# Logical Event Contingency Planning for Y2K

Robert L. Moore and Roberta H. Krupit
*Coastal Research and Technology, Inc.*

*Traditional contingency planning methods do not work well for logical events, such as year 2000 (Y2K) problems. And although there are many different types of logical event contingency plans, those that work well for some situations may be of no use for others. This article discusses the unique aspects of logical event contingency plans and helps you plan appropriate strategies to deal with logical events.*

Any information technology (IT) system needs its computing resources to operate correctly (system integrity) and should maintain the value of its information (data integrity). Unfortunately, adverse events may damage data integrity or system integrity or both.

Contingency planning for IT systems focuses on preserving, enabling recovery, or the graceful degradation of system or data integrity. Unfortunately, contingency mechanisms that work in one adverse situation may be hopelessly inadequate in another. Consequently, different contingency plan types exist for different "disasters." The two primary types are physical and logical event plans.

Traditional IT contingency plans address physical events such as flood, fire, earthquake, war, or loss of power. Many sources discuss traditional contingency planning, which often emphasizes replication and physical separation to guard against physical disasters. Such traditional planning does not consider logical events.

A logical event (LE) strikes all sites that have similar information configurations (software, data, and firmware)—no matter how widely separated. The impacted system also may corrupt IT that is down the information flow stream. LE contingency planning has two subcategories: plans for incompletely understood systems and plans for well-understood systems. Because the Y2K problem is an LE, discussion of LE contingency planning is appropriate.

LE contingency planning borrows from systems security principles, which include auditing, system modeling, input and output validation, and instrumentation. This article outlines the properties that a contingency plan strives to preserve, suggests techniques for further investigation in Y2K LE contingency planning, and explains LE contingency plans with respect to traditional plans.

Y2K planning is about risk management. Risk management involves

- Identifying risks (potential threats and vulnerabilities).
- Analyzing risks by evaluating, categorizing, and prioritizing them.
- Planning for risks.

Contingency planning reduces to

- Creating mechanisms to identify the events that trigger contingency actions.
- Defining what the contingency actions are (either automatic or manually executed actions).
- Preparing the responsible parties by documenting who they are and by training them to be ready.

### Weak Contingency Planning

An occasional protest to LE contingency planning is to offer a weak (or no) contingency plan followed with, "What more can be done? I can't plan for every possibility!" For example, a weak plan might call for canceling vacations and putting support personnel on call. This is merely a beginning that, unfortunately, does not take advantage of all available information.

Procedures to deal with a disruption should address the event's most probable serious consequences. Scale the plan to fit the consequences. For instance, if you run out of paper when printing E-mail, you acquire more paper—you do not buy a paper mill. However, if you run the U.S. Mint and you frequently run out of paper for money, perhaps you *should* buy a paper mill (or mint more coins). Planning for every event is impossible and counterproductive. It is prudent to analyze the adverse events that could occur (for example, Y2K problems) and construct mechanisms to preserve, enable recovery, or gracefully degrade system or data integrity.

## Contingency Plans and Integrity

An IT system should be reliable, correct, and accurate. These integrity principles divide into two categories: data and system integrity. Data and system integrity include accuracy, completeness, consistency, timeliness, authenticity, authorization, precision, compliance with laws, regulations, organization policies, and procedures, and evidence that all of the proceeding properties are fulfilled [4]. IT that loses system or data integrity is worse than useless—it may be misleading and even dangerous. Integrity principles are fundamental requirements to reliable IT operation.

Contingency plans state how to manage the planned degradation, preservation, or restoration of system and data integrity. Not all adverse events impact all integrity principles. To create a contingency plan for a particular system, one addresses that system's requirements with respect to a catastrophe by documenting how to handle the integrity of specific requirements.

## Physical Event Contingency Plans

Traditional contingency planning, whether from a computer-age viewpoint or not, assumes that problems are physical in nature. Adverse events take

the form of flood, fire, earthquake, war, terrorism, riot, hurricane, tornado, sabotage, loss of electricity, equipment failure, flu epidemic, and so on.

The distinguishing feature of physical problems is distance. A gas station explosion might impact business operations at a restaurant half a block away but will not impact operations at another gas station 100 miles away. Physical "disasters" have less direct impact as distance from the disaster increases.

Traditional contingency plans use the localization of physical disasters by emphasizing IT duplication in physically protected or remote locations. There is more to contingency planning than creating backups, of course. Other parts of physical event contingency planning include educating staff in contingency procedures, ensuring adequate management and security controls for operation during an event, cost analysis of recovery options, and mechanisms to rapidly transfer control to alternate sites. In any case, the ultimate safeguard in a physical event contingency plan is a remote operations center (ROC) that faithfully duplicates the capabilities of the primary operations center [7].

Fortunately, disaster planning literature covers the creation of an ROC. Consequently, even though the design, creation, and operation of an ROC are not easy, they are sufficiently documented that we need not revisit them here except in contrast to LE plans.

## Logical Event Contingency Plans

The logical and physical worlds are fundamentally different, as are logical and physical events. A logical event, such as a bug in command and control ($C^2$) software that shuts down pumping operations at a gas station based on a quarterly pump maintenance query, will not impact a neighboring restaurant but may impact every gas station with the same $C^2$ software. Indeed, LEs may corrupt otherwise operational systems down the information stream from the impacted system. A plan to address physical events is unlikely to help with an LE (and vice versa).

Adverse logical events are the combination of threats and vulnerabilities, a combination that is possible due to bugs at some level (requirements, design, or implementation). Logical events can follow failure to anticipate possible data forms, hazards that could attack a system (like vulnerability to a virus), unintended intercomponent interactions, or design assumptions that eventually destabilize the system (Y2K).

Computer security events are typically LEs. Computer security events provide clear examples of logical "disasters" (such as viruses and Internet worms). Methods to mitigate security events are similar to methods for logical events in general and Y2K events in particular.

IT security focuses on preserving a system and its information content against malicious attempts to make data and resources unavailable, unreliable, inaccurate, or inefficient. IT security begins by trying to avoid events and concludes by building mechanisms to deal with events should they happen anyway. System development tries to avoid LE problems but uses contingency plans to address expected or unexpected potential threats. An LE contingency plan focuses on preserving a system and its information content against events that make data and resources unavailable, unreliable, inaccurate, or inefficient.

The comparison of the Y2K LE with a virus is indeed appropriate. Y2K consequences, although not malicious, degrade IT much like viruses.

How an LE impacts IT depends on the "distance" between the various components. Coupling, cohesion, and similarity of function and form determine distance. As understanding of a system increases, logical contingency plans grow from generic catch-all to specific plans. This understanding provides a means to develop more detailed solutions, such as "checksums" and logging mechanisms on automated actions and rapid debugging measures [4, 6].

## Calculating Inputs from Valid Outputs or Outputs from Valid Inputs

Creating "checksums" from inputs and outputs depends on determining what valid inputs and outputs could be:

1. Using documentation, test cases and results, maintainer and user system knowledge, and accumulated live inputs and outputs surmise a set of acceptable inputs and outputs. This is a blueprint for what will and will not be allowed as inputs and outputs. This method is essentially anecdotal. However, it is also extremely practical in its simplicity (no special techniques or tools needed) and in its efficiency (the data is readily available). Take care that knowledge of historical inputs and outputs does not unintentionally exclude legitimate future input or output variations.
2. Calculate the weakest precondition (wp) or strongest post-condition (sp) to derive a set of conditions that must hold true either before or after execution, for the routine to function correctly. wp(S,R) is "the set of all states such that execution of [routine] S begun in any one of them is guaranteed to terminate in a finite amount of time in a state satisfying [expected result] R" [3]. Similarly, sp(S,I) represents that if [input condition] I is true, execution of S results in sp(S,I) true if S terminates [2].

Both wp and sp are obscure and have minimal automated support. However, they can be calculated almost completely mechanically. wp and sp are easiest to derive when good software engineering practices have gone into a routine (for instance, cyclomatic complexity is low, nesting is low, the routine is not too large, and the code is structured). Calculating wp and sp also requires more mathematical ingenuity when dealing with loops. The most practical method to determine wp or sp is to perform the calculations in sections with good software engineering or non-loop structures and to heuristically estimate what should be true in the spirit of wp and sp in the harder sections (see sidebar "Calculating Weakest Precondition for a Simple C Routine" on next page).

### Y2K LE Contingency Planning

Most Y2K contingency planning is LE contingency planning. Writing a Y2K contingency plan depends on how much is known about a system. Y2K triage determines how much is known and divides Y2K-impacted systems into two categories:

Category 1 – Critical and noncritical systems that because of fiscal, technical, or time constraints or mission-related decisions will not be worked on with respect to Y2K.

Category 2 – Critical systems that will be examined,[1] possibly fixed, and tested and for which adequate resources exist to accomplish these tasks.

Category 1 systems need plans that address Y2K consequences at a macroscopic level (*generic* plans). Conversely, contingency plans for Category 2 systems are based on information derived from analyzing and possibly repairing the systems (*specific* plans).

## Specific LE Contingency Plans

Compared to systems in Category 1, much is known about Category 2 systems. Information on what a Category 2 system does, how it does it, which sections of the code deliver what functions, etc., is usually available. This information is the basis of all subsequent contingency planning. It helps answer both technical (how do I guard against event X?) and management (which functions are important to me?) questions. (Generic LE contingency planning techniques, which are discussed later, may also apply to a system eligible for a specific plan, but not vice versa.) Specific plans feature an array of techniques[2,3] including input and output validation, auditing, and code instrumentation [4, 6]. An LE contingency plan's goals include the following:

- Detect problems quickly.
- Determine a specific cause for the problem.
- Determine a repair or, if no feasible repair exists, reduce or prevent further impact from the problem.
- Recover information about damage from the problem so that anything lost can be restored.
- Demonstrate due diligence in anticipating, avoiding, and mitigating problems.

The following subsections outline ideas for specific plans.

### Input and Output Validation

Recovery time and cost are cut when problems are noticed quickly. One way to notice problems is to automatically validate inputs and outputs [4]. An understanding of inputs, outputs, and their interrelationship requires insight into both the data form and function[4] (see sidebar below). To use input and output validation as a Y2K defense

- Determine valid outputs or inputs or both, usually at a subroutine level. (Determining outputs is often easiest.)
- Given known, valid outputs, calculate valid inputs (or conversely, valid outputs from inputs (see sidebar below,

## Calculating Weakest Precondition for a Simple C Routine

The following example shows how to calculate wp(S,R). (wp is discussed in the sidebar "Calculating Inputs from Valid Outputs or Outputs from Valid Inputs" on previous page). In this example, S is the program to be executed. R is a statement containing as much information as possible about what is hoped will be true after S executes. In this example, wp(S,R) is used to detect conditions that would cause R to not be true after S executes–that is, error conditions. Once wp(S,R) is calculated, it can be used as a built-in data check to see if errors will occur. Any data inputs that falsify wp(S,R) will cause the routine represented by "S" to produce unexpected results.

One way to use wp(S,R) is to place S in the context of a statement such as "if wp(S,R) then do S else report error." The following example demonstrates working out the full wp(S,R) calculation. Doing the full calculation for most programs is too tedious. Usually, contingency error-trap conditions are derived through a combination of formal wp calculations in easy code segments and heuristic estimates of what the calculations should look like in more difficult code segments. (See http://www.coastalresearch.com/ for definitions and more examples of both the formal and the heuristic techniques.)

In this example, the programmer interpreted the statement "the C compiler made by company X is Y2K compliant" to mean that any software compiled using that compiler would be compliant. The derived wp(S,R), if checked before the routine S executes, provides an error detector to guard against the programmer's misinterpretation.

Let S =

```
"       #include <stdio.h>
        #include <time.h>
        #include <stdlib.h>

        int maintenancedue (int lastservice)
        {
         int maintenance, servicedue, currentyear
         long currentdate;
         struct tm *t;

         time(&currentdate);
         t = localtime(&currentdate);
         currentyear = (int) t->tm_year;

①        servicedue = 5 + lastservice;
②        if (currentyear >= servicedue)
③        maintenance = currentyear;
         else
④        maintenance = servicedue;
⑤        return maintenance;
        }"
```

For a more detailed explanation of the mathematics that follow, see http://www.coastalresearch.com/. Note that "$\vee$" means "OR" or "union," "$\wedge$" means "AND" or "intersection," "$\Rightarrow$" represents "implies," "F" is false, and "T" is true. The calculation uses the fact that if the symbol "a" may be expressed as the sequence of symbols "b;c", then wp(a,R) = wp(b;c,R) = wp(b,wp(c,R)).

R = "currentyear $\leq$ maintenancedue = max(currentyear,lastservice+5) $\leq$ currentyear+5," because the "business rules" (known to the source code maintainers or users) say that maintenance occurs every five years at the maximum. The rules also say how the current year and the maintenance year are related.

wp(S,R) = wp(①;②;⑤,R) =

wp(①;②,wp(⑤,R)) =

wp(①;②,wp(maintenancedue = maintenance, currentyear $\leq$ maintenancedue = max(currentyear,lastservice+5) $\leq$ currentyear+5)) = wp(①;②, currentyear $\leq$

"Calculating Weakest Precondition for a Simple C Routine").

- Add source code to beginning or end of each subroutine (or whatever level valid input and output sets were determined) to check incoming (or outgoing) data to verify that it is "within range." If the input (or output) data is out of range, take whatever action needed, e.g., issue warning messages, write an error log, or halt execution.

Input/output validation is a good practice that may even provide a defense against the most pernicious Y2K error—unrecognized data corruption just short of system failure.

## Event and Data Auditing

Audit information records when and how events occur—critical information in planning and executing recovery. Event auditing logs the actions and logs calls by routines, calls to routines, the order in which calculations take place, etc., to localize a problem's cause. Data auditing records the transactions against each data item. This traces where data corruption exists, how far it has spread, and perhaps what might be done to correct it. Together, event and data auditing diagnose a system by identifying an LE's cause and effect [6]. A specific contingency plan explains how to collect the information and what to do with it.

## Using Debuggers and Source Code Instrumentation to Build "Audit" Trails

Running an "instrumented" or "debug" version of a system accumulates audit-like information to find and diagnose errors after they have occurred. Although collecting information by using test or debug tools is not traditional auditing, these tools are practical because standard audit trails are often not detailed enough. For instance, test instrumentation tools add code to an application that records every execution branch choice and perhaps even value settings. (Some debuggers have a trace mode that provides equivalent information.) Likewise, debugging tools allow "break points" to be set in software. Program execution is suspended when a break point is encountered, allowing queries on variable values. Some debuggers permit interaction; other debuggers and all instrumentation tools provide only post-execution information.

Presuming there will be no bugs from debug or instrumentation interaction with the source code at compile time, you can use the following procedure of instrumented and debug source code versions to track down and diagnose Y2K events

- Create an instrumented version of the application. Create a debug version of the application.
- Retain a non-debug, noninstrumented version of the application for execution during all periods when Y2K events are unlikely. (Debugging and instrumentation data generation may slow the application.)

maintenance = max(currentyear, lastservice+5) ≤ currentyear+5) = wp(①, wp(②, currentyear ≤ maintenance = max(currentyear, lastservice+5) ≤ currentyear+5))

However, wp(②, currentyear ≤ maintenance = max(currentyear, lastservice+5) ≤ currentyear+5) = [(currentyear ≥ servicedue) ∨ (currentyear < servicedue)] ∧ [(currentyear ≥ servicedue) ⟹ wp(maintenance = currentyear, currentyear ≤ maintenance = max(currentyear, lastservice+5) ≤ currentyear+5)] ∧ [(currentyear < servicedue) ⟹ wp(maintenance = servicedue, currentyear ≤ maintenance = max(currentyear, lastservice+5) ≤ currentyear+5)] = [T] ∧ [(currentyear ≥ servicedue) ⟹ (currentyear ≤ currentyear = max(currentyear, lastservice+5) ≤ currentyear+5)] ∧ [(currentyear < servicedue) ⟹ (currentyear ≤ servicedue = max(currentyear, lastservice+5) ≤ currentyear+5)] = [(currentyear ≥ servicedue) ⟹ (currentyear ≤ currentyear = max(currentyear, lastservice+5) ≤ currentyear+5)] ∧ [(currentyear < servicedue) ⟹ (currentyear ≤ servicedue = max(currentyear, lastservice+5)≤currentyear+5)]=R'

wp(①, R') = wp(servicedue = 5 + lastservice, [(currentyear ≥ servicedue) ⟹ (currentyear ≤ currentyear = max(currentyear, lastservice+5) ≤ currentyear+5)] ∧ [(currentyear < servicedue) ⟹ (currentyear ≤ servicedue = max(currentyear, lastservice+5) ≤ currentyear+5)]) = [(currentyear ≥ 5 + lastservice) ⟹ (currentyear ≤ currentyear = max(currentyear, lastservice+5) ≤ currentyear+5)] ∧ [(currentyear < 5 + lastservice) ⟹ (currentyear ≤ 5 + lastservice = max(currentyear, lastservice+5) ≤ currentyear+5)]

If S executes only when "[(currentyear ≥ 5 + lastservice) ⟹ (currentyear ≤ currentyear = max(currentyear, lastservice+5) ≤ currentyear+5)] ∧ [(currentyear < 5 + lastservice) ⟹ (currentyear ≤ 5 + lastservice = max(currentyear, lastservice+5) ≤ currentyear+5)]" (the result of the wp(S,R) calculation above) is true, R will be satisfied. (To calculate the truth or falseness of this quantity, note that for two symbols "a" and "b," "a ⟹ b" is false only if "a" is true and "b" is false.) Inputs that falsify the result of the calculation are error conditions and should be guarded against.

To see how this works in practice, suppose that it is now the new millennium. Let the current year be 2001, represented as currentyear=101 in the source code (check the C definitions). Supposing the last maintenance was in 1995, represented as lastservice = 1995, S might be supplied by a database to the routine S. Executing S should return that the maintenance is due now, in 2001. Instead, the routine returns 2000.

A follow-on routine that flags equipment for maintenance by checking to see if the maintenance due date is *equal* to the current year would return false–and continue to return false forever. To a casual reader (or to a hurried Y2K analyst under pressure to get the job done), the code looks fine. But if the result of the wp(S,R) calculation is checked, it will be found to be false. Using the wp(S,R) as a "guard" before executing S would prevent this error.

As a practical matter in this example, an experienced C programmer could have found the error with less work than calculating wp(S,R). In more complicated routines, or where someone with enough time and experience to read the code is unavailable, calculating wp(S,R) using a combination of formal and heuristic techniques holds the advantage.

- On any date or time when a Y2K event is likely, use the instrumented version of the application in place of the standard version. If a Y2K error appears, use the "audit" information from the instrumented version to identify the instruction sequence that occurred during the erroneous run.
- Use the debug application version to investigate the erroneous execution sequence found with the instrumented application.

Specific tools are not necessary to gather audit-like information. Anyone who can write and compile code can add debug and trace statements. Tools merely make life easier. In any case, audit information is valuable in following an event's cause and effect.

### Audit Through Application Models

Some Y2K assessment and repair tools construct comprehensive source code models through reverse engineering techniques. During assessment, they help the programmer locate potential Y2K problems and may even provide insight to fix problems. To audit using these models simply means that if a Y2K event occurs, the model, rather than being used just as an error predictor, can help diagnose the problem. The model constitutes a "holistic" audit in its ability to localize problems given the real-life information about what happens when an error occurs.

Presuming that an up-to-date model of the source code and its interactions and dependencies can be maintained, a model can be used to rapidly diagnose errors as follows:

- Use a reverse engineering (see sidebar "Reverse Engineering") tool to model the application, beginning with the implementation level and working toward a design-level understanding. Identify all real and suspected date data usage. (Deriving this information is essentially what goes on during a sophisticated Y2K bug search.) Keep this model current throughout revisions. If possible, trace the user's experience-based "business rules" understanding through the application. This trace may help in understanding the side-effects of Y2K problems.
- When a Y2K bug occurs, note the functional area and as much other information as possible about the bug. Pinpoint the bug in the reverse engineered model.
- Using the model's control and data flow information, along with the data usage information, trace the bug manifestation to the code and data flows that caused the bug.
- Follow the bug as it propagates through the code and data to find all bug implications.
- Repair all problems caused by the bug. Update the model to reflect the repairs.

## Generic LE Contingency Plans

Limited information about systems in Category 1 confines generic LE contingency plans to addressing broad possibilities. Generic plans end up being like physical event plans; however, generic plan techniques apply equally to systems in Categories 1 and 2. Generic contingency plan mechanisms include service degradation, internal recovery, commercial recovery, cooperative recovery, and combination recovery strategies [7].

### Service Degradation Strategies

Service degradation strategies are useful when IT is partially operational. Service degradation involves

- Reduction of Service: Some, but not all, functionality is available [7]. This strategy works when part of the system experiences problems, but a work-around bridges the gap. The work-around might not be desirable or meet all requirements, but it allows something like business-as-usual pending repairs. For instance, when a central calendar management system is inoperable, anyone planning events is inconvenienced. Replacing the calendar system with a temporary text file to share information may suffice.
- Manual Replacement of IT-Based Service: IT tasks can sometimes be performed manually. This occurs when manual calculations substitute for automated functions, or paper records replace on-line data. Knowledge about the manual procedures often exists, since the IT service superseded the original manual procedures [7]. Manual procedures do not mean abandoning automation—if a corporate accounting program on a mainframe is unavailable, perhaps a personal computer spreadsheet could substitute.
- Withdrawal of Service: Functions without immediate operational impact (planning, research and development, etc.) are dispensable during an LE. The functionality may be too complex, require too much precision, or be too time-consuming for manual execution [7]. Because the functionality cannot be acceptably executed without the unavailable system and the functionality is not immediately critical, withdrawal of the service is the best choice.

## Reverse Engineering

Reverse engineering is defined as "the process of analyzing a subject system to (1) identify the system's components and their interrelationships and (2) create representations of the system in another form or at a higher level of abstraction." [1] (See Figure 1.)

Reverse engineering covers a variety of techniques; only a few are relevant here. Beginning with source code, reverse engineering can, for example, produce control and data flow information (often represented graphically) both between and within routines, identify the ripple effects of changing one piece of source code with respect to other code, and even deduce the domain of valid inputs and outputs.

Reverse engineering depends heavily on automated assistance. Many fine research tools are free (see http://gulf.uvic.ca/~kenw/toolsdir/). Building and maintaining a source code date usage model using a reverse engineering tool can have a significant payoff not just in diagnosing Y2K problems before critical dates occur but after supposedly corrected problems crop up as well.

## Internal Recovery Strategies

Contingency plans frequently use internal recovery strategies. These strategies feature a "can do" attitude of "the pressure is on—let's get the job down *now*!"

- Work Round-the-Clock: Working extra hard sometimes gets a job done quickly and well. However, experience teaches that work produced under pressure is often poorly done. Plans that use this strategy need to provide details on getting the best from people in a short time, maintaining morale under pressure, and choosing the right people.
- Train and Assign Extra People: Any project might benefit from extra hands. Unfortunately, a crisis is not the time to bring on new people. There may be exceptions to this if the people are of high ability, thoroughly trained, and familiar with the work but (through some quirk of fate) are working elsewhere. This strategy requires forethought on getting good people, training them adequately, and integrating them into the team.
- Have Employees On Call: Many industries use an "oncall" strategy to have employees available during unforeseen events. Unless a spot repair solves the problem, though, calling employees in at midnight might accomplish no more than creating a bleary-eyed work force. Important details for this tactic include having a checklist of trivial repairs to attempt before calling employees in, diagnostics to determine when a problem is solvable from on-call resources, and arrangements in case of unreliable communications.
- Information Preservation: A classic "disaster" recovery aid is source code and data backup. System backups are prudent. Preserving data and program code frequently, particularly if multiple versions are retained (in case the most recent backup was done *after* an event occurred but *before* the event was noticed), ensures a baseline exists. Be sure to verify that the procedure to recover saved information works *before* it is needed. Be aware, too, that backups may be of limited use during a Y2K LE, because (1) the backup may be unreadable by the Y2K-impacted system, and (2) the backup itself might include Y2K errors.

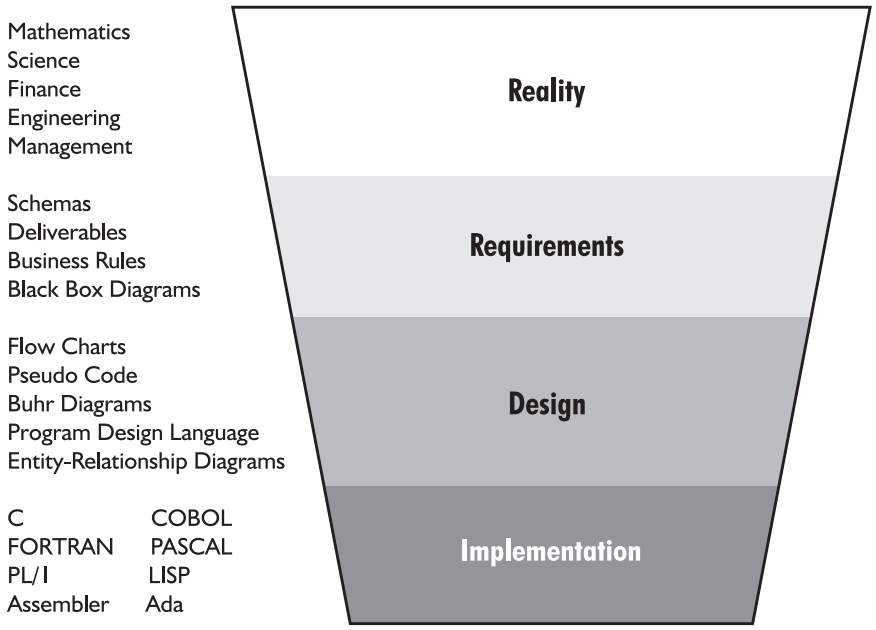


Figure 1. *Software abstraction levels.*

## Commercial Recovery Strategies

Commercial recovery strategies help when an organization cannot recover from an event because of technical, personnel, or political issues but can "hire" a solution.

- Contracting Tasks to Others: Hiring supplemental employees may aid recovery from an event. Teams can temporarily expand to produce results more rapidly. Alternatively, added employees can free internal resources to concentrate on recovery. Either strategy creates difficulties similar to those in the above "train and assign extra people" solution. Important details in using this strategy include determining the types of available help and having a purchase order for services pre-approved.
- Commercially Available System Alternatives: Are there commercially available equivalents to an internally developed system? Depending on the urgency of repairs, their technical feasibility, how good a replacement the commercial alternative is for the existing component, and the cost of the repairs vs. the commercial substitute, buying a replacement is an efficient recovery method. It may require great effort to fit the commercial substitute into the existing infrastructure; so, seriously consider the impact before using this tactic.

## Cooperative Recovery Strategies

Cooperation between organizations with similar systems and problems might facilitate more robust systems and more rapid post-event recovery. Banding together also gives cooperating organizations a louder voice to vendors who are making fixes and provides other opinions on how to proceed. A drawback is that cooperation helps those who did not work hard to meet the challenges while providing little benefit to those who did their homework. Cooperation also risks exposing sensitive information to potential competitors. A contingency plan that addresses this strategy helps limit the risks while enhancing the advantages by establishing nondisclosure agreements, exploring the strengths of each party, and determining administrative procedures for cooperation.

## Combination Recovery Strategies

Combining the strategies above gives a more robust overall solution. For in-

stance, hiring temporary on-call employees might be as effective as using internal employees while avoiding a morale impact on long-term staff.

## Summary

A good contingency plan accounts for the importance of the IT being protected, the contingency mechanisms' costs and benefits, and the ability of system developers, maintainers, and managers to implement the plan. Paramount in contingency planning is knowing a system's vulnerabilities, determining real-world threats, understanding the combination of threats and vulnerabilities, and then choosing appropriate contingency mechanisms. ◆

### About the Authors

Robert L. Moore is a senior software engineer for Coastal Research and Technology, Inc. in the National Security Agency (NSA) Year 2000 Oversight Office. He is the author of Y2K compliance criteria widely used in the U.S. intelligence community and a variety of articles on software reengineering, reverse engineering, and Y2K issues. Prior to Y2K work, he worked on software reengineering projects for NSA's software engineering center. He is a certified software test engineer and has a master's of science degree in applied mathematics.

718 Meadow Field Court
Mount Airy, MD 21771
Voice: 301-688-9943
Fax: 301-688-9494
E-mail: rlmoore@romulus.ncsc.mil

Roberta H. Krupit is a senior software engineer for Coastal Research and Technology, Inc. in the National Security Agency Year 2000 Oversight Office. She has worked on software reengineering projects for NSA and the Office of Naval Intelligence.

### References

1. Chikofsky, Elliot J. and James H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, January 1990, pp. 13-17.
2. Gannod, Gerald C. and Betty H. C. Cheng, "Using Informal and Formal Techniques for the Reverse Engineering of C Programs," *Proceedings of the 1996 International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, Calif., 1996, pp. 265-274.
3. Gries, David, *The Science of Programming*, Springer-Verlag, New York, 1981, Chaps. 1, 9-12, 16.
4. Mayfield, Terry, J. Eric Roskos, Stephen R. Welke, and John M. Boone, "Integrity in Automated Information Systems," C Technical Report 79-91, Institute for Defense Analysis, 1991, Sections 2.1, 2.2, 3.6-3.9, 3.12. (Available by writing to INFOSEC Awareness, Attn: V/NISC, National Security Agency, 9800 Savage Road, Ft. George G. Meade, MD 20755-6753 or at http://www.radium.ncsc.mil/tpep).
5. Mohan, C., Kent Treiber, and Ron Obermarck, "Algorithms for Management of Remote Backup Data Bases for Disaster Recovery," *Proceedings of the 9th Annual International Conference on Data Engineering*, IEEE Computer Society Press, Los Alamitos, Calif., 1993, pp. 511-518.
6. National Computer Security Center, "A Guide to Understanding in Audit in Trusted Systems," National Computer Security Center, 1987, Section 5-6 (Available by writing to INFOSEC Awareness, Attn: V/NISC, National Security Agency, 9800 Savage Road, Ft. George G. Meade, MD 20755-6753 or at http://www.radium.ncsc.mil/tpep).
7. QED Information Services, Inc., *Disaster Recovery: Contingency Planning and Program Evaluation*, Chantico, Port Jefferson, New York, 1985, Chap. 4.

### Notes

1. Some systems from Category 2 may be returned to Category 1 if Y2K examination reveals that the systems will be impossible to repair within existing resource constraints.
2. A useful related technique is process isolation—separating data records into two sets: one set for application X and one set for application Y to limit data-propagated errors in X from corrupting Y (and vice versa). Algorithms may be adapted from [5].
3. Least privilege or role enforcement (restricting processes to just the accesses and abilities they need for the current moment's action) is another related mechanism. For instance, if the YY part of DDMMYY increments years since 1900, YY could overflow as the counter moves from 31 December 1999 to 1 January 2000 (that is, YY = 100). To restrict any process that tries to write DDMMYY to a database to no more than six characters still allows Y2K problems to occur but prevents an accidental overwrite of adjacent data items.
4. This is true if inputs from random number generators are counted as outside inputs. Consequently, the random number is a known quantity as an input, even if it is not known until run time.

## IEEE/EIA 12207 Standard for Software Lifecycle Processing

The new commercial standard IEEE/EIA 12207, "Information Technology – Software Life Cycle Processes," is available from the Defense Automated Printing Service (DAPS) at no charge. The standard comes in three parts:

- IEEE/EIA 12207.0, "Standard for Information Technology – Software Life Cycle Processes."
- IEEE/EIA 12207.1, Guide for ISO/IEC 12207, "Standard for Information Technology – Software Life Cycle Processes – Life Cycle Data."
- IEEE/EIA 12207.2, Guide for ISO/IEC 12207, "Standard for Information Technology – Software Life Cycle Processes – Implementation Considerations."

Other military and federal specifications also are available from DAPS.

Defense Automated Printing Service
Building 4/D
700 Robbins Avenue
Philadelphia, PA 19111-5094
Help Desk: 215-697-6257/6396 DSN 442-6257/6396
Fax: 215-697-1462
E-mail: roy_bowser@daps.mil
Internet: http://www.dodssp.daps.mil