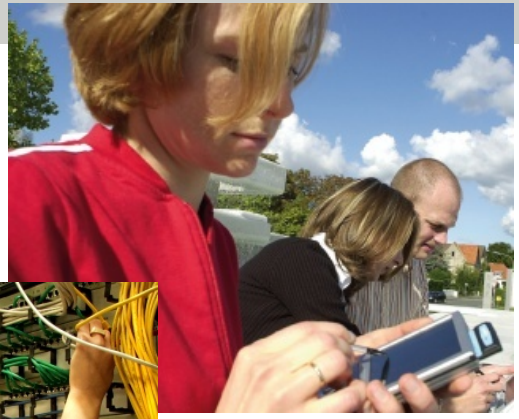
PADERBORN UNIVERSITY
The University for the Information Society

Embracing Scribunto

A path to Entity Class Management

EMWCon Fall 2017



IMT: Zentrum für Informations- und Medientechnologien

Tobias Oetterer
Center for Information and Media Technologies

Introduction

Image from https://clipartfest.com

Tobias Oetterer
Center for Information and Media Technologies

Introduction

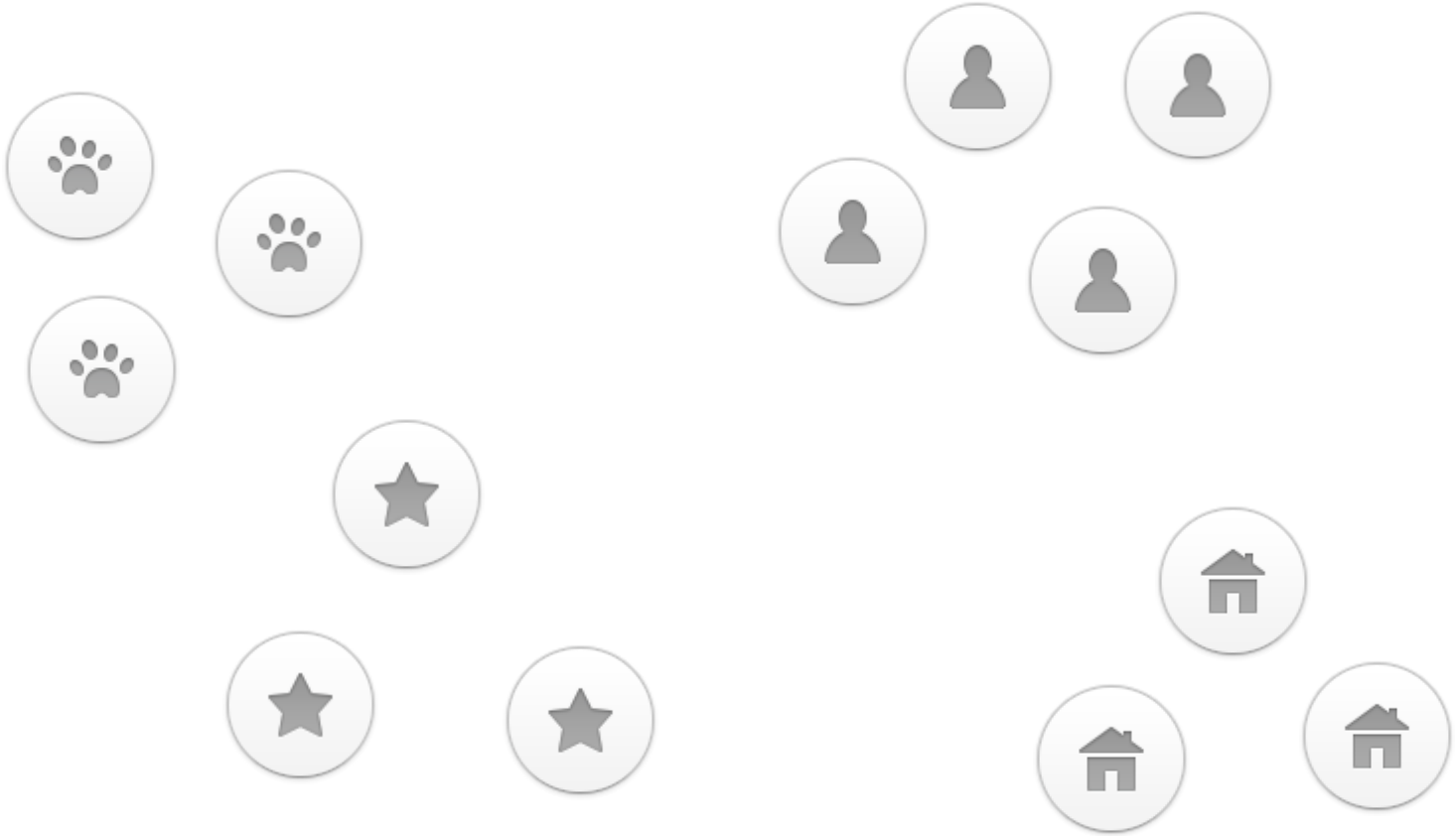
Let's assume, you want to create a new entity class for your wiki. Furthermore, let's assume, it is already designed and ready to be implemented. What now?

I want you to take part in the experience I gained during the last years and my gradual shift from the conventional use of templates/forms to an extended system utilizing Extension:Scribunto. I'll show you how a division of class description and implementation could look like, introduce a way of maintaining a unified codebase, give you a feeling of the possibilities lua offers, when taken to extreme, and finally let you have a first peak how creation, deployment and management of entity classes in a wiki could look like.

Be prepared to be confronted with opinions, see a struggling wikidev learn through failure and expect to be asked for insight into your own techniques.

Tobias Oetterer
Center for Information and Media Technologies

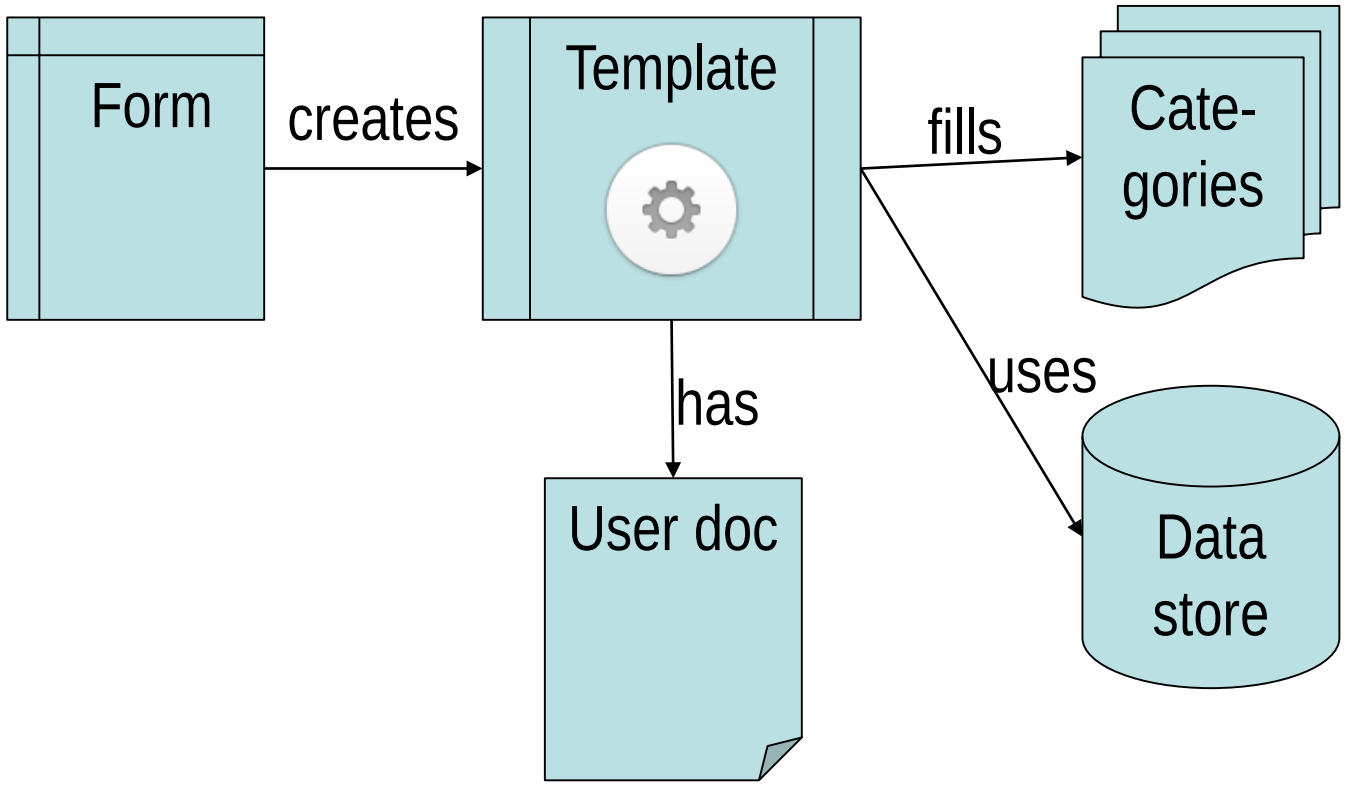
Introduction: What are entity classes



Images from http://www.whoishostingthis.com/resources/free-icons/

Tobias Oetterer
Center for Information and Media Technologies

Entity Class Engine



Tobias Oetterer
Center for Information and Media Technologies

Terms

Overloaded

- Class
- Property
- Category

Tobias Oetterer
Center for Information and Media Technologies

A short Introduction to Scribunto and Lua

Scribunto is an extension that allows for the embedding of lua scripts in MediaWiki.

> “[Lua] is a lightweight multi-paradigm programming language designed primarily for embedded systems and clients”.
>
> https://en.wikipedia.org/wiki/Lua_(programming_language)

For further reading, please see
https://en.wikibooks.org/wiki/Scribunto:_An_Introduction

Tobias Oetterer
Center for Information and Media Technologies

Entity class wish list

- Manageable codebase

Tobias Oetterer
Center for Information and Media Technologies

Maintainable codebase: Why

Let's address the elephant in the room: templates are a mess when dealing with process logic. Wikimarkup is not meant to handle complex code and templates were originally designed to show predefined text snippets.

There is some help available, when using templates this way:

- Extensions like ParserFunctions, Variables, and Arrays
- Encapsulate functionality in sub-templates
- Comments at strategic positions
- Syntax highlighter gadget [0]

Of course, you need to stick to the regiment and have the self-control to code correctly. Also, when using encapsulation, you should maintain an overview of dependencies.

[0]: https://www.mediawiki.org/wiki/Extension:Gadgets#List_of_gadget_scripts

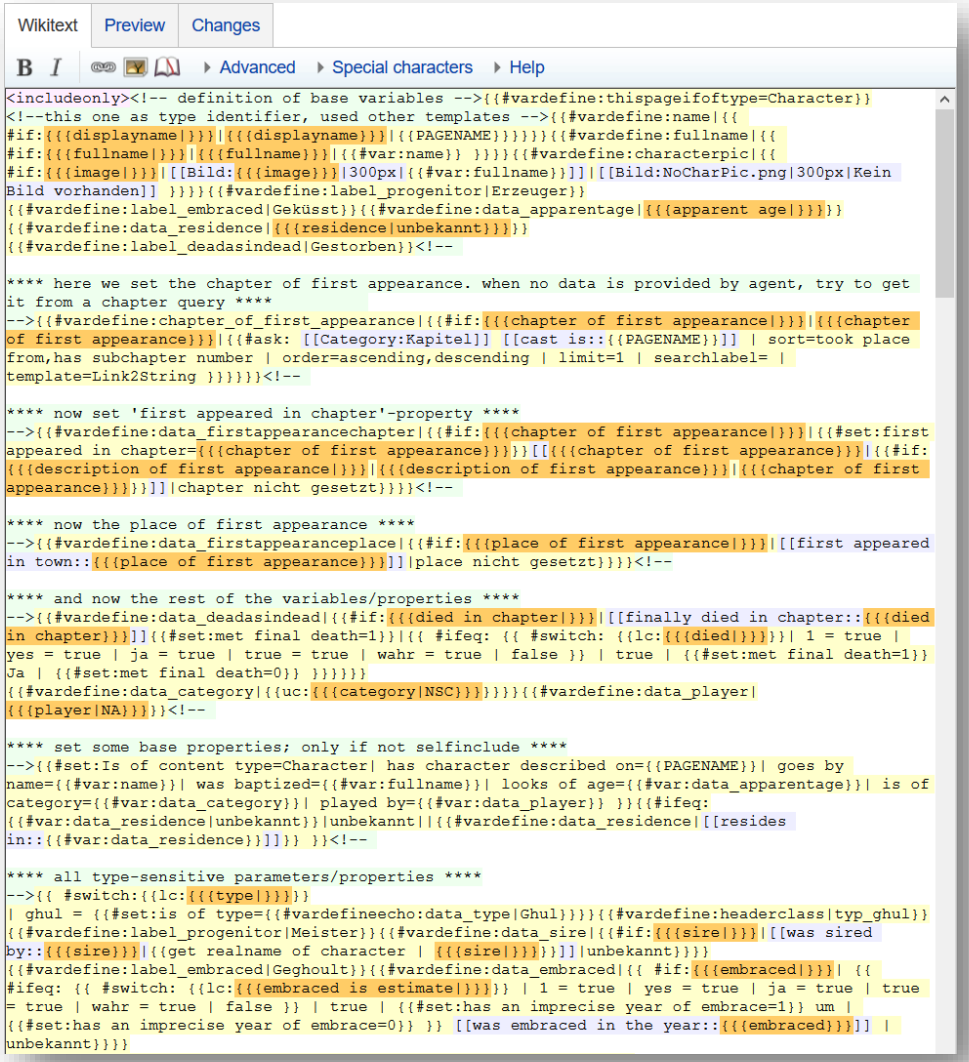
Tobias Oetterer
Center for Information and Media Technologies

Maintainable codebase: Why?

This is a parser function heavy template viewed in editor

You can see that it utilizes comments to improve readability. Also apparent are the improvements from the Syntax highlighter Gadget.

Back then this was a heavy duty template and ran badly performance-wise. Also: gadgets weren't available, yet.



Tobias Oetterer
Center for Information and Media Technologies

Maintainable codebase: How?

This is how code could look like in lua on its module page.

Here, asides from Scribunto the Extension:SyntaxHighlight is used to increase readability.

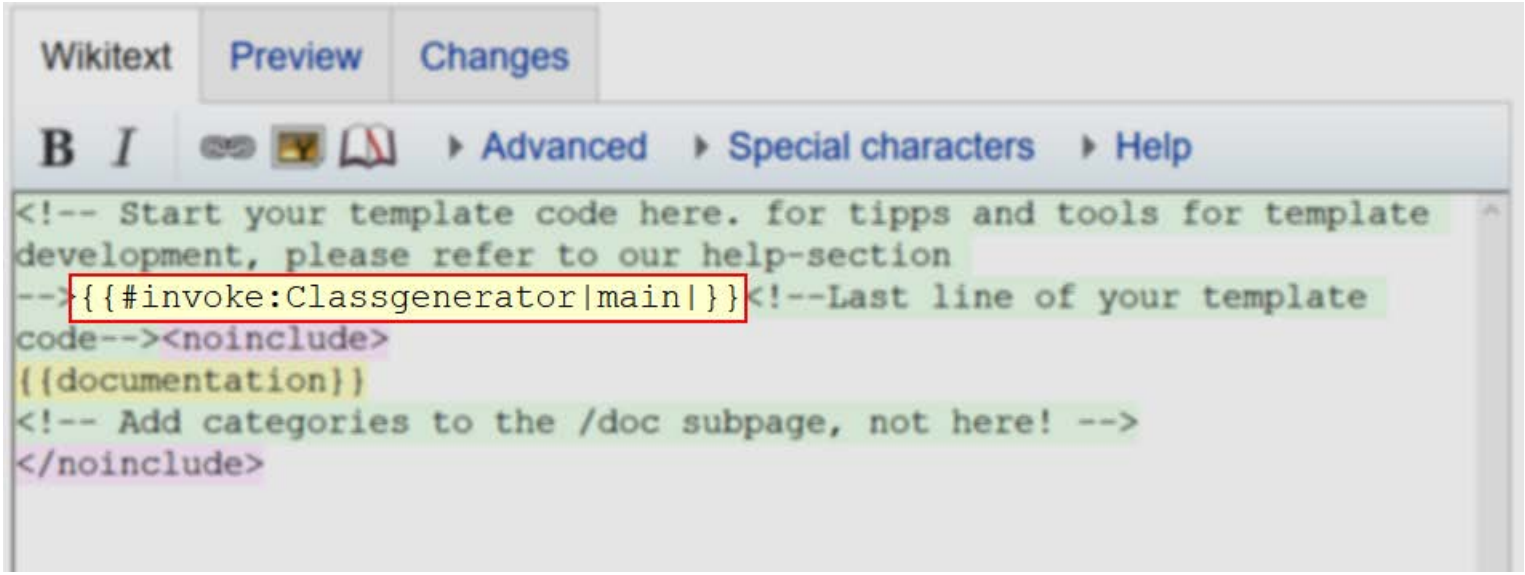
I sugguest, installing the Wikieditor and also Extension:CodeEditor to have an easier life editing modules.

Note, that I switched to documenting with the help of a luadoc parser – albeit manually.

```lua
--- This returnes entity's uid and a data handler, containing entity's data.
-- @function ._initDataCore
-- @tparam base self this instance
-- @tparam table|string|nil data either a table holding entity data or a string
-- holding a uid or nil (in which case, we try to get data from frame's arguments).
-- @treturn string uid
-- @treturn dataHandler
_initDataCore = function(self, data)
    local dataHandler = class.myFactory:newDataHandlerWith(
        self:getPropertyHandler()
    )
    local uid, loadingData
    if data == nil then
        -- get data from frame args
        uid, loadingData = _initializationDataFromArguments(self)
        _private[self].initializedWithFrameArgs = true
    elseif type(data) == 'table' then
        -- use directly
        uid, loadingData = _initializationDataFromDataTable(self, data)
    else
        -- try to init self with data as uid
        uid, loadingData = _initializationDataFromStore(self, data)
    end
    -- inject uid and cid and load into dataHandler
    loadingData[mw.ipso.const.nameOfEntityIdProperty] = uid
    loadingData[mw.ipso.const.nameOfClassIdProperty] = self.class.name
    dataHandler:load(loadingData)
    if not dataHandler:verify() then
        -- data verification failed. fetch errors
        self:addError(dataHandler:getErrors())
    end
    return uid, dataHandler
end
```

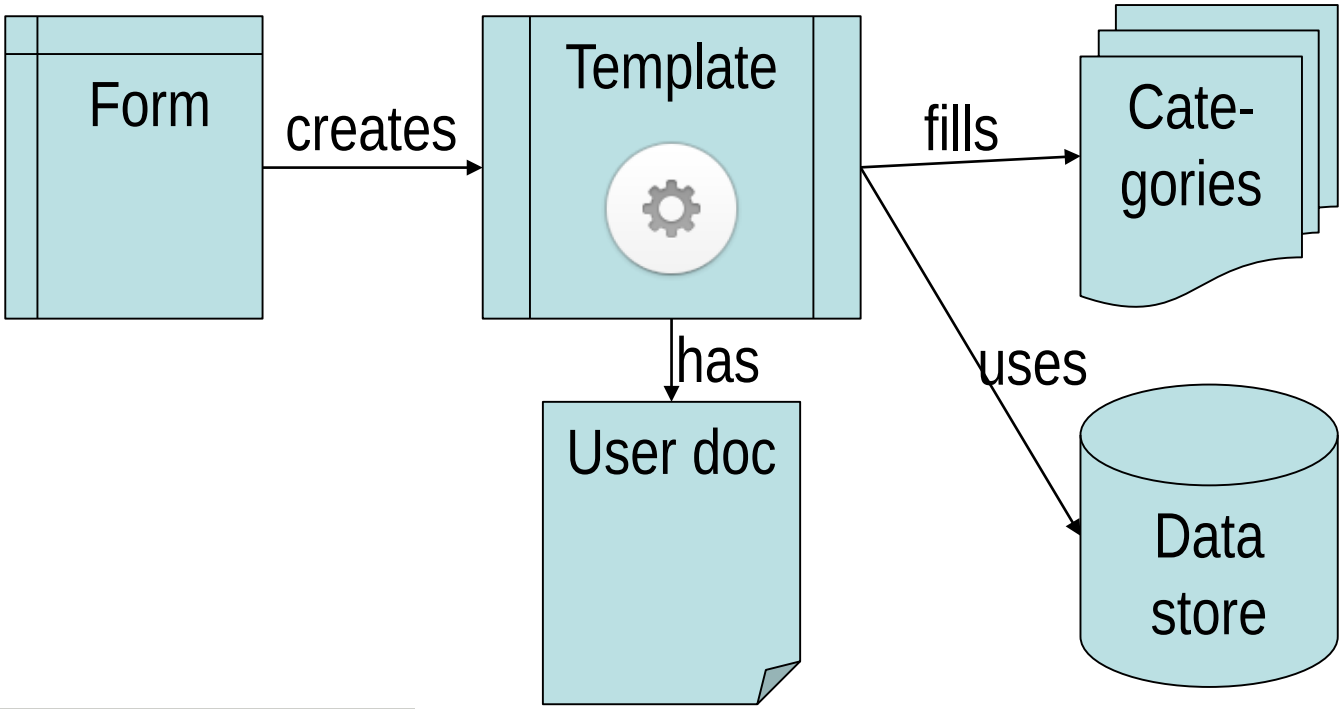
Tobias Oetterer
Center for Information and Media Technologies

Maintainable codebase: How?



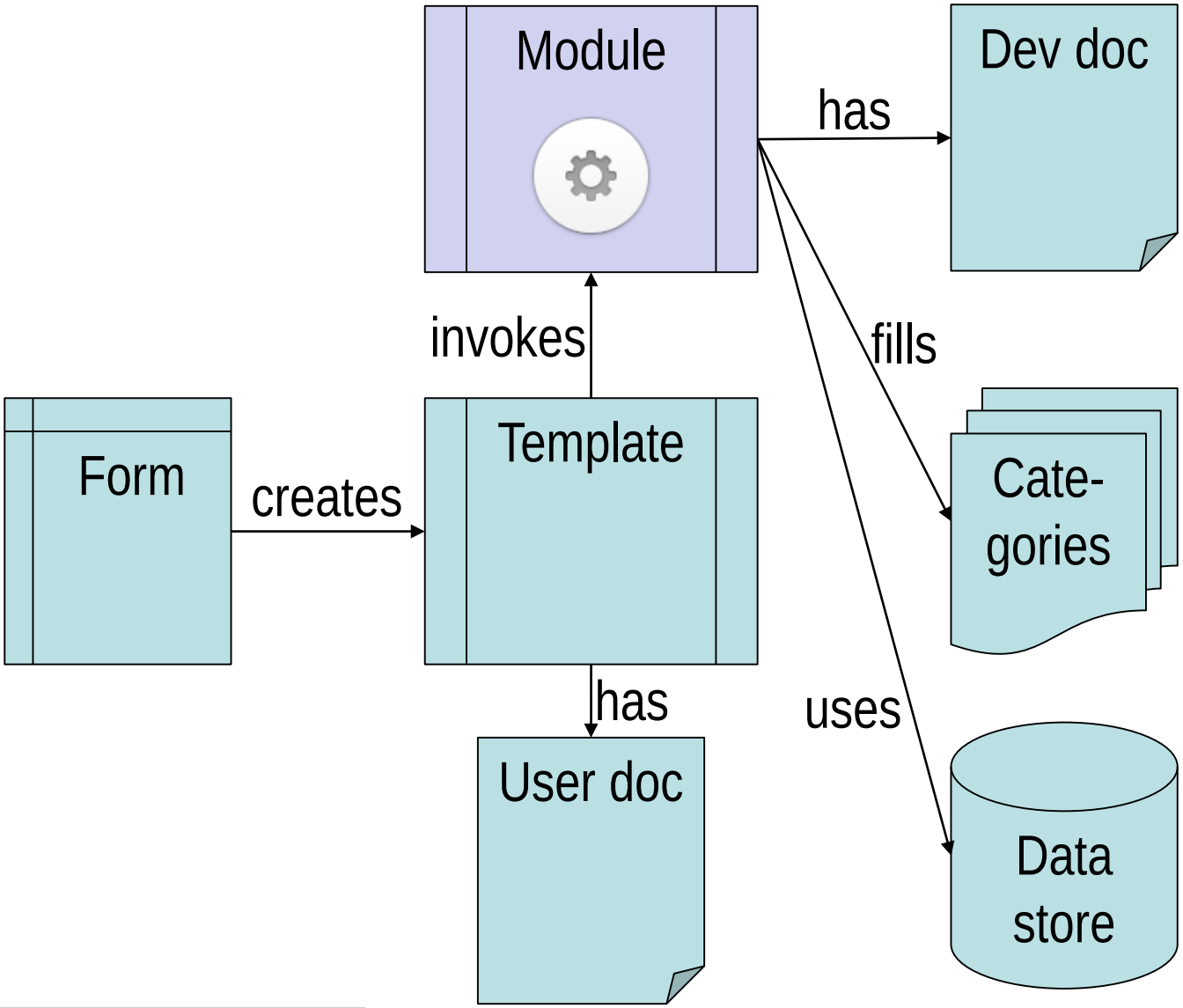
Tobias Oetterer
Center for Information and Media Technologies

Entity Class Engine



Tobias Oetterer
Center for Information and Media Technologies

Entity Class Engine



Tobias Oetterer
Center for Information and Media Technologies

Entity class wish list

- Manageable codebase
- Plausible data

Tobias Oetterer
Center for Information and Media Technologies

Plausible data: Why?

I don't trust user input. Even when using PageForms [0] and hiding the normal edit tab, there is still the possibility, that a user edits the page source directly. And I need to rely on two things:

- Mandatory data should be present
- Restriction of allowed values should not be superseded

When adding an entity to my data store, I need to trust these two assumptions. An entity failing its integrity check

- Must not be stored
- Should be flagged down for inspection

[0]: https://www.mediawiki.org/wiki/Extension:Page_Forms

Tobias Oetterer
Center for Information and Media Technologies

Plausible data: How then?

To test for existence of mandatory fields was easily done with a

{{#if:{{{mandatory|}}}}||…}}

statement.

For predefined values I used an array [0]: first define a list of valid values and later test, if your parameter is in it.

Testing for dynamic values was done similarly, only the array was initialized by an #ask-query.

This latter method was quite performance hungry.

[0]: https://www.mediawiki.org/wiki/Extension:Arrays

Tobias Oetterer
Center for Information and Media Technologies

Plausible data: How now?

Now, I have all the information I need to verify my user input stored in an entity property definition. All the testing is done based on this configuration.

Furthermore, I mostly dropped the idea of testing for dynamic lists of allowed values. However, with Scribunto this can be done easily and less performance-hungry than with templates.

```
properties = {
    name = {
        description = 'Every toy has its name',
        disabled = false,
        label = 'Name',
        link = false,
        list = false,
        mandatory = true,
        form = true,
        store = 'has name',
        type = 'string',
        values = false,
    },
    type = {
        description = 'And every toy has a certain type.',
        disabled = false,
        label = 'Type',
        link = true,
        list = false,
        mandatory = true,
        form = true,
        store = 'has assigned type',
        type = 'string',
        values = { 'car', 'teddy', 'barbie', 'lego' },
    },
    manufacturer = {
        description = 'This company produced the toy.',
        disabled = false,
        label = 'Made by',
        link = 'manufacturer',
        list = true,
        mandatory = false,
        form = true,
        store = 'has assigned type',
        type = 'string',
        values = false,
    },
},
```

Tobias Oetterer
Center for Information and Media Technologies

Entity class wish list

- Manageable codebase
- Plausible data
- Division of configuration and code

Tobias Oetterer
Center for Information and Media Technologies

Division of configuration and code: Why?

Not embedding values for your class-properties in your code has obviously certain benefits:

- Ease of adaption
- Ease of reusability
- Ease of assessment

Tobias Oetterer
Center for Information and Media Technologies

Division of configuration and code: How?

Easy:

```
config = mw.loadData('Module:MyClass/config')
-- or
config = mw.text.jsonDecode( require('Data:MyClass config.json') )
```

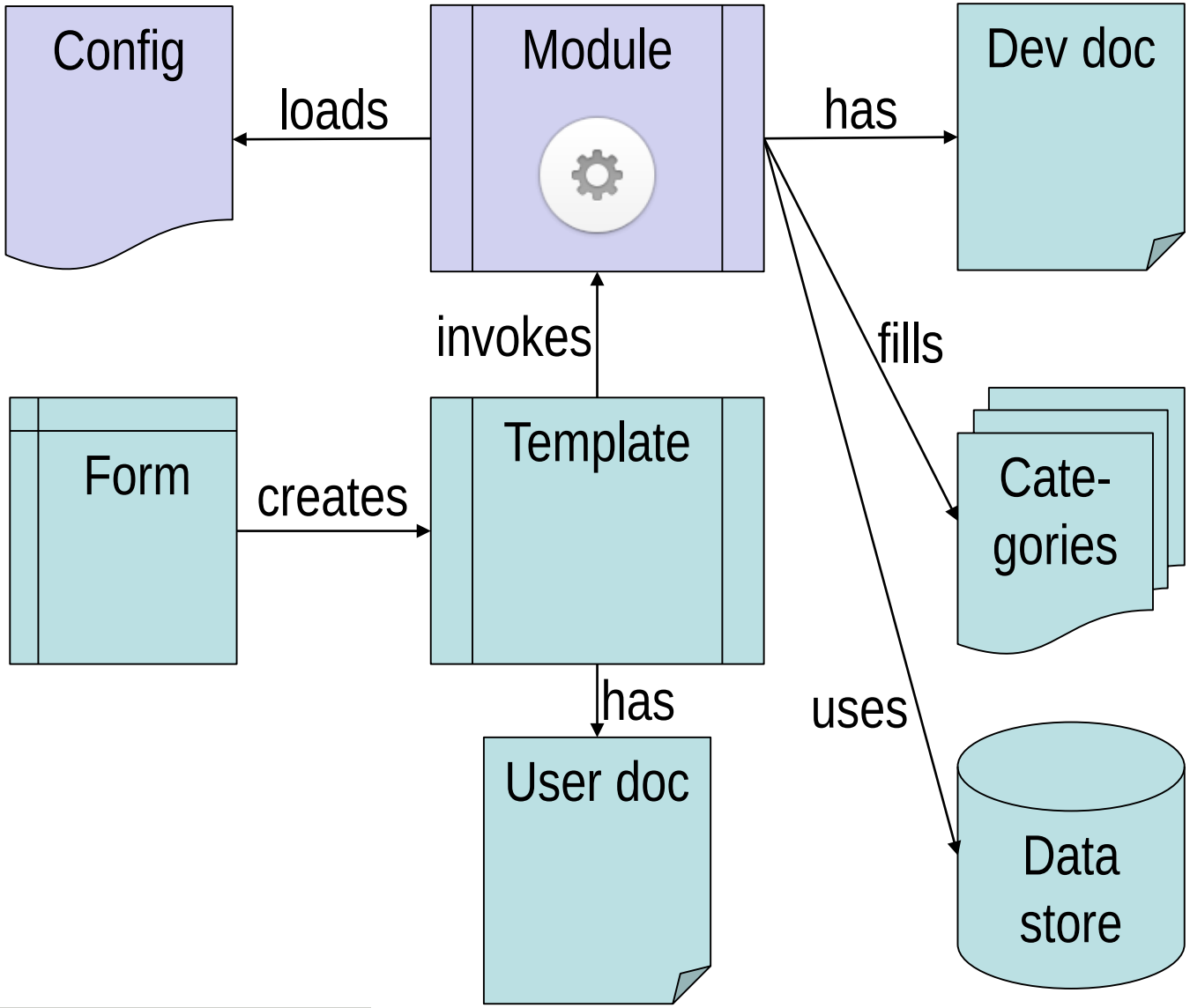
You define your configuration structure and place it on a second page, e.g. as a lua module or a json file. Then simply proceed to import this configuration and access it, when and wherever needed.

Techniques for that are

- Global variables (global in the module context, not super global)
- Class properties
- Import again on every use (lua has a caching mechanism)

Tobias Oetterer
Center for Information and Media Technologies

Entity Class Engine



Tobias Oetterer
Center for Information and Media Technologies

Entity class wish list

- Manageable codebase
- Plausible data
- Division of configuration and code
- Unified codebase

Tobias Oetterer
Center for Information and Media Technologies

Unified codebase: Why?

Building lua modules for the ECs sees a pattern emerging:

- Fetch data from template arguments
- Do plausibility tests (data verification)
- Preprocess its data
- Store Semantic properties
- Display the infobox and/or errors

This led to the idea to create a unified code base. Through that, it was possible to:

- Create new classes faster
- Introduce new features more easily

Tobias Oetterer
Center for Information and Media Technologies

Unified codebase: How?

The obvious answer is an inheritance pattern. So I looked for a way to incorporate OOP into lua.

With the help of metatables, this can be done natively. However, I wanted a clean division of classes and instances and a tested inheritance pattern.

After some investigation [0], I opted for a module called middleclass [1] and created a base class with all the commonly used methods. All that was left to do was to create a small shell class for every EC that inherits from this “foundation” class.

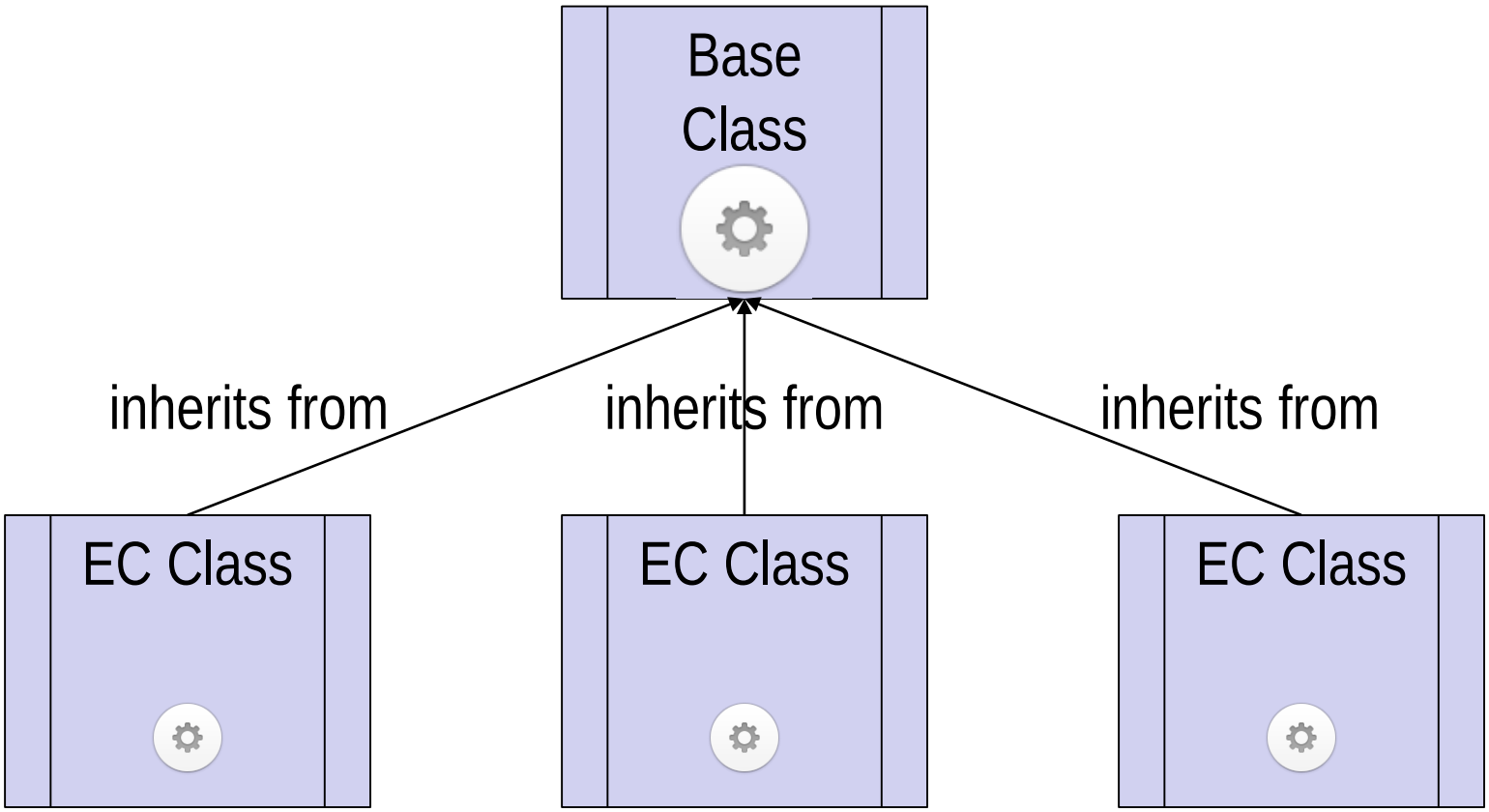
Each EC’s lua class also does additional, individual data processing and of course output formatting.

[0]: http://lua-users.org/wiki/ObjectOrientedProgramming

[1]: https://github.com/kikito/middleclass

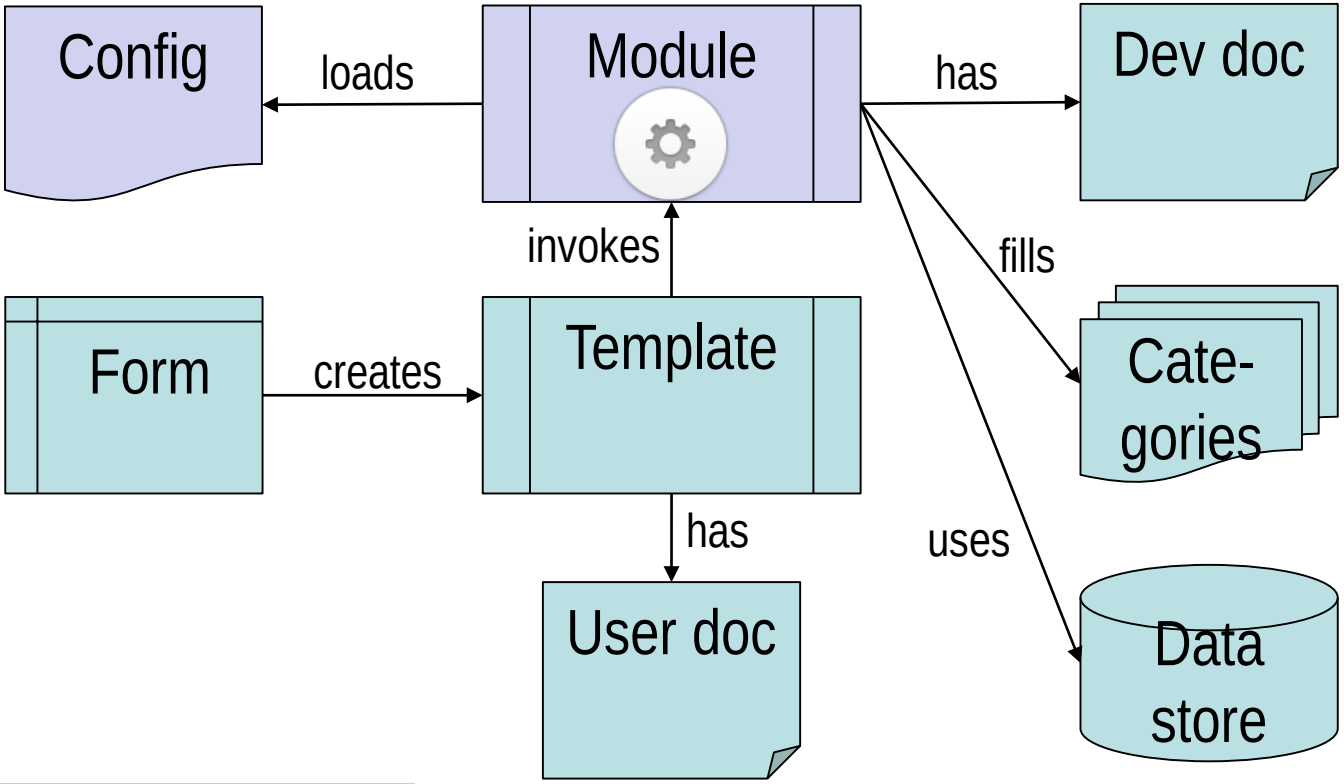
Tobias Oetterer
Center for Information and Media Technologies

Unified codebase: How?



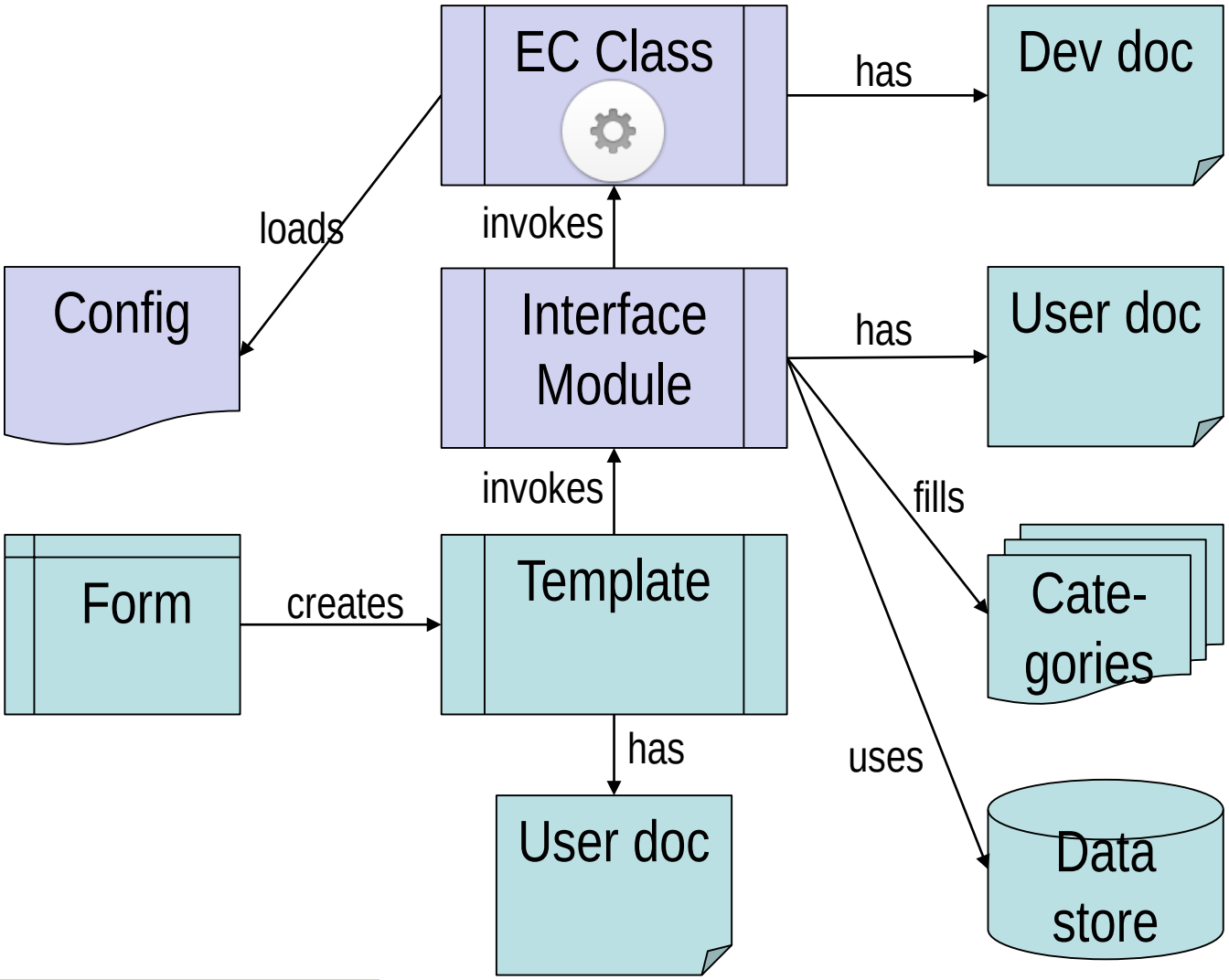
Tobias Oetterer
Center for Information and Media Technologies

Entity Class Engine



Tobias Oetterer
Center for Information and Media Technologies

Entity Class Engine



Tobias Oetterer
Center for Information and Media Technologies

Entity Class Engine

This module implements template {{Classgenerator}}.

Usage [Bearbeiten]

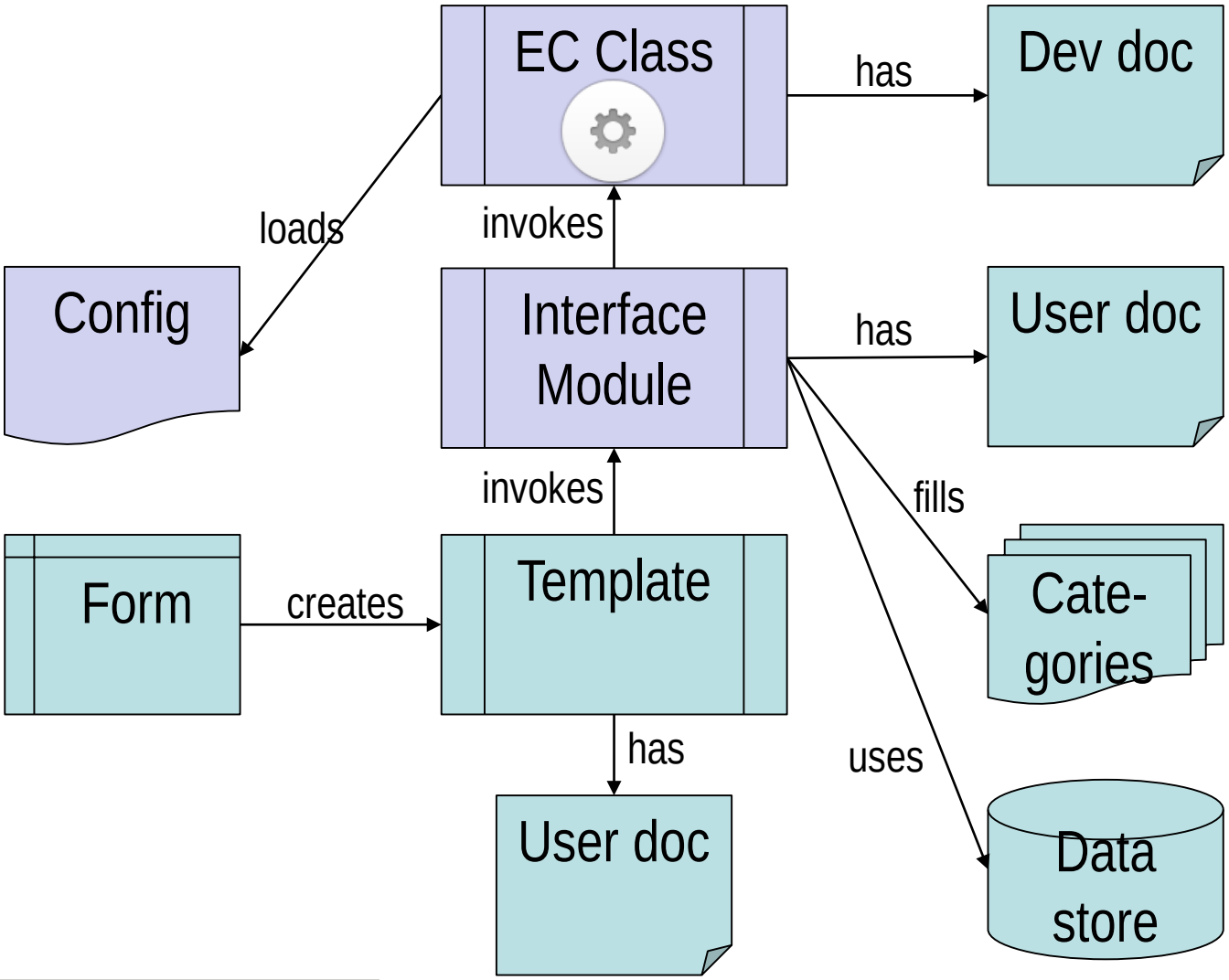
```
{{#invoke:Classgenerator|main}}
```

The above documentation is transcluded from Modul:Base/doc. (edit | history)
Subpages of this module.

```lua
local p = {}
local Class = require('Module:Classgenerator/class')
local getArgs = require('Module:Arguments').getArgs

function p.main(frame)
    local args = getArgs(frame)
    local me = Class:new(mw.title.getCurrentTitle().prefixedText)
    me:initFromArgs(args)
    me:storeData()
    me:addInfobox()
    me:addPageBody()
    return me:render()
end

return p
```

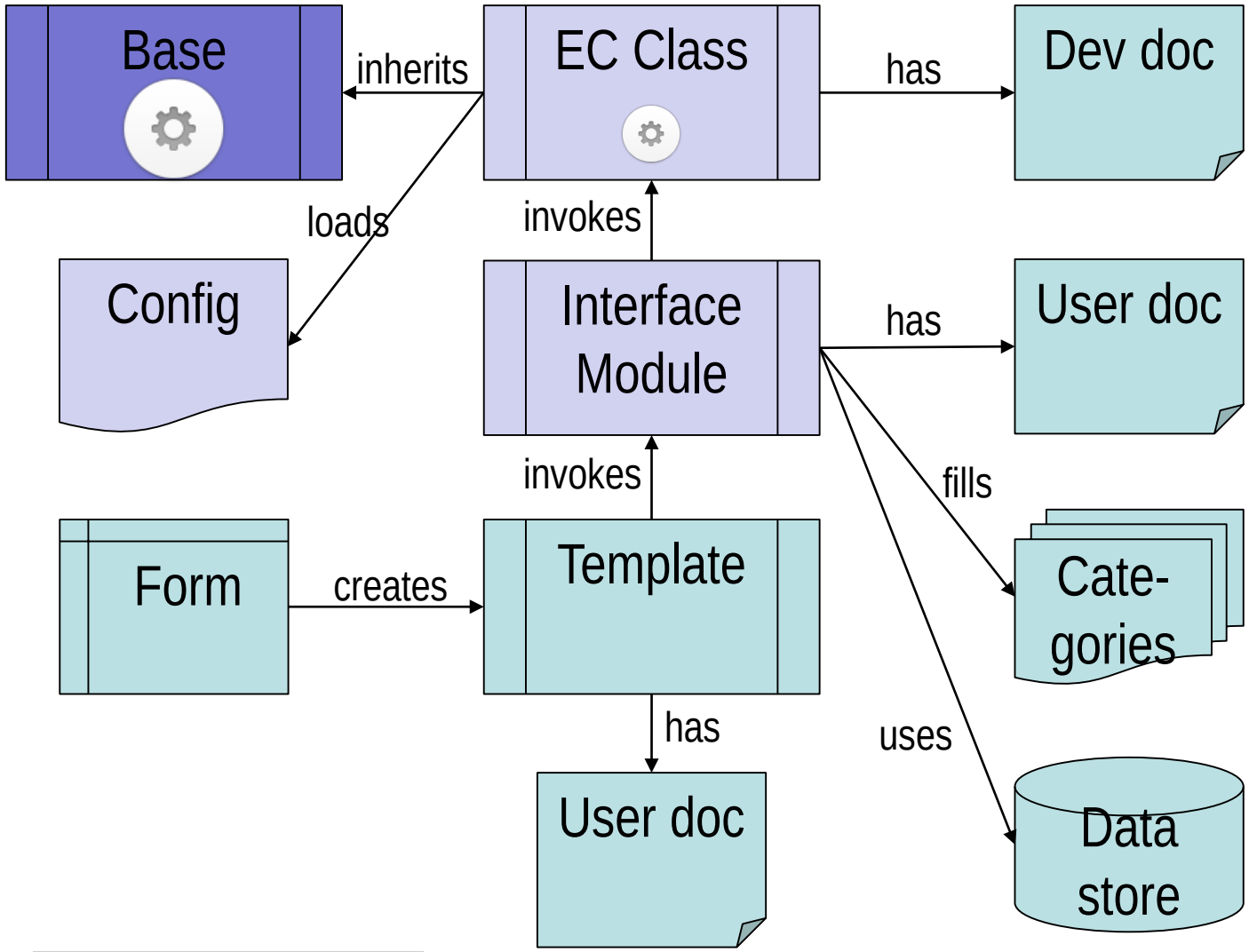
Tobias Oetterer
Center for Information and Media Technologies

Entity Class Engine



Tobias Oetterer
Center for Information and Media Technologies

Entity Class Engine



Tobias Oetterer
Center for Information and Media Technologies

Entity class wish list

- Manageable codebase
- Plausible data
- Division of configuration and code
- Unified codebase
- Auxiliary pages

Tobias Oetterer
Center for Information and Media Technologies

Auxiliary pages: Why?

Having a nice lua class at the core of an entity class still poses the task of creating the EC's form and template documentation manually.

This is insofar frustrating as to the availability of some of the necessary information inside your EC configuration:

- Mandatory or optional
- Single or list item
- List of allowed values
- Label and description

```
properties = {
    name = {
        description = 'Every toy has its name',
        disabled = false,
        label = 'Name',
        link = false,
        list = false,
        mandatory = true,
        form = {
            default = '{{PAGENAME}}',
            size = 40,
        },
        store = 'has name',
        type = 'string',
        values = false,
    },
    type = {
        description = 'And every toy has a certain type.',
        disabled = false,
        label = 'Type',
        link = true,
        list = false,
        mandatory = true,
        form = {
            input_type = 'radiobutton'
        },
        store = 'has assigned type',
        type = 'string',
        values = { 'car', 'teddy', 'barbie', 'lego' },
    },
    manufacturer = {
        description = 'This company produced the toy.',
        disabled = false,
        label = 'Made by',
        link = 'manufacturer',
        list = true,
        mandatory = false,
        form = {
            input_type = 'combobox',
            placeholder = 'pick one, please!'
        },
        store = 'has manufacturer',
        type = 'string',
        values = false,
    },
},
```

Tobias Oetterer
Center for Information and Media Technologies

Auxiliary pages: Why?

But since there was the “foundationclass”, every EC’s class was inheriting from, it was relatively easy to add functionality to

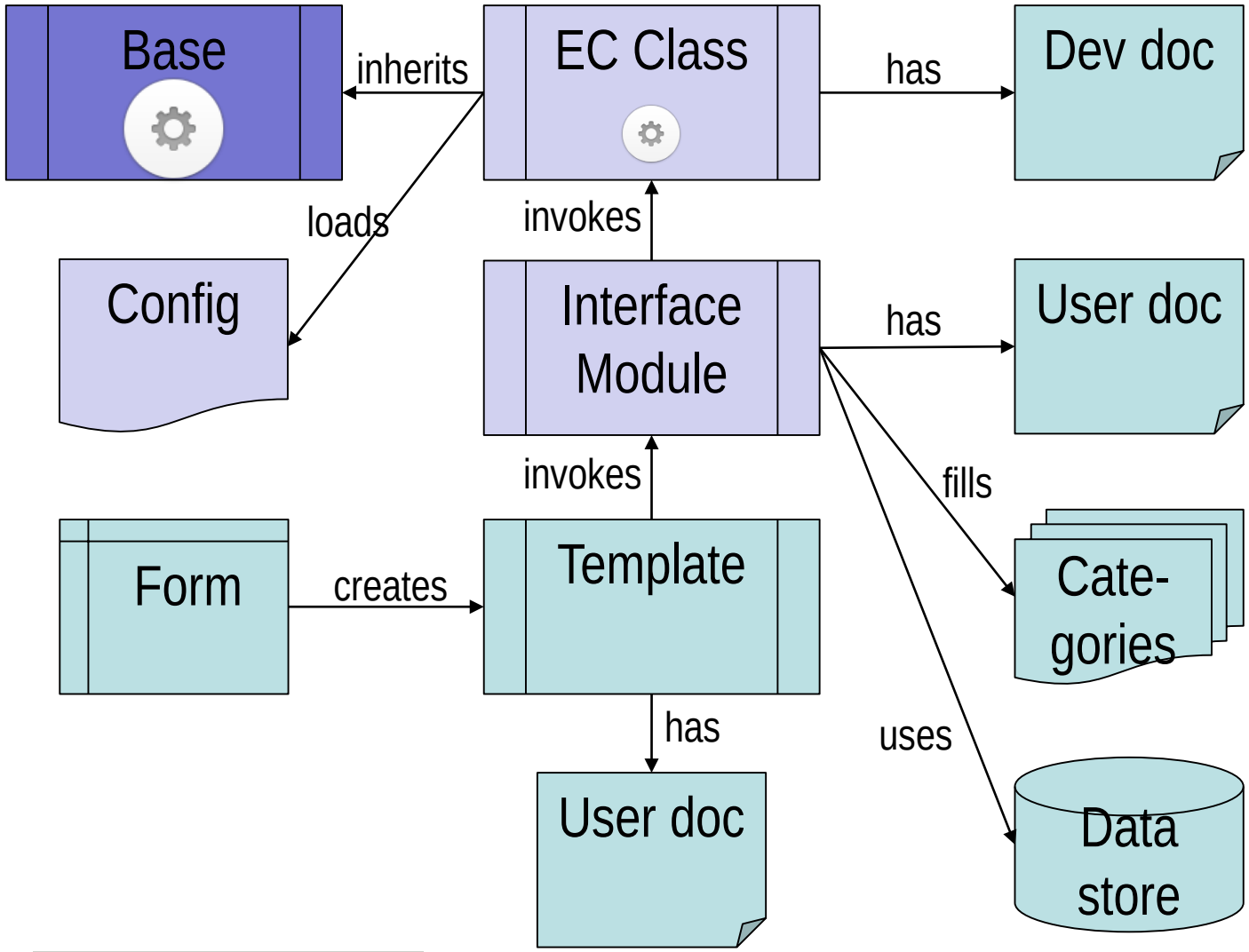
- Create a (static) method that produces template documentation
- Create a (static) form printer method

Admittedly, the form generator only works as-is for the most basic forms. For more complex requirements, the EC class has to implement its own function. However, there are methods that can be reused.

```
properties = {
    name = {
        description = 'Every toy has its name',
        disabled = false,
        label = 'Name',
        link = false,
        list = false,
        mandatory = true,
        form = {
            default = '{{PAGENAME}}',
            size = 40,
        },
        store = 'has name',
        type = 'string',
        values = false,
    },
    type = {
        description = 'And every toy has a certain type.',
        disabled = false,
        label = 'Type',
        link = true,
        list = false,
        mandatory = true,
        form = {
            input_type = 'radiobutton'
        },
        store = 'has assigned type',
        type = 'string',
        values = { 'car', 'teddy', 'barbie', 'lego' },
    },
    manufacturer = {
        description = 'This company produced the toy.',
        disabled = false,
        label = 'Made by',
        link = 'manufacturer',
        list = true,
        mandatory = false,
        form = {
            input_type = 'combobox',
            placeholder = 'pick one, please!'
        },
        store = 'has manufacturer',
        type = 'string',
        values = false,
    },
},
```

Tobias Oetterer
Center for Information and Media Technologies

Entity Class Engine



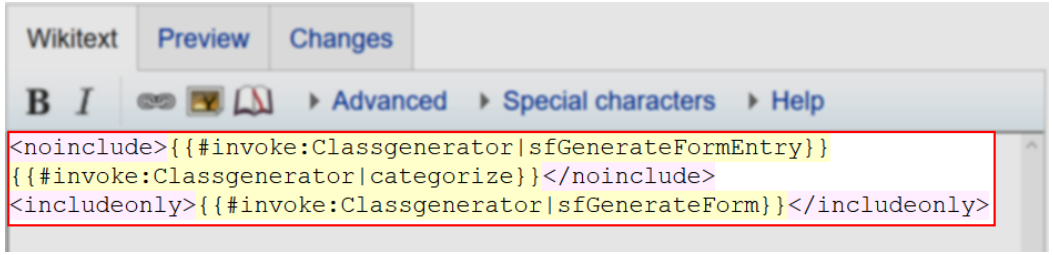
Tobias Oetterer
Center for Information and Media Technologies

Entity Class Engine

```lua
local p = {}
local Class = require('Module:Classgenerator/class')
local getArgs = require('Module:Arguments').getArgs

function p.categorize(frame)
    return tostring(Class:categorize())
end

function p.categoryPage(frame)
    return tostring(Class:categoryPage())
end

function p.explainDataStore(frame)
    return tostring(Class:explainDataStore())
end

function p.gardeningCategoryPage(frame)
    return tostring(Class:gardeningCategoryPage())
end

function p.sfGenerateForm(frame)
    return tostring(Class:sfGenerateForm())
end

function p.sfGenerateFormEntry(frame)
    return tostring(Class:sfGenerateFormEntry())
end

function p.templateDocumentation(frame)
    return tostring(Class:templateDocumentation())
end

function p.main(frame)
    local args = getArgs(frame)
    local me = Class:new(mw.title.getCurrentTitle().prefixedText)
    me:initFromArgs(args)
    me:storeData()
    me:addInfobox()
    me:addPageBody()
    return me:render()
end

return p
```

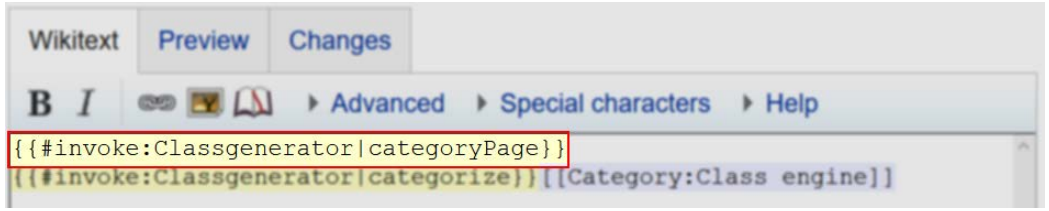
Tobias Oetterer
Center for Information and Media Technologies

Auxiliary pages: How?



Source of form page

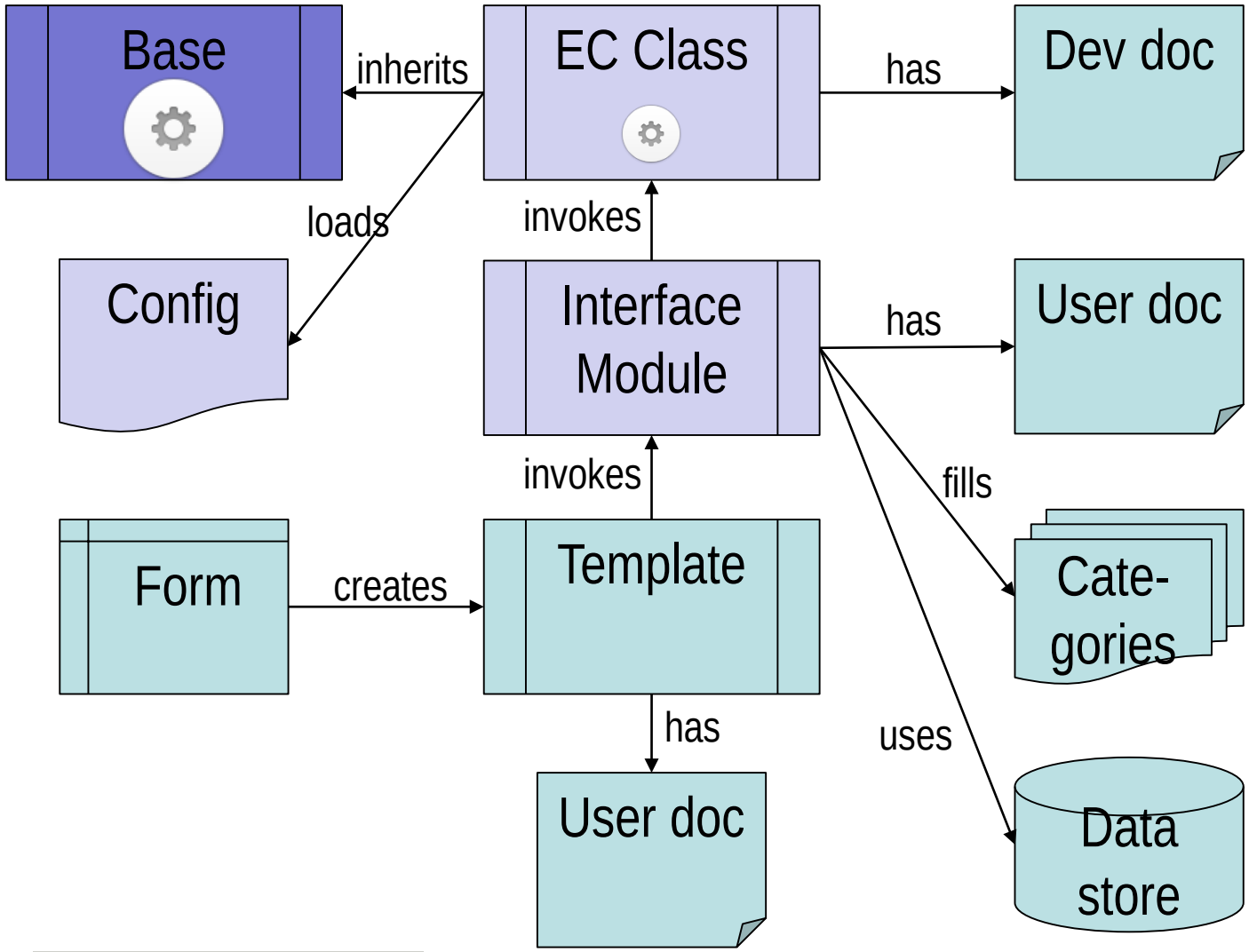
Source of template‘s doc page



Source of category page

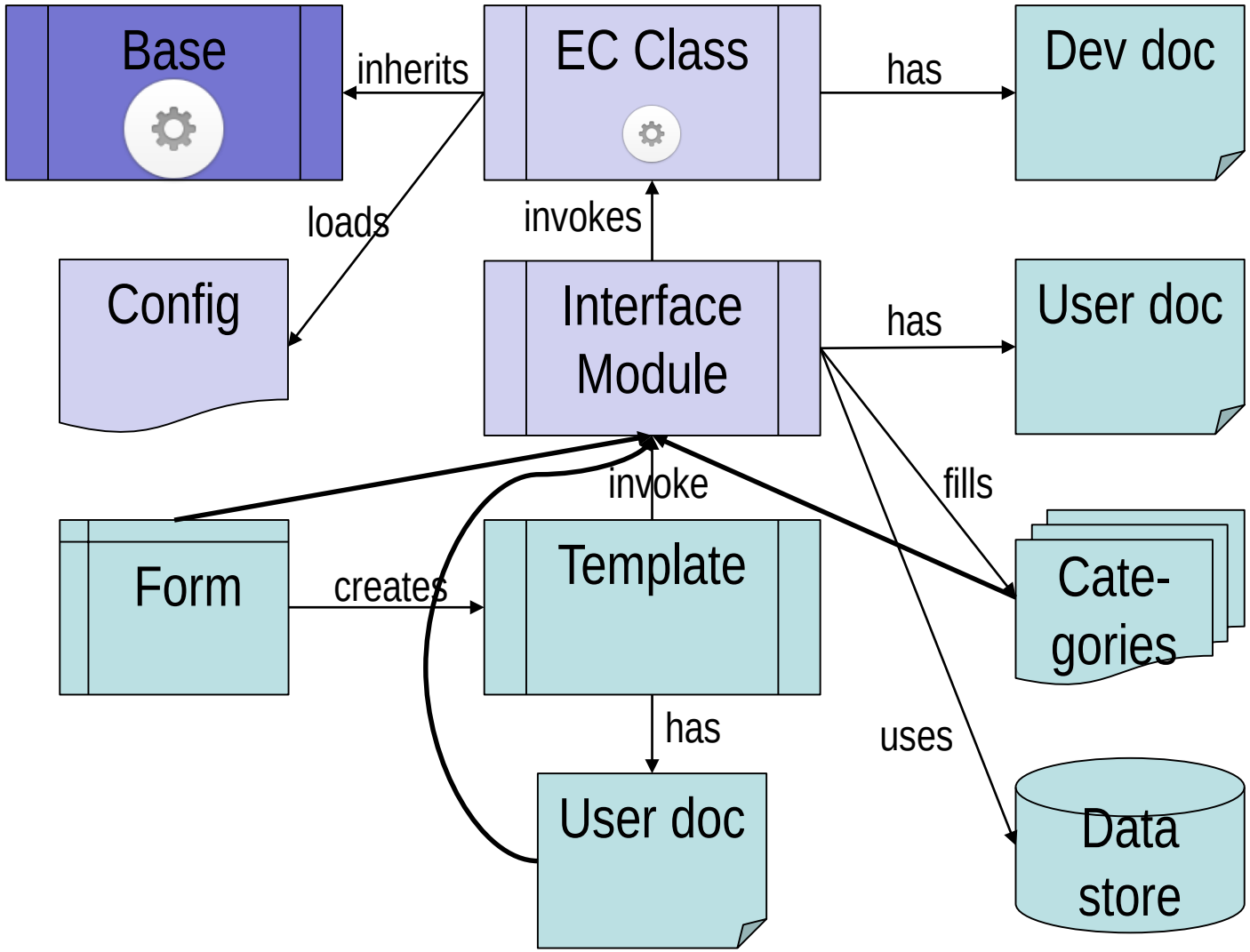
Tobias Oetterer
Center for Information and Media Technologies

Entity Class Engine



Tobias Oetterer
Center for Information and Media Technologies

Entity Class Engine



Tobias Oetterer
Center for Information and Media Technologies

Entity class wish list

- Manageable codebase
- Plausible data
- Division of configuration and code
- Unified codebase
- Auxiliary pages
- Class generation

Tobias Oetterer
Center for Information and Media Technologies

Class generation: Why?

Creating an EC now means the following tasks:

- Creating an interface module
- Creating the class module and adapting it, when necessary
- Creating the EC configuration module
- Creating all the module documentation pages
- Creating a template husk and its documentation
- Creating a form (if needed)
- Creating EC categories (if wanted)

Furthermore having a look at an EC configuration file, one can see a clear an well defined structure. This calls for more automation.

Tobias Oetterer
Center for Information and Media Technologies

Class generation: How?

This was the last class generated by hand. The resulting EC had the following functionalities:

- Provide an interface (form) to help defining an EC configuration
- Generate an overview page of all the information for the resulting class
- Create all the necessary pages for the EC, including
  - Module documentations
  - Template husk and user documentation
  - Form
  - Categories
  - Semantic properties

Unfortunately, the extension used for page creation cannot write in module namespace, so the creation of the modules has to done manually, still.

[0]: https://www.mediawiki.org/wiki/Extension:AutoCreatePage

Tobias Oetterer
Center for Information and Media Technologies

Class generation: How?

```
general = {
    category = 'Toys',
    delimiter = ',',
    description = 'Elements found in a Sandbox',
    entityTitle = '',
    gardeningCategory = 'Broken Toys',
    isSubObject = false,
    namespace = '',
    suppressErrors = false,
    suppressWarnings = false,
    store = 'smw',
},
form = {
    sections = {},
    allowFreeText = false,
    buttons = {'save', 'preview', 'changes', 'cancel'},
},
template = {

},
infobox = {
    'name',
    'type',
    'manufacturer'
},
```

```
properties = {
    name = {
        description = 'Every toy has its name',
        disabled = false,
        label = 'Name',
        link = false,
        list = false,
        mandatory = true,
        form = true,
        store = 'has name',
        type = 'string',
        values = false,
    },
    type = {
        description = 'And every toy has a certain type.',
        disabled = false,
        label = 'Type',
        link = true,
        list = false,
        mandatory = true,
        form = true,
        store = 'has assigned type',
        type = 'string',
        values = { 'car', 'teddy', 'barbie', 'lego' },
    },
    manufacturer = {
```

Tobias Oetterer
Center for Information and Media Technologies

Class generation: How?

GLOBAL-Section | FORM-Section | TEMPLATE-Section | PARAMETERS-Section

This section deals with all the class's global data, i.e. all data you need for your "template" part as wenn as for your form, ...

Objectclass

Title*	Test
Description*	Short description
Category	Category name
Gardening category*	individual
Gardening category name*	Category name
Namespace	Namespace
Cargo table	Testtest
Default restricted	User group
Default delimiter*	,
Global costum config	field1 = value,

(*) Pflichtfeld

Save page | Show preview | Show changes | Cancel

Tobias Oetterer
Center for Information and Media Technologies

Class generation: How?

Inhaltsverzeichnis [Verbergen]

Your 2do checklist

1. ✓ You created all your modules
2. ✓ You created all auxilliary pages
3. ✓ Your cargo table does exist! (Note: When you change your cargo table's declaration, you need to rebuild it, though)
4. ✓ You are done. Remember, when you want to change parameters in your config, edit this page and write your config file new

Data Storage

Title	MediaWiki Extension
Änderungsdatum	02.03.2016 11:17:29
Store available	cargo
Cargo table	extension
Namespace	Extension

Eine Extension ist eine funktionelle Erweiterung der Core MediaWiki Software.

Extensions werden vom IMT im System installiert und über dieses Template Kunden verfügbar gemacht.

Pages	
Module	Module:Extension
Class	Module:Extension/class
Config	Module:Extension/config
Template	Template:Extension
Form	Form:Extension
Category	Category:Mediawiki extensions
Gardening	Category:Extension pages with errors

Declaration for CARGO table "extension" [show]

Page status

This status may be inaccurate, if you just created pages. In that case, run jobs and clear cache of this page.

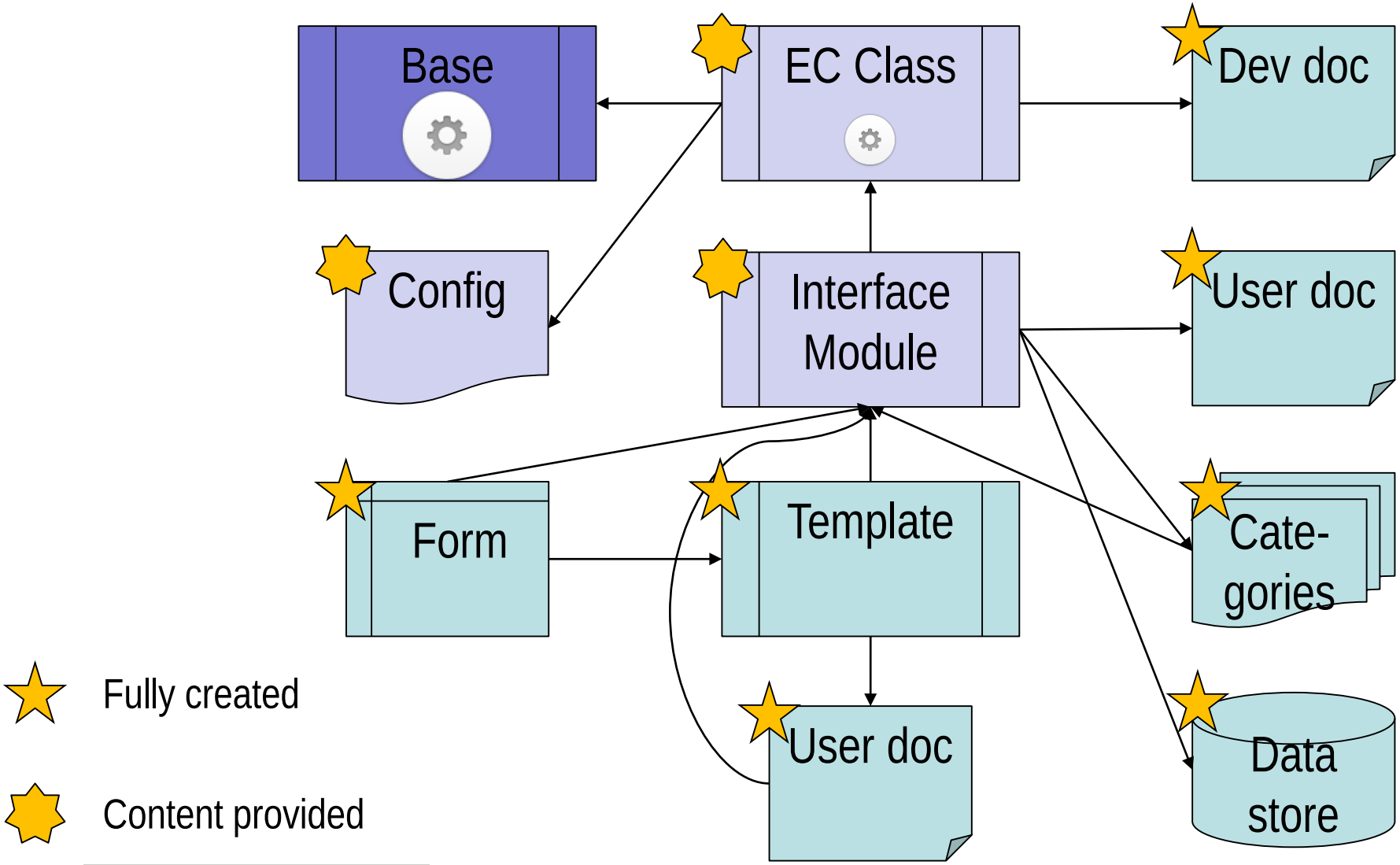
Module:Extension	✓
Module:Extension/doc	✓
Module:Extension/class	✓
Module:Extension/class/doc	✓
Module:Extension/config	✓
Module:Extension/config/doc	✓
Template:Extension	✓
Template:Extension/doc	✓
Form:Extension	✓
Category:Mediawiki extensions	✓
Category:Extension pages with errors	✓

Page autocreation

All pages present!

Tobias Oetterer
Center for Information and Media Technologies

Entity Class Engine



Tobias Oetterer
Center for Information and Media Technologies

Entity class wish list

- Manageable codebase
- Plausible data
- Division of configuration and code
- Unified codebase
- Auxiliary pages
- Class generation
- Class management

Tobias Oetterer
Center for Information and Media Technologies

Class management: Why?

At this point there is one control page for every EC in the wiki. This control page

- gives an overview of the EC
- provides functionality to create all the necessary pages for the EC

Updates to the configuration have to be done manually, as do exports and imports of the EC's engine and data pages.

Class management should have the following features

- Give an overview of classes, including interdependencies and foreign key relations
- Creation, update and decommission of ECs
- Export/import of EC definitions
- Export and import of the EC engine pages
- Creation, export and import of packages (collection of classes)
- API access

Tobias Oetterer
Center for Information and Media Technologies

Class management: How?

My current project: A MediaWiki Extension that provides a solid lua base class for EC data

- Input
- Processing
- Storage
- Output

Or Ipso (working title). The codebase is at 75%, missing “only” the form creation feature.

Building on that, the IpsoCM will hopefully provide an administrative Interface inWiki for all the aforementioned features. No release date yet.

Tobias Oetterer
Center for Information and Media Technologies

What are your thoughts?

From: http://www.magicalmaths.org/download-free-question-and-answer-images/