

**Risks of faults intrinsic to software languages.
Trade-off between design performance and application safety**

G. Motet

**Fondation pour une Culture de Sécurité Industrielle
& Université de Toulouse, INSA, LATTIS
Gilles.Motet@icsi-eu.org**

Abstract

A lot of products embed software applications carrying out complex functions (e.g. cars, aircrafts, and medical equipments). More and more authority for control is placed on these applications whose failures may lead to accidental loss. Numerous methods have been developed to prevent these failures. These methods apply either to the programs developed or to the design activities. However, the programming or modelling languages used to operationalise the solutions as software applications are rarely questioned. These languages constitute the technology of realization of the program. On one hand, the language features are selected to increase the development performance and to decrease the software application costs. On the other hand, these features may be at the origin of specific types of faults which constitute the intrinsic risks of these languages. Therefore, the choice of a language or of a subset of a language, leads to a decision-making issue of how to deal making safety and performance trade-offs. The first part of the paper analyzes the evolution of the programming languages. We show that these changes were aimed at the convergence of design performance improvement with designed application safety. The introduction of the object-oriented technologies breaks this commonality. They cut the development expenditures but introduce new types of faults. The decision makers such as the critical software application producers (e.g. the aircraft manufacturers) and the authorities (e.g. the avionics certification authorities) have to deal with this trade-off. These new technologies cannot be just rejected as they are more and more often used in certain domains (e.g. mobile phones, internet applications). The proposed constraint on using them seem to be too restrictive and are not justified. In particular, the safety levels of software programs developed applying these constraints are not assessed. The second part of the paper addresses these questions. It specifies the problem and it proposes a method to estimate the risk of faults in object-oriented programs. Thus, the decision-makers can elaborate rules for using object-oriented languages establishing a trade-off between the wished-for development performance and the required safety levels.

1. Software applications, programs and technologies

1.1 Safety and economic characteristics of software applications

Software technologies are seeing increased use to operationalise functions of products. Numerous arguments justify their use. We will focus on safety and economical characteristics which will be illustrated on systems embedded in automobiles.

Software technologies allow complex behaviours to be specified and carried out quite easily. They provide functionalities asked for by consumers which cannot be implemented by another technology. For instance, certain car shock absorbers are controlled by programs to increase comfort. These behaviours can adapt to driving conditions by processing values from actuators (speed of the car, road curvature, etc.). This capability is also used for spark ignition, fuel injection and valve control to reduce emissions. The costs of these complex

functionalities are not excessive. They are introduced initially for up-market cars and soon become available for bottom-of-the-range models.

Software technologies intrinsically improve systems safety as they are not affected by ageing. For instance, electronic ignition used to control spark timing avoids the tuning of the points. Failures are rarely due to the production step, which is limited to the loading of a binary code in a memory. Any faults triggered during this phase are also easily detected. This leads to low production costs for very reliable products.

Complex software functions are more and more often developed to increase product safety. ABS (Anti-lock Braking Systems) and ESP (Electronic Skid Prevention) systems prevent accidents. Airbag control systems enhance the protection of passengers at the moment of an accident. Mercedes-Benz has developed the “Pre-Safe System” which detects the imminence of a potential accident and anticipates future actions of the driver. For instance, the vehicle’s hydraulic braking circuits are pressurized to be operational as soon as the driver presses the pedal.

General Motor predicts that the average car will have 100 million lines of code by 2010 (source IBM). Today, more than 80% of the innovation in a car comes from the intensive use of software technologies. This significant role for software technology is also present in aircraft. The Airbus A380 required the development of over one billion lines of software code. Programs are also an important part of mobile phones (up to 5 million lines of software code) and medical systems (medical robots, pacemakers, etc.). Some data highlight the key position of software technology in the R&D effort of industrial sectors (TNO, 2005): automotive: 2002 (22%), 2015 (expected rate 35%); medical equipment: 2002 (25%), 2015 (33%); aerospace: 2002 (35%), 2015 (45%); consumer electronics: 2002 (42%), 2015 (60%); telecom equipment: 2002 (52%), 2015 (65%).

The greater role played by software has other implications. More and more authority for control is transferred from the product’s users to software applications embedded in these products: for example braking pressure diminution by the ABS when the driver presses the brake pedal or, conversely, the braking of one wheel by the ESP without driver action. The previously mentioned “Pre-Safe System” may also provoke an automatic speed reduction when an accident appears imminent. This means that, firstly, the guarantee of the correctness of the software applications is essential for the safety of the users of these products. Failures of software systems embedded in cars have been partly to blame for several accidents leading to death or injuries. Elims (2004) estimates that 4 deaths and 300 injuries a year in the UK are due to failures of software embedded in cars. His estimation is based on product recalls. Secondly, the failures may also impact the firms which produce the systems. More and more often they lead to legal actions from users or authorities. The judicial proceedings of the American Federal Court against a car manufacturer for potential air pollution due to an erroneous emissions-control program are an example (USDJ, 1999) (CEPA, 2003). Moreover, the integration of the products in complex systems increases the safety requirements in order to avoid domino effects; the failure of a software application, which may lead to the malfunctioning of coupled products, etc.

1.2 Correctness and design performance of software programs

The low production costs and the high reliability of software technology lead to widespread use of software applications in all the products which provide more and more complex and

critical functions. However, these two operational qualities (cost-effective and safe) are only available in operation if they were previously realised during the design. Even if no faults can be introduced in the program at run-time, their presence due to designer error will be duplicated in each software application derived from this program. Faults are systemic, rather than being due to random causes or wear-out mechanisms¹. Moreover, the higher the development expenditure is, the higher the sale price of the products embedding software is. The two requirements (low costs and high reliability) inherent in the software technology exploitation then are dependent upon the program development.

For many years the achievement of these requirements was assessed *a posteriori* by looking at the development results. This approach leads to catastrophic observations:

- Few programs were found to be correct at delivery time, and now this rate will certainly be higher. The reuse of the same program in thousands of products leads to a relatively large number of failures. For instance the suit against Toyota previously mentioned (CEPA, 2003) affected 330 000 cars. Moreover, the safety-critical nature of numerous software applications, even of one copy, may cause severe harm. In 1993, the baggage management system of Denver airport delayed the airport opening for 16 months. The excess costs were estimated at \$500 million for the airport and \$480 million for users (mainly the airlines) (De Neufville, 1994). Problems on the same kind of computerized system also occurred at the opening of the new terminal 5 of Heathrow in 2008 (Millward, 2008).
- Delays during development and cost overruns are still significant.

They are several explanations for these observations. Two of them are characteristic of software applications:

- Applications sizes are growing quickly. The cost is rarely compensated for by increases in engineers' productivity. A salary cost of \$250 per line of code² leads to \$250 million for an application of 1 million lines of code. A human error rate of 10^{-3} per line of code leads to one thousand faults in such an application.
- Software applications are increasingly coupled to provide new functionalities. For instance, the CAN bus of a car allows data to be exchanged between software components. The engine control software application interacts with the engine (to control the valves, injectors, etc.), receives data from the exhaust pipe (to limit emissions), from the air conditioner (to prevent the engine from stalling when it is idling and the air conditioner switches on), etc. This coupling of correct applications leads to the occurrence of new types of integration faults.

Numerous methods have been proposed to improve safety and design performance. They act on two features: **the program** and **the activity** of the designers. However, the problems have been perpetuated due to two main causes: the use of inefficient development methods and tools and the inefficient use of potentially good methods. To overcome these problems, methods whose efficiency is proved or at least estimated *a priori* should be used. Moreover, the effectiveness of their use should be proved or at least estimated *a posteriori*. Some examples will be quoted in sections 1.3 and 1.4 illustrating the difficulty of establishing these estimations. We will highlight the consequences on the trade-offs done between development performance and the safety of the software applications.

¹ We will not consider malicious intents or applications whose specifications provide functions intrinsically hazardous leading to harm.

² \$1000 for a space shuttle line of code, several hundred for an avionics software, and \$50 for a conventional software application.

1.3 Methods applied to the programs

With regard to the absence of faults in programs, multiple fault removal techniques have been proposed. Unfortunately, their actual effectiveness has not been assessed. Organizations superimpose numerous techniques on top of each other to prove the seriousness with which the development firm or the organisation providing guarantees (e.g. certification authorities) takes the problem. When a new type of fault is discovered *a posteriori* (during the software application's operation), a new technique handling this fault is added. This fault type is considered to be exceptional because the fault was discovered recently. Then, the technique is only imposed for programs implementing highly-critical functions (e.g., associated with a failure rate less than 10^{-7} or than 10^{-9} per hour). In fact, this approach is purely cost-driven. We accept to invest more time, that is, more money, in verifying critical functionalities. Nevertheless, the new fault type may be due to the use of a new software technology or a new software development process. Therefore, this type of fault will occur frequently and the detection technique should also be used for non-critical applications. Moreover, old methods can become useless as new technologies or new process may eliminate the faults they detect. Thus, trade-off between safety and economic requirements seems to be based on a simple and rational principle: more expenditure leads to more safety. However, the actual effectiveness of the techniques used is estimated neither in economic terms nor in terms of safety. Probably, the numerous techniques are redundant and used simultaneously would allow many faults to be detected several times over. However, as they are applied in sequence and as faults are corrected after detection, this fact is not assessed.

Measurements of the efficiency of use of different dependability techniques have rarely been made. And yet, this aspect is fundamental for safety as well as from an economic viewpoint. For instance, behaviour-proof techniques are certainly promising. For example, SPARK (Barnes, 2003) allows expected properties of the system to be expressed and then demonstrated. These techniques have been used in the avionics domain (Chapman, 2000). Their effective use depends on the relevance of the properties checked and thus on the skill of the engineers. Moreover, the complex relationships between uses of multiple techniques are rarely dealt with. For example, functional testing, whose use effectiveness depends on the behaviours actually activated, often assumes that a given behaviour is obtained by a single internal run of the program. This assumption is sometimes wrong. For instance, aircraft altitude provided to the pilot is calculated by different systems and programs according to the aircraft's location (high altitude, approaching, close to the airport). So, one function (provide altitude) is implemented by 3 software systems in a transparent way for the user. Thus, as functional tests handle values in a continuous domain (e.g., the aircraft altitude is a real number), they are completed by structural tests examining the internal features of the program processed during its operation. However, these two techniques are generally applied separately whereas an efficient use depends on the understanding of their subtle interactions.

The lack of evaluation of the efficiency of the fault avoidance techniques and of their use leads developers to superimpose the techniques required. Engineers often consider that the constraints imposed by this work overload could be harmful to software application safety. They weaken and diffuse the technical responsibility for detecting faults. This increased spending could be chosen in a more informed way and the responsibilities to treat a given class of faults better allocated if the efficiency of each of the techniques could be assessed. This assessment is very complex as no statistical figures are available. For instance, in the

avionics domain, the mechanisms handling failure rates up to 10^{-4} per hour can be justified by realistic data from flights. However, affirming that a given technique allows the failure rates to be reduced to 10^{-7} or to 10^{-9} per hour is an assertion for which there is no contrary evidence (McDermid, 2006).

The adoption of an increasing number of techniques dependent on the defined Safety Integrity Level (SIL) is also debatable from a cost point of view. The economic impact of these techniques is difficult to establish. Early detection of faults reduces the global costs of development. If the costs of corrective maintenance are added to the cost of use of the safety techniques, overall costs are greater for the lowest levels of integrity (the lowest requirements of safety). Two reasons for this are specific to the software technology:

- Spending on corrective maintenance is very high because of the need for diagnosis of the fault in a complex program, correction without introduction of new faults (no side-effects), re-use of all the techniques of safety to assure that the guarantee is preserved.
- More importantly, the correction of the fault is essential because the failure of a software application is not due to a statistical chance affecting a single copy of the product. The fault exists in all the products which use the program. The expenses of recalls of all these products are thus cumulative.

In most domains, no trade-off is made between the achievement of the two objectives of development performance and safety. Indeed, safety methods are actually considered as dogmas, even if the statutory texts or the standards often use the term "should" instead of "shall". Therefore, the expense incurred to ensure the correctness of the design of a software application is often equivalent to or greater than other design costs. A study of the efficiency of safety methods should probably cut these costs by reducing the number of methods used or the duration spent for using them. The comparison of the design of a software program with the manufacturing of a product raises interesting questions. The improvement of the manufacturing process of a hardware product leads both to reductions in the costs of checking and to very low rates of failure. Why should this not be true for the process of software development? How could human errors be reduced to prevent the introduction of faults in the programs?

1.4 Methods applied to design activities

Again, numerous methods have been proposed to control the development made by a designer or by a design team. These include the following (without being exhaustive):

- A development process which structures the activity in stages whose objectives and types of results are specified;
- Guidelines which stipulate good practices to realize a stage (e.g., design patterns) or to express the results (e.g., programming guidelines);
- Tools which automate all or any of the stages (automatic generation of code, re-use of components, etc.).

These methods are often thought of as benefiting safety as well as development performance. Indeed, by preventing faults, they reduce the eventual expense of searching for and correcting them at the end of the project. McDermid (2001) mentions that the global costs of development of programs depend very little on their level of criticality (SIL). He moderates this conclusion by clarifying that it can be biased by the higher skill of engineers allocated to

more critical programs. However, again, the efficiency of these prevention methods and of their use has not been evaluated, in particular because they concern human activities. Therefore they never replace the detection techniques applied to the programs later on.

Techniques applied to programs and those concerning the software development activity are often correlated. Certain standards specify the techniques to be used at each stage of development. Besides the generic standard IEC-61508 (1998), numerous sector-based standards exist (as the DO-178B (1992) in avionics); some are derived from the IEC-61508 (for example EN-50128 in the railway domain). The multitude of standards is due first of all to the absence of measurement of the efficiency of the methods to achieve safety. Every sector establishes its beliefs and makes a trade-off to decide on the list of its own good practices.

Benedicktsson (2000) led a theoretical study on the costs and the contributions of practices associated with each SIL (Safety Integrity Level) of the IEC 61508. This study was based on the COCOMO II model. It quantified:

- the effort of checking, which it demonstrated to increase with higher levels of integrity (SIL), and
- the effort of development, which it showed to decrease with higher SIL levels .

The synthesis of these two estimations shows that the global effort of development remains constant whatever the SIL. These figures confirm the fact that in the software domain also, investments in safety are profitable. However they are contradicted by the actual costs of the projects according to their criticality (ranging from \$30 to \$1000 per line of code) as the SIL level increases. The extra expenditures for designing critical software applications could correspond to the maturity of the company developing the software. Thus, the global effort is the same whatever the SIL is. But the money spent for one person/month is higher for developing critical applications. In particular, outsourcing is frequently used for conventional software development. The relationship between the maturity level of firms and the SIL of the developed programs is highlighted by the study led by Benedicktsson (2001). He shows the parallel between Safety Integrity Levels (SILs) defined by the IEC 61508 and the necessary Capability Levels (CLs) to reach SIL. The trade-off made between development performance and application safety consists in accepting extra expenditures to pay for the firm's capability.

1.5 Languages as software technologies

On the one hand, the designer and the program are identified as essential attributes of development performance and safety. On the other hand, the programming or modelling language used to operationalise the solutions as software applications are rarely questioned. These languages constitute the technology of realization of the programs. In other domains, the intrinsic risks associated with the technologies used are studied, generally before considering them for implementation. In software engineering, the software technology is seen as a neutral element, having little impact on development and safety. Even if ease of development and the reduction of faults are sometimes quoted in the chapters introducing books on the different programming languages, a systematic study would be useful concerning

- the risks of faults that are inherent to languages, and the ease of controlling them to guarantee safety,
- the impact of the language on development performance,
- the trade-off necessary between these 2 objectives when using language features.

In section 2, we explain the absence of work on these questions by presenting the evolution of software languages used to develop critical applications (avionics, automotive, nuclear, railroad, etc.). We will show that the evolutionary changes simultaneously improved development performance and the safety of the developed programs. We will analyze the origins of this fact. In section 3, we will show that new "Object-Oriented" programming languages question this observation and suggest that these trade-offs should be reconsidered.

2. More safety and more design performance

In this section, we analyze the evolution of programming languages, considering design performance and software application safety as criteria. Four phases will be highlighted. In the first there was a technical viewpoint focused on run-time requirements (section 2.1). This was followed by a second phase with a human point of view centred on the designer (section 2.2). The integration of these two components (socio-technical viewpoint) arose later as the third phase (section 2.3). The fourth phase also took into account the organization of the group of people involved in the development of complex programs (section 2.4). We conclude showing that these changes in programming languages aimed at the convergence of two objectives: the improvement of design performance with the safety of the designed applications (section 2.5). As these two requirements are jointly taken into account, no trade-off is necessary at the design stage to select the best language and to use it in the best way.

2.1 Technical viewpoint

Historically, the first programming languages had one goal: to provide operational statements, that is, features allowing the expected run-time behaviours to be described. Such statements included arithmetic operations on variables and constants, loops, etc.

In this period, safety was assessed by functional testing, executing the programs. The presence of faults, often considered as fatalities, was noticed *a posteriori* by discovering failures during program execution. The program was then modified and the functional test repeated. Numerous research works dealt with functional testing (Myers, 1979), for instance, to ensure the activation of all the potential behaviours by the test sequences.

Fault detection was improved thanks to static analysis methods allowing run-time errors to be detected *a priori*, that is, without execution of the program. These errors are specific to the statements of the language: division by zero, operation overflow, violation of the range when accessing an array, etc. Inspections were completed by automatic tools such as ASTREE (Cousot, 2002), successfully used on avionics software applications.

Thus, requirements concerning the design performance and the program safety were considered as independent issues, handled in sequence. The possible trade-off between performance and safety was not questioned. Moreover, the programming language was not considered as having any impact either on the ease of design or on the program safety. The language was only a technical tool used to implement a system.

2.2 Human errors

As previously mentioned, software technologies are not affected by wearing-out. Run-time failures are only due to the errors of the designer who involuntarily introduces faults into the program. In the second phase of programming language evolution, work focused on the designer and his/her relationship with the program. At first, program design faults were considered. A fault is identified by a wrong statement which affects the program structure in the same way as a defect affects a mechanical part.

Numerous good practice guidelines were developed. Each firm set standards, which were then extended to different industry fields such as automotive, nuclear, and aerospace. These programming rules constrain the programming language use. They aim to prevent the introduction of faults during the design phase. They facilitate the expression and the understanding of the programs. For instance, the way a variable is named was specified. For example, the name « Flight_Altitude » is more comprehensible than « X ». This prevents misunderstandings. However, these guidelines did not reconsider the features of the programming languages but only the way they should be used. Some features were excluded (e.g. “goto” statement) because they were seen to be hazardous.

2.3 Socio-technical approach

During the third phase, the operational statements of the language were extended to provide features allowing faults to be prevented or making their detection easier.

At first, paradigms of the approach used by the designer were expressed as features of the language. They avoid the gap between the cognitive model and its formulation as a program. The first example of such a feature is the subprogram, used to express a problem as composed of sub-problems. The distinction between the input and output concepts which is a basic design paradigm was introduced during the 1980s as subprogram parameter attributes (language Ada (1983)).

Fault detection was made easier by introducing data typing as a language feature. For instance, car speed or fuel tank volume which are two different design concepts, can be expressed as variables of two different types rather than using the predefined type “integer”. Therefore, the assignment of a speed by an expression formulating a volume is detected as wrong. Later on, the standard ISO 15942 (2000) evaluated the impact of the use of Ada features on the ease of fault detection for various detection techniques. Firstly, this highlights the fact that the designer can not only prevent faults (cf. previous guidelines) but also can influence fault diagnosis efficiency. Secondly, this shows that the language features used to program intrinsically impact the efficiency of the use of the fault detection techniques. Even if these appreciations of language features are now available, they are considered by only a few firms to influence the selection of language features, and the way they have to be used.

Guidelines were proposed about the use of these new features of the language (SPC, 1997). These guidelines are fundamental as the new features have no operational contribution: correct programs may be developed without the use of these features. However, their use increases the program correctness likelihood.

These features are software implementations of the concept of safety margins conventionally used for other technologies. They allow redundant elements to be introduced in the program. They are useful if human errors occur. They avoid an error of the designer leading to a fault in the software application by preventing its occurrence or facilitating its detection. So, they increase the software application's safety. Unfortunately, a lot of designers consider that these redundant statements decrease development performance as well as the speed of application execution. This assertion is wrong. Indeed, most of the new features which increase safety have no impact on the executable application generated from the code of the program. For instance, by specifying two different types for the two variables "car speed" and "tank volume", assignment inconsistencies are detected in the program, whereas these two variables will be both implemented by integers in the executable code. On the contrary, defining these two variables as integers in the program, the assignment of a speed by a volume will be considered as correct. The fault will be maintained in the program.

These features allow the requirements to be expressed by the designer in addition to his/her solution of them. Of course, only the solution of the problem is useful in operation, that is, the statements which produce the required behaviour. However, the expression of the requirements allows the consistency of the solution to be checked (Motet, 1996). Some tools propose the extension of conventional languages by introducing a formal annotation language. These annotations are comments on the programs. This illustrates the fact that they have no operational objective as comments do not produce executable code. However, they do permit the consistency between the requirements and the program statements to be checked during the test phase (e.g. ANNA (Luckham, 1987)) or by static analysis (e.g. SPARK (Chapman, 2000)).

The comparison between requirements (what is the problem?) and the proposed solution (how is it solved?) is fundamental for software technologies. Indeed, these technologies do not propose fault models comparable to those that exist for hardware technologies: for instance, a break in an electronic connection, a crack in a mechanical piece. Faults in programs are systemic and their types are infinite. For this reason, authors define software faults as adjudged causes of software application failures (Laprie, 1990). Therefore, the only way to detect an incorrect design is to show that the behaviour expected by designer is not the one formulated and performed by the program (Leveson, 1995).

2.4 Organisational approach

Numerous people are involved in the development of complex software applications. For instance, more than 300 people take part in the design of a flight manager. Therefore, an organization must be put in place, defining the role of the various people in a structured development process. This large number of participants causes new types of faults associated with the collective activity. These faults have to be prevented or detected and removed with specific approaches.

The development of complex programs requires the sharing or the reuse of subprograms created by other persons during the current development or the previous ones. Design activity leads to the contribution of new pieces of information but also the loss of old ones which are not useful for the implementation. However, these last pieces of information can be indispensable to other persons for avoiding or detecting faults. This characteristic of the development is at the origin of the failure of the first Ariane 5 launch. The automatic pilot of the launcher used the rocket attitude (relative position in relation to the vertical) computed

using gyroscopic data. The subprogram, designed for Ariane 4, was operational for many years and was reused for Ariane 5. However, the realization in Ariane 4 made an assumption concerning the launcher acceleration: it should be less than a given maximum value. The specification of this constraint, not useful for the implementation, was absent in the documentation and, in particular, in the program. The extensive previous use of this subprogram was considered to justify not doing additional tests. Unfortunately, the acceleration of Ariane 5 was greater than the maximum value and an operational error occurred during the first flight. Two other human design errors, also due to the lack of consideration of human organization to support the design, prevented the first error from being correctly handled. This led to the swivelling of the launcher and its explosion. (More details are provided in an annex of Motet [1996]).

Programming languages therefore came to provide features taking the human organization of development into account. For instance, *packages* put together subprograms relative to a common functionality (for example, to manage a graphical display) or to a given component (for example, to control an engine). These features aim at preventing faults. Other features make fault detection easier. For instance, a precondition expresses a hypothesis which must be valid to authorise a subprogram execution. A post-condition formulates a property on the results provided. If the language provides features to express such conditions, the notion of contract between various designers can be expressed in a program as assumptions and assertions (Meyer, 2000).

Finally, features such as the exception mechanisms were added to languages to detect and to handle errors occurring at run-time. They allow the developer to specify how violations of a contract should be treated. These features existed in the language used to develop the software application embedded in Ariane 5. However, as they were not mandatory and as their contribution were not understood, they were not used effectively. Additionally, the idea of raising an exception signalling the occurrence of an unhandled error at run-time was criticized. Such criticisms were comparable to opinions considering that the thermometer signalling the temperature is at the origin of the illness and therefore should not be used. This example shows that, even if features improving safety are available, their use must be encouraged and explained by guidelines.

The preservation of information introduced at design time, even if it is not useful in operation, introduces redundant data also helpful to check the global consistency of programs developed by parallel activities done by numerous teams. Again, they constitute safety margins to prevent or to detect organizational errors.

2.5 Toward a static organization of the elements of information

The previous sections showed that the progress of languages leads to the introduction of features aiming at preventing the faults or making their detection easier when they are due to the designer (cf. section 2.3) or to the organization (cf. section 2.4).

In spite of the difficulties mentioned and thanks to guidelines, these new features are being increasingly used. They facilitate the management of software complexity, which is one of the causes of faults introduced by the individuals or the organizations. This better control increases development performance reducing the number of faults and therefore the cost of

detecting their presence, diagnosing their location and correcting them. Thus, these features improve development performance as well as the safety of the designed programs.

This convergence of two requirements often considered to be contradictory, is due to the fact that these features aim at structuring the operational elements of information (data, treatments, relationships) of the programs as a static organization. For instance, operational statements receive explicit names (cf. section 2.2), they are structured as subprograms (cf. section 2.3) organized into packages (cf. section 2.4). This static organization of the information facilitates the localization of the handled pieces of information. Thus, it prevents faults or makes their detection and correction easier. Unfortunately, this static organization was shattered when new object-oriented technologies were used.

3. Object-Oriented Technologies: performance vs. safety?

3.1 Object-Oriented Technologies

Object-Oriented Technologies were proposed in the 1980's (Meyer, 1998). They introduce a new design paradigm: inheritance. As previously, objects agglomerate their own data and processing code. However, they also inherit information from other objects which are their "parents". The left part of figure 1 shows an example of such an organization. Object³ A possesses a program fragment (called *method*) *m()* and a data element (called *attribute*) *x*. Object B, son of A⁴, defines a method *n()*. It also possesses the method *m()* coming from its parent A. The inheritance mechanism also impacts the object C which offers 3 methods: two are locally defined (*p()* and *m()* [whose behaviour is redefined from the method *m()* defined originally in A]), whereas *n()* is inherited from B but not redefined. This approach increases development performance. The right part of figure 1 illustrates these explanations.

³ To simplify the presentation we will use the term « object » whereas the object-oriented languages are based on « classes » whose objects are instances.

⁴ The figures use the conventional object-oriented notation: the arrows point from offspring to parents.

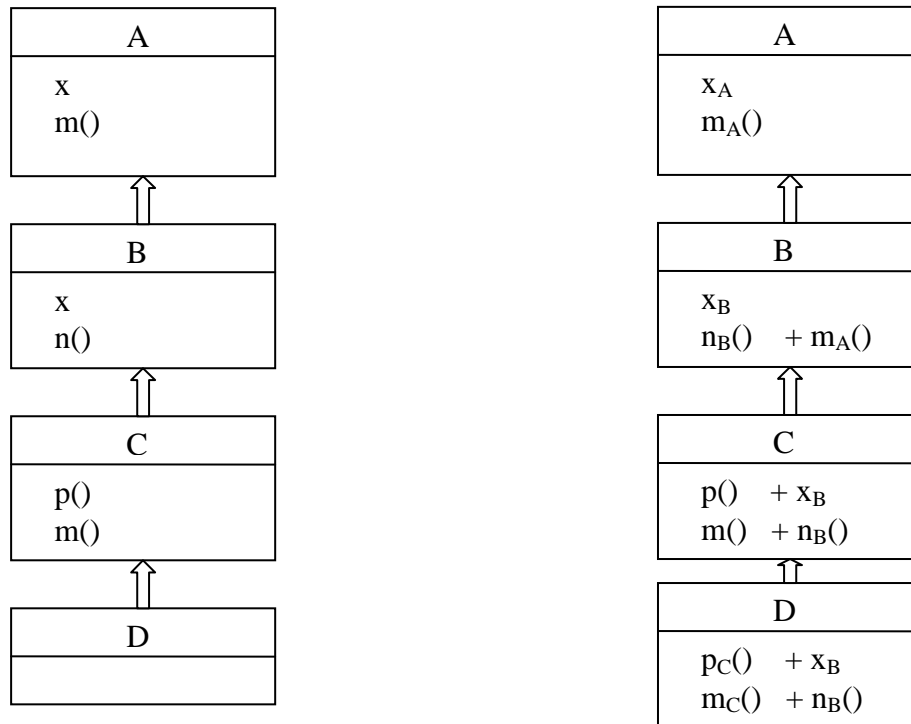


Figure 1. Object-oriented design example

Thanks to this inheritance mechanism, the elements of information can be capitalised upon. Inheritance also allows the specialization of the semantics of the elements of information. Thus, abstract objects can be defined. Then more specialized objects are derived whilst preserving elements of information from the abstract objects. For example, a generic object defining the control of a car engine (ancestor object) can be specialized in a descendant, depending on the characteristics of the engine (e.g., the number of cylinders) and of the car (e.g. the presence of an air conditioner). This new object can also be specialized taking into account more specific characteristics such as emissions regulations in various countries. Thus, several descendants are defined. The increase in the development performance is, for instance, shown by a study commissioned by Hugues Space and Communication (Port, 1999).

Due to the inheritance mechanism, the elements of information provided by an object (for instance, the services described by methods) are not necessarily described in the object. They can be inherited. In figure 1, method $n()$ defined in B is also offered by C and D. However, a piece of information provided by an object with a given semantics (a given meaning) can be available in a descendant object with another semantic due to the *redefinition* mechanism. For instance, in figure 1, method $m()$ defined in A is available in B. It is also available in C and D. However, this method provides another behaviour due to its redefinition in C. So, two definitions of one method $m()$ exist: one in A and B and another in C and D.

The inheritance mechanism also concerns the data supported by attributes. For instance, the attribute x of A is propagated to B where it is redefined. As methods use attributes to process treatments, knowledge of the actual behaviour of methods may be hard. For instance, we assume that $m()$ uses x . These two pieces of information are specified in A. So, $m_A()$ uses x_A . The redefinition of $m()$ in C uses the attribute x redefined in B. So, $m_C()$ uses x_B . Thus, the static organization of the elements of information in the structure of a program shatters. A structured organization still exists (objects assembled with inheritance relationships).

However, the availability of the elements of information is governed by inheritance propagation rules. The designer or the designers' teams are faced with new difficulties to control the available information and, in particular, to locate them in the structure. Moreover, the addition of an element of information has local but also global effects. For instance, if a method $p()$ is added to A in figure 1, it will be available in A and B but will be substituted in C and D by the method defined in C. The redefinition of methods reduces the knowledge of the actual behaviour provided by an inherited method in a given object. Inheritances and redefinitions are at the origin of new types of faults which have to be controlled to develop critical systems. Let us remember that an industrial program contains from several millions to a billion elements of information in a structure (statements, data, etc.). So, we can imagine the complex effects of their diffusion in an object-oriented program. The richness and the complexity of such an organization are similar to those offered by Internet. Most web sites propose elements of information inherited from other ones using redirections.

The Object-Oriented Technology in Aviation group (OOTiA, 2004) defines the problems. It provides a list of faults intrinsic to Object-Oriented technology and shows the impact of this new technology on verification techniques. However, the first reaction to this trade-off between the development performance offered by the object-oriented languages and the safety of the application consisted in reducing the diffusion capabilities by the limitation of the number of parents (only one), the length of the chain of inheritances and the number of methods per object. These technical solutions are not based on a deep understanding of the issues defined. Moreover, their use does not guarantee that a given safety level is actually reached. An understanding of the problem is essential to establish an appropriate trade-off between development performance and safety assurance.

Object-oriented technologies introduce a break which cannot be handled by adapting recipes coming from the conventional technologies. This break is similar to the one occurring during the 1980s and 1990s. During this period, the size of programs grew from thousands to millions of lines. Leveson showed that this first break required the control of new issues (Leveson, 1995). She compared them to the ones occurring when high pressure technology was used to design the steamships sailing on the Mississippi river. The adaptation of the methods used to design the low pressure engines was not efficient and was at the origin of numerous accidents (Leveson, 1994). The current approaches used to prevent software faults cannot be adapted to object-oriented technologies. The safety problems have to be reconsidered and to do this they must first be stated.

The second part of the paper aims at specifying the problem. In order to establish an appropriate trade-off between development performance and safety assurance, measurements of these two criteria must be provided. First, the effect of the information propagation on performance development and on safety assurance has to be understood. The paper focuses on this last aspect: why are propagation mechanisms at the origin of faults and how can their impacts on program safety be measured?

3.2 New safety measurement approach

A program is a structure which organizes pieces of information: treatments (e.g., methods) and data (e.g., attributes). For instance, statements and data are included in subprograms which can be nested or embedded in modules (also called packages or units). Software application safety in the past was increased thanks to features leading to a static structuring of these pieces of information. Their localizations in the structure were made easier for the

designers. Statements which reduce this localisation were excluded by guidelines (e.g., global variables, multiple exits of loop and subprograms). This good knowledge of the organization of the information facilitates the understanding of the program, thus avoiding the introduction of faults or making their detection easier. On the contrary, this static structuring reduces the efficiency of the elements of information. They are just defined locally.

The object-oriented approach introduces a dissemination mechanism of the pieces of information in the structure (inheritance mechanism). The localization of the available information in the structure is therefore more difficult. For instance, the method `n()` available in object D on figure 1 is coming from its definition in B. This definition is located 2 levels higher. This distance reduces the understanding and increases the risk of design faults. Moreover, the designer does not distinctly know the exhaustive list of the pieces of information available in any element of the structure. This requires a global view of the program which is difficult for complex industrial applications.

This diffusion phenomenon can be compared with thermodynamic systems. A system composed of isolated parts reduces the heat exchange. The system behaviour is static. Its thermal energy is inefficient as it does not produce any work. This is similar to conventional programs: pieces of information are confined in elements of the structure. On the contrary, the thermal diffusion between parts of the system makes its energy useful thanks to its dynamic evolution. The information diffusion due to the inheritance mechanism also reinforces the usefulness of the pieces of information. The thermodynamic system efficiency is conventionally assessed by the system *entropy*. The concept of entropy was applied by Shannon to information theory (Shannon, 1948) to measure the relevance of a piece of information. We use it to assess the difficulty of accessing a piece of information. It is then extended to measure the difficulty of understanding a program.

3.3 Estimation of the safety performance

Let us consider two design models of one system composed of 3 entities (A, B and C). The first model comes from an object-oriented design approach. It is described on the left part of figure 2. 'A' defines two methods `m()` and `o()`. B inherits from A and defines `n()`. C inherits from B and defines `p()`. The right part of figure 2 presents the same entities providing the same methods, coming from a conventional design approach. Due to the absence of the inheritance mechanism, the definitions of the methods are repeated.

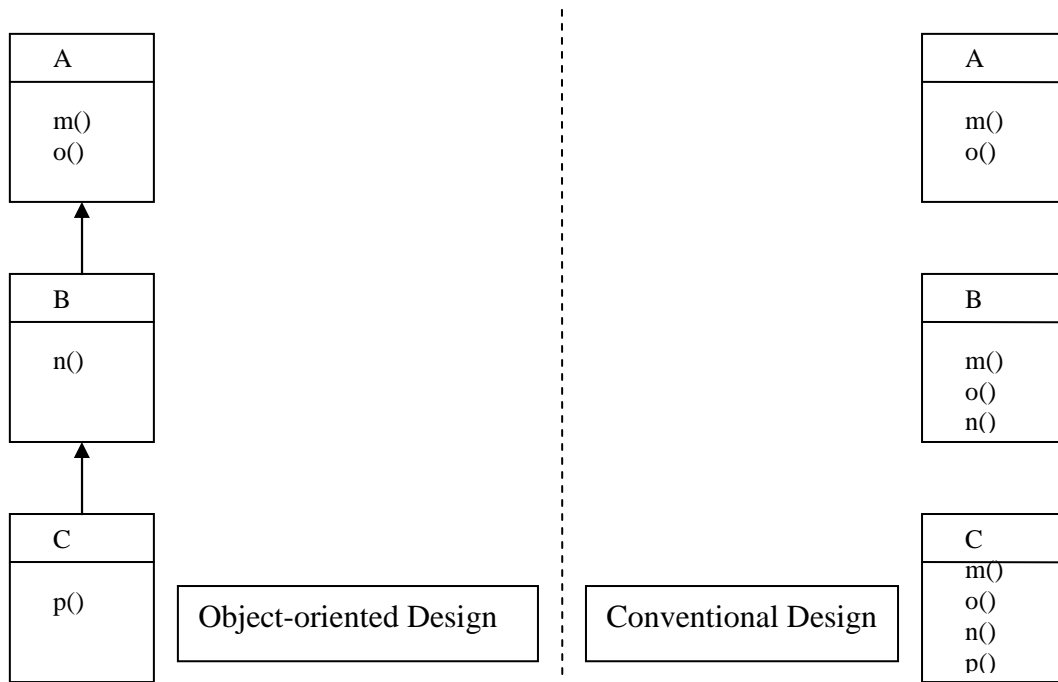


Figure 2. Object-oriented and conventional designs of an application.

The design model described on the right part of figure 2 makes very easy the identification of the methods provided by each entity (A, B or C). This knowledge of the available pieces of information reduces the probability of faults when the methods are designed or used. However, such a design model is voluminous (repeated information), which reduces the design efficiency.

In order to assess the risk of faults, figure 3 represents the identification process for each method provided by C. On the right part (conventional design), the process is direct: the methods $m_C()$, $o_C()$, $n_C()$ and $p_C()$ specified in C are immediately identified. Note that we give the label $m_C()$ to denote the method $m()$ accessible from C as this one is distinguished from the method $m()$ of B ($m_B()$) and from that of A ($m_A()$). Indeed these methods are duplicated, that is, defined again.

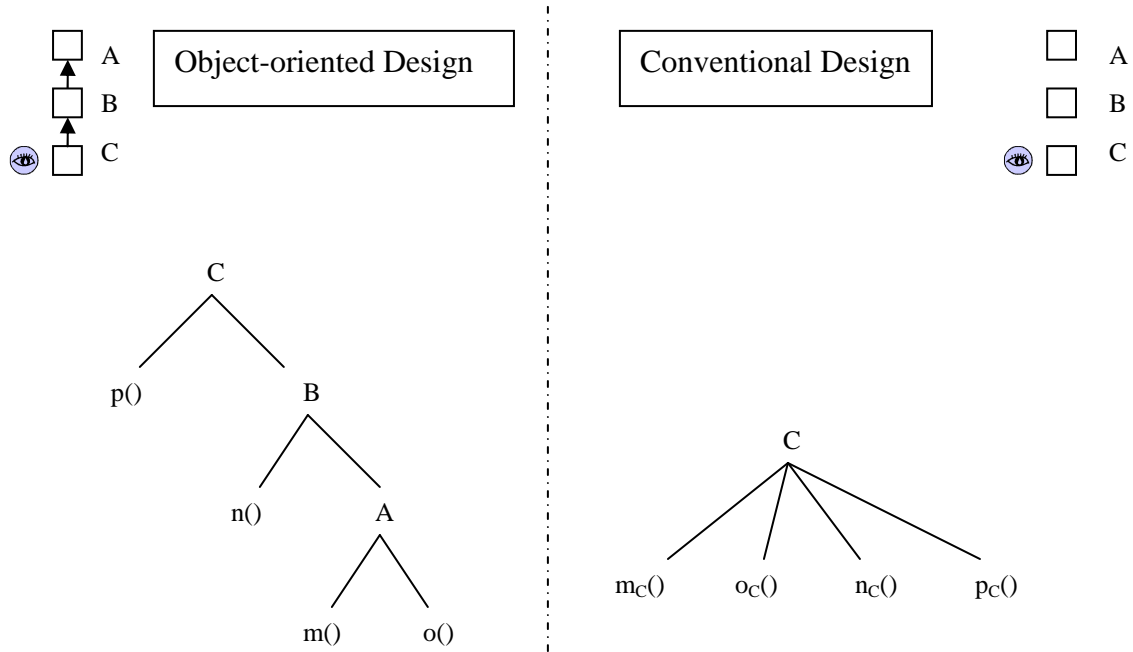


Figure 3. Identification of a piece of information

In the object-oriented design model, the process for identifying the methods available in C is more complex (left part of figure 3). C contains p() locally specified and also the methods inherited from the parent B. So, it is necessary to look at B to identify these. These methods are composed of n() specified in B and of the methods inherited from A. Looking at the contents of A, m() and o() are identified.

Information theory establishes that the difficulty of distinguishing a piece of information ‘e’ in set ‘E’ containing n elements is $\log_2(n)$ (Shannon, 1948).

$$I_E(e) = \log_2(n) \text{ with } |E| = n.$$

So, the difficulty to distinguish m_C in C in the conventional design model (right part of figure 3) is

$$I_C(m) = \log_2(4) = 2 \text{ (result 1).}$$

For the object-oriented design model (left part of figure 3), to identify m() in C, it is necessary to establish that it is inherited from B which must be distinguished from p(). As C is expressed as containing p() and the elements available in B, the difficulty $I_C(B)$ to distinguish B from p() is $\log_2(2)$ (2 pieces of information in C). In the same way, it is necessary to deduce that in B, m() comes from A distinguished from n(). So, the difficulty $I_B(A)$ is assigned by $\log_2(2)$. Finally, in A, m() must be distinguished from o(). So, the difficulty $I_A(m)$ is also assigned with $\log_2(2)$. To conclude, the difficulty to identify m() in C in the object-oriented design model is:

$$I_C(m) = I_C(B) + I_B(A) + I_A(m) = \log_2(2) + \log_2(2) + \log_2(2) = 1 + 1 + 1 = 3 \text{ (result 2).}$$

The results 1 and 2 show that the difficulty in identifying $m()$ in C is greater for the object-oriented design ($I_C(m) = 3$) than for the conventional design ($I_C(m) = 2$). This conclusion seems to be obvious. However, it is now quantified and will be used to define the trade-off between design performance and application safety.

Figure 4 synthesizes the assessments $I_C(y)$ of the difficulties in detecting in C each element 'y' in the object-oriented design model (left part) and in the conventional design model (right part).

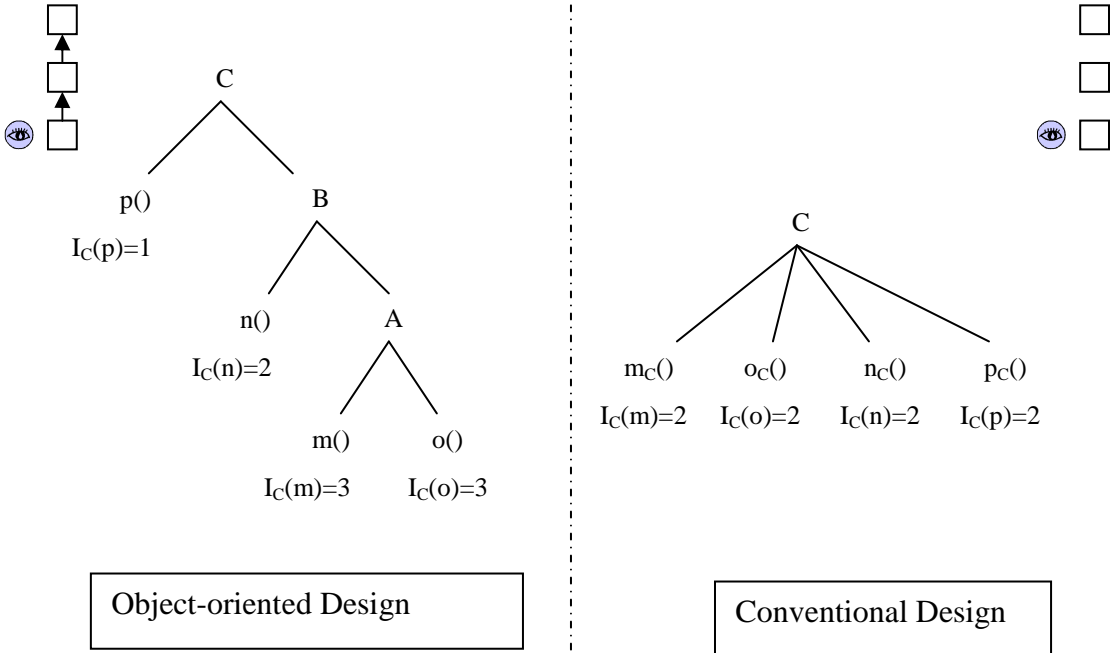


Figure 4. Difficulty in identifying each element from C.

The difficulty in detecting any element from C in the object-oriented design model is assessed as: $E_C(OOD) = (3+3+2+1) / 4 = 2.25$

The difficulty in detecting any element from C in the conventional design model is assessed as: $E_C(CD) = (2+2+2+2) / 4 = 2$

The same calculus is done for each class and an average value defines the global measurement. This metrics is called MESS (Metrics based on Entropy for Structured Sets). For our example, the two following results are obtained:

- MESS(CD) = 1.64 for the conventional design, and
- MESS(OOD) = 1.78 for the Object-oriented design

Assuming that the likelihood of the presence of faults is proportional to the identification difficulty and that the safety is inversely proportional to this difficulty, we notice that the safety of the conventional design model ($E(CD) = 1.64$) is somewhat better than those of the object-oriented design model ($E(OOD) = 1.78$).

The following section will show that this tendency can be reversed when the information use frequency is taken into account.

Gaudan (2008) evaluates the efficiency of MESS as a metric compared with other Object-Oriented metrics. MESS was applied to an Object-Oriented program of a module of a flight manager. This prototype contains 50 classes and 236 definitions of methods. It also inherits from 12 classes of the API Java J2SE. This experiment aims at appreciating the pertinence of the proposed metrics, that is, to determine if it is a good predictor of the presence of faults in Object-Oriented programs.

We compared the result of our metric (MESS) applied to each class with the risk of faults provided by 2 industrial tools (Findbugs and Eclipse TPPT platform) on the same classes. Figure 5 synthesizes the results. For each class, a dot is represented. Its abscissa is the value provided by the tools and the ordinate is the MESS value.

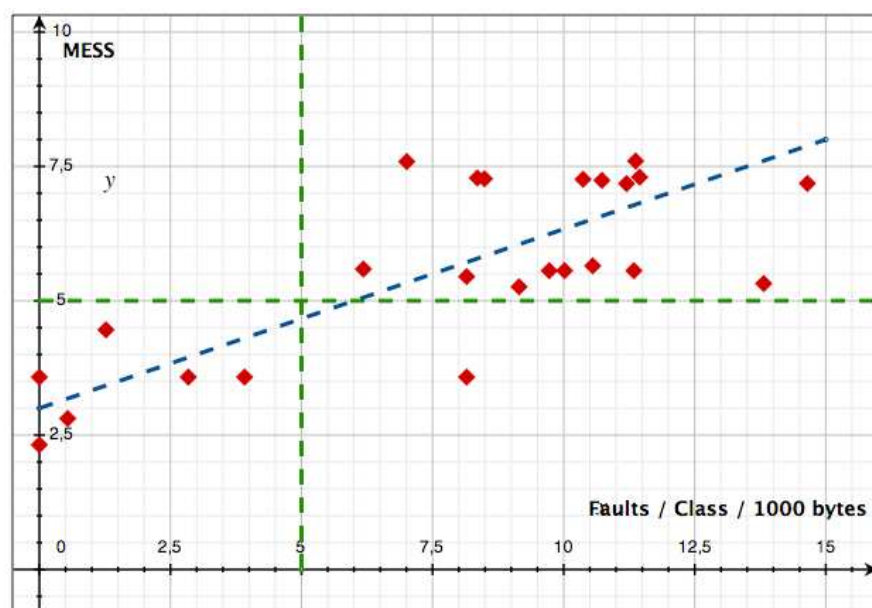


Figure 5. Estimation of the efficiency of MESS

The dotted line on the figure highlights a correlation. This means that MESS provides a comparable assessment to that given by the two tools. However, these tools only handle a predefined list of faults. As this list is certainly not exhaustive, numerous types of faults are obviously not considered. On the contrary, MESS aims at assessing the understanding of the program by the designer. A bad understanding may lead to many different kinds of fault.

Figure 5 also suggests the existence of a threshold. When MESS is greater than 5, the risk of fault is high. Contrary to guidelines proposed (e.g., volume 3 of (OOTiA, 2004)), MESS < 5 specifies an objective (fault risk level) which can be achieved by various architectures of the application (classes, inheritance relationships, distributions of methods in classes, etc.). Certification authorities as well as standards could therefore be defined, not in terms of design constraints but in terms of the objectives of the control of the risk to be achieved.

3.4 Estimation of development performance and trade-off

In the 2 design models analysed, the number of structural elements is the same (A, B and C). Moreover, these structural elements provide the same pieces of information. However, 4 methods are specified in the object-oriented design model whereas 9 methods have to be

expressed in the conventional design model. The number of the specified elements, for instance, the number of methods in the example, is first used to assess the design performance. So, the object-oriented approach is more efficient.

Integrating the 2 measures to trade off the 2 requirements (performance and safety) also requires considering the rate of the use of the pieces of information provided by the model (methods in the examples). Indeed, even if it contains a fault, a piece of information which is specified but not used will not lead to a program failure. To illustrate this factor, consider two organizations of 4 files (m, n, o, p). The first one (conventional) places the 4 files in one directory C. The second one, the object-oriented approach, consists of a directory C containing one file p and one sub-directory B. B contains one file n and a directory A. This last one includes 2 files m and o. These two organizations are the same as the ones illustrated by figure 2. The average rate evaluation indicates that the conventional organization is more efficient to assess any information. This calculus implicitly assumes that the pieces of information are accessed with the same probability. However, if the file p is more often accessed (9 times as opposed to once for the others), then the previous conclusion will be reversed as shown below.

The probability of use is therefore an important criterion which must be taken into account to assess the ease of identifying the pieces of information and then to estimate the reliability of the designed program. It characterizes the performance of the design model considering the usefulness of the introduced pieces of information.

The use rate has no impact on the ease of identification of the elements of the conventional design model. This assertion is false for an object-oriented design model. In particular, the object-oriented approach may be safer than the conventional approach as shown below.

Figure 6 provides an example of specifying the numbers of uses of methods in the object-oriented design model. These values are underlined. For instance, method m() of A is called five times, of B twice and of C once. To compare the two design models, figure 7 reuses the same data for the conventional design model.

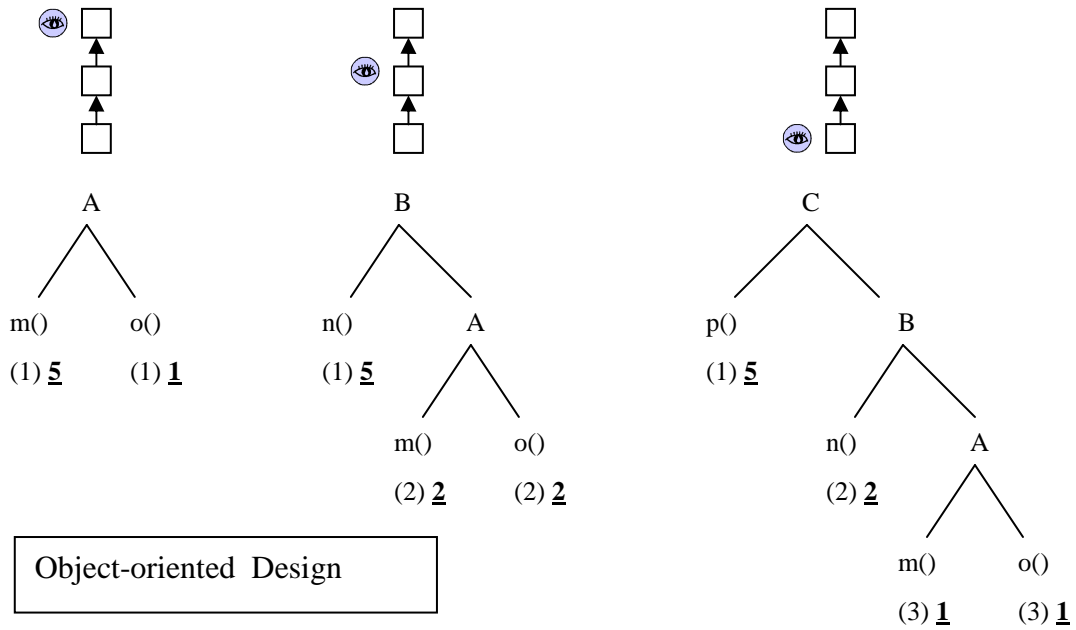


Figure 6. Use of elements of the object-oriented design model

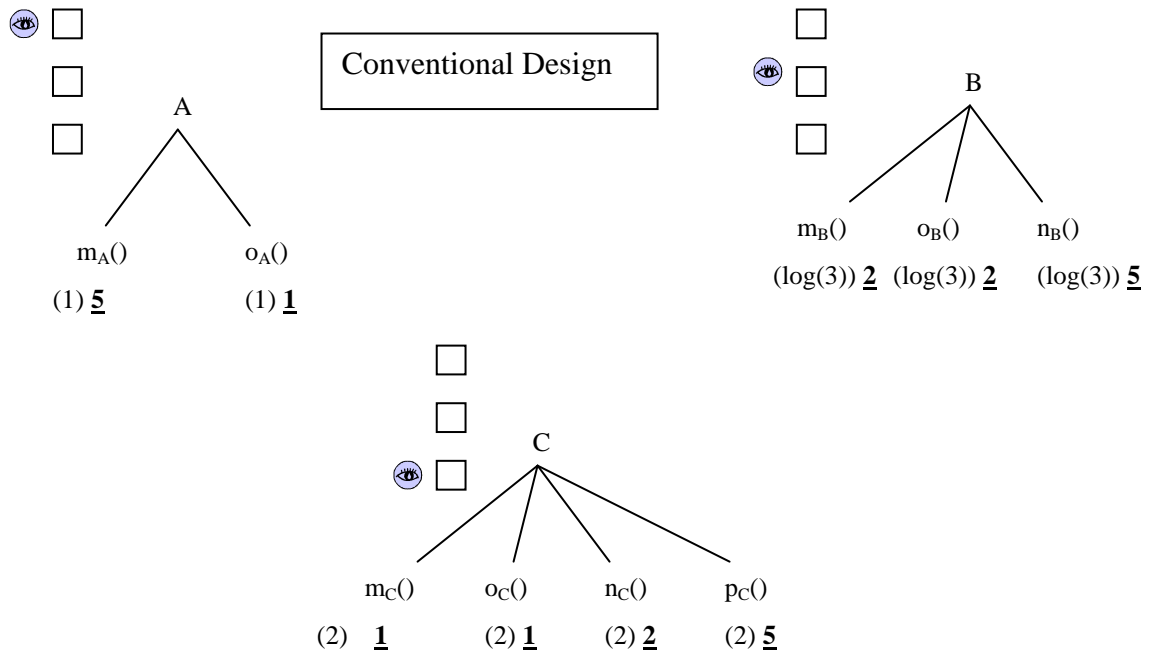


Figure 7. Use of elements of the conventional design model

The ease of operation (i.e., integrating the actual use) to identify the pieces of information (methods) is therefore the average of the ease $I_X(y)$ if identification of each piece of information 'y' in X, weighted with the number of uses of this piece of information.

$$E = (5 \times I_A(m) + 1 \times I_A(o) +$$

$$\frac{2 \times I_B(m) + 2 \times I_B(o) + 5 \times I_B(n) + 1 \times I_C(m) + 1 \times I_C(o) + 2 \times I_C(m) + 5 \times I_C(p)}{24}$$

The division by 24 comes from the total number of uses of the methods.

This formula is valid for both designs. However, the actual values of $I_X(y)$ depend on the design choices. For the object-oriented design model, using the values synthesized in figure 6, we obtain: $E(OOD) = 1.42$. For the conventional design model, using the values synthesized in figure 7, we obtain: $E(CD) = 1.63$.

We deduce that operational safety, that is, safety taking actual use into account, is better for the object-oriented model (lower difficulty in identifying an element) than for the conventional one.

4. Conclusion

In the past software technologies progressed by introducing new features in their languages that simultaneously increased the development performance and the safety of the applications developed. This result was obtained by constraining the organization of the elements of information provided in a program at static locations in the program structure.

The object-oriented approach provokes a break in this trend. It adds the inheritance mechanism, allowing the diffusion of elements of information into the structure. Thus, an element of information defined in a component of the structure is also available in other structural components without any additional specification. On one hand, this property increases development performance. On the other hand, it makes more difficult the identification of the pieces of information actually provided by the structural elements, potentially leading to more design faults. However, we have showed that object-oriented technologies are not intrinsically more dangerous. However, their use requires the definition of methods of trade-off between the two criteria: safety and development performance.

We then introduced the contribution of Shannon's work based on entropy to estimate the difficulty in identifying a piece of information in a model or program and thus to assess the reliability of the software applications. Various design models providing the same functionalities can be proposed and then evaluated. These estimations facilitate the trade-off between various designs integrating development performance and application reliability. We noted the importance of the rate of use of the elements of information in such estimations.

This work will lead to the derivation of good practices whose actual effect on the reliability can be assessed. The proposed guidelines will certainly be more complex than those currently available, such as "no more than 6 levels of inheritance" because the proposed measurement considers the structure as well as the distribution of the pieces of information and their use together. However, the proposed measurement will allow the actual contribution of the guidelines to safety to be assessed and their use justified.

References

Ada, 1983, Reference Manual for the Ada Programming Language, American National Standards Institute Ansi/Mil-Std-1815A.

Barnes J., 2003 High Integrity Software. The Spark Approach to Safety and Security, Addison-Wesley.

Benedicktsson O., 2000 Safety Critical Software and Development Productivity, Proceedings of the 2nd World Congress on Software Quality.

Benedicktsson O., Hunter R.B., McGettrick A.D., 2001 Processes for Software Critical Systems, Software Process: Improvement and Practice, vol. 6, Issue 1, 47-62.

CEPA, 2003, Toyota Agrees to Pay \$7.9 Million Settlement, California Environmental Protection Agency, News Release, Air resources Board, 2 March.

Chapman R., 2000 Industrial Experience with SPARK, Ada Letters, Volume XX , Issue 4, ACM, 64-68.

Cousot P., 2002 Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation, Theoretical Computer Science, Vol. 2771-2, 47-103.

de Neufville R., 1994 The Baggage System at Denver: Prospects and Lessons, Journal of Air Transport Management, Vol. 1, No. 4, 229-236.

DO-178B, 1992 Software Considerations in Airborne Systems and Equipment certification, DO-178B & ED-12B, RTCA. Inc. & EUROCAE, December.

Elims M., 2004 On wheels, Nuts and Software, Proceedings of the 9th Australian Workshop on Safety Critical Systems, Australian Computer Society, ACM, 67-76.

EN-50128, 2001 Railway Applications. Communications, Signalling and Processing Systems. Software for Railway Control and Protection Systems.

Gaudan S., Motet G., Auriol G. 2008 , Metrics for Object-Oriented Software Reliability Assessment – Application to a Flight Manager, Proceedings of the 7th European Dependable Computing Conference (EDCC) Kaunas, Lithuania, IEEE, 2008.

Geffroy J-C. & Motet G., 2002 Design of Dependable Computing Systems, Kluwer Academic Publishers.

IEC-61508, 1998 Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems, International Electrotechnical Commission (IEC).

ISO-15942, 2000 Guide for the Use of the Ada Programming Language in High Integrity Systems, ISO/IEC TR 15942 International Standards Organization, 2000.

Laprie J.-C. (editor), 1990 Dependability: Basic Concepts and Terminology, Springer-Verlag.

Leveson N.G., 1994 High-Pressure Steam Engines and Computer Software, Software, Vol. 27, N° 10, IEEE, 65-73.

Leveson N.G., 1995 Safeware. System Safety and Computers, Addison-Wesley Publishing, 1995.

Luckham D., 1987 ANNA: a Language for Annotating Ada Programs, Lecture Notes in Computer Science, Vol. 260, Springer.

McDermid J., 2001 Software Safety : Where's the Evidence ? , Proceedings of the Sixth Australian Workshop on Industrial Experience with Safety Critical Systems and Software, Vol. 3, ACM, 1-6.

McDermid J. & Kelly T., 2006 Software in Safety Critical Systems : Achievement and Prediction, Nuclear Future, Vol. 2, N° 3, 140-145.

Motet G., Marpinard A., & Geffroy J-C., 1996 Design of Dependable Ada Software, Prentice Hall, 1996.

Meyer B., 1998 Object-Oriented Software Construction, Prentice Hall.

Meyer B., 2000 Design by Contract, Prentice Hall, 2000.

Millward D., 2008 Heathrow Engulfed by Baggage Chaos, Telegraph, 26 Feb.

Myers G.J, 1979., The Art of Software Testing, Wiley.

OOTiA, 2004 Handbook for Object-Oriented Technology in Aviation, Object-Oriented Technology in Aviation group, <http://shemesh.larc.nasa.gov/foot>.

Port D. & McArthur M. 1999 A Study of Productivity and Efficiency for Object-Oriented Methods and Languages, Proceedings of the Sixth Asia Pacific Software Engineering Conference, IEEE.

SPC, 1997 Ada 95 Quality and Style, Software Productivity Consortium, Springer.

TNO, 2005 TNO & IDATE, Software Intensive Systems in the Future, Information Technology for European Advancement (ITEA) European programme, Final report.

Shannon C.E., 1948 A Mathematical Theory of Computation, Bell System Technical Journal, vol. 27.

USDJ, 1999 U.S. Sues TOYOTA For Clean Air Act Violation. Claims 2.2 Million Cars Have Illegal Emission Control Monitoring Systems, US Department of Justice, Washington, DC, 12 July.