The Seven Principles of Software Engineering

C. Ghezzi, M. Jazayeri, D. Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, 1991.

Rigour and Explicitness

Rigour and explicitness are a necessary complement to unstructured creativity.

Separation of Concerns

Separately deal with different individual aspects of the problem (such as time, qualities, views, size).

Modularity

Identify *modules* (units of division of work), then deal with intra-module and inter-module details.

Abstraction

Identify and focus on the important aspects of the problem, thus obtaining purpose-specific *models*.

Anticipation of Change

Identify aspects of the product and process that are likely to change, and protect from their changes.

Generality

Solving a more general (less constrained) problem is often easier, and provides reuse opportunities.

Incrementality

Successively produce better approximations to a solution by improving on the previous solution.

The Impact of Bad Specifications

Specification errors are the most *numerous* errors:

- 64% of all errors are specification errors.
- 36% of all errors are programming errors.

Specification errors are the most *tenacious* errors:

- 19% of all errors are specification errors and are detected before delivery.
- 45% of all errors are specification errors and are detected *after* delivery.
- 9% of all errors are programming errors and are detected before delivery.
- 27% of all errors are programming errors and are detected after delivery.

Specification errors are the most *costly* errors:

- A specification error caught while designing costs 2.5 times more than while specifying.
- A specification error caught while programming costs 5.0 times more than while specifying.
- A specification error caught while integrating costs 36.0 times more than while specifying.

Considering that:

- Correcting specification errors represents 66% of the total error correction cost.
- Correcting design errors represents 25% of the total error correction cost.
- Correcting programming errors represents 9% of the total error correction cost.

and considering that the total error correction cost represents 50% of the total cost of a software, then we have that correcting specification errors represents 33% of the total cost of a software!

Specification Template (for the CCP course)

Given ⟨arguments and their types⟩
[*such that* ⟨pre-condition on arguments⟩],
program ⟨name⟩ [*modifies* ⟨some arguments⟩ *and*] *returns* ⟨results and their types⟩
such that ⟨post-condition on arguments and results⟩
[, *without modifying* ⟨some remaining arguments⟩].
[*Examples*: ⟨…⟩.]
[*Counter-examples*: ⟨…⟩.]

Role of the Pre-Condition

- If the pre-condition on the arguments does *not* hold,
  then the program *may* return *any* results!
- If the pre-condition on the arguments *does* hold,
  then the program *must* return results that satisfy the post-condition!

Role of Well-Chosen (Counter-)Examples

- In *theory*: They are redundant with the pre/post-conditions.
- In *practice*:
  + They often provide an intuitive understanding that no assertion or definition could achieve.
  + They often help eliminate risks of ambiguity in the assertions by illustrating delicate issues.
  + If they contradict the pre/post-conditions, then we know that something is wrong somewhere!

A Sample Specification

Given two integer-arrays A[1..M] and B[1..N]
such that A and B are non-decreasingly ordered,
program merge *returns* an integer-array C[1..M+N]
such that C is the non-decreasingly ordered permutation of the union of A and B,
without modifying A and B.
Example: merge ([1,4,4,6] , [2,3,4] , [1,2,3,4,4,4,6]).

Comments

- The used concepts of "non-decreasingly ordered array", "permutation of an array", and "union of two arrays" are assumed to be understood by the reader in the same way as by the writer. This also explains the role of the examples.
- The program must be *deterministic*, because "C is *the* …", and not "C is *a* …".
- The array upper bounds M and N are *implicit arguments* (and should thus also not be modified).
- *Formalising* specifications (as advocated by many) often gives rise to long formulas (see the next slide for a sample formalisation), which is unnecessary for our objective:
  + We aim at the *manual* construction of correct programs, not at their *automated* construction: constructing those long formulas and manually manipulating them would be more errorprone.
  + We aim at the manual *construction* of correct programs, not at their automated *verification*.
  + New formal symbols need to be informally explained anyway, so that one can verify (!) whether they indeed capture the informal intentions.

Example: Formalisation of the Specification of a merge Program

Pre-condition: ordered(A,1,M) and ordered(B,1,N)

Post-condition: permutation(A,M,B,N,C) and ordered(C,1,M+N)

where:

- ordered(X,L,U) if-and-only-if for all integers I such that $L \leq I < U$ we have that

$$X[I] \leq X[I+1]$$

  (i.e., integer-array X[L..U] is non-decreasingly ordered)

- permutation(A,U,B,V,C) if-and-only-if for all integers I we have that

$$\text{number}(I,C,U+V) = \text{number}(I,A,U) + \text{number}(I,B,V)$$

  (i.e., integer-array C[1..U+V] is a permutation of the union of integer-arrays A[1..U] and B[1..V])

where:

- number(E,X,U) = the number of integers J such that $1 \leq J \leq U$ where we have that

$$X[J] = E$$

  (i.e., the number of occurrences of integer E in integer-array X[1..U])

The Seven Sins of the Specifier

Source: Bertrand Meyer. On Formalism in Specifications. *IEEE Software* 2(1):6–26, 1985.

Noise	The presence in the text of an element that doesn't carry information relevant to any feature of the problem. Variants: **Redundancy**, **Remorse**.
Silence	The existence of a feature of the problem that is not covered by any element of the text.
Overspecification	The presence in the text of an element that corresponds not to a feature of the problem but to features of a possible solution.
Contradiction	The presence in the text of two or more elements that define a feature of the problem in an incompatible way.
Ambiguity	The presence in the text of an element that makes it possible to interpret a feature of the problem in at least two different ways.
Forward Reference	The presence in the text of an element that uses features of the problem not defined until later in the text.
Wishful Thinking	The presence in the text of an element that defines a feature of the problem in such a way that a candidate solution cannot realistically be validated with respect to this feature.

The Programming Language

Data Types

- Booleans: *boolean* (values: *true* and *false*)
- Integers: *integer* (values: …, –3, –2, –1, 0, 1, 2, 3, …)
- Arrays: *array*[⟨lowbound⟩..⟨upbound⟩] *of* ⟨type⟩ (empty when lowbound = upbound + 1)

Primitive Statements

- Simple Assignment: ⟨variable⟩ ← ⟨expression⟩

Composition Mechanisms

- Sequential Composition: ⟨statement⟩ ; ⟨statement⟩
- Conditional Composition:
  if ⟨condition⟩ *then*
  ⟨statement⟩
  [*else*
  ⟨statement⟩]
  fi
- Iterative Composition:
  while ⟨condition⟩ *do*
  ⟨statement⟩
  od

Program Correctness

Definition: The *state* of a program P at a moment M consists of the values of the variables of P at M.

Definition: An *assertion* is an affirmation regarding a program state.
Examples: The pre/post-conditions of specifications and proof invariants (see below) are assertions.

Definition: A program P is *partially correct* with respect to a specification S
if, each time P terminates on arguments that satisfy the pre-condition (including the types) of S,
P returns results that satisfy the post-condition (including the types) of S.

Definition: A program P is (*totally*) *correct* with respect to a specification S
if P terminates on all arguments that satisfy the pre-condition (including the types) of S
and P is partially correct with respect to S.

Notation

Let P be a program statement, and let Q and R be assertions involving the variables of P.
Then the notation:

$$\{ Q \} \ P \ \{ R \}$$

means that P is totally correct w.r.t. the specification with pre-condition Q and post-condition R.

Hoare's Semantic Laws

Simple Assignment

$$\{ Q[X/E] \} \; X \leftarrow E \; \{ Q[X] \} \quad \text{(to be read from right to left)}$$

$$\text{or:} \quad \{ Q[X] \} \; X \leftarrow E \; \{ Q[X/X_0] \text{ and } X = E[X/X_0] \} \quad \text{(to be read from left to right)}$$

(where X_0 is the initial value of X)

Sequential Composition

if $\{ Q \} \; P_1 \; \{ R \}$ **and** $\{ R \} \; P_2 \; \{ S \}$
then $\{ Q \} \; P_1 \; ; P_2 \; \{ S \}$

Conditional Composition

if $\{ Q \text{ and } B \} \; P_1 \; \{ R \}$ **and** $\{ Q \text{ and not } B \} \; P_2 \; \{ R \}$
then $\{ Q \}$ *if* B *then* P_1 *else* P_2 *fi* $\{ R \}$

Iterative Composition

if $\{ \text{Inv and } B \} \; P \; \{ \text{Inv} \}$
then $\{ \text{Inv} \}$ *while* B *do* P *od* $\{ \text{Inv and not } B \}$

Proving Programs by Computational Induction

Given a specification S, with pre-condition Pre and post-condition Post, and a program P of the form:

⟨initialisation⟩ ;
while ⟨condition⟩ *do*
  ⟨body⟩
od
[; ⟨conclusion⟩]

a *proof by computational induction* of total correctness of P with respect to S proceeds in 2 steps:

1. **Proof of Partial Correctness** of P with respect to S:
   Find an assertion Inv, called the *invariant*, that holds each time ⟨condition⟩ is evaluated, expressing *what has already been done so far*, i.e., prove that Inv indeed holds the 1st time:

   { Pre } ⟨initialisation⟩ { Inv } (1)

   and prove that after the last time (i.e., when the loop ends), the program terminates correctly:

   { Inv *and not* ⟨condition⟩ } ⟨conclusion⟩ { Post }
   or: Inv *and not* ⟨condition⟩ *implies* Post (2)

   and prove that if Inv holds the nth time, then Inv will indeed hold the $n+1^{st}$ time, if any:

   { Inv *and* ⟨condition⟩ } ⟨body⟩ {Inv} (3)

2. **Proof of Termination** of P on all arguments that satisfy Pre:
   For instance, find for each loop an integer function F on the program variables, called the *variant*, that is decreasing towards a lower bound during each iteration, i.e., prove that F indeed returns a lower-bounded integer:

   for all arguments satisfying Pre, function F returns a lower-bounded integer (4)

   and prove that execution of the loop body indeed decreases the value of F:

   $$\{\ \text{Inv}\ \textit{and}\ \langle\text{condition}\rangle\ \textit{and}\ F(\ldots) = f\ \}\ \ \langle\text{body}\rangle\ \ \{\ F(\ldots) < f\ \} \qquad (5)$$

   Other proof methods exist. For instance, the variant may increase towards an upper bound.

Comments on Program Proving by Computational Induction

- Distinguish between *specification variables*, *program variables*, and *proof variables*.
- The invariant often is "similar" to the post-condition Post.
- The invariant (resp. variant) *cannot* be correct (except for weird programs) if it does not involve all (resp. some of) the variables that appear in ⟨condition⟩ or that are modified by ⟨body⟩.
- The proof *cannot* be correct (except for over-specific specifications) if it does not appeal to the whole pre-condition and the whole conditions of all *if…then…else* and *while* statements.
- The proof *cannot* be correct if the invariant and the variant do not satisfy *each* of their conditions.
- If a proof step fails, then backtrack to a previous proof step and fix it or change the (in)variant.

Evaluation of Program Proving by Computational Induction

- **Advantages:**
  + The methodology really produces *proofs* in the classical understanding of the term, because they are based on axioms and inference rules.
  + The proof reasoning is made on the (static) *text* of the program, but not on its multiple — often infinitely many — (dynamic) executions. Program proving is thus more powerful than program testing!
  + Whereas program testing only aims at *detecting* the existence of errors, program proving is likely to also help in *locating* the errors and in *correcting* them.
- **Disadvantages:**
  – Correctness proofs only prove (if correct!) the correctness of the program w.r.t. its specification, but *nothing* regarding the hardware and software platform on which the program will be run.
- Program proving and program testing are thus *complementary*, and also prototyping.
- (In)variants may be *difficult* to find for (uncommented) programs that one has not written oneself, and explicit, rigorous proofs may be *long and difficult*.
  This is *not* a disadvantage of program proving (compared to program testing), but rather evidence that *programming itself is difficult* !
  Proofs only make explicit the reasoning that was — or ought to have been — made anyway.
  Explicitly doing such proofs teaches us the risks we take when relying on untrained intuition.
- Is it possible to use this program proving methodology constructively, that is to actually *construct* correct programs right away? Yes, see the next chapter!
  Program constructing is *easier* than proving, because one controls the actual solution process.

Constructing Programs by Comp'l Induction

Given a specification S, with pre-condition Pre and post-condition Post,
a *construction by computational induction* of a program P of the form:

⟨initialisation⟩ ;
while ⟨condition⟩ *do*
  ⟨body⟩
od
[; ⟨conclusion⟩]

such that P is totally correct with respect to S proceeds in 6 steps:

1. **Intuitive Idea:** Describe the solution idea that you will follow during the program construction.
2. **General Situation:** Using an *invariant* (assertion) Inv, describe the program state that is to always hold before ⟨condition⟩ is evaluated, expressing *what has already been done so far*. Using a *variant* (function) F, describe the integer quantity that is to change during each iteration.
3. **Initialisation:** Infer ⟨initialisation⟩ such that:

$$\{\ \text{Pre}\ \}\ \langle\text{initialisation}\rangle\ \{\ \text{Inv}\ \} \tag{1}$$

4. **Loop-Condition and Conclusion:** Infer ⟨condition⟩ and ⟨conclusion⟩ (if necessary) such that:

$$\{\ \text{Inv}\ \textit{and not}\ \langle\text{condition}\rangle\ \}\ \langle\text{conclusion}\rangle\ \{\ \text{Post}\ \}$$
$$\textbf{or:}\quad \text{Inv}\ \textit{and not}\ \langle\text{condition}\rangle\quad \textit{implies}\quad \text{Post} \tag{2}$$

5. **Loop-Body:** Infer ⟨body⟩ such that:

$$\{\ \text{Inv}\ \textit{and}\ \langle\text{condition}\rangle\ \}\ \ \langle\text{body}\rangle\ \ \{\text{Inv}\} \tag{3}$$

and such that:

$$\text{for all arguments satisfying Pre, function F returns a lower-bounded integer} \tag{4}$$

and such that:

$$\{\ \text{Inv}\ \textit{and}\ \langle\text{condition}\rangle\ \textit{and}\ \text{F}(\ldots) = \text{f}\ \}\ \ \langle\text{body}\rangle\ \ \{\ \text{F}(\ldots) < \text{f}\ \} \tag{5}$$

Other methods exist. For instance, the variant may increase towards an upper bound.

6. **Documentation:** Comment the resulting program with at least its specification, and comment each of its loops with its invariant and variant.

Comments on Program Construction by Computational Induction

- Use diagrams and introduce appropriate notations and properties whenever convenient.
- If Step 5 in turn needs a loop, then choose an appropriate methodology and apply it.
- Devising (in)variants is *not* an additional and artificial difficulty in program construction, but rather an essential step thereof, even if not made explicit or unconscious.
- Program constructing is *easier* than proving, because one controls the actual solution process.

The Induction Principle

Let P be a conjectured property of natural numbers (i.e., the integers that are ≥ 0).
(For example, "the factorial of any natural number N is larger than or equal to N" is such a property.)

To prove that P(N) holds for *any* natural number N, we proceed in 2 independent steps:

- **Base Case:** Prove that P(0) holds.
- **Step Case:** Prove that P(N) holds for $N > 0$, assuming that P(M) holds for all $0 \leq M < N$.

Indeed, once these two cases are proven,
they interact so as to achieve that P(N) holds for *any* natural number N:

1. By the base case, we have that P(0) holds, unconditionally.
2. By the step case, we have that P(1) holds, because P(0) holds by 1.
3. By the step case, we have that P(2) holds, because P(0), P(1) hold by 1, 2.
4. By the step case, we have that P(3) holds, because P(0), P(1), P(2) hold by 1, 2, 3.
5. … and so on, until infinity! …

This principle, known as *complete mathematical induction*, can be generalised for any domain (not just natural numbers), whether lower-bounded or upper-bounded.

Proving Programs by Structural Induction

Given a specification S_P, with pre-condition Pre_P and post-cond. $Post_P$, and a program P of the form:

⟨initialisation⟩ ;
while ⟨condition⟩ *do*
  ⟨body⟩
od
[; ⟨conclusion⟩]

a *proof by structural induction* of total correctness of P with respect to S_P proceeds in 3 steps:

1. **Identification** of an *auxiliary program* A within P, which performs *what remains to be done*, and identification of a specification S_A for A, with pre-condition Pre_A and post-cond. $Post_A$, such that some function F on its variables has a lower-bounded domain according to Pre_A:
2. **Proof of Total Correctness** of A with respect to S_A:
   Prove the *base case*:

$$\{\ Pre_A \textit{ and } F(\ldots) = f_0 \text{ with } f_0 \text{ minimal}\ \}\ \ A\ \ \{\ Post_A[f_0]\ \} \tag{1}$$

   and prove the *step case*:

$$\{\ Pre_A \textit{ and } F(\ldots) = f_i \text{ with } f_i \text{ non-minimal}\ \}\ \ A\ \ \{\ Post_A[f_i]\ \}$$
$$\textbf{if}\ \ \{\ Pre_A \textit{ and } F(\ldots) = f_j < f_i\ \}\ \ A\ \ \{\ Post_A[f_j]\ \} \tag{2}$$

3. **Proof of Total Correctness** of P with respect to S_P, assuming total correctness of A w.r.t. S_A:

$$\{\ Pre_P\ \}\ \ \langle\text{initialisation}\rangle\ ;\ A\ \ \{\ Post_P\ \} \tag{3}$$

Comments on Program Proving by Structural Induction

- Distinguish between *specification variables*, *program variables*, and *proof variables*.
- The auxiliary program A often is the given program P without its initialisation statements: the methodology of structural induction amounts to "jumping on the running train."
- The post-condition Post_A of the auxiliary program A is a *generalisation* of Post_P.
- The post-condition of the auxiliary program A *cannot* be correct (except for weird programs) if it does not involve all the variables that appear in ⟨condition⟩ or that are modified by ⟨body⟩.
- The proof *cannot* be correct (except for over-specific specifications) if it does not appeal to the whole pre-condition and the whole conditions of all *if*…*then*…*else* and *while* statements.
- If a proof step fails, then backtrack to a previous proof step and fix it, or change the auxiliary program A, or change the specification of A.

Evaluation of Program Proving by Structural Induction

- *Advantages and disadvantages*: the same as for program proving by computational induction.
- Specs of auxiliary programs may be *difficult* to find for programs that one has not written oneself, and explicit, rigorous proofs may be *long and difficult*.
  This is *not* a disadvantage of program proving (compared to program testing),
  but rather evidence that *programming itself is difficult* !
  Proofs only make explicit the reasoning that was — or ought to have been — made anyway.
  Explicitly doing such proofs teaches us the risks we take when relying on untrained intuition.
- Is it possible to use this program proving methodology constructively,
  that is to actually *construct* correct programs right away? Yes, see the next chapter!
  Program constructing is *easier* than proving, because one controls the actual solution process.

Constructing Programs by Struct'l Induction

Given a specification S_P, with pre-condition Pre_P and post-condition $Post_P$,
a *construction by structural induction* of a program P of the form:

⟨initialisation⟩ ;
while ⟨condition⟩ *do*
  ⟨body⟩
od
[; ⟨conclusion⟩]

such that P is totally correct with respect to S_P proceeds in 5 steps:

1. **Intuitive Idea:** Describe the solution idea that you will follow during the program construction.
2. **Generalisation:** Using a specification S_A, with pre-condition Pre_A and post-condition $Post_A$, of an *auxiliary program* A, which performs *what remains to be done*, generalise the specification S_P such that some function F on its variables has a lower-bounded domain according to Pre_A:
3. **Auxiliary Program:** Infer ⟨condition⟩ and ⟨conclusion⟩ (if necessary) such that:

$$\{\ Pre_A \textit{ and } F(\ldots) = f_0 \text{ with } f_0 \text{ minimal}\ \}\ \ A\ \ \{\ Post_A[f_0]\ \} \qquad (1)$$

and infer ⟨body⟩ such that:

$$\{\ Pre_A \textit{ and } F(\ldots) = f_i \text{ with } f_i \text{ non-minimal}\ \}\ \ A\ \ \{\ Post_A[f_i]\ \}$$
$$\textbf{if}\ \ \{\ Pre_A \textit{ and } F(\ldots) = f_j < f_i\ \}\ \ A\ \ \{\ Post_A[f_j]\ \} \qquad (2)$$

4. **Initialisation:** Infer ⟨initialisation⟩ such that:

$$\{\ \mathrm{Pre_P}\ \}\ \langle\text{initialisation}\rangle\ ;\ \mathrm{A}\ \ \{\ \mathrm{Post_P}\ \} \tag{3}$$

5. **Documentation:** Comment the resulting program with at least its specification, and comment each auxiliary program with its specification.

Comments on Program Construction by Structural Induction

- Use diagrams and introduce appropriate notations and properties whenever convenient.
- If Step 3 in turn needs a loop, then choose an appropriate methodology and apply it.
- Specifying auxiliary programs is *not* an additional & artificial difficulty in program construction, but rather an essential step thereof, even if not made explicit or unconscious.
- Program constructing is *easier* than proving, because one controls the actual solution process.

Computational Induction vs. Structural Induction

Summary and Comparison

In a proof / construction by *computational* induction,
partial correctness of the *entire* program is established
by proving that an *invariant* holds each time the loop-condition is evaluated.
This invariant expresses *what has already been done so far*, after some iterations.
This is proven by *simple* mathematical induction on the *computation* length (the number of iterations).
Termination of the entire program is established separately.
Criticism: It is unnecessary to prove partial correctness and termination separately,
and it is less natural to reason on the computation length rather than on the values of the variables.

In a proof / construction by *structural* induction,
total correctness of the *auxiliary* program is established
by proving that its *post-condition* holds after a finite number of iterations.
This post-condition expresses *what remains to be done*, after some iterations.
This is proven by *simple / complete* mathematical induction on the *structure* of some variable.
Total correctness of the entire program is established
from the total correctness of the auxiliary program and the *initialisation* statements.
Criticism: It is less natural to wonder “what remains to be done” than “what has already been done.”

How to Choose the “Right” Methodology?

The choice of a proof/construction methodology is neither arbitrary, nor a matter of personal taste, as, for a given problem, the reasoning or program may be much simpler in one method than in the other. Some choice *heuristics* can be formulated:

- The methodology of *structural* induction is definitely superior
  when the question of “what remains to be done?” *can* be answered *easily*
  without referring to “what has already been done so far.”
  (Example: binary search.)
- Conversely, the methodology of *computational* induction seems superior
  when the question of “what remains to be done?” *can* only be answered *with difficulty*
  without referring to “what has already been done so far.”
  However, a good specialisation of the specification of the auxiliary program can often be found.
  (Example: the plateau problem.)
- Finally, the methodology of *computational* induction seems superior
  when the question of “what remains to be done?” *cannot* be answered
  without referring to “what has already been done so far.”
  However, a good specialisation of the specification of the auxiliary program can often be found.
  (Example: array compression.)

Consequences of Applying Either Methodology

Experience shows that much “better” programs are constructed
when following either of these two methodologies rather than reasoning outside them!