Making Systems Explainable*

1st Oscar Nierstrasz
feenk GmbH, Wabern, Switzerland
ORCID: 0000-0002-9975-9791

2nd Tudor Gîrba
feenk GmbH, Wabern, Switzerland
ORCID: 0000-0002-2987-9624

***Abstract*—What makes software systems explainable?**

As we develop and maintain software, we have questions to ask about the code, but piecing together the answers remains hard. The main interface the classical IDE offers is a text editor for the source code. Code, documentation, and the running system are disconnected.

In this keynote presentation, we will show how software systems can be made explainable with the help of three interacting technologies: (i) *live notebooks* that can be used to create narratives that link documentation, source code, and running applications, (ii) *example methods* that not only perform tests, but produce live examples that can be used within narratives, to explain use cases, scenarios and features, and (iii) a *moldable inspector* that can be easily extended with live *custom views* to answer domain-specific questions about software systems.

With the help of running examples we will show in the keynotes presentation how these technologies work together to provide a radically different kind of development experience.

***Index Terms*—Smalltalk; program comprehension; notebooks; testing; debugging**

What does it mean for a system to be explainable?

For a software system to be *explainable*, it should empower developers to answer questions about it. Questions like:

- *How does this work?*
- *Where is this feature implemented?*
- *Where is this event handled?*
- *What does this code do?*
- *How do I use this API?*
- *Where should I hook in this new feature?*

Questions like these are not just about the code, but also about the running system, and narratives about how it works. Furthermore, the answers often entail multiple levels of abstraction to be addressed simultaneously. As a consequence, an explainable system must not only be *queryable* and *explorable*, but it must be able to *offer narratives* that link stories with source code and live objects. These narratives serve to explain the system not only to developers, but also to other stakeholders.

What prevents current systems from being explainable?

Current IDEs are mostly focused on *editing of source code*. Sure, there are debuggers, and plugins for analysis, but the main paradigm is: stare at your code, edit it, compile, run your tests, run the code, fix bugs, repeat.

Documentation is largely static, and at most provides links to other documentation, or to source code. Tests provide a path to live objects only if they fail, and if one is brave enough to use a debugger to explore the live program state. Debuggers offer only low-level inspector views of the state of objects. Various plugins may exist but such specialized tools tend not to offer ways to tailor their views to the specific questions raised within the domain of a particular application.

How do we make systems explainable?

We claim that we can make systems explainable by shifting the focus in the IDE from editing source code to exploring live objects, source code, and narratives that link them. Such an IDE would allow us to seamlessly navigate between these three aspects.

Glamorous Toolkit,[1] or *GT*, is a *moldable* development environment which is designed so that its tools can be easily "molded" to the needs of a particular domain or application. Although there are many parts to GT, the key components for making systems explainable are (i) a live notebook (Lepiter) that allows developers to construct interactive narratives about code and live objects, (ii) a coding environment (aka "Coder") that links source code to live examples, and (iii) a moldable object inspector (Inspector) that allows developers to cheaply add custom views for exploring and navigating the web of objects of a domain.

These components make systems explainable by enabling querying, navigation and story-telling for developers, and also for other stakeholders.

Really? Explain the pieces to me

A. Lepiter

Lepiter [1] is a live notebook, connecting documentation of models and source code with live, running examples.

Pages in a notebook consist of linked text interspersed with various kinds of snippets, such as embedded code, live visualizations, and code snippets that can be evaluated. As in the Coder and the Inspector, one navigates from view to view with the help of Miller Columns,[2] so from a given page, one can navigate to another Lepiter page, a Coder view of a class or method, or an Inspector view of a live object (Figure 1).

One can therefore create narratives either by (i) telling stories in Lepiter pages, or (ii) following a trail of cascading views.

* Invited keynote presentation at VISSOFT 2022.

[1] https://gtoolkit.com
[2] https://en.wikipedia.org/wiki/Miller_columns

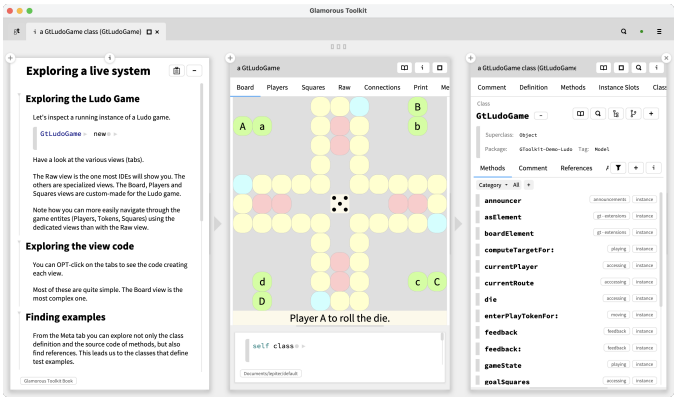

Fig. 1. Navigating from a Lepiter note to a custom view of live instance, and from there to the source code.

B. Examples

Examples are live objects produced by executing *example methods* [2], [3], [9].

Example methods are just like unit tests, except that they return an object instead of simply passing or failing. This means that the result of a test is not just a status report, but a *live object* that can be explored.

An example can be used as input to another example method, thus allowing tests to be chained. (So when a test fails, its dependent tests don't fail, as they are not run [4].)

Examples can be used to create narratives, to explain use cases, scenarios and features. Since examples can be chained, it is possible to create a chained sequence of examples that illustrate a particular scenario. Such examples can also be embedded into a notebook page that documents and explains the steps of the scenario, while linking to the live, explorable instances (Figure 2).

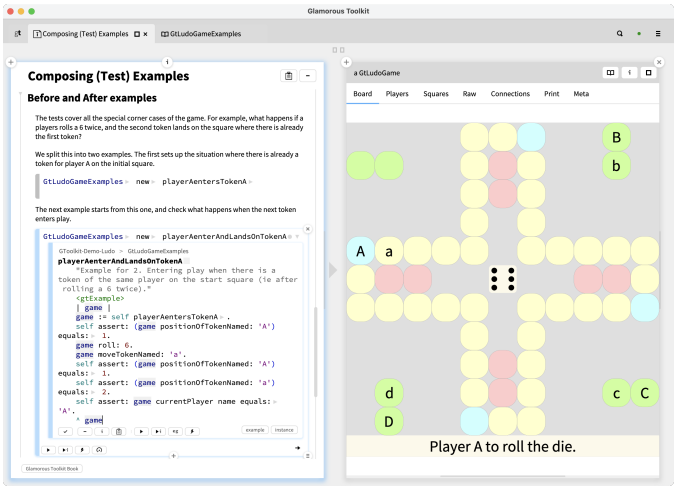

Fig. 2. Explaining a scenario by composing examples that illustrate steps in a computation

C. The Moldable Inspector

The Inspector [5] extends the usual debugging object inspector with interactive playgrounds, Miller columns and custom views. Rather than being an incidental tool that is only used in a debugging context, the Inspector serves as the environment in which the other tools live, as *we are always inspecting some kind of object*. The Miller columns are the primary mechanism for navigating through a chain of connected objects. Each column provides a view of an object, and a playground that allows you to send messages to that object. Each playground is, in turn, a complete Lepiter page that can combine annotated text, live code snippets, and other entities. As a consequence the Inspector offers two ways to navigate. You can either click in the interactive view to open another view in the adjacent column, or you can evaluate a code snippet (or follow a link) to inspect the result in new column.

The Inspector is *moldable* in the sense that it can be cheaply extended with *custom views* that are specific to a given context (be it a domain, an application, or even an individual object) during development. [6], [7]. The key insight is that every software system raises its own questions and issues, so generic views are less likely to be helpful in making such systems explainable. By making it easy and cheap to add custom views, the Inspector is elevated from being an obscure debugging aid, to a central tool for exploring and explaining complex software systems.

Whereas live interactions and playground queries provide a way to navigate through a web of connected objects, custom views do the opposite! Each custom view consists of (i) an encapsulated query, to produce the object(s) of the view, and (ii) a simple, interactive visualization of the result. Thus, the point of custom views is to enable developers to quickly find answers to common questions just by selecting the tab of interest, without having to navigate or code a query (Figure 3).

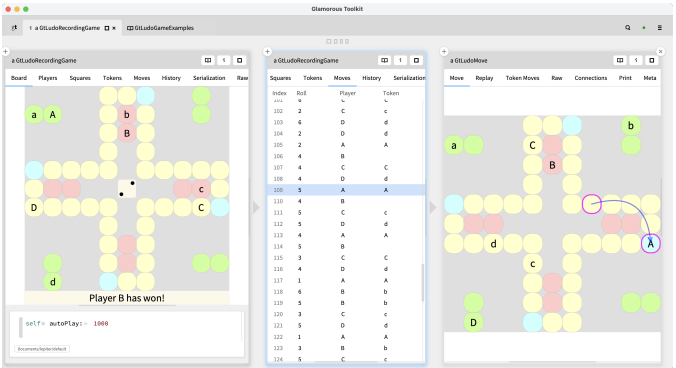

Fig. 3. Custom views of a Ludo game object and its moves help to explain the game logic

WHY IS IT CHEAP TO BUILD CUSTOM VIEWS?

Tools are only effectively moldable if it is cheap to adapt them. In the current release of GT,[3] there are 1974 custom GT

[3]DEV v0.8.1735, 26 July 2022

views, consisting of 22 932 lines of code, or just short of 12 lines per view. Of course there is a lot of heavy lifting being done somewhere else, but in practice, with a bit of experience, these views are easy to create (Figure 4).

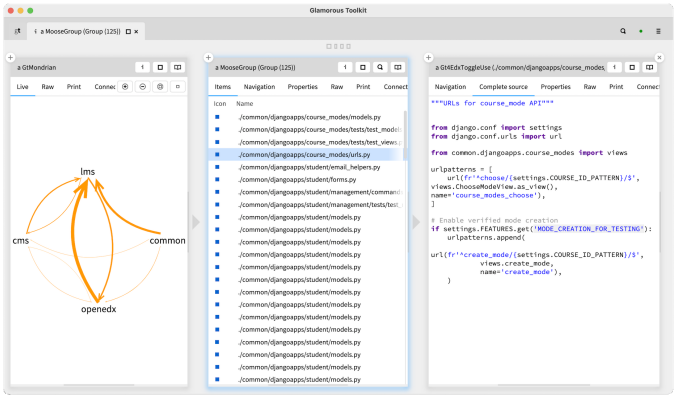

Fig. 4. Custom views for static analysis of a Python project

The key insight, however, is that GT is built with a graphical stack that has just *one rendering tree* [8]. This makes it easy to create flexible visualizations combining arbitrary elements, *i.e.*. text, graphics, and widgets.

How does all this impact software development?

Just like testing should not be an afterthought, *making systems explainable should be part of the development process*.

In short, you should: (i) document narratives about the software you are building as well the process you are following itself, (ii) create meaningful examples as you develop, and (iii) create customs views to expose what's important.

Lepiter influences development in several ways: Live documentation is not an add-on, but an integral part of a software system. Lepiter is the single starting point for most development tasks. For example, we used a Lepiter page even to find out how many custom views there are in GT (Figure 5). Systems become explainable by linking code and examples in live documentation. Creating narratives that consist of live documentation becomes part of the development process. As you develop software, you document your process by creating tagged notes that describe features and use cases, while linking code and examples. Whenever we have a question, we devise a custom narrative to make the problem explicit. It's the narrative that drives the rest. This is the essence of what we call Moldable Development.[4]

Examples also form a key component of both the software and the process: Example-driven development [9] forces you to develop the system in a way that it can be explored, visualized and explained, just like TDD encourages you to develop systems so they become testable! If you need a feature, write an example (test). If you find a bug, write an example. If you want to explain something, write an example.

Finally, *custom views* reach their potential if they are introduced as the need for them arises: If you need to navigate or

[4]https://moldabledevelopment.com

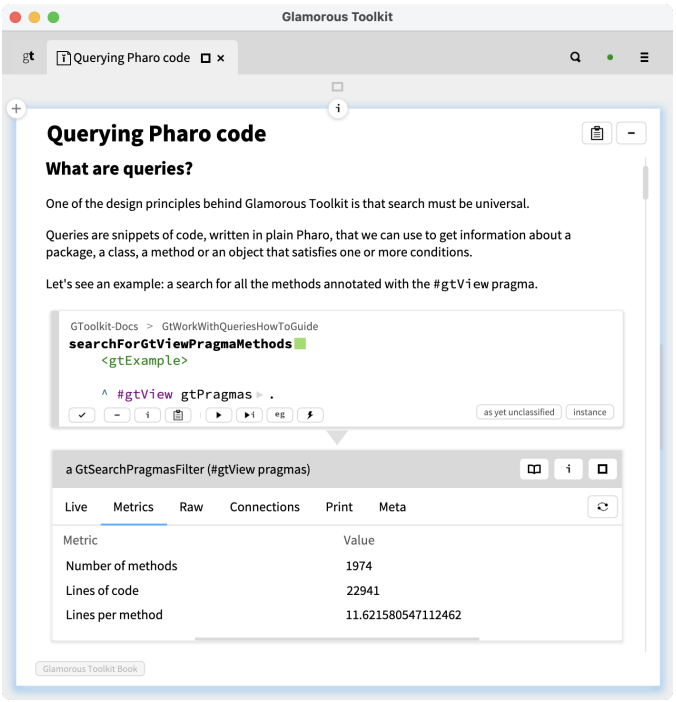

Fig. 5. Querying Pharo code

write a query, create a view. If you need to explain something about a domain concept, create a custom view. Apply the principles of GQM (Goal - Question - Metric) to developing custom views — don't introduce a view just because an easy visualization is available, but because there is a demonstrated need arising from domain-specific questions. These questions may come from the developers, but they may also come from the customer, the user, or other stakeholders. Create the views as you develop to enhance both debugging and documentation. Custom views answer questions about domain objects in an application. These answers become words in narratives in live notebooks, to make systems explainable.

Is that all there is?

Well, no. There are many other important components that help to make systems explainable with GT. For example, *Spotter* [10] is context-aware tool that helps you search for anything in the IDE. GT is also a language workbench powered by the mature *SmaCC* compiler-compiler framework that enables the reverse engineering, parsing and editing of foreign languages [11]. All these are integrated in a uniform environment made out of visual and interactive operators that can be combined to create many unexpected development experiences.

GT is an enabling technology for Moldable Development. Once we can explain the inside of systems, we get to unlock new opportunities at the level of the development process and even for how we create new business value.

SO, HOW WOULD YOU SUM THAT UP?

Systems become explainable by making it easy to create narratives that link documentation, code, and live objects. This means that domain concepts become explicit, so it is easy to navigate between concepts and their instances, and it is easy to have dedicated views that expose the underlying concepts.

Instead of seeing a software system as a static collection of source code files, it should be responsive and live, integrating narratives about the code and the running application. Alternatively, if the application is the analysis of a foreign codebase, the resulting analysis model should be equally live and responsive.

The result is that an explainable system is one which supports a *dialogue* between the developer and the system. The narratives and the views arise from the questions developers ask of the system.

Acknowledgments

We thank Pooja Rani for her feedback on a draft of this paper.

REFERENCES

[1] T. Gîrba, "Introducing Lepiter: Knowledge management + multi-language notebooks + moldable development," 2021. [Online]. Available: https://lepiter.io/feenk/introducing-lepiter--knowledge-management--e2p6apqsz5npq7m4xte0kkywn/

[2] M. Gaelli, "Test composition with example objects and example methods." in *Proceedings of the ECOOP '03 Workshop on Object-oriented Language Engineering for the Post-Java Era*, ser. LNCS, vol. 3013, Jul. 2003, pp. 143–153, abstract only — full version availabe as technical report IAM-03-009. [Online]. Available: http://scg.unibe.ch/archive/papers/Gael03aTestComposition.pdf

[3] M. Gaelli, R. Wampfler, and O. Nierstrasz, "Composing tests from examples," in *Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007*, vol. 6/9, Oct. 2007, pp. 71–86. [Online]. Available: http://www.jot.fm/issues/issue_2007_10/paper4.pdf

[4] A. Kuhn, B. V. Rompaey, L. Hänsenberger, O. Nierstrasz, S. Demeyer, M. Gaelli, and K. V. Leemput, "JExample: Exploiting dependencies between tests to improve defect localization," in *Extreme Programming and Agile Processes in Software Engineering, 9th International Conference, XP 2008*, ser. Lecture Notes in Computer Science, P. Abrahamsson, Ed. Springer, 2008, pp. 73–82. [Online]. Available: http://scg.unibe.ch/archive/papers/Kuhn08aJExample.pdf

[5] A. Chiş, T. Gîrba, O. Nierstrasz, and A. Syrel, "The Moldable Inspector," in *Proceedings of the 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2015. New York, NY, USA: ACM, 2015, pp. 44–60. [Online]. Available: http://scg.unibe.ch/archive/papers/Chis15a-MoldableInspector.pdf

[6] A. Chiş, T. Gîrba, J. Kubelka, O. Nierstrasz, S. Reichhart, and A. Syrel, "Moldable tools for object-oriented development," in *PAUSE: Present And Ulterior Software Engineering*, B. M. Manuel Mazzara, Ed. Springer, Cham, 2017, pp. 77–101. [Online]. Available: http://scg.unibe.ch/archive/papers/Chis17a-MoldableToolsPAUSE.pdf

[7] ——, "Exemplifying moldable development," in *Proceedings of the Programming Experience 2016 (PX/16) Workshop*, ser. PX/16. New York, NY, USA: ACM, 2016, pp. 33–42. [Online]. Available: http://scg.unibe.ch/archive/papers/Chis16b-ExemplifyingMoldableDevelopment.pdf

[8] T. Gîrba, "One rendering tree," 2020. [Online]. Available: https://medium.com/feenk/one-rendering-tree-918eae49bcff

[9] ——, "An example of example-driven development," 2019. [Online]. Available: https://medium.com/feenk/an-example-of-example-driven-development-4dea0d995920

[10] A. Chiş, T. Gîrba, J. Kubelka, O. Nierstrasz, S. Reichhart, and A. Syrel, "Moldable, context-aware searching with Spotter," in *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2016. New York, NY, USA: ACM, 2016, pp. 128–144. [Online]. Available: http://scg.unibe.ch/archive/papers/Chis16a-MoldableContextAwareSearchingWithSpotter.pdf

[11] J. Brant, J. Lecerf, T. Goubier, S. Ducasse, and A. Black, "Smacc: a compiler-compiler," 2017. [Online]. Available: https://books.pharo.org/booklet-Smacc/html/smacc.html