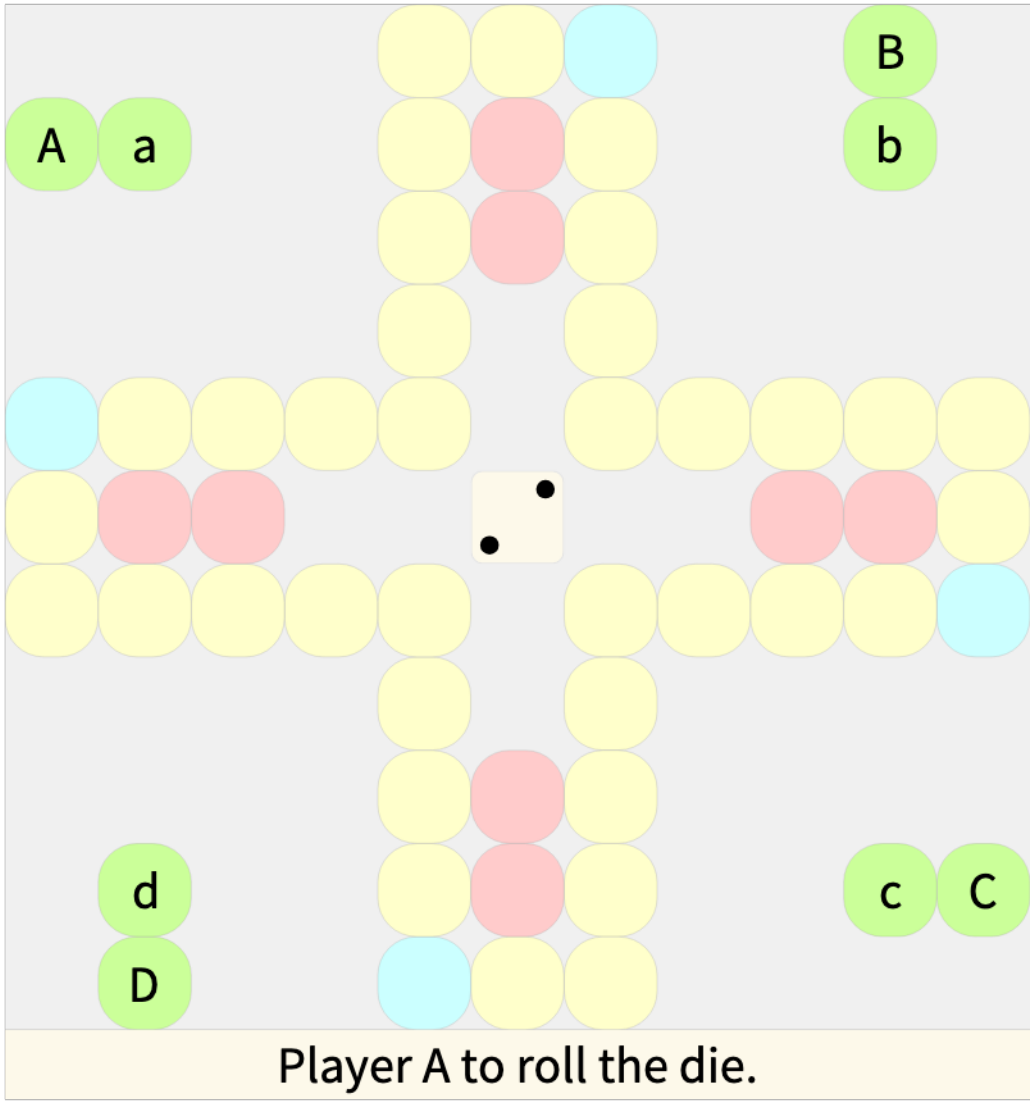
A bit of Smalltalk

oscar.nierstrasz@feenk.com

Appetizer

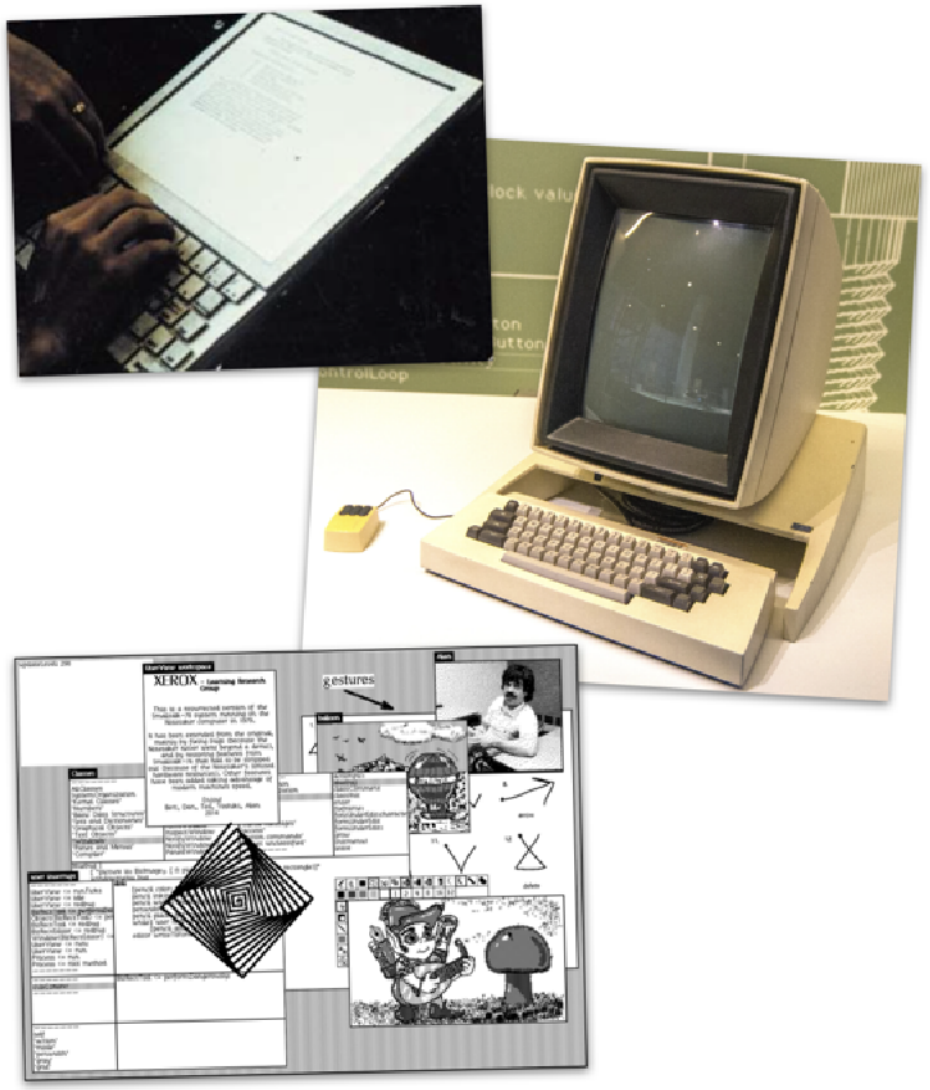
Ludo as a running example of a live system



Outline

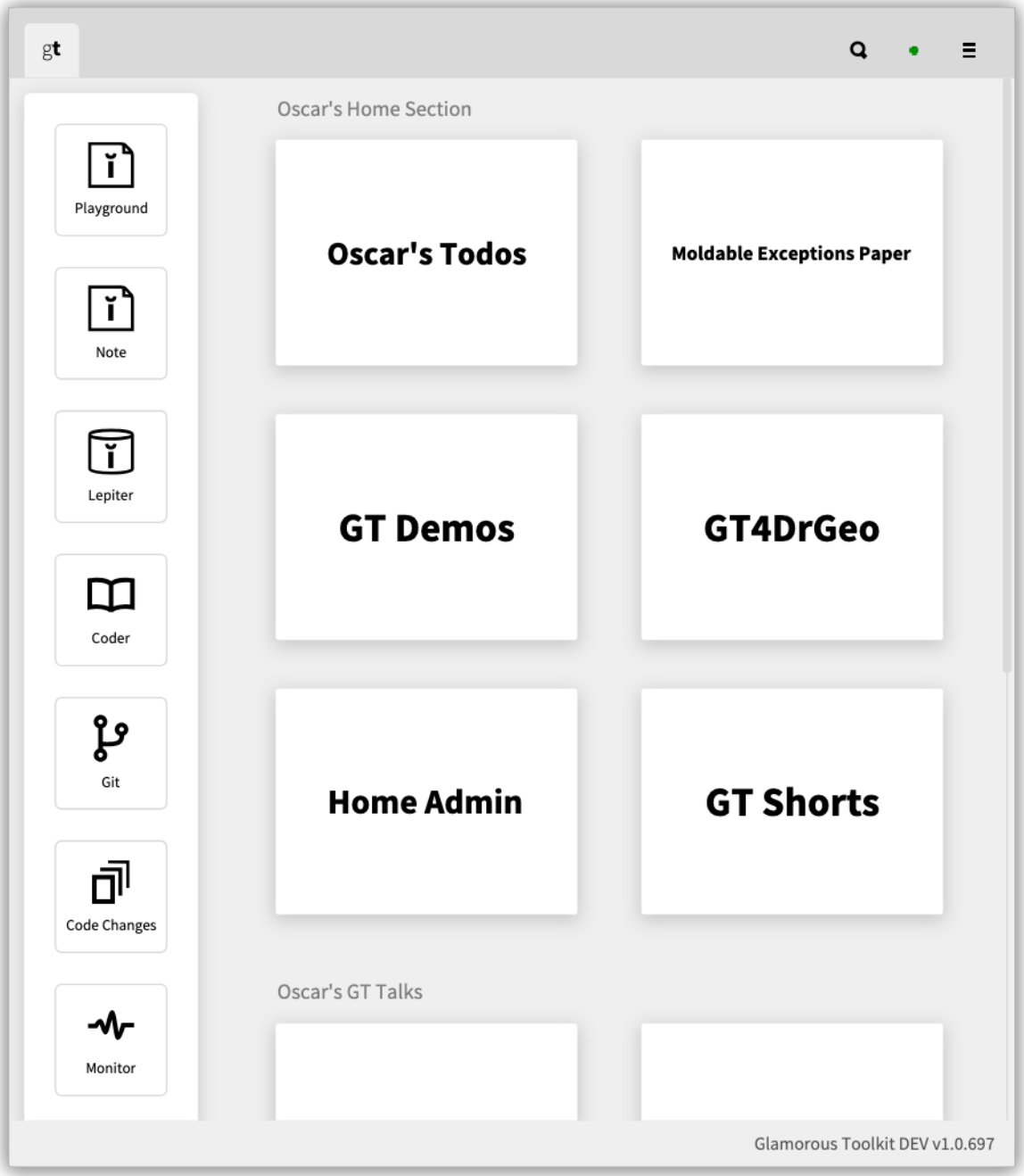
- What is Smalltalk?
- Smalltalk basics — syntax, sending messages
- Live programming with interactive views
- Testing with composable examples
- Live documentation with Lepiter notebooks
- Take home messages

Smalltalk was the first purely object-oriented language and environment, with the first interactive, graphical IDE.



Glamorous Toolkit is a *moldable* development environment with native windows, software analysis tools, and a visualization engine.

Gt is built on top of Pharo, a modern, open-source Smalltalk.



Smalltalk is a *live* programming system.

Objects live inside an image running on a VM.

Changes to classes and methods are logged as you program.

Two rules

1. Everything is an object
2. Everything happens by sending messages

Numbers, strings, booleans and blocks are objects

Primitive data types are full-fledged objects.

```
1
```

```
'hello'
```

```
true
```

```
[ 3 + 4 ]
```

1 explicit reference

nil

Classes are objects

We can send messages to classes:

```
SmallInteger maxVal
```

1 explicit reference

nil

Methods are objects

We can inspect methods and send them messages.

```
4 even
```

```
SmallInteger >> #even
```

```
(SmallInteger >> #even) sourceCode
```

1 explicit reference

nil

Everything is an object

This page is an object

```
thisSnippet page
```

The Ludo game is an object

```
GtLudoGame new
```

Tools are objects

```
GtCoder forClass: SmallInteger
```

This slideshow is an object

```
GtPresenterSlideShow create:
SmalltalkIntroSlideshow new
```

The GT World is an object

```
GtWorldElement new
```

1 explicit reference

nil

Everything happens by sending messages

A *message* is a request for a service.

A *method* is code to fulfil that request.

```
3 + 4
```

1 explicit reference

nil

All computations are message sends

Arithmetic operations, comparisons, and evaluating blocks are just message sends.

```
3 /▸ 4
```

```
(3 =▸ 4) not
```

```
[ 3 +▸ 4 ] value▸
```

▸ 1 explicit reference

nil

Object creation is a message send

You create an object by sending a message to a class.

```
OrderedCollection new
```

```
Fraction numerator: 3 denominator: 4
```

```
Set with: 3 with: 4
```

1 explicit reference

nil

Changes to the system are message sends

You create a class by sending a message to its superclass:

```
Object  subclass: #MyClass
```

You can ask a class to compile a method:

```
MyClass  compile: 'foo ^ 42'
```

```
MyClass  new foo
```

We can remove the class by asking the Smalltalk object to do it:

```
Smalltalk removeClassNamed:  'MyClass'
```

1 explicit reference

nil

You can change (almost) anything!

```
Smalltalk removeClassNamed: 'Object'.
```

1 explicit reference

nil

Three kinds of messages

Unary messages

```
5 factorial
```

```
Float pi
```

```
GtLudoGame new
```

Binary messages

```
3 + 4
```

```
'hello', ' ', 'there'
```

```
1 << 10
```

Keyword messages

```
3 raisedTo: 10 modulo: 5
```

```
GtLudoGame new roll: 5
```

1 explicit reference

nil

Precedence

First unary, then binary, then keyword

```
2 raisedTo: 1 + 3 factorial
```

is the same as:

```
2 raisedTo: ( 1 + (3 factorial ))
```

Warning!

```
1 + 2 * 3
```

is the same as:

```
(1 + 2) * 3
```

Use parentheses to get the order you want.

```
1 + (2 * 3)
```

1 explicit reference

nil

Syntax — Numbers

Radix numbers (binary):

```
2r101010
```

Hex:

```
16r2A
```

1 explicit reference

nil

Syntax — Strings and Symbols

Characters:

```
$a
```

Strings:

```
'hello world'
```

```
'hello world' first = $h
```

Comments:

```
"Ceci n'est pas un string"
```

```
GtLudoGame comment
```

Symbols:

```
#ThereIsOnlyOneOfMe
```

True or false

```
#ThereIsOnlyOneOfMe == 'ThereIsOnlyOneOfMe'
```

```
#ThereIsOnlyOneOfMe = 'ThereIsOnlyOneOfMe'
```

1 explicit reference

nil

Syntax — Constants

`true`, `false` and `nil` are built-in constants.

```
(nil =>  '') == false | true
```

1 explicit reference

nil

Syntax — Arrays

Literal arrays

```
#( 1 $a #foo #( I am 'nested' ) )
```

Dynamic arrays

```
{ 6 * 7 . #( 'hello' 'there' )}
```

1 explicit reference

nil

Syntax — Blocks

Blocks can take any number of arguments.

```
[ 3 + 4 ] value
```

```
[ :x | x + 1 ] value: 10
```

```
[ :x :y | x ** y ] value: 3 value: 2
```

Blocks are first class values.

```
[ :aBlock :anArg | aBlock value: anArg ]
    value: [ :aNumber | aNumber + 1 ]
    value: 1
```

1 explicit reference

nil

Syntax — Pseudo variables

`self` and `super` both refer to the object itself.

```
self == super
```

1 explicit reference

nil

Method Syntax

Pharo constructs:

Messages	Unary
	Binary
	Keyword
Variables	Temporary
	Argument
	Global
	Self
	Super
	ThisContext
Arrays	Array
	LiteralArray
Literals	Integer
	Float
	Character
	Symbol
	String
Others	Pragma
	Return
	Block
	Assignment
	Cascade
	Comment

Method:

GToolkit-Demo-Ludo > GtLudoGame

```
computeTargetFor: aToken

    "Given a token to move, determine which square it should move to.
    There are 3 cases for the target square."

    | route targetIndex |
    route := self currentRoute .

    "(1) a token enters play on the first square of the route"
    (self die topFace = 6 and: [ aToken isInStartState ]) ifTrue: [
        aToken enterPlay.
        ^ route first ].

    self
        assert: aToken isInPlay
        description: 'Token ' , aToken name , ' is not in play'.

    "(2) a token in play moves forward to another square on the route"
    targetIndex := (route indexOf: aToken square) + self die topFace.
    targetIndex <= route size ifTrue: [ ^ route at: targetIndex ].

    "(3) the roll would go past the end of the route (we stay where we are)"
    ^ aToken square
```

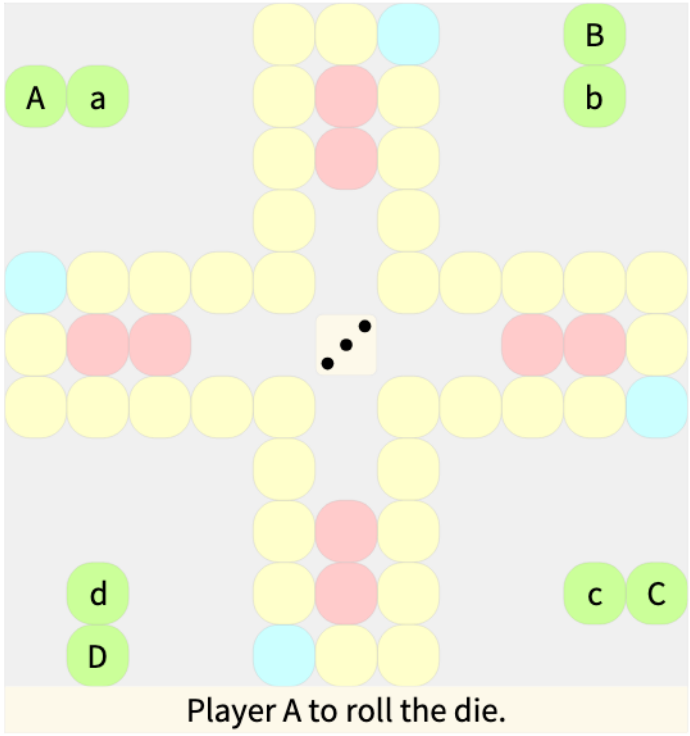
playing instance

Exploring a live system

Let's inspect a running instance of a Ludo game.

```
GtLudoGame ▸ new ▸ .
```

From the instance we can see various *views*, navigate to the *source code*, and explore *examples*.



▸ 1 explicit reference

nil

Changing a running system

We would like to add an *autoplay* feature to a running game.

```
game := GtLudoGame new.
```

We can control the game programmatically.

```
game roll: 6.
game moveTokenNamed: 'a'.
```

We can automate a single move:

```
game playerToRoll ifTrue: [ game die roll ].
```

```
game playerToMove
    ifTrue: [ game moveTokenNamed:
                game tokensToMove atRandom name ].
```

And we can combine the actions.

```
game playerToRoll ifTrue: [ game die roll ].
game playerToMove
    ifTrue: [ game moveTokenNamed: game tokensToMove
atRandom name ].
```

We can extract this now as a method.

```
1 to: 100 do: [ :n | game autoPlay ].
```

1 explicit reference

nil

Composing (Test) Examples

An *example* is a test method that returns the object under test:

GToolkit-Demo-Ludo > GtLudoGameExamples

emptyGame

```
<gtExample>
| game |
game := self gameClass new.
self assert: game isOver not.
self assert: game winner equals: 'No one'.
self assert: game currentPlayer name equals: 'A'.
self assert: game playerToRoll.
self assert: game playerToMove not.
^ game
```

example instance

▸ 1 explicit reference

nil

Recording and visualizing token moves

Tracking the moves

```
GtLudoRecordingGameExamples new
playerAovershootsGoal .
```

Autoplaying a game with history

```
game := GtLudoRecordingGame new .
```

As we play moves, we see the history update.

```
game autoPlay: 1.
```

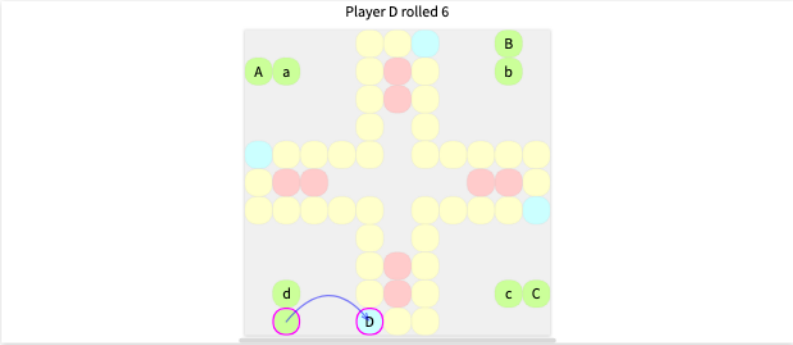
1 explicit reference

nil

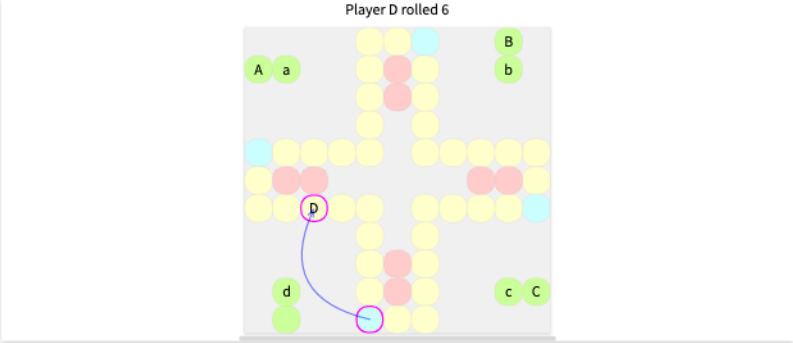
Live documentation of the game logic

We can use live examples to document the logic of an application.

The initial move



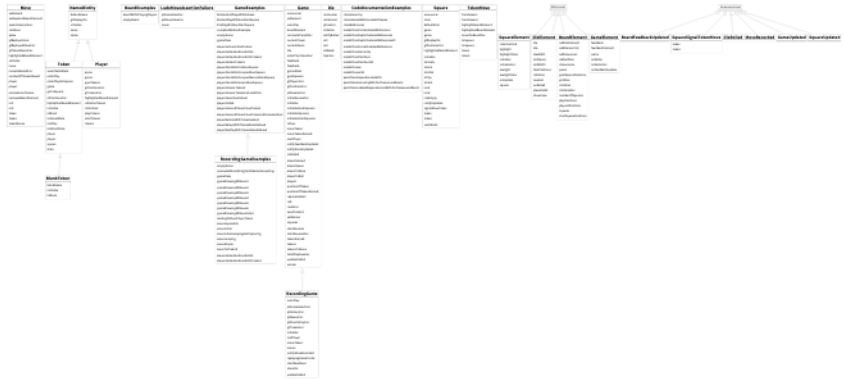
A regular move



nil

Live documentation of the technical solution

We can easily obtain a UML diagram for the whole implementation, but it is a bit cluttered:

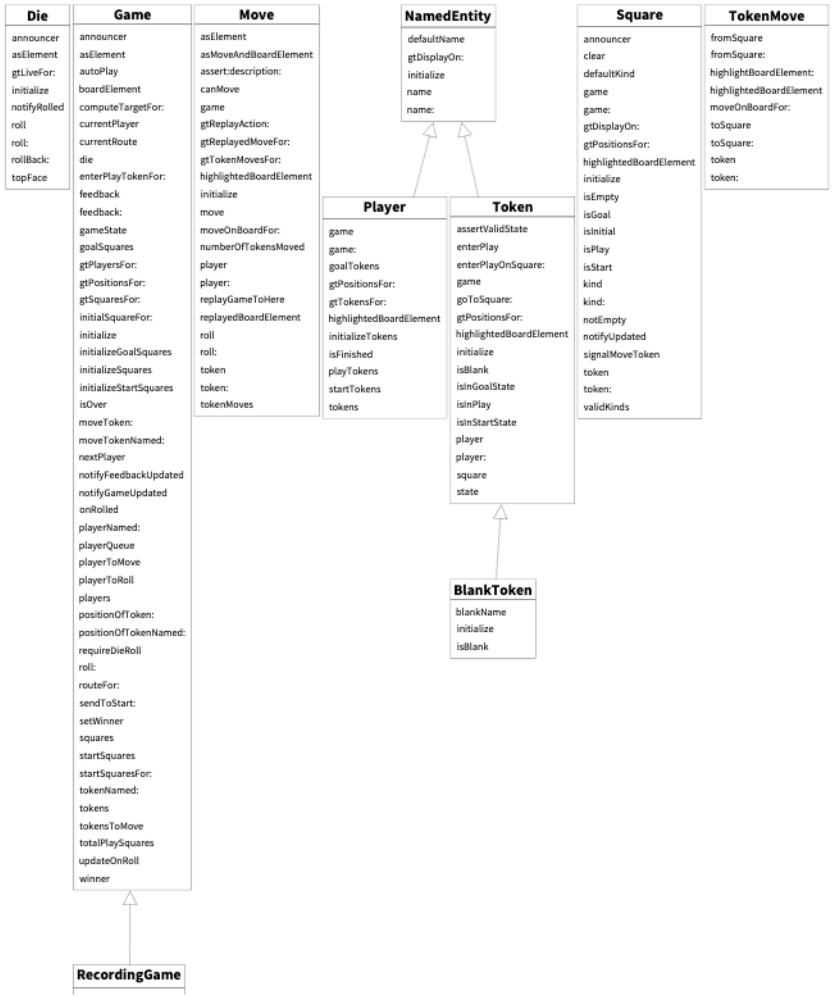


▶ 1 explicit reference

nil

Model classes

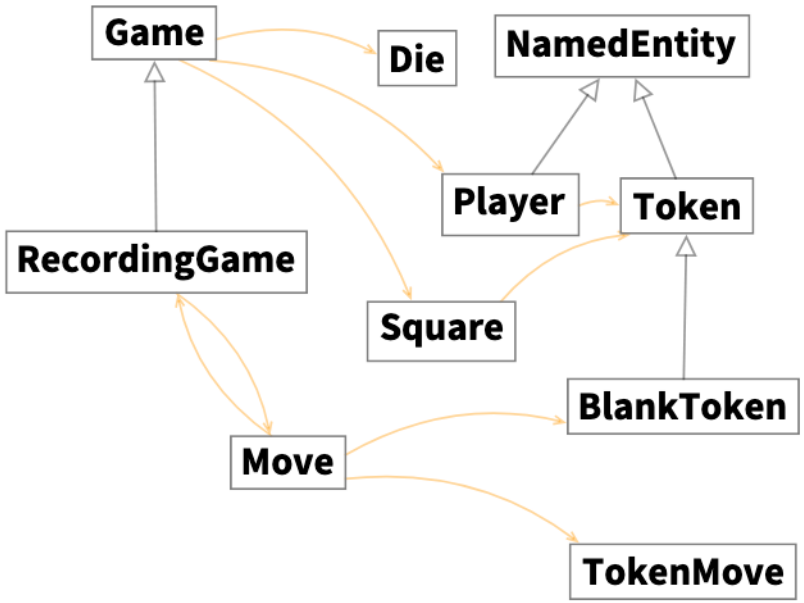
It can be more useful to focus just on a subset of the implementation, such as just the model classes.



nil

Relationships between classes

We can also visualize the relationships between classes.



▸ 1 explicit reference

nil

Take home messages

- Smalltalk enables *live programming*
- Example methods enable *composable tests* and live interaction with test results
- *Moldable development* brings together coding and exploration of live systems

What's next?

- Download GT from gtoolkit.com
- Explore the Glamorous Toolkit Book
- Learn about Pharo — go to books.pharo.org
- Have fun!