Working Toward a White Box Approach
Transforming Complex Legacy Enterprise Applications

It has long been the assumption that, once built, all computer systems are Turing machines—that is, devices that perform one small, deterministic step at a time—and are, therefore, deterministic in nature.¹ For every set of inputs, there is always the same set of outputs no matter what part of the system.

Practical application development involves manual programming of the requirements needed to achieve a business objective. The process of programming has never been free of defects and, therefore, entails a high degree of uncertainty.² Over the years, quality processes have been developed to detect defects at every step, and all system development life cycles require thorough testing phases.

One approach to systems and application development utilizes a set of transparent closed-loop tools to eliminate uncertainty and defects within a well-bounded set of processes. Central to this approach is a logical systems repository (LSR), which, when implemented, allows for automation to drive application development and legacy systems migration from requirements to production release, which helps achieve better accuracy, consistency, and, consequently, increased speed and lower costs. The goal is to represent every element of complex computer systems (i.e., data, procedures, infrastructure) as data fully available for analysis and modeling. Alternate software development life cycle approaches can leverage that capability.

What Is a White Box and What Is an LSR?

Virtually every important business process enabled
by computer systems at an organization involves a certain level of complexity. Complexity arises from how computer systems are constructed and connected to each other. There are six levels of system complexity:

- **Level 1: Size**—Large modules/components composed of single blocks of >50 lines of code signal complexity. Size can be an indicator that the block of code is dealing with multiple responsibilities and needs to be broken down and componentized. Keeping track of when a component needs to be updated or connected becomes more difficult with a large body of code.

- **Level 2: Hard-wired interfaces**—Hard-wired components with brittle interfaces have reduced reusability, which potentially leads to the development of redundant code. When any component undergoes a change, the corresponding components that interface with it must be incorporated in appropriate updates and testing (figure 1). Studies show that interfaces with low cohesion tend to degrade the cohesion of all the classes implementing them, compared to the classes that do not implement interfaces, significantly improving maintainability.3

- **Level 3: Depth of the call stack**—When components within a system call other components, increases in this chain of invocations result in a longer control flow (call stack) (figure 2), and complexity of dependencies rises. When a downstream component needs to be changed or connected, all upstream and downstream impacts need to be considered. Measures such as implementation length and volume represent different levels of complexity and can impact several quality factors, including maintainability, efficiency and performance.4

- **Level 4: Hierarchy of the call stack**—When components are called by multiple other components, the hierarchy (multiple callers/parents) of the call stack becomes tangled (figure 3). A single child component can be shared by multiple parent components. When a child must be changed or connected, the impact to all parents needs to be considered. Measured in McCabe’s cyclomatic complexity, call stack hierarchy has been found to correlate with several quality factors, such as maintainability, reliability and testability.5

- **Level 5: Multiple technologies and applications**—When business processes cross generations of applications and infrastructure, the existence of multiple technologies that affect process and organizational boundaries can obfuscate the call stack (figure 4). How computer systems are implementing a business process becomes increasingly difficult to document and trace because a component in one process or language can be calling a component in a completely different process or technology. Tracing these dependencies manually is difficult and requires team members conversant in multiple programming languages and systems.

- **Level 6: Black box components**—Multiple vendor products (black boxes) further complicate traceability because how code is flowing through these closed products and how they are processing information is unknown (figure 5).

**Limitations of a Manual Approach to Design and Development**

Large, complex IT projects tend to fail. In a study published by The Standish Group, researchers reviewed more than 50,000 IT projects between

---

**FIGURE 1**
Changing Hard-Wired Components

![Changing Hard-Wired Components](image1)

**FIGURE 2**
Increasing Call Stack Depth

![Increasing Call Stack Depth](image2)
1992 and 2004 and found that only 29 percent could be classified as successes. An enabler for these failures is often found to be complexity itself. Due to complexities, such as the six types described, many large programs are fraught with risk, including security vulnerabilities, because it is not humanly possible for architects and developers to manually keep track of unlimited amounts of dependencies and relationships in their visual short-term memory. Changes are often made to components without a full view of all interconnected components in the same system or in upstream/downstream systems. Project teams must rely on either the prior experience of subject matter experts (SMEs) or integration testing to discover such dependencies and the ensuing unintended consequences.

The high costs of manual migration—data migrations are often 200 percent of the cost of system acquisition—force organizations to either postpone modernization or resort to spot upgrades necessitated by obsolete systems, resulting in a patchwork of bandages holding a brittle system together. There is also a high degree of correlation between Halstead’s effort, one of many measurements developed by Maurice Halstead at IBM to determine the mental effort required to develop or maintain a program, and McCabe’s cyclomatic complexity measures.

Complexities impact quality; introduce the risk of disruption to process and data; create uncertainty in terms of project timeline, milestone performance and resource requirements; and, ultimately, increase costs.

Furthermore, complex systems typically exhibit additional challenges for manual implementation, such as:

- Deployments are large in terms of application code, systems integrations and databases changes.
- Business logic is difficult to extract and is often duplicated, sometimes inconsistently.
- Hundreds of contributors modify source code using inconsistent techniques and coding patterns.
- Obsolete or dead code is not used, but it still needs to be maintained, increasing the nonproductive total cost of ownership.
- Documentation can be out of date with weak traceability to the source code.

**An Alternative Approach**

An alternative to the manual approach is to use a white box approach that exposes all the internal control flows, data lineage, traceability between business processes, functions supported by a system, flow of logic, business rules and data elements of a system as data and lets those data points drive decisions in architecture, design, development and testing. Elements of data include business functions, use cases and user stories, business logic and business rules, code flows and cyclomatic paths, data elements and data lineage, interfaces and dependencies, and infrastructure.

> “The high costs of manual migration...force(s) organizations to either postpone modernization or resort to spot upgrades necessitated by obsolete systems...”
To measure the benefits of a white box approach to system migration and interconnectivity, an example of a specific project in which the system supports mission-critical business processes can be used. This system is complex and involves multiple technologies. It handles, among other functions, the processing of pharmaceutical prescription claims. This system implements regulated processes such as drug utilization reviews to determine if a drug prescribed to a patient by a physician is safe, considering other drugs, conditions, allergies; prior authorization; coverage; and pricing. Errors can result in significant harm to patients. Several studies have found that when implemented correctly, computerized physician order entry (CPOE) systems reduce medication errors and overall patient harm as opposed to manual recording.16

When applied to these functions, even a system that conforms to Six Sigma, a set of techniques and tools for process improvement, exhibits a level of accuracy that falls short of the quality level required by the enterprise. At that level of accuracy, organizations that process more than a billion prescriptions a year will encounter more than 3,000 errors per year, a level that is not acceptable when patient lives are at stake.17 As noted, on account of the increased accuracy of a data-driven design, implementation and testing process supported by a white box approach, it is an optimal solution when a system needs to be modernized (e.g., from Broadvision, C++, CORBA to contemporary technologies); updated for new regulations (i.e., the US Health Insurance Portability and Accountability Act [HIPAA] D.0 or International Classification of Diseases [ICD].10); connected to other consumers (for white labeling the underlying dependencies. This approach enables project teams to determine how business processes are being supported by the associated computer systems; how systems implement use cases; how the use cases are implemented across components, technologies, languages and systems; and the code, data and infrastructure assets involved. For an existing system, this information needs to be extracted with the use of parsers that will not only perform lexical and semantic analyses but span these control and data flows automatically.

This approach is a powerful solution for validating model-based software engineering principles14 and is now more relevant than ever as technical debt and the associated complexity of IT systems have quickly multiplied. White box testing approaches are also important for the validation of big data systems.15 This is because the level of testing paths to be exercised increases rapidly with size and complexity of data and is not easy to comprehensively visualize manually as mentioned. Furthermore, in the absence of visibility into impact from changes to a specific area of the system, it is difficult to assess the level of testing required in terms of number of paths of logic on the data constructs and corresponding scenarios to be tested. Managing the technical debt via replatforming or modernization requires an effort to get to a white box state that exposes internal control flows for all the collaborating systems within an enterprise.

When working with black box components, project teams do not have access to component source code or underlying data structures, which adds complexity. The white box approach analyzes and maps the configuration of such components and exhaustively details all interactions with those components. This level of visibility exposes all relevant flows involving the black box components and any related defects. The data are then stored in a logical system repository (LSR)—a database that houses all the previously listed elements of the internal controls and their relationships.

“The white box approach analyzes and maps the configuration of such components and exhaustively details all interactions with those components.”

When working with black box components, project teams do not have access to component source code or underlying data structures, which adds complexity. The white box approach analyzes and maps the configuration of such components and exhaustively details all interactions with those components. This level of visibility exposes all relevant flows involving the black box components and any related defects. The data are then stored in a logical system repository (LSR)—a database that houses all the previously listed elements of the internal controls and their relationships.

 dependencies. This approach enables project teams to determine how business processes are being supported by the associated computer systems; how systems implement use cases; how the use cases are implemented across components, technologies, languages and systems; and the code, data and infrastructure assets involved. For an existing system, this information needs to be extracted with the use of parsers that will not only perform lexical and semantic analyses but span these control and data flows automatically.

This approach is a powerful solution for validating model-based software engineering principles14 and is now more relevant than ever as technical debt and the associated complexity of IT systems have quickly multiplied. White box testing approaches are also important for the validation of big data systems.15 This is because the level of testing paths to be exercised increases rapidly with size and complexity of data and is not easy to comprehensively visualize manually as mentioned. Furthermore, in the absence of visibility into impact from changes to a specific area of the system, it is difficult to assess the level of testing required in terms of number of paths of logic on the data constructs and corresponding scenarios to be tested. Managing the technical debt via replatforming or modernization requires an effort to get to a white box state that exposes internal control flows for all the collaborating systems within an enterprise.

When working with black box components, project teams do not have access to component source code or underlying data structures, which adds complexity. The white box approach analyzes and maps the configuration of such components and exhaustively details all interactions with those components. This level of visibility exposes all relevant flows involving the black box components and any related defects. The data are then stored in a logical system repository (LSR)—a database that houses all the previously listed elements of the internal controls and their relationships.

dependencies. This approach enables project teams to determine how business processes are being supported by the associated computer systems; how systems implement use cases; how the use cases are implemented across components, technologies, languages and systems; and the code, data and infrastructure assets involved. For an existing system, this information needs to be extracted with the use of parsers that will not only perform lexical and semantic analyses but span these control and data flows automatically.

This approach is a powerful solution for validating model-based software engineering principles14 and is now more relevant than ever as technical debt and the associated complexity of IT systems have quickly multiplied. White box testing approaches are also important for the validation of big data systems.15 This is because the level of testing paths to be exercised increases rapidly with size and complexity of data and is not easy to comprehensively visualize manually as mentioned. Furthermore, in the absence of visibility into impact from changes to a specific area of the system, it is difficult to assess the level of testing required in terms of number of paths of logic on the data constructs and corresponding scenarios to be tested. Managing the technical debt via replatforming or modernization requires an effort to get to a white box state that exposes internal control flows for all the collaborating systems within an enterprise.

When working with black box components, project teams do not have access to component source code or underlying data structures, which adds complexity. The white box approach analyzes and maps the configuration of such components and exhaustively details all interactions with those components. This level of visibility exposes all relevant flows involving the black box components and any related defects. The data are then stored in a logical system repository (LSR)—a database that houses all the previously listed elements of the internal controls and their relationships.

Case Study: How a White Box Makes a Difference

FIGURE 5
Complicating Traceability

| Javascript in Browser | Javascript Server Pages | API | Black Box Product | Database |

“The white box approach analyzes and maps the configuration of such components and exhaustively details all interactions with those components.”

When working with black box components, project teams do not have access to component source code or underlying data structures, which adds complexity. The white box approach analyzes and maps the configuration of such components and exhaustively details all interactions with those components. This level of visibility exposes all relevant flows involving the black box components and any related defects. The data are then stored in a logical system repository (LSR)—a database that houses all the previously listed elements of the internal controls and their relationships.

When working with black box components, project teams do not have access to component source code or underlying data structures, which adds complexity. The white box approach analyzes and maps the configuration of such components and exhaustively details all interactions with those components. This level of visibility exposes all relevant flows involving the black box components and any related defects. The data are then stored in a logical system repository (LSR)—a database that houses all the previously listed elements of the internal controls and their relationships.

When working with black box components, project teams do not have access to component source code or underlying data structures, which adds complexity. The white box approach analyzes and maps the configuration of such components and exhaustively details all interactions with those components. This level of visibility exposes all relevant flows involving the black box components and any related defects. The data are then stored in a logical system repository (LSR)—a database that houses all the previously listed elements of the internal controls and their relationships.

When working with black box components, project teams do not have access to component source code or underlying data structures, which adds complexity. The white box approach analyzes and maps the configuration of such components and exhaustively details all interactions with those components. This level of visibility exposes all relevant flows involving the black box components and any related defects. The data are then stored in a logical system repository (LSR)—a database that houses all the previously listed elements of the internal controls and their relationships.
and data elements were scanned, identified, verified with subject matter experts (SMEs) and stored in the LSR database, which was then used to drive implementation of the future state.

Making a Difference in Visibility and Transparency: Reverse Engineering

The LSR ensures high confidence in full code coverage. It ensures that no functionality is lost from the current state, regardless of its disposition in the future state—e.g., rewrite, redesign or remove. The factory code parsers handle all programming languages and code file types used in the current state.

In the case study, this approach reduced reliance on SMEs for initial discovery activities, allowing them to focus more on review and validation. Moreover, it clearly identified and listed all reverse-engineering tasks required. Further, the factory process/tools assisted the SMEs in performing data lineage analysis within and across all technology platform boundaries. Data lineage analysis detailed how each data element involved in the call stack would evolve through its life cycle.

For this case study, a system was selected that captures and manages patient preferences in pharmacy benefit management, and it was modernized and migrated to a new architecture. It important to note the size (figure 6) and complexity of the system (figure 7).

The metrics presented in figures 6 and 7 provide a view into how difficult it would be for an architect or developer to manually assess dependencies, impacts and optimal methods for updates, migration or interconnectivity.

Automated tools were used to extract the call stack across all languages and systems. The infrastructure footprint and a comprehensive inventory of functions, use cases, business rules, components and data elements were scanned, identified, verified with subject matter experts (SMEs) and stored in the LSR database, which was then used to drive implementation of the future state.

FIGURE 6
Size of the System

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>Language</th>
<th>Number of Components</th>
<th>Total Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>JavaScript</td>
<td>55</td>
<td>39,738</td>
</tr>
<tr>
<td>2</td>
<td>JSP</td>
<td>81</td>
<td>37,380</td>
</tr>
<tr>
<td>3</td>
<td>C++</td>
<td>483</td>
<td>131,248</td>
</tr>
<tr>
<td>4</td>
<td>XSL/MQ</td>
<td>27</td>
<td>24,253</td>
</tr>
</tbody>
</table>

FIGURE 7
Complexity of the System

<table>
<thead>
<tr>
<th>Level</th>
<th>Language</th>
<th>Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Size</td>
<td>Max lines of code is 9,751</td>
</tr>
<tr>
<td>2</td>
<td>Hard-coded interfaces</td>
<td>Present</td>
</tr>
<tr>
<td>3</td>
<td>Depth of call stack</td>
<td>Max: 11, Median: 5</td>
</tr>
<tr>
<td>4</td>
<td>Hierarchy of call stack</td>
<td>64 percent of 9,000 functions have more than one parent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Max parents for a business function: 42</td>
</tr>
<tr>
<td>5</td>
<td>Number of technologies</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>Vendor products</td>
<td>1 (Datapower)</td>
</tr>
</tbody>
</table>
The approach involved code parsing that spanned platforms and process boundaries yet allowed users to navigate and reverse navigate through the call hierarchy within and across all technology platforms. The SMEs performed ad hoc and on-demand analysis and assessed the impact of any change. The LSR acted as the sole central source of information required for forward engineering activities.

A Different Forward Engineering Life Cycle Driven by the LSR

With full and comprehensive visibility into how the system was implemented, a white box approach was used to build out the future state, including modernization, refactoring, migration and interconnectivity.

The white box approach ensured that all functionality was preserved, while allowing a coarse functionality to be broken down into granular units as required. The approach helped SMEs and developers to achieve clear separation of architectural concerns during the design and coding phases. The approach delivered reports that showed components that were either not forward engineered or that lacked traceability from the target state back to the current state. The visibility improved management of the developer workload and made engineering productivity more predictable.

Furthermore, the LSR was used as a source of information for all systems components, business rules and data elements to auto-generate code either partially or completely per target design. The LSR served as the meta model that code generators typically require to convert functions to code objects and components; they sourced information from the LSR about data elements inbound and outbound, business rules, and the hierarchy of callers. When necessary, the functionality was enhanced manually to increase its business fit and value. The business SMEs/analysts were able to inject new user stories or update an existing user story as required. The approach also focused on the performance of the forward-engineered functionality, such that it matched or enhanced the current production performance.

The White Box Implementation Process Used in the Case Study

The implementation of the white box process with regard to the pharmacy case study involves reverse and forward engineering using LSR.

Reverse Engineering

At a high level, the reverse engineering phase is a six-step process (figure 8) that establishes the LSR and the reverse engineering dashboards. The dashboard views are customized to the needs of various project roles (e.g., views that are specific to business analysts, quality engineers and business users). In addition, users are allowed to run prebuilt or custom queries over LSR for analysis.

Forward Engineering

The forward engineering phase (figure 9) involves the future state design of every component in the LSR, along with its traceability to its current state. Every component is assessed for its fit in the future state architecture and moved to the suitable architecture layer in the future state. For example, business logic trapped in web pages could be moved to an appropriate restful application programming interface in the new and modern architecture, making it easier to maintain, reuse and improve user experience. This enables business transformation and enhancements as opposed to a plain
Benefits

The case study exhibits the numerous benefits of the white box approach, including:

- Identification of where a change in the system can be performed the most efficiently and safely and be best integrated into existing business capabilities
- Assessment of impact to all system components, interfaces and data elements
- Traceability to a new system to ensure that no business rule is inadvertently left behind
- Determination of the sequence and logical grouping of components that need to be changed
- Identification of opportunities for secure enclaves based on actual data flows to enhance security
- Demonstration of the impact to downstream systems across multiple interfaces
- Prediction of effort and time and, thereby, resource needs
- Accurate and definite selection of elements of the approved software development life cycle management standard operating procedure (SOP) of the organization, reducing duplication of work and improving team efficiency
- Creation of maps with cyclomatic flow, resulting in comprehensive test coverage, including test cases for unit testing and integration testing and for business simulation and user acceptance
- Selection of data to be used for testing to help assert certain branches of code to exercise all the code (further improving test coverage)

It is important to measure and understand the impact of these incremental capabilities in the effort, cost, elapsed time, resource requirement and SME dependence. An improvement in accuracy of the migrated system will represent better patient experience and eliminate disruptions that can have clinical impacts by enabling users (i.e., patients and physicians) faster access to their prescriptions and accurate fulfillment and delivery of their medicines.

Results

The patient preferences function within the service delivery system for pharmacy benefit management...
was replatformed using both the white box approach and traditional approaches to support an objective comparison between the two. The white box approach resulted in a zero-defect release because all scenarios of the system’s behavior were modeled and exercised. Application of a large number of controls eliminated disruption in the reengineering and system connectivity to downstream applications.

Although improved accuracy was the primary objective, a number of other improvements also occurred using the white box approach, compared to the traditional approach to system migration, including:

- The number of person days required to analyze the existing code base was reduced from 55 to 10.
- The number of person hours to migrate the functionality was reduced from 4,629 to 1,612.
  - The project team size was reduced by 50 percent (from 14 to 7).
  - The cycle time was reduced by about 30 percent (down from 2 months to one month and a half).
- There was a reduction of about 80 percent in person hours of SME time.

**FIGURE 10**

Planning the White Box Approach

<table>
<thead>
<tr>
<th>Element of the White Box Approach</th>
<th>Objective Description</th>
<th>Purpose Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evaluate the target architecture for alignment with enterprise strategy.</td>
<td>Management: Align, Plan and Organize (APO) AP003 Managed Enterprise Architecture</td>
<td>Represent the different building blocks that make up the enterprise and its interrelationships, as well as the principles guiding their design and evolution over time to enable a standard, responsive and efficient delivery of operational and strategic objectives.</td>
</tr>
<tr>
<td>Perform a failure modes and effects analysis on all the failure modes determined as unintended cyclomatic flows discovered.</td>
<td>Governance: Evaluate, Direct and Monitor (EDM) EDM03 Ensured Risk Optimization</td>
<td>Ensure that the enterprise's risk appetite and tolerance are understood, articulated and communicated and that the risk to the enterprise value related to the use of I&amp;T is identified and managed.</td>
</tr>
<tr>
<td>Determine a sequence of changes in atomic groups based on dependencies, value of functionality, effort and cost eliminating risk of disruption.</td>
<td>Management: AP002 Managed Strategy</td>
<td>Support the digital transformation strategy of the organization and deliver the desired value through a road map of incremental changes. Use a holistic I&amp;T approach, ensuring that each initiative is clearly connected to an overarching strategy. Enable change in all different aspects of the organization, from channels and processes to data, culture, skills, operating model and incentives.</td>
</tr>
<tr>
<td>Generate a plan including milestones, timeline, effort and cost while using a closed-loop process to capture certification and test scenarios upfront.</td>
<td>Management: AP005 Managed Portfolio</td>
<td>Execute the strategic direction set for investments in line with the enterprise architecture vision and I&amp;T road map. Consider the different categories of investments and their alignment with strategic objectives, enterprise worth and risk. Move selected programs into the active products or services portfolio for execution. Monitor the performance of the overall portfolio of products and services and programs, proposing adjustments as necessary in response to program, product or service performance or changing enterprise priorities.</td>
</tr>
</tbody>
</table>
Implementation and Monitoring
The next step is to apply corresponding controls during the build and monitor phases (figure 11).

These control objectives are implemented in a highly transparent, data-driven and, therefore, audit-friendly approach. These controls not only help optimize outcomes, but they also demonstrate the underlying drivers for decision making. Therefore, they improve the systems' recovery and resilience; in a changing situation, those underlying factors can be updated to rapidly come up with a new set of designs, plans and control regimes.

Conclusion: Delivery of Incremental Value
With today’s computing capability and automated discovery tools, it is possible to create a white box closed-loop methodology to fully realize the promise.

- The white box approach determined that a target of 90 percent code coverage was required based on dependencies. All branches were exercised in testing.
- Cycle time was reduced by more than 40 percent.

Implementing COBIT Controls
The white box approach enables and empowers implementation of very specific controls that improve an organization’s capability to govern, plan for an optimal outcome, build accurately, and measure effectiveness and efficiency of delivery. Specifically, COBIT can be used with the white box approach using a recommended set of objectives.19

Planning
The first step to implement a controls regime that ensures accuracy and predictability is to plan for the optimal outcome while eliminating risk (figure 10).

<table>
<thead>
<tr>
<th>FIGURE 11</th>
<th>Implementing the White Box Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Element of the White Box Approach</strong></td>
<td><strong>Objective Description</strong></td>
</tr>
<tr>
<td>Ensure traceability for every function and business rule from the current state to the future state aligned with the plan.</td>
<td>Management: Build, Acquire and Implement (BIA) BAI02 Managed Requirements Definition</td>
</tr>
<tr>
<td>Provide the LSR through an organizational change management process so that everyone on the team is at par with the subject matter experts.</td>
<td>Management: BAI08 Managed Knowledge</td>
</tr>
<tr>
<td>Utilize certification results to measure the value of functions delivered and track utilization in production to reconcile value delivered.</td>
<td>Management: BAI11 Managed Projects</td>
</tr>
<tr>
<td>Provide audit outputs from the LSR to demonstrate sequence, coverage, traceability and the integrity of supported business processes.</td>
<td>Management: Monitor, Evaluate and Assess (MEA) MEA03 Managed Compliance With External Requirements</td>
</tr>
</tbody>
</table>
of control frameworks such as COBIT. Once business and nonfunctional requirements are fully described in an LSR, systems can be fully automated, with all the benefits that accrue when replacing manual effort with technology, such as better accuracy, lower cost and a faster turnaround. White box methods can reduce dependence on scarce resources of all types (technical and business), shorten cycle times significantly and eliminate technical defects prior to release.

In addition, once a system is fully characterized in an LSR, "what if" and other analyses can be used to assess the impact of new requirements, services refactoring and migration to new platforms. White box engineering is a valuable process that can be used to fulfill the long-held promise of deterministic systems. The impact of changes when interconnected is no longer a mystery.

Furthermore, the LSR enables more advanced business capabilities. In the system discussed herein, the LSR allowed for both a deeper analysis and traceability constructs for how each claim was processed through the underlying systems, including adjudication. This capability, termed "claims neurology," allowed an implementation of long short-term memory (LSTM) deep learning neural nets to effectively predict margin performance of future claims.

Endnotes


3 Abdeen, H.; H. Sahraoui; O. Shata; “How We Design Interfaces, and How to Assess It,” 2013 Institute of Electrical and Electronics Engineers (IEEE) International Conference on Software Maintenance, Netherlands, September 2013, https://ieeexplore.ieee.org/document/6676879


5 Ibid.


16 Singh, H.; S. Mani; D. Espadas; N. Petersen; V. Franklin; L. Petersen; “Prescription Errors and Outcomes Related to Inconsistent Information Transmitted Through Computerized Order Entry: A Prospective Study,” Archives of Internal Medicine, vol. 169, iss. 10, 25 May 2009, https://jamanetwork.com/journals/jamainternalmedicine/fullarticle/773518

