humane assessment

the missing software engineering method

tudor gîrba
www.tudorgirba.com
www.humane-assessment.com

Assessment is expensive and you already pay for it

The goal of assessment is decision making. Everyone makes decisions. Managers decide about the overall development. Architects decide the broad technical direction. And so do software engineers: they decide daily the course of the implementation.

In fact, several studies report that software engineers spend up to 50% of the time assessing the state of the system to know what to do next. In other words, assessment accounts for half of the development budget. These are just the direct costs. The indirect costs can be observed in the quality of the decisions that result from it.

Assessment must be recognized explicitly and approached as a distinct discipline. It is too important to do otherwise. Only by making it explicit can it be optimized. The challenge is significant because it requires a paradigm shift. The promise lies in the costs that can be decreased when going from ad-hoc to structured. But, the good news is that the budget is already allocated and is being spent.

Software systems are not only large, but they are complex in contextual ways. To be effective, assessment must be tailored to deal with the context of the system and of the problem at hand.

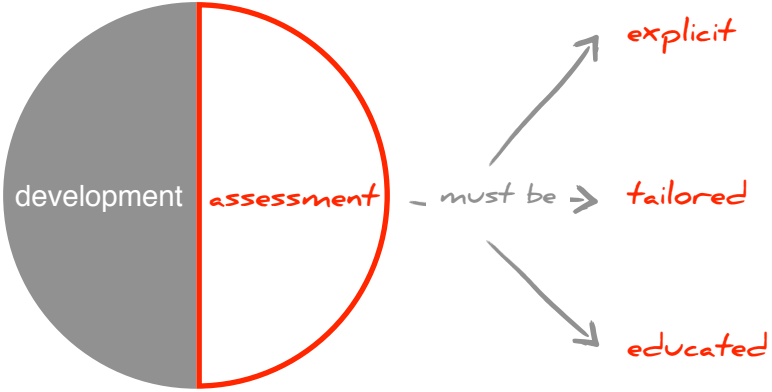
The ability to assess a situation is a skill. Like any skill, it needs to be educated. From a technological point of view, a tool that checks software is software, too, and software engineers already know how to produce software. The challenge is to shift the focus from what to develop, to how to check what to develop.

Assessment is expensive.

Make it manageable by making it explicit.

Make it effective by tailoring it to your needs.

Make it efficient by educating your skills.



Assessment accounts for as much as half of the software development effort. These are just the direct costs. The indirect costs can be observed in the quality of the decisions that result from it.

Currently, it is approached ad-hoc and ineffective. This must be changed.

Software assessment needs rethinking

Software assessment is a critical activity that is too expensive to approach in an ad-hoc manner.

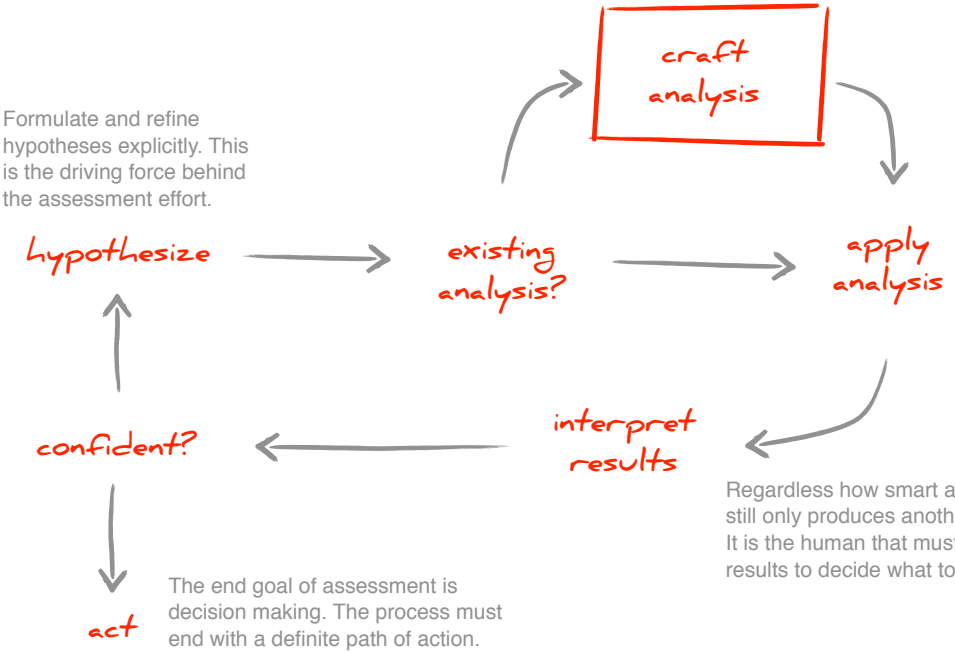
In practice developers mostly rely on code reading. This solution provides fine grained details, but it does not scale when we want to reason about a system as a whole. To put it in perspective, a person that reads one line in two-seconds would require approximately one month of work to read a quarter of a million lines of code.

Nobody has this amount of time at their disposal. Thus, the reading is typically limited to a part of the system, while the overview is left for the drawing board from the architect's office. Following this strategy, most decisions tend to be local, while the strategic ones are mostly based on inaccurate or dated information.

To rectify the situation you need tools that help us crunch the vastness of data. However, not any tool does the job. Many tools exist to deal with various aspects of data and software analysis, but most of them take the oracle way: they offer some predefined analyses that provide answers to standard questions.

Receiving a standard answer is great when you have a standard question. However, it turns out that most of the time our questions are not quite standard. Software systems tend to present many contextual details due to various factors: different technologies, different third party libraries, different architecture decisions and so on. In these situations, regardless how smart the analysis is, it is of limited use if it does not allow us to contextualize it.

The essence of the humane assessment method consists in crafting tailored tools for carrying out dedicated analyses.



Formulate and refine hypotheses explicitly. This is the driving force behind the assessment effort.

Software systems are complex and they present plenty of context-specific problems. A custom problem requires a custom solution. To be effective, it is critical to craft an analysis tool for it. This is an activity that must be captured explicitly as part of the development process.

Regardless how smart an analysis is, it still only produces another set of data. It is the human that must interpret the results to decide what to do next.

The end goal of assessment is decision making. The process must end with a definite path of action. Only then it has practical impact.

Adopting Humane assessment

Decisions must be taken everyday. Every time you scrutinize a software artifact to figure out what to do next, you are actually performing an assessment.

Assessment has to be embedded in the development process and in the organization.

Assessment must become a daily reality. The core idea of the humane assessment method is for engineers to craft and rely on tailored tools. This offers a general guideline for approaching assessment in a systematic way.

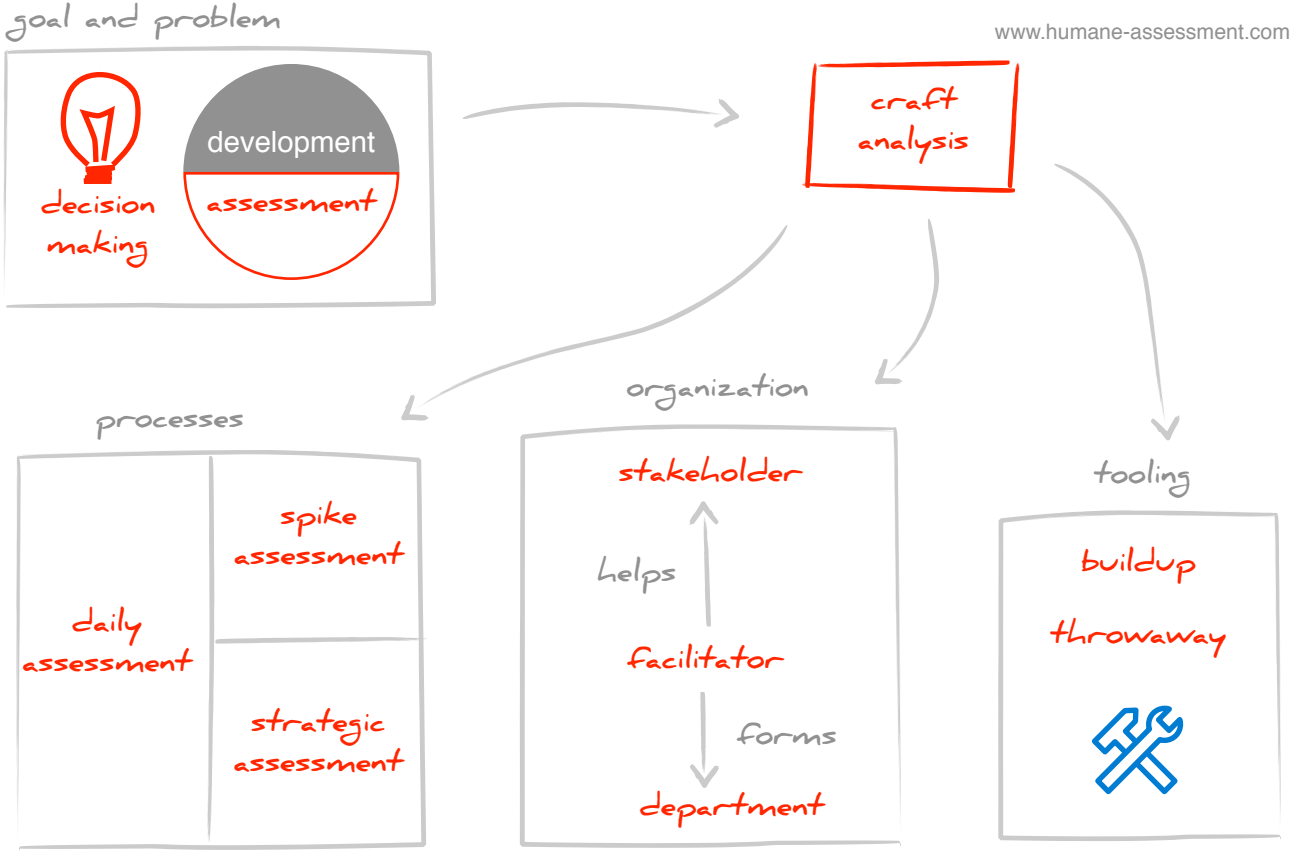
However, to get the method effective during development, we need to take a step further and integrate it in subprocesses of the overall development process.

Assessment must always have a problem at the center. Depending on the nature of this problem, there are several ways to integrate the method in the development process: as a daily assessment routine to capture, document and improve the state of the system; as spike assessments to support fast decision making; or as strategic assessments to support coarse grained decision making.

Assessment requires dedicated skills. To ensure their presence, you have to capture it in the organization through explicit roles that carry the dedicated knowledge. Two distinct roles are critical: the stakeholder and the facilitator.

A stakeholder can be either a technical or non-technical team member. She owns the problem and is the driver of the assessment effort. The facilitator is an engineer whose main goal is to ensure a smooth dialogue between the stakeholders and the system. He makes certain that the cost of assessment remains accessible for the stakeholders.

An important component of assessment is indeed the tool infrastructure. Moose offers such an infrastructure that makes assessment practical. However, in the end, it is the skills and activities of the team that make the difference.


goal and problem
www.humane-assessment.com
development
assessment
decision making
craft analysis
processes
spike assessment
daily assessment
strategic assessment
organization
stakeholder
helps
facilitator
forms
department
tooling
buildup
throwaway

Moose makes assessment practical

Assessment requires a tool infrastructure that makes it practical to build custom analysis solutions for your systems and your problems. You need to control your effort with an appropriate infrastructure.

The Moose analysis platform was designed for it. Moose is an extensive platform for software and data analysis that makes software assessment practical.

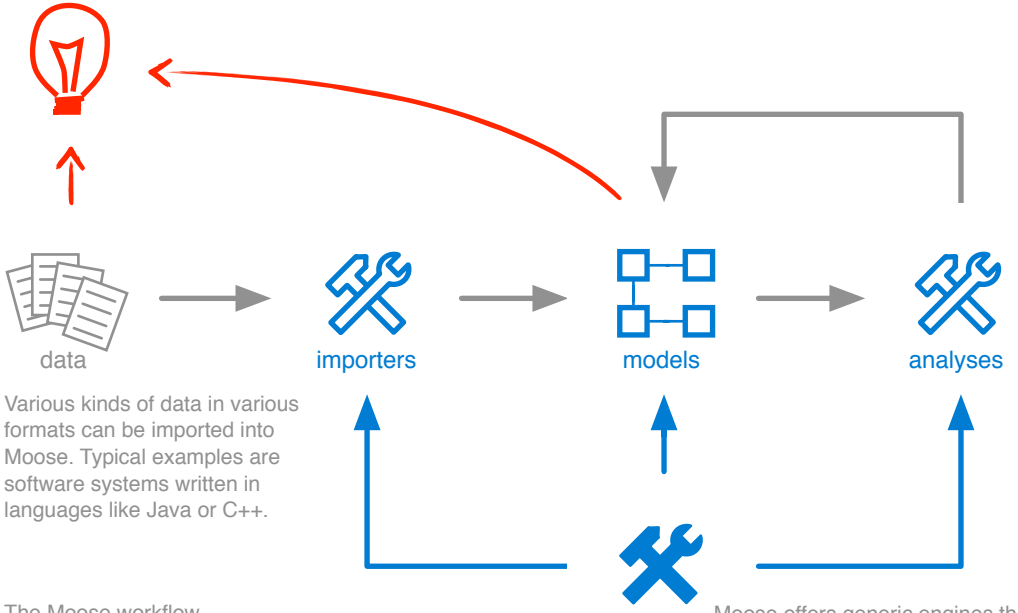
It is a free and open-source project started in 1996. Since then it grew and it is now developed in several research groups and it is increasingly being applied in industrial contexts. In total, the effort spent on Moose raises to more than 150 person-years of research and development.

The design of Moose is rooted in the core idea of placing the analyst at the center and of empowering him to build and to control the analysis every step of the way.

Data from various sources and in various formats is imported and stored in a model. For example, Moose can handle out of the box various languages such as: Java, Smalltalk, or C/C++.

On top of the created model, the analyst performs analyses, combines and compares the analyses to find answers to specific questions. Moose enables this in two distinct ways. On the one hand, Moose comes with multiple predefined services such as: parsing, modeling, metrics computation, visualization, data mining, duplication detection, or querying. These basic services are aggregated into more complex analyses like computation of dependency cycles, detection of high level design problems, identification of exceptional entities and so on. On the other hand, Moose is more than a tool. Moose is a platform that offers an infrastructure through which new analyses can be quickly built and can be embodied in new interactive tools.

Using Moose we can easily define new analyses, create new visualizations, or build complete browsers and reporting tools altogether.



Various kinds of data in various formats can be imported into Moose. Typical examples are software systems written in languages like Java or C++.

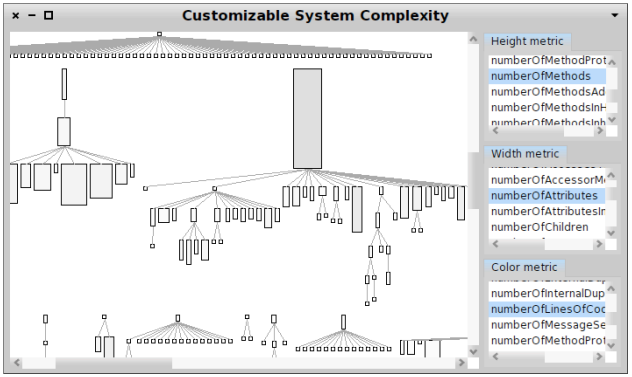
Moose offers various tools that deal with metrics, clustering, querying, visualizing, and interactive browsing. A key concept is that the result obtained after applying a specific analysis tool are fed back into the model and are available for further analysis. This enables an iterative process in which the analysis is built and refined gradually.

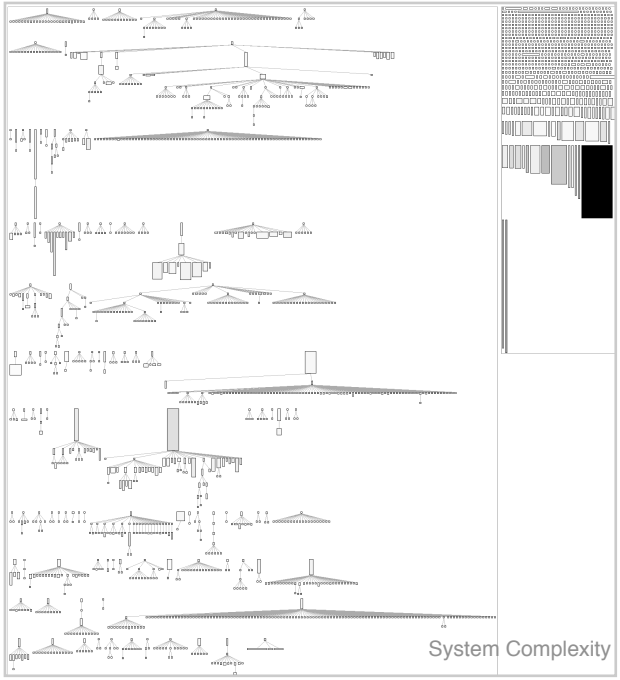
The Moose workflow
www.moosetechnology.org
www.themoosebook.org

Moose offers generic engines that enable the analyst to craft new tools fast: from parsers, to models, and to visual and interactive tools.

Visualizing class hierarchies

The System Complexity is a polymetric view that shows classes as nodes and inheritance as edges. Furthermore, each node is enhanced visually with three metrics: the height is given by the number of methods, the width is given by the number of attributes, and the color is given by the number of lines of code.

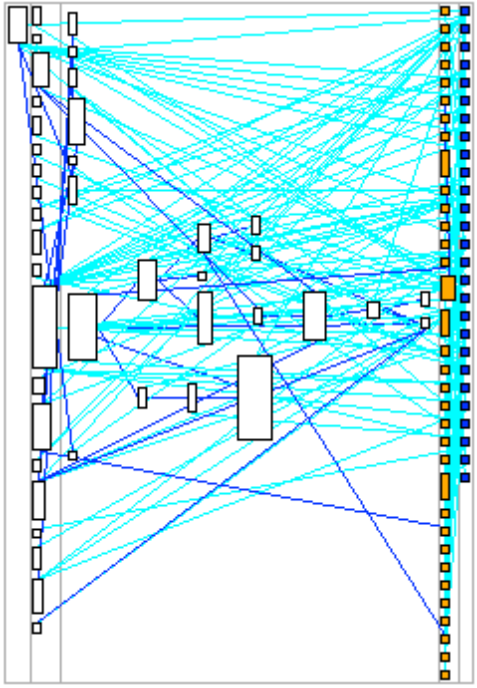




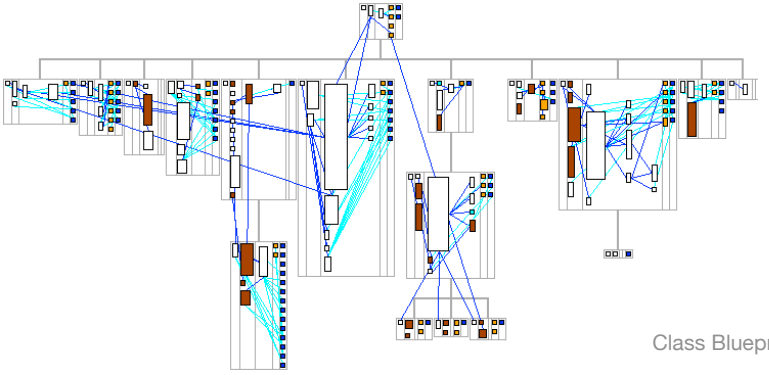
Visualizing the internal structure of classes

While the System Complexity visualization shows the overall class hierarchies, the Class Blueprint shows the internals of classes.

The figure to the right shows an example of a class with multiple methods. The figure to the bottom shows a larger hierarchy.



Class Blueprint

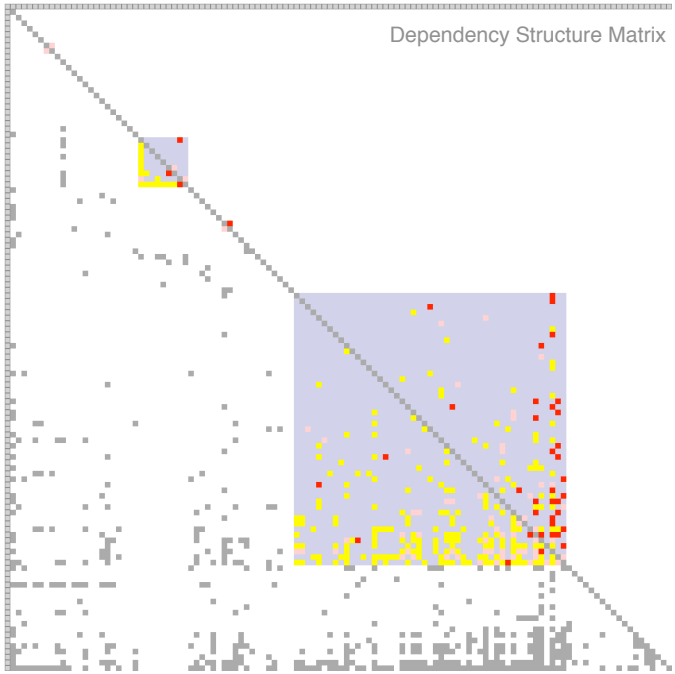


Visualizing dependencies between modules

The Dependency Structure Matrix is a visualization that shows the dependencies between modules at a coarse grained level.

The list of modules is shown on both rows and columns and each dot represents a dependency. The algorithm tries to place all dots below the diagonal. If a dot appears above the diagonal, it signifies that a cycle was detected in the system. The visualization highlights these cyclic dependencies with a shaded background.

In the example to the right, we have a system with more than 100 packages. In total there are four cycles, of which one formed by multiple modules.

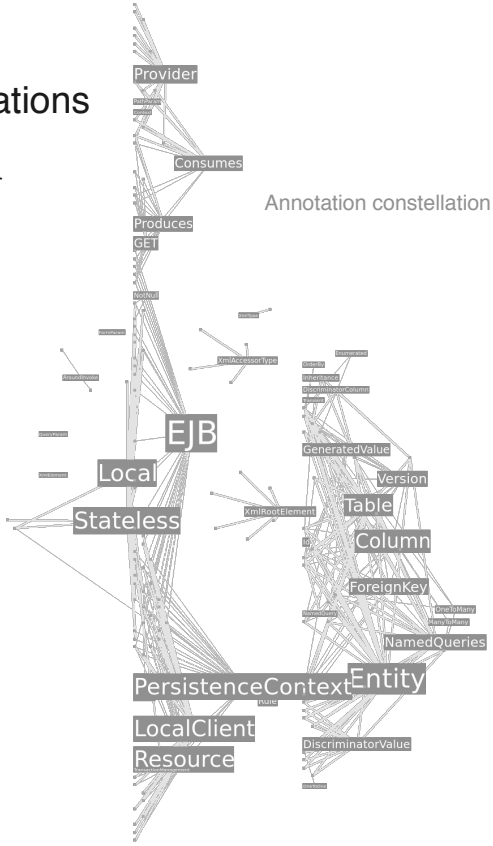


Other software visualizations

The previous pages show a couple of classic visualizations. While they provide valuable views over a software system, they are only some of the visualizations offered. Two of these are shown to the right.

The first reveals the complexity introduced by source code annotations. In the example, we see the types of annotations used in a system and how they are related.

The second shows how source code duplications appear between parts of the system. In the example, we see the same packages on each column, and a connection between them denotes a duplication: internal if the line is horizontal, or external if the line is oblique.



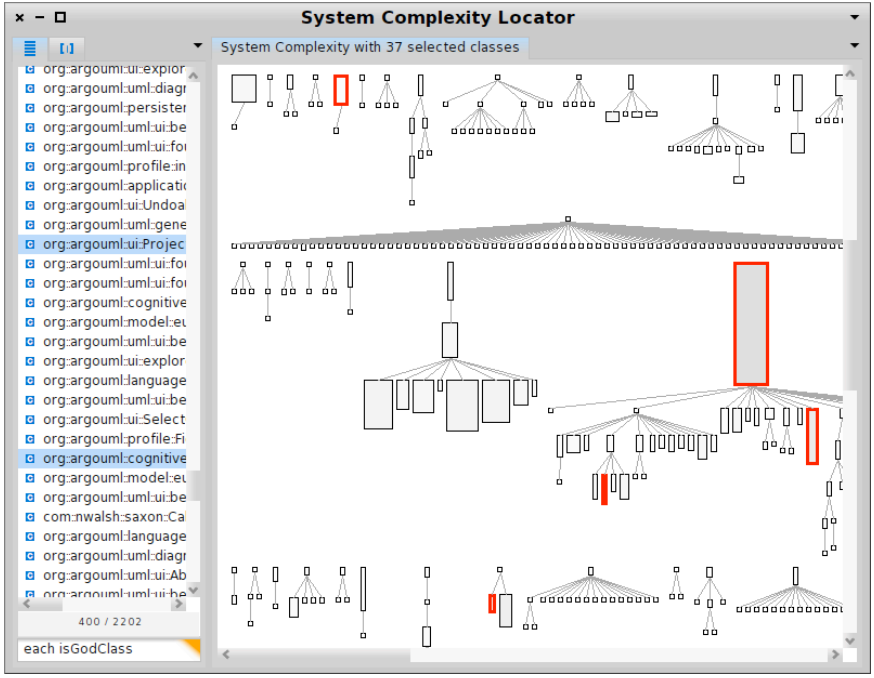
Annotation constellation

Side-by-side duplications

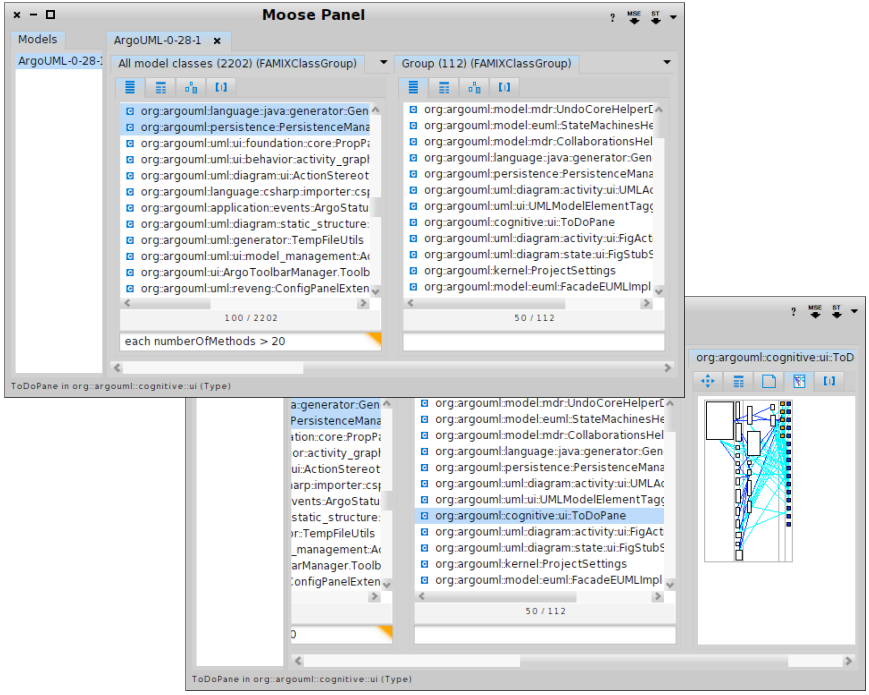
Querying models

Visualizations provide maps of what otherwise is intangible information. Using these maps we can get a sense of the structure of the large amounts of data. Another mechanism for understanding data is querying. Queries are instructions given to the computer to retrieve a part of the data that conforms to the given predicate. Thus, they help the analyst to focus on smaller set of data.

Moose offers a rich scripting API. Furthermore, it also offers the possibility to relate multiple views, such as query results and visualizations. For example, the picture to the right highlights a set of classes detected as being flawed on an overall System Complexity visualization.



Browsing models



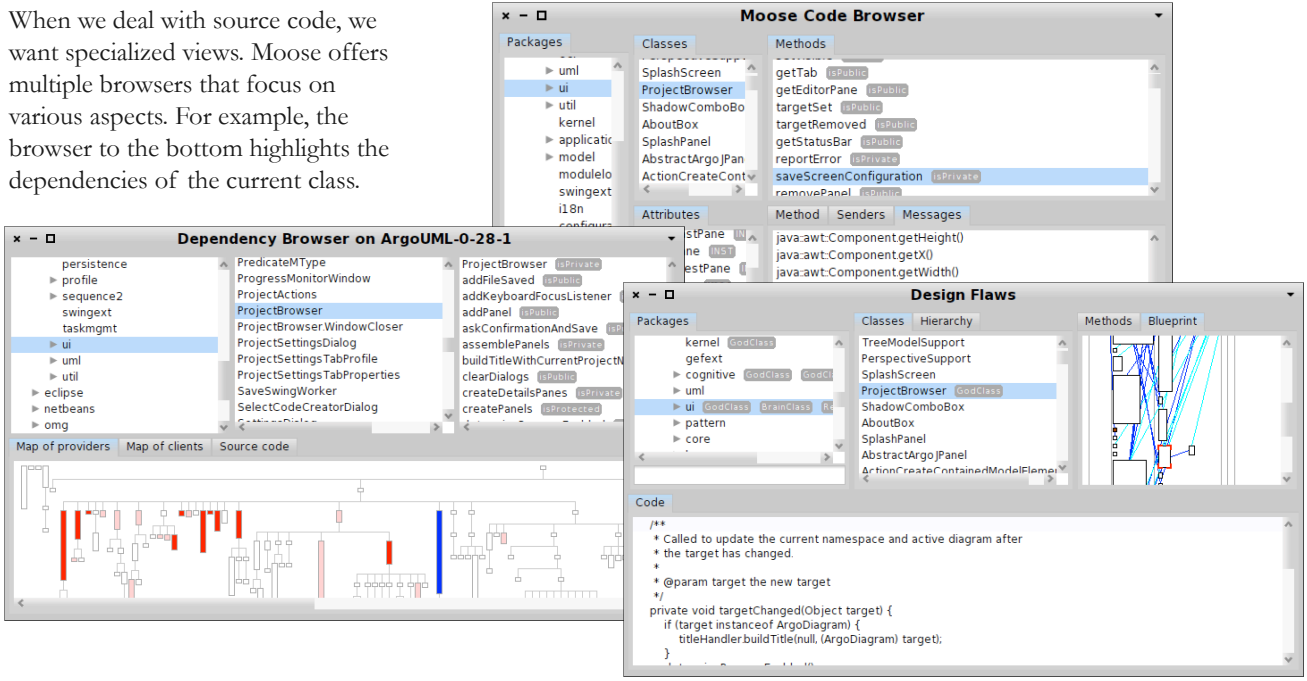
Moose offers multiple tools and mechanisms for exploration. The basic exploration tool offers a generic user interface that can navigate through any model, be it of source code or otherwise.

The tool employs a Finder-like design: selecting any entity spawns the details of this entity to the right. The details can be presented in various forms (such as simple lists, tables, or visualizations), each of these forms offering a high degree of interaction.

For example, the screenshot at the top shows a list of classes to the left and the result of a query is spawned to the right. The screenshot to the bottom shows various visualizations.

Browsing source code

When we deal with source code, we want specialized views. Moose offers multiple browsers that focus on various aspects. For example, the browser to the bottom highlights the dependencies of the current class.



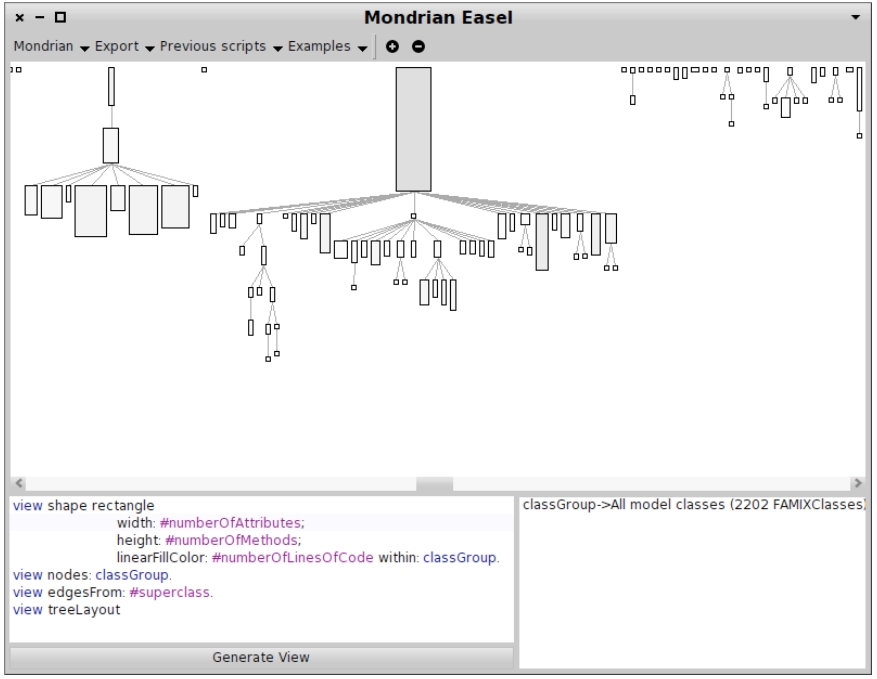
Painting custom visualizations

Often, we need dedicated visualizations. Moose allows the analyst to build such specific visualizations through a domain specific scripting interface.

The screenshot to the right shows the Mondrian Easel, a dedicated editor for such visualizations. The presented example, shows how concise it is to draw a System Complexity view of a set of classes.

Using this infrastructure, we can build in a concise and declarative way visualizations that exhibit patterns specific to the data and the problem of interest.

Most visualizations in Moose are drawn using this engine.



Crafting custom tools

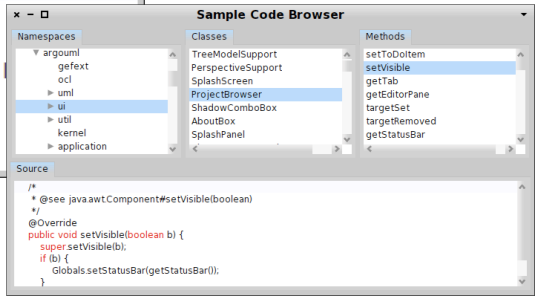
Large models hold many details. To tame these models, we need interactive browsers that help us understand their inner structure and relationships.

Moose offers Glamour, an engine for building custom interactive tools. For example, the script to the right builds a complete code browser that displays packages, classes, methods and source code.

Using this engine, Moose enables the analyst to craft quickly custom tools that target custom data sets, or custom navigation use cases.

Workspace

```
| browser |
browser := GLMTabulator new title: 'Sample Code Browser'.
browser
    row: [ :r | r column: #namespaces; column: #classes; column: #methods ];
    row: #details.
browser transmit to: #namespaces; andShow: [ :a |
  a tree
    title: 'Namespaces';
    display: [ :model | model allNamespaces select: [ :each | each isRoot ] ];
    children: [ :namespace | namespace childScopes ];
    format: [ :namespace | namespace stubFormattedName ] ].
browser transmit from: #namespaces; to: #classes; andShow: [ :a |
  a list
    title: 'Classes';
    display: [ :namespace | namespace classes ];
    format: [ :class | class stubFormattedName ] ].
browser transmit from: #classes; to: #methods; andShow: [ :a |
  a list
    title: 'Methods';
    display: [ :class | class methods ];
    format: [ :method | method stubFormattedName ] ].
browser transmit from: #methods; to: #details; andShow: [ :a |
  a text
    title: 'Source';
    display: [ :method | method formattedSourceText ] ].
browser openOn: MooseModel root allModels last.
```

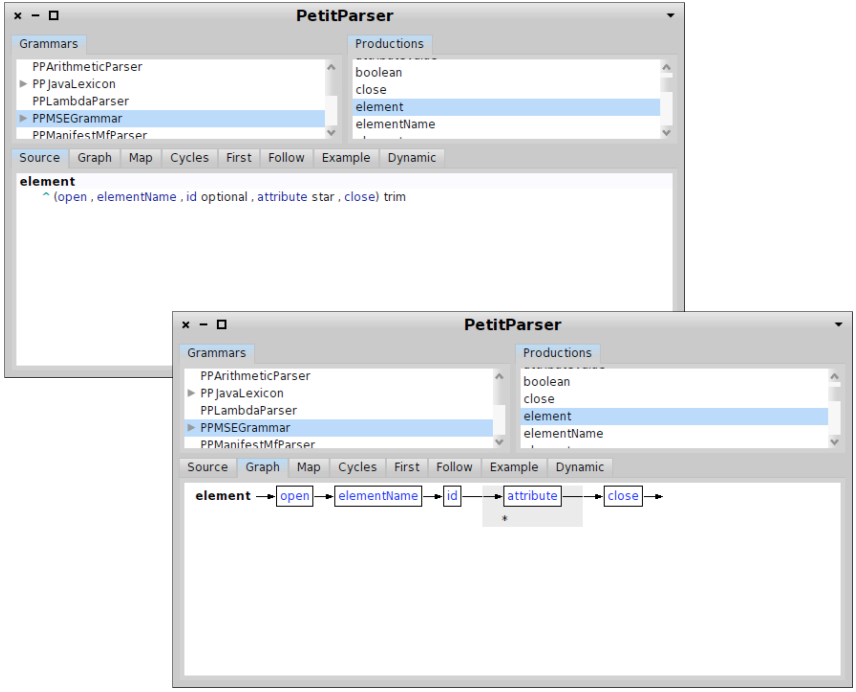


Parsing new languages

To analyze software systems, we first need to be able to parse them. Moose comes with PetitParser, a generic framework for defining dedicated parsers.

For example, the pictures to the right shows the PetitParser development tool open on a part of the grammar of the MSE file format, the default format for models import/export. The development tool offers multiple perspectives including a graphical representation, a grammar cycle detection, and a debugger.

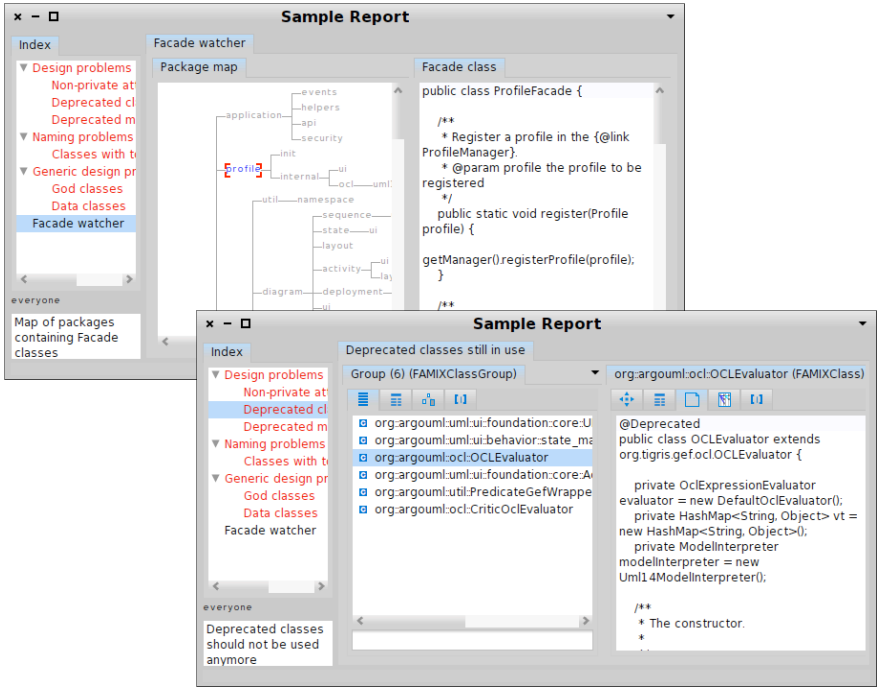
This infrastructure enables the analyst to analyze various data sources spanning from domain specific languages to custom configuration files.



Building live reports

While a significant part of Moose is focused on providing interactive capabilities to build and perform analysis, it is often desirable to carve in stone a set of concerns that we want to watch for, and afterwards have a tool to check them and to produce a report. Moose offers a report building engine that enables the analyst to define custom concerns.

In the example to the right, we have a report on a source code model checking several concerns. The concerns are marked in red because violations were found in the system. This, these concerns can act like tests against the model. Furthermore, concerns can also be visualized various ways.



the book

the book
that shows
the outside
the inside and
the philosophy of
the Moose platform

by Tudor Gîrba

www.themoosebook.org

Success story: Daily assessment in Scrum projects

Our client was responsible for the development and maintenance of multiple software projects. The teams were following a Scrum-based development process, but as the size of the projects increased, the need for a continuous assessment of the systems became apparent to ensure the quality and compliance of the work. We were mandated to improve this state by introducing daily assessment.

We approached these issues in two ways: (1) we introduced custom reporting tools, and (2), we coached the teams to make their decisions explicit, and to check them against the reality of the system.

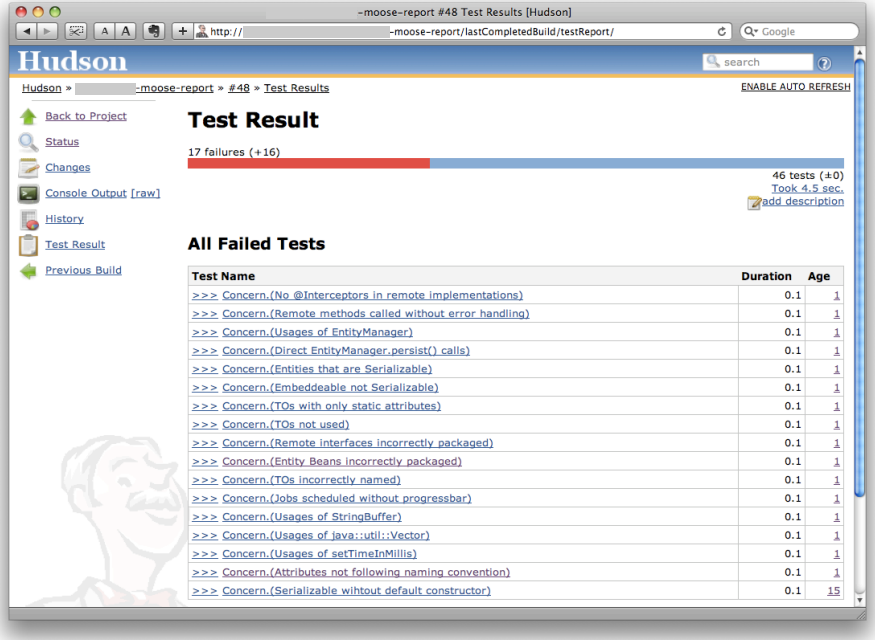
To get stakeholders to define their concerns explicitly, we held several interactive workshops in which we identified problems and showed how these can be answered fast. We then encoded these concerns into rules that addressed several areas including quality assurance and documentation.

We used Moose to craft the reporting tools. These tools were integrated into the continuous integration process and provided continuous and contextual feedback about how the concerns were reflected in the system. An example of such a report can be seen on the next page.

The most important component of our work was to affect the development process to get the team take the actual state of the system into account and to correct the situation continuously.

First, the concerns were made explicit and captured in form of rules that were integrated into the continuous reporting. The detected problems were discussed on a daily basis in a dedicated standup meeting. If the rule was considered valid, it was either resolved immediately, or planned for a later iteration. If it was determined that the concern was not valid, the rule was adjusted.

The quality of the projects improved in a short amount of time with only little overall overhead, and the confidence of the team and management increased.



Example of a Moose report integrated into the Hudson continuous integration server.

Success story: Strategic assessment of architecture conformance

Our client's IT-Architecture department published a set of architectural guidelines and coding conventions for the internal and external software systems. They needed a way to verify compliance of one of their software system.

Our role was to review the application and verify its conformance to the guidelines.

The first step was to obtain an overview of the system. We based our analysis on reading both technical and business documentation and on analyzing the actual source code.

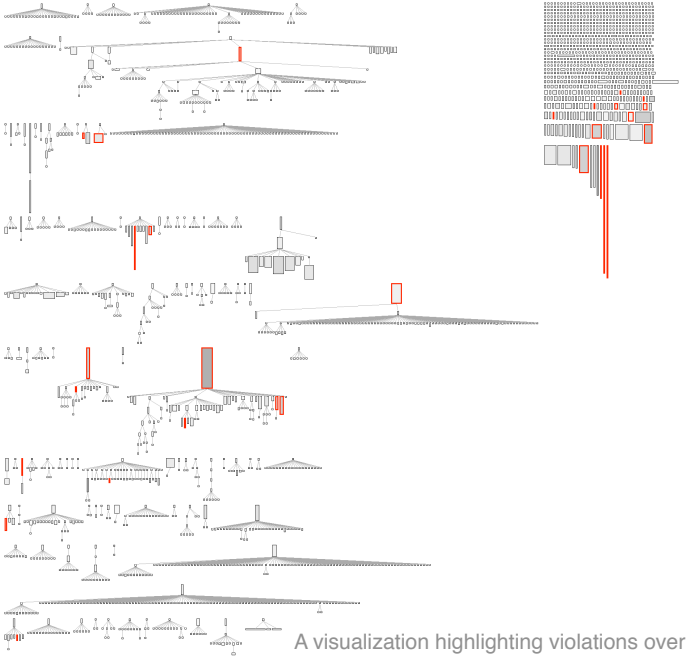
Using the architecture descriptions, we focussed on checking the architectural layers and interface boundaries. We validated the coarse grained architectural rules. We used visualization techniques and applied several custom detection strategies to highlight points of interest and irregularities in the code.

On closer inspection through queries and selective code reading, 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.

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.

We also provided a summary of the overall code quality. For this purpose we used several techniques such as metrics, queries and visualizations. Among others we 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.



A visualization highlighting violations over the structure of the system

Success story: Custom analysis tool for custom configurations

The client had a large software system composed of several hundred subsystems, each being written in Java. The overall system was based on a custom engine that was put together using a multitude of custom configuration files specified in XML and in another custom language. Given the particularities of the project, a dedicated tool was needed to analyze it.

We received the task of crafting a dedicated tool that would be able to provide an overview of these configurations and expose their various relationships.

The prerequisite for any data analysis is the specification of the meta-model for the data. Thus, the first step was the analysis of the various configuration files with the aim of capturing the class diagram. The diagram on the next page shows the anonymized result of the configuration analysis. Once the meta-model specified, we encoded it in the tool.

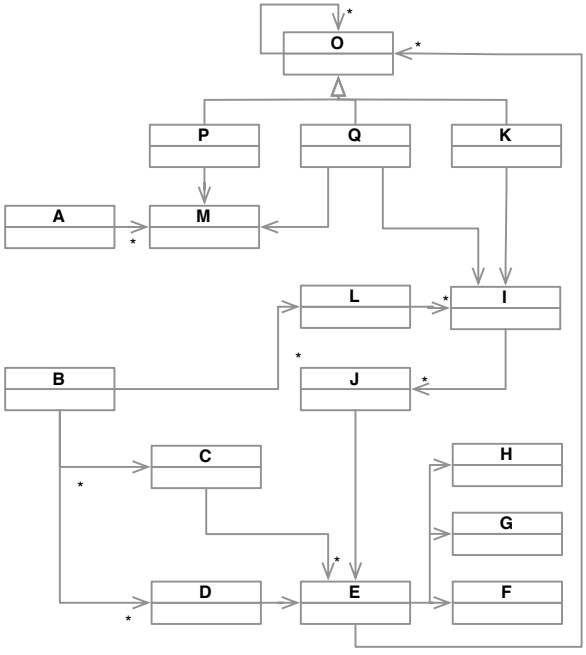
The next step was to create the importer to take all configuration files as input and then to create a coherent model.

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. Thus, our importing solution needed to deal with all these differences.

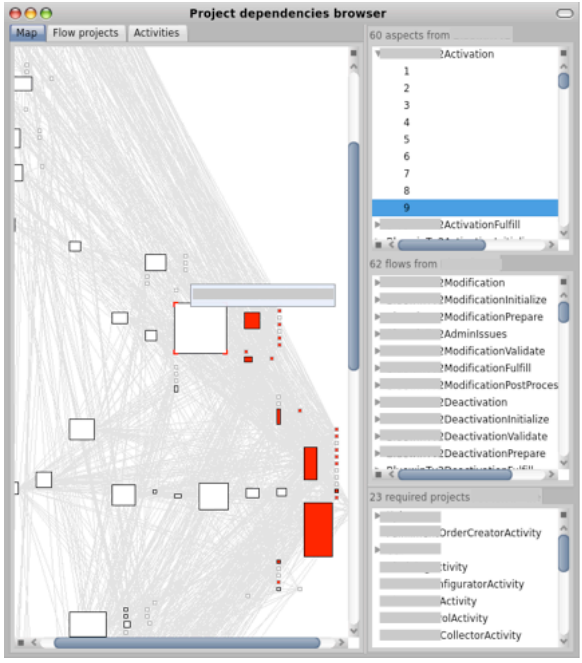
The last step was to create several analyses and visualizations on top of this model. These views were then integrated into an interactive tool. When used by developers, the tool revealed several unexpected dependencies and incomplete specifications in the system.

For putting together this solution, we used the tool building capabilities of Moose.

All in all, the development effort for the prototype totaled a mere 10 person-days. Placed in the overall context of the multiple hundred person-years of development effort spent in the project, it was an insignificant investment with high return on investment.



A schematic view of the complex meta-model



A part of the custom tool for dependency overview

Success story: Analysis of systems 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, a large amount of sources have accumulated. 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.

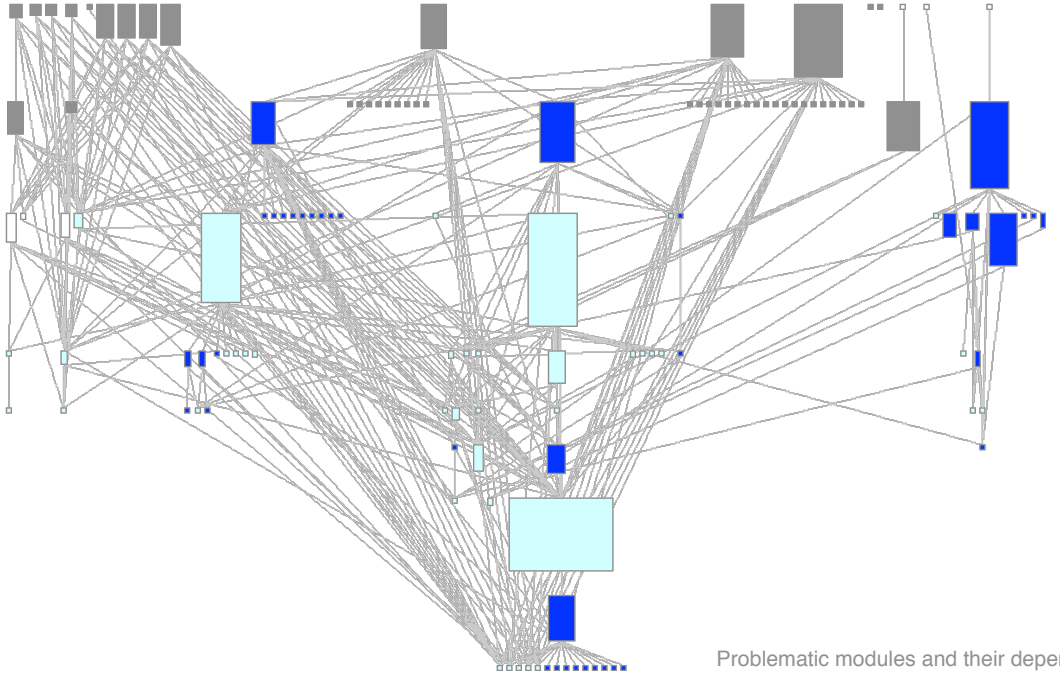
We were mandated to create a dedicated infrastructure for supporting the assessment of programs written in this language and to relate them to the overall forms structure.

One challenge was posed by the lack of documentation of both the language and the format of the exported forms. Thus, we started with a reverse engineering effort. We adopted an iterative approach through which we combined developer interviews for recovering the meaning of instructions, and testing to check the accuracy of the produced parsers and importers.

After two dozen days of effort, we produced a first working version of the importer that could be used to produce a picture of the systems.

Using this importer, we enabled the team to start encoding their own concerns to ease development and maintenance. It soon became apparent that the automatic checking revealed many areas for improvement. Thus, the team adopted the process of continuous assessment and checked these concerns on a regular basis.

Another byproduct of this project consisted in a set of interactive browsers and visualizations meant to ease program comprehension. These were also used to ease the dialogue between the technical team and management concerning the status of the systems.



Problematic modules and their dependencies.

Services	Description	Results	Example scenarios	Duration
Assessment coaching	We coach companies to integrate assessment tools and practices in the development process and in the organization.	We help setting up assessment departments and we coach facilitators and stakeholders to work together on a daily basis.	Teams that want to control the evolution of the architecture, or that want to instill continuous quality assurance.	weeks or months
Strategic assessment	We help companies make strategic decisions by assessing the systems from their portfolios.	We work together with the technical staff, we compile the results as a report, and we facilitate the decision making process.	Checking the conformance to wanted standards and designs; Supporting strategic decisions of rewrite / reengineer.	weeks
Tooling buildup	We offer the service of producing custom solutions for solving specific problems or for reporting on proprietary systems.	We use Moose as the underlying technical platform to deliver both exploratory and complete tools that provide support for decision making.	Analysis tools for proprietary languages, for domain-specific languages, for meta-data, or for proprietary configuration systems.	weeks

Courses	Description	Audience	Duration
Humane assessment primer	This course offers an introduction in the area of software and data assessment, and provides an overview of the strategies to integrate it in the development process and in the organization. The sessions are dialog-oriented and cover issues related to managing assessment both from a technical and a process standpoint.	engineers architects managers	1-2 days
Moose shepherd	This is a technical introductory course to the Moose analysis platform. It teaches the use of querying, code metrics, and visualizations. It also provides an introduction to parsing and to scripting custom concerns, visualizations and interactive browsers. The course sessions are accompanied by hands-on exercises which are typically based on case studies provided by the participants.	engineers architects	5 days
Moose hunter	This advanced course is focused on uncovering the full potential of Moose through exercises of scripting and of extending the base functionality with new and custom detections, models, visualizations and tools. The course can be customized to match the context of the participants.	engineers architects	2-5 days

Tudor Gîrba attained his PhD in 2005, and he now works as a consultant. His main expertise lies in the area of software engineering with focus on software and data assessment.

Since 2003, he leads the work on the Moose analysis platform. He published all sorts of peer reviewed scientific articles, he served in program committees for several dozen international conferences and workshops, and he is regularly invited to give talks and lectures.

He developed the humane assessment method, and he is currently helping companies to assess and to manage large software systems and data sets.

www.tudorgirba.com
tudor@tudorgirba.com