2IP15 Programming Methods

From Small to Large Programs

Tom Verhoeff
Eindhoven University of Technology
Department of Mathematics & Computer Science
Software Engineering & Technology Group

www.win.tue.nl/~wstomv/edu/2ip15

Overview

- Looking back
- Looking ahead
- An anecdote about design methodology

Source Code Should Be Written with Utmost Care

- Source code is not only intended for the compiler.

  Source code should be readable and verifiable by other engineers.

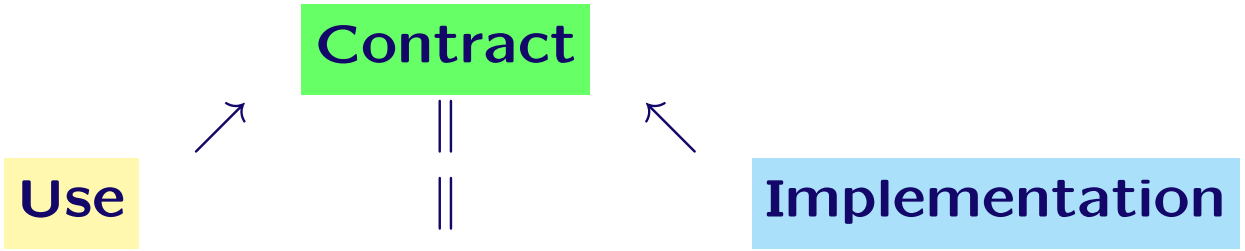
- Adhere to a coding standard:

  Pay attention to layout, (javadoc) comments, naming, structure.

- This prevents mistakes, and eases finding and repairing of defects.

Manage Complexity: Modularization

- Separation of Concerns: *Divide & Conquer* (and hence *Rule*)
- For each facility (function, type, iterator, package, . . .), separate



  Use and implementation are based on the (same) contract.

  Use and implementation are not directly 'coupled'.

- Always *program to an abstraction* (contract, interface), not to a 'concretion' (use, implementation).
- Divide & conquer serves many purposes, including maintainability.

Procedural Abstraction

- Abstract from data operated on (through parameters)
- Abstract from realization of operation (when viewed from usage)
- Abstract from context of use (when viewed from implementation)
- Guidelines for functional decomposition

Errors and Exceptions

IEEE terminology: failure, defect (fault, bug), mistake, error

For non-private methods:

- the precondition should be as weak as possible:
  unless unacceptable for performance reasons.
- the contract specifies relevant exceptions and their conditions:
  `@pre P` and `@throw E if ! P`

You are encouraged to check preconditions, e.g. via `assert`,
also in **`private`** methods. Assertion checking is disabled by default!

Data Abstraction

Abstract Data Type = set of (abstract) values and corresponding operations: construct, destroy, query, modify

In Java

Specification: `class` name, `public` method headers and contracts, public invariant
Optionally: public constant names

Implementation: `private` instance variables, private (rep) invariant, abstraction function, public method bodies
Optionally: public constant values, private methods

N.B. Variable of `class` type is a reference: *aliasing*!

Java Built-in Protections for Modularization

- Functions (class methods): local variables
- Data types (classes): instance variables, methods

Modifier	Access Level			
	Class	**Package**	**Subclass***	**World**
`private`	*Yes*	No	No	No
no modifier	*Yes*	*Yes*	No	No
`protected`	*Yes*	*Yes*	*Yes*	No
`public`	*Yes*	*Yes*	*Yes*	*Yes*

*Outside package

Step-by-step (Test-driven) ADT Development Plan

1. Gather and analyze requirements.
2. Choose requirement to develop next.
3. Specify class & methods informally: javadoc summary sentences.
4. Specify formally: model with invariants, signatures, and contracts.
   Class w/o implementation: no data rep, empty method bodies.
5. Create a corresponding unit test class.
6. Implement rigorous tests.
7. Choose data representation and implement class methods.
8. Test the implementation.
9. Refactor and retest.

Iteration Abstraction

- Problem: Visit each item of a collection exactly once
- Abstract from type of collection, type of items, how to iterate
- An *iterator object* maintains the state of the iteration.
- In general, an iterator object implements `Iterator<T>` providing methods **`boolean`** `hasNext()`, `T next()`, and optionally `remove()`.
- To use enhanced **`for`** statement, collection implements `Iterable<T>`, i.e., provides a method `iterator()` returning an `Iterator<T>`.
- **`for`** `(` *Type Identifier* `:` *Expression* `)` *Statement*

SOLID Object-Oriented Design Principles

- **S**ingle Responsibility Principle (SRP, see Lecture 2)
- **O**pen Closed Principle (OCP, see Lecture 10)
- **L**iskov Substitution Principle (LSP, see Lecture 4)
- **I**nterface Segregation Principle (ISP, see Lecture 13)
- **D**ependency Inversion Principle (DIP, see Lecture 8)

Goal: to manage change

Taxonomy of Design Patterns

- Creational patterns

  Factory Method, Singleton

- Structural patterns

  Composite, Façade, Adapter, Decorator

- Behavioral patterns

  Strategy, Iterator, Observer, Command, State, Template Method

- Concurrency patterns

  "`SwingWorker`"

Reflection on Method

- Adhering to a method (systematic way of working) has a price.
- It may cost extra development time, code, and execution time.
- Without method, complexity quickly becomes unmanageable, which costs even more.
- Method makes project and product quality better controllable.
- Method reduces time for getting it to work: defect localization and correction, change and reuse.

Looking Ahead

- Techniques to develop algorithms: stepwise refinement

  Correctness by construction

  Example: *MaxSegSum*

- JML: Java Modeling Language

  Tools to support formally verified software development

- Domain-Specific Languages (DSLs), language technology

  Meta-models, models

  - Text-to-Model (T2M): lexical analysis, parsing
  - Model-to-Model (M2M): transformation
  - Model-to-Text (M2T): formatting, source code generation

In the Hoare triple

$$\{\, P \,\} \quad S \quad \{\, Q \,\}$$

- P is a predicate on the program variables, called precondition
- S is a program (fragment) involving some program variables
- Q is a predicate on the program variables, called postcondition

Hoare triple $\{\, P \,\}\ S\ \{\, Q \,\}$ is valid if and only if

> every execution of S that starts in a state satisfying P terminates in a state satisfying Q.

Hoare Triples: Example

For integer program variables x and r, the Hoare triple

$$\{\, x \geq 0 \,\} \quad \mathit{SquareRoot} \quad \{\, 0 \leq r \;\wedge\; r^2 \leq x < (r+1)^2 \,\}$$

expresses that $\mathit{SquareRoot}$ computes the integer square root r of x, where x is known to be non-negative.

Axiom of $skip$

$skip$ is the 'do-nothing' statement characterized by the axiom:

$\{\, P \,\}\ skip\ \{\, Q \,\}$ holds iff $[P \Rightarrow Q]$

The *weakest* precondition of $skip$ for postcondition Q is Q.

Assignment

Incorrect (and not so useful if it were correct):

$$\{\, true \,\}\ x := E\ \{\, x = E \,\}$$

Incorrect in case E contains x: e.g. $\{\, true \,\}\ x := x + 1\ \{\, x \stackrel{?}{=} x + 1 \,\}$

Not so useful: postcondition is not general and other variables are not taken into account

Consider a payment from your wallet:

$$\{\, P \,\}\ wallet := wallet - drink\ \{\, wallet \geq fruit \,\}$$

What is the weakest precondition P to leave enough money for fruit after buying a drink?

Axiom of Assignment

Assignment to program variable x is characterized by the axiom

$\{\, P \,\}\ x := E\ \{\, Q \,\}$ holds iff $[P \Rightarrow Q(x := E)]$

where $Q(x := E)$ is obtained from Q by substituting E for every free x.

The weakest precondition of $x := E$ for postcondition Q is $Q(x := E)$.

When E is not defined in all states: also require $[P \Rightarrow \text{def}.E]$.

Axiom of Assignment: Example

What is the weakest precondition P for

$$\{\, P \,\}\ wallet := wallet - drink\ \{\, wallet \geq fruit \,\}$$

We calculate:

$$\begin{array}{ll} & (wallet \geq fruit)(wallet := wallet - drink) \\ \equiv & \quad \{\ \text{subst.}\ \} \\ & wallet - drink \geq fruit \\ \equiv & \quad \{\ \text{algebra}\ \} \\ & wallet \geq fruit + drink \end{array}$$

MaxSegSum

Given integer array $f[0..N)$ with $0 \leq N$.

Determine maximum sum of all consecutive segments of f

MaxSegSum: Formal Specification

$$
\begin{array}{l}
[\![\ \textbf{con}\ N\colon\ int;\ \{\,0 \le N\,\} \\
\qquad\quad f\colon\ \textbf{array}\ [0..N)\ \textbf{of}\ int; \\
\quad \textbf{var}\ r\colon\ int; \\
\quad MaxSegSum \\
\quad \{\,R : r = (\max a, b : 0 \le a \le b \le N : S.a.b)\,\} \\
]\!]
\end{array}
$$

where

$$S.a.b \;=\; (\sum\, i : a \le i < b : f.i)$$

for $0 \le a \le b \le N$.

The problem is trivial if f contains no negative numbers: $S.0.N$.

History: $\mathcal{O}(N^3)$ solution is easy; $\mathcal{O}(N^2)$ ...; $\mathcal{O}(N \log N)$ ...

MaxSegSum: Invariant

A loop is needed to traverse the array

Generalize postcondition: replace constant N by variable n

Rewrite postcondition $R : r = G.N$, where

$$G.n \;=\; (\max a, b : 0 \leq a \leq b \leq n : S.a.b)$$

Invariant: $P.1 : r = G.n$ with $P.0 : 0 \leq n \leq N$

Finalization: guard $n \neq N$

Initialization: $r, n := 0, 0$, because $G.0 = 0$

MaxSegSum: Partial Program

$$
\begin{array}{l}
[\![\ \textbf{var}\ n\colon\ int; \\
\quad r, n := 0, 0 \\
\ ;\{\,\text{inv}\ P.0 : 0 \le n \le N,\ P.1 : r = G.n\,\}\ \{\,\text{bnd}\ N - n\,\} \\
\quad \textbf{do}\ n \neq N \rightarrow \{\,0 \le n < N \wedge P.1\,\} \\
\quad\ ;\quad \text{“update } r\text{”}\ \{\,(P.0 \wedge P.1)(n := n + 1)\,\} \\
\quad\ ;\quad n := n + 1\ \{\,P.0 \wedge P.1\,\} \\
\quad \textbf{od}\ \{\,P.0 \wedge P.1 \wedge n = N\,\} \\
\quad \{\,R : r = G.N\,\} \\
]\!]
\end{array}
$$

The hole “update r” still needs to be refined

MaxSegSum: Establish $P.1(n := n+1)$

Concerning $P.1(n := n+1)$ calculate

$$
\begin{array}{ll}
 & G.(n+1) \\
= & \quad \{ \text{ def. of } G.n \ \} \\
 & (\max a, b : 0 \le a \le b \le n+1 : S.a.b) \\
= & \quad \{ \text{ split off } b = n+1, \text{ using that } 0 \le n+1 \ \} \\
 & (\max a, b : 0 \le a \le b \le n : S.a.b) \max (\max a : 0 \le a \le n+1 : S.a.(n+1)) \\
= & \quad \{ \text{ def. of } G.n, \bullet \text{ def. } H.n = (\max a : 0 \le a \le n : S.a.n) \ \} \\
 & G.n \max H.(n+1) \\
= & \quad \{ P.1, \text{ introduce } s \text{ satisfying } P.2(n := n+1) \text{ with } P2 : s = H.n \ \} \\
 & r \max s
\end{array}
$$

Extra variable s introduced with strengthened invariant ($P.2$)

MaxSegSum: Partial Program

$$
\begin{array}{l}
[\![\ \textbf{var}\ n,\ s:\ int; \\
\quad r, n, s := 0, 0, 0 \\
\quad ;\{\,\text{inv}\ P.0 : 0 \le n \le N,\ P.1 : r = G.n,\ P.2 : s = H.n\,\}\ \{\,\text{bnd}\ N - n\,\} \\
\quad \textbf{do}\ n \neq N \rightarrow \{\,0 \le n < N \wedge P.1 \wedge P.2\,\} \\
\qquad ;\quad \text{“update } s\text{”}\ \{\,P.0(n := n+1) \wedge P.1 \wedge P.2(n := n+1)\,\} \\
\qquad ;\quad r := r \max s\ \{\,(P.0 \wedge P.1 \wedge P.2)(n := n+1)\,\} \\
\qquad ;\quad n := n+1\ \{\,P.0 \wedge P.1 \wedge P.2\,\} \\
\quad \textbf{od}\ \{\,P.0 \wedge P.1 \wedge P.2 \wedge n = N\,\} \\
\quad \{\,R : r = G.N\,\} \\
]\!]
\end{array}
$$

The hole “update s” still needs to be refined

MaxSegSum: Establish $P.2(n := n+1)$

Concerning $P.2(n := n+1)$ calculate

$$
\begin{array}{ll}
 & H.(n+1) \\
= & \quad \{ \text{ def. of } H.n \ \} \\
 & (\max a : 0 \le a \le n+1 : S.a.(n+1)) \\
= & \quad \{ \text{ split off } a = n+1, \text{ using that } 0 \le n+1 \ \} \\
 & (\max a : 0 \le a \le n : S.a.(n+1)) \max S.(n{+}1).(n{+}1) \\
= & \quad \{ \ S.a.(n{+}1) = S.a.n + f.n \text{ if } a < n; \text{ and } S.(n{+}1).(n{+}1) = 0 \ \} \\
 & (\max a : 0 \le a \le n : S.a.n + f.n) \max 0 \\
= & \quad \{ \ + \text{ distributes over max for non-empty range: } 0 \le n \ \} \\
 & ((\max a : 0 \le a \le n : S.a.n) + f.n) \max 0 \\
= & \quad \{ \text{ def. of } H.n \ \} \\
 & (H.n + f.n) \max 0 \\
= & \quad \{ \ P.2 \ \} \\
 & (s + f.n) \max 0
\end{array}
$$

MaxSegSum: Complete Program

$$
\begin{array}{l}
[\![\ \textbf{var}\ n, s\colon\ int; \\
\quad r, n, s := 0, 0, 0 \\
\ ;\{\,\text{inv}\ P.0 : 0 \le n \le N,\ P.1 : r = G.n,\ P.2 : s = H.n\,\}\ \{\,\text{bnd}\ N - n\,\} \\
\quad \textbf{do}\ n \neq N \rightarrow \{\,0 \le n < N \wedge P.1 \wedge P.2\,\} \\
\qquad s := (s + f.n)\ \max\ 0\ \{\,P.0(n := n+1) \wedge P.1 \wedge P.2(n := n+1)\,\} \\
\quad\ ;\quad r := r\ \max\ s\ \{\,(P.0 \wedge P.1 \wedge P.2)(n := n+1)\,\} \\
\quad\ ;\quad n := n + 1\ \{\,P.0 \wedge P.1 \wedge P.2\,\} \\
\quad \textbf{od}\ \{\,P.0 \wedge P.1 \wedge P.2 \wedge n = N\,\} \\
\quad \{\,R : r = G.N\,\} \\
]\!]
\end{array}
$$

Complexity: $\mathcal{O}(N)$, i.e. linear

MaxSegSum: Stripped Program

$$
\begin{array}{l}
[\![\ \textbf{var}\ n, s\colon\ int; \\
\quad r, n, s := 0, 0, 0 \\
\ ;\textbf{do}\ n \neq N\ \rightarrow \\
\qquad s := (s + f.n)\ \max\ 0 \\
\quad ;\quad r := r\ \max\ s \\
\quad ;\quad n := n + 1 \\
\quad \textbf{od} \\
\quad \{\, r = (\max\, a, b : 0 \leq a \leq b \leq N : S.a.b) \,\} \\
\quad \{\, \text{where}\ S.a.b = (\sum\, i : a \leq i < b : f.i) \,\} \\
]\!]
\end{array}
$$

Magically simple program text, but hard to understand operationally

Correct by construction and calculation

Could you discover this without calculations?

Brilliance

Michael Jackson

Software Requirements & Specifications:
a lexicon of practice, principles and prejudices

Addison-Wesley, 1995

ISBN 0–201–87712–0

www.win.tue.nl/~wstomv/quotes/software-requirements-specifications.html

Brilliance

Some years ago I spent a week giving an in-house program design course at a manufacturing company in the mid-west of the United States.

On the Friday afternoon it was all over.

The DP Manager, who had arranged the course and was paying for it out of his budget, asked me into his office.

Brilliance

'What do you think?' he asked.

He was asking me to tell him my impressions of his operation and his staff.

'Pretty good,' I said. 'You've got some good people there.'

Program design courses are hard work; I was very tired; and staff evaluation consultancy is charged extra.

Anyway, I knew he really wanted to tell me his own thoughts.

Brilliance

'What did you think of Fred?' he asked. 'We all think Fred is brilliant.'

'He's very clever,' I said. 'He's not very enthusiastic about methods, but he knows a lot about programming.'

'Yes,' said the DP Manager. He swivelled round in his chair to face a huge flowchart stuck to the wall: about five large sheets of line printer paper, maybe two hundred symbols, hundreds of connecting lines.

Brilliance

'Fred did that. It's the build-up of gross pay for our weekly payroll. No one else except Fred understands it.'

His voice dropped to a reverent hush.

'Fred tells me that he's not sure he understands it himself.'

Brilliance

'Terrific,' I mumbled respectfully. I got the picture clearly.

Fred as Frankenstein, Fred the brilliant creator of the uncontrollable monster flowchart.

That matched my own impression of Fred very well.

'But what about Jane?' I said. 'I thought Jane was very good. She picked up the program design ideas very fast.'

Brilliance

'Yes,' said the DP Manager. 'Jane came to us with a great reputation. We thought she was going to be as brilliant as Fred. But she hasn't really proved herself yet.

We've given her a few problems that we thought were going to be really tough, but when she finished it turned out they weren't really difficult at all. Most of them turned out pretty simple.

She hasn't really proved herself yet — if you see what I mean?'

I saw what he meant.

(End of quote)