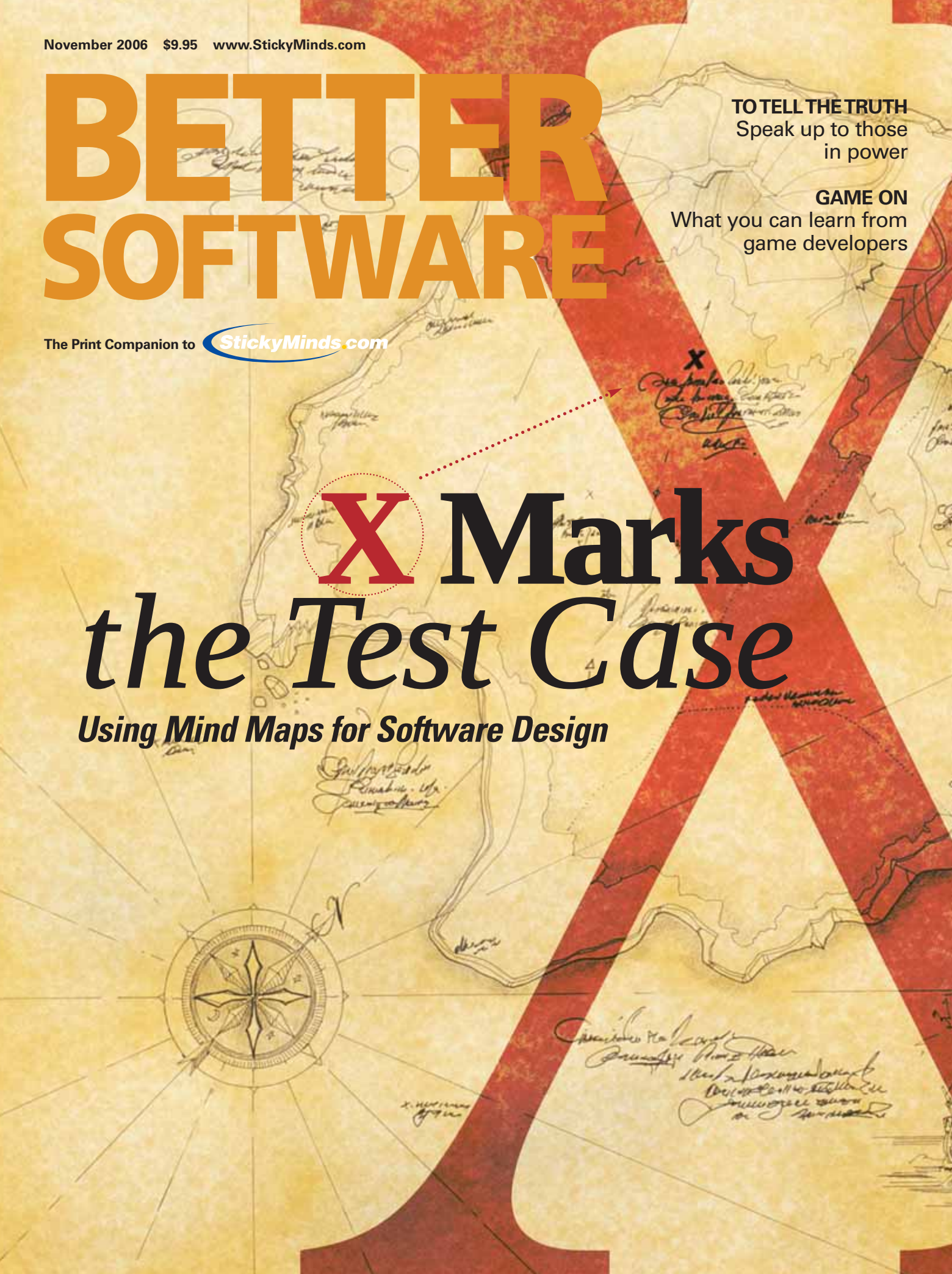
November 2006 $9.95 www.StickyMinds.com

BETTER SOFTWARE

The Print Companion to StickyMinds.com

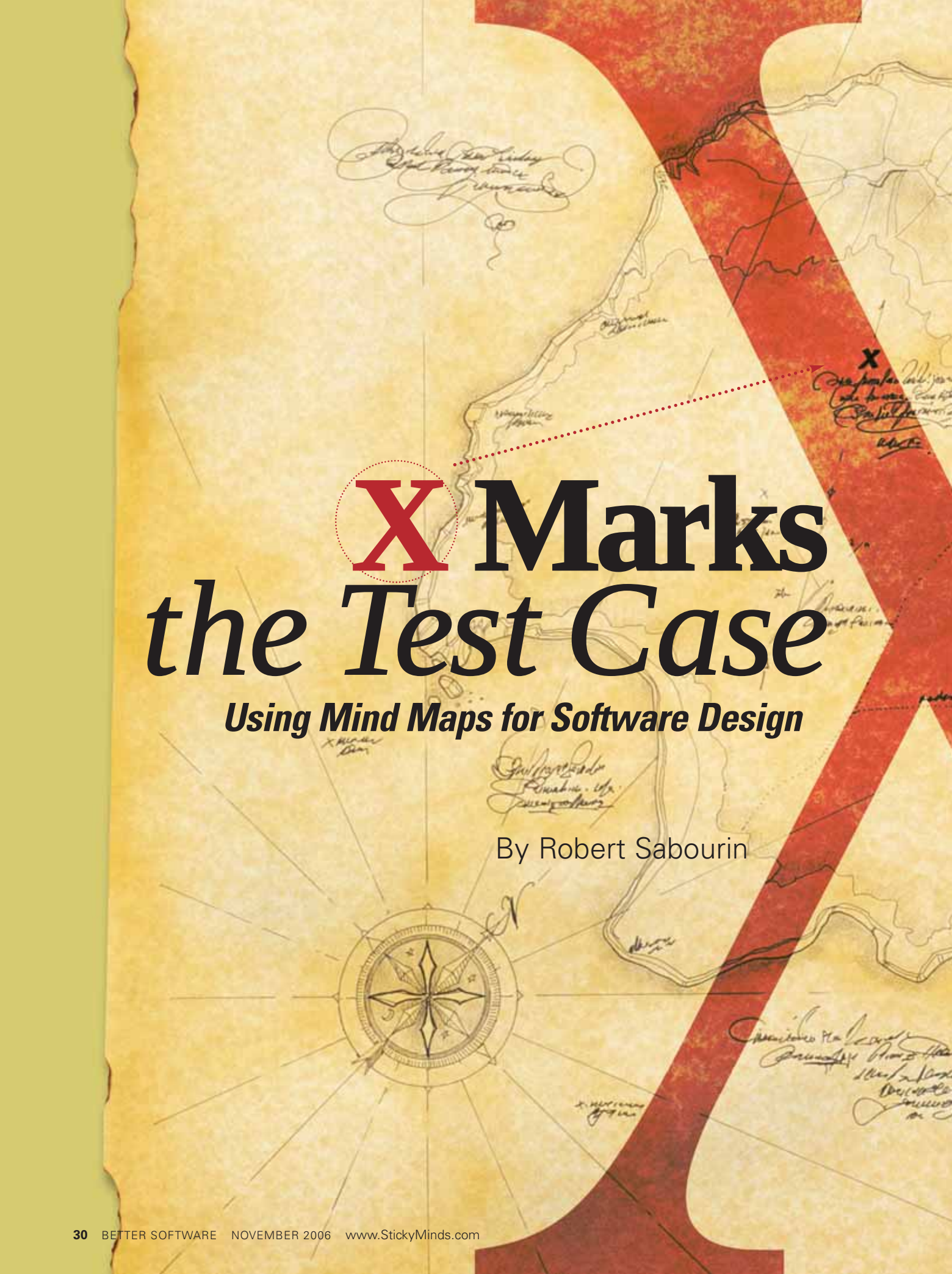
TO TELL THE TRUTH
Speak up to those in power

GAME ON
What you can learn from game developers



X Marks the Test Case

Using Mind Maps for Software Design



X Marks the Test Case

Using Mind Maps for Software Design

By Robert Sabourin

Mind maps are a way to explore and document ideas and their relationships in a simple diagram. Important concepts are recorded as words or pictures and connected with lines indicating their relationships. While in college, Tony Buzan developed mind maps to save time in creating and reviewing notes (see the StickyNotes for more information). Later, he popularized mind maps through lectures, books, and courses. Today mind maps are used to improve memory, reading skills, note taking, creativity, performance, and brainstorming.

I first learned about mind maps from my son Mark when he was in grade school. Mark was taught how to use mind maps to describe the plot, characters, theme, and events of stories read in class. The first map he showed me was for the children's classic *Charlotte's Web*. I was quite impressed that a simple drawing could meaningfully describe so many concepts. I started to dabble with mind maps to solve some of the software engineering projects I had at the office.

You can create mind maps with just pencil and paper, but a number of automated tools are available (see the StickyNotes for links). FreeMind is a widely available open source tool that provides a quick entry into the mind-mapping universe. I use FreeMind when I teach testing to my undergraduate software engineering students and to testing professionals in the field. When I am designing commercial test cases, I often use Mind Manager from Mindjet. Mind Manager is reasonably priced and allows me to develop very complex mind maps that can be easily integrated into other documents and shared across other projects.

Mind mapping is a great technique to use when you've run through the standard test design approaches, but you're still looking for that "killer test case." Mind maps help generate great test ideas and get the creative juices flowing. This article shows how I use mind maps to help in three different test design areas: equivalence classes (as part of domain testing), usage scenarios, and quality factors.

Mind Maps to Help Define Equivalence Classes

One of the more common test case design techniques I use is called equivalence class partitioning. I learned this technique in *The Art of Software Testing* by Glenford Meyers. Today it is widely taught as a key element of domain testing (see the StickyNotes for some additional resources).

An equivalence class is a set of tests that test the same thing. All tests in an equivalence class expose the same bugs. Whenever I test an application, I use equivalence classes to help choose what data and conditions to exercise under test. It provides good test coverage while reducing the number of test cases.

I use mind maps to visually represent equivalence classes so I can decide where to focus my testing. Normally I try to identify as many classes as I can, and then, depending on how much effort I choose to invest, I select the equivalence classes that offer me the most value as a tester.

I use a three-step approach to create a mind map for an equivalence class:

- Identify the variables.
- Identify classes based on application logic, input, and memory (AIM).
- Identify invalid classes.

Identify the variables

I create a mind map for each variable under test. The variable is the center of the mind map, and it is important that we choose variables that are relevant to what the application must accomplish. In my mind map, I choose a variable from all the possible things that could or should influence the behavior of the application I am testing. My tests will be questions related to how the application will behave given different values for that variable.

A variable can be a parameter entered by the user or a condition that may influence the application under test. It is important to note that variables can be a combination of two or more input fields, conditions, or states. For example, a user filling out an insurance application needs to enter the starting and ending date of the claim period, so he selects the date range by entering two parameters. Therefore, I can use the variable "date range" in my test case design rather than two different variables for the starting and ending dates. Usually parameters required in a group can be combined, and the group of parameters is a variable!

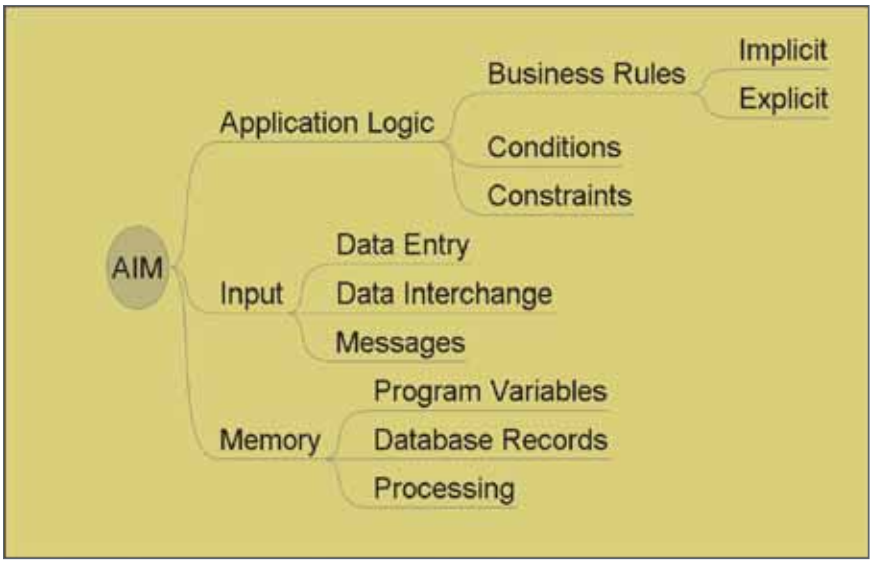


Figure 1: The AIM model

Identify classes based on AIM

The AIM model in Figure 1 is the basic template I use to begin defining an equivalence class mind map for a variable. Note that I consider explicit (documented) and implicit requirements when identifying classes based on the application logic.

I position the variable in the center of the map, then I draw three branches with end nodes labeled Application Logic, Input, and Memory.

On the Application Logic or "A" node, I place one branch for each class related to the application's expected behavior based on business rules or processing algorithms. If I were testing income tax software and the variable in question was the income level, then I would define at least one application logic class for each income level identified in the requirements. In business applications, "A" classes are those defined by ranges of values.

On the Input or "I" node, I place equivalence classes based on the way data is input. There are many ways data can get to an application, and I use the "I" class to focus on the ways the application will deal with data before it is processed. "I" classes could be the length of an input field based on the number of characters allocated in a form. For example, a form may have an income value field that allows six characters. So classes can be numbers that are represented in six digits or fewer. If we are restricted to integers, this gives a range from 000000 to 999999. These values are based on how the data is entered, not how the data is processed. If different numeric notations are allowed, we can also have classes based on the different notations (for example, with or without numeric edit symbols such as a thousands separator, decimal point, or currency symbol). If the application reads data from a file, the field formats for the file will define a selection of potential "I" classes.

On the Memory or "M" node, I place equivalence classes based on the way data is processed or stored in the program. To identify "M" classes, you must have some knowledge of how the software is being written. When we base tests on the actual source code or low-level design, we are said to be doing white box testing. I often consult developers, design documents, or database schemas to identify "M" classes for variables being tested.

Different "M" classes can be defined based on the variable types from the database schema. I have seen different variable types used in the database and program logic, and I encourage you to explore and identify classes based on both.

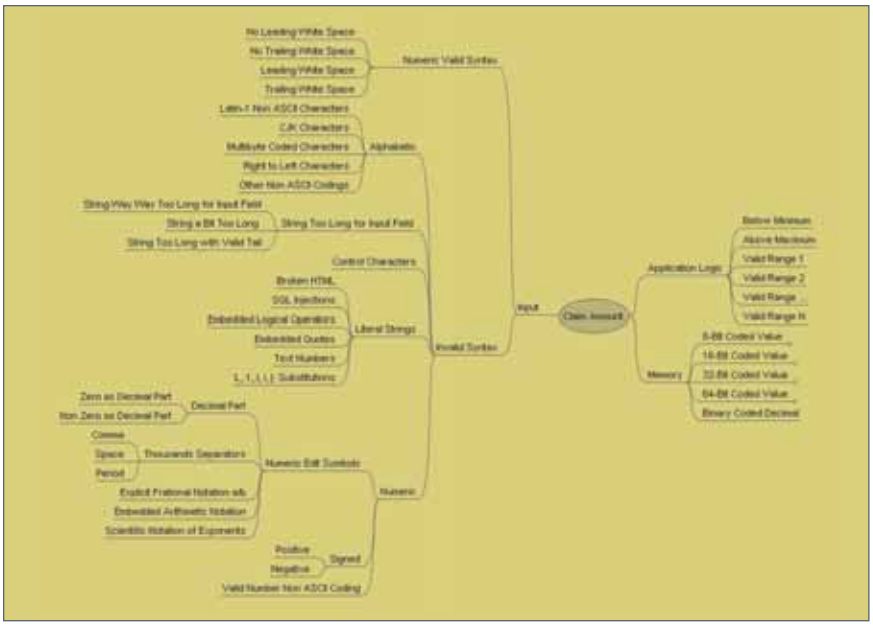


Figure 2: Equivalence class mind map

Let's examine a typical equivalence class mind map (see Figure 2). This mind map describes a set of equivalence classes related to a user's entering a claim amount into a Web-based insurance application.

I chose the variable Claim Amount as the center of the mind map. Branches that radiate from Claim Amount break down possible values based on AIM.

Classes defined in Application Logic relate to ranges of values. Note that the invalid classes Below Minimum and Above Maximum are also included.

Input classes are defined in branches of valid and invalid syntax. Under Valid Syntax I have included data that is entered with or without leading or trailing spaces. Under Invalid Syntax I have included classes based on different data entered, codings from different locales, long strings, control characters, literal strings, and different numeric notations. While I begin by listing as many possibilities as I can, I may remove some later if I feel the risk is low and the testing effort isn't justifiable.

Memory-related classes are included for different representations of numeric data coded as integers.

I can review the map with developers, customers, and project team members to make sure the handling of Claim Amount is consistent with their expectations and any project requirements.

Figure 3 shows a mind map that identifies equivalences related to a date field. The "A" classes relate to whether the date is in the past, present, or future. The "I" classes relate to the syntax of the date as entered by a user. No fields related to "M" are included here.

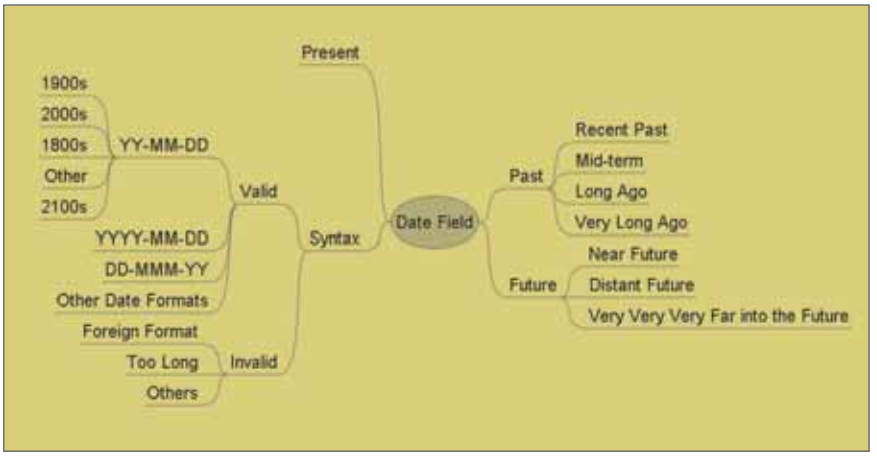


Figure 3: Map identifying equivalences related to a date field

Identify invalid classes

For each node of the mind map I often create two branches, one for valid classes and the other for invalid classes.

For application logic classes, I define valid classes for ranges of values established within the specifications of the business requirements. I define invalid classes for values outside the ranges established within the business requirements.

For input classes, it's valuable to look at invalid classes based on invalid type, invalid syntax, and invalid length of variables. An invalid type occurs when a variable is expected to have one type of data but a user enters an incorrect type. A common example is a user typing alphabetic text into a numeric field. Another situation is a user entering decimal values into an integer input field.

Invalid syntax often occurs in processing values such as date, time, and money. For example, many possible date formats exist depending on how the day, month, and year are represented and the order in which they are entered. For example DD-MM-YY, MM-DD-YY, and YY-MM-DD are three different possibilities that could be used for a date-input field. Depending on the application under test, I may want to choose at least one class of each possible date format a user could enter.

Invalid length entries can trigger bugs, such as buffer overflows and security risks. I define invalid classes for string lengths that are a bit too long, a lot too long, and drastically too long. For example, what would happen if an input field designed to process up to eighty characters gets a string of 100 million characters pasted into it? Does the input process correctly identify the invalid data, or does it attempt to process it?

Mind Maps to Identify Usage Scenarios

John Kennedy's inspirational inaugural address challenged Americans to "… ask not what your country can do for you …" but rather to "… ask what you

can do for your country." Usage scenarios are a similar challenge to your software: "Test not only what your application can do for your user but also what your user can do with your application."

Usage-scenario test cases walk through what a user will typically do with the system from beginning to end and exercise many different capabilities of an application. Almost all the bugs I find that interfere with typical usage scenarios are high priority. When I am squeezed for time, I focus on usage-scenario testing.

Mind maps help me identify the ways in which the application under test helps users achieve their goals. I first identify the types of users, and then I identify three different types of activities for each user—normal work activities, crisis or extreme activities, and infrequent activities.

Here is an example I use when I teach test-case design: Imagine you are testing the Wrap-O-Matic, a machine designed to wrap chocolates. The machine takes an input stream of chocolates and provides as output beautifully wrapped and packaged chocolates. If I want to make sure I tested all possible usage scenarios, I would create the mind map in Figure 4 to help guide my test-case designs.

Note that for the Wrap-O-Matic I have identified many different system users. An operator makes sure the machine is running, an auditor reviews reports about the wrapping-production status, a loader makes sure the Wrap-O-Matic has all consumables and chocolates required, an unloader takes wrapped chocolates away, a health inspector makes sure wrapped chocolates conform to the health code, and a maintainer makes sure the system is in tip-top shape.

Using this mind map to define usage scenarios that could be tested, I would choose which scenarios to test and which to ignore based on the risk associated with each one.

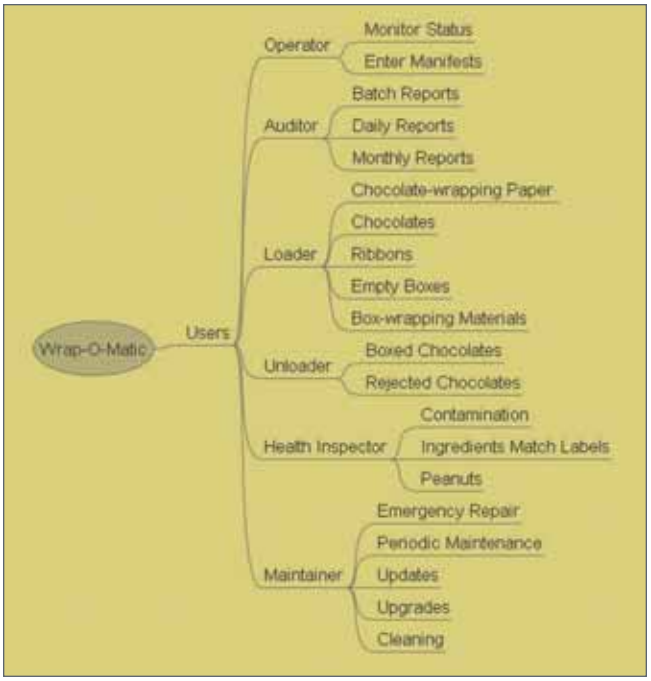


Figure 4: Wrap-O-Matic mind map

Mind Maps to Identify Quality Factors

A quality factor is a characteristic of the software project which, if missing, will lead to failure or negative consequences. Typical quality factors are usability, performance, scalability, maintainability, and other important "-ilities." I find it instructive to use a mind map to explore quality factors to determine whether they are important and, if so, how to test them. Often it is difficult to discover what quality factors are important to our clients. Mind maps can be an effective tool to communicate with requirements analysts, developers, and end-users to identify important quality factors.

Figure 5: Mind map to identify important usability characteristics

Figure 5 illustrates a mind map developed to identify which characteristics of usability should be tested for a specific Web application. Usability is broken down into three dimensions represented by branches of the mind map: ease of use, understandability, and engagibility. Further definition of these characteristics is done by building this map and discussing quality factors with different project stakeholders.

Quality factors are among the most challenging aspects of a software project to test. The general approach for mapping the testing of a quality factor is to identify which factors are important and then determine scale, meter, and the required targets for each factor (see Figure 6).

You will see that misunderstandings and ambiguities are exposed and resolved in real time as you build the mind map!

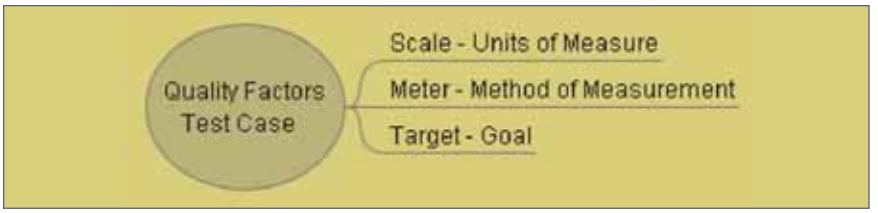


Figure 6: Mapping the testing of a quality factor

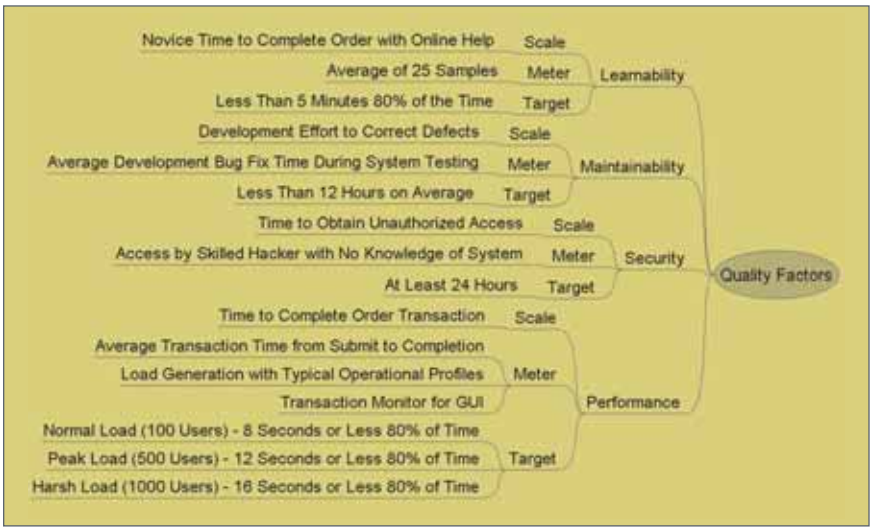


Figure 7: Quality factors mind map

Scale is the unit of measure I will use to test the factor. Meter is the experiment or method I will use to make the measurement. Target is the actual goal or expected result. In Figure 7 note that for performance testing I have one scale but three targets depending on the load of the system being tested. I find the maps are a useful way to review my understanding of the system's quality-factor requirements. Then I can speak with developers and requirement analysts to make sure we are in sync.

Your Next Step

I suggest that you start using mind maps in designing test cases based on equivalence classes for important test variables. Choose a variable that is sufficiently complex (for example, one related to potential system-security breaches, such as user type including users with different privileges). Draw a map using the AIM model and then brainstorm independently with developers, users, and other testers. As you meet with them, draw the map to document information. You will see that misunderstandings and ambiguities are exposed and resolved in real time as you build the mind map. Once you have perfected the basic mind-mapping technique, apply it to other areas such as identifying usage scenarios and quality factors. **{end}**

Robert Sabourin has more than twenty years of management experience leading teams of software development professionals to consistently deliver projects on time, on quality, and on budget. He is an adjunct professor of software engineering at McGill University and has spoken at conferences around the world, including the STAR conferences. Robert has published many software development and testing articles and is the author of the popular children's book I am a Bug!, *designed to teach software testing to children.*

Sticky Notes

For more on the following topics, go to www.StickyMinds.com/bettersoftware

- Tony Buzan and mind maps
- Mind-mapping tools
- More on equivalence classes

Benefits of Mind Maps

Visual • Testers benefit from mind maps because they provide an intuitive, visual representation of part of the application under test.

Communication • Mind maps assist in communication between technical and non-technical project stakeholders. By sitting with others, a tester can build or modify mind maps that describe important aspects of the system. This collaboration can lead to a more realistic and complete test design.

Creative • Mind maps encourage lateral thinking. When drawing a mind map, ask yourself "What if?" questions to determine how the map could be changed. Mind-mapping software allows for the easy restructuring of the map while it is being developed. Branches can be created, nodes can be split, and new ideas can be added at any time. Tools also allow for commenting, annotations, and inclusion of additional test, tables, or images.

Design Documentation • The mind map can be used as an important and reusable design artifact. Before detailing a test script or exploratory testing charter, create a mind map. Mind maps can be reused to generate many different types of tests or to select different test data for existing tests, thus serving as the basis for many different test cases.

Note Taking • In addition to describing part of the application under test, mind maps can be used by testers to organize their notes, especially as part of exploratory testing. While I am doing exploratory testing, I often keep a mind-mapping tool running to make visual notes of what I have discovered. It's easy to review mind maps with other testers, developers, and project stakeholders.

A Word of Caution • Sometimes seemingly simple mind maps can grow very large! If the mind map is getting too big, consider restructuring it by reorganizing the branch structure or by creating more than one map for the variable, scenario, or quality factor you are exploring.