International Conference on Innovations
in Computer Science and Engineering (iCiCSE2019)
26 - 28 June 2019, Miri Sarawak, MALAYSIA

Linguistic Engineering

Mohamed E. Fayad, PhD

Full Professor and Entrepreneur
Computer Engineering Department
Charles W. Davidson College of Engineering, San José State University
One Washington Square, San José, CA 95192-0180
E-mail: m.fayad@sjsu.edu, https://www.amazon.com/-/e/B001IXQCFU

Half-day Tutorial Pamphlet

International Conference on Innovations in Computer Science and Engineering (iCiCSE2019), 26 - 28 June 2019, Miri Sarawak, MALAYSIA

Linguistic Engineering

Mohamed E. Fayad, PhD
Full Professor and Entrepreneur
Computer Engineering Department
Charles W. Davidson College of Engineering, San José State University
One Washington Square, San José, CA 95192-0180
E-mail: m.fayad@sjsu.edu, https://www.amazon.com/-/e/B001IXQCFU

Half-Day Tutorial

Abstract
Linguistic Engineering is a rapidly developing field of research. A firm background in language technology and linguistic engineering is extremely valuable in the context of manipulating large datasets. A linguistic engineer has knowledge of language technology as used in computer applications including search engines, all uses of language technology in computer applications, and all of the possible forms of applied linguistics. The author(s) are captivated with the unification and stability modeling of natural language from an engineering and computational perspectives, as well as the study of appropriate engineering approaches to linguistic questions.

Natural language is properly considered as "the system of all systems". As such, we examine every known concept (noun or noun phrase) is a unified and stable pattern that has functional and non-functional requirements and ultimate design. Every concept has one unique responsibility and can be uniquely defined by its functional and non-functional requirements as a unified and stable pattern. By employing this unified approach, we postulate that any concept can be defined such that the definition is adequate and complete for use in any field of knowledge.

We claim that this approach can be employed to provide an intrinsic and complete understanding of any concept and the language can be evolved based on knowledge.
Linguistics engineering is inherently interdisciplinary but most important, it can have a powerful impact on every field of human knowledge, such as all of the applications of engineering, science and academics, and specifically in the fields of software engineering, computer engineering, system engineering, artificial intelligence, law, philosophy, theology, cognitive science, social science, psychology, and government, among others.

Linguistics engineering has theoretical and applied components. Theoretical linguistics engineering focuses on issues in cognitive science, and applied linguistics engineering focuses on the practical understanding, concept modeling to put human language into use in a concise way in any field of knowledge. The authors have the vision to generate a common, unified, stable pattern language - a knowledge map - of a subject, domain, concepts to be applied in all fields of knowledge. Linguistics engineering will lead to the advancement or enhancement of the other aspects of the linguistics

International Conference on Innovations in Computer Science and Engineering (iCiCSE2019), 26 - 28 June 2019, Miri Sarawak, MALAYSIA

Linguistics Engineering Approaches and Case Studies

Tutorial Outline

[1] Software Stability Model (SSM) and the Art of Abstractions

[2] Linguistic Engineering Templates

[3] Linguistic Engineering Technologies

[4] Concept/Class Responsibility and Collaborations

[5] Analysis Stable Analysis Pattern (Case Study)

[6[Data Stable Design Pattern (Case Study)

[7] Leakage Stable Design Pattern (Case Study)

[8] Data Leakage Stable Architecture Pattern (Case Study)

[9] Stable Machine Learning Knowledge Map (Case Study)

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.

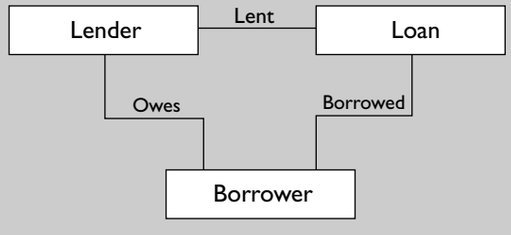
Case Study I—The Loan

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.

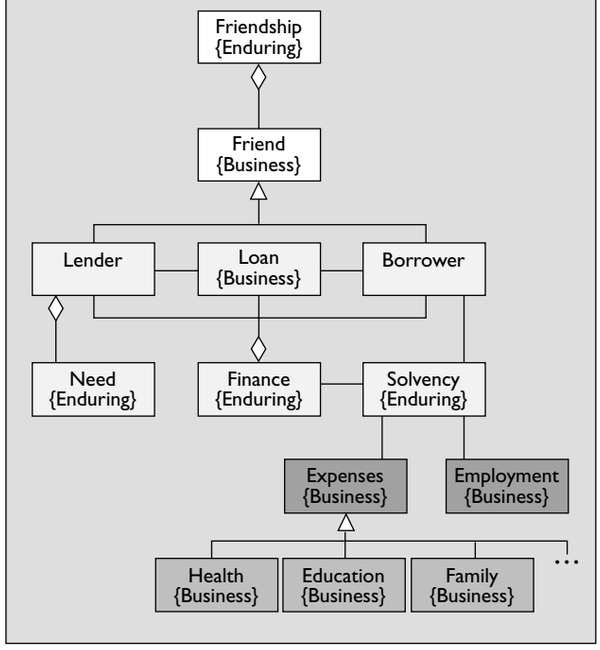
What else is missing from this model? Note there is nothing in the model concerning why Eric wants the loan paid back. Nor is there anything in the model documenting why Bob will not or cannot pay back the loan. Does Eric have a pressing need for the money? Is Bob capable of repaying the loan? Is Bob 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

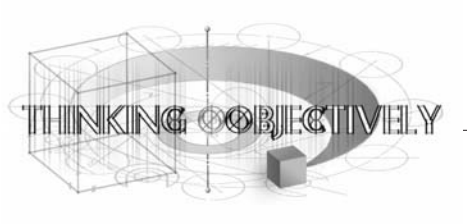
of the employment object. Externally, however, employment turns into some form of income, regardless of whether it is zero or six figures. This external aspect of employment—the fact that income depends on it—remains constant for the duration of the problem. Combined, the fact that the employment object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



Mohamed Fayad

Accomplishing Software Stability

The kitchen proves an ideal setting to illustrate how stability and change work together within software.

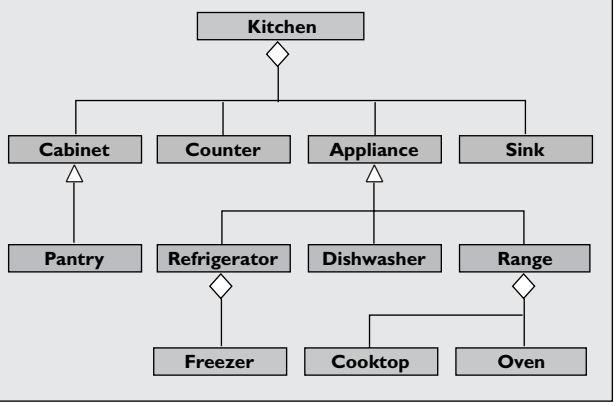
A descriptive example of the use of Enduring Business Themes (EBTs) and Business Objects is the typical kitchen. The concept of a kitchen depends on a variety of objects that do not remain constant over time. These objects can be considered "Industrial Objects."

Our kitchen model can be thought of as a tree of aggregations and generalizations. Many of the branches or leaves of the tree can change and the kitchen model breaks down. In this sense, the kitchen model is not stable over time.

Similarly, the problem with software systems today is that Industrial Objects like those found in the kitchen model (see Figure 1) are frequently observed to be within the core of design. Likewise, Industrial Objects are too often finely intertwined within the system implementation. In such systems, changes over time usually result in changes in the Industrial Objects. The results are unstable designs and an unstable code base.

If the designer properly considers those aspects of a kitchen that do not change over time the design of our kitchen is radically different. A proper design centers on enduring themes and Business Objects. Enduring themes are those concepts of the system that would remain constant over time. Externally, Business Objects also remain constant over time, but may have internal, intangible processes that change. In the case of the kitchen, the model centers on concepts such as lighting, storage and preservation, heating, and convenience. Objects such as the range and the refrigerator become secondary and are replaced easily as new technologies become available.

Figure 1. An unstable classical model of a kitchen.



To find the EBTs and Business Objects for any model, you must have enough experience in the domain of the problem to decide whether or not a concept is core to the system. Experience working in an area builds an innate feel for what is and what is not enduring. Often, this is not enough. A professional chef might be a good person to consult when trying to find the EBTs of a kitchen. A very experienced, professional chef might say, "I've always had pots, pans, a stove, an oven, a set of cabinets, and a refrigerator. These must be enduring themes of the kitchen." This chef would be wrong. The

PAUL WATSON

objects that he or she identified are, in fact, Industrial Objects—not EBTs. Despite the fact this chef has never been without these objects, these objects are nowhere near enduring enough to be considered EBTs.

To find the EBTs of a kitchen, one must distance oneself from the material items thought to be typical kitchen items. Instead, look at the "why's" of these items. Why does the kitchen have a refrigerator? Why does the kitchen have a range? Attaching objects to the answers to these questions yields the EBTs of a kitchen.

Although the model of the kitchen using these EBTs is more complex than the traditional kitchen model, it more accurately represents the essence of a kitchen and is a more stable model. Those objects that can change in the not-so-distant future are moved toward the periphery of the kitchen model while the hubs of the model are replaced with stable EBTs and Business Objects.

The EBTs of the kitchen not only include cooking, but livability, food storage, convenience, cleanliness, and cuisine. If any of these themes are changed or removed from the model, the model and the kitchen itself change dramatically. For example, if the livability theme is taken out of the model, the model is no longer a useful design for a kitchen in a home, but possibly for a kitchen in the back of a restaurant. If the cuisine changes, it may also be necessary to change some structural features of the kitchen. In other words, the kitchen is only as stable as its EBT constituents.

The cooking EBTs must be present, of course, as it is an aggregate of those tools used within a kitchen to prepare food. The cuisine theme represents the food cooked within a given kitchen. Of course, it is an aggregate of food and the recipes used to prepare the food. Food storage and preservation is an aggregation of those objects that perform the cold and dry storage duties of a kitchen. Livability is the theme that makes the kitchen part of the home. Cleanliness is something that must be maintained within a kitchen to ensure food does not become tainted with disease. Finally, convenience is what makes the kitchen easy to use and keeps the cook working in the kitchen happy.

The two Business Objects in the model are "light" and "recipe." Both are Business Objects instead of EBTs because they both contain internal processes that may change without changing their external relationship to the problem. Recipes for given cuisine, for example, can contain many different processes that result in the same dish. Light can be produced in many different ways yielding the same, comfortable, homey light.

A summary of the identification criteria for enduring business themes, business objects, and industrial objects.

	Enduring Business Themes	Business Objects	Industrial Objects
Stability Over Time	Stable over time	Externally stable	Unstable
Adaptability	Adaptable without change	Adaptable through internal change	Not necessarily adaptable
Essentiality	Essential	Essential	Replaceable
Intuition	Intuition	Intuition and reading	Reading only
Explicitness	Implicit	Implicit or explicit	Explicit
Commonality to the Domain	Core	Core	Peripheral
Tangibility	Conceptual	Semi-tangible	Tangible

Identification Criteria

Experience points to seven tests as criteria for distinguishing EBTs, Business Objects, and Industrial Objects. While there may be exceptions, these tests generally provide a very reliable method for identification (see the table).

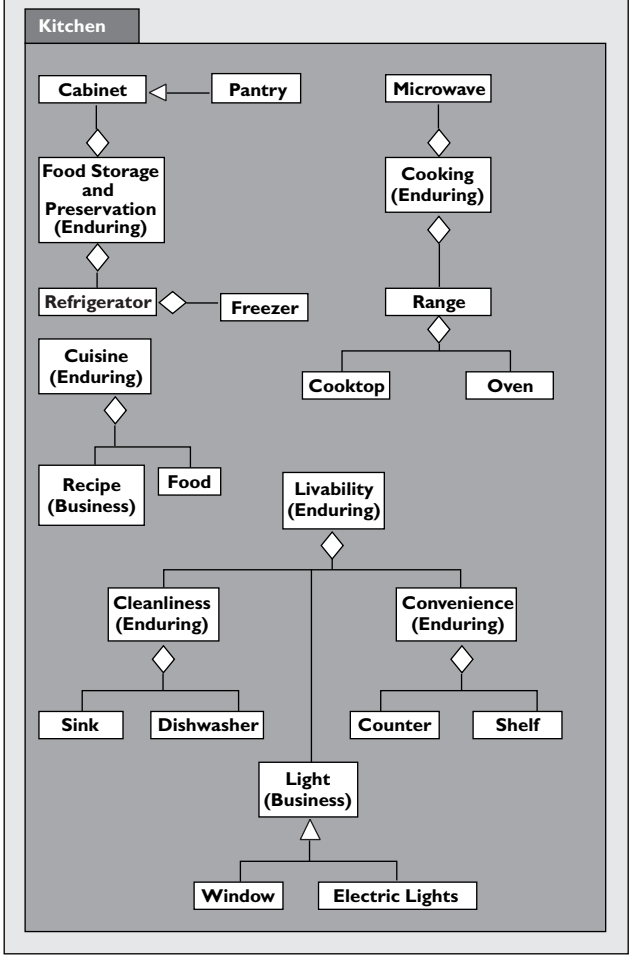
Stability. Probably the most important criterion for identifying EBTs and Business Objects is they must be enduring. Both must be stable over time. Where they differ is in *how* they are stable. EBTs are completely stable both internally and externally. They have never and will never change. All the

EBTs identified in Figure 2 are completely stable. Kitchens must always facilitate cooking and provide a livable environment. Likewise, friends must always consider both their friendship and their finances when reconciling loans, and warehouses must always store goods in an efficient manner and serve their customers if they expect to stay in business.

Business Objects can change internally. Externally, however, they must remain the same. The warehouse case study has a perfect example of this. In the warehouse study, the storage schema is a Business Object. Externally, it must always be a method of storing goods in the warehouse. Internally, however, it is an aggregate of Industrial Objects. Any of these Industrial Objects can change or be replaced. As long as the storage schema is still externally the same, the model is still valid. Item is another Business Object in this model. Internally, it does not matter if the item is a bathroom slipper or a telephone directory. Externally, it is a storable good and that is all that matters to the system.

Industrial Objects can be changed at will. If the system is designed correctly, changing these objects should not cause changes to ripple through the entire system. For example, in a kitchen, the range can be replaced with either a barbecue or a food replicator. As long as it still facilitates the theme of cooking, the model remains intact. Do not take this to the extreme, however. In the warehouse example, one cannot replace the loading dock with a banana and expect this to work. The Industrial Objects must continue to fit with the Business Objects and EBTs to which they are attached.

Figure 2. A more stable model of the kitchen problem.



Adaptability. An object is considered adaptable if it can be used in a system despite changes the system may go through. Some objects are adaptable without any changes to the objects themselves. Others require some changes to the internal processes of the objects. Still others must change completely.

EBTs cannot change at all. How they function internally as well as how they interact with the system externally cannot change. Therefore, EBTs are adaptable without making any changes to the EBTs themselves. It does not matter what changes the system goes through. The EBTs of that system remain constant. They must adapt to the changes without changing themselves [1–3].

Business Objects can only change internally. The processes used inside Business Objects and the other classes that make up Business Objects can change. Everything else about the Business Object must stay the same. Thus, Business Objects are adaptable through internal change.

Thinking Objectively

A very experienced, professional chef might say, "I've always had pots, pans, a stove, an oven, a set of cabinets, and a refrigerator. These must be enduring themes of the kitchen." This chef would be wrong.

Finally, all aspects of Industrial Objects can change. As long as the Industrial Object is reasonable, the system should still be able to work with it. Industrial Objects do not have to be adaptable to change. If a change to a system warrants the removal or replacement of an Industrial Object, that should be fine and the system model should account for this possibility.

Essentiality. EBTs and Business Objects are both essential features of any system. For example, if one removes the storage EBT from the warehouse model, the warehouse is destroyed. A warehouse cannot function without the theme of storage. Similarly, if the cooking theme is removed from the kitchen, the kitchen turns into a room that can store food. The same thing would happen if Business Objects are removed from any of the systems. If the loan object is removed from the loan model, then the model is no longer of a loan between friends. If the customer object is removed from the warehouse model, then the warehouse cannot turn a profit and will go out of business.

The only nonessential objects are Industrial Objects. For example, you can easily remove the range from the kitchen model. As long as there is still some facility to cook food in the kitchen, the model is still a model of a kitchen. Aisles and bin combinations may be removed from the warehouse model and replaced with some other storage medium. As long as there is a way to store things in the warehouse, the warehouse concept remains intact.

Implicitness/Explicitness. A common error in systems requirements analysis is EBTs are almost never explicitly defined in the problem statement. EBTs are frequently left out of all supporting documentation. For example, there is no mention in the problem statement for the warehouse system that the warehouse must serve its customers. The customer service theme must come from knowledge about how a warehouse works and how it survives. The customer service theme is implicit.

Business Objects are sometimes stated in the problem statement or supporting documentation, but with equal frequency, they are not identified. Again, in the warehouse example, the customer Business Object is mentioned and it is explicitly stated that customer information must be stored. However, there is no mention in the problem statement of the storage schema Business Object. Modes of stocking products change over time, but the need to model each storage schema that is used is enduring. This, again, comes from knowledge of how a warehouse works and what must be provided to make a warehouse run efficiently. Frequently, Business Objects are left out of the analysis and design because they are implied.

Industrial Objects can almost always be found by reading the problem statement or its supporting documentation. In the warehouse case study, all of the Industrial Objects in the model were taken directly from the problem statement. Why is this the case? Remember the problem statement states the problem at hand. Consequently, it must state things in terms of what is happening now. It is not usually concerned with stability or change. Therefore, many of the objects mentioned in problem statements will be Industrial Objects.

However, this means the old way of documenting systems requirements must be changed. It is critical to identify both the EBTs and Business Objects early in the analysis of the problem. In short, it is important to define both the implicit and explicit features of the problem during analysis and design.

Intuition. Because EBTs are almost never explicitly stated in the problem statement, one must either have a great deal of experience in the field in question or one must read between the lines of the problem statement. One must use intuition when finding EBTs. For example, there is some

mention of the movement of goods in the warehouse problem. However, there is no explicit statement in the problem stating this movement is important. Intuition must be used to determine this.

Business Objects are sometimes explicitly stated in problem statements. When they are not, however, one must again use intuition to find them. The storage schema and order-filling system objects of the warehouse model were found using intuition. The customer and organization objects are also Business Objects, but they are more commonly addressed and no intuition is required to find them.

Finding Industrial Objects is often a simple chore of extracting them from documentation. Since they are often explicitly stated somewhere, no intuition is required to find them. If one simply reads a problem statement and extracts all the nouns from it, one will find almost all of the Industrial Objects relevant to the system. Therefore, it is common to see objects such as product, location, and forklift in the analysis products of a warehouse system.

Tangibility. Tangibility helps to verify that EBTs are actually EBTs and Business Objects are actually Business Objects. EBTs are themes. As such, they are almost never tangible items. Themes are very conceptual. Take "livability" from the kitchen model as an example. This is not a concrete object. One cannot even associate a concrete object with "livability" and capture the entire essence of livability.

Business Objects are *semi-tangible.* This may be difficult to grasp at first. Take "customer" from the warehouse case study as an example. It is a Business Object. One can easily associate a human or a company or some other tangible item with a customer.

Finally, Industrial Objects are almost always concrete. One can see a range in a kitchen or walk down an aisle in a warehouse. If an object in a model represents a concrete entity, then it is most likely an Industrial Object. It does not take much thought to see why. If an object in a system represents a concrete real-world entity and that entity changes, then that object in the system must change with it. Since concrete entities usually change both internally and externally, the object in the system cannot be a Business Object. Therefore, it must be an Industrial Object.

Commonality to the Domain. Commonality to the domain deals with where the object should go in a system. Both EBTs and Business Objects are stable. The rest of the system should be built around these things. Therefore, the EBTs and Business Objects lie at the core of the system and should be modeled as such. Those things that can change, the Industrial Objects, should be moved as far to the periphery as possible so that changes in them do not have a ripple or cascading effect through the rest of the system.

Classifying artifacts of complex applications as EBTs, Business Objects, and Industrial Objects is a vital technique for systems analysis and design. Applying these concepts allows software engineers to change the way we define problems. Therefore, these concepts yield an important paradigm shift in systems engineering—an approach that specifically targets software stability. C

References

1. Fayad, M. *Software Stability.* MightyWords, 2001
2. Fayad, M. and Cline, M. Aspects of software adaptability. *Commun. ACM 39*, 10 (Oct. 1996), 58–59
3. Fayad, M. and Laitinen, M. *Transition to Object-Oriented Software Development.* (Aug. 1998) John Wiley & Sons, New York, NY.

Mohamed Fayad (fayad@cse.unl.edu) is J.D. Edwards Professor at the University of Nebraska, Lincoln.

If the reader is interested in learning more about these concepts, he or she will find some case studies posted at www.cse.unl.edu/~fayad—check Case Studies. If the reader would like to submit a sample and specific domain problem, I would be happy to model it.

Mohamed E. Fayad

How to Deal with Software Stability

All that exist are heuristics for determining an EBT, a BO, or an industrial object.

In my previous column I introduced the concepts of enduring business themes (EBTs) and business objects (BOs) as essential artifacts used to achieve software stability [2, 3]. This third case study compares and contrasts a classical model built by an expert in the field of software engineering, Peter Coad, and a model of the same problem based on Enduring Business Themes and Business Objects.

The project is to create a warehouse tracking system allowing goods storage and order filling to be performed more quickly and more efficiently. The warehouse is physically organized into aisles of bins. Each bin contains items, and each pair of bins and items has a line item associated with them for clerical purposes. Items arrive at the warehouse on pallets. Since pallets do not have to contain the same item, they also have line items for clerical purposes. When a customer places an order, that order and the line items on it are translated into a pick list and list line items. The items indicated by the pick list are finally retrieved from their appropriate locations in the warehouse and moved to the loading dock for shipment. The tracking application must also keep track of customer information and customer organizations to better serve repeat customers.

Coad's model of this problem [1] has several aggregations and finely intertwined relationships between classes. Many of these classes could be classified as Industrial Objects (IOs). Consequently, minor changes to this system could result in major changes to this model and would require a reengineering process to fix. Finding EBTs for a warehouse tracking system is no trivial task. However, once it is done and if it is done properly, the resulting model will be far more stable and robust to change.

Warehouses are used, of course, for storing goods. Thus, one of the obvious EBTs that should be incorporated into the model should be goods storage. People also want their goods stored and retrieved in a timely manner. Therefore, the warehouse must be run and the storage systems must be structured efficiently. It is easy to see that efficiency is another EBT worth consideration. Other EBTs that immediately come to mind are shipping, receiving, order handling, customer service, and movement of goods.

There are several BOs involved in a warehouse. Several of them exist in [1]. First and foremost, the warehouse itself is a BO. The goods to be stored are also BOs. It does not matter to the system what goods are being stored. To the system, goods are goods. In [1], the "Item" class represents goods, so, for consistency's sake, they will remain members of the "Item" class in the EBT-based model. Customer, organization, and order, from [1], are also BOs. The rest of the objects in his model, however, are IOs. They can all be changed completely without changing the basic premise of a warehouse.

Although this may not appear obvious, the EBT-based model is far more adaptable. Most of the associations between classes have been moved to associations between EBTs and between BOs, which will never change in such a way as to affect the system. All the changes will occur in the IOs, which have been moved toward the periphery of the system.

TERRY MIURA

Identification Heuristics

Some heuristics for finding enduring themes immediately come to mind. Identification of EBTs and BOs does not come easily. Engineers used to classical methods of object-oriented modeling will have to change their thought processes drastically to cope with these new concepts. These heuristics are designed to ease this transition in thought.

First and foremost, EBTs and BOs must be enduring. In other words, they must be stable over time. In many cases, determining whether or not something is enduring simply takes experience in a given field. However, most of the time it takes a little more. Field experience allows one to know how long given concepts or objects have been involved with a given system. A long history in the field, however, does not necessarily translate into longevity in the future.

Determining whether or not an object will be enduring takes a knowledge of the system, a modicum of intuition, and common sense. It is obvious that a new process introduced to a system is less likely to be enduring than a process that has had a place in the system for years. However, as evidenced by the now infamous "millennium bug," not even a millennium of time can always be considered enough time for something to become enduring.

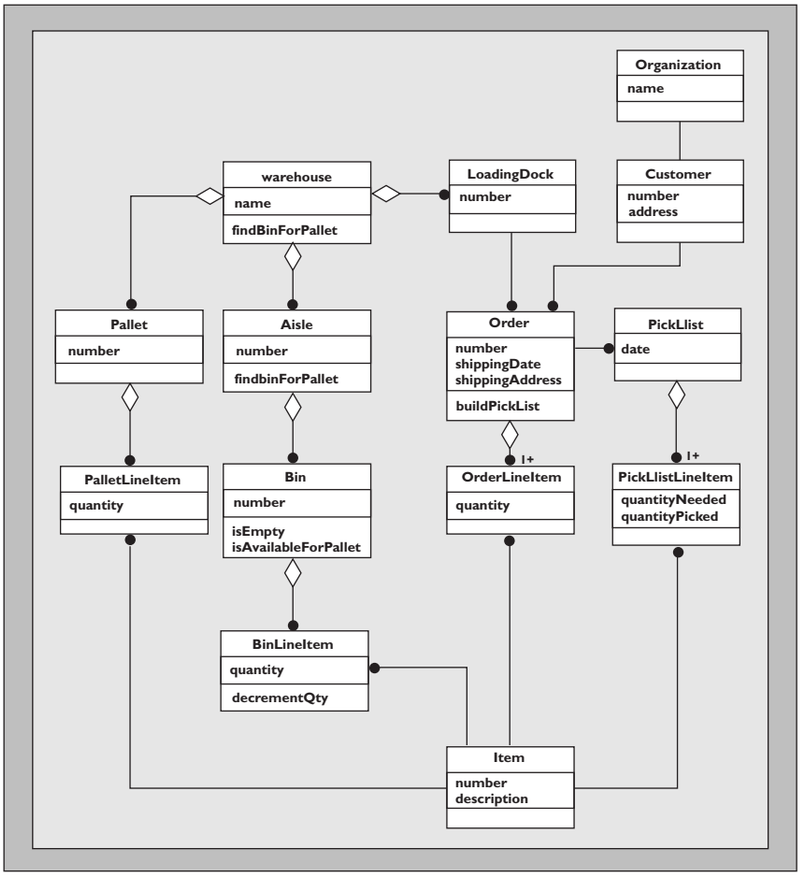


Figure 1. A model of a warehouse tracking system [1].

Industrial Object Identification

Identifying IOs is easy. Usually, the objects one designs into a classical object model are IOs. To determine whether an object is an IO, one must simply ask: "Can this thing be replaced with something else? Will the system remain viable if some other object replaced this one?" If the answer to either of these questions is "yes," then the object is most likely an IO.

One should also look at the "tangibility" identification criterion. If the object represents some physical entity, that object is very likely to be an IO. If it represents a piece of machinery, a button, an input device, or an output system, it must be replaced any time the physical entity it represents is replaced. Thus, it should be considered an IO and placed toward the periphery of stable designs.

Top-Down Identification

Once one gets a feel for identifying IOs, identification of BOs and EBTs becomes far easier. There are at least two viable methods for finding BOs and EBTs. One method is to take a top-down look at the system. Start with the whole system.

Then, start breaking off conceptual pieces of the system. "What is the system used for? What must it be? Why are we building this system?" For example, in a kitchen, the system serves as a meeting area and a living area and is used for preparing food. Warehouses, as another example, store commodities, serve customers, and serve as hubs for the shipping of goods. Name each of these subunits and continue to break them down. Once IOs, which are easy to identify, are encountered, stop and back up one level. This level of "objects" contains good candidates for BOs and EBTs.

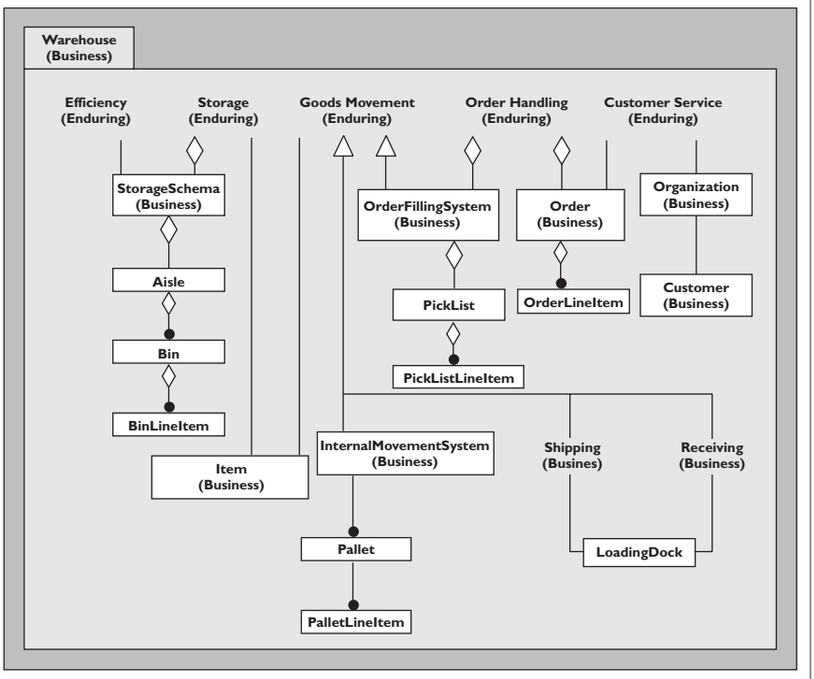


Figure 2. An EBT-based model of the warehouse tracking system.

Bottom-Up Identification

Another method is to start at the bottom and work upward. Start by building a classical model of the problem at hand. Then, find the IOs in the model. Again, this should be easy. It is likely most of the objects in the classical model will be IOs. Try to group these objects and find the concepts covering them. For example, in the kitchen, one might pair the refrigerator and the freezer. "What are the refrigerator and freezer used for?" one might ask. Well, they keep things cold, obviously. "Why keep things cold?" Food must be stored cold to preserve it. Aha! We have found the EBT of food preservation. Continue this grouping and finding of concepts until it cannot be continued. The only task remaining is to separate the EBTs from the BOs.

As evidenced by the now infamous "millennium bug," not even a millennium of time can always be considered enough time for something to become enduring.

Note the bottom-up method of identification will only find those EBTs and BOs that cover IOs that already exist in a classical object model. Many other EBTs and BOs should probably still be considered in order to form a complete analysis of the problem at hand. For example, it is highly unlikely that a bottom-up identification scheme will ever have found the "livability" EBT of the kitchen model shown in the case study.

No Silver Bullets

There are no silver bullets. Just as there are no hard-and-fast rules for finding EBTs and BOs, there are no silver-bullet heuristics for separating the two. There are a few decent heuristics that can help, though. One method of separating EBTs from BOs is to examine the tangibility of the theme and object candidates. Usually, EBTs are themes—they are conceptual, overlying entities of a system. BOs, on the other hand, are far more tangible

objects. They are overlying processes or methods or step-by-step instructions. While BOs are more conceptual than IOs, they are usually more tangible than EBTs. Consider, for example, the recipe that was documented in the kitchen case study. Recipes do not change over time even though the entities used to prepare recipes do. Also consider the "storage schema" object in the warehouse case study. In order to be efficient at storing and retrieving commodities, warehouses should have some method of storing goods in an orderly fashion. That method may change, but there will always be some efficient method of storing commodities. One cannot hear, see, or feel the recipe contained in a chef's brain. Nor could one pick up a storage schema. However, both the storage schema and the recipe can be translated to something more tangible than a nebulous theme.

EBTs are those themes that remain constant for a given system. They should be thought of as those stable objects that should make up the core of software systems.

Conclusion

The heuristics for finding EBTs and BOs that were documented in this column are merely suggestions. There is no guaranteed method for determining what is an EBT, what is a BO, or what is an IO. All that exist are a series of heuristics or rules of thumb that can be used to help in the process. Nevertheless, EBTs are a vital analysis artifact. The identification of EBTs may be one of the only methods a software engineer has at his or her disposal that can help design stable software over time.

Used properly, EBTs and BOs can create a stable foundation around which stable software systems can be built. The identification of these stable foundation entities will require a drastic change in the way software engineers think about problems. However, the time and money saved by engineering more stable systems far offset the cost of this mental retraining.

EBTs are those themes that remain constant for a given system. They should be thought of as those stable objects that should make up the core of software systems. BOs are also stable objects around which to build software systems, but unlike EBTs, the internal processes of BOs may change if the need arises. Externally, however, BOs are as stable as EBTs.

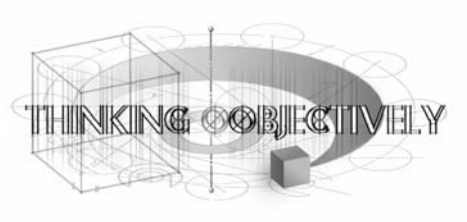
Problem analysis using EBTs and BOs requires far more intuition and knowledge about the problem domain than more classical problem analysis techniques. EBTs and BOs are rarely stated in problem statements or supporting documentation. Their identification requires an in-depth analysis of the problem domain and the business systems the problem is dealing with. This may require far more interaction between software engineers and those people who will be using the resulting software system. Finally, the identification of EBTs and BOs will require more of a "gut reaction" than most software engineers are used to relying on. Again, it will require some time to complete this mental retraining.

This is not a silver bullet. The use of EBTs and BOs is not a replacement for other good software engineering practices and common sense. It is obvious that, when analyzing problems, concentrating on those aspects of the problem that will remain stable will yield a more stable analysis of the problem. Once we learn to identify these stable foundations upon which to build our solutions, we have made the first step toward accomplishing software stability. C

References

1. Coad, P., North, D., and Mayfield, M. *Object Models Strategies, Patterns, and Applications.* Yourdon Press, Prentice-Hall, Engelwood Cliffs, New Jersey, 1995.
2. Fayad, M.E. and Alltman, A. Introduction to software stability. *Commun. ACM 44*, 9 (Sept. 2001), 95–98.
3. Fayad, M.E. Accomplishing software stability. *Commun. ACM 45*, 1. (Jan. 2002), 111–115.

Mohamed Fayad (fayad@cse.unl.edu) is J.D. Edwards Professor at the University of Nebraska, Lincoln.



Mohamed Fayad and Shasha Wu

Merging Multiple Conventional Models in One Stable Model

Stability models may demand greater investment in analysis, but when used wisely, savings in development and maintenance costs more than make up for it.

Unlike the products of other engineering fields, there is no physical deterioration to cause software to fail. Most software defects and deterioration are caused by changes in software [2]. Generally, these changes cannot be avoided. Such changes are often the result of the natural evolution of business processes and changes in the underlying requirements of software systems to meet these evolving needs. To achieve software stability is to balance the seemingly contradictory goals of stability over the software life cycle with the need for adaptability and extensibility.

To accomplish this balancing act, we hypothesize that certain refinements can be applied to conventional OO analysis and design techniques. However, such refinements must not overly complicate our conventional approach. In software, we view simplicity and elegance as synonymous. Simplicity is considered an important characteristic of a good model. To demonstrate the simplicity and elegance of the Software Stability Model (SSM), we apply it to a study of open-pit mining (The detailed description of this problem can be found at ww.cse.unl.edu/~fayad/SoftwareStability/OOPSLA01-DesignFest-Final1.doc).

The Study

In an open-pit mine, mechanical shovels load material from depots into dump trucks. These trucks then transport material to various destinations. Ore is transported to mineral processing facilities to be processed and prepared for market. Waste is delivered to dumps.

A scheduling or dispatching system is required to assign the empty trucks to their proper destinations. This system checks the status of each truck and shovel. As soon as a truck is free, it is assigned to a shovel that will be free when the truck arrives. If all the shovels are busy, the dispatcher system selects the shovel with minimum wait time.

A conventional model for this problem is given in Figure 1. The model illustrates how the transport system works. Ore is placed into the ore extraction openings, and is then transported to a mineral processing facility. Similarly, waste is placed into waste removal openings and is then transported to waste dumps. Dump trucks are assigned to these openings according to a schedule. Shovels load ore and waste into these trucks.

The project's business objective is to maximize the mine's profitability and satisfy production quotas. The project will accomplish this by optimizing the equipment, personnel, and the overall efficiency of the mining operation.

We can readily imagine changes in production techniques or mining conditions that will require changes to this software model. Perhaps conveyor belts are used as the main transportation methods instead of trucks. In this method, loaders carry materials from the depot and dump it into feeders. The feeder feeds a crusher at a steady rate and the crusher reduces the size of rocks to a proper size for transporting by conveyor belt. The whole system must be consistent in terms of capacity of materials flow. The loader must handle the required volume of material over time (for example, tons/hour.) The feeder must feed the crusher at the same rate, and the crusher must deliver the same amount of material of the proper size to the conveyor. Conveyors, in turn, must have the proper width and speed to transport the material. If any element of this system fails to maintain the

PAUL WATSON

required flow rate, material congestion occurs, and the whole system fails to achieve its goal of transporting the designated quota of material to the destination.

With the introduction of this new technique, the conveyer belt, the old model fails to satisfy the business requirements. We must modify it. The new class diagram might be generated as in Figure 2a. We note that the new model doesn't bear a close resemblance to our original model, nor would it satisfy the requirements of our first open-pit mine.

Other locations may use a pipeline as the transportation mode. With this pipeline technique, material is loaded into feeders by loaders. Each feeder passes the material to a crusher and then on to a ball mill. Ball mills reduce the size of materials and prepare them for transportation through pipelines. The milled material is poured into mixers, where water is added to make slurry. The pipeline then carries the material to the destination.

With conventional models, the tendency is to remodel (as in Figure 2b). Again, the result is a substantial change.

As these examples illustrate, conventional OO modeling is subject to instability. Many changes to the business process most likely to lead to changes in the previous models, prompting a reengineering process. If the previous work is reused, the tendency is to attach new objects to the existing model similar to the way that barnacles adhere to the hull of a ship.

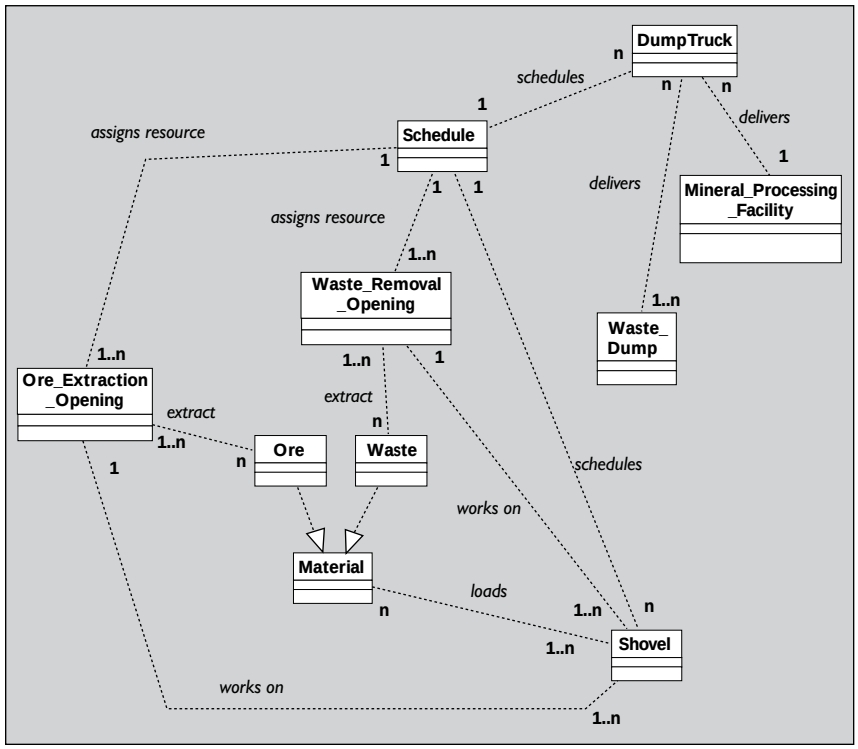


Figure 1. Typical class model for open-pit mining.

The Stability Approach

With the stability approach (see Figure 3), we first define the purpose of the system and delineate why the system is needed.

Although this aspect of analysis is not unique to the SSM, the model formalizes the analysis and focuses the analysis activities around the concept of Enduring Business Themes (EBTs). EBTs are the core abstractions of a problem domain. These themes are unlikely to change over time. For example, one important theme of the open-pit mining problem is "efficiency." The concept of efficiency clearly has a role as an EBT, because efficiency is part of the business objective for building the system. When we derive a schedule for trucks, we are actually trying to make the system work more efficiently, hence, more profitably.

Another important theme is "concurrency." Consider the interaction between different components of the transport system; they are assigned to work together when they are available. Dump trucks are assigned to shovels when they are available, and they only work together when shovels are available at the same time. In other words, they have concurrency. This concept is also for

material flow through the conveyor belt and pipeline systems.

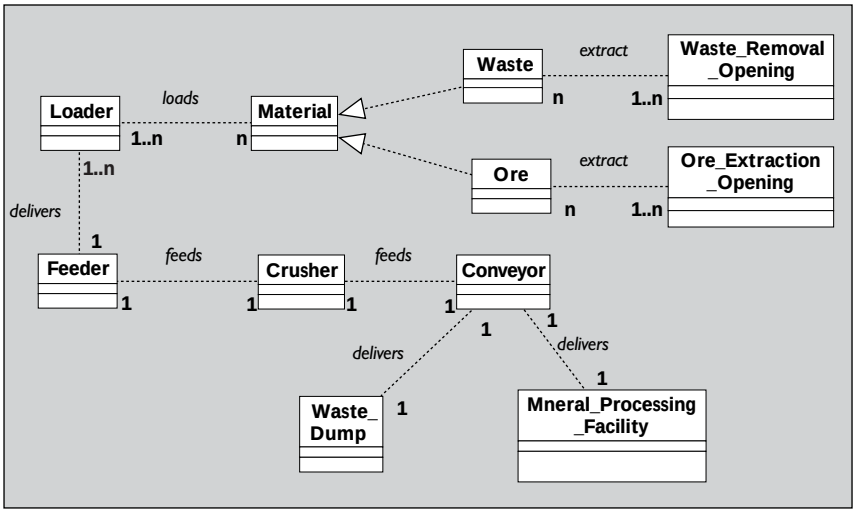
All of the components in the system have to pass a specific amount of material to the next component within a specified amount of time. All of these material flows have to be equal in a system and all the components of the system must be available and working at the same time to be concurrent. Without this concurrency we cannot imagine a transport system; the result is congestion and failure. Therefore, concurrency is a very important measurement and one of the core purposes of our scheduling system.

After identifying the EBTs associated with our problem, we can identify and associate those business objects (BOs) that provide the necessary abstractions of the processes underscoring the business operations (such as origin, destination, and transport). The third step of the SSM is to define those Industrial Objects (IOs) that refine (or instantiate) the various BOs. In this way, our model exhibits stability over changes. If we substitute feeders, loaders, crushers, and conveyors for shovels and dump trucks, the core model is unchanged. It is through changes to the peripheral objects that we achieve the balance between changes and our goal of stability. Despite the various modifications to support various instances of the problem, our EBTs and BOs remain stable.

In addition, the model remains well organized over the course of many design iterations. Ultimately, we believe this approach has tremendous benefits for software teams. This study shows the SSM is elegant and extensible.

The layered approach of the

Figure 2a. Class diagram of conventional model for conveyor belt transportation.



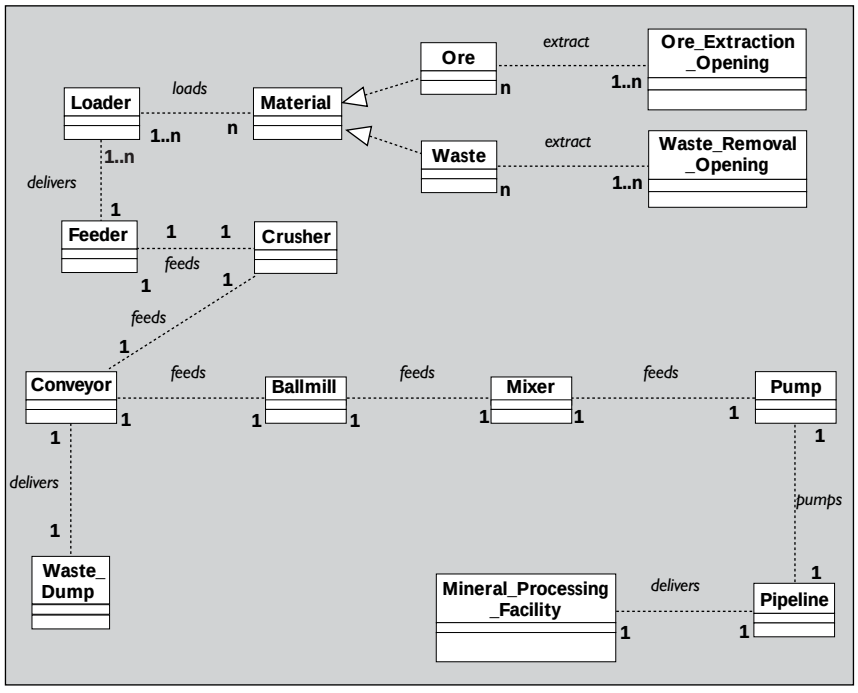


Figure 2b. Class diagram of conventional model for pipeline transportation.

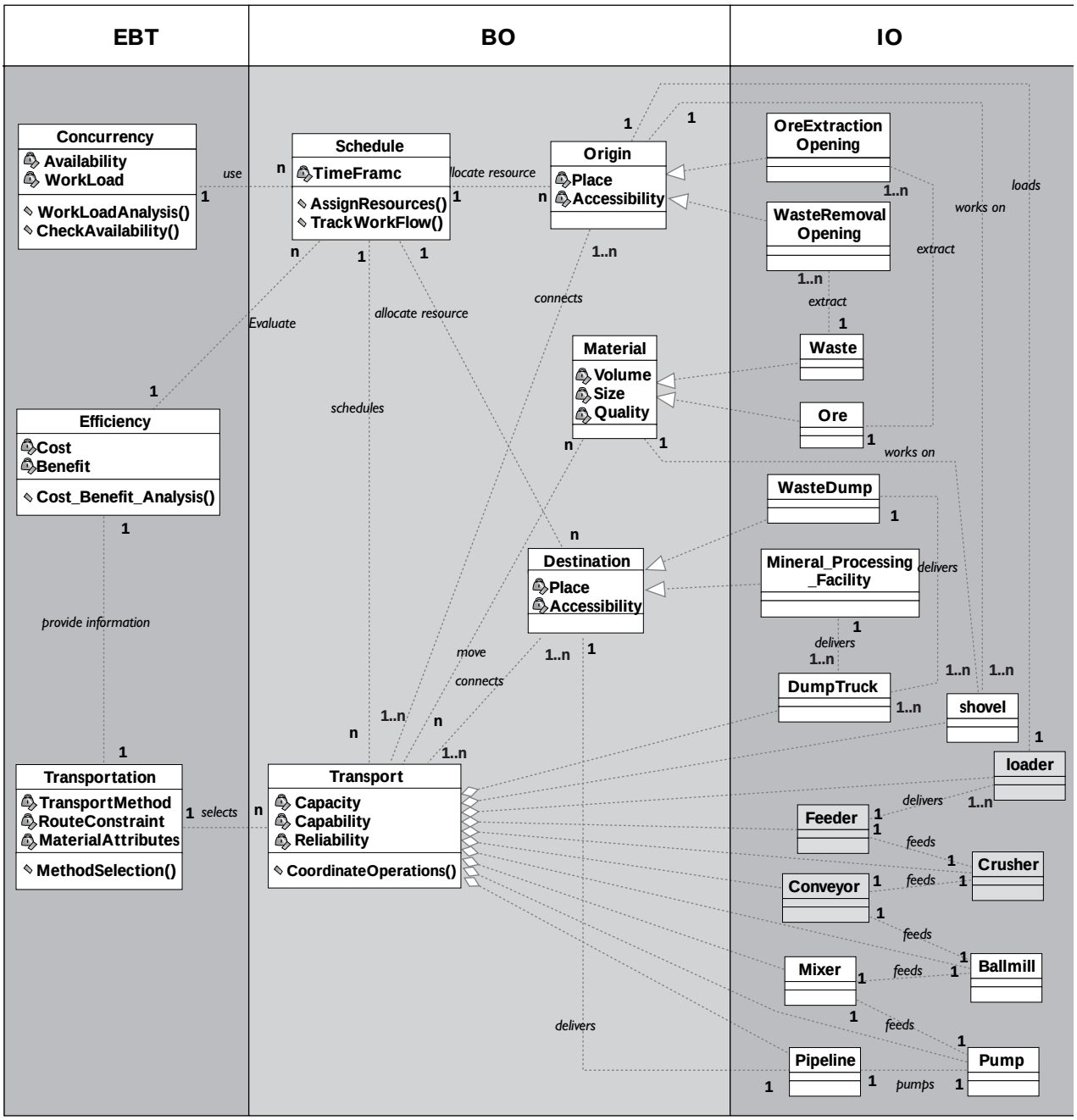


Figure 3. Class diagram of stability model for transport system.

SSM provides an important technique for decomposing a problem space. By adding the focus of identifying EBTs and BOs, the SSM forces software engineers to clearly rationalize the role of each object described within a problem space.

Due to their higher level of formalism, we believe SSMs are more measurable and testable than conventional models. EBTs describe the goal of the system [1, 2]. BOs

are the plan to reach those goals and IOs are the objects that perform the work to realize the plan [1, 2]. The sequences and logical relationships among those objects are clear. By tracing the objects top-down, we can test and verify the use of every class in the diagram systematically. We will address the issues of measurability and testability in future columns.

Although the stability approach requires a high level of discipline in analysis and a high level of rigor, this is inevitable in any approach that tries to reduce maintenance costs. The three-layer method for identifying EBTs, BOs, and IOs is central to the Stability Approach. System designers can provide a precise, well-structured, adaptable, and maintainable solution for a problem only if they understand why the objects are necessary to the problem.

Conclusion

The common ideas of conventional models try to describe the problems and solutions using real-world objects and by providing a mechanism to guide the subsequent programming process. However, in a stability model, the EBTs and BOs answer the questions: "Why do we build this system?" and "How does the system achieve its goal?" Thus, the key to stability modeling is to identify aspects of the environment in which the software will operate without change, and to cater the software to these areas [1].

In conventional models, engineers often conclude the analysis when a solution to the problem is identified. On the other hand, the stability model more accurately reflects a designer's reasoning and the underlying process used in the analysis and exposed through EBTs and BOs. Therefore, we conclude that stability models are easier to understand and evaluate.

While stability models demand a greater investment in analysis, our practical experience has shown that, when used wisely, the savings in development and maintenance costs can more than make up for the additional time spent during analysis. We also believe the stability approach has potential to reduce or eliminate the cost of the reengineering cycles commonplace in software engineering projects. We view the software stability approach as an essential refinement to existing OO analysis and design processes. Moreover, we do not note any added complexity in SSMs. Instead, our study suggests SSMs are concise, elegant, and inherently adaptable and extensible. C

Mohamed Fayad (fayad@sjsu.edu) is a professor of computer engineering in the Department of Engineering at San José State University, San José, CA.
Shasha Wu (shwu@cse.unl.edu) is a Ph.D. student in the Department of Computer Science and Engineering at the University of Nebraska, Lincoln.

References

1. Fayad, M. Accomplishing software stability. *Commun. ACM 45*, 1 (Jan. 2002), 95–98.
2. Fayad, M. and Altman, A. An introduction to software stability. *Commun. ACM 44*, 9 (Sept. 2001), 95–98.