SESUG Paper 221-2018

When Reliable Programs Fail: Designing for Timely, Efficient, Push-Button Recovery

Troy Martin Hughes

ABSTRACT

Software quality comprises a combination of both functional and performance requirements that together specify not only *what* software should accomplish, but also *how well* it should accomplish it. Recoverability, a common performance objective, represents the timeliness and efficiency with which software or a system can resume functioning following a failure. Thus, requirements for high availability software often specify the recovery time objective (RTO), the maximum amount of time that software may be down following an unplanned failure or a planned outage. While systems demanding high availability will require redundant hardware, network, and additional infrastructure, software, too, must facilitate rapid recovery. Moreover, in environments in which system or hardware redundancy is infeasible, recoverability can be improved only through software development best practices. Because even the most robust code can fail under duress or due to unavoidable or unpredictable circumstances, software reliability must incorporate recoverability principles and methods. This text introduces the TEACH mnemonic that describes the guiding principles that software recovery should be *timely*, *efficient*, *autonomous*, *constant*, and *harmless*. Moreover, the text introduces the SPICIER mnemonic that describes discrete phases in the recovery period, each of which can benefit from and be optimized with TEACH principles. Software failure is inevitable, but its adverse effects can be minimized by incorporating recoverability best practices into SAS® software design and development.

INTRODUCTION

Reliability describes a measure of performance against failure, often defined mathematically as the probability (or actuality) of error-free operation or mean time to failure (MTTF), thus the amount of time a product functions before it fails. Because a software failure typically represents only a need to restart (or occasionally reprogram), software reliability is often referenced as mean time *between* failures (MTBF), distinguishing that the software lifespan does not terminate when it fails, unlike a burned-out lightbulb that becomes useless. Regardless of which nomenclature is utilized, reliability should be sought in software design, especially in SAS software that underpins critical infrastructure, serves an extensive user base, produces dependent processes or data products, or is intended to be enduring. Achieving software reliability is the most crucial step in minimizing the cumulative recovery time that a development team spends restoring code and functionality following failures—because code that doesn't fail in the first place won't need to recover.

Notwithstanding, even the most robust, reliable, defect-free code will fail on occasion. Environmental, network, system, data, and other external issues can cause lapses in connectivity, memory errors, and other failures that result in unavoidable program termination. Planned outages, including upgrades and modifications to SAS software, its infrastructure, network, and hardware also require halting operational SAS programs. While *recoverability* reflects the ability to restore functionality or availability of a system or software following a failure or planned outage, *recovery* represents the state of restored functionality or availability, and *recovery period* the time and effort required to restore the system or software. As recoverability principles are embraced in software design, the recovery period decreases and software reliability correspondingly improves.

An ideal recovery is *timely*, *efficient*, *autonomous*, *constant*, and *harmless*—tightly intertwined principles depicted through the TEACH mnemonic. Timeliness reflects the speed with which the recovery occurs, including discovery of the failure, how long it takes to restart the code, and how long it takes the code to surpass the point of failure. Efficiency demonstrates code that does not require excessive processing power or developer effort, the latter of which can be eliminated through autonomous recovery design. Autonomy reflects code that identifies failure, terminates automatically if necessary, and can be restarted with the push of a button without further developer intervention. Code constancy demonstrates that a

recovery strategy is enduring and thus not an expedient "quick fix" that will need to be improved or undone later. Harmless code should both fail and recover without damaging its environment, such as causing other processes to fail or producing invalid data products.

While TEACH depicts recoverability principles that should be applied during software design and development, the SPICIER mnemonic identifies specific components of the recovery period, each of which can be optimized through development best practices:

1. ***S**top the **P**rogram* if code did not fail safely, because failure does not necessarily denote that code has stopped running.
2. ***I**nvestigation* entails gaining a full understanding of the source, cause, duration, impact, and potential remedies of the failure.
3. ***C**leanup* can often be eliminated through software development best practices but includes actions such as removing or "backing out" faulty data that have entered a permanent data store due to poor quality control.
4. ***I**nternal* issues and remedies represent code modifications that must be made following a failure to facilitate recovery.
5. ***E**xternal* issues and remedies also represent modifications to code, but on modules that interface with external data, such as monitoring its connectivity status, structure, format, content, or quality.
6. ***R**erunning* code is the final phase in the recovery period and, after especially heinous failures that have crippled critical infrastructure, the point at which SAS developers head to Buffalo Wild Wings to unwind.

While signaling higher quality software, an investment in recoverability will not be warranted in all situations. Production code that runs with no external dependencies in a stable SAS environment likely experiences few failures per year, thus reliability would be only marginally improved through decreased recovery periods. SAS code similarly that is ad hoc, has no dependencies, is not critical, or is intended to have a brief lifespan also may not benefit substantially from adherence to recoverability principles, which often require additional quality assurance, quality control, and process tracking mechanisms to implement. However, developers should be aware of recoverability objectives and principles and, when appropriate and beneficial to software requirements, be able to build SAS software that confidently, completely, and consistently recovers from failure.

RECOVERABILITY PRINCIPLES

While the TEACH principles of recoverability are closely intertwined, timeliness is the most measurable attribute and thus often acts as a proxy for other principles. Mean time to recovery (MTTR) represents the average amount of time that specific software requires to recover from failure over its lifespan or some other defined period. By some definitions, recovery is complete when software begins executing again. However, in data analytic development that often transforms data from one state to another through serialized extract-transform-load (ETL) processes, business value may not be achieved simply by typing "Run." Thus, recovery can also be defined organizationally as the point at which software execution surpasses the previous point of failure. In this sense, if code failed while extracting the 20th of 25 tables from an external database, recovery would not be considered complete until the program had been executed again and had successfully extracted the 20th table. This is not to suggest that the initial 19 tables would need to be re-extracted, only to say that the 20th table would need to be extracted successfully to constitute recovery. Other definitions of recovery are even more stringent and require that dependent data products be valid and available. In this sense, a failure that corrupted a critical data set would not be considered remedied until the data set—or possibly reports or other data products produced from the data set—had been replaced with valid output.

It becomes clear that recovery must be conceptualized similarly by all stakeholders. Thus, a developer who restarts code minutes after a failure and at that point considers recovery to be complete may frustrate a stakeholder who instead does not view recovery as complete until he can access reports produced by the failed code. Especially where reliability or recoverability metrics are captured by an organization, the recovery period must be measured and recorded through consistent methodology if used to calculate MTTR. Moreover, if stakeholders have established a recovery time objective (RTO) as a

software requirement, the RTO should be conceptualized through the same methodology because the RTO becomes the standard against which MTTR is measured. Everyone must be on the same page.

As a proxy for other recoverability principles, timeliness is often utilized to measure not only MTTR and the entire recovery period, but also individual components of the recovery period to identify areas that can most be improved. Figure 1 demonstrates the phases from failure to functionality through the recovery period. By optimizing each of these steps through recoverability principles, the ultimate objective in "designing for recovery" is to develop software that programmatically minimizes or bypasses each step.

Figure 1. Recovery Period Phases

Efficiency represents the effort that a person or process expends to facilitate software recovery in relation to elapsed or expended time. Thus, a developer could work inefficiently in investigating or remedying a failure if he is unfamiliar with the SAS environment, the SAS language, or the specific code. Code that is difficult to read or poorly documented can also unnecessarily delay recovery because it is not intuitive and thus more difficult to maintain and modify. A failure that requires substantial cleanup of the SAS system because the software produced spurious data, metadata, or data products could also be an inefficient use of a developer's time, because that cleanup likely could have been eliminated or occurred automatically were more rigorous software development methods emplaced. While software failure is never a pleasant experience, through proactive design and development, developers can ensure that their time and effort are maximized during the recovery period.

Software itself also must recover efficiently. If SAS software doesn't recognize that an internal failure has occurred and the failure goes unnoticed for hours or days, this is an inefficient use of the SAS processor, which was effectively spinning its wheels without purpose. When the program must be rerun at the close of the recovery period, efficient code also should intelligently bypass any unnecessary steps that don't need to be completed. Under ideal conditions, checkpoints (i.e., restore points) have been recorded that demonstrate code modules that have completed successfully. This permits process flow to skip to the point of failure. For example, re-extracting 19 tables that had ingested without error only an hour before would be tremendously inefficient, thus a checkpoint could allow the software to recover from failure by jumping to the 20th table.

Autonomy requires that software recover with minimal or no developer intervention. From automatic detection of the initial failure, to effortless cleanup, to code that can be rerun without further modification, autonomy can dramatically expedite recovery. Process metrics collected during software failure can demonstrate the type and location of defects or errors, especially if they can be compared with baseline metrics that demonstrate previous process success. By anticipating and identifying specific error types in exception handling routines, software can often terminate gracefully. And, when code does terminate gracefully from an anticipated failure, such as a temporarily dropped network connection or a memory error, the path to recovery can also be predicted and thus automated because neither internal nor external issues will need to be investigated or recoded. Where new or unanticipated errors occur causing failure, quality control business rules within software can be modified to include routines that identify those new events and exceptions; in this way, through continuous quality improvement (CQI), software will continue to grow more robust and autonomous over time.

Constancy reflects stability in code over time, as well as stability in how software is executed. Stable code should be dynamic and intelligent enough to determine how and where to restart after a failure. Thus, rather than highlighting sections of code to be run manually or creating quick fixes to be used to jump start code, developers should be able to run the same code in its entirety every time. Thus, temporary solutions that can run code *today* may be timely but, if they introduce new defects or technical debt that must be dealt with *tomorrow*, their added risk is often prohibitive. When failures do occur and emergency or corrective maintenance is required, always fix the source, not the symptoms.

Harmless software does not negatively affect its environment when it fails and throughout the recovery process, such as by producing faulty, incomplete, corrupt, or otherwise invalid data, metadata, data

products, or output. To be harmless, code should automatically detect a failure when it occurs and handle it in a predictable, responsible manner; only through predictability comes the assurance that dependent processes and data products are not negatively affected. In most cases, it is far more desirable to have data products that are unavailable or delayed than those that are dead wrong. The recovery process also should not harm external processes or alter the environment unfavorably. Thus, if a developer modifies SAS library names in a production environment as an expedient solution to rerun software that has failed (because it referenced library names in a separate development environment), this could cause unrelated processes to fail. Code bandages may be appropriate under extraordinary circumstances, but when those bandages lack quality or display commensurately less quality than software requirements specify, wounds quickly can fester and even spread to previously unaffected infrastructure.

1. STOP PROGRAM

Failure comes in all shapes and sizes but doesn't necessarily denote program termination or even that errors were encountered during execution. Failure can occur not only when code produces warnings or errors or terminates abruptly, but also when it enters a deadlock, infinite loop, or continues to execute albeit producing invalid data or output. Thus, a common goal of exception handling routines is to channelize these disparate failure outcomes to a path in which processing continues, the *fail-safe path* (i.e., *happy path, happy trail*), in which processing fails gracefully. When software does not fail safe, infinite loops, syntax errors, or faulty business logic can be encountered that produce invalid data or results, thus decreasing or eliminating production value. Worse are the cases in which developers or analysts are unsure of whether a process completed correctly, and thus are forced to endure a fishing expedition to validate a data product, output, or other results.

Although all production software should fail gracefully when errors are encountered, if an unanticipated error is encountered for the first time, software may fail in an unsafe or unpredictable manner. In these cases, if software is still executing but errors have been detected, the software should be terminated manually. For example, if an ETL program that performs daily data ingestion requires tables to have 40 fields but on Monday a table is encountered that only has 37 fields, the process flow should be robust enough to automatically recognize and resolve this exception. The specific resolution would be controlled by business rules, but might include immediate program termination with notification to stakeholders, as well as preventing dependent processes from executing. If the system is not robust and does not detect the 37-feld discrepancy, later code could continue to execute and create invalid results, even if no actual runtime errors were produced. Thus, when software doesn't natively stop after encountering a failure, exception handling and conditional logic should be emplaced to detect and respond automatically.

Timely, efficient, and autonomous termination can often be achieved through exception handling routines that detect runtime errors and provide graceful program termination. For example, the following SAS output demonstrates an error that is produced when the Ghost data set doesn't exist:

```
data temp;
    set ghost;
ERROR: File WORK.GHOST.DATA does not exist.
run;

NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.TEMP may be incomplete. When this step was stopped there
were 0 observations and 0 variables.
WARNING: Data set WORK.TEMP was not replaced because this step was stopped.
%put &SYSCC;
1012
```

The SAS automatic macro variable &SYSCC reflects the value of the highest error encountered during processing, thus 1012 reflects the runtime error. Testing &SYSCC after each procedure, DATA step, or other module offers one way to determine programmatically whether a runtime error has occurred; thereafter, the evaluation of &SYSCC and can be used to alter program flow dynamically. In a separate text, the author demonstrates exception handling methods that detect error and other unwanted program states and redirect process flow toward program resumption or graceful termination.[i]

Software failures, however, are not always signaled by accompanying runtime errors. If the following code is run after the previous code, it runs without producing a runtime error but is still considered to have failed because the intended result—a data set having data—isn't produced:

```
data temp2;
    set temp;
run;

NOTE: There were 0 observations read from the data set WORK.TEMP.
NOTE: The data set WORK.TEMP2 has 0 observations and 0 variables.
```

It turns out that despite encountering a runtime error, the first DATA step nevertheless creates the data set Temp which has zero observations and zero variables. The second DATA step executes without warning or error, despite having invalid input (i.e., a null data set). Thus, in testing for conditions that should direct a program to abort, software must examine more than warning and error codes and must validate business rules and prerequisite conditions. The following code first tests to ensure that the data set contains at least one observation and, if that requirement is not met, the %RETURN statement causes the macro to abort:

```
%macro validate(dsn=);
data _null_;
    call symput('nobs',nobs);
    stop;
    set &dsn nobs=nobs;
run;
%if %eval(&nobs=0) %then %do;
    %put NO OBSERVATIONS!;
    %return;
    %end;
%mend;

%validate(dsn=work.temp);
```

Despite diligence in software exception management, programs will occasionally need to be terminated manually. An entire SAS server can crawl or stall if SAS processes have consumed too many resources, thus memory-hogging programs may need to be terminated and either restarted or reengineered. At other times, a developer may realize that an operational process has started creating errant output, such as HTML reports that are invalid or blank, indicative of some internal process failure. These errors can often occur when data integrity constraints are omitted, and erroneous or unwanted data enter a process flow unfiltered. For example, if report production necessitates that a categorical text field have only five discrete values, the inclusion of additional values might not produce runtime errors but could corrupt dependent processes and data products.

Terminating software manually reinforces the reality that software can fail inexplicably on any line of code. The hapless developer who trips over a cable and disconnects his machine from the server could do so while *any* line of code is executing. Thus, designing for recoverability requires developers to espouse this mindset and ask themselves after each line of code, "What will happen if my code fails *here*?" This principle reflects that code should always fail safe in a harmless, predictable manner—because software that fails safe and is harmless will not require cleanup, thus this agonizing step in the recovery process can be avoided.

2. INVESTIGATION

When program termination has resulted from a scheduled outage due to maintenance or modification, this step won't be necessary because no unknown exists. But in cases in which a failure has occurred in production software, analysts, developers, management, and other stakeholders will be clamoring not only for immediate restoration but also for answers. From a risk management perspective, key stakeholders are often not concerned with great technical detail, but rather with more substantive facts surrounding the failure:

- Was the failure discovered immediately?

- Was it discovered through automated means or manually?
- What was the cost or damage of the failure?
- Was the failure internally or externally caused?
- Was the failure anticipated, in that it exploited a previously identified vulnerability?
- Was the cause of the failure remedied?
- Could this specific type of failure occur again?
- Did this failure uncover additional vulnerabilities that should be remedied?

Log files are a recommended best practice for production software because they can help validate program success through a baseline (i.e., clean) log as well as facilitate the identification of runtime or logic errors. If the source of the failure is captured in the log file, developers can often refactor software to be more robust, thus obviating the possibility of at least *that* specific error from occurring in the future. In other cases, the log file may exist but may not contain sufficient information to debug code. This also occurs when errors in business rules or logic may not be perceived immediately, thus relevant log files may have been deleted before they could be analyzed. In these latter cases, the code may require extensive debugging and test runs to reproduce and recapture the failure with sufficient information to understand the underlying defect.

Log files are beneficial in identifying and investigating new errors, yet developers have other tools for identifying predictable failures. A more proactive approach is to capture successful and unsuccessful process completion metrics in control tables that monitor the state of software as it executes. In the previous example, the macro %VALIDATE only wrote output to the log alerting developers to the null data set. In actual software, exception handling would instead have altered program flow to route around the failure; this exception could have been recorded in a control table, allowing data-driven processing to catch and handle the error. Moreover, in recording the type, location, and time of error in a control table, as well as other relevant attributes, the control table can become the “black box” to which developers reliably turn when disaster strikes software.

Maintenance of a risk register is another best practice for production code that is deployed operationally. The register proactively identifies known defects, faulty logic, and other weaknesses in code that could potentially cause program failure, either through a runtime error or through production of invalid data or data products. Developers often manage the register, recording pertinent details about defects, such as the conditions that would cause code to fail, the likelihood of discovery of the failure, the negative impact (i.e., cost) that a specific failure would have, a brief description of an option or options to remedy the defect, and metrics quantifying and prioritizing the severity of the defect in relation to other defects. The risk register can be viewed as a measure of technical debt because it demonstrates all known functional and performance deficiencies that exist in operational code. It also can be utilized to demonstrate the robustness of software to stakeholders, because it identifies specific factors or events—internal and external—that will cause software to fail. In so doing, decision makers can choose either to accept the risks of faulty code or mitigate or eliminate these risks through refactoring and reengineering.

After a failure, stakeholders may want to know whether the cause was anticipated (i.e., notated in the risk register) and might choose to elevate the prioritization of the defect that caused it. And, if the defect was not previously notated or anticipated, it should be added to the risk register and prioritized against other known defects. This risk management methodology prevents developers from espousing the unhealthy pattern of overcorrection, in which available resources suddenly shift toward the cause of the most recent failure. Thus, unless the most recent failure is truly prohibiting functional recovery, its remediation should be weighed against other existing technical debt on the risk register, rather than in isolation. This formal prioritization of risks helps ensure that the greatest threats to system or software reliability (or, ultimately, business value in general) are tackled first.

Efficiency in investigation is greatly improved by savvy SAS developers who are intimately familiar with the code base and organizational infrastructure. Because they already have a working understanding of process flows, dependencies, requirements, and limitations, they can more intuitively discern what might have broken and what might have caused it. These developers also may have designed and engineered the original code, and thus may be highly motivated to demonstrate that external (not internal) factors felled “their” program. But especially in cases in which non-native developers are responsible for

investigating the failure of legacy code, program organization, documentation, and standardization will facilitate more efficient and effective understanding and later modification.

3. CLEANUP

After investigation of a failure, the logical next step is either to rerun the code or to refactor or reengineer it if defects are discovered and prioritized for immediate completion. Especially in cases in which no defect exists and the software terminated due to unavoidable external circumstances, such as temporary network connection failure, the ideal recovery pattern is to restart the program without further intervention. Push-button simplicity can be thwarted, however, if software failure produces wreckage in the form of invalid or ambiguous data sets, metadata, control tables, reports, or other data products. The principle that recovery be harmless dictates that wreckage not be produced in the operational environment, thus the site of failed software should not resemble an accident scene, with bits of bumpers, broken glass, and the fetid stench of motor oil and transmission fluid seeping into the asphalt. Moreover, if developers are required to expend energy retrofitting or refurbishing the production environment following a failure, the objectives of timeliness, efficiency, and autonomy will also suffer.

A common path to undesirable wreckage results when invalid data enter a data process flow due to a lack of data integrity constraints and ultimately poison a permanent data store and its metadata. For example, consider the transactional permanent data set Food.Food_Truck with the categorical variable Food_Item that should contain values of only burrito or taco. Failure to enforce data integrity on ingested transactional updates could result in invalid data populating and polluting permanent data. These invalid observations would need to be backed out, likely through a manual process, and re-ingested once the appropriate quality control gate had been emplaced. Associated metadata that represented the quantity and nature of the failed transactional update would also need to be manually corrected to ensure that the transactional update was not counted twice. Note that in this case, despite no runtime errors having been produced, the process flow would still be considered to have failed because faulty data would have been ingested into the permanent data store. The inclusion of quality control gates, especially those that enforce the integrity of third-party data structure, format, content, and quality are critical to eliminating painful cleanup that must occur when invalid data are irresponsibly ingested.

Sometimes the design of SAS software creates ambiguous results that obfuscate the point from which code execution should resume following a failure. This often occurs in long process flows that repetitively modify permanent data sets through successive transformations. The following code simulates this confusion in a multistage ETL process flow in which successive transformation processes are demonstrated, all of which modify the permanent data set Food.Food_Truck:

```
data food.food_truck; /* represents initial ingestion */
    infile datalines delimiter=',';
    length food_item $20;
    input food_item $;
    datalines;
burrito
Burrito
Tacos
taco
;
run;

data food.food_truck; /* represents first transformation */
    set food.food_truck;
    food_item=lowcase(food_item);
run;

data food.food_truck; /* represents second transformation */
    set food.food_truck;
    if food_item in('burrito','burritos') then food_item='burrito';
    else if food_item in('taco','tacos') then food_item='taco';
run;
```

The preceding code executes without any runtime errors but, if the code does crash (due to external issues), an ambiguity must be resolved because the permanent data set Food.Food_Truck will have been created but will not be complete. In fact, in an actual ETL process flow that embodies this repetitive data set naming convention, a dozen or more disparate modules each could have overwritten the same permanent data set, thus causing exceptional confusion about the state of that data following a failure. Worse yet, if this permanent data set were to feed dependent processes or users who were waiting for its completion, those processes or analysts might detect that Food.Food_Truck existed but not be aware that it was invalid due to the process failure. And, if those dependent processes executed without further data set validation, cascading failures could erupt and cripple the infrastructure.

Some developers do prefer to use this repetitive naming convention for a couple reasons. First, in complex data process flows, clutter can be reduced by maintaining only a few data sets that are overwritten repeatedly. However, in production software, temporary data sets should not be too distracting because software should be executed in a consistent and automated manner. Second, storage limitations in some environments may prohibit fifteen copies of a data set—albeit temporary—from being produced as a process flow executes. Where storage space is limited, repetitive data set naming may be the only viable solution to handling large files, but caution must be exercised because no backup or restore data will exist if the data set is corrupted in a later transformation.

A preferred methodology is to maintain intermediate views of a data flow through permanent—not temporary—data sets that demonstrate incremental progress. The obvious advantage is that following many failures, data recovery can begin from these restore points that demonstrate the point of last known successful completion, rather than from the beginning of the process flow. Restore points also allow developers to forego the painful process of log parsing, data diving, and the creation of one-time code patches to facilitate recovery, thus facilitating timely, efficient, and autonomous recovery. The improved code now only runs the TRANS macro if INGEST completes without error, and produces intermediate permanent data sets to aid in process validation and recovery:

```
data food.control; /* create control table */
    infile datalines delimiter=',';
    length module_no 8 module $20 dt 8;
    input module_no module $;
    format dt datetime19.;
    datalines;
1,INGEST
2,TRANS1
;
run;

data food.ingest; /* represents initial ingestion */
    infile datalines delimiter=',';
    length food_item $20;
    input food_item $;
    datalines;
burrito
Burrito
Tacos
taco
;
run;
%macro ingest;
%if %eval(&SYSERR=0) %then %do; /* test ingest module */
    data food.control;
           set food.control;
           if module="&SYSMACRONAME" then dt=datetime();
    run;
    %end;
%else %abort;
%mend;

%ingest;
```

```
%macro trans;
data food.food_truck; /* represents transformation */
    set food.ingest;
    food_item=lowcase(food_item);
run;
%if %eval(&SYSERR=0) %then %do;
    data food.control;
           set food.control;
           if module="&SYSMACRONAME" then dt=datetime();
    run;
    %end;
%else %abort;
%mend;
%trans;
```

In this example, testing the value of the SAS automatic macro variable &SYSERR demonstrates whether a warning or error occurred and, if none has, the control table is updated with the date and time of completion, thus validating the process. In an actual scenario, the ingestion module could be a complicated data pull that takes several hours to complete. Thus, if software fails, developers can pinpoint the last known point of success by accessing the control table and recovery will be able to start from that point. Moreover, dependent processes or users who rely on the data set Food.Food_Truck only need to access the control table Food.Control to verify that the data set (and its prerequisite processes) have completed without warning or runtime error. A more complete example utilizing a control table with automatic checkpoints is demonstrated in the Rerun section.

In actual implementation, because control tables operate as an information bridge between the ETL code that is executing and the processes (or SAS users) that are waiting for the code to complete, caution must be exercised to ensure that two parties don't attempt to access the control table simultaneously. Thus, while a control table is being updated by the ETL process, the table will be exclusively locked so that other sessions will not be able to access it. And, correspondingly, for the split-second that users or processes are interrogating a control table to gain process metrics, a shared lock will exist that prohibits the ETL process from modifying the table. In a separate text, the author demonstrates use of the LOCKITDOWN macro, which eliminates the possibility of data collision errors in a multi-user environment, by requiring all processes to test data set availability before attempting access.[ii] All control tables used in multi-user environments should be secured with LOCKITDOWN or similar lock control methodology.

The above code also does not redirect program flow once a failure has occurred. In a more realistic exception handling scenario, rather than simply aborting the program, use of %RETURN could be implemented to terminate the immediate macro while capturing the error and propagating it (through inheritance) to other processes. In this manner, processes that depended on the failed module would be prevented from starting, but other unrelated processes would be allowed to continue, thus squeezing the most value from the code, despite the failure state. Preventing dependent processes from executing after detection of a failure is one of the most critical steps taken toward eliminating unnecessary wreckage that will require cleanup.

4. INTERNAL ISSUES AND MODIFICATION

Once software has failed and stopped, the cause and a potential remedy have been investigated, and any necessary cleanup has occurred, the next step often is simply to rerun the software. This straightforward scenario occurs when a hapless SAS developer has tripped over and unplugged a network cable, causing brief alarm as code begins to fail unexpectedly, and later jubilation as the cable is plugged in without further incident. Other straightforward scenarios can include failures caused by memory or resource errors. For example, when a junior analyst Cartesian joins two billion-record data sets to each other, the SAS server may gurgle amid troubled, agonal respirations, but likely will stall before completing the absurd task. In these scenarios, terminating the offending process manually will often allow software to restart with no further modifications to code or the SAS infrastructure.

But if defects have been identified in the code, or if management or developers have decided that the reliability of the software or system needs to be improved, emergency maintenance may be warranted. In the most exigent circumstances, hardware, network, or SAS code modifications may be required before recovery can be achieved, thus developers may be under the gun by management and other stakeholders. In some cases, less significant defects may only require subtle code refactoring, thus improving performance while maintaining equivalent functionality. In other words, the software will do the same thing—just more reliably or accurately—after developers complete the facelift.

One example of a performance enhancement involves another hapless SAS developer who noted that his code always ran the first time he executed it, but failed consistently thereafter, thus requiring him to terminate the SAS session after each run. The problem was that global macro variables created in the first execution were persisting into subsequent executions, thus causing cross-contamination and failure. When %GLOBAL or %LOCAL statements are utilized to initialize macro variables, they will initialize variables to null values if the macro variables do not exist but, if a macro variable already exists, these statements do nothing. This is illustrated in the following code sample:

```
%macro test;
%global var;
%do i=1 %to 3;
    %let var = &var &i;
    %end;
%put &var;
%mend;

%test;
```

The first time the code is executed, it correctly prints "1 2 3". However, when executed a second time from the same SAS session, it prints "1 2 3 1 2 3" because the original value of &VAR persists, despite the %GLOBAL declaration. Performance can be improved by adding a line of code that utilizes %SYMEXIST to first test the existence of the variable &VAR and to reinitialize it to Null if necessary:

```
%macro test;
%if %symexist(var) %then %let var=;
%else %global var;
%do i=1 %to 3;
    %let var = &var &i;
    %end;
%put &var;
%mend;

%test;
```

The revised code is now more robust and can be rerun without having to terminate the SAS session and resume. Other examples of internal issues that must be remedied can be much more extensive, such as when a control table or other data-driven methodology is implemented to foster more dynamic, data-driven process control. Implementation of quality assurance quality control measures can be a huge undertaking and, in fact, the lines of code representing performance enhancements may equal or even surpass the amount of code fulfilling the actual functional requirements. Thus, the risk of not implementing these controls must be assessed in relation to the anticipated benefit and cost of their inclusion.

5. EXTERNAL ISSUES AND MODIFICATION

Sometimes the cause of a failure may be external, such as a failed connection to an external database administered by a third party. In this case, the hapless developer can't just plug in the cable, and instead must begin making phone calls to attempt to ascertain the cause of the failure and its expected duration. Thus, in many cases in which an external cause is identified during investigation, all a development team can do is sit around and wait. This also can occur when some component within the SAS server or infrastructure fails but developers lack the administrative permissions or expertise to remedy the problem. SAS administrators can quickly find themselves single-threaded in these situations as they struggle alone

to bring a system back online, also emphasizing the need to identify and eliminate single points of failure in personnel that can cause delays during the recovery period.

But when connectivity to an external data source has been lost, developers still may be able to take steps to retrofit their code if higher levels of robustness and reliability are desired. For example, if the code failed due to a connectivity issue, but continued attempting to process dependent DATA steps, this opportunity allows developers the chance to implement exception handling routines that immediately recognize the failure and reroute processing, either to unrelated processes or to program termination. Another best practice is to maintain a SAS module that only tests connectivity of the third-party data source and thus can confirm before attempting a data pull whether connectivity is live. And, if connectivity is lost, developers can reuse this same SAS module to repetitively test the connection programmatically so code can resume the second connectivity is regained. This is a far more efficient and autonomous solution than a developer manually testing the connection every 20 minutes until success is achieved.

Sometimes an external connection initially will have worked but fails midway through an extraction process. In a production system, metrics about all successful and unsuccessful data pulls hopefully have been compiled and saved, thus these metadata can be used to guide the program upon resumption directly back to the point of failure. Thus, if a web services pull fails during the extraction of 1,000 XML documents, only those records that were not pulled should be queried when the code is reinitiated, substantially reducing the recovery period.

Third-party data are especially prone to causing failure because their structure, format, content, and quality lie outside the responsibility and control of the development team. This underscores the importance of erecting control gates immediately after data ingestion to validate all data attributes. Reprising the food truck example from above, if data values that include enchilada and chimichanga are sneaking into SAS data flows and polluting the results, introduction of a single line of code can remove values that are discovered that lie outside the data schema.

```
data food.food_truck; /* represents transformation */
    set food.food_truck;
    if food_item in('burrito','burritos') then food_item='burrito';
    else if food_item in('taco','tacos') then food_item='taco';
    else if food_item not in ('taco','burrito') then delete; /* Control Gate */
run;
```

In an actual control gate, the quality control mechanism should report to developers and users that enchiladas and chimichangas are being denied access, so that business rules can be inspected and potentially modified either to explicitly include or exclude these tasty observations. In this example, however, the DELETE statement serves to eliminate potential contamination caused by these observations, while other business rules might instead specify a replacement value be installed. In a separate text, the author demonstrates how to enforce data constraints dynamically through external schemas that model and validate data through exception reporting.[iii] Thus, through dynamic coding, software should remain flexible to its environment while providing feedback to stakeholders.

6. RERUN PROGRAM

Once any necessary modifications to code, hardware, or other infrastructure have been made, the SAS program should be tested, validated in the production environment, and finally executed. Because the cause of the failure was identified and remedied (or, at the very least, evaluated and included in the risk register to be prioritized for future completion), in an ideal scenario, the developer presses a button or schedules his job and confidently walks away. Often, however, execution is neither this straightforward nor autonomous, and complications can arise when the code either has not failed safe or when restore points do not exist to specify where to resume operation.

When SAS code fails in an end-user development environment, developers often make a quick fix, re-attempt the offending module, and then run the remaining code from that point. End-user developers are well placed to flexibly make these modifications on the fly because they are, by definition, the only users of the code. While this "code and fix" mentality is appropriate for end-user development, robust operational environments that host critical infrastructure should espouse software development best

practices. Thus, a more robust solution is to engineer SAS software that intelligently determines where it should begin executing. This professional design allows SAS practitioners other than the original developers to execute the code—including during both normal operation and after a failure has occurred.

For example, if a data pull fails when pulling the 50th of 60 tables from an external database, upon resumption, the software should be able to identify that 49 tables already have been extracted and thus only extract the remaining eleven tables. Hapless developers—the same bunch who trip over their network cables—might be tempted to modify the software with a quick-fix that only pulls the remaining eleven tables. Although timely, this manual solution would be *inefficient* because it would require additional development work, *inconstant* because the code would need to be modified again when the next full data pull was required, and would *lack autonomy* because the developer would have inserted himself into the recovery process. A more autonomous yet less timely solution would be to run the entire process and attempt to extract all 60 tables. Yet, this duplication of effort would be neither timely nor efficient, and where connectivity issues might exist (because a network failure already had occurred), software should aim to extract data as quickly as possible. Moreover, in organizations that measure MTTR as time from the initial failure to the point at which software is later run and surpasses that failure point, this latter strategy can drastically slow the recovery period by delaying availability and functionality.

Thus, a superior solution is to design modular code that intelligently identifies where it has failed and succeeded, and records process metrics in a control table that can be used to drive processing both under normal circumstances and exceptional circumstances, including during and following catastrophic failure. Through these best practices, timely, efficient, autonomous, constant, and harmless recovery can be achieved. Although the previous example in the Cleanup section introduces the methodology and benefits of control tables, the following example provides more detailed information. For additional reference and examples, the use of control tables in data-driven and concurrent processing models is demonstrated in a separate text by the author.[iv]

The following sections of code demonstrate the %BUILD_CONTROL macro that creates a control table if it doesn't exist. This improves autonomy because if the control table is accidentally deleted or corrupted, it can be deleted and will regenerate automatically. Note that the %SYSFUNC macro function tests only for the existence of the control table, so if another data set already exists with the same file name, the code as currently written will fail. The %UPDATE_CONTROL macro is responsible for updating the control table with the date of completion for each module that completes without runtime error:

```
%macro build_control();
%if %sysfunc(exist(controller))=0 %then %do;
    data controller;
           length process $40 date_complete 8;
           format date_complete date10.;
           if ^missing(process);
    run;
    %end;
%mend;

%macro update_control(process=);
data control_update;
    length process $40 date_complete 8;
    format date_complete date10.;
    process="&process";
    date_complete=date();
run;
proc append base=controller data=control_update;
run;
%mend;
```

The primary business rules are located in the %NEED macro, which specifies that a process should not be executed if it has already been executed on the same day. The %NEED macro is called immediately before each module in the ETL process to determine if that module needs to be executed or if can be skipped. Thus, if code failed after successfully completing one module, upon recovery, that first module would automatically be skipped. And, if the code is executing for the first time of the day under normal

circumstances, all modules will be required so all modules will be executed. This logic exemplifies constancy, because it allows the same code to be executed both under normal circumstances as well as following program failure:

```
%macro need(process=);
%global need;
%let need=YES;
%let now=%sysfunc(date());
data _null_;
    set controller;
    if strip(upcase(process))="%upcase(&process)" and date_complete=&now then call
symput('need','NO');
run;
%mend;
```

Two modules, %INGEST and %PRINT, are included here as representations of much larger modules that would exist in an actual ETL process flow. Most importantly, the automatic macro variable SYSERR tests module completion status and, only if no warnings or runtime errors were encountered, the %UPDATE_CONTROL macro is called to record this successful completion in the control table. Note that in an actual infrastructure, additional information about process failures likely would be collected during this process and passed to the control table. In addition, failure of a prerequisite process (such as %INGEST) would necessitate that dependent processes (such as %PRINT) not initiate, but for demonstration purposes, this added business logic is omitted. Thus, although this code does not fail safe, it does eliminate redundant processing that often occurs during the recovery period:

```
%macro ingest();
%let process=&SYSMACRONAME;
data ingested;
    set incoming_data;
run;
%if &SYSERR=0 %then %update_control(process=&process);
%mend;

%macro print();
%let process=&SYSMACRONAME;
proc print data=ingested;
run;
%if &SYSERR=0 %then %update_control(process=&process);
%mend;
```

The %ETL macro represents the engine that drives processing, by first building the control table if necessary and by subsequently testing the need for each module before calling it. In an actual scenario, this software might be scheduled to run once a day, thus ingesting and processing new transactional data sets. To prevent the code from inefficiently performing redundant operations, successful execution of individual modules is tracked via the control table. And, if one of the modules does fail during execution, the success of individual processes can be evaluated, and data-driven processing will immediately direct execution toward the correct module:

```
%macro etl();
%build_control;
%need(process=ingest);
%if &need=YES %then %ingest;
%need(process=print);
%if &need=YES %then %print;
%mend;

%etl;
```

CONCLUSION

Recoverability represents an often overlooked component in the quest for reliability in data analytic development. Although SAS developers should strive to develop fault-tolerant, defect-free software that

resists failure and which fails gracefully when necessary, even the most robust, reliable software can be toppled by external or unavoidable circumstances. Design and development with a recoverability mindset can ensure that software recovers in a timely, efficient, autonomous, constant, and harmless manner—consistent with the TEACH recoverability principles. All stakeholders must realize that highly available software requires developers to design not only for reliability but also for recoverability.

REFERENCES

[i] Hughes, Troy Martin. 2014. Why Aren't Exception Handling Routines Routine? Toward Reliably Robust Code through Increased Quality Standards in Base SAS. Midwest SAS Users Group (MWSUG).

[ii] Hughes, Troy Martin. 2014. From a One-Horse to a One-Stoplight Town: A Base SAS Solution to Preventing Data Access Collisions through the Detection and Deployment of Shared and Exclusive File Locks. Western Users of SAS Software (WUSS).

[iii] Hughes, Troy Martin. 2013. Binning Bombs When You're Not a Bomb Maker: A Code-Free Methodology to Standardize, Categorize, and Denormalize Categorical Data Through Taxonomical Control Tables. Southeast SAS Users Group (SESUG).

[iv] Hughes, Troy Martin. 2014. Calling for Backup When Your One-Alarm Becomes a Two-Alarm Fire: Developing Base SAS Data-Driven, Concurrent Processing Models through Fuzzy Control Tables that Maximize Throughput and Efficiency. South Central SAS Users Group (SCSUG).

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Name: Troy Martin Hughes
E-mail: troymartinhughes@gmail.com