

Fragility of Evolving Software

Kim Mens

ICTEAM, Université catholique de Louvain, Louvain-la-Neuve, Belgium

Software systems are fragile with respect to evolution. They consist of many software artefacts that make implicit assumptions about one another. When such artefacts get replaced by newer versions, some of these assumptions may get invalidated, thus causing subtle evolution conflicts. We refer to this phenomenon as *fragility of evolving software*.

A particular instance of fragility in class-based object-oriented programming languages is the *fragile base class problem* [5, 7]. It occurs when a base class (a class from which other classes are derived through inheritance) gets replaced by a newer version. When the derived classes make certain assumptions about the base class that the evolved base class no longer provides, this can cause subtle conflicts. E.g., suppose that a base class B implements a method m in terms of an auxiliary method n . Now suppose that a derived class D of B overrides the implementation of n , with the intention not only of adapting the behaviour of n , but also that of m which is defined in terms of n . Independently, however, the base class B evolves into a newer version B' where m no longer depends on n for performance reasons. After this evolution, D 's assumption that m depends on n is no longer valid, and D 's overridden implementation of n no longer affects m , thus causing an unexpected behavioural conflict.

Another instance of fragility is the *fragile pointcut problem* in aspect-oriented programming [2, 6]. It may occur upon evolution of an aspect-oriented program, which contains pointcuts expressed over a base program. Pointcut expressions express execution points, called join points, in the base program where the aspects need to be applied. Pointcuts are fragile, because the base program is oblivious of their existence. When the base program evolves, this may

have unexpected effects on the pointcuts expressed over that base program (such as accidental captures or misses of join points). *Advice fragility* is a related problem caused by the obliviousness of aspect code with respect to base code [1]. It arises when the advice code (i.e., the aspect code that will be woven with the base code at the join points) is too tightly coupled to the base code. This may cause problems when the base code evolves in such a way that the advice code no longer fits with that base code.

In general, fragility problems arise when the assumptions made by a software artefact about another artefact get invalidated upon evolution of that other artefact. In a sense, those assumptions, which are often not documented explicitly, constitute a kind of implicit contract to be respected upon software evolution. Solutions to the fragility problem therefore typically involve providing a means to define such an *evolution contract* explicitly, detecting possible breaches of that contract upon evolution, classifying the possible conflicts that may arise when the contract gets breached, and proposing appropriate solution strategies for each of those possible kinds of conflicts.

For example, as a solution to the fragile base class problem exemplified above, *reuse contracts* [7] document the so-called *specialisation interface* [3] of the base class, i.e. its internal calling dependencies, as well as how the derived class depends on the base class. In our previous example, the reuse contract would document, amongst others, that in base class B method m calls n , and that the derived class D ‘refines’ B by overriding the implementation of n . Furthermore, an evolution operator would describe that B' ‘coarsens’ B by removing an internal calling dependency (method m in B' no longer calls n). A two-dimensional classification of possible *reuse con-*

flits can then be made in terms of the evolution operators and derivation dependencies. E.g., the above conflict where the base class coarsens a method m by removing its dependency on n , whereas the dependent class overrides n , would be flagged as an ‘inconsistent method’ [7]. Based on this classification, for each type of conflict¹ a corresponding solution strategy can be proposed. E.g., for the inconsistent method above either the method m in B' should keep its dependency on n or, alternatively, D should be replaced by a D' that overrides not only n but also m .

More recently, *usage contracts* [4] were proposed as an alternative mechanism to solve some of the problems caused by base class fragility. Usage contracts explicitly document the expected structural regularities that a base class wants its dependent classes to conform to. These regularities can be verified automatically when implementing or modifying the dependent classes, flagging potential conflicts immediately so that they can be corrected as soon as they arise.

In the case of the fragile pointcut problem, pointcuts are fragile w.r.t. evolution of the base code. A possible solution [2] consists in making the pointcuts more robust by declaring them in terms of an intermediate pointcut model, rather than directly in terms of the base code. This model forms a kind of evolution contract between the base code and the pointcut expressions, by making explicit some of the assumptions the pointcuts make about the base code. Furthermore, it enables detecting, after evolving the base code, whether the base code still satisfies the pointcut model. If it doesn’t, the mismatch between the base code and the model gives an indication of potential mismatches (accidental or missed captures) the pointcuts have with respect to the base code, and appropriate solutions can be proposed (by adapting either the base code or the pointcuts that refer to it).

The examples above illustrate only some instances of fragility problems and their solutions. Our claim is that the problem of fragility applies in general to any kind of adaptable systems, and so do the proposed solution approaches. The problem always amounts

to a lack of documentation on the implicit assumptions between a base entity and its dependent entities, and the solution always involves documenting these assumptions more explicitly in a kind of evolution contract, and verifying upon evolution whether the contract remains respected. By classifying the types of possible conflicts (depending on the kind of evolution and the kind of dependency) and their corresponding solution strategies, appropriate solutions to these conflicts can be proposed.

In the near future we will investigate the problem of fragility in the area of dynamic software evolution, and context-oriented programming in particular. In fact, this example will be worked out in more detail in a separate contribution submitted to this BENEVOL2013 seminar.

References

- [1] A. Cymment, N. Kicillof, R. Altman, and F. Astesauain. Improving AOP systems’ evolvability by decoupling advices from base code. In *RAM-SE’06 workshop at ECOOP’06*, pages 9–21, 2006.
- [2] A. Kellens, K. Mens, J. Brichau, and K. Gybels. Managing the evolution of aspect-oriented software with model-based pointcuts. In *Proceedings of ECOOP’06*, pages 501–525. Springer-Verlag, 2006.
- [3] J. Lamping. Typing the specialization interface. In *Proceedings of OOPSLA ’93*, pages 201–214. ACM Press, 1993.
- [4] A. Lozano, A. Kellens, and K. Mens. Usage contracts: Offering immediate feedback on violations of structural source-code regularities. Under revision.
- [5] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In *Proceedings of ECOOP’98*, pages 355–382. Springer-Verlag, 1998.
- [6] C. Noguera, A. Kellens, D. Deridder, and T. D’Hondt. Tackling pointcut fragility with dynamic annotations. In *RAM-SE’10 workshop at ECOOP’10*, pages 1–6. ACM Press, 2010.
- [7] P. Steyaert, C. Lucas, K. Mens, and T. D’Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proceedings of OOPSLA’96*, pages 268–285. ACM Press, 1996.

¹Other possible conflicts are conflicting method interfaces, unimplemented methods, or accidental method captures. [7]