Residues in Diagnostics



Concept by Dmitry Vostokov · DumpAnalysis.org

Residues in Diagnostics

A survey of memory dumps, traces, logs, and project records

Dmitry Vostokov · DumpAnalysis.org

Version 17 · 12 November 2026

Developed with AI assistance (GPT-6 Astra Max and Fable 5.1 Extra) and checked against the sources listed in this article.

Abstract

A function has returned, but its arguments may still be recognizable in a raw stack. What corresponds to this situation in a trace or a log? Our published Error Thread pattern makes the analogy with Execution Residue explicit. Circular Trace, Palimpsest Messages, Recovered Messages, and Incomplete History describe other situations, each with its own evidence. A Linux Stack Log lists addresses recovered from a core dump; a Windows kernel stack can retain traces of the routines that wrote the dump. We examine these examples alongside work on data lifetime, event ordering, and defect modeling. We then follow the relationship between software construction and execution through architectural patterns, the Software Narratology Square, and our early work on project postmortems. Barry O'Reilly's residuality theory gives the word residue a different architectural meaning. The publication record dates the two uses and shows where their concerns overlap.

Keywords: Execution Residue; memory dump analysis; trace and log analysis; defect modeling; software architecture; software narratology; project postmortems; residuality theory.

What survives a function return

When we open a crash dump, the debugger can usually show us some of the calls that were active at capture. A different question is whether the same dump contains anything from a call that has already returned. Unwinding the current stack and examining the raw memory occupied by the stack answer different questions. A return changes the current frames; it does not necessarily clear the bytes that held a parameter, a return address, or a local value. Later calls may overwrite some of those bytes and leave others intact. [1, 47]

In April 2008 we described Execution Residue using a small Windows program that creates two threads. In one raw stack, values associated with thread startup remained recognizable. In the thread that crashed, exception handling had used some of the same stack space. The usual backtrace and the raw contents therefore told us different things about the two threads. We called out Thread Startup Residue and Exception Handling Residue in that account and later examined Execution Residue in managed space. [1, 2]

In Clean Raw Stack Execution Residue, we investigated high CPU use in Visual C++ 2008 Express. The main thread's raw stack in the dump held addresses from many earlier calls. We reproduced the problem in a second process, cleared part of the main thread's raw stack in the debugger, and let its message loop run for one iteration. When we stopped again, we could examine what that iteration had left in the cleared region. [76]

An address that resolves to a function name does not establish that the function ran. This was the point of our earlier Coincidental Symbolic Information pattern. Historical Information asks the complementary question: what earlier behavior does a static dump actually preserve? Rough Stack Trace describes the use of candidate code addresses, while Region Strata compares what changed or persisted in successive snapshots of a region. These are different inferences from memory, even when they concern the same bytes. [46, 47] [51, pp. 932, 940]

A hang after printing

In a published case, an application hung after printing. The current stack showed a thread waiting in a printer object's destructor during process shutdown. The list of unloaded modules contained the Windows fault-reporting DLL. That clue led us to the thread's raw stack, where we found evidence of earlier exception handling. The recovered exception context pointed to heap corruption. The current wait located the thread; it did not by itself explain what had happened before shutdown. [52]

Hidden Exception and Stored Exception describe related ways in which an exception record can remain available outside the immediate context. Unloaded Module likewise concerns evidence of an image that is no longer loaded. The catalog distinguishes information captured by instrumentation from changes caused by the instrumentation itself. Information in a dump can belong to the reporting of a failure as well as to the activity that preceded it. [51, pp. 504, 580, 584, 1071, 1156]

What survives a Windows bug check

A Windows blue screen (BSOD) reports a system stop. A complete memory dump taken at that point can contain user-mode stacks as well as kernel stacks. In Version 5 of our Windows course, Exercise C1B names Execution Residue, Past Stack Trace, and Glued Stack Trace in the manual reconstruction of a WOW64 process stack. [68, p. 10] Exercises C11 and C12 cover recovering missing information from a raw stack and looking for signs of past behavior. [68, pp. 45–46] Reconstructing a process stack and identifying the cause of the system stop are different tasks.

Slide 50 links to a worked bug-check case, *Hunting for a Driver*. The stop code was RDR_FILE_SYSTEM (0x27). WinDbg rejected the first address we examined as an I/O request packet (IRP). Disassembly identified a saved argument in the caller's frame. We read the IRP pointer from that slot and verified it with `!irp`; the caller was still on the stack. The packet's device pointer led to an invalid device object with a null driver-object pointer. This explained why the dispatch failed; it did not identify which driver first made the object invalid. [68, p. 50] [69]

Writing the crash dump can itself change the evidence. In our 2021 kernel-space Execution Residue example, the dump-writing components `crashdmp` and `dump_diskdump` had overwritten earlier values on the kernel stack. We obtained the stack bounds with `!thread`, then used `dps` to examine `KiPreBugcheckStackSaveArea`, a copy saved before those components ran. That copy still contained an address resolving to `win32kfull!vExpandAndCopyText`. The value survived in the copy; it does not establish that this graphics routine caused the stop. [70]

From a Linux core dump to a Stack Log

In 2015 we described the Linux form of Execution Residue, including return addresses and pointers to strings left in a stack region. [71] Our advanced Linux course shows the difference between that material and the current backtrace. In GDB, `bt` places the selected thread in `nanosleep`, reached through `sleep` and `procNE`. With `x/512a $sp-2000` we also find values below the current stack pointer that resolve to return sites in `work_7` through `work_1`. Those functions are absent from the backtrace. Slide 44

shows the earlier call chain as a Past Stack Trace. In this course we use the term Braided Stack Trace for the arrangement illustrated on slide 45: parts of different call paths survive in the same stack region. Exercise Adv5 models this arrangement. [72, pp. 39–45, 65–66]

On slide 51 we turn the raw values into a Stack Log. Each row gives the stack location, the candidate return address, its module, symbol, and offset. The rows follow locations in a single snapshot. Their order is not an event timeline, and the application did not emit these rows while it ran. Exercises Adv2 and Adv3 use ARM64 or x64 core dumps to construct such listings. Adv8 examines residue statistics and visualization; Adv9 compares stack evidence from different runs after normalization. [72, pp. 51, 53–54, 98, 100–101]

The published counterparts in traces and logs

Error Thread is the direct published comparison with Execution Residue. In May 2014 we described following an error message or exception stack trace back through messages from the same Thread of Activity. We compared that collection with the surviving parts of earlier calls in a raw stack and with a stack trace ending at the error. The analogy concerns the history of the activity. It does not imply that the log buffer wrapped or that a message was recovered from damaged bytes. [3] [4, p. 125]

Circular Trace provides a different answer when the question is what physically survives overwrite. In our June 2009 example, displayed entry 3799 carried a timestamp from 28 May; the very next entry, 3800, carried one from 27 May. The later part of the listing then advanced toward 28 May again. The transition marked the wrap in a circular recording. Neighboring entries in the displayed file belonged to different portions of the recording; the application had not executed a day’s work backwards. [48]

Table 1. Other published patterns for interpreting a partial recording.

Published pattern	What can be read from the recording
Palimpsest Messages	Part or all of a message has been erased or overwritten. Surviving fields and related messages may constrain its former contents. [4, p. 229] [67]
Recovered Messages	A message may be undecodable because tracing metadata is missing, or its contents may be corrupt. Message Context, and sometimes source code, makes part of the information recoverable. [4, pp. 191, 252] [66]
Inter-Correlation and Intra-Correlation	Related messages can be compared across traces or within one trace. A correspondence between records is distinct from the bytes that survive in a particular record. [4, pp. 167, 170]
Hidden Error and Indirect Message	Related messages give information about something that was not reported directly. The inferred event is not itself a recovered message. [4, pp. 153, 162]
Phantom Activity	A trace shows activity without its expected external trigger. The published pattern relates this appearance to internal state that was not correctly updated. The message can be new even when the state is old. [4, p. 234]

A missing reply illustrates why these distinctions matter. The original Incomplete History entry first establishes what the code records: calls and returns, and, in the more restrictive case, exception handling as well. It also allows for tracing that stopped before a response. An outgoing call with no

visible return has a different significance in code known to trace all three paths than in a component that never logs returns. [53]

The fifth-edition catalog also distinguishes a message whose own data is cut short (Truncated Data) from a trace whose recording ends early (Truncated Trace). A Trace Window can exclude an interval or a field that is still in the source. Sparse Trace and Visibility Limit concern restrictions on what the tracer observes. None of these observations, in isolation, establishes that a previously recorded message was overwritten. [4, pp. 279, 346, 351–352, 360]

The lifetime of an observation

The lifetime of data need not coincide with the lifetime of the operation that produced it. Chow and colleagues followed sensitive data and its copies in whole-system simulations and found that some copies remained in memory beyond their intended use. Our work on memory analysis as archaeology examined what overwriting does to evidence left by an earlier computation. In a different setting, Schiffer distinguished an artifact in use from the context in which it is recovered; Loy examined physical residues on stone tools. What these examples share is a question about survival. The processes by which a blood trace and a machine address are identified are quite different. [5, pp. 115, 263] [7, 49, 54, 55]

A record has a collection history of its own. The NIST guides to digital forensics and log management describe acquisition, preservation, retention, and disposal. RFC 3227 distinguishes sources by volatility. A restart may erase process memory while leaving disk records intact. In Dapper, sampling deliberately limits which distributed requests are represented in a trace. Absence has a different meaning in a sampled trace, an overwritten buffer, and a disk file collected after a restart. [8, 9, 12, 56]

OpenTelemetry’s log data model gives this distinction a concrete expression. Timestamp denotes the time of the event; ObservedTimestamp denotes the time the collection system observed it. An old event received late and a new event reporting old state are different cases. Lamport’s happened-before relation supplies a further distinction: a send precedes its corresponding receive, whereas the display order of messages from different processes does not by itself establish a causal order. [13, 14]

The same distinction appears outside logs. On Linux, a terminated, waitable child retains status that its parent can collect. Finding that status is evidence of termination; it does not mean the child is still executing. In model-based fault detection, a residual has another meaning again: it is a discrepancy derived from observed and modeled behavior. A saved measurement can outlive the experiment, while the residual computed from it is a result of a comparison. [10, 62]

Stressors and the modeling of events and defects

O’Reilly uses a stressor to explore what happens to an architecture when a condition changes. A condition may persist, as with the loss of a server, and it may concern the business rather than a component fault. [25, 28] An event model describes occurrences and their order; a defect model describes a mechanism by which behavior can depart from a requirement. In the dependability terminology of Avizienis and colleagues, a fault is a cause attributed to an error, and an error is a state that can lead to a service failure. The occurrence of a stressor alone establishes none of these outcomes. [15]

Two published trace patterns make the event distinction concrete. In Event Sequence Order, image-load events in a trace can reveal an unexpected service startup order. The observed order matters because the system has an expected ordering of activities. In State and Event, the example is a windowing system receiving an end-session event. A message about that event and a message reporting the state in which the system was working do not describe the same occurrence. In each case the pattern concerns what a trace records about an event model; recognizing it does not identify a defect on its own. [38, 39]

In Software Defect Construction (2009), we described simulating bugs and abnormal conditions to study reproduction and dump analysis. In the 2010 work collected as Models of Software Behavior, we described running example applications, services, or drivers to produce the memory layouts associated with particular analysis patterns. The original article illustrates Multiple Exceptions with a program in which three threads encounter exceptions. [5, p. 108] [73] In Generative Debugging (2011), the behavior of a blocked activity was distinguished from the structures involved: in the published example, a thread and a pending I/O request. The Metadefect Template Library description that followed included Residue<> among its behavior classes and stacks, heaps, regions, and buffers among its structural parameters. It also described support for tracing. Residue thus has a documented place in our defect-modeling work, apart from the interpretation of a particular dump. [5, pp. 130–131] [6, 34, 37]

Our September 2009 Counterfactual Debugging posts examined a loop that writes beyond a local variable. In Dereference Fixpoints we asked whether the program would crash or keep running. [74] In Data Ordering we compared two builds of the same source. The first crashed when a write reached the stack base, as the dump showed. Changing a compiler setting reversed the order of the variable and its pointer on the stack. In the second build the pointer held its own address, and the loop kept running. The two runs showed how a change in stack layout could give the same defect different symptoms. [75]

Burrows’s Chubby paper gives an example of the underlying problem in a distributed system. A former lock holder’s request can reach a file server after another holder has acquired the lock. Chubby’s sequencer lets a cooperating server check which acquisition the request belongs to. The trace of a request arriving late establishes its arrival. Whether the operation was accepted is a separate fact about the receiving server. The identity and sequence information have meaning because the protocol defines them. [57, §2.4]

Reiter’s account of model-based diagnosis allows more than one diagnosis to be consistent with the observations and the system description. Sampath and colleagues give a precise treatment of diagnosability for partially observed discrete-event systems: whether the observations eventually distinguish a failure. In software, Yuan and colleagues’ LogEnhancer selected program state to include at existing log sites and evaluated the result on eight applications. These works address, in different settings, how much a recorded observation can distinguish. A match between two sequences of messages is weaker evidence than a distinction the model or instrumentation actually preserves. [16, 58, 59]

Architectural decisions in execution artifacts

A computational system moving between changed conditions leaves different traces of its earlier states. What survives depends in part on the architecture. SQLite illustrates this without requiring two different applications: the same database engine has two documented ways of protecting a

transaction. Both support recovery, but they leave different information in the accompanying file. [63, 65]

Table 2. What SQLite writes in two transaction modes [63, 65].

Mode	Contents of the accompanying file	Relation to the main database
Rollback journal	Original contents of pages about to be changed.	Changes are written to the database. A hot journal can supply earlier page images during recovery.
Write-ahead log (WAL)	New page images are appended to the WAL. A commit is marked there.	The main database can retain the earlier page contents until checkpointing transfers the committed changes.

After a commit in WAL mode, the latest committed values can still be in the WAL. Opening the main database file as a sequence of pages and ignoring its associated WAL can therefore give a different picture from that seen by a database reader. Conversely, finding an old page in a rollback journal does not show that the application wrote that old value as its new result. The architectural decision changes the meaning of the bytes that survive. [63, §§2–2.2] [65, §§3.5, 4.4–4.5]

Records also depend on what the architecture treats as an event. In Fowler's account of Event Sourcing, recorded application events can be replayed to rebuild application state. Dapper's spans, by contrast, represent selected work and its relationships across services. A sequence of state-changing business events and a sampled distributed trace are both histories of aspects of execution. Replaying the former and reading the latter answer different questions. Our account of the semiotics of memory and trace analysis likewise distinguishes a representation from its connection with the process that produced it. A transaction identifier is intelligible as a symbol because a protocol assigns it meaning. Its presence alone does not certify a successful transaction. [11, 12, 40]

Bridging software construction and execution

In our 2013 description of Motif, client–server exchanges, publisher–subscriber notifications, database queries, and plug-in initialization appear as recognizable arrangements of trace messages. The article relates software narremes to implementation patterns, Macrofunctions to design patterns, and Motifs to architectural patterns, allowing overlap between the levels. An exchange can reveal a client and a server even when their names have changed between products. One message saying that a request was sent tells us considerably less about their relationship. [4, p. 214] [30]

Bridging the Great Divide connects this trace work with our earlier use of pattern catalogs to relate software construction and memory dump analysis. The Pattern-Based Software Diagnostics presentation was published in transcript form in August 2013. In the collected theoretical articles, Projective Debugging describes an example in which a model derived from trace information reproduced a synchronization deadlock. That result describes the behavior of the projected model. Its agreement with a trace does not restore events absent from the recording. [5, pp. 169, 209] [31, 32]

There were already other ways of examining the relationship between a system and its architectural description. Murphy, Notkin, and Sullivan's software reflexion models compare relationships derived from source code with a high-level model. Kazman and Carrière's Dali workbench reconstructs

architecture from available implementation evidence. The distinction in our Motif work is the use of recognizable interactions in *execution artifacts*. A relationship present in source code and a relationship exercised by one recorded request are different kinds of evidence. [30, 33, 60]

The Software Narratology Square

Our 2009 definition of software narratology distinguished the operations in a program, the path taken in an execution, its representation in a trace, and the trace as displayed. Selecting a different interval can change the story a reader sees while the stored trace remains identical. The March 2012 articles on narrative planes and the Software Narratology Square placed software traces alongside three other kinds of account. The later definition that month explicitly included both requirements use cases and reports of interactions during incidents. [41–44]

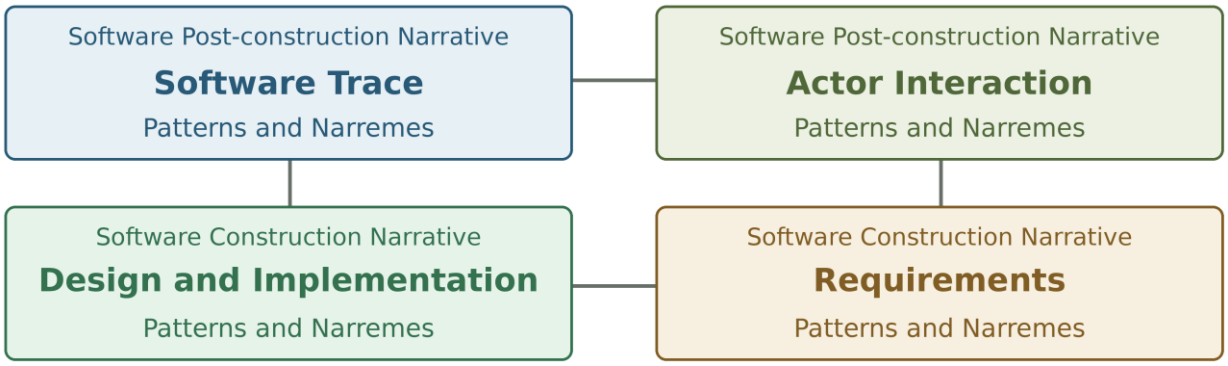


Figure 1. The Software Narratology Square, redrawn from our diagram of 11 March 2012. Both construction and post-construction narratives have patterns and narremes. [43] [45, p. 14]

The upper and lower rows answer different questions. A requirement describes an intended interaction. An incident account describes what an actor experienced. Design and implementation describe how the software was constructed; a trace records selected events from a particular execution. A reported action does not acquire a timestamp in the program's trace simply because it appears in the incident narrative. This is one reason the distinction between construction and post-construction matters when only part of a recording survives. [42–45]

The seminar transcript, *Software Narratology: An Introduction to the Applied Science of Software Stories*, was first published in April 2012. Page 14 of the June 2015 revision discusses the square. The next pages distinguish story, plot, and presentation, and describe supporting material such as scenario accounts, images, and interviews. The request/reply and entry/exit examples illustrate a narreme, a basic unit of a software narrative. A surviving reply can make an earlier request intelligible without supplying its missing payload. The presentation also describes including event descriptions during construction for use in later diagnosis. [45, pp. 14–19, 24–26]

There is a parallel in our published Dump Context pattern. The issue description, collection circumstances, system information, and earlier captures can accompany a memory dump. An earlier analysis may explain why a second dump was requested; it is evidence about the investigation rather than another record of the execution captured in that second dump. [64]

From crash dump analysis to project postmortems

In December 2007 we introduced the Management Bits and Tips blog. The contemporary index links entries on Project Failure Analysis Patterns dated 18 and 25 December. Our summary of 4 January 2008 explicitly describes the transfer of crash dump analysis patterns to project failures. In the 2010 anthology announcement we explained the analogy in terms of teams as processes and individuals as threads. The published collection, *Critique of Management Reason*, includes the introduction and three parts of Project Failure Analysis Patterns on pp. 47–53. [17–20]

A distinction matters here. The subject of those articles was the transfer of crash and hang analysis ideas to organizational project postmortems. Project postmortems themselves already had a literature. In 1996 Collier, DeMarco, and Fearey described collecting project measures alongside a participant survey, a debriefing, and a project-history meeting. A schedule can record when a milestone moved. It does not preserve everything the participants knew about a dependency at the time. Their account explains why the measures and the participants' recollections were collected separately. [61]

Walsh and Ungson treat organizational memory as information retained not only by individuals but also in organizational arrangements and routines. That distinction helps explain how a decision can outlive its project in two different senses: a record may preserve why it was made, while a procedure can continue to determine who does the work. In their account of production incident postmortems, Lunney and Lueder likewise distinguish the incident narrative from follow-up actions and their owners. A project review and a production incident review have different scopes, even when both draw on records and recollections. [21, 22, 61]

Residuality in software architecture

Barry O'Reilly uses *residue* in an architectural design method. In his 2020 introductory paper, it denotes what remains of a system under a particular stressor. The description can include people, information flows, infrastructure, and software. [25, 26] The initial residue may be damaged or unable to function in the changed context. The architect examines and augments it, so the resulting description may include changes never deployed. [28] A value recovered from a stack comes from a particular execution. Our Counterfactual Debugging example answered a what-if question by running the second build. [75] In O'Reilly's method, the architect can ask what would remain under a stressor the system has never encountered. [25, 28]

O'Reilly's electric-vehicle charging example makes the difference concrete. Server loss during a charging session exposes a dependency between stopping the charge and unlocking the vehicle. The customer can be left attached to the charging point. The example records how the changed condition would be detected and uses a matrix to show which components it affects. [28] The scenario identifies a dependency in the design. An incident trace would answer a different question: did that dependency prevent a particular vehicle from being unlocked?

O'Reilly discusses failure mode and effects analysis (FMEA) and the Architecture Tradeoff Analysis Method (ATAM) in evaluating an architecture. His empirical procedure uses a different set of stressors from those used to develop the design. [28] The charging example illustrates the reasoning; it does not establish how the method performs across projects. His 2021 philosophical treatment addresses changing conditions, his 2022 paper examines random simulation and attractor networks, and his 2023 paper discusses the limits of architectural representations. [27, 35, 36]

O'Reilly uses *residual causality* for the way earlier software design decisions can restrict what people are able to do later. [50] A stored decision can document why a rule was introduced; a rule still enforced by software can constrain action. These are different consequences of the same decision. O'Reilly also places architectural decisions in a broader discussion of uncertainty and process. [29]

The publication history

The December 2007 project entries and January 2008 summary document our transfer of dump patterns to project failures. April 2008 dates the named Execution Residue pattern. The two Counterfactual Debugging posts and the message-loop experiment appeared in September 2009. [74–76] The Metadefect Template Library description followed in 2011, the Software Narratology Square in 2012, the architectural Motif mapping in 2013, and the explicit Error Thread analogy in 2014. These dates refer to particular published descriptions. [1, 3, 6, 17, 18, 30, 43]

On the architectural side, O'Reilly and Mac Críosta's precursor appeared in the July 2018 issue of *Cutter Business Technology Journal*. O'Reilly's 2019 antifragility paper preceded his 2020 introduction to residuality theory. [23–26] The chronology establishes earlier publication of our named diagnostic pattern and its trace analogy. It does not establish priority over O'Reilly's architectural method or show that either body of work influenced the other. Reasoning from material remains, organizational memory, and architectural reconstruction all have earlier, independent histories. [7, 21, 33, 54]

In trace and log analysis, the counterpart to Execution Residue depends on what the recording preserves. Messages from the activity leading to an error are the subject of Error Thread. The older records at the wrap in our 2009 listing are the subject of Circular Trace. A missing return in code that records requests, returns, and exceptions is the situation described by Incomplete History. In each case, we also need to know how the recording was produced. [3, 48, 53]

References

Page numbers for [4] and [5] refer to the author's copies identified in those entries. The same page numbers appear in the public tables of contents linked there. For the software narratology presentation [45], the first publication and the revision used for page citations are distinguished. Page numbers for the Windows and Linux slide decks [68, 72] count the title slide as page 1. Bracketed pattern names in the blog references identify the relevant catalog entries.

[1] Vostokov, D. "Crash Dump Analysis Patterns (Part 60)" [Execution Residue (unmanaged space, user)]. Software Diagnostics Library, 29 April 2008. https://www.dumpanalysis.org/blog/index.php/2008/04/29/crash-dump-analysis-patterns-part-60/

[2] Vostokov, D. "Crash Dump Analysis Patterns (Part 60b)" [Execution Residue (managed space)]. Software Diagnostics Library, 17 October 2011. https://www.dumpanalysis.org/blog/index.php/2011/10/17/crash-dump-analysis-patterns-part-60b/

[3] Vostokov, D. "Trace Analysis Patterns (Part 86)" [Error Thread]. Software Diagnostics Library, 20 May 2014. https://www.dumpanalysis.org/blog/index.php/2014/05/20/trace-analysis-patterns-part-86/

[4] Vostokov, D. Trace, Log, Text, Narrative, Data: An Analysis Pattern Reference for Information Mining, Diagnostics, Anomaly Detection. Fifth edition. OpenTask, March 2023; public revision 5.00, February 2023. Author-supplied copy consulted: revision 5.06, August 2026. ISBN 978-1-912636-58-7. Entries discussed: Circular Trace, p. 78; Error Thread, p. 125; Hidden Error, p. 153; Incomplete History, p. 159; Indirect Message, p. 162; Inter-Correlation, p. 167; Intra-Correlation, p. 170; Message Context, p. 191; Motif, p. 214; Palimpsest Messages, p. 229; Phantom Activity, p. 234; Recovered Messages, p. 252; Sparse Trace, p. 279; Trace Window, p. 346; Truncated Data, p. 351; Truncated Trace, p. 352; Visibility Limit, p. 360. https://www.dumpanalysis.org/trace-log-analysis-pattern-reference Public title pages and contents: https://www.dumpanalysis.org/TAPR/Trace-Log-Text-Narrative-Data-Analysis-Pattern-Reference-Fifth-Edition-TOC.pdf

[5] Vostokov, D. Theoretical Software Diagnostics: Collected Articles. Fourth edition. OpenTask, 2024; public revision 4.00, February 2024. Author-supplied copy consulted: revision 4.01, April 2024. ISBN 978-1-912636-91-4. See Models of Software Behavior, p. 108; Archaeological Foundations for Memory Analysis, p. 115; Generative Debugging, p. 130; Metadefect Template Library, p. 131; Bridging the Great Divide, p. 169; Projective Debugging, p. 209; Software Diagnostics as Archaeology, p. 263. Public title pages and contents: https://www.dumpanalysis.org/TSD/Theoretical-Software-Diagnostics-Fourth-Edition-TOC.pdf

[6] Vostokov, D. “Metadefect Template Library (Part 0).” Software Diagnostics Library, 1 May 2011. https://www.dumpanalysis.org/blog/index.php/2011/05/01/metadefect-template-library-part-0/

[7] Schiffer, M. B. “Archaeological Context and Systemic Context.” American Antiquity 37(2), 1972, pp. 156–165. https://doi.org/10.2307/278203

[8] Kent, K., Chevalier, S., Grance, T., and Dang, H. Guide to Integrating Forensic Techniques into Incident Response. NIST SP 800-86, August 2006. https://doi.org/10.6028/NIST.SP.800-86

[9] Kent, K., and Souppaya, M. Guide to Computer Security Log Management. NIST SP 800-92, September 2006. https://doi.org/10.6028/NIST.SP.800-92

[10] Isermann, R. “Model-based fault-detection and diagnosis—status and applications.” Annual Reviews in Control 29(1), 2005, pp. 71–85. https://doi.org/10.1016/j.arcontrol.2004.12.002

[11] Fowler, M. “Event Sourcing.” 12 December 2005. https://martinfowler.com/eaaDev/EventSourcing.html

[12] Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Google Technical Report dapper-2010-1, April 2010. https://research.google/pubs/dapper-a-large-scale-distributed-systems-tracing-infrastructure/

[13] OpenTelemetry. “Logs Data Model.” Specification, accessed 8 November 2026; especially Timestamp, ObservedTimestamp, and trace context fields. https://opentelemetry.io/docs/specs/otel/logs/data-model/

[14] Lamport, L. “Time, Clocks, and the Ordering of Events in a Distributed System.” Communications of the ACM 21(7), 1978, pp. 558–565. https://doi.org/10.1145/359545.359563

[15] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. “Basic Concepts and Taxonomy of Dependable and Secure Computing.” IEEE Transactions on Dependable and Secure Computing 1(1), 2004, pp. 11–33. https://doi.org/10.1109/TDSC.2004.2

[16] Reiter, R. “A Theory of Diagnosis from First Principles.” Artificial Intelligence 32(1), 1987, pp. 57–95. https://doi.org/10.1016/0004-3702(87)90062-2

[17] Vostokov, D. “Management Bits and Tips blog.” Software Diagnostics Library, 18 December 2007. https://www.dumpanalysis.org/blog/index.php/2007/12/18/management-bits-and-tips-blog/

[18] Vostokov, D. “ManagementBits update (December, 2007).” Software Diagnostics Library, 4 January 2008. https://www.dumpanalysis.org/blog/index.php/2008/01/04/managementbits-update-december-2007/

[19] Vostokov, D. “Forthcoming Management Bits: An Anthology from Reductionist Manager.” Software Diagnostics Library, 10 July 2010. Announces the project under its earlier title; see [20] for the published paperback. https://www.dumpanalysis.org/blog/index.php/2010/07/10/forthcoming-management-bits-an-anthology-from-reductionist-manager/

[20] Vostokov, D. Critique of Management Reason: Management Bits and Tips from Can't. OpenTask paperback, Version 1.0, November 2025, ISBN 978-1-906717-13-1. Project Failure Analysis Patterns, pp. 47–53. Paperback title pages and contents: https://www.dumpanalysis.org/CMR/Critique-of-Management-Reason-TOC.pdf Leanpub version updated 16 January 2026; the author's description gives the title history: https://leanpub.com/bits

[21] Walsh, J. P., and Ungson, G. R. "Organizational Memory." Academy of Management Review 16(1), 1991, pp. 57–91. https://doi.org/10.5465/amr.1991.4278992

[22] Lunney, J., and Lueder, S. "Postmortem Culture: Learning from Failure." In Beyer, B., Jones, C., Petoff, J., and Murphy, N. R. (eds.), Site Reliability Engineering: How Google Runs Production Systems, chapter 15. O'Reilly Media, 2016. Online edition accessed 8 November 2026. https://sre.google/sre-book/postmortem-culture/

[23] O'Reilly, B. M., and Mac Críosta, G. "No More Snake Oil: Architecting Agility in a Complex Environment." Cutter Business Technology Journal 31(7), July 2018, pp. 21–26. ISSN 2475-3718 (print), 2475-3742 (online). Web posting: 4 September 2018. Archived posting (2 April 2023; accessed 8 November 2026): https://web.archive.org/web/20230402153941/https://www.cutter.com/article/no-more-snake-oil-architecting-agility-complex-environment-500536

[24] O'Reilly, B. M. "No More Snake Oil: Architecting Agility through Antifragility." Procedia Computer Science 151, 2019, pp. 884–890. https://doi.org/10.1016/j.procs.2019.04.122

[25] O'Reilly, B. M. "An Introduction to Residuality Theory: Software Design Heuristics for Complex Systems." Procedia Computer Science 170, 2020, pp. 875–880. https://doi.org/10.1016/j.procs.2020.03.120

[26] O'Reilly, B. M. "There Is No Spoon"—The Path to Residuality Theory: A Collection of Articles from Cutter Consortium, 2018–2021. Cutter Consortium, 2021. Includes "There Is No Spoon": Residuality Theory & Rethinking Software Engineering, Business Agility & Software Engineering Excellence Executive Update 21(5), 2020, posted 20 July 2020, pp. 1–7. Archived collection (2 April 2023; accessed 8 November 2026): https://web.archive.org/web/20230402061305id_/https://www.cutter.com/sites/default/files/APM/2021/The%20Path%20to%20Residuality%20Theory%20Collection.pdf

[27] O'Reilly, B. M. "Residuality and Representation: Toward a Coherent Philosophy of Software Architecture." Procedia Computer Science 224, 2023, pp. 91–97. https://doi.org/10.1016/j.procs.2023.09.015

[28] O'Reilly, B. M. Residues: Time, Change, and Uncertainty in Software Architecture. Leanpub, version updated 2 June 2024. https://leanpub.com/residuality

[29] O'Reilly, B. M. The Architect's Paradox: Uncertainty and the Philosophy of Software Architecture. Leanpub, version updated 18 September 2025. https://leanpub.com/architectsparadox

[30] Vostokov, D. "Trace Analysis Patterns (Part 74)" [Motif; "Motifs" in the original post]. Software Diagnostics Library, 19 July 2013. https://www.dumpanalysis.org/blog/index.php/2013/07/19/trace-analysis-patterns-part-74/

[31] Vostokov, D. "Bridging the Great Divide." Software Diagnostics and Observability Institute; web page undated. Also collected in Theoretical Software Diagnostics, fourth edition, p. 169. https://www.dumpanalysis.org/the-great-divide

[32] Vostokov, D., and Software Diagnostics Services. Pattern-Based Software Diagnostics: An Introduction. Presentation transcript. OpenTask, August 2013. ISBN 978-1-908043-49-8. https://www.dumpanalysis.org/introduction-pattern-based-software-diagnostics

[33] Kazman, R., and Carrière, S. J. Playing Detective: Reconstructing Software Architecture from Available Evidence. Technical Report CMU/SEI-97-TR-010, Software Engineering Institute, Carnegie Mellon University, October 1997. https://www.sei.cmu.edu/library/playing-detective-reconstructing-software-architecture-from-available-evidence/

[34] Vostokov, D. "Generative Debugging (Part 0)." Software Diagnostics Library, 15 April 2011. https://www.dumpanalysis.org/blog/index.php/2011/04/15/generative-debugging-part-0/

[35] O'Reilly, B. M. "The Philosophy of Residuality Theory." Procedia Computer Science 184, 2021, pp. 809–816. https://doi.org/10.1016/j.procs.2021.03.101

[36] O'Reilly, B. M. "Residuality Theory, random simulation, and attractor networks." Procedia Computer Science 201, 2022, pp. 639–645. https://doi.org/10.1016/j.procs.2022.03.084

[37] Vostokov, D. "Software Defect Construction." Software Diagnostics Library, 16 June 2009. https://www.dumpanalysis.org/blog/index.php/2009/06/16/software-defect-construction/

[38] Vostokov, D. "Trace Analysis Patterns (Part 33)" [Event Sequence Order]. Software Diagnostics Library, 24 November 2010. https://www.dumpanalysis.org/blog/index.php/2010/11/24/trace-analysis-patterns-part-33/

[39] Vostokov, D. "Trace Analysis Patterns (Part 70)" [State and Event]. Software Diagnostics Library, 4 May 2013. https://www.dumpanalysis.org/blog/index.php/2013/05/04/trace-analysis-patterns-part-70/

[40] Vostokov, D. "Pattern-Oriented Observability (Part 2): Semiotics of Memory, Trace, and Log Analysis." Software Diagnostics and Observability Institute, web page undated; accessed 8 November 2026. https://dumpanalysis.org/pattern-oriented-observability-part2

[41] Vostokov, D. "Software Narratology: A Definition." Software Diagnostics Library, 12 June 2009. Also collected in Theoretical Software Diagnostics, fourth edition, p. 58. https://www.dumpanalysis.org/blog/index.php/2009/06/12/software-narratology-a-definition/

[42] Vostokov, D. "Software Narrative Planes." Software Diagnostics Library, 11 March 2012. Also collected in Theoretical Software Diagnostics, fourth edition, pp. 145–146. https://www.dumpanalysis.org/blog/index.php/2012/03/11/software-narrative-planes/

[43] Vostokov, D. "Software Narratology Square." Software Diagnostics Library, 11 March 2012. Original diagram inspected. Also collected in Theoretical Software Diagnostics, fourth edition, p. 147. https://www.dumpanalysis.org/blog/index.php/2012/03/11/software-narratology-square/

[44] Vostokov, D. "What is a Software Narrative?" Software Diagnostics Library, 20 March 2012. Also collected in Theoretical Software Diagnostics, fourth edition, p. 144. https://www.dumpanalysis.org/blog/index.php/2012/03/20/what-is-a-software-narrative/

[45] Vostokov, D., and Memory Dump Analysis Services. Software Narratology: An Introduction to the Applied Science of Software Stories. OpenTask, April 2012. ISBN 978-1-908043-07-8. Public listing: 26-page paperback. Complete author-supplied presentation transcript consulted: revision 2.0, June 2015, 27 PDF pages; cover and slide title: Software Narratology: Introduction, Version 1.0. The revision date is stated on the copyright page. See p. 14 (square), p. 15 (story, plot, and presentation), pp. 16–19 (supporting context and narremes), and pp. 24–26 (anticipatory construction and product independence). https://dumpanalysis.org/introduction-software-narratology

[46] Vostokov, D. "Crash Dump Analysis Patterns (Part 24)" [Coincidental Symbolic Information]. Software Diagnostics Library, 30 August 2007. https://www.dumpanalysis.org/blog/index.php/2007/08/30/crash-dump-analysis-patterns-part-24/

[47] Vostokov, D. "Crash Dump Analysis Patterns (Part 34)" [Historical Information]. Software Diagnostics Library, 6 November 2007. https://www.dumpanalysis.org/blog/index.php/2007/11/06/crash-dump-analysis-patterns-part-34/

[48] Vostokov, D. "Trace Analysis Patterns (Part 3)" [Circular Trace]. Software Diagnostics Library, 3 June 2009. https://www.dumpanalysis.org/blog/index.php/2009/06/03/trace-analysis-patterns-part-3/

[49] Vostokov, D. "Archaeological Foundations for Memory Analysis." Software Diagnostics Library, 22 April 2010. https://www.dumpanalysis.org/blog/index.php/2010/04/22/archaeological-foundations-for-memory-analysis/

[50] O'Reilly, B. M. "The Machine in the Ghost: Autonomy, Hyperconnectivity, and Residual Causality." Philosophies 6(4), article 81, 30 September 2021. https://doi.org/10.3390/philosophies6040081

[51] Vostokov, D., and Software Diagnostics Institute. Encyclopedia of Crash Dump Analysis Patterns: Detecting Abnormal Software Structure and Behavior in Computer Memory. Third edition, Version 3.00. OpenTask, September 2020. ISBN 978-1-912636-30-3. Public title pages and contents: https://dumpanalysis.org/ECDAP/Encyclopedia-Crash-Dump-Analysis-Patterns-Third-Edition-TOC.pdf

[52] Vostokov, D. “Blocked thread, historical information, execution residue, hidden exception, dynamic memory corruption, incorrect stack trace and not my version: pattern cooperation.” Software Diagnostics Library, 23 April 2009. https://www.dumpanalysis.org/blog/index.php/2009/04/23/blocked-thread-historical-information-execution-residue-hidden-exception-dynamic-memory-corruption-incorrect-stack-trace-and-not-my-version-pattern-cooperation/

[53] Vostokov, D. “Trace Analysis Patterns (Part 19)” [Incomplete History]. Software Diagnostics Library, 5 April 2010. https://www.dumpanalysis.org/blog/index.php/2010/04/05/trace-analysis-patterns-part-19/

[54] Loy, T. H. “Prehistoric Blood Residues: Detection on Tool Surfaces and Identification of Species of Origin.” Science 220(4603), 1983, pp. 1269–1271. https://doi.org/10.1126/science.220.4603.1269

[55] Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., and Rosenblum, M. “Understanding Data Lifetime via Whole System Simulation.” 13th USENIX Security Symposium, 2004, pp. 321–336. https://www.usenix.org/conference/13th-usenix-security-symposium/understanding-data-lifetime-whole-system-simulation

[56] Brezinski, D., and Killalea, T. Guidelines for Evidence Collection and Archiving. BCP 55, RFC 3227, February 2002, especially §2.1, Order of Volatility. https://www.rfc-editor.org/rfc/rfc3227

[57] Burrows, M. “The Chubby Lock Service for Loosely-Coupled Distributed Systems.” 7th USENIX Symposium on Operating Systems Design and Implementation, 2006, pp. 335–350, especially §2.4, Locks and Sequencers. https://research.google/pubs/the-chubby-lock-service-for-loosely-coupled-distributed-systems/

[58] Sampath, M., Sengupta, R., Lafortune, S., Sinnamohideen, K., and Teneketzis, D. “Diagnosability of Discrete-Event Systems.” IEEE Transactions on Automatic Control 40(9), 1995, pp. 1555–1575. https://doi.org/10.1109/9.412626

[59] Yuan, D., Zheng, J., Park, S., Zhou, Y., and Savage, S. “Improving Software Diagnosability via Log Enhancement.” ACM Transactions on Computer Systems 30(1), article 4, 2012. Extended version of the ASPLOS 2011 paper. https://doi.org/10.1145/2110356.2110360

[60] Murphy, G. C., Notkin, D., and Sullivan, K. “Software Reflexion Models: Bridging the Gap between Source and High-Level Models.” Third ACM SIGSOFT Symposium on the Foundations of Software Engineering, 1995, pp. 18–28. https://doi.org/10.1145/222124.222136

[61] Collier, B., DeMarco, T., and Fearey, P. “A Defined Process for Project Postmortem Review.” IEEE Software 13(4), July 1996, pp. 65–72. https://doi.org/10.1109/52.526833 Open copy: https://courses.cs.umbc.edu/undergraduate/345/spring12/mitchell/readings/aDefinedProcessForProjectPostMortemReview.pdf

[62] Linux man-pages project. “wait(2): wait for process to change state.” See Notes on retaining a terminated, waitable child’s status until it is collected. Accessed 8 November 2026. https://man7.org/linux/man-pages/man2/wait.2.html

[63] SQLite. “Write-Ahead Logging.” See §§2.1–2.2 on checkpointing and concurrency. Accessed 8 November 2026. https://sqlite.org/wal.html

[64] Vostokov, D. “Crash Dump Analysis Patterns (Part 301)” [Dump Context]. Software Diagnostics Library, 21 October 2025. https://www.dumpanalysis.org/blog/index.php/2025/10/21/crash-dump-analysis-patterns-part-301/

[65] SQLite. “Atomic Commit in SQLite.” See especially §3.5 on writing the rollback journal and §§4.4–4.5 on recovering from incomplete changes. Accessed 8 November 2026. https://sqlite.org/atomiccommit.html

[66] Vostokov, D. “Trace Analysis Patterns (Part 107)” [Recovered Messages]. Software Diagnostics Library, 3 May 2015. https://www.dumpanalysis.org/blog/index.php/2015/05/03/trace-analysis-patterns-part-107/

[67] Vostokov, D. “Trace Analysis Patterns (Part 108)” [Palimpsest Messages]. Software Diagnostics Library, 13 May 2015. https://www.dumpanalysis.org/blog/index.php/2015/05/13/trace-analysis-patterns-part-108/

[68] Vostokov, D. Advanced Windows Memory Dump Analysis with Data Structures. Version 5. Software Diagnostics Services, 2024. Public slides, 54 pages. Exercise C1B, p. 10; C11 and C12, pp. 45–46; pattern links and case studies, pp. 49–50. https://www.patterndiagnostics.com/Training/Advanced-Windows-Memory-Dump-Analysis-Version5-Slides.pdf Companion book: OpenTask, revision 5.01 (March 2024), ISBN 978-1-912636-95-2. The published contents list C1B on p. 77, C11 on p. 293, and C12 on p. 301: https://www.dumpanalysis.org/AdvWMDA/Advanced-Windows-Memory-Dump-Analysis-Version5-TOC-Sample-Exercise.pdf

[69] Vostokov, D. "Hunting for a Driver." Software Diagnostics Library, 6 July 2009. Case linked from [68, p. 50]. https://www.dumpanalysis.org/blog/index.php/2009/07/06/hunting-for-a-driver/

[70] Vostokov, D. "Crash Dump Analysis Patterns (Part 60c)" [Execution Residue (unmanaged space, kernel)]. Software Diagnostics Library, 3 October 2021. https://www.dumpanalysis.org/blog/index.php/2021/10/03/crash-dump-analysis-patterns-part-60c/

[71] Vostokov, D. "Crash Dump Analysis Patterns (Part 60, Linux)" [Execution Residue]. Software Diagnostics Library, 19 December 2015. https://www.dumpanalysis.org/blog/index.php/2015/12/19/crash-dump-analysis-patterns-part-60-linux/

[72] Vostokov, D. Linux Core Dump Analysis — Advanced: Data Structures and Generative AI. Software Diagnostics Services, 2026. Public slide preview, 130 pages. Past behavior and stack traces, pp. 39–45; Stack Log and GDB exercises, pp. 51–54; Braided Stack Trace and Exercise Adv5, pp. 65–66; residue statistics and comparisons across runs, pp. 98–101. https://www.patterndiagnostics.com/Training/Advanced-Linux-Core-Dump-Analysis-Slides-Preview.pdf Course listing: https://www.dumpanalysis.org/advanced-linux-core-dump-analysis

[73] Vostokov, D. "Models for Memory and Trace Analysis Patterns (Part 1)." Software Diagnostics Library, 16 May 2010. https://www.dumpanalysis.org/blog/index.php/2010/05/16/models-for-memory-and-trace-analysis-patterns-part-1/

[74] Vostokov, D. "Counterfactual Debugging: Dereference Fixpoints." Software Diagnostics Library, 11 September 2009. https://www.dumpanalysis.org/blog/index.php/2009/09/11/counterfactual-debugging-dereference-fixpoints/

[75] Vostokov, D. "Counterfactual Debugging: Data Ordering." Software Diagnostics Library, 15 September 2009. https://www.dumpanalysis.org/blog/index.php/2009/09/15/counterfactual-debugging-data-ordering/

[76] Vostokov, D. "Clean Raw Stack Execution Residue." Software Diagnostics Library, 23 September 2009. https://www.dumpanalysis.org/blog/index.php/2009/09/23/clean-raw-stack-execution-residue/