Abstract Interfaces for Device Independency

Jianjun Hu, Yakub Sailanawala
CSE870 Advanced Software Engineering MiniProject
Computer Science and Engineering
Instructor: Dr. B. Cheng
Michigan State University
Contact Information: chengb@cse.msu.edu

April 12, 2002

Abstract
A key issue in designing software for embedded systems is to make it flexible so that it survives replacement of certain hardware devices with minimum or no changes. This effort provides enormous savings of time and money during the life cycle of the system. The objective of the original paper is to propose a principle and a procedure to achieve these goals. These concepts and principles have been applied or extended in many other branches of software engineering. The same procedure is extended for requirement specification reuse and for defining program families. In terms of abstract interface design, Lampson provided a generalized design principles and hints in computer system design. Two other papers have used the same technique to achieve device independence without citing original paper. One of the papers uses an Abstract-Interface module named AIDO to separate the parallel I/O APIs from the underlying file systems and make the application programs portable and still efficient. The other uses a similar device interface module to integrate different input devices. It is clear that the principle of abstract interface design and the proposed procedure will have further influence in effort to achieve device independency or interface standardization.

Keywords: Abstract Interface Design, Device Independent Module, Commonality Analysis, Assumption List, Product Families, Requirement Reuse, And Abstract Devices, Control Systems.

1 Overview of Designing Abstract Interface for Device Independency [1]
Embedded real-time software, among other things, is used to control several hardware devices [HWD]. These HWDs mostly have complicated and sometimes restrictive interfaces. Changes in hardware device interfaces often lead to widespread changes in the whole system. To avoid this, embedded systems should be designed as two groups of components: 1) The device interface module (DIM) containing the device dependent code, 2) The remaining software components that should be independent of device specific details. This paper discusses how to design good abstract interfaces for DIM to accommodate possible changes of supporting hardware while minimizing, the cost for these changes.

DIMs should meet the following goals:

- DIMs should be small and easy to change. They should be the only components that change if and only if the HWD changes.
- DIMs should hide most of the HWD specific details thus making the user programs easier to write and change.
- Different programs accessing a particular HWD should follow a device-sharing protocol specified in the DIMs.
- Device use can be made more efficient by centralizing device access code in DIMs.

To design a DIM satisfying these goals, we need to define an abstract interface for this DIM. This abstract interface will not only specify the characteristic and the services performed by the DIM and hence the HWD but also provide access functions (APIs) and events which will be used by the user programs to access them. To design an abstract interface we need to obtain its dual description, one that lists all the assumptions about the characteristics of the HWD the user programs are allowed to make, e.g. “The HWD provides information from which wind-speed can be determined”. The other description specifies the APIs and the Events. The
assumption list, in the dual description, explicitly states the assumptions about the characteristics that are implicit in the functional description. The assumption list written in prose is easier to understand by non-programmers. These descriptions must be reviewed by the users, hardware engineers, experienced programmers etc. The design should be modified based on their comments and then reviewed again, repeating the process until a fairly stable and consistent dual description is achieved.

Some of the design goals for DIMs can be conflicting. Some design problems and possible tradeoffs are:

- Designing abstract interfaces becomes difficult if there are many variations in the HWD. Abstract interfaces covering all possible variations can make the DIM large and inefficient.
- For devices that have several sets of characteristics that change independently, it is recommended to bundle these characteristics into different sub-modules. This way the user gets a single view of the HWD while facilitating changes to the DIM.
- Certain characteristics of a HWD may change (e.g. higher resolution) and must be made available to the user programs so that the user can take advantage of them. These characteristics can be specified either at system generation time or at run time. Some guidelines for making a choice are:
  - Characteristics with low cost of variability should be specified at runtime.
  - Characteristics with high cost of change and low probability of change should be specified at system generation time.
  - Characteristics with high cost and high probability of change should have both of these choices available or should be set to some conservative value, which is suitable for a variety of changes, at system generation time.
  - Sometimes interdependence among HWDS can result in interdependence among their respective DIMs. These interdependencies should be specified in the assumptions list and exposed to the user. Also, sometimes due to these dependencies or otherwise, HWDS can require some external information, which can be either supplied by the user program or it can be exchanged between programs of different DIMs. If this information requirement is common to a class of replacement devices then it should be supplied via user programs. This means that if the information requirement changes then the user programs need to change too. On the other hand, if the information requirement is specific to a particular HWD then it can be exchanged among DIM programs.
  - Any future changes or enhancements in the HWD should be mentioned in assumptions even if not implemented.

The idea of designing software adaptable to a range of hardware devices has been generalized to family oriented software development. The concept of hiding device specific details or encapsulation has become a cornerstone of object-oriented programming. Another extension is the effort to achieve platform independency using intermediate abstract interface modules or layers. Examples include Java Virtual Machine and AIDO module in cross-platform parallel file system design.


It is observed that inaccurate or poorly validated requirements are generally the main reason behind delays in software project. Parnas et al. in [1] describe a procedure for designing embedded system software that will survive changes in family of replaceable hardware, by defining abstract interfaces that correspond to independent modules. The authors of [2] present a similar idea for decomposing a control system into modules. The decomposition is based on obtaining a high-level (or abstract) description of requirements that is reusable over a family of control systems that display similar behavior.

The Four Variables model gives an abstract view of the Software Controlled Systems. The four variables are, MON – the physical variables of the system that are being monitored, INPUT – the measured values of the monitored variables that are passed on to the controller, OUTPUT – the output generated by the controller that runs the actuators which will manipulate the physical process, and CON – the physical quantities, of the system, that are controlled. In addition to these four variables we have four mappings, REQ – this represents the requirements of the control systems and it maps MON to CON, IN – the sensors use this relation to transform MON to INPUT, SOFT – Based on the input information the software controller generates output that works the actuators and this function is captured by the SOFT relation that maps INPUT to OUTPUT, and OUT – the actuators use this relation to map OUTPUT to CON. The composition of the IN, SOFT and OUT functions
delivers the requirements (REQ) of the control system. Changes in the sensors and actuators would result in changes in IN and OUT relations respectively and hence the controller. These changes may be independent of the changes in requirements. In order to minimize or even eliminate the impact of changes in IN and OUT relations on SOFT we would want the SOFT to depend heavily on the REQ relation. This can be achieved by decomposing SOFT into \( \text{IN}^{-1}, \text{SOFT}_{\text{REC}}, \) and \( \text{OUT}^{-1} \). The \( \text{IN}^{-1} \) relation reconstructs an estimate of the observed MON values, for a given INPUT. Similarly, \( \text{OUT}^{-1} \) maps the CON variables to OUTPUT needed for the actuators for manipulating them. This separates the sensor and actuator dependent portion of the SOFT relation in modules represented by \( \text{IN}^{-1} \) and \( \text{OUT}^{-1} \) respectively and the \( \text{SOFT}_{\text{REQ}} \) thus left, is clearly an abstraction for the requirements and it will survive changes in sensors and actuators.

The above approach is demonstrated by a case study in the control mechanism of two mobile robots. One of the robots is Pioneer built by ActivMedia, Inc., which comes with a rich set of control library functions and advanced sensors. The other is a lego-bot built with Lego building blocks and uses off-the-shelf sensors and motors for motion. The set of requirements for both these platforms is limited to following: (a) Avoid obstacle if detected, (b) Recover from collisions and continue exploration, and (c) Move forward at full speed in absence of obstacles.

The monitored variables (MON) are \( \text{CollisionDetected} \) – a Boolean variable, \( \text{Range} \) – distance from the obstacle, and \( \text{ObstacleOrientation} \) – this variable indicates if the obstacle is to the left, right or straight ahead. The controlled variables (CON) are \( \text{Speed} \) – ranging from 0 - 100 and \( \text{Heading} \) – ranging from \(-180\) to \(180\) and indicates the degree the robot may have to turn to avoid an obstacle. The goal is to come up with SOFT relation so that the \( \text{SOFT}_{\text{REQ}} \) is as close to the requirement listed above as possible. This goal is achieved by encapsulating the sensors and actuators dependent information in the \( \text{IN}^{-1} \) and \( \text{OUT}^{-1} \) modules. For example, Pioneer uses sonar sensors for distance sensing and the lego-bot uses infrared sensors. Both these sensors have different range and accuracy. But we hide this information with a scaling function that maps both the range to one from 0 – 100. This scaling function forms a part of the \( \text{IN}^{-1} \) module. It is observed that the infrared sensor is not reliable at distances less than 200 feet so the scaling function for the infrared sensor maps all values between 0-200 to 0, thus taking care of a possible inaccuracy. Another aspect is controlling the speed of the robot. As far as the \( \text{SOFT}_{\text{REQ}} \) goes the speed varies from 0 – 100 (range of the \( \text{Speed} \)) and it is the responsibility of the speed--control function in the \( \text{OUT}^{-1} \) module to interpret this value and produce identical effects in both the machines. The speed-control function for Pioneer will just use a scaled value of this number as a parameter for a library function whereas the lego-bot will have a more complex speed-control function which will manipulate some low-level hardware devices.

Structuring the software specifications of a control system by obtaining a high-level specification of the system requirements that is independent of sensors and actuators related information will result in software that is reusable, with minor and easy to make changes, over members of a family that displays that same high level behavior.

3 \textbf{Hints for computer system design [3]}

In [1], Britton \textit{et. al.} proposed a procedure to design the device interface module for anticipated changes of hardware devices. They also provided some practical advices to deal with conflicting decisions or trade-offs during device interface module design. Although their discussion is centered on device interface module design, these techniques are also applicable to interface design of other systems. However, they only provided some general guidelines for interface design of device interface module. In different application context, these guidelines must be careful interpreted.

In [4] Lampson summarized some general hints for computer system design based on the author’s design experience of many computer systems. It discusses hints for three aspects of computer system design: functionality, speed and fault-tolerance. Lampson extended the interface design guidelines of DIM [1] and gave some hints for functionality design that depends largely on interface design. Interface design of computer systems is more general than that of DIM, but they share some basic principles.

Lampson [4] discussed the requirements of interface design in computer system, which includes simplicity, completeness and sufficiently small and fast implementation. This is similar to DIM where lack of essential capabilities and including device-dependent codes in DIM will make it difficult to confine the influence of device changes to the rest of the system. The first hint for interface design of computer systems is trying to keep the interface simple, that is, to capture minimum essentials of an abstraction. Generalization usually means extra cost. Complex interface means its implementation will probably be large, slow and complicated. Simple
interface also makes the cost of each service to be predictable. Usually neglecting the underlying cost of services of interfaces often lead to inefficient user programs. Abstraction through common interfaces is often the source of severe difficulty in upper layer applications. The second important hint for interface design is not to hide power. Higher-level interface should not sacrifice power of lower level to the generality of interface. This concept is similar to the transparency concept in another paper [7] of Parnas. What we want to hide by abstraction is the “undesirable” details. In this aspect, it is clear that the objective of interface design of computer system is not to provide device independence for possible device changes as in DIM in [1]. In DIM design [1], there is an issue related to changing characteristics of virtual devices, in which case system generation parameters or real-time variables can be used to allow reconfiguration. Lampson [4] extended this idea and suggested that procedure arguments of an interface can provide flexibility.

There are some design hints for the continuity (or sustainability) of computer systems. One is to keep basic interfaces stable. This is also the case in the DIM design, where hardware interface changes won’t affect the rest of the system thorough the stable interface of DIM. Another hint is to keep secrets of the implementation. Secrets in DIM design mean the hardware details, which may vary among different candidate hardware devices. In the case of computer system, “secrets are assumptions about an implementation that client programs are not allowed to make”. However there is a trade-off here. Sometimes, exposing some implementation (or hardware) characteristics can improve the efficiency significantly.

In summary, [4] extend the interface design issues to more general interface design problems of computer systems. Some of the hints are equally applicable to DIM design of original paper [1].

4 Commonality Analysis: A Procedure for defining Families [4]

The Host At Sea [HAS] buoys float in sea and collect environment related data and broadcast it at regular intervals. These HAS buoys, irrespective of the differences in their equipments and technique of collecting and transmitting data, have the exact same purpose and so they form a family. If we can obtain an abstract description of the similarities among these buoys and possible variations due to different equipment and changes over time, before their production, then we can achieve immense savings in terms of time and cost by accounting for the same in their design. The process of defining families is referred to as designing abstract interfaces in [1]. The family members are generalizations of the class of replaceable hardware devices mentioned in [1]. The author of [4] gives a structured format for all the elements that go into designing abstract interfaces (or defining families) and detail description of the process for obtaining them. Commonality Analysis [CA] refers to both, the end product in the form of a document as well as the process.

Contents of the CA document, which is also the end product of this analysis, are:

- **Introduction**: Outlines the requirements for a family, ways for generating code and documentation. It also contains the scope and expected use of the family.
- **Overview**: Description of the family and its relation to other families.
- **Terminology**: Standard terms used to describe the family
- **Commonalities**: List of assumptions that are true for all the members of the family. The Assumption List of [1] is what the authors of [3] call commonalities
- **Variability**: In this section we list characteristics of the families that may change over time (e.g. resolution).
- **Parameters of Variation**: CA also includes the range of values over which these characteristics may vary. These values may be numeric or non-numeric; they may be choices of algorithms or functions for computation. CA should also consists specifications of the time these values are bound to the program like, system generation time or run–time or system specification time. The values that are decided at system specification time give rise to sub-families for each fixed value. This issue is discussed in [1], which also gives guidelines for making this choice based on the cost of variation and the likelihood of it.
- **Issues**: List of difficult to resolve, system development issues and possible solutions or alternatives. Maintaining such a list is useful for reviewers, designers and engineers.
- **Appendices**: This includes various issues that transpired as a result of this analysis but may not be directly related to the family.

Commonality Analysis is performed by a team of experts, facilitated by a moderator, during a series of meetings. The moderator guides the discussion and should be well versed in the CA process so that he can identify precise and clear definitions, important issues and other aspects of commonality analysis. Each CA
team also has a recorder who records the team’s decisions, which are displayed for everyone to see during the meeting. Generally the moderator is the recorder. At the end of each meeting every member receives an up-to-date copy of the document. The CA process is divided into the following stages:

- **Prepare**: It is the moderator’s responsibility to see to it that the team has all the resources needed for CA.
- **Plan**: The team decides on the purpose and scope of the family, which will make up the Purpose and Overview sections of the document.
- **Analyze**: During this stage the team discusses all the aspects of the family but for the quantification of the variability parameters. Thus at this stage all but one section of the document is filled.
- **Quantify**: Quantification of the variability parameters is done during this stage. After the completion of this stage the CA document is complete.
- **Review**: CA document is reviewed by users, various experts like system designers, hardware engineers that are external to the CA team.

This is an iterative process. This process can take around 25 weeks, but the time depends on the family.

The CA procedure outlined above has been used on several projects at Lucent and has resulted in 2-3 times in productivity gains. The document resulting from this process serves as an excellent source of reference and training aid for future designers, developers and engineers. It also provides a launch pad for the evolution of the family. The whole process gives the designers an opportunity to think about various issues regarding requirements for the family members that they might otherwise not have thought of.

5 Device Independent Navigation and Interaction in Virtual Environments [5]

This paper [5] proposed an architecture aiming at achieving independence of virtual environment applications from particular available input devices. A special intermediate module named Mapper [Fig 2] is introduced to separate the device dependent system components (input device drivers) with those devices independent ones (Virtual Environment application). Since Mapper module is aimed to integrate several available devices, a logger module is provided to register available input devices. A provider layer is put on the Mapper to coordinate the request and data transfer from Mapper to the application. Compared to the efforts of [1] to extract the commonality of the possible devices and provide standard interface to upper layers, Mapper has some significant extensions for device independence and is different in some important facts to Device Module Interface (DIM) in [1].

One of the major objectives of Mapper module is the ability to integrate different available input devices and emulating missing device capabilities. This is different to the assumption in [1] that usually only one device is assumed to exist at one time. With the different configuration of hardware input devices, Mapper has the capability to reconfigure the use of a set of available input devices interactively based on the user preference list. It also allows sharing of input devices among multiple applications and automatic selection (by an selection algorithm) of devices and interactions appropriate for the task, which is significantly different to DIM in [1]. This concept supports device independence at a higher level.
Another major extension of Mapper compared to DIM in [1] is that Mapper introduced a layer concept [Fig.3]. In [1], DIM interfaces often have to struggle with the trade-off of exposing detail levels. When it is too general, we may lose the power of the underlying devices. When it is too detailed, we lose device exchangeability. In Mapper, different levels of service to application layers are provided and then application programmers can easily control this trade-off by choosing services of appropriate layers provided by Mapper. On the one hand, the device layer allows some special applications to exploit the power of devices. On the other hand, other layers are device independent and provide device independent interface for application development with different level of efficiency. They provide even more abstract service suitable for development of virtual environment applications. For example, interaction layer supports generic interaction data classes for simplified construction of GUI. The intelligent selection of appropriate input devices for a specific task is carried out on this layer. Navigation layer provides services for application to control the user’s avatar in virtual environment. Metaphor layer provides a set of pre-configured choices of navigation styles for some most known applications.

This paper [6] should have cited paper [1] since it uses a Mapper module, which is very similar to DIM module to achieve the device independence and reduce the application change when new input devices are available. It also considered the balance of levels of exposing the details of devices. It also talked about transforming the raw data into standard form or emulating the standard capability of supported interfaces. In addition, the design of the interface of the interaction layer in Mapper must depend on the assumption of the possible data forms required for the different applications. That means that analyzing the commonality function requirements also need to establish an assumption list and an interface list as [1] does. So the procedure proposed in [1] can be applied here and can considerably ease the design process. Instead, this paper doesn’t discuss how they come up with the interfaces at different layers of Mapper.

6 An Abstract-Device Interface for Implementing Portable Parallel –I/O Interfaces [6]

The idea of using intermediate module similar to DIM [1] has been unconsciously applied to implement abstract-device interface for parallel I/O (ADIO) of parallel file systems to achieve portability and efficiency in [6]. ADIO is a layer between parallel I/O APIs and the APIs of different specific file systems. Compared to DIM in [1], ADIO is not designed to be used by application programs directly. Instead, this interface module is only expected to be used to implement a set of device independent parallel I/O APIs which will make the application program independent of the APIs of different underlying file systems. The concept is similar to using Java Virtual Machine to achieve cross-platform capability for application programs.

The objective of ADIO is to facilitate implementation of any existing or new parallel-I/O API on any existing or new file systems. Here the file system is equivalent to the device in [1]. ADIO is composed of a small set of basic operations/functions to performing parallel I/O. Other parallel-I/O interfaces can be implemented on top of ADIO. And for each specific file system, an ADIO must be implemented correspondingly in an optimized way. So implementing an ADIO on any file system make it available to all other parallel-I/O APIs while implementing a new set of parallel I/O APIs makes it work on all underlying file systems.

The ADIO design begins by investigating the interface and functionality provided by different parallel file systems and high level libraries and then deciding how the functionality could be supported by ADIO interfaces in a portable and efficient way. ADIO implements all the essential abstract operations of file systems including: open and close, contiguous reads and writes, noncontiguous reads and writes, non-blocking reads and writes, see, test and wait, file control and others. However, in contrast to [1], this paper doesn’t provide some guidelines and procedure for this abstraction process. It also doesn’t discuss the trade-off of the levels of exposing the details of underlying file systems.

The authors have implemented several parallel-I/O APIs such as MPI-IO, Intel PFS, IBM PIOFS, PASSION in terms of the abstract interfaces provides by ADIO. They also implemented a set of ADIOs on different file systems such as PFS, PIOFS, Unix and NFS. One of the issues is that some file systems don’t provide some abstract operations in ADIO, so some indirect implementations of these abstract interfaces of ADIO must be used. They tested the performance of introducing the additional layer of ADIO and found that the overhead is negligible.
This paper should have cited paper [1] in the sense that its objective and mechanism to achieve device independence by ADIO is exactly the same with DIM in [1]. Here we may look the different underlying file systems as the possible changes of devices. In addition, [1] provides a detailed procedure about how to design the interface of DIM and can be perfectly applied in ADIO design as well. Most of the issues discussed in [1] when designing DIM interfaces can also be used in ADIO interface design, which is a weak point in this paper.

Reference:


