3.5 Design of Abstract Interfaces

Text for Chapter 3.5

[HBPP81] Heninger Britton, K., Parker, R. A., and Parnas, D. L. *A procedure for designing abstract interfaces for device interface modules*. In “Proc. of the 5th Int'l. Conf. on Software Engineering – ICSE 5”, pp. 195–204 (Mar. 1981).

Additional Background for Chapter 3.5

[Par77] Parnas, D. L. *Use of abstract interfaces in the development of software for embedded computer systems*. NRL Report 8047, Naval Research Lab., Washington DC, USA (3 June 1977). Reprinted in Infotech State of the Art Report, Structured System Development, Infotech International, 1979.

A predecessor report of [HBPP81] with more examples.

[PaWe85] Parnas, D. L. and Weiss, D. M. *Active design reviews: Principles and practices*. In “Proc. of the 8th Int'l Conf. on Software Engineering – ICSE 8”, London (Aug. 1985).

How to organize the review of documentation.

Applying Information Hiding to Embedded Systems

- the external interface is what is likely to change
- use an abstract interface to hide the actual interface

Motivation for Abstract Interface Design Rules

- much of the complexity of embedded real-time software: special-purpose hardware devices
  - example A-7 avionics:
    - ▷ 21 devices, arbitrary interfaces (value encodings, timing quirks)
    - ▷ changes during and after development
    - ▷ device “adequate” but does not meet specification exactly
    - ▷ device replaced by better one
    - ▷ new connections between devices
- hide details inside device interface modules
- but *which* details?

Device Interface Modules

- software module structure:
  1. hardware-hiding module
     - 1.1 extended computer module
     - 1.2 device interface module
       - *1.2.1 air data computer*
       - *1.2.2 angle of attack sensor*
       - ...
  2. behaviour-hiding module
  3. software decision module
- provide virtual devices
  - example: virtual altimeter
    - ▷ provides value of type `range` instead of bit string
    - ▷ raw data is read, scaled, corrected, and filtered

Design Goals for Device Interface Modules

- confine changes
- simplify the rest of the software
- enforce disciplined use of resources
- code sharing
- efficient use of devices

Definitions

for:

- interface
- abstraction
- abstract interface
- device interface module
- secret of a device interface module

Definition: Interface

Definition 15 (Interface)
The interface between two programs consists of the set of assumptions *that* each *programmer needs to make about the other program in order to demonstrate the correctness of his own program.*

- more than syntax
- analogous definition for the interface program–device

Definition: Abstraction

Definition 16 (Abstraction)
An abstraction of a set of objects is a description that applies equally well to any one of them.

- each object is an instance of the abstraction
- an abstraction models some aspects, but not all
- example: differential equations
  (electrical circuits, collections of springs and weights, . . .)

Appropriateness of an Abstraction

- appropriate for a given purpose:
  easier to study the abstraction than the actual system
  - example: map

Definition: Abstract Interface

Definition 17 (Abstract interface)
An abstract interface is an abstraction that represents more than one interface.

- exactly the assumptions included in all of the interfaces that it represents

Definition: Device Interface Module

Definition 18 (Device interface module)
A device interface module is a set of programs that translate between the abstract interface and the actual hardware interface.

- implementation possible only if all assumptions in abstract interface are true of actual interface

Definition: Secret of a Device Interface Module

Definition 19 (Secret of a device interface module)
A secret of a device interface module is an assumption about the actual device that user programs is not allowed to make.

- secret is an information about the current device which needs not be true for others

Undesired Event Assumptions

- interface between programs A, B includes assumptions of A about B and of B about A
- B': does not make any assumptions about A
  - extra error checking and reporting in B'; more expensive
- development version of A-7:
  device interface modules that assume
  undesired events by user programs can occur
- production version of A-7: checking omitted
  - compiler switch
- error checks in the requirements: never omitted

Design Approach

- two partially redundant descriptions of the interface:
  1. assumption list characterizing the virtual device
  2. programming constructs embodying the assumptions
- review and iterate

Description 1: Assumption List Characterizing the Virtual Device

- study devices available or under development
  - advertisements of vendors
  - journals
  - . . .
- make list of common characteristics
  - device capabilities
  - modes
  - information requirements
  - behaviour
  - proper use

- these are the assumptions
- example:
  "The device provides information from which barometric altitude can be determined."
  - only devices satisfying this assumption will replace the current barometric altitude sensor
  - no common assumption on the format of the information
- many assuptions appear inocuous
  - record anyway
  - review might prove them false

Description 2: Programming Constructs Embodying the Assumptions

- access programs
  - name, parameter types, value returned
  - limitations
  - effect on the device
- signalling events

The Descriptions are Partially Redundant

- specifications for the programming constructs imply the assumptions
- access program specifications additionally provide form of data exchange
  - example:
    altimeter device interface module
    might not provide barometric altitude directly,
    but two or three quantities from which it can be computed
  - a design change would change the access program specification but not the assumption list

Different Purposes of the Two Descriptions

1. assumption list: state assumptions explicitly
   - explicit: invalid assumptions are easier to detect
   - prose: easier to review for non-programmers
   - review by programmers, users, hardware engineers
     - ▷ valid?
     - ▷ general enough?
2. programming constructs: direct use in user programs
   - review by programmers who have worked with similar programs
     - ▷ typical user programs supported well?
     - ▷ efficient implementation possible?

- consistency is essential
  - assumptions clearly embodied in the programming construct specifications
  - programming construct specifications should not imply additional capabilities

Reviews

- ask the expert *why* something *cannot* change
  - "active design review"
  - for details see [PaWe85]

Iterative Process for the A-7

- tried to list assumptions first
- many subtle assumptions became apparent only when designing programming constructs
- review of assumptions revealed errors in programming constructs
- several cycles of review
  - internally at NRL (several times)
  - by A-7 maintenance team (informal, then formal)

Example: Development of the Air Data Computer (ADC)

- a sensor that measures
  - barometric altitude
  - true airspeed
  - the mach number representation of airspeed

Excerpt of an Early Draft

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 inaccuracy in one ADC sensor may affect 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

program name	parameter type	parameter information
G_ADC_ALTITUDE	p1:distance;O	altitude assuming 29.92 inches sea level pressure
G_ADC_MACH_INDEX	p1:mach;O	mach
G_ADC_TRUE_AIRSPEED	p1:speed;O	true airspeed
G_ADC_FAIL_INDICATOR	p1:logical;O	true if ADC failed

Problems with This Early Draft

- current ADC hardware and most replacement devices have built-in test capability – no access
- when ADC is in failed state, no values specified for access functions
- ranges of measured values not specified
- user programs must poll to detect changes in validity
- not clear whether module performs device-dependent corrections to the raw sensor values

Excerpt of Draft for Formal Review

assumption list

1. 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 inches of mercury.
2. All of these measurements are based on a common set of sensors; therefore an inaccuracy in one ADC sensor will affect all measurements.
3. 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.
4. The ADC is capable of performing a self-test upon command from the software. The result of this test is returned to the software.
5. The minimum measureable value for mach number and true airspeed is zero. The minimum barometric altitude measureable is fixed after system generation time, as are the maximum value and resolution for all measurements.

access program table

program name	parameter type	parameter information
G_ADC_BARO_ALTITUDE	p1:distance;O	corrected altitude assuming sea level pressure = 29.92 inches mercury
G_ADC_MACH_INDEX	p1:mach;O	corrected mach
G_ADC_RELIABILITY	p1:logical;O	true if ADC reliable
G_ADC_TRUE_AIRSPEED	p1:speed;O	corrected true airspeed
TEST_ADC	p1:logical;O	true if ADC passed self test

event table

event	when signalled
@T(ADC unreliable)	When "ADC reliable" changes from true to false

Problems with the Later Draft

- correction for actual sea level pressure is device-dependent
  - therefore better do inside DIM
  - future hardware may do this automatically
- only one reliability indicator for three values
  - current hardware: only one indicator; OK
  - future hardware: might have independent sensors
- some devices might not be able to measure speeds as low as zero

Excerpt of 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. <DELETED>

<DELETED>

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 programs 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, an SLP of 29.92 will be assumed.

access program table

program name	parameter type	parameter information
G_ADC_ALTITUDE	p1:distance;O	corrected altitude assuming SLP=29.92 or user supplied SLP
	p2:logical;O	true if altitude valid
G_ADC_MACH_INDEX	p1:mach;O	corrected mach
	p2:logical;O	true if mach valid
G_ADC_TRUE_AIRSPEED	p1:speed;O	corrected true airspeed
	p2:logical;O	true if true airspeed valid
S_ADC_SLP	p1:pressure;I	sea level pressure
TEST_ADC	p1:logical;O	true if ADC passed self test

event table

event	when signalled
@T(altitude invalid)	When "altitude valid" changes from true to false
@T(airspeed invalid)	When "true airspeed valid" changes from true to false
@T(mach invalid)	When "mach valid" changes from true to false

Design Problems – Tradeoffs and Compromises

- design goals in conflict:
  - small device interface modules
  - device-independent user programs
  - efficiency
- ultimate goal:
  minimize expected cost of the software
  over its entire period of use

Major Variations Among Available Devices

- sometimes differences are more than skin deep
  - example: Inertial Measurement Set (IMS)
- full simulation does not separate concerns
- solution: two modules

Devices with Characteristics that Change Independently

- failure to fully separate
  - example: Projected Map Display Set (PMDS)
- solution: module within module

Virtual Device Characteristics that are Likely to Change

- they cannot be hidden:
  user programs *must* behave differently if these characteristics change
  - examples:
    - ▷ measurement resolutions
    - ▷ number of positions on switches
    - ▷ max. displayable value
- a solution: symbolic constants
  - are system generation parameters

- problem:
  initial assumption wrong that
  all values known at system generation time
- solutions:

cost for variability	likelihood of change	solution
low	*	run-time variable (+ access prgs.)
high	low	system generation parameter
high	high	run-time variable with option to bind earlier
		conservative value for all devices, bind early

Device Dependent Data to/from Other Modules

- device dependent characteristics that vary at run-time
  - example: enter drift rate of IMS at run-time through panel
- reporting and displaying device dependent errors
- solution: restricted interface
  - mark these assumptions and and access programs as "restricted"
  - append to normal interface

Removable Interconnections Between Devices

- device interdependences for hardware convenience
  - example: Doppler and Ship Inertial Navigation Set share a data path
    - ▷ someone assumed the software never needs both simultaneously
  - can hide nature but not existence of connection
- hardware connection might be removed later
- similar: concurrent access to capabilities restricted within a single module
- solution: upward compatible interface
  - show interdependence now
  - maybe remove later

Interconnections Through Possible Failures?

- device A provides information, device B uses it
- device A can fail, invalidating the data of B
- if computer can detect failure of A:
  - device interface module of B can and should hide interconnection by simulating the detection of a failure of B
- if computer cannot detect failure of A:
  - users of B must expect undetectable failures
  - the interconnection itself can and should be hidden

Reporting Changes in Device State

- by signalling events or by access programs?
  - problem: depends on the (changing) requirements of user programs
- solution:
  specify always both,
  implement only what is used

Devices That Need Software Supplied Information

- information from outside device interface module
  - example: current IMS device needs to know whether aircraft is above $70°$ latitude
    - ▷ latitude not calculated within IMS module
- how to get information?
  (a) device interface module provides access program
  (b) device interface module programs call other programs
- solution: depends on whether information requirement is common to the replacement devices
  - if yes: provide access program

Virtual Devices that Do not Correspond to Hardware Devices

- a 1-to-1 relationship not always gives clear interfaces
  - some related capabilities scattered among several hardware devices
    - ▷ example: weapons-related capabilities of A-7
  - some unrelated capabilities occur in the same device for physical convenience
    - ▷ example: weapons release device fills two roles
  - some groupings explained by history only
- solution:
  - one virtual device for weapons release
  - one virtual device for weapon data

Bottom Line

- the basic definition of abstraction gives good guidelines even in hard design problems
- we can do a better job with a systematic procedure and a principle

When Won't It Work?

success depends on:

- the oracle assumption
  - our ability to predict change
- existence of commonality between actual interfaces
  - interface programs smaller than applications programs
- the Big "Big-Box" Assumption
  - the application is big enough to justify the effort for an abstract interface

Abstract Interface Design as an Application of Fundamental Principles

- being explicit about assumptions and design decisions
- encapsulation of likely change

- abstract interface module can solve the embedded computer system problem by hiding the embedding from the computer
- external interface modules are just a special case
  - use same method for other information hiding modules