

# Misuse Cases: Use Cases with Hostile Intent

Ian Alexander

**H**umans have analyzed negative scenarios ever since they first sat around Ice Age campfires debating the dangers of catching a woolly rhinoceros: “What if it turns and charges us before it falls into the pit?” A more recent scenario is “What if the hackers launch a denial-of-service attack?” Modern systems engineers can employ a misuse case—the negative form of a use case—to document and analyze such scenarios.<sup>1–3</sup> A misuse case is simply a use case from the point of view

of an actor hostile to the system under design. Misuse cases have many possible applications and interact with use cases in interesting and helpful ways.

## Eliciting security requirements

Security requirements exist because people and the negative agents that they create (such as computer viruses) pose real threats to systems. Security differs from all other specification areas in that someone is deliberately threatening to break the system. Employing use and misuse cases to model and analyze scenarios in systems under design can improve security by helping to mitigate threats.

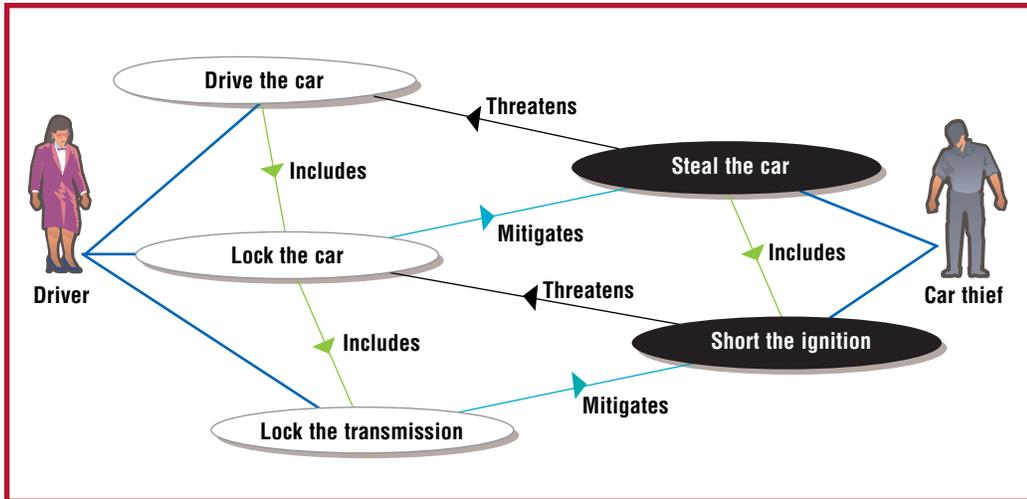
Some misuse cases occur in highly specific situations, whereas others continually threaten systems. For instance, a car is most likely to be stolen when parked and unattended, whereas a Web server might suffer a denial-of-service attack at any time.

You can develop misuse and use cases recursively, going from system to subsystem levels or lower as necessary. Lower-level

cases can highlight aspects not considered at higher levels, possibly forcing another analysis. The approach offers rich possibilities for exploring, understanding, and validating the requirements in any direction. Drawing the agents and misuse cases explicitly helps focus attention on the elements of the scenario.

Let’s compare Figure 1 to games such as chess or Go. A team’s best strategy consists of thinking ahead to the other team’s best move and acting to block it. In Figure 1, the use cases appear on the left; the misuse cases are on the right. The misuse threat is car theft, the use-case player is the lawful driver, and the misuse-case player the car thief. The driver’s freedom to drive the car is at risk if the thief can steal the car. The driver must be able to lock the car—a derived requirement—to mitigate the threat. This is at the top level of analysis. The next level begins when you consider the thief’s response. If he or she breaks the door lock and shorts the ignition, this requires another mitigating approach, such as locking the transmission. In

Misuse cases—a form of use cases—help document negative scenarios. Use and misuse cases, employed together, are valuable in threat and hazard analysis, system design, eliciting requirements, and generating test cases.



**Figure 1. Use/misuse-case diagram of car security requirements. Use-case elements appear on the left; the misuse cases are on the right.**

this way, what begins as an apparently simple hardware-only design might eventually call for software subsystems. We can analyze more complex threats to e-commerce and other commercial systems in the same way.

Figure 1 also shows that threat and mitigation form a balanced zigzag pattern of play and counterplay. This “game” is reflected in an inquiry cycle style of development. Both use and misuse cases can include subsidiary cases of their own kind, but their relationships to cases of the opposite kind are not simple inclusion. Instead, misuse cases threaten use cases with failure, and appropriate use cases can mitigate known misuse.

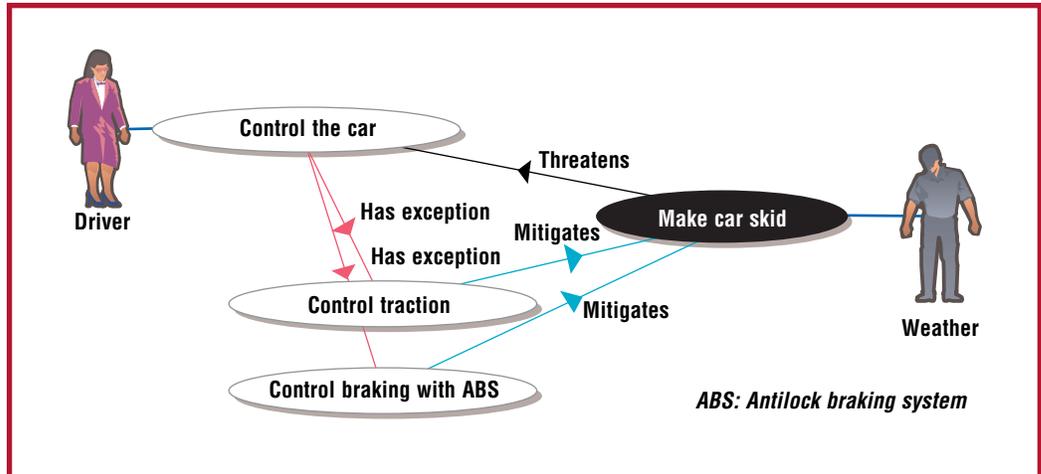
After you know the mitigation approaches, you can proceed with development by trading off user requirements (countering misuse) and system constraints (such as cost, weight, size, and development schedule). When you don’t know the mitigation approaches, you can proceed with both development and use/misuse-case analysis, which is initially top-down. You can identify, study, prototype, evaluate, and select mitigation approaches. When mitigations demand new subsystems or components, the new devices in turn usually engender new types of threat. You can analyze these threats to evaluate the need for further countermeasures. Analysis and design become intertwined as design choices crystallize and the system requirements become more specific.

Mitigation measures rarely neutralize security threats. Thieves pick locks and break into systems through unsuspected access paths. However, partial mitigations are still useful as long as they afford a realistic increase in protection at reasonable cost. Neutralizing all possible threats is wishful thinking and can’t be stated as a requirement. For example, some drivers leave their engines running when they leave their vehicles for short periods. Can designers protect against this sort of misuse? There are plainly more cases to consider than those in Figure 1.

### Eliciting safety requirements from failure cases

Failure Mode Effects Analysis and related techniques traditionally evaluate safety threats. They relate possible causes to possible effects, make assumptions and measurements, and calculate probabilities, leading to an assertion that the system is sufficiently safe. However, the analysis depends on accurately identifying possible failure modes. No work is done on unimagined failures. Therefore, a technique like misuse-case analysis that seeks to identify possible causes of system failure can feed Failure Mode Effects Analysis with plausible threats to systems.

Karen Allenby and Tim Kelly describe a method for eliciting and analyzing functional safety requirements for airplane engines by employing a type of use case.<sup>4</sup> Because functional hazards should be intimately derived



**Figure 2. Eliciting and analyzing car safety requirements through use and misuse cases. Weather is the negative agent.**

from system requirements, they see a need for a way to derive hazards from known system functions, proposing use cases for this purpose. However, they don't suggest associating negative agents with use cases. Their method tabulates the failures and the causes, types, effects, and possible mitigations. Mitigations often involve subsystems, implying a recursive decomposition. However, because their use cases describe potentially catastrophic failures and their effects, calling them misuse cases or failure cases seems reasonable.

Safety requirement scenarios don't necessarily involve a human agent (though this is possible through sabotage or terrorism). The negative agent is usually the failure of a safety-related device such as a car brake or an inanimate external force such as dangerous weather. When drivers lose control on ice-covered roads, it can be advantageous to anthropomorphize the weather as an agent "intending" to make the car skid (see Figure 2). This approach captures the force of an easily understood metaphor to emphasize the requirement for control in different weather conditions. Because human language is metaphoric, it is wise to express requirements in this way.<sup>5</sup>

Misuse cases can help elicit appropriate solutions in the form of subsystem functions such as traction and automatic braking controls. These functions handle exception events (such as skidding) through carefully programmed responses. These responses satisfy requirements that can be written as exception-handling scenarios. Such scenarios

can either form the exception subsections of larger use cases (such as "Control the car") or can be pulled out into exception-handling use cases in their own right, as illustrated in Figure 2, in which explicit "has exception" links connect the new use cases.

Once you have identified such exception-handling use cases, you don't need misuse cases except to justify design decisions. Justifications are important to protect design elements and requirements from being struck down during reviews, especially when time and money are scarce. When necessary, you can employ use-case tools to readily display or hide misuse cases (see the "Tool Support for Use and Misuse Cases" sidebar).

As with security requirements, you can develop safety requirements recursively, going from system to subsystem levels, or lower. Again, bottom-up and middle-out approaches are possible. The explicit presence of misuse cases should let domain experts validate safety requirements more easily and accurately.

### **Interplay of design, functional, and nonfunctional requirements**

The examples given so far illustrate how misuse cases interplay with system and subsystem functions. But you can also see a misuse case as a hostile action that serves as a functional goal, such as "Steal the car." Such threats are traditionally handled by writing nonfunctional requirements and standards governing quality, such as "The

## Tool Support for Use and Misuse Cases

Use and misuse cases are fundamentally textual structures and can certainly be handled as simple word-processing documents without special tools. The diagrams are also often quite simple, with easy-to-draw notations. However, a requirements tool specialized for use cases can keep numerous cases organized and consistent through the life of a large project. A tool can automatically produce diagrams and metrics, check consistency, and guide requirements elicitation while providing more or less detailed templates.

I have produced such a toolkit for my own use called Scenario Plus. It's a free set of add-ons for Telelogic's DOORS requirements-management tool that retains traceability, archiving, and data-exchange features. Readers can download the toolkit from the Web at [www.scenarioplus.org.uk](http://www.scenarioplus.org.uk). The figures in this article reflect some of its graphical capabilities. More important is the way the templates organize use-case text and handle links between cases.

The interplay of use and misuse cases consists of four combinations, namely relationships to and from each kind of case. Table A shows the four-part rule governing the automatic creation of relationship types according to the sources and targets of relationships between use and misuse cases. For example, the toolkit sees a link from a misuse case to a use case as a threat and labels it "threatens."

The toolkit implements this mechanism to construct links. The requirements engineer names the target case within a scenario step in either the primary scenario or an alternative path of the source case, and underlines the name. The analysis tool then scans for underlined phrases and tries to match them with existing use- or misuse-case names

| ID    | Use Cases  | Links to Included Use/Misuse Cases                 |
|-------|--|--|
| UC-30 | <b>2.1.3 Lock the Car</b>  |  |
| UC-31 | <b>2.1.3.1 Primary Scenario</b>  |  |
| UC-35 | System automatically <u>Locks the Transmission</u> to prevent the Car thief from <u>Stealing the Car</u> . | UC-49 Lock the Transmission<br>UC-70 Steal the Car |

**Figure A. Automatic creation of links between misuse and use cases through searching for underlined use case names with simple fuzzy matching.**

car shall be constructed to the intrusion resistance (quality) defined in STD-123-456."

Does this mean that use cases define functions and misuse cases define nonfunctional requirements? Possibly, but only if you look at the situation traditionally. Another way of looking at the role of misuse cases is to observe that the typical response to a threat is for the designers to create a subsystem (such as a lock), whose function (preventing intrusion) is to mitigate that threat. In short, you can say that the misuse case elicits the subsystem function (see Figure 3).

Actually, both use and misuse cases can

help to elicit functional and nonfunctional requirements, if you accept the distinction. However, Table 1 makes clear the dynamic interplay between different types of requirements and that one person's nonfunctional attribute is another person's distinct subsystem function. If so, perhaps use cases can cover all types of requirements or even (as some have claimed) replace them.

### Eliciting "-ility" requirements

Misuse cases can help document the types of nonfunctional or quality requirements that engineers often call the "-ilities": reliability, maintainability, portability, and so on.

**Table A**

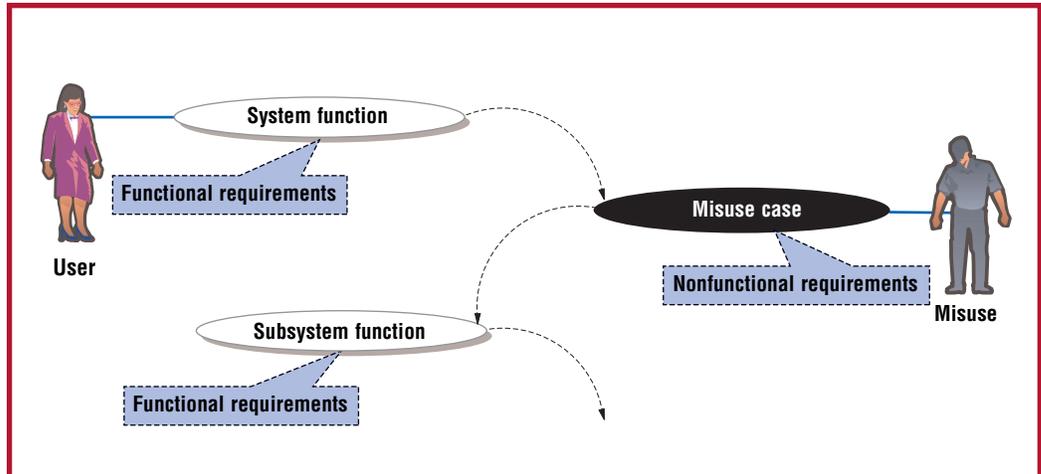
**Rule governing creation of relationships between use and misuse cases**

|             | Case type | Source case |           |
|-------------|-----------|-------------|-----------|
|             |           | Use         | Misuse    |
| Target case | Use       | Includes    | Threatens |
|             | Misuse    | Mitigates   | Include   |

(their goals). For example, the tool fuzzily matches the phrase "Stealing the car" with the misuse-case goal "Steal the car" (see Figure A). If it finds no match, the tool asks the user whether it should create a new case. The tool then links the cases according to the rule defined in Table A.

This mechanism enables engineers to write use-case steps simply and readably in English. Users can display the created links, view them on the diagram as relationships between use and misuse cases, and navigate them as usual in the requirements database.

Users can choose to show or hide misuse cases (along with their negative agents and relationships). The tool automatically draws the misuse-case agents on the right side of the diagram to keep them apart from ordinary actors.



**Figure 3. Interplay of use and misuse cases with functional and nonfunctional requirements.**

### Reliability, maintainability, and portability

Whereas security threats stem directly from genuinely hostile agents, you can elicit and analyze reliability requirements as threats caused by agents that are not necessarily intelligent. Such agents include human error, storms, design errors (such as software bugs), and interference on telecommunication links, which can cause software crashes and other types of failure.

Maintainability and portability requirements can also benefit from use/misuse-case treatment. Here, the negative agents could be an inflexible design or a wired-in device dependence. These simple examples illustrate that—contrary to the view that use cases only discover functions—use/misuse-case analysis can be applied to many types of requirements.

### Other “-ilities”

You can also apply misuse-case solutions to usability, as when a novice operator confused by the user interface becomes a negative agent. You can also apply the approach to hardware aspects such as storability and transportability to solve the threats of icy

weather and rough handling to delicate components.

Here, simplicity is a virtue. Misuse cases involve fundamental situations affecting many types of systems. If misuse cases were expressed only in terms of complex calculus, they might have limited applicability. Their simplicity suggests robustness in that they form strong arguments in favor of designing systems in particular ways.

Although I wouldn’t advocate blindly creating hundreds of misuse cases for all possible requirements, especially when the challenges in question are well known in advance, the technique does appear to be widely applicable to elicit and justify different types of requirements. Does this mean you can model use cases and forget about writing nonfunctional requirements and constraints? The jury is out. Some things that are easy to describe in a few words—the system must be delivered in two years, the unit cost must be no more than \$250, and the mean time between failures must be at least 10,000 hours of operation—are not ideally written in scenarios. Although it is often helpful to think through scenarios to elicit constraints

**Table I**

### Applicability of use and misuse cases for eliciting different types of requirements

| Elicitation through       | Use case            | Misuse case          |
|---------------------------|---------------------|----------------------|
| Functional requirement    | Important mechanism | Useful, but indirect |
| Nonfunctional requirement | Possible            | Important mechanism  |

and nonfunctional requirements, they are often better summarized as statements.

## Eliciting exceptions

An exception-handling use case can describe how the system under design will respond to an undesirable event to prevent a possibly catastrophic failure. The response can lead to the resumption of normal operations or to a safe shutdown, as when a train stops after it passes a danger signal. Misuse-case analysis is one way to hunt down possible exceptions. Sometimes it is worth documenting misuse-case scenarios in detail; other times, just the name of the misuse case can identify gaps in system requirements.

A clear relationship exists between exception classes and the negative actors who initiate misuse cases. These classes are simply-named categories of exception, generic situations that cause systems to fail. You can generate candidate exception scenarios and elicit requirements to prevent system failure from a proven list of exception classes. However, such lists are rare.

You can also employ simple requirements templates to elicit exceptions. Good templates help elicit and validate requirements simply because they remind us of questions to ask, such as “Could there be any portability requirements here?” To elicit exceptions, you step through all the scenarios in the use cases, asking, “Could anything go wrong here?” This is effective and general, but not guaranteed to find all possible exceptions.

For each template heading or misuse case, there can be several requirements, but if only one is found—or if you confirm that there is no requirement—the approach is worthwhile. In other words, a template, like a misuse case, implies an inquiry method.

However, devising threats and negative agents with misuse cases is sometimes a more powerful technique than simply stepping through a template or thinking about exceptions, for several reasons.

- Inverting the problem from use to misuse opens a new avenue of exploration, helping to find requirements that might have been missed.
- Asking “Who might want this to go wrong?” and “What could they do to make this go wrong?” in the misuse-case approach contributes to searching sys-

tematically for exceptions by using the structure of the scenarios themselves as a guide, with a more specific intent than a plain search for exceptions provides.

- Knowing explicit threats offers immediate justification for the search and indicates the priority of the discovered requirements.
- Personifying and anthropomorphizing the threats adds the force of metaphor, applying the powerful human faculty of reasoning about people’s intentions to requirements elicitation.
- Making elicitation into a game makes the search enjoyable and provides an algorithm for it—“Team 1 outthinks Team 2’s best move, and vice versa.” The stopping condition is whether the threat’s size and probability justify the cost of the mitigation, given its probability of defeating a threat. There is an obvious parallel here with cost-benefit analysis.
- Providing a visual representation of threat and mitigation makes the reasoning behind the affected requirements immediately comprehensible.

I find both templates and a scenario-directed search for exceptions useful in requirements elicitation. Misuse cases offer an additional way toward that holy grail, the “complete” set of requirements.

## Eliciting test cases

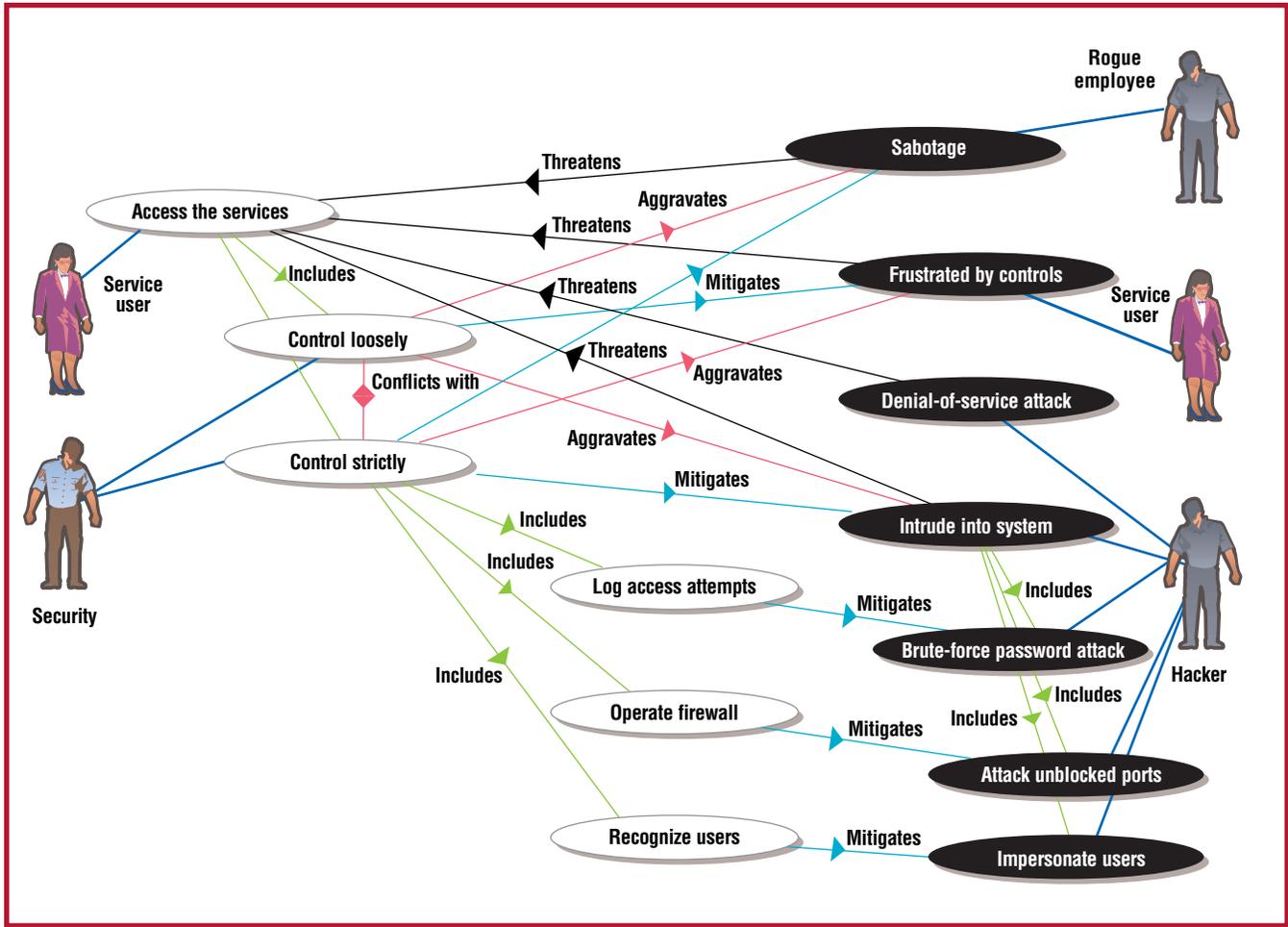
Any scenario can lead to a test case. Good testing goes beyond happy-day scenarios to explore boundary conditions and exceptions.

Misuse cases can clearly help identify exceptions and failure modes, any of which might be worth testing or verifying by other means. The habit of thinking out negative scenarios is arguably an essential skill for the test engineer.

Products of use/misuse-case analysis that can contribute to effective test planning include

- Specific failure modes (especially useful for real-time, embedded, and safety-related systems)
- Security threats (especially useful for distributed commercial and government systems)
- Exception-handling scenarios (always useful, often directly translating to test scripts)

**Good testing goes beyond happy-day scenarios to explore boundary conditions and exceptions.**



**Figure 4. Use and misuse cases for Web portal security.**

A test engineer could view misuse cases as existing purely to ensure better system testing. A quality engineer could equally well argue that their purpose is to improve the quality of delivered systems.

### Design trade-offs with misuse cases

An important element of system design is to satisfy any conflicting user demands. The situation is complicated by the fact that each design choice opens up new possibilities for both use and misuse. Designers must therefore trade off one option against another.

For example, Web portal users must be able to access the provided services. This access can be threatened by many security assaults, from sabotage by rogue employees to sophisticated attacks by hackers. Security itself can also threaten system use if it is so strict that it frustrates lawful users and leads

them to seek alternative services. On the other hand, loose controls that are more comfortable for such users invite misuse. Figure 4 illustrates these dilemmas by adding “aggravates” and “conflicts with” relationships between cases.

Once designers identify the threats, mitigations, and possible conflicts between design options, they can make informed choices in light of system goals and requirements. Misuse cases, their relationships, and the use cases that ultimately are not implemented (say, loose security control) form part of the justification for system design. They could all be discarded, but only at the risk of repeating the same trade-off arguments during the next system upgrade. Misuse cases thus have a definite role to play during system design and in addressing design issues and trade-offs during in-service operations and maintenance.

## Putting use and misuse cases to work

My colleagues and I at DaimlerChrysler are investigating the appropriate forms of use cases to help recycle reuse requirements for control software in cars.<sup>6-8</sup> Use cases can assist not only software development, which is the domain addressed by popular accounts,<sup>9,10</sup> but also hardware and interfaces of all kinds. A large system comprising many subsystems, such as a car, must be analyzed in successively greater detail in subsystem models to cope with the complexity of the functionality. Use-case models can also help express the purpose of system and subsystem features to different audiences.

Engineers can apply use and misuse cases at any system level. This approach dovetails well with both the recursive decomposition inherent in the software engineering life cycle and with participative, inquiry cycle approaches that invite groups of people to solve problems in stages by asking suitable questions. Use and misuse cases help users and engineers communicate about development issues. For example, a project developing automotive software to control audio entertainment and safety announcements must not only consider the driver's entertainment needs—a basic use case—but also allow traffic announcements to override them. Indeed, if a safety announcement is to override both entertainment and traffic announcements, the software system must consider interactions between a range of subsystems.

Cars increasingly rely on software to perform functions previously provided by hardware. When a car has separate radio, compact disc player, and warning systems, the radio and CD player can't interfere electronically with the warning system, but neither can the entertainment be faded under the warning system's control to let users hear the warnings. Therefore, designers are beginning to reduce the radio and CD player to minimal hardware and implement their control functions in software. This integration not only permits desirable new behavior such as fading audio but also opens the door for undesirable interactions between subsystems. So, the entertainment subsystem must inherit whole-car scenarios and develop its own more detailed use and misuse cases to handle them. This process is the use-case equivalent of system decompo-

sition and information hiding. Use/misuse-case analysis can contribute to each stage of system development, alongside other processes that identify objects and define messages to be passed between them.

## Getting started with misuse cases

The best way to get started is with a small, informal workshop in which you and other stakeholders identify negative agents that might threaten your system. You then brainstorm a list of misuse cases for each agent. If you already have use cases, search for ways in which misuse cases could threaten them. This might lead you to create more misuse cases, or it might lead to relationships between existing use and misuse cases. You can then consider how to mitigate the misuse cases, should they arise. This can lead to creating new subsystem functions in the form of use cases. A conflict analysis might then be appropriate. You might also want to consider the costs and benefits of differing design approaches and alternative subsystem functions.

Some misuse cases require little documentation; others might need a fully worked-out primary scenario for a use case. Subsystem use cases created in this process will need to be documented as usual and examined to see whether they are susceptible to their own misuse cases. This can require several iterations. Later in the project, you might want to consider all misuse cases as candidate test cases to ensure the system under design behaves as expected.

Large projects with an established methodology might need to prepare misuse-case guidelines to include with the other standards. These guidelines should cover applying misuse cases to requirements elicitation, analysis, design trade-offs, and verification.

**T**he interplay of use and misuse cases during analysis can help engineers elicit and organize requirements more effectively. Functional and nonfunctional requirements thus elicited similarly interplay with each other—and with design decisions—throughout the design process. Thinking about misuse cases will probably pick up numerous issues that could have caused systems failures or added time and ex-

**Some misuse cases require little documentation; others might need a fully worked-out primary scenario for a use case.**

## About the Author



**Ian Alexander** is an independent consultant and trainer who specializes in requirements engineering. He collaborates with DaimlerChrysler Research and Technology on reusing requirements in different series of cars. His principal research interest is improving the requirements engineering process by modeling business goals, processes, constraints, and scenarios. He received an MA in natural sciences from Cambridge University and an MSc in computer science from Imperial College, London. He serves on the committee of the BCS Requirements Engineering Specialist Group and heads the Requirements Engineering section of the IEE Professional Network for Systems Engineers. He is a Chartered Engineer and an IEE member. Contact him at [iany@easynet.co.uk](mailto:iany@easynet.co.uk).

pense to development projects. Use/misuse-case analysis is not only promising but also complements existing analysis, design, and verification practices. ☞

## References

1. I. Jacobson et al., *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, Boston, 1992.
2. G. Sindre and A.L. Opdahl, "Eliciting Security Require-

- ments by Misuse Cases," *Proc. 37th Conf. Techniques of Object-Oriented Languages and Systems, TOOLS Pacific 2000*, 2000, pp. 120–131.
3. G. Sindre and A.L. Opdahl, "Templates for Misuse Case Description," *Proc. 7th Int'l Workshop Requirements Eng.: Foundation for Software Quality (REFSQ 2001)*, 2001.
4. K. Allenby and T. Kelly, "Deriving Safety Requirements Using Scenarios," *Proc. 5th Int'l Symp. Requirements Eng. (RE 01)*, IEEE CS Press, Los Alamitos, Calif., 2001, pp. 228–235.
5. C. Potts, "Metaphors of Intent," *Proc. 5th Int'l Symp. Requirements Eng. (RE 01)*, IEEE CS Press, Los Alamitos, Calif., 2001, pp. 31–38.
6. I. Alexander, "Use/Misuse Case Analysis Elicits Non-Functional Requirements," to be published in *Computing and Control Eng. J.*
7. I. Alexander and T. Zink, "Systems Engineering with Use Cases," to be published in *Computing and Control Eng. J.*
8. I. Alexander and F. Kiedaisch, "Towards Recyclable System Requirements," *Proc. 9th IEEE Conf. and Workshop Eng. Computer-Based Systems*, IEEE Press, Piscataway, N.J., 2002.
9. A. Cockburn, *Writing Effective Use Cases*, Addison-Wesley, Boston, 2001.
10. D. Kulak and E. Guiney, *Use Cases: Requirements in Context*, Addison-Wesley, Boston, 2000.

For more information on this or any other computing topic, please visit our Digital Library at <http://computer.org/publications/dlib>.

# CALL FOR Articles and Reviewers

IEEE  
**Software**

## The State of the Practice of Software Engineering

This special issue will focus on the actual current practice of software engineering in industry—what software engineering practice is, and, by implication, is not. The issue will present an accurate, baseline view of practice upon which other practitioners, and software engineering researchers, can build as they prepare us for the future. It will describe current practice, not judge it. This issue will not, for example, focus on "best practices."

Submissions must present information that accurately characterizes software engineering as currently practiced. Surveys, for example, are appropriate; case studies, unless their findings can be generalized to a broad spectrum of practice, are not. Studies of particular domains of practice are welcome. The issue will consist of contributed papers as well as invited articles from people intimately familiar with software practice.

Submit articles online at <http://cs-ieee.manuscriptcentral.com>.

Publication: November/December 2003

Submission deadline: 15 April 2003

Guest Editor: Robert Glass, [rlglass@acm.org](mailto:rlglass@acm.org); 1416 Sare Rd., Bloomington, IN 47401