Ignorance Hiding

Based on:

Programmer-Client Interaction in Writing Program Specifications

Daniel M. Berry
Orna Berry

Problem -1

Most software is produced by a professional programmer to meet requirements of a client who is not a programming or computing professional.

Problem -2

All too often, the resultant software is not what the client wants.

- **The programmer misunderstood what client wants.**
- **The client misunderstood what he or she wants.**

Problem -3

The software, though correct with respect to the specifications does;

- **too little**
- **too much**
- **the wrong thing**

Problem -4

With modern programming methods and tools, it is quite reasonable nowadays to expect good compliance with specifications.

So, assuming that programming is perfect, the problem of obtaining specifications stating exactly what the client wants remains.

Problem -5

Thus the major problem in the production of software meeting the client's requirements is in obtaining mutually satisfying specifications,

which specify exactly what the client wants,

which, perhaps, anticipates future needs,

from which the programmer may write the required software.

Problem -6

We know that the source of more than half of the errors found during testing, acceptance testing, and production use is poor specifications.

In these slides, "specifications" means "requirements specifications" and is used interchangeably with "requirements" even though specifications are normally only a written manifestation of the requirements.

Problem -7

The difficulties preventing sufficient mutual understanding to arrive at suitable specifications are that

the programmer knows little or nothing about the client's domain,

Problem -8

the client knows little or nothing about what is possible with a computer,

or worse than that, the client believes that experience with a home computer makes an expert!

A big ga p here!

Problem -9

We believe that it is the job of the programmer, as the professional in the situation to bridge the gap to the client.

It is up to the programmer to get the client to teach him or her about the client's domain.

It is up to the programmer to teach the client enough about computing to make intelligent choices.

Problem -10

If the client cannot do an adequate teaching or learning job, it is up to the programmer to teach the client how to teach or learn.

Finally, it is up to the programmer to be able to learn what the client is teaching.

Background -1

This talk describes an experience by the two authors as programmer and client, for the purpose of producing specifications for a statistical experiment simulation program.

Background -2

The first author was, and still is, a computer scientist and played the programmer in the experience.

The second author, at the time, was a statistician, and played the client in the experience.

Background -3

The first author knows very little about probability and statistics and, in fact, had developed a phobia for anything remotely smacking of the subjects.

The second author, at the time, knew very little about computing beyond the use of BMDP and the barest rudiments of FORTRAN *coding* (as opposed to programming).

Background -4

The second author has since then seen the light and is now a computer scientist!

Background -5

The native languages of the authors are different.

Finally, the authors were, and still are, married to each other!

Thus, the potential for misunderstanding is clear.

Background -6

At the time, the second author was an M.A. candidate at Tel Aviv University in statistics.

Her thesis research was on the validity of conclusions drawn from experiments involving a sequence of observations in which the data become unavailable from some point on (truncation) or in which some but not all data are missing (censored).

Background -7

She had to write a program that permitted numerous simulations of experiments with large observation vectors.

An experiment was to generate a pair of vectors of observations, then to both

- **truncate one of the pair at some random point, and**
- **censor one of the pair at some random points.**

Background -8

Finally, she was to compare conclusions drawn from the full vector to those drawn from the truncated and censored vectors.

Drawing conclusions and comparing them involved calculating some very well-known statistical measures (well-known to statisticians!).

Difficulties -1

All had to be finished in 10 weeks (extended by one more week at the end):

- **writing the program**
- **running several experiments**
- **documenting the program**
- **writing the thesis**
- **defending the thesis**
- **correcting the program or the thesis based on the committee's recommendations (i.e., requirements)**

Difficulties -2

Why the short fuse? The advisor was leaving the country for the summer.

At the time, second author did not know how to program, but the advisor, the author of the BMDP statistical package, does.

The three-person committee could, in principle, require new calculations at any time, even at the defense.

Difficulties -3

There were certainly some committee members in the department who took particular delight in delaying a student by finding interesting new calculations to perform.

Difficulties -4

Thus, it was clear to the first author, the experienced computer scientist, that the program had to be right the first time and had to do any calculations that even the most diabolical committee member could ever, ever think of.

There was simply no time to write the program a second time or even to modify it.

Difficulties -5

The first author was personally familiar with patchwork programs that never quite worked right that resulted from thinking of new features as the program was being written.

He was familiar with the agony of having to revise almost every line of a program to accommodate the ever so small new feature that a client asked for late in the development.

Difficulties -6

He was painfully familiar with interface problems discovered late in a program's development.

These all had to be avoided at all costs!!! (Well... not *all* costs!)

Difficulties -7

In consideration of the second author's inexperience with computing, the advisor agreed to allow the first author to help the second author write the specifications for the program.

Of course, the second author had then to write the program herself.

Form of specification -1

The authors and the advisor agreed that the specification would consist of two main parts:

1. **list of inputs in the form of a typed, constrained variable declaration for each input (constraints specify bounds)**

Form of specification -2

2. **given input meeting the types and constraints of part 1, a list of all desired output, i.e., values and tables**

No details about how the calculations were to be done were to be specified!

Actually, the phobia of the programmer would make sure of that!

Initial specification -1

The programmer asked the client to write a first draft specification

This is what he got!

Initial specification -2

Input:

- ***n*** **(sample size) int**
- θ **(parameter of exponential distribution) real**
- ***p*** **(percentage of data to be lost) real**
- ***t*** **(truncation of the survey) real**

Initial specification -3

Output:

1. ***observations*** **2 vectors of size *n*, each obs is a real no.**
2. **new observation vectors without the lost data (no info as to which positions lost data)**
3. **the 2 vectors of truncated data**

Initial specification -4

For each of 1, 2, 3:

4. $p.25$ $p.50$ $p.75$

5. $p(t=\frac{1}{2})$, $p(t=1)$, $p(t=2)$

6. **table of nonparametric test each one of the elements of (4) and (5) in the output paragraph**

Initial specification -5

	below the stats	above the stats	tot tot
vect1	$n1b$	$n1a$	$n1$
vect2	$n2b$	$n2a$	$n2$
tot	$n1b+n2b=nb$	$n1a+n2a=na$	n

and the hypergeometric prob. $p\{n1a\}$

Initial specification -6

7.
   a. **limits of the confidence intervals between the statistics in 4 and 5 in the output paragraph** $[l1, l2]$ **in 95% confidence level**
   b. **for each appropriate pair of the above stat, if it fell into its c.i. [confidence interval] or not**

Initial specification -7

Clear, huh?

The programmer winced hard and knew that he had his work cut out for him!

Floundering -1

In an attempt to understand these specifications in order to refine them into something useable, the programmer asked the client what in heaven's name was going on here!

The client began to talk.

Floundering -2

The programmer heard lots and lots of statistical buzzwords:

take an observation vector
truncate it
censor it
concatenate two observation vectors
time *t* of a vector at probability *p*

Floundering -3

probability *p* of a vector at time *t*

take the delta sum of a vector

the standard error of a vector at probability *p*

the standard error of a vector at time *t*

over and over again.

Floundering -4

They were meaningless to the programmer,

but

they were clearly meaningful to the client or anyone in the statistics field.

Inspiration -1

"*ah hah!*" said the programmer.

We have an abstract data type,

observation vector

or

vector

for short,

in which each observation is a real number and whose operations include

Inspiration -2

create
truncate
censor
concatenate
time
probability
delta sum
standard error.

Inspiration -3

By nudging the client for more operations on observation vectors that will be needed, the programmer was able to produce a preliminary list of the operations of the ADT vector.

Inspiration -4

The programmer then asked the client to identify for each operation

- **the list of parameters and their types**
- **the type of the return value, if any**

Inspiration -5

The programmer nudged the client for assurance that

- **each operation is well understood by statisticians**
- **each operation is either**
  - **trivially implemented, or**
  - **defined by a formula well-known to statisticians**

Inspiration -6

With these assurances, the programmer felt confident that the ADT was implementable.

Refinement -1

Now was the time to begin refining the specifications into a useable document.

The programmer decided to produce a description of the main program using the ADT and its operations as primitive.

Refinement -2

Again, the specifications would show only the input and the output and not show how they or the ADT would be implemented.

So the programmer and client began a refinement cycle.

Refinement -3

As the programmer and the client worked together to make the specifications more precise, complete, etc., the programmer knew that they would discover

- **additional needed operations for the ADT and**
- **that parameters, parameter types, and return types of existing operations would be changed.**

Refinement -4

Each such change would require another iteration on the specifications with all of the above and below mentioned nudging.

Refinement -5

Discovering a new operation for the ADT caused

- **adding a description of the operation to the ADT description,**
- **nudging again for assurances of implementability, and**
- **rewriting the specifications to use the new ADT.**

Refinement -6

Occasionally the programmer discovered the client using the same operation in more than one way!

How could he do that even though he did not understand the operations?

The client used the same verb with a different set or a different number of noun phrases, i.e., a type inconsistency.

Refinement -7

For example,

- **some applications of either empirical standard error operation had a size parameter of type int and others did not,**
- **some applications of either theoretical standard error operation had a θ parameter and others did not.**

Refinement -8

When programmer caught these, he nudged the client for a resolution.

Maybe each use was really of a different operation.

Maybe the earlier uses were wrong.

Maybe the new use is wrong.

Refinement -9

Maybe both are right, and the operation should be overloaded.

Maybe both are right, and the operation takes a variable number of parameters, with missing ones being set to default values.

Refinement -10

In these cases,

size was recognized as independent of the vector (surprise!) and therefore a necessary parameter of the operation so that all applications were changed to provide size.

θ was recognized as calculable from other data and therefore unnecessary; so, the θ parameter was removed, and applications were changed to provide no θ.

Refinement -11

Once a resolution was achieved, the ADT was updated to reflect the changes, and the client was nudged for assurance of well-definedness.

The specification of the program was rewritten to reflect the new ADT.

Refinement -12

There were a number of questions and modifications that the programmer thought of from his experience.

Spelled out identifiers would be clearer.

Program lacked generality—it was locked to one run, with fixed uniform sized vectors, with the same θ, loss rate, and truncation threshold in each case.

Refinement -13

There were magic constants, e.g., the 95% confidence level; why?

All of these fixed values were made input parameters of the program.

The programmer's debugging sense warned him to have the program print out a lot of additional information that would help trace computations and debug the program.

Refinement -14

It turned out that all of the wouldn't-it-be-nice-if-you-also-calculated-xxx information requested by the advisor and other members of the committee were covered by this debugging output!!!

The Specifications -1

First we give the abstract data type:

The Specifications -2

Then we give the actual input-output specifications:

Results -1

The specifications were completed and approved by programmer, client, and advisor after about 4 weeks of spirited discussion.

The client turned into the FORTRAN programmer of the specifications.

She implemented the ADT as a collection of FORTRAN subroutines and functions that shared access to a named common area that the main program did not see.

Results -2

The program ran after very few debugging runs.

The sources of the bugs were very easy to locate because of the extra debugging output the first author had insisted on.

Results -3

The second author wrote the thesis, defended it, and filed it one day before the advisor left the country.

Also, during the writing of the thesis, the second author delivered a baby!

Amazing Occurrence -1

The authors were prepared for the usual feedback on the specifications that normally occurs as the specified program is being written.

They were prepared to make all changes necessary to the requirements document to keep it consistent and up-to-date with the program.

It had to be ready to stick into the thesis on a moment's notice.

Amazing Occurrence -2

But, once the specifications were accepted and typed, *they were not changed at all* except to correct minor typographical errors.

There was none of the usual feedback from the designing, coding, testing, debugging, and running of the program onto the specifications.

***No* inconsistencies were discovered.**

***No* new functions were discovered.**

Amazing Occurrence -3

This is the first time in either author's experience that a requirements document had not undergone change as the program was being implemented and remained faithful to that program as implemented.

This was also the first time the first author had ever participated in the development of a program about which he had no real understanding (and he still does not).

Amazing Occurrence -4

The fact that the requirements document did not have to be changed at all is even more surprising!

Reasons for Amazing Event -1

So what were the factors that allowed this amazing event to happen?

More importantly what are the techniques that can be applied to increase the chances of duplicating this event with other specifications?

Beginners' luck???

Reasons for Amazing Event -2

Introspection showed three main techniques that were applied beyond normal common sense:

- **abstract data typing**
- **strong typing**
- **Jewish motherhood**

Abstract Data Typing -1

You are already very familiar with the concept.

Here, the encapsulation that is normally used to hide implementation details (so that they can be changed) was used to

hide ignorance

of domain concepts about which the programmer had not even the foggiest notion and to allow him to work with them in a consistent manner.

Abstract Data Typing -2

All he had to do was be certain that the concepts were well enough understood that they could be implemented, and implemented straightforwardly.

Then he could fake his way through the problem so well that he could find inconsistencies in what the client was saying.

Strong Typing -1

Strong typing is that property of a programming language that assures that the types of all (sub)expressions can be computed at compile time.

Strong Typing -2

The type of a constant is immediate:

1	**int**
3.14	**real**
true	**bool**
'x'	**char**

Strong Typing -3

The type of a variable is declared:

```
int i,j,k
real x,y
bool b
char c
```

Strong Typing -4

The required types of language constructs is known

OK **if true then .... else .... fi**

OK **if b then ... else .... fi**

BAD **if x then ... else .... fi**

Strong Typing -5

The type of a procedure is the list of types of its parameters and the type of its returned value, if any:

proc(int)int factorial
proc(real)char real_to_char
proc(int,int)bool equal
proc(char)void print_char

Strong Typing -6

It is possible to check applications for type consistency without knowing the body *or meaning* of procedure!

OK **i:=factorial(j)**
" **c:=real_to_char(x)**
" **if equal(i,j) then ...**
" **print_char(c)**

BAD **c:=factorial(j)**
" **i:=factorial(c)**
" **i:=factorial(i,j)**

Strong Typing -7

In the case of this experience, strong typing was sufficient to allow the programmer to find *all* inconsistencies in what the client was saying and in non-final drafts of the specification.

It helped the programmer fake understanding of the operations by giving him a tool to at least be able to tell when an operation was being used correctly or consistently with the other uses.

Strong Typing -8

Is this surprising?

Not really!

It is well known how effective strong typing is at identifying program errors prior to run time and in preventing these errors from ever occurring at run time.

There have been numerous studies.

Redundancy!

Strong Typing -9

Some languages that used not to be strongly typed are now, e.g.,

C $\rightarrow$ C++

LISP $\rightarrow$ Common LISP

Even before C++, people used lint to approximate strong typing.

Jewish Motherhood -1

Jewish motherhood is the ability to keep nudging the client so that he or she feels guilty when he or she has failed to tell you something!

It is also the ability to detect when client is not telling you something important.

Jewish Motherhood -2

***How to be a Jewish Mother* by Dan Greenburg**

explains how to do it and makes the point that you have neither to be Jewish nor a mother to be a Jewish mother.

An irish waitress and an italian barber can be Jewish mothers.

(Your nationality) programmers can (and must be) Jewish mothers.

Jewish Motherhood -3

You have to be able to nudge the client to make sure that you have captured what he or she wants.

Are you sure that this is what you want?

Are you certain that this is what you want?

Are you positive that this is what you want?

Are you sure that you are certain that this is what you want?

Jewish Motherhood -4

Are you sure that you are positive that this is what you want?

Are you certain that you are sure that this is what you want?

Are you certain that you are positive that this is what you want?

Jewish Motherhood -5

Are you positive that you are sure that this is what you want?

Are you positive that you are certain that this is what you want?

... (ad infinitum et nauseum)

You have to be able to inculcate guilt and detect when the client is waffling

Jewish Motherhood -6

One caveat!

There is a danger of the client learning how to be a Jewish child, i.e., to be immune to the guilt effects.

So you have to learn when to let up.

Specification Languages -1

The experience suggests some important properties for a specification language, that

- **it permit natural language phrases,**
- **it permit the definition of abstract data types, and**
- **it have a processor which does full type checking.**

Specification Languages -2

Given these requirements for a specification language, it is clear that we are talking about a strongly typed program design language with some ADT definition construct.

Specification Languages -3

The paper describes an attempt by the authors to use Ada as a such a program design language in which to express the specification.

Specification Languages -4

The ADT vector was expressed as a package observation_vector defining the type vector.


```
with TEXT_IO; use TEXT_IO;
package OBSERVATION_VECTOR is
    type VECTOR is private;
    function create_observations(size:INTEGER;theta:FLOAT)return VECTOR;
    function create_kept_observations(v:VECTOR;loss_rate:FLOAT
        name:STRING)return VECTOR;
    function create_truncated_observations(v:VECTOR;time:FLOAT
        name:STRING)return VECTOR;
    function concat(v1,v2:VECTOR)return VECTOR;
    function size(v:VECTOR)return INTEGER;
    function time(v:VECTOR;prob:FLOAT)return FLOAT;
    function probability(v:VECTOR;time:FLOAT)return FLOAT;
    function delta_sum(v:VECTOR)return INTEGER;
    function time_theoretical_std_error(size:INTEGER;prob:FLOAT)
        return FLOAT;
    function time_empirical_std_error(size:INTEGER;v:VECTOR;
        prob:FLOAT)return FLOAT;
    function prob_theoretical_std_error(size:INTEGER;time:FLOAT)
        return FLOAT;
    function prob_empirical_std_error(size:INTEGER;v:VECTOR;
        time:FLOAT)return FLOAT;
    function above(v:VECTOR;time:FLOAT)return INTEGER;
    function below(v:VECTOR;time:FLOAT)return INTEGER;
    procedure print_name(v:VECTOR);
    function difference_std_error(val1,val2:FLOAT)return FLOAT;
    function theta(v:VECTOR)return FLOAT;
end OBSERVATION_VECTOR;
```


Specification Languages -5

The specification itself was given in the form of a main program that withed the package.

The purpose of this exercise was to trick the Ada compiler to do the type checking that we wanted.

Specification Languages -6

The program consists only of declarations of input variables to establish their types and a bunch of applications of put (overloaded printing routine that accepts any printable value as its in parameter) to the results of applications of package-defined functions to these input variables and to for-loop index variables.

Specification Languages -7

Things like

```
for each time ∈ {$1 over 2$,1,2}
    ...
od
```

in specification are expressed as

Specification Languages -8

```
declare t:float;
    time_set constant array(1..3) of float
        := (1.0/2.0,1.0,2.0);
begin for i in time_set'range loop
        i:=time_set(i);
    ...
end loop; end;
```

This way, in the body, the uses of t will be properly type checked.

Specification Languages -9

What happened when we ran the Ada version of the specification through an Ada compiler?

We found one error that was actually a typo in the published specification.

Specification Languages -10

The typo was a missing string parameter in applications of the operations create_truncated_observations and create_censored_observations.

These errors are not in the FORTRAN program.

Specification Languages -11

So we were very lucky not to have had a serious error.

A serious error is one in which interface inconsistencies are not noticed until integration testing.

In a larger problem, it would have been even luckier not to have many more serious errors.

Specification Languages -12

A type checking processor for a program design language would certainly help to eliminate these errors.

Such errors are, in fact, *stupid errors*, errors that are algorithmically avoidable.

Specification Languages -13

Ada is not really a good program design language because it does not really allow arbitrary natural language text.

Only one program design language we know of that meets all of our requirements for a specification language.

Specification Languages -14

Ada-SDP, from Mayda Ltd. in Israel.

It recognizes Ada keywords.

Any string of text between keywords is an application of a procedure which has parameter words.

Specification Languages -15

If you

declare push $value into $stack

and there are types value and stack,

and you write

push plate into plate_stack

the Ada-SDP processor recognizes this as application of

push $value into $stack

Specification Languages -16

If, in addition, plate and/or plate_stack have been declared, the processor checks that they are declared of types value and stack respectively.

Unfortunately, the company no longer exists, so we will have to be very careful and do things by hand.

Conclusion

We consider

- **the reasons for success**
- **what needs to be done in the future**

Reasons for Success -1

It is always nice to believe that a method that worked on one problem will work on all others

However, the reality is that each problem seems to beget or require its own method or its own variant of a method

Reasons for Success -2

There are a number of reasons that the above described method worked in this case and one cannot expect that the success will generalize:

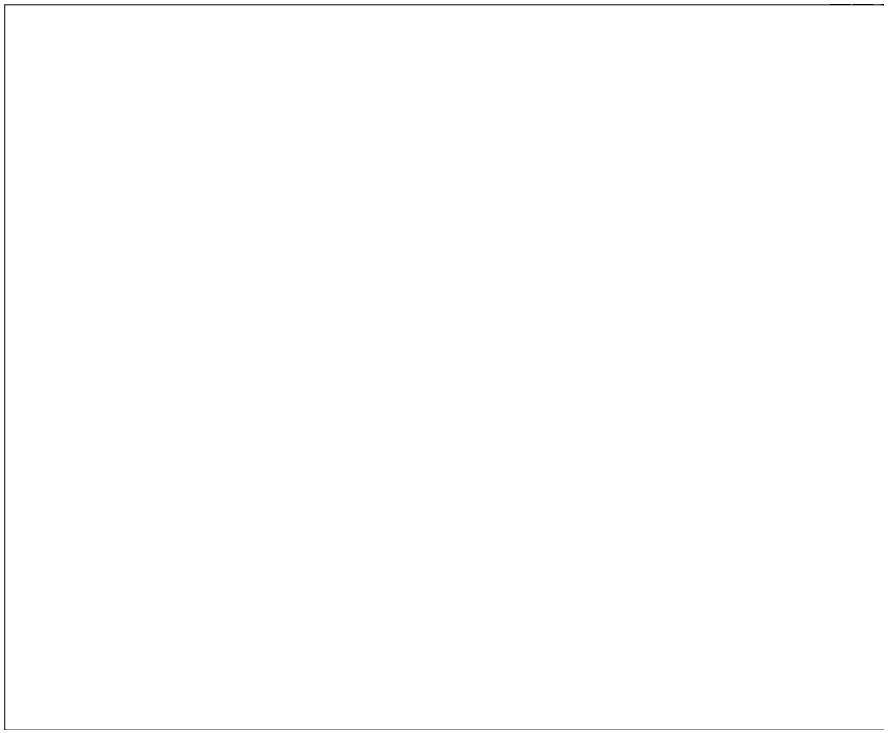
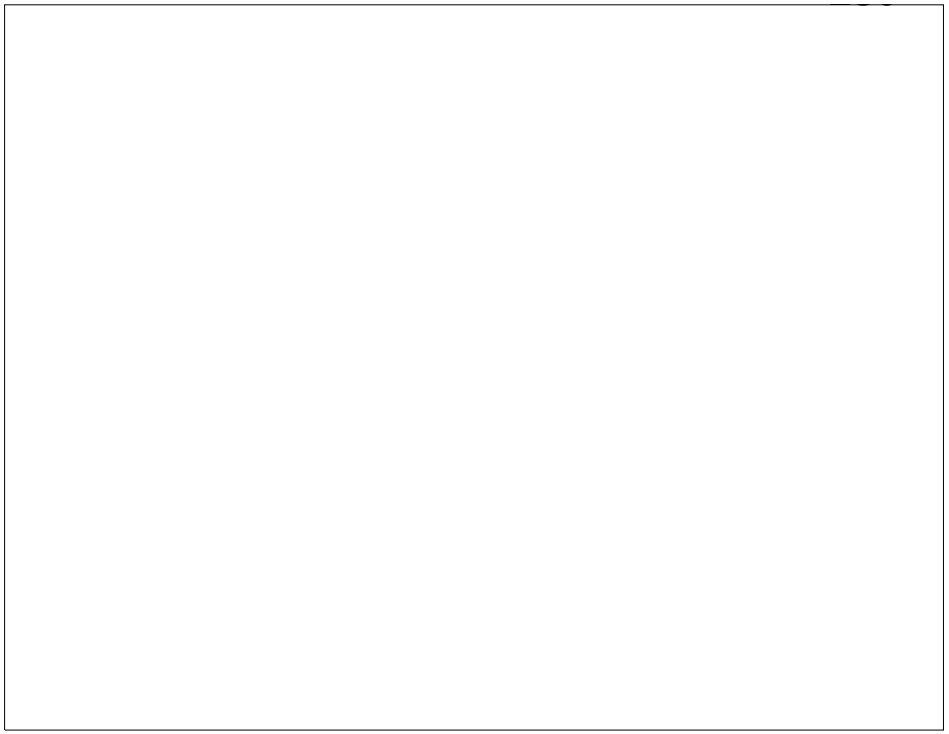
- **beginner's luck**
- **the small size of the problem**
- **the fact that there was only one client representative**

Reasons for Success -3

- **the fact that there was only one programmer or requirements analyst**
- **the very fact, as suggested by Alstad, that the programmer knew so little about the domain and therefore did not fall into any tacit assumption tarpit**
- **the fact that the problems had such a strong mathematical basis and the only real problem was organizing the output**

For the Future

We need to try the method out on

- **larger problems**
- **with groups of client representatives**
- **with groups of requirements analysts**
- **with harder non-formalizable problems**

to see how generally applicable the method is.