Windows Software Trace Analysis

Accelerated

Dmitry Vostokov
Software Diagnostics Services

Published by OpenTask, Republic of Ireland

OpenTask books and magazines are available through booksellers and distributors worldwide. For further information or comments send requests to press@opentask.com.

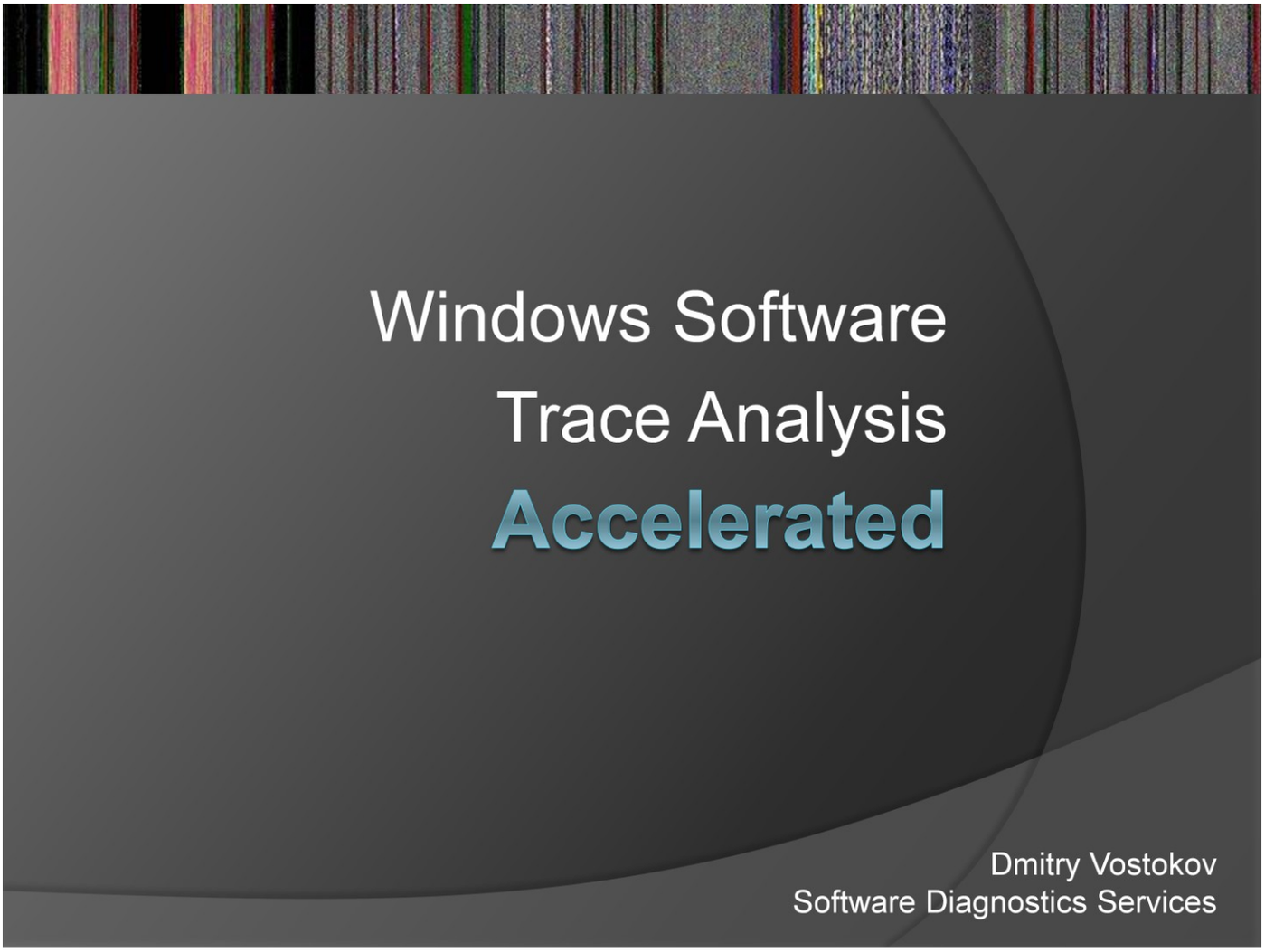
A CIP catalogue record for this book is available from the British Library.

ISBN-l3: 978-1-908043-42-9 (Paperback)

Revision 2 (February 2016)

Contents

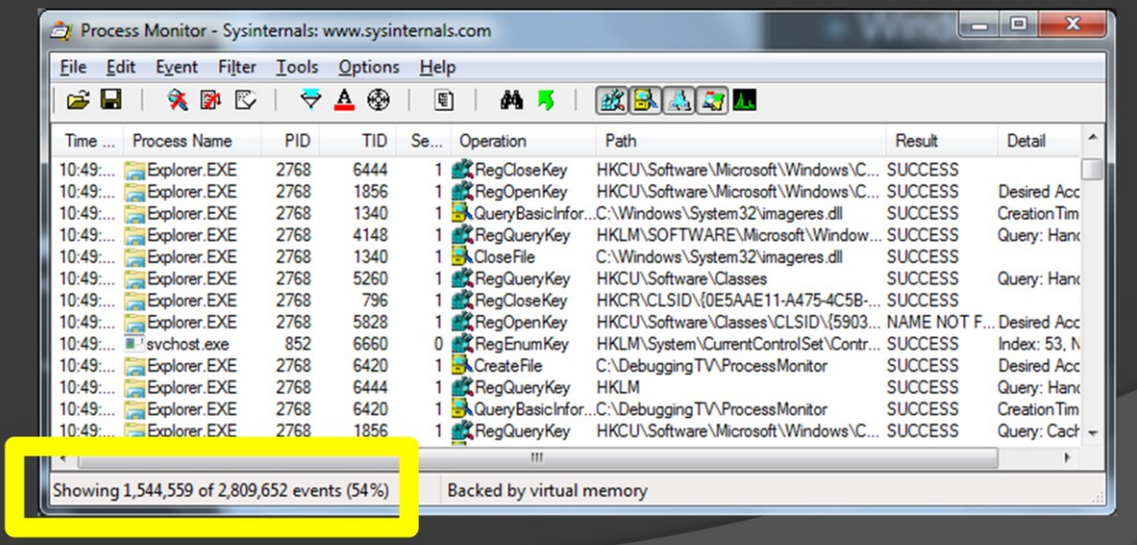
Presentation Slides and Transcript



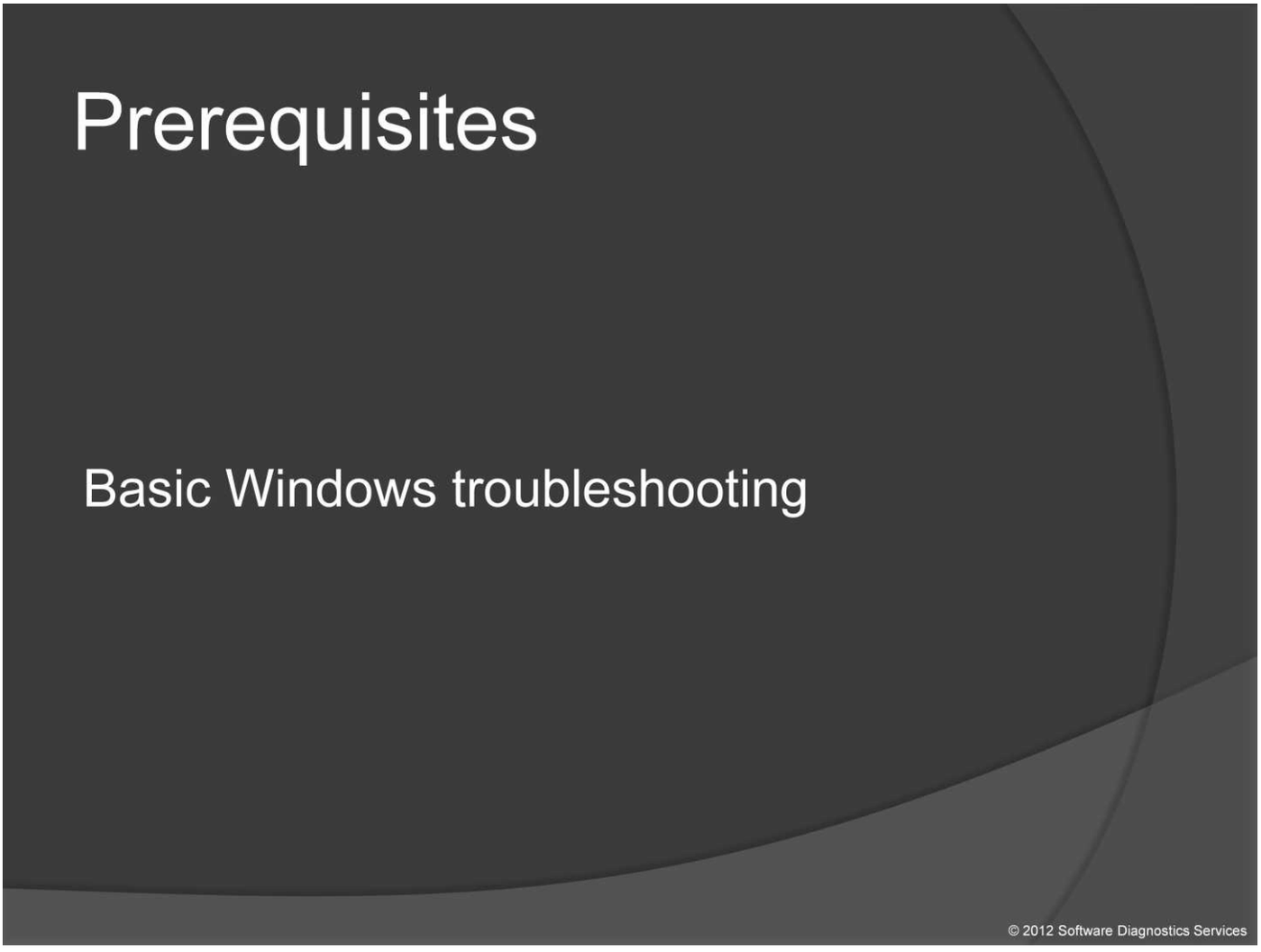
Hello Everyone, my name is Dmitry Vostokov, and I'll teach this training course. This course was originally planned for 2 training sessions 2 hours each but due to circumstances beyond my control, I split it into 4 one hour sessions. The training will also be recorded so you get recording links after each session on the next day.

What's it all About?

- General Trace Analysis Patterns
- Windows context

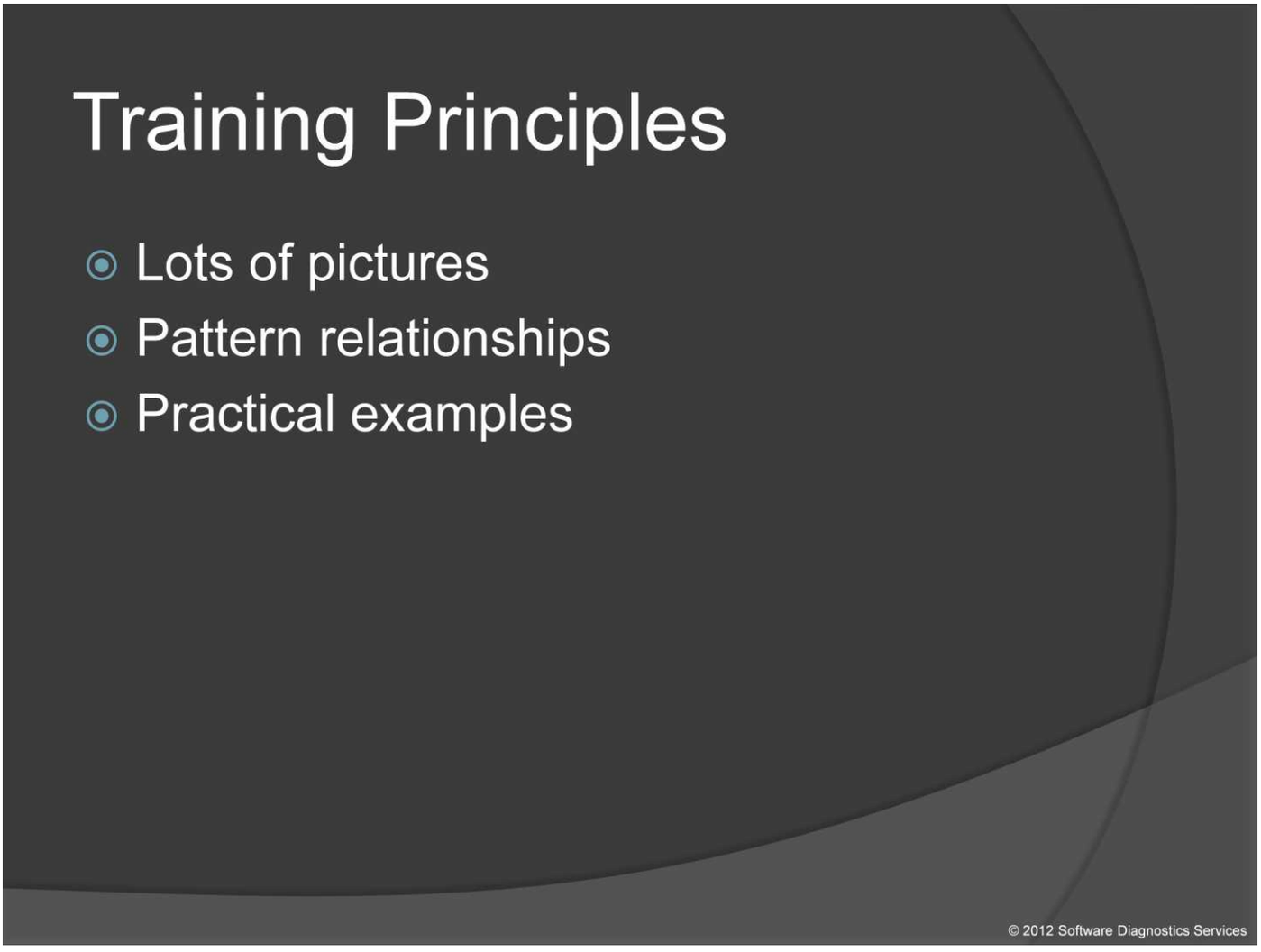


Just a few words about the need for this training. Almost 10 years ago I started doing Windows software diagnostics full time as a member of technical support and escalation team in a large global software vendor. The job required analysis of software traces and similar to that of Process Monitor format with messages from hundreds of processes and thousands of threads totalling millions of lines. Gradually I became aware that we need a similar pattern-driven system like we had for memory dump analysis. After a few patterns such as **Periodic Error**, I was stuck in devising more. At this time through my independent reading, I accidentally became acquainted with Narratology, a discipline that studies narration, narrative stories. So this became the foundation for what later I named as Software Narratology, a new approach to the study of software narrative, stories of computation. After this training, you can find a slide with resources that contain a recorded video about Software Narratology presentation that provides an overview in a wider software post-construction context. Viewing software traces as narrative helps in devising general patterns to structure trace analysis independent from OS and products. So this training teaches these patterns in Windows context, and you can apply them to your specific environment and product domain problems. I use Process Monitor as a tool because it is widely used and not tied to specific products.

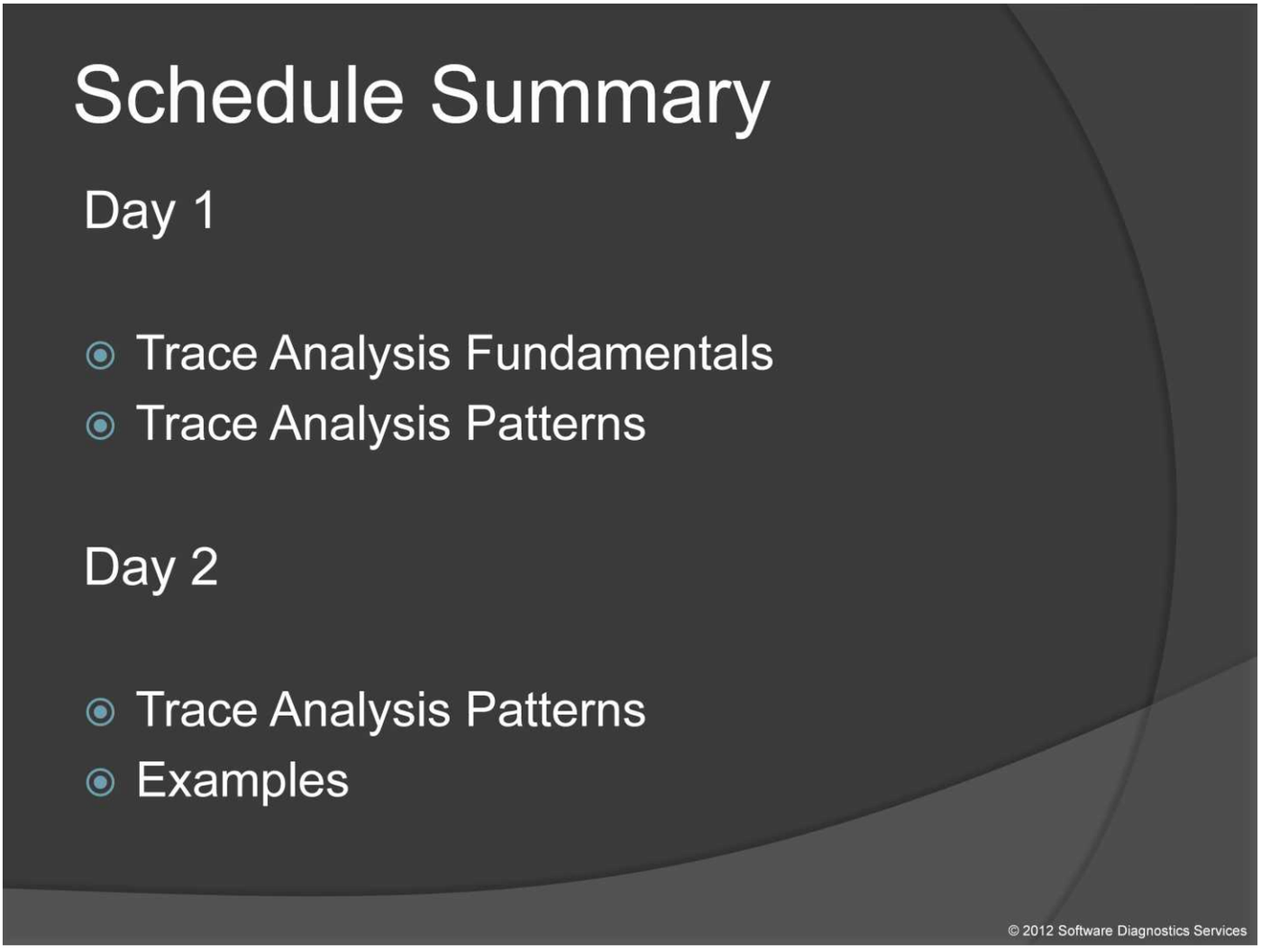


These prerequisites are hard to define. Some of you have software development experience and some not. However, one thing is certain that to get most of this training you are expected to have basic troubleshooting experience. Most troubleshooting nowadays is done by reading software traces and logs: event logs or Process Monitor logs for example. If you troubleshoot terminal services environments, then you might be familiar with the so-called Citrix CDF tracing.

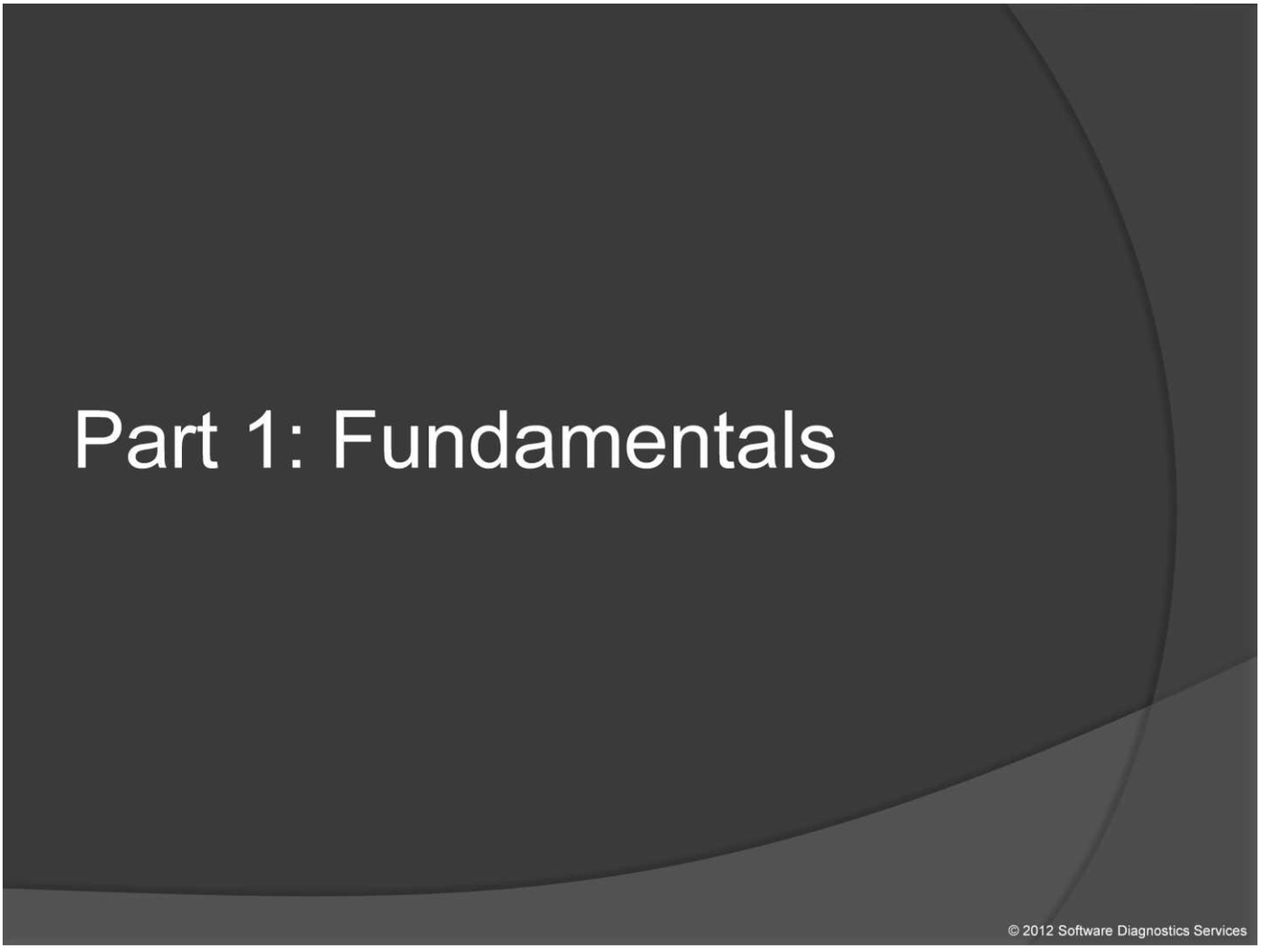
Our primary goal is to learn Windows software trace analysis in an accelerated fashion. So first we review absolutely essential fundamentals necessary for software trace analysis. Then we learn about software trace analysis patterns that were classified into several categories. We cover more than 60 patterns. Finally, we practice pattern-driven software log analysis. At the end of the training, I also give you a detailed reference where you can read more on software trace analysis using the approach advocated here. One note: this training is about software trace analysis and not about software trace implementation, internals and collection methods, tricks and tips although we might briefly cover that during the training.



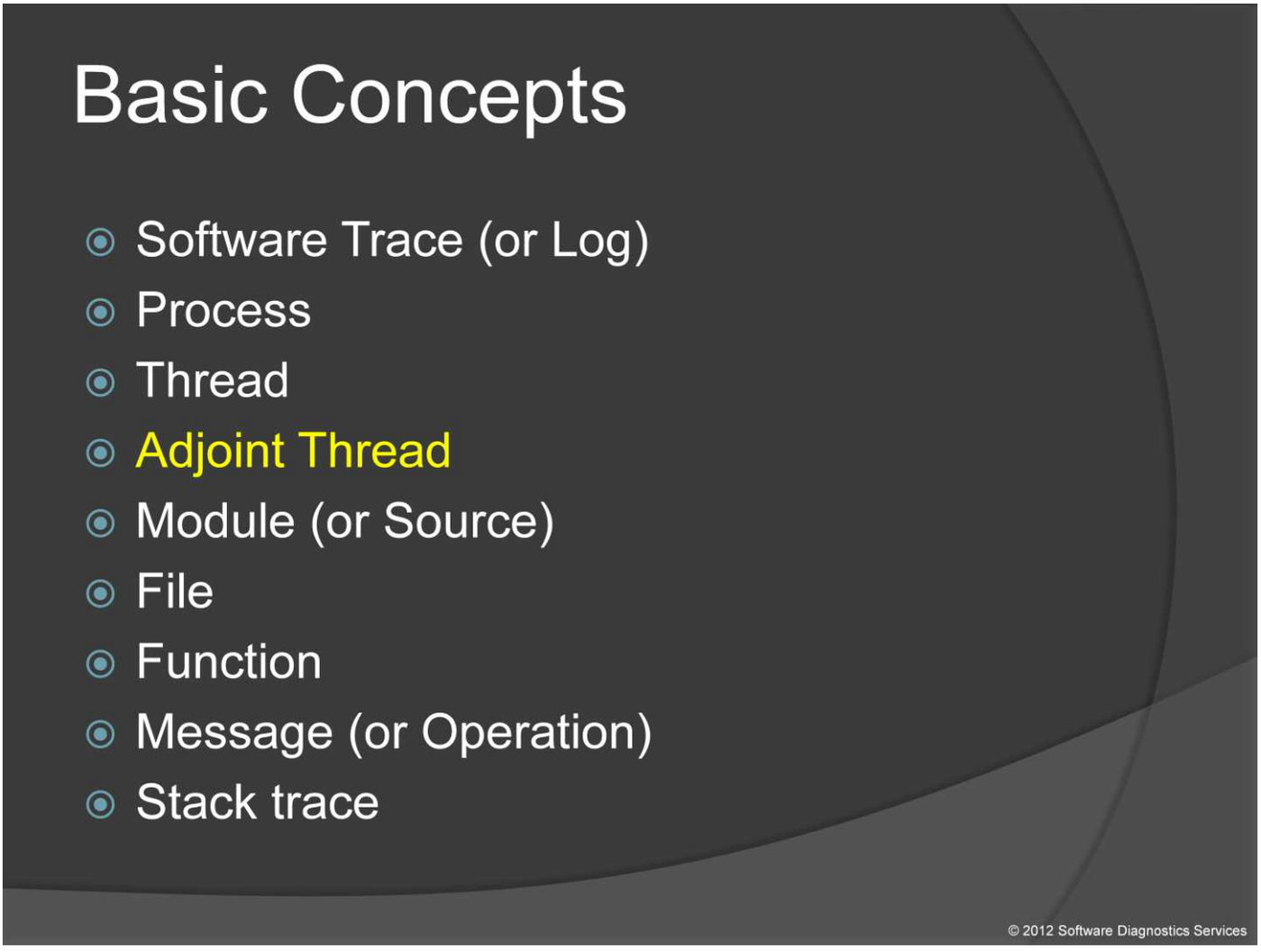
For me, there were many training formats to consider, and I decided that the best way is to concentrate on patterns first and then do hands-on exercises at the end of the training to see the novel method in action.



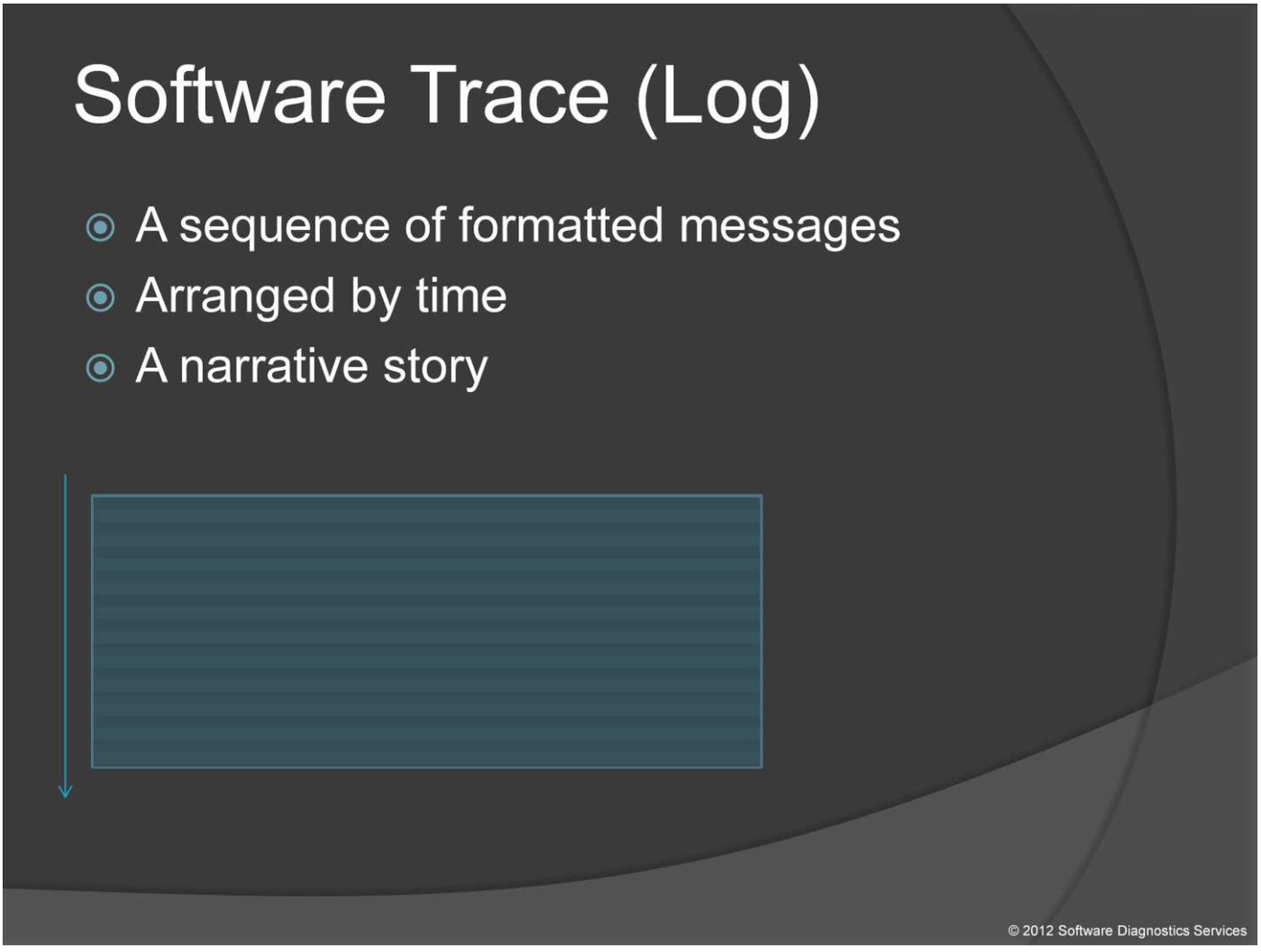
This is a roughly planned schedule. We first review essential fundamentals (mostly Windows-specific) necessary for trace and log analysis although the pattern-driven method we use is general and can also be used for other OS.



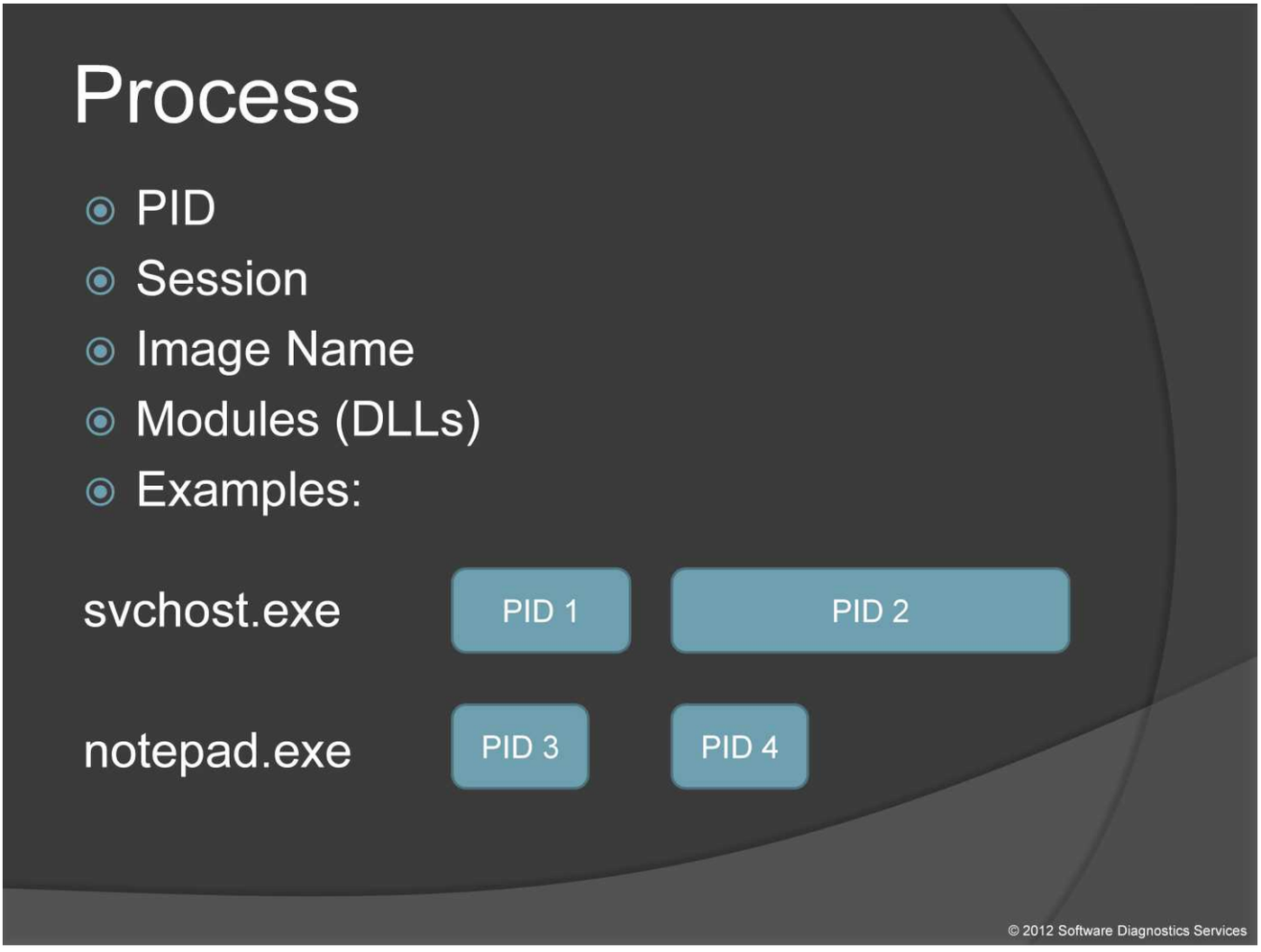
Now I present you some pictures that explain certain basic concepts related to Windows software traces and logs.



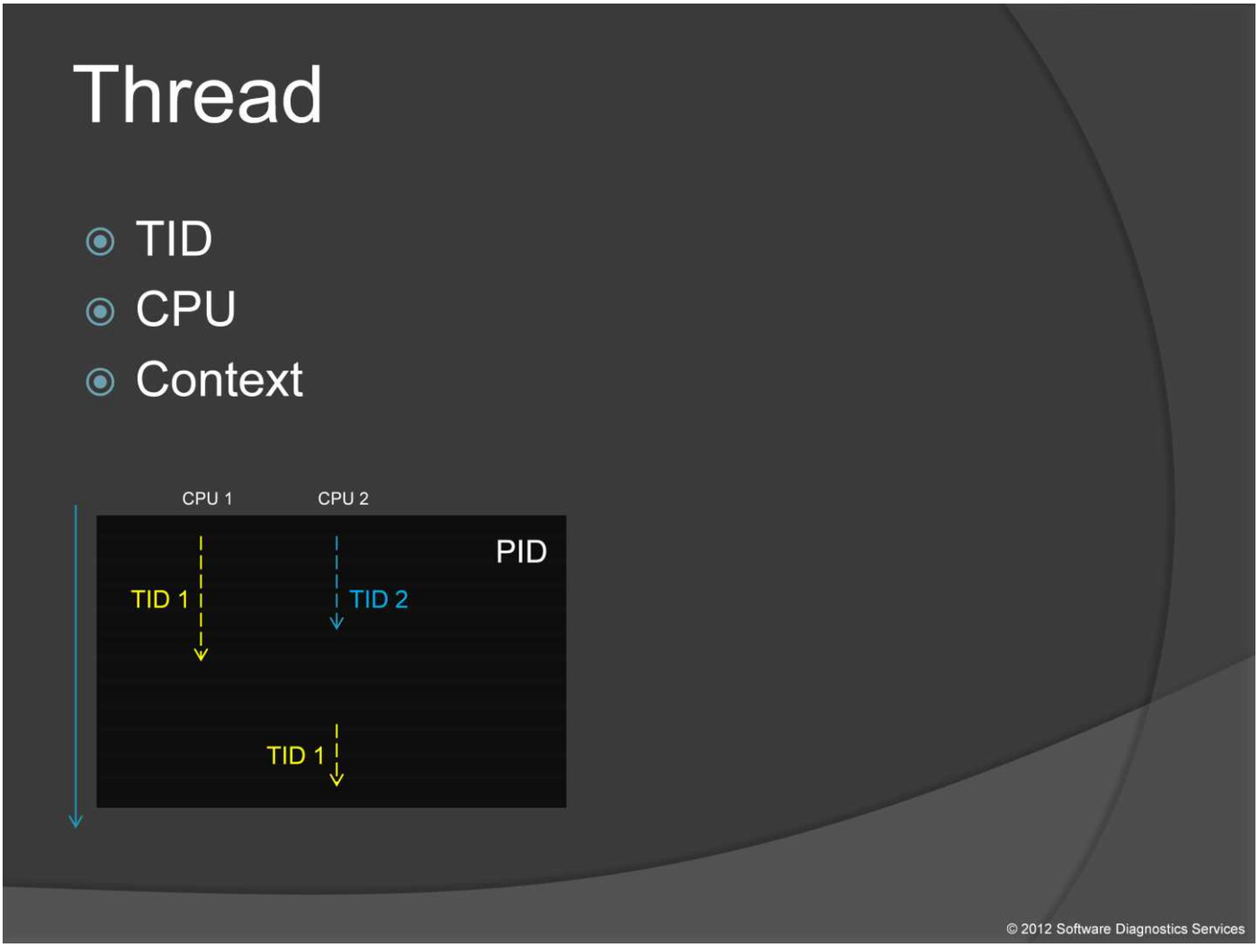
These basic concepts include processes, threads, modules, source code files, source code or API functions, and stack traces. One additional concept stays out. It is called **Adjoint Thread,** and we introduce and discuss it after we review threads. Together with threads, adjoint threads are absolutely essential for software trace analysis.



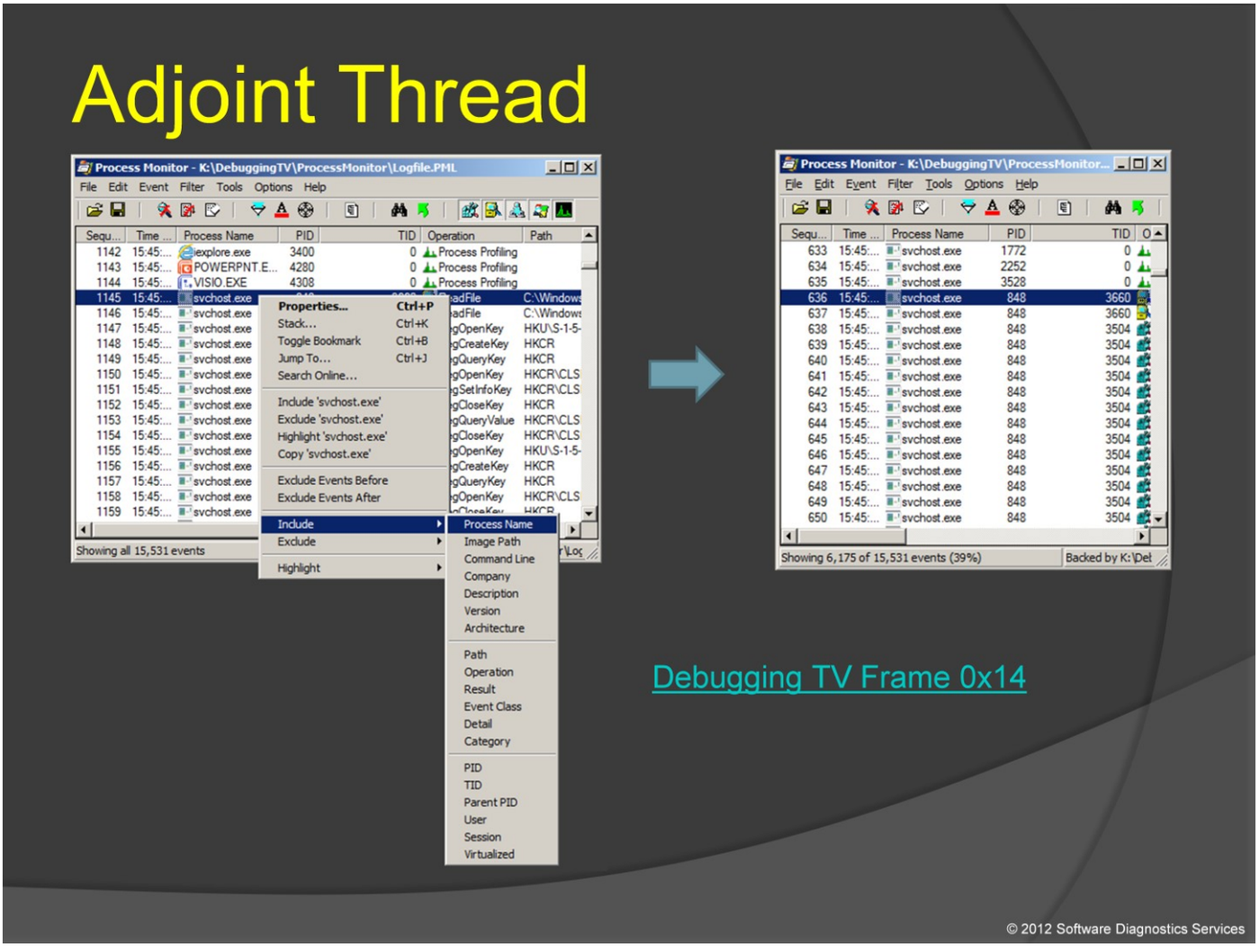
What is a software trace actually? For our purposes, it is just a sequence of formatted messages sent from running software. They are usually arranged by time and can be considered as a software narrative story. In this training, we confine ourselves to the analysis of such logs and what patterns to look for.



Windows process is a container for resources such as memory, files, and synchronization objects. Even OS kernel itself can be considered as a process itself, usually called just System. Each process has its own process identifier, PID and belongs to some user session. For example, there can be several users logged into terminal services environment. Each process has its own image name such as notepad.exe and a list of associated loaded DLL modules. An image name is also a module. It is important to remember that there can be several processes running each having the same image name, for example, 2 instances of notepad executable. The list of DLLs in both instances most of the time is identical. At the same time, it is possible that one image name covers completely different processes because on launch a process loads different modules for different purposes. Here an example is svchost executable. On a running Windows system, you can find many such svchost processes. When we analyze software logs, we can filter messages related to specific PID or image name to find any abnormal behaviour according to the expected message flow. A typical example here: after the middle of the full trace we no longer see any more messages from specific PID, not even any termination or graceful process end messages.

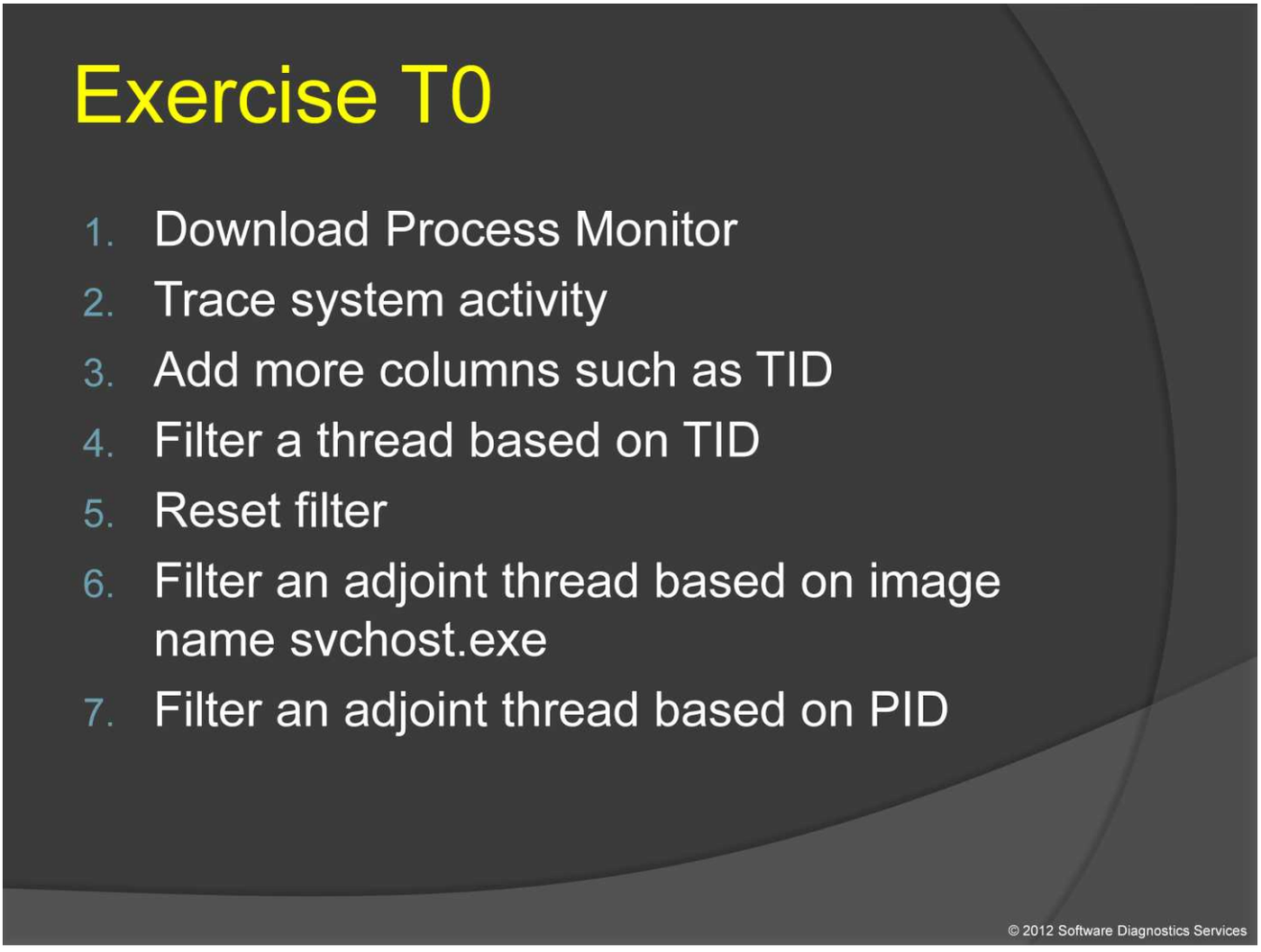


A thread is an execution unit and is owned by some process. Remember that trace messages come from some thread because we need to execute some code to emit a trace message. Each thread is executed on some CPU and, in general, can have its CPU changed during execution history. Filtering by threads, for example, allows us to find any anomalous behavior such as blocked execution activity and various execution delays. Here in the pictorial example, we see a discontinuity for TID 2 and a delay in TID 1.

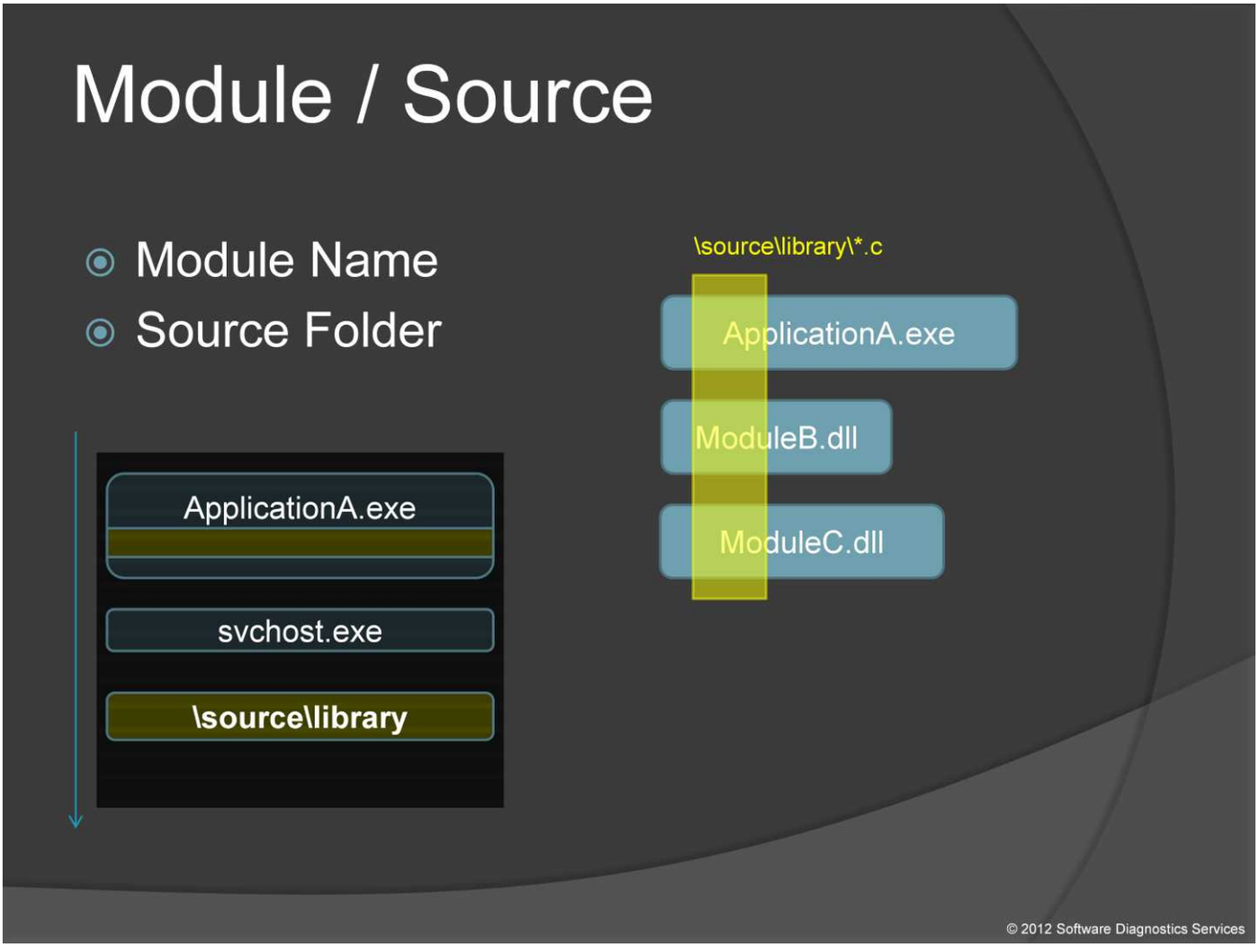


If a thread is a linear ordered flow of activities associated with particular TID as seen from trace message perspective through time we can also extend this flow concept and consider a linear flow of activities associated with some other parameter such as PID, CPU or message text. Such messages will have different TIDs associated with them but some chosen constant parameter or column value in trace viewing tool. The name **adjoint** comes from the fact that in threads of activity TID stays the same but other message attributes vary but in adjoint threads we have the opposite. Please also refer to Debugging TV Frame 0x14 presentation that also contains links to additional articles and examples. In Process Monitor and Excel we use exclusive and inclusive filtering to form adjoint threads. By applying complex filtering criteria we get adjoint threads from other adjoint threads, for example, an adjoint thread with specific PID and file activity formed after an inspection of an adjoint thread with the same image name, such as svchost.exe.

Debugging TV Frame 0x14:
http://www.debugging.tv/Frames/0x14/DebuggingTV_Frame_0x14.pdf



Now I do a preliminary exercise to show adjoint threading in Process Monitor. It is very simple so you can repeat it in your environment.

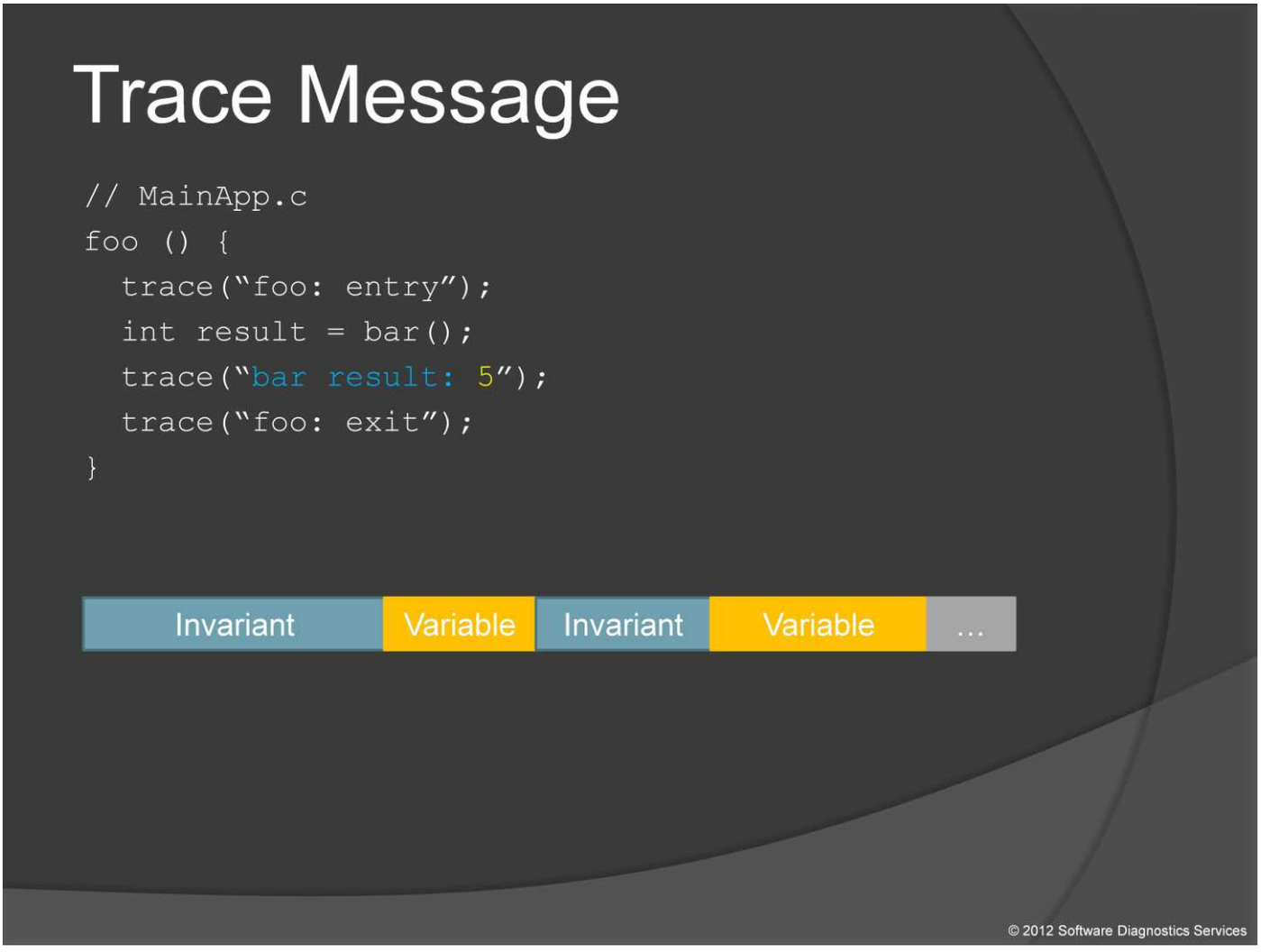


Trace messages come from a thread that belongs to a PID but the code to emit them resides in source code files. Some source code files can also be reused such as static library code and included in different modules. Such DLL modules can also be loaded into different processes, for example, hooking modules. Therefore, Source or Module (in a simpler) case is another grouping of messages based on subsystem and functional division that may include several source code files. By module or source filtering we can see subsystem activities.

File and Function

```
// MainApp.c
foo () {
  trace("foo: entry");
  // do stuff
  trace("foo: exit");
}
```

Source code consists of files, and each file has some functions inside that do actual tracing. With file or function filtering we can see the flow of certain functionality that is more fine-grained than source or module adjoint thread of activity.



Formatted trace message is just a sentence with some invariant and variable parts. In fact, it is possible to trace invariant and variable parts, the so-called **Message Invariant** and **Data Flow** patterns that we consider later.

Stack Trace

```
// MainApp.c
main() {
  trace("start");
  foo();
}

foo() {
  trace("foo: entry");
  bar();
}

bar() {
  trace("bar: entry");
  // do stuff
}
```

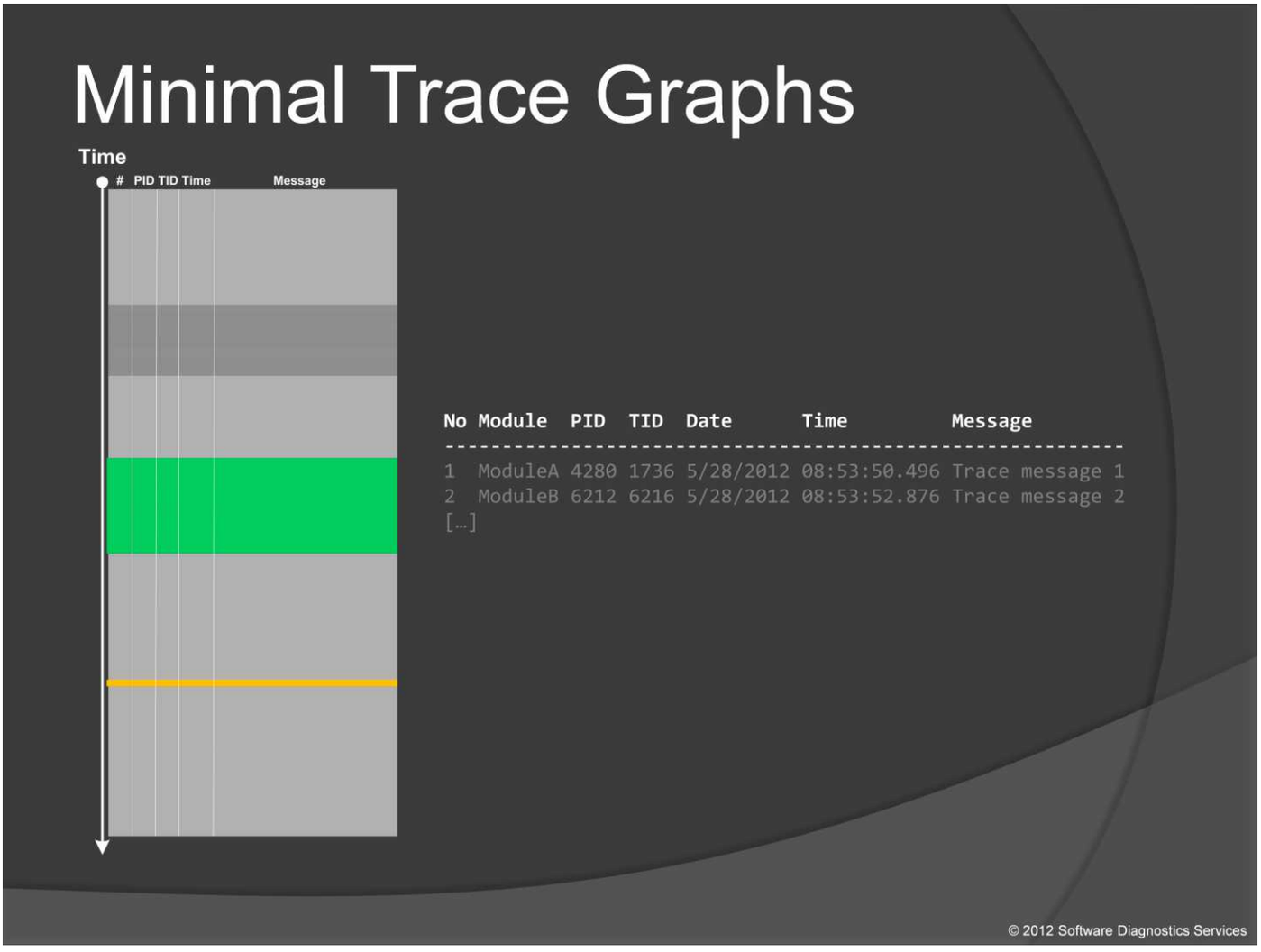
Because each trace message originated from some function in the source code, it has an associated stack trace similar to live debugging scenario where we put a breakpoint at a trace message code location. Also please note that stack traces (or backtraces) are read from bottom to top as in a debugger.

A few words about software trace recording tools. In this slide, I only put my own favorite tools. These tools record the sequence of events in a linear order. We do not consider tools that monitor some parameters in the system and provide aggregate statistical snapshots or graphs such as Performance Monitor or its newest incarnations. This is because the underlying tracing is still event based and statistical sampling can be considered as a trace event itself such as **Counter Value** pattern we consider later. Here I also provided links to tools I personally and frequently use. MessageHistory records GUI events similar to Spy++ from Microsoft Visual Studio. CDFControl is used in Citrix terminal services environments.

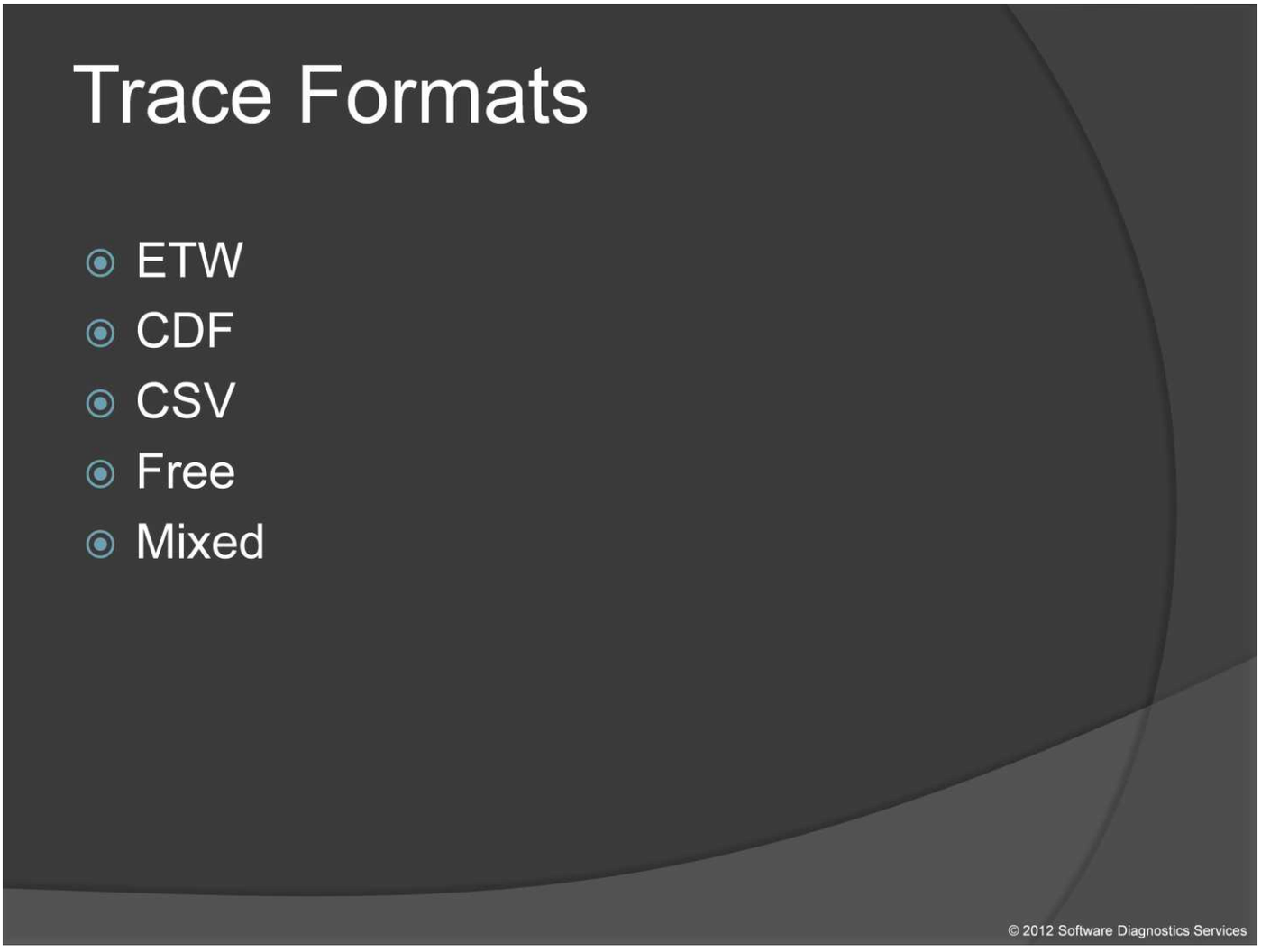
Process Monitor: http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx
MessageHistory: http://support.citrix.com/article/CTX111068
CDFControl: http://support.citrix.com/article/CTX111961

These are my favorite analysis tools, and we would use two of them in exercises. They are Process Monitor and Microsoft Excel.

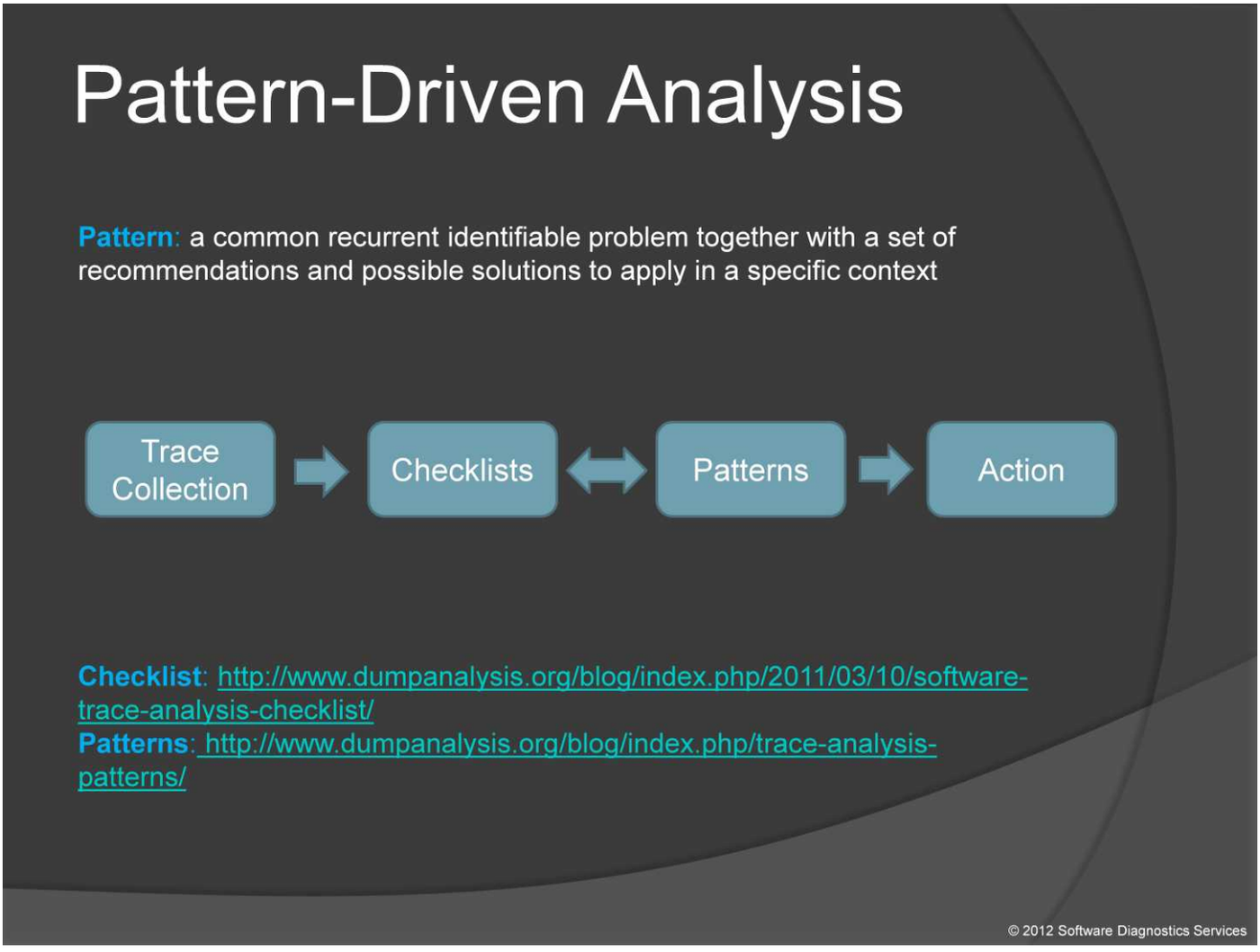
CDFAnalyzer: http://support.citrix.com/article/CTX122741



In order to illustrate trace analysis patterns graphically, we use the simplified abstract pictorial representation of a typical Windows software trace. It has all essential features such as message number, time, PID, TID, and message text itself. Sometimes, for illustration purposes, I also provide trace fragments in a textual form where I include or exclude certain message attributes (or columns) such as message source or module, time or date.



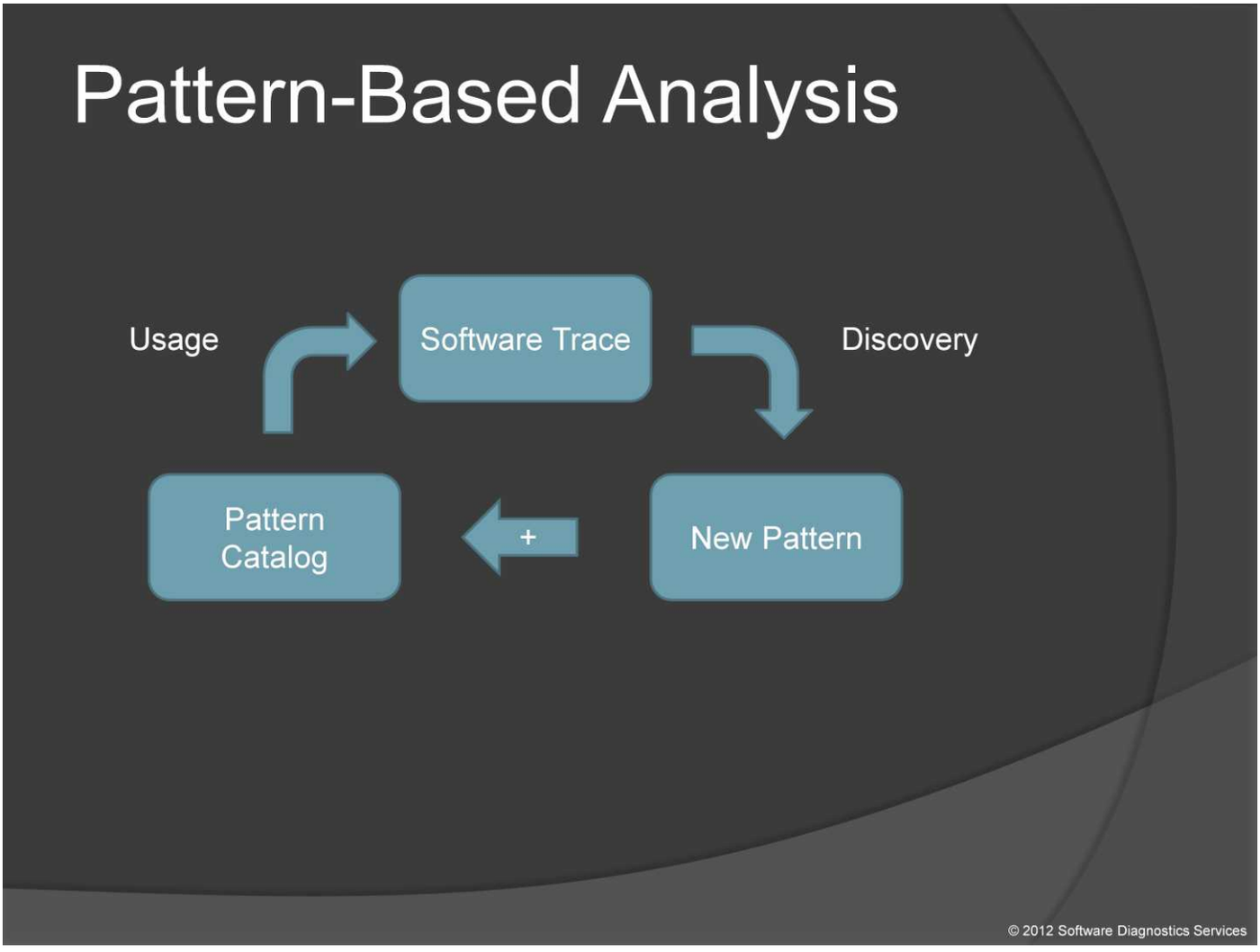
By formats, we consider already formatted traces ready for human inspection. The first two formats are formats we use for illustration of patterns. CSV format is the normal format of values separated by comma, space or tab ready for import into some table manipulation software such as Microsoft Excel. The rest of formats are really product dependent or designed for some specialized purpose. By **Mixed** format, we mean state snapshots aggregated with change history of some parameters.



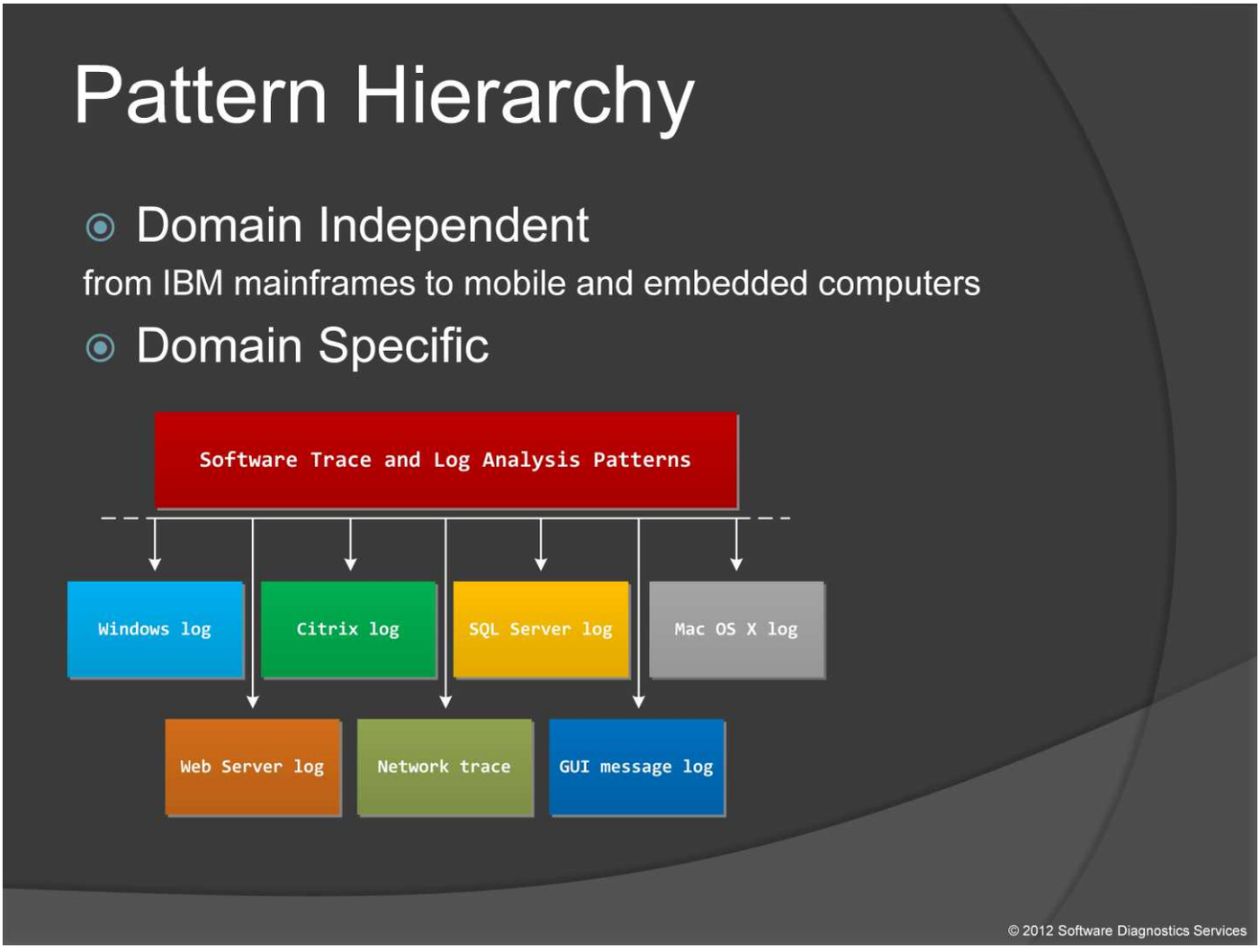
A few words about logs, checklists, and patterns. Software trace analysis is usually an analysis of a text for the presence of patterns. Here checklists can be very useful. One such checklist is provided as a link.

Checklist: Memory Dump Analysis Anthology, Volume 6, page 297
http://www.dumpanalysis.org/blog/index.php/2011/03/10/software-trace-analysis-checklist/

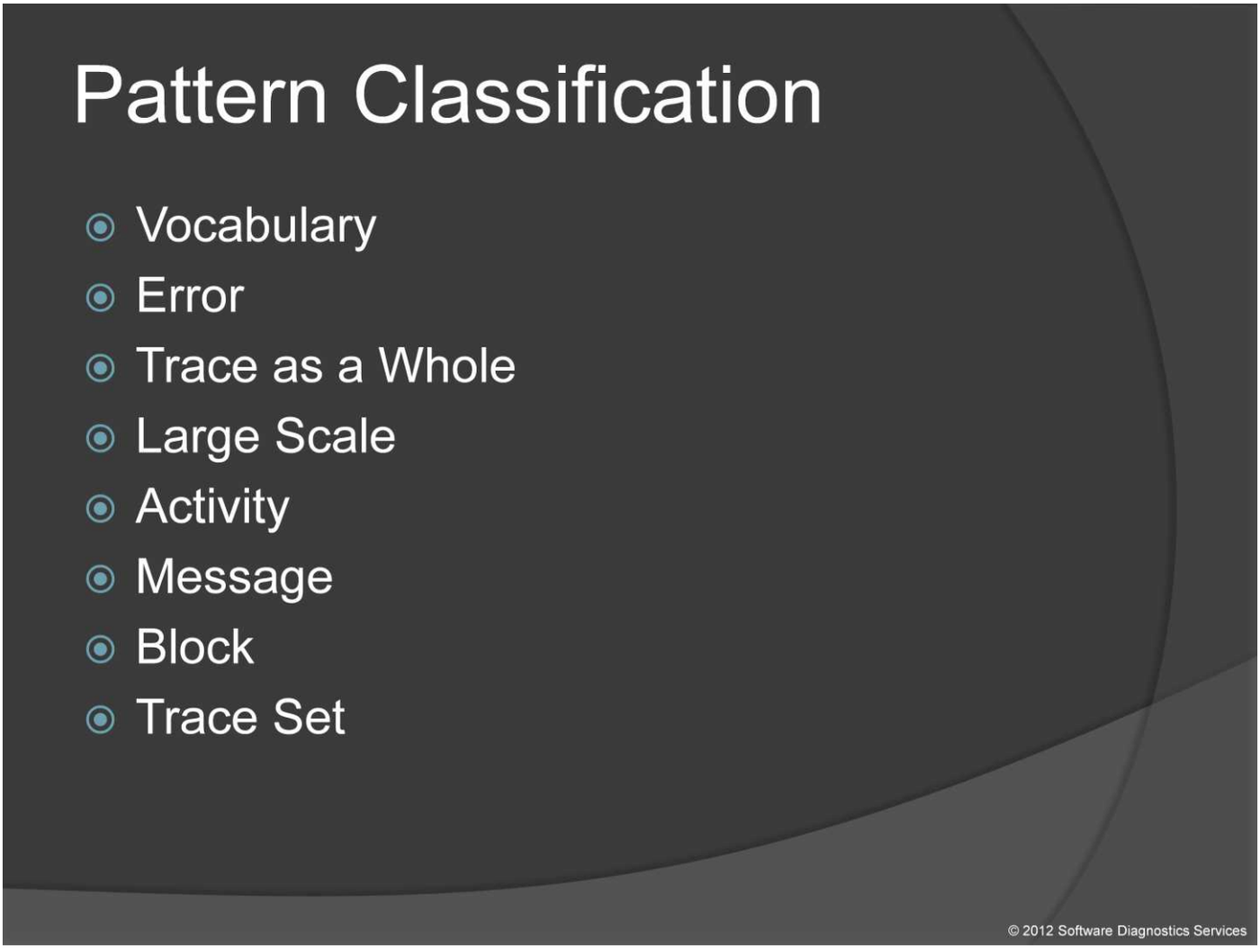
Patterns: Memory Dump Analysis Anthology volumes
http://www.dumpanalysis.org/blog/index.php/trace-analysis-patterns/



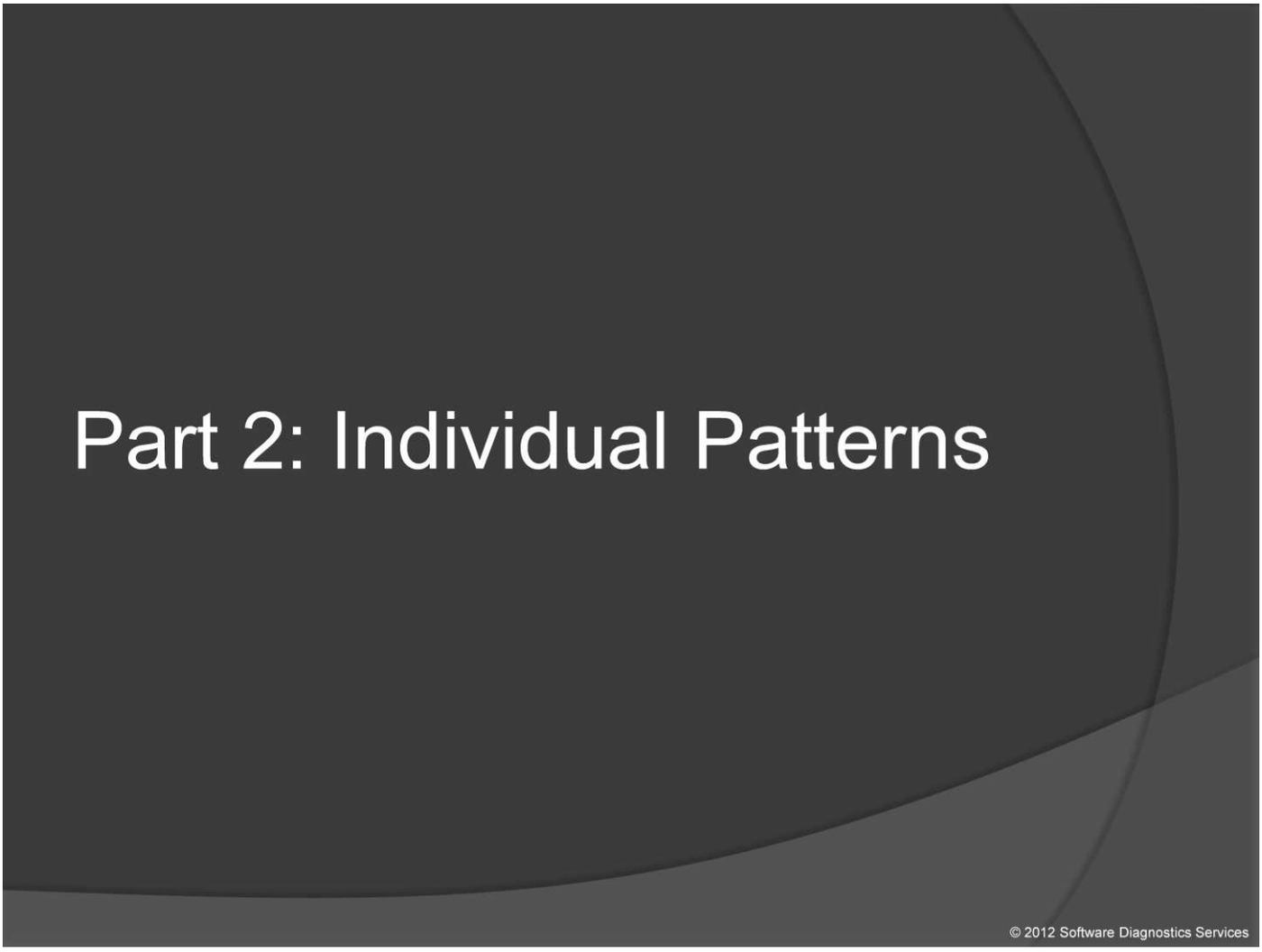
Pattern catalogs are rarely fixed. New patterns are constantly discerned especially at a domain-specific level such as product- and platform-specific patterns. For example, today I found yet another missing pattern and currently adding it to trace analysis pattern catalog.



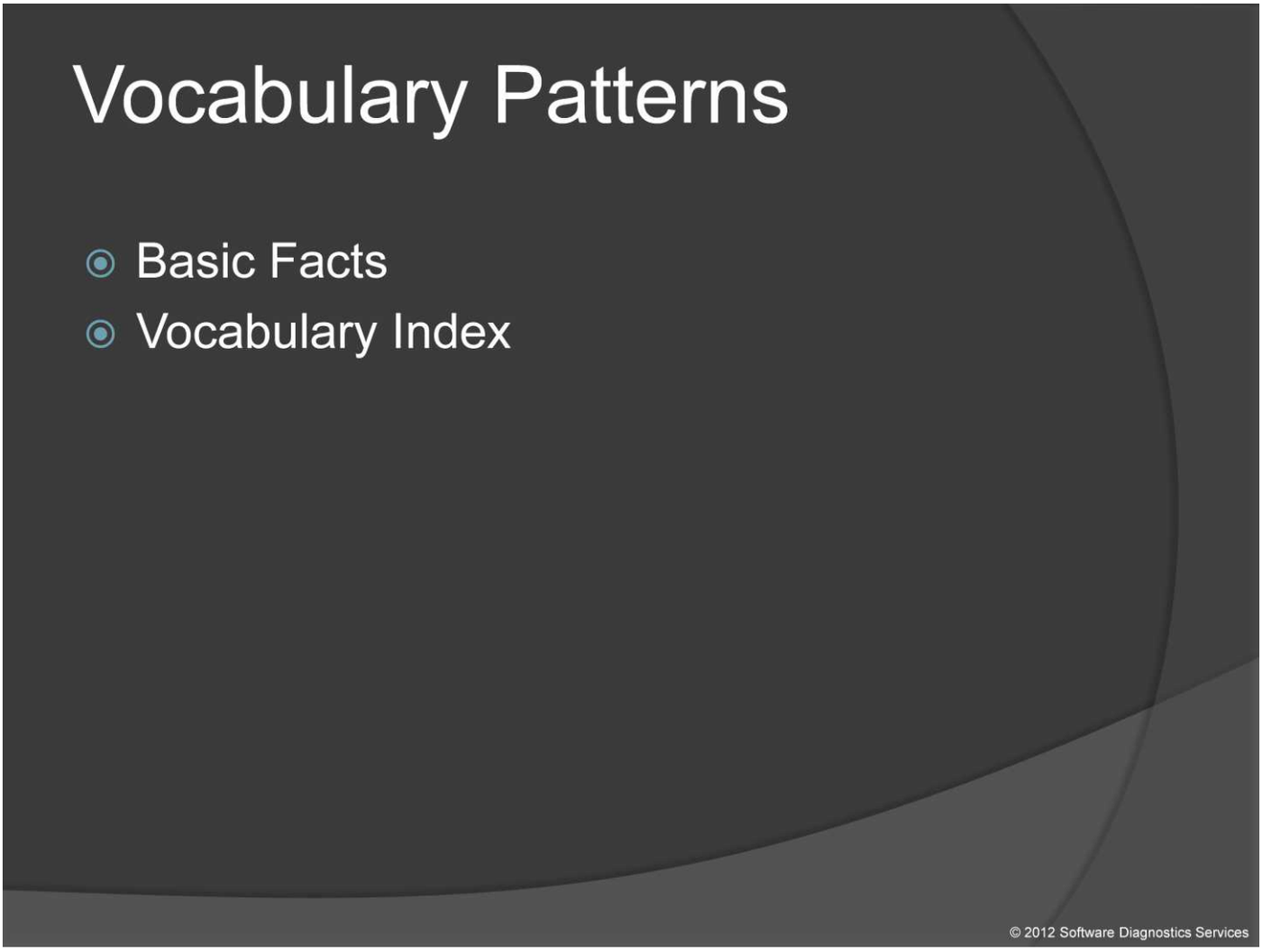
I repeat that these patterns are general, domain-independent and can be applied for analysis of different software logs even on IBM mainframes. The next level is domain specific patterns such as product or OS error message, etc. Of course, to get most of the patterns, you need to know your specific problem domain. An analogy would be a story about some country where you don't know anything about its history, meaning of events and places, but you still can recognize the general story structure like a person was waiting for two hours to get an audience.



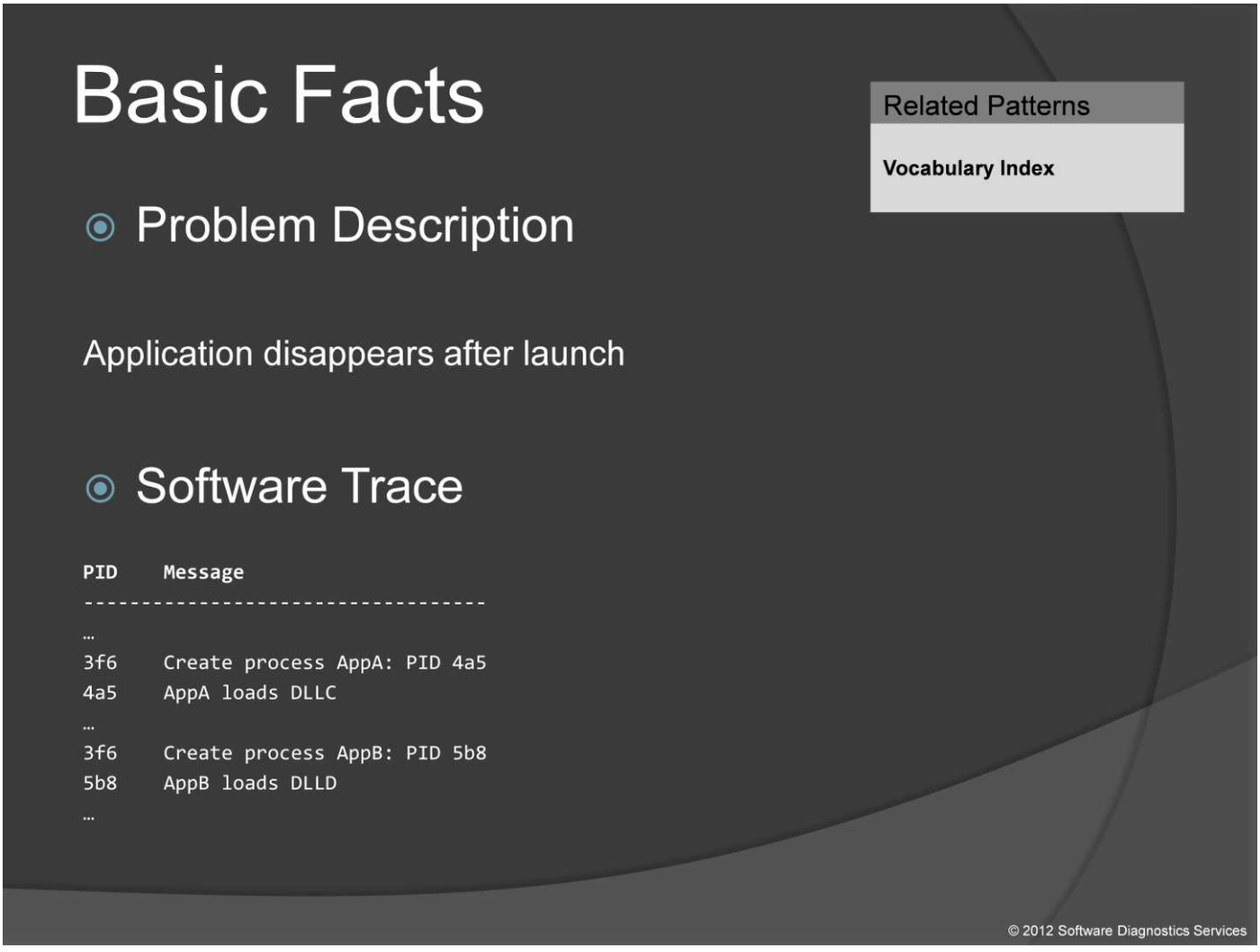
Recently I classified patterns which are now numbered more than 60 at the time of this writing into several categories. Vocabulary category consists of patterns related to problem description. Error category covers general error distribution patterns. We also consider traces as wholes, their large scale structure, activity patterns, patterns related to individual trace message structure, patterns related to collections of messages (the so-called blocks) and finally patterns related to several traces and logs as a collection of artifacts from software incident.



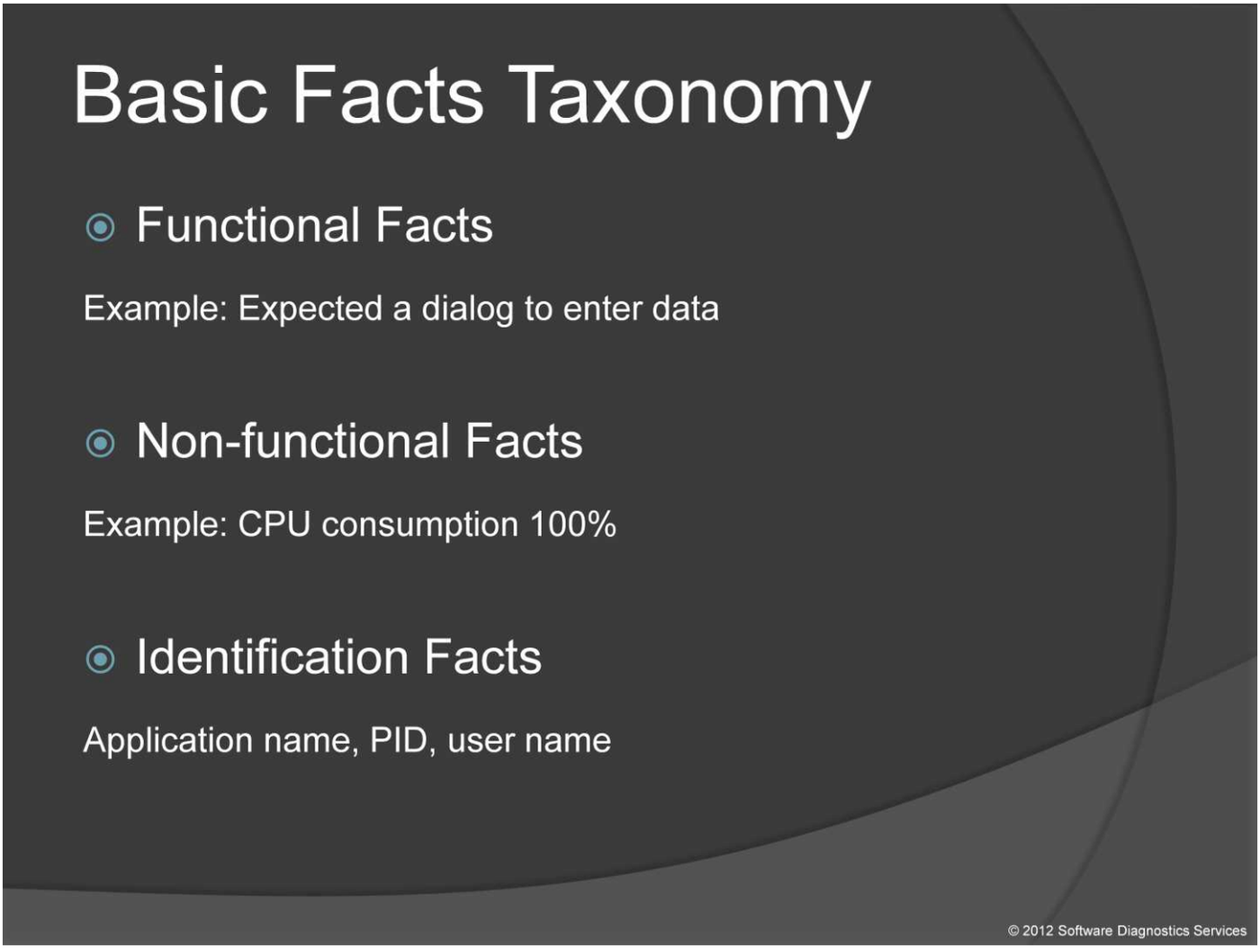
Now we come to **Part 2** which is about individual patterns where I also provide you pictures and explanations including related patterns as they form some kind of analysis pattern network. You can later use these slides as a reference.



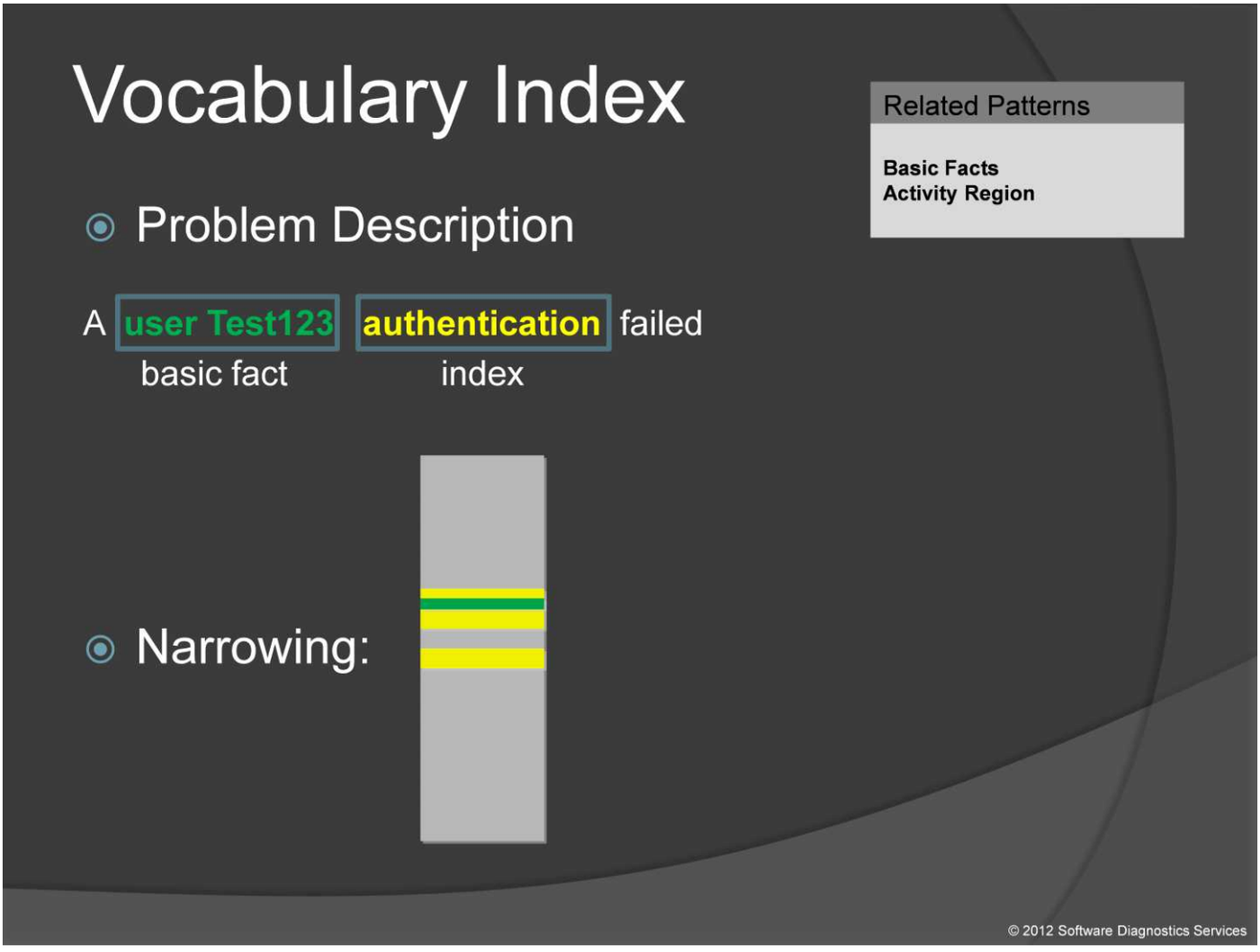
The first block of patterns we cover are vocabulary patterns. These are patterns related to problem description from a user point of view.



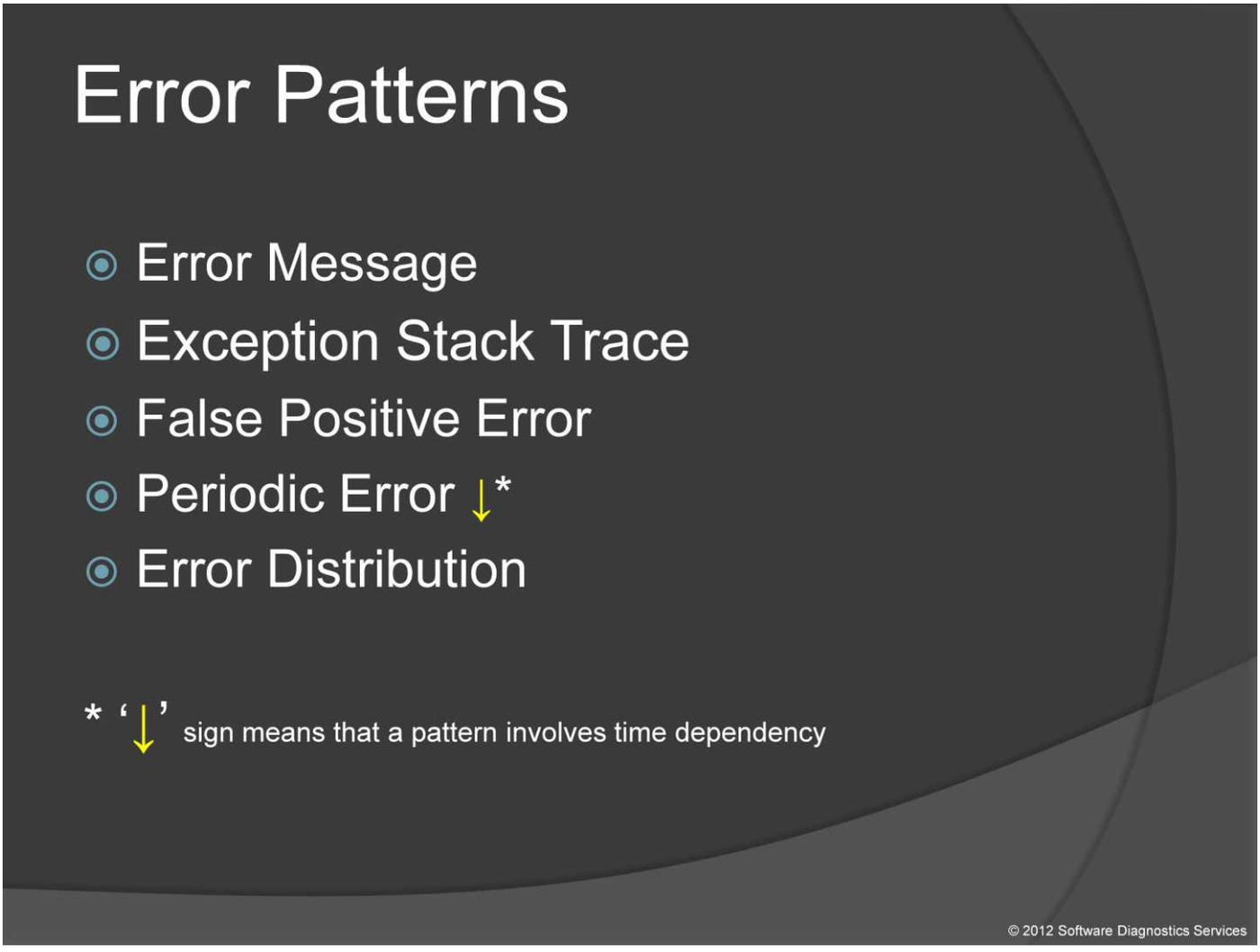
A typical trace is a detailed software narrative that may include lots of irrelevant information with useful messages like needles in a haystack. However, it is usually accompanied by a problem description that lists essential facts. Or we hope it lists essential facts. Therefore, the first task of any trace analysis is to check the presence of **Basic Facts** (or it is usually called Supporting Materials) in the trace. If they are not visible or do not correspond, then the trace was possibly not recorded during the problem or was taken from a different computer or under different conditions. Here is a negative example: An application disappears after launch. We look at the trace and find several application launches. Which one to look for? What if they all disappear suddenly because the tracing ends before they finish? We see how vital are basic facts. Is it still possible to diagnose which one disappeared? Sometimes it is possible, for example, here it might be a discontinuity, disappearance of all messages related to that application PID till the end of the trace compared with other applications messages that populate the trace till the end unless the application finishes correctly. Here we can use the so-called **Adjoint Thread** pattern to filter trace messages for specific PID and compare their **Partition** and **Characteristic Message Blocks** (other patterns) and search for error patterns and any **Guest Components** (the latter is suddenly appearing trace messages from a loaded module that we never see loaded in normal working scenarios).



Software is written according to some requirements such as functional ones such as what it is supposed to do and what users are expected to see and get during the interaction, and non-functional ones such as a requirement that software execution should not exceed resource usage. Deviations from such requirements map to facts in software problem descriptions. The 3rd type of facts is problem instance specific such as a user name or PID that help to identify specific software interaction in software execution story. Suppose you analyze a user session problem from terminal services environment. You get a software trace with a hundred of users, and you need a username to search for and problem process PID to filter its threads.

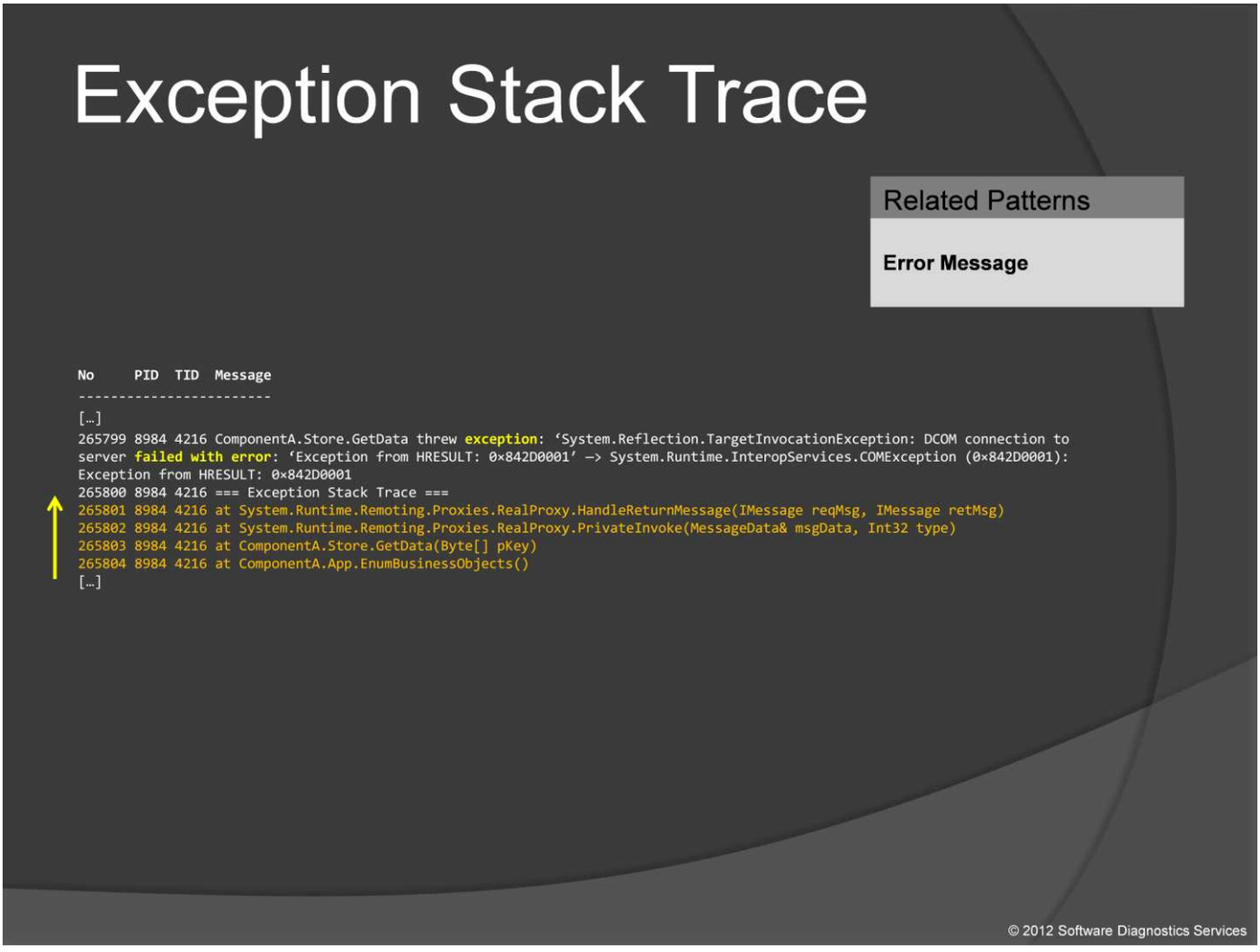


What will we do confronted with 10 million trace messages recorded during an hour with an average trace statement current of 3,000 msg/s from dozens of modules and having a short problem description even if it has some basic facts? One solution is to try to search for a specific vocabulary relevant to the problem description, for example, if a problem is authentication failure, then we might try to search for words related to authentication. We call such list of words drawn from troubleshooting domain vocabulary a **Vocabulary Index** by analogy with book index. In our trace example, the search for "authentication" jumps straight to a smaller **Activity Region** (another pattern) of authentication modules starting from the message number #1,380,010 and the last "password" occurrence is in the message #3,380,490 and that narrows initial analysis region to just 500 messages.

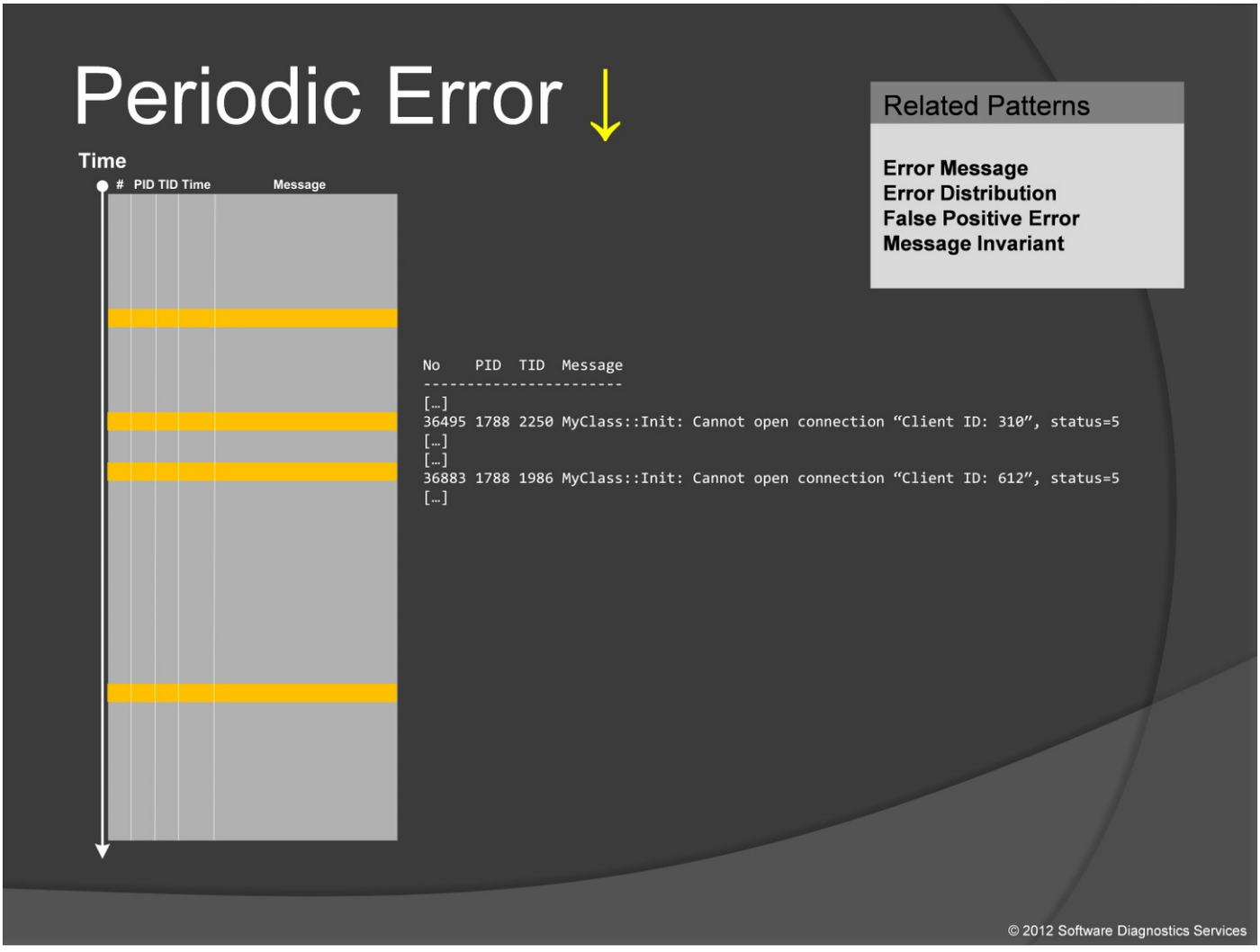


The next block of patterns we cover are error patterns. These patterns are related to error and failure messages either explicitly stating that there is an error or doing that indirectly via error code, abnormal function return value or NT status values in failure range.

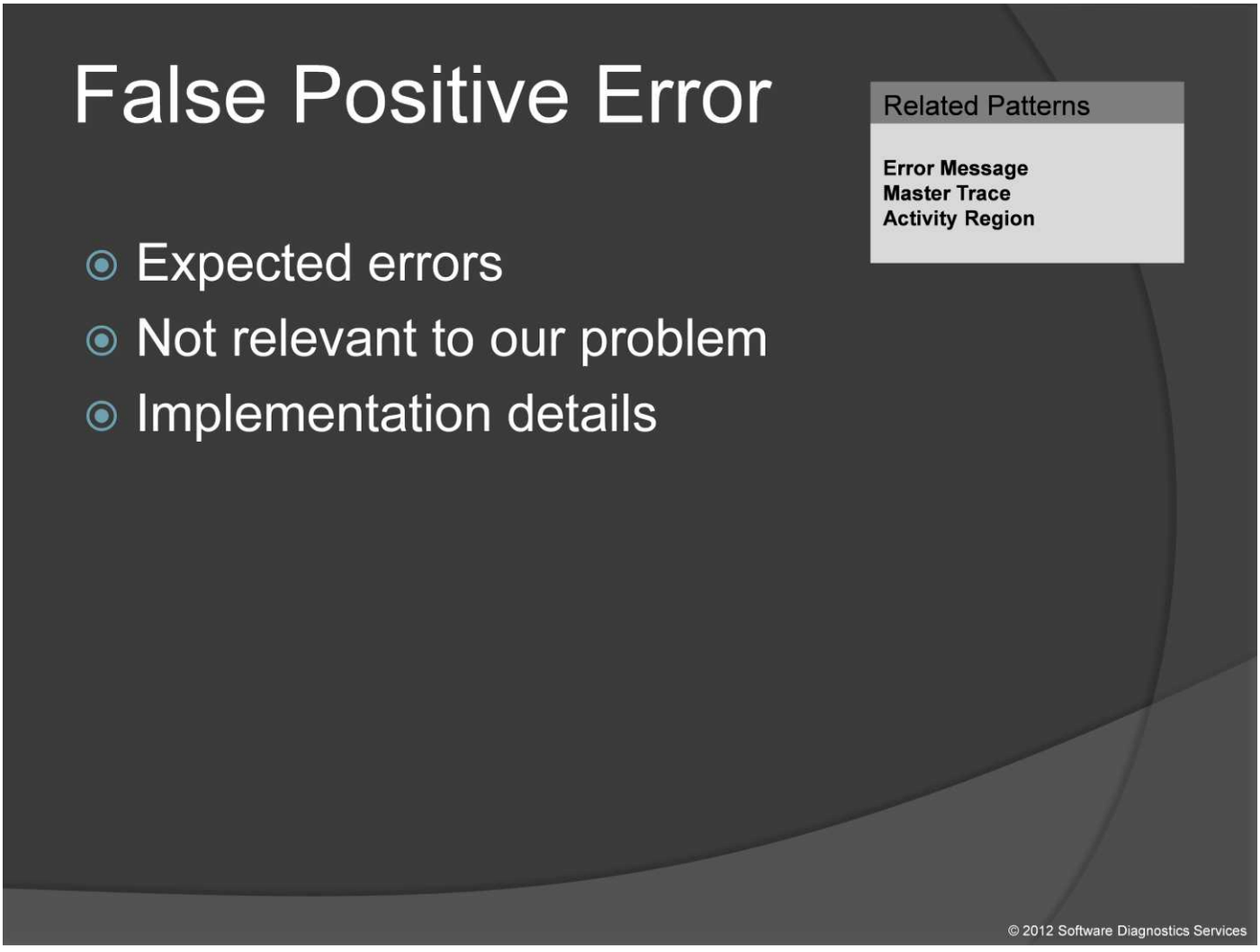
This error message can be reported either explicitly (”operation failed”) or implicitly as an operation status value such as 0xC00000XX or a value different from a normal result such as when we get 5 instead of 0. It is considered a good implementation practice to indicate in trace messages specifically whether a number value was supplied for information only and should be ignored by technical support or software maintenance engineer. Some error messages may contain information that is not relevant to the current software incident, the so-called **False Positive Errors** (another pattern we discuss later). Some tracing architectures and tools include message information category for errors and warnings, such as Citrix CDF (ETW-based) where you can filter by error category to get **Adjoint Thread** of errors. Note, that the association of a trace statement with an error category is left at the discretion of a software engineer writing code, and you can have error messages that do not belong to error category. Errors you find or are interested in can be repeated throughout the log (**Repeated Error** pattern), they can also be unevenly distributed throughout the trace or log (**Error Distribution** pattern) and if filtered by their data value can show this error **Data Flow** across threads, processes, and modules.



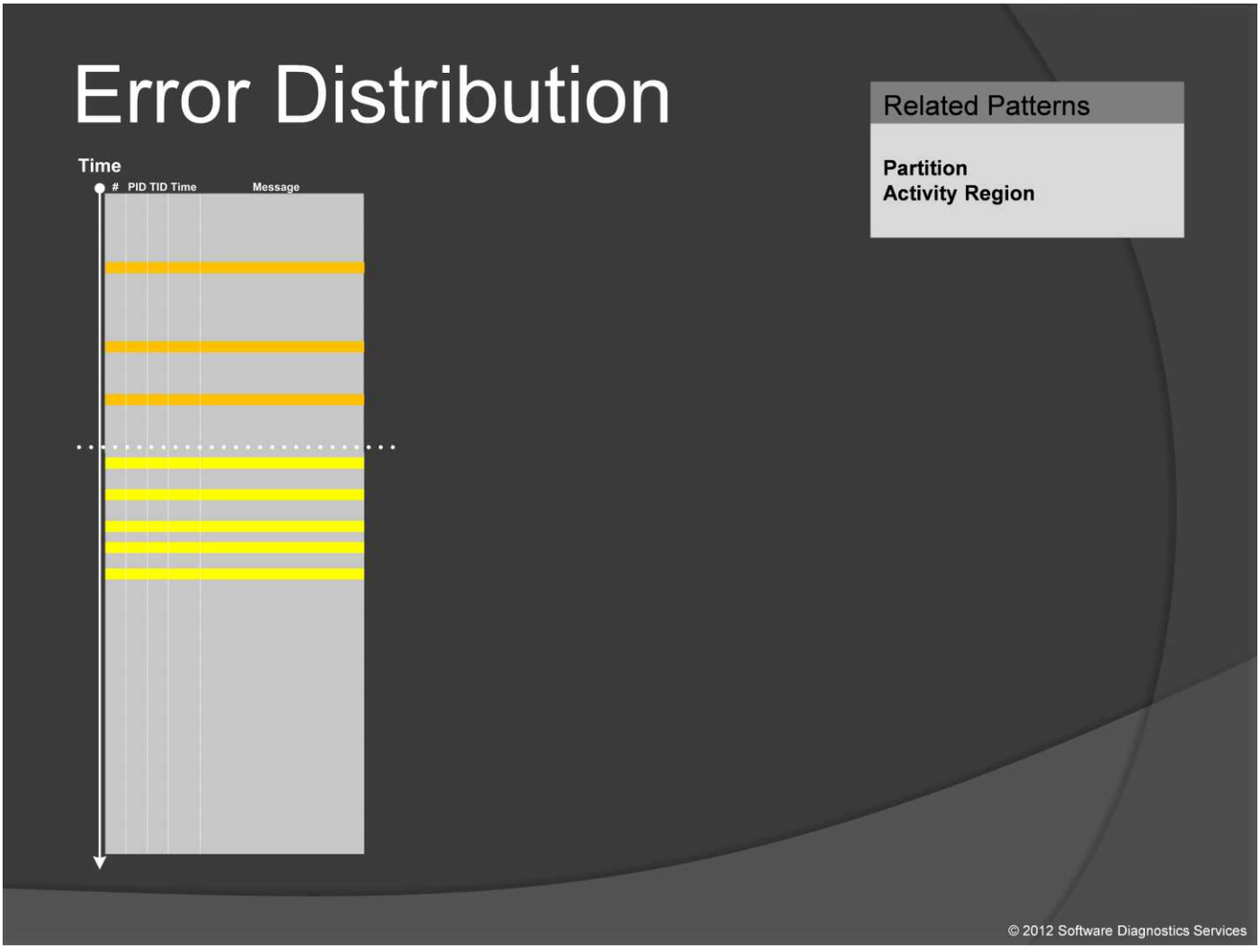
Often analysis of software traces starts with searching for short textual patterns, like a failure or an exception code or simply the word "exception". And indeed, some software components are able to record their own exceptions or exceptions that were propagated to them including full stack traces. This is all common in .NET and Java environments. The slide shows a typical example based on real software traces. These stack traces are similar to stack traces we see in memory dumps. In the embedded stack trace we see that App object was trying to enumerate business objects and asked Store object to get some data, and the latter object was probably trying to communicate with the real data store via DCOM.



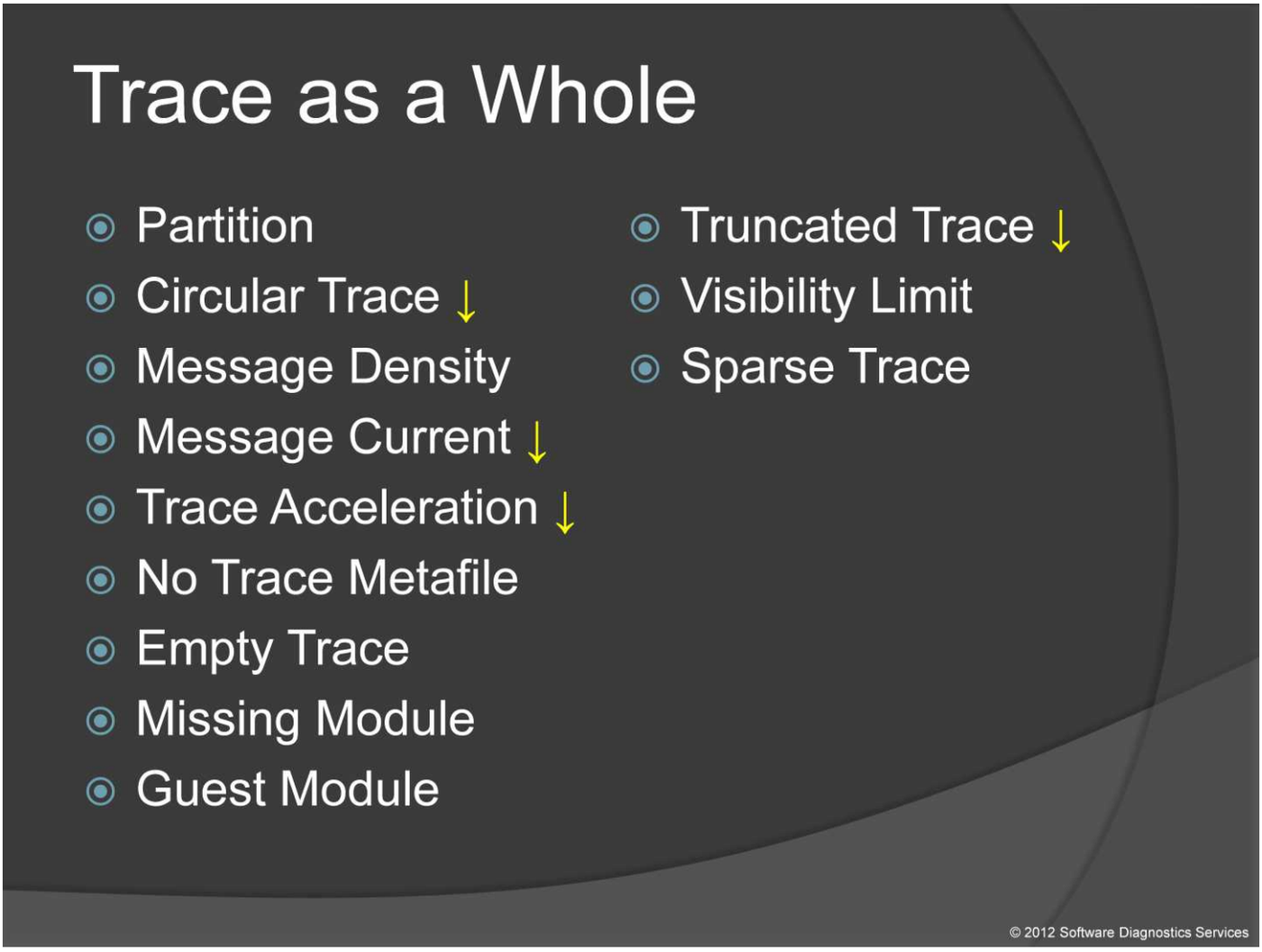
This is an **Error Message** that is observed periodically many times in a trace or log file. In fact, it may not be exactly the same trace message. It may differ in some reported values having the same **Message Invariant** structure (another message level pattern).



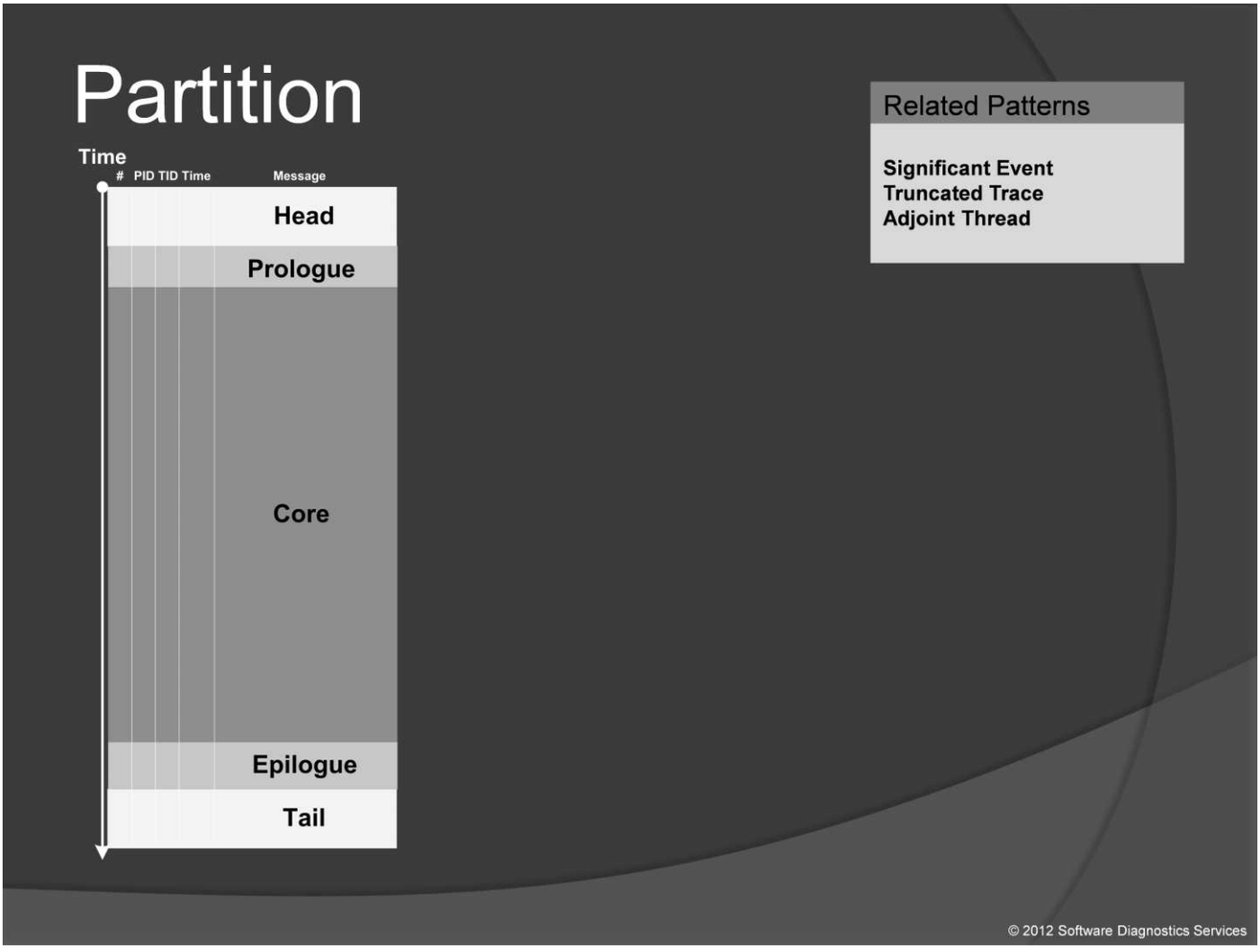
Software might report errors that are false positive, as not relevant to the reported problem or expected as a part of implementation detail, for example, when a function returns an error to indicate that a bigger buffer is required or to estimate its size for a subsequent call. To know if some error message is false positive or not we compare the same **Activity Regions** in a trace from problem scenario to a trace from the normal working scenario or the so-called **Master Trace**.



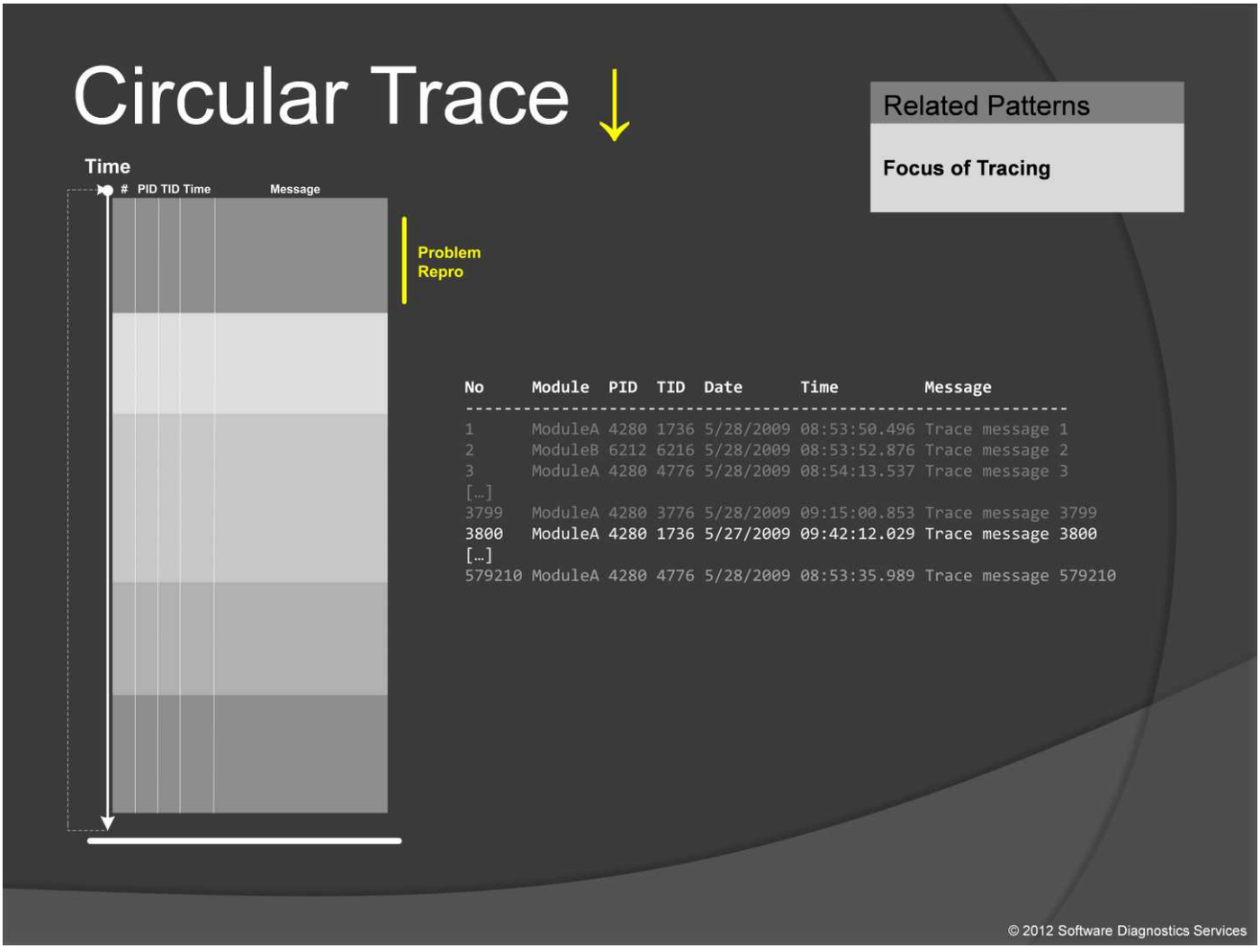
Sometimes we need to pay attention to **Error Distribution**, for example, the distribution of the same error across a software log space or different error messages in different parts of the same software log or trace (providing effective **Partition** of the trace into error **Activity Regions**).



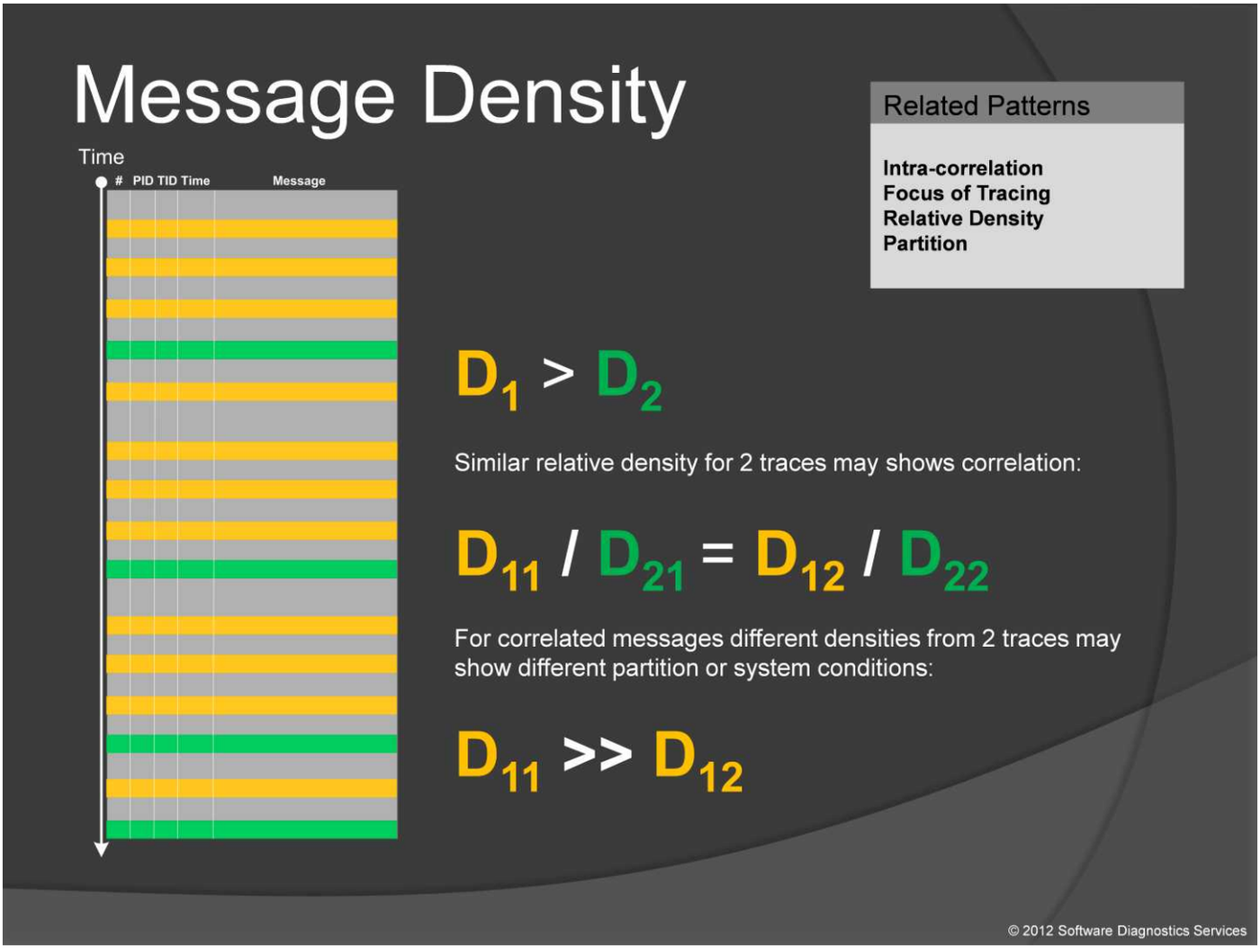
The third block of patterns we cover are patterns that are related to software trace or log as a whole. Here we ignore trace message contents and treat all messages statistically.



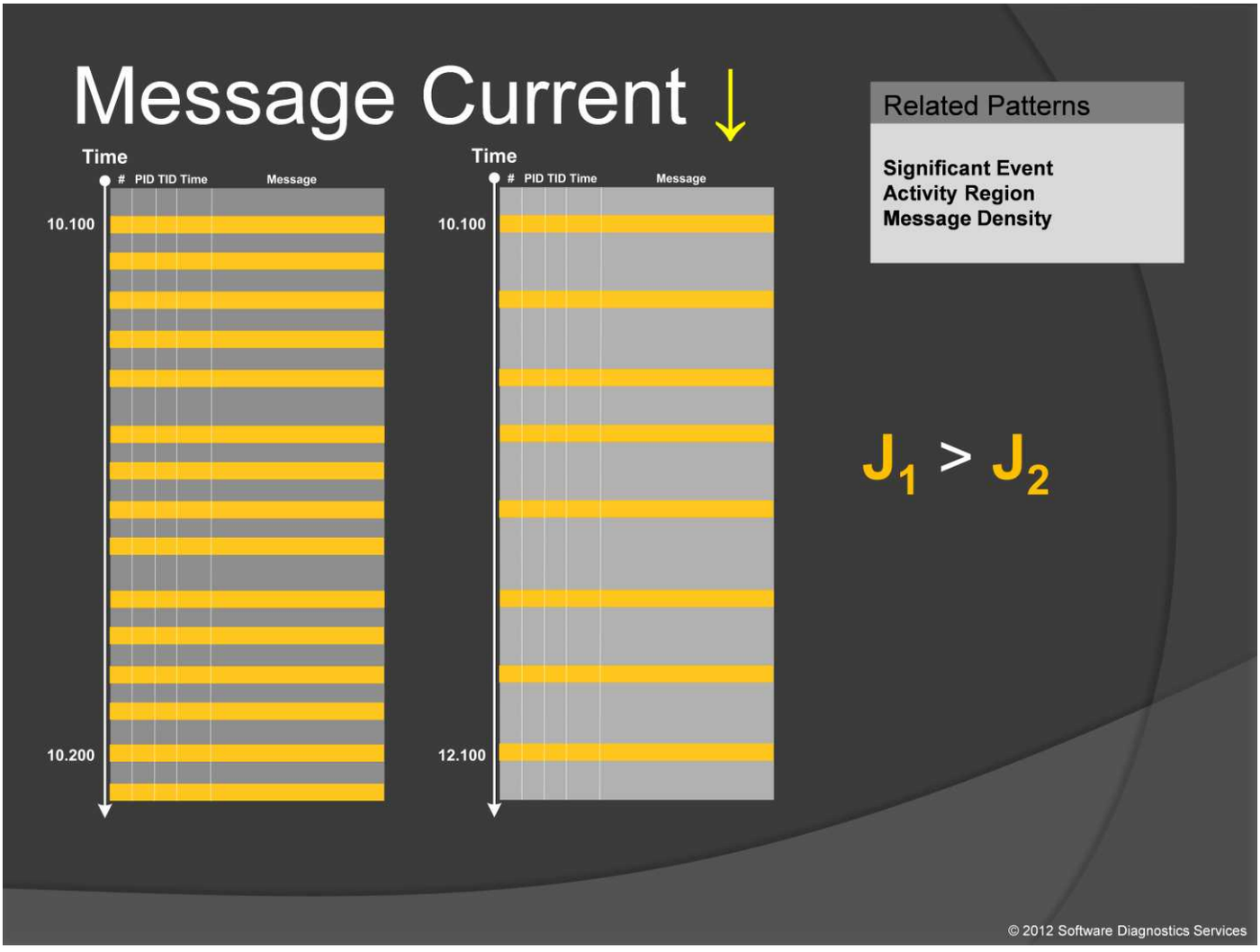
Here we introduce a software narratological (like a software story) partitioning of a trace into Head, Prologue, Core, Epilogue and Tail segments. Some elements such as Head and Tail may be optional and combined with Prologue and Epilogue. This is useful for comparative software trace analysis. Suppose, a trace started just before the problem reproduction steps or some particular **Significant Event** and finished just after the last reproduction steps or after another **Significant Event**. Then its core trace messages are surrounded by prologue and epilogue messages. What is before and after are not really needed for analysis (like noise). The size of a core trace segment need not be the same because environments and executed code paths might be different. However, often some traces are **Truncated** (another pattern). Please note that such partitioning can be done for any filtered trace such as **Adjoint Thread of Activity**.



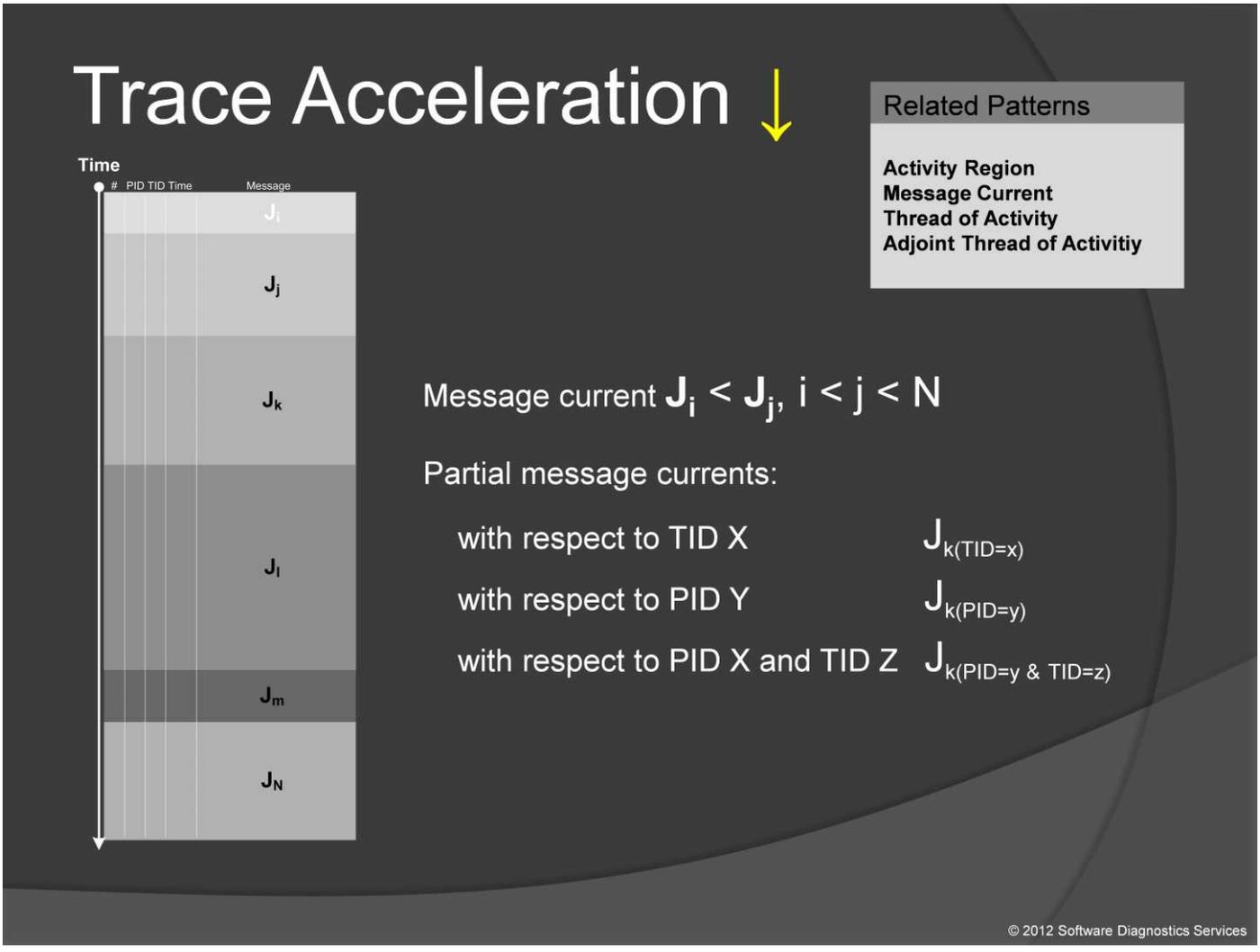
One of the common problems with tracing is rare and non-reproducible incidents. If the amount of tracing messages is small per second, it is possible to record events for hours or days. However, if we need to trace all events to do filtering later, then trace files can grow uncontrollably. Some tools enable circular tracing and after reaching particular file size, the tracing stream overwrites older messages. Such circular traces can be detected from timestamps, like in the example on this slide. Here the analysis **Focus of Tracing** region is found at the beginning of the trace because as soon as elusive and hard to reproduce problem happened the tracing was stopped.



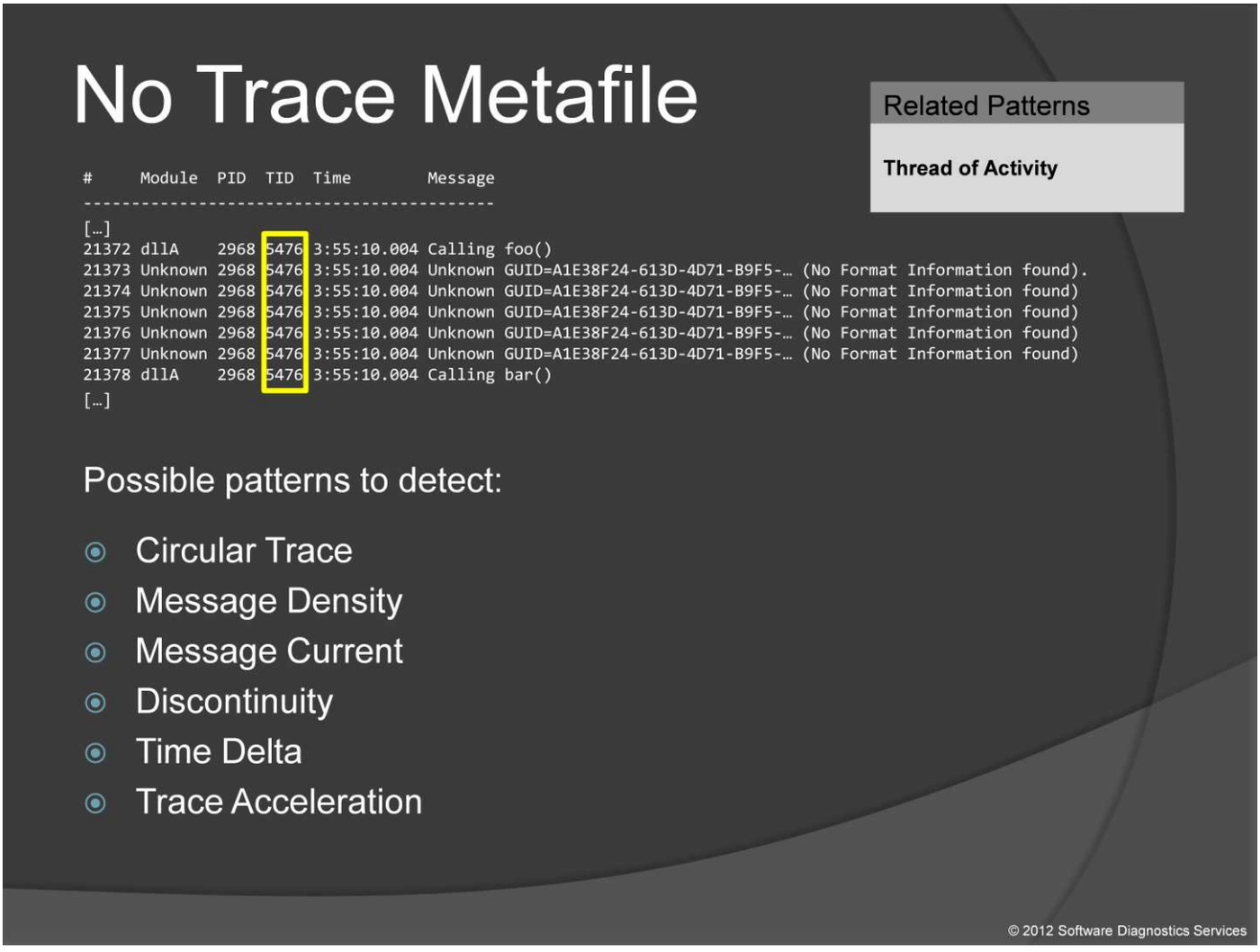
Sometimes we have several disjoint **Foci of Tracing** and possible false positives. We wonder where we should start our analysis and assign relative priorities for troubleshooting suggestions. Here **Message Density** pattern can help. The statement or message density is simply the ratio of the number of occurrences of the specific trace statement (message) in the trace to the total number of all different recorded messages. Consider this software trace with two frequent error messages and their corresponding trace densities D_{11} and D_{21}. The second index is for a trace number. Suppose their relative ratio is 6. Another trace for the same problem was collected at a different time with the same errors. It has much more total messages and only a few error messages of interest from the first trace. However, the ratio of densities is approximately the same, and this suggests that error messages are correlated. However, for the second trace statement density is 10 times lower and this suggests these problems might have started much later at some time later after the start of the trace recording, much bigger noise part (head part) from **Trace Partition**. We also look at **Relative Density** pattern again when we consider trace set patterns, patterns for trace message collections and sets.



Message Current (also called statement current) is the number of messages per unit of time. This is similar to velocity, first-order derivative. A trace can also be partitioned into **Activity Regions** with different currents and current can also be measured between significant events such as process start and exit. High trace current can be an indication of high CPU activity if a spiking thread has trace messages. Together with **Message Density,** this pattern can help in prioritizing troubleshooting suggestions.

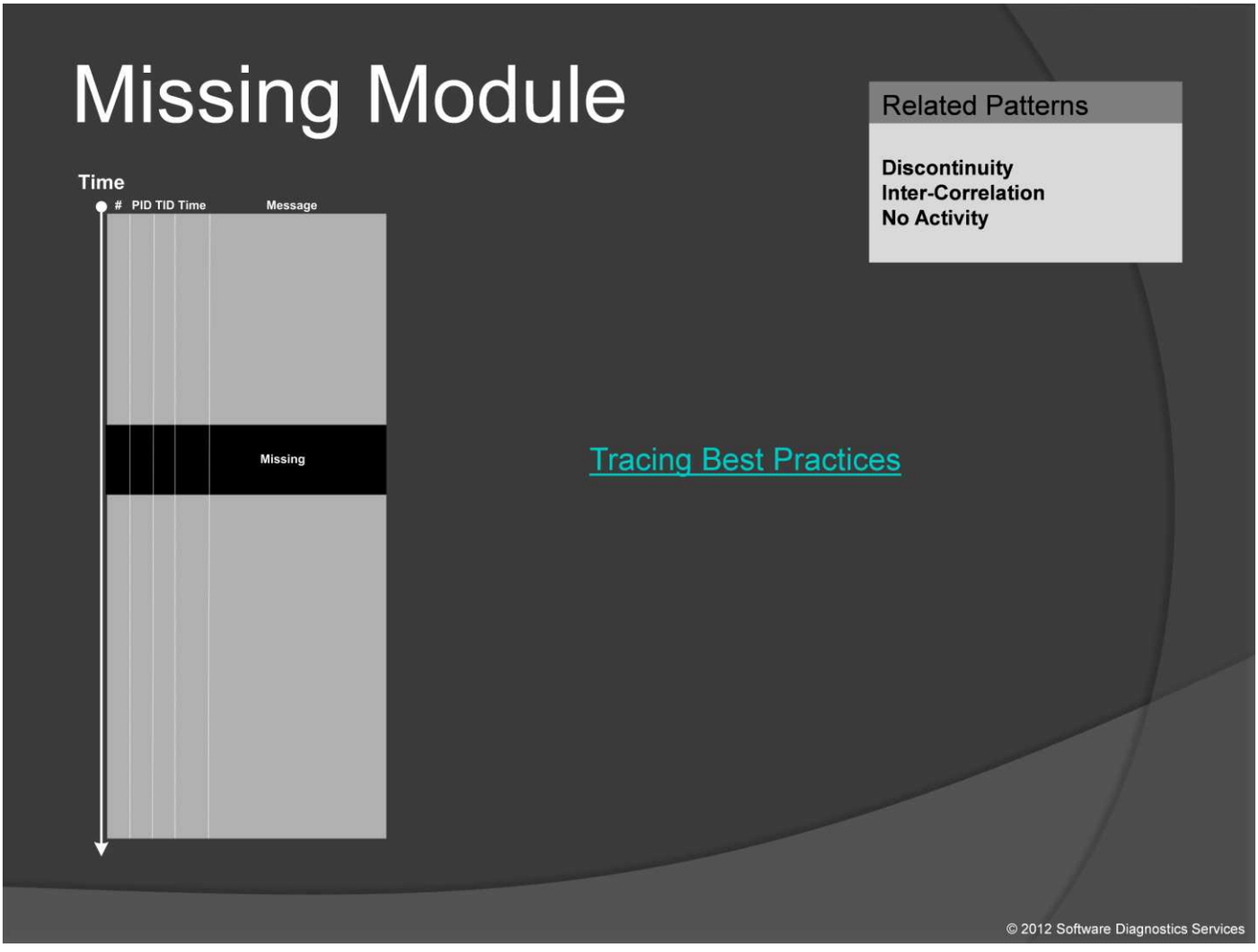


Sometimes we have a sequence of **Activity Regions** with increasing values of **Message Current** like depicted on this slide. The boundaries of regions may be blurry and arbitrarily drawn, of course. Nevertheless, the current is visibly increasing or decreasing. You can see an analogy with physical acceleration, second-order derivative. We can also metaphorically use here the notion of a partial derivative for trace **Message Current** and **Acceleration** for **Threads of Activity** and **Adjoint Threads of Activity**.



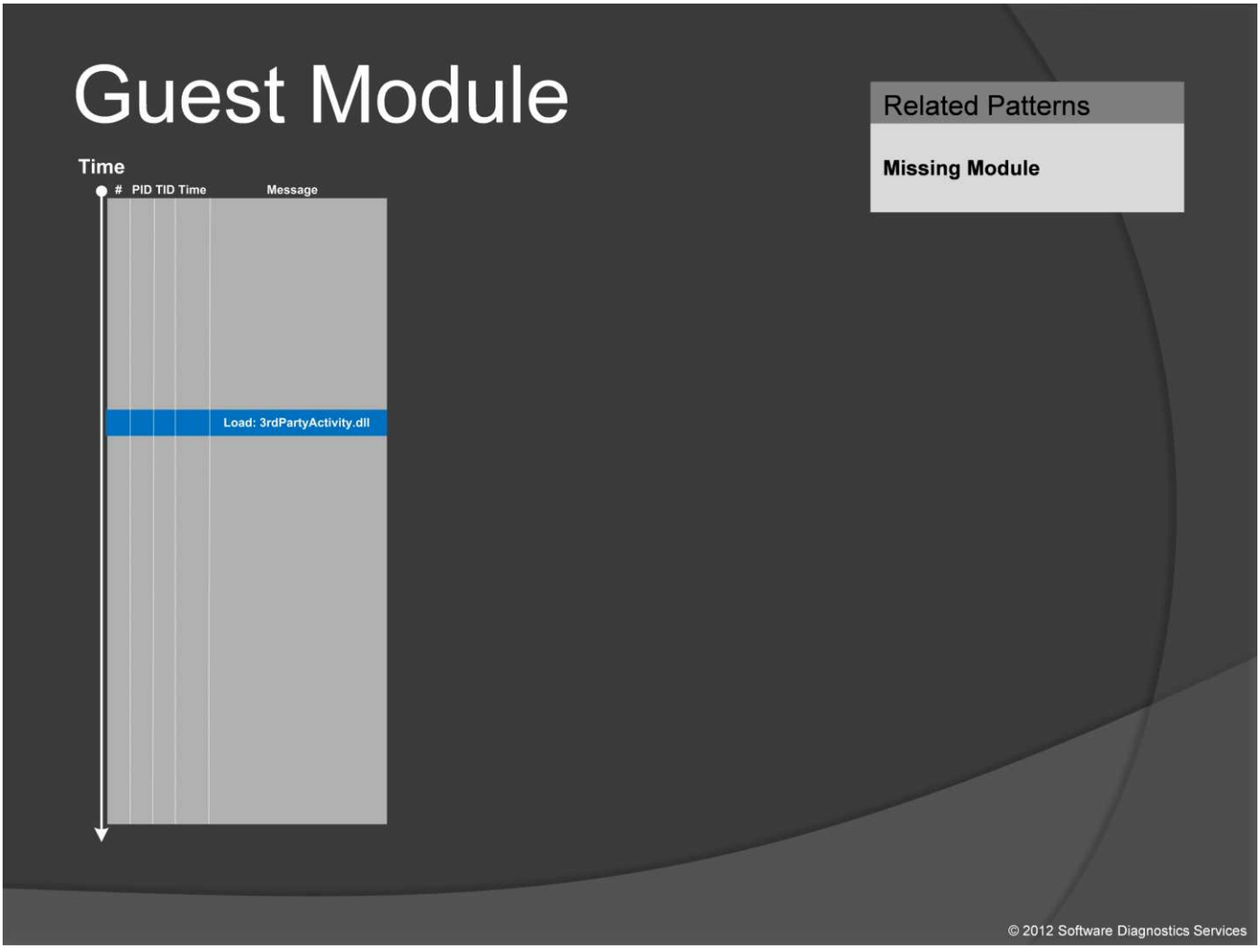
If you do live debugging and/or memory dump analysis you know that symbol files are necessary in order to map binary addresses to their proper symbolic forms such as function names. Modern software tracing implementations such as Microsoft Event Tracing for Windows and Citrix Common Diagnostics Format based on it do not store message text and formatting information in recorded message stream. After recording, similar to postmortem debugging, the trace is opened and associated necessary symbol files are used to convert stored short binary data to expanded human-readable textual format. In the world of Microsoft Tracing for Windows such files are called TMF files, Trace Meta Files. In some cases when we don't have TMF files it is still possible to detect broad behavioural patterns such as **Circular Trace**, **Message Density** and **Current**, **Discontinuity**, **Time Delta**, and **Trace Acceleration**. By looking at **Thread of Activity,** we can also sometimes infer the possible component or module name based on surrounding trace messages with present TMF files, especially when we have source code access. For example, in the trace above it can be dllA or any other module that *foo* function calls.

Empty Trace ranges from a totally empty trace where only a meta trace header (if any) describing overall trace structure is present to a few messages where we expect thousands. This is also an extreme case of **Truncated Trace**, **No Activity,** and **Missing Components** patterns. Also please note that an empty trace file doesn't necessarily have a zero file size because a tracing architecture may pre-allocate some file space for block data writing such as Citrix CDF tracing. I put some recommendations on this slide.

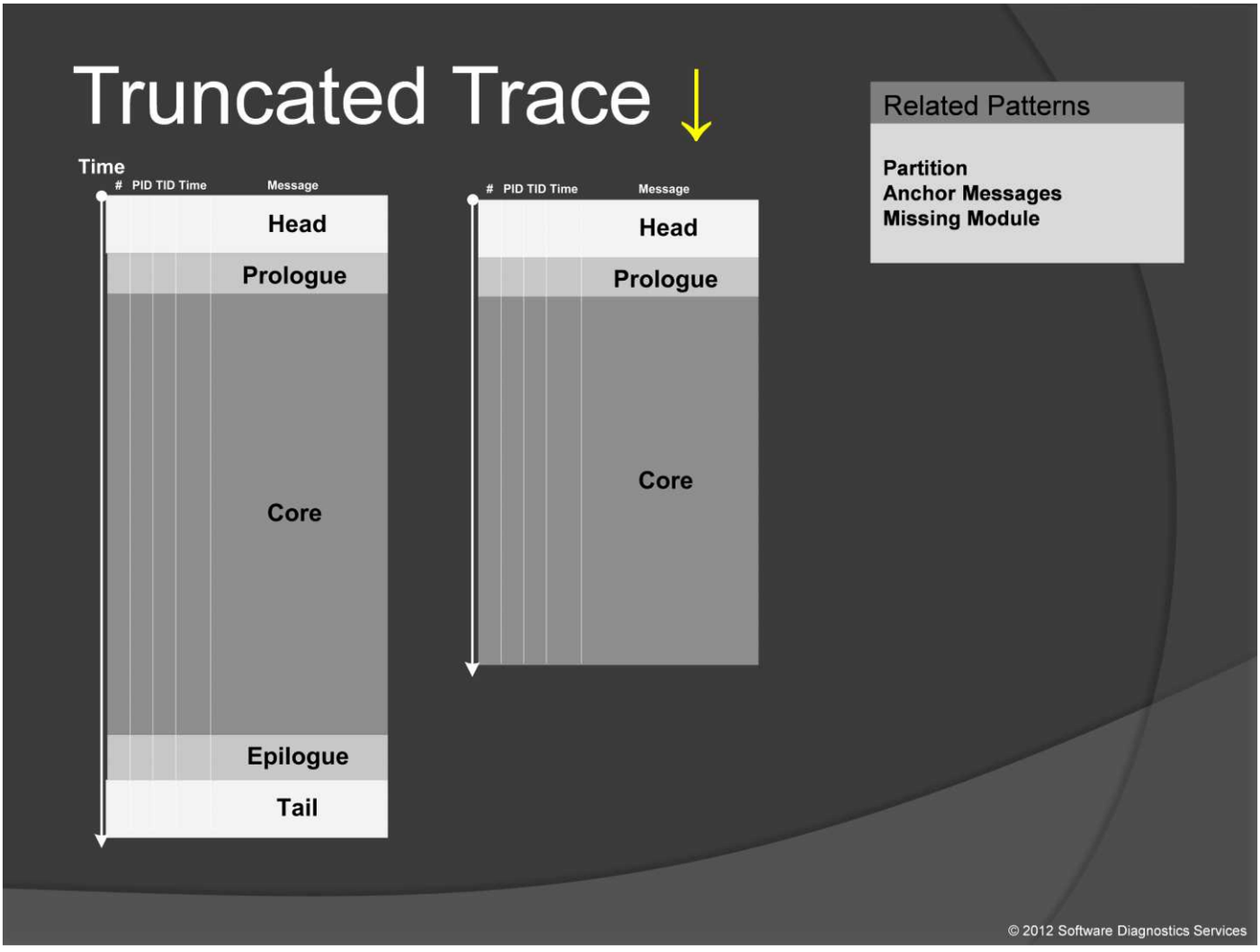


Sometimes, we don't see trace messages we expect and wonder whether a module was not loaded, its container process ceased to exist or simply it wasn't selected for tracing. In many support cases, there is a trade-off between tracing everything and the size of trace files. Customers and engineers usually prefer smaller files to analyse. However, in the case of predictable and reproducible issues with short duration, we can always select all modules or deselect a few (instead of selecting a few). I also put a link to Citrix support article for Citrix CDF tracing best practices and it can be applied to other software traces as well. **Missing Module** pattern is closely related to **Discontinuity** pattern with a possibility of a sudden and silent gap in trace statements that could have happened because not all necessary modules or components were selected for tracing. Sometimes, in cases when a module was selected for tracing but we don't see any trace output from it other traces from different tracing tools such as Process Monitor can give us an indication, for example, showing a load failure message. This is an example of trace **Inter-Correlation** pattern that we cover later.

Tracing Best Practices: http://support.citrix.com/article/ctx121185

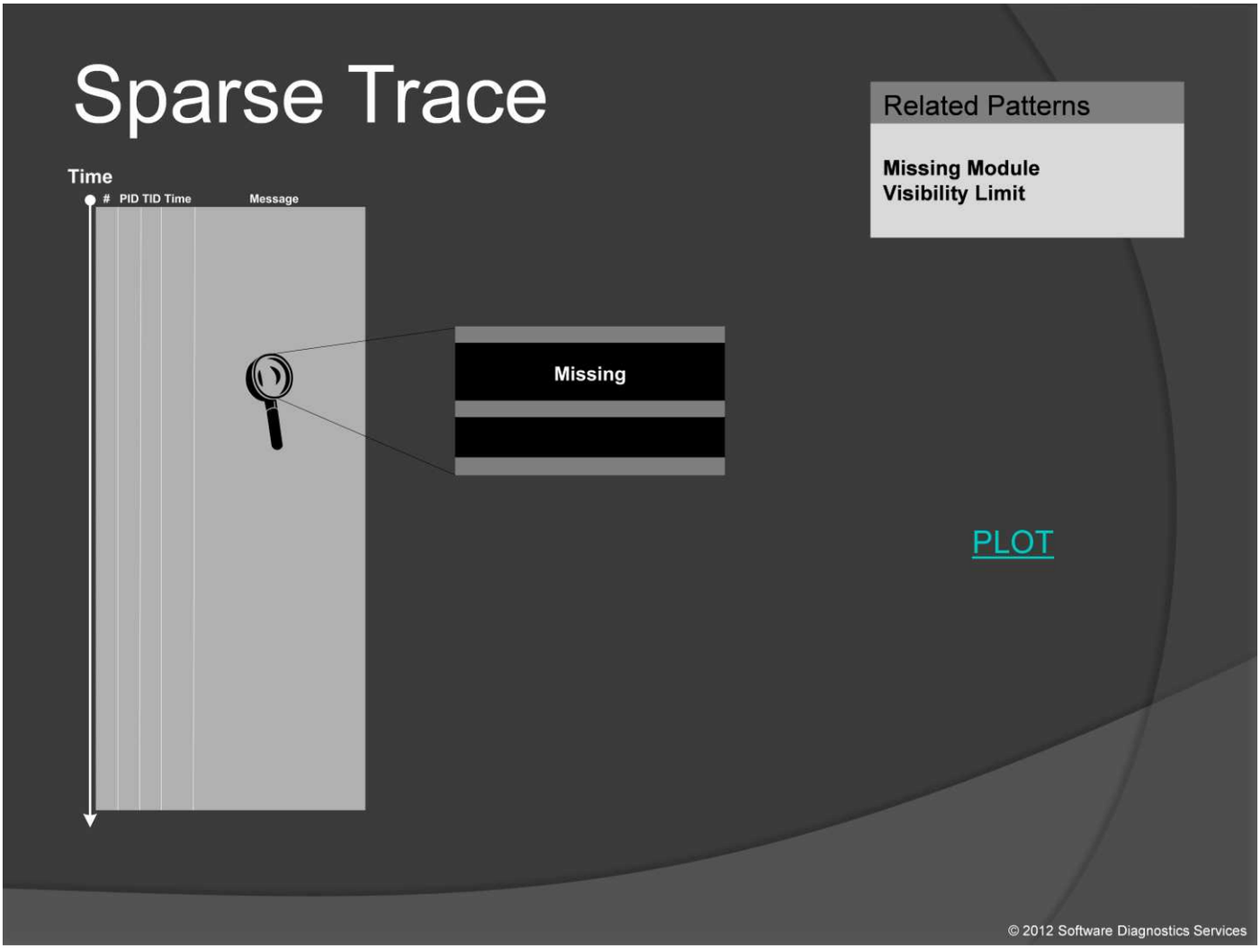


Often, when comparing normal, expected (or working) and abnormal (or non-working) traces we can get clues for further troubleshooting and debugging by looking at module load events. For example, when we see an unexpected module load event in our non-working trace, its function (and sometimes even module name) can signify some differences to pay attention to. **Guest Module** pattern is different from **Missing Module** pattern we covered previously. Although in the latter pattern a missing module in one trace may appear in another trace but the module name is known apriori and expected. In the former pattern, a module is unexpected. For example, in the trace picture above, the appearance of *3rdPartyActivity* DLL may suggest further investigation if the activity is related to functional activity in a normal working trace. Another example is WER module.



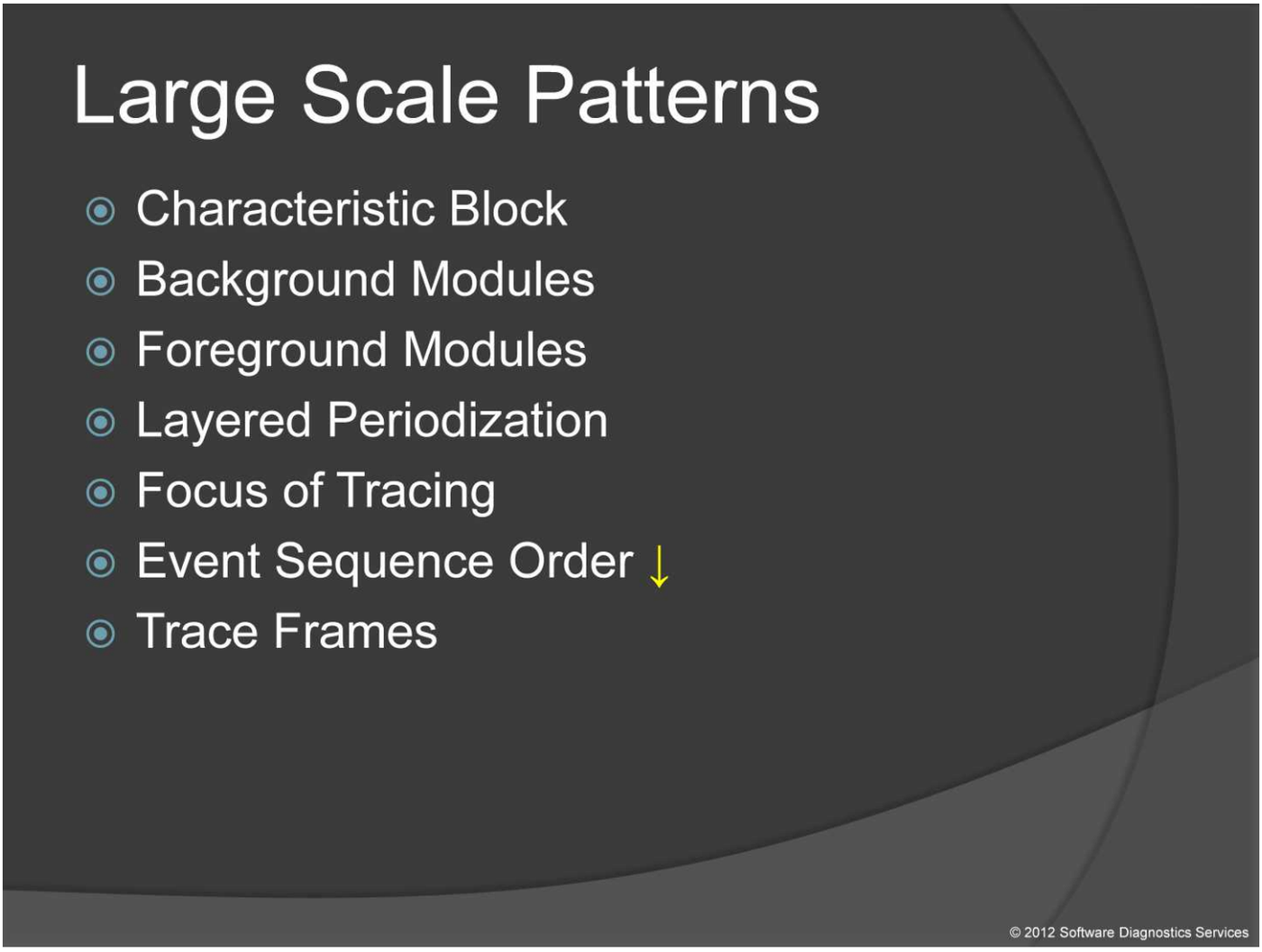
Sometimes a software trace is **Truncated** when a trace session was stopped prematurely, often when a problem didn't manifest itself visually. We can diagnose such traces by their short time duration, missing **Anchor Messages** or **Modules** necessary for analysis. My favourite example is user session initialization in a terminal services environment when problem effects are visible only after the session is fully initialized, and an application is launched. The module that monitors process creation events was included for tracing and we expect a full process launch sequence from csrss.exe to an application executable. However, a trace only shows the launch of the *winlogon.exe* executable and other trace messages for a few seconds after that.

Sometimes it is not possible to trace from the very beginning of a user or system interaction. Moreover, internal application tracing cannot trace anything before that application start and its early initialization. The same is for system wide tracing which cannot trace before the tracing subsystem or service starts. Therefore, each log has its intrinsic **Visibility Limit** in addition to possible **Truncation** or **Missing Modules** patterns that cover cases we can avoid.



Recall that behind any trace statement is source code fragment (the so-called Program Lines of Trace, **PLOT** abbreviation). This pattern covers the missing trace statements in source code. Potentially it is possible to trace every source code line or implementation language statement but in practice trace statements in source code are added only in places when a developer finds it useful to aid possible future debugging. Monitoring tools also trace certain public API and specific functionality such as file and registry access or network communication. Therefore, it is often a case that when we don't see anything in a trace or see very little, this is because particular source code fragment was not covered by trace statements. This **Sparse Trace** pattern is different from **Missing Module** pattern where some modules were not included for tracing explicitly although there is tracing code there. It is also different from **Visibility Limit** pattern where tracing is intrinsically impossible. As a result, after an analysis of such traces, technical support and escalation engineers request to add more trace statements and software engineers extend tracing coverage iteratively as needed.

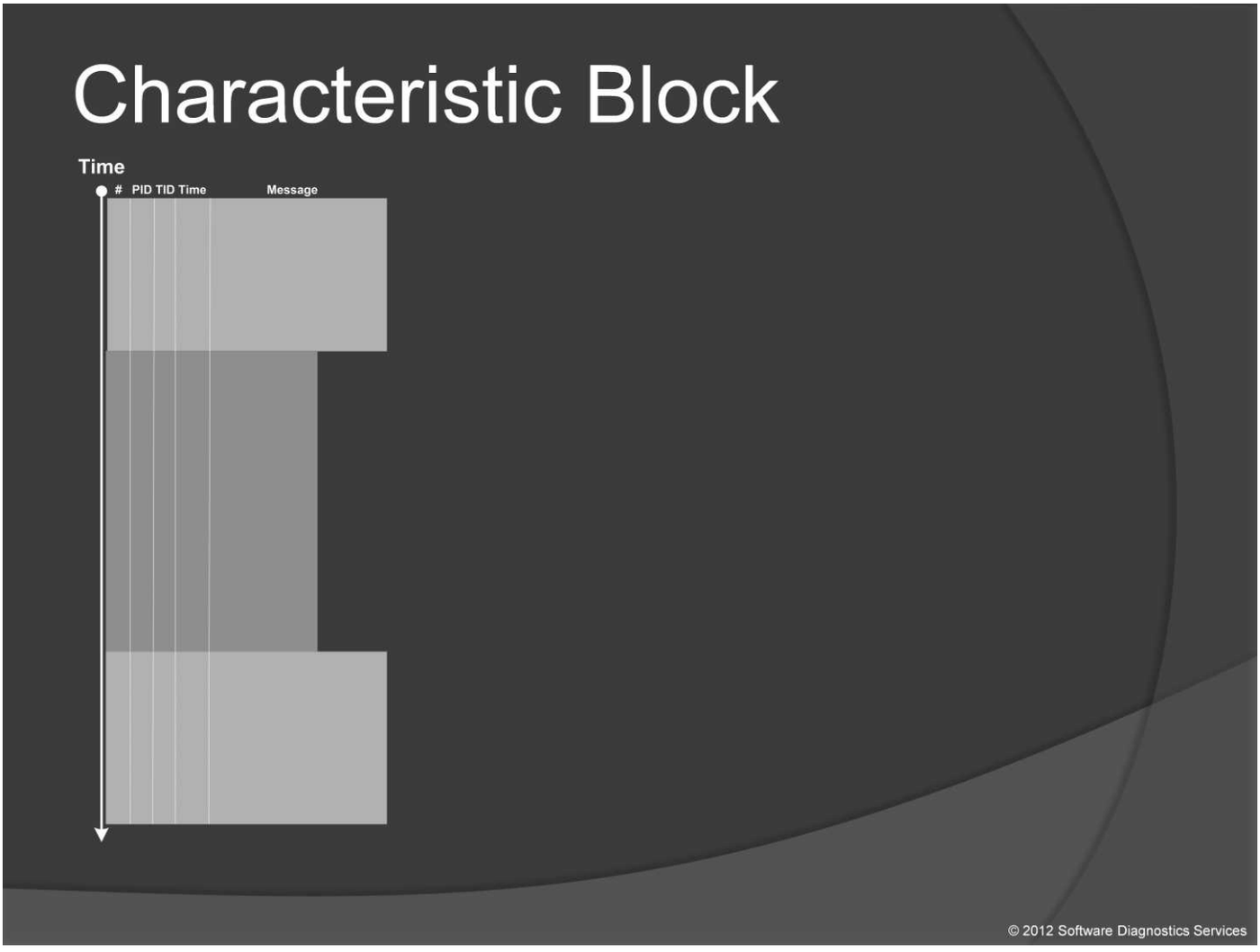
PLOT: Memory Dump Analysis Anthology, Volume 5, page 272
http://www.dumpanalysis.org/blog/index.php/2010/05/06/basic-software-plots-part-0/



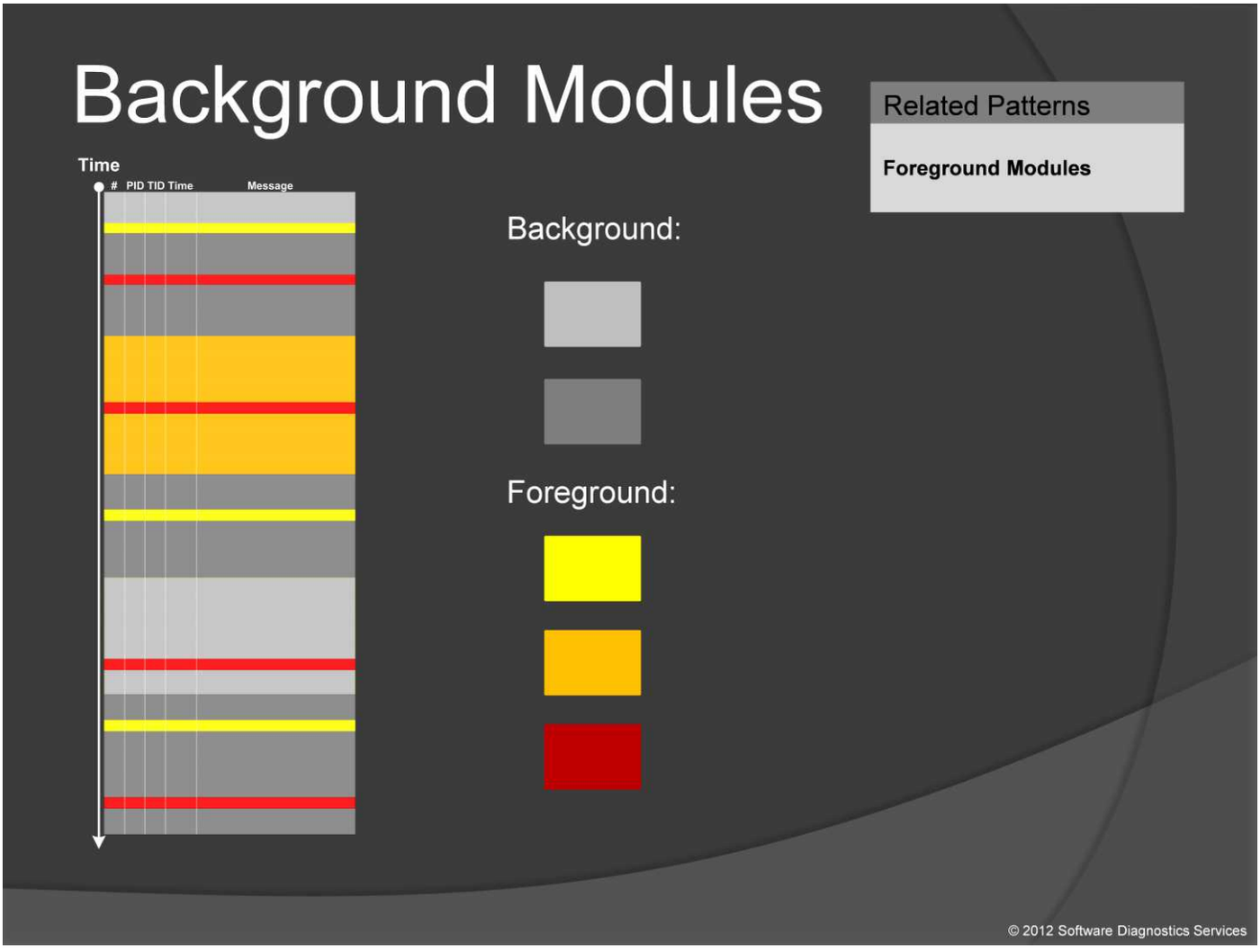
The fourth block of patterns we cover are large scale trace patterns. They are about coarse grain structure of software traces and logs where the division unit is often a module or some high-level functionality.

Bird's Eye Binary View

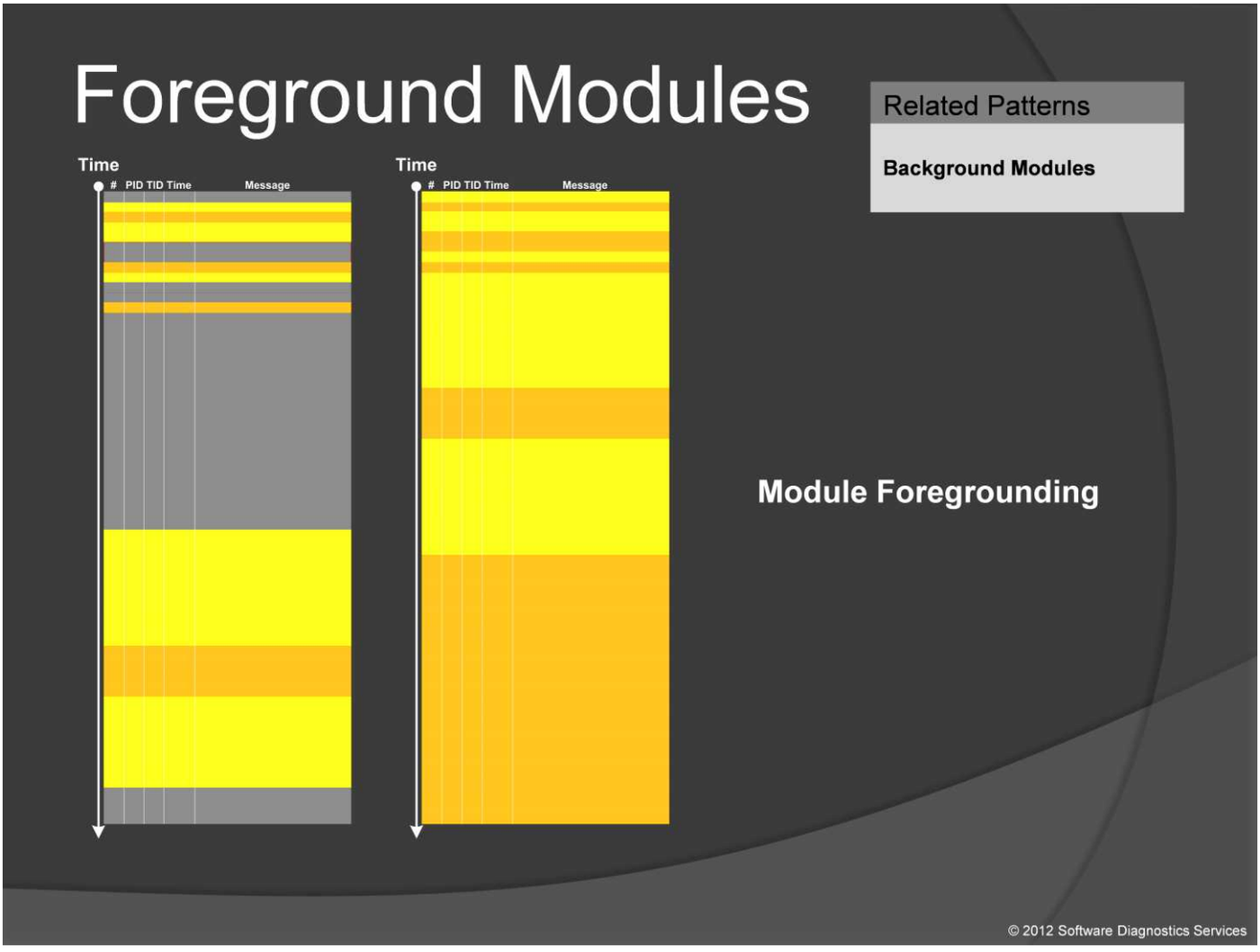
Here's a bird's eye view of binary software traces: Process Monitor log and Citrix CDF file. However, we are concerned with formatted textual representations.



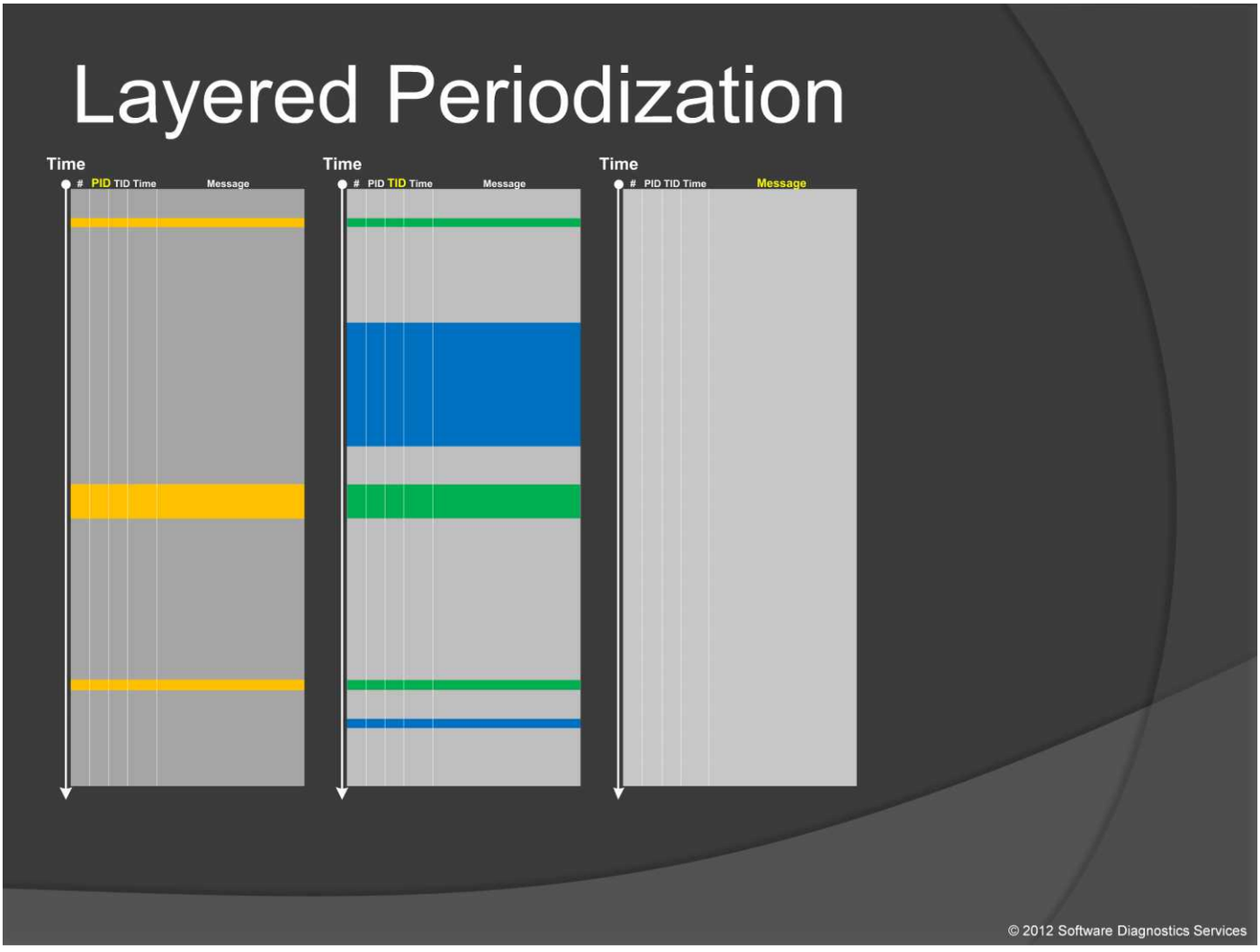
Textual representations can also be viewed from bird's eye perspective. Irregularities in formatting make it easier to see the coarse blocked structure of a software trace or log. Typical examples here are uniform debugging message stream and some very long repeated activity like retries. Such blocks of output can be seen when scrolling trace viewer output but if a viewer supports zooming it is possible to get an overview and jump directly into a **Characteristic Message Block**. For example, we can open a log file in a word processor, choose the smallest font possible and select multipage view. Sometimes this pattern is useful to ignore bulk messages and start the analysis around block boundaries.



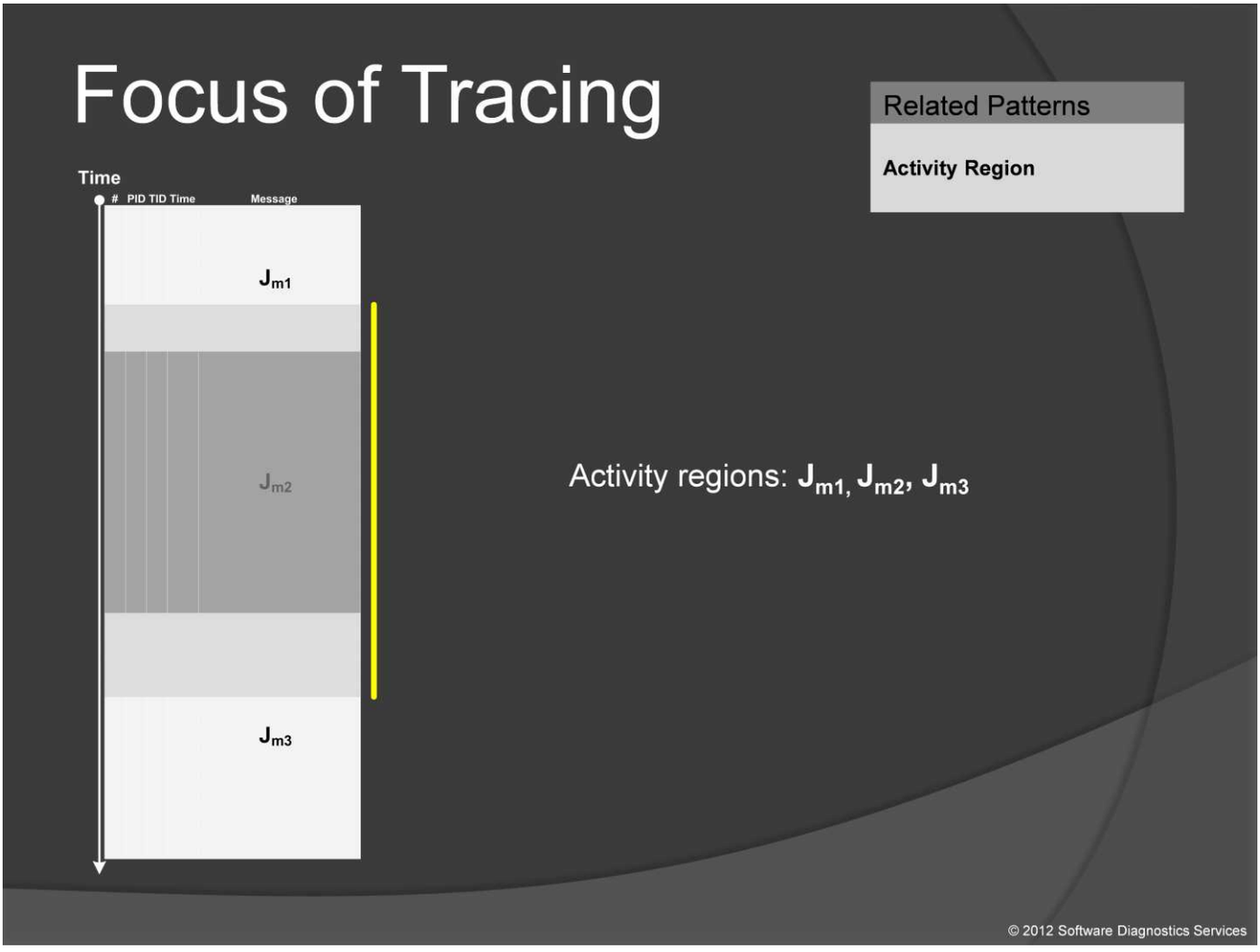
To illustrate **Background and Foreground Modules** (also called **Components**) pattern let's suppose we are troubleshooting a graphical user interface issue using a software trace containing the output from all components of the problem system. User interface modules and their messages are foreground for a trace viewer (or a person) against numerous background modules such as database, file, and registry access, shown in shades of grey. So we see that choice of background and foreground components depends on a problem.



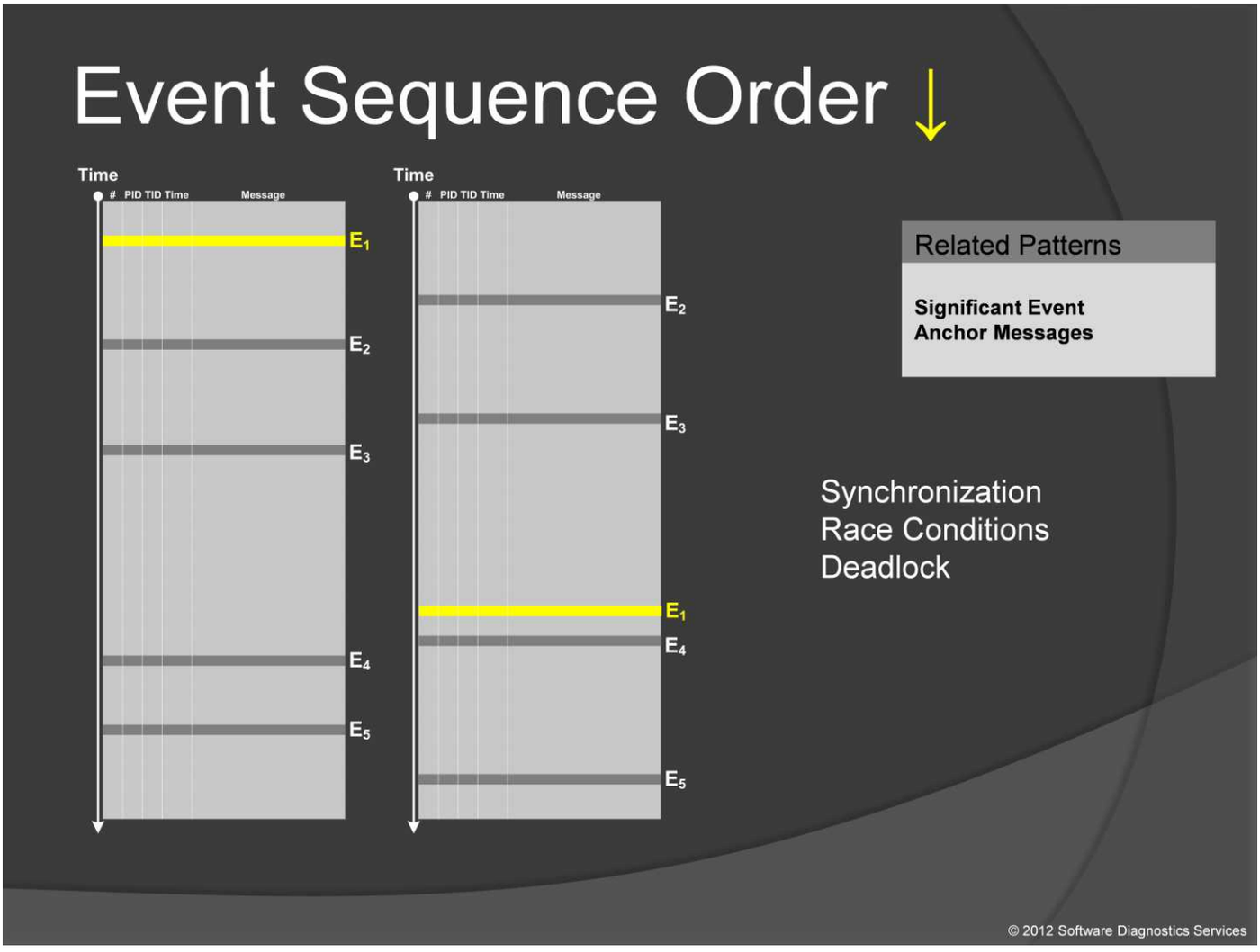
Trace viewers such as CDFAnalyzer and Process Monitor can filter out (or exclude) background component messages and present only foreground modules (that we call ***module*** or ***component foregrounding***). Here background modules can be considered as noise to filter out. Of course, this process is iterative, and parts of what once was foreground become background and candidates for further filtering.



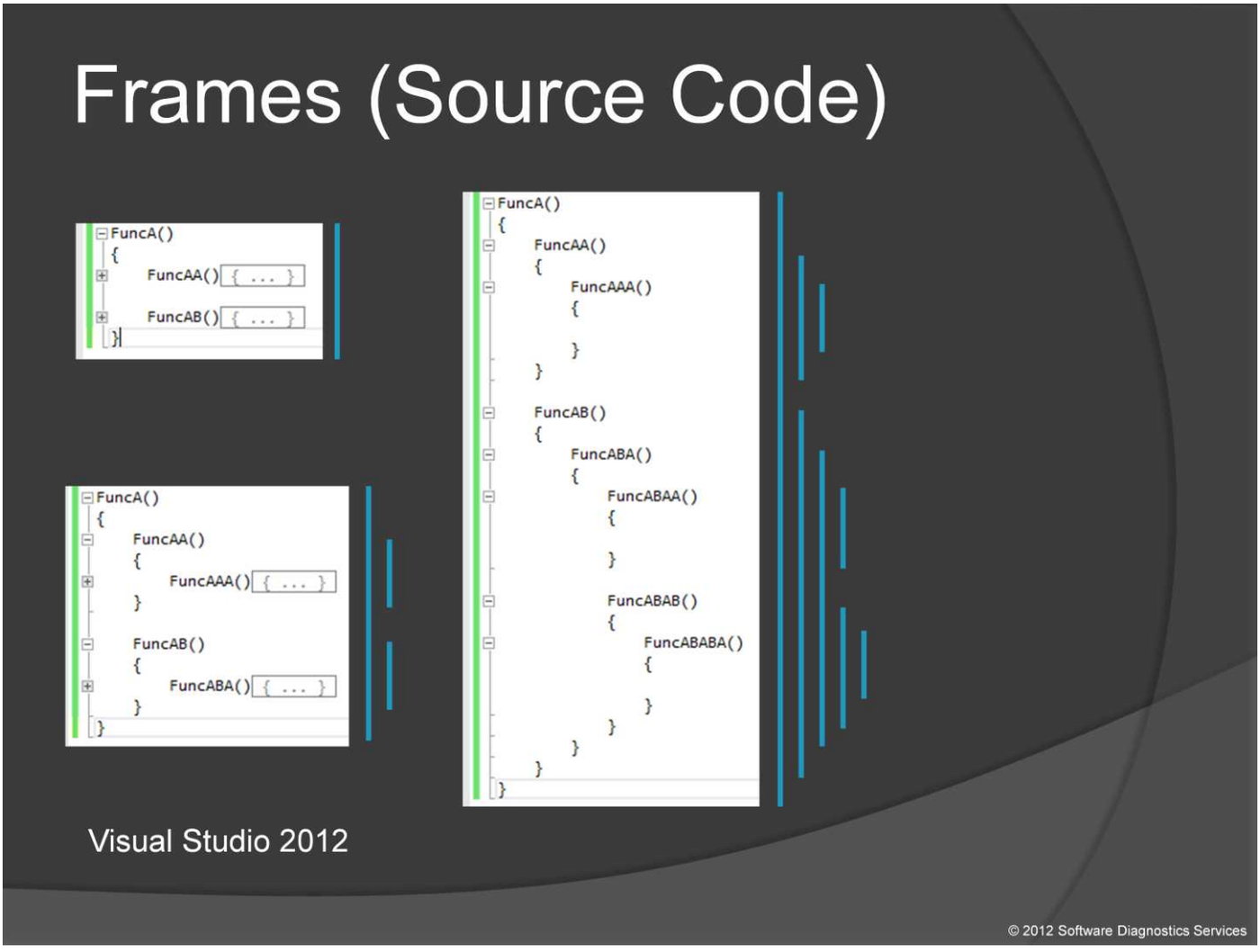
The periodization of software trace messages may include individual messages, then aggregated messages from threads, and then aggregated threads into processes and finally individual computers (in a client-server or similar sense). This is best illustrated graphically. We see a message layer on the right, and then goes thread layer where different colors correspond to different TIDs. Then on the left we see process layer where other different colors correspond to different PIDs. It is also possible to have a different periodization based on modules, functions, and individual messages. For such periodization, we should remember that different threads can enter the same module or function.



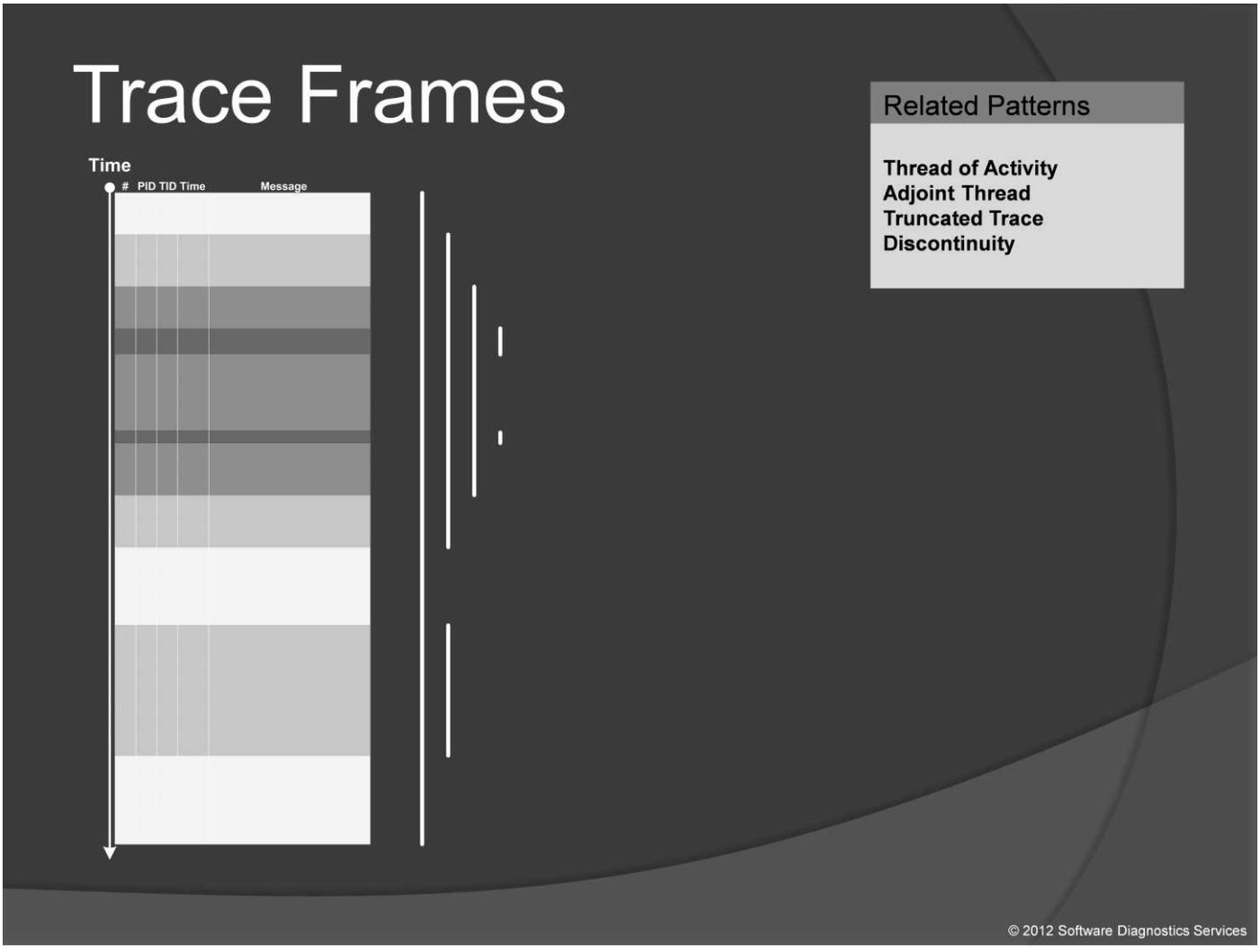
A software trace or log consists from the so-called **Activity Regions** with syntactical and visual aspects of trace analysis whereas **Focus of Tracing** brings attention to changing semantics of trace message flow, for example, in Citrix terminal services environment, from logon messages during session initialization to database search. Here is a graphical illustration of this pattern where tracing focus region spans 3 regions of activity.



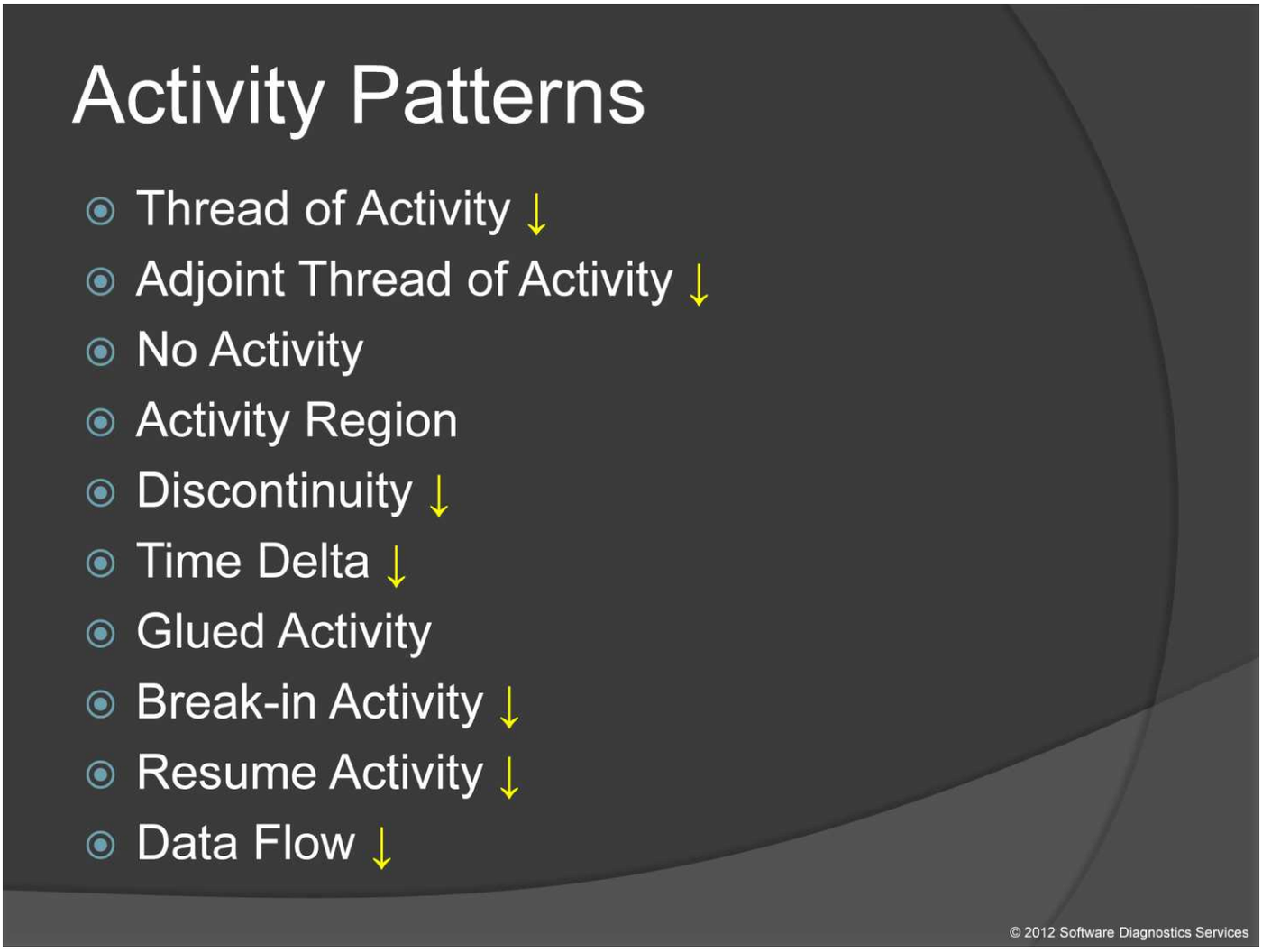
In any system, there is an expected **Event Sequence Order** as a precondition to its normal behaviour. Any out-of-order events should raise the suspicion bar as they might result or lead to synchronization problems such as race conditions and deadlocks. It need not be a sequence of trace messages from different threads but also from processes, for example, image load events in Citrix CDF traces can indicate a misconfiguration in a session initialization process startup order.



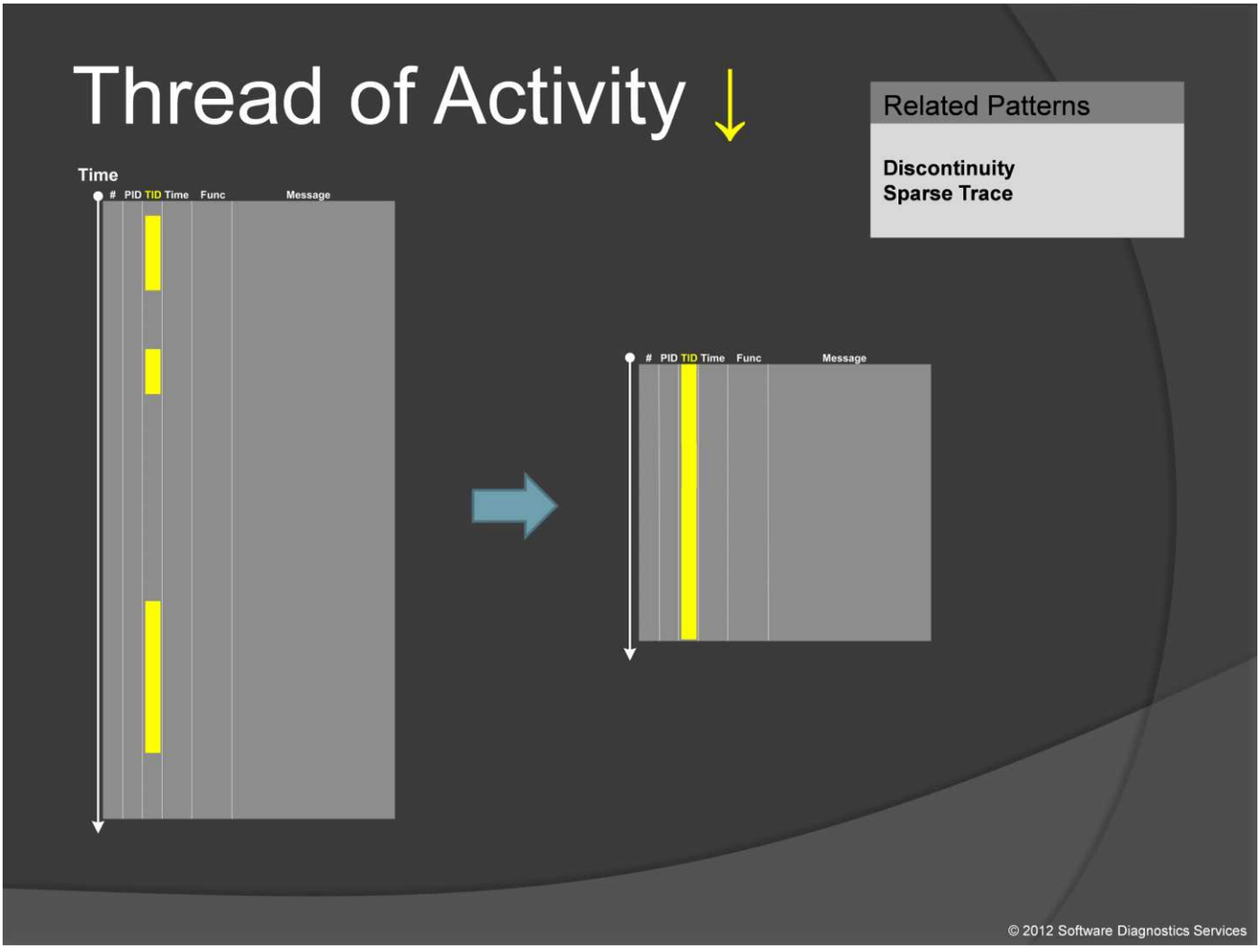
Our next pattern is **Trace Frames** and to make it more understandable, I found a good example in Visual Studio editor where we can expand and collapse nested function declarations. I highlighted frames in blue lines to the right of each screenshot.



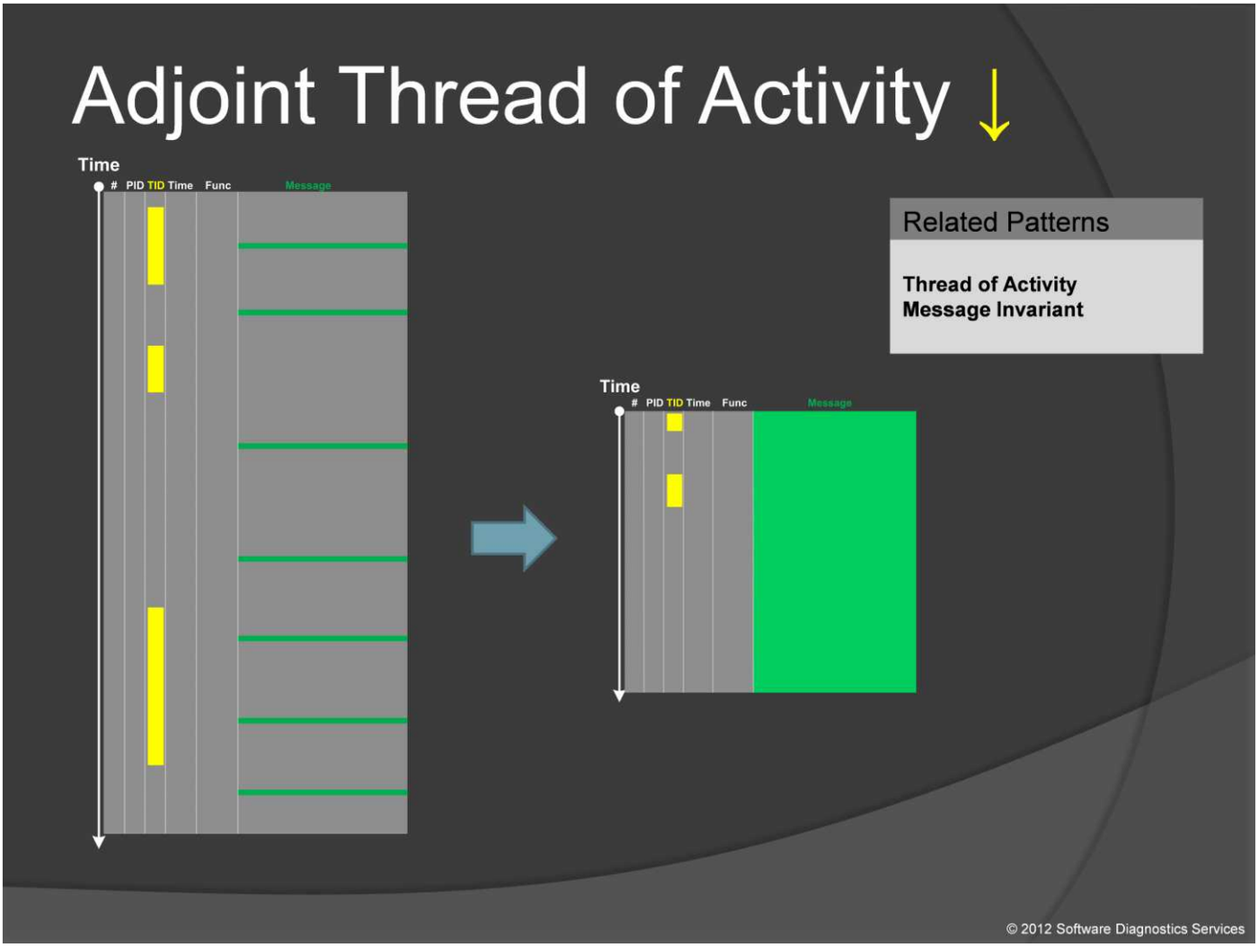
Similar to source code, we also have frames in software traces. Some products use indentation in textual logs. At the level of a software trace or filtered thread or **Adjoint Thread of Activity,** we can clearly see **Discontinuities**.



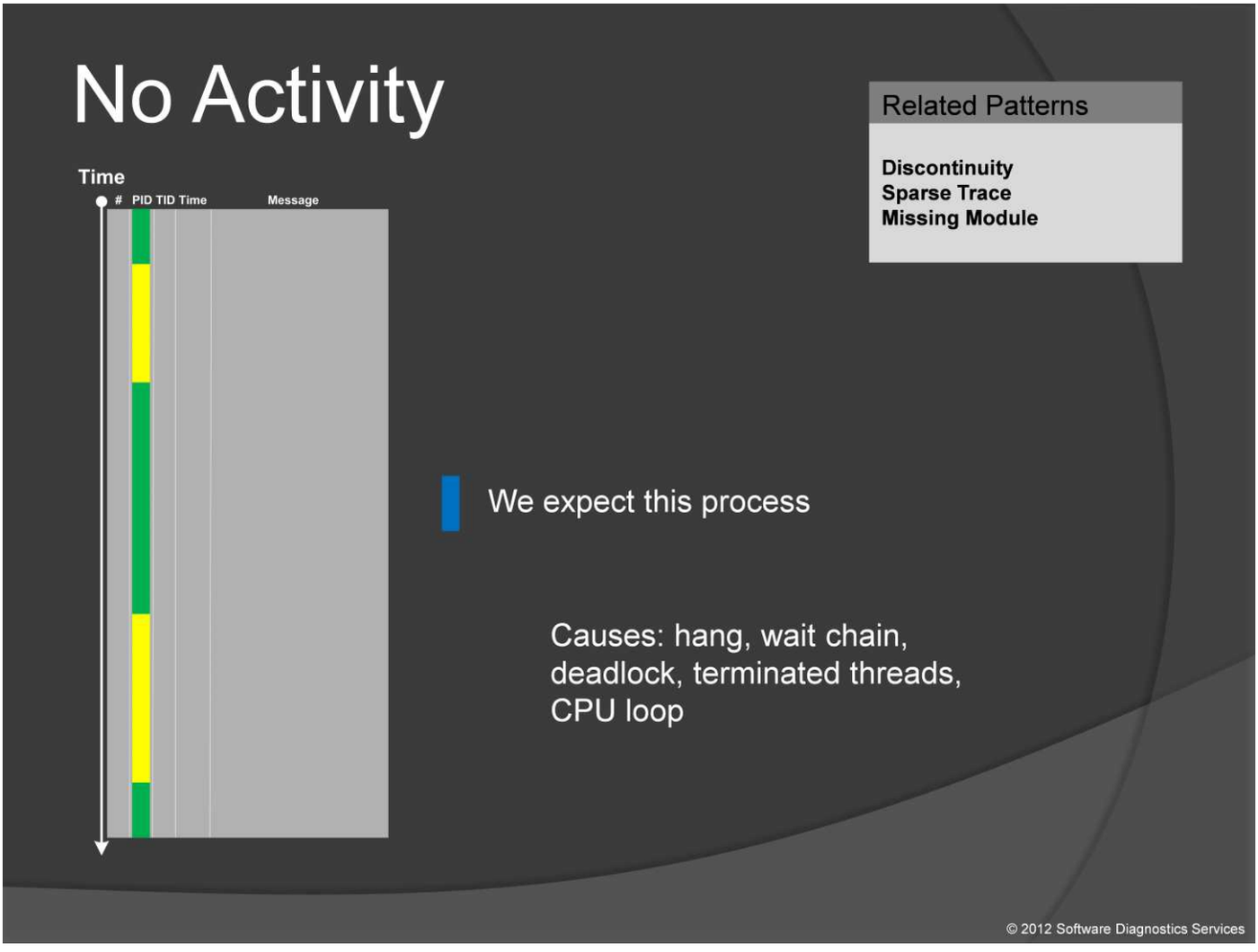
The fifth block of patterns is related to various software activities we see in logs and traces. Most of them involve time dependency.



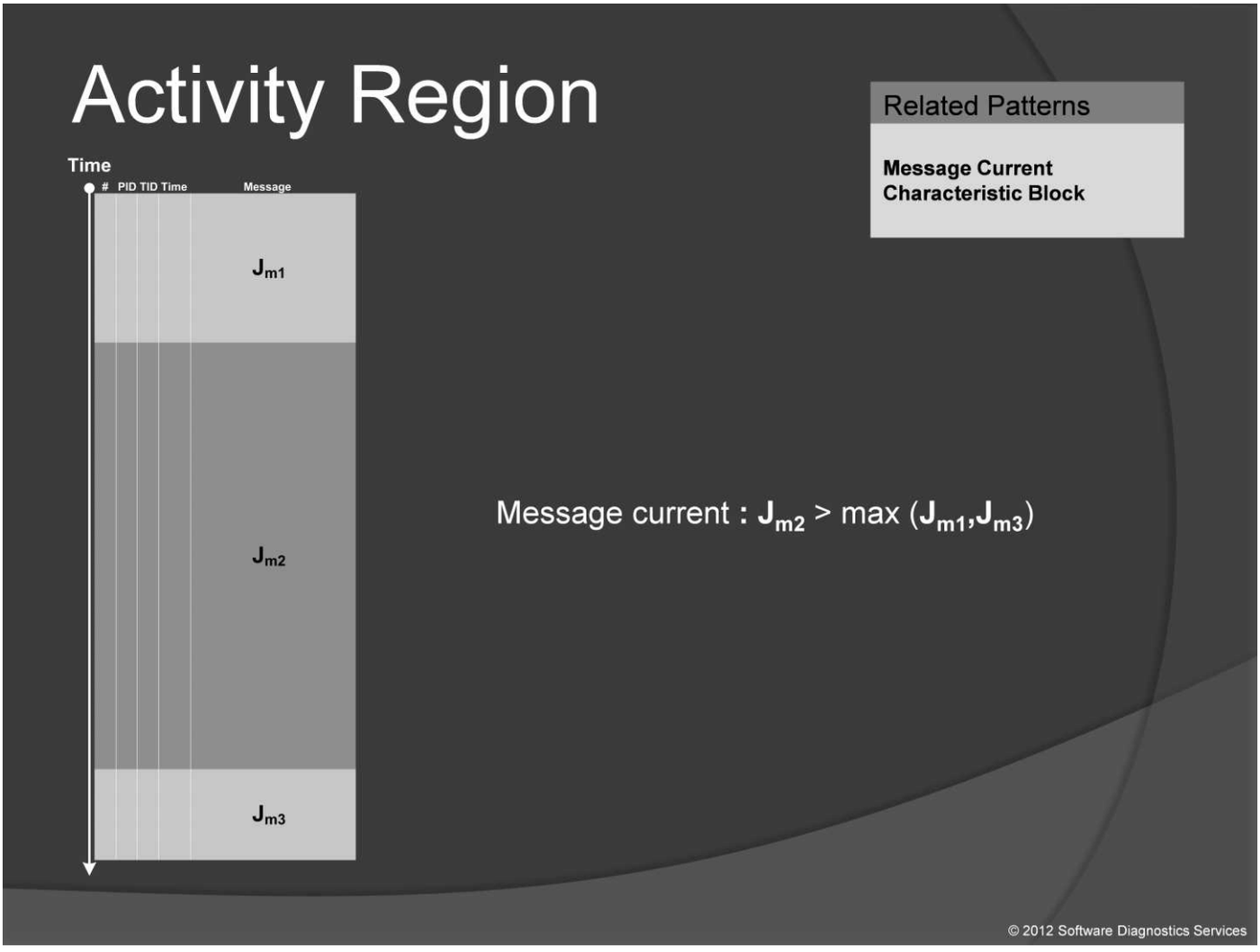
This pattern is about trace messages associated with particular TID. Usually, when we see an error indication or some interesting message we select its current thread and investigate what happened in this process and thread before. By looking at threads, we can spot discontinuities that can be signs of inter-process communication and even CPU activity if code loops are not covered by trace statements (the so-called **Sparse Trace** pattern). Here a supplemental memory dump may reveal stack traces and help in further diagnostics of abnormal software behaviour.



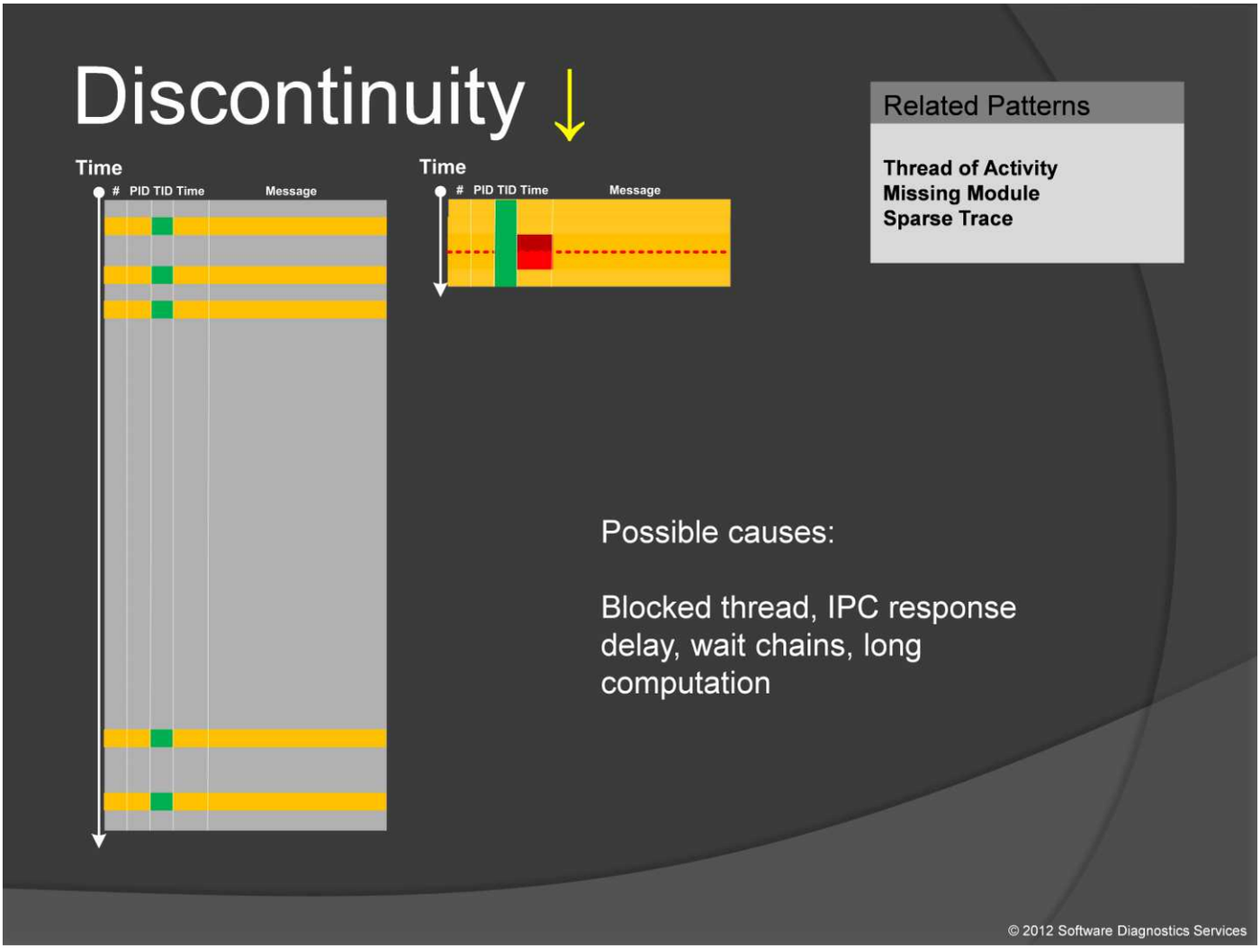
As I mentioned in the introduction, **Adjoint Thread** is an extension of **Thread of Activity** pattern. In the picture, we see a message stream where some messages are coming from specific TID shown in yellow color. Suppose we are interested in some specific trace **Message Invariant** such as related to "CreateProcess". In such trace messages, usually, there is some message invariant part saved in short binary form possibly in GUID format for fast recording together with some data as process image name. Later, when this message is formatted for display, any variant part is formatted and appended to it. However, because of the invariant part, it is possible to filter such messages and form an **Adjoint Thread of Activity**.



A **Discontinuity** pattern is seen when some activity ceases to manifest itself in trace messages. Its limit is **No Activity** where the absence of activity can be seen at a thread level or at a process level or at a module level with the latter is similar to **Missing Module** pattern. The difference from the latter pattern is that we know for certain that we selected our modules for tracing but don't see any trace messages at all. If a process started before tracing session and there is no activity we can think of it as hanging caused by thread wait chains, deadlocks. It is also possible that certain parts of the code are not covered by trace statements (the so-called **Sparse Trace** pattern), and they are simply looping. Here a memory dump would be helpful.



When looking at long traces with millions of messages, we can see regions of activity where **Message Current** (J_m, msg/s) is much higher than in surrounding temporal regions or partition a trace into **Characteristic Message Blocks** where gaps between them are filled with some background activity.



Sometimes we may see delays in application or service startup, terminal user session initialization, long response times and simply the absence of response. All these problems can be reflected in software traces showing sudden time gaps in **Threads of Activity**. There can be different causes for discontinuities such as threads blocked in inter-process communication but with preselected timeout, there may be some CPU intensive computation not covered by trace statements in source code (**Sparse Trace** pattern). It could also be the case of **Missing Modules** not selected for tracing.

Time Delta ↓

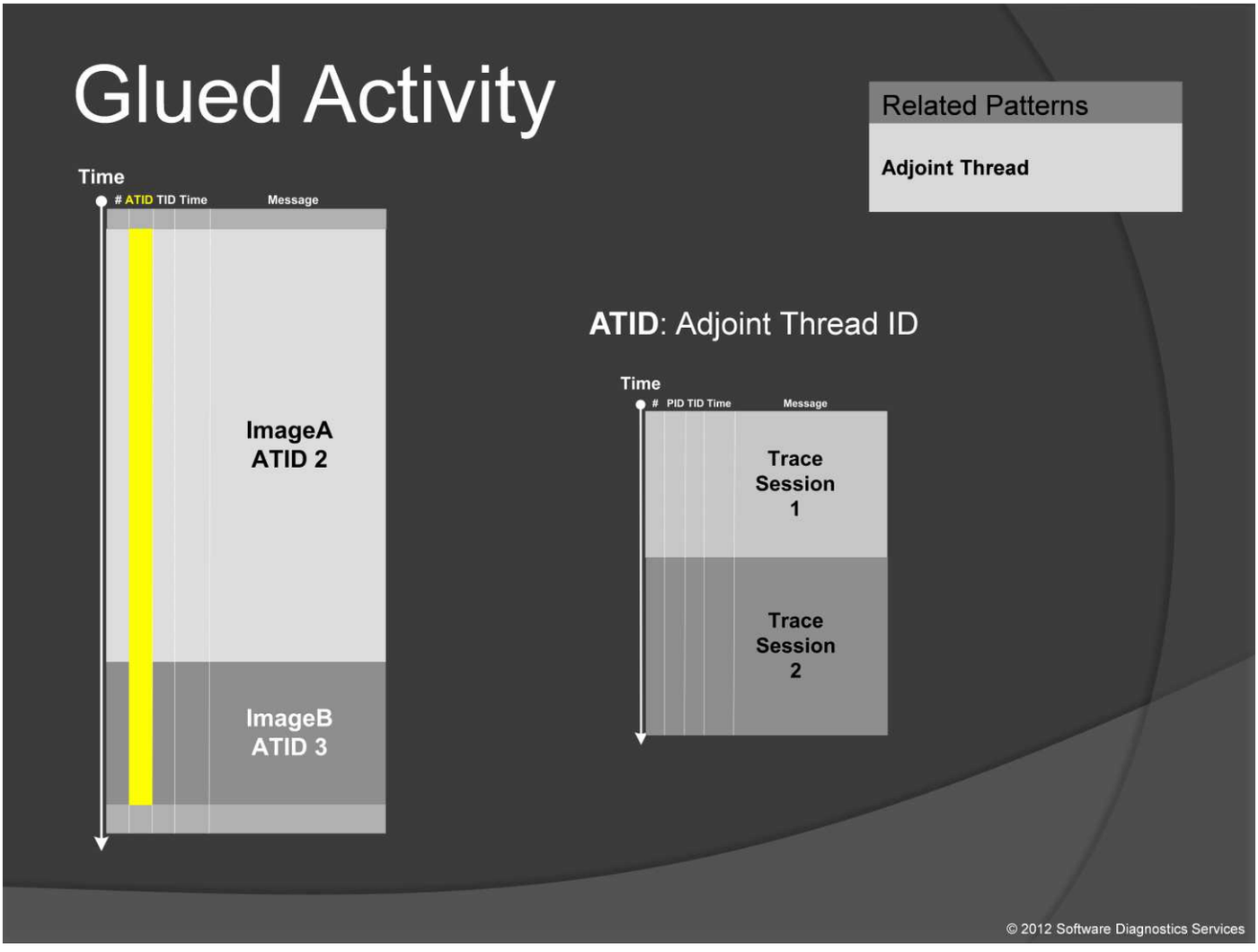
Related Patterns

Basic Facts
Thread of Activity
Discontinuity
Significant Event

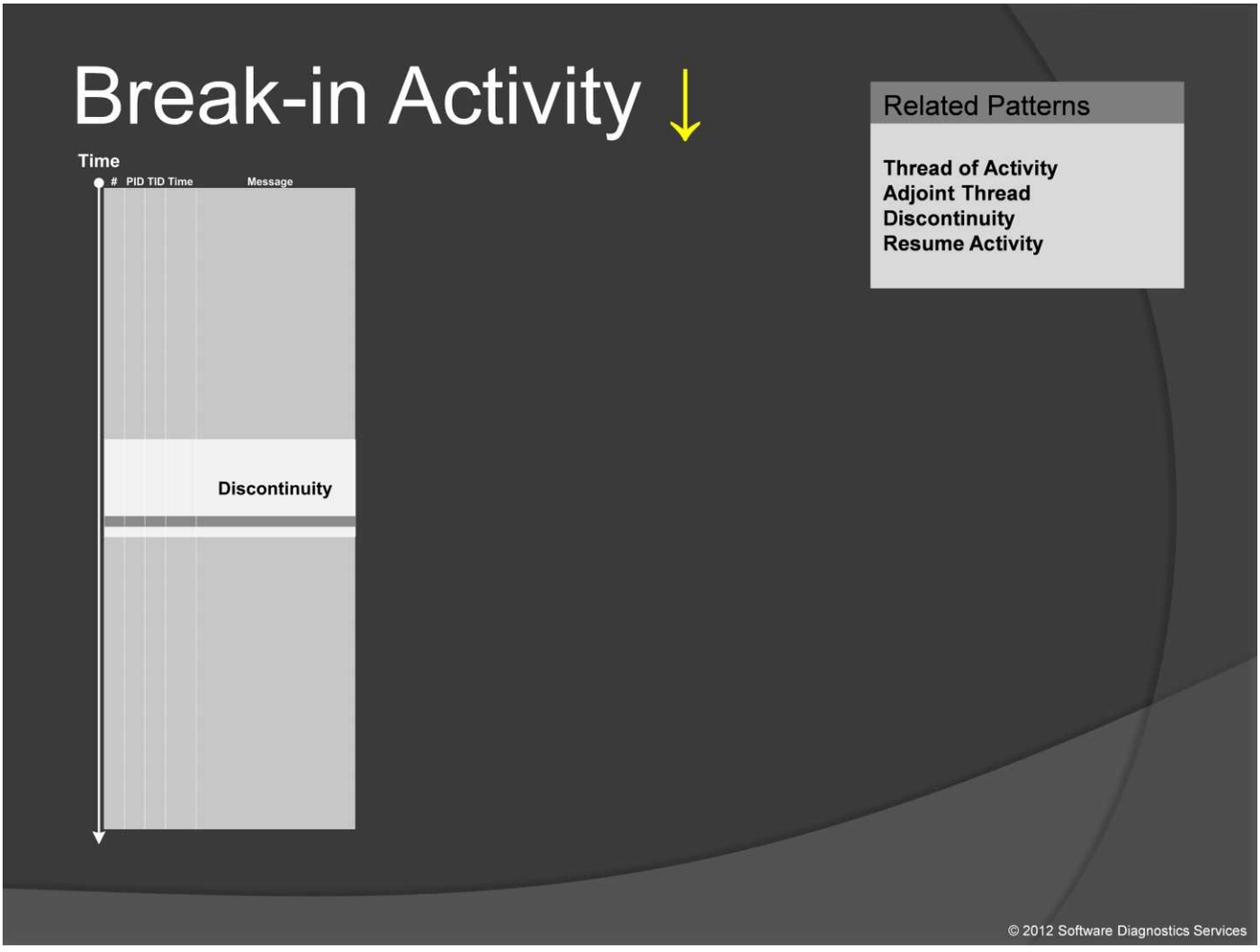
```
#     Module PID  TID  Time         File    Function  Message
-------------------------------------------------------------------------------------------------
6060  dllA   1604 7108 10:06:21.746 fileA.c DllMain   DLL_PROCESS_ATTACH
24480 dllA   1604 7108 10:06:32.262 fileA.c LaunchApp Exec Path: C:\Program Files\CompanyA\appB.exe

30 seconds of discontinuity till the end of full trace
```

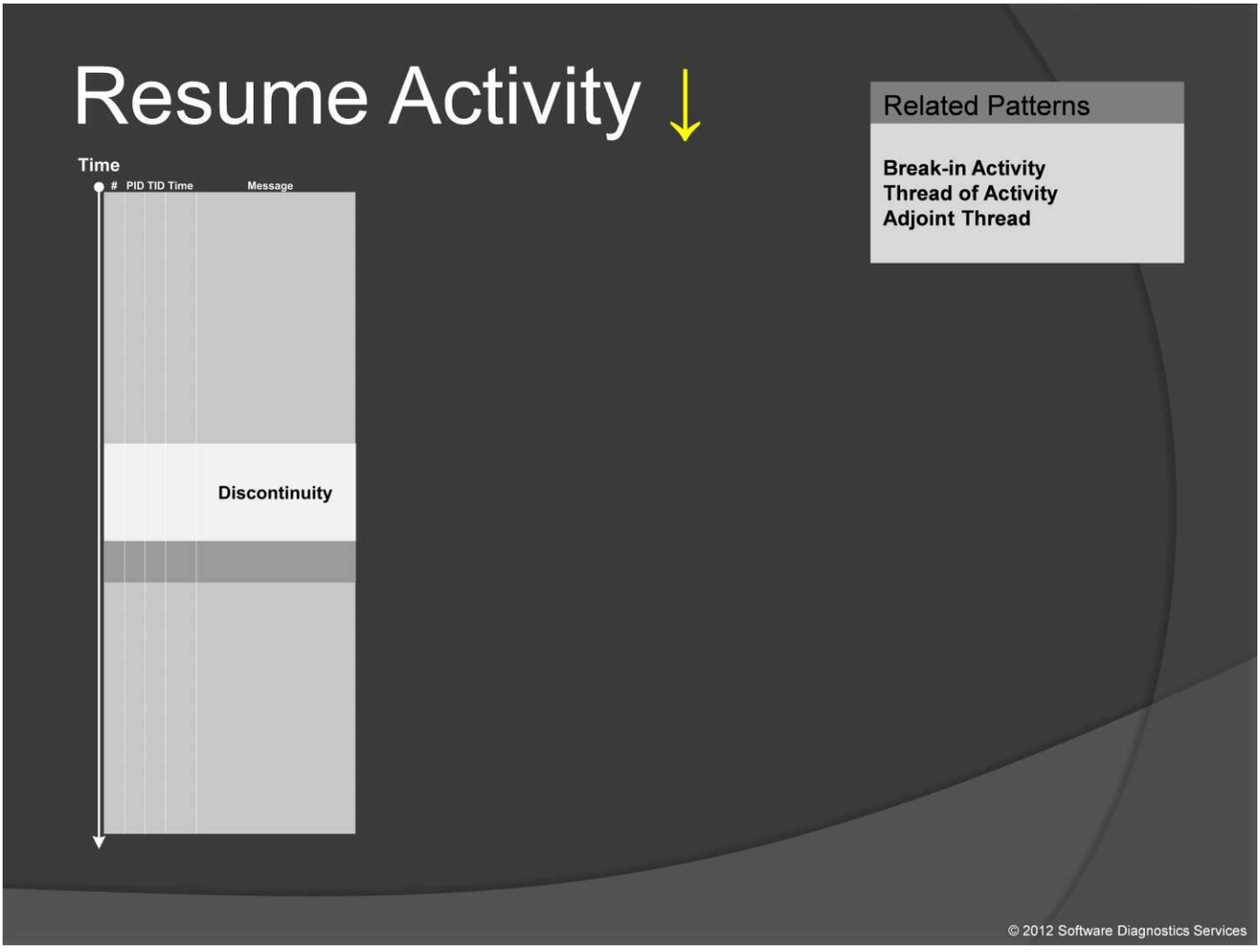

Time Delta pattern is closely related to **Discontinuity**. This is a time interval between **Significant Events** we are interested in or just some found delay as the example in this picture. In the trace fragment, we are interested in *dllA* activity from its load until it launches *appB.exe*. We see that the time delta was only 10 seconds. But after launch, there is another 30 seconds delay until the tracing was stopped. When troubleshooting delays their time should be included in **Basic Facts** (or supporting information) accompanying software traces and logs.



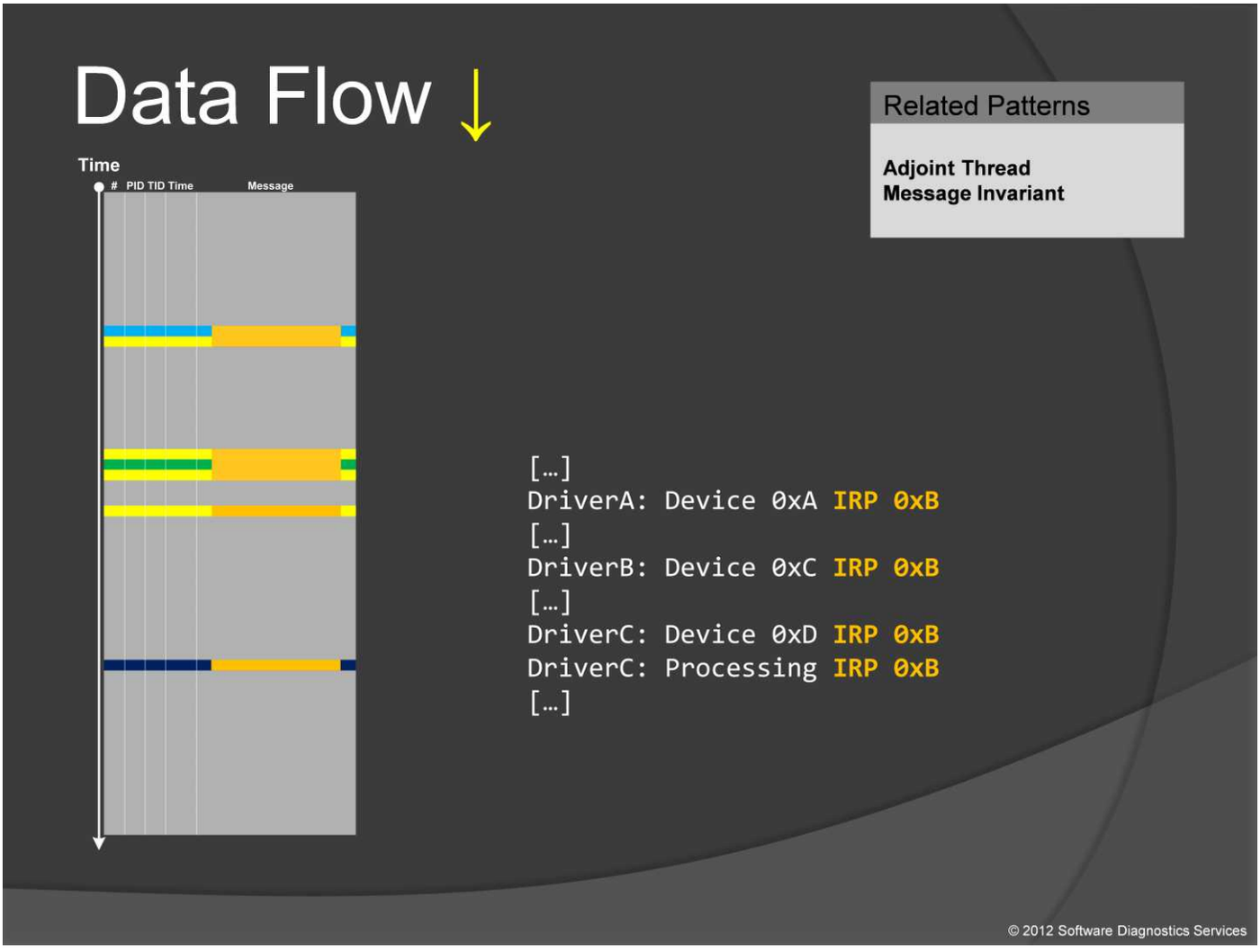
Adjoint Thread invariants that we name as ATIDs (like TIDs for threads) can be reused giving rise to software traces where two separate execution entities (different ATIDs such as 2 and 3 on this picture) are glued together in one trace. I've actually seen PID reuse, at least, two times and don't really know whether this was a bug in a tool because I thought PIDs are unique and are not reused. But in the case of other OS and adjoint thread invariants which can be reused, I still leave this pattern as is and abstract it for ATIDs. Other similar examples might include different instances of modules sharing the same name, source code or even, in general, periodic tracing sessions appended to the end of the same trace file although we think that the latter should be a separate pattern.



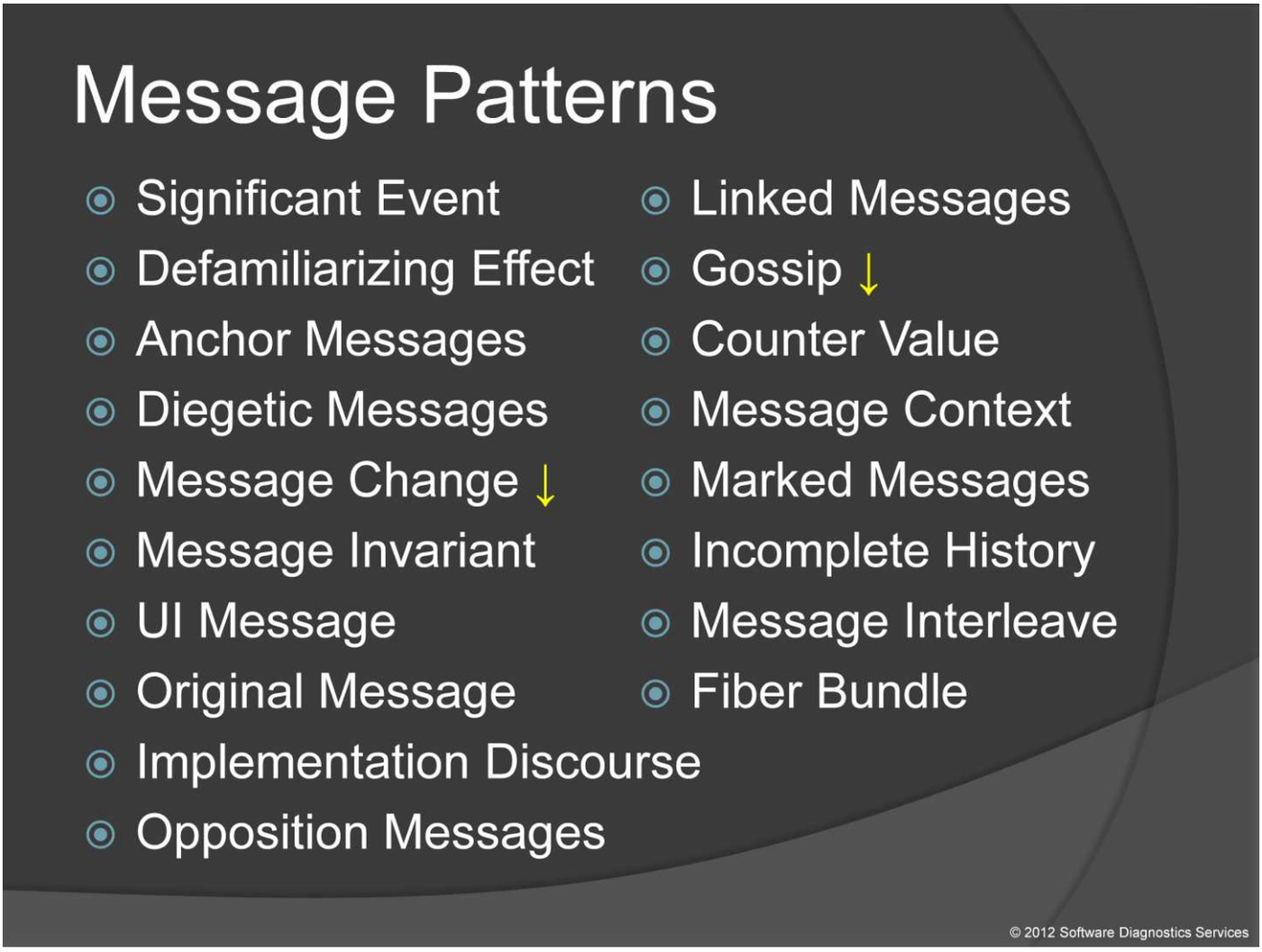
This pattern covers a message or a set of messages that surfaces just before the end of **Discontinuity** of a **Thread of Activity** or **Adjoint Thread** and possibly triggered Activity continuation (**Resume Activity** pattern we cover next).



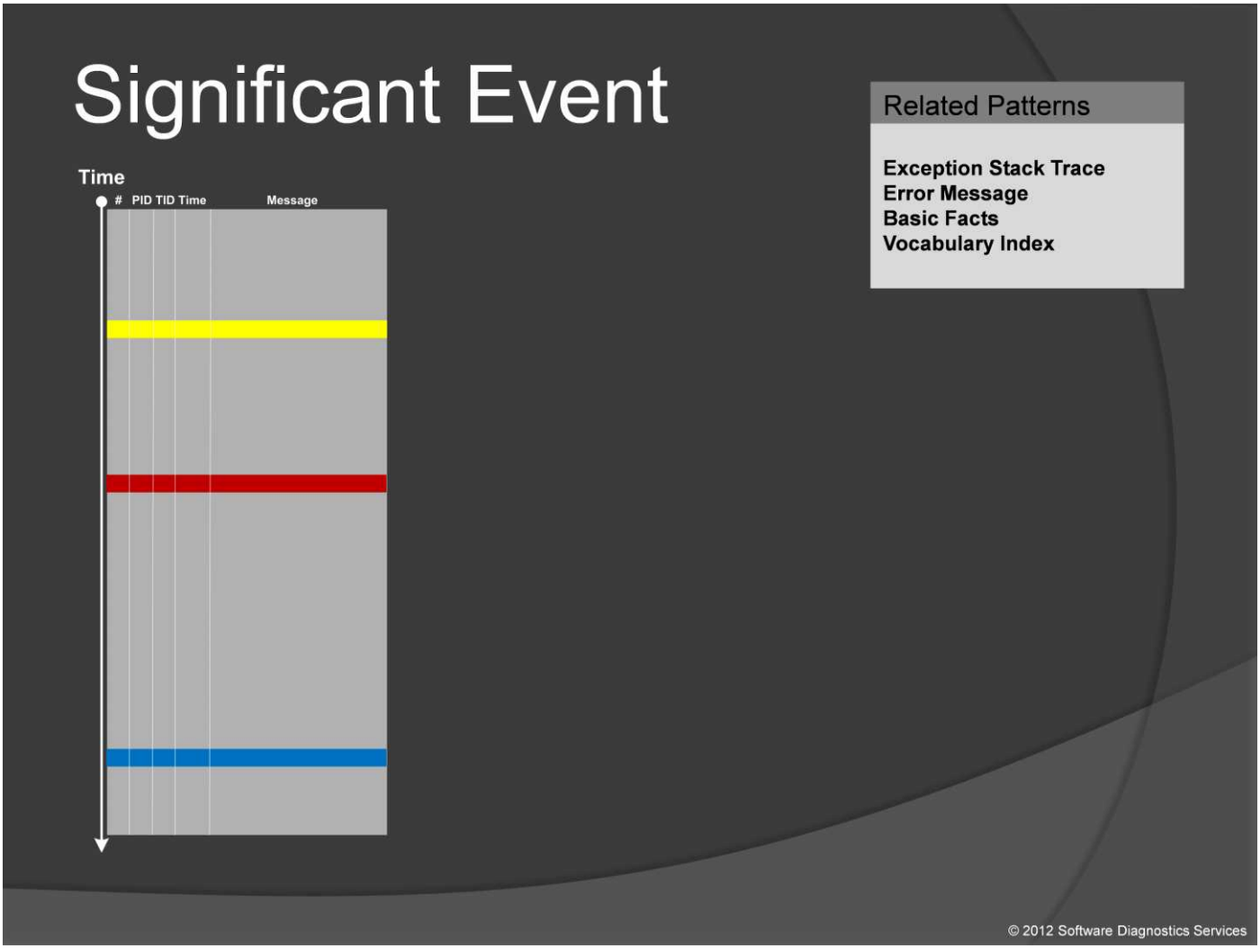
If **Break-in Activity** pattern we covered previously may be unrelated to a **Thread of Activity** or an **Adjoint Thread** which has a **Discontinuity** then **Resume Activity** pattern highlights messages from that thread after discontinuity.



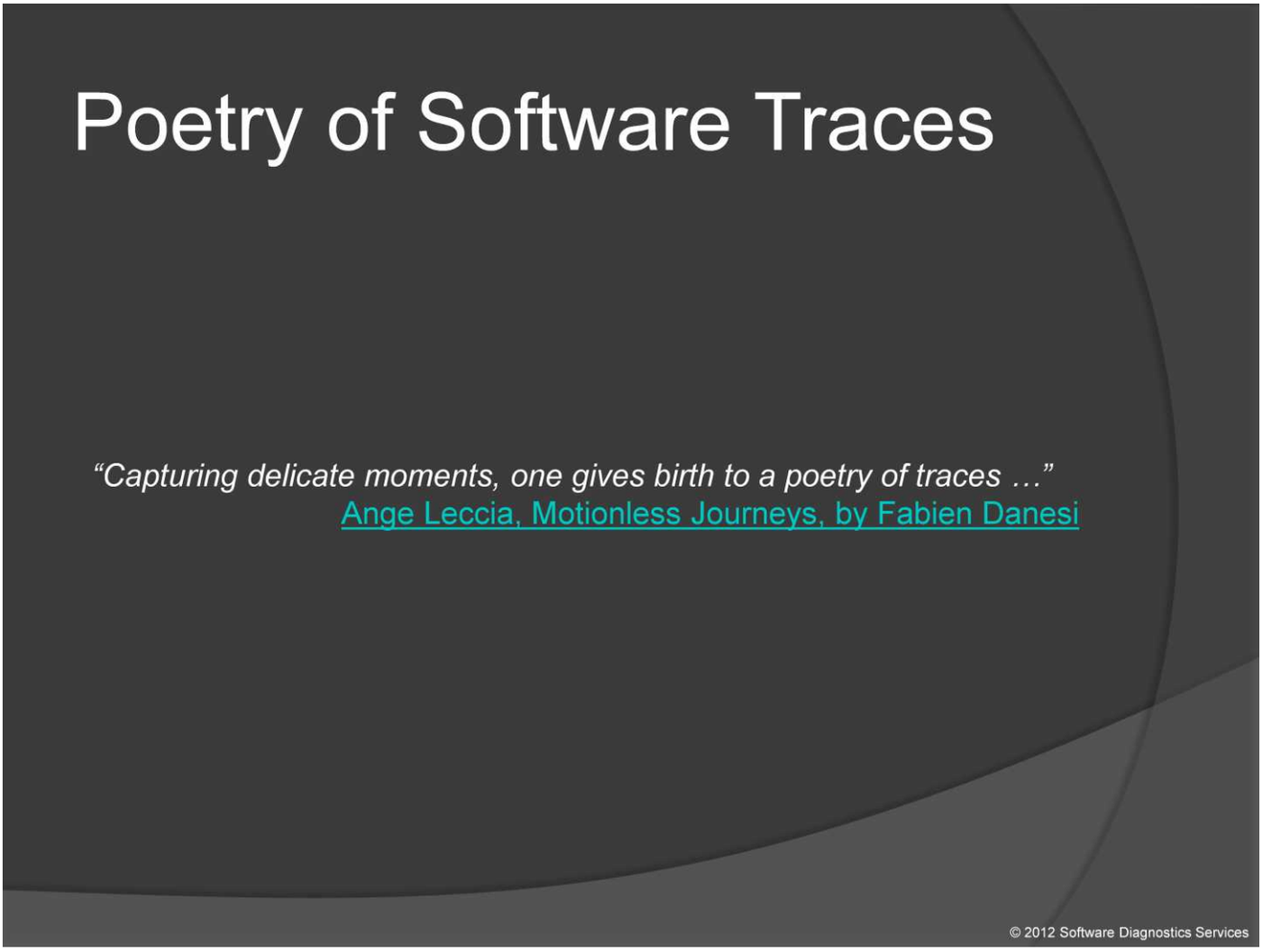
If trace messages contain some character or formatted data that is passed from module to module or between threads and processes it is possible to trace that data and form a **Data Flow** thread similar to an **Adjoint Thread of Activity** we have when we filter by a specific **Message Invariant**. However, for **Data Flow,** we may have completely different message types. Here I illustrate **Data Flow** by a hypothetical driver communication case where the same IRP (I/O Request Packet) is passed between devices. Note that by **Data Flow** we mean any data which can be just an error or exception propagating through.



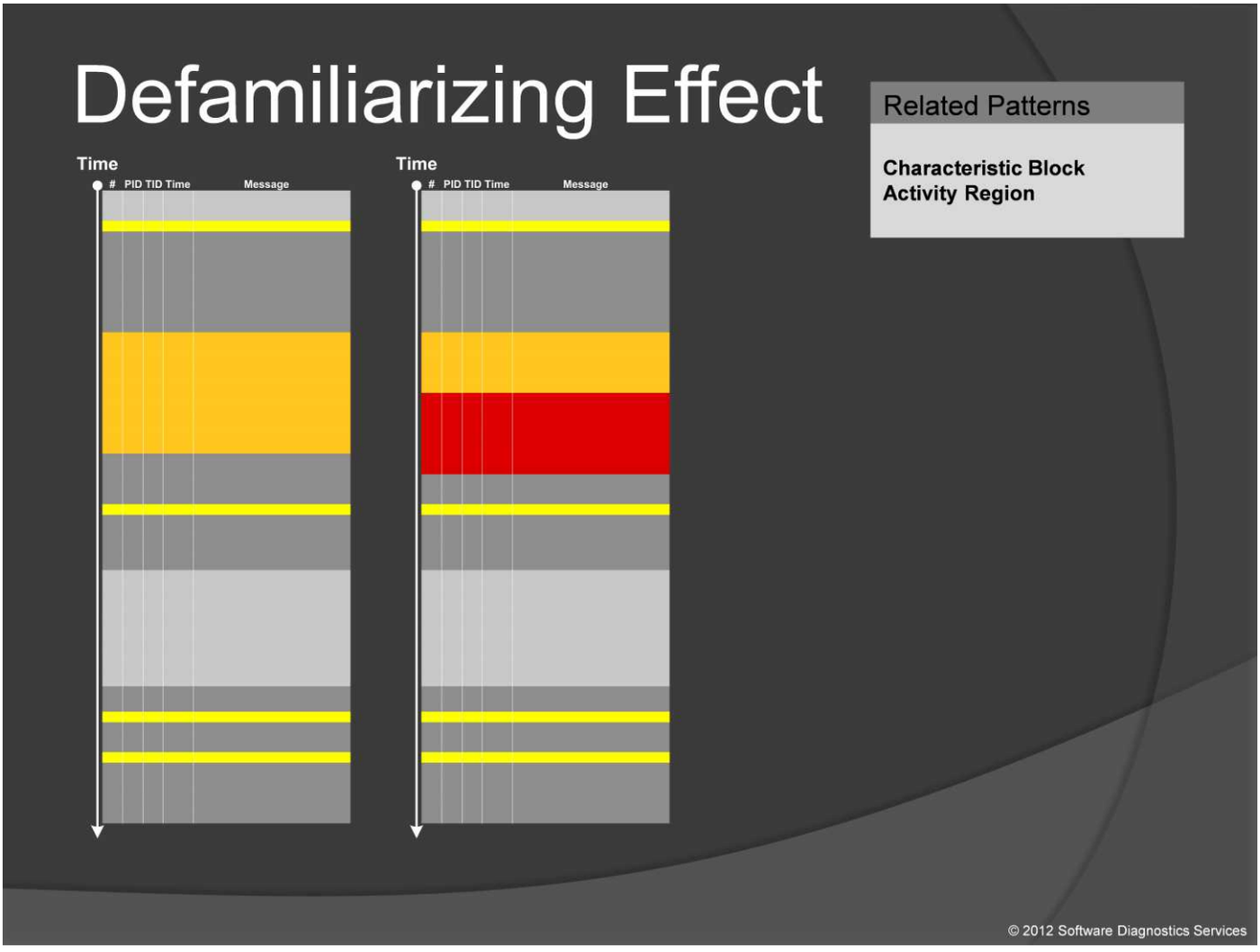
The sixth block of patterns we cover are message patterns or patterns at the level of an individual message.



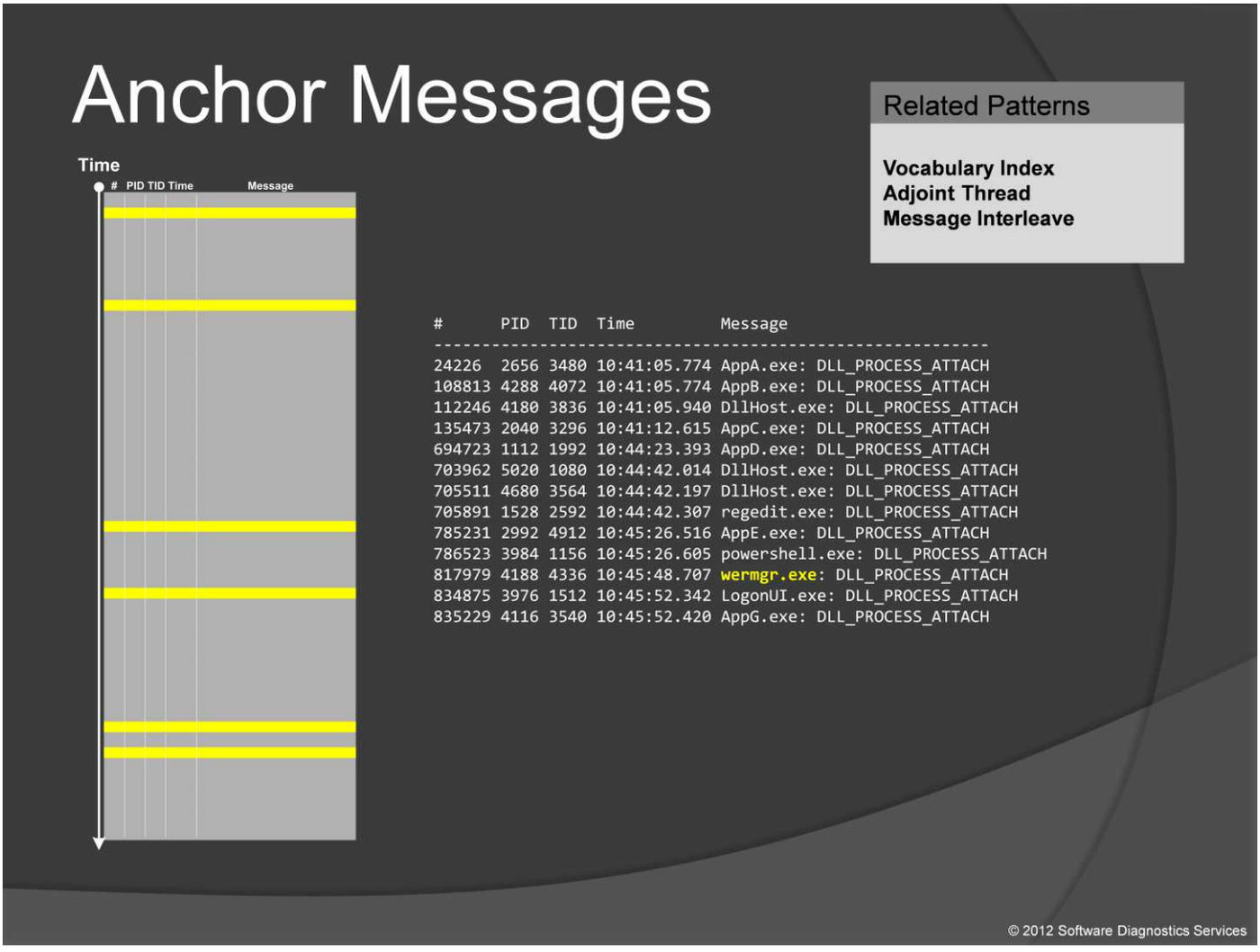
When looking at software traces and logs and doing either a search for or just scrolling certain messages have our attention immediately. We call them **Significant Events**. It could be a recorded **Exception Stack Trace** or an **Error Message**, a **Basic Fact**, a trace message from **Vocabulary Index**, or just any trace message that marks the start of some activity we want to explore in depth, for example, a DLL is loaded into a process space, a process is started, or a certain function is called. The start of a trace and the end of it are trivial significant events and are used in deciding whether the trace is **Circular**, and also in determining the trace recording interval (**Time Delta** pattern) or its average **Message Current**.



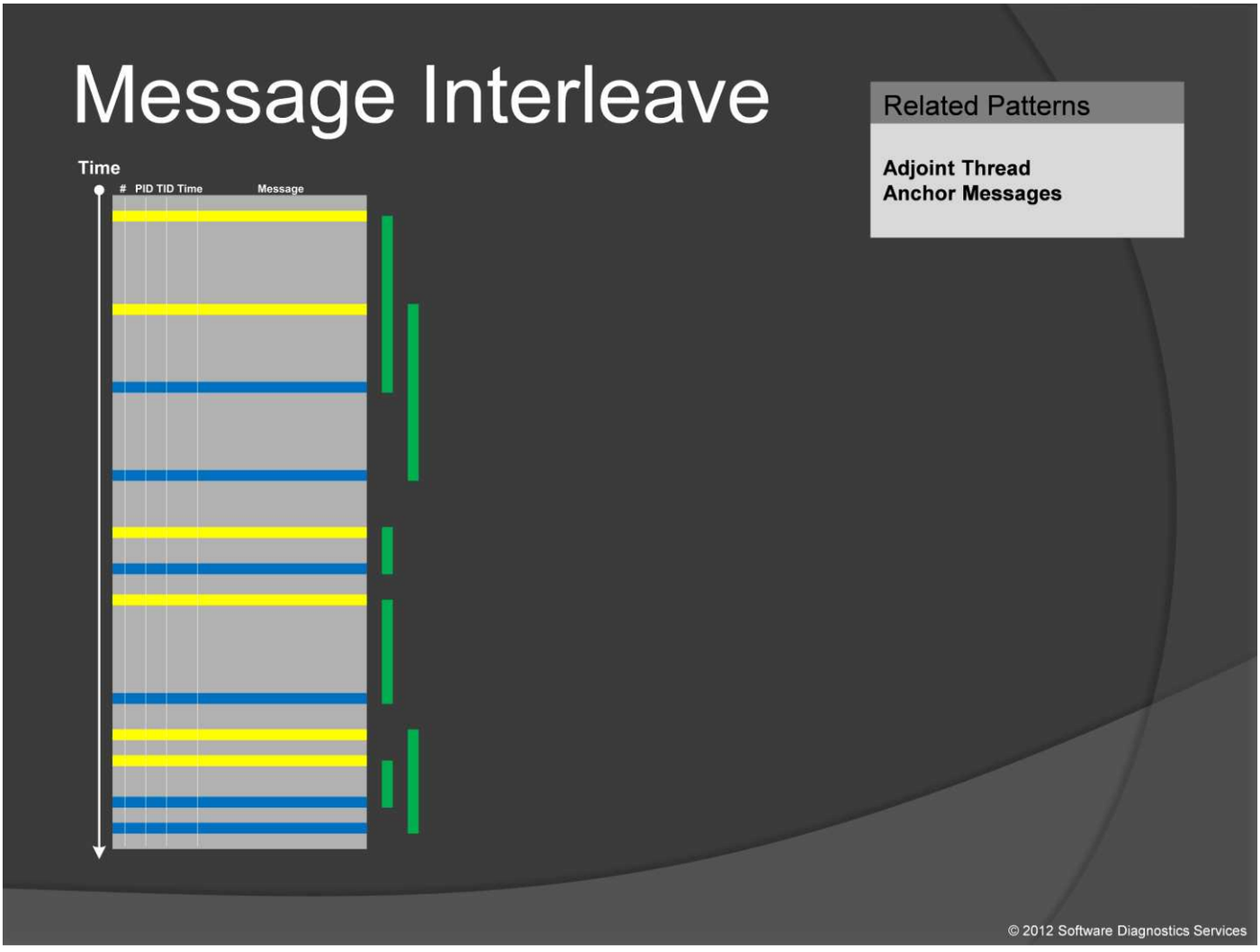
A bit of poetry before we continue with further patterns.
From http://www.plpfilmmakers.com/motionless-journeys



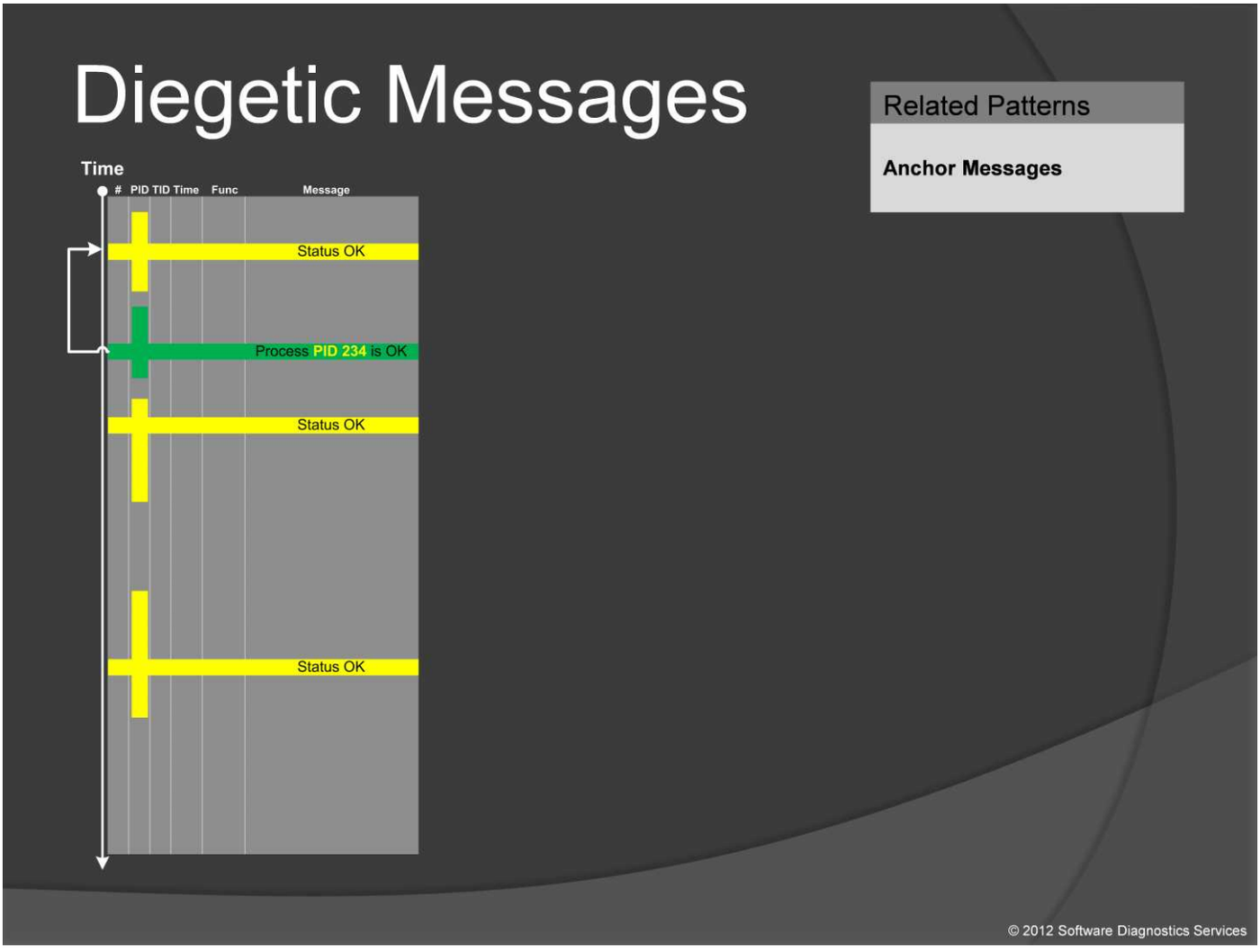
Here like in poetry we see sudden unfamiliar trace statements across the familiar landscape of **Characteristic Blocks** and **Activity Regions**. On the left, we see familiar traces and on the right a new trace from a problem system.



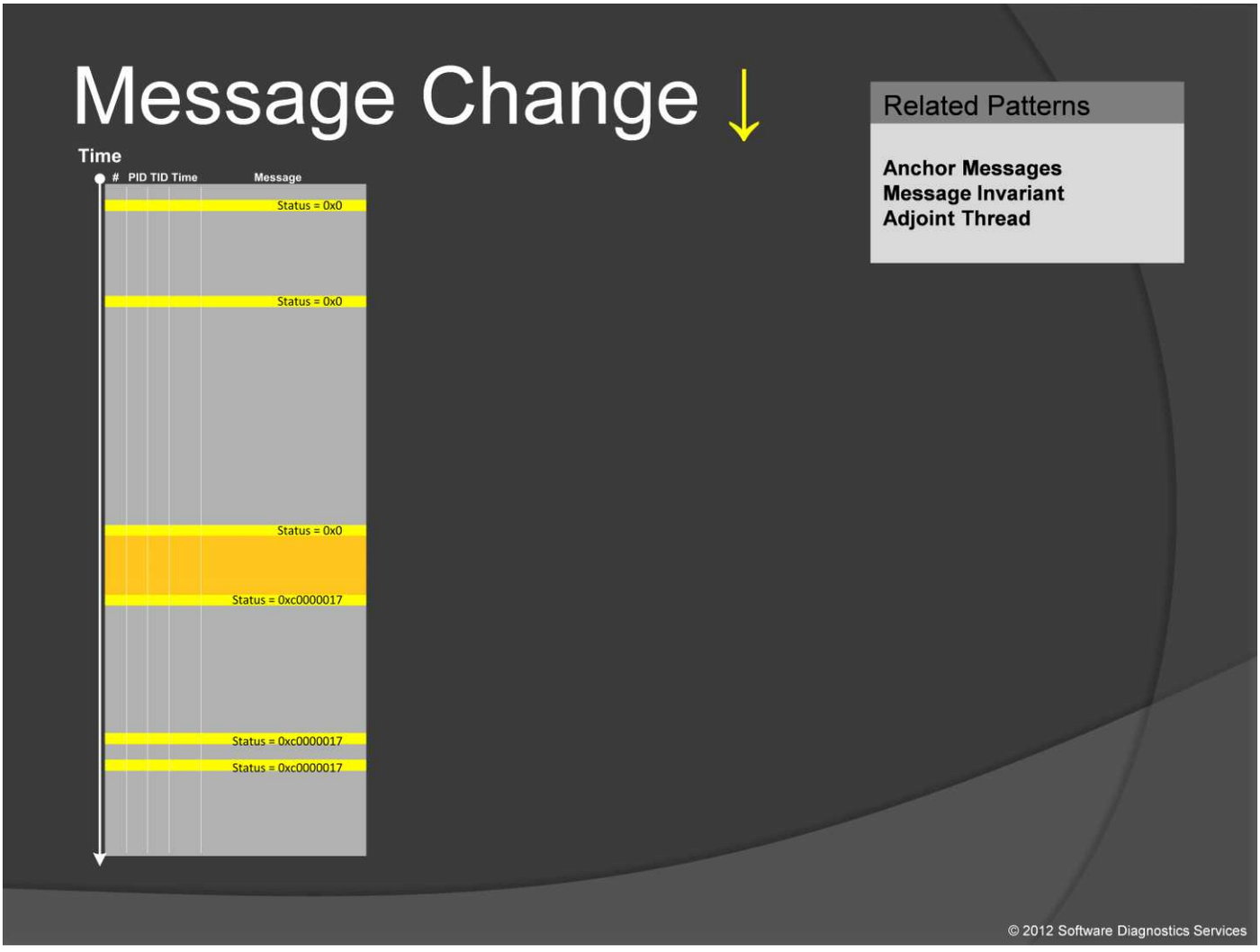
When a software trace is very long, it is useful to partition it into several regions based on a sequence of **Anchor Messages**. The choice of them can be determined by a **Vocabulary Index** or an **Adjoint Thread of Activity**. For example, an ETW trace with almost 1,000,000 messages recorded during a remote desktop connection for 6 minutes can be split into 14 segments by the adjoint thread of DLL_PROCESS_ATTACH message. Then each region can be analyzed independently for any anomalies, for example, for the answer to the question why *wermgr.exe* was launched.



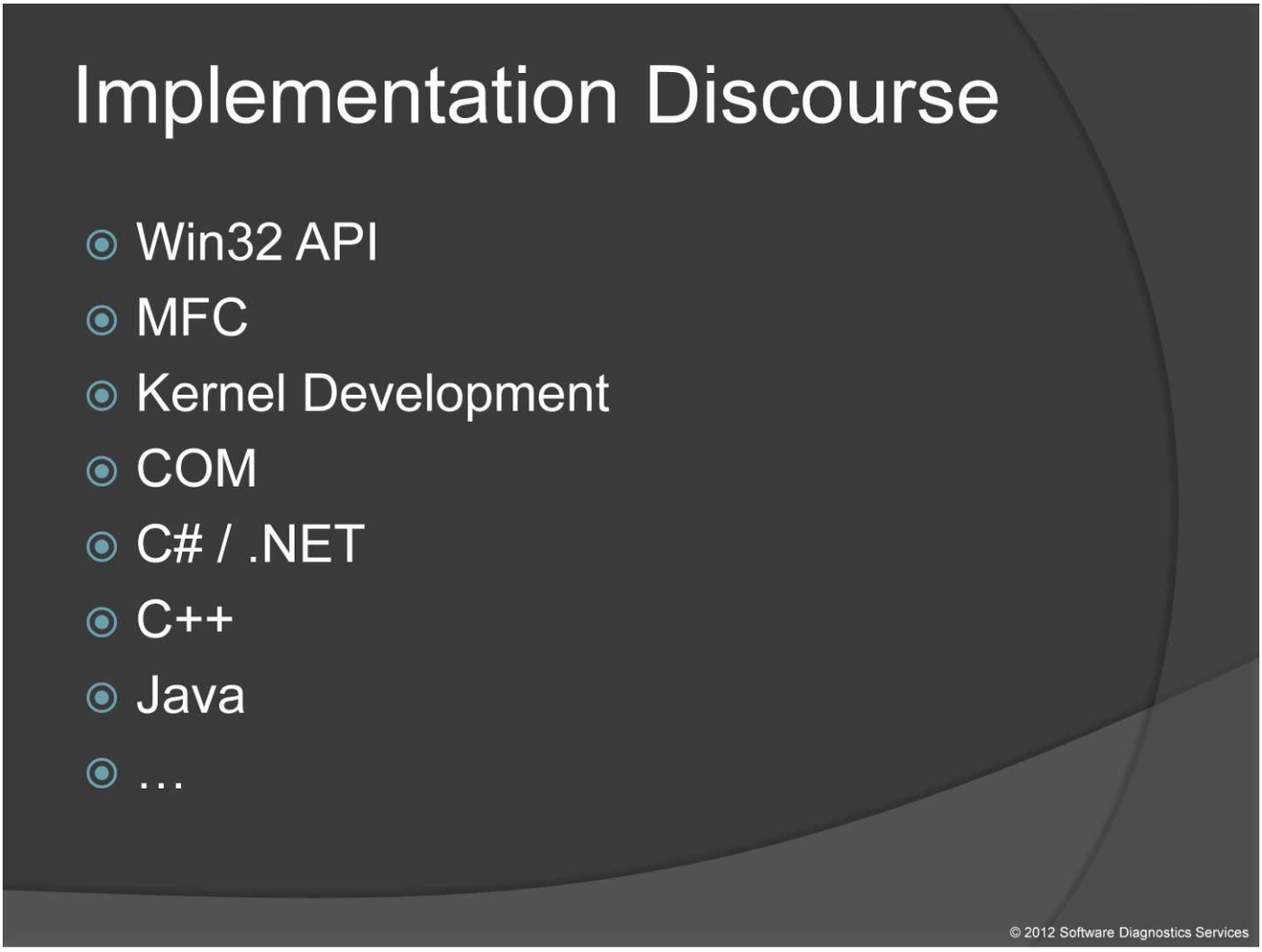
This pattern addresses several trace segmentations by interleaving regions of one set of **Anchor Messages** with another set of **Anchor Messages**. Here, on this picture, we interleave an **Adjoint Thread** of DLL_PROCESS_ATTACH messages with the adjoint thread of DLL_PROCESS_DETACH messages. Another example could be “open” and “close” or “load” and “unload”, etc.



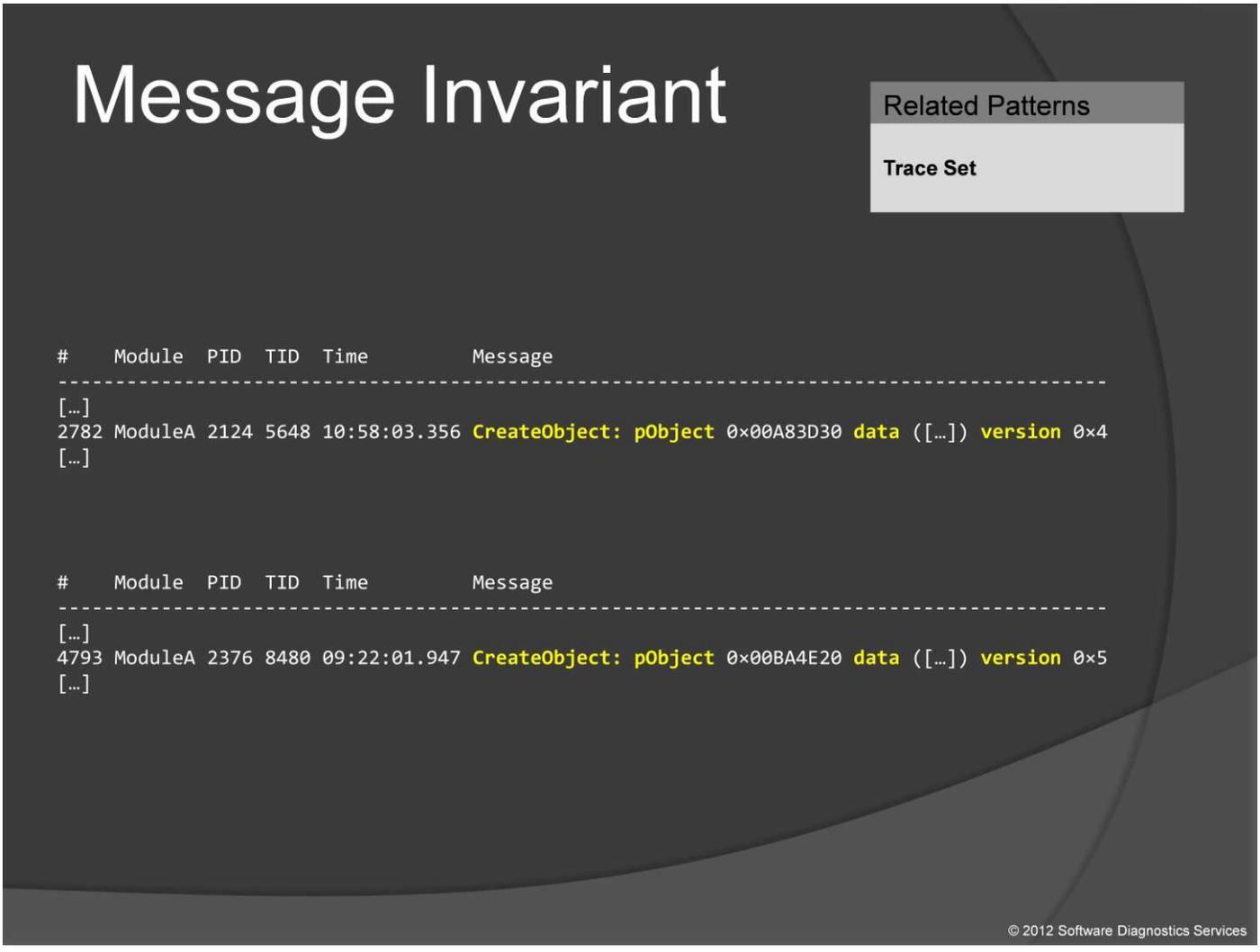
In general, we have processes or modules that trace themselves and processes or modules that query about other processes, components and subsystems. In the picture, you see the difference between diegetic (in green color) and non-diegetic trace messages (in yellow color) for PIDs. Some modules may emit messages that tell about their status but from their message text, we know the larger computational story. A typical example here is a session initialization process startup sequence in terminal services.



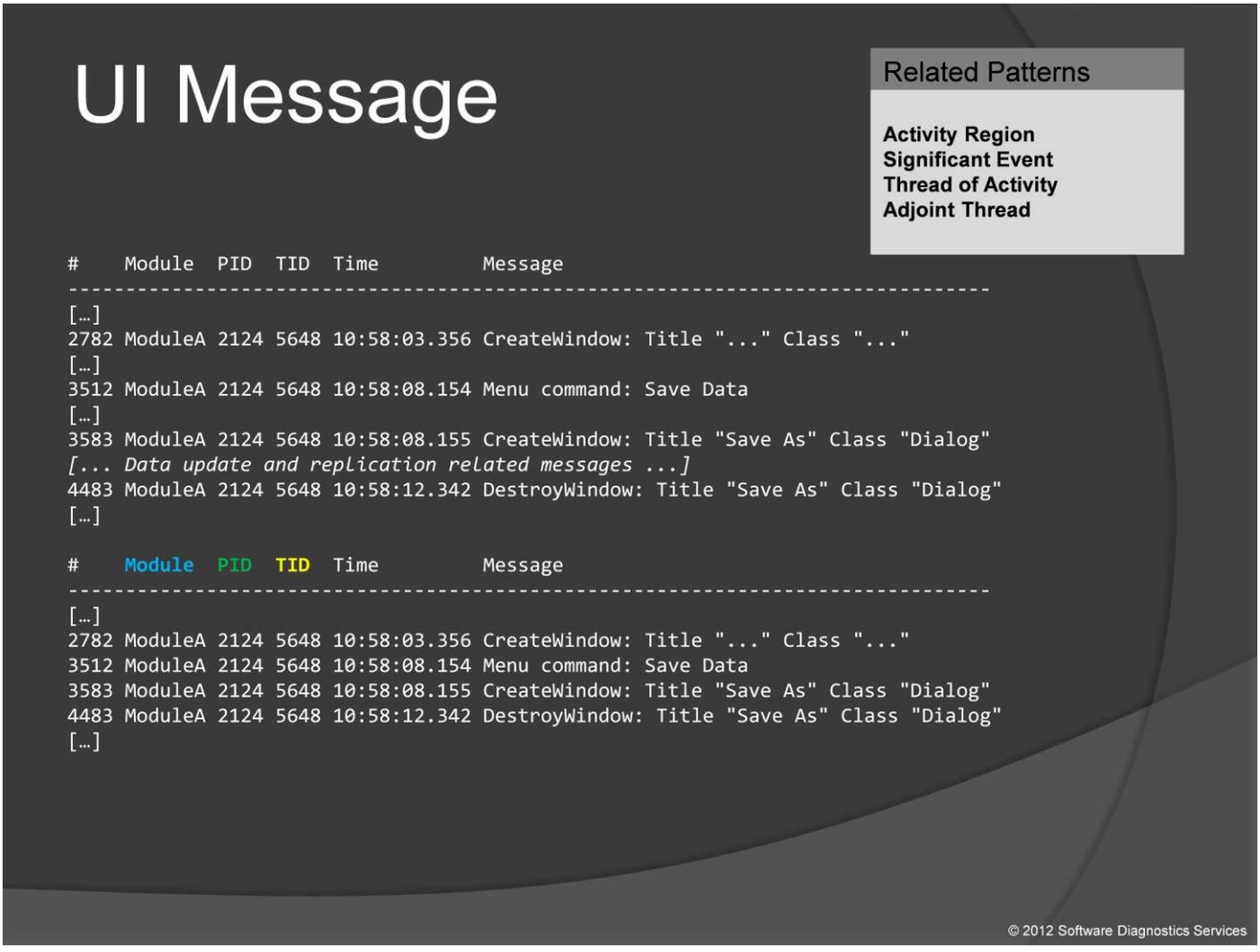
Often when we find an **Anchor Message** related to our problem description which has some variable part such as status or progress report we are interested in its evolution throughout a software trace (by creating an **Adjoint Thread** for this **Message Invariant** or simply by a search filter). And then we can check messages between changes.



If we look at any non-trivial trace, we would see different **Implementation Discourses**. Modules are written using different languages and adhere to different runtime environments, binary models, and interface frameworks. All these implementation variations influence the structure, syntax and semantics of trace messages. For example, .NET debugging traces differ from file system driver or COM debugging messages.



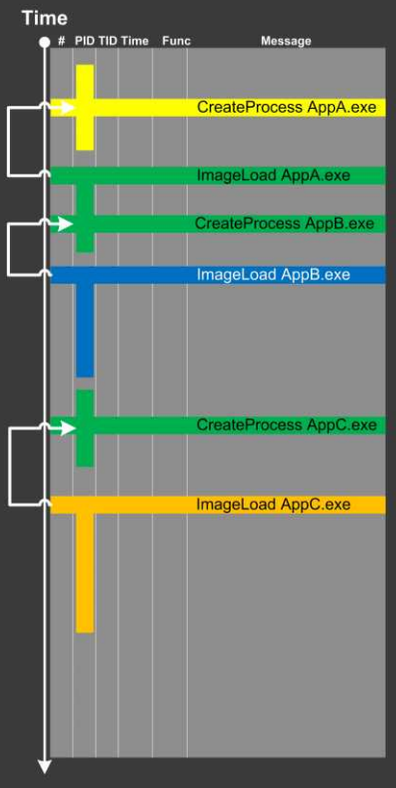
We already encountered **Message Invariants** before in the context of a single software trace. Here we illustrate it for a **Trace Set** of working (normal) and non-working (abnormal) software traces. Recall that most of the time software trace and log messages coming from the same source code fragment (the so-called **PLOT**, Program Lines of Trace) contain invariant parts such as function and variable names, descriptions, and mutable parts such as pointer values and error codes. In a comparative analysis of several trace files, we are often interested in message differences. For example, in one troubleshooting scenario, certain objects were not created correctly for one user. We suspected a different object version was linked to a user profile. Separate application debug traces were recorded for each user, and we could see version 0×4 for the problem user and 0×5 for all other normal users.



This pattern is very useful for troubleshooting system-wide issues because we can map visual behavior to various **Activity Regions** and consider such messages as **Significant Events**. Filtering by TID, we can create a **Thread of Activity** and filtering by module or PID we can create an **Adjoint Thread of Activity**.

This pattern deals with software trace messages where certain **Message Invariant** is repeated several times, but only the first message occurrence has significance for software trace analysis. One such example is shown here for library load events. We form an **Adjoint Thread** and look for ".dll" messages. We are interested in the first occurrence of a specific name because in our troubleshooting context we need to know the time it was loaded first. This pattern is called **Original Message** and not **First Message** because in other contexts and cases the message wording and data might not be the same for the first and subsequent messages.

Linked Messages



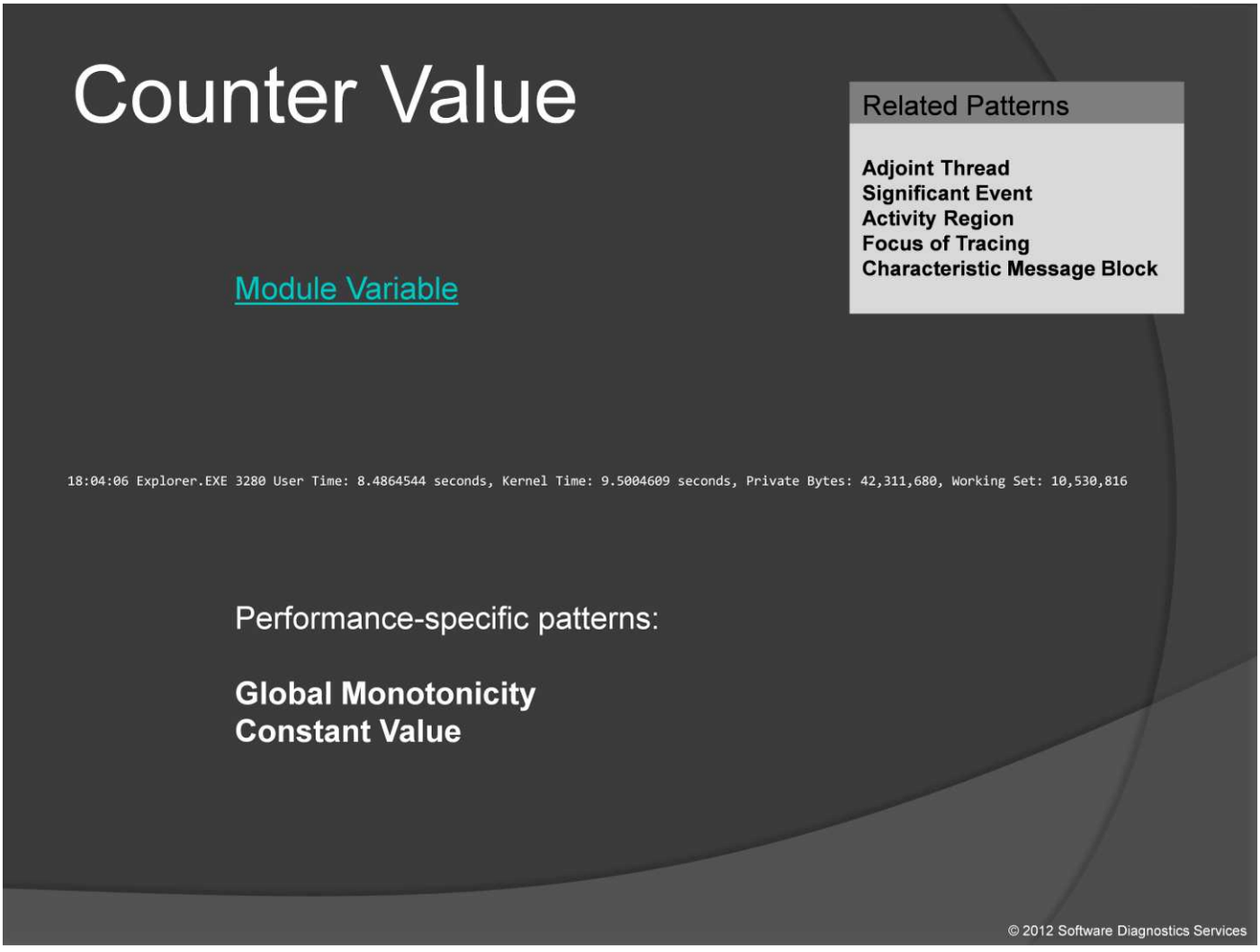
Related Patterns

Adjoint Thread

```
#      PID  Message
-----------------------------------------
[…]
128762 1260 CreateProcess: PPID 1260 PID 6356
[…]
128785 6356 ImageLoad: AppA.exe PID 6356
[…]
131137 6356 CreateProcess: PPID 6356 PID 6280
[…]
131239 6280 ImageLoad: AppB.exe PID 6280
[…]
132899 6356 CreateProcess: PPID 6356 PID 8144
[…]
132906 8144 ImageLoad: AppC.exe PID 8144
[…]
```

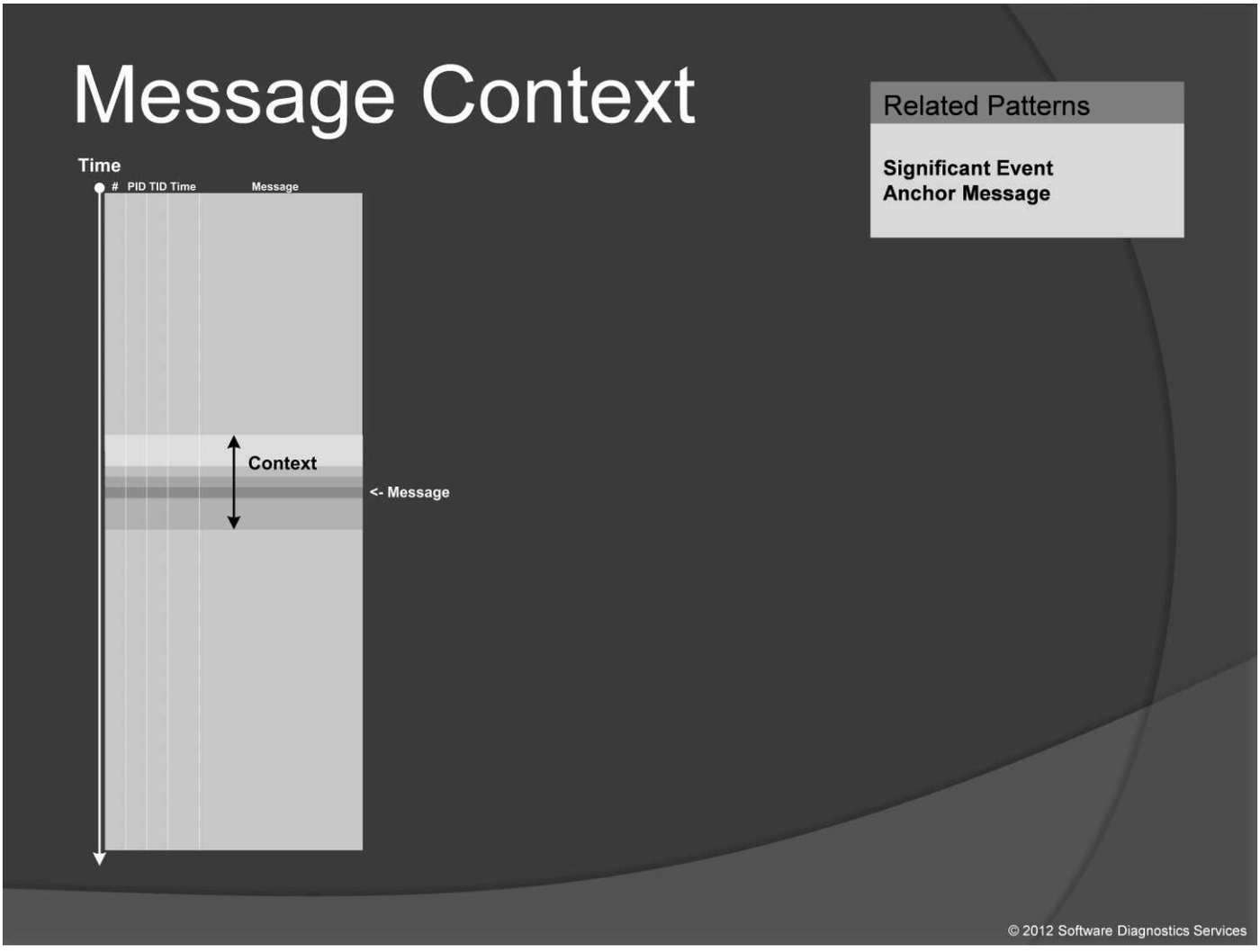

Sometimes we have the so called **Linked Messages** through some common parameter or attribute. One such example is illustrated in this slide, and it is related to kernel process creation notifications. Here we got **Adjoint Thread** for a module that intercepts such events (not shown on trace example for visual clarity). We see messages linked through PID and PPID (Parent PID) parameter relationship.

This pattern has a funny name. It is called **Gossip** instead of **Duplicated Message** to allow the possibility of syntax and semantics of the same message to be distorted in subsequent trace messages from different **Adjoint Threads**. On this slide in the first top example, you see a distortion free example of the same message content seen in different modules. To make analysis easier, when constructing an **Event Sequence Order** or **Message Interleave** it is recommended to choose messages from one source instead of mixing events from different sources as in the second bottom example on this slide.

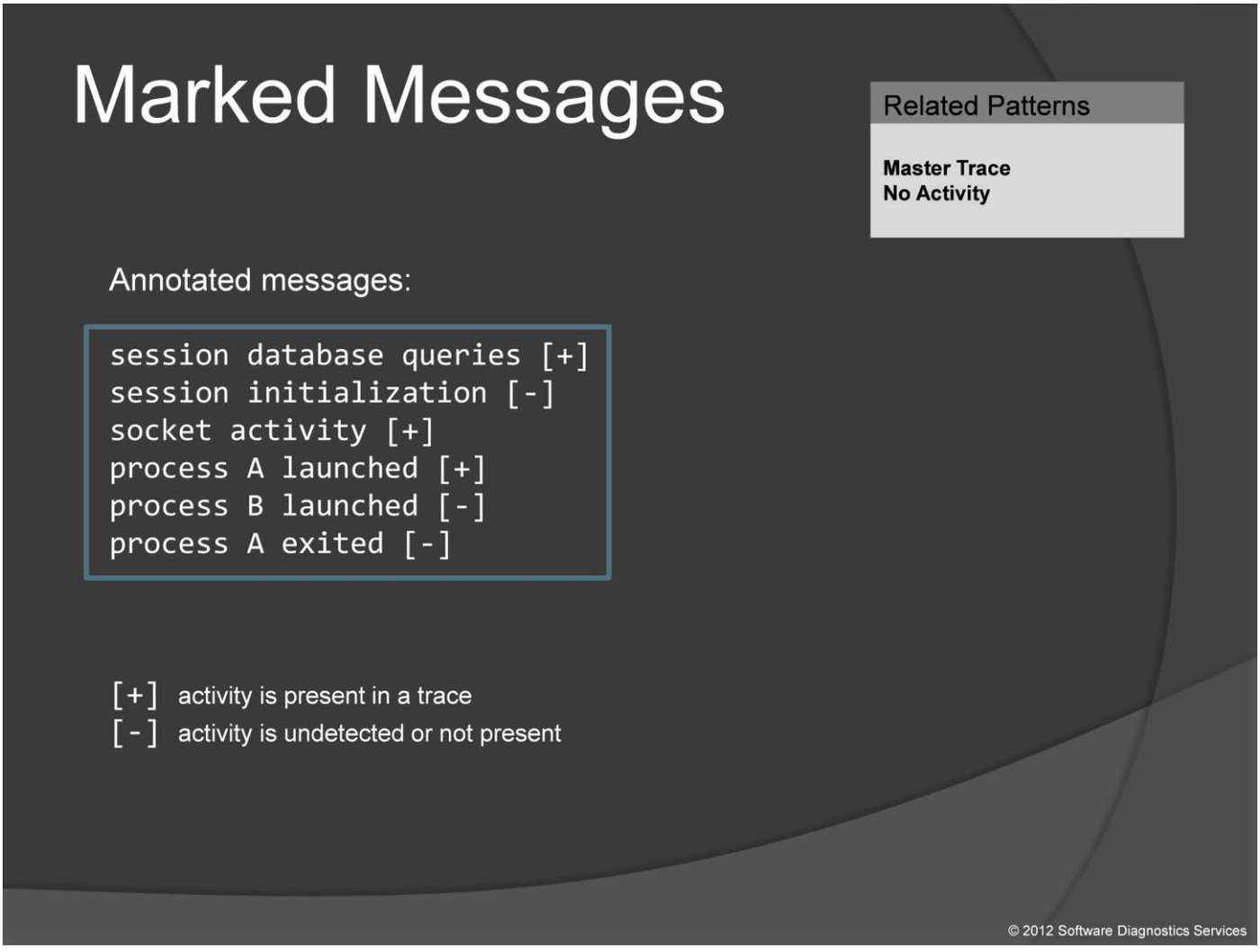


This pattern covers performance monitoring and its logs. A **Counter Value** is some variable in memory, for example, a **Module Variable** from memory dump analysis patterns, that is updated periodically to reflect some aspect of the state. It can also be a derivative variable calculated from different such real variables and presented in trace messages. Here, I provide an example of profile messages from Process Monitor (edited a bit for space). Profile events might be filtered off by default. The sequence of profile events is a software trace itself so all other trace analysis patterns such as **Adjoint Thread** (different colors on graphs), **Focus of Tracing**, **Characteristic Message Block** (for graphs), **Activity Region**, **Significant Event**, and others can be applicable here. Beside that there are also **Performance**-specific patterns such as **Global Monotonicity** and **Constant Value** but they are beyond the scope of this training.

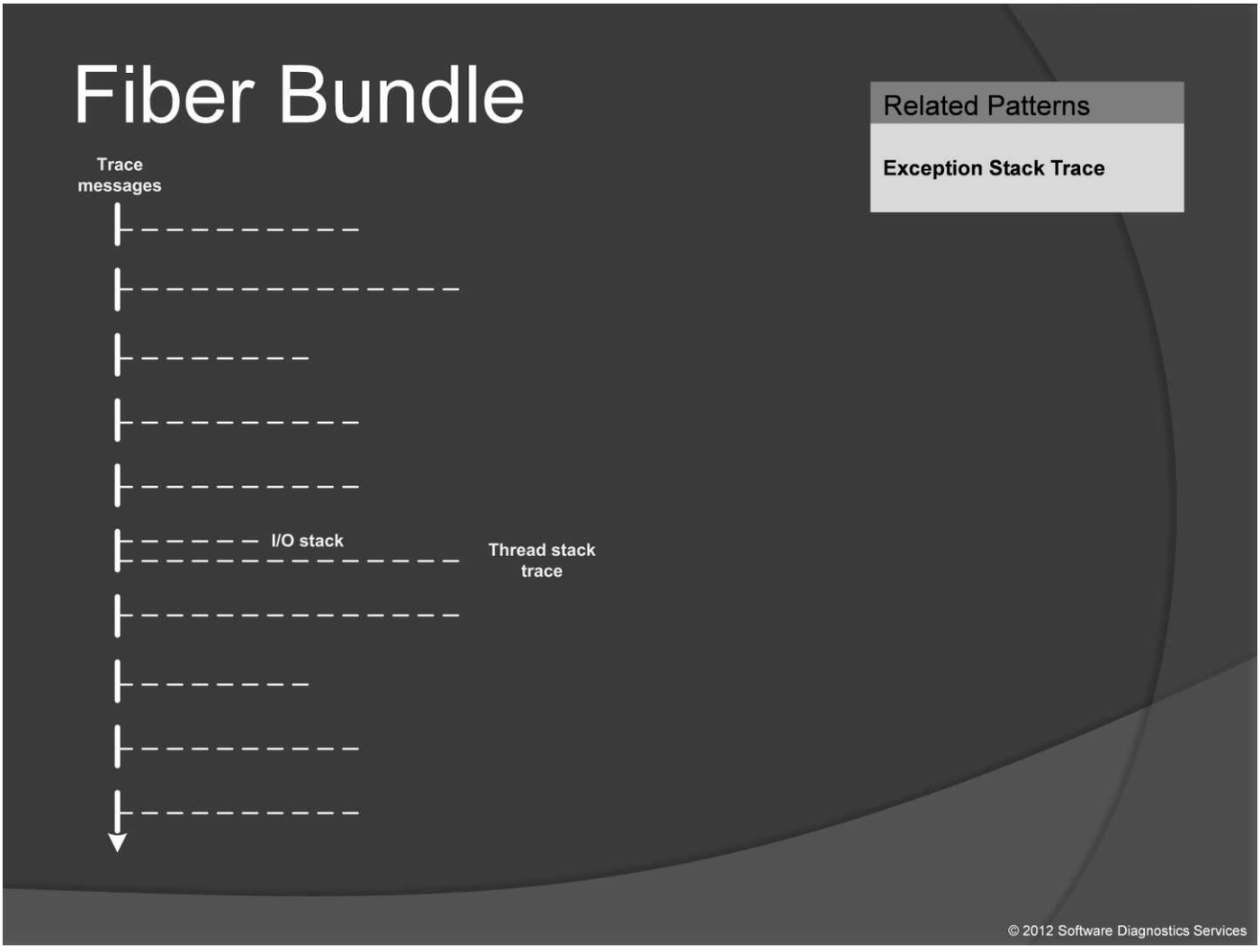
Module Variable: Memory Dump Analysis Anthology, Volume 7, page 98
http://www.dumpanalysis.org/blog/index.php/2011/12/03/crash-dump-analysis-patterns-part-157/



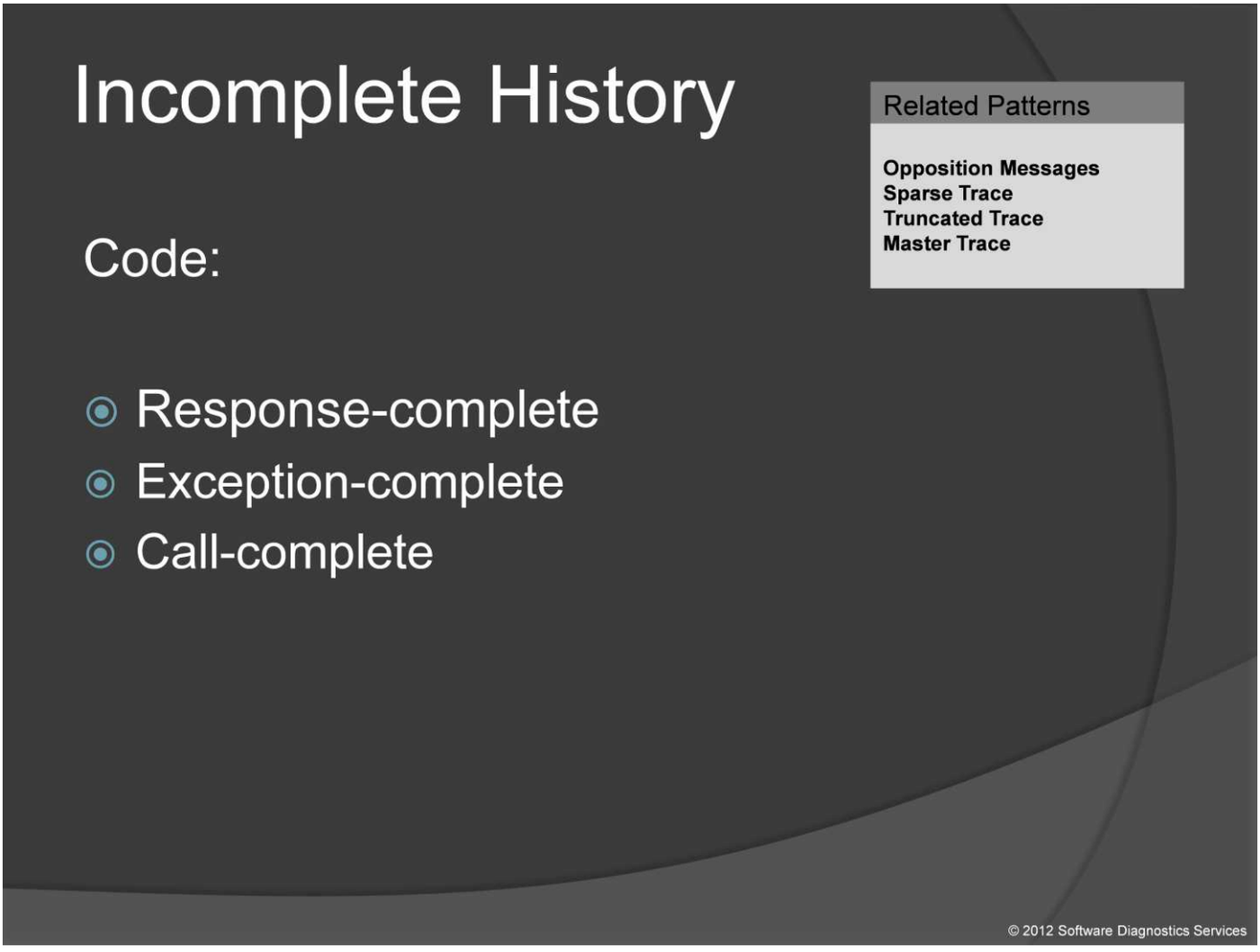
Most of the time we analyze not isolated messages but in a surrounding **Message Context,** which is a set of messages having some relation to the chosen message and usually found in the message stream in some close proximity.



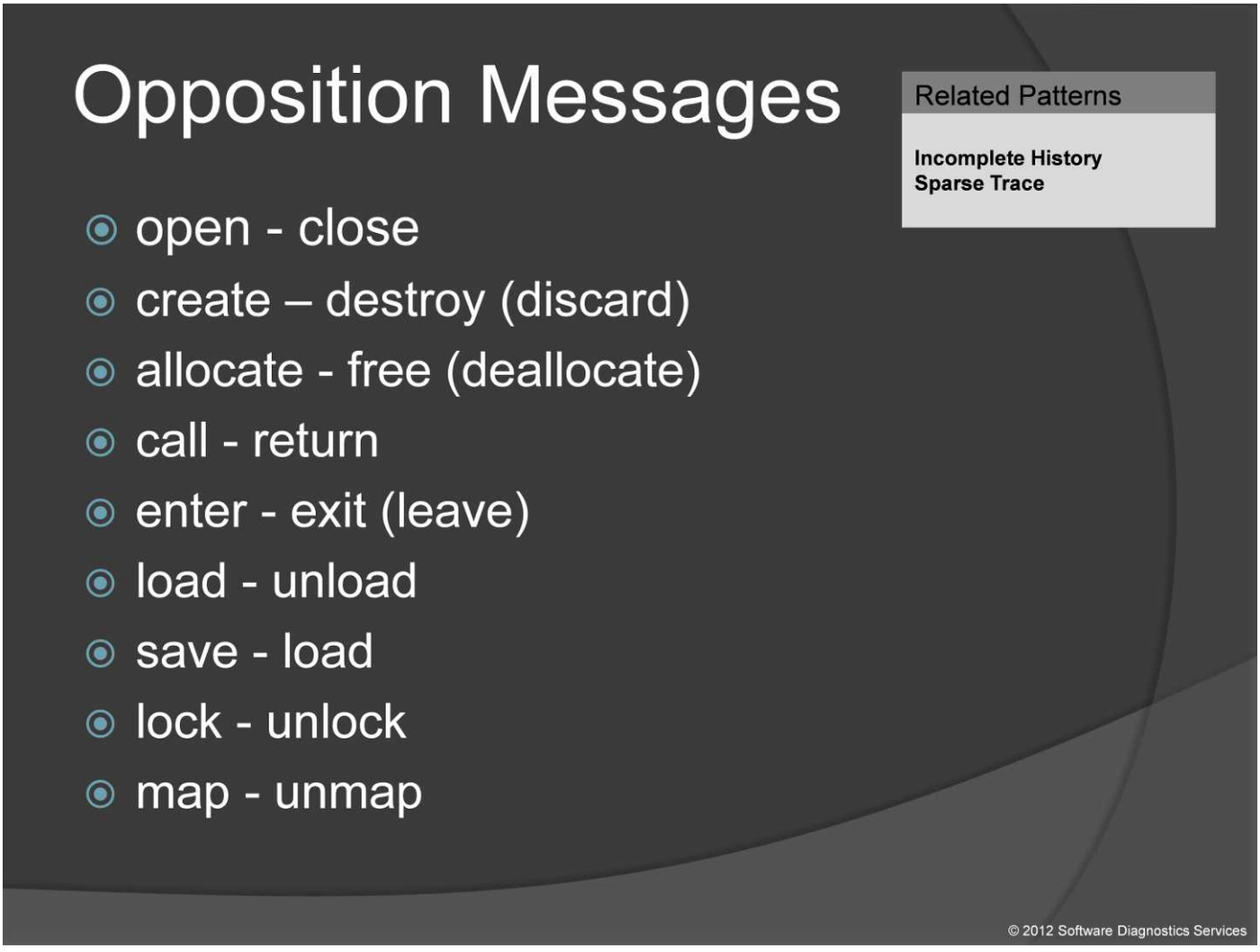
This pattern groups trace messages based on having some feature or property. For example, marked messages may point to some domain of software activity such as related to functional requirements and, therefore, may help in troubleshooting and debugging. Unmarked messages include all other messages that don't say anything about such activities (although may include messages pointing to such activities indirectly we are unaware of) or messages that say explicitly that no such activity has occurred. We can annotate any trace or log after analysis to compare it with a **Master Trace** pattern (which is a normally expected trace corresponding to functional requirements). Sometimes a non-present activity can be a marked activity corresponding to all inclusive unmarked present activity (for example, **No Activity** pattern).



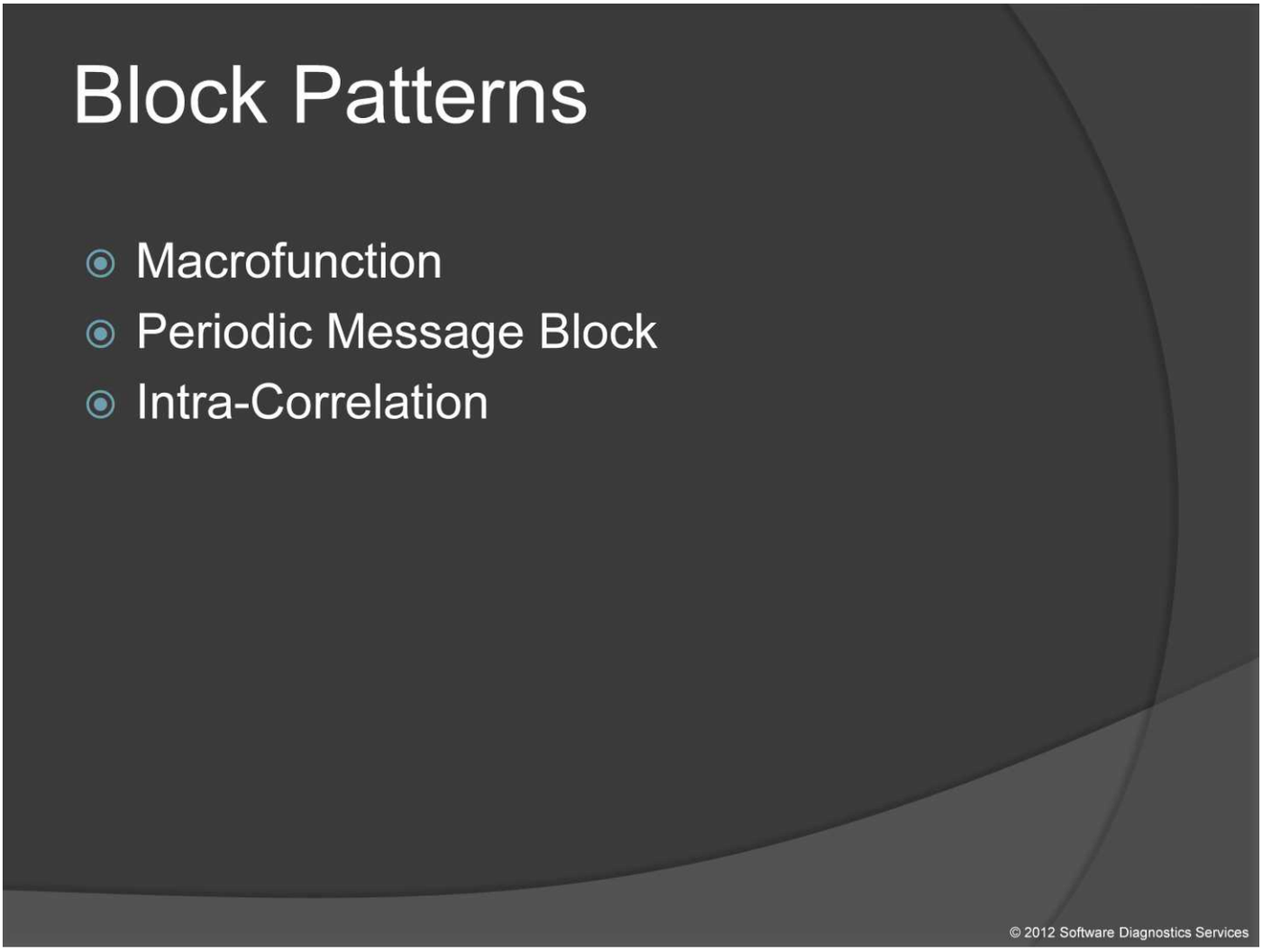
The modern software trace recording, visualization and analysis tools such as Process Monitor provide stack traces associated with trace messages. We can consider stack traces as software traces as well and, in a more general case, bundle them together (or attach as fibers) to a base software trace or log. For example, a trace message, that mentions an IRP can have its I/O stack attached together with a thread stack trace with function calls leading to a function that emitted the trace message. Note, that this pattern is different from **Exception Stack Trace,** which is just a reported stack trace formatted as trace messages in software trace or log.



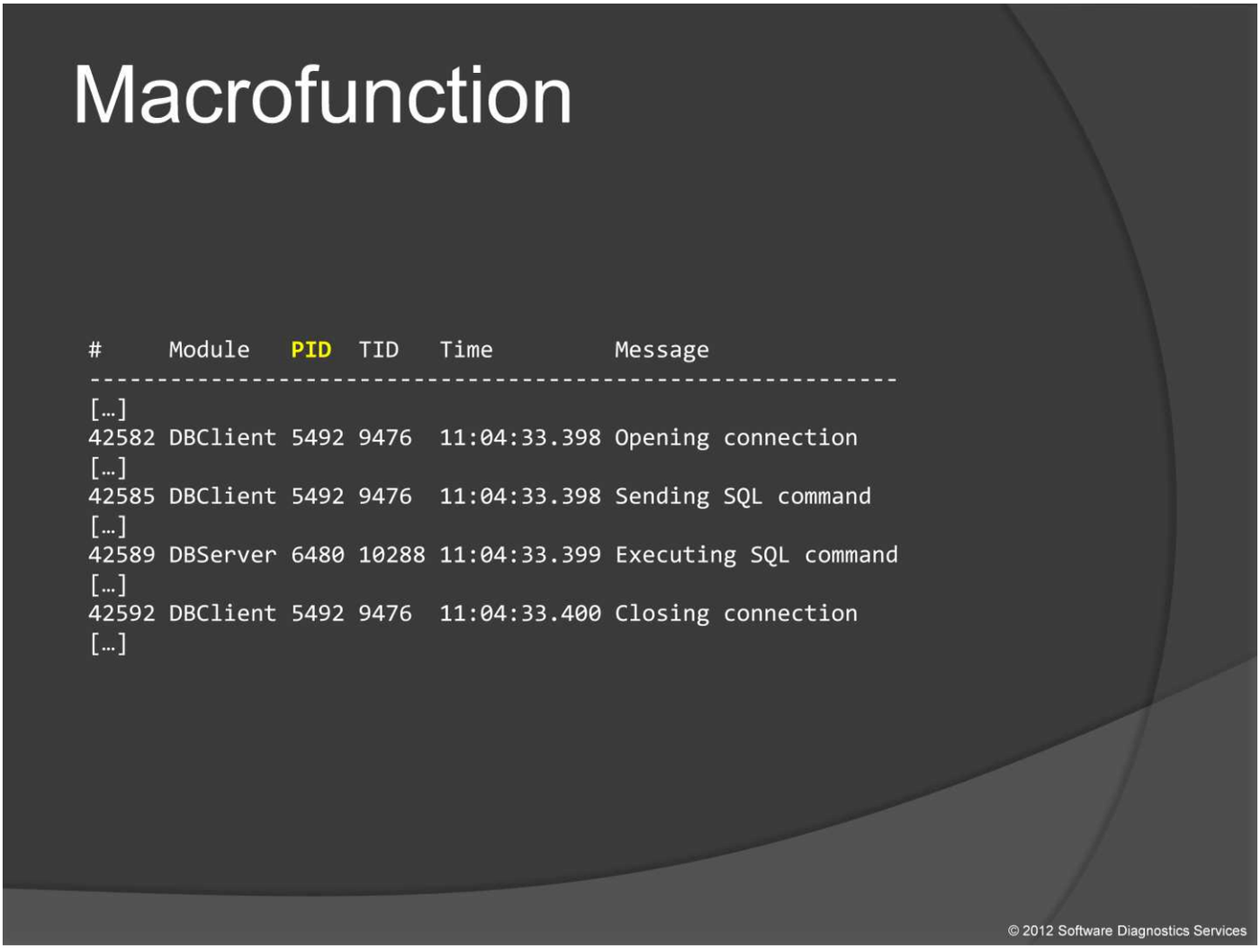
The large part of a typical software trace consists of requests and responses, for example, function or object method calls and returns. The code that generates trace messages is called *response-complete* if it traces both requests and responses. For such code (except in cases where tracing is stopped before a response, **Truncated Trace**) the absence of expected responses could be a sign of blocked threads or quiet exception processing (handled exceptions). The code that generates trace messages is called *exception-complete* if it also traces exception processing. Response-complete and exception-complete code is called *call-complete*. If we don't see response messages for call-complete code we have **Incomplete History,** and this might be because of either execution problems or **Sparse Trace** (that is code is not covered by tracing). In general, we can talk about the absence of certain messages in a trace as a deviation from the standard trace sequence template corresponding to a use case (the so-called **Master Trace** pattern).



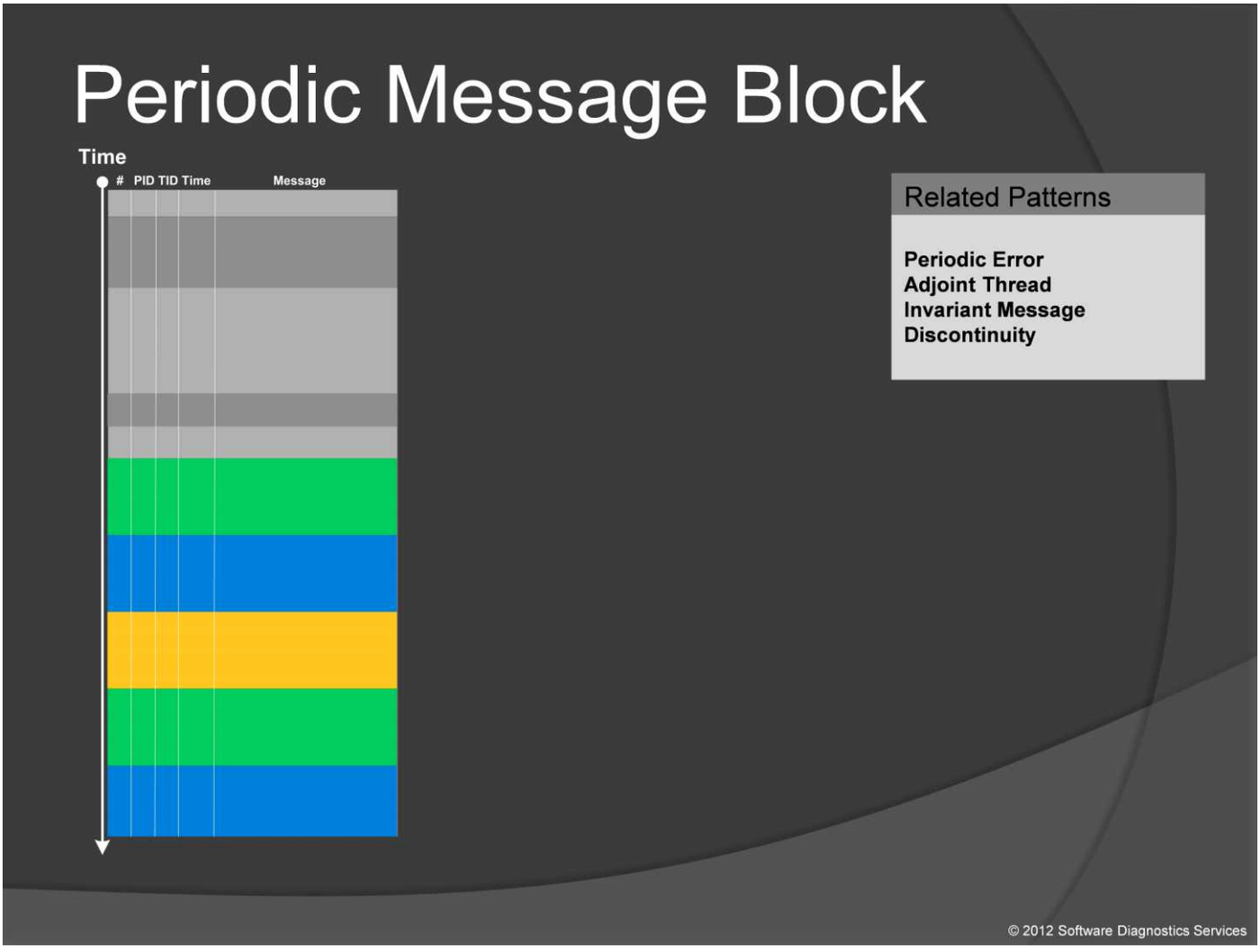
This pattern covers the pairs of opposite messages usually found in software traces and logs such as on this slide list. The absence of an opposite may point to some problems such as synchronization and leaks or **Incomplete History** (such as wait chains). There can always be a possibility that a second term is missing due to **Sparse Trace** not covered by sufficient trace statements in the source code, but this is a poor implementation choice that leads much to confusion during troubleshooting and debugging.



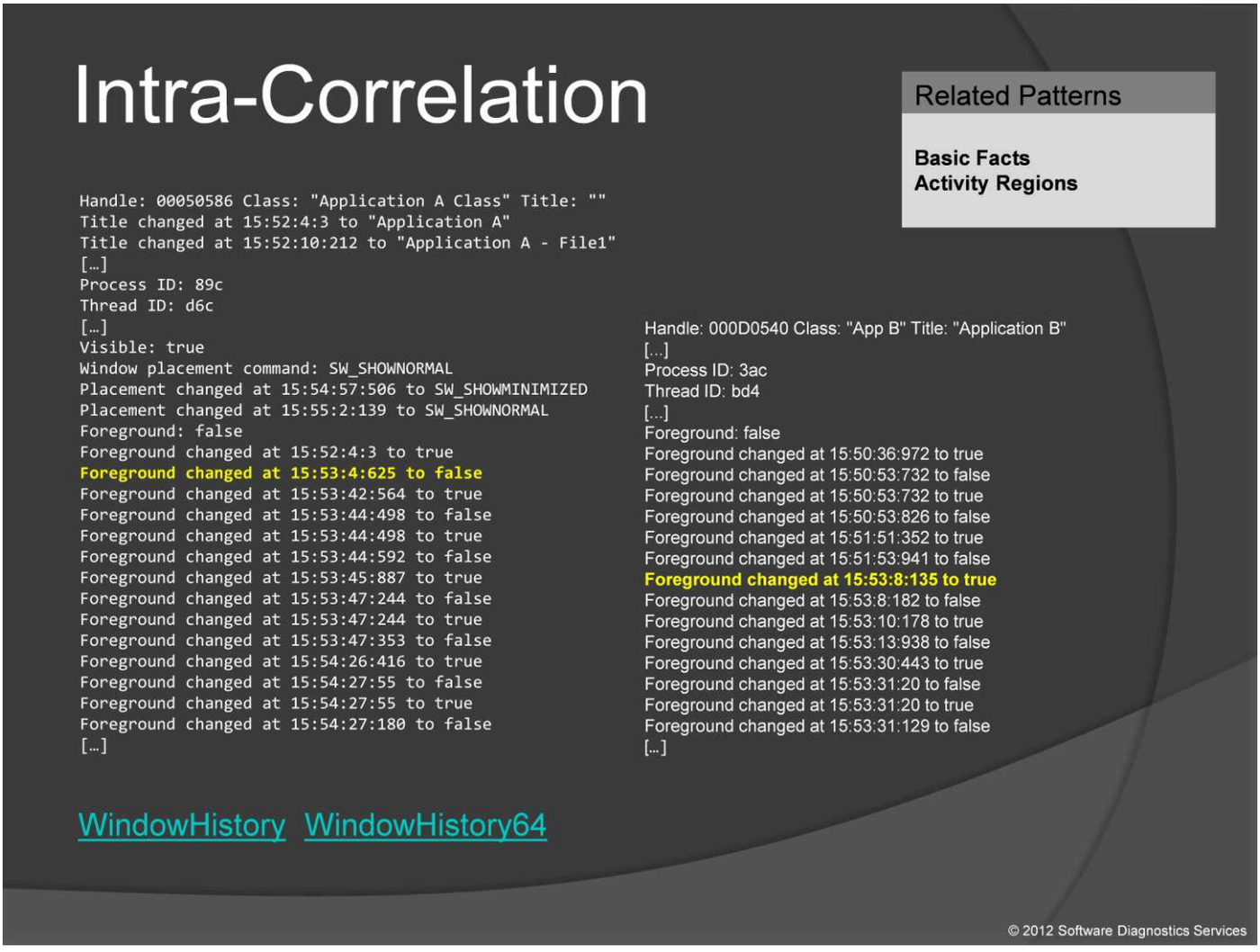
The seventh block of patterns we cover are patterns of message aggregates, message blocks.



Several trace messages may form a single semantic unit that we call **Macrofunction** in comparison to individual trace messages that serve the role of microfunctions. In this slide, we provide an example of a software log fragment for an attempt to update a database. We can consider **Macrofunction** as a use case. It involves different PIDs for a client and a server. So we see that these macrofunctions need not be from the same ATID (Adjoint TID).

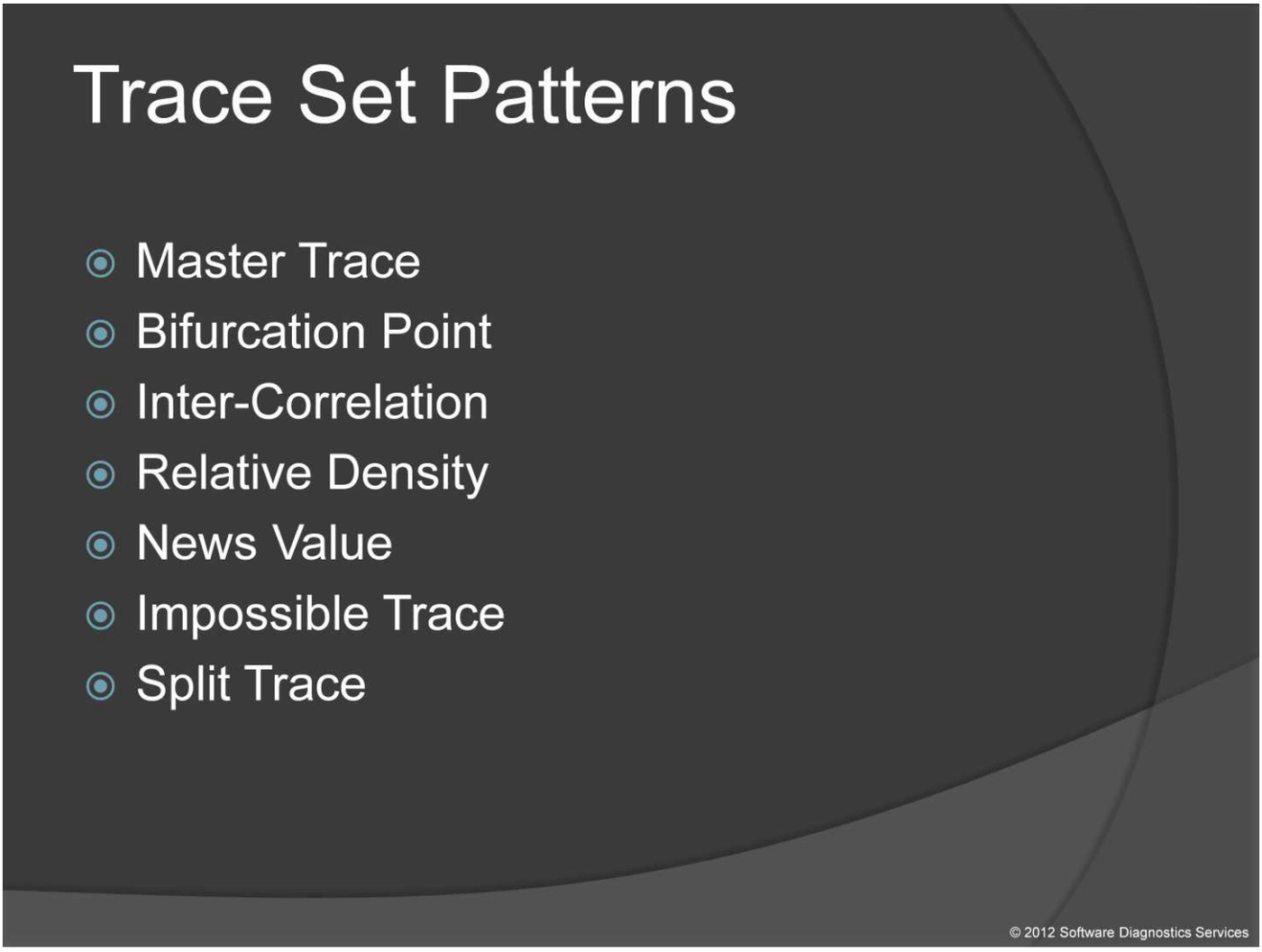


This pattern is similar to **Periodic Error** pattern but not limited to errors or failure status reports, for example, when some **Adjoint Thread** (such as messages from specific PID) stop to appear after the middle of the trace and after that there are repeated blocks of **Invariant Messages** from different PIDs with their threads checking for some condition (such as waiting for event) and reporting timeouts. There can be **Discontinuities** between the same message blocks from **Periodic Message Blocks**.

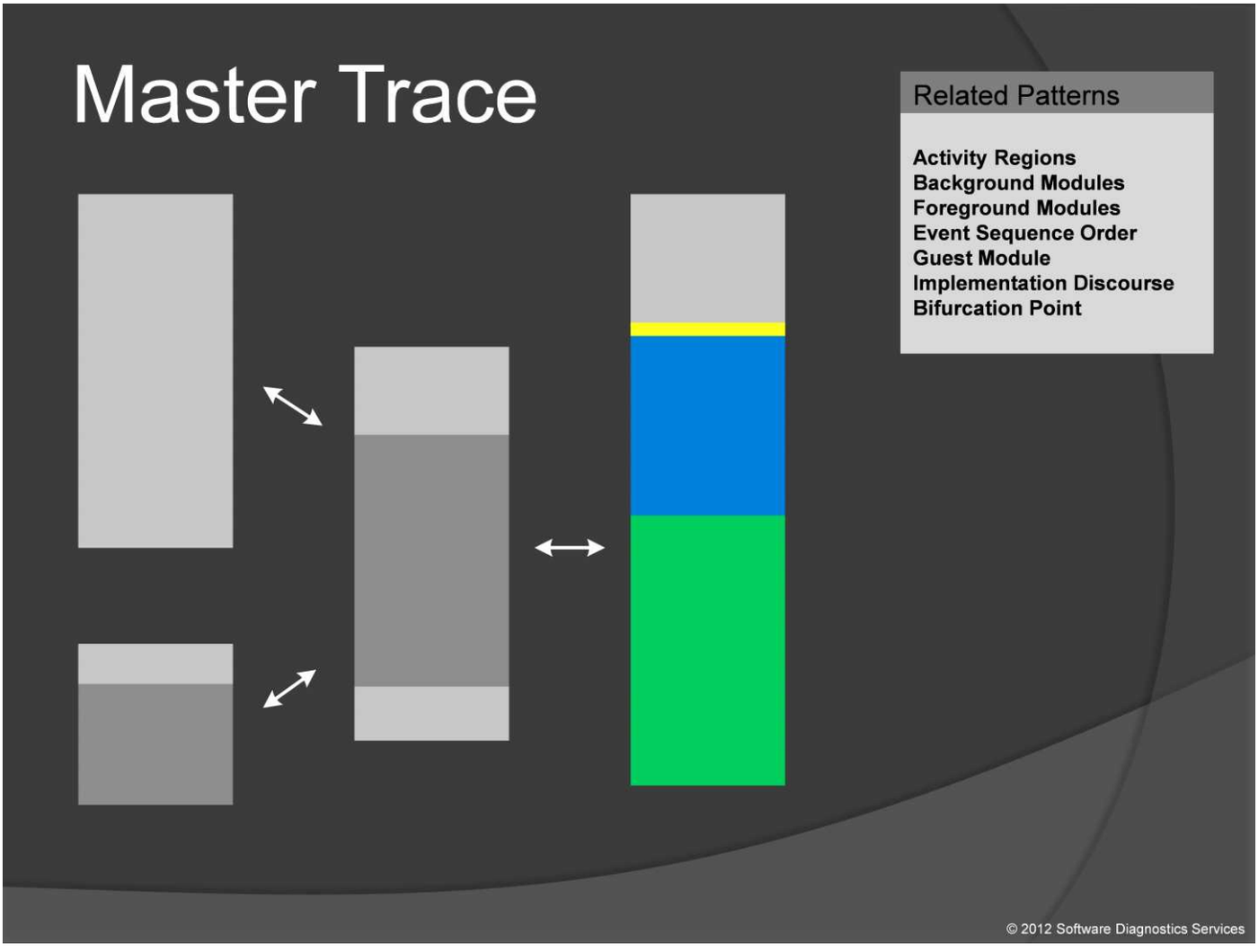


Sometimes we see a functional activity in a trace or see some messages that correspond to **Basic Facts** from problem description. Then we might want to find a **correlation** between that **Activity Region** or facts and another part of the trace. If that intra-correlation fits into our problem description we may claim a possible explanation or, if we are lucky, we have just found, an inference to the best explanation. In this slide, there is an example of Citrix WindowHistory tracing tool. A third-party application was frequently losing the focus, and the suspicion was on a terminal services client process. The trace fragment on the left was found corresponding to that application. Corresponding terminal services client window trace fragment didn't have any foreground changes, but another application main window had lots of them (the fragment on the right). We can see that most of the time when Application A window loses focus Application B window gets it.

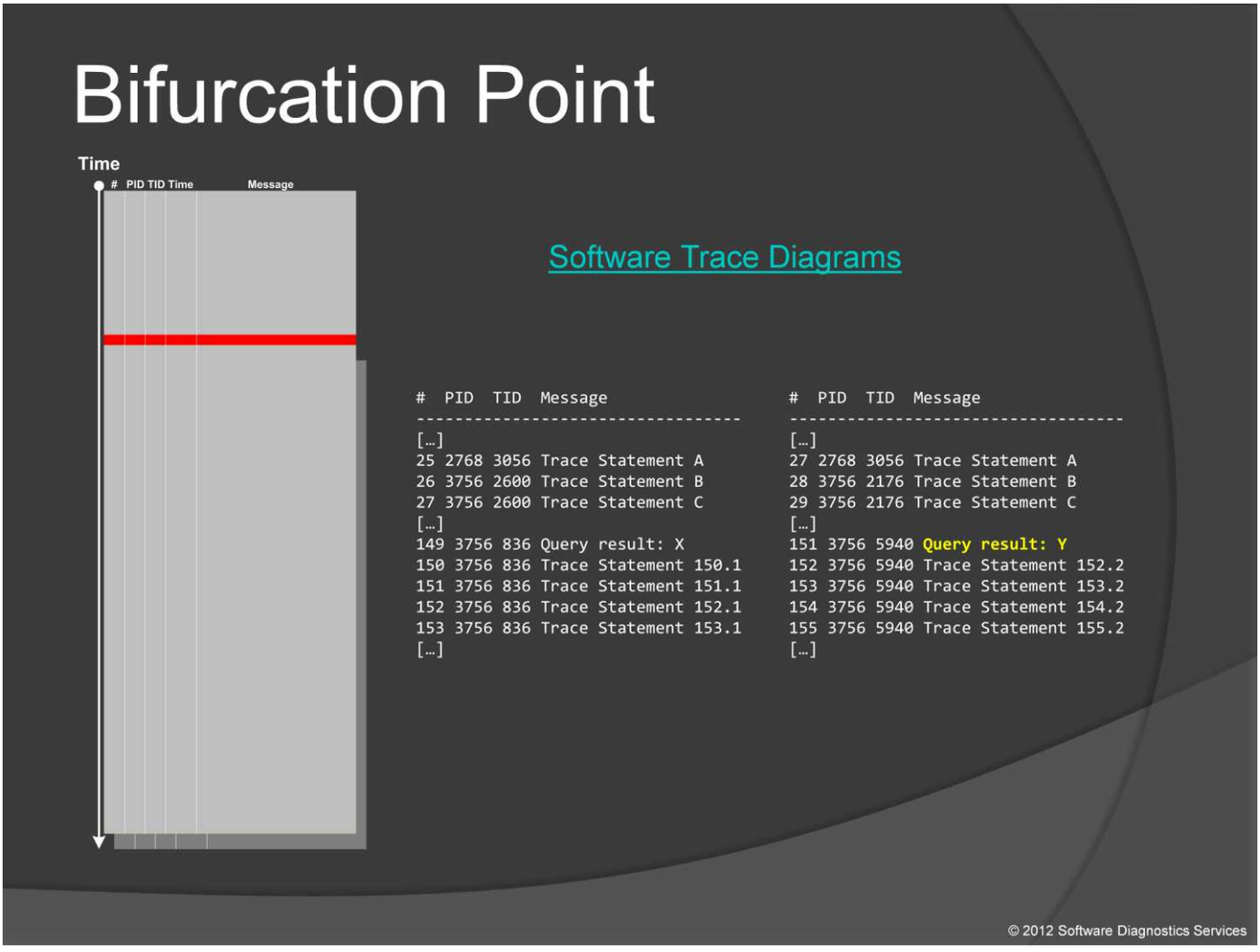
WindowHistory: http://support.citrix.com/article/CTX106985
WindowHistory64: http://support.citrix.com/article/CTX109235



The eighth block of patterns we cover are patterns for trace sets when we have several software traces and logs.

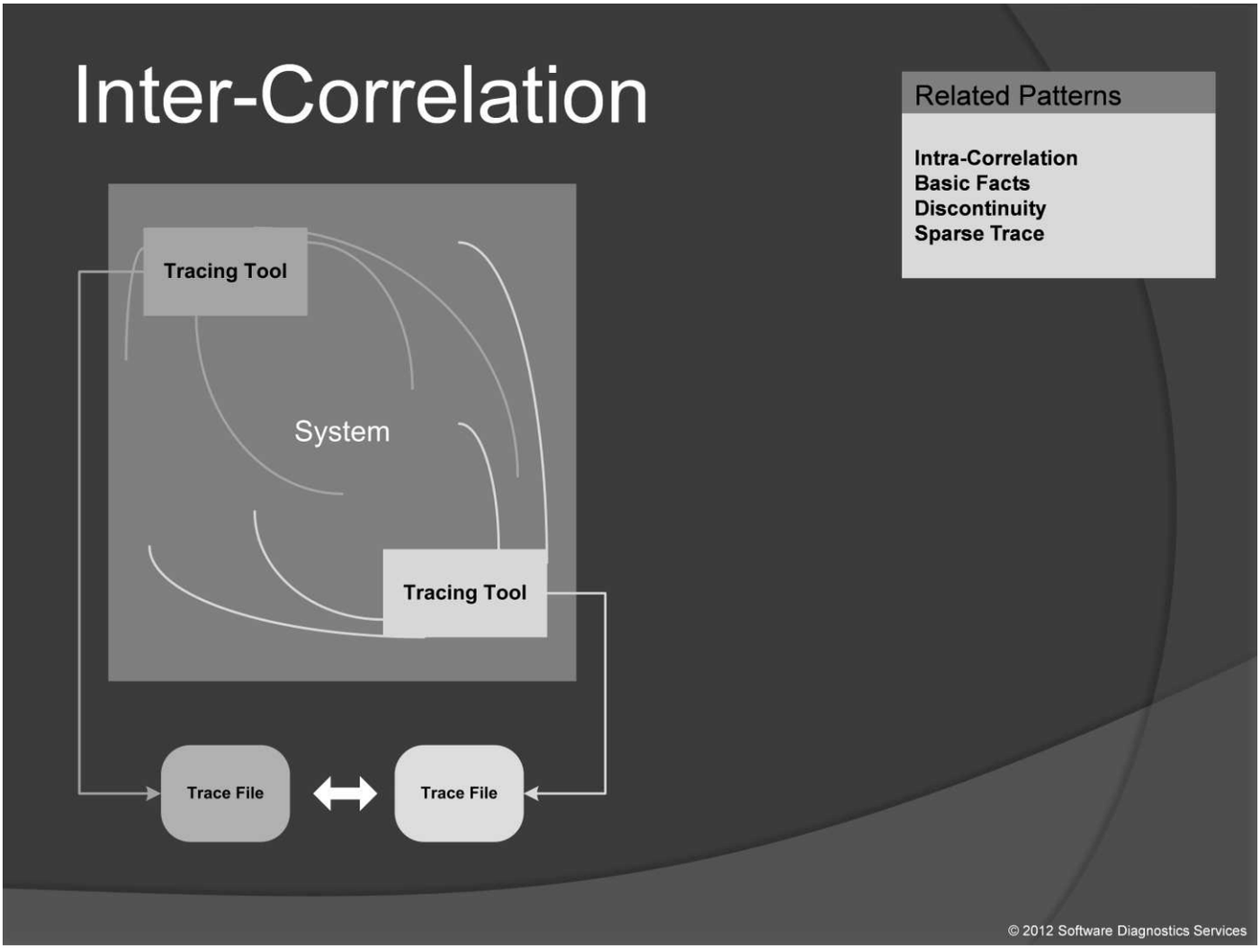


When reading and analyzing software traces and logs we always compare them to a **Master Trace,** which is a standard software trace corresponding to the functional use case. When looking at the software trace from a system we either know the correct sequence of **Activity Regions**, expect certain **Background** and **Foreground Modules**, certain **Event Sequence Order** or mentally construct a model based on our experience and **Implementation Discourse**. Software engineers usually internalize software master traces when they construct code and write tracing code for supportability. In the case of other people supporting a product during a post-construction phase, it is important to have a repository of traces corresponding to **Master Traces**. This helps in finding deviations after **Bifurcation Point**, for example. For a product that is meant to run on a variety of systems, working scenario traces can correspond to **Master Traces**.

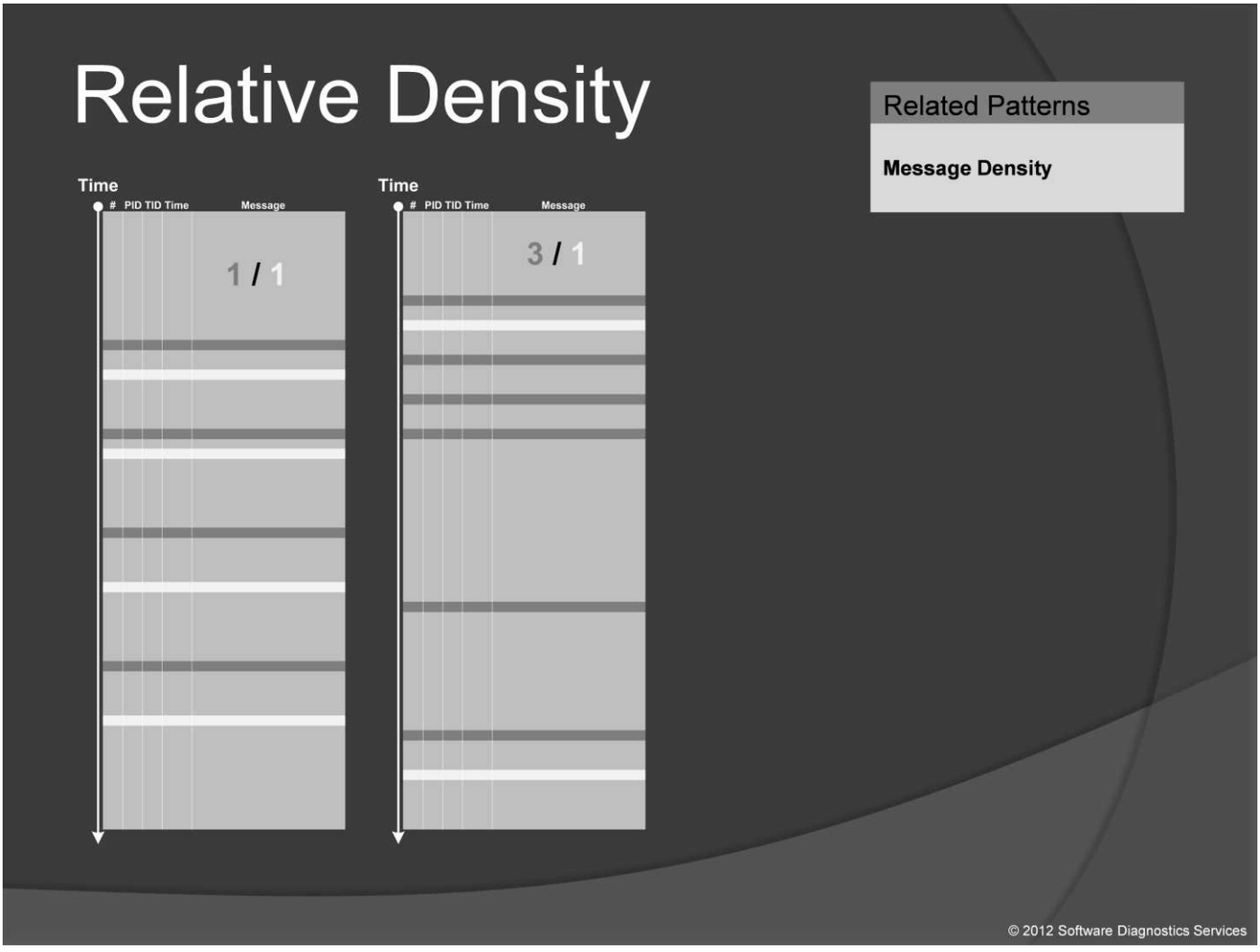


This pattern name is borrowed from Catastrophe Theory. It means a trace message after which software traces diverge for working and non-working abnormal scenarios. One such abstracted example is illustrated on this slide. On the left, we have a software trace from a normal working environment and on the right we have a software trace from a problem non-working environment. We see that messages A, B, C and further are identical up to Query result message. However, the last message differs greatly in reported results X and Y. After that, message distribution differs greatly in both size and content. Despite the same tracing time, say 15 seconds, **Message Current** is 155 msg/s for working and 388 msg/s for the non-working case. **Bifurcation Points** are easily observed when tracing noise ratio is small and, in a case of full traces, could be achieved by filtering irrelevant **Background Modules**. In this slide, I also put a link to the proposed software tracing diagrams.

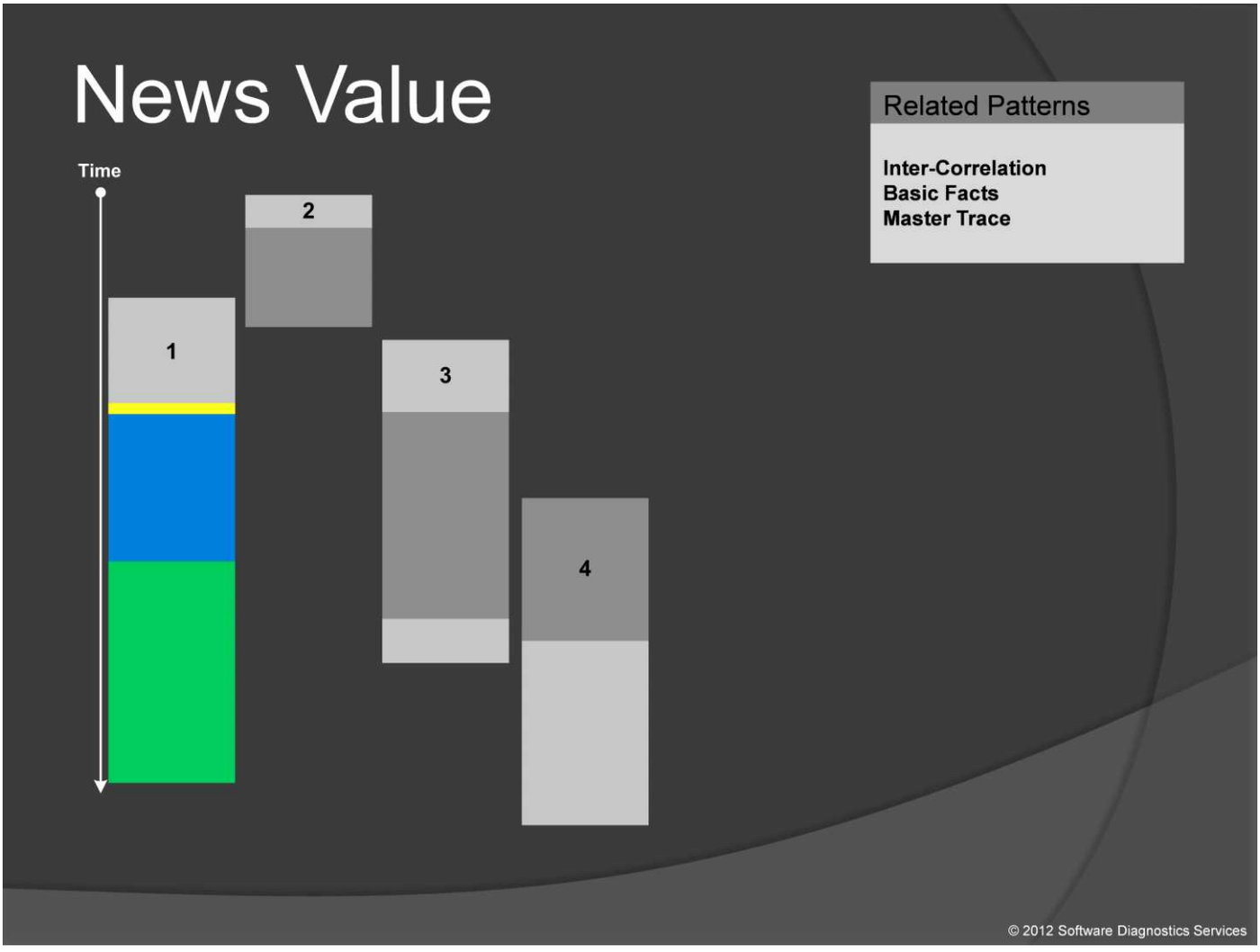
Software Trace Diagrams: Memory Dump Analysis Anthology, Volume 7, page 279
http://www.dumpanalysis.org/blog/index.php/2012/07/03/software-trace-diagrams-stdiagrams/



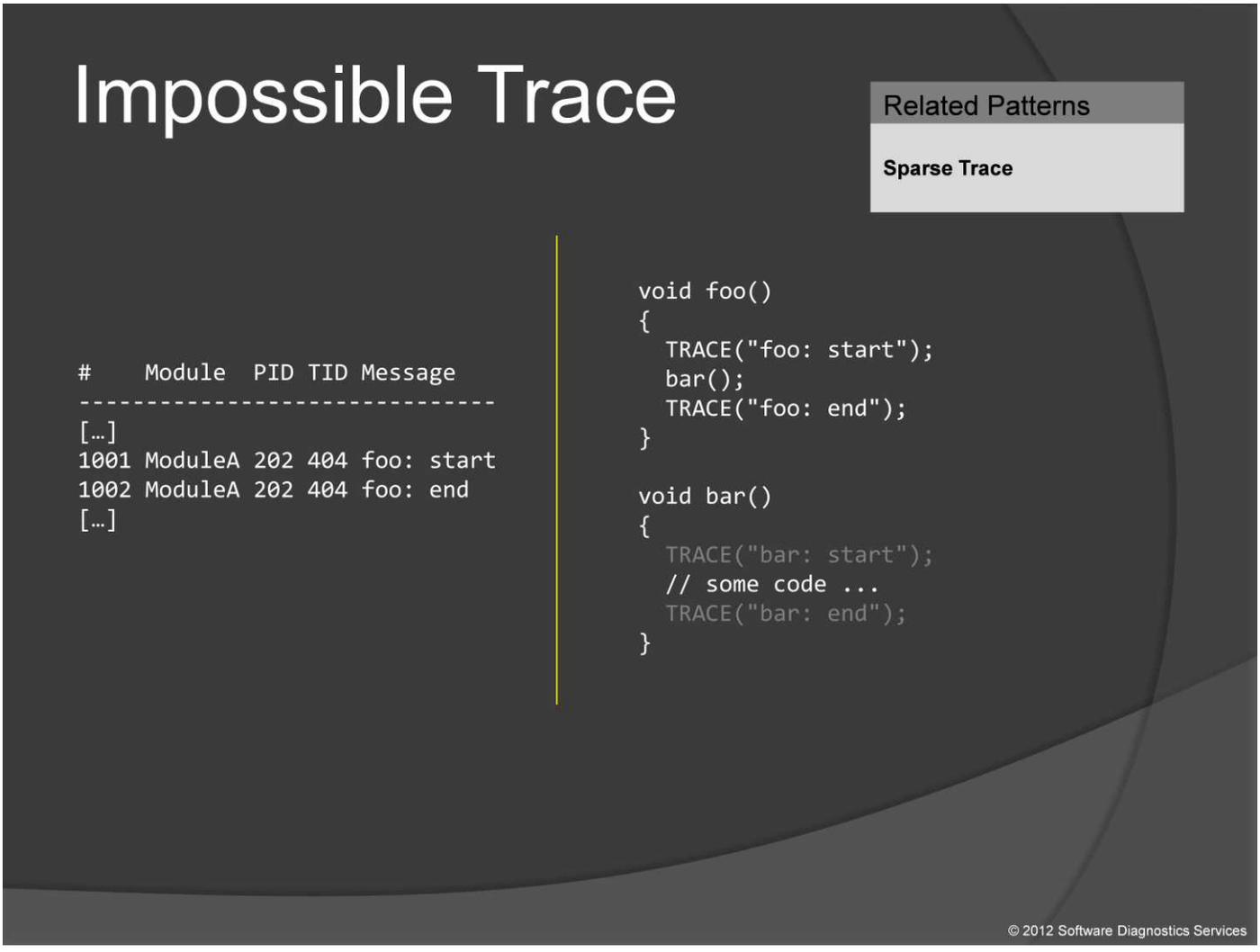
This pattern is analogous to the previously described **Intra-Correlation** pattern but involves several traces from possibly different trace tools recorded (most commonly) at the same time or during an overlapping time interval. However, the purpose of using different tracing tools is to cover system events more completely. One of the examples we can provide is when we have **Discontinuity,** and its interval events are covered by a different tool.



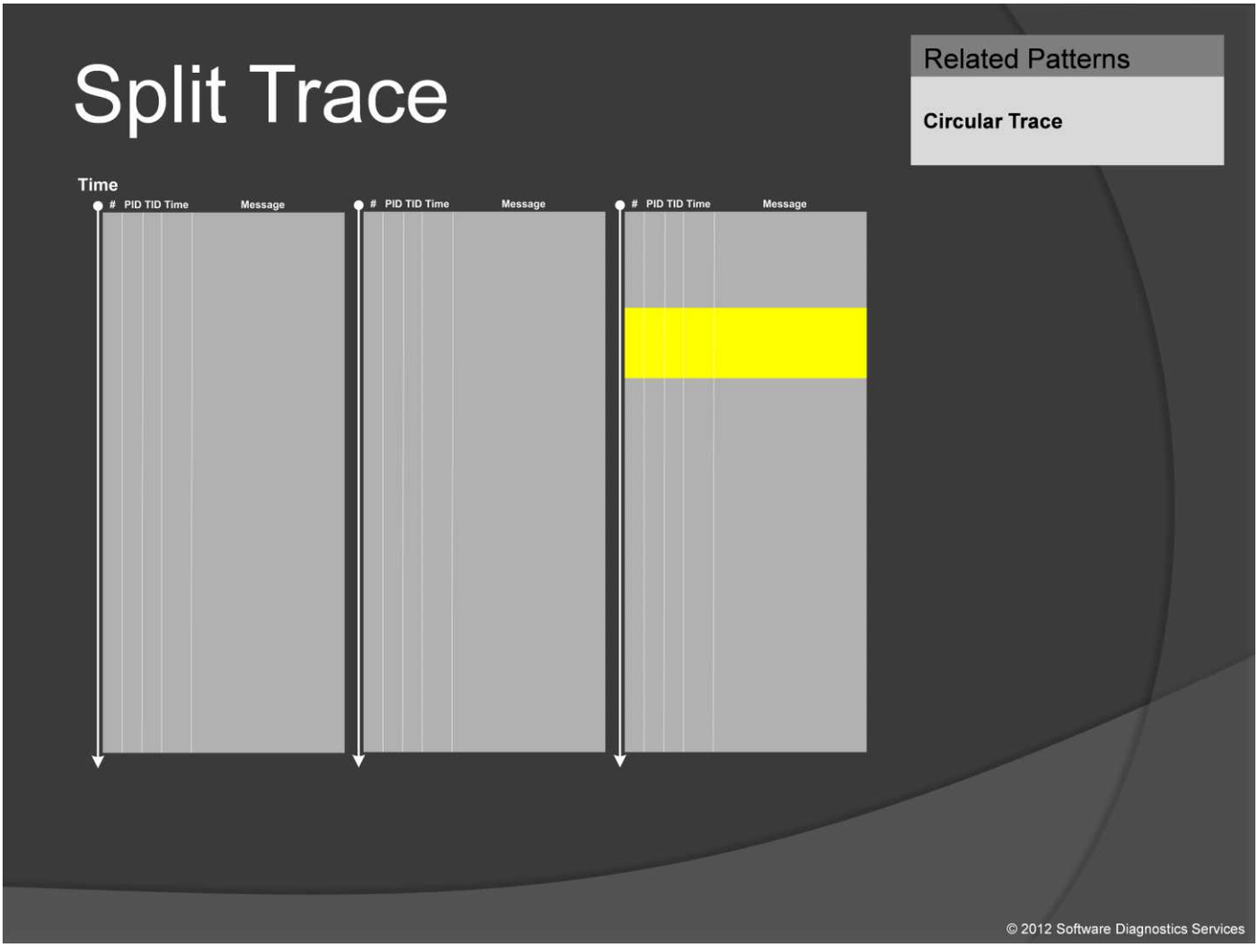
This pattern describes anomalies related to the semantically related pairs of trace messages, for example, “data arrival” and “data display”. Their **Message Densities** can be put in a ratio and compared between working and non-working scenarios. Recall that **Message Density** is a ratio of the number of specific messages to the total number of messages in a software trace. Because the total number of trace messages cancel each other, we have just the mutual ratio of two message types. One hypothetical example is shown on this slide. We see on the left picture the ratio of “data arrival” to “data display” is 1/1. On the right picture we see the increased ratio of “data arrival” to “data display” messages (3/1) and this accounts for reported visual data loss and sluggish GUI.



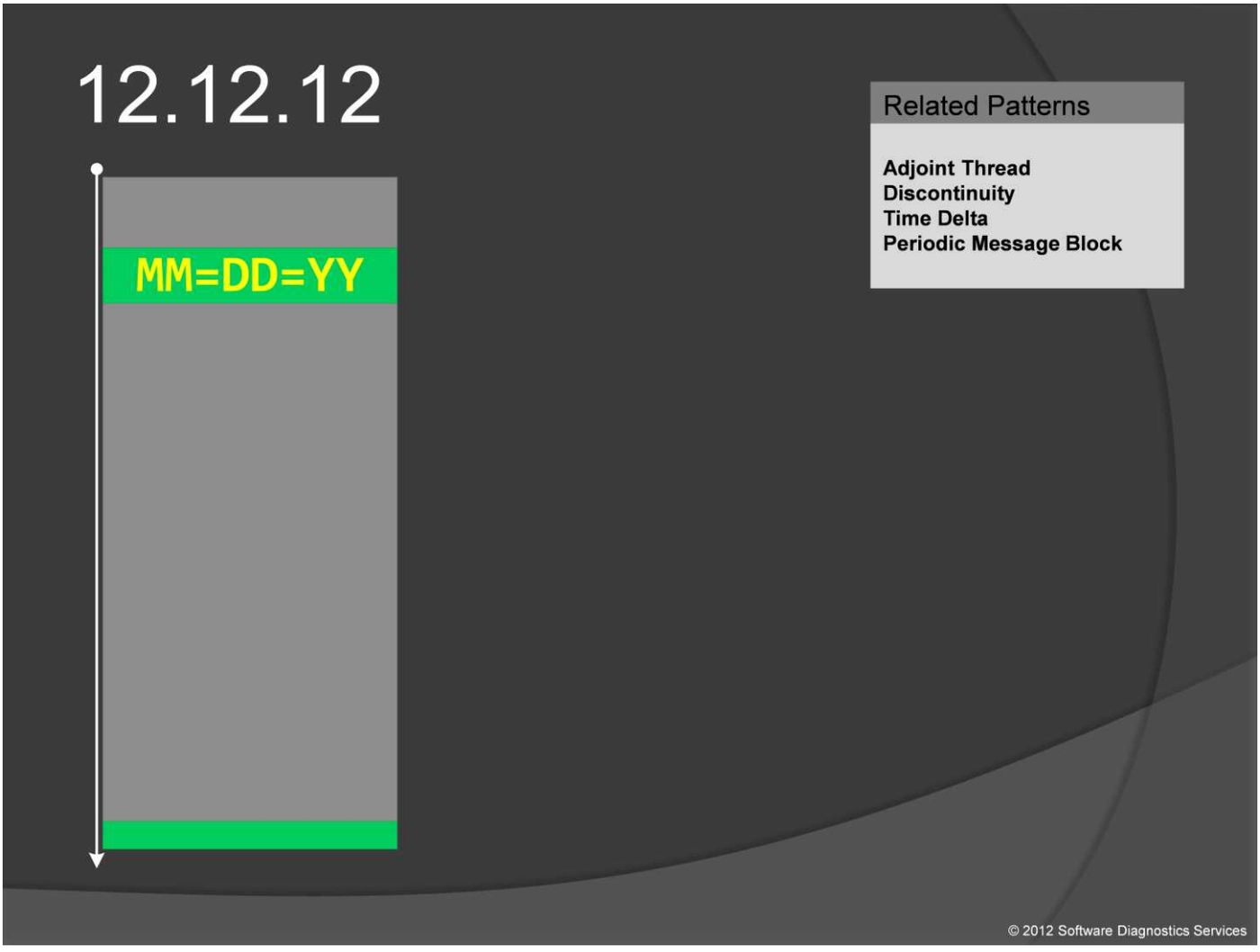
This pattern assigns relative importance to software traces for problem-solving purposes, especially when related to problem description (**Basic Facts**), recent incidents and timestamps of other supporting artifacts such as memory dumps. Assessing value is often done in conjunction with trace **Inter-Correlation** analysis of the most recent logs. For example, in relation to a trace on the left (1st), only the 3rd trace has some value as the other two were recorded either earlier or later.



Although rarely (at least for myself) but it happens that when we look at a trace and then say it's an **Impossible Trace**. For example, we see on the trace fragment shown on the left of this slide that the function *foo* had been called. However, if we look at the corresponding source code on the right we would see that something is missing: the function *bar* must have been called with its own set of trace messages we don't see in the trace. Here we might suspect that the runtime code was being modified, perhaps by patching. In other cases of missing messages, we can also suspect thrown exceptions or local buffer overflows that led to wrong return address skipping the code with expected tracing statements. The mismatch between the trace and the source code we are looking at is also possible if the old source code didn't have *bar* function called (**Sparse Trace** pattern).

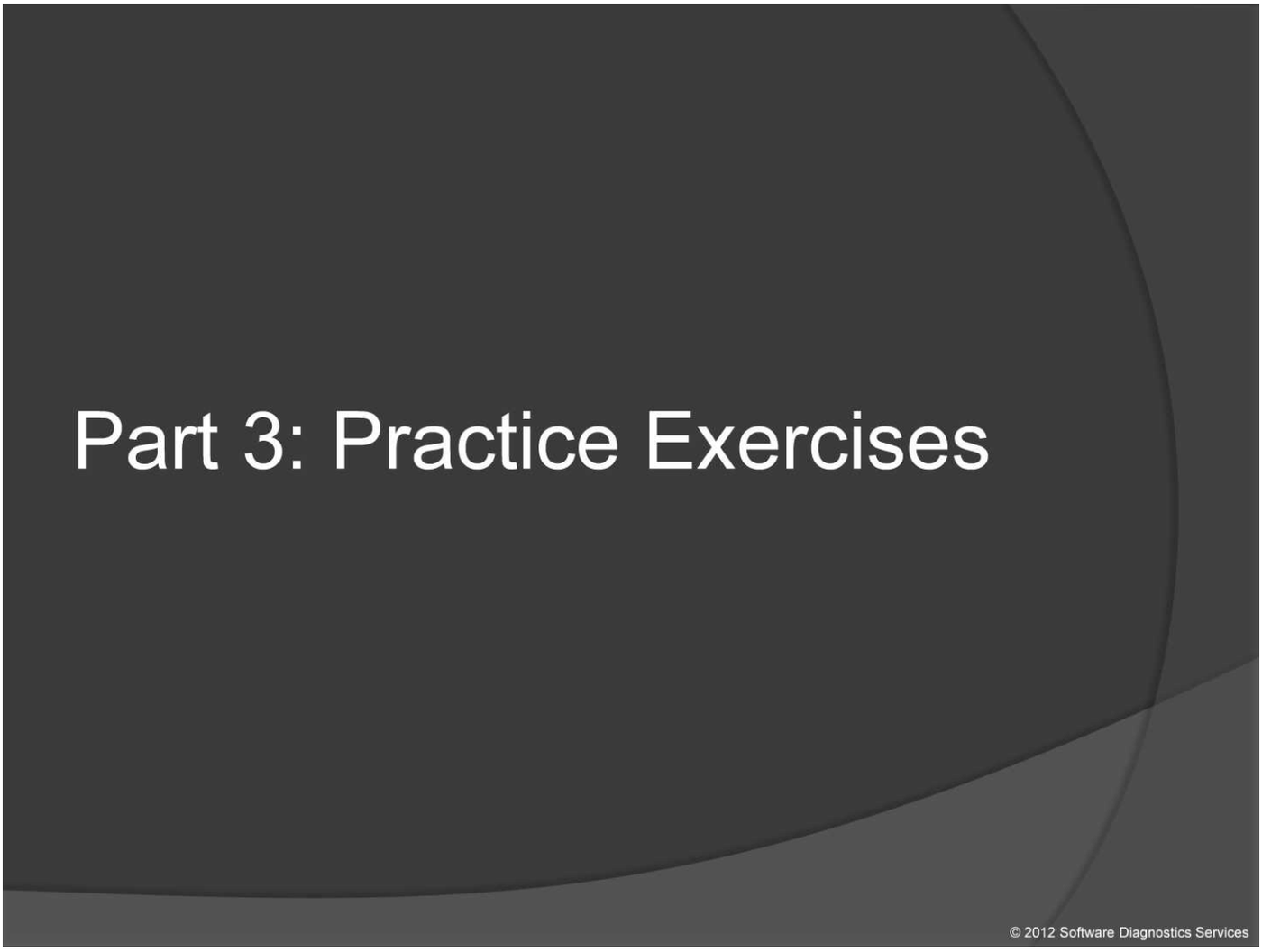


Some tracing tools such as Citrix CDFControl have the option to split software traces and logs into several files during long recording. Although this should be done judiciously, it is really necessary sometimes. So, what should we do if we get several large trace files, and we want to use some other analysis tool such as Citrix CDFAnalyzer? If we know the problem happened just before the tracing session was stopped, we can look at the last such file from the file sequence. If we have small files, then we should recommend **Circular Trace**. Another method if we need adjoint threading is exporting into CSV files and importing into Excel.

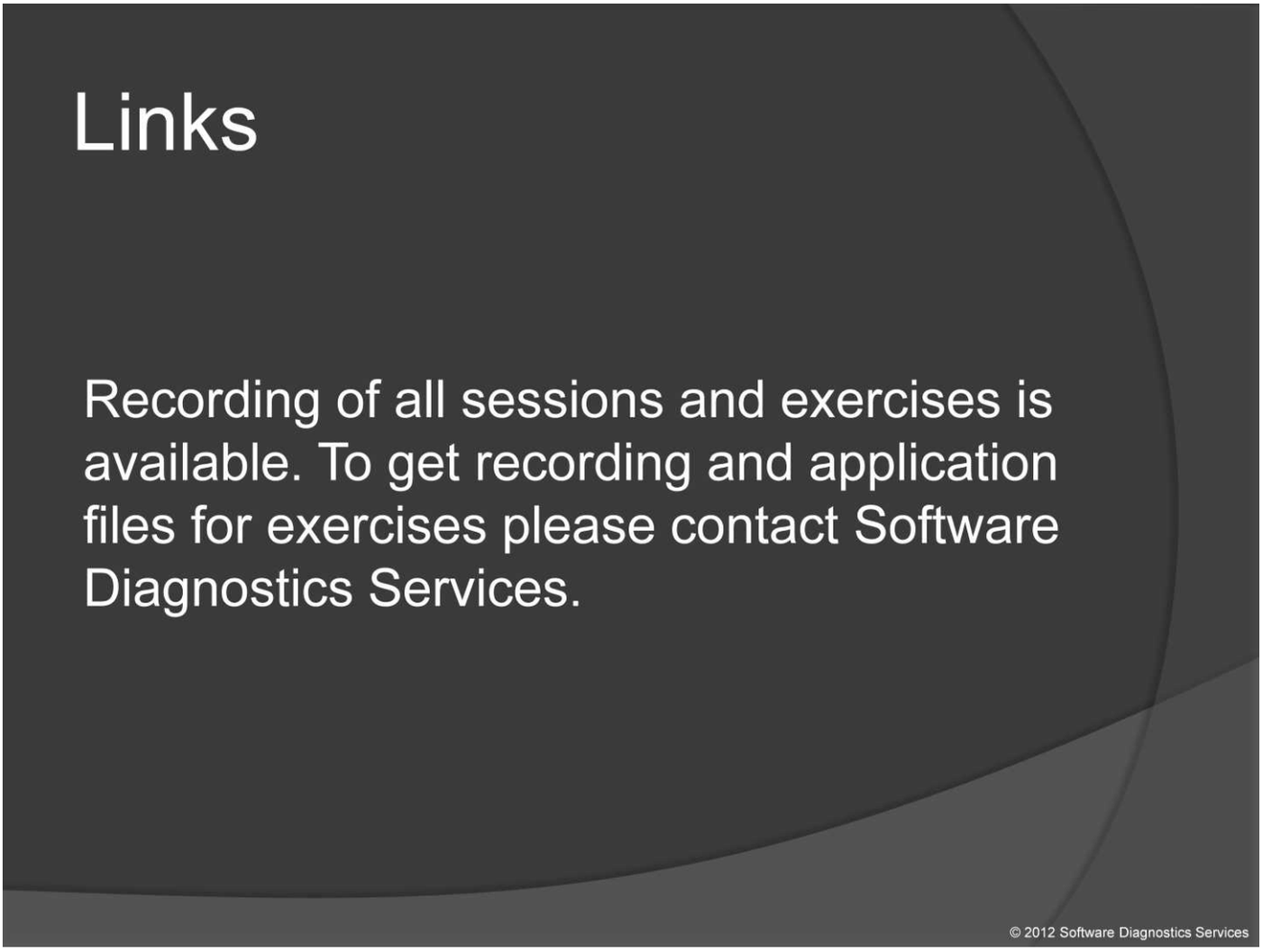


A bit of a diversion to finish with patterns. We just passed 12.12.12 and let's apply our patterns to it. Suppose we trace personal or world events each year and form an **Adjoint Thread** with messages having DD=MM=YY timestamp invariant (or filter them). Then we clearly see a **Discontinuity** before the next Century with much bigger **Time Delta** than between such messages. Also, these messages form **Periodic Message Block** in relation to the full trace.

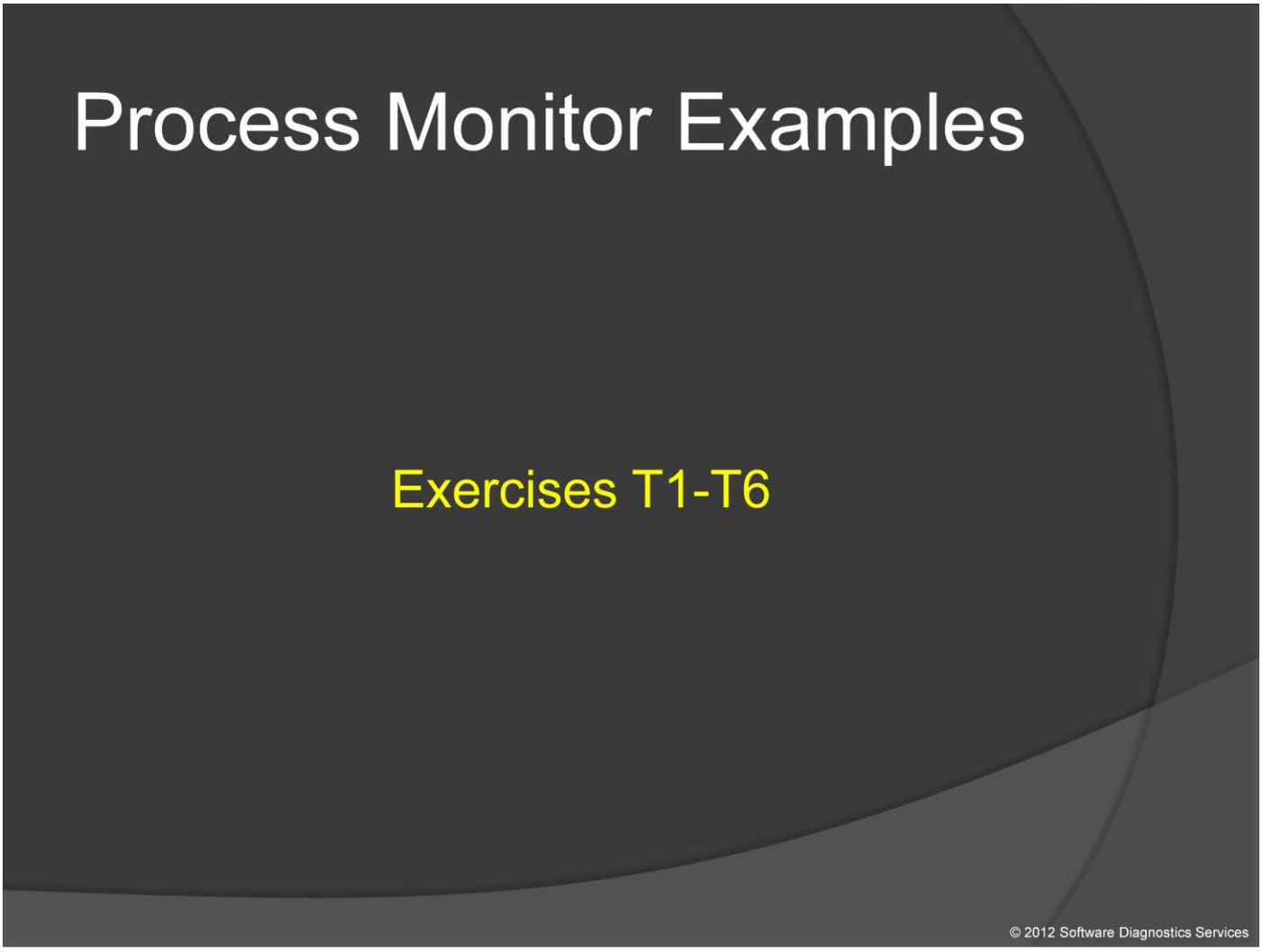
Practice Exercises



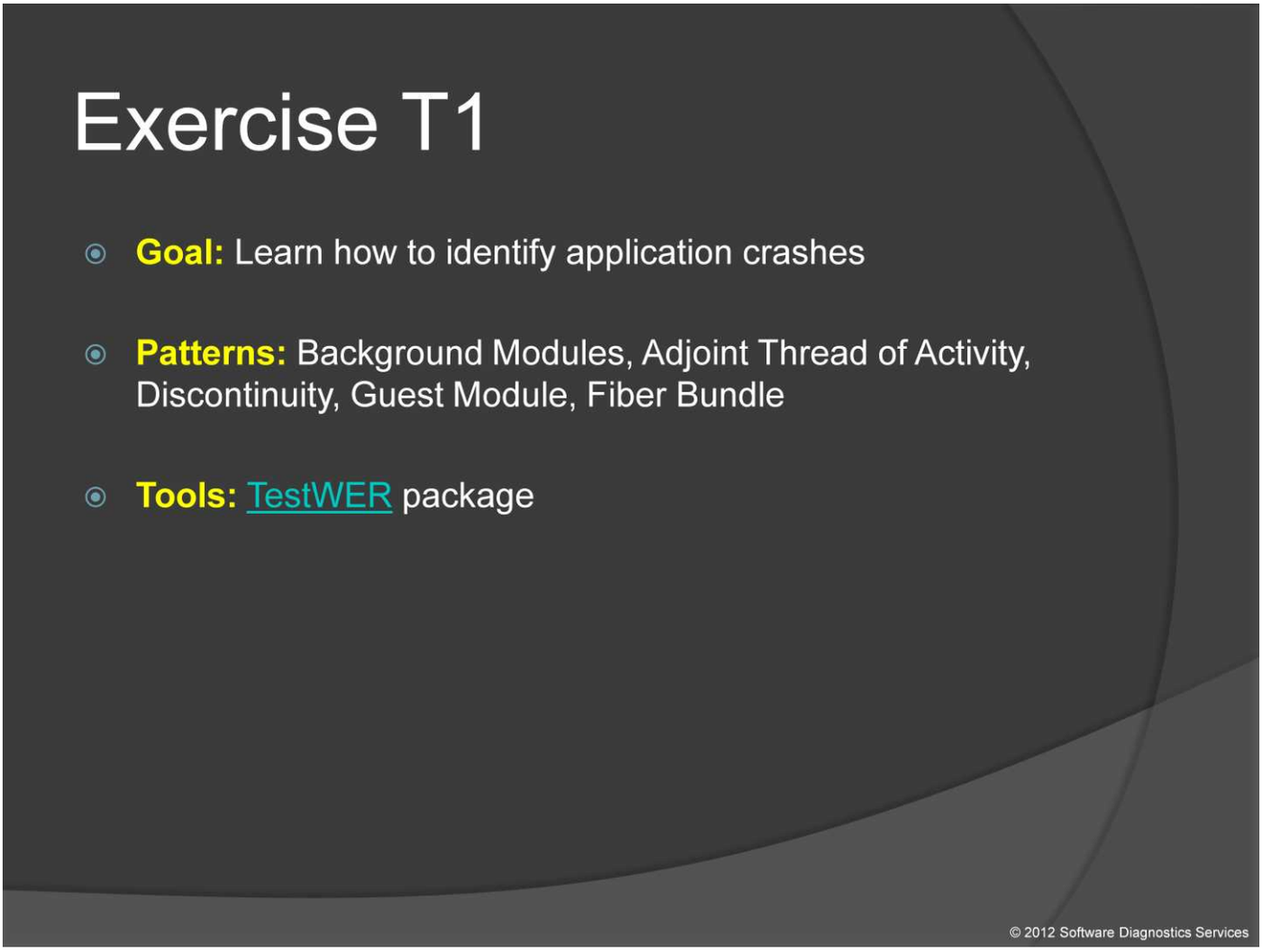
Now we come to practice. The goal is to show you some frequent trace analysis patterns in context.



If you bought this course in ebook or paperback format from Amazon, B&N or any other bookshop, please forward the proof of purchase such as an invoice to training@patterndiagnostics.com



We demonstrate frequent patterns using Process Monitor. The analysis patterns and techniques used there are very similar to Citrix CDF file analysis tools, for example, and indeed in one of the exercises we look at terminal session initialization process launch sequence. Almost all exercises require modeling applications we created specifically for this training, and we provide download links if you want to reproduce tracing sessions. We have found that step-by-step instructions become very complicated for GUI analysis tools as environment including screen resolution may vary greatly. Such instructions are very useful for command line tools with the predictable output. So we decided to provide demonstrations and record it for your reference.



TestWER: http://support.citrix.com/article/CTX111901

Exercise T2

- **Goal:** Learn how to identify CPU consumption, profile processes and threads
- **Patterns:** Activity Region, Characteristic Message Block, Periodic Message Block, Thread of Activity, No Activity, Counter Value, Sparse Trace
- **Tools:** AppT2

Exercise T3

- **Goal:** Learn how to calculate message current and density
- **Patterns:** Activity Region, Thread of Activity, Time Delta, Message Current, Message Density, Relative Density
- **Tools:** AppT3

Exercise T4

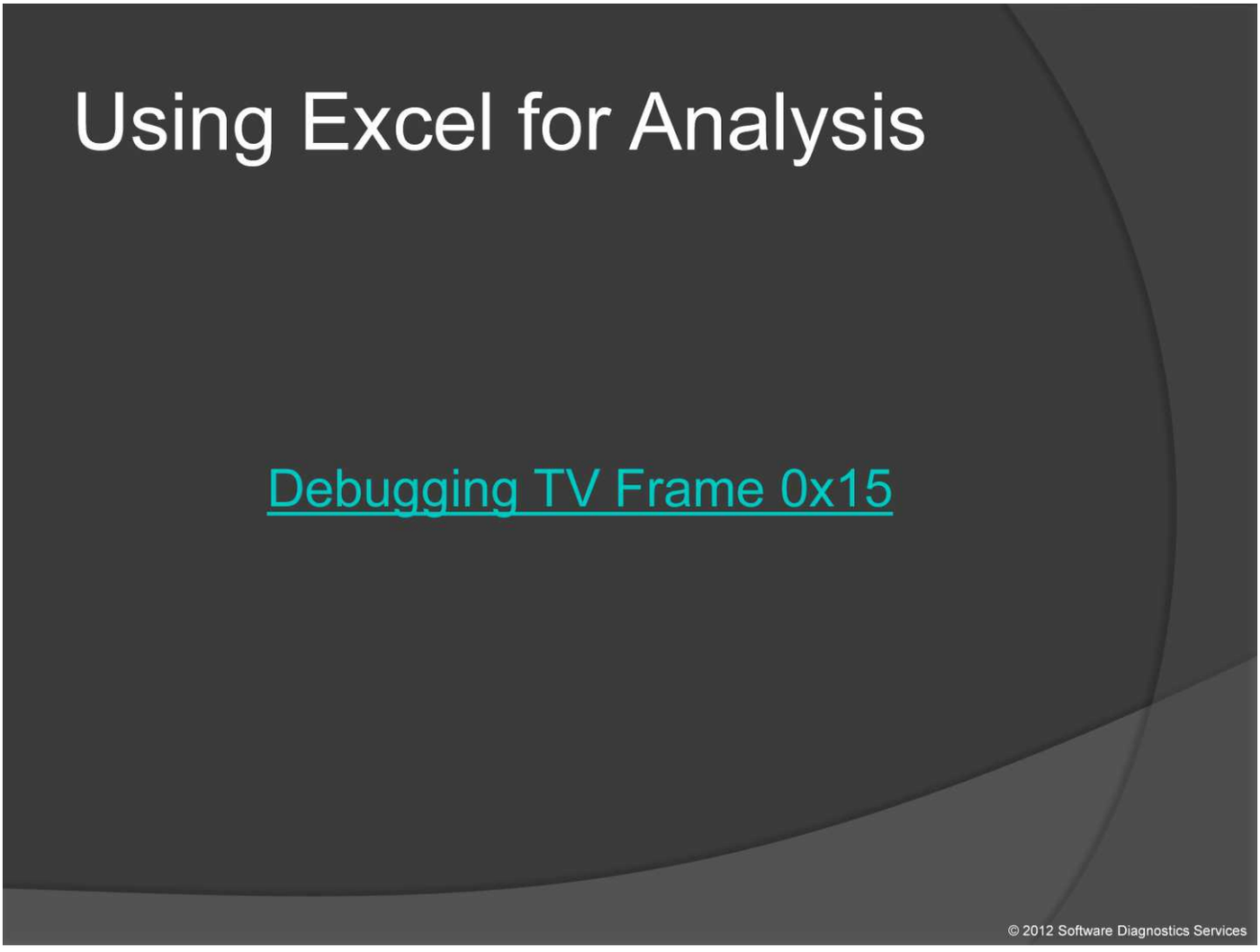
- **Goal:** Learn how to compare software traces
- **Patterns:** Master Trace, Characteristic Message Block, Bifurcation Point
- **Tools:** AppT4

Exercise T5

- **Goal:** Learn process startup sequence for terminal services session
- **Patterns:** Adjoint Thread of Activity, Anchor Messages, Message Interleave

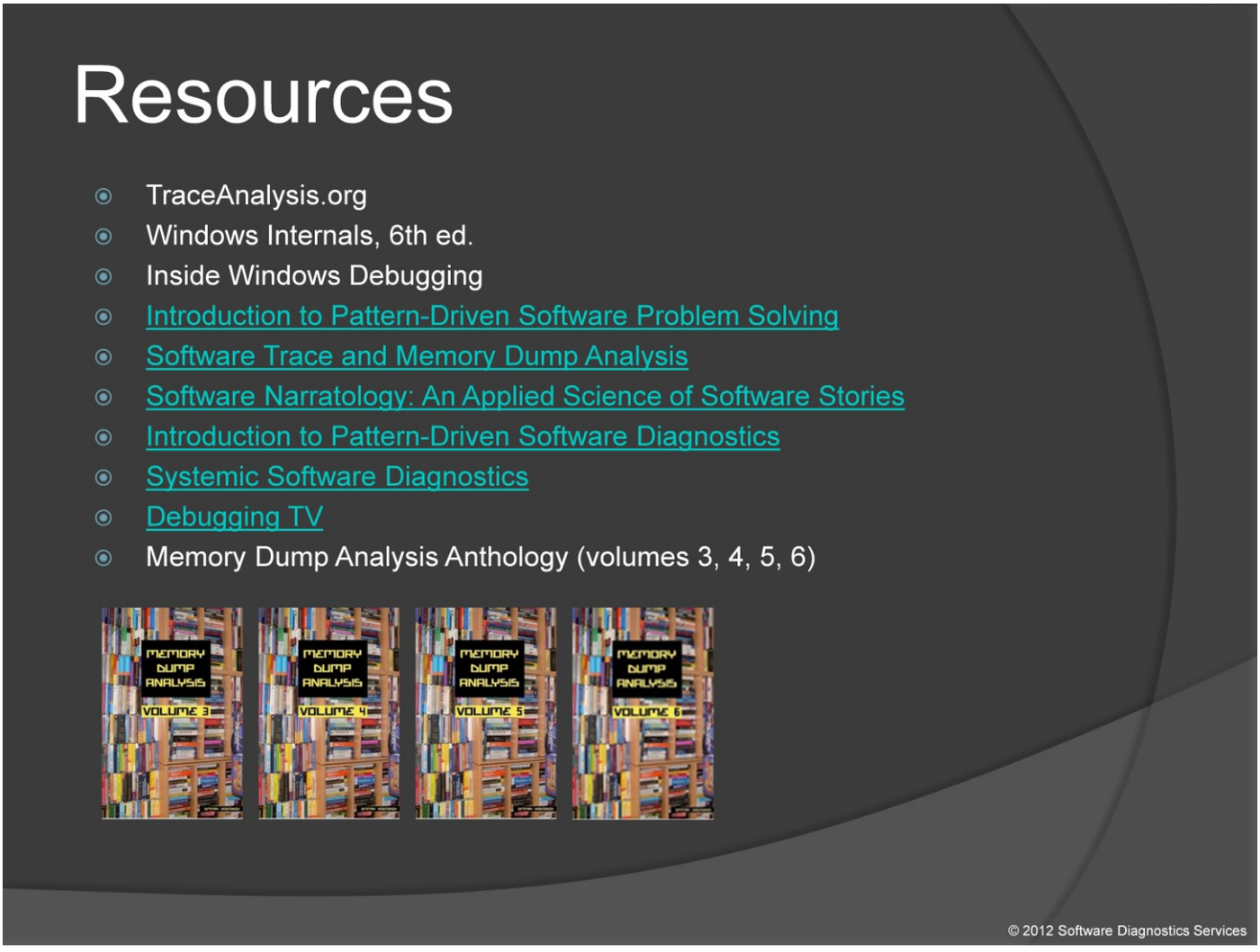
Exercise T6

- **Goal:** Learn how to work with split traces
- **Patterns:** Split Trace, Adjoint Thread of Activity, Fiber Bundle
- **Tools:** AppT6



We decided to include a link to Excel analysis presentation in order to provide the better exercise using different tools.

Debugging TV Frame 0x15: http://www.youtube.com/watch?v=Nx-ORaTmEPc
You can also download a video file from www.debugging.tv



Introduction to Pattern-Driven Software Problem Solving:
http://www.patterndiagnostics.com/PDSPSI-materials

Software Trace and Memory Dump Analysis:
http://www.patterndiagnostics.com/STMDA-materials

Software Narratology: An Applied Science of Software Stories:
http://www.patterndiagnostics.com/Introduction-Software-Narratology-materials

Introduction to Pattern-Driven Software Diagnostics:
http://www.patterndiagnostics.com/Introduction-Software-Diagnostics-materials

Systemic Software Diagnostics:
http://www.patterndiagnostics.com/systemic-diagnostics-materials

Debugging TV:
http://www.debugging.tv

Note: Memory Dump Analysis Anthology**,** Volumes 7, 8a, 8b, and 9a contain new patterns not covered in this training.

App Source Code

```
// AppT2 - Accelerated Windows Software Trace Analysis training
// Copyright (c) 2012 Memory Dump Analysis Services
// GNU GENERAL PUBLIC LICENSE
// http://www.gnu.org/licenses/gpl-3.0.txt

#include "stdafx.h"
#include <windows.h>
#include <process.h>

#define WAIT (1000*60)

void thread_one(void *wait)
{
	wchar_t buf[] = L"Hello Trace!";
	DWORD dwLoops = (DWORD)wait/60;

	while (--dwLoops)
	{
		HANDLE hFile = CreateFile(L".\\AppT2.dat",  GENERIC_READ | GENERIC_WRITE, 0, NULL,
					CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
		DWORD dww = 0;
		WriteFile(hFile, buf, sizeof(buf), &dww, NULL);
		CloseHandle(hFile);
		Sleep(60);
	}
}

void thread_two(void *wait)
{
	DWORD dwTicks = GetTickCount();
	while (true)
	{
		if (GetTickCount()-dwTicks > (DWORD)wait)
			break;
	}
}

int main(int argc, WCHAR* argv[])
{
	_beginthread(thread_two, 0, (void *)WAIT);
	_beginthread(thread_one, 0, (void *)WAIT);

	Sleep(WAIT);

	return 0;
}
```

```
// AppT3 - Accelerated Windows Software Trace Analysis training
// Copyright (c) 2012 Memory Dump Analysis Services
// GNU GENERAL PUBLIC LICENSE
// http://www.gnu.org/licenses/gpl-3.0.txt

#include "stdafx.h"
#include <windows.h>
#include <process.h>

#define WAIT (1000*60)

void thread_one(void *wait)
{
	DWORD dwLoops = (DWORD)wait;

	while (--dwLoops)
	{
		HANDLE hFile = CreateFile(L".\\AppT3.dat",  GENERIC_READ | GENERIC_WRITE, 0, NULL,
					CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
		CloseHandle(hFile);
		Sleep(1);
	}
}

void thread_two(void *wait)
{
	Sleep((DWORD)wait);
}

int main(int argc, WCHAR* argv[])
{
	_beginthread(thread_two, 0, (void *)WAIT);
	_beginthread(thread_one, 0, (void *)WAIT);

	Sleep(WAIT);

	return 0;
}
```

```
// AppT4 - Accelerated Windows Software Trace Analysis training
// Copyright (c) 2012 Memory Dump Analysis Services
// GNU GENERAL PUBLIC LICENSE
// http://www.gnu.org/licenses/gpl-3.0.txt

#include "stdafx.h"
#include <windows.h>
#include <process.h>

#define WAIT (1000*60)

void thread_one(void *wait)
{
	DWORD dwLoops = (DWORD)wait/60;
	DWORD dww = 0;
	wchar_t szFile[_MAX_PATH] = {0};

	HANDLE hFile = CreateFile(L".\\AppT4.cfg",  GENERIC_READ | GENERIC_WRITE, 0, NULL,
				OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
	if (GetLastError() == ERROR_ALREADY_EXISTS)
	{
		ReadFile(hFile, szFile, sizeof(szFile), &dww, NULL);
	}
	CloseHandle(hFile);

	while (--dwLoops)
	{
	    hFile = CreateFile(szFile,  GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING,
				FILE_ATTRIBUTE_NORMAL, NULL);
		Sleep(60);
		CloseHandle(hFile);
	}
}

void thread_two(void *wait)
{
	Sleep((DWORD)wait);
}

int main(int argc, WCHAR* argv[])
{
	_beginthread(thread_two, 0, (void *)WAIT);
	_beginthread(thread_one, 0, (void *)WAIT);

	Sleep(WAIT);

	return 0;
}
```

```
// AppT6 - Accelerated Windows Software Trace Analysis training
// Copyright (c) 2012 Memory Dump Analysis Services
// GNU GENERAL PUBLIC LICENSE
// http://www.gnu.org/licenses/gpl-3.0.txt

#include "stdafx.h"
#include <windows.h>
#include <process.h>

#define WAIT (1000*60)

void thread_one(void *wait)
{
	DWORD dwLoops = (DWORD)wait;
	DWORD dww = 0, dwCode = 0;

	HANDLE hFile = CreateFile(L".\\AppT6.dat",  GENERIC_READ | GENERIC_WRITE, 0, NULL,
				OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
	if (GetLastError() == ERROR_ALREADY_EXISTS)
	{
		ReadFile(hFile, &dwCode, sizeof(dwCode), &dww, NULL);
	}

	while (--dwLoops)
	{
		if (++dwCode > (5*(DWORD)wait)/2)
		{
			throw dwCode;
		}
		SetFilePointer(hFile, 0, NULL, FILE_BEGIN);
		WriteFile(hFile, &dwCode, sizeof(dwCode), &dww, NULL);
		Sleep(1);
	}

	CloseHandle(hFile);
}

void thread_two(void *wait)
{
	Sleep((DWORD)wait);
}

int main(int argc, WCHAR* argv[])
{
	_beginthread(thread_two, 0, (void *)WAIT);
	_beginthread(thread_one, 0, (void *)WAIT);

	Sleep(WAIT);

	return 0;
}
```