Mohamed E. Fayad and Adam Altman

An Introduction to Software Stability

Studying the design and development that produces stable (or unstable) software.

There is little doubt the field of software engineering, like all other engineering fields, has helped make life what it is today. With software controlling more equipment, software engineering is becoming an integral part of our lives. However, unlike many other engineering fields, the products produced through software engineering are largely intangible. Also, unlike the products of other engineering fields, software products are unlikely to remain stable over a long period of time.

This column is the first in a series offering insight into the central themes of software stability. In this series we examine the unique characteristics of software that distinguish it from other engineering fields. We also study the artifacts of analysis, design, development, and other factors that tend to produce stable or unstable software products.

In hardware areas, failure rates of products often start high, drop low, and then go high again. Early in a hardware product's lifecycle, there are some problems with the system. As these problems are fixed, the failure rate drops. However, as hardware gets old, physical deterioration causes it to fail. In other words, the hardware wears out and the failure rate rises again.

Software, on the other hand, is not subject to hardware's same wear and tear. There are no environmental factors that cause software to break. Software is a set of instructions, or a recipe, for a piece of hardware to follow. There are no moving parts in software; nothing can physically deteriorate. Software should not wear out. Unfortunately, it does. Countless authors in the field of software engineering have identified this problem. However, the software engineering techniques outlined by many software-engineering authors have not achieved an adequate amount of stability in software projects.

This problem is more than just an inconvenience for software engineers and users. The reengineering required for these software products does not come without a price. It is common to hear of reengineering projects costing hundreds of thousands, to millions of dollars. This does not take into account the time wasted by continual reengineering.

Software defects and deterioration are caused by changes in software. Many of these changes cannot be avoided. They can, however, be minimized. Currently, when a change is made to a software program, most of the time the entire program is reengineered. It doesn't matter if the change is due to new technology or a change in clientele. This reengineering process is ridiculous. If the core purpose of the software product has not changed, why, then, must the entire project be reengineered to incorporate a change?

The concepts of "enduring business themes" (EBTs) and "business objects" (BOs) have been introduced as a proposed solution to this problem. The idea in this case is to identify aspects of the environment in which the software will operate that will not change and cater the software to these areas. The majority of the engineering done on a software project should be done to fit the project to those areas remaining stable. This yields a stable core design and, thus, a stable software product.

MARTIN MAYO

Changes introduced to the software project will then be in the periphery, since the core was based on something that remains and will remain stable. Since whatever changes that must be made to the software in the future will be in this periphery, it will only be these small external modules that will be engineered. And, thus, we avoid the endless circle of reengineering entire software projects for minor changes [2, 3].

It has been suggested that EBTs should be modeled and implemented as functions [1]. This, however, does not yield much extra software stability. When a modern software system changes, it is the classes in the model that change. Changes in these classes ripple through the system, causing other classes and relationships to be reengineered. It does not matter if one function within a class remains constant; this does not guarantee changes in a class would not cause a ripple effect throughout the rest of the system. Therefore, EBTs and BOs should be modeled as objects forming the core of software systems, not as functions.

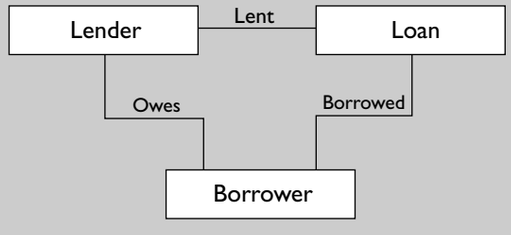
Case Study I—The Loan

Consider the following example: One year ago, Bob experienced some financial difficulty. In a moment of desperation, he asked his best friend, Eric, for a loan. Eric lent him $1,000. A year's time passed, and Bob did not repay the loan. Eric would like the money back.

We proposed this scenario to several teams in software engineering classes to analyze using classical problem analysis techniques as well as OMT or UML notation. Team responses ranged from "Take Bob to court," to "Break Bob's legs." All the suggestions were solutions; none had to do with problem analysis.

The teams identified three classes in the problem statement: the borrower (Bob), the lender (Eric), and the loan. Unfortunately, this is an incorrect analysis of the problem (see Figure 1). It does not matter how these classes are arranged in the object model. The entirety of the problem cannot be analyzed and cannot be accurately modeled using only these three classes. Using only these classes, one cannot identify any differences between this model and a model of a loan between a bank and a customer or a loan shark and his client.

Figure 1. A flawed classical analysis of the loan program.



What is missing from this model? To perform an accurate analysis of this problem, several other factors must be taken into consideration. First, the concept of friendship must be inserted into the analysis to distinguish this problem from other loan problems.

Using the EBTs of friendship and finance, this problem becomes slightly more complicated, but the problem is modeled far more accurately. Therefore, it becomes far easier to find a correct solution to the problem. Since the solution will be correct in the beginning, costly changes to the software will be avoided.

However, there are still many factors missing from this model. There is more to using EBTs and BOs than simply identifying concepts such as "friendship" and "finance," and objects such as "friend," and throwing them into an object model. The use of EBTs and BOs requires an entirely new way of thinking. Model entities must now be thought of in terms of stability and conceptuality. This new paradigm in software engineering requires vast changes in our thinking. This model can no longer be confused with other models for loans between different entities, but it is still not thorough enough to fully analyze the problem and arrive at a stable solution.

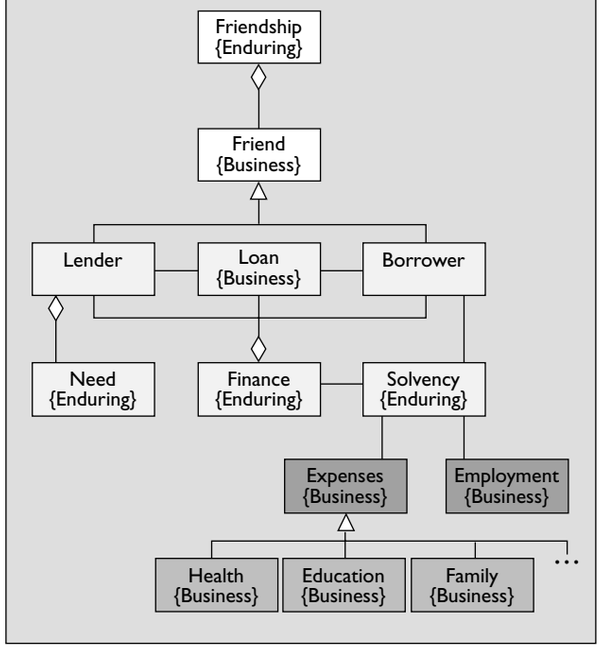
What else is missing from this model? Note there is nothing in the model concerning why Eric wants the loan paid back. Nor is there anything in the model documenting why Bob will not or cannot pay back the loan. Does Eric have a pressing need for the money? Is Bob capable of repaying the loan? Is Bob employed? Do Bob's current expenses prevent him from repaying the money? In the past, has Bob demonstrated he is trustworthy and responsible?

By answering these questions, one can easily model the problem at hand and arrive at an appropriate solution. Now a payment schedule considering both Eric's need for the reconciliation of the loan and Bob's financial situation can be determined.

Notice this new object model contains several objects not explicitly mentioned in the problem statement. In this model, BOs are represented as objects with the "business" stereotype, and EBTs are represented as objects with the "enduring" stereotype (see Figure 2). Take note of the EBTs and BOs being used in this model. The EBTs are finance, friendship, need, and solvency. Finance and friendship are required themes since this model is a loan between friends. The friendship object contains these attributes and operations relating to the friendship between the two parties involved.

If it were possible to actually rate a friendship or give it a computable value, the rating would be this object's responsibility. The finance theme is an aggregate of the loan and the parties involved, and has the responsibility of reconciling the loan based on the borrower's solvency. Solvency is calculated based on the borrower's expenses and employment, of which are BOs. The borrower's solvency combined with the lender's need should be used to create a schedule for repaying the debt. All of these objects are categorized as EBTs because they are intangible themes that remain both internally and externally stable throughout the life of the problem.

Figure 2. A thorough analysis of the loan problem using EBTs and Business Objects.



There are several BOs identified in this model. All BOs are labeled as such since they are partially tangible and will remain externally stable throughout the life of the problem. They might, however, change internally. For example, take the family BO. Although family dynamics may constantly change over time, to the problem, the borrower's family is always his or her family. Family members may become ill, get married, get divorced, pass away, or do other things that cause internal family processes to change. Externally, though, as far as the problem is concerned, families remain constant; the family may cause the borrower to have certain expenses. How these expenses are calculated depends on the current family dynamic.

The employment BO works similarly. Employment is the borrower's source of income. Its internal processes allow the calculation of income that factors into the borrower's solvency. Changes in the borrower's employment cause the process whereby the borrower's income is calculated and change the internal processes

of the employment object. Externally, however, employment turns into some form of income, regardless of whether it is zero or six figures. This external aspect of employment—the fact that income depends on it—remains constant for the duration of the problem. Combined, the fact that the employment object is externally stable and internally volatile make the employment object a BO.

Our research suggests accomplishing stability requires far more thorough analysis than was thought to be required in preliminary analyses of the problem—far more than a cursory analysis of the key entities or roles. These and other issues will be discussed in future columns in this series on software stability. For further examination, see case studies posted at www.cse.unl.edu/~fayad. C

References

1. Cline, M., Mike G., and Howard Y. Enduring business themes (EBTs); sidebar in *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, M. Fayad, D. Schmidt, and R. Johnson, Eds., John Wiley and Sons, New York, 1999.
2. Fayad, M. and Mauri L. *Transition to Object-Oriented Software Development*. John Wiley and Sons, New York, 1998.
3. Fayad, M. *Software Stability*. Four E-books, MightyWords, Inc., 2001

Mohamed E. Fayad (fayad@cse.unl.edu) is J.D. Edwards Professor at the University of Nebraska, Lincoln.
Adam Altman is a graduate student in computer science at Stanford University.