Domain discovery

February 21, 2021

feenk

We cover the whole discovery lifecycle.

From working with domain experts to recovering knowledge from existing systems and data and to creating executable specifications.

Visual prototypes

We capture domain expert input into executable prototype. Then we make the prototype show visual domain representations. Projecting multiple views facilitates a multi faceted discovery.

Reverse engineering

When new systems have to accommodate existing data sources, like APIs, databases or file formats, or when legacy systems already exists in the domain, we reverse engineer these and integrate the understanding in the domain discovery.

Executable specifications

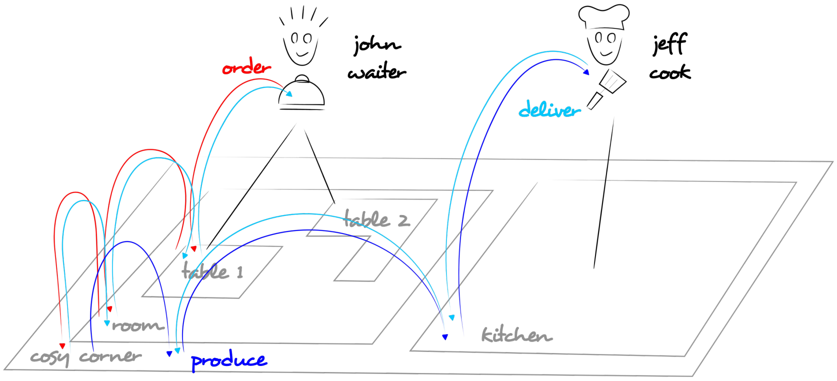
The domain discovery is driven by a prototype system that is visualized in many ways to capture the various facets of the domain.

Once the system exists, we assemble the views into larger narratives that document the domain. These essentially act as executable specifications.

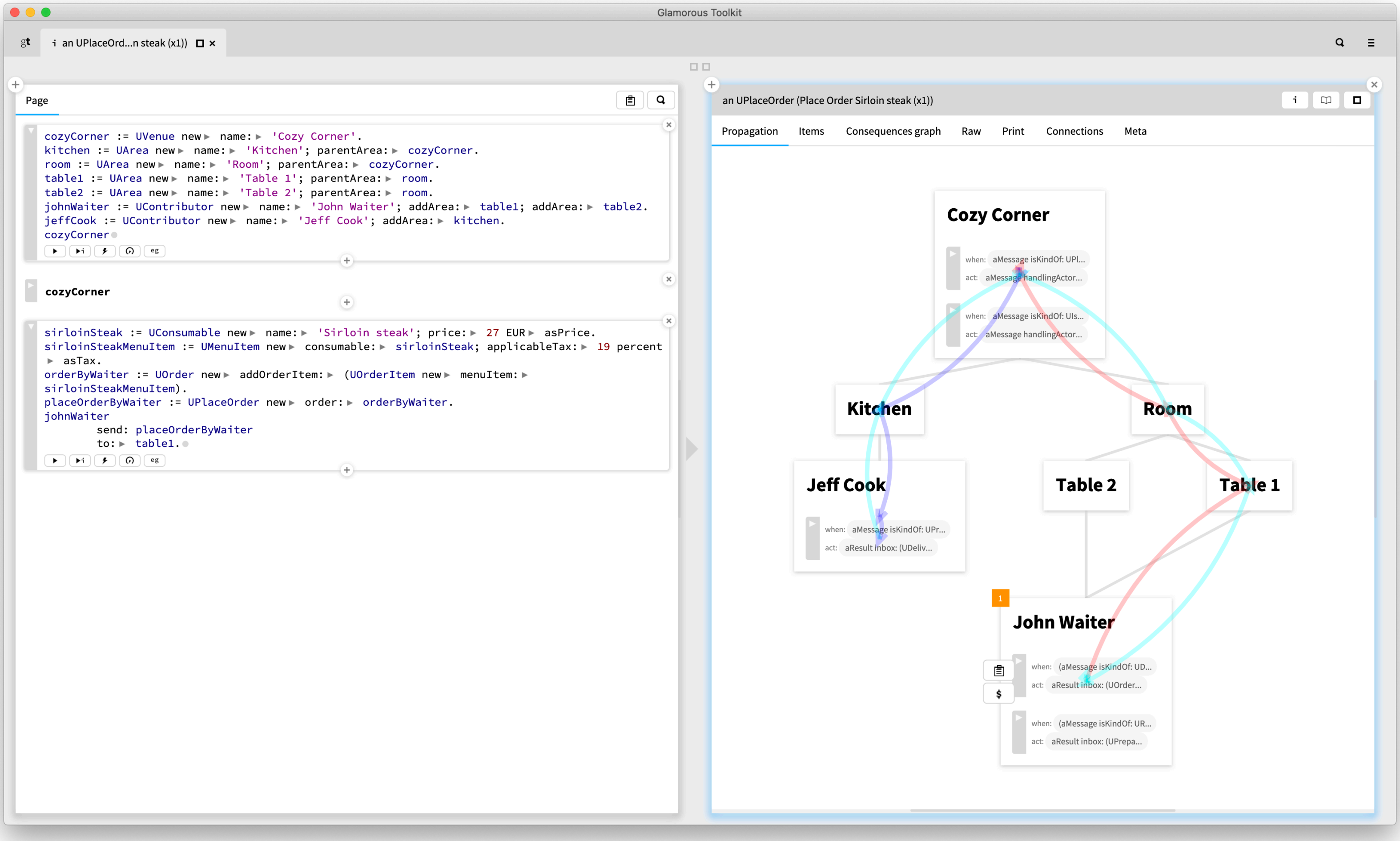
Visual prototypes

The domain discovery often starts from drawings on the whiteboard. These drawings offer a common language between domain experts and technical people.

Like this one right here, depicting a flow in a restaurant.



As soon as some an idea exists, we capture it in an executable prototype. And then we make the prototype show the same picture. For example, here we see code on the left and an interactive domain depiction on the right. Projecting multiple views facilitates a multi faceted discovery.



```
cozyCorner := UVenue new▶ name:▶ 'Cozy Corner'.
kitchen := UArea new▶ name:▶ 'Kitchen'; parentArea:▶ cozyCorner.
room := UArea new▶ name:▶ 'Room'; parentArea:▶ cozyCorner.
table1 := UArea new▶ name:▶ 'Table 1'; parentArea:▶ room.
table2 := UArea new▶ name:▶ 'Table 2'; parentArea:▶ room.
johnWaiter := UContributor new▶ name:▶ 'John Waiter'; addArea:▶ table1; addArea:▶ table2.
jeffCook := UContributor new▶ name:▶ 'Jeff Cook'; addArea:▶ kitchen.
cozyCorner
```

```
cozyCorner
```

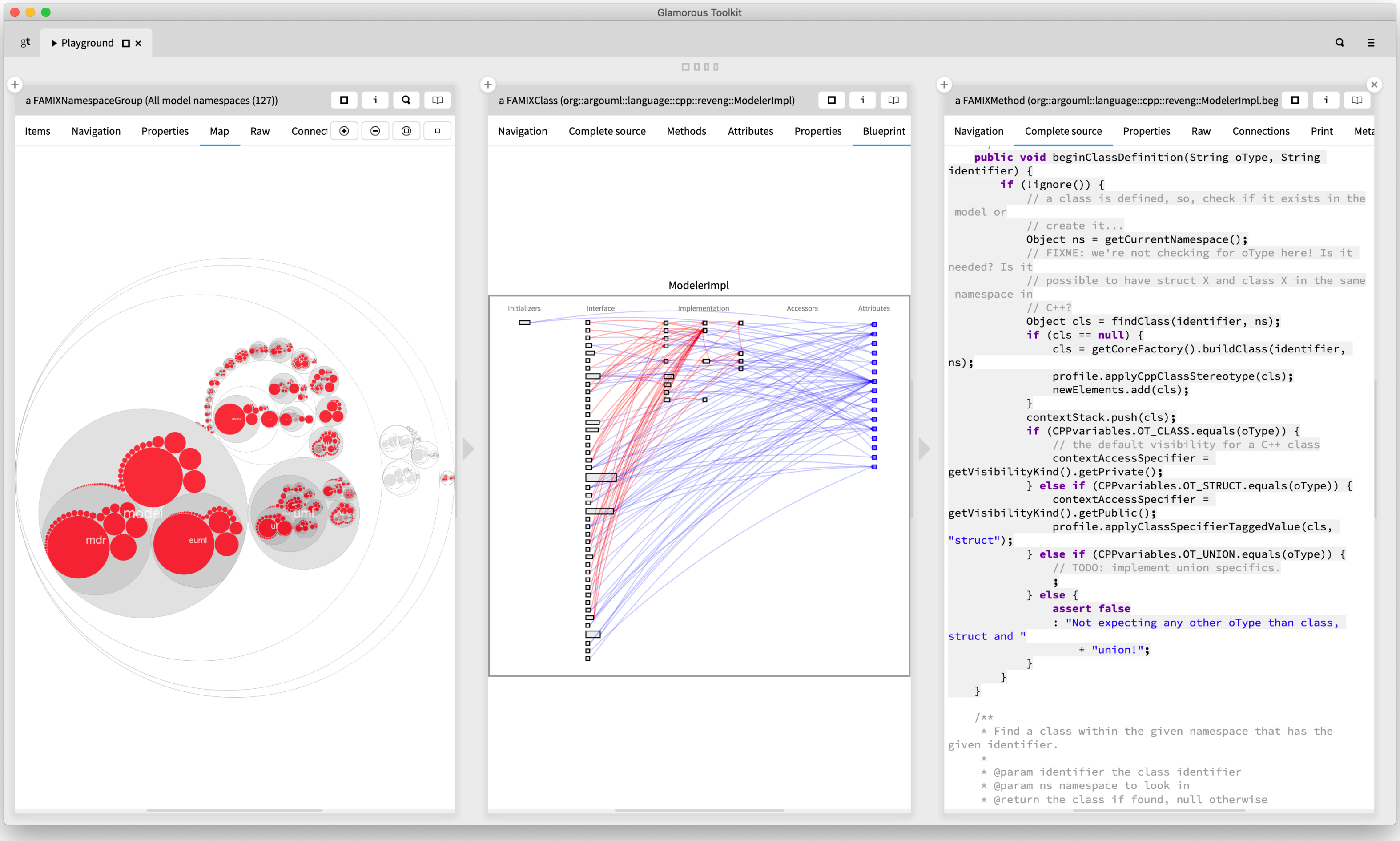
```
sirloinSteak := UConsumable new▶ name:▶ 'Sirloin steak'; price:▶ 27 EUR▶ asPrice.
sirloinSteakMenuItem := UMenuItem new▶ consumable:▶ sirloinSteak; applicableTax:▶ 19 percent
▶ asTax.
orderByWaiter := UOrder new▶ addOrderItem:▶ (UOrderItem new▶ menuItem:▶
sirloinSteakMenuItem).
placeOrderByWaiter := UPlaceOrder new▶ order:▶ orderByWaiter.
johnWaiter
        send: placeOrderByWaiter
        to:▶ table1.
```

```
when: aMessage isKindOf: UPl...
act: aMessage handlingActor...
when: aMessage isKindOf: UIs...
act: aMessage handlingActor...
when: aMessage isKindOf: UPr...
act: aResult inbox: (UDeliv...
when: (aMessage isKindOf: UD...
act: aResult inbox: (UOrder...
when: (aMessage isKindOf: UR...
act: aResult inbox: (UPrepa...
```

Reverse engineering of legacy systems and data sources

Often new systems have to accommodate existing data sources, like APIs, databases or file formats. Other times, legacy systems already exists in the domain.

We reverse engineer these and integrate the understanding in the domain discovery.

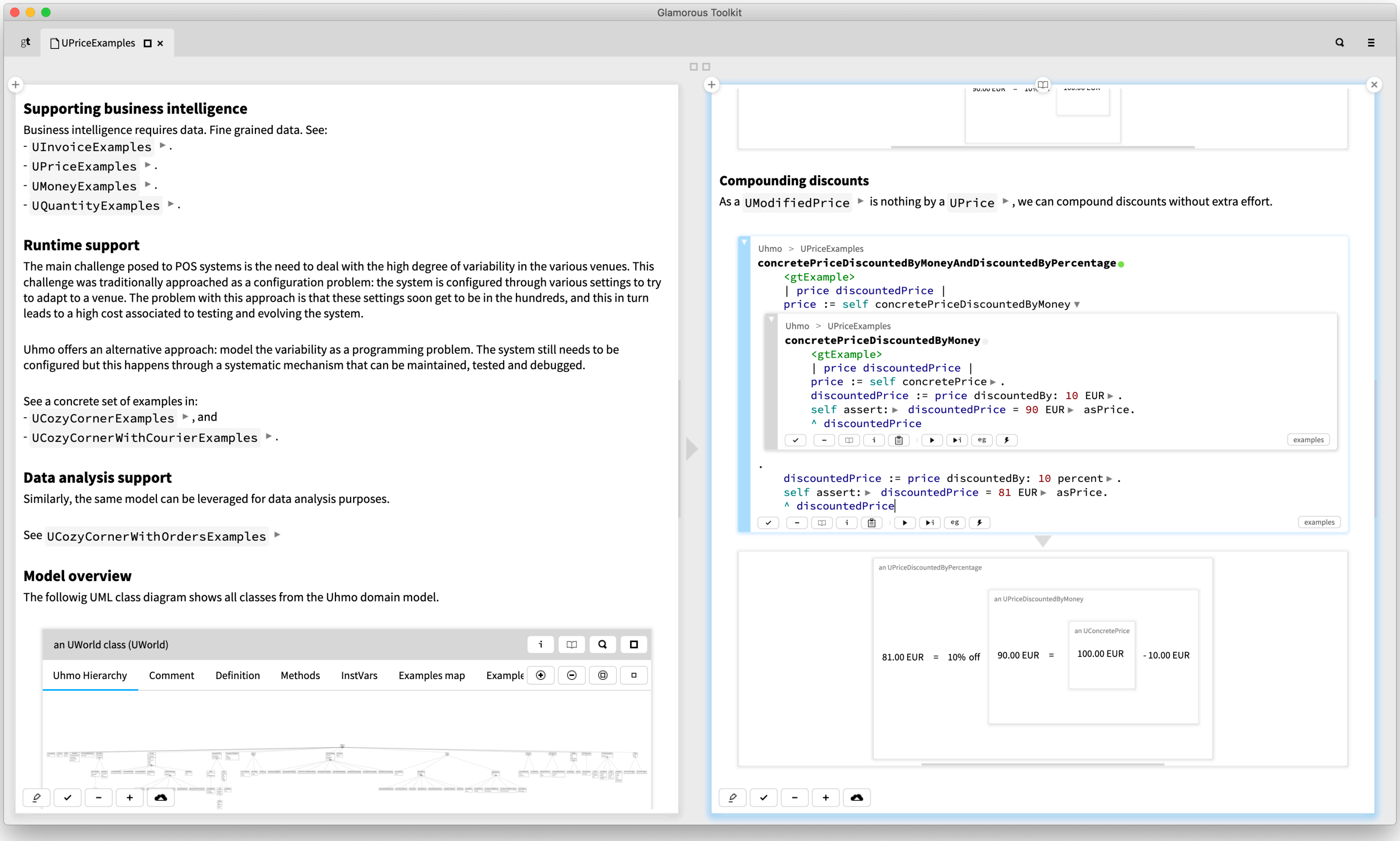


```
    public void beginClassDefinition(String oType, String
identifier) {
        if (!ignore()) {
            // a class is defined, so, check if it exists in the
 model or
            // create it...
            Object ns = getCurrentNamespace();
            // FIXME: we're not checking for oType here! Is it
needed? Is it
            // possible to have struct X and class X in the same
 namespace in
            // C++?
            Object cls = findClass(identifier, ns);
            if (cls == null) {
                cls = getCoreFactory().buildClass(identifier,
ns);
                profile.applyCppClassStereotype(cls);
                newElements.add(cls);
            }
            contextStack.push(cls);
            if (CPPvariables.OT_CLASS.equals(oType)) {
                // the default visibility for a C++ class
                contextAccessSpecifier =
getVisibilityKind().getPrivate();
            } else if (CPPvariables.OT_STRUCT.equals(oType)) {
                contextAccessSpecifier =
getVisibilityKind().getPublic();
                profile.applyClassSpecifierTaggedValue(cls,
"struct");
            } else if (CPPvariables.OT_UNION.equals(oType)) {
                // TODO: implement union specifics.
                ;
            } else {
                assert false
                : "Not expecting any other oType than class,
struct and "
                    + "union!";
            }
        }
    }

    /**
     * Find a class within the given namespace that has the
given identifier.
     *
     * @param identifier the class identifier
     * @param ns namespace to look in
     * @return the class if found, null otherwise
```

Executable specifications

The domain discovery is driven by a prototype system that is visualized in many ways to capture the various facets of the domain.

Once the system exists, we assemble the views into larger narratives that document the domain. These essentially act as executable specifications.



Supporting business intelligence

Business intelligence requires data. Fine grained data. See:
- `UInvoiceExamples` ▸.
- `UPriceExamples` ▸.
- `UMoneyExamples` ▸.
- `UQuantityExamples` ▸.

Runtime support

The main challenge posed to POS systems is the need to deal with the high degree of variability in the various venues. This challenge was traditionally approached as a configuration problem: the system is configured through various settings to try to adapt to a venue. The problem with this approach is that these settings soon get to be in the hundreds, and this in turn leads to a high cost associated to testing and evolving the system.

Uhmo offers an alternative approach: model the variability as a programming problem. The system still needs to be configured but this happens through a systematic mechanism that can be maintained, tested and debugged.

See a concrete set of examples in:
- `UCozyCornerExamples` ▸, and
- `UCozyCornerWithCourierExamples` ▸.

Data analysis support

Similarly, the same model can be leveraged for data analysis purposes.

See `UCozyCornerWithOrdersExamples` ▸

Model overview

The followig UML class diagram shows all classes from the Uhmo domain model.

Compounding discounts

As a `UModifiedPrice` ▸ is nothing by a `UPrice` ▸, we can compound discounts without extra effort.

```
Uhmo > UPriceExamples
concretePriceDiscountedByMoneyAndDiscountedByPercentage
    <gtExample>
    | price discountedPrice |
    price := self concretePriceDiscountedByMoney ▾
        Uhmo > UPriceExamples
        concretePriceDiscountedByMoney
            <gtExample>
            | price discountedPrice |
            price := self concretePrice ▸ .
            discountedPrice := price discountedBy: 10 EUR ▸ .
            self assert: ▸ discountedPrice = 90 EUR ▸ asPrice.
            ^ discountedPrice
.
    discountedPrice := price discountedBy: 10 percent ▸ .
    self assert: ▸ discountedPrice = 81 EUR ▸ asPrice.
    ^ discountedPrice
```

feenk

We make your systems explainable.

We are consultants.
We are researchers.
We are authors.

We bring a unique experience. We cover the whole spectrum, from a single line of code to decisions made at the company executive level.

Our work is based on state-of-the-art scientific work, much of which we personally authored. We actively create new tools and techniques for thinking with and about software systems.

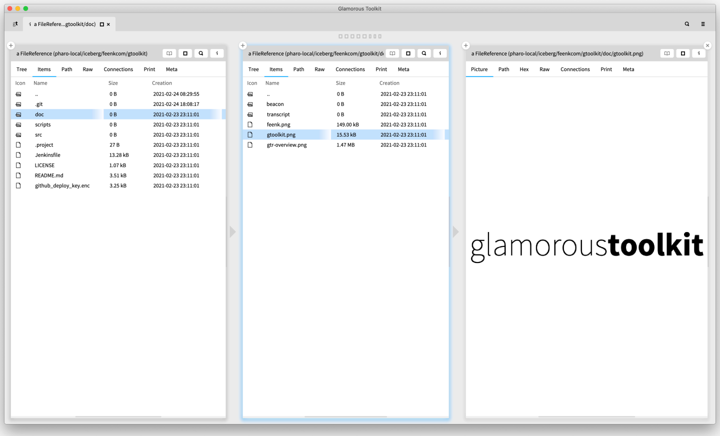
Our work has been validated for more than a decade of working with highly difficult problems in legacy systems.

feenk

Glamorous Toolkit is the moldable environment.

Glamorous Toolkit is our highly integrated and moldable environment. It is a software analysis platform. A live notebook. A knowledge management platform. A rich visualization engine. A powerful query tool. A fancy editor.

But, most importantly, it can be molded in many ways to fit the context of the system at hand. This ability is crucial. Through it, decision making becomes both highly effective and a beautiful experience.



a FileReference (pharo-local/iceberg/feenkcom/gtoolkit)

Tree | Items | Path | Raw | Connections | Print | Meta

Icon	Name	Size	Creation
	..	0 B	2021-02-24 08:29:55
	.git	0 B	2021-02-24 18:08:17
	doc	0 B	2021-02-23 23:11:01
	scripts	0 B	2021-02-23 23:11:01
	src	0 B	2021-02-23 23:11:01
	.project	27 B	2021-02-23 23:11:01
	Jenkinsfile	13.28 kB	2021-02-23 23:11:01
	LICENSE	1.07 kB	2021-02-23 23:11:01
	README.md	3.51 kB	2021-02-23 23:11:01
	github_deploy_key.enc	3.25 kB	2021-02-23 23:11:01

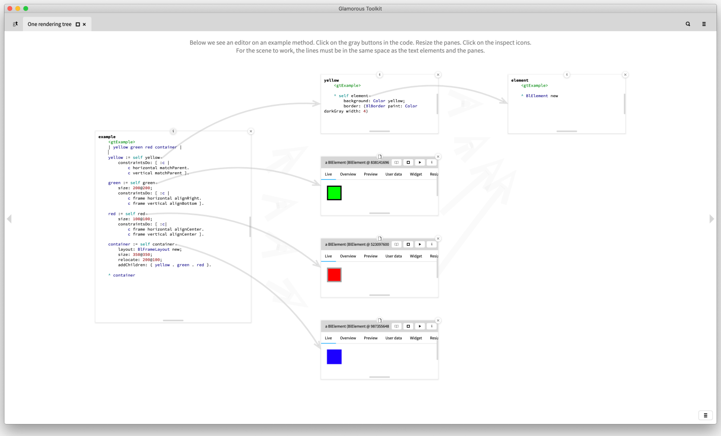
a FileReference (pharo-local/iceberg/feenkcom/gtoolkit/d

Tree | Items | Path | Raw | Connections | Print | Meta

Icon	Name	Size	Creation
	..	0 B	2021-02-23 23:11:01
	beacon	0 B	2021-02-23 23:11:01
	transcript	0 B	2021-02-23 23:11:01
	feenk.png	149.00 kB	2021-02-23 23:11:01
	gtoolkit.png	15.53 kB	2021-02-23 23:11:01
	gtr-overview.png	1.47 MB	2021-02-23 23:11:01

a FileReference (pharo-local/iceberg/feenkcom/gtoolkit/doc/gtoolkit.png)

Picture | Path | Hex | Raw | Connections | Print | Meta

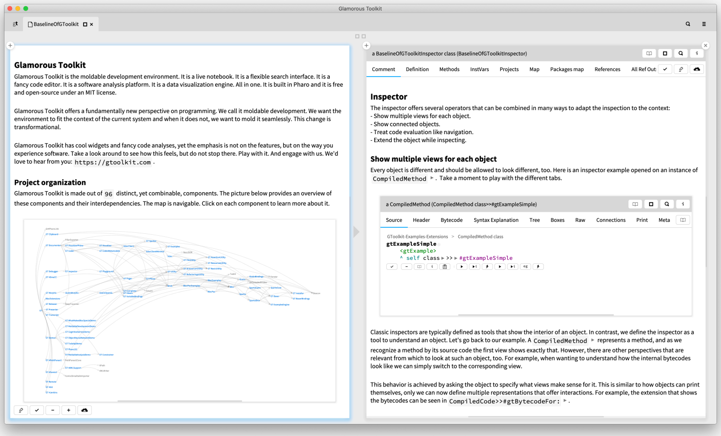

Glamorous Toolkit
gt
One rendering tree
a BlElement (BlElement @ 838141696)
a BlElement (BlElement @ 523097600)
a BlElement (BlElement @ 987355648)
Live
Overview
Preview
User data
Widget
Resiz

Below we see an editor on an example method. Click on the gray buttons in the code. Resize the panes. Click on the inspect icons. For the scene to work, the lines must be in the same space as the text elements and the panes.

```
example
    <gtExample>
    | yellow green red container |

    yellow := self yellow.
        constraintsDo: [ :c |
            c horizontal matchParent.
            c vertical matchParent ].

    green := self green.
        size: 200@200;
        constraintsDo: [ :c |
            c frame horizontal alignRight.
            c frame vertical alignBottom ].

    red := self red.
        size: 100@100;
        constraintsDo: [ :c|
            c frame horizontal alignCenter.
            c frame vertical alignCenter ].

    container := self container.
        layout: BlFrameLayout new;
        size: 350@350;
        relocate: 200@100;
        addChildren: { yellow . green . red }.

    ^ container
```

```
yellow
    <gtExample>

    ^ self element.
        background: Color yellow;
        border: (BlBorder paint: Color
darkGray width: 4)
```

```
element
    <gtExample>

    ^ BlElement new
```



Glamorous Toolkit

Glamorous Toolkit is the moldable development environment. It is a live notebook. It is a flexible search interface. It is a fancy code editor. It is a software analysis platform. It is a data visualization engine. All in one. It is built in Pharo and it is free and open-source under an MIT license.

Glamorous Toolkit offers a fundamentally new perspective on programming. We call it moldable development. We want the environment to fit the context of the current system and when it does not, we want to mold it seamlessly. This change is transformational.

Glamorous Toolkit has cool widgets and fancy code analyses, yet the emphasis is not on the features, but on the way you experience software. Take a look around to see how this feels, but do not stop there. Play with it. And engage with us. We'd love to hear from you: `https://gtoolkit.com` .

Project organization

Glamorous Toolkit is made out of `96` distinct, yet combinable, components. The picture below provides an overview of these components and their interdependencies. The map is navigable. Click on each component to learn more about it.

a BaselineOfGToolkitInspector class (BaselineOfGToolkitInspector)

Comment | Definition | Methods | InstVars | Projects | Map | Packages map | References | All Ref Out

Inspector

The inspector offers several operators that can be combined in many ways to adapt the inspection to the context:
- Show multiple views for each object.
- Show connected objects.
- Treat code evaluation like navigation.
- Extend the object while inspecting.

Show multiple views for each object

Every object is different and should be allowed to look different, too. Here is an inspector example opened on an instance of `CompiledMethod` ▸ . Take a moment to play with the different tabs.

a CompiledMethod (CompiledMethod class>>#gtExampleSimple)

Source | Header | Bytecode | Syntax Explanation | Tree | Boxes | Raw | Connections | Print | Meta

GToolkit-Examples-Extensions > CompiledMethod class

```
gtExampleSimple
    <gtExample>
    ^ self class ▸ >> ▸ #gtExampleSimple
```

Classic inspectors are typically defined as tools that show the interior of an object. In contrast, we define the inspector as a tool to understand an object. Let's go back to our example. A `CompiledMethod` ▸ represents a method, and as we recognize a method by its source code the first view shows exactly that. However, there are other perspectives that are relevant from which to look at such an object, too. For example, when wanting to understand how the internal bytecodes look like we can simply switch to the corresponding view.

This behavior is achieved by asking the object to specify what views make sense for it. This is similar to how objects can print themselves, only we can now define multiple representations that offer interactions. For example, the extension that shows the bytecodes can be seen in `CompiledCode>>#gtBytecodeFor:` ▸ .

gtoolkit.com