Identification Model of the Object Oriented Technology risks, for Avionics Certification

S Gaudan, G Motet, E Jenn, S Leriche

Identification Model of the Object Oriented Technology risks, for Avionics Certification

S. Gaudan[1,2], G. Motet[2], E. Jenn[1], S. Leriche[1]

1: Thales Avionics, 105 avenue du Général Eisenhower, 31036 Toulouse cedex 1, France
2: LESIA-INSA Toulouse, 135 avenue de Rangueil, 31077 Toulouse cedex 4, France

Abstract

The introduction of any new technology in an existing industrial process has a dual effect: on the one hand, it is expected to bring some well-identified benefits, but on the other hand, it also brings a certain number of new risks. It is the role and responsibility of decision makers, designers, and developers to estimate and balance these two aspects, taking into account the very specific characteristics of their industrial domain. In the domain of software systems, for instance, object-oriented technologies have been demonstrated to increase software quality and productivity, but they simultaneously bring some specific risks that must be carefully characterized and handled, especially when they are integrated in the development of software applications for critical systems.
In the avionics domain, the OOTiA document proposes a first and informal identification of some of these risks. However this identification process misses the formal background that would guarantee its consistency and completeness.
In this paper, we suggest to fill this gap by considering the management of software risk as a specific case of industrial risk management. To achieve this goal, we propose a generic model for the identification of software risks. This model provides the information required by the subsequent phases of risk management: risk estimation, risk acceptance, and risk mitigation.

Keywords

Risk management, Object-Oriented Technology, Avionics applications, Certification.

1 Motivations

Today, systems are for a large and ever growing part controlled by software sub-systems, and these sub-systems undertake more and more complex, and more and more critical functions.

As a consequence, non-functional requirements on these sub-systems represent a growing proportion of all requirements they have to comply with. Software components must be flexible, maintainable, and reusable; they must be able to cope and scale smoothly with complexity, and they have to satisfy all safety and security objectives. Hence, suppliers of those systems are faced with a multidimensional optimisation process involving many criteria: production cost, performance, resource consumption, reliability, portability, maintainability, etc. This search for optimality is usually achieved by changing and enhancing processes, but another possibility is to change the technologies involved in these processes.

Usually, analyses of software risks focuses on functions — for example to assess the effect of a software failure on the system by means of a software FMECA —, but rarely on the technologies used for the implementation of these functions. The approach is somewhat different in other technical domains where the technology itself is the subject of risk assessment. In the mechanical domain, for instance, numerous studies are carried out to determine the lightest, the most robust, or generally speaking, the most adapted *new* material that will satisfy a given set of (non functional) requirements. Similarly, it seems important that suppliers of critical software systems benefit from the evolution of available software technologies to improve the main quality criteria of their products.

In the avionics domain, the introduction of a new technology is a long-term financial stake: a product development phase is long (around 5 years) and its life span is around 30 years or more. It is consequently essential to be innovative and to be able to foresee what will be the *next* optimal software technologies to support *future* airborne functions. For example, flight management systems, maintenance systems, or data base management systems of future aircrafts will perform more complex functions, but they will have to be more profitable, easier to develop, less expensive, more portable, more reliable, etc.

Java seems definitely to be one of these "new" technologies, as illustrated by the number of studies already realised — and currently going on — concerning its use in the context of critical embedded systems. Java actually shows many interesting features that makes it a *potential* good candidate for the replacement of current programming languages: true object-orientation, simplicity, portability, richness of the application programming interface and toolset, and *last but not least*, pop-

ularity. These features and their corresponding underlying concepts clearly improve various quality factors. However, they also introduce new potential risks (e.g., new design fault patterns, etc.) that shall be estimated and compared to potential benefits.

Furthermore, in the avionics domain, introducing these new technologies raises specific compliance demonstration problems with the standard recommendations for airborne software. These recommendations, expressed in the DO-178B standard — established more than ten years ago when object-oriented technology was not even considered in airborne systems — must be reinterpreted in the Java context: how are traceability requirements declined in the presence of polymorphism?, what is a consistent code reuse in the presence of inheritance? how is test coverage estimated?, etc.

In this context, the avionics community (aircraft manufacturers, equipment suppliers, research laboratories, certification authorities, etc.) has elaborated a document listing some of the issues raised by the introduction of Object Oriented Technologies (OOT) in aviation, and giving some solutions to address them (OOTiA, ref. [8]). This document expresses the main concerns of avionics domain experts on the compliance of OOT with DO-178B objectives, and more generally with safety requirements at the origin of this objectives. It is worth noting that the OOTiA is an input document of the on-going DO-178 update process (towards DO-178C), which is expected to be completed by the end of 2008. So, is it likely that OOTiA's recommendations will eventually become new requirements for avionics software developers.

The current study is an effort towards a systematic and exhaustive analysis of issues raised by OOT in general and Java in particular. It will eventually serve as a means to demonstrate the ability of Java to support the delivery of services that can justifiably be trusted.

2 Framework of the Study

2.1 The Need for a Methodological Framework

Neither the current DO-178B standard nor the OOTiA document propose any methodological framework to address OOT related problems in a systematic way. Therefore, our objective is to elaborate such a framework to complete the OOTiA's intuitive and informal list of issues on the basis of the well-founded risk management concepts expressed in ISO's Guide 73 [2] and other standards [1, 3].

2.2 Risk Management Process

The Risk Management process adopted in numerous industrial domains (e.g., chemistry, pharmacy, nuclear energy, etc.) is based on four tasks:

1. Risk analysis, i.e., *risk identification* and *risk estimation*
2. Risk evaluation, i.e., *hierarchical organisation of the risks by establishing classes of risks*
3. Risk acceptance: *definition of a threshold of acceptability of the risk in the hierarchy established previously (political choice)*
4. Risk treatment: *means of reduction of the risk to make it acceptable, to reduce its estimation*

This paper is focused on the first phase of the first task, i.e., the risks identification.

2.3 Identification of sources

In the OOTiA document, problems are identified in a list that describes briefly all issues, but very few information is given on how this identification process has been carried out. To complete this analysis, guarantee its completeness, and find adequate mitigation means, the first phase consists in identifying the *sources* of these risks. To reach this goal, we propose a model that gives a formal framework for the risks and sources identification process. This model is presented in the next section. Later, in Section 4, we illustrate its application on one particular issue raised by the OOTiA. We conclude by introducing the perspectives and the way for the following tasks to be led to pursue this process of Object-Oriented Technology Risk Management.

3 The Identification Model

3.1 Introduction

A formal identification of risks sources requires an abstract *identification model* which will be instantiated for any specific risks. Currently, the elaboration of such a model is the core of numerous discussions concerning various industrial domains, notably under the aegis of the French AFNOR (Association Française pour la NORmalisation) and ISO (International Standardization Organization). In this context, we consider that we may benefit from results obtained in these different domains to obtain a generic, rich, and stable model.

In what follows, we first propose a model resulting from numerous discussions and confrontations with situations coming from various industrial domains. This model constitutes a proposal for the workgroups that elaborate and update the standards quoted previously. Then, we will show that this model instantiates perfectly for risks associated with software technologies.

In the sequel of the document, the identification model is illustrated by a simple example in which a building located in a mountainous area might be destroyed by an avalanche.

3.2 General concepts

As illustrated on the example given in Figure 1, the identification model is broken down in four components.

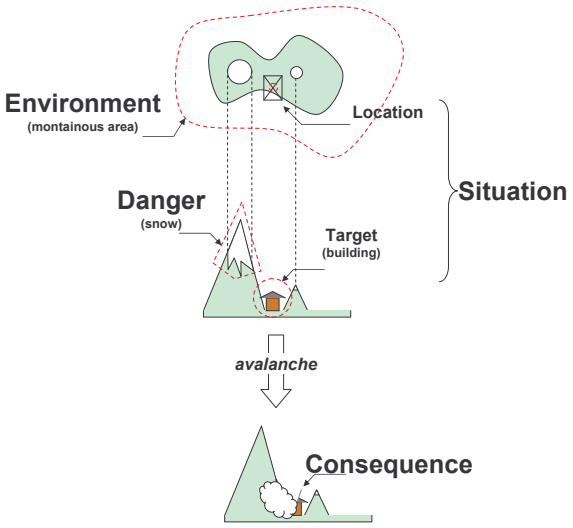

Figure 1: The identification model phases

- The **danger**. In the example, the danger is related to the presence of snow at the top of a mountain. The mass of snow represents a large (potential) energy that might be transferred to another object (e.g., a cottage, a skier, etc.), and lead to its destruction.

- The **environment of the danger**. In the example, the environment is the geographic area possibly touched by the avalanche. The area may be further decomposed in several sub-areas corresponding to various exposition levels to the danger. For instance, a cottage located on the mountainside is more likely to be damaged than a cottage more distant from the mountain.

- The **dangerous situation**. In the example, the dangerous situation is the presence of a building in a mountainous area. It is worth noting that this situation is the result of the *decision* — or risk-taking — to build the cottage in this environment.

- The **consequence** of the danger. In the example, a possible consequence of the danger is the deterioration of the building.

Those four components are detailed in the following sections.

3.3 The danger

A **danger** is the element responsible for the negative effects associated with a risk. It can be split up into four main elements:

- The **dangerous phenomenon** is a physical or abstract characteristic of an entity (the *actor*) that can be "transferred" to another entity (the *target*). In our previous example, the dangerous phenomenon is the potential energy of the snow. This energy can be first transformed into a kinetic energy during the avalanche, and then transferred by a shock to the building.

- The **actor** of the danger is the element that owns the physical or abstract characteristic that determines the *dangerous phenomenon*. For instance, the snow is the actor because it *owns* some potential energy due to its mass and relative height.

- The **dangerous property** is the actor's characteristic that determines the dangerous phenomenon. In our example, the dangerous properties of the snow are its mass and height, because this mass and this height are at the origin of the potential energy.

- The **dangerous event** (or initiator event) represents the activation of the danger. In our example, the snow at the top of the mountain cannot lead to any damage as long as it doesn't move. The event that activates the danger is the uncoupling of the snow that makes it move, "transforming" its potential energy into kinetic energy.

3.4 The environment of the danger

The **environment** of the danger represents the elements that have an influence on the propagation of the dangerous phenomenon.

In our example, the environment covers all geographic elements (characterized by their height, slope, presence of trees, etc.) that have an influence on the trajectory, velocity, and energy transfer characteristics of the snow mass.

3.5 The dangerous situation

The danger and its environment are not sufficient to determine the presence of a risk. For a risk to occur, a target must be placed in a **dangerous situation** characterised by two elements:

- the **target**, which is the entity susceptible to be affected by the effects of the danger. In the example, the target is the building.

- the **location** of the target in the environment of the danger. In our example, the exposition to the danger increases as the building gets closer to the mountain base.

3.6 The consequences

The **consequences** of a danger with respect to a given dangerous situation are modelled by two elements:

- the **harmful interaction** (or harmful event) is the interaction between the dangerous phenomenon and the target in the dangerous situation. In our example, it corresponds to the shock between the moving mass of snow and the building.
- the **damage** represents the negative effects of the harmful interaction on the target. In our example, the damages are the more or less severe deteriorations of the building.

4 An example in Java

In the next sections, the identification model is now instantiated in the software domain and illustrated on one particular issue identified in the OOTiA document.

4.1 OOTiA issue presentation

The identification model presented in the previous sections is applied on a problem first identified by Offutt *et al.* (ref. [7]) in a paper dedicated to the design faults related to sub-typing and polymorphism. This issue, called the "State Definition Anomaly" (SDA), was later integrated to the list of issues given in the OOTiA.

The risk associated with this example arises from the following constraint defining the correct use of inheritance (subtyping): "*any property ϕ satisfied by a type T, must be satisfied by every subtype of T*". This principle was first stated by Liskov *et al* in [5]; it is now known as the Liskov Substitutability Principle, or LSP.

In practice, this principle induces the following constraint on the redefinition of methods (overriding) in an inheritance hierarchy: *if a method `m` satisfies a contract C* (see [6]) *when applied on any object t of type T, all overriding methods of `m` in subtypes of T must satisfy* at least *the same contract*, i.e.:

- pre-conditions of the overriding method must be as strict as or less strict than those of the overridden method, and
- post-conditions of the overriding method must be as strict as or stricter than those of the overridden method. For example, if the overridden method has to return a result in a particular domain $[a, b]$, the overriding method has to return a value in a domain $[a', b']$ such that $[a', b'] \subseteq [a, b]$.

This principle also concerns other aspects of a method specification such as the invariants, the history conditions, the frame conditions, etc.

For example, if a method initialises a class attribute, any overriding method has to perform an initialisation so that any valid sequence of statements applied on an object of the parent class remains valid when applied on any object of any subclass. Indeed, a violation of this rule can lead to disastrous system behaviour. For example, an attribute not correctly initialised and used by a subclass object may generate a run-time error.

This problem is one particular case of the general "State Definition Anomaly" (Offutt *et al*), defined as follows: "the refining methods implemented in the descendant must leave the ancestor in a state that is equivalent to the state that the ancestor's overriden method would have left the ancestor in" ([7]).

According to our identification model, the SDA can be roughly modelled as follows:

- The *danger* is due to inheritance.
- The *environment* of the danger is the program, because it is through its structure that the potentially harmful effects of the inheritance are propagated.
- The *dangerous situation* corresponds to the definition of an overriding method (the target) in a given class (the location).
- The *consequences* represent the design fault (here, a contract violation).

Figure 2 gives the initial structure of a program potentially subject to a SDA.

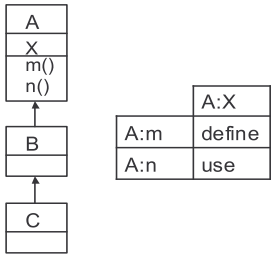

Figure 2: Initial program structure

Methods `m()` and `n()` defined in class `A` (noted `A:m()` and `A:n()`), interact with the attribute `X` (noted `A:X`). Method `A:n()` has a pre-condition `X` $\in [4, 20]$, that requires an explicit initialisation of attribute `X` before method `A:n()` is invoked (the default value is not accepted). Method `A:m()` defines the attribute `A:X` in a way consistent with the pre-condition of method `A:n()`. So, the following sequence of statements is valid:

```
main() {
 ...
  a = f(); // return an object from the
           class A, class B or class C.
  a.m();
  a.n();
 ...
}
```

If the programmer wants to override the method `m()` in the class `C`, he is facing the risk of SDA. Indeed, if he overrides `A:m()` in class `C`, and if `C:m()` does not verify the contract of interaction with the class attributes (i.e., `m` doesn't initialize attribute `A:X`), an inconsistent sequence of instructions can occur (see Figure 3).

Execution of the following sequence of instructions

```
a.m();
a.n();
```

on an object `a` of type `C`, actually calls the methods:

```
a.C:m();
a.A:n()19
```

where `A:n()` uses the attribute `A:X` which is not correctly initialized previously by `C:m()`.

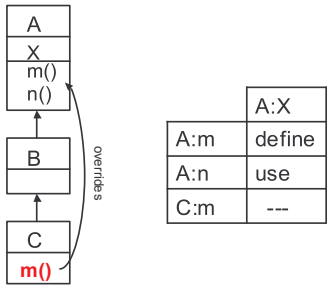

Figure 3: Program after overriding `m()`

Now, let's give the detailed identification model for the SDA risk, and identify the sources of this risk.

4.2 The danger

Inheritance is at the origin of numerous design faults, the SDA being *one* typical example. So, in what follows, inheritance is essentially considered as a "dangerous feature", as emphasis is placed on its potential negative effects. However, this mechanism also shows many advantages, and numerous positive effects (or benefits) are expected in terms of productivity, safety, etc. The balance between risks and benefits is considered during the risk evaluation and risk acceptance phases (cf. section 2.2), which are not considered any further in the scope of this paper.

Now, let's decline the risk identification model in the context of software technologies, and illustrate its application to the SDA case.

- The *actor* of the danger is a particular design or programming language. In the SDA example, the actor is any object-oriented language supporting inheritance, such as Java, for example.
- The *dangerous property* is a specific feature of the actor (here, a language), such as a particular programming paradigm, a particular construct, etc. In our example, inheritance is the dangerous property because it determines new and possibly complex propagation of information (data and control transfer) between various software components.
- The *dangerous phenomenon* is the propagation of information as determined/allowed by the dangerous property. In our example, the transfer of information concerns the propagation of state definition contracts through the inheritance hierarchy.
- The *harmful event* is the usage of the language feature — in our example: inheritance — carrying the dangerous property in a given program.

4.3 The environment

The *environment* of a particular software risk is the program in which the dangerous feature is used. The structure of this program (in particular, the class inheritance structure, the method number and complexity, etc.) is one element of the environment that may influence the propagation of a given dangerous phenomenon.

In the SDA example, the inheritance tree (notably the long chains of inheritances and the numbers of inherited methods) is an essential component of the risk environment, since it determines the propagation of information concerning state definition.

4.4 The dangerous situation

The *dangerous situation* is the situation where a programmer modifies a program that uses a given dangerous property. This modification exposes a

- *target* method or class in
- a particular *location* in the program to the dangerous phenomenon (corresponding to the dangerous property).

In our example, the dangerous situation is the one where the programmer overrides method `A:m()` in a subclass `C` of `A`. The target is the overriding method `C:m()`, and the location where the target is exposed to the danger is class `C`.

4.5 The consequences

The *consequences* of a risk are defined by two aspects:

- The *harmful interaction* represents the occurrence of a conflict between the information transferred by the dangerous phenomenon *(inherited contracts)* and the information carried by the target *(local contract)*. This conflict leads to an inconsistency between the different informations. In our SDA example, the harmful interaction corresponds to the violation of the contract of method `A:m()` inherited by its overriding method `C:m()`. More precisely, it corresponds to the fact that `C:m()` does not initialise `A:X`, whereas `A:m()` does it.
- The *damage* represents the provision of an erroneous contract which will eventually lead to an erroneous code. In our example, the erroneous contract translates to the following erronous sequence where `A:n()` is called although `A:X` is not initialized:

```
a.m();
a.n();
```

It is worth noting that the damages severity could be defined as the number of invocations of the faulty method, or also as the difficulty to detect the design fault.

5 Conclusion

In this document, we have proposed an identification model of the risks associated with a programming or design language.

This model takes into account

- the dangerous language constructs (danger),
- the program structure (environment),
- the types and locations of the modifications performed on the program and the programmer expertise (dangerous situations).
- the problems generated by the modification (consequences)

The identification of sources represents the first stage of the overall risk management process. The next phase consists in quantifying this risk. Indeed, controlling a risk requires the identification of the dangerous property *and* the estimation of its potential effects.

For this reason, we have defined a set of estimation means on the previous model for each of the factors contributing to a given risk:

- Means to estimate the *magnitude* of the dangerous phenomenon,
- Means to estimate the *occurrence level* of a dangerous property and consequently its influence on the programmer,
- Means to estimate the *exposition level* of the target to the risk,
- Means to estimate the *severity* of the damages.

These estimation means allow designers to accept or refuse risks on the basis of factual and quantified information.

For a given software risk, a specific metric is associated with each estimation means. In the SDA example, the magnitude of the dangerous phenomenon is measured by a function depending on (i) the number of inheritance levels, (ii) the number of inherited methods, and (iii) the complexity of the methods' contracts.

Acceptability thresholds are then defined based on these metrics. We think that such thresholds should be part of the recommendations of future avionics standards concerning OOT, the choice of a particular threshold depending on the software application's criticality.

If a risk is considered non acceptable (i.e., the acceptability threshold is reached), it shall be mitigated. In the software domain, a possible technique consists in applying fault prevention techniques to lower the magnitude, exposition, etc. A classical technique is the provision for coding rules. An example of such a coding rule is given in the OOTiA document where the authors suggest to apply the "Six Deep Rule" that recommends the depth of the inheritance tree to be lower than 6. Our methodological framework supports the elaboration of such recommendations, but on a firm and sound basis.

Our current activity leads to perform a systematic analysis of the issues identified in the OOTiA, according to our methodological framework. The next phase will then consist in completing this study by an analysis focused on the specific risks introduced by the Java language [4].

References

[1] AS/NZS 4360. Risk management. Standards Australia, 2004.

[2] ISO/IEC Guide 73. Risk management, vocabulary, guidelines for use in standards. International Organization for Standardization, 2002.

[3] AFNOR. Management du risque - lignes directrices pour l'estimation des risques. Norme 50-252, 2005.

[4] Roger T. Alexander, James M. Bieman, and John Viega. Coping with java programming stress. *IEEE Computer*, 33(4):30–38, 2000.

[5] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[6] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

[7] J. Offutt, R. Alexander, Y. Wu, Q. Xiao, and C. Hutchinson. A fault model for subtype inheritance and polymorphism. In PRC, editor, *In Twelfth IEEE International Symposium on Software Reliability Engineering (ISSRE'01)*, pages pages 84–95. IEEE, November 2001.

[8] OOTiA. Handbook for object-oriented technology in aviation, Octobre 2004.

6 Glossary

FMECA: Failure Modes, Effects and Criticality Analysis

ISO: International Standard Organisation

OOT: Object Oriented Technology

OOTiA: Object Oriented Technology in Aviation

SDA: State Definition Anomaly