JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 10, November-December 2004

Generating Complete, Unambiguous, and Verifiable Requirements from Stories, Scenarios, and Use Cases

Donald Firesmith, Software Engineering Institute, U.S.A.

Abstract

Simple scenarios and stories are typically used for requirements engineering in the Agile community (e.g., eXtreme Programming). Use case modeling has also been a popular requirements elicitation and analysis technique for many years. However, stories, scenarios, and use cases typically exhibit a great informality that violates the traditional guidance in the requirements engineering community that requirements should be complete, unambiguous, and verifiable. This is why many professional requirements engineers use these techniques only as tools for informal requirements elicitation, analysis, and validation. Instead during requirements analysis and specification, more experienced requirements engineers tend to develop and specify more formal textual requirements that are complete, unambiguous, and verifiable.

This column will show how to transform incomplete and vague stories, scenarios, and use cases into a proper set of complete, unambiguous, and verifiable requirements.

1 THE CHALLENGE

Over the last five years, members of the Agile community (e.g., users of minimal formality development methods such as eXtreme Programming) have strongly recommended the production of simple stories and scenarios as the primary form for requirements during requirements engineering. Proponents claim many benefits for their use including greatly improved productivity and customer satisfaction. Unlike the more structured and formal approaches that generate individually identified and specified textual requirements, stories and scenarios seem to be easier to learn and use. Because they rely on people's natural ability to read and tell stories, simple scenarios are often used by stakeholders with no more training than the reading of a short overview article on the subject in a popular journal. Coupled with close collaboration with customers and other stakeholders, stories and scenarios are also claimed to better deal with rapidly changing requirements.

Cite this column as follows: Donald G. Firesmith: "Generating Complete, Unambiguous, and Verifiable Requirements from Stories, Scenarios, and Use Cases", in *Journal of Object Technology*, vol. 3, no. 10, November-December 2004, pp. 27-39. http://www.jot.fm/issues/issue_2004_11/column3

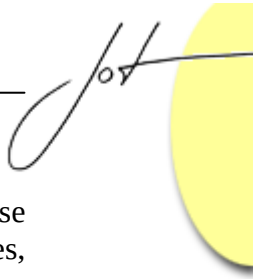
Similarly, use case modeling has been a popular requirements elicitation and analysis technique for even longer, especially in the object community. Use cases can provide significantly more structure than simple stories, although many developers of use cases seem to document individual use case paths using little more than a paragraph of narrative text. In practice, only a relatively small number of use case writers seem to be careful about (1) including preconditions and postconditions, (2) differentiating requirements for ancillary information, and (3) adequately specifying exceptional paths instead of merely documenting the "sunny day scenarios."

Although stories, scenarios, and use cases do have their uses and advantages, their informality causes them to violate the common traditional guidance from the requirements engineering community that states that requirements should be complete, unambiguous, and verifiable [Firesmith 2003]. Stories, scenarios, and use cases are usually incomplete because they typically do not contain all information required such as relevant preconditions and postconditions. For example, an ATM must be able to enable customers to withdraw money from their bank accounts, but ATMs must only do this under certain circumstances such as only if the customer has sufficient funds in the account. Being written in unstructured, simple text, stories, scenarios, and use cases often contain a great amount of ambiguity that makes it impossible to verify them. In fact, it is often difficult to decide just what parts of them are intended to be requirements.

Stories, scenarios, and typical use cases are missing important information and rely on the domain knowledge of their readers. This incompleteness, ambiguity, and lack of verifiability also means that the readers of requirements "specified" as stories, scenarios, and use cases must recognize the implicit hidden assumptions and fill in the missing information. Unfortunately, different readers will interpret the requirements differently, based on their own individual experiences and assumptions. The resulting system is like a house built on shifting sands; without a firm foundation of proper requirements, the architecture, design, implementation, and testing of the resulting system suffers.

It has been well known for many years that the costs and schedule "saved" by skimping on requirements engineering can be lost many times over during the rest of development and operations [Boehm and Papaccio 1988]. Perhaps worst of all, requirements deficiencies can also cause critical safety hazards as systems become more mission and safety critical. For example, one study of 34 safety incidents showed that 44% of them were primarily due to inadequate requirements specifications [HSE 1995].

This is why many professional requirements engineers use stories, scenarios, and use cases only as tools for informal requirements elicitation, analysis, and validation. Instead, the more experienced requirements engineers tend to develop and specify complete, unambiguous, and verifiable textual requirements during requirements analysis and specification. Thus, a major challenge for requirements engineers is the task of turning informal, ambiguous, incomplete, and unverifiable stories, scenarios, and use cases into complete, unambiguous, and verifiable requirements.



The next two sections of this column will use a standard example to illustrate the use of stories, scenarios, and use cases. Then the next section will turn the resulting stories, scenarios, and use cases into complete, unambiguous, and verifiable requirements.

2 SIMPLE STORIES AND SCENARIOS

ATM Example

We will use the requirements for an automated teller machine (ATM) as our example in this column to clarify the weaknesses of stories, scenarios, and use cases as ways to specify requirements. The ATM has several advantages as an example. It is large enough to illustrate most aspects of requirements elicitation and analysis without being too large and overwhelming. And because ATMs are familiar to everyone, no special domain knowledge is required.

Restricting ourselves to the most common user of the ATM (i.e., a customer as opposed to the role of someone servicing the ATM), there are typically 4 different ways of using an ATM. For the purpose of this column, we will restrict ourselves to the most common case of withdrawing money from an account.

Simple Textual Story

A typical simple textual story for withdrawing funds from the ATM might go something like the following:

- A customer walks up to an ATM and inserts his bank card. The ATM responds by welcoming the customer and requesting that he enters his PIN number. After authenticating the customer using information on the bank card, the ATM displays several options and asks the customer what he would like to do. Once the customer decides to withdraw funds, the ATM displays the customer accounts and asks the customer which account he would like to withdraw funds from. Once the customer selects the account, the ATM asks how much money he would like to withdraw from the account. The customer chooses $60. The ATM forwards the request to the Bank, which approves the withdrawal. The ATM dispenses the withdrawn amount, prints out a receipt, and asks the customer if he would like another transaction. The customer declines, and the ATM displays a greeting for the next customer.

The preceding example story has several potential problems from a requirements standpoint:

- First of all, it assumes a traditional ATM architecture which uses bank cards to identify customers and the entry of PIN numbers to authenticate the customer's identity. This architecture constraint mandates the least secure (worst) approach to access control and precludes the current or future use of much more secure and convenient security controls such as the use of biometrics (e.g., thumb print reader). There is also ambiguity because if bank card is used. For example, the ATM can

either not temporarily take the bank card (swipe card) or else temporarily hold the bank card and then return it upon completion of transactions.

- The story ignores preconditions implementing bank rules such as the need for sufficient funds, the absence of a hold on the account, and having not already exceeded the daily maximum withdrawal amount. It also assumes that the ATM can communicate with the bank, that the bank's computer is not down, that the ATM stores sufficient funds, and that the receipt printer has paper.
- The story does not state what interactions are trigger events (i.e., what the customer and bank computer do) and what interactions are requirements (i.e., how the ATM must respond to these triggers). Requirements are not explicitly stated as such (e.g., using the word "shall"). Requirements are also not explicitly identified with a project-unique identifier (PUID).

Simple Scenario

Unlike stories, scenarios are typically more specific in that they use actual objects and data. A typical simple scenario for withdrawing funds from the ATM might go something like the following:

- Mr. John David Smith walks up to ATM number 15856 and provides adequate information for the ATM to successfully identify him and authenticate his identity. The ATM welcomes Mr. Smith by name and requests that Mr. Smith select a type of transaction (i.e., withdrawal funds from an account, deposit funds into an account, obtain account balance, or transfer funds). Once Mr. Smith selects withdraw funds, the ATM displays Mr. Smith's accounts including: checking account number 1593 4782 1594 1947, savings account number 1593 4782 1853 9977, and vacation account number 1593 4782 2292 2999. Mr. Smith selects his checking account, which has an available balance of $3,496.75. The ATM presents 5 options including withdraw $20, withdraw $40, withdraw $60, withdraw $100, and "enter an amount to withdraw in increments of $20 up to a maximum of $200." Mr. Smith selects $60. The ATM builds a withdrawal authorization transaction requesting the $60 from Mr. Smith's checking account that includes the date and time (8 October 2004 at 4:16PM), the ATM identifier (15856), the transaction type (withdrawal), the account type (checking), the account number (1593 4782 1594 1947), the transaction amount ($60), and the new available balance ($3,436.75). The ATM encrypts the transaction information, digitally signs the transaction, and sends it to the bank computer on the leased line connecting them. The ATM receives an encrypted and digitally signed response approving the withdrawal amount at 8 October 2004 at 4:17PM. After decrypting the response, verifying that it was sent by the bank, and verifying that it was not corrupted, the ATM dispenses three $20 bills and prints out a receipt with the date and time (8 October 2004 at 4:17PM), the ATM identifier (15856), the last 4 digits of the account number (1947), the ATM location (100 Main Street, Metropolis, New York, 10010), the withdrawal amount ($60), and the new available balance ($3,436.75). The ATM asks Mr. Smith if he would like another transaction. Mr.

Smith declines another transaction, his session ends, and the ATM displays a standard greeting for the next customer.

This example scenario has several advantages over the example story from a requirements standpoint:

- First of all, it no longer contains an architectural constraint in the sense of a specific mechanism for performing identification and authentication.
- It is much more specific about what happens.

However, the example scenario also has several potential problems from a requirements standpoint:

- The scenario is at too low of a level of abstraction. Although it contains more details than the story, the scenario still does not clearly distinguish mandatory requirements from ancillary information.
- The scenario is actually more of a specification of a single test case than a specification of the requirement(s) being tested.
- As before, the scenario does not clarify preconditions and postconditions. It also makes the same "sunny day" assumptions that every thing works properly. It does not explicitly identify requirements as such and distinguish them from ancillary information.

3 USE CASE MODELING

A use case is a general way of using a system to achieve a goal (i.e., something of benefit to an actor). Although a use case is typically represented as a named oval on a use case diagram, a use case is actually typically specified as a collection of normal and exceptional paths (a.k.a., courses) through the use case. Extra important information such as preconditions and postconditions can and should also be specified. If you think of a use case as a very large procedure, then use case paths can be thought of as execution paths through the use case. Use case paths can be either normal (i.e., successful in that the goal is achieved) or exceptional (i.e., the use case fails). Use case paths also define associated equivalence classes of test scenarios (i.e., each scenario in the equivalence class flows down the same execution path through the use case).

If we select "Withdraw Funds" as our use case, then some of the possible use case paths include:

- Normal Paths:
  - Preset amount of funds successfully withdrawn
  - Customer determined amount of funds successfully withdrawn
- Exceptional Paths:
  - Account Overdrawn
  - Withdrawal Amount Requested Would Result in Insufficient Funds
  - Excessive Daily Withdrawals

— Hold on Account
— Connection to Bank Lost
— ATM has Insufficient Funds
— Etc.

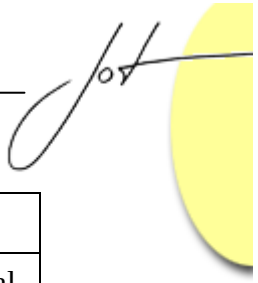
Note that there are typically many more "rainy day" paths than "sunny day" paths through a use case. This is because there are typically many more ways for things to go wrong than to go right.

A use case path can be specified in multiple ways. In addition to using paragraphs of narrative text (i.e., stories), two better and more popular approaches are to use swim lane diagrams and structured lists of interactions between actors and the system being specified.

Use Case Path as a Swim Lane Diagram

A common way to document a use case path is to use a sequence diagram or equivalently a swim lane diagram. Each swim lane documents the actions performed by either the system or one of its actors. The following is a possible swim lane diagram for the "Customer determined amount of funds successfully withdrawn" "sunny day" path through the "Withdraw Funds" use case.

Customer	Automatic Teller Machine	Bank
Identify and authenticate self to ATM		
	Verify authenticated identity	
	Request transaction category	
Select withdrawal		
	Request withdrawal account	
Select customer account		
	Notify preset withdrawal amounts	
Build and then request withdrawal amount		
	Build secure withdrawal authorization transaction	
	Send withdrawal authorization transaction to bank	



		Authorize transaction
		Build secure withdrawal authorization transaction
		Send withdrawal authorized transaction to ATM
	Dispense money and record of transaction	
	Query further transaction	
Answer no		
	Display greeting for next customer	

The example swim lane specification of a use case path has several advantages over the example story and scenario from a requirements standpoint:

- Its structure clearly differentiates the differing responsibilities for the customer, ATM, and bank.
- It is succinct, avoiding both implementation constraints and superfluous test data.
- Note that in this case, this example swim lane diagram has included optional internal actions (e.g., the building of secure transactions). Some proponents of use case modeling recommend that only externally visible interactions be used, arguing that requirements only involve externally visible behavior. However, this information is usually implicit in the interactions and provides useful information regarding preconditions.

However, the example swim lane specification of a use case path still has several potential problems from a requirements standpoint:

- The use case path specification still ignores preconditions implementing bank rules such as the need for sufficient funds, the absence of a hold on the account, and having not already exceeded the daily maximum withdrawal amount. It also assumes that the ATM can communicate with the bank, that the bank's computer is not down, that the ATM has sufficient funds, and that the receipt printer has paper. Even if the author of the use case path had included use case level preconditions, these preconditions would not apply to exceptional paths, which violate the use case preconditions and have their own exceptional preconditions.
- The use case path specification still fails to explicitly state what interactions are trigger events (i.e., what the customer and bank computer do) and what interactions are requirements (i.e., how the ATM must respond to these triggers).
- The interactions are very brief, making them potentially cryptic and very ambiguous to those unfamiliar with the application domain. Because the reader is

almost certainly familiar with ATMs and their associated rules, the reader can fill in this missing information. Unfortunately, this does not work when building systems in less familiar application domains. Neither does it ensure that different readers will make the same assumptions.

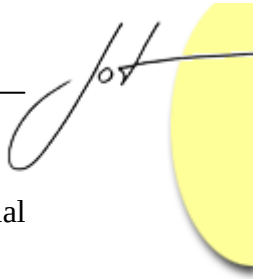
Use Case Path as a List of Use Case Path Interactions

A more complete and less ambiguous version of the use case path specification would be a sequence of interactions in the conversation between the system and its actors. The most obvious addition is the use of complete sentences. The interaction between two entities (system and actor) is made explicit by listing both in the sentence. The initiator of the interaction that was previously shown by positioning the action in a swim lane is made clear by making the initiator the subject of the sentence. The difference between ancillary information (interactions initiated by actors) versus requirements (interactions initiated by the system) is optionally made explicit by the use of the word “shall” on those interaction sentences, the subject of which is the system under discussion.

1. The customer identifies and authenticates himself/herself to the ATM.
2. The ATM (requests / shall request) the type of transaction be selected by the customer.
3. The customer selects to make a withdrawal from the ATM.
4. The ATM (presents / shall present) the withdrawal options to the customer.
5. The customer builds a withdrawal amount that is divisible by 20 that is less than the ATM maximum withdrawal amount.
6. The customer notifies the ATM of this withdrawal amount.
7. The ATM (shall build /builds) a secure withdrawal authorization transaction.
8. The ATM (shall send /sends) the secure withdrawal authorization transaction to the bank.
9. The bank sends a secure withdrawal authorized transaction back to the ATM.
10. The ATM (dispenses / shall dispense) the selected money to and print out a record of the transaction for the customer.
11. The ATM (queries / shall query) the customer if a further transaction is desired.
12. The customer signifies that he or she is finished to the ATM.
13. The ATM (displays / shall display) a standard greeting for next customer.

The example list of use case path interactions has several advantages over the story, scenario, and swim lane use case path examples from a requirements standpoint:

- Its standard structure as a list of interactions makes it easier to know how to produce it; just write down the interactions in chronological order.
- It avoids implementation constraints and superfluous test data.
- It is becoming clearer that interactions initiated by external actors are not requirements, but rather ancillary information, whereas interactions initiated by the system should be viewed as mandatory requirements.



However, the example list of use case path interactions still has several potential problems from a requirements standpoint:

- Preconditions and postconditions are still implicit. Yet, the path's preconditions are the reasons why that specific path was taken through the use case and thus critical to understanding the use case path. Also, some preconditions and postconditions must be valid at specific points in the path and should therefore be connected to their associated transactions.
- The differences between preconditions, triggers, resulting system actions, and postconditions are not yet obvious

4 BUILDING TEXTUAL REQUIREMENTS

Now, we take the preceding stories, scenarios, and paths through use cases and turn them into more properly specified requirements. This will be done by a combination of adding missing information and being more rigorous about how the information is specified.

A summary of the use case path can be produced by including the interaction's preconditions. The result is either how goal is achieved for normal paths or a failure to achieve the goal for exceptional paths.

Finally, a set of relatively complete textual requirements can be produced from the use case path interactions by using the following standard format: "If a trigger occurs when certain preconditions hold, then the system shall perform a required set of actions and shall be left in a required set of postconditions." Various quality requirements (e.g., performance, security) can be added to the triggers, preconditions, required actions, and postconditions as appropriate or else defined elsewhere.

Use Case: Withdraw Funds
Use Case Path: Customer Determined Amount of Funds Successfully Withdrawn
Result: The ATM enables authorized customers to successfully withdraw customer-determined amounts from their accounts.
Path-Defining Preconditions: 1) The ATM is in service. 2) The customers have been successfully identified and authenticated. 3) The requested amounts can be dispensed using the denominations stored by the ATM. 4) The ATM can securely communicate with the bank.

[1] Note that the ATM system itself only enforces a small number of bank rules (e.g., ensuring that the customer is successfully identified and authenticated, ensuring that the requested amount does not exceed single ATM withdrawal maximum, and ensuring that the ATM can dispense the requested withdrawal amount). The majority of the bank rules (e.g., requiring sufficient funds, not going over daily withdrawal

5) The bank computer approves the transactions.[1]
6) The bank's responses have not been corrupted.
7) The bank's responses can be verified to have come from the bank.

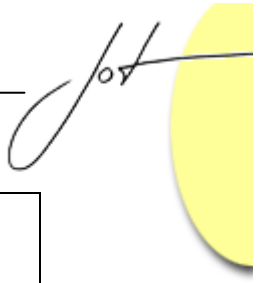
Textual Requirements

ID	Trigger	Preconditions	Actions	Postconditions
1	If a customer identifies and authenticates himself to the ATM	when the ATM is in service,	then the ATM shall verify the authentication and request that the customer chooses a transaction type (i.e., withdrawal, deposit, balance query, transfer)	and the ATM shall record the customer's authenticated identification.
2	If the customer selects to make a withdrawal from the ATM	when the ATM stores the customer's authenticated identification,	then the ATM shall request that the customer select one of the customer's account from which to make the withdrawal	and the ATM shall record the choice of 'withdrawal' made by the identified and authenticated customer.
3	If the customer selects an account from which to withdraw the funds	when the ATM stores the choice of withdrawal made by the identified and authenticated customer,	then the ATM shall request the customer to select or build the desired withdrawal amount	and the ATM shall record the choice of account from which to make the withdrawal.
4	If the customer requests a withdraw amount	when the withdrawal amount is a multiple of the denomination that the ATM can dispense and less than the ATM maximum individual withdraw amount, and the ATM stores the choice of withdrawal from a specific account made by the identified and authenticated customer,	then the ATM shall build a secure [2]withdrawal authorization transaction for the requested amount from the requested account and send the transaction to the bank	and the ATM shall record that the withdrawal authorization transaction was sent to the bank.
5	If the bank sends a secure notification to the ATM that the withdrawal request is authorized	when the ATM has not waited too long[3] for authorization for the customer's withdrawal request, the bank notification has not been	then the ATM shall dispense the approved funds and dispense a written record of the transaction to the customer	and the ATM shall debit the withdrawal amount from its recorded cash inventory and ask if the customer wishes another transaction.

maximums, account open, no hold on the account, etc.) are verified by the bank, which uses these rules to determine if the customer is authorized to make the requested withdrawal. Note also that communication between the ATM and the bank is typically minimized because of transaction costs so the ATM's actions can largely be grouped into two sequential sets: (1) building a secure transaction request and (2) acting on the bank's response.

[2] The definition of the term 'secure' can be found in separate security requirements.

[3] The definition of too long can be either defined here or found in separate performance requirements.



		corrupted, and the bank response can be verified as actually having come from the bank,		
6	If the customer signifies to the ATM that he is done,		then the ATM shall prepare for the next customer	and the ATM will no longer have possession of the customer's means of identification and authentication.

The following table uses a single common interaction to clearly show the difference between the four approaches. The story is simple and at a very high level of abstraction. The scenario provides much more detail at the level of a test case. The swim lane version of the use case path interactions is at an extremely high level of abstraction, whereas the interaction list version uses complete sentences. The final textual requirements version is complete in terms of its inclusion of trigger events, preconditions, mandatory system actions, and mandatory system postconditions.

Approach	Interaction
Story	The ATM forwards the request to the Bank, which approves the withdrawal.
Scenario	The ATM builds a withdrawal authorization transaction requesting the $60 from Mr. Smith's checking account that includes the date and time (8 October 2004 at 4:16PM), the ATM identifier (15856), the transaction type (withdrawal), the account type (checking), the account number (1593 4782 1594 1947), the transaction amount ($60), and the new available balance ($3,436.75). The ATM encrypts the transaction information, digitally signs the transaction, and sends it to the bank computer on the leased line connecting them.
Use Case Path Swim Lanes	Build secure withdrawal authorization transaction. Send withdrawal authorization transaction to bank.
Use Case Path Interaction List	The ATM shall build a secure withdrawal authorization transaction. The ATM shall send the secure withdrawal authorization transaction to the bank.
Textual Requirement	If the customer requests a withdraw amount when the withdrawal amount is a multiple of the denomination that the ATM can dispense and less than the ATM maximum individual withdraw amount, and the ATM stores the choice of withdrawal from a specific account made by the identified and authenticated customer, then the ATM shall build a secure withdrawal authorization transaction for the requested amount from the requested account and send the transaction to the bank and the ATM shall record that the withdrawal authorization transaction was sent to the bank.

Although the preceding textual requirements clearly will require more work to produce than the simple "requirements" of stories, scenarios, and use case path specifications, the

extra work should be quickly recovered based on the resulting time and effort saved during downstream activities such as architecting, design, implementation, integration, and test. Because the textual requirements are still in the natural language of the customer and stakeholders, they will still be easy for them to read, understand, and review. Any remaining difficulty in readability is due to the necessary increase in complexity required to deal with the actual complexity of the application domain, and this problem of readability can be mitigated by breaking the sentence into its for component parts. Also, it is better to understand that unavoidable complexity of complete requirements during requirements engineering rather than later in the project when the resulting errors are much more difficult and expensive to correct. Finally, if it is necessary, traditional means such as formal specification languages can be used to generate formally-specified requirements once the preceding semiformal requirements have been produced.

5 CONCLUSION

Although very valuable as requirements elicitation, analysis, and initial validation tools, stories, scenarios, and use case path specifications are typically inadequate for specifying requirements because they are incomplete, ambiguous, and therefore unverifiable. For example, they usually do not address preconditions and postconditions, which have a huge influence on the meaning of the requirements. Similarly, they do not tend to state the triggering events that cause them to be true. They also do not typically clarify the distinctions between requirements (i.e., what the system must do and what postconditions it must ensure) and ancillary information (e.g., triggering events produced by actors and preconditions that may or may not be ensured). This column has provided examples and guidance on how to transform stories, scenarios, and use case path specifications into complete, unambiguous, and verifiable textual requirements.

ACKNOWLEDGEMENTS

Many thanks to Gurinder Dhingra and Peter Capell for reviewing this column and providing helpful comments and suggestions. May their projects always be blessed with complete, consistent, correct, feasible, and verifiable requirements.

REFERENCES

[Boehm and Papaccio 1988] Barry W. Boehm and Philip N. Papaccio. "Understanding and Controlling Software Costs," *IEEE Transactions on Software Engineering*, vol. 14, no. 10, October 1988, pp. 1462-1477.

[Firesmith 2003] Donald Firesmith: "Specifying Good Requirements", in *Journal of Object Technology*, vol. 2, no. 4, July-August 2003, pp. 77-87. http://www.jot.fm/issues/issue_2003_07/column7

[HSE 1995] *Out of Control: Why Control Systems Go Wrong and How to Prevent Failure* (2nd Edition), Health and Safety Executive (HSE), 1995.

Disclaimers

The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

The views and conclusions contained in this column are solely those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Software Engineering Institute, Carnegie Mellon University, the U.S. Air Force, the U.S. Department of Defense, or the U.S. Government.

About the author

Donald Firesmith is a senior member of the technical staff at the Software Engineering Institute (SEI), where he helps the US Government acquire large, complex, software-intensive systems. Working in industrial software development since 1979, he has worked primarily with object technology since 1984 and has written 5 books on the subject. During the last four years, he has developed the world's largest (1,100+ webpage), free, and open source informational website of reusable process engineering components. Based on the OPEN Process Framework (OPF), it is located at http://www.donald-firesmith.com. Currently writing a book on the engineering of safety requirements, he can be reached at dgf@sei.cmu.edu.