A Procedure for Interface Design

David Lorge Parnas, P.Eng., Ph.D, Dr.h.c., Dr.h.c., FRSC, FACM, FCAE

SFI Fellow, Professor of Software Engineering
Director of the Software Quality Research Laboratory (SQRL)
Department of Computer Science and Information Systems
Faculty of Informatics and Electronics
University of Limerick

Abstract

Improperly designed interfaces can make modular programs almost indistinguishable from “monoliths”. An interface encapsulates design decisions only if it need not be changed when those design decisions are changed. This talk presents a principle for interface design and a procedure for applying that principle. The principle and procedure are illustrated with a variety of examples.
What is an Interface?

The interface between two programs consists of all the assumptions the developers of each program made about the other.

- More than just syntax, format, and type information.
- If one component of a system makes an assumption about another component of the system, and a change in the second component makes that assumption false, the first component will have to be changed.

In other words:

If the correctness of one component of a system can only be demonstrated by making an assumption about another component of the system, and a change in the other component makes that assumption false, the first component will have to be changed.

If we want to avoid the “ripple effect”, we must be aware of the assumptions implicit in an interface and design interfaces that are unlikely to change.
Examples of Implicit Assumptions

Mailing List Software
The main interface is a data structure that will store only
• addresses that contain a five digit postal code
• a street name
• house number before the street name
• etc.

• This software can only be used in the USA.
• This software won’t handle incomplete addresses.
• This software won’t “allow” four digits to be added to the zip code.

This type of distributed assumption can be very expensive. If many parts of the software are based on that assumption, they will have to be changed.
Applying “Information Hiding” to “Embedded Systems”

In so-called embedded systems it is the external interface that is likely to change.

In many of those systems, the software is perceived as a minor component and changes in the external system will be made without taking software costs into account.

In this situation, we should use an “abstract interface” to “hide” the actual interface.

Current fad is to ignore requirements (not determine or document them) that are likely to change.

My approach is the opposite; it identifies the changeable requirements and designs the software to make subsequent changes easier.
Abstract Interfaces

What do we mean by abstract?

1. **Not** vague, theoretical or highly mathematical; abstract means - expressing a general property.

2. Abstract implies a many-to-one mapping.

3. The abstraction represents many things equally well.

4. The abstraction models some aspects of the real things, but not all.

5. Eliminating detail is the approach: The interesting issue is *which* details should be eliminated. Eliminate the details that are likely to change.

6. **Examples of abstractions:**
   
   1. Circuit diagrams - Physical layout is hidden.
   1. Graphs - Meaning of nodes and arcs is hidden.
   1. Algorithms - abstract from restrictions of programming languages.
   1. Abstract Data types - Representation is hidden.
Abstractions

Why are abstractions useful?

• If all properties of the abstract system correspond to properties of the real system, we can learn about the real system by studying the abstraction.

• Abstraction is simpler (in principle);
  • It has less information, but
  • It may appear more complex because it is described using unfamiliar notation.

• Results about abstraction may be “reused”.
  • They apply in many situations - those situations that share the abstraction and differ only in things that are abstracted from.
Abstract Interfaces

What is an abstract interface?

• One interface that represents many possible actual interfaces equally well.
• An interface that models some properties of actual interface but not all.
• A (proper) subset of the set of assumptions in the actual interface.

All of the assumptions must be true. A lie is not an abstraction.

• All things true of the abstract interface must be true of actual interface.
• An infinite stack is not an abstraction; no stack is infinite.
• Every stack has a limit.
• A stack with a parameter representing maximum capacity is an abstraction.
• The parameterised interface applies to many stacks.
• It will not apply to all stacks if you assume that the parameter’s value must be a constant independent of the values stored in the stack.

Abstractions can introduce restrictions but do so consciously.
How can Interface Hiding modules be used in Embedded Systems?

1. Study the real-world interfaces to learn what is possible, likely, unlikely.
2. Define an abstract interface that covers those you are likely to encounter.
3. Procure applications programs based on abstract interface.
4. Procure (separately) programs that translate between real interface and the abstract interface.

**SOFTWARE QUALITY RESEARCH LABORATORY**
If you Design the Abstract Interface Correctly:

“Real-World” changes that affect actual interface affect only the interface programs provided that the assumptions in the abstract interface remain valid.

The main program gets its input using the abstract interface programs provided for that purpose. The main program is written without knowledge of the actual interface.

The main program calls on the interface programs to send output using the abstract interface. It is written without knowledge of the actual interface.

If the interface is well designed, there will be no unnecessary effort in translation from an external format to an internal one.
Simple Example--A Date Interface

Possible formats in actual interfaces:
February 10, 1941 - (month day-in-month, year)
10 Februar 1941- (day-in-month, month, year)
10 February 41 - (day-in-month, month, last-two-digits-of year)
10.2.1941 - (day-in-month.integer-encoded-month.year)
2/10/1941 - (integer-encoded month/day-in-month/year)
41.2.10  - (last-two-digits-of-year, integer-encoded-month, day-in-month)
41 February 10 - (last-two-digits-of-year, month day-in-month)
41,41 - (day-in-year, last-two-digits-of-year)
41,41 - (last-two-digits-of-year, day-in year)
15015 - (days since the first day of 1900)

Abstract Interface

get_year, get_month, get_day, get-days-since-1900
get-day-in-year, get-years-since-1900
get-month-as-integer, ...

Note assumptions about range of years.
Avoiding Double Translations: An Example

Assume:

- The European Standard Date Format is used as the concrete interface.
- An application that computes interest on a daily basis and uses “days-since-1900” internally.
- Some other application program, planning the money supply, also uses the “days-since-1900” internally.
- They communicate using the standard concrete interface.

Under these conditions a double conversion takes place unnecessarily. If both programs use the abstract interface program “get/set days-since-1900” and use that as their internal representation, no conversions take place. The interface must provide all conceivable representations, but only those used need be implemented.
Reflections on Y2K

I started using this example in 1978.

What would have happened if all commercial programs had been written in terms of an abstract interface using “access programs” or “methods” like those on the previous slide?

• We would have had to change the implementation of the date module, but not the programs that use the date module.
• There would have been no space or time penalties.

What else is likely to change?
How to Design an Abstract Interface

1. Prepare a list of assumptions about properties of all the possible real-world interfaces to be encountered.
2. Have it reviewed by people who are in a position to know about what’s out there and what might change. -- Revise it; get new reviewers.
3. Express these assumptions as a set of access programs (operators, methods) allowing possible revisions and outputs. This will be the basis of a module specification.
4. Perform consistency checks - using ignorant reviewers.
   - Verify that any property of the access program set is implied by the assumptions.
   - Verify that all assumptions are reflected in the interface specification.
   - Verify that you can write bulk of system using these access programs; if not, go to 1.

Implementation:
- Developer implements system using the access programs defined in B. The programs must depend only on information in the specification and be correct for any implementation of those access programs that satisfies the specification.
- Implement access programs without knowledge of application.
Illustration of this Procedure for the Address Holder

Initial assumptions:
The following items of information will be contained in addresses and can be identified by analysis of the input data; this information is the only information that will be relevant for our computer programs:

- Last name
- First name
- Organisation
- Street address (house number, street name)
- City, state and zip code
Illustration of this Procedure for the Address Holder

Objections

• There might be a title, e.g. Dr., Professor, Kanzler, Taoniste
• The postal code might not be there.
• There might be no house number.
• There might be no street name.
• There might be a P.O. Box.
• It might be the name of a company,

Refined assumptions - based on objections

Many additional access programs are needed.
Get-title, Get-family name, Get-postal-code, etc.
Provision must be made for addresses with missing elements.
Refining/extending the Interface for a Proper Subset of the Interfaces

1. Some useful applications programs may not be generally applicable.
Specialisation (refinement) by adding access programs not generally implementable
   - get-civil-service-grade
   - get-county
Restrict use of the specialised program to applications that require it.
Sell/deliver as extensions for special needs customers.

2. Refinement by stating additional properties of access programs.
   - postal code is less than 10,000S
   - use postal-code as an array index in restricted applications.
Deviant Actual Interfaces

- Example: Box numbers as complete address
  These represent a failure of the initial design process.

Revise the design to make it an abstraction that is more widely applicable.

The quickest correction is often a mistake. Go back to early false decision and reconsider decisions.

Whenever you introduce an abstraction remember that it represents a set of real things and consider whether or not that set is the right set.

• SOFTWARE QUALITY RESEARCH LABORATORY •
When Won't it Work?

Success depends on our ability to predict change (oracle assumption).

- We cannot predict the future perfectly.
- Our software will be better if we try.
- Eisenhower: Plans are useless, planning is essential.
- Major unexpected deviation can destroy the effort.

Success depends on existence of commonality between actual interfaces

- If you abstract from a very diverse set, you can say very little.
- Interfaces that are good for everything are also good for nothing.
- Abstract from things that don’t matter for your application programs.

Success depends on the Big “Big-Box” Assumption.

- If the application program is small, simple, easy to rewrite, this doesn’t pay.
Review

An interface is equivalent to a set of assumptions.
The abstract interface is a precise, formally specified, interface.
The abstract interface is a model of all “expected” actual interfaces.
Explicit, systematic design is needed.
Review is essential.
Contractor/implementor is more tightly constrained than in conventional procedure -- application program is not allowed to make assumptions that limit applicability.

**Actual interface is met by writing additional programs--not by modifying programs that were written based on the abstract interface definitions.**
Some More Difficult Examples A-7 DIMs

In the remaining slides, we will consider examples from the design of the software for the A-7, a U.S. Navy Attack Aircraft.

In this application there are many specialised devices and they are frequently replaced.

The idea of a Device Interface Module (DIM) becomes important but the interface must be a carefully designed interface.

The idea of a DIM is old. The issue here is interface design.
Goals for Device Interface Modules (DIMs)

1. Confining the impact of device changes on the software.
   - User must not be able to do things that only work on one device.
   - User must have no need to bypass the module.
   - DIM should be minimal.

2. Simplifying the rest of the software

3. Enforced disciplined use of shared resources (protocols)

4. Code sharing

5. Efficient code (product of expert programmers)
Iterate

- Find assumptions that are valid for all devices.
- Embody those assumptions in a set of access programs.
- Review for consistency.
- Review for sufficiency.

Software designers may not be qualified as reviewers.
Use active design reviews - see Parnas/Weiss.
An Example: History of a simple module

A reasonable first draft: Air, Data, Computer (ADC)

Excerpts from an Early Draft of the ADC Abstract Interface

Assumption List

1. The ADC provides a measure of barometric altitude, mach number, and true airspeed.
2. The above measurements are based on a common set of sensors. Therefore an problem in one ADC sensor may invalidate any of these outputs.
3. The ADC provides an indication if any of its sensors are not functioning properly.
4. The measurements are made assuming a sea level pressure of 29.92 inches of mercury.

Access Program Table

<table>
<thead>
<tr>
<th>Program name</th>
<th>Parameter type</th>
<th>Parameter information</th>
</tr>
</thead>
<tbody>
<tr>
<td>G_ADC_ALTITUDE</td>
<td>pl:distance;0</td>
<td>alt. assuming 29.92&quot; sea level pressure</td>
</tr>
<tr>
<td>G_ADC_MACH_INDEX</td>
<td>pl:mach;0</td>
<td>mach</td>
</tr>
<tr>
<td>G_ADC_TRUE_AIRSPEED</td>
<td>pl:speed;0</td>
<td>true airspeed</td>
</tr>
<tr>
<td>G_ADC_FAIL_INDICATOR</td>
<td>pl:logical;0</td>
<td><strong>true</strong> if ADC failed</td>
</tr>
</tbody>
</table>

- SOFTWARE QUALITY RESEARCH LABORATORY -
An Example: History of a simple module

Problems with first draft reported by reviewers:

• Cannot use built-in test. (lacking capability ‘0)
• No report of failed-state values. (missing information)
• Ranges not specified.
An Example: History of a simple module

Assumption List second draft

- The ADC provides measurements of the barometric altitude, true airspeed and the mach number representation of the airspeed of the aircraft. Any known measurement errors are compensated for within the module. Altitude measurements are made assuming that the air pressure at sea level is 29.92" of mercury.

- All of these measurements are based on a common set of sensors; therefore an inaccuracy in one ADC sensor will affect all measurements.

- User programs are notified by means of an event when the ADC hardware fails. If the access programs for barometric altitude, true airspeed and mach number are called during an ADC failure, the last valid measurements (stale values) are provided.

- The ADC is capable of performing a self-test upon command from the software. The result of this test is returned to the software.

- The minimum measurable value for mach number and true airspeed is zero. The minimum barometric altitude measurable is fixed after system generation time, as are the maximum value and resolution for all measurements.
### Access Program Table - second draft

<table>
<thead>
<tr>
<th>Program name</th>
<th>Parameter type</th>
<th>Parameter information</th>
</tr>
</thead>
<tbody>
<tr>
<td>G_ADC_BARO_ALTITUDE</td>
<td>pl:distance;0</td>
<td>corrected altitude assuming slp = 29.92&quot; mercury</td>
</tr>
<tr>
<td>G_ADC_MACH_INDEX</td>
<td>pl:mach;0</td>
<td>corrected mach</td>
</tr>
<tr>
<td>G_ADC_RELIABILITY</td>
<td>pl:logical;0</td>
<td>true if ADC reliable</td>
</tr>
<tr>
<td>G_ADC_TRUE_AIRSPEED</td>
<td>pl:speed;0</td>
<td>corrected true airspeed</td>
</tr>
<tr>
<td>TEST_ADC</td>
<td>pl:logical;0</td>
<td>true if ADC passed self test</td>
</tr>
</tbody>
</table>

### Signalled Event Table

<table>
<thead>
<tr>
<th>Event</th>
<th>When signalled</th>
</tr>
</thead>
<tbody>
<tr>
<td>@T (ADC unreliable)</td>
<td>When “ADC reliable” changes from true to false</td>
</tr>
</tbody>
</table>
University of Limerick

An Example: History of a simple module

Problems with the later draft:

- Sea level pressure correction must be internal.
- Reliability of different outputs could be independent.
- Minimum readings might not be 0.
The Published Version: Assumption List

1. The ADC provides measurements of the barometric altitude, true airspeed, and the mach number representation of the airspeed of the aircraft (mach index). Any known measurement errors are compensated for within the module.

2. User programs are notified by means of events when one or more of the outputs are unavailable. A user program can also inquire about the reliability of individual outputs. If the access functions for barometric altitude, true airspeed, and mach number are called while the values are unreliable, the last valid measurements (stale values) are provided.

3. The ADC is capable of performing a self-test upon command from a user program. The result of this test is returned to the user program.

4. The minimum, maximum and resolution of all ADC measurements are fixed after system generation time.

5. The ADC will compute its outputs on the basis of a value for Sea Level Pressure (SLP) supplied to it by a user program. If no value is provided, SLP of 29.92 will be assumed.
Access Function Table

<table>
<thead>
<tr>
<th>Program name</th>
<th>Parameter type</th>
<th>Parameter information</th>
</tr>
</thead>
<tbody>
<tr>
<td>G_ADC_ALTITUDE</td>
<td>pl:distance;0</td>
<td>corrected altitude assuming SLP = 29.92 or user supplied SLP</td>
</tr>
<tr>
<td></td>
<td>p2:logical;0</td>
<td>true if altitude valid</td>
</tr>
<tr>
<td>G_ADC_MACH_INDEX</td>
<td>p1:mach;0</td>
<td>corrected mach</td>
</tr>
<tr>
<td></td>
<td>p2:logical;0</td>
<td>true if mach valid</td>
</tr>
<tr>
<td>G_ADC_TRUE_AIRSPEED</td>
<td>p1:speed;0</td>
<td>corrected true airspeed</td>
</tr>
<tr>
<td></td>
<td>p2:logical;0</td>
<td>true if true airspeed valid</td>
</tr>
<tr>
<td>S_ADC_SLP</td>
<td>p1:pressure;I</td>
<td>sea level pressure</td>
</tr>
<tr>
<td>TEST_ADC</td>
<td>l:logical;0</td>
<td>true if ADC passed self test</td>
</tr>
</tbody>
</table>

Event Table

<table>
<thead>
<tr>
<th>Event</th>
<th>When signalled</th>
</tr>
</thead>
<tbody>
<tr>
<td>@T(altitude invalid)</td>
<td>When “altitude valid” changes from true to false</td>
</tr>
<tr>
<td>@T(airspeed invalid)</td>
<td>When “true airspeed valid” changes from true to false</td>
</tr>
<tr>
<td>@T(mach invalid)</td>
<td>When “mach valid” changes from true to false</td>
</tr>
</tbody>
</table>
Design problems--Tradeoffs and compromises

Major variations among available devices: Example, Position measurement.
  • Sometimes differences are more than skin deep.
  • Full simulation of ideal device does not separate concerns.
  • Solution: two or more modules.

Devices with several independent characteristics (Example map display)
  • Failure to fully separate.
  • Solution: Module within module.
  • Changeable characteristics that cannot be hidden
  • Introduce System generation parameters
    • Generation time
    • Load time

Device dependent data to/from other modules (example diagnostic data)
  • Restricted interfaces (R rating)
Design problems--Tradeoffs and compromises

Device interdependence
- provide UE messages that hide the interdependence (e.g. 2 messages for one failure)

How to report an event (as a signal or by an access program)
- Specify both events and access programs.
- Implement only what is used.

Devices that need software supplied information.
Is this on the interface or hidden?
- Hidden, use other module’s programs to get information
- On interface, provide program to receive information.

Choose the one that makes the interface less likely to change.
Design problems--Tradeoffs and compromises

Correspondence between real and virtual devices
- virtual device may combine several physical devices.
- virtual device may be only part of a physical device.

The devices that will change together stay together.
If characteristics can change independently, try to separate them.
Conclusions:

The basic definition of abstraction gives good guidelines even in hard design problems.

Good interface design requires careful and explicit study of the possible changes.

We can do a better job with a systematic procedure and a principle.

Subject matter expertise must be obtained. Being a good programmer is not enough to be a good software designer (necessary, not sufficient).