Humane assessment by example

Humane assessment is a method for making software engineering decisions.

This document offers examples of how the various facets of the method works in practice. Specifically, it provides stories of daily, spike and strategic assessment, and it shows how integrating custom tools in the development process can lead to better solutions fast.

All described case studies were solved using the same techniques and even the technology, namely the Moose analysis platform which was designed precisely to enable custom analyses at a low cost. While the technology choice is important, it is secondary to the idea that assessment is a discipline that can be served by a small set of tools and associated skills.

For further information about humane assessment, please consult the official webpage:
www.humane-assessment.com

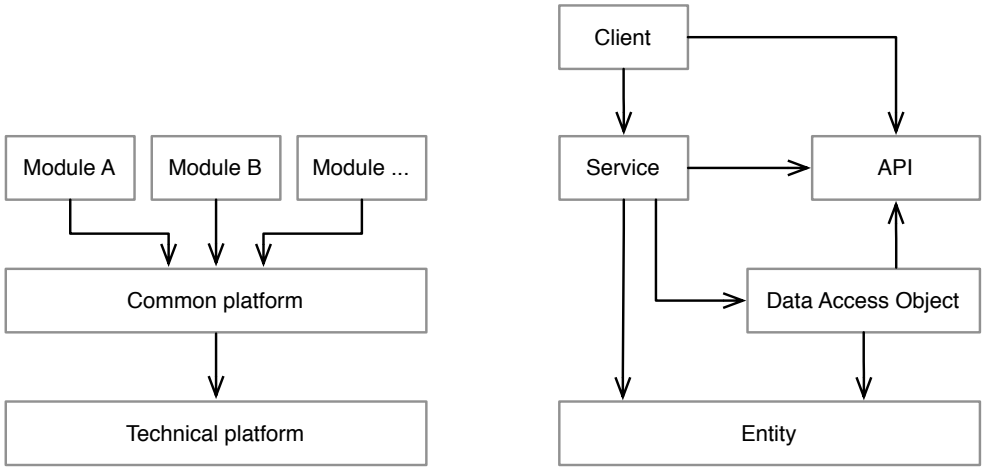
daily assessment

www.humane-assessment.com/guide/daily

Ensuring a cohesive architecture

A Scrum team was developing a Java-based product. Although the team was pressured to release within a space of two months, the developers decided to adopt the daily assessment process to control the architecture.

The teams started by drawing the architecture. The pictures were clean and they revealed clear rules that should be followed. Here are some examples:

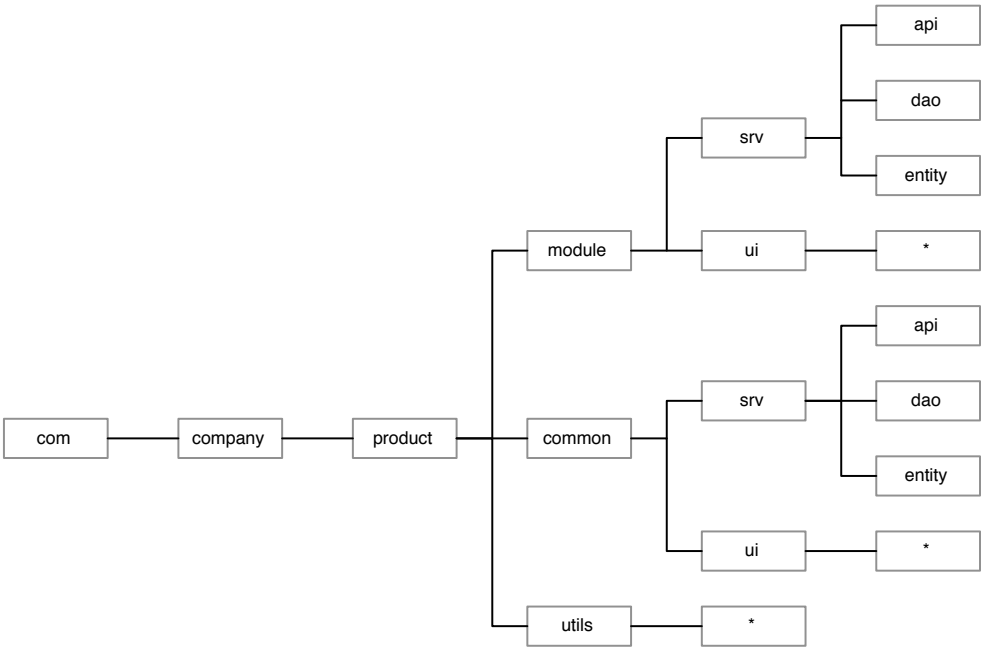


initial drawings capturing various facets of the architecture

Detecting architectural inconsistencies relies on mapping the conceptual components on the actual implementation. In practice, this turned out to be difficult as there was no easy way to locate the presented concepts such as Module A, or all the API classes belonging to that module.

Suddenly, rules that were supposed to be easy to express became expensive to implement. The root problem came from a lack of structure in the code. Thus, we decided to reshape the structure, and to introduce tighter naming conventions.

We started from the package structure. To obtain a convention accepted by everyone, we created a visual map of all packages in the system, and in a succession of blitz-workshops, we asked the teams to use color coding to identify what dimensions they would like to see captured in the names of packages. After a couple of hours of effort, we obtained a scheme similar to the one depicted below.



the grammar capturing the package naming convention

The next step was to encode the convention in a set of rules. The diagram essentially shows a grammar, thus, the easiest way to encode it was to produce a small parser based on it, and to require the fully qualified of all packages to be parseable by the parser. Shortly afterwards, the rule was integrated in the continuous integration and through the daily assessment process we reshaped the package organization.

Once packages got ordered, we could install more strict rules at class level. For example, all public service interfaces had to be placed in the API package, or all classes annotated with @Entity had to be in the entity package. The latter example, was detected using the following checker:

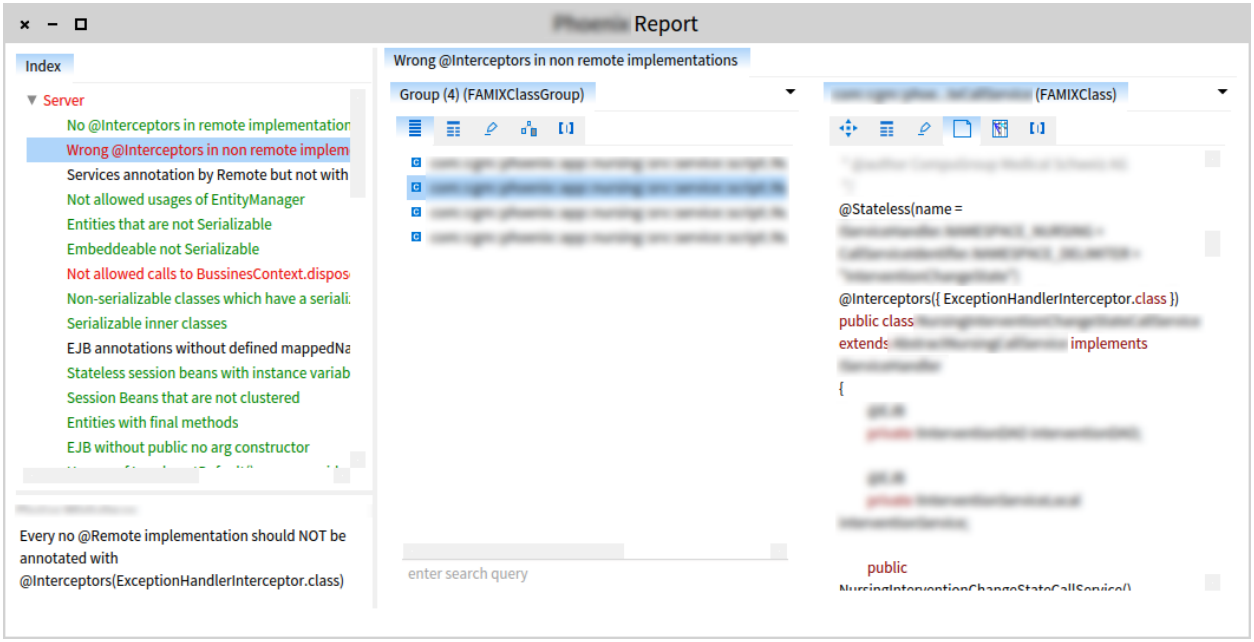
```
model allClasses select: [:each |
  (each isAnnotatedWith: 'Entity') and: [
    ('*::entity::*' match: each mooseName) not ]]
```

After less than a month of work, the code landscape had underwent a dramatic transformation. The team was happy, yet we were not through: we had to still check the architectural constraints. At this point, the detections became as simple as:

```
model allClasses select: [:each |
  each isCommon and: [
    each providerClasses anySatisfy: #isModule]]
```

As expected, we identified several violations, some of which would have led to problematic deployments. The critical ones were fixed

fast, while the rest were marked as exceptions and dealt with in the following months after the release. But, the most important gain was that further rules could be added easily precisely because all components could be easily located.



an interactive report holding various checkers

Even though the surgery was large involving significant renaming operations that affected also the structure in the version control, the team met the deadline without doing any overtime. This stands as evidence for how code cleaning is often not expensive once the problem is explicit and controllable.

But, that is not all either. For a long time, the naming convention was held tight and the small errors were corrected promptly. However, a couple of years after the initial schema choice, a large amount of code was introduced that did not comply with the defined naming rule. As a consequence, several other architectural rules got broken, too. The problem was immediately discussed in the daily assessment stand-up, and the team identified that the cause was the introduction of a module working with web services. Given that the original system did not work with web services, the specific requirements were not taken into account. Thus, it became clear that the situation needed to be re-assessed and a deeper discussion and rethinking of the structure followed.

A good naming convention is not only useful for defining further analyses, but it also is a catalyzer for communication as it helps capture the intention of otherwise abstract entities.

Driving a clustering migration

The company wanted to migrate one of their system to a clustering solution that was based on JBoss features. This feature was considered to have strategic importance and the customers expected to get it running within a couple of months.

To make use of JBoss clustering, the server has to be properly configured. Given that this is not an easy task, and the team had little experience with it, we needed to ensure that the settings were correct. To achieve a better understanding, we created a simulation environment:

- We set up a cluster with several instances of JBoss running,
- We created programmatically multiple clients that connected to a set of services served by the cluster,
- We made sure that we spawn enough client programs to get at least one connection to each cluster node,
- We then took a one node down, and
- We checked that all clients continued to be served.

Having this environment, allowed us to iterate over the assumptions very fast and within the space of a few days, we ensured that we got all settings right. The conclusion of the experiment, was that the only thing left to ensure was that all remote service classes in the system needed to be annotated with @Clustered. The template looked something like:

```
@Remote(IService.class)
@Stateless(name = "XYZService",
           mappedName = "ejb/service/XYZService")
@Interceptors({ ExceptionHandlerInterceptor.class,
              MethodTraceLogInterceptor.class })
@Clustered
public class XYZService implements IService { ... }
```

To ensure that all services did follow this design, we created a checker. Essentially, the checker listed all classes annotated with @Remote and that were not annotated with @Clustered:

```
model allClasses select: [ :each |
   (each isAnnotatedWith: 'Remote') and: [
      (each isAnnotatedWith: 'Clustered') not ]]
```

The system had more than 60 such services that were developed by three teams. To ensure that all services get annotated, we integrated the checker in the continuous integration and monitored

it through the daily assessment process. While at first adding the annotation appeared to be a straightforward activity, we discovered along the way that some services should not be clustered because they had other node specific contracts (e.g., writing a report file and saving it to disk). In these cases, we created another checker that explicitly forbade those services to be clustered. All teams worked through each of the services, and after a short while, all of them got annotated.

At this point, we still had to ensure that our initial assumption held true also for a large system with many services. To check this assumption, we wanted to repeat the original simulation experiment with the real system in place. In order for us to do that, we needed to ensure that we had samplers for each service class.

To achieve this, we again resorted to a Moose checker. This time, the checker identified all clustered services that were not exercised by at least one sampler class (we used a performance infrastructure to simulate clients, and thus we knew that samplers had to be in the performance path):

```
model allClasses select: [:each |
  (each isAnnotatedWith: #Remote) and:
  [(each isAnnotatedWith: #Clustered) and:
  [(each superclassHierarchy flatCollect:
    #clientClasses) noneSatisfy: [:client |
      client isInPerformanceTestPath ]]]]
```

We integrated this checker in the continuous integration and worked through the to do list through the daily assessment process. After a short while all relevant services had at least one sampler, and we carried on with our experiment. The experiment showed that our assumption was correct and that the system was properly clustered.

All in all, the whole effort amounted to about ten person-days distributed in tiny work chunks among the developers. The cost was small compared with how reliable the deployment of the new version happened.

Ensuring correct database schema generation

The team developed a Java enterprise system, and the persistence was developed using the Java Persistency API. For example, a typical one to many relationship looked as depicted below:

```
@OneToMany
public Set<SomeRecord> getRecords() {...}
```

At some point, the team decided that it would be useful, at least for development purposes, to generate the schema out of the code and reinstall in a DB. All went well, but at some point while doing performance analysis, the team noticed that when generating the schema out of the standard annotations, the foreign keys were not present in the case of one-to-many relationships.

This was cumbersome. On the one hand it was very handy to be able to replace the schema at any point, on the other hand, the schema was incomplete. To solve the issue, every @OneToMany annotation had to be augmented by a @ForeignKey annotation. For example, the code above should have looked like:

```
@OneToMany
@ForeignKey(name=”FOREIGN_KEY” inverseName=”PRIMARY_KEY”)
public Set<SomeRecord> getRecords() {...}
```

To ensure that no foreign key got forgotten, the team decided to check for the pattern continuously. The detection rule turned out to be rather simple:

```
(model allAnnotationTypes entityNamed: #’javax::persistence::OneToMany’)
  annotatedEntities select: [:each |
    (each isAnnotatedWith: ‘ForeignKey’) not ]
```

Developers have a stake in the system, too, and their interests should be served as well.

spike
assessment

www.humane-assessment.com/guide/spike

Optimizing a JBoss cache

A major problem appeared after launching a system into production: the end users reported that the system was sometimes very slow. Because this was a critical system, the problem was considered of crucial importance.

To identify the problem, we started by reproducing it in production. It was particularly interesting that the slowness occurred only sometimes, and not all the time. After instrumenting the system locally, we saw that some parts of the executed code were related to a cache. This was consistent with the original reports and we deemed it to be a relevant path of investigation.

The problem with working on a cache is that its behavior depends highly on the usage. Simply working on the cache locally would not guarantee success in production. We needed to exercise our system with real user activity. However, given how critical the system was, we were not allowed to deploy experimental solutions in production.

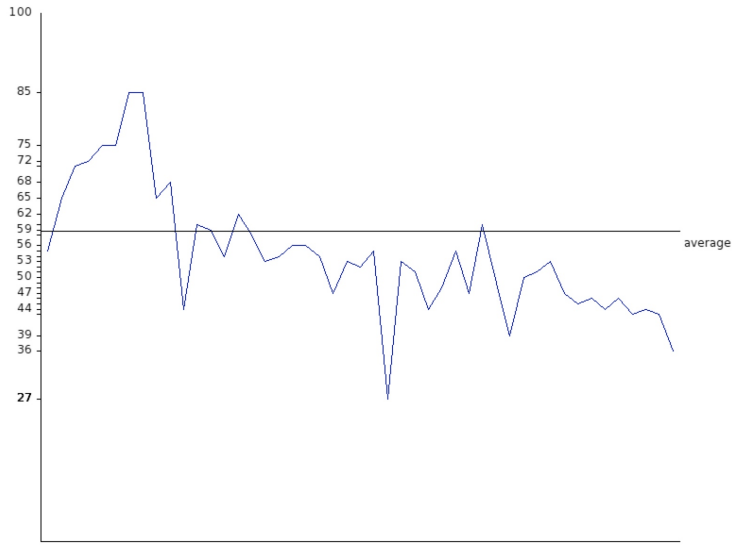
However, the production system had a strong logging infrastructure that logged all cache requests and the associated hits and misses. An excerpt from the large log file could look like:

```
2012-03-13 16:10:56,876 TRACE  CachingDelegate: Cache hit /service/Region
QueryTO [depth=-1, parentDepth=1, namespace=Region, regioncode=IACA,
itemcodes=[APA4725090], codesystem=, elementkind=ITEM,
descriptionIncluded=false, validityTestMode=false] hash=-742458864
2012-03-13 16:10:58,470 TRACE  CachingDelegate: Cache MISS /service/Region
QueryTO [depth=1, parentDepth=2, namespace=Region, regioncode=Structure,
itemcodes=[], codesystem=, elementkind=ITEM, descriptionIncluded=false,
validityTestMode=false] hash=2003270021
2012-03-13 16:10:58,563 INFO   ScriptCalls: ScriptCall called with Parameter:
namespace[nnn;Core]
2012-03-13 16:10:58,798 INFO   ActionService: getActionsByDate: patId=8909580,
caseId=3807672, from=Wed Dec 21 00:00:00 CET 2011
2012-03-13 16:10:58,923 TRACE  CachingDelegate: Cache hit /service/Region
AnotherQueryTO [regioncode=null, itemCodes=[], namespace=Region,
codeSystem=domain.entry, elementKind=Region, validityDate=Tue Mar 13 00:00:00
CET 2012, descriptionIncluded=false, locale=de_CH, validityTestMode=false]
hash=570633577
2012-03-13 16:10:59,266 TRACE  CachingDelegate: Cache MISS /service/
AnotherRegion QueryTO [depth=1, parentDepth=-1, namespace=AnotherRegion,
regioncode=630, itemcodes=[I_23451], codesystem=, elementkind=ITEM,
descriptionIncluded=false, validityTestMode=false] hash=-1116635756
```

In these log files, we could see which cache request was a hit and which one was a miss, and we could see for each request the

corresponding parameters. Using this information, we could plot the cache hit ratio over time.

We built a parser that extracted the cache information and built a browser featuring charts and querying possibilities. The chart confirmed the original suspicion that the cache behaves both poorly (with a hit ratio of under 60%), and randomly (sometimes a high ratio, sometimes a very low one).

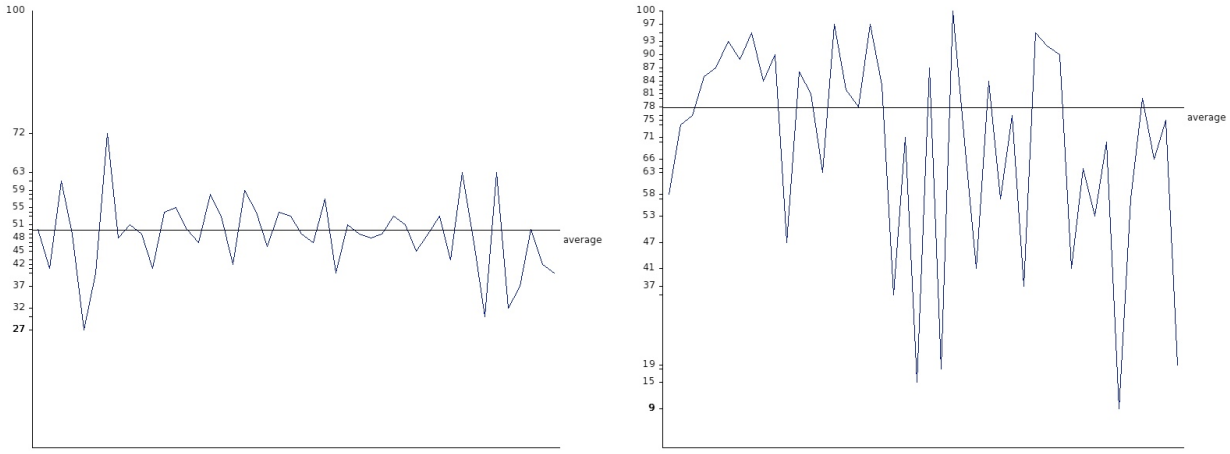


a first plot of the cache hit ratio

Given that we could not work directly on the production system, we installed a simulation environment that mimicked user activity:

- We copied the production database in a separate installation,
- We extracted the cache related entries from the large log file,
- We created a client that replayed the cache requests at time intervals specified in the original log (we actually used a fast forward factor of 20x to speedup the experiment), and
- We ensured the accuracy of the experiment by comparing the chart corresponding to the new log output with the original chart.

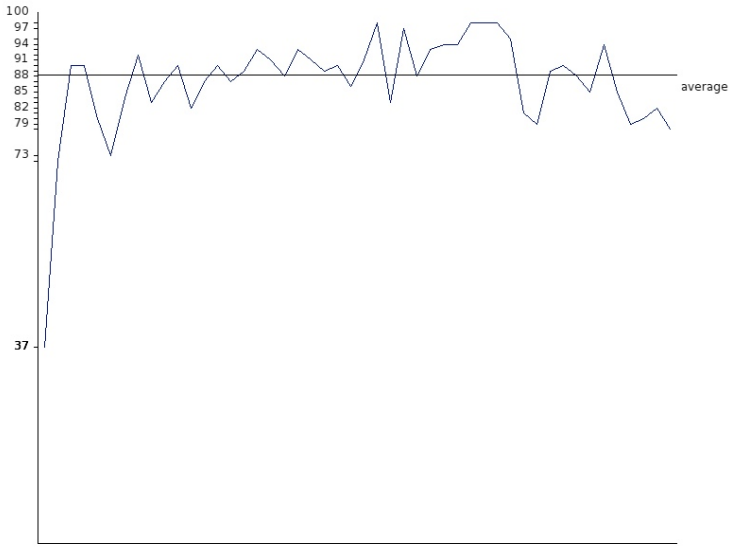
Once we obtained a simulation environment, we went deeper in the code. After some code exploration, we noticed that there should be two different cache zones. Indeed, when looking in the original log, we identified that there were two types of requests with different parameters. Because our tool allowed us to query the data interactively, we could easily split the logs and plot each part separately.



revealing the difference between the two zones of the cache

We observed that the two parts were not behaving the same way. While the first one was slow, it was predictable, and in fact, it was not affecting the use cases reported as being slow. The slowness and randomness came from the second zone in the cache.

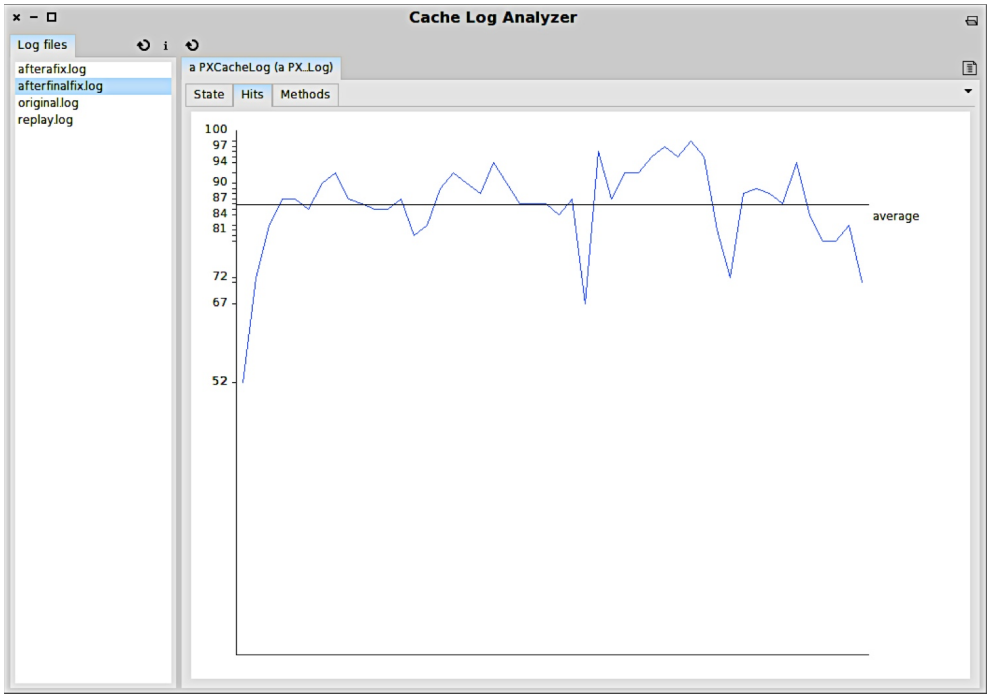
At this point, we knew where to focus in the code, and not long afterwards we identified the cause: because of a coding mistake, the whole cache was emptied instead of only throwing away the least used entries. The modification was rather easy, and after replaying the experiment, we got a much better performance.



plotting the hit ratio to confirm the cache improvement

The overall effort took somewhere around five person-days. At the end, we had a solution and due to the systematic assessment, we were confident that the solution would solve the end user problem. Once we deployed it, we learnt that we were indeed correct.

This was possible due to the systematic assessment approach. First, by setting up a simulation environment we could test and compare problems and solutions Second, using a visualization helped us to understand the dynamic nature of the cache. Third, the interactive capabilities of the browser allowed us to query the log entries in several ways until we found a revealing split.



a screenshot of the interactive browser used to slice the log entries

All in all, the investment of building tools paid off quickly because it helped us check our assumptions against the data in almost real time.

Guarding against a memory leak

In JBoss 5 there exists a bug in the implementation of the memory pools used for the allocation of beans. However, the bug is only apparent in special circumstances when (1) the system has cyclic dependencies between beans, and (2) the pool is smaller than the maximum amount of beans wanted at a time.

In other words, if you have cyclic dependencies between beans you are susceptible to experience memory leaks when using JBoss 5. The team discovered this issue after a painful couple of months of chasing the problem in a production system that was accumulating memory leaks. They found that the issue was caused by circular dependencies between two beans.

The code looked like this:

```
@Remote(...)
@Stateless(...)
public class XYZService {
  @EJB
  private ABCService abc;
  ...
}

@Remote(...)
@Stateless(...)
public class ABCService {
  @EJB
  private XYZService xyz;
  ...
}
```

One way to guard against the bug is to avoid circular dependencies completely. In our case, breaking the problematic cycle was not possible in the short term because the cost was too high. However, we still could guard against the bug by leveraging a technical loophole in JBoss 5: setting the beans pool size to be larger than the amount of beans used at a time prevented the problem. However, to find out what this right size is, we needed to form an idea of the runtime scenarios, and the first step in this direction is to know what cycles between beans there are throughout the whole system.

The detection of these cycles might seem less straightforward because the cycle does not appear explicitly in the code given that both references point to the interfaces, and not to the implementation. The actual cycle only happens at runtime via an injection mechanism.

Thus, to detect our problem statically, we have to manufacture the dependency, and with the right tools, the detection became inexpensive:

```
(model allClasses select: #isRemote)
  cyclesToAll: [ :each |
    each attributes flatCollectAsSet: [ :attribute |
      attribute declaredType withSubclassHierarchy ]]]
```

The expression gets all the beans from the system, and for each of these will look at the possible cycles induced by the types of the attributes to all the sub types of the declared type.

Using this checker, we identified another cycle that was originally not detected. This second cycle was easy to refactor and was consequently fixed immediately. For the second cycle, we ensured that the pools were large enough, and we marked the issue as a temporary exception to the rule.

The checker was integrated in the continuous integration process to ensure that nobody introduces more such cycles. Later, when the project pressure allowed for it, the problematic cycle was removed as well.

Identifying tables used by entities

While going through the logs of a production system, the system administrator observed some suspicious InvalidStateException errors coming from hibernate. The exceptions looked like:

```
org.hibernate.validator.InvalidStateException:
     validation failed for: com.example.model.Concept
     ...
```

One of the assumptions was that these errors were due to some database problems. To check the assumption, he needed to know which tables were involved in the errors. The problem was that the logs only provided the names of the Java classes without the names of the tables involved in the mapping. Thus, one thing we learnt is that we needed to extend the logging infrastructure to export the table names as well.

However, this still did not solve the existing problem because the system was already in production and could not be changed easily. To dig into the problem, he would have needed to open Eclipse, search for the involved classes and identify the associated tables. The Java code associated with the error looked like a regular hibernate annotation:

```
@Entity
@Table(name="CONCEPT")
public class Concept {...}
```

In our example, com.example.model.Concept is associated with the CONCEPT table. However, given that he did not have access to such an environment, he was stuck. He wished of having a simple file with the mapping of all classes. It turned out that the problem is straightforward with Moose. The below script retrieves the mapping and puts it in a simple tab separated file that can be viewed with Excel:

```
model allClasses do: [ :each |
   (each isAnnotatedWith: #Table) ifTrue: [
   Transcript
      show: each mooseName;
      tab;
      show: ((each annotationInstances detect: [:ann | ann name = #Table])
             attributes detect: [:attr | attr name = #name]) value;
      cr ] ]
```

The whole cycle, from problem identification to obtaining the mapping, took somewhere around 15 minutes. After this small investment, the original problem became easily approachable.

What to override?

The project consisted in integrating two existing legacy systems, SystemX and SystemY. The connection point happened through an interface from SystemX consisting of more than a hundred methods that the latter system had to implement.

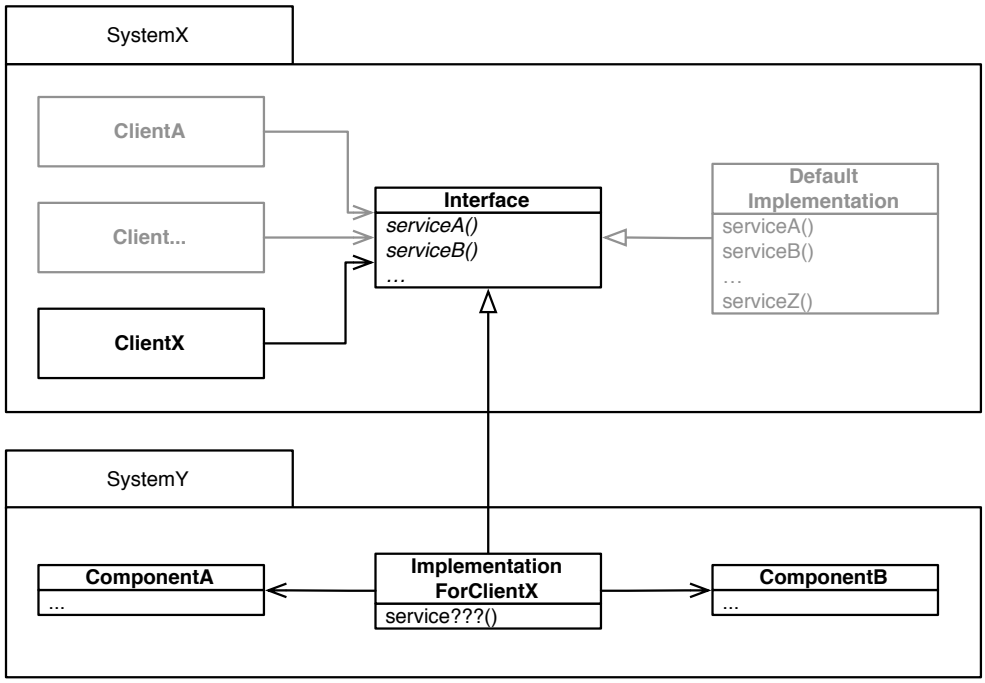


diagram of the connections between SystemX and SystemY. Only ClientX had to be considered.

Given that the runtime scenario only involved a limited set of usages, it was enough to provide real implementations only for a handful of methods. This limited the cost of the integration significantly.

The team still needed to know precisely which methods had to be implemented. The first strategy employed by the team was to exercise the two systems with various functional tests, and check for exceptions. However, this provided no guarantee that no problems will appear in production. To answer the question completely, we queried the interface usages from ClientX:

```
(model allClasses entityNamed: ‘Interface’) methods select: [ :each |
   each clientTypes anySatisfy: [ :client |
      client name = ‘ClientX’ ] ]
```

The analysis revealed that all methods were already properly implemented. Yet, this time there was certainty.

Later, a new version of SystemX was released and had to be integrated. Based on the already existing analysis, we discovered a missing method that was not documented and would have easily gone undetected.

With the new release, while testing the new version, the team also noticed a new exception occurring. At a closer look it seemed that new code contracts were introduced in other parts of the code together with new functionalities. Starting from the known problem, we reverse engineered the code and built a new set of rules. Running those rules revealed several more overriding needs. The complete effort was measured in a few hours.

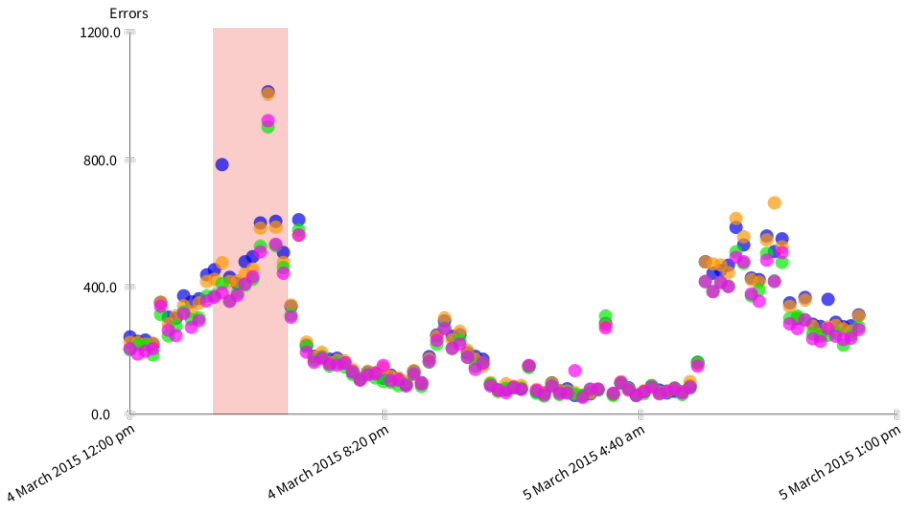
Not all problems can be easily tested functionally. It is often significantly more effective to check them statically.

Evaluating the cause of a peak problem

The users of a mission critical hospital system reported that the behavior of the system got significantly slower, close to unusable, during 3 and 4 PM. The issue transformed into a crisis, received immediate top management attention and solving it was given top priority.

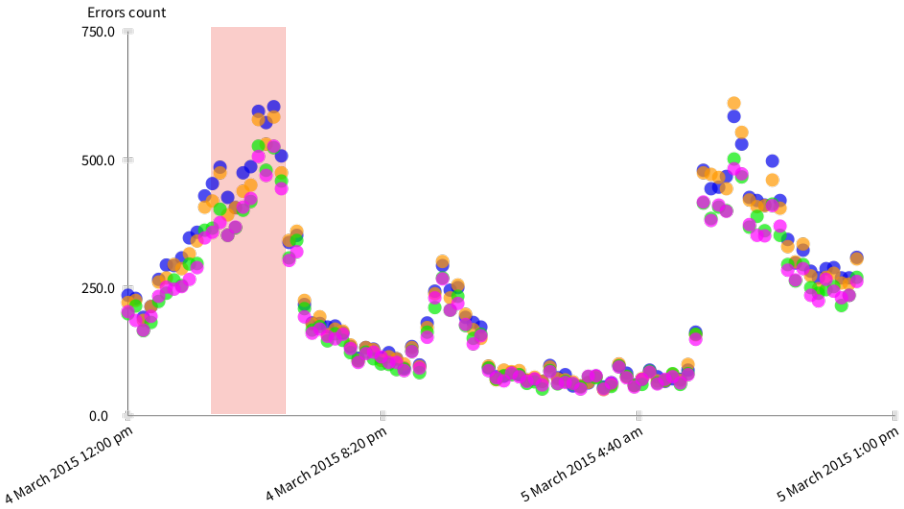
After a brief evaluation the system administrators concurred that the issue is due to a logical error in the code, and handed over the problem to developers. Looking closer at the situation, something did not add up, and we went back to the system administrators to investigate further. They explained that the problem was quite obvious. They showed us the log files of the production installation, and indeed they showed that during 3 and 4 PM, thousands of exceptions of a certain kind of `SomeStrangeError` were thrown. They concluded that this is abnormal as they did not encountered these exceptions on any other installation, and hence they deemed it to be the root of the performance peak problem.

At a closer investigation, the `SomeStrangeError` came from the underlying middleware and it was unlikely to have anything to do with performance. This hypothesis was confirmed by the support from the middleware supplier. Knowing this, we investigated the log files ourselves, only this time we parsed them and extracted all exceptions reported by the four server nodes and plotted them on a chart as seen below.



All errors from the log files. Each circle denotes 15 minutes worth of errors. The color of the circle denotes a distinct server node.

Indeed, during 3 and 4 PM there was a spike in errors. However, given that some of the reported exceptions were of other nature, we further trimmed the chart to show only the exceptions of the `SomeStrangeError` kind.



The same chart as above showing only the problematic `SomeStrangeError`.

The picture reveals a different situation. While it is true that the spike in errors correlated with the performance slowdown during 3 and 4 PM, at the same time there was another spike of errors at a later time in the same day that was not correlated with a slowdown. As a consequence, it was unlikely that the errors were the cause of the slowdown.

Our assessment took a couple of hours, and we used this chart to argue our case with the management and system administrators. In the end, the cause of the performance slowdown turned out to be an undocumented backup job that was triggered between 3 and 4 PM.

Taking a step back, it is certainly true that the technical state of our production system was far from ideal, but the current problem was not due to that state. When the system administrators looked at the logs, they used a text editor which tends to focus the attention on the details. However, the real pattern only revealed itself when looked at a larger granular level through dedicated tools that allowed us to quickly parse, trim, and visualize the data.

Chasing troublesome announcements

A while ago, we got a major problem in the browsing engine of Moose. The engine was designed to help developers build browsers easily. Once the model of a browser was defined, the renderer produced the actual user interface that had to be kept in sync with the conceptual model. The problem was that in certain cases, the renderer did not display correct values.

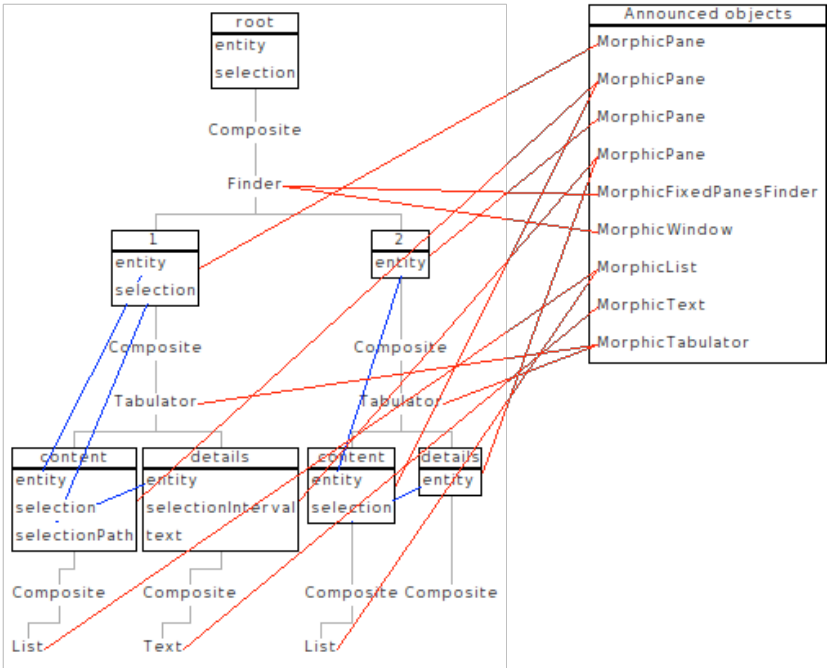
We knew that the problem was related to the communication mechanism between the model and the actual user interface. To handle the interaction between the objects from the browser model and the actual user interface widgets, the engine used announcements objects to implement an observer pattern.

Given that the communication did not happen via direct calls, it was difficult to get the proper overview by using only the code browser. Furthermore, the engine relies on a prototype-based design and it deep copies its model objects every time there is a significant interaction. Thus, a large part of the behavior is only to be understood at the objects level, rather than at the class level.

After several dozen days of investigations involving several people, we got to capture the situation in a testable scenario, but even so, we could not find the cause. There were simply too many objects around that obfuscated the situation.

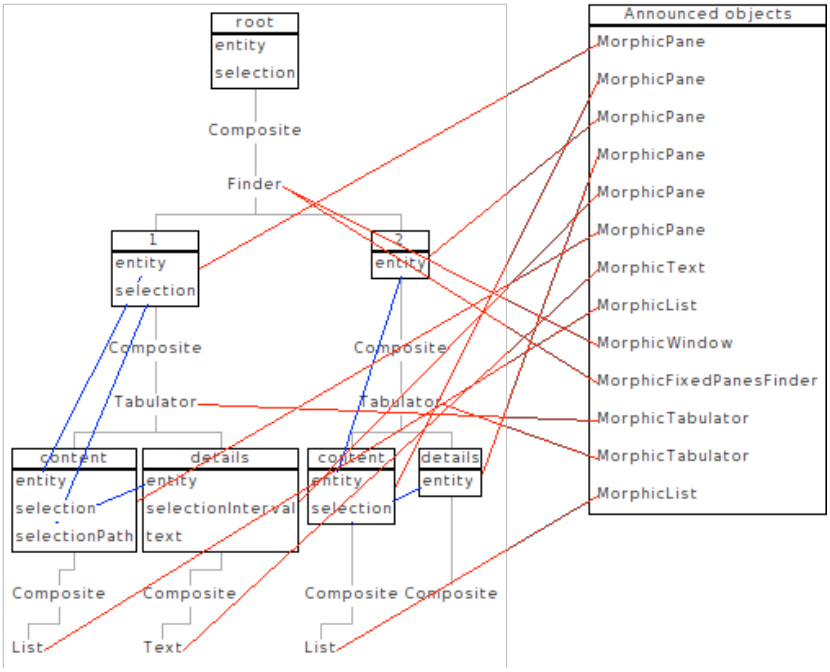
At this point we changed the debugging strategy and built a visualization to help us understand these objects and their connections. On the one hand, we needed to visualize the tree structure of the model. This is depicted as a tree of boxes on the left hand side in the picture below. On the other hand, we wanted to see the connections between each model object and the corresponding rendering object. The connections are shown with red lines leading to the rendering objects on the right.

The picture revealed the problem nicely: there were several objects from the model that were linked to the same rendering object. For example, the second MorphicPane from the top right had two red edges connected to it. There should have been exactly one such edge for each object on the right. This meant that the problem was certainly related to how announcements were copied.



the initial visualization how the objects to the right are connected to more objects from the left

This reduced the scope of search to a set of about 10 methods.
After a brief inspection, the solution boiled down to a one line fix.



the visualization that showed the effect of the fix

Not all problems can be captured in a useful manner from a functional point of view. Even if we had a failing test, we still could not get to the root of the problem. In our case, employing a data-oriented route helped us identify the problem much faster.

strategic
assessment

www.humane-assessment.com/guide/strategic

Recovering data flow mappings to support a strategic decision

The client was developing a long lived, difficult to understand, embedded system that was already deployed on many remote sensors. The system had an interface that allowed administrators to modify the system configuration remotely for each sensor, but this interface only worked live, and the client wanted to add an offline mode as well.

The team identified two possible solutions, and the management was faced with a strategic decision:

1. Virtualize the whole system and build an interface on top of it that treated it like a black box, or
2. Reverse engineer the system to build upon the existing backup mechanism and use it for administrative purposes as well.

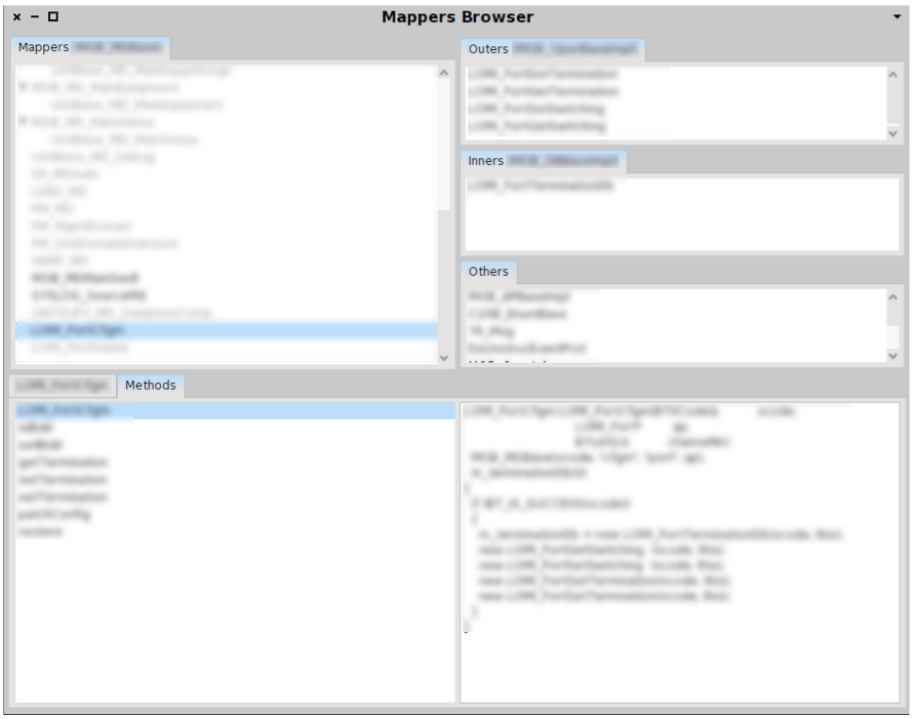
Option 1 was cheaper in the short run, but it was not desired due to lags and brittleness. Option 2 was only possible if the backup model could be easily mapped on the interface model.

We were approached to help with the decision. As the key decisional element was related to the simplicity of the mapping, we focused on recovering it from the existing code.

The first step was to choose a sizable subsystem as a concrete example. After several quick interviews and browsing of the code, we identified that all classes representing the interface model inherited from a OuterBase class, and the backup classes inherited from an InnerBase class.

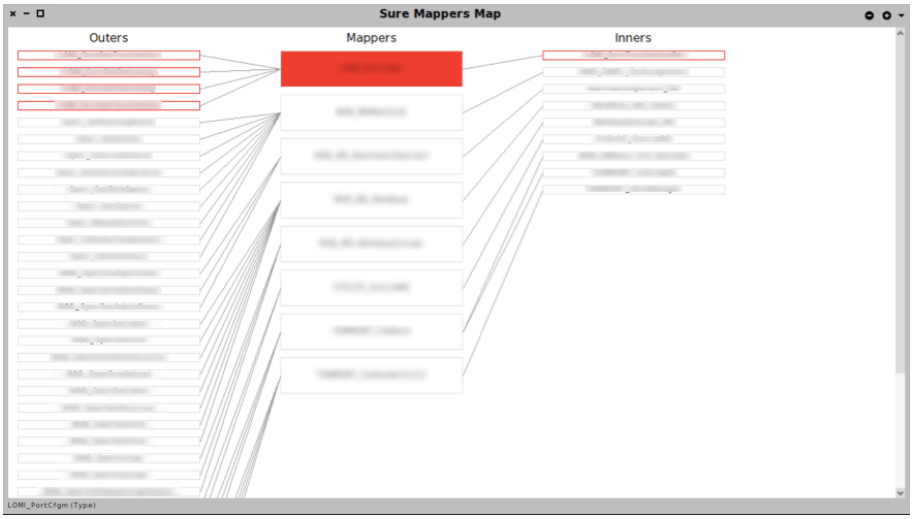
To find out where the actual transformation happens, we searched for the methods that received an OuterBase class as a parameter and create an InnerBase class. We quickly got to a few places, all of them inheriting from a MapperBase class. We double checked that all subclasses indeed offered similar mappings both through queries and by going through the comments.

At this point, in order to find patterns, we identified that we would benefit from a dedicated browser that would focus on the mappers and show the related inner and outer classes.



a dedicated browser for investigating possible mappings

Using this browser, we could quickly traverse many mappers and identify several mapping heuristics. To help the team understand the mappings, we encoded the heuristics into an interactive map.



a dedicated interactive visualization showing the mappers and the corresponding inner and outer concepts

This map showed that multiple outer concepts map on the same inner object, and this implied that it would require a significant work to build the original requirement. Finally, option 1 was chosen. The overall assessment effort was measured in a few days, and as a result, a strategic decisions This time, based on facts.

Supporting a large performance optimization

The client had a critical problem: a key enterprise system was slow. This caused great havoc among end-users, and the whole project was threatened with cancellation. This problem got into the attention of top management, and a strategic project was started.

We were appointed to lead the project. The first priority was to clarify the goal. All the stakeholders set at the table and agree on an initial set of use cases that should be made faster.

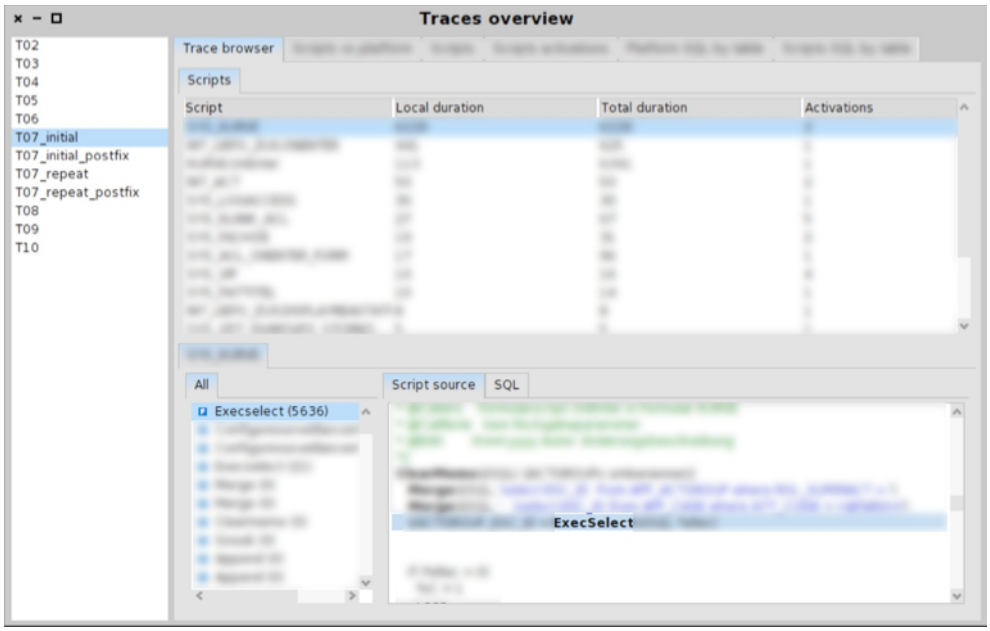
The only reliable way to make a system faster is by means of measurements. Once the use cases were established, we setup to measure the performance. The system was based on a combination of Java, Delphi and a proprietary scripting language, and a significant part of the initial problems seemed to be located at the interaction between the Delphi and the scripting language. The problem was that there existed no infrastructure for exactly addressing this contextual situation.

Thus, we setup to build a tool that would enable us to measure reliably. Given the heterogeneity of the technology, we needed a way to collect the dynamic measurements. This being a long lived enterprise system, we noticed that it came with a strong logging infrastructure. We decided to use it as a basis for data collection:

- We extended the logging infrastructure to handle multiple sources, and
- We introduced in various engines logging statement holding measurement information.

Once we had a log file with detailed measurement information, we built an importer that extracted the execution model and integrated it in a custom browser. Because we did not know where exactly the problems can come from, at first, we built a tool that presented highly detailed information. In particular, it linked the static model of the code with the dynamic information gathered from the log. This way we could easily query various patterns.

This first exercise required approximately 20 person-days mostly due to the variability present in the log files and due to ensuring accuracy of measurements. Once we had a first version of the tool, we managed to identify several issues within a few hours.



the initial browser brought together both static and dynamic information

During this initial phase, we noticed that comparing before and after measurements is key to providing quick feedback. However, while our tool was highly detailed, it was not fast enough mainly due to the static analysis.

Thus, we reshaped the tool to only focus on the dynamic part and be able to analyze large log files within seconds. Given that the original infrastructure needed to parse the log files was in place, reshaping the tool took approximately two days.

As in any project that requires strategic decisions, the distillation of the technical results to non-technical people is critical to ensure good decisions. In our case, we presented a before and after measurements to show that we can reach optimizations of up to factor 10. This led to management confidence and to the expansion of the project to improve further use cases.

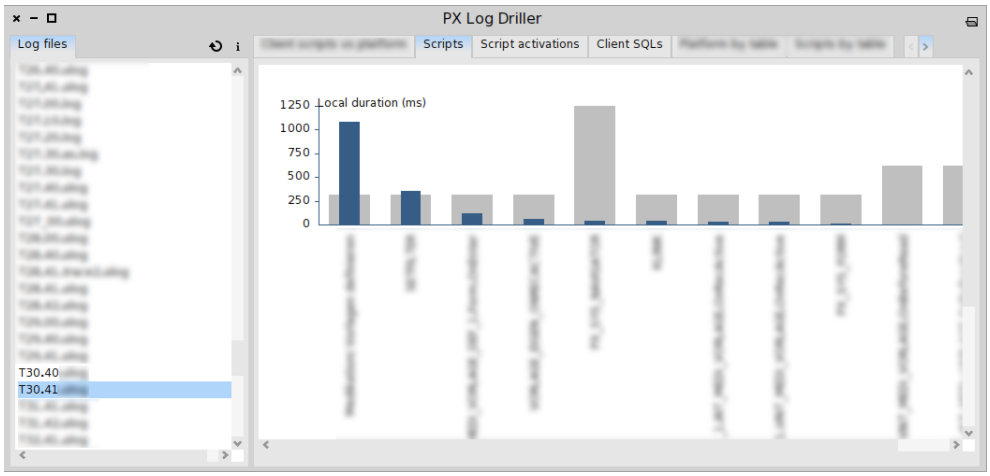
Some problems, such as a poor SQL statement, were easily solvable, but others required more domain knowledge to reorganize the code but still keep the same functionality. To address this issue, we brought in the team multiple developers as required by the tackled use cases. Often they would join the team for only one or two days.

The tool was crucial in getting this process to work. The tool was both simple to use and presented enough contextual information

that developers had a high rate of managing to solve the issue in a short amount of time.

To make the most of the tool, we continuously evolved it with detections capturing patterns that we already fixed. For example, we noticed at some point that it is useful to provide a split of the SQL queries by the affected table, and we built it in. Or we saw that we often need to search for textual patterns in the queries, and we built it in.

In total, we approached more than 40 use cases and we involved a total of 10 developers over a period of two months. The project was considered a success, and it essentially took four months less than the original estimation.

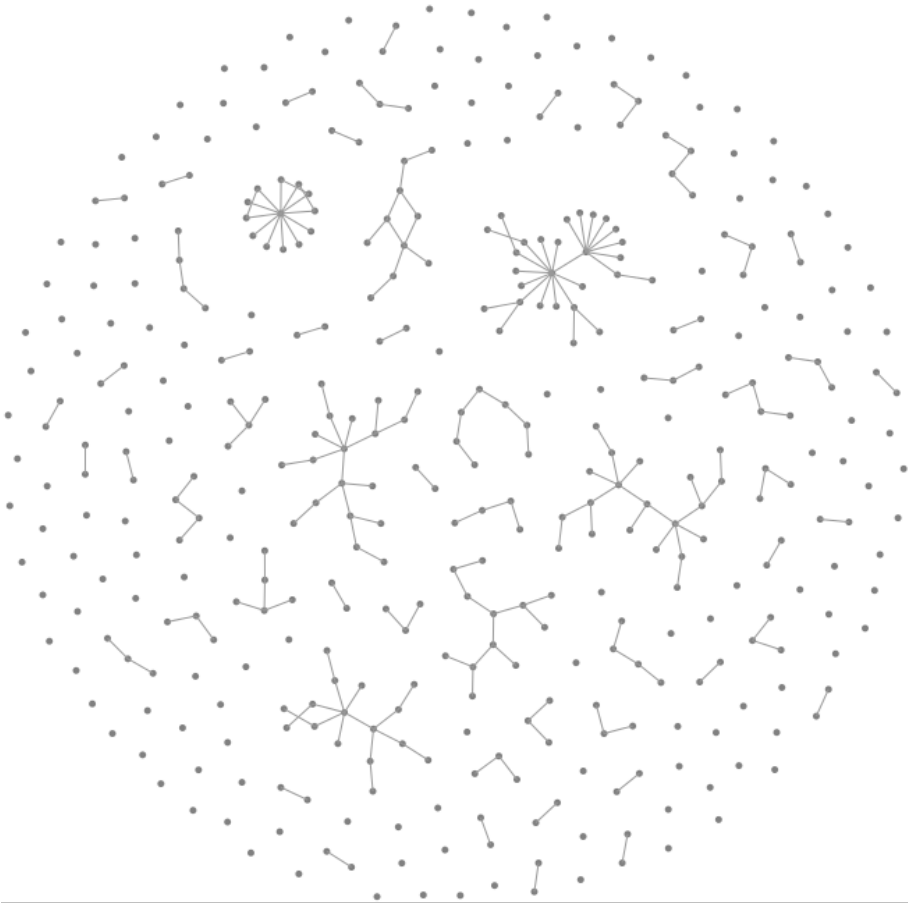


an example of a performance report split by each involved script

Evaluating a refactoring path

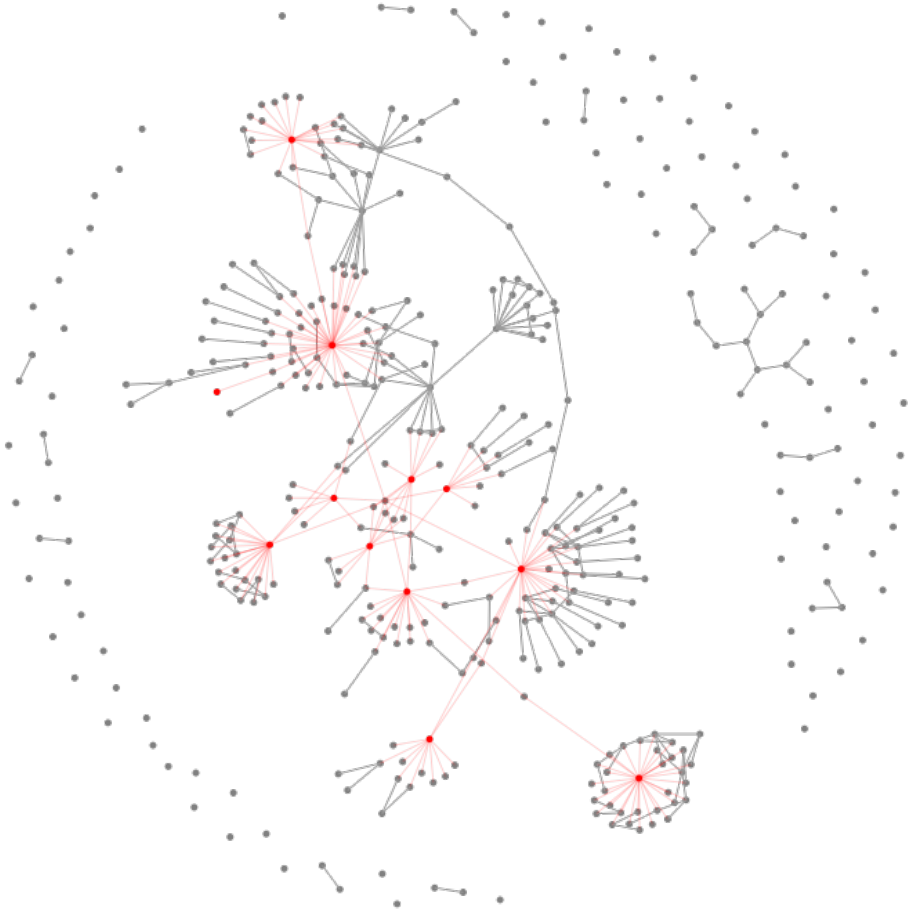
This is a story of finding a refactoring path to split a large class of almost 1000 methods that was central to the user interface framework. The class was responsible for handling the theming of widgets. The problem was that adding new features to the theming behavior was close to impossible. We needed to find a way to split the class. The question was how to do it and how to be confident that it is the right path.

To understand the problem, we first took a look inside the class. One way to split a class, or a module in general, is to identify cohesive concepts inside. To this end, the visualization below shows all methods and all their inter-calls. We notice that the graph is sparsely connected, thus not providing a useful guidance to find good boundaries for splitting. A similar picture came out also when taking the attributes of the class into account.



Visualizing the methods inside the class and their inter-calls.
The visualization reveals many unrelated methods.

If the internal implementation does not provide meaningful leads, we have to look at how the class is being used. Thus, the picture below shows the same methods in gray, but this time also in the presence of a dozen significant widgets that are using the theming class denoted with red circles and red edges.



Visualization of the methods of the class in gray and a dozen significant client classes in red.

From this picture, we noticed that the methods cluster very strongly around red circles leading to the conclusion that the theming logic should be placed closer to each individual widgets classes. The overall assessment took half a day, and the result informed a key decision which was part of a project that redesigned and reimplemented completely the graphical user interface engine over more than two years to develop.

Estimating a strategic model change

The system was a strategic piece that managed the raw data in a large transportation company, with many other projects relying on the data managed by the system. It was built for over 25 years through a combination of Delphi and PL/SQL.

The team was faced with a dilemma: continue with the existing technology and design or move to a newer service-based approach that was closer to the rest of the eco-system. We were asked to help with evaluating the situation.

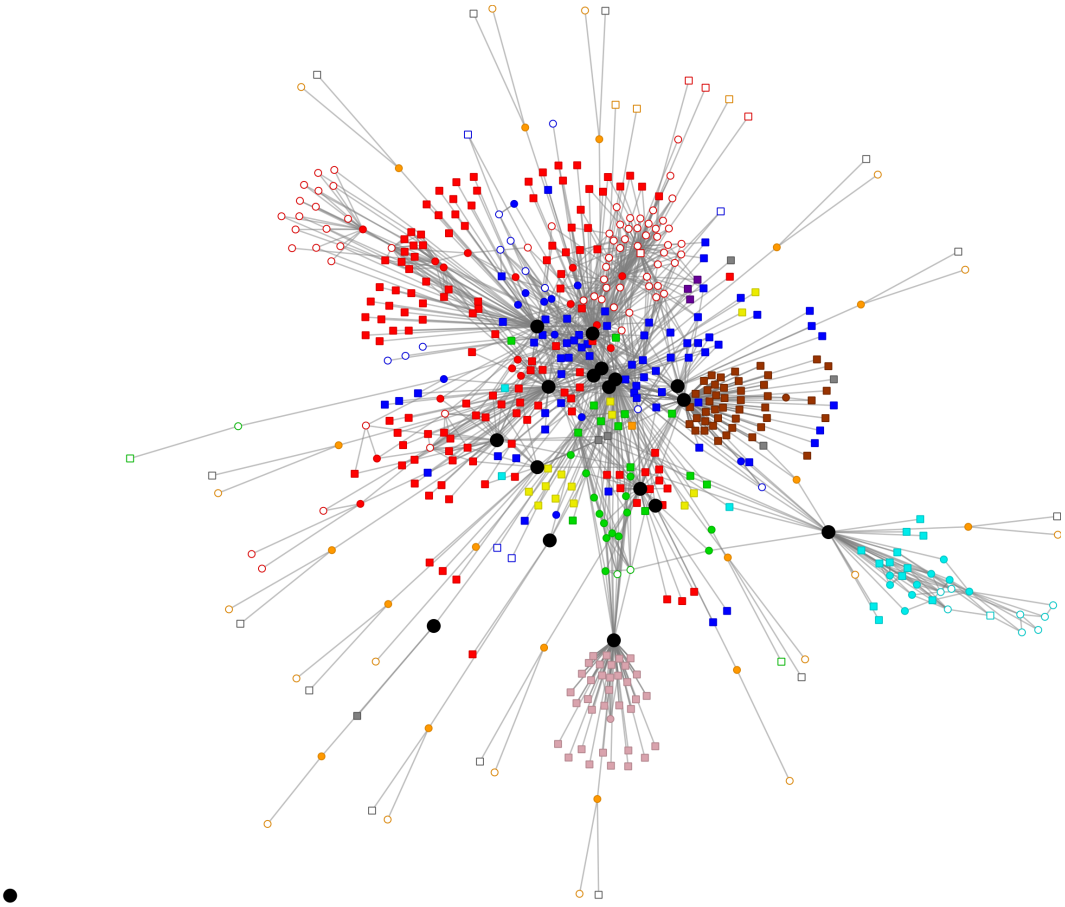
One near future challenge was related to a change that had to be accommodated in the data model. The change in question was due to the need of accommodating finer grained semantics in the data. The system relied a dozen schemas, and only one of them, but a central one, was supposed to be directly affected by the change. The question was how widely was the entities from this schema used throughout the system.

To answer this question, we constructed a tool that allowed us to reason about the complete system and especially about the interaction between Delphi and PL/SQL. The system was comprised of several sub-projects, most of which offered a client with a user interface that interacted directly with the database. The client was built out of Delphi source and forms, and both these artifacts could potentially utilize parts of the database either through dedicated components or direct SQL in strings. The PL/SQL side used stored procedures to specify server side behavior. The system did not access tables directly, but only through views.

The picture below shows the overview of those relationships. Black circles denote Delphi projects. Colored circles show PL/SQL stored procedures, and colored squares represent PL/SQL views. The colors are given by the different schemas in the database in which the entity resides. The schema that was subject to change was the blue one. Furthermore, filled color shapes denote the entities that are accessed directly from Delphi.

We distinguished several issues. There are several projects that work with isolated schemas such as the one close to the cyan entities, and there are also several projects that work with multiple schemas and they tend to be in the center of the picture. The blue entities are scattered throughout the space suggesting that they

are intertwined with other parts of the system. Furthermore, the fact that most blue entities are filled shows that they are used heavily in the Delphi side.



Dependencies between Delphi and PL/SQL entities.
Black circles represent Delphi projects. Colored circles are PL/SQL stored procedures.
Colored squares are PL/SQL views. Each color denotes a different schema.

This visualization provided a frame of reference, and we complemented it with several other analyses that focused on narrower issues and that led to both spike and strategic assessments. The overall effort of building the tool took about a person-month, which when compared with the overall investment in the system can be considered as insignificant. The definitive decision of choosing the future technology was still to be made at the time of the writing, but the existence of the tool and the resulting assessments provided the team with both confidence and insights that were not available before.

Checking architectural conformance of an outsourced system

The client was in charge with developing multiple systems both internally and through outsourcing. Once in production, all these systems were ran and mostly maintained in-house. To ensure a better integration and limit maintenance costs, the IT-Architecture department published a set of architectural guidelines specifying multiple aspects including the technology stack, security, exception handling and other patterns.

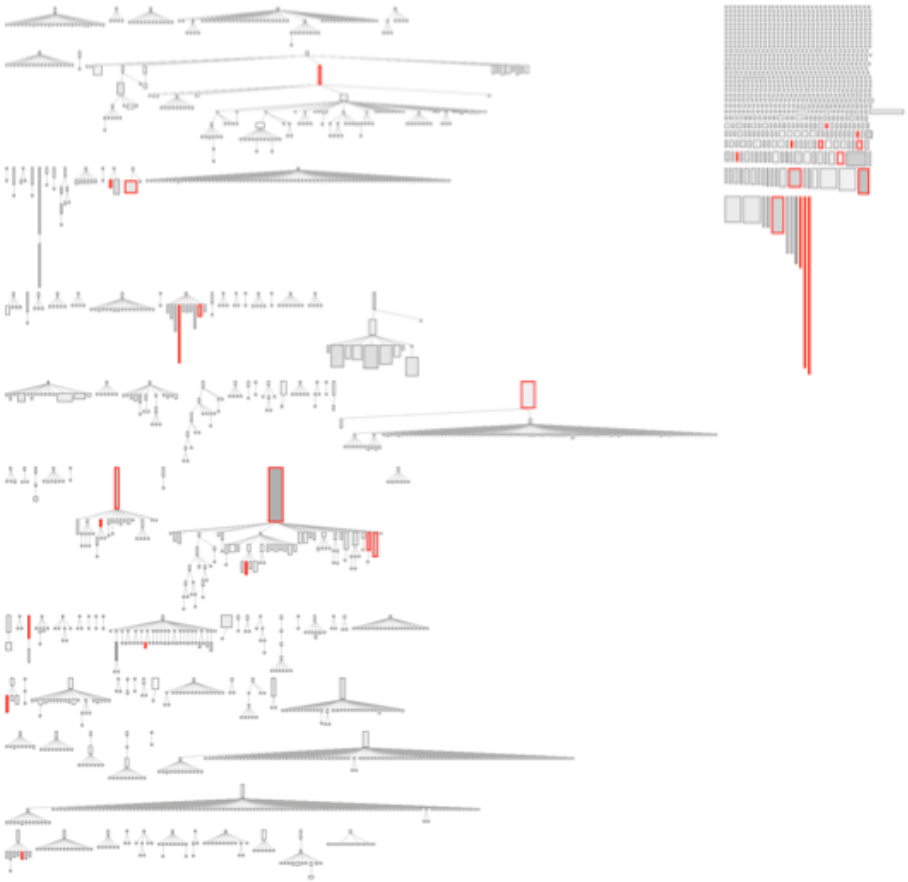
All projects were urged to follow these guidelines. However, these guidelines were only present on paper and violations could easily occur in the actual implementations. We were mandated to help checking how systems conform to these guidelines.

The first target was provided by an outsourced system that was approaching the end of first release. Because the outsourcing development team was not available, we based our analysis on interviews with the architects and on analysis of the documentation and the source code.

Using the textual architecture guidelines, we focussed on checking the architectural layers and interface boundaries. Indeed, we validated the coarse grained architectural rules. However, when we queried the system in more details, we identified a number of guidelines violations. For example, one of the detected shortcomings of the application was the poor exception handling that violated the architectural constraints.

We used several Moose-based visualizations and applied custom detection strategies to highlight points of interest and irregularities in the code. An example of such a visualization showing violations over the overall system structure is displayed below.

Once the non-conforming parts were identified, we proposed concrete recommendations for how to improve the structure of the system to conform to the desired guidelines. Among others we also pinpointed how the logic is distributed over the system and how the code duplication should be refactored. The architectural violations were scheduled for refactoring before the application was to go into production.



architectural violations highlighted on the overall class hierarchy

The overall project required less than a dozen person-days of effort. During the project, we did not limit our activity at pointing the violations, but we made it a point of identifying possible strategies for rectifications. This was possible because the stakeholders were available and could decide what parts have a deep impact on future maintenance and on integration with other systems.

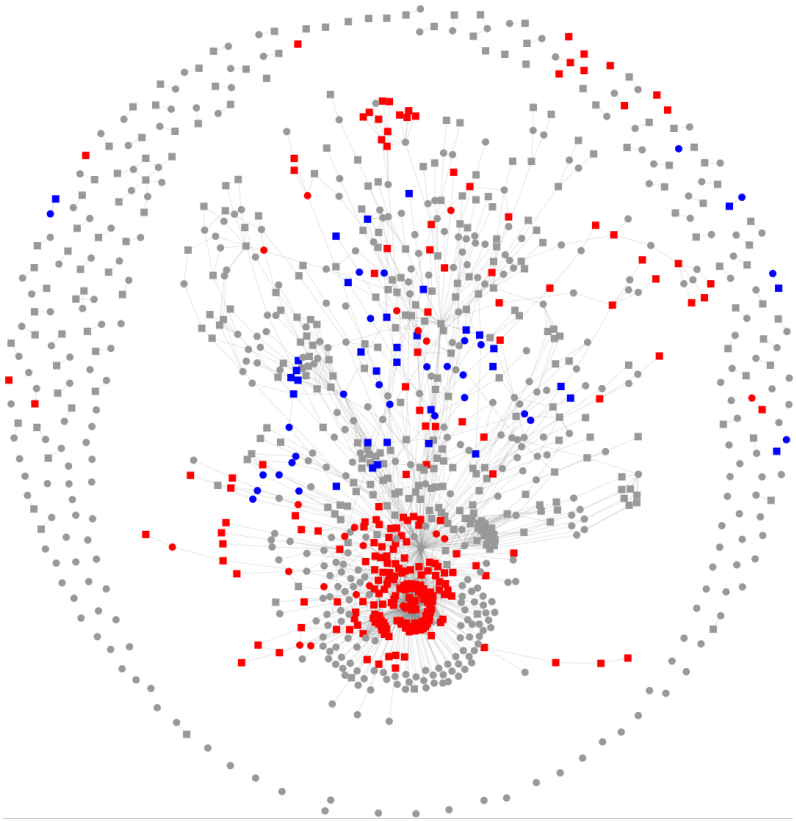
An interesting side effect was that we also detected several inconsistencies in the actual guidelines. These were clarified and corrected by the architects. Furthermore, the toolset could be reused on other projects.

www.humane-assessment.com/guide/buildup

Evaluating the splitting of an Angular-based system

The company had a large system with a user interface based on Angular 1 and totaling more than 300’000 lines of code. The team had already split the server side into (micro)services, and had started since one year to consider splitting the client side as well into smaller units to match the different development paces for different components. Splitting the client, however, turned out to be more difficult than initially estimated.

To expose the size and nature of the problem, we constructed an infrastructure to enable custom analyses over Angular code, and we used it for detecting several problematic patterns. The initial focus of the team was to split the system into two large subparts. To communicate the size of the problem, we created the visualization below to show all components and templates from the system together with all their inter-dependencies. Furthermore, we highlighted with red and blue the two components to show that they are significantly intertwined.



All components and templates. Red and blue denote the two components.

To put in perspective the implications of the picture above, let us consider how dependencies are defined in an Angular 1 application. In our example, we have a `moduleX.js` file defining a `someComponentA`, and relying on a template defined in `moduleX.template.html`.

```
moduleX.js
angular.module('moduleX', [])
  .config(function config($routeProvider) {
    $routeProvider.when('urlX', {
      templateUrl: ‘moduleX.template.html’, … }) })
  .component(‘someComponentA’ …)
```

The template defines a dependency to `some-component-b`.

```
moduleX.template.html
<some-component-b>
  <div> … </div>
</some-component-b>
```

`some-component-b` is defined in `moduleY` under the logical name of `someComponentB`. Furthermore, the component relies on a template `componentB.template.html`.

```
moduleY.js
angular.module('moduleY', [])
  .directive(‘someComponentB', function () {
     return {
       templateUrl: ‘componentB.template.html’, … } } )
```

The template includes `some-component-a` which corresponds to the `someComponentA` component defined above, thus leading to a cyclic dependency between `moduleX` and `moduleY`.

```
componentB.template.html
<some-component-a …>
  …
</some-component-a>
```

Typically, developers reason about such dependencies through text searches. However, in the case of Angular, this approach is hampered by the fact that the same component appears under two different looking names: a camel-case one in JavaScript sources, and a dash-separated one in HTML. Given the intricacies of dependencies seen in the visualization above, approaching them with a basic text search is far from being effective.

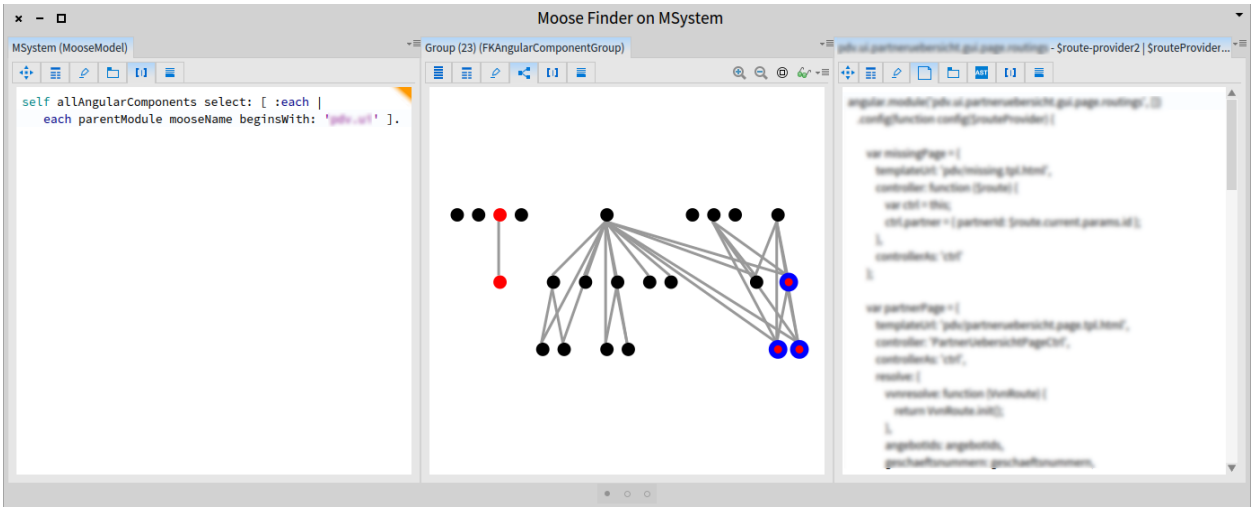
To produce the visualization we first created an importer that built an object model of the system. As can be observed in the example above, the JavaScript code specifies dependencies between components and templates through JavaScript strings. To provide a meaningful model, the tool had to take into account these semantics of Angular. Based on this tool, we could construct

several custom made queries and analyses to reveal impediments for the splitting project.

For example, an important input for finding the splitting path is the knowledge of which parts of the system is one component requiring. The following query answer this question for the `red` subpart:

```
red := (model allAngularComponents select: #isMRed),
       (model allAngularTemplates select: #isMRed).
all := red flatCollectAsSet: [:each |
       each withDeepCollect: #includedComponentsAndTemplates ].
all \ red
```

Some of these returning components and templates residing in common modules might be used in either the `red` or the `blue` subpart. Knowing where this happens can inform the splitting of external modules. The picture below shows the case of such a module. In the first pane we select all components coming from a nominated module. In the second pane we visualize all dependencies between these components, and highlight with red and blue the components that are used from the `red` and `blue` subparts. We can see that some are used in both `red` and `blue`, some only in `red`, and some are not used in either of them.



Investigating all components belonging to a common parent module.

Another kind of dependencies comes from how Angular allows one to define and inject services and other variables throughout the code. For example, the code below shows a `moduleZ` defining a `ServiceA` that relies on `ServiceB`.

```
moduleZ.js
angular.module('moduleX', [])
   .factory('ServiceA', function(ServiceB) { … })
```

Given that the two sub-parts were supposed to be top level, none of the services defined in them should be used outside of their boundaries. The following query reveals all defined variables that are defined in the red subpart, but are used from outside of this part:

```
self allAngularInjectables select: [ :each |
  each isMRed and: [
  each usedByInjectables anySatisfy: [ :user | user isMRed not] ] ].
```

The effort of constructing the initial infrastructure that allowed us to answer relevant questions took approximately 8 days. This was based on an out-of-the-box version of Moose that also comes with a basic JavaScript parser but that has no knowledge about Angular. The main effort was invested in understanding all variations and constructs that Angular provides.

Taking a step back, the cost of 8 days is close to irrelevant especially when compared with the overall cost of having a team of three dozen people not acting on a strategic need for one year.

Browsing a configuration system

The client approached us with the following problem: it became much too expensive and risky to extend the system. The system was made out of several hundred subsystems, each being written in Java. These subsystems were interconnected through dedicated interfaces. When asked, the client told us that there are some configuration files that describe how the connection should happen. He also mentioned that the configurations were using a declarative XML format.

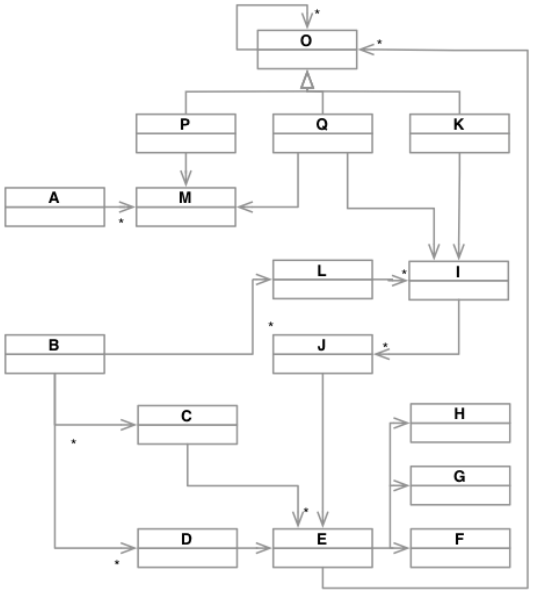
Upon further investigation, we identified that the overall system was indeed based on a custom engine that was put together using custom configuration files. Only there were more than 1000 such files, and they were describing more than simple connections. We also discovered that a scripting language was used to augment the information from the XML with extra constraints.

The problem was foremost one of perception: management believed the system to be modular and simple to configure. Yet, there was no formal description of the structure of the XML, and when we asked the technical team, they had a hard time explaining what each part of the XML meant.

We made it our goal to tackle this essential problem by offering an analysis tool that would help both management and the technical team to obtain an overview of the system.

The prerequisite for any data analysis is the identification of the structure for the data. Thus, as a first step, we analyzed manually several configuration files with the aim of capturing the structure in a diagram. The diagram below shows the anonymized result of the configuration analysis. The diagram helped to communicate with the client, and at the same time it served as a blueprint for the actual implementation of the model.

Afterwards, we created the importer that the configuration files. An important challenge was posed by the unification of data: because the system was developed over a decade, there were multiple ways of expressing the same information, and often these variations were all present in the same file.



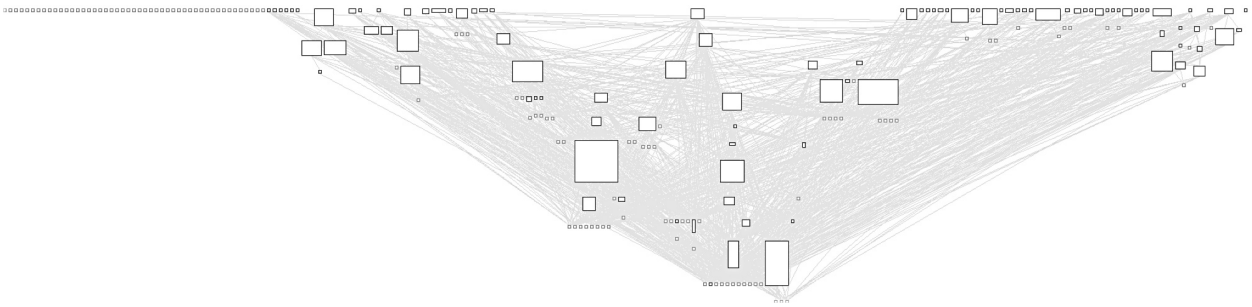
Our importing solution needed to deal with all these differences. To ensure both a speedy implementation and an accurate one, we used two strategies:

- We worked closely with the technical team to identify the meaning between various mappings. We would take examples from the configurations and they would formulate hypotheses related to what these examples meant in the system.
- We encoded the hypotheses in queries, and ran them against the complete set of configuration files. When a hypothesis was satisfied, we continue to build on it. When we encountered mismatches between a hypothesis and the configurations, we looked at the exceptions and went back to the technical team.

A further challenge was posed by importing the dependency information out of the adjacent scripts. While the XML syntax was clear, the syntax of the scripting language was less so. One approach was to build a complete parser for the language, but as our main aim was to build evidence of where the strategic problem comes from, we searched a solution that allowed us to extract dependency information without understanding the whole script. Indeed, we could identify a pattern that could easily be encoded in a partial parser.

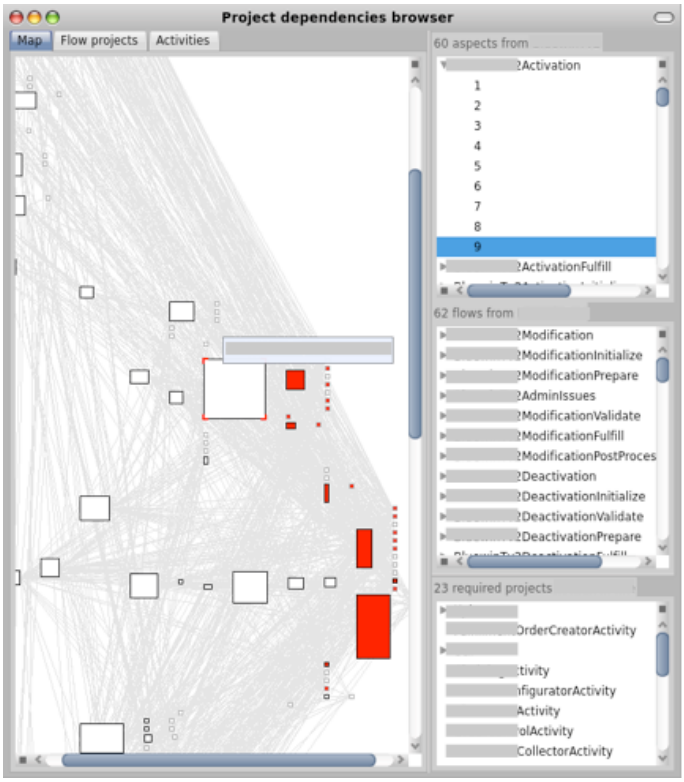
Using this approach we were able to build a first consistent model that had enough coverage to start performing actual analyses. At this stage, we could quickly see where the original problem came from: due to a lack of organization, there were simply too many dependencies. The visualization below depicts each system as a

box and each dependency through a line. The boxes are laid out such that the more a component was used, the more it appeared at the bottom.



an overview of all configuration dependencies

To ease the communication and further exploration, we augmented the view with metrics and we integrated all of these into an interactive browser.



a custom interactive browser for understanding configuration dependencies

The browser revealed the shape of the system for the first time, and the developers could identify multiple unexpected dependencies. Furthermore, the tool provided the evidence for why it was expensive to change the system.

Supporting multiple assessments of a system written in a proprietary language

The client had a mission-critical long living system written in multiple languages. The system offered a rich user interface built through many interconnected forms. Furthermore, an important feature of the system consisted in a proprietary language that could be used to customize or to create new modules.

Over more than a decade of development, the environment accumulated a large amount sources for multiple projects. While the proprietary tools offered some support for developing in this language, they offered no analysis infrastructure and because of that mistakes could easily occur.

We were mandated to create a dedicated infrastructure for supporting the assessment of programs written in this language and to relate them to the overall forms structure.

The lack of documentation about the structure of language posed a significant challenge. We started with a reverse engineering effort and we adopted an iterative approach through which we combined:

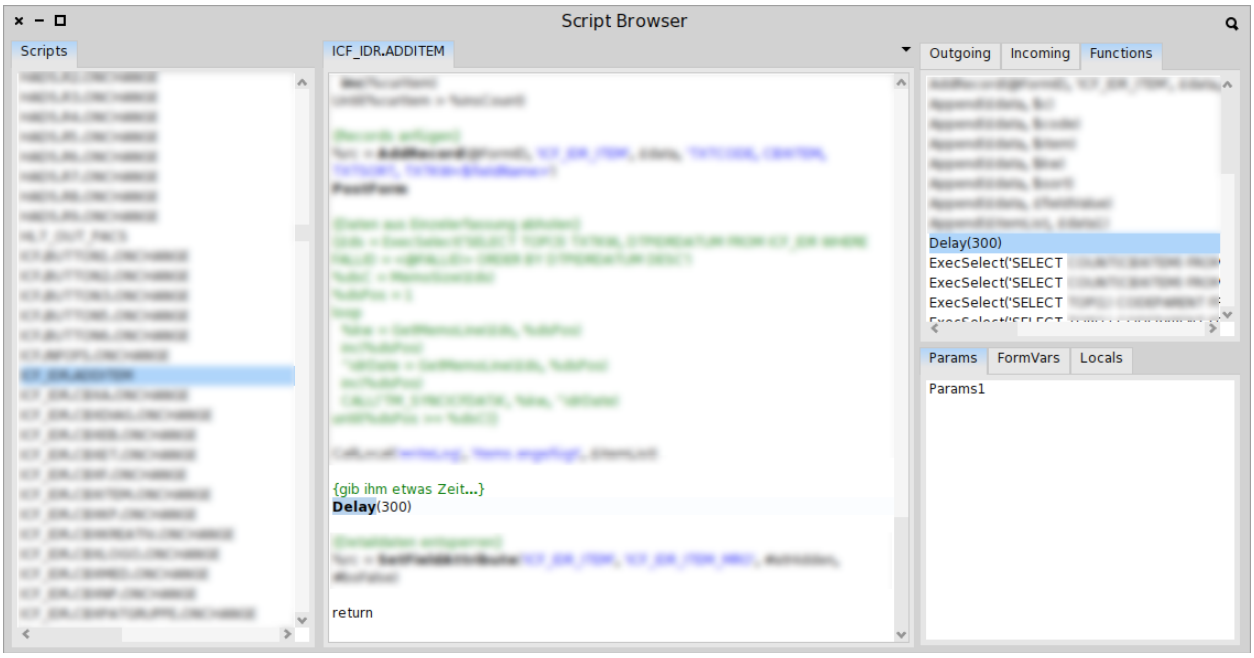
- Multiple developer interviews for recovering the meaning of programming instructions,
- Testing to check the accuracy of the produced parsers and importers, and
- Running variations of example scripts against the real system.

After two dozen days of effort, we produced a working version of the importer that could be used to produce a picture of the systems. The importer was based on several custom-made parsers built on top of the parsing infrastructure of Moose.

Using this importer, the team could start encoding their own concerns and ease maintenance. For example, applying the checkers on the first installation revealed more than 60 syntactic errors in production code. These errors were ignored by the runtime interpreter, but they were the source of great confusion for developers because something that appeared to be right was mysteriously not executed. Given that many scripts could easily have thousands of lines of code, the problem was significant. This

detection alone made the original buildup investment more than worth it. Thus, the team adopted the process of continuous assessment and checked these concerns on a regular basis for several projects.

Once the team became more confident in using analyses, we created several other tools on top of the code model to support various scenarios. For example, we create a browser that allowed the developer to search and navigate through all scripts using one single interface - as opposed to having to click on visual form builders multiple times to get to a script. Furthermore, the browser provided extra information about the call graph and the state of the script which brought an extra level of transparency.

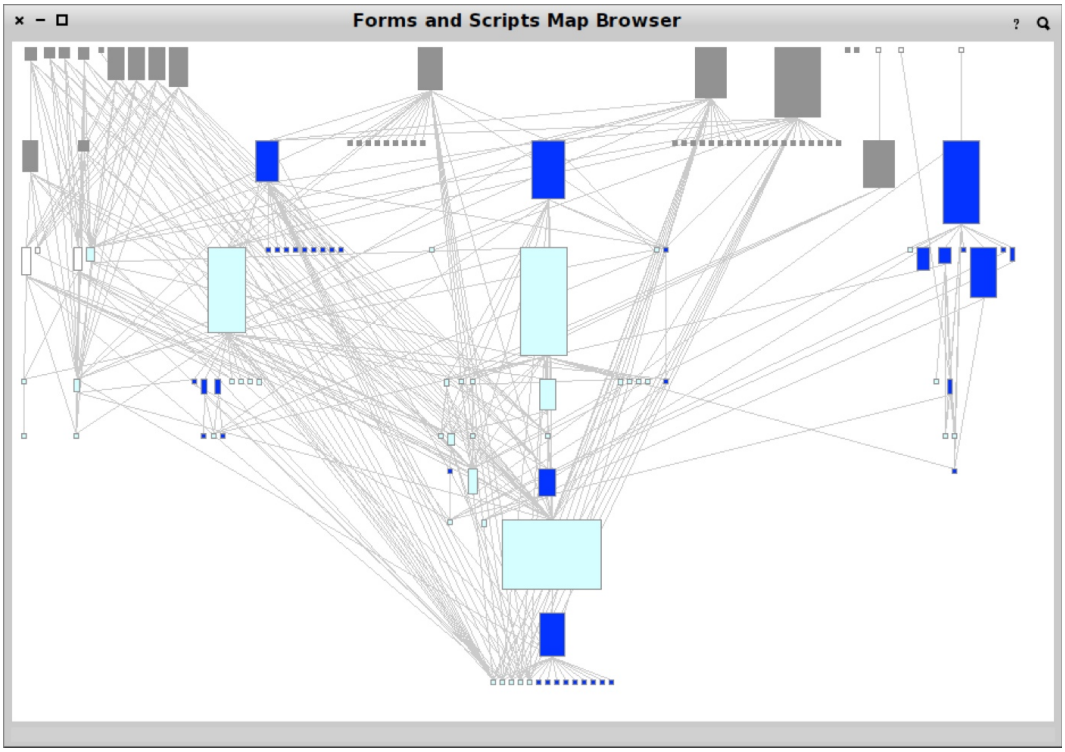


a custom browser for manipulating scripts

Another byproduct of this project consisted in a set of interactive browsers and visualizations meant to ease the dialogue between the technical and non-technical people concerning the status of the systems. In one case, a product owner needed to build the case for obtaining more refactoring budget. He knew that there were problems in the system, but he did not know how to present it to top management. To help with this problem, we constructed in half a day a dedicated browser that showed the dependencies in the system. This was possible precisely because the infrastructure of obtaining the code model was already present.

An example of a part of the system can be seen below. The visualization was interactive and it showed two types of modules

colored in two shades of gray, and two shades of blue depending on selection. To highlight the dependencies, the browser highlighted in blue the modules that were depending on the current selection. The product owner went on in the management meeting and said: "In theory, our system is built as a tree of dependencies. In reality, it looks different. This mismatch needs to be rectified." The CEO even clicked through the system asking for further clarifications. Finally, the budget was allocated.



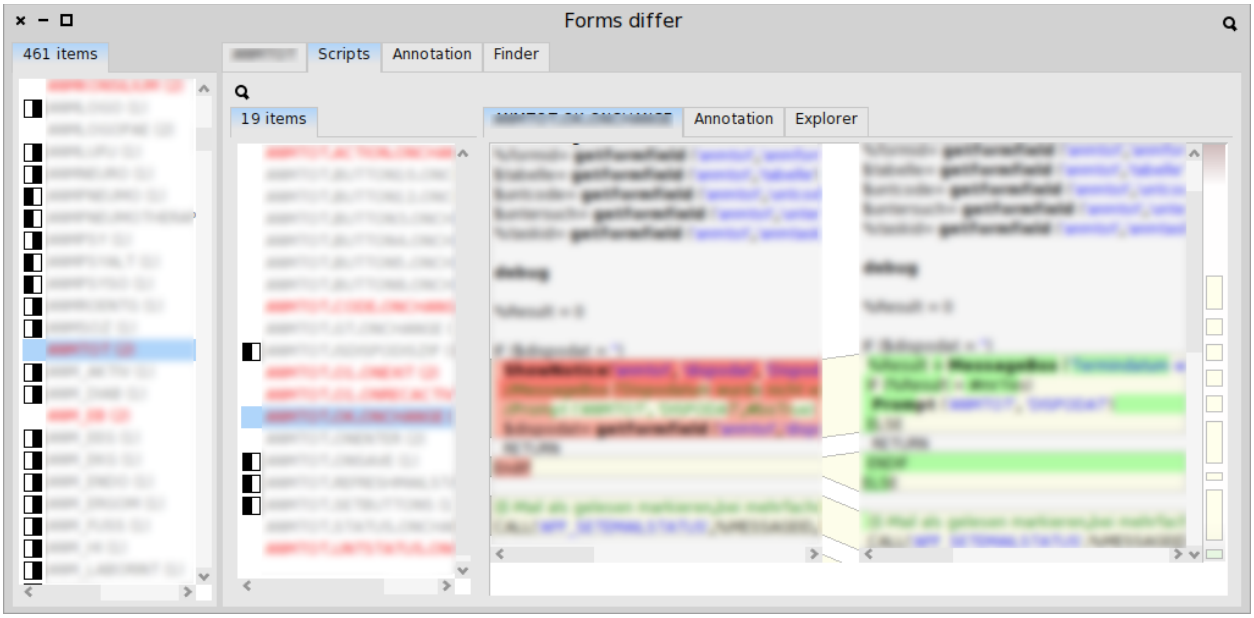
an interactive browser showing dependencies between modules

On another occasion, the client needed to merge two existing installations into a single one. The challenges were multiple fold. One of them was to identify the differences between the forms and scripts. At first, the developers started to investigate by using the standard text based tools, but it soon became apparent that this route has little chances of success given the sheer size and intricacies of the scripts and forms. The developers required a tool to ease their investigation task.

Scripts were indeed pure text and the global ones were located in separated files. Thus for those scripts it would have been enough to simply use a text diffing tool. However, many of the scripts were located inside the forms, and because these forms had complicated formats that even encoded the embedded scripts,

generic tools were rendered useless. To alleviate the problem, we constructed a dedicated browser for diffing the two code bases.

The screenshot below shows a glimpse of the tool. Given that the infrastructure for analyzing one version existed, the cost of the new tool was measured in a few days. The tool allowed the developers to go through the entire system within a day.

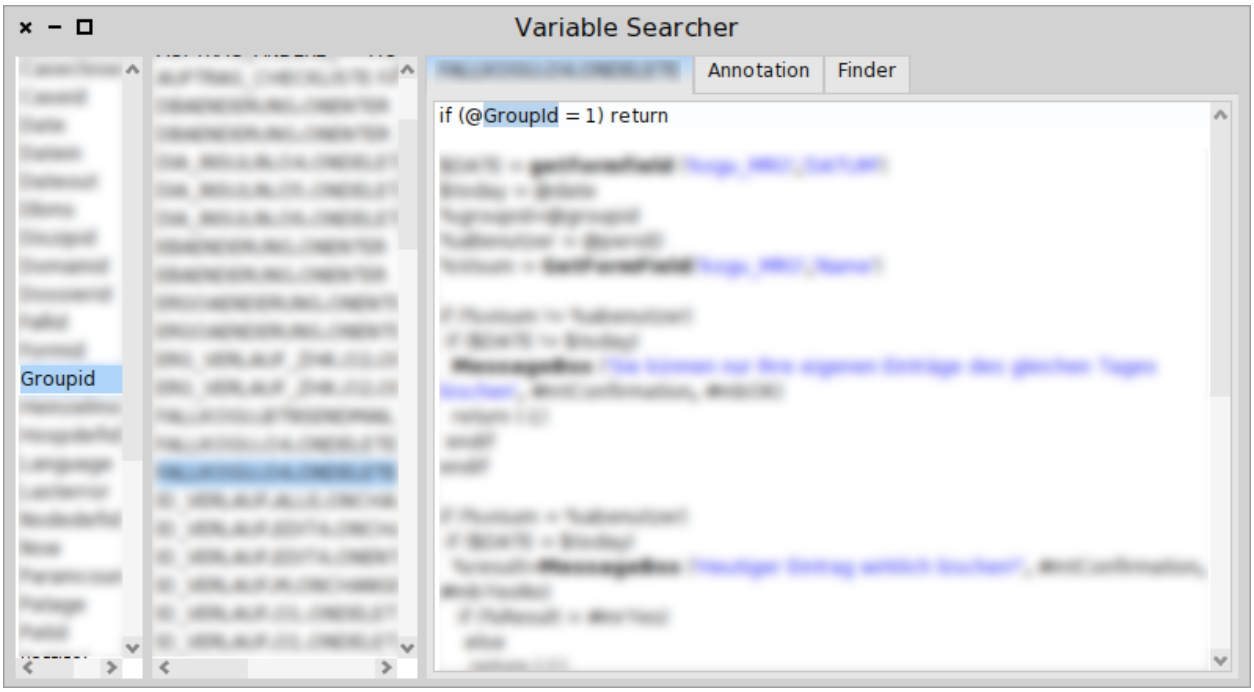


a dedicated browser showing detailed differences between two code bases

In the same project, the team identified another problem: merging the two systems also implied that the database ids for various entries needed to be modified as well. The question was: how will this change affect the scripts. We needed to find all cases of using explicit ids in the scripts. However, given that the scripting language was a low level one, and that the code base was filled with numbers of all sorts, there was no way to automatically identify where the problematic ids were being used simply by looking for the numbers.

To solve the problem, we built another dedicated browser that employed several heuristics to provide hints of code zones that could be problematic. Building the browser required less than a day, and going through all locations required just a few hours. To make things even easier, the browser also provided a simple way to annotate the locations and to produce a report of the annotations. This list was then used for fixing the problematic places during the migration. This was a strategic problem that

could essentially be answered with an insignificant amount of effort.



a dedicated browser showing direct usages of ids for given variables

These are but of few of the use cases served by the original infrastructure. The original investment in a tooling buildup paid off multiple times.

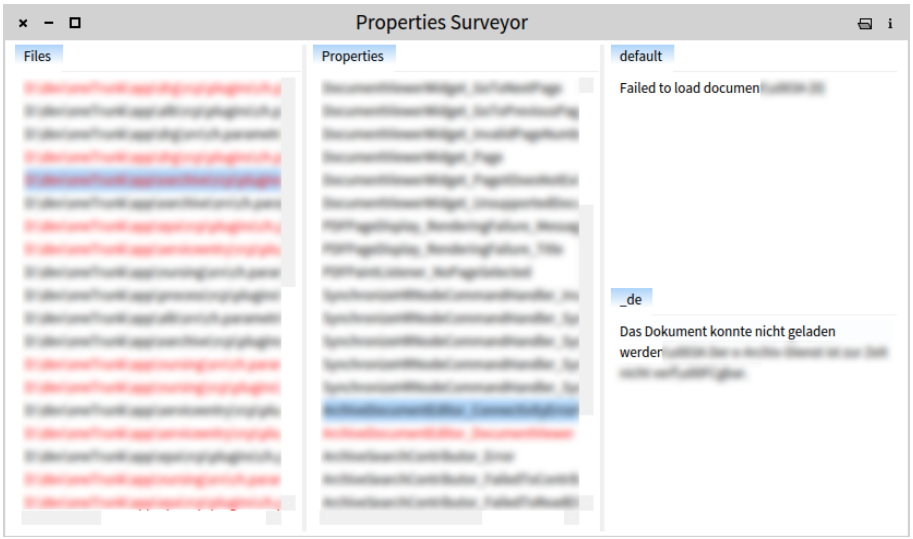
Identifying missing translations

The system had to work in a multi language environment, and for this purpose it worked with a typical Java internationalization framework that relied on translations being placed in files having a suffix for each language. A sample could look like: sample_en.properties, sample_de.properties ...

There were multiple thousands of such property items and they had to be translated in three languages. The properties were added and removed by engineers during the regular development work. Afterwards, the files were processed by the translation team to ensure the quality of the translation.

However, the translation team did not know what were the required properties to be translated, and even if they spent a significant amount of time manually checking properties, often translations were missing.

To solve the issue, we built a browser that revealed the missing translations. The tool relied on several heuristics, but essentially, it collected all properties and checked to see that all language variations have entries for each of them. The effort took around two days and it was enough to transform the situation into a manageable one.



a browser that showed in red all files and properties that were missing translations