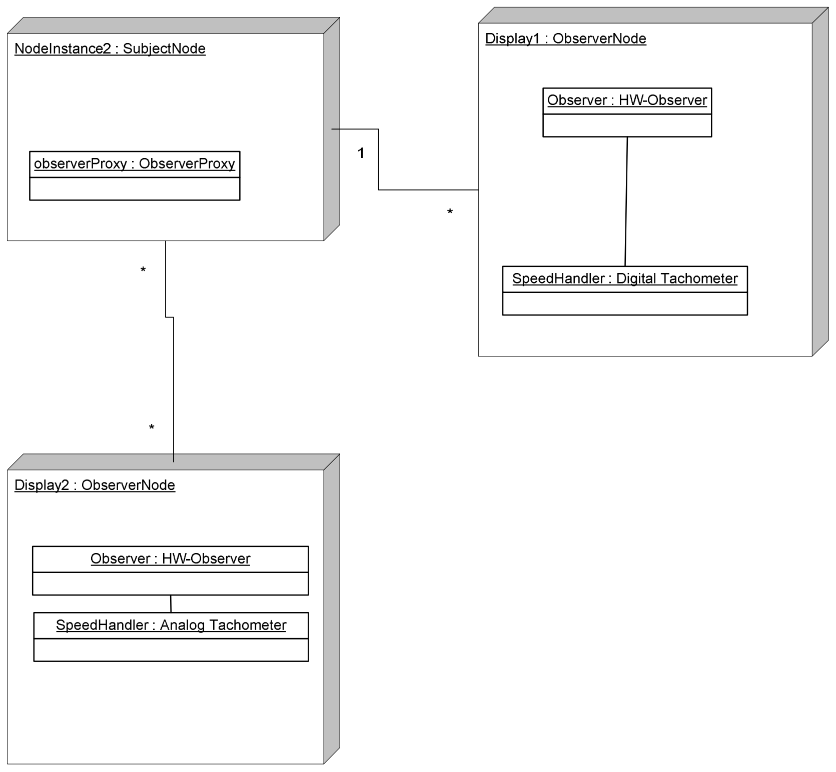
NodeInstance2 : SubjectNode
observerProxy : ObserverProxy
Display1 : ObserverNode
Observer : HW-Observer
SpeedHandler : Digital Tachometer
1
*
*
*
Display2 : ObserverNode
Observer : HW-Observer
SpeedHandler : Analog Tachometer

Figure 8: Class diagram depicting the structure of each Concrete observer.

Consequences

The consequences of applying the HWobserver pattern are the increased flexibility: to a shared bus new observers can be attached at any time thus making the entire system very flexible. The pattern can be implemented using standardized Hardware components, which can be off the shelf microcontrollers with hardware components such as embedded Controller Area Network (CAN) or Ethernet controllers. One disadvantage is that the bandwidth requirements for the bus will increase with the amount of data distributed on the bus, thus the bandwidth of the bus must be either scalable or sufficiently high in order to accommodate the maximum bandwidth which is required. In some cases the bus will never be able to provide the required bandwidth, e.g. in the case of 24-port gigabit switches; the requested bus speed will be at least 24-30GHz, for ONE subject or 576GHzfor 24 (the latter a more realistic requirement - in non-blocking switches).

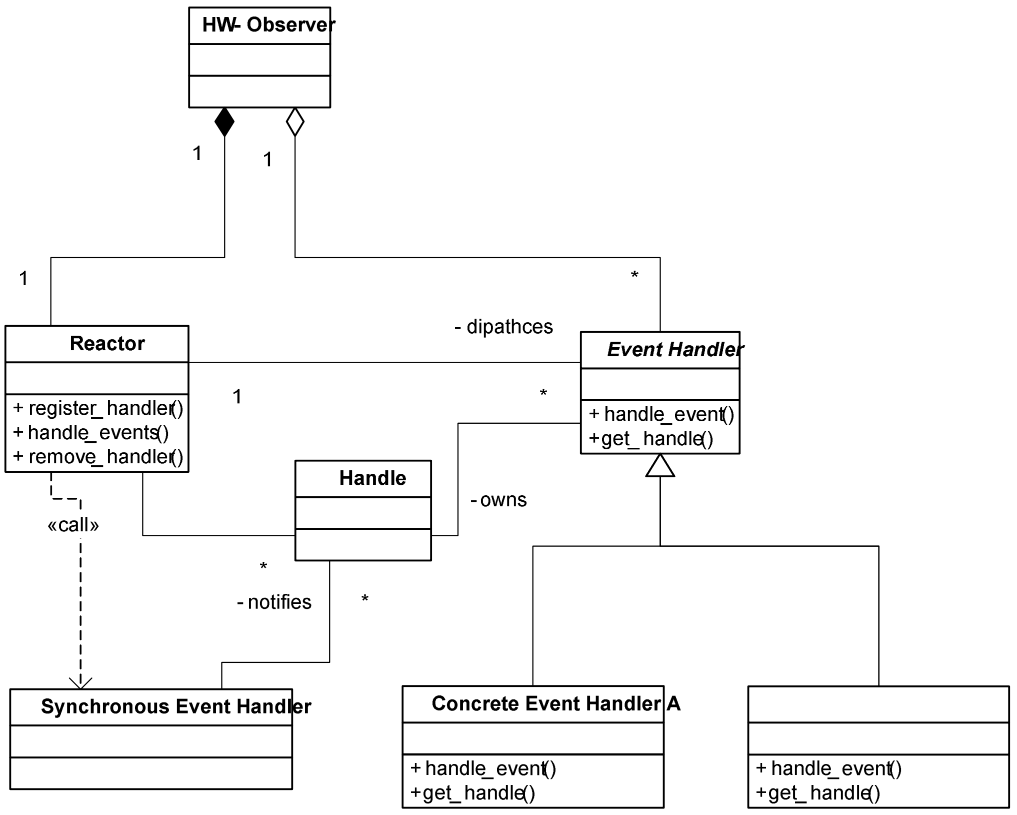

Figure 9: Class diagram depicting the structure of each Concrete observer.

Description Tools/Processes

One of the questions this paper will raise is the need for a design pattern which is usable within a broader community. Software developers are currently the only formalized users of this concept; therefore some of the modeling tools, methods and processes are briefly mentioned.

UML

UML Unified Modeling Language was invented by Grady Booch, Ivar Jacobson and Rumbaugh. The term “unified” is very important: the 3 inventors found out in the mid-1990’s that 3 different OO languages depicting the same (or almost the same) semantics confused the community, so they joined together to create more synergy using the strengths of the OO knowledge of each participant. The rest is history; the “gang of three”-notation is THE notation for OO modeling and Design. UML is governed by OMG the object management group, OMG does also govern the UML profiles and the CORBA specification.

ROPES

Rapid Object-oriented Process for Embedded Systems (ROPES) is an object-oriented development process. This process is (like most processes today), divided into several cycles. ROPES is intended to be scalable from small projects up to very large projects, which is accommodated of several kinds of development cycles: ***nanocycles***, ***Microcycles*** and ***Macrocycles***. The ***nanocycles*** are very

short cycles where a set of classes and relations are put into the model. The result is an executable model which is debugged or tested in the small scale.
A set of nanocycles builds a ***Microcycle*** whose outcome is a ***prototype***. A prototype is a model build of executable and tested code, code which is customer ready.

ROPES uses 5 views of architecture which are:

• Deployment View
• Subsystem and Component View
• Distribution View
• Safety and Reliability View
• Concurrency and Resource View

The two latter views make ROPES differ from other processes such as TUP, where ROPES has real-time capabilities not described by TUP.

TUP

(Danish) Technological Unified Process (TUP), is an iterative process much like ROPES, where ROPES dictates the cases to be implemented are those which constitutes the highest risks. TUP is not specific about the use cases to implement first, only that they must be representative for the full system. TUP is highly based upon the requirements identified during the use-case modeling phase; the use case model is augmented with 4 other views:

Logical View
Process View
Implementation View
Deployment View

Resulting in the 4+1 view known from TUP. TUP is a base of the Model Based

MARTE

Modeling and Analysis of Real-Time and Embedded systems (MARTE), is an UML profile currently being in the finalization process. MARTE is an abbreviation for “Modeling and Analysis of Real-Time and Embedded systems”. MARTE is intended to be a successor for SPT.

SoC

SoC is an abbreviation for System on a Chip, and is an UML profile capable of modeling transfer through channels between modules.

SPT

SPT is an abbreviation for Schedulability Performance and Time. This profile is intended to give quantitative approach when modeling real-time embedded systems. SPT is expected to be superseded by MARTES.

SysML

SysML is an abbreviation for Systems Modeling Language. SysML is a profile to UML augmenting UML with parameters which can be used to interface to other tools such as analysis tools.

Conclusion

There is no doubt that UML and design patterns can be used when designing both hardware and software but there are two topics which should be addressed when using design patterns for hardware: Prerequisites and Consequences. In some cases the prerequisites determines the applicability of the pattern: consider the singleton GOF pattern which easily can be used in hardware, but a special case of it: The state pattern is not usable at all, neither in hardware nor real-time software (due to memory allocation). The conclusion here is that some patterns are applicable in both hardware and software, and some are not; and even for those which are applicable the consequences can differ or have another nature.

Another interesting topic is the level of abstraction required; Hardware developers tend to go down to the "clamp" level where Software developers aims at the highest possible abstraction level - actually there is no surprise here: Software is an abstraction for hardware, and developing software or hardware at the boundary between the two abstraction levels is as being in the twilight zone;
in many cases you want the freedom of being able to abstract from implementation details (that is what is Hardware and what is software), however in some cases you must make the decision due to interface or performance constraints of your design. The digital designer who operates here have two choices: either be able to change the abstraction level according to needs or only operate with hardware components which can be modeled with high abstraction level, but still implicitly meet all constraints necessary to make the design operate.

Another point is that HW designers do not know UML - UML is a software modeling tool, with some hardware capabilities. The need for an educational extension for hardware developers is obvious. On the other hand software developers know nothing about hardware constraints, and the need for either a tool or education is also obvious.

Putting it all together the need for a set of Hardware/Software Co-design patterns is clearly needed. These patterns must have a nature where the level of abstraction is flexible in a way where the users (the "NEW Digital Designers") can hatch and instantiate the pattern and still make sure that the constraints at any time are met
.
But there are obstacles which must be taken into account: If a project is started that shall create a set of design patterns which interoperate[2], then note that up to 80 per cent of all projects of this kind fails (according to Booch [1]). This is a good reason not to do it yourself (especially if someone already have done it). With other words starting such a project is a great RISC.

Another point: Mining patterns do not provide value for a company - contrarily: public patterns are a risk of publishing a company's intellectual property! And a pattern miner might come into the

[2] Also known as frameworks

position that he does not look productive, even if he creates the bullets for the other developers in the company.

Bibliography

[1] Grady Booch.
Object Solutions - Managing the Object Oriented Process. Addison-Wesley, 1996.

[2] Frank Bushmann, Kevling Henney, and Douglass C. Schmidt.
Pattern-Oriented Software Architecture Volume 4 - A Pattern Language for Distributed Computing.
John Wiley and sons, 2007.

[3] Frank Bushmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal.
Pattern-Oriented Software Architecture Volume 1 - A System of Patterns.
John Wiley and sons, 1996.

[4] Bruce Powell Douglass.
Real-Time Design Patterns, Robust Scalable Architecture for Real-Time Systems.
Addison Wesley, 2003.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
Design Patterns - Elements of Reusable Object-Oriented Software.
Addison Wesley,1995.

[6] Flemming Hansen. OO Design Patterns, Course number 11731, October 2007.

[7] Craig Larman.
Applying UML and Patterns - An Introduction to Object-Oriented Analysis and Design and Iterative Development.
Prentice-Hall, 2004.

[8] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Bushmann.
Pattern-Oriented Software Architecture Volume 2 - Patterns for Concurrent and Networked Objects.
John Wiley and sons, 1996.