A natural measure for denoting software system complexity

Jacques PRINTZ, Professor Emeritus
Chair of Software Engineering
Conservatoire National des Arts et Métiers, 2 Rue Conté
PARIS Cedex 75141 – France
Mail : jacques.printz@cnam.fr

Summary

INTRODUCTION 1
CYCLOMATIC NUMBER MEASUREMENT CONSIDERED HARMFUL 4
PROGRAM TEXT LENGTH MEASUREMENT 6
PROGRAMMERS ACTIVITY REVISITED 7
CLASSIFICATION OF BUILDING BLOCKS 9
COSTS OF INTERFACES 11
THE COMPLEXITY OF INTEGRATION 12
INTEROPERABILITY 16
TEMPORARY CONCLUSION 17
APPENDIX 19
REFERENCES 20

Figures

Figure 1 – PIP Programmer at work 3
Figure 2 – Building-block programming 4
Figure 3 – Sequential flow of control 4
Figure 4 – Parallel flow of control 5
Figure 5 – Invisible edges with event programming 5
Figure 6 – Program and test working together 8
Figure 7 – Overall test development process 9
Figure 8 – Minimisation of edges 10
Figure 9 – Cost of interfaces 12
Figure 10 – Integration tree layout 13
Figure 11 – Degenerated integration tree 13
Figure 12 – Nominal structure of a building block 14
Figure 13 – Building blocks composition 15
Figure 14 – Shared data and performance 16
Figure 15 – Interoperability, from business architecture to functional architecture 16

Introduction

The problem of complexity measurement is as old as programming, when programming became a major problem for the software industry in the sixties. The fact is clearly attested in the two NATO reports on software engineering [A14]. Von Neumann himself give a lot of attention to complexity in the last decade of his shortened lifetime.
The problem of measure is a very classical one in physical science and everybody aware of history of science (epistemology) knows that it is an extremely difficult subject. With information measurement we enter in a kind of no man's land with pitfalls everywhere where it is difficult to separate the observer (the user of the information) and the phenomenon of information itself.
In this communication I propose to define complexity measurement as follows:

Program A or software system A is more complex than program B or software system B if the cost of development/maintenance of A is greater than the cost of B.

This measure is just common sense based on the fact that for a project manager cost (that means a compound of human cost, development time and expected quality) is the only thing worthwhile of his attention.

To define the complexity measure more precisely, we need to define:

- what is "cost",
- what is "development".

NB: To simplify, we will use the term complexity in a broad sense, and we will make no difference between intrinsic complexity of the problem and the complexity (i.e. complication) of the solution which aggregates technologies and project organization. For in depth discussion, see ref. [A8].

Since the very beginning of software engineering the software cost has been associated with programming effort, generally expressed in man-month or man-day, while development was defined as the sum of activities such as: program definition / requirements (with customers and end-users), software design, programming (development / maintenance) + unit testing, integration [see ref. A14, A20].

Programming effort is the amount of programmer energy needed to deliver a program or a software system which satisfies the level of quality required by the target organization which will use it.

The term "programmer energy" refer to the human part of the programming activity which is basically a translation, in a linguistic sense, of customer requirements into an executable program on a specific computer platform. We know that programmer performance may vary in a large range from 1 to 15, as reported in the literature (NB: I can testimony of variation from 1 to 10). So we need to define more formally what do we mean by programmer performance.

In one of my books [A1], I have defined what I have called the "Perfect Isolated Programmer", or PIP. A PIP is a kind of human robot, or a Maxwell's daemon, with well formalized capacities (in the same sense as we define "perfect gas" in thermodynamics), able to perform the translation SYSTEM REQUIREMENTS → WORKING SOFTWARE. By definition a PIP is not submit to environment disturbance and human impedimenta. PIP effort is expressed in time unit (work-hour) which is similar to an "energy". If several PIP act together to create a program, one has to take in account the effort needed for the communications between them in order to maintain coherence, for example the effort needed to establish contracts between them to define the interfaces and to verify that they respect the contracts.

The PIP programmer effort will be split in two parts:

- A part for writing source code and test code to validate the program, i.e. ***the programming effort*** as understood by X-programming or agile community which promote the Test-Driven-Development approach.
- Another part for writing documentation to communicate, either face to face or group discussions, or Internet mailing, etc., i.e. ***the communication effort***.

PIP programmer's activity may be shown as depicted in figure 1. It creates three types of text : programs, tests and documentation.

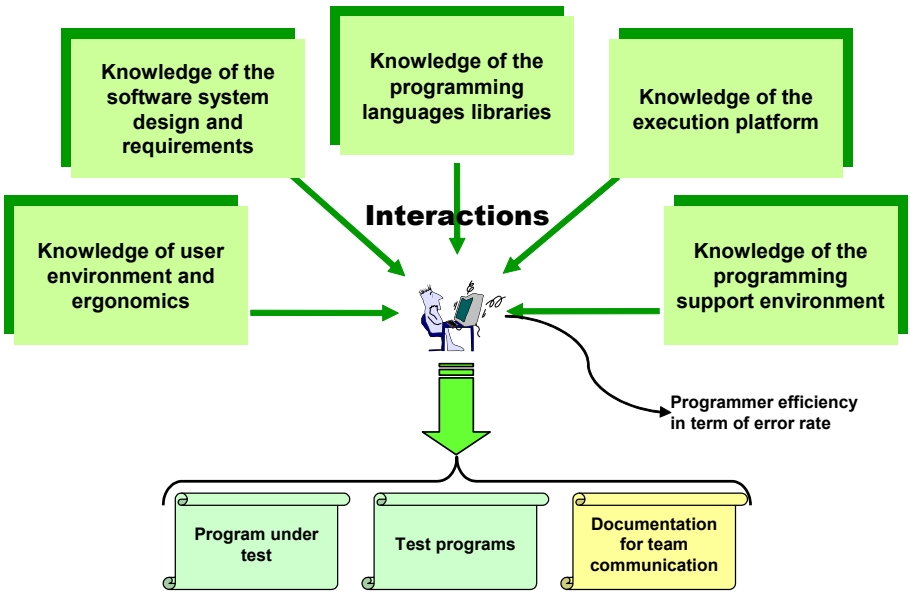

Figure 1 – PIP Programmer at work

When we observe real programmers productivity at large scale, one states a remarkable stability of around 4.000 instructions per man-year (with a year of about 220 working days of 8 hours), or around 20 source line instructions per normal working day. Of course, instructions counting must be done with extreme care as defined in [A2]. When we observe the phenomenon at small scale with toy programs, it simply disappears. Instructions counting is the foundation of widely used cost estimation models such as COCOMO or Function Point. So programmer productivity is a statistical measure as exactly is the temperature. At atomic level, temperature is meaningless, because the measurement apparatus itself disturbs too greatly the temperature measurement (Cf. Heisenberg uncertainty relations).

Programs are developed chunk by chunk by individual programmers or small groups of 2-3 programmers which are members of a programming team of around 7±2 developers and a team leader. For large or very large projects, up to 50-60 teams may be active at the same time. In that case, team organization become the main risk due to the interactions between teams; see [A18]. In large projects, a significant amount of effort must be dedicated to communicate, establish and validate the interface contracts.

Contract information may use constraint languages such as OCL, now part of UML, or more specialized languages like PSL (IEEE Standard N°1850).

Returning to the definition we will now examine some situations to validate the size of tests as a good candidate for an effective measure of complexity.

Programming activity in well managed projects may be depicted as in figure 2. Teams are organized to product elementary pieces of code (called building blocks, or BB) ready to integrate.

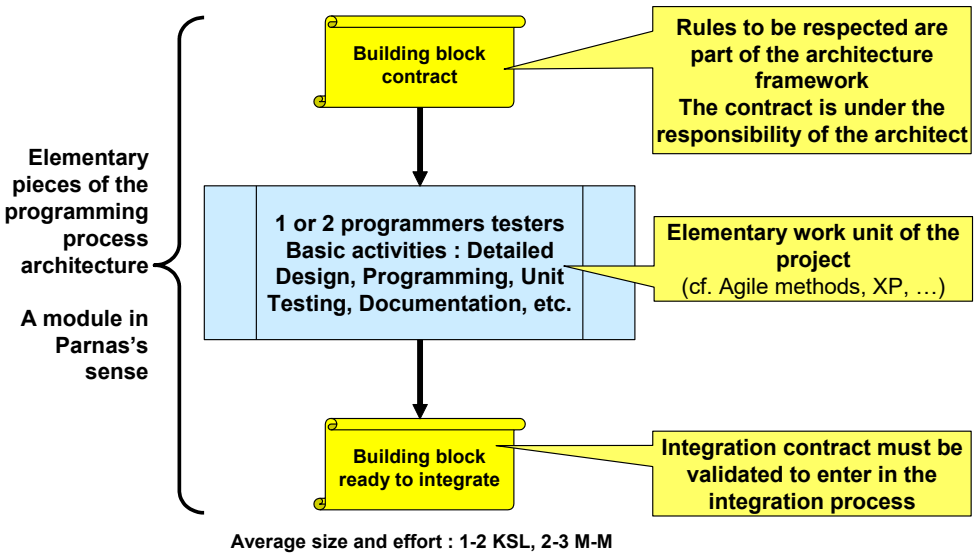

Figure 2 – Building-block programming

Cyclomatic number measurement considered harmful

Cyclomatic number, also called Mac Cabe complexity measure [A3], is one of the most popular complexity measure implemented in most CASE/QUALITY products to "measure" quality, but programming quality, i.e. semantic aspects, measurement is another story.

However, when we consider the Mc Cabe measure from the project manager point of view, it has really no meaning. Of course everybody agree that from graph theory [A4] point of view, cyclomatic number means something important, but the problem is what does it mean for the programming effort point of view ?

With very simple examples, which are true counter-examples (in the meaning of K.Popper "falsification" ; see ref. [A5]), one can show easily why cyclomatic number complexity has no meaning.

The following picture shows the most simple design pattern with sequential execution of *N* modules, M1, M2, …, M-*N* executed sequentially. Think to a man-machine interface (MMI) where screens are captured in a fixed order.

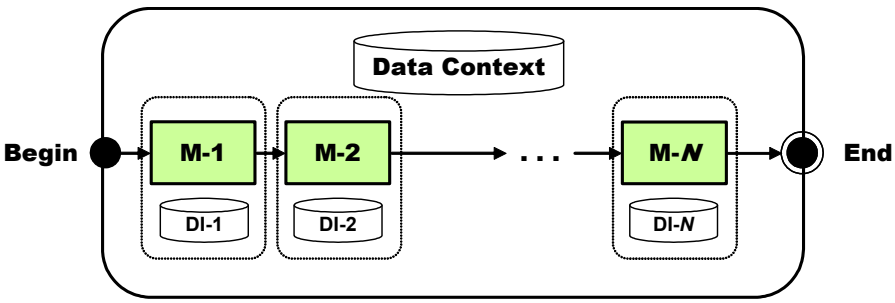

Figure 3 – Sequential flow of control

The cyclomatic number of the pattern is 1.

The next picture shows another pattern where the end-user can filled the screens in any order. Basically the tests are the same.

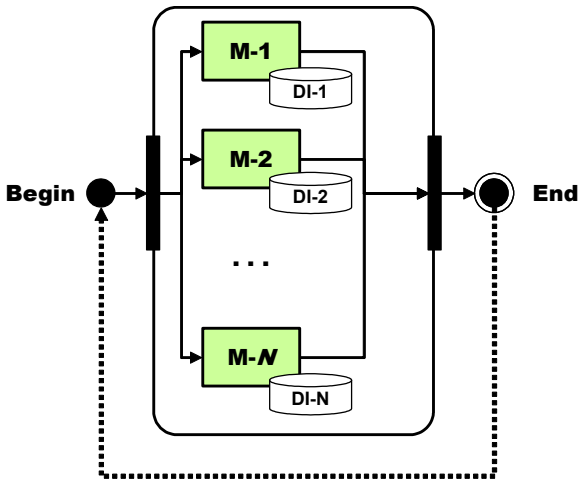

Figure 4 – Parallel flow of control

In that case the cyclomatic number is *N*.
The difference with the previous one is the loop from END to BEGIN, a very few instructions indeed. But the tests to validate the two patterns are almost the same and they have basically the same cost.
Moreover, in figure 3, the cyclomatic number give an appearance of simplicity despite the fact that the programmers may have created a large data context to exchange information between modules. On the contrary, in figure 4, the programmers have organized the data in order to use a transaction programming style. Data are well isolated each others; the modules work as ACID transactions. Despite the fact that the cyclomatic number *N* is much greater, the software architecture is much better.
Thus the cyclomatic number alone gives a "complexity" measure totally inappropriate and deceptive. In order to take in account the real complexity based on the number of tests, the programmer has to consider the source_code×data matrix (cross references) which depends mainly on the data architecture which is basically the same in both cases.
Another misleading situation comes from the event programming style allowed now by languages like Java, a risky style for the beginners.
Figure 5 illustrates this situation.

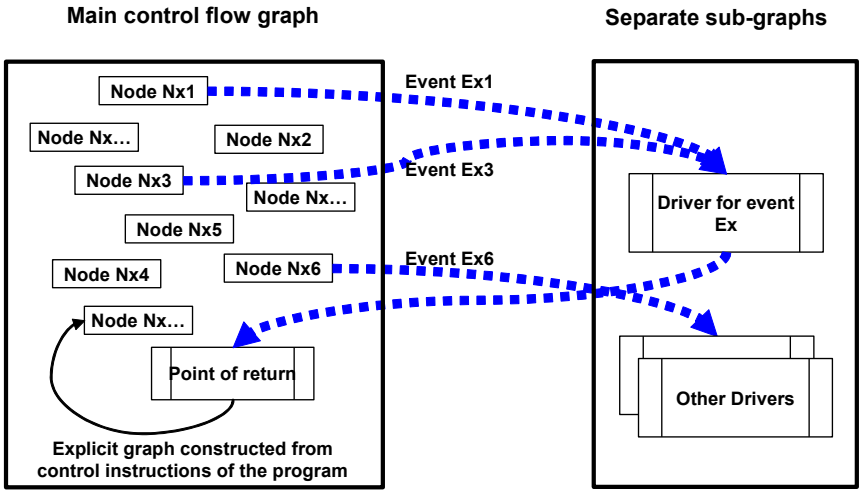

Figure 5 – Invisible edges with event programming

The doted lines represent invisible edges, which will not be caught neither by quality tools, nor by compilers made unable to optimize the generated code. Any sequence of code able to raise the event *Ex* will create an additional edge. The computed cyclomatic number will be much smaller than in the real execution. It's like having invisible GOTO statements hidden in the source code.
Thus, again, the cyclomatic number gives a "complexity" measure totally inappropriate and deceptive. Worse, it may encourage event programming style which is the most difficult to master and needs a lot of experience in real-time and system programming. You will notice that the number of lines of code, in that case, will probably decrease, but the number of tests will increase dramatically.

Program text length measurement

In the early age of programming, in the sixties, managers in charge of large programming teams stated another remarkable fact which is a relation between the length/size of a program expressed in number of kilo instructions (KSL/ kilo source line) effectively written by the programmers [see ref. A2, A20] and the effort, generally expressed in man-month, to delivered them to a customer. Numerous statistical studies [see A6 for details] has shown that the relation, most of the time, had the form of a power law:

$$Effort = k(KSL)^{1+\alpha}$$

where α is a positive number >0, in fact in a range between [0.01 to 0.25] ; k is a constant which depends on the programmer capability and experience, and also on the overall project environment such as documentation. The effect of k on the effort is linear, and for this reason k is called the ***cost factor***.
The most interesting parameter in this experimental relation is the number α. But what α exactly means? What are the conditions which allow to say that α has such or such value? For example the classification of program types [such as: a) simple programs as in business applications, b) algorithmic driven such as compiler, data mining or complex data transformations, c) real time and events programming ; see ref. A6] plays an important role, but also team maturity in the sense of capability to work together as explained in the CMMI approach. The number α says something about complexity of the interactions between modules within the software system, as well as interactions between human actors, at individual level or at team level, within the development organization. For this reason α is called the ***scale factor***.
In my book [A1], I have demonstrated that α largely depends of the hierarchical structure of the interactions. When the system architect and the programmers strictly apply the Parnas's modularity principles [A7], then the whole program organization is hierarchical, and the *Effort*/*KSL* relation holds. It shows the importance of a rigorous classification of functional blocks at design time. The PIP programmer of the COCOMO estimation model is supposed to master the Parnas's modularity principles.
With regards to our definition, a system software program which is composed from the integration of N modules M1, M2, ..., Mn with lengths of KSL_1, KSL_2, ..., KSL_n, that is to say of a total length of $KSL_1 + KSL_2 + ... + KSL_n$ is more costly than the simple sum of each module taken independently.
The difference is given by:

$$D = k(KSL_1 + KSL_2 + ...KSL_n)^{1+\alpha} - k\left[(KSL_1)^{1+\alpha} + (KSL_2)^{1+\alpha} + ... + (KSL_n)^{1+\alpha}\right]$$

The question is: what is the meaning of D?

If our initial relation is true (i.e. experimentally verified), then *D* denotes a) the effort to break the whole software into *N* modules, plus b) the effort to establish interface contracts between the modules, and plus c) the effort to integrate the *N* modules, that is to say to verify that the contracts have been enforced. In other word, that means that the number α says something about the complexity of integration whose cost *D* depends on the number of interactions between modules. Thus, counting the relations between modules is a key element of the measurement process. For more details see ref. [A1, A18].

Now we have to examine the question of the legitimacy to use program length to denote program complexity.

In the algorithmic information theory established by A.Kolmogorov and G.Chaitin [A12], the complexity of a problem is expressed by the length of the "program" to solve it. So, in this theory we have a clear indication that the textual length is well founded and means something important.

We have not enough place in this communication to establish a more abstract link between the effort relation and his mathematical form as a power law. Power laws are very common in nature, especially in social sciences, as it has been established by several authors [A9]. It is not surprising to find them in this highly human activity that is programming [A10]. For a more detailed discussion, see [A8, chapter 16].

Despite the well founded legitimacy of program length as a measurement of complexity, something important is still missing from the project manager point of view.

We know since the very beginning of programming that a program has always two faces:

1. a statically one which is effectively well denoted by the program length and structure,
2. a dynamically one which is, for the moment, denoted by nothing, or in a weak sense by the scale factor α which take in account the combinatory aspect of the integration tree.

To go further, we need to revisit briefly the nature of programming activity.

Programmers activity revisited

Every programmer knows that programming effort cover two types of activities:

- the first one, and most well known, is the writing of programs in one or several programming languages,
- the second one is the testing activity, in his classical sense of validation and verification. Testing has been for a long time the hidden face, for not to say the shaming face of programming.

With eXtreme programming and agile development methods emphasis has been put on the so called "test driven development" approach. K.Beck [A13] has strong words : « *you don't get to choose whether or not you will write tests – if you don't, you aren't extreme: end of discussion* » ; this is an interesting position statement, quite new in the academic and industry software engineering landscape.

In the real world, materials have defects and can break (strength of materials is a pillar of engineering sciences) , electromagnetic waves are noisy, a circle is never perfect, and so on. In the real world programming, real programs, written by real programmers, contains also programming and design flaws, because programmers work with an error rate, as everybody. We have statistics about the number of residual defects during software operation.

So testing is now recognized as being as much important as programming, for not to say more important. From a logical point of view, the program affirms something about the

real world information processing that it models (a kind of theorem), but the tests establish the validity of the affirmation (a kind of proof). For a mathematician, a theorem without a proof has no value, and for him the demonstration is more important and often more interesting than the theorem itself. A good example is the recent demonstration by A.Wiles of Fermat's last theorem (more than 200 pages of demonstration for a "one line" theorem). In physic we have the same thing with physical laws and the experiments which justify them (think to the LHC equipment at CERN in Geneva to show the existence of Higg's boson for the gravitational force).
We will not enter in this paper into a philosophical discussion about the epistemological status of what we call the "proof" of the program. What is important here is to notice that beside the program text, it exists another text which establishes its validity, even if this validity is in essence statistical, as in reliability theory.

With regard testing, the programmer's activity may be described as in figure 6.

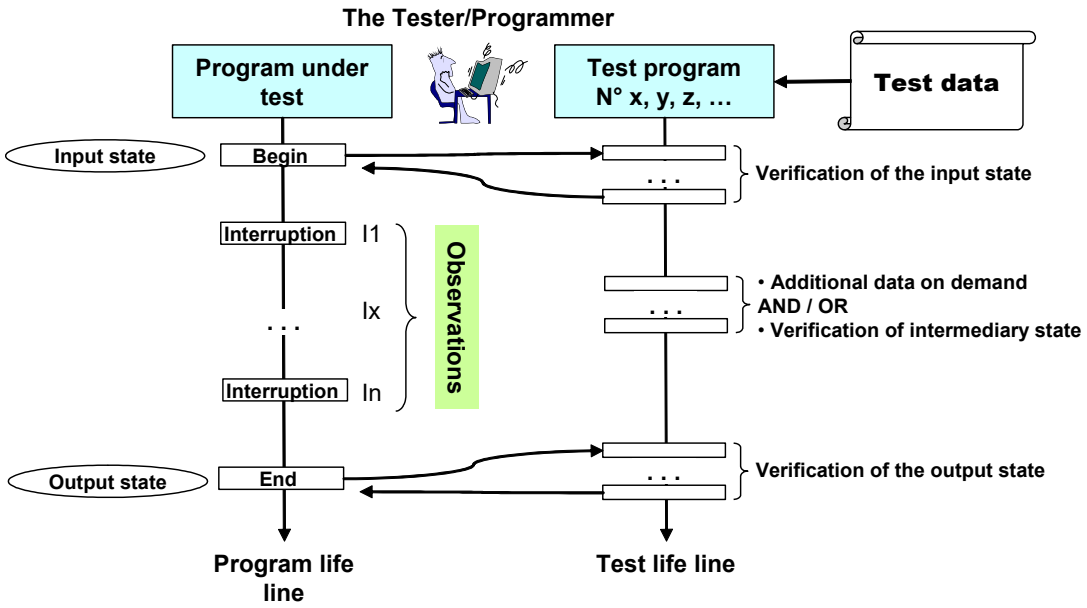

Figure 6 – Program and test working together

The program under test is written in some specialized languages such as debugger languages, or tests languages such as TTCN or PSL, or most of the time in the same language as the program under test. Test data must be carefully selected, according to the test objective. In fact, the test program and the corresponding test data are the specification of an experiment whose successful execution, under the control of the programmer/tester, will establish the validity of a part of the program under test. Data displayed from the program under test execution or accepted by the test program may be written in some universal data language such as ASN.1 or XML.
To validate the whole program under test, several test programs "experiment" must be written.
Different methods exist for designing "good" tests. Model based methods seem the most promising for integration; for more details see [A23, A24].

The overall test process can be depicted as in figure 7.

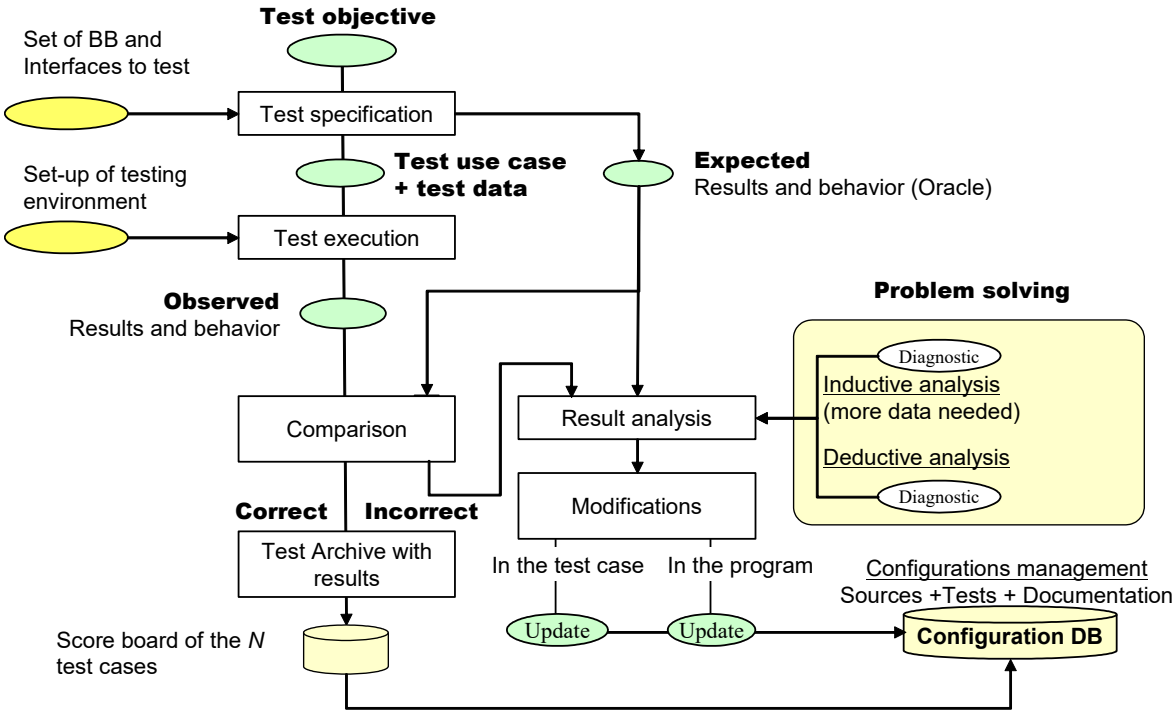

Figure 7 – Overall test development process

It is clear that the test programs and the activities round them reflect the dynamical aspect and the behaviour of the program under test, with enough data to explore the combinatory of the mapping between input and output states. Test data may be quite difficult to compute. For more details on test, see [A17].

Classification of building blocks

From the project manager and architect point of view, there are, roughly speaking, two types of building blocks (BB).

- First type BB are those that we can call Functional BB, or FBB, because they code the semantic part of the software system. FBB correspond exactly to what D.Parnas called module in his famous paper [A7]. FBB interact each others to form services which have a meaning from the business process point of view.
- Second type BB, or Service BB, represent more abstract functions which are use as an enhancement of an existing programming language. Early languages as FORTRAN and COBOL have extensively used that type of language extension with built-in functions like *SORT* or *SIN*, etc. Operating system functions or protocol functions are good examples of such extension. API (Application Programming Interface) play the same role. Today Java programmers may use, to enhanced their productivity, a large variety of SBB. An expert programmer knows hundredth of API. SBB are used as services or macro-instructions to code the FBB. The peculiarity of SBB is that they may be called from everywhere in the software system.

From the project manager point of view FBB and SBB have quite different status. By definition, SBB are abstract functions independent each others. The development cost of a SBB is totally independent of its usage, it is truly context-free. The cost to use a SBB is the same everywhere, either in a small program or in a large one. When a SBB is mature enough its usage cost from a programmer point of view is the same as any

available built-in functions of the programming language. Writing y=SIN(x) has the same programming cost that writing y=x+1 or y=SQRT(x).
Thus, complexity, from the project manager point of view, comes only from the interactions between FBB, and not from the interactions between FBB and SBB.
SBB are in fact a factor of simplicity despite the numerous references to them in the FBB. Again, this is another good example of the lack of meaning of the cyclomatic number.
Figure 8 explains the situation.

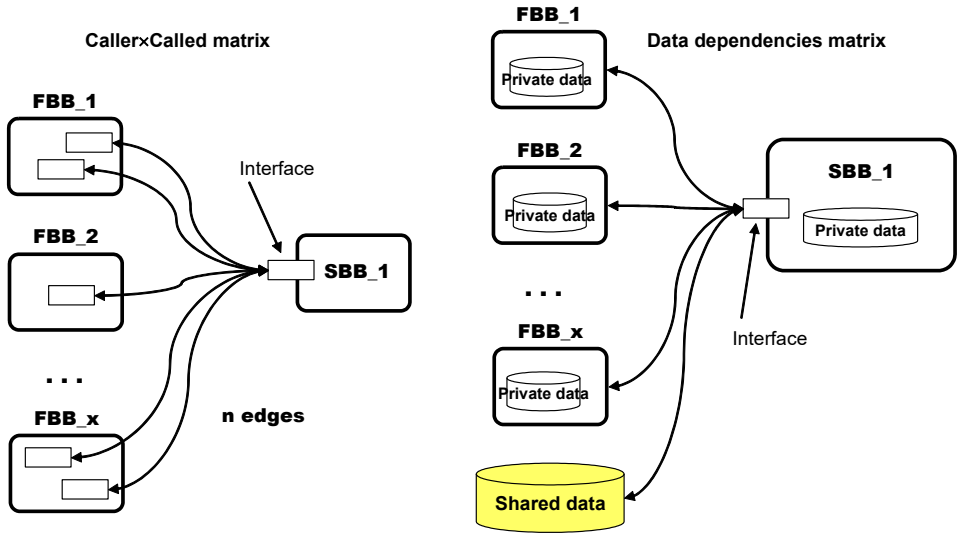

Figure 8 – Minimisation of edges

The cyclomatic complexity number CN is obtained with the graph theory well known formula : $CN = e - n + 2 \times p$, where e is the number of edges, n is the number of nodes, and p is the number of non connected sub-graphs.
SBB_1 is a black box such as any language instruction or built-in function. The link between an invocation point from an FBB to the SBB is not a normal edge from the tester point of view. Thus the number of edges to take in account to compute the cyclomatic complexity is not e but $e - k$ where k is the number of edges connecting the FBBs to the SBBs ; this number may be computed using the caller×called matrix.
Moreover, all the nodes and edges belonging to the SBBs must also be subtracted, because the corresponding complexity will be hidden by the interfaces to the SBBs (again, see D.Parnas modularity principles, in [A7]). In the same way, the shared data between the FBBs, have no influence on the SBBs as long as the programmers respect strictly the interface contracts (to be checked by formal reviews, under the responsibility of the integration team leader).
Concerning the data, it is to be noted that if m data items within the FBB have dependencies with n data item from the SBB, the cardinality of the set of data×data relations is $Card = m \times n$ (at least). If an interface has been defined, the cardinality becomes $Card = m + n$. While SBB is tested independently, then the number of data path to validate the integration of the FBB with the service blocks SBB is just m.
The simplification brought by the modularity is well reflected by our complexity measure because modularity means less tests to develop.
Identification of SBB is the result of the abstraction process which is the essence of good design for reuse ; and also for the rigorous foundation for DSL, see ref. [A26]. The

effort to develop such a software system, using the *Eff/KSL* relation, is less than to develop the same software system when SBB has not been identified. The ratio of SBB code in a program is an indicator of a good design.
According to our definition, a software system using SBB is less complex as any other because less tests are needed. So the definition still works.

Costs of interfaces

Once the design of the software system is completed, the situation is as follows:

- SBB have been identified either in the software system itself, or reuse from an already existing one, or available in a specialized library. In the SOA approach this is the role of WSDL.
- Interactions between FBB are specified by contracts, according to figure 9.

FBB are connected to each other in order to form business services. The flow of control between FBB may be represented using UML activity diagrams such as flow graphs, or much better by finite states machines. However the flow of control is just one way to define interfaces between FBB. Interfaces may also be defined from coupling between FBB through shared data and/or events, or by remote procedure call (RPC) invocations through the network. It is one of the most difficult aspect of good design to document them all. If some of the coupling have been forgotten, that means that it exists somewhere in the software system covert channels nowhere documented. Consequently, they will escape to systematic integration tests. Such a situation occurs when two or more programmers negotiate an interface without to inform the software architect. Prevention of such design flaws is the role of quality assurance (inspections, design reviews, …). For more details about interfaces see [A8, chapter 15] and the various kind of matrix representation such as N^2 matrix [see A21, A8 chapter 2].
Note that the most common N^2 matrix are FBB×FBB (Caller/Called matrix), FBB×Data, FBB×Events, but that Data×Data (functional dependencies), Events×Events (event management by abstraction level) must also be considered. More complex matrix of higher order are necessary to describe the relations between contracts, implementation, technologies. All these matrix have an impact on complexity and the cost of integration.
Cost of an interface, in a first approximation, may be depicted as in figure 9. The cost is split into elementary costs such as:

- Cost of the interfaces specification which needs explicit agreement between the programmers and actors which will use it (the end-users for man-machine interface).
- Cost of quality assurance during the programming of building blocks (inspections, reviews) to ensure that rules have been correctly applied in order to detect and remove defects early.
- Cost of the test programs development (in parallel to BB programming) to be run at integration time, in order to validate the integration of the FBB.

More complex situations may be observed with man-machine interface (MMI) [see A15] or with system interoperability in System of System (SoS) approach, now common in the enterprises and the administrations.
The sum of all these costs corresponds to the integration effort.
We can now refine our initial definition as follows:

> *Integration of the building blocks of system A is more complex than integration of the building blocks of system B is the cost of integration of A is greater than the cost of B. The cost of integration depends on the number of FBB and on the number of relations between them to validate.*

Complexity of the system is measured by the cost of integration of its FBB.

Now, the last problem is to enumerate and classify all the various interactions and relations that exist between the FBB and the way to track them.

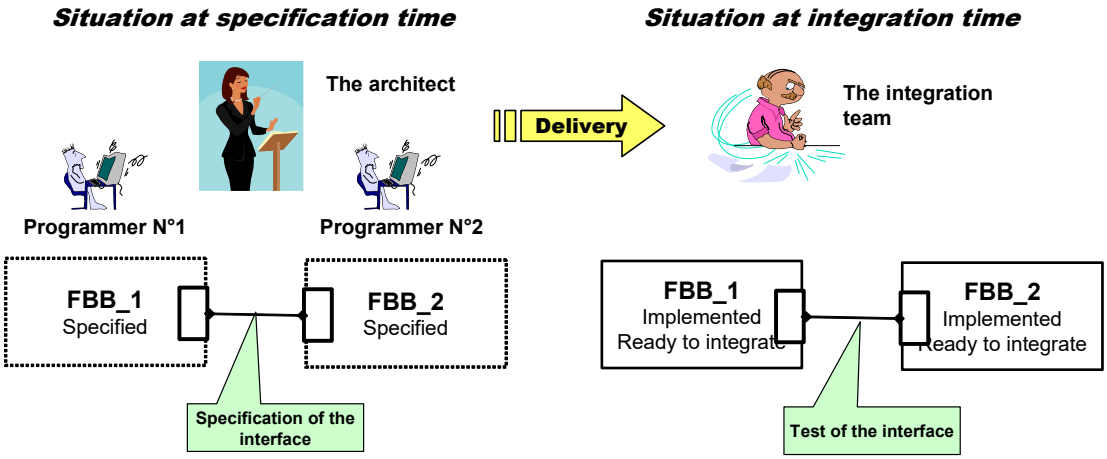

Figure 9 – Cost of interfaces

The specification of the interface and the proof of concept is part of the contract, as well as the test of the interface which in fine will validate the contract implementation.

The complexity of integration

The complexity of integration depends on:

- the number of building blocks to integrate,
- the number and nature of interfaces between blocks (synchronous or asynchronous flow of control, shared data, messaging, raising and catching of events, remote services invocation, return codes of services invocations),
- the size of an integration step, that is to say the number of BB to be integrated to form a new BB ; we will call this important number the "SCALE" of the integration.

For the integration step, human ergonomics constraints recommend to integrate with a SCALE = 7±2 BB at a time. With *n* blocks interacting, the errors are to be searched in the set of parts $(2^n - 1)$ formed by interacting blocks (NB : that shows why "big bang" integration is just stupid !).

Note that by definition the service blocks SBB may be validated independently of the context. Consequently, an efficient integration strategy will be to start the integration process, a) first with the in depth validation of SBB using a simulation test environment under control of the integration team, and then b) to go to the nominal integration environment using the FBB. Doing that way, one has the assurance that in case of errors with the SBB, the fault is in the calling context, not in the SBB itself.

The integration process is depicted in figure 10.

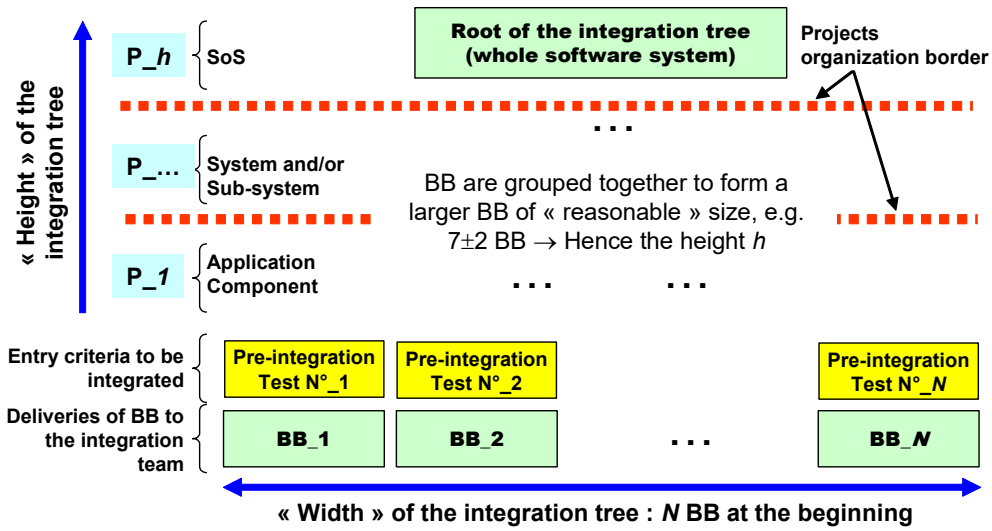

Figure 10 – Integration tree layout

At the beginning of the integration process (step 0) each BB is tested from the integration point of view. Tests are written according to the information contained in the contracts between the development teams and the integration team. They are mainly black-box tests, based on the interfaces known of the integration team. Robustness is an important aspect of pre-integration in order to check the behaviour of the BB when wrong data are injected, and more generally when interfaces are violated. The number of wrong cases to consider for these tests may be very high, according to the programming style. At each step, additional integration tests must be provided to validate the new BB constructions.

The height of the tree varies like the *log* of the width of the tree (the base of the *log* is the scale 7±2 if one respects human ergonomics and communication constraints). For example a software system of size 500 KSL constructed with 250 BB of 2 KSL each in average, will have a height of 3. Number of additional tests to provide will be 30+4+1=35. At each step, there are less tests to provide but the size of the test programs (and effort to develop them) will be larger. In total, the number of tests set (test cases) for integration will be : 250 (step_0, initial) + 30 (step_1) + 4 (step_2) +1 (the whole system) = 285.

The strict application of the Parnas's modularity principles forbids the construction of degenerated integration trees as in figure 11.

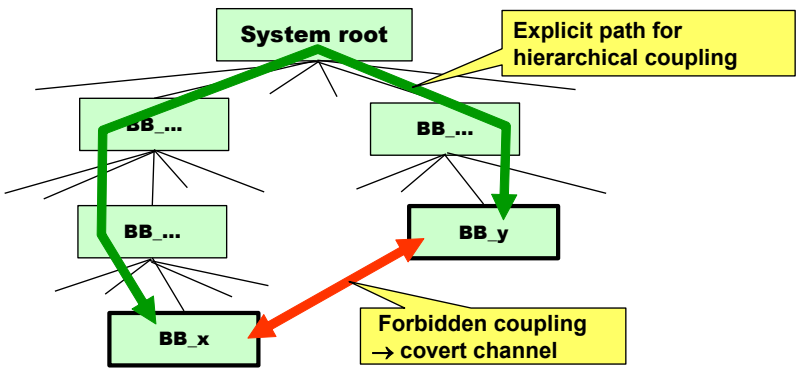

Figure 11 – Degenerated integration tree

The effort at each step may be defined qualitatively as follows:

- Step_0: Effort for the *N* pre-integration tests of the BB ready to integrate,
- Step_1: Effort for *N1* new constructed BB (a % of step_0 effort) with $N1 = \frac{N}{scale}$,
- Step_2: Effort for *N2* new constructed BB (a % of step_0 + step_1 effort) with $N2 = \frac{N1}{scale}$,
- and so on, until the root.

Again, we have a power law, **if and only if the tree is rigorously hierarchical**.
This approach could be associated with what is called "recursive graphs" in [A19]. The weight to be used is what we called "cost".

Avoiding covert channels depends on the programming style, and also on the capability and experience of the programmers. Let's catch a glimpse at the nominal structure of a BB.

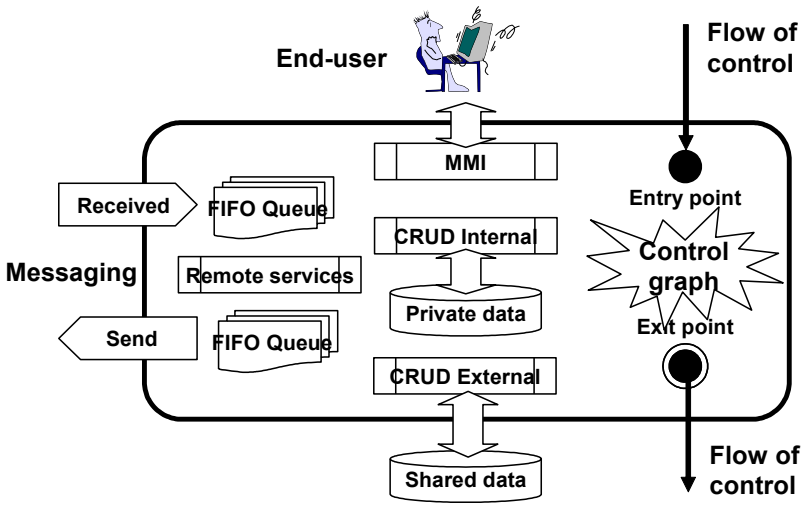

Figure 12 – Nominal structure of a building block

To master the BB environment, the integration manager must know:

- The flow graph between BB, i.e. the activity diagram connecting the BB each others, what is also called the caller/called matrix, or N^2 matrix in system engineering [see A21].
- The data graph between BB and the referenced data (private and shared), also called data-dictionary, indicating the type of the reference (CRUD/ Create, Retrieve, Update, Delete). Flow graph + data graph allow to detect data inconsistency as for example a wrong R or U operation on a data that has been D previously.
- The event graph in order to know which BB RAISE and/or CATCH such or such events, and filter the authorization for SEND/RECEIVED network operations.

Private data and shared data must have the property of transactional memory (ACID properties) in order to avoid data inconsistency when the BB execution is interrupted for any reason. On transactional memory, see [A8, A16]. Transactional memory means that the source code which modify the memory must be programmed in the transaction programming style ; on transactions, see ref. [A27].

Consequently, if the programmers are not aware of the architectural rules to be applied, and if the application of the rules is not rigorously verified (quality assurance) the tree will not be hierarchical. Covert channels will be unavoidable.
We know from distributed system architecture that n processes having each m states, when interleaving, will create a global state with m^n elementary states to check [see A22]. The m states are clearly related with the existence of shared data. If the Parnas's principle is rigorously applied, i.e. m=0 states, the global state is 1, $\forall n$. This is a supporting evidence to strictly control the size of the shared data and to use transaction processing style. Notice that it is true with SBB, by definition of SBB.
Let's now see how the composition process works.

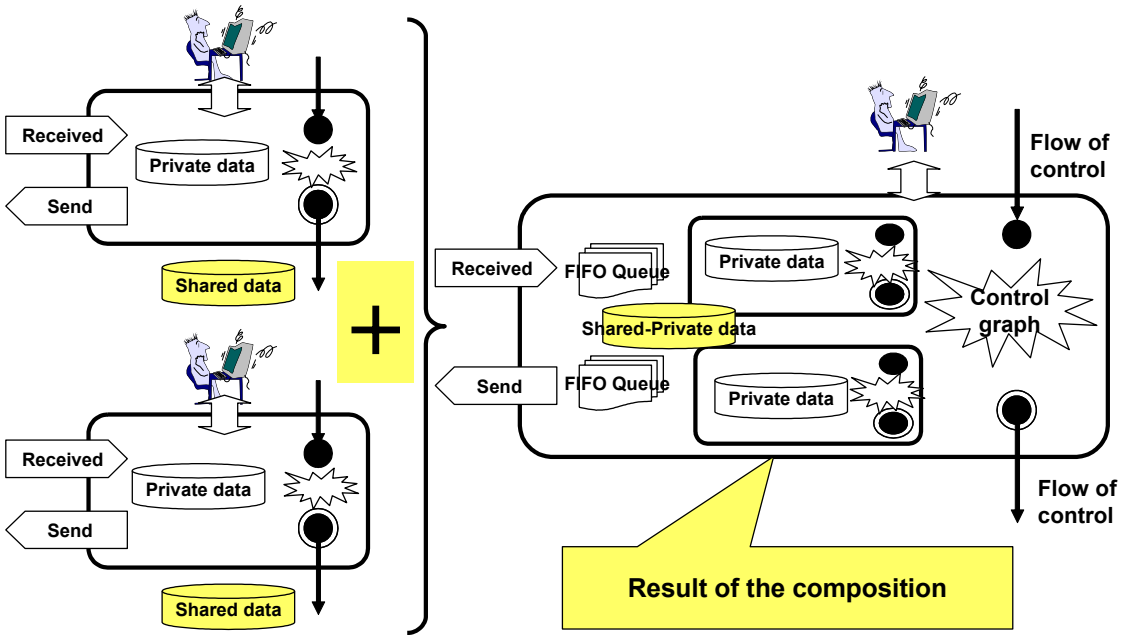

Figure 13 – Building blocks composition

What was private data remains private, but what was shared may become private to the new composed BB. Doing that (this is an emerging architecture property) one give the new BB the capacity to control its environment. Notice that the new BB is now in conformance with Parnas's principles. It is a way to master combinatory explosion.
Usage of shared data has generally two main causes:

1. The first one is performance or safety (recovery from a failure), sometimes for good reasons,
2. The second one comes, almost always, from programmers incompetence, especially when they are not aware of distributed system pitfalls, ignorant of transaction processing, or tempted by event programming.

Let's have a look on the performance problem explained in figure 14.
Case N°2 is in conformance with the Parnas's principles, but several messages, or large messages, may be necessary to transfer the information context between BB_1 and BB_2, thus creating a potential saturation of the network which eventually will create performance problems.
Decision of the architect depends on the perception of risks. In all cases the access to the shared and private data must be carefully control.

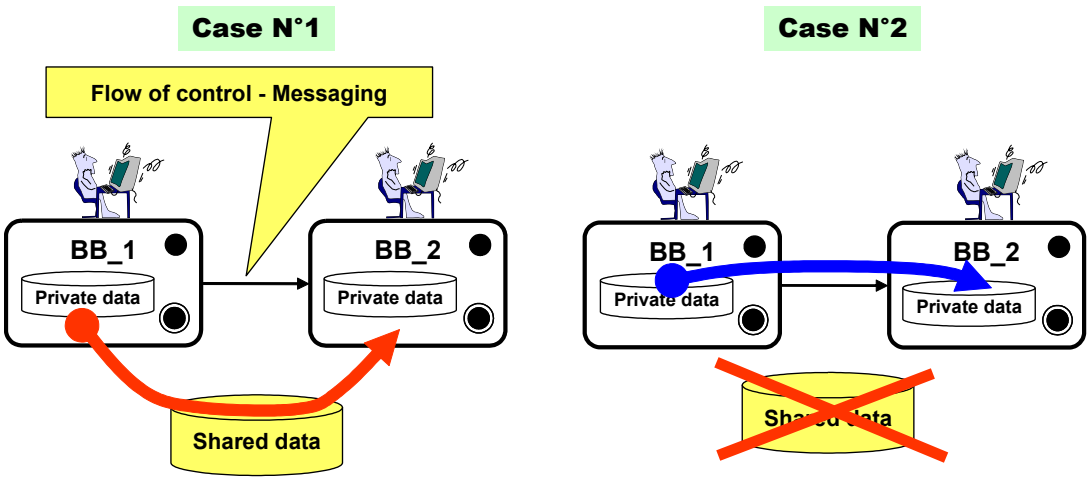

Figure 14 – Shared data and performance

If the architect decides that for performance reason a context has to be defined, extreme caution must be addressed to the programmers. Consider a set of BB sharing the same context CNTXT. The programmers of BB_x which want to be sure that the CNTXT is sound are encouraged to use defensive programming style ; consequently, before to use the context they have to check it by on-line tests (semantic invariant, pre and post conditions, etc.). The size of on-line tests depends on the size of the shared data and of the property associated with the data, but also from the history of the successive modifications of CNTXT. If something wrong occurs, a diagnostic must be sent to the system administrator. That shows how carefully the design of shared data must be done. If the programming team is not aware of the danger of shared data, a quality catastrophe is unavoidable.

Interoperability

The case of interoperability is interesting because this is a classical pattern with information systems and system of systems (SoS) ; see [A19, A25]. It shows how a "good" technical solution with a "software bus" may be entirely corrupted if the architecture of data has not been done correctly.
The "theory" of interoperability is summarized in figure 15.

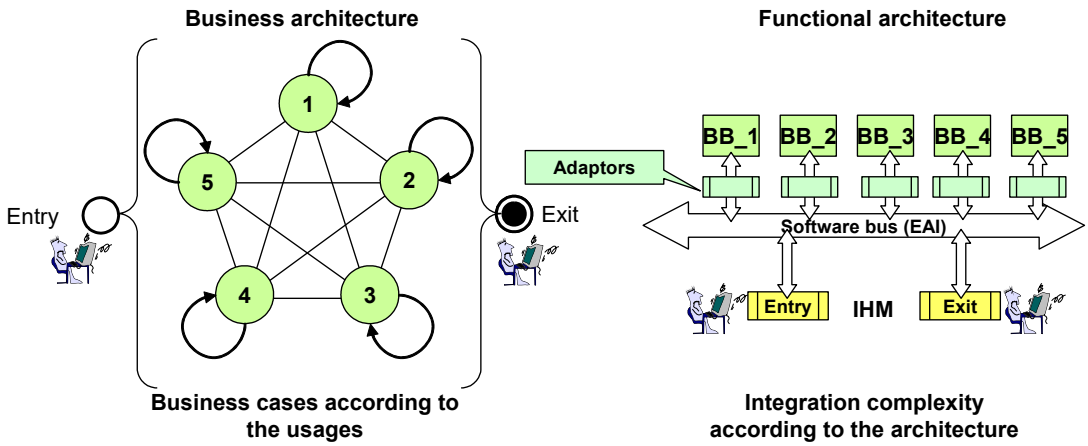

Figure 15 – Interoperability, from business architecture to functional architecture

The flow graph of the functional architecture is simpler than the one of the business architecture (linear against quadratic). It corresponds to the deployment of a software bus, with adaptors to translate the messages exchanged. If we look at the data belonging to each BB, and if we try to organize them according to a classification with criteria private/public, the situation is like this:
Each BB has his own private data, but if we consider the shared data we have to consider what is common for each pair (i.e. $\frac{n(n-1)}{2}$ pairs), common to each triplet, and so on, until we find what is common to all. This time we are facing the set of parts, i.e. $2^n - 1$ which is an exponential. That shows why if the deployment of the EAI is not associated with an in depth reengineering of the data architecture for the whole system, and also a reorganization of the applications, the situation will probably be worst. Consequently, to take advantage of the introduction of an EAI or ESB, a complete reengineering of the various BB is mandatory in order to analyse the data model of the software system. The architecture with the EAI looks like sound to someone not aware of the pitfalls, but if we measure it with the tests to validate it, it still remains a very complex one.
If we try to integrate systems which have redundancy, the shared data will be automatically large. Integrate them will oblige to decide who is responsible of what.
All these examples show the extreme importance of the data reference graph and its impact on testing strategy as far as complexity is concerned. Data architecture must be done first.
With the events, complexity is still evolving in the wrong way. In an on-line distributed architecture, which offers facilities like PUBLISH/SUBSCRIBE, the natural tendency, if not rigorously controlled by the architect, will be to publish to all. Doing that, one creates invisible links between the publisher and many subscribers for which the message may have no meaning. The arrival of a message to a receiver which is not the good one may provoke inappropriate reactions according to the real situation.
A "PUBLISH to all" command will normally generate appropriate tests in all BB that are potential receivers. Again, size of tests to provide for validating the PUBLISH command are a much better complexity indicator than the flow graph.

Temporary conclusion

Measuring, or trying to measure, complexity of a software system with the size of tests to validate it has several advantages.

- It's a real measure. Size of tests versus quality is tractable. If well managed more tests means better quality.
- More complexity means more tests: this is common sense that everybody can understand. Each time new features are required by the customer, requirements must be translated first in term of "how much test" are to be added and re-executed to validate the new features, from the customer point of view, in order to warrant the system SLA.
- It focuses the attention of the software architect on what will facilitate the testability of the software system. It will create a virtuous circle round the "design to test" concept as our hardware colleagues did since the very beginning of computer age.
- It focuses the attention of the software architect on dangerous software constructions which have the appearance of simplicity, i.e. a few lines of code, like in event programming, but that will create invisible connections and

coupling between unexpected chunks of code; consequently new test programs costly to write will be necessary to warrant quality.

- It gives a rigorous meaning to what agile and X-programming communities call Test-Driven Development when they claim: “Do tests before the programs”.
- A program is similar to a theorem, but like in mathematics, the proof/test is the only valuable thing. What will be the status of a program/theorem without a proof/test? Just magic, not engineering.
- It is a way to make the risk visible to the customers with simple reasoning like this: “your requirements will add complexity; complexity means more tests; tests are costly and difficult to design. If you don’t accept to pay for more tests, you will have to accept more risks”, and so on.
- Its shows why integration is an intrinsic part of software architecture, and must not be separated from it: testable architecture, or design to test are key aspects of software engineering.

To conclude with a comparison with other fields of engineering such as aeronautics, nobody would accept to flight on an aircraft whose behaviour has not been proven correct and safe, with all available means, humanly and technically ; in civil engineering, the responsibility of the architect is committed during ten years. With software engineering state of the art, a lot of techniques are now available to improve the quality and warrant SLA as stated by maturity models like CMMI, but too often the general management don’t yet accept to pay for it. The risk is then transferred to the customer and to the public, in total contradiction with what is claimed by quality management.

Appendix

To go further we have to define a programme of work qualitatively and quantitatively.
Qualitatively, we have to check with the academic and industrial community that measuring complexity by length of tests is pertinent.
Quantitatively, it is not so easy, because industrial statistical data are needed.
A first direction could be to capture test effort, for example integration test effort with regards the complexity of the integration tree based on the system design. It should be easy with software organizations claiming to use architectural patterns like MVC, EAI/ESB, SOA. As recommended by the CMMI, effective quality may be measured by the number of residual defects discovered during a given unit of time of system use (as for reliability measurement). Measuring quality by residual defects has been used intensively by NASA SEL many years ago and is still pertinent. Counting of effort is a basic measurement for project management fully mastered at level 2 of the CMMI scale. Thus working with organizations at level 2 could provide pertinent data, but level 3 organizations would be better.
A second direction could be to define a unit for test length measurement, in the same way units of programming effort have been defined for cost estimation models. For example, COCOMO use a unit which is the effort to develop a chunk of 1.000 lines of code (one instruction per line, without comments ; see ref. [A2]). This is a statistical measure extracted from projects data analysis ; same thing with Function Point measurement based on data transformations.
Test length, as previously seen, means:

- Length of test programs. Length is easier to define if tests are expressed in some test languages or expressed in a classical programming language, in particular if the programs under test have been instrumented. Otherwise, it could be difficult.
- Length of test data to be used by test programs. Such data are easy to express in data language like ASN.1 or XML.
- Length of programs developed to compute pertinent data.

Notice that organizations claiming to achieve CMMI level 4 or 5 should have such metrics available on demand for evaluation purpose.

We are currently in the process of creating a working group on “Integration and complexity” within CESAMES (see web site at http://www.cesames.net), an association chaired by Professor Daniel Krob, to validate this new approach.

Thanks to those which have made comments and help to improve the third revision of this paper.

References

[A1] J.Printz, *Productivité des programmeurs*, Hermès, 2001.
[A2] R.Park, *Software size measurement: a framework for counting source statements*, Technical report, CMU/SEI-92-TR-20.
[A3] T.J.Mc Cabe, *A complexity measure*, IEEE TSE, Vol. SE-2,N°4, Dec. 1976.
[A4] C.Berge, *Théorie des graphes et ses applications*, Dunod 1963.
[A5] K.Popper, *The logic of scientific discovery*, 1968 (En français chez Payot).
[A6] B.Boehm, *Software engineering economics*, 1981, and *Software cost estimation with COCOMO II*, 2000.
[A7] D.L. Parnas, *On the criteria to be used in decomposing systems into modules*, Communications of the ACM, Vol 15(12), 1972.
[A8] J.Printz, *Architecture logicielle*, Dunod 2009.
[A9] G.K Zipf, *The principle of least effort*, Hafter Publishing Co., 1965 – B.Mantelbrot, *Les objets fractals*, Flammarion.
[A10] M. Schooman, *Software engineering*, Mac Graw Hill, 1988.
[A11] C.Arndt, *Information measures – Information and its description in science and engineering*, Springer, 2004.
[A12] Chaitin, *Information randomness and incompleteness*, World Scientific, 1987 – See also J-P.Delahaye, *Information, complexité et hazard, Hermès*, 1994 – Handbook of theoretical computer science, MIT Press, Vol. A, Chap. 4.
[A13] K.Beck, *eXtreme programming explained*, 2000; and also: *Test driven development*, 2003.
[A14] NATO Science committee, *Software engineering*, Garmisch, 1968 et Rome, 1969
[A15] I.Horrocks, *Constructing the user interface with statecharts*, Addison-Wesley 1999.
[A16] J.Larus, C.Kozyrakis, *Transactional memory*, Communications of the ACM, 07/08 Vol 51 N°7; N.Shavit, *Transactions are tomorrow's loads and stores*, Communications of the ACM, 08/08 Vol 51 N°8; T.Harris & alii, *Composable memory transaction*, Communications of the ACM, 08/08 Vol 51 N°8; C.Cascaval & alii, *Software transaction memory: why is it only a research toy*?, Communications of the ACM, 11/08 Vol 51 N°11.
[A17] J-F.Pradat-Peyre, J.Printz, *Pratique des tests logiciel*, Dunod 2009.
[A18] J.Printz, *Ecosystème des projets informatiques*, Hermès, 2006.
[A19] Y.Caseau, D.Krob, S.Peyronnet, *Complexité des systèmes d'information : une famille de mesure de la complexité scalaire d'un schéma d'architecture*, pôle System@tic.
[A20] F.Brooks, *Mythical man-month*, Addison-Wesley, 1975 (new edition in 2000).
[A21] INCOSE *System Engineering Handbook*, version 3.2, January 2010.
[A22] E.Clarke, & alii, *Model checking*, MIT Press, 1999.
[A23] M.Utting, B.Legeard, *Practical model-based testing*, Morgan Kaufmann 2007.
[A24] B.Legeard & alii, *Industrialiser le test fonctionnel*, Dunod, 2009.
[A25]Y.Caseau, *Urbanisation, SOA et BPM*, Dunod, 2008.
[A26]S.Kelly, J.Tolvanen, *Domain-specific modeling*, Wiley, 2008.
[A27]J.Gray, A.Reuter, *Transaction processing: concepts and techniques*, Morgan Kaufmann, 1993.