Accelerating Application Refactoring

A Practical Guide

Philippe Guerin, Senior Software Architect, CAST

The Trouble with Application Refactoring

"Why is it taking so long?"
"Do we really need a 20-person team on this?"
"Why is the changed system misbehaving?"

We're all faced with questions like these from our business stakeholders and leadership. They know our systems are complex, but have little appreciation for what that means, nor patience or funding for us to modernize. Not to mention the years of cost savings and turnover that deplete the collective knowledge of our legacy software. We know we must refactor some of our core systems in order to keep up with "digital." But, there's no pleasure in having to resort to trial-and-error methods, software archeology taking costly wrong turns, and at times unwittingly introducing production defects.

In this guide we share how the essential activities of application refactoring can be significantly accelerated by applying Software Intelligence. We, at CAST, have a long history with application modernization and refactoring, so as a first point we establish the typical refactoring approaches we see enterprises take. In these approaches, we've found a set of common motions, which are typical activities undertaken to refactor applications. We then explore these motions in some detail, how they are typically undertaken today, and how to accelerate them.

CAST is the company behind the world's most advanced 'MRI for Software', which essentially reverse engineers and automatically 'understands' software systems built with any mix of 3GL, 4GL, Mobile, Web, Middleware, Framework, Database, Mainframe technologies. Throughout this document, we use examples from CAST Imaging, our interactive architecture visualization capability, to illustrate the accelerated refactoring motions.

Not all Refactoring Approaches Are the Same

Probably the most common reason to refactor is to make the application more modular and hence easier to change. Another common reason is to take advantage of new capabilities available, either as frameworks or PaaS services from the major cloud providers. And of course, refactoring is sometimes undertaken to fix specific issues, or to reduce maintenance overhead or to beef up the security posture of an application.

Whatever the case, refactoring by its very nature is driven by non-functional aspirations. Even if the end goal is to deliver higher levels of functionality more quickly and robustly. The extent of refactoring will depend on the level of disruption an enterprise or the IT organization is willing to tolerate. Among all

existing motivations to refactor thus far, we've seen several refactoring archetypes. Below, we describe these common approaches to refactoring, and explain some of the typical activities, or "motions", each of these approaches entails.

So, let's start with the refactoring approaches.

Do Nothing

With all the talk about modernization and cloud, it would seem this is not really an approach. Yet, in our experience, this is the most common refactoring approach taken by most enterprises today. Remember, we're talking about application refactoring here. Wrapping an application up and moving the virtual machine to the public cloud is precisely a "do nothing" approach. If you are accustomed to your enterprise system and are generally satisfied with its performance, you may choose not to make any enhancements. However, this can still be a missed opportunity to gain a competitive edge in the long run. This approach does not require any specific application refactoring motions.

The Band Aid

As the name suggests, this approach consists of focusing on problem areas and implementing stopgaps to get the system working in new ways. This approach is also sometimes taken to opportunistically apply modern technology to legacy systems—melding the old and the new to improve the organization's performance and productivity. These quick fixes are usually low risk as they're surgical and simply put an end to specific issues. But, if not implemented with a good sense of impact on the integration points to the rest of the existing system, the approach could lead to unresponsive components. Typically, work is done at the boundaries of the legacy system because the team doesn't control or even understand the core system. Some examples of motions that take place in this approach are:

- Framework replacement or insertion
- Rewriting a vulnerable component
- Putting in a DAO (data access object)
- Adding a wrapper to a legacy component

Gradual Replacement

Wholesale application modernization doesn't need to be done overnight. Some organizations take a planned approach to upgrade legacy systems one piece at a time, perhaps leveraging an open source UI component to enhance the user experience or a database access framework to improve performance. The benefit of this approach is that you can quickly model future state by using off the shelf components to determine what works and what doesn't. No need to overhaul the entire system

when only certain elements need to be optimized. You can break the system into zones or portions—an advantage for those who want to test the waters when it comes to legacy system enhancements. However, having too many components and integrations may create compatibility and complexity issues. Some examples of motions that take place in this approach are:

- Functional decomposition of existing system
- Review of the "as-is" architecture
- Plan the "go-to" architecture
- Identify candidates for componentization
- Remove, replace or add a framework

Rip and Replace

For this approach, transformation is the bottom line. To take this approach there is usually a somewhat urgent need to retire the legacy system and rebuild a new one. Rip and replace is a complete overhaul, which may be very risky from a business stakeholder perspective. This is an aggressive approach to extend your organization's competitive edge. Although the solution is "high risk, high reward", the major changes may create organizational adoption challenges for teams that are accustomed to their legacy processes. Some examples of motions that take place in this approach are:

- Review of the "as-is" architecture
- Plan the "go-to" architecture
- Rewrite or add a module, features or a component

Continuous Refinement

This is the ideal refactoring cadence that should be the eventual end state for all critical applications. You can think of it as maintenance refactoring. When in this mode, the organization knows the application well, there isn't a need to make drastic changes, and the team is comfortable making minor adjustments. These include cleaning, consolidating, and updating code. The system is in a healthy state and merely undergoes modernization to remove problem areas and inefficiencies, or to adjust to performance requirements. There is no urgency, and most changes are highly proactive. This is ideal for organizations whose existing IT systems serve their purpose well. The only danger of this approach is to fall too far behind the curve of the tech landscape, finding it hard to get qualified resources to support and update legacy systems in the future. Some examples of motions that take place in this approach are:

- Code clean up and bug fixing
- Rewrite, expose or add a component
- Replace a framework
- Optimize transactions

Successful refactoring is part art and part process. Because in many cases you don't know what you don't know about your existing systems, refactoring projects are hard to predict and force into a specific process. In the next section we deconstruct the typical refactoring motions and explain how we can make them more procedural, predictable and efficient.

Common Motions – Summarized

Whether it's a full rewrite of a legacy system or a move to public cloud, some activities are common to most types of refactoring projects. We want to distill these common refactoring motions – the typical activities that one has to do when refactoring a large, complex codebase. There are probably many ways to describe the typical activities that take place in modernization or refactoring projects. In our experience, these common motions broadly fall into two categories. **Analysis**, that is to understand the current state or plan the future state, and **Action**, which is to actually do the refactoring in the codebase.

Analysis Motions

1. Review the "as-is" architecture – Discovery of the "as-is" architecture of the system in order to understand the overall design as it's been implemented in the existing application. That review can lead to both technical and functional discovery of the system and it naturally precedes most of the other refactoring motions.

2. Plan the "go-to" architecture – A new architecture should be defined explicitly enough that developers can follow along, and architects can check compliance of implementation. If components are targeted for replacement, all connections to those components can be tracked and goals set for achieving "zero connections".

3. Identify candidates for componentization – This entails looking at the existing software architecture to see where a functional component can be turned into a microservice or a better separation of concerns can be introduced.

4. Identify obsolescence or vulnerabilities – this involves analysis of frameworks and third party or open source components to determine where the most urgent risks are.

Action Motions

5. Decouple a community of components – this can take many forms and is often linked to isolating a service, an API, a transaction or a whole layer.

6. Rewrite, expose or add a component – Whether it's for vulnerability, or to update the functionality and business rules, a component-wise approach to refactoring often makes sense.

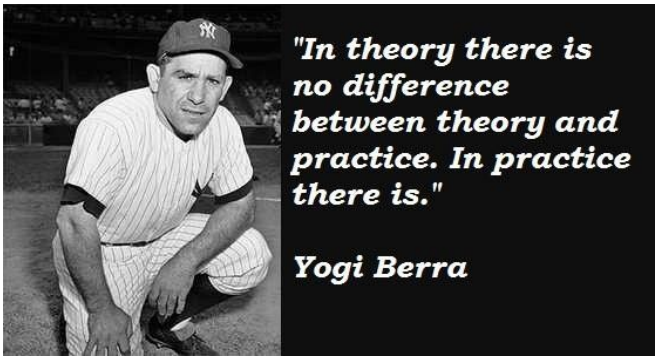
7. Insert a framework – A new framework can be useful for common functionality – better tested, easier to maintain and more future-ready than building your own.

These are not representative of all the motions that take place in application refactoring. Just the most common we've seen in many years working on modernization with enterprise IT applications teams.

Common Motions – Explored

In this section we look at each of these motions in more detail, exposing the challenges in each one and explaining how each motion can be expedited through machine-assisted Software Intelligence. For each motion we describe the current status quo – that is the typical approach you might take, and what that approach looks like with Software Intelligence. These approaches are placed side-by-side, so we can easily compare.

Motion 1: Review the "as-is" architecture



Most of the time the existing system architecture is unknown. Even if there is some documentation, it's either not detailed enough or has fallen behind the implemented reality. Almost always the real "as-is" architecture in practice is not the same as what you might think. This motion is usually a starting point for most of the Action motions, like decoupling components, or even for just planning the new architecture. Hence, you will see reference to this motion built into most of the remaining motions we describe in this section.

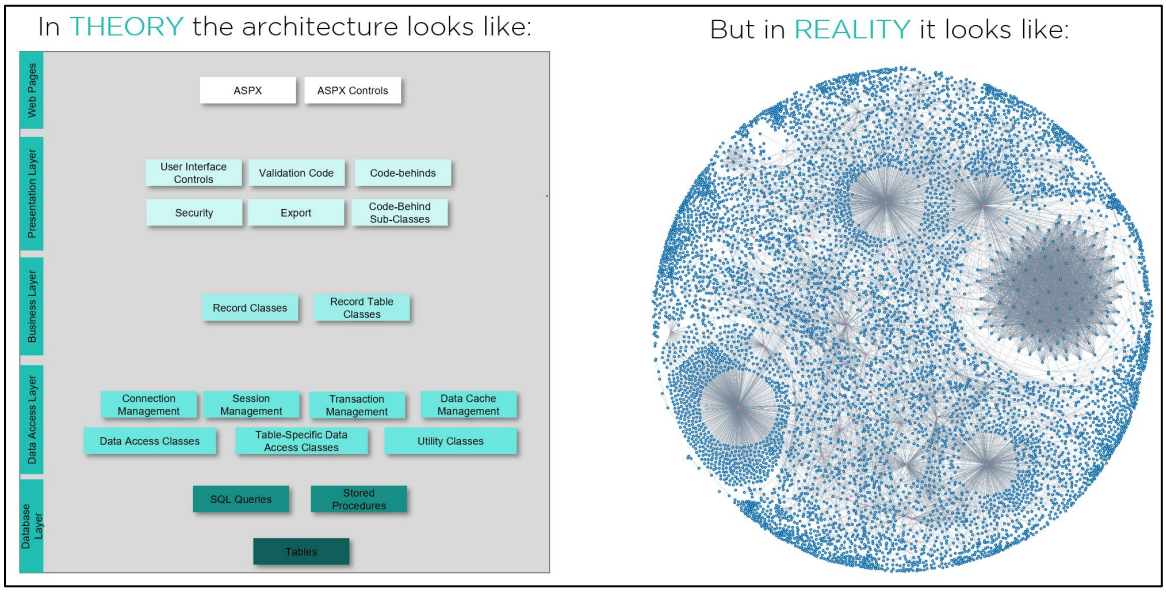


Figure 1 - Architecture document vs. the real "as-is" architecture

Architecture exploration is a part of all the motions described below. It's part of any work done in the context of large systems. This is probably the most pervasive motion in all application development and especially in maintenance. It is a necessity when doing any kind of refactoring of an existing system.

Status Quo	With Software Intelligence
The typical process of discovery is to start with one component, usually at the data end or the UX end of the chain, and then see where it leads. One would not discover the entire system, just the paths through it that are needed for the specific refactoring motion. Assuming bi-weekly sprints of 4-5 features each, each sprint is likely to require a week of "as-is" architecture exploration effort for senior developers. Sometimes several developers, if the application spans multiple tech stacks. It is not atypical to have to explore as many as 1000 components in the context of one refactoring project. Matter of **weeks**.	This becomes an entirely different approach, because the technical staff have access to the full map of the system. They can navigate to the impacted components for a refactoring project, or just navigate the architecture to understand how to better maintain portions of the system. The Software Intelligence would be available as a set of blueprints that can be quickly navigated and change impacts quickly explored. Rather than weeks, the process would take an hour or two to examine the same scope of around 1000 components. Matter of **hours**.

Motion 2: Plan the "go-to" architecture

Unless building something completely greenfield, an architectural diagram should always include existing components, as no system is built in isolation. Typically, this planning process is a step in one of the other, more action-oriented motions of refactoring. For example, as we see in Figure 4, as it relates to identifying obsolete components, we have a step at examining the current architecture, followed by planning the "go-to" architecture.

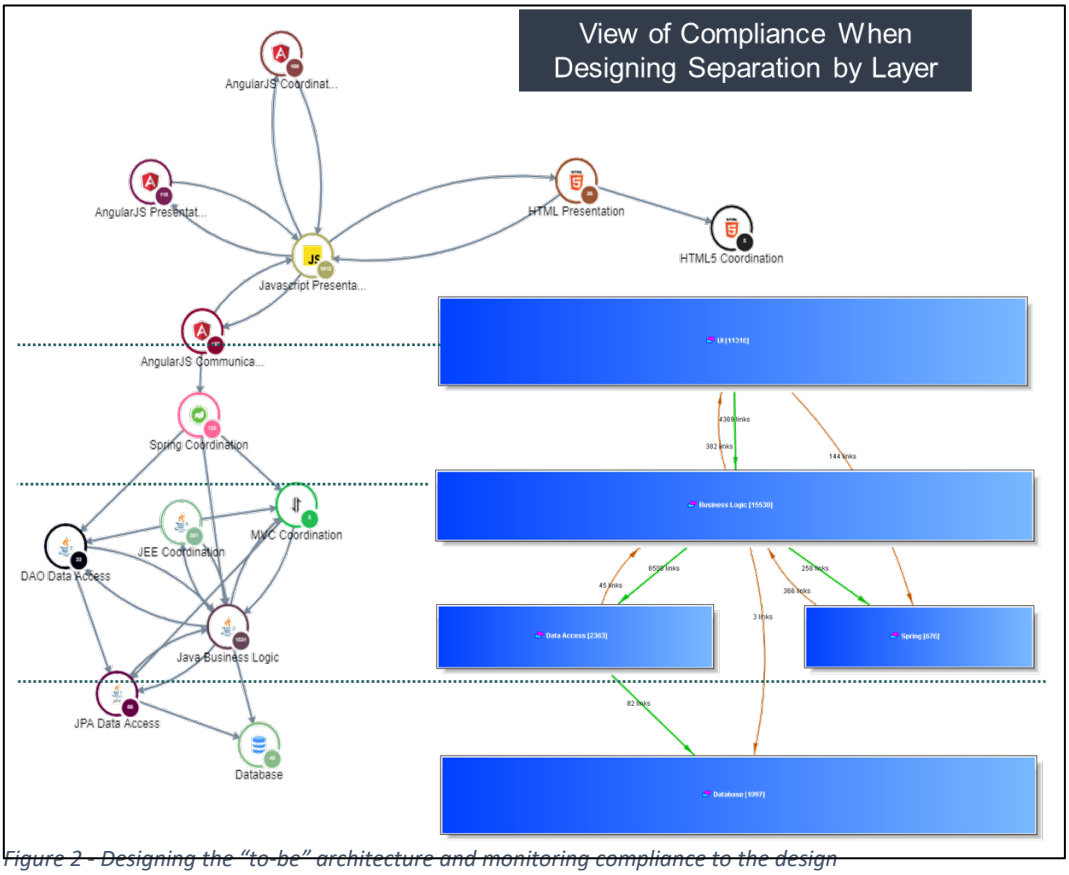


Figure 2 - Designing the "to-be" architecture and monitoring compliance to the design

A common approach to building modern systems is establishing a separation of concerns**.** The idea being that

each type of activity (user interface, business logic, data handling) is handled by a specific layer. This helps to modularize and isolate components of a system, creating layers of abstraction and making it more flexible to change. Those layers can be defined in the "go-to" architecture and checked for compliance. This is typically an exercise that involves multiple software architects to bring together an understanding of existing system architecture.

Like others, this motion starts with an understanding of the current architecture, but then gets into understanding the potential "to-be" scenarios and then sorting through these scenarios to agree on the best plan. Also, best in class teams build in a method to ensure the new architecture actually gets implemented by the development teams.

Status Quo	With Software Intelligence
A cabal of architects, likely representing different parts of the system, work together over a period of many weeks to formulate a visual description of a system that can be analyzed. Then a theoretical "to-be" architecture is drawn up and eventually agreed upon by the senior architects. That theoretical description looks something like the diagram on the left side of Figure 1. It is not connected to the current "as-is" architecture, so as the team implements it's possible they will run into some unknowns that will derail or delay the project. Also, during implementation the only way to check adherence to the new design is to keep a member of the architecture cabal allocated as a member of each dev team at least half time. This allows the designers to get something like direct oversight over implementation. **Months** of planning. **Man-years** of implementation oversight.	The same team of architects as in the status quo scenario will start with the "as-is" architecture as described in Motion 1. They look at different solution scenarios in the context of the "as-is" and trade off the ideal solution against the reality of getting there from the "as-is" state. Once a good compromise is found – the optimal tradeoff – then the "to-be" plan is drawn, as seen in Figure 2. The blue boxes in this figure represent the separation of concerns, codified as an architecture check in the Software Intelligence platform. These checks are looked at periodically, perhaps with each sprint, for compliance and progress. That is to make sure no new constructs are written that go against the "to-be" state and progress is made to rewrite constructs that currently go against the "to-be" architecture. **Days or weeks** of planning. **Man-days** of implementation oversight.

Motion 3: Identify candidates for componentization

The challenge is to look at all the interconnections between components to see where you might have good break points to cluster components. Typically, this is the common motion behind establishing a new microservice within or next to the legacy system. Usually the legacy is a big ball of COBOL or even Java, that has proven to work well over time, housing complex and well-tested business rules, and does not need to be discarded any time in the near future.

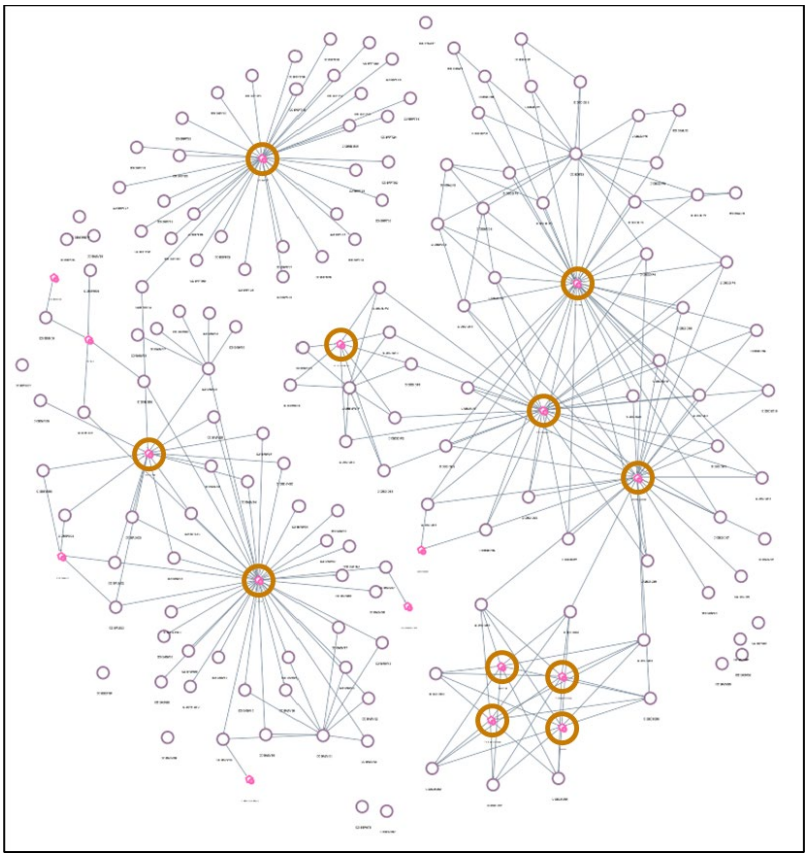


Figure 3 - The highlighted components that can be combined into a separate entity

The process here is typically to identify a cluster of components in a specific functional area, which can be "cordoned off" into a new layer, API or microservice. Then the impacts of that change need to be assessed and the project can then be estimated and implemented.

The team will typically start with a functional entry point and explore all the components going down the chain of that function. Then look at all the related areas and figure out which components need to also be reviewed to either include inside the new microservice, or to be interfaced to it from the outside. Once the microservice is zoned off, an impact analysis project is undertaken to understand what else is dependent on that component.

Status Quo	With Software Intelligence
The bulk of the exploration will be manual, having to jump through all related components by using a combination of grep and reading code. This will be a multi-week activity. The scope of work is unpredictable, and the effort is hard to estimate. There is a risk that some components are left out once testing begins, thus causing issues in live use or extending the project length. Depending on the level to which separation of concerns already exists, this exercise may be more involved. With vertical concerns separated, that is layers, this exercise is limited to just a layer at a time. With horizontal separation of concerns also implemented, the scope of exploration will be smaller.	The team will look at the visuals of the system to examine the functional component and all of its interconnections within the first couple hours of the project. As shown in Figure 3, the components that require the least work to repackage can be selected and modeled as if they were all part of one component, such as an API or a microservice. This modeling exercise should be just drag-and-drop, so in a matter of hours the team can model many alternative scenarios and make an informed tradeoff about draw the functional boundaries of the new microservice.
Weeks of exploration. **High risk** of leaving out components.	**Hours** of exploration. **Negligible risk** of leaving out components.

Motion 4: Identify obsolescence or vulnerabilities

In a system with thousands of components, many of which are already taken from open source, it's important to keep on top of versions and known vulnerabilities. Older versions of OSS components typically have more known vulnerabilities and can become completely obsolete. Staying on top of this can be an enormous task, and there are a number of tools in the Software Composition Analysis (SCA) space that can help. Once the obsolete components are identified, they need to be replaced or the architecture needs to be altered in order to do without them. Motion 6 discusses replacing or rewriting a component and Motion 7 explains the process of inserting a new framework.

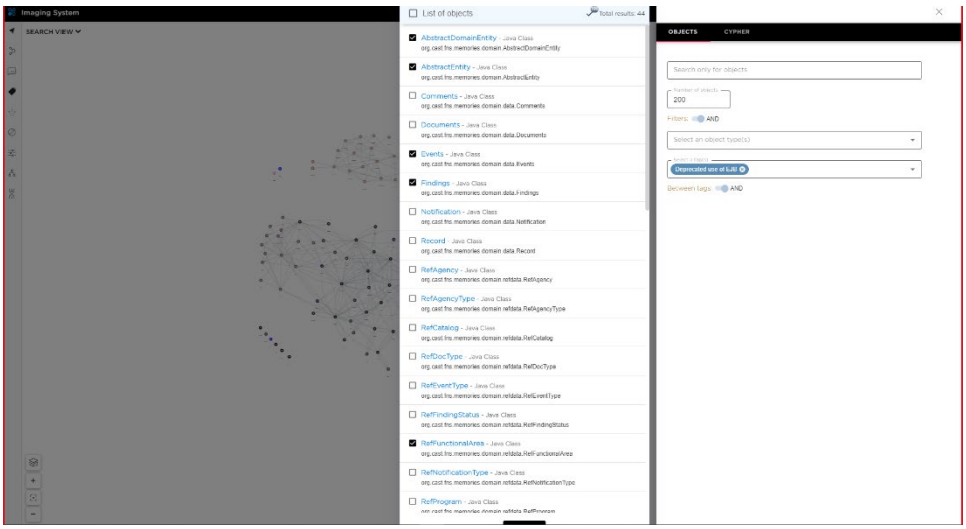


Figure 4 – Results of search for obsolete components

This motion starts with the output from the SCA tool, identifying the components or frameworks that are affected. Then the real work of assessing the impact and workload of these changes begins.

Status Quo	With Software Intelligence
Once the list of identified frameworks is dumped out of the SCA solution, a developer is assigned the task of finding that component in the code and figuring out what can be done to remedy the situation. Usually, the developers assigned to this task will be using a combination of grep searches and walking through code to see where the frameworks or components need to be replaced. This will be a couple weeks of work, and then the set of activities in Motion 6 will take over.	The team will run a quick search on the identified components, and how these components look in the context of the "as-is" architecture. The definitive list is analyzed, with an automated impact analysis to be able to estimate the approximate time to replace each component. Then the list of obsolete or vulnerable components can be prioritized based on risk vs. effort to create an action plan. Overall, this should be about a half day of work, before actual implementation, as described in Motion 6.
Matter of **weeks**. **Low certainty** of prioritization.	Matter of **hours**. **High certainty** of prioritization.

Motion 5: Decouple a community of components

There are many types of decoupling that could take place in refactoring an application. It could just be one small component, a microservice, an API or a whole layer. In either case, the first step is to see how that decoupling will look in the context of the surrounding components. The main action is to first identify the community of components and identify the various points of liaison this community has with the rest of the system. These liaisons can be replaced with ones that have a low level of coupling, such as an API or an HTTP call, thus removing all strong liaisons, such as a direct object instantiation for example.

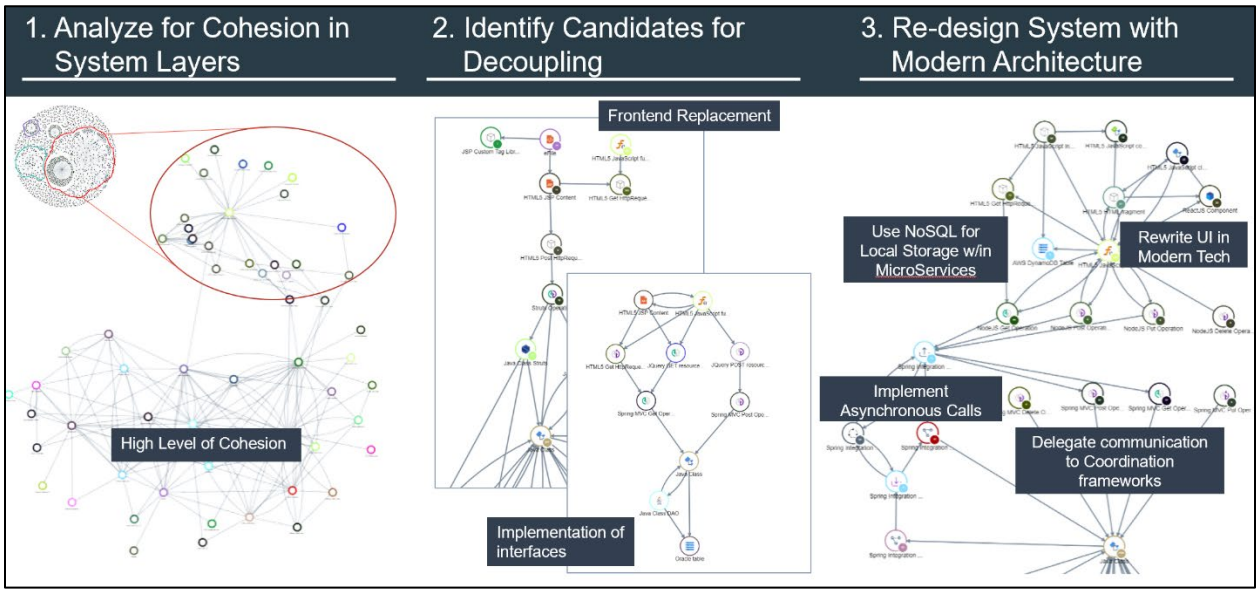


Figure 5 - The steps in decoupling the Front End of an application

If we take the example of decoupling the front-end interface from the back-end business processes and data – a common motion that we see in reorganizing legacy systems, there are several steps that one must take:

First you need to understand what those layers look like today. What is the level of cohesion? How many components are involved? What are their touchpoints that will have to be altered?
Then you need to look at which of those components need to be decoupled. A set of components will comprise the new Front End layer design. There will be another layer of components to form the interface to all other services/components in the application. And then you implement the desired changes. In the example we're using, as you can see in Figure 5, some of the changes might be a rewrite of a UI component or use of a different local storage facility in combination with a microservice.

The key is to have visibility into the multiple dependencies that can affect a decoupling effort. Using a software intelligence capability to visualize these dependencies can speed up the effort significantly.

Status Quo	With Software Intelligence
Depending on the scope of the de-coupling, the paths through the system that would need to be discovered could range from just several to dozens. The discovery process described in the Motion 1 section can therefore span hundreds of components and take several man-weeks. Once an understanding of the system is established, depending on the end-state architecture, the team may have to do some more discovery to complete an impact analysis. Sometimes up to 30-50% of the effort can be wasted trying to understand impact of decoupling changes and component dependencies. And, due to the unknown unknowns, the project timeline can be off by 25-30% or more.	The team isolates the components that are being decoupled and studies the visuals that show the impacts across various components and layers. Those impacts are enumerated quickly, so the team can size the effort and assign appropriate resources. The whole process should take a half day or so, and the ensuing implementation project timeline should be 95% accurate. This motion is perhaps the one that takes the biggest advantage of Software Intelligence capabilities.
Weeks of planning. **Low accuracy** project estimates.	**Days** of planning. **High accuracy** project estimates.

Motion 6: Rewrite, expose or add a component

If a component has even minor changes, or a new component needs to be added to the system, all the upstream and downstream components may be affected and need to be checked. Of course, testing is part of that process, but even in planning the change it's necessary to visualize how the change will propagate through the system.

Aside from the coding of the new component, most of the upfront effort here is in the impact analysis to understand what else needs to be rewritten outside the affected component.

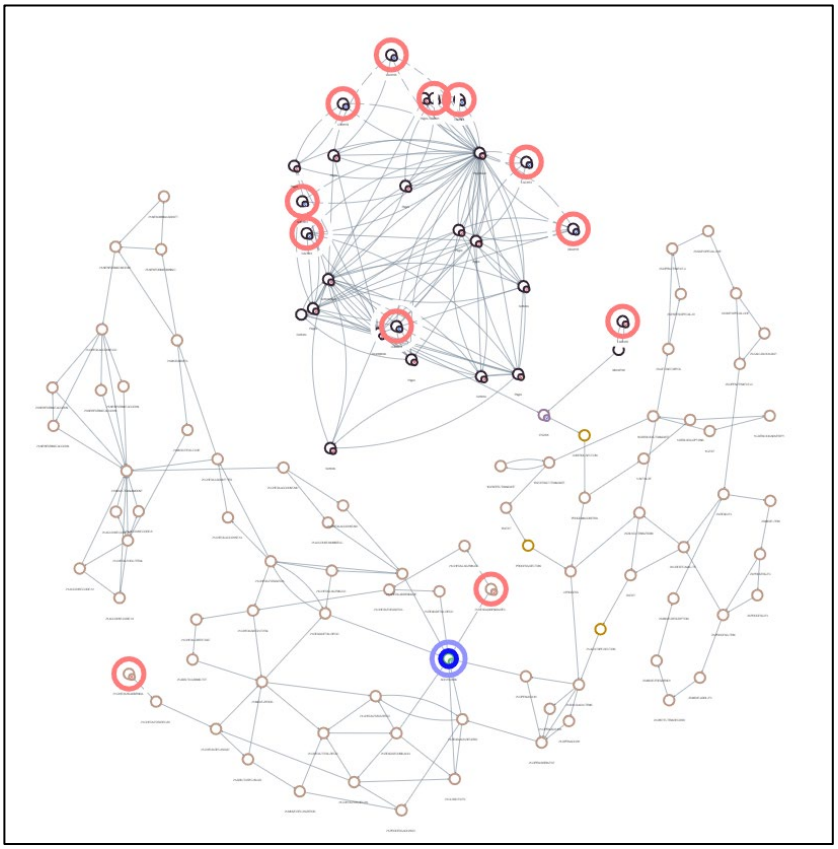


Figure 6 - Looking at the impact propagation of one system component

Status Quo	With Software Intelligence
Since the focus is on one component, typically the developer will start with that component, do some grep searches to see where this component is referenced, and look inside the component to see what happens downstream. If it's a new component, it's a similar process. This is not as involved as Motion 5, where a whole layer is impacted, but still takes time. Surveys estimate that the impact analysis is around 60% of the work effort of modifying or creating a component in the context of an existing system. **Weeks** of analysis.	The team looks at the component and all upstream and downstream elements, as shown in Figure 6. Since the impact assessment is relatively instantaneous, the software intelligence can be used in the planning and prioritization process to estimate effort upfront with relative accuracy. **Hours** of analysis.

Motion 7: Insert a framework

Like component replacement, but more significant in its impact, a new framework must be checked for impact across all the components it interfaces with. First to see how that framework can replace existing components and then, based on the capabilities of the framework, whether there are opportunities presently or at a later time to take advantage of those new capabilities. Then, to see where in the rest of the application will be impacted once the framework is removed or replaced with a new version.

Replacing a framework is more complex than just one component. First the components currently doing the job need to all be identified. Then, all their interactions amongst themselves and their ecosystem.

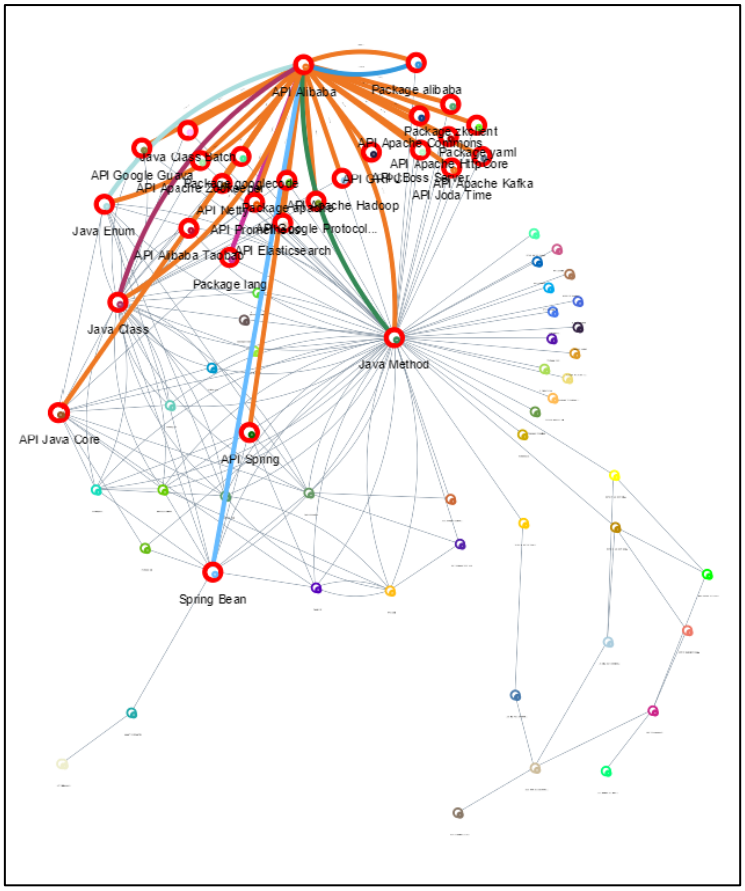


Figure 7 - Exploring an obsolete framework and its surrounding components

Status Quo	With Software Intelligence
This, like in many of the other motions described here, can be a time-intensive, trial and error process of reading code, looking at configurations and following links. Once the impact areas are identified, the project can be planned and executed. A lot of manual effort, with a high degree of uncertainty. If additional capabilities that come with the framework are also looked to be used, then the team needs to identify which components should be rewritten to take advantage of those new capabilities. **Weeks** of analysis.	Within minutes the team can see all upstream and downstream elements, as shown in Figure 7. These views can be used in the planning and prioritization process to estimate effort upfront and make decisions about timing. Also, while looking at the system components around the new framework, with Software Intelligence this becomes an opportunity to drive higher compliance with the framework. Non-compliant constructs can be flagged, enumerated and tracked with the goal of reducing them over time. **Hours** of analysis.

Practical Guidance – Building Your Refactoring Muscle

Anywhere application refactoring takes place, it is necessary to review the existing software, look at the changes to that software and assess the impact of those changes on the rest of the codebase. Once a system gets larger than 100 components, it becomes difficult, nigh impossible for any human being to assess all the interactions to know where the risks are. Most modern revenue-carrying systems are in the realm of 5,000 components or above. As we further componentize our systems into APIs and microservices, those numbers will only increase.

Classical Agile methods have always called for one out of five or ten sprints to be refactoring sprints. With modernization work all around us, and Agile becoming mainstream, the ability to refactor systems is not a one-off challenge. It is a capability that organizations must develop and maintain. Refactoring is an ongoing discipline at world-leading technology organizations. Turning that into a machine-assisted capability by leveraging Software Intelligence is the only way to keep up with modern technology demands of any enterprise.

About CAST Imaging

All the images presented in this document are from CAST Imaging, the world's most advanced MRI for software. CAST Imaging is based on award winning Application Intelligence Platform (AIP) technology, the result of 20 years and $200 million in R&D. The AIP can analyze dozens of common programming languages, scores of common frameworks, database structure, web services, and all configurations to generate a complete end-to-end view of a software system. CAST Imaging turns that metadata into an easily navigable view for architects, engineers and developers to see how their system really functions. To find out more, or get a live demo, please contact CAST or visit www.castsoftware.com/Imaging.

About the Author

Philippe Guerin is a Senior Software Architect at CAST and leads the team of CAST Solutions Architects. He is an expert in application modernization, architectural design, and software intelligence. Over the last 10 years Philippe has been helping businesses, government agencies, management consultancies, cloud vendors and systems integrators assess large IT organizations application landscapes and accelerate transformation efforts. You can contact Philippe directly at p.guerin@castsoftware.com.

Sharing and Attribution