CSci 658-01: Software Language Engineering Abstraction

H. Conrad Cunningham

24 February 2018

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848

University, MS 38677
(662) 915-5358

Advisory: The HTML version of this document requires use of a browser that supports the display of MathML. A good choice as of February 2018 is a recent version of Firefox from Mozilla.

TODO: - Combine or coordinate with other Data Abstraction and Modular Design material - Expand to discuss the new Backpack module system

Abstraction

Introduction

As computing scientists and computer programmers, we should remember *Simplicity is good; complexity is bad.*

The most effective weapon that we have in the fight against complexity is *abstraction.* What is abstraction?

Abstraction is *concentrating on the essentials and ignoring the details.*

Sometimes abstraction is described as *remembering the "what" and ignoring the "how".*

Kinds of Abstraction

Large complex systems can only be made understandable by decomposing them into modules. When viewed from the outside, from the standpoints of users, each *module* should be simple, with the complexity hidden inside.

We strive for modules that have simple interfaces that can be used without knowing the implementations. Here we use *interface* to mean any information about the module that other modules must assume to be able to do their work correctly.

Two kinds of abstraction are of interest to computing scientists: *procedural abstraction* and *data abstraction.*

Procedural abstraction: the separation of the logical properties of an *action* from the details of how the action is implemented.

Data abstraction: the separation of the logical properties of *data* from the details of how the data are represented.

When we develop an algorithm following the top-down approach, we are practicing procedural abstraction. At a high level, we break the problem up into several tasks. We give each task a name and state its requirements, but we do not worry about how the task is to be accomplished until we expand it at a lower level of our design.

When we code a task in a programming language, we will typically make each task a subprogram (procedure, function, subroutine, method, etc.). Any other program component that calls the subprogram needs to know its interface (name, parameters, return value, assumptions, etc.) but does not need to know the subprogram's internal implementation details. The internal implementation can be changed without affecting the caller.

In data abstraction, the focus is on the problem's data rather than the tasks to be carried out.

Procedures and Functions

Generally we make the following distinctions among subprograms:

- A *procedure* is (in its pure form) a subprogram that takes zero or more arguments but does not return a value. It is executed for its effects, such as changing values in a data structure within the program, modifying its reference or value-result arguments, or causing some effect outside the program (e.g., displaying text on the screen or reading from a file).
- A *function* is (in its pure form) a subprogram that takes zero or more arguments and returns a value but that does not have other effects.
- A *method* is a procedure or function often associated with an object or class in an object-oriented program. Some object-oriented languages use the metaphor of message-passing. A method is the feature of an object that receives a message. In an implementation, a method is typically a procedure or function associated with the (receiver) object; the object may be an *implicit parameter* of the method.

Of course, the features of various programming languages and usual practices for their use may not follow the above pure distinctions. For example, a language may not distinguish between procedures and functions. One term or another may be used for all subprograms. Procedures may return values. Functions may have side effects. Functions may return multiple values. The same subprogram can sometimes be called either as a function or procedure.

Nevertheless, it is good practice to maintain the distinction between functions and procedures for most cases in software design and programming.

In Haskell, the primary unit of procedural abstraction is the pure function. Haskell also groups functions and other declarations into a program unit called a `module`. A `module` explicitly exports selected functions and keep others hidden.

Using Top-Down Stepwise Refinement

This section focuses on procedural abstraction. Later sections focus on data abstraction.

A useful and intuitive design process for a small program is to begin with a high-level solution and incrementally fill in the details. We call this process top-down stepwise refinement. Here we introduce it with an example.

Developing a square root package

Consider the problem of computing the nonnegative square root of a nonnegative number x. Mathematically, we want to find the number y such that

$y \geq 0$ and $y^2 = x$.

A common algorithm in mathematics for computing the above y is to use Newton's method of successive approximations, which has the following steps for square root:

1. Guess at the value of y.
2. If the current approximation (guess) is sufficiently close (i.e. good enough), return it and stop; otherwise, continue.
3. Compute an improved guess by averaging the value of the guess y and x/y, then go back to step 2.

To encode this algorithm in Haskell, we work top down to decompose the problem into smaller parts until each part can be solved easily. We begin this *top-down stepwise refinement* by defining a function with the type signature:

```
sqrtIter :: Double -> Double -> Double
```

We choose type `Double` (double precision floating point) to approximate the real numbers. Thus we can encode step 2 of the above algorithm in Haskell as follows:

```
sqrtIter guess x
    | goodEnough guess x = guess
    | otherwise          = sqrtIter (improve guess x) x
```

We define `sqrtIter` to take two arguments–the current approximation `guess` and number `x` for which we need the square root. We have two cases:

- When the current approximation `guess` is sufficiently close to `x`, we return `guess`.

  We abstract this decision into a separate function `goodEnough` with type signature:

  ```
  goodEnough :: Double -> Double -> Bool
  ```

- When the approximation is not yet close enough, we reduce the problem to another application of `sqrtIter` itself to an improved approximation.

  We abstract the improvement process into a separate function `improve` with type signature:

  ```
  improve :: Double -> Double -> Double
  ```

  To ensure termination of `sqrtIter`, the argument `(improve guess x)` on the recursive call must get closer to a value that satisfies its base case.

The function `improve` takes the current `guess` and `x` and carries out step 3 of the algorithm, thus averaging `guess` and `x/guess`, as follows:

```
improve :: Double -> Double -> Double
improve guess x = average guess (x/guess)
```

Here we abstract `average` into a separate function as follows:

```
average :: Double -> Double -> Double
average x y = (x + y) / 2
```

The new guess is closer to the square root than the previous guess. Thus the algorithm will terminate assuming a good choice for function `goodEnough`, which guards the base case of the `sqrtIter` recursion.

How should we define `goodEnough`? Given that we are working with the limited precision of computer floating point arithmetic, it is not easy to choose an appropriate test for all situations. Here we simplify this and use a tolerance of 0.001.

We thus postulate the following definition for `goodEnough`:

```
goodEnough :: Double -> Double -> Bool
goodEnough guess x = abs (square guess - x) < 0.001
```

In the above, `abs` is the built-in absolute value function defined in the standard Prelude library. We define square as the following simple function (but could replace it by just `guess * guess`):

```
square :: Double -> Double
square x = x * x
```

What is a good initial guess? It is sufficient to just use 1. So we can define an overall square root function `sqrt'` as follows:

```
sqrt' :: Double -> Double
sqrt' x | x >= 0 = sqrtIter 1 x
```

(A square root function `sqrt` is defined in the Prelude library, so a different name is needed to avoid the name clash.)

Making the package a Haskell module

We can make this package into a Haskell module by putting the definitions in a file (e.g., named `Sqrt.hs`) and adding a module header at the beginning as follows:

```haskell
module Sqrt
    (sqrt')
where
    -- give the definitions above for functions sqrt',
    --   sqrtIter, improve, average, and goodEnough,
```

The header gives the module the name `Sqrt` and defines the names in parenthesis as being *exported* to other modules that *import* this module. The other symbols (e.g., `sqrtIter`, `goodEnough`) are local to (i.e., hidden inside) the module.

In the above Haskell code, the symbol "`--`" denotes the beginning of an end-of-line comment. All text after that symbol is text ignored by the Haskell compiler.

The Haskell module for the Square root case study is in file `Sqrt.hs`. Limited testing code is in module `TestSqrt.hs`.

Top-down stepwise refinement

The program design strategy known as *top-down stepwise refinement* is a relatively intuitive design process that has long been applied in the design of structured programs in imperative procedural languages. It is also useful in the functional setting.

In Haskell, we can apply top-down stepwise refinement as follows.

1. Start with a high-level solution to the problem consisting of one or more functions. For each function, identify its type signature and functional requirements (i.e., its inputs, outputs, and termination condition).

   Some parts of each function are abstracted as "pseudocode" expressions or as-yet-undefined function calls.

2. Choose one of the incomplete parts. Consider its type signature and functional requirements. Refine the incomplete part by breaking it into subparts or, if simple, defining it directly in terms of Haskell expressions (including calls to the Prelude or other available library functions).

   When refining an incomplete part, consider the various options according to the relevant design criteria (e.g., time, space, generality, understandability, elegance, etc.)

The refinement of the function may require a refinement of the data being passed. If so, back up in the refinement process and readdress previous design decisions as needed.

If it not possible to design an appropriate refinement, back up in the refinement process and readdress previous design decisions.

3. Continue step 2 until all parts are fully defined in terms of Haskell code and data and the resulting set of functions meets all required criteria.

For as long as possible, we should stay with terminology and notation that is close to the problem being solved. We can do this by choosing appropriate function names and signatures and data types. (In later chapters, we examine Haskell's rich set of builtin and user-defined types.)

For stepwise refinement to work well, we must be willing to back up to earlier design decisions when appropriate. We should keep good documentation of the intermediate design steps.

The stepwise refinement method can work well for small programs, but it may not scale well to large, long-lived, general purpose programs. In particular, stepwise refinement may lead to a module structure in which modules are tightly coupled and not robust with respect to changes in requirements. A combination of techniques may be needed to develop larger software systems.

Using Data Abstraction

A design technique that can help make a program robust with respect to change in the data is to use data abstraction. As in the previous subsection, let's begin with an example.

Rational number arithmetic

For this example, let's implement a group of Haskell functions to perform rational number arithmetic, assuming that the Haskell library does not contain such a data type.

In mathematics we usually write rational numbers in the form $\frac{\texttt{x}}{\texttt{y}}$ where `x` and `y` are integers and `y` $\neq$ `0`.

For now, let's assume we have a special type `Rat` to represent rational numbers and a constructor function

```
makeRat :: Int -> Int -> Rat
```

to create a rational number instance from its numerator `x` and denominator `y`. That is, `makeRat x y` constructs rational number $\frac{\texttt{x}}{\texttt{y}}$.

Further, let us assume we have selector functions `numer` and `denom` with signatures

```
numer, denom :: Rat -> Int
```

that each take a Rat argument and return the numerator and denominator, respectively. That is, they satisfy the equalities:

```
numer (makeRat x y) == x
denom (makeRat x y) == y
```

We consider how to implement rational numbers in Haskell later, but for now let's look at rational arithmetic using the constructor and selector functions above.

Given the knowledge of rational arithmetic from mathematics, we can define the operations for unary negation, addition, subtraction, multiplication, division, and equality.

```
negRat :: Rat -> Rat
negRat x = makeRat (- numer x) (denom x)

addRat, subRat, mulRat, divRat :: Rat -> Rat -> Rat
addRat x y = makeRat (numer x * denom y + numer y * denom x)
                     (denom x * denom y)   -- x + y
subRat x y = makeRat (numer x * denom y - numer y * denom x)
                     (denom x * denom y)   -- x - y
mulRat x y = makeRat (numer x * numer y)
                     (denom x * denom y)   -- x * y
divRat x y = makeRat (numer x * denom y)
                     (denom x * numer y)   -- x / y

eqRat :: Rat -> Rat -> Bool
eqRat x y = (numer x) * (denom y) == (numer y) * (denom x)
```

Above we give the type signatures for all four functions in the same type declaration by listing them separated by commas.

These functions all use the type `Rat`, constructor function `makeRat`, and selector functions `numer` and `denom` assumed above. They do not depend upon any specific representation for rational numbers.

The above six functions work on rational numbers as a *data abstraction* defined by the type `Rat`, constructor function `makeRat`, and selector functions `numer` and `denom`.

The goal of a data abstraction is to separate the logical properties of *data* from the details of how the data are represented.

Rational number data representation

Now, how can we represent rational numbers?

For this package, we define a type synonym `Rat` to denote this type:

```
type Rat = (Int, Int)
```

For example, `(1,7)`, `(-1,-7)`, `(3,21)`, and `(168,1176)` all represent $\frac{1}{7}$.

As with any value that can be expressed in many different ways, it is useful to define a single *canonical* (or *normal*) form for representing values in the rational number type `Rat`.

It is convenient for us to choose a rational number representation `(x,y)` that satisfies the following property, which we call an *invariant*:

> `y > 0`, `x` and `y` are relatively prime, and zero is denoted uniquely by `(0,1)`.

By *relatively prime*, we mean that the two integers have no common divisors except 1.

By *invariant*, we mean that the logical assertion always holds for every rational number created by `makeRat` and manipulated only by the operations in the `RationalCore` and `Rational` modules.

This representation has the advantage that the magnitudes of the numerator `x` and denominator `y` are kept small, thus reducing problems with overflow arising during arithmetic operations.

We thus provide a function for constructing rational numbers in this canonical form. We define constructor `makeRat` as follows.

```
makeRat :: Int -> Int -> Rat
makeRat x 0 = error ( "Cannot construct a rational number "
                      ++ showRat (x,0) ++ "\n" )
makeRat 0 _ = (0,1)
makeRat x y = (x' `div` d, y' `div` d)
    where x' = (signum' y) * x
          y' = abs' y
          d  = gcd' x' y'
```

Above we use features of Haskell we have not used in the previous examples:

- Instead of leaving the `(x,0)` case undefined, we define an explicit `error` call that returns a custom error message as a `String`.
- To concatenate two strings, we use the infix `++` (read "append") operator. (We discuss `++` in the chapter on lists.)
- Putting backticks (`` ` ``) around an alphanumeric function name enables us to use that function as an infix operator. The function `div` denotes integer division. Above the `` `div` `` operator denotes the integer division function used in an infix manner.

- The `where` clause introduces `x'`, `y'`, and `d` as a local definitions within the body of `makeRat`. It can be called from within `makeRat` but not from outside the function. In contrast, `sqrtIter` in the Square Root example is at the same level as `sqrt'`, so it can be called by other functions (in the same Haskell module at least).

  The `where` feature allows us to introduce new definitions in a top-down manner–first using a symbol and then defining it.

- Instead of defining the types of the local definitions `x'`, `y'`, and `d`, we use *type inference*.

  These parameterless functions could be declared

  ```
  x', y', d :: Int
  ```

  but it was not necessary because Haskell can infer the types from the types involved in their defining expressions.

  Type inference can be used more broadly in Haskell, but explicit type declarations should be used for any function called from outside.

The function `signum'` (similar to the more general function `signum` in the Prelude) takes an integer and returns the integer `-1`, `0`, or `1` when the number is negative, zero, or positive, respectively.

```
signum' :: Int -> Int
signum' n | n == 0 =  0
          | n > 0  =  1
          | n < 0  = -1
```

The function `abs'` (similar to the more general function `abs` in the Prelude) takes an integer and returns its absolute value.

```
abs' :: Int -> Int
abs' n | n >= 0 =  n
       | n <  0 = -n
```

The function `gcd'` (similar to the more general function `gcd` in the Prelude) takes two integers and returns their greatest common divisor.

```
gcd' :: Int -> Int -> Int
gcd' x y = gcd'' (abs' x) (abs' y)
    where gcd'' x 0 = x
          gcd'' x y = gcd'' y (x `rem` y)
```

Prelude operation `rem` returns the remainder from dividing its first operand by its second.

Given `makeRat` defined as above, we can define `numer` and `denom` as follows:

```
numer, denom :: Rat -> Int
numer (x,_) = x
```

```haskell
    denom (_,y) = y
```

Finally, to allow rational numbers to be displayed in the normal fractional representation, we include function `showRat` in the package. We use function `show`, found in the Prelude, here to convert an integer to the usual string format and use the list operator `++` to concatenate the two strings into one.

```haskell
    showRat :: Rat -> String
    showRat x = show (numer x) ++ "/" ++ show (denom x)
```

Unlike `Rat`, `makeRat`, `numer`, and `denom`, function `showRat` (as implemented) does not use knowledge of the data representation, but it is used by `makeRat`. We could optimize it slightly by allowing it to access the structure of the tuple directly.

Modularization

There are three groups of functions in this package:

1. the six public rational arithmetic functions `negRat`, `addRat`, `subRat`, `mulRat`, `divRat`, and `eqRat`
2. the public type `Rat`, public constructor function `makeRat`, public selector functions `numer` and `denom`, and string conversion function `showRat`
3. the private utility functions called only by the second group, but just reimplementations of Prelude functions anyway

As we have seen, `Rat`, `makeRat`, `numer`, `denom`, and `showRat` are the *interface* to the *data abstraction* that hides the information about the representation of the data. We can *encapsulate* this group of functions in a Haskell module as follows. This source code must also be in a file named `RationalCore.hs`.

```haskell
    module RationalCore
        (Rat, makeRat, numer, denom, showRat)
    where
        -- Rat, makeRat, numer, denom, showRat definitions
```

We can encapsulate the utility functions in a separate module, which would enable them to be used by several other modules.

However, given that the only use of the utility functions is within the data representation module, we choose not to separate them at this time. We leave them in the data abstraction module. Of course, we could also eliminate them and use the corresponding Prelude functions directly.

Similarly, `negRat`, `addRat`, `subRat`, `mulRat`, `divRat`, and `eqRat` use the core data abstraction and, in turn, extend the interface to include rational number arithmetic operations. We can encapsulate these in another Haskell module that imports the module giving the data representation. This module must be in a file named `Rational.hs`.

```
module Rational
    ( Rat, makeRat, numer, denom, showRat, -- from RatioalCore
    negRat, addRat, subRat, mulRat, divRat, eqRat )
where
    import RationalCore
    -- negRat, addRat, subRat, mulRat, divRat, eqRat definitions
```

Other modules that use the rational number package can import module `Rational`.

This modular approach to program design and implementation offers the potential of scalability and robustness with respect to change.

The key to this *information-hiding* approach to design is to identify the aspects of a software system that are most likely to change from one version to another and make each a design *secret* of some module.

The secret of the `RationalCore` module is the rational number data representation used. The secret of the `Rational` module itself is the methods used for rational number arithmetic.

Alternative rational number data representation

In the rational number data representation above, constructor `makeRat` creates pairs in which the two integers are relatively prime and the sign is on the numerator. Selector functions `numer` and `denom` just return these stored values.

An alternative representation is to reverse this approach, as shown in the following module (in file `RationalDeferGCD.hs`.)

```
module RationalDeferGCD
    (Rat, makeRat, numer, denom, showRat)
where

type Rat = (Int,Int)

makeRat :: Int -> Int -> Rat
makeRat x 0 = error ( "Cannot construct a rational number "
                      ++ showRat (x,0) ++ "\n" )
makeRat 0 y = (0,1)
makeRat x y = (x,y)

numer :: Rat -> Int
numer (x,y) = x' `div` d
    where x' = (signum' y) * x
```

```haskell
                y' = abs' y
                d  = gcd' x' y'

    denom :: Rat -> Int
    denom (x,y) = y' `div` d
        where x' = (signum' y) * x
              y' = abs' y
              d  = gcd' x' y'

    showRat :: Rat -> String
    showRat x = show (numer x) ++ "/" ++ show (denom x)
```

This approach defers the calculation of the greatest common divisor until a selector is called.

The invariant for this rational number representation requires that, for `(x,y)`,

> `y` $\neq$ `0` and zero is represented uniquely by `(0,1)`.

Furthermore, function `numer` and `denom` satisfy the equalities

```haskell
    numer (makeRat x y) == x'
    denom (makeRat x y) == y'
```

where `y' > 0`, `x'` and `y'` are relatively prime, and $\frac{\texttt{x}}{\texttt{y}} = \frac{\texttt{x'}}{\texttt{y'}}$.

Question:

> What are the advantages and disadvantages of the two data representations?

Like module `RationalCore`, the design secret for module `RationalDeferGCD` is the rational number data representation.

Regardless of which approach is used, the definitions of the arithmetic and comparison functions do not change. Thus the `Rational` module can import data representation module `RationalCore` or `RationalDeferGCD`.

Figure 1 shows the dependencies among the modules we have examined in the rational arithmetic case study.

We can consider the `RationalCore` and `RationalDeferGCD` modules as two concrete instances (Haskell `modules`) of a more abstract module we call `RationalRep` in the diagram.

The module `Rational` relies on the abstract module `RationalRep` for an implementation of rational numbers. In the Haskell code above, there are really two versions of the Haskell module `Rational` that differ only in whether they import `RationalCore` or `RationalDeferGCD`.

We could also replace alias `Rat` by a user-defined type to get another alternative definition of `RationalRep`, as long as the interface functions do not have to work with types other than `Int`.

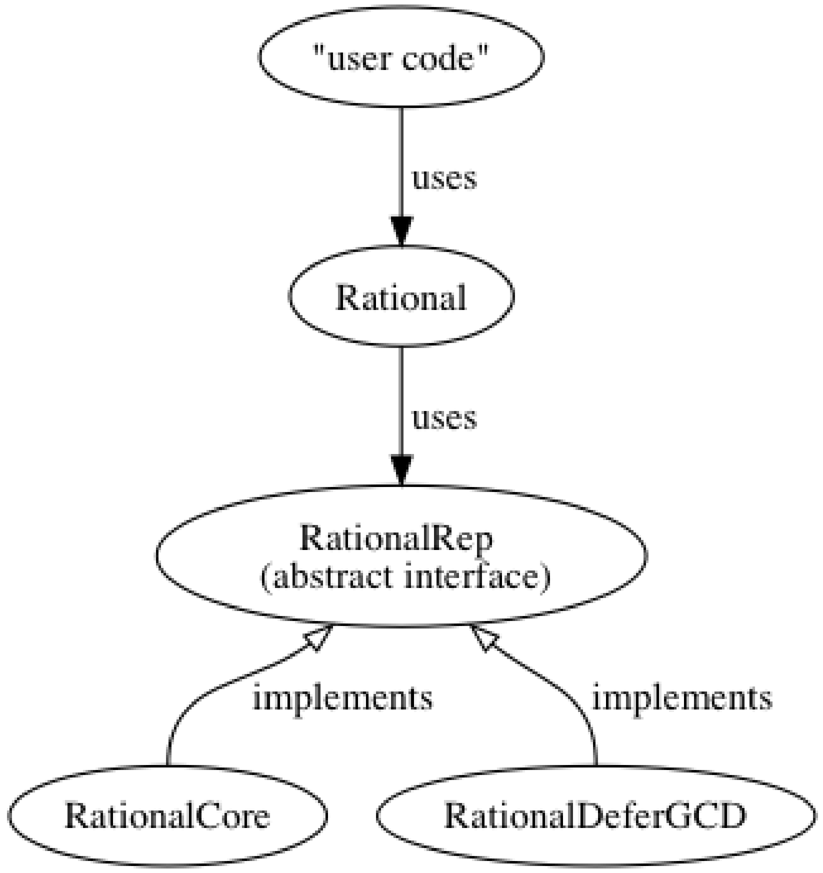

Figure 1. Module Dependencies for Rational Arithmetic Case Study

Modular Design and Programming

Now let's step back from the rational arithmetic case study and consider the general issues of data abstraction and modular design and programming.

Software engineering pioneer David Parnas defines a *module* as "a work assignment given to a programmer or group of programmers" [Parnas 1978]. This is a *software engineering* view of a module.

In a programming language like Haskell, a `module` is also a program unit defined with a construct or convention. This is a *programming language* view of a module.

Ideally, a language's module features should support the software engineering module methods.

Information-hiding modules

According to Parnas, the goals of *modular design* are to [Parnas 1972]:

1. enable programmers to understand the system by focusing on one module at a time (i.e., *comprehensibility*).
2. shorten development time by minimizing required communication among groups (i.e., *independent development*).
3. make the software system flexible by limiting the 'number of modules affected by significant changes (i.e., *changeability*).

Parnas advocates the use of a principle called *information hiding* to guide decomposition of a system into appropriate modules (i.e. work assignments). He points out that the connections among the modules should have as few information requirements as possible [Parnas 1972].

In the Parnas approach, an information-hiding module:

- forms a *cohesive* unit of functionality *separate* from other modules
- *hides* a design decision (its *secret*) from other modules
- *encapsulates* an aspect of system likely to change (its secret)

Aspects likely to change independently of each other become secrets of separate modules. Aspects unlikely to change can become interactions (connections) among modules.

This approach supports the goal of changeability (goal 2). When care is taken to design the modules as clean abstractions with well-defined and documented interfaces, the approach also supports the goals of independent development (goal 1) and comprehensibility (goal 3).

Information hiding has been absorbed into the dogma of contemporary object-oriented programming. However, information hiding is often oversimplified as merely hiding the data and their representations [Weiss 2001].

The secret of a well-designed module may be much more than that. It may include such knowledge as a specific functional requirement stated in the requirements document, the processing algorithm used, the nature of external devices accessed, or even the presence or absence of other modules or programs in the system [Parnas 1972, 1979, 1985]. These are important aspects that may change as the system evolves.

Interfaces

It is important for information-hiding modules to have well-defined and stable interfaces.

According to Britton et al, an *interface* is a "set of assumptions ... each programmer needs to make about the other program ... to demonstrate the correctness of his own program" [Britton 1981].

An interface includes the type signature of each function (i.e., name, arguments, and return value) and the constraints on the environment and argument values (e.g., the invariants).

An *abstract interface* is an interface that does not change when one module implementation is substituted for another [Britton 1981; Parnas 1978]. It concentrates on module's essential aspects and obscures incidental aspects that vary among implementations.

Information-hiding modules and abstract interfaces enable us to design and build software systems with multiple versions. The information-hiding approach seeks to identify aspects of a software design that might change from one version to another and to hide them within independent modules behind well-defined abstract interfaces.

We can reuse the software design across several similar systems. We can reuse an existing module implementation when appropriate. When we need a new implementation, we can create one by following the specification of the module's abstract interface.

Haskell information-hiding modules

As we have seen, in Haskell the `module` construct can be used to encapsulate an information-hiding module. The features exported form part of the interface to the module. One module can `import` another module, specifying its dependence on the interface of the other module.

We define each Haskell module in a separate file. The Haskell compiler can compile a module independently of others except that the modules it depends on must be previously compiled. The Haskell build and package management tools `cabal-install` and `stack` support Haskell modules as their primary units.

The interface of a Haskell module consists of the names and type signatures of its exported types and functions plus the constraints on the functions and the expected properties of the "objects" manipulated.

In the Rational Arithmetic case study, we defined two information-hiding modules:

1. "RationalRep", whose secret is how to represent the rational number data and whose interface consists of the data type `Rat`, operations (functions) `makeRat`, `numer`, `denom`, and `showRat`, and the constraints on these types and functions

2. "Rational", whose secret is how to implement the rational number arithmetic and whose interface consists of operations (functions) `negRat`, `addRat`, `subRat`, `mulRat`, `divRat`, and `eqRat`, the other module's interface, and the constraints on these types and functions

We developed two distinct Haskell modules, `RationalCore` and `RationalDeferGCD`, to implement the "RationalRep" information-hiding module. We developed one distinct Haskell module, `Rational`, to implement the "Rational" information-hiding module. Haskell module `Rational` can be paired (i.e., by chaning the `import` statement) either of the other two variants of "RationalRep".

Unfortunately, Haskell 2010 has a relatively weak module system that does not support multiple implementations as well as we might like. There is no way to declare that multiple Haskell modules have the same interface other than copying the common code into each module and documenting the interface carefully. We must also have multiple versions of `Rational` that differ only in which other module is imported.

Together the Glasgow Haskell Compiler (GHC) release 8.2 (July 2017) and the Cabal-Install package manager release 2.0 (August 2017) support a new extension, the Backpack mixin package system. This new system remedies the above shortcoming. In this new approach, we would define the abstract module "RationalRep" as a signature file and require that `RationalCore` and `RationalDeferGCD` conform to it.

Further discussion of this new module system is beyond the scope of this chapter.

Invariants

As we saw in the Rational Arithmetic case study, a module that provides a data abstraction must ensure that the objects it creates and manipulates maintain their integrity–always have a valid structure and state. An invariant for the data abstraction can help us design and implement such objects.

Invariant: A logical assertion that must always true for every "object" created by the public constructors and manipulated only by the public operations of the data abstraction.

Often, we separate an invariant into two parts.

Interface invariant: An invariant stated in terms of the public features and abstract properties of the "object".

Implementation (representation) invariant: A detailed invariant giving the required relationships among the internal features of the implementation of an "object"

An interface invariant is a key aspect of the abstract interface of a module. It is useful to the users of the module, as well to the developers.

In the Rational Arithmetic case study, the interface invariant for the "Rational-Rep" abstract module is the following.

For all integers `x` and nonzero integers `y`,

```
~~~{.haskell}
    numer (makeRat x y) == x'
    denom (makeRat x y) == y'
~~~
```

where `y' > 0`, `x'` and `y'` are relatively prime, $\frac{x}{y} = \frac{x'}{y'}$ and if `x'` $= 0$, then `y'` $= 1$.

An implementation invariant guides the developers in the design and implementation of the internal details of a module. It relates the internal details to the interface invariant.

We can state an implementation invariant for the `RationalCore` module as follows.

For all integers `x` and nonzero integers `y`,

```
~~~{.haskell}
    makeRat x y == (x',y')
~~~
```

where `y' > 0`, `x'` and `y'` are relatively prime, $\frac{x}{y} = \frac{x'}{y'}$ and if `x'` $= 0$, then `y'` $= 1$.

The implementation invariant implies the interface invariant. Although `makeRat` does quite a bit of work, `numer` and `denom` are simple.

We can state an implementation invariant for the `RationalDeferGCD` module as follows.

For all integers `x` and nonzero integers `y`,

```
~~~{.haskell}
    makeRat x y == (x,y)
~~~
```

In this module implementation, `makeRat` is trivial, thus `numer` and `denom` must do most of the work to establish the interface invariant.

We will return to the invariant concepts in later chapters.

Design criteria for interfaces

What makes a good interface for an information-hiding module?

In designing an interface for a module, we should also consider the following criteria. Of course, some of these criteria conflict with one another; a designer must carefully balance the criteria to achieve a good interface design.

Note: These are general principles; they are not limited to Haskell or functional programming. In object-oriented languages, these criteria apply to class interfaces.

- **Cohesion:** All operations must logically fit together to support a single, coherent purpose. The module should describe a single abstraction.
- **Simplicity:** Avoid needless features. The smaller the interface the easier it is to use the module.
- **No redundancy:** Avoid offering the same service in more than one way; eliminate redundant features.
- **Atomicity:** Do not combine several operations if they are needed individually. Keep independent features separate. All operations should be *primitive*, that is, not be decomposable into other operations also in the public interface.
- **Completeness:** All primitive operations that make sense for the abstraction should be supported by the module.
- **Consistency:** Provide a set of operations that are internally consistent in

  - naming convention (e.g., in use of prefixes like "set" or "get", in capitalization, in use of verbs/nouns/adjectives),
  - use of arguments and return values (e.g., order and type of arguments),
  - behavior (i.e., make operations work similarly).

  Avoid surprises and misunderstandings. Consistent interfaces make it easier to understand the rest of a system if part of it is already known.

  The operations should be consistent with good practices for the specific language being used.

- **Reusability:** Do not customize modules to specific clients, but make them general enough to be reusable in other contexts.

- **Robustness with respect to modifications:** Design the interface of an module so that it remains stable even if the implementation of the module changes. (That is, it should be an abstract interface for an information-hiding module as we discussed above.)

- **Convenience:** Where appropriate, provide additional operations (e.g., beyond the complete primitive set) for the convenience of users of the module. Add convenience operations only for frequently used combinations after careful study.

We must trade off conflicts among the criteria. For example, we must balance:

- completeness versus simplicity
- reusability versus simplicity
- convenience versus consistency, simplicity, no redundancy, and atomicity

We must also balance these design criteria against efficiency and functionality.

Exercises

TODO: Add more exercises for the techniques and features introduced in this section. Make sure what is here still make sense.

For each of the following exercises, develop and test a Haskell function or set of functions.

1. Develop a Haskell module (or modules) for line segments on the two-dimensional coordinate plane using the *rectangular coordinate* system.

   We can represent a line segment with two points–the starting point and the ending point. Develop the following Haskell functions:

   - constructor `newSeg` that takes two points and returns a new line segment

   - selectors `startPt` and `endPt` that each take a segment and return its starting and ending points, respectively

   We normally represent the plane with a *rectangular coordinate* system. That is, we use two axes–an `x` *axis* and a `y` *axis*–intersecting at a right angle. We call the intersection point the *origin* and label it with 0 on both axes. We normally draw the `x` axis horizontally and label it with increasing numbers to the right and decreasing numbers to the left. We also draw the `y` axis vertically with increasing numbers upward and decreasing numbers downward. Any point in the plane is uniquely identified by its `x`-coordinate and `y`-coordinate.

Define a data representation for points in the rectangular coordinate system and develop the following Haskell functions:

- constructor `newPtFromRect` that takes the `x` and `y` coordinates of a point and returns a new point
- selectors `getx` and `gety` that takes a point and returns the `x` and `y` coordinates, respectively
- display function `showPt` that takes a point and returns an appropriate `String` representation for the point

Now, using the various constructors and selectors, also develop the Haskell functions for line segments:

- `midPt` that takes a line segment and returns the point at the middle of the segment
- display function `showSeg` that takes a line segment and returns an appropriate `String` representation

Note that `newSeg`, `startPt`, `endPt`, `midPt`, and `showSeg` can be implemented independently from how the points are represented.

2. Develop a Haskell module (or modules) for line segments that represents points using the *polar coordinate* system instead of the rectangular coordinate system used in the previous exercise.

   A polar coordinate system represents a point in the plane by its *radial coordinate* `r` (i.e., the distance from the *pole*) and its *angular coordinate* `t` (i.e., the angle from the *polar axis* in the reference direction). We sometimes call `r` the *magnitude* and `t` the *angle*.

   By convention, we align the rectangular and polar coordinate systems by making the origin the pole, the positive portion of the `x` axis the polar axis, and let the first quadrant (where both `x` and `y` are positive) be the smallest positive angles in the reference direction. That is, with a traditional drawing of the coordinate systems, we measure and the radial coordinate `r` as the distance from the origin measure the angular coordinate `t` counterclockwise from the positive `x` axis.

   Using knowledge of trigonometry, we can convert among rectangular coordinates `(x,y)` and polar coordinates `(r,t)` using the equations:

   ```
   x = r * cos(t)
   y = r * sin(t)
   r = sqrt(x^2 + y^2)
   t = arctan2(y,x)
   ```

   Define a data representation for points in the polar coordinate system and develop the following Haskell functions:

- constructor `newPtFromPolar` that takes the magnitude `r` and angle `t` as the polar coordinates of a point and returns a new point
- selectors `getMag` and `getAng` that each take a point and return the magnitude `r` and angle `t` coordinates, respectively
- selectors `getx` and `gety` that return the `x` and `y` components of the points (represented here in polar coordinates)
- display functions `showPtAsRect` and `showPtAsPolar` to convert the points to strings using rectangular and polar coordinates, respectively,

Functions `newSeg`, `startPt`, `endPt`, `midPt`, and `showSeg` should work as in the previous exercise.

3. Modify the solutions to the previous two line-segment module exercises to enable the line segment functions to be in one module that works properly if composed with either of the two data representation modules. (The solutions may have already done this.)
4. Modify the solution to the previous line-segment exercise to use the Backpack module system.
5. Modify the modules in the previous exercise to enable the line segment module to work with both data representations in the same program.
6. Modify the solution to the Rational Arithmetic case study to use the Backpack module system.

Acknowledgements

I adapted and revised much of this work in Summer and Fall 2016 from my previous materials.

- Using Top-Down Stepwise Refinement (square root module) from my earlier implementations of this algorithm in Scala, Elixir, and Lua and from section 1.1.7 of Abelson and Sussman's *Structure and Interpretation of Computer Programs*
- Using Data Abstraction (rational arithmetic module) mostly from chapter 5 of my *Notes on Functional Programming with Haskell*, from my Lua-based implementations, and from section 2.1 of Abelson and Sussman's *Structure and Interpretation of Computer Programs*
- Modular Design and Programming from my Data Abstraction and Modular Design notes, which draw from the ideas of several of the references listed below

In 2017, I continued to develop this work as parts of chapters 1 and 2 of a draft textbook. In Spring 2018, I separated this material back into a more focused new chapter on abstraction.

I maintain these notes as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the notes to HTML, PDF, and other forms as needed.

References

[**Abelson-Sussman 1996**] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs* (SICP), Second Edition, MIT Press, 1996.

[**Bird-Wadler 1998**] Richard Bird and Philip Wadler. *Introduction to Functional Programming*, Second Edition, Addison Wesley, 1998. [First Edition, 1988]

[**Britton 1981**] K. H. Britton, R. A. Parker, and D. L. Parnas. "A Procedure for Designing Abstract Interfaces for Device Interface Modules," In *Proceedings of the 5th International Conference on Software Engineering*, pp. 195-204, March 1981.

[**Chiusano-Bjarnason 2015**]] Paul Chiusano and Runar Bjarnason, *Functional Programming in Scala*, Manning, 2015.

[**Cunningham 2001**] H. Conrad Cunningham and Jingyi Wang. Building a Layered Framework for the Table Abstraction, In *Proceedings of the ACM Symposium on Applied Computing*, pp. 668-674, March 2001.

[**Cunningham 2004**] H. Conrad Cunningham, Cuihua Zhang, and Yi Liu. Keeping Secrets within a Family: Rediscovering Parnas, In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, pp. 712-718, CSREA Press, June 2004.

[**Cunningham 2010**] H. Conrad Cunningham, Yi Liu, and Jingyi Wang. Designing a Flexible Framework for a Table Abstraction, In Y. Chan, J. Talburt, and T. Talley, editors, Data Engineering: Mining, Information, and Intelligence, pp. 279-314, Springer, 2010.

[**Cunningham 2014**] H. Conrad Cunningham. *Notes on Functional Programming with Haskell*, 1994-2014.

[**Dale-Walker 1996**] Nell Dale and Henry M. Walker. *Abstract Data Types: Specifications, Implementations, and Applications*, D. C. Heath, 1996. (Especially chapter 1 on "Abstract Specification Techniques")

[**Horstmann 1995**] Cay S. Horstmann. *Mastering Object-Oriented Design in C++*, Wiley, 1995. (Especially chapters 3-6 on "Implementing Classes", "Interfaces", "Object-Oriented Design", and "Invariants")

[**Meyer 1997**] Bertrand Meyer. *Object-Oriented Program Construction*, Second Edition, Prentice Hall, 1997. (Especially chapter 6 on "Abstract Data Types" and chapter 11 on "Design by Contract")

[**Mossenbock 1995**] Hanspeter Mossenbock. *Object-Oriented Programming in Oberon-2*, Springer-Verlag, 1995. (Especially chapter 3 on "Data Abstraction")

[**Parnas 1972**] D. L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules," *Communications of the ACM*, Vol. 15, No. 12, pp.1053-1058, 1972.

[**Parnas 1976**] D. L. Parnas. "On the Design and Development of Program Families," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 1, pp. 1-9, March 1976.

[**Parnas 1978**] D. L. Parnas. "Some Software Engineering Principles," *Infotech State of the Art Report on Structured Analysis and Design*, Infotech International, 10 pages, 1978. Reprinted in *Software Fundamentals: Collected Papers by David L. Parnas*, D. M. Hoffman and D. M. Weiss, editors, Addison-Wesley, 2001.

[**Parnas 1979**] D. L. Parnas. "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 1, pp. 128-138, March 1979.

[**Parnas 1985**] D. L. Parnas, P. C. Clements, and D. M. Weiss. "The Modular Structure of Complex Systems," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 3, pp. 259-266, March 1985.

[**Thompson 2011**] Simon Thompson. *Haskell: The Craft of Programming, Third Edition*, Pearson, 2011.

[**Weiss 2001**] D. M. Weiss. "Introduction: On the Criteria to Be Used in Decomposing Systems into Modules," In *Software Fundamentals: Collected Papers by David L. Parnas*, D. M. Hoffman and D. M. Weiss, editors, Addison-Wesley, 2001.

Terms and Concepts

Procedural abstraction, top-down stepwise refinement, abstract code, termination condition for recursion, Newton's method, Haskell `module`, module exports and imports, rational number arithmetic, data abstraction, properties of data, data representation, invariant, interface invariant, implementation or representation invariant, canonical or normal forms, information hiding, module secret, encapsulation, interface, abstract interface, design criteria for interfaces, software reuse, Haskell `where` local definition, type inference, use of Haskell modules to implement information-hiding modules rational number arithmetic, data abstraction, abstract properties of data, data representation, invariant, interface invariant, implementation or representation invariant, canonical or normal forms, information hiding, module secret, encapsulation, interface, abstract interface, design criteria for interfaces, software reuse, Haskell `where` local definition, type inference, use of Haskell modules to implement information-hiding modules