Gallows

a scaffold generator for Ruby on Rails employing ExtJS 3.0

or

how I managed to get my internship done without slipping into disaster

A thesis written by Matteo Settenvini,
student of the Computer Science course,
Faculty of Sciences, University of Padua

between August and September 2009
<matteo@member.fsf.org>

Supervisor:
Prof. Francesco Ranzato, University of Padua, dept. of Mathematics
<ranzato@math.unipd.it>

External Tutor:
Dott. Massimo Pegoraro, Diginess snc, Rubano (PD)
<info@diginess.com>

*This page has been intentionally left blank
for layout purposes.*

1. About this document

This is my bachelor's degree thesis, and it will present the work done in the months of August-September 2009 as an intern at Diginess snc in Rubano (PD). It is roughly divided in two parts: a first part which talks about the project itself and is aimed mainly at people interested in the technologies involved, and a second, more general part, that talks about my work experience *per se*, and contains (what I hope are) useful informations in order to get the maximum result when taking your first internship or employment as a programmer.

To be fair, this wasn't my first work experience. I managed some small one-man projects in the last six-to-seven years, I worked as a journalist for roughly six months in 2007, and had had both an internship of two months and then a regular job as an apprentice IT technician for a small company in Schio (VI) for a year and a half. There, I worked as head of help desk, managed customer relationships, and did some pre-selling of an Israelian software, as well as a little programming.

Therefore, even if my work experience is still limited, I hope to give you some insights that are coming from someone who, at least, has a fresh memory about how hard it is to transit from the university classes to the job market nowadays.

If you don't agree on some or all of the points presented therein, please feel free to contact me at matteo@member.fsf.org, specifying at the beginning of the subject line: "*Your Thesis:*". This will enable me to answer to you in a shorter time. I'm always open to suggestions and constructive criticism.

2. Disclaimer

All the code and other details relating to the project which has been implemented during my internship are property of Diginess snc, a web-agency located in Via Paolo da Sarmeola 4, Sarmeola di Rubano (PD), Italy.

Mr. Pegoraro, Diginess' CEO and my tutor during the internship, hereby grants permission of disclosure of these informations for didactic and study purposes, limited to the contents of this thesis only, which is released under a Creative Commons 3.0 By-NC-SA license.

Therefore, this document is considered falling outside the NDA signed by both parts at the beginning of the collaboration.

For acknowledgement,

Dott. Massimo Pegoraro
Diginess snc

3. License information

Table of Contents

4. A conceptual map

4.1. Implementing Gallows

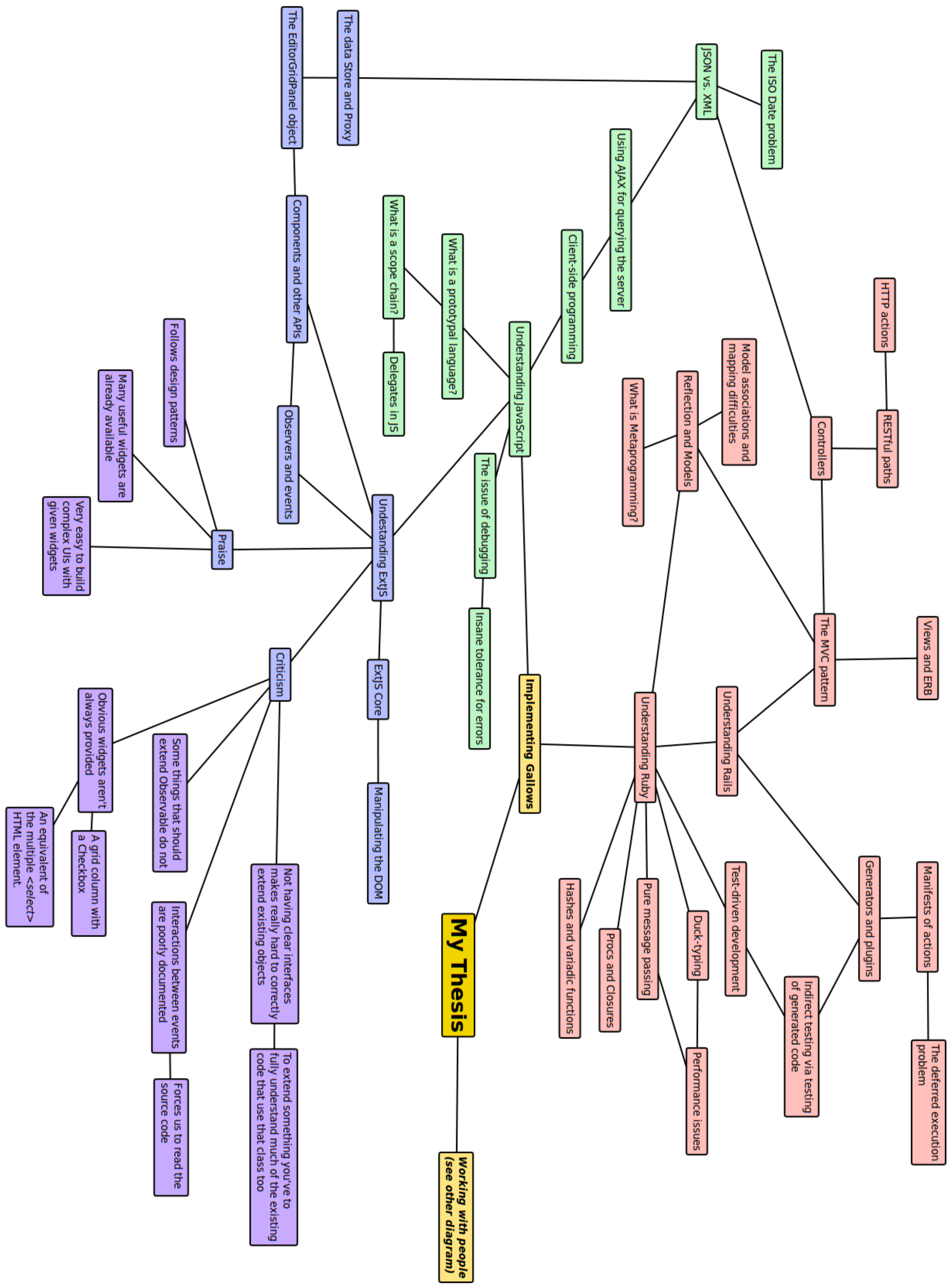

My Thesis
Implementing Gallows
Working with people (see other diagram)
Understanding ExtJS
Praise
Follows design patterns
Many useful widgets are already available
Very easy to build complex UIs with given widgets
Components and other APIs
The EditorGridPanel object
The data Store and Proxy
Observers and events
ExtJS Core
Manipulating the DOM
Criticism
Obvious widgets aren't always provided
An equivalent of the multiple <select> HTML element
A grid column with a Checkbox
Some things that should extend Observable do not
Interactions between events are poorly documented
Forces us to read the source code
Not having clear interfaces makes really hard to correctly extend existing objects
To extend something you've to fully understand much of the existing code that use that class too
Understanding JavaScript
What is a scope chain?
Delegates in JS
What is a prototypal language?
Client-side programming
Using AJAX for querying the server
JSON vs. XML
The ISO Date problem
The issue of debugging
Insane tolerance for errors
Understanding Ruby
What is Metaprogramming?
Reflection and Models
Model associations and mapping difficulties
Hashes and variadic functions
Procs and Closures
Pure message passing
Duck-typing
Performance issues
Test-driven development
Indirect testing via testing of generated code
Understanding Rails
The MVC pattern
Controllers
RESTful paths
HTTP actions
Views and ERB
Generators and plugins
Manifests of actions
The deferred execution problem

Figure 4.1: Conceptual map of this document (Gallows Implementation)

4.2. Working with people

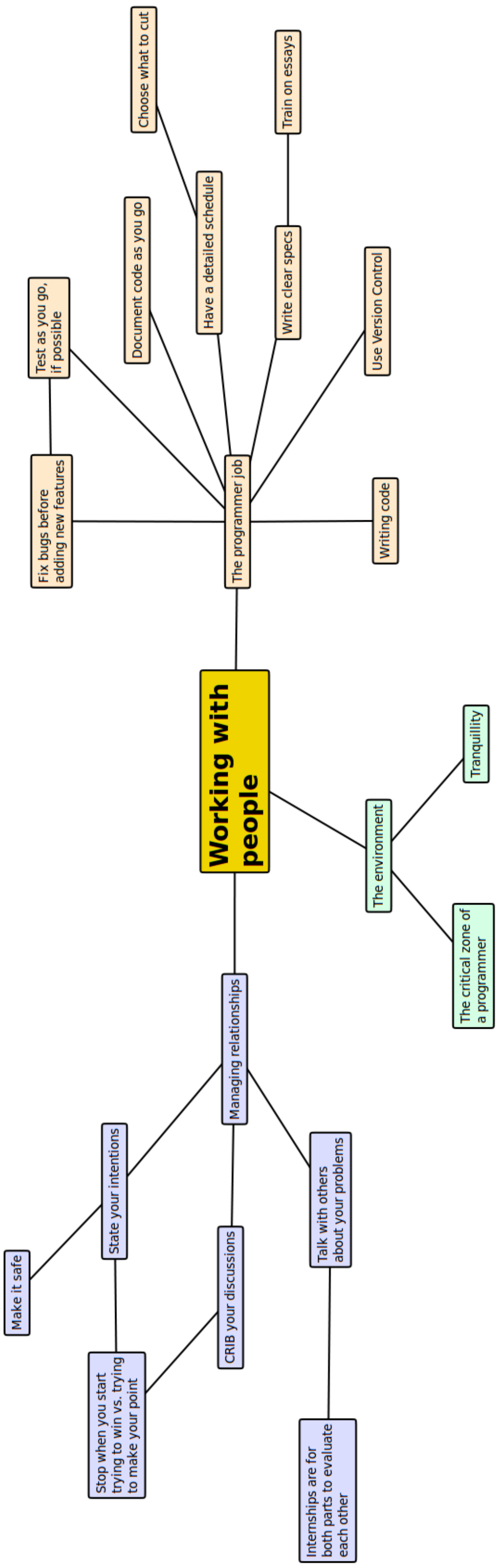


Figure 4.2: Conceptual map of this document (Working with People)

5. Introduction

Nobody reads documentation. That's an hard fact; mostly because, quite frankly, programmers are traditionally bad at writing. They may be good at writing for a compiler, which doesn't get fed up easily, but with humans they always take the wrong approach: a lot of incomprehensible acronyms and Star Trek references that no-one understands. Therefore, while writing this thesis, I asked myself if I could manage to make it slightly more interesting than the average two-hundred-page doorstop. If, by the end of this introduction, I've got an hook on your attention, maybe we're going somewhere.

Therefore, I asked me the classical marketing question: who is my audience? Bottle-glassed young nerds, more prone to poke at their keyboard with nicotine-stained pudgy fingers than reading an oh-so-boring thesis work, severe teachers in tweed jackets, or your average white-collar make-it-short tie-wearer?

The answer, perhaps surprisingly, is neither of them. This thesis is written for you. Yeah, you, the one sitting in front of this page. Really, that's not as obvious as it seems. Even if the technicalities are all here – this isn't an essay on the sex of echidnas, after all –, this work is meant to be enjoyable even to people with only little knowledge on the domain at hand.

“Nobody reads documentation. That's an hard fact.”

I tried to be clear and to explain things in an understandable manner. Even if you don't give a damn about how JSON interacts with an ExtJS Store class to populate an `EditorGridPanel` generated by a Ruby meta-class which introspects on database model to gather up associations between tables, it shouldn't matter. There should be enough insights and ideas interspersed in the text that you can still get something out of this. Why? To make it useful; because an internship is mostly and foremost a work of building experience. And this experience must be shared, so that the same old mistakes aren't repeated again and again.

Hence, I divided conceptually this thesis of mine in two distinct and independently readable parts. The first dives into the gory details of my project. This may be of interest to you only if you own, or plan to own someday, a degree in Computer Science.

If you're not interested in it, you could still find pleasurable what I say in the second part, which is about how to get on productively in your workplace as a programmer (but many things apply to other fields as well), about how to get your project on Rails, how to possibly end your work in schedule and even with most of it working! Woh-hoo!

So, in a sense, this is not conventional homework, reporting how I did this and did that, and then smithed the details in place. I always hated throw-away thesis from students that only wants to graduate with the least effort. This is a more open dissertation about how I got from the academic classes to the workplace, how I managed a small project, and how I survived the shock.

Keep on reading. The best has still to come.

6. Acknowledgements

I have to thank the whole of my family along with Erika, for the strenuous support they showed me in all these years, even when I was ready to give up and go to work in banana plantations in Cuba (why I didn't do that, still I don't know). They're the ones that most deserves my gratitude, and without them I wouldn't be here today. Thanks, people.

In addition, my gratitude encompasses also my best and greatest friends, which have been always available when I needed someone to help me or cheer me up. Other than my girlfriend, Josefine (the best woman in the whole world!), the list must start with the people which I learned from the most in these last years of university: Paolo Santi, Luca Vezzaro, Marco Trevisan, Carlo Maria Massimo, Filippo Paparella, Enrico Iori and many others I don't have the space or the memory to thank them here (but you're great, and you know who you are!). On with the list, my acknowledgements also to Egidio Fabris, Alessandro Dalla Vecchia, Allison Steenson, Alessandro Tognetti, Daniela Luvisetto, Andrea Zaupa, Elisabetta Soccodato and all those I always forgot to thank even though they're the best of the crew. Thank you for bearing with me.

I also have to thank people at Diginess for the nice and friendly space they reserved for me in their team. Working with them has been a pleasure I hope to continue to enjoy in the future.

“Programming without ethics is just that: a binary blob of 0s and 1s.”

Due to my strenuous support of free (as in speech) systems, my thanks also to Richard Matthew Stallman, Lawrence Lessig and Eben Moglen, whose work has always been mostly influential to me. They started all this, and without them I wouldn't have been interested much in computers; programming without ethics is just that: a binary blob of `0`s and `1`s, ready to turn out being a new `0x7C0`.

Last but not least, I want to thank all the professors of the Dept. of Maths and CS, which have been very influential in my upper-grade studies. I can't tell who's best, because all of them score high in my opinion. To make it short, you gave me much. Everyday I try and use that knowledge for what I hope is the best: teach others ethics on the workplace, work shoulder-to-shoulder with people without fearing getting my hands dirty, and never give up when the compiler throws dozens of lines of ugly C++ template errors at me.

Finally, this thesis is dedicated to my professor of Maths in high school, Marco Zoso, which always believed I could make it, even with my marks of 6 out of 10 in the aforesaid subject. To Marco: look at me, now that I did it, I can stand on the roof and shout “*I'm a golden god!*” like Russell Hammond in *Almost Famous*.

In fides,

[illegible]

This page has been intentionally left blank for layout purposes.

PART I: IMPLEMENTING GALLOWS

7. The scope of the project

The requirements agreed at the beginning of the project were abuot the complete analysis and implementation of a scaffolding system in the Ruby language, to be able to quickly generate, given a starting model, controllers and views for the Rails framework. The generated views were to use the ExtJS JavaScript framework in order to provide a nifty interface that would let the users get in-place editing of data (where possible), dynamic searches and updates, allow for mass destroy of items and enable nested editing of associated entities.

Great care was therefore taken to make sure that the developers who should modify by hand the generated views wouldn't need to alter the provided library nor, as far as possible, would need invasive changes to the existing models. That isn't to say it is possible to do that: one specific requirement set by the stakeholder was exactly the capability of customize the code as needed. To this aim, it was decided to make the whole subsystem an independent plug-in of the target application, activatable on demand but insulated by the rest of the environment as much as possible.

The goal has been simplicity of usage for both the developer and the end users. Ideally, once modified the generated configuration file, the final views and controllers should require little editing if at all.

7.1. The problem we're trying to solve

Doing data-entry in web applications is increasingly common. What once was done via questionable contraptions in Microsoft Visual Basic is now being moved to the cloud (usually, retaining the same ugly mess). This has some immediate benefits: more people can work concurrently, informations are easily accessible to whoever has a simple standard-compliant browser, and it's easier to guarantee what you create will survive earthquakes, WW3 and your favorite company's Wall Street crack-down. This to the mere expense of some more complexity.

The problem with developers is that they don't have much time at hand and, quite frankly, that creating forms on the web is plain boring. Visual Basic was easy: you took a widget, dragged it on you canvas, and you had it set up. Whereas, on the web, you've still to write by hand a lot of glue to check the data, send it to the server, store it away in a database, and then worry about SQL injections and such. What a pain.

Enter the Ruby on Rails world. Not only it provides a clean way to write all the glue, it also provides most of the glue itself. No more broken teapots in your kitchen. Writing a full-fledged web application can be matter of minutes – leave or take the design part.

The procedure of generating the views is usually called *"scaffolding"*

In the specific, there are some generators provided along with the framework that, if instructed correctly, will automatically build a model for you, create the needed tables in a database and set up table associations. They allow to check for data integrity, provide a controller able to receive and respond to requests via HTML or as a web-service and even build some default views you can apply some CSSs to. These may contain web forms for the CRUD[1] actions, already drafted for you. This is done via a single command entered on the console, and three-four minutes of editing.

1 Create – Read – Update – Destroy: these are the main actions that databases understand. Mapping these in one way or another in your application is usually desirable, even if some entities may miss the views for some of them.

The procedure of generating the views is usually called “*scaffolding*”. This is because they aren't final and they'll require manual intervention, but for testing the application and checking it works, they're already fine out of the box.

Therefore the request to improve upon the default Rails scaffold generator. I was asked to make a generator with the following features:

- It had to be a generator for Ruby on Rails;
- The views had to use ExtJS 3.0 (when the project begun, it wasn't yet released, and of beta quality);
- It had to be easily customizable;
- It had to be possible to edit also entities associated to the one currently displayed;
- The generated code had to require as few modifications as possible before being usable as production code;
- It had to be pleasurable to the sight.

At the end of the creation of the scaffold generator, I were to employ it to create all the controllers and the views for the company's flagship product, an e-commerce solution. My system was to be used to generate all the views of the administrative section – thus allowing to enter new products, categories, taxonomies, manage customers, orders and shipments, and so on.

To these requirements I identified some other desirable not-functional requirements:

- It should run on different platforms, and thus using as few browser-incompatible features as possible;
- It had to be localizable;
- The subsystem should be a plug-in, and not interfere with the functioning of the rest of the application.

7.2. Technologies involved

The technologies involved in the project will be tackled separately in their respective sections. It suffices to say here that the project used the Ruby and Javascript scripting languages, the first server-side and the latter client-side. Other than that, the pages and other artifacts generated by the scaffolder use well-known standards like HTML, XML, CSS, AJAX and JSON. They are the glue that keeps all together. They won't be treated here in detail, except for JSON, because prior knowledge is given as guaranteed.

7.3. Other related projects

There are a plethora of other generators for Ruby on Rails; listing them all here would be pointless. It suffices to say that the company was dissatisfied with the plug-in they were using up until the beginning of the project, which was ActiveScaffold. This was due to its inability to provide sensible customization options of the code it generated. More, it forced upon the developer some alterations in the behavior of the model which risked to be quite invasive.

A scaffold generator which used ExtJS was already in the works (called extjs-scaffold), however it wasn't employable as a basis of development for the following reasons:

1. It used ExtJS 2.0 instead of the new version;

2. It basically generated a unique huge page containing all the Javascript code needed to run the view, once for each entity you'd launch it for. This quickly would have led to duplicated code; to carry a modification to one of the views would have meant to change all the others manually;

3. No support for in-place editing, nor for the editing of associated entities;

4. Under the GPL license. Even if I cannot say I agree, the company did want a proprietary solution under a NDA-agreement. Therefore, they bought an ExtJS license (else, ExtJS itself would have been released to non-paying customers under the GPL license too) and asked me to create a closed-source implementation of the scaffolder.

8. Ruby and Rails

8.1. Ruby

I've always had a certain passion for programming languages; I find it amusing to see how different people come up with the strangest contrivances in order to express the same basic things, and of the beautiful ways they untangle gordian knots. In my peregrinations across different idioms and dialects, I was very happy to stumble upon Ruby, a scripting language that has been born in the first nineties by the hand and the mind of Yukihiro “Matz” Matsumoto.

Above all, it's the impression of cleanliness that makes Ruby a gem. Since the beginning, the whole focus of development has targeted ease of expression above performance. Ruby is purely object-oriented (nothing isn't an object!), and offers different shortcuts as useful syntactic sugar. This almost enables you to read a Ruby program in plain English out loud.

Inheriting most of its features from languages such as Perl, Python and Lisp, Ruby has all the things you may expect from a modern language: you need to write something out procedurally? It's there. Functionally? There. Design patterns? Built in the standard library. Dynamic type-checking? You bet it! The next version of the language, Ruby 2.0, should also improve greatly on the areas the language is weaker, namely poor performance during execution and the slowness of garbage-collection.

I'll proceed to dive into some of the Ruby foundations which are needed in order to understand the rest of my work. These may not be immediately clear without an introduction. I won't encompass all the features Ruby provides; readers interested may refer to [10] for further informations.

8.1.1. Pure message passing

As it stands, Ruby is purely object-oriented and uses pure-message passing. As you may suspect, it's also weakly typed. This means you have much more power at your hands, and power is something easy to misuse without some self-control. For example:

```
class CarEngine
  def start
    “Vroooom!”
  end
end

class NuclearMeltdown
  def start
    “Destroying planet Earth!”
  end
end

class Baffled
  def unknown
    “I'm not called 'start'”
  end
end

classes = [CarEngine, NuclearMeltdown, Baffled]
obj = classes[rand(classes.size)].new

puts obj.start

# This will output one of these:
=> “Vroom!”
=> “Destroying planet Earth!”
=> NoMethodError: undefined method `f' for #<C:0xb74747f0>
```

As you can see, nothing stops us from calling a certain method on a class even when we don't know its runtime type. This is the meaning of *true message-passing*: the sending of the message is tried at runtime, and there's no type-checking involved until it's finally delivered[2]. If things go bad, a *NoMethodError* exception will be thrown, and we will be able to rescue from it. However, this also means that we must be careful not to call the wrong method on the wrong object, or it could act in some unexpected way – like making planet Earth explode.

I've got to say that mostly, if variables are named consistently in your programs, this isn't a problem. Ruby and, as we soon will see, Rails in particular will enforce a large number of conventions which, if abode, will save countless hours of debugging.

2 Purists will assert that no type-checking is involved at all, even during delivery – this is quite true.

8.1.2. Functional capabilities

Some of the things that make Ruby really pleasurable are its functional features. That is, it's very easy to, say, print all the company names given a certain employee. For example:

```
@companies.collect(&:employee).each { |e| puts e.name }
```

Let's walk through this, because it'll allow us to understand most of the code of Gallows in just one line:

1. `@companies` is a reference to an instance variable, thus it's clear this code will be called inside a class method. Remember: in Ruby the first declaration means also allocation and instantiation. All instance variables are prefixed with a '`@`', class variables with two (e.g. `@@class_variable`).

2. We're calling the `collect` method on the `@companies` object, which arguably is of an `Enumerable` type. Of course, we can't be sure until we run the code, but that's not a problem most of the time. We're duck-typing[3], so we can expect it to be an `Array` or similar. Unit and functional testing, of course, will be extremely useful in order to limit the number of errors.

3. The `collect` method walks through an `Enumerable` object, and builds an `Array` with all the return values of the passed block, which is invoked once for each array element. A block is a closure, retaining the informations about the enclosing scope; as you may recall[4], a block can be saved and re-used later, with the guarantee that every variable referenced in the block will still be available at the point of invocation. The garbage collector will take care of not deallocating them until the block itself will be freed (given all other references were already dropped). The syntax `&:employee` is just a shorthand for: `{ |company| company.employee }`. The two syntaxes are equivalent.

4. For each element of the returned array, pass it to an anonymous block which takes an argument as a formal parameter, and print to the console the result of calling the method '`name`' on the actual parameter.

As a side note, just doing `@companies.each { |c| puts c.employee.name }` here is much more efficient, however the point of the example was exactly to show the syntax, so please bear with us. See also [10:pp. 53-57]

You can define methods using a block and, for example, invoking it for each element of an array, like this:

```
def method(array, param)
  # do something
  array.each { |element| yield param, element }
end

method(%w{uno due tre quattro}, 10) { |p,e| puts “#{p} -> #{e}” }
```

3 Duck-typing is just a simple principle: “*If it walks like a duck, and it quacks like a duck, then it is a duck.*”

4 Lisp and Scheme programmers will certainly be familiar with the concept of saving some code to run it later; in Ruby this is a `Proc` object, which is obtained from a block via the *lambda* operator.

8.1.3. Mixins

An important concept in Ruby is that you can modify at any time a pre-existing object or class (which, in turn, is also an object). This is done by adding methods to it or deleting others.

For example, to add a new method for Array, at any point in your code do:

```
class Array
  def ucase
    self.map { |e| e.to_s.upcase }
  end
end

puts [1, 'fo0', 5.23, “bAr”].ucase.inspect
=> ["1", "FO0", "5.23", "BAR"]
```

Given this great flexibility, Mixins come into play. They are just sets of methods which may refer to a `self`[5] object inside of them, which will take the correct value upon execution. Mixins are very handy; for example, even if Ruby doesn't support multiple inheritance directly, they can be used to say that a certain class supports a subset of features, and thus conforms dynamically to a certain interface (that is to say, that it will receive the messages you expect it to receive).

A common showcase of the usefulness of mixins is that of the `Enumerable` module. It is included in such classes as Array, Hash, String and many others. These “*includers*” just need to define some documented methods, namely `each` or `<=>`, the comparison operator, if they want to support also additional features like sorting (Hashes won't, for example).

Thus, without the need of a static delegate, and without the need to replicate code or to inherit from a certain class, mixins provide the foundations to share some code which will be injected at runtime in the right place.

8.2. Rails

Rails is a framework used to build rich web-applications. Its designers wanted to:

1. Abstract as much as possible from the underlying system, especially from different DBMSs, and avoiding hard-coding SQL queries which are so common in other frameworks.

2. Build something high level, which let's you define easily and quickly web forms and services – the most boring part of the work of a web programmer.

3. Enforce good design practices, especially TDD[6], and avoid the dirty mess most applications written in other languages quickly become (PHP seem to be quite up on the list on this).

4. Last but not least, have fun.

5 Like in Python, and unalike C++ and Java, the *this* reference is called *self* in Ruby.
6 Test-driven development: write your tests first, *before* the rest of the code!

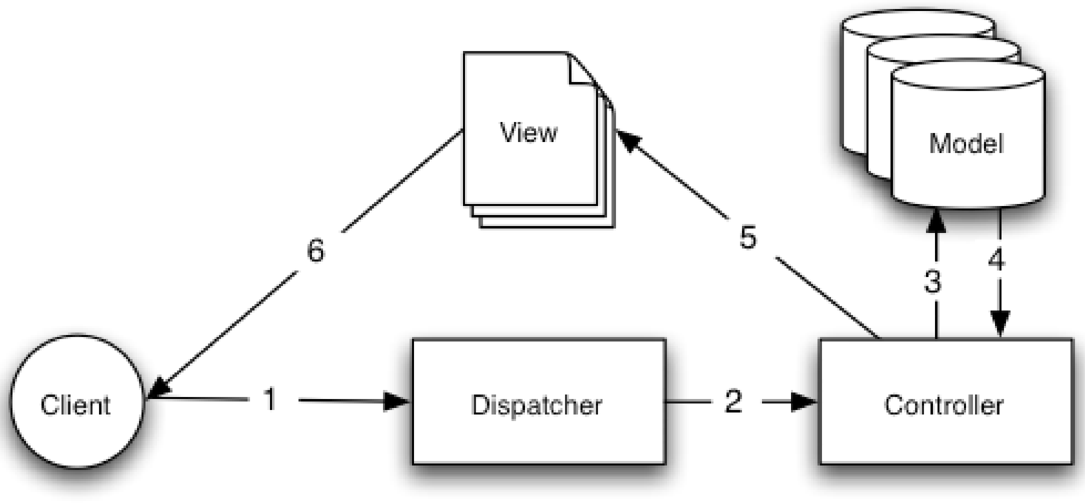


Figure 8.1: The MVC pattern

To these extents, they devised a framework which strictly adhere to the MVC pattern, and thus (mainly) divides your work in three parts:

1. Writing models to map database entities into your applications, also containing the business logic in order to preserve database integrity (unfortunately, this isn't managed at the DBMS level, which is a quite controversial point among Rails programmers). This may involve performing complex calculations on the entities themselves.

2. Writing controllers which to a determinate action respond doing something (typically, most entities will have CRUD actions associated) and showing up the result to the user via a view, depending on different factors. For example, they could show the result as HTML, as a PDF, as XML (for web-services), and so on.

3. Writing views (and their helpers), in order to present items to the user. Partial views will let us to reuse code as much as possible, rendering just one item of a list and repeating the same fragment – by inclusion – in a more complete layout.

8.2.1. MVC: Models

Models usually map directly entities which are present in the database, but may add new “ghost” entities that haven't a correspondence to a db table, or may add virtual[7] methods and functionality, mostly to ensure the data stays valid.

Relationships between entities are usually expressed by calling one of these methods at class level, which meta-generate the needed virtual attributes:

- `belongs_to :entity` → The current table has a foreign key for the table `entities` named `entity_id`.

- `has_many :entities` → In the table `entities` there is a foreign key called `this_class_id`.

7 In the context of models, we will use the term *virtual method* from now on to indicate a method which isn't a direct mapping of a database attribute. If you come from a C++ background, please accept the clash in terminology.

- `has_one :entity` → In the table `entities` there is a foreign key called `this_class_id`.
- `has_and_belongs_to_many :entities (:through => :assoc)` → This is the *many-to-many* association, via a join table called `assocs`. In the `assocs` table, there'll be the `this_class_id` and `entity_id` foreign keys.

Optionally, these associations may also mandate the behavior to keep when deleting or updating associated records. For example, to maintain data integrity they may call delete on cascade, or nullify foreign keys.

8.2.2. MVC: Controllers

A controller takes a request, which has been routed to map to a controller action, and performs some query or data manipulation on a model. Then, it returns a visual representation of a response.

For example, I may visit the `/products/11/edit` path, which invokes the `edit` action on a `ProductsController`, passing a `product_id` of `11` in the parameters. This action can respond in different ways depending on, say, the mime-type of the request. If we made the request in the form of `text/html`, it may render a HTML page with a form for editing this particular product.

On form submittal, the data may go to `/products/11` with a POST action[8]. It would then call the `update` method of `ProductsController` with a hash of the posted data. The controller would then try to update the model with the right data, eventually checking if the user has the permissions to do so. Then, it may render a view informing the user of some errors, redirect her back to the list of products, and so on, depending on the preferred course of action.

8.2.3. MVC: Views

Views are decoupled from controllers, and just represent the data prepared by them. A small collection of tags, alike those of JSP or PHP, are available to insert code into a page. The parser of these views is called ERB.

Views may use Helpers for more complex methods, in order to keep them as simple as possible (that is, as much HTML and as few Ruby code as possible). Moreover, views can include other views as *partials*. This maximizes code re-usage across different views.

They aren't limited to HTML; it is very well possible (and often done) to employ ERB also for XML, JSON, RSS, CSS, text and JavaScript files output.

8 HTTP supports four types of actions, even if the most used are two: POST and GET. The other two are PUT and DELETE. This is why we can query the same path but with different HTTP verbs and have the controller respond correctly for each of them.

9. Javascript and ExtJS

9.1. JavaScript as a prototypal language

ECMAScript, of which the most famous dialect is called JavaScript, is a client-side object-oriented scripting programming language, which has been created to run snippets of code inside the user's browser so to provide dynamic functionalities inside web pages. Its short syntax, ease of parsing, and expressiveness have made it the language of choice for the web. ECMAScript is an open standard, and its name is a compromise between the organizations involved in standardizing it back in 1997, especially Netscape and Microsoft. Brendan Eich, the creator of JavaScript, is on record as saying that "*ECMAScript was always an unwanted trade name that sounds like a skin disease.*"

[E]ven if JavaScript is object-oriented, it's also prototype-based.

Lately, given its massive support coming from browsers ensuring it will be a language well supported in future, and due to the critical mass of developers already familiar with the language, JavaScript has been adopted also outside the web arena, for example in the upcoming release of the open Desktop Environment called GNOME 3.0. Nowadays, interpreters for JavaScript are fast and reliable, and even building a new interpreter from scratch is feasible in an acceptable time-frame.

But what makes JavaScript unalike to many other languages around? We talked about Ruby before. What are the main differences from another object-oriented scripting language? It turns out that, even if JavaScript is object-oriented (though it can be used in a procedural fashion if one wants to), it's also **prototype-based**.

This means that there's no notion of "class" built-in in ECMAScript. Every object gets instantiated starting from a pre-existent one, and methods can be freely copied across instances. In practice, upon calling the operator `new`, all methods defined into the `prototype` property of the original object[9] are made available to the new one.

Even more intriguing, all objects act like associative arrays. For example, a "*startEngine*" method in a "*car*" object may be referred both as `car.startEngine` or as `car['startEngine']`. The last notation is the one used to access named values from associative arrays, which are just unordered sets of properties and values.

In Javascript, functions are first class citizens. The intrinsic nature of the language makes easy to copy, share, change scope or re-assign methods and functions.

9.2. Constructing an object

Objects in JavaScript have a built-in member in their `prototype` property called *`constructor`*. This is the function which the operator `new` invokes upon instantiation (an empty constructor is automatically generated). In some frameworks, access to members of a superclass is usually implemented via a *`superclass`* property of the constructor; ExtJS does this for us[10]. Having such a

9 That is, the original *function*. We'll use the term *object* from now on, since you're probably accustomed to the word coming from OO-programming but keep in mind that, at the core, it's just a function slightly on steroids.

10 If we had to achieve this in plain JavaScript, we'd have to explicitly call the method of the function/object we're extending, like in C++, where you can call a superclass method via static member resolution.

member lets us to delegate at run-time a method's execution to the superclass.

Each function inside a object has a 'this' reference passed. However, this isn't statically determined, as we'll see further: it is very well possible (and often done) to change the reference to the this implicit object before executing a method, thus altering the scope in which the function will run.

Needless to say, abusing this may render the code very hard to debug. Fortunately, a wise usage of this feature enables us to write better and easier-to-understand code.

Before continuing, let us make an example of an object or two:

```
var Dog = function (config)
{
  this.isAngry = config.isAngry;
};

Dog.prototype = {
  legs: 4
  ,bark: function ()
  {
    if (!this.isAngry)
      alert (“Whoof!”);
    else
      alert (“Growl!”);
  }
};

var labrador = new Dog ({ isAngry: false });
var pitbull  = new Dog ({ isAngry: true });
pitbull.bark = function ()
{
   Dog.prototype.bark ();
   alert (“I've got ” + this.legs + “ legs!”);
};

labrador.bark ();
pitbull.bark ();
```

As you can see, we're creating a Dog “class” (which is just a function), and copying the configuration from a hash passed as a parameter. Then we proceed defining the prototype which will be applied to all new instances of the Dog “class”, which contains a property and a method, bark.

Finally, we create two objects, override the bark method just for the pitbull object, and inside it we call the superclass method explicitly.

So, what would you expect it to print? Most people would answer:

```
“Whoof!”
“Growl!”
“I've got 4 legs!”
```

Which is wrong. It'll print two times “Whoof!”, and then “I've got 4 legs!”. This is because you've to manage things by hand, since JavaScript wasn't intended from the beginning to offer OO-related functionalities. To achieve the expected result, we must call:

```
Dog.prototype.bark.call (this);
```

inside `pitbull.bark ()`. This will set the correct object for the invocation; else, the `this` reference inside `Dog.prototype.bark ()` will be set to the `Dog.prototype` object. Confusing? You bet it is.

In the same way, inside the constructor of a derived object you've to take care to call the constructor of the superclass via `call` (or its closely-related friend, `apply`).

A common problem that makes things even more difficult in JavaScript, is that variables are function-scoped instead that block-scoped. That is, this snippet will print three times the letter 'c' (the `setTimeout ()` function just waits for the number of provided milliseconds before calling a certain function passed as an argument)[11]:

```
var list = [ 'a', 'b', 'c' ];
var item;
for (var i = 0; i < list.length; ++i)
{
   item = list[i];
   setTimeout (function () { alert(item); }, 1000);
}
```

If you want it working, you've to use a loop closure. Fortunately, ExtJS gives us already some funky methods to make easy to inherit from other objects, loop onto arrays, copy methods across instances and so on.

9.3. ExtJS and its features

ExtJS is a standalone JavaScript framework which stems from some prior work on YUI. It tries to target a plethora of different browsers and systems, and to provide a smooth and aesthetically pleasing interface that resembles mostly a normal desktop application.

A full-fledged set of widgets comes with the framework, including grids, charts, most form elements, box containers like those of Gtk+ or Nokia's Qt+, and so on. Data providers, readers and writers are available via `Ext.data.Store` proxies. At the kernel of the framework, there's a component named Core, which is responsible for manipulating and updating the DOM, but also of allowing one to extend objects emulating OO-inheritance, copying around properties, and in general covering the gap that leaves JavaScript behind in respect to fully object-oriented languages.

ExtJS makes using AJAX calls quite easy; the main data interchange format is JSON – but also XML is supported.

We'll talk more of ExtJS when we'll talk about the ways Gallows did need to extend its basic functionalities to bend it to its will.

11 This example courtesy of Jim R. Wilson.

10. Gallows

Gallows is a scaffold generator that uses ExtJS 3.0 to achieve CRUD actions on a editable grid, as well as on some associated controller views. The Gallows generator produces a grid view of records starting from a pre-existing model. The capability of modifying, creating and deleting items is achieved via AJAX, by setting up ExtJS 3.0 to use a JSON renderer.

It is necessary to run the generator two times: the first round it creates a stub of configuration file, which you can alter to achieve the wanted result in views. The second time, it generates a controller and the right JSON-based views.

The advantage of this approach is to allow you to get automatically some `join` capabilities across different tables/models, and to associate the right widget to the right field when more than one may apply. (Hopefully) sensible defaults will be auto-generated. The controller name (which can be namespaced) must match the model name.

10.1. Usage

As previously anticipated, using Gallows means going through some steps, which are summarized in figure 10.1 (in pseudo-UML: readability has been favored to the following of the standard).

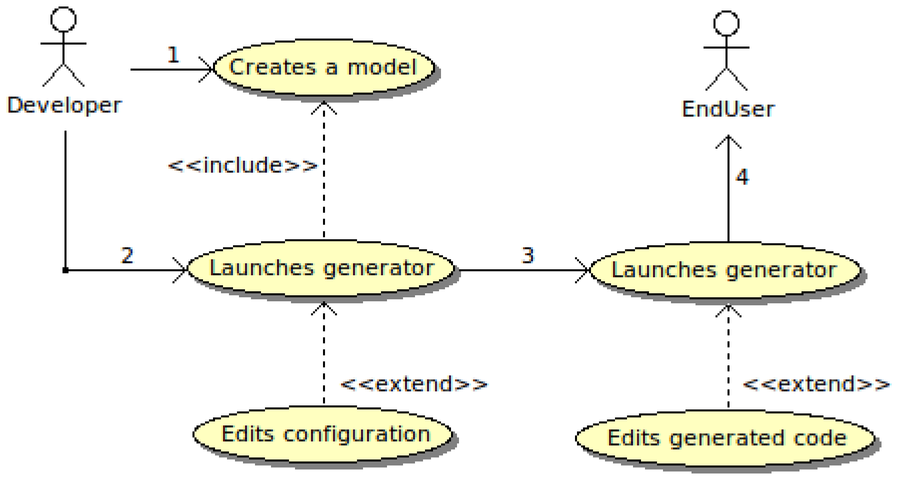


Figure 10.1: Conceptual work-flow with Gallows

The developer starts the process after finishing writing at least a stub of a model. In other words, the table must be in the database. After that, she invokes the generator a first time, which will return a YAML configuration file in the `config/gallows` subdirectory.

She proceeds editing the configuration file, removing fields she doesn't want to be included in the view, and eventually adjusting their types for virtual model methods. By default these will be kept with type “`method`”, not being able to infer their dynamic type.

Once done so, she runs the generator again, and now she gets a controller and all the views. She just need to fix the desired widgets in the views where necessary, and to enforce some checks in the JavaScript code. Then she checks if the layout (especially the new viewport) suits her needs, and if all virtual methods she wants to pass to the view are specified in the header of the controller.

Example: `./script/generate gallows NewControllerName`

This will create:

1. *First round*: a configuration file in `config/gallows`
2. *Second round:*
   - → a controller as you specified ;
   - → the `index.html.erb` and `layout.html.erb` files for views ;
   - → `new.js.erb` and `edit.js.erb` views for editing ;
   - → the related partial views: `_item_editor.js.erb`, `_record_store.js.erb`, `_i18n.js.erb`, `_vieport.js.erb`;
   - → shared JavaScript and CSS files in `public/*` if needed.

The developer should then proceed:

1. Editing the `_record_store.js.erb` file, in order to check that all fields she wants to manage are there with the correct type.
2. Editing the `index.html.erb` file, in order to change – if needed – the default widget used to display a certain value. Here it is also possible to add validations (ExtJS' `VTypes`) that will be used when updating or saving a record. Some basic extra validations are provided in the JavaScript library I wrote.
3. Doing the same for the `_item_editor.js.erb` file. When integrating Gallows into Digicommerce, I've separated more frequently used or mandatory fields from less important ones, and put them into different tabs. This was done so that the user finds the interface less cluttered. Here you can also call a factory which creates a window to manage entities directly associated to the current one.
4. Editing the `_viewport.js.erb` file. It is also possible to use the same file for all views by defining a unique layout. This greatly reduces code duplication.
5. Checking the generated controller, to see if it includes all the desired virtual methods and if all entities needed are in the eager-loading list. Now it's also the time to use the search conditions in the controller to further narrow the search results if they have to be manually applied to virtual methods.
6. Going get a snack and do relax! It's done.

10.2. Overall architecture

I did my best to keep the client-side and the server-side subsystems well compartmented. This allows for easy substitution at a later time, reducing rework times and new features' implementation. Also, I used standard Ruby methods where available, so that customizing the code and extending it should be trivial for a Rails programmer with some experience. The system as a whole should provide an interface most developers are already familiar with.

I maintained changes required to be carried on to models at a minimum, which was one of the major complaints with the `ActiveScaffold` previous code-base.

Thus, at a later time, it should be fairly easy to substitute the ExtJS part with another framework (for example, an extended jQuery), while retaining all the server side parts for responding via JSON to AJAX calls.

10.3. Plug-in initialization

Since Gallows will generate controllers employing its internal library for sending JSON data in response to AJAX queries, it is necessary to register the `.extjs` extension to be served with a new mime-type. We do this into `rails/init.rb`. We set the mime-type to “`text/html`”, due to some requirements imposed by ExtJS when receiving responses to multipart-data form submittals.

See http://www.extjs.com/deploy/dev/docs/?class=Ext.form.BasicForm for a full explanation.

10.4. The generator itself

The generator is the most easy part of Gallows. It takes just a model name, and by running a manifest in a “create” context it copies and instantiates all the views and the controller starting from ERB templates. It is mostly glue.

10.4.1. The manifest

Inside the generator, a manifest is defined. Its function is to register a series of actions whose execution will differ depending on the context in which the generator will be called. Valid actions are, for example: defining a directory, saying where a file must go, if it is a template or not, and so on. Templates will contain ERB code that will be parsed and instantiated depending on the generator command line parameters and context.

This means that all actions are deferred to after the manifest creation. The manifest is returned by the main generator method, and depending on the context in which the generator will run, a different action will occur for each defined instruction. For example, if the generator was called in the “`generate`” context, a “`directory`” instruction will create a new folder. Instead, in the “`destroy`” context, all actions of the manifest will be executed backwards, and a “`directory`” instruction will result in the removal of the specified folder (if empty).

What happens is that `Rails::Generator::Base`, our superclass, will be mixed-in with the module appropriate for the context; we don't need to know, when defining a `Manifest`, what it'll be.

10.4.2. Support for nested routes

I overrode some of the default actions, and added a couple new, in `lib/gallows/custom_generator_commands.rb`. Specifically, I added a check for the presence of ExtJS – which has to be manually downloaded by the user –, a check for the presence of the Silk icon set and new method to allow the definition of nested routes.

This method was needed to improve upon Rails default one which is used upon resource `creation`. The original function added routes with no notion of nesting when they were namespaced. Here's the code for `Create`; for the other types of commands, `Destroy` and `List`, they are just no-ops. The code reads as follows:

```
# takes the controller class name instead of
# a underscore name, and can be nested
def add_resource_routes(*resources)
  resource_list = resources.map do |r|
    resources.map { |r| route_for_nested_resource r }
  end.join("\n\n\n")

  sentinel = 'ActionController::Routing::Routes.draw do |map|'
  logger.route "map.resources #{resource_list}"

  unless options[:pretend]
    gsub_file 'config/routes.rb',
              /(#{Regexp.escape(sentinel)})/mi do |match|
      "#{match}\n#{resource_list}"
    end
  end
end

private
def route_for_nested_resource(r)
  list = r.split('::')
  recur = lambda do |list, i, parent|
    indent  = '  ' * (i+1)
    el      = list[i].underscore

    if i == list.size-1
      return "#{indent}#{parent.nil? ? 'map' : parent}.resources" +
             "#{el.to_sym.inspect}"
    end

    "#{indent}map.namespace(#{el.to_sym.inspect}) do |#{el}|\n" +
    "#{recur.call(list, i+1, el)}\n" +
    "#{indent}end"
  end

  return recur.call list, 0, nil
end
```

Finally, I had to force inclusion employing a private method (`include`). This is a Rails design issue, and is documented on the web – I used "`send`" in order to bypass visibility constraints:

```
Rails::Generator::Commands::Create.send  :include, Gallows::Generator::Commands::Create
Rails::Generator::Commands::Destroy.send :include, Gallows::Generator::Commands::Destroy
Rails::Generator::Commands::List.send    :include, Gallows::Generator::Commands::List
```

10.5. The configuration file

The YAML configuration file for a certain controller and its associated views is a standard YAML file, which defines:

- The fields/actions to show in the grid columns;
- The fields to use for a full-text search filtering;
- The columns that have to be displayed but cannot be altered by the end user;
- (Optional) A list of columns that require confirmation before committing changes.
- (Optional) A list of columns in the model to initially hide

The "`columns:`" list contains the list of columns of the final grid. The first level under "`columns:`" sets the header name (substituting underscores with whitespace).

Under that level, a list of mappings follows for each single column, since you could have more than one widget in a cell. The fundamental keys you want to check are:

- *field*: the name of the attribute or action to display; it's taken from the model.
- *type*: the type the data has in Ruby; useful mainly to correctly format Dates/DateTimes and such. The types available are: `:string`, `:text`, `:integer`, `:float`, `:decimal`, `:datetime`, `:timestamp`, `:time`, `:date`, `:binary`, `:boolean`, `:method`, `:image`, `:file` . Of these:
    - *:method* is different in which it generates stubs in the generated code ready for a custom AJAX request;
    - *:file* gives you the capability of uploading files to the server, and displaying a link to the uploaded one.
    - *:image* is like *:file*, however instead of a link it shows a thumbnail of the image.

The fields listed in "`editing_disabled:`" cannot be modified directly through the grid. This usually applies at least to the record `id` value.

"`require_confirmation:`" lists some fields that, if changed, will require the user to confirm a dialog before proceeding. This is used mainly for edits that may trigger a series of server-side actions (like sending an e-mail to a group of users, confirming an order, or changing a password).

The "`hidden_columns:`" list, if filled in, will contain some columns which will be available to the user, but won't be visible by default. They can be added to the view via the drop-down menu which is shown on hover on each column header.

10.6. How the data must be passed

At the core of Gallows, stands the `ExtjsHelper` class. It provides just four public methods that are called from the controllers, and then delegates the correct behavior to its internal methods. The UML class diagram for `ExtjsHelper` is shown in Figure 10.2.

Before proceeding to comment each of the provided methods, I'd like to show how a generated controller looks like. Take, for example, the following `CategoriesController`:

```
require 'gallows'

class Admin::CategoriesController < Admin::AdminController
  # these are the associated entities to load together with this model
   @@associated = [:child_nodes, :products]

  # put here the list of virtual methods of the model you want to serve
  # to the page via XML or JSON.
  @@virtual_methods = [:name]

  # GET /admin/categories
  # GET /admin/categories.xml
  # GET /admin/categories.extjs
  def index
    offset = params[:offset].to_i || 0
    limit  = params[:limit]

    # add here other :conditions if you need them
    options = Gallows::ExtjsHelper.search_conditions(params,
                            :include => @@associated)

    total_count       = Category.count(options)
    options.merge!(:limit => limit, :offset => offset)
    @category = Category.all(options)

    respond_to do |format|
      format.html  # index.html.erb
      format.xml   { render :xml => @Category.to_xml(:methods => @@virtual_methods) }
      format.extjs { render :json => Gallows::ExtjsHelper.to_json(@category,
                                              :offset => offset,
                                              :results => total_count,
                                              :include => @@associated,
                                              :methods => @@virtual_methods) }
    end
  end

  # GET /admin/categories/edit.js
  # GET /admin/categories/edit.xml
  def edit
    # Not used by ExtJS:
    # @category = Category.find (params[:id])

    respond_to do |format|
      format.js   # edit.js.erb
      format.xml  { render :xml => @test }
    end
  end


  # PUT /admin/categories/1.extjs
  # PUT /admin/categories/1.xml
  def update
    @category = Gallows::ExtjsHelper::update_from_params(params)

    respond_to do |format|
      format.extjs do
        render :json => Gallows::ExtjsHelper.to_json(@category,
                                              :include => @@associated,
                                              :methods => @@virtual_methods)
        end
      format.xml   { head :ok }
    end
  rescue => e
    status = case e.class
      when ActiveRecord::RecordInvalid, ActiveRecord::RecordNotSaved
        :ok
      when ActiveRecord::RecordNotFound
        :not_found
      else
        :bad_request
      end

    respond_to do |format|
      format.extjs { render :json => Gallows::ExtjsHelper.to_json(e.message, :success => false),
                            :status => status }
      format.xml   { render :xml => e.message, :status => status }
    end
  end

  # omitted methods: show, create, new, destroy.
end
```

I omitted the code for the `show`, `create`, `new` and `destroy` actions, since they are trivial to determine given the above example.

Let me comment this code a little bit further:

1. The `@@associated` member is an Array which specifies the entities you want to eager-load along with the current one. It is recommended you include each entity which appears in the

ExtJS data Store for this controller – that is, any entity in use by the views. This will greatly reduce the number of SQL queries needed when displaying a page.

2. The `@@virtual_methods` array specifies those methods of the model whose return values have to be passed along with the response. This allows you to fine-tune exactly what you want to include and what you don't need. All “normal” model attributes (that map directly to database columns) are passed on by default.

3. The index view is related to the main grid, and the `index` method's responsibility is to prepare the data. The HTML response would not need any data to work, since all the data is retrieved on a second call to `index` both done via AJAX and requesting a response in the `.extjs` format. Thus, an obvious optimization doable here would be to move the HTML response at the beginning, before any query has been made. I left it like this for clarity. The `index` method follows these steps:

    1. It gets the search conditions from the parameters. Please note how this works: it returns a normal Ruby `Hash` of conditions, and then uses it against a normal `ActiveRecord` model. I took some care to do this in this way, so that it's easy to re-use these conditions to further filter the results, for example when you've got some data which can't be retrieved directly during the query. A concrete example is that of localizations made via Globalize[12]. Since translated strings reside in different tables, it means that some important fields to be searched upon are virtual attributes in the models. In this way, it is possible to impose manually a search also on these attributes; if things go bad, this can be done at Ruby-code level – instead than via SQL queries.

    2. It does a first query upon the given search conditions; this allows us to correctly paginate the results by counting the number of records that match the query.

    3. It repeats the query, now limiting the number of results to the specified parameter, and also applying the proper offset.

    4. Finally, if the result is in the `.extjs` format, it calls `to_json` in order to prepare data for the response.

4. The edit view is trivial: it just renders the given view. The `.js` view, which displays the single-item editor in a ExtJS Window, doesn't require any object for ERB.

5. The update view has more strong error checking than that provided by default by Rails, but your mileage may vary. Apart from that, it's fundamentally just two lines:

    1. Updating an object via a call to `update_from_params`.

    2. Rendering the result to JSON.

12 http://globalize-rails.org/

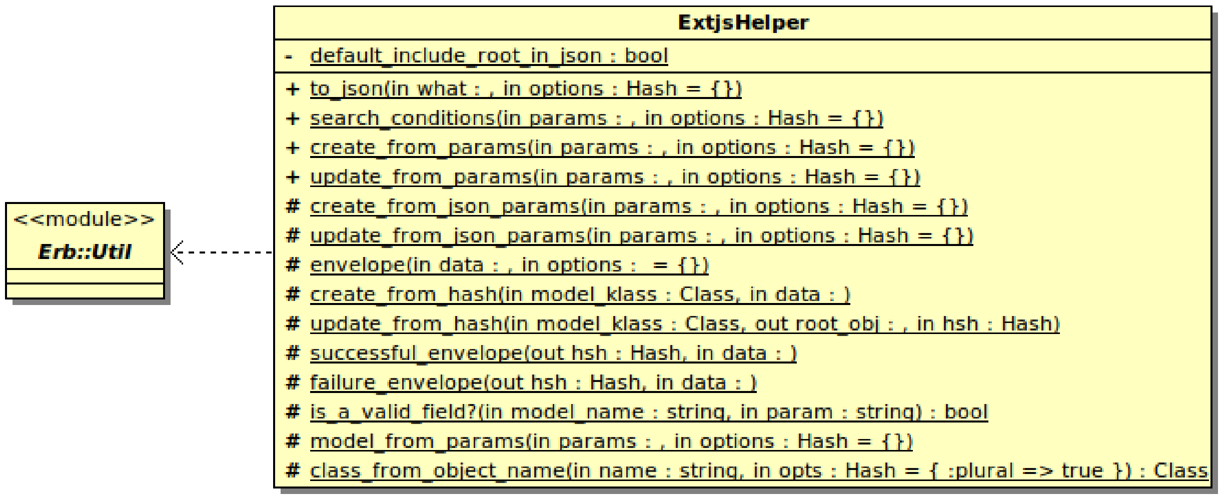

ExtjsHelper
- default_include_root_in_json : bool
+ to_json(in what : , in options : Hash = {})
+ search_conditions(in params : , in options : Hash = {})
+ create_from_params(in params : , in options : Hash = {})
+ update_from_params(in params : , in options : Hash = {})
# create_from_json_params(in params : , in options : Hash = {})
# update_from_json_params(in params : , in options : Hash = {})
# envelope(in data : , in options : = {})
# create_from_hash(in model_klass : Class, in data :)
# update_from_hash(in model_klass : Class, out root_obj : , in hsh : Hash)
# successful_envelope(out hsh : Hash, in data :)
# failure_envelope(out hsh : Hash, in data :)
# is_a_valid_field?(in model_name : string, in param : string) : bool
# model_from_params(in params : , in options : Hash = {})
# class_from_object_name(in name : string, in opts : Hash = { :plural => true }) : Class
<<module>>
Erb::Util

Figure 10.2: ExtjsHelper class diagram

Now, let's explore the `ExtjsHelper` class. Of the four public methods you can see in Figure 10.2, `to_json` is the most important. It will prepare the response for ExtJS putting it into an envelope. An example of the envelope format is given in section 10.6.1.

Let's get a quick tour through these library functions, in order:

1. `to_json`: This method forces the `ActiveRecord::Base.include_root_in_json` property to `false`, for easier management of JSON responses by ExtJS. It will have cure to set it back to the previous value at the end of the call (*note for maintainers*: put this in an “`ensure`” block). It will put the data passed as a parameter in an envelope through a call to the corresponding protected method.

2. `search_conditions`: given the hash of the request parameters, it will build a hash of options to be passed to the various `find*` methods of the current model class. All searches are performed through a SQL “`LIKE '%query%'`” statement (properly sanitized to avoid SQL injection attacks). Extending this to use something like Xapit, Ferret or similar fulltext search engines should be matter of changing a couple of lines in the source. The returned hash includes a `:order` item, and can be further manipulated in the controller.

3. `create_from_params`: given the parameters' hash, it just redirects the call to `create_from_json_params` if the response must be in pure JSON (when `params[:format] is 'extjs'` and `params.has_key? 'json'`). Else, if we must return JSON as text/html, we have to first escape the return value of `create_from_json_params` because of HTML entities. Finally, if we don't have to return JSON, we return a standard Ruby object to the caller, after calling `update_from_hash` with `model_klass.new` as the second parameter.

4. `update_from_params`: it behaves exactly like `create_from_params`, except it first tries to retrieve the object to update and raises an `ActiveRecord::RecordNotFound` exception if needed.

5. `create_from_json_params`: gets the `model_from_params`, decodes the JSON data via `ActiveResource::Format::JSONFormat.decode`, and then delegates `create_from_hash` the object construction.

6. `update_from_json_params`: follows the same structure of `create_from_json_params`, retrieving one or more objects first. If we're mass updating some records (support for mass updates is already built-in Gallows, if needed), a lambda function called “`find_and_update_object`” delegates `update_from_hash` for each member of the passed array.

7. `envelope`: checks the passed parameters; if `:success => true` it builds a `successful_envelope`, else a `failure_envelope` with the passed data.

8. `create_from_hash`: delegates `update_from_hash`, using `model_klass.new` as the root object.

9. `update_from_hash`: updates each association defined in the model if it's also defined in the hash items – or any other virtual method for which the `root_object.respond_to?`. It also normalizes to arrays all single-item values passed to setters of `has_many` associations. Of course, it is left to the developer to set the right methods as `protected_attrs` in the model; especially authentication-related or destructive ones.

10. `successful_envelope` and `failure_envelope`: they just add two root objects to the JSON response: a `results` value, with the number of results, and a `json` one, with the actual data.

11. `is_a_valid_field?`: given a model_name, tries to “constantize” it to a model's `Class.` Then, it checks if the given field name is a column of the model.

12. `model_from_params`: returns a `model_name` and `Class`, given the request parameters.

13. `class_from_object_name`: given a string, tries to transform it in a `Class` of the same name (eventually once singularized it).

A sample of what happens when a index page is asked to the server is shown in Figure 10.3. At first, a HTML page is returned, containing the JavaScript code to initialize ExtJS, and to create the main grid, store and viewport.

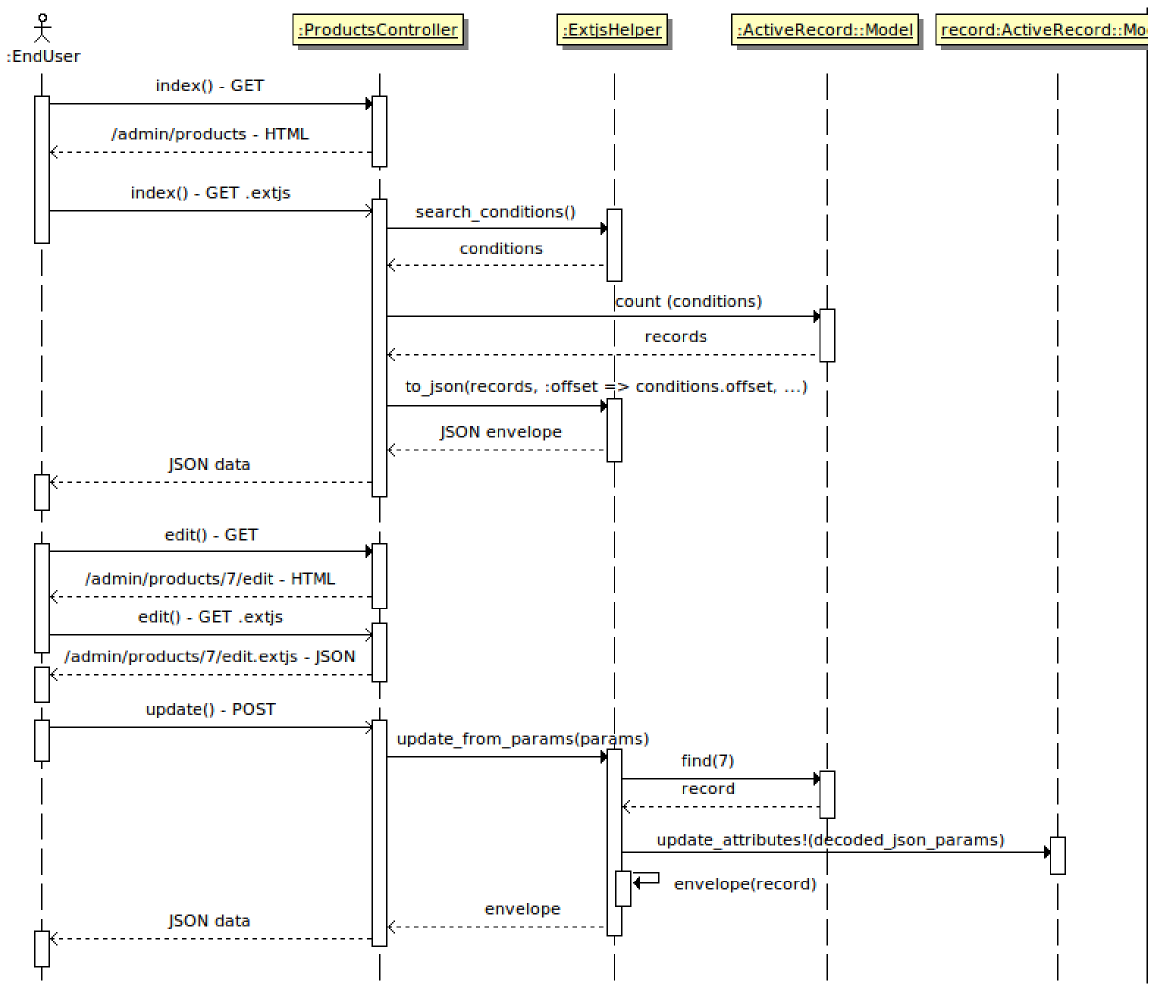


Figure 10.3: Updating an item - sequence diagram

Then, a JSON query is made from the store associated to the main grid to the same controller, which may filter the results based on a `_query` parameter, and order them by the field in the `_sort` parameter, in the (ascending o descending) `_order` direction. Pagination is achieved by sending the `_offset` and `_limit` parameters.

If the user requests to edit a single row, an AJAX GET request is made to the controller, which triggers the edit action. This injects some JavaScript code in a unique namespace (via a random-generated ID) inside the HTML page, and executes it. This code creates a ExtJS window and shows it to the user, employing the form elements put in the `_item_editor.js.erb` file. The developer will have taken care to check that each widget is the right one for the goal it has to achieve – for example, description fields, while being strings, may need HTML editors instead of simple `<textarea>` fields.

When the form is submitted, the element is updated, and the returned response is parsed from an hidden `<iframe>` to a fake XHR object. See 10.8.4. for more details. Then, the whole grid is reloaded from the store – it isn't computationally optimal, but it ensures there are less problems with other people editing the same data. However, note that during the scope of the project no real locking mechanism has been provided since not required by the stakeholder.

10.6.1. The envelope

Each JSON response to ExtJS need to have some fields that tell us:

1. if the request was successful or not;
2. the number of results – this is useful mainly for pagination;
3. an array of results.

For example, a valid response may be:

```
{"results": 1, "json": [
  {"position": 3, "products": [
    {"position": 8, "manufacturer_id": null, "created_at": "2009/09/03
14:52:16 +0200", "youtube": null, "updated_at": "2009/09/03 14:52:25 +0200",
"published": true, "deleted_at": null, "attachment": null, "weight": null,
"the_parent_record_id": "10", "id": 9, "free": false, "created_by": null,
"approved": true, "updated_by": null, "sku": null, "name": "Feeney, Hyatt and
Wiegand", "scribd": null, "meta_product_id": 1, "default_image":
"/var/www/digicommerce/public/upload/product/default_image/9/product_example.
jpg", "hilight": true}
  ], "created_at": "2009/09/03 14:52:10 +0200", "updated_at": "2009/09/11
15:15:10 +0200", "deleted_at": null, "id": 10, "created_by": null,
"child_nodes": [
    {"position": 1, "created_at": "2009/09/03 14:48:54 +0200", "updated_at":
"2009/11/08 23:08:13 +0100", "deleted_at": null, "id": 4, "created_by": null,
"updated_by": null, "parent_id": 10, "name": "corrupti", "image": null},
{"position": 3, "created_at": "2009/09/03 14:52:10 +0200", "updated_at":
"2009/09/11 15:15:10 +0200", "deleted_at": null, "id": 6, "created_by": null,
"updated_by": null, "parent_id": 10, "name": "qui", "image": null}
  ], "updated_by": null, "parent_id": 0, "name": "alias", "image": null}
], "offset": 0, "success": true}
```

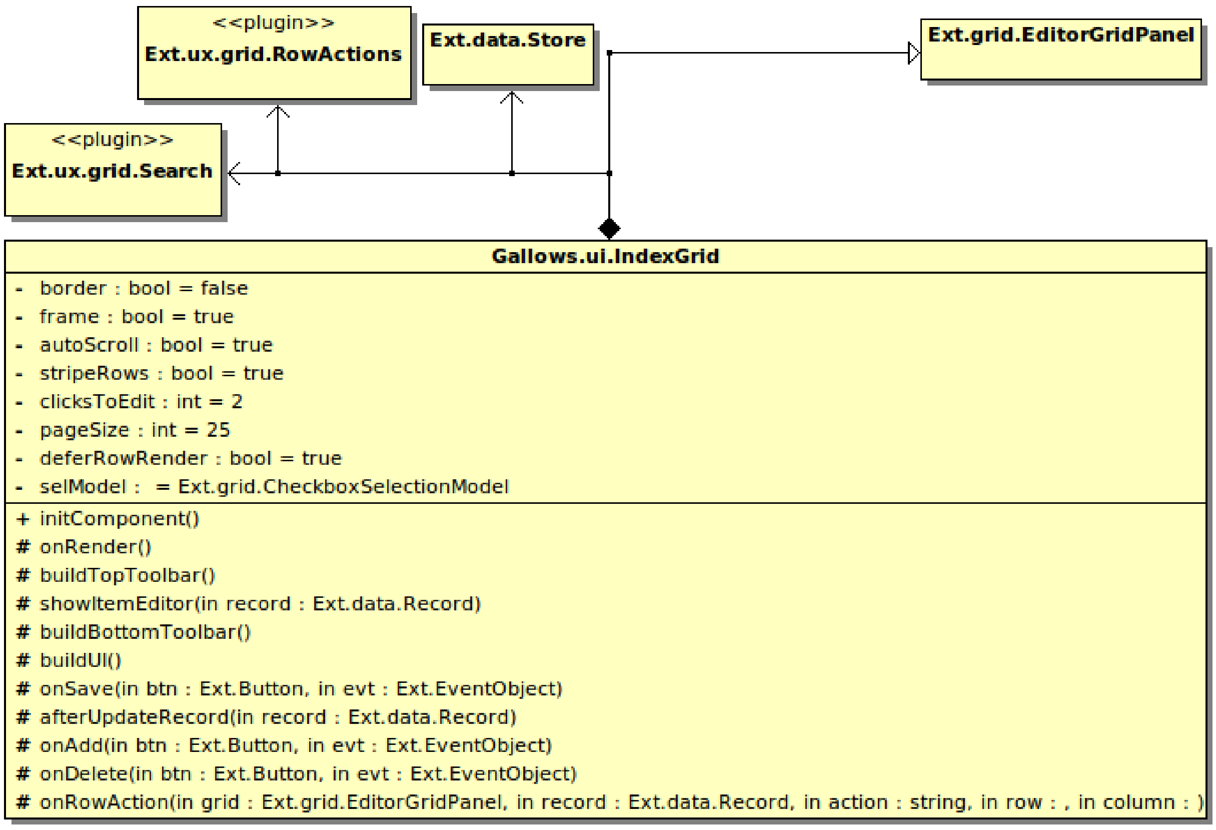
I've inserted a couple of ends of line to make it clearer; smell the love. However, for a machine this is surprisingly easy to parse. I've put in bold red the envelope itself; “`json`” is the name of each response's JSON root. In regular orange, you've got the first level of entities (a category). In dark-yellow italic, you've got the second (nested) level of entities: associated products and child categories. Needless to say, it may be desirable not to return all products for each category – a potentially expensive operation –, but allowing for a separate query when the need arises. In this case, products are in the eager-load list inside the `CategoriesController`.

10.6.2. The ISO date problem

Unfortunately, the JSON specification lacks of a standard or normative way to express dates. It would seem that the best way to exchange dates would be via ISO 8601[13]. Problem is, Rails and ExtJS have different ideas on what the standard should be. I ended up forcing `"dateFormat": "c"` on the ExtJS data Store columns that expected a `DateTime` object, in order to ensure compatibility.

13 http://www.iso.org/iso/catalogue_detail?csnumber=40874

10.7. The main grid



<<plugin>>
Ext.ux.grid.RowActions
Ext.data.Store
Ext.grid.EditorGridPanel
<<plugin>>
Ext.ux.grid.Search
Gallows.ui.IndexGrid
- border : bool = false
- frame : bool = true
- autoScroll : bool = true
- stripeRows : bool = true
- clicksToEdit : int = 2
- pageSize : int = 25
- deferRowRender : bool = true
- selModel : = Ext.grid.CheckboxSelectionModel
+ initComponent()
# onRender()
# buildTopToolbar()
# showItemEditor(in record : Ext.data.Record)
# buildBottomToolbar()
# buildUI()
# onSave(in btn : Ext.Button, in evt : Ext.EventObject)
# afterUpdateRecord(in record : Ext.data.Record)
# onAdd(in btn : Ext.Button, in evt : Ext.EventObject)
# onDelete(in btn : Ext.Button, in evt : Ext.EventObject)
# onRowAction(in grid : Ext.grid.EditorGridPanel, in record : Ext.data.Record, in action : string, in row : , in column :)

Figure 10.4: Class diagram for the IndexGrid widget

The main grid provides most of the functionalities needed by the pure read-only part of the application: it shows a list of items and the actions needed for their editing, be it in-place or via a separate pop-up window. See for example Figure 11.3 at page 49.

It has got its own `Ext.data.Store` linked, which acts in a RESTful way, and queries the controller for the type of entities to be displayed. This very grid, with its store overridden, is also used to provide editing of associated entities, as described in section 10.11.

Due to the priority in which the plug-ins, the columns and their components are instantiated[14], the grid is defined to explicitly call a `gallowsInit ()` member function in `Columns` put into the column model, if present. The grid object itself is passed as a parameter to this function, since `Columns` don't behave like `Ext.Components` (for some obscure design reason taken by ExtJS developers). This is needed so the column has a notion of its “container”.

This function is then used to register for some events, like click ones. Of course, manipulating directly the grid somewhat breaks code encapsulation and rises coupling – but we know that grid columns can only be inside a grid, so this isn't a particular problem.

Moreover, this is the only way to manage clicks in a cell at a column level, instead than at grid level – which would have brought coupling to stellar levels and probably would have required some down-casting too. Employing this method, while not particularly elegant, allows for keeping more

14 *Before they're rendered, some fields or other members may be undefined, but we may need to do something in-between construction, building and on-screen rendering.*

code in a single-place. This contributes to make it easier to read and maintain.

10.8. Custom ExtJS widgets

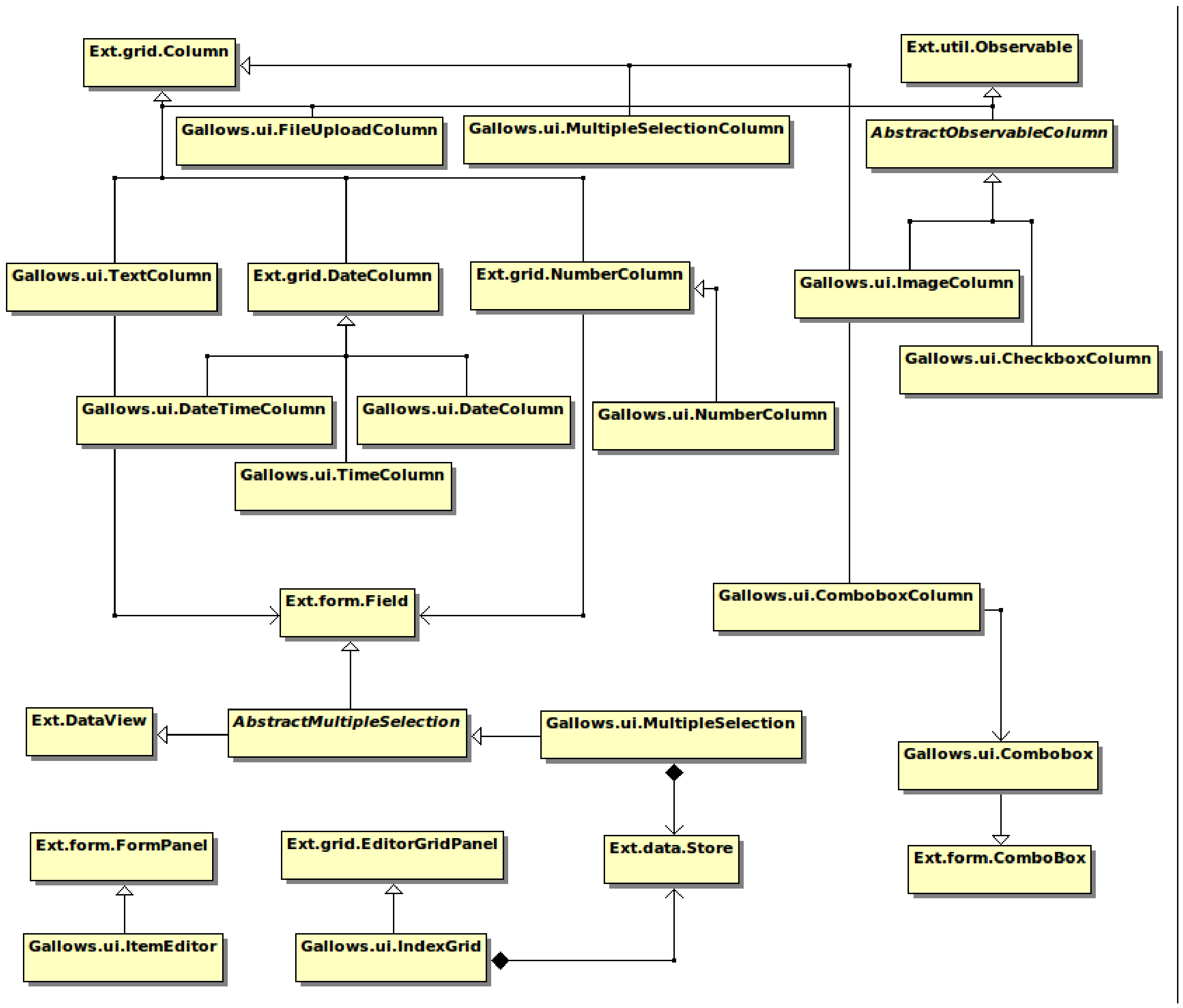


Figure 10.5: The custom widgets implemented for ExtJS

One of the first problems which arises when extending JavaScript object in ExtJS, is that you don't know what you have to re-implement. What should be a “base abstract class” have no notion of abstract methods. You just have to do a wild guess, implement the methods that looks as they should be redefined, and cross your fingers. This mostly works, until you trigger some strange function which in turn fires back with a method in the base class that needed to be redefined. Put into the pot the difficulties related to properly debug JavaScript code, and you've understood why this has been so much a source of trial-and-error for me – and of countless hours thrown away.

Sometimes, ExtJS uses some internal and undocumented property (like the infamous “`isColumn: true`” property in things that want to behave like a Ext.grid.Column) to assert that a certain object behaves in a certain way. This instead of, e.g., checking the prototype scope chain or using the `instanceof` JavaScript operator. I had to read hundreds of lines of code before getting out of some of these mystique errors. Other experienced plug-in authors around the web do use the same private properties to fool ExtJS in working as they want. I can't say I like it (it's bound to break if they change some internal representation, sooner or later), but for now it's the only way to make it work.

See the footnote at page 51 for an example.

Over the course of the project, I ended up implementing a long list of custom widgets. Most of them override just few functions from the ExtJS original ones, however in a couple of cases (most notably: `IndexGrid`, `CheckboxColumn`, `ComboboxColumn` and `MultipleSelection`), this required extensive changes and re-implementation.

A class diagram showing up how everything fits together on the ExtJS side of things, is shown in Figure 10.5.

Debugging is so hard, because the only effective way (let me rephrase it: the *only-way-full-stop*) to see where your code is failing is by using the Firebug Firefox extension. While quite a nice piece of code, you end up wondering what's happening more often than is necessary; it's slow, conditional breakpoints aren't implemented properly – mostly they don't work at the time of this writing, going up the scope chain of JavaScript object in a tree structure makes you crazy above the third level of nesting, the DOM has a lot of properties you don't care about and that should be hidden, etc.

Overall, I don't feel any shame in stating that I resorted more than once to put "`alert('something')`" in my code. We developers are in a desperate need for serious JavaScript debugging support. And wait! If in Firefox more or less debugging works, it isn't guaranteed it will when using the most feared nightmare of each of us web developers: Microsoft Internet Explorer. Try to debug your code there, and then come back to me...

10.8.1. Emulating multiple inheritance

Sometimes it'd be useful to allow for multiple inheritance in JavaScript. However, this is totally unsupported and left up to you, even with ExtJS. One compelling case in which this is particularly useful, is to make a certain widget extend some other class overriding just some limited functionality, and also make it inherit from an Observable object (from the Observer design pattern).

To solve this, I bent and twisted ExtJS provided methods to my will. For example:

```
Ext.ns ('Gallows.ui', 'Gallows.data');
Ext.ns ('Ext.util.Format');

var AbstractObservableColumn = Ext.extend (Ext.grid.Column, {});
Ext.applyIf (AbstractObservableColumn.prototype, Ext.util.Observable.prototype);

Gallows.ui.CheckboxColumn = Ext.extend (AbstractObservableColumn,
{
    constructor : function (config)
    {
        Ext.apply (this, config);

        this.addEvents ({ click: true });
        if (!this.id) this.id = Ext.id ();
        this.scope = this;

        Gallows.ui.CheckboxColumn.superclass.constructor.call (this, arguments);

    } // ~ constructor

    ,gallowsInit : function (grid)
    {
        this.grid = grid;
        this.grid.on ('render', function ()
        {
            var view = this.grid.getView();
            view.mainBody.on ('mousedown', this.onMouseDown, this);
        }, this);
    }

    ,onMouseDown : function (e, t)
    {
        if (this.readonly == true || this.isEditable === false) return;
        //alert (t.className);

        if (t.className && t.className.indexOf ('x-grid3-cc-' + this.id) != -1)
        {
            e.stopEvent ();
            var index  = this.grid.getView ().findRowIndex (t);
            var record = this.grid.store.getAt (index);
            record.set (this.dataIndex, !record.data[this.dataIndex]);
            this.fireEvent('click', this, e, record);
        }
    } // ~ onMouseDown


    // OTHER METHODS...

};
```

Here, I start creating a temporary variable "`AbstractObservableColumn`", which is a function/object that adds the `Column` capabilities to an empty object (`{}`).

Then, I proceed copying all those methods of `Observable` that aren't already defined into `Column`. The order in which this should be done is debatable, and should be assessed from time to time; this is okay for what I wanted to do.

The call to the `superclass` constructor will initialize just `Column`, since the `constructor` of `Observable` has not overrode that of `AbstractObservableColumn`. If I needed to make sure also the `Observable` constructor was called, I should have called it explicitly.

In the constructor, now I can add the "`click`" event listener, and use it normally inside the widget instead of passing every time through the (ugh!) Grid column model, which is the method ExtJS mandates. The `gallowsInit()` function is called by my `IndexGrid` constructor if present, and emulates the behavior of ExtJS `Components` – that is, it passes itself as an argument to this function. This is needed to circumvent initialization order problems; I'm still not sure why ExtJS developers didn't devise such a capability in the first place.

10.8.2. Emulating blurring

In-place editing inside a `EditorGrid` is done via the ExtJS Editor class, which wraps a HTML form widget and de-activates when the `onBlur` event is fired. Then it takes care to perform an update on the `Store` with the value inside the widget (and the Store performs an AJAX call to the server to update the record).

Unfortunately, the `onBlur` event is available only on standard HTML form elements, and there was no easy way to tell the editor to deactivate upon the receiving of some other event.

Since I also needed the same widgets (without the `Editor` wrapper) to be re-usable for the single-item editor form, which is put inside a separate window on a “`edit`” action on a `Row`, I wanted those widgets to fire an “`blur`” event when clicked outside of them. I didn't need also to manage keyboard events for firing `blur`, given I wanted this behavior to apply just to the multiple-selection widgets I instantiated – they are usable only with the mouse, at this time.

To solve this problem I delegated the '`click`' event on the whole HTML document for the blurring of these widgets. Then I overrode the click event inside my multiple selection custom widget not to propagate the click up on the event chain.

With a few more details, I had this to work: if the multiple selection is displayed and you click inside it, the click is managed by the widget to change its internal state. For example:

```
Ext.ns ('Gallows.ui', 'Gallows.data');

if (Ext.isEmpty (Gallows.i18n))
    throw "Cannot load the internationalization file. Check your includes.";

var AbstractMultipleSelection = Ext.extend (Ext.form.Field, {});
Ext.apply (AbstractMultipleSelection.prototype, Ext.DataView.prototype);

Gallows.ui.MultipleSelection = Ext.extend (AbstractMultipleSelection,
{
    // ...OTHER METHODS HERE...
    // inside the constructor:
    //   this.addEvents ("blur");

    ,onBeforeShow : function ()
    {
        Ext.getDoc ().on ('click', this.blurIfOutside, this);
    }

    ,onBeforeHide : function ()
    {
        Ext.getDoc ().un ('click', this.blurIfOutside, this);
    }

    // We need to fire a blur event when clicking outside
    // this element, because it's what the Ext.Editor expects
    // us to do in order to stopEditing (). However, since
    // this is a div, a "native" blur event can't be set.
    // Therefore, we do this manually.
    // private
    ,blurIfOutside : function (ev, el)
    {
        if (!this.isVisible ()) return;
        var target = Ext.fly (el);

        // Bubbles up 'til the main document or window
        while (target !== null)
        {
            if (target == this.el)
                return;
            target = target.parent ();
        }

        // If here, this is the main document, so we blur:
        this.fireEvent ("blur", this);
    }
};
```

As you can see, I took care to delete the closures associated with the widget upon hiding, and to re-create them on show. Thus, no leaks should occur at least as far as blurring for multiple selections are concerned.

10.8.3. Returning field names the Rails way

One of the problems I faced was with the management of the standard Rails field names when outputted for a form. As you may know, Rails generates fields in the form with names like this:

```
artist[lastname]
```

…so that when in the controller you call `@artist.update_attributes!(params)` all the right magic works, since `params` is a `Hash` in the form:

```
{
  offset => 0,
  _method => 'put',
  artist => {
    firstname => 'James',
    lastname => 'Hetfield'
  }
}
```

However, ExtJS doesn't like these kind of parameter names inside `Ext.form.Combobox` and `Ext.DataView`, just to name two, since they use an `XTemplate` to render a list of items. The standard regular expression which is used to catch parameters inside a template allows only for digits, letters, '`-`' and '`#`' (plus a little bit of other magic[15] you really don't want to read).

It turns out that you can override the '`tpl`' configuration parameter to use `XTemplate` syntax for in-line evaluation. For example, for a `Combobox`, you could move from the default to:

```
  '<tpl for="."><div>{[values["' + this.displayField + '"]]}</div></tpl>'.
```

And there you go, problem solved – hopefully. However, due to some way `data.Store` works in ExtJS, such a change wasn't easily applicable to Gallows and would have required extensive changes to the code-base and the generator itself – and time was running out. So I leave this here as a testament for whoever will have to maintain my project, so they can create a branch of Gallows and know where to start working.

10.8.4. The issue with file uploads

In the middle of development, I discovered I had left out from my prior design a fundamental component: file uploads. To be more precise, I had in schedule their implementation, however I deferred it and then discarded the problem as easily solvable up until I had to tackle the issue.

The problem is, I had coded all my generated controllers, libraries and so on to rely exclusively on JSON AJAX calls. However, it is simply not possible to do so in the case of file uploads: you can't access any write-property (just read-only ones) of form file upload fields, and they have to send data via POST, possibly using a specific mime-type in order to avoid problems (`multipart-data`). The form submit action would result in a page reload, something we don't want.

ExtJS plays relatively nice, and allows us to mask a **response** to a normal POST request of a form submittal as a fake XHR one (XHR is the short form for `XmlHttpRequest`, the function JavaScript uses to post and return data via AJAX).

The trick is to use an hidden `<iframe>` in HTML as the target of the form action, and then parsing the returned data as JSON (which has to have a `text/html` mime-type), then building a vanilla XHR object client-side.

10.8.5. Implementing a multiple selection widget

As already stated in section 10.8., Custom ExtJS widgets, one of the main complaints I have with ExtJS is that you don't exactly know what to override in subclasses of supposedly "abstract" interfaces, such as `Ext.form.Field`.

15 http://extjs.com/deploy/dev/docs/source/Template.html#cfg-Ext.Template-re

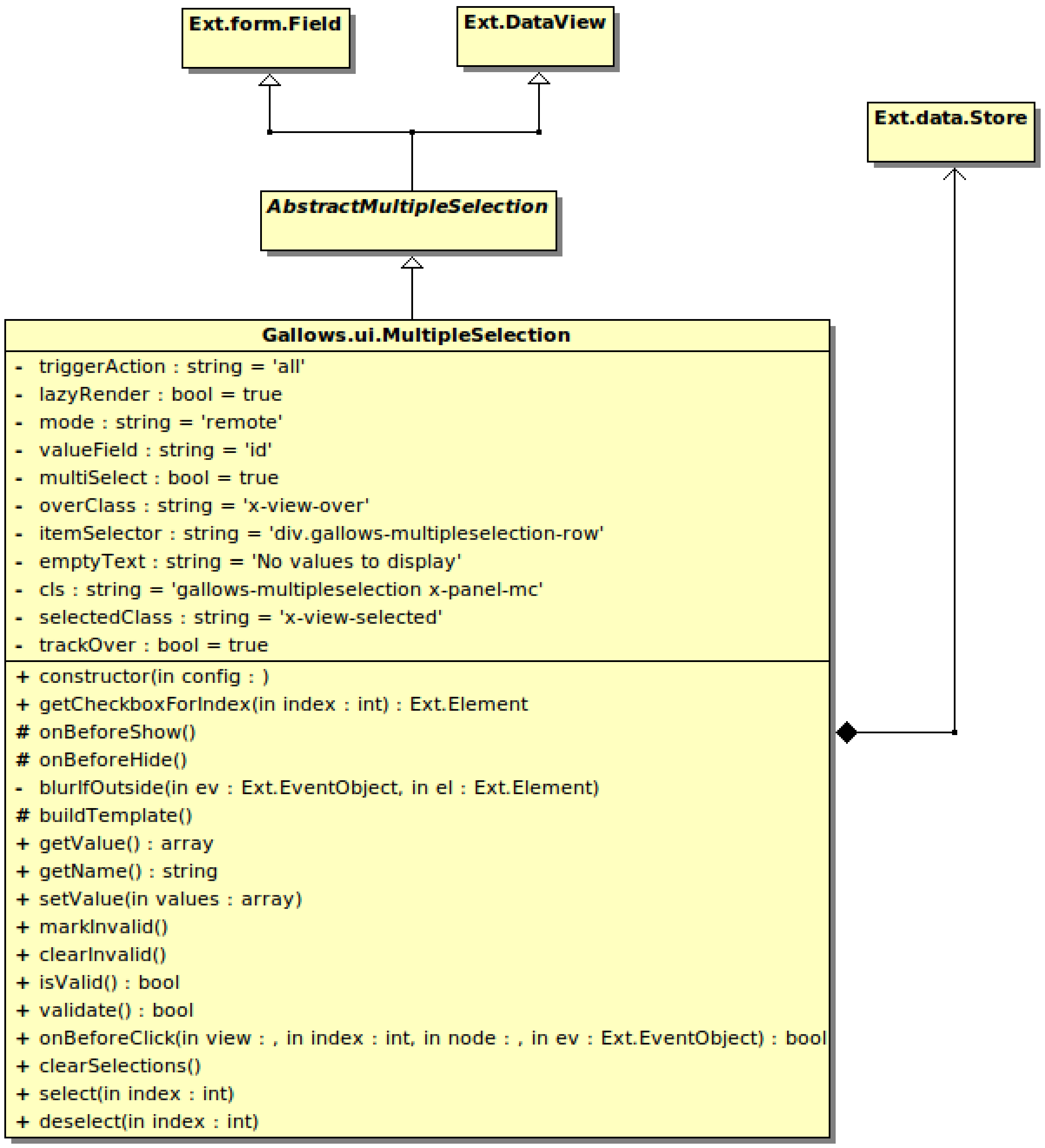

Ext.form.Field
Ext.DataView
Ext.data.Store
AbstractMultipleSelection
Gallows.ui.MultipleSelection
- triggerAction : string = 'all'
- lazyRender : bool = true
- mode : string = 'remote'
- valueField : string = 'id'
- multiSelect : bool = true
- overClass : string = 'x-view-over'
- itemSelector : string = 'div.gallows-multipleselection-row'
- emptyText : string = 'No values to display'
- cls : string = 'gallows-multipleselection x-panel-mc'
- selectedClass : string = 'x-view-selected'
- trackOver : bool = true
+ constructor(in config :)
+ getCheckboxForIndex(in index : int) : Ext.Element
# onBeforeShow()
# onBeforeHide()
- blurIfOutside(in ev : Ext.EventObject, in el : Ext.Element)
# buildTemplate()
+ getValue() : array
+ getName() : string
+ setValue(in values : array)
+ markInvalid()
+ clearInvalid()
+ isValid() : bool
+ validate() : bool
+ onBeforeClick(in view : , in index : int, in node : , in ev : Ext.EventObject) : bool
+ clearSelections()
+ select(in index : int)
+ deselect(in index : int)

Figure 10.6: Class diagram for the MultipleSelection widget

A `MultipleSelection` widget mostly offers the same functionalities of a HTML `<select>` form field, allowing of course to select more than just one entry. You can see it in action in Figure 11.1: by clicking on a item you can select or de-select it at once. Clicking outside the widget will save the data to the underlying `Store` automatically, and trigger a grid refresh.

In this case I wanted `Ext.DataView` to behave as a `Ext.form.Field` because it's the only way for an `Ext.grid.Editor` to wrap it and allow in-place editing in the main grid. Unsurprisingly, I had to add some "*private*" members manually to have it work, even after the copying of properties from the two already mentioned prototypes.

To sum it up, the `MultipleSelection` widget:

1. Fakes a blur event to have it working with `Ext.grid.Editor`, as described in 10.8.2.

2. Has its own Store, which points to a RESTful controller which manages a first-level associated of the currently selected entity.

3. It implements empty validation methods, because there should be no way to make a boolean-per-item selector non-valid.

4. It uses a JavaScript array as the value of a hidden field for a backing store, so there's something to submit when this widget is inside a proper HTML form. Every time an item is (de)selected, the array is updated accordingly, along with the `Ext.data.Store`.

Given everything, this solution proved to work quite well, even if it did take quite some effort to be coded, due to the constraints ExtJS imposes and to the difficulty of assessing the methods to override.

10.8.6. Customizing the default Combobox

The default ComboBox is datastore-agnostic. I wanted something that could immediately be inserted into a grid, and provide search functionalities upon an associated entity. Thus, I've extended the basic widget to always provide a remote store, which is RESTfully linked to a controller. It is thus imperative, in order to have everything working correctly, that the developer generates not only the scaffold for the main entity, but also all related ones, so they're able to respond to JSON queries.

10.8.7. Other customized widgets

I wrote also some other customized widgets, mostly columns that map directly some Rails attribute type to some data in the view. For example, I created separated columns for `Date`, `DateTime` and `Time`. I also implemented simple `Image` and `FileUpload` columns: the former shows a resized image (the controller must accept a data stream for the setter, and return an URL for the getter of the property), or a link to the file.

Most other extensions to ExtJS are documented in the code and used internally, and thus it is not worth reporting them here one-by-one.

10.9. The viewport

In order to provide correct dynamic resizing of ExtJS widgets, the whole document needs to be wrapped in a `Viewport`. This takes over the HTML page, so you can't add normal tags and expect them to show up. To allow the developers to edit the scaffold, and to define their own custom layouts, I provided a partial view, `_viewport.js.erb`, which can be overridden on a layout or single-view basis.

In this file it is possible to change what is to be drawn into the page, placing a container in one of the four main areas defined by the `Viewport`: north, south, east and west.

The default file looks like this:

```
Ext.ns ('Gallows.ui');

Gallows.ui.viewportConfig = function (config)
{
    return Ext.apply ({
      layout: 'border',
      items: [{
          layout: 'fit',
          region: 'north',
          html: '<h1 class="x-panel-header"><%= "Editor for Products" %></h1>',
          autoHeight: true,
          border: false,
          margins: '0 0 5 0'
      }, {
          border: false,
          region: 'center',
          layout: 'fit',
          items: [ // This is where the grid will be rendered.
            Gallows.local.userGrid
          ]
      }]
    }, config);
};
```

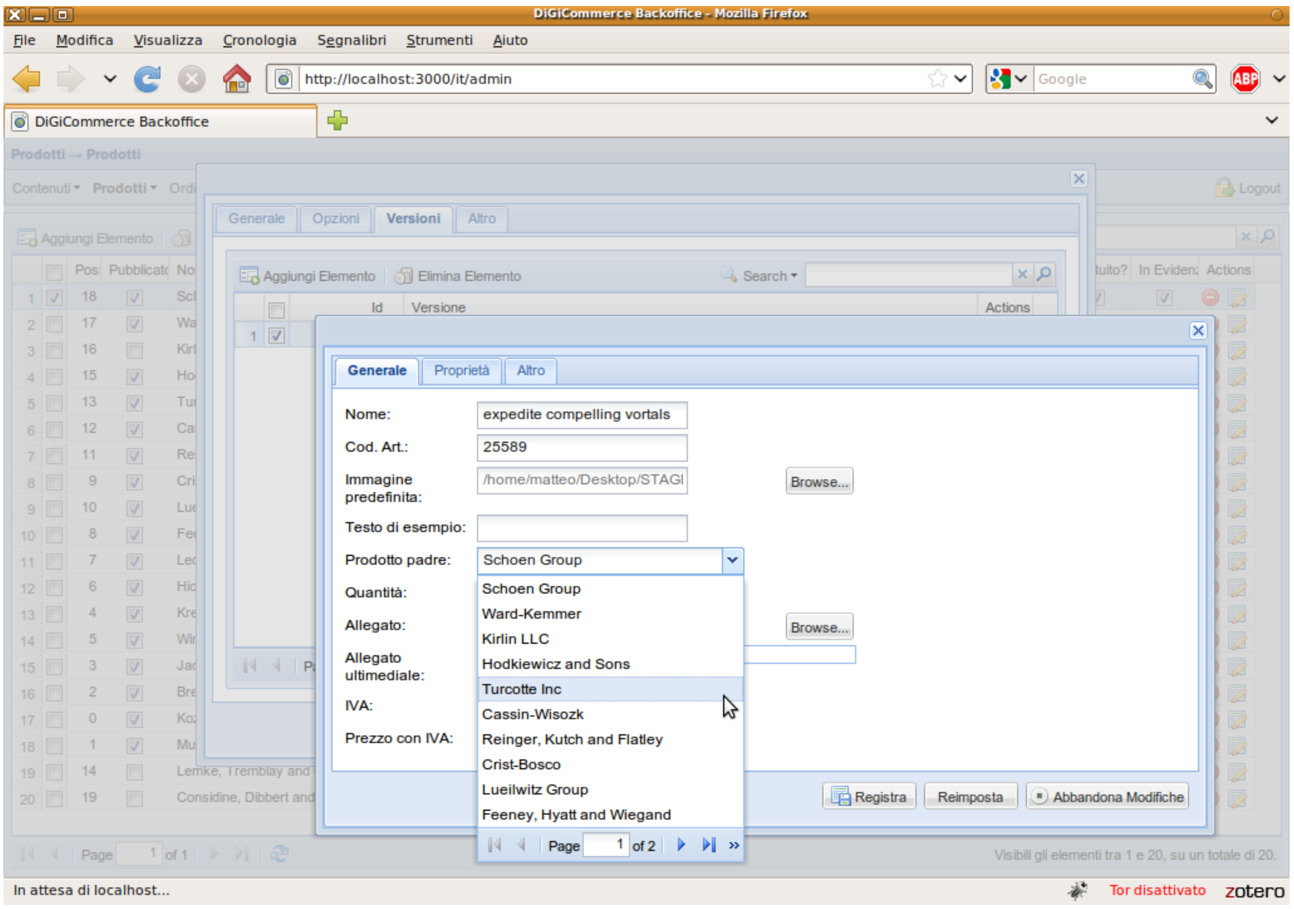
As you can see, adding new areas or widgets is fairly easy.

10.10. *Simple internationalization support*

One of Gallows requested features was of keeping it easily portable to different languages and conventions. To this extent, a `_i18n.js.erb` file was provided, containing both translations of localizable strings employed in the views, and widely-used formats for currencies, numbers, dates and so on. Given it's an ERB template, the translation can easily be managed via Globalize or other Ruby tools (my personal advice is of using GNU Gettext).

10.11. *Associated entities and nested windows*

One of the compelling problems Gallows tries to solve is allowing to edit also entities associated to the active one. In the configuration files, each column listed as `entity.property` or `entities.property` (that is, containing a dot), is recognized as describing an association.


DiGiCommerce Backoffice - Mozilla Firefox
File Modifica Visualizza Cronologia Segnalibri Strumenti Aiuto
http://localhost:3000/it/admin
DiGiCommerce Backoffice
Generale Opzioni Versioni Altro
Aggiungi Elemento Elimina Elemento Search
Id Versione Actions
Generale Proprietà Altro
Browse...
Registra Reimposta Abbandona Modifiche
Page 1 of 2
In attesa di localhost...
Tor disattivato zotero

Nome:	expedite compelling vortals
Cod. Art.:	25589
Immagine predefinita:	/home/matteo/Desktop/STAGI
Testo di esempio:	
Prodotto padre:	Schoen Group
Quantità:	
Allegato:	
Allegato ultimediale:	
IVA:	
Prezzo con IVA:	

Schoen Group
Ward-Kemmer
Kirlin LLC
Hodkiewicz and Sons
Turcotte Inc
Cassin-Wisozk
Reinger, Kutch and Flatley
Crist-Bosco
Lueilwitz Group
Feeney, Hyatt and Wiegand

Visibili gli elementi tra 1 e 20, su un totale di 20.

Figure 10.7: Editing an associated entity in a new modal window. The underlying grid will be updated automatically upon saving.

Gallows is clever enough to do the right thing on the second pass. If you have a singular form (`entity.property`), he knows you're describing a one-to-one or a many-to-one association. Thus, it employs widgets like a researchable combo-box to allow you to choose the right entity from a list. Managing the identifiers, updating the associated entities and such is done automatically and is transparent to the user.

If, instead, we have a plural form, Gallows deduces we want a one-to-many or many-to-many relationship, and acts accordingly, for example with the multiple selection widget defined in `multiple-selection.js`.

But what happens if we want to add a grid inside the standalone editor window, and allow recursive in-place or nested editing of associated entities?

In that case, I provided the `Gallows.ui.create_nested_grid_editor()` function so it is easy to implement this functionality. It is defined into `nested-editor-factory.js`.

An example of its usage is provided here, aimed at showing the versions associated to a product. The configuration you need to specify into `_item_editor.js.erb` goes like this (for a moderately complex example):

```
var nestedEditor = Gallows.ui.create_nested_grid_editor ({
  grid_columns: [
      { header: 'Id', width: 10, sortable: true, dataIndex: 'id',
        xtype: 'gallows-numbercolumn', format: '0', editable: false },
      { header: 'Versione', width: 50, sortable: true, dataIndex: 'to_label',
        xtype: 'gallows-textcolumn', editable: false }
  ]
  ,store_columns: [ /* taken from the versions' _record_store.js.erb; */
    { name: 'id' },
    { name: 'to_label' },
    { name: 'name' },
    { name: 'sku' },
    { name: 'default_image' },
    { name: 'sample' },
    { name: 'product_id' },
    { name: 'quantity', type: 'int' },
    { name: 'attachment' },
    { name: 'updated_at', type: 'date' },
    { name: 'created_at', type: 'date' },
    { name: 'media_files', convert: function (v) {
                         return v ? Ext.pluck (v, 'id') : undefined;
                       }, fieldName: 'media_files.file' },
<%-
  unless @product.nil?
    @product.meta_product.meta_attributes.each do |p|
      %><%=
      "    { name: '#{javascript_escape p.name}'#{property_type_for_extjs_store p} },\n"
      %><%
    end
  end
-%>
    { name: 'vat', type: 'int' },
    { name: 'price', type: 'float' }
  ]
  ,authenticity_token: '<%= form_authenticity_token %>'
  ,nested_controller_url: '<%= admin_product_versions_url I18n.locale, @product %>'
  ,nested_controller_name: 'versions'
});
```

I also wanted to show how we decide exactly what fields to add to the store based on a dynamic meta-product which is present in the Digicommerce (not unalike a template for products). As you can see, nothing prevents you to manipulating, adding or removing fields dynamically. This is a testament to the flexibility that Gallows provides.

Of course, once done so, you still need to add this grid to the right tab, always inside `_item_editor.js.erb`.

10.12. Distributing as a Gem

Even if the plug-in form was sufficient for our purposes, I decided to add a Rake[16] task to automatically build a stand-alone and platform-independent package. In the Ruby world, these packages are called “gems”. Thus, running `rake package` will give you a ready-to-use and immediately exportable gem in the `pkg/` subdirectory.

10.13. Why testing is so hard

Testing code generators is quite hard. There is some research going on to make this easier[17], however this is either difficult to implement, lacks some already-available framework for the language at hand, or still requires a lot of work. Mostly, this happens because you've not much

16 Mostly, Rake is Ruby's `make`.

17 For example, see Baldan, König, Stürmer “*Generating Test Cases for Code Generators by Unfolding Graph Transformation Systems*”, Springerlink 2004.

control on the code you're going to generate, and about how it'll intermix with an application you don't know how will be coded. This is the same reason why libraries or toolkits are hard to test, too.

In the specific, Gallows makes some assumptions on entity, attributes and associations names (which are mostly the same Rails does), however it doesn't give you the same flexibility to force them to something else than the expected standard. In other words, if you did too much mix-n-mojo in your app, expect some trouble.

Anyway, Gallows is hard to test because you can either choose to test generated code – and then you don't have much control about what's the input, and it's quite hard to verify the correctness of a computer program –, or to simply unit test the methods the generator employs, and then you do need to implement a parser and a lexer. In our case, implementing a JavaScript parser would have required more time than that spent on implementing Gallows alone.

Pejoratively, there's no effective means to date to test Javascript code adequately (nor to debug it, if it comes to that). Selenium IDE[18] seems promising, but it has still some road to do before becoming reliable enough.

Thus, testing was postponed to the generated code, after integration with the Digicommerce. That seemed the more natural and effective course of action at the time.

18 http://seleniumhq.org/

11.Integrating Gallows with the Digicommerce

I've included some screenshots in the following pages to show some of the work that has been done to integrate Gallows with the proprietary Digicommerce e-commerce solution. Since everyone loves screenshots, in the next sheets you can see how the administrative section of the e-commerce is presented.

I'm happy to say that the integration part went smoothly and required just a few days of monkey-patching to complete (mostly, copying column header names across the old and the new interface, and localizing in Italian the application). As far as I know, the company is already using my code in production.

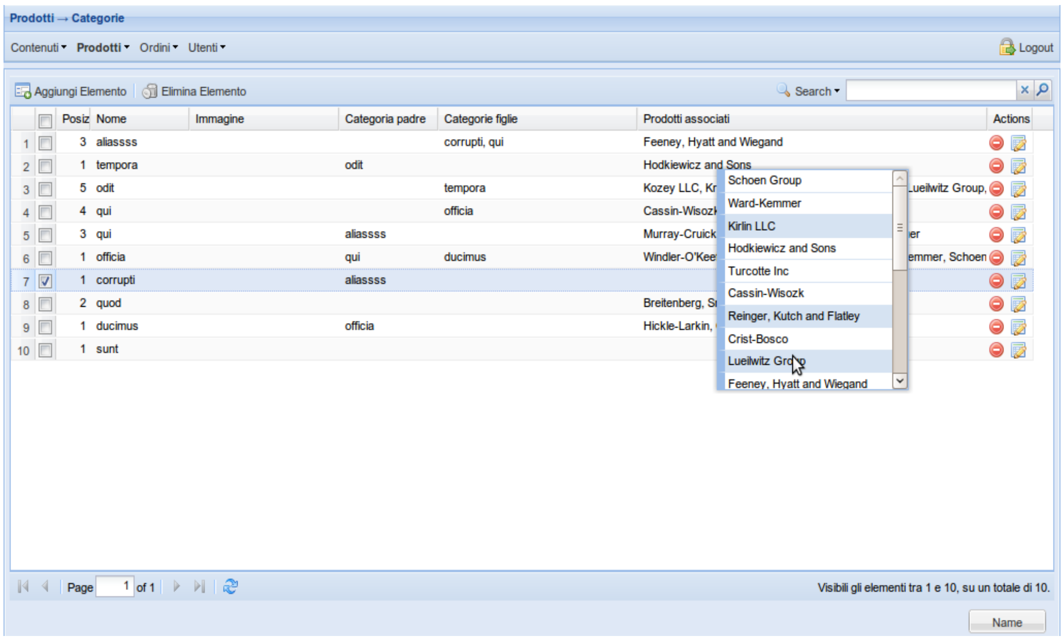
11.1. Why the generated code hasn't been tested adequately

Yes, it's an hard truth. Testing has been done just by hand, and with no automated means. It mostly has involved following for each page the same routine of testing the CRUD actions. This isn't something I take pride of.

This happened due to the lack of time. Testing was deferred to this stage because testing Gallows directly was too hard. However, I expected to find a series of already implemented tests for the old administrative interface I had had to replace, and to fix them to work with the new ExtJS GUI.

Unfortunately, the whole Digicommerce application lacks tests of any sort. This is something the company is aware of, and they'll work to fix it somewhere in the future; however, as of now, they've just a system of e-mail notifications when an exception arises in production. Then they take a look at the back-trace and jump directly to fix the issue, hoping it doesn't happen again and that something else has not been broken by the patch.

I presented the problem to the stakeholder, and I was told to prefer implementing new functionalities over testing. Given the time at hand, I had to make the hard choice. Thus, testing was broken-heartedly left out.



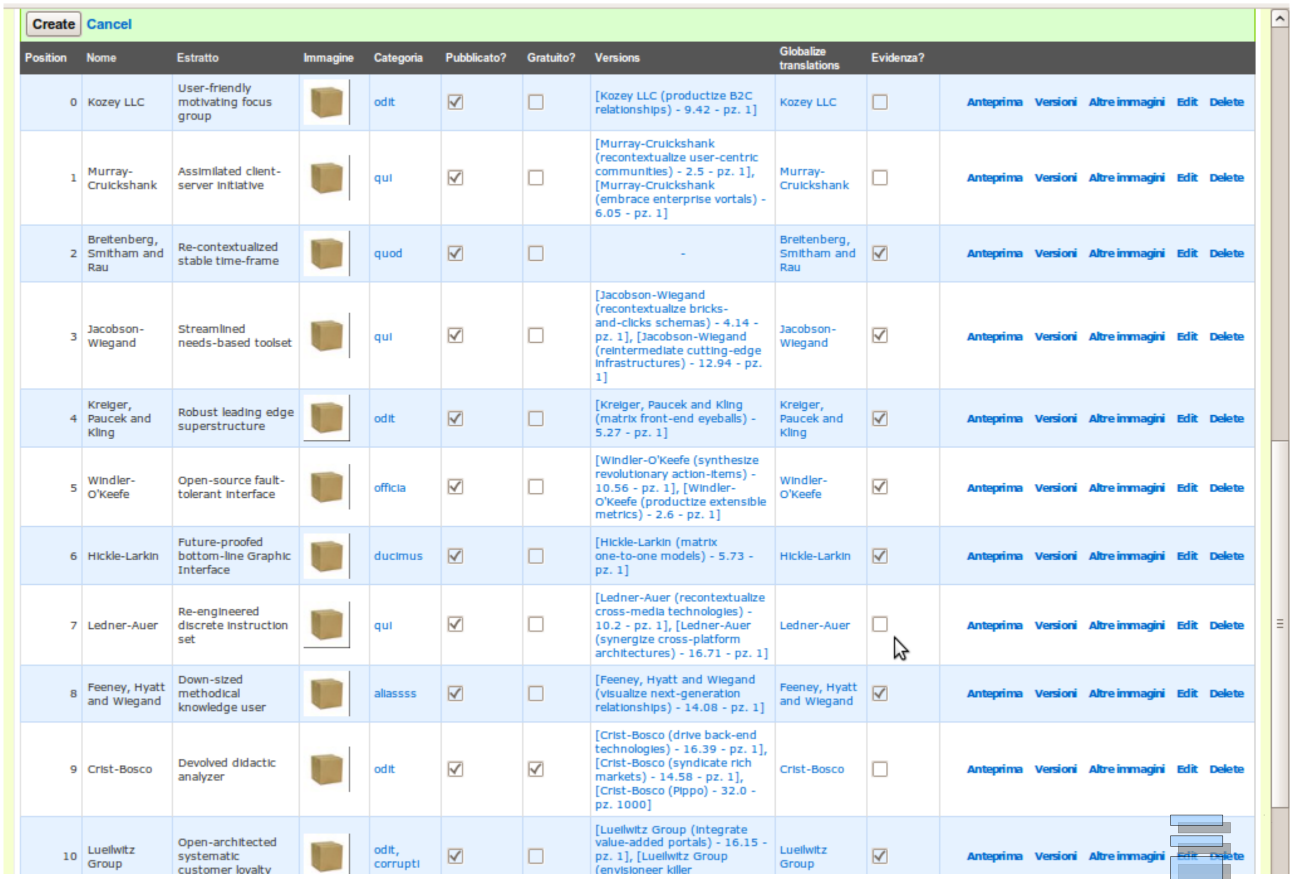
		Posiz	Nome	Immagine	Categoria padre	Categorie figlie	Prodotti associati	Actions
1	☐	3	aliassss			corrupti, qui	Feeney, Hyatt and Wiegand	
2	☐	1	tempora		odit		Hodkiewicz and Sons	
3	☐	5	odit			tempora	Kozey LLC, Kr … Lueilwitz Group,	
4	☐	4	qui			officia	Cassin-Wiszok	
5	☐	3	qui		aliassss		Murray-Cruick … er	
6	☐	1	officia		qui	ducimus	Windler-O'Kee … emmer, Schoen	
7	✓	1	corrupti		aliassss			
8	☐	2	quod				Breitenberg, S	
9	☐	1	ducimus		officia		Hickle-Larkin,	
10	☐	1	sunt					

Schoen Group
Ward-Kemmer
Kirlin LLC
Hodkiewicz and Sons
Turcotte Inc
Cassin-Wisozk
Reinger, Kutch and Flatley
Crist-Bosco
Lueilwitz Group
Feeney, Hyatt and Wiegand

Page 1 of 1

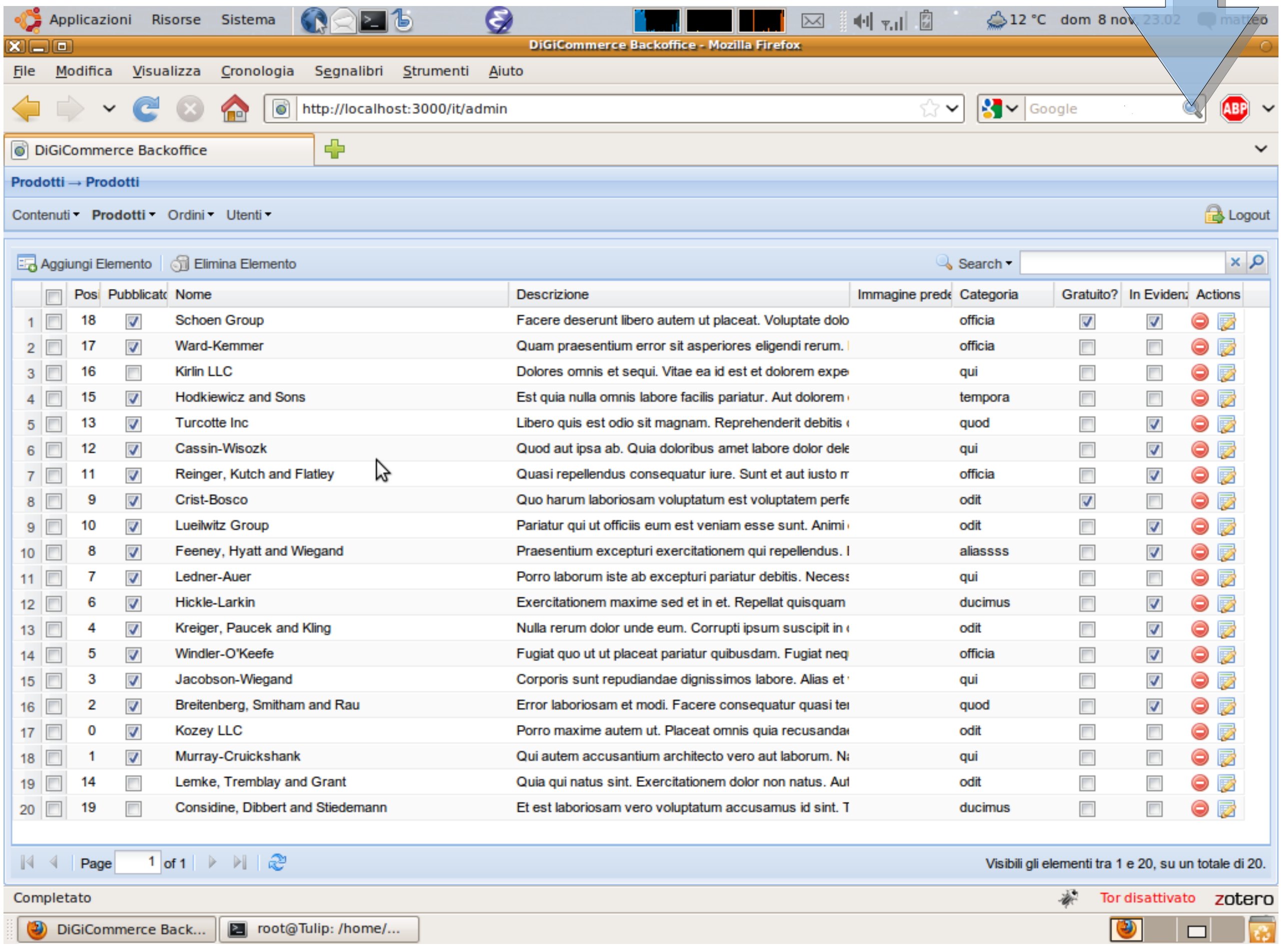
Visibili gli elementi tra 1 e 10, su un totale di 10.

Figure 11.1: An example of in-place editing, using a multiple-selection custom widget



Position	Nome	Estratto	Immagine	Categoria	Pubblicato?	Gratuito?	Versions	Globalize translations	Evidenza?	
0	Kozey LLC	User-friendly motivating focus group		odit	✓	☐	[Kozey LLC (productize B2C relationships) - 9.42 - pz. 1]	Kozey LLC	☐	Anteprima Versioni Altre immagini Edit Delete
1	Murray-Cruickshank	Assimilated client-server initiative		qui	✓	☐	[Murray-Cruickshank (recontextualize user-centric communities) - 2.5 - pz. 1], [Murray-Cruickshank (embrace enterprise vortals) - 6.05 - pz. 1]	Murray-Cruickshank	☐	Anteprima Versioni Altre immagini Edit Delete
2	Breitenberg, Smitham and Rau	Re-contextualized stable time-frame		quod	✓	☐	-	Breitenberg, Smitham and Rau	✓	Anteprima Versioni Altre immagini Edit Delete
3	Jacobson-Wiegand	Streamlined needs-based toolset		qui	✓	☐	[Jacobson-Wiegand (recontextualize bricks-and-clicks schemas) - 4.14 - pz. 1], [Jacobson-Wiegand (reintermediate cutting-edge infrastructures) - 12.94 - pz. 1]	Jacobson-Wiegand	✓	Anteprima Versioni Altre immagini Edit Delete
4	Kreiger, Paucek and Kling	Robust leading edge superstructure		odit	✓	☐	[Kreiger, Paucek and Kling (matrix front-end eyeballs) - 5.27 - pz. 1]	Kreiger, Paucek and Kling	✓	Anteprima Versioni Altre immagini Edit Delete
5	Windler-O'Keefe	Open-source fault-tolerant interface		officia	✓	☐	[Windler-O'Keefe (synthesize revolutionary action-items) - 10.56 - pz. 1], [Windler-O'Keefe (productize extensible metrics) - 2.6 - pz. 1]	Windler-O'Keefe	✓	Anteprima Versioni Altre immagini Edit Delete
6	Hickle-Larkin	Future-proofed bottom-line Graphic Interface		ducimus	✓	☐	[Hickle-Larkin (matrix one-to-one models) - 5.73 - pz. 1]	Hickle-Larkin	✓	Anteprima Versioni Altre immagini Edit Delete
7	Ledner-Auer	Re-engineered discrete instruction set		qui	✓	☐	[Ledner-Auer (recontextualize cross-media technologies) - 10.2 - pz. 1], [Ledner-Auer (synergize cross-platform architectures) - 16.71 - pz. 1]	Ledner-Auer	☐	Anteprima Versioni Altre immagini Edit Delete
8	Feeney, Hyatt and Wiegand	Down-sized methodical knowledge user		aliassss	✓	☐	[Feeney, Hyatt and Wiegand (visualize next-generation relationships) - 14.08 - pz. 1]	Feeney, Hyatt and Wiegand	✓	Anteprima Versioni Altre immagini Edit Delete
9	Crist-Bosco	Devolved didactic analyzer		odit	✓	✓	[Crist-Bosco (drive back-end technologies) - 16.39 - pz. 1], [Crist-Bosco (syndicate rich markets) - 14.58 - pz. 1], [Crist-Bosco (Pippo) - 32.0 - pz. 1000]	Crist-Bosco	☐	Anteprima Versioni Altre immagini Edit Delete
10	Lueilwitz Group	Open-architected systematic customer loyalty		odit, corrupti	✓	☐	[Lueilwitz Group (integrate value-added portals) - 16.15 - pz. 1], [Lueilwitz Group (envisioneer killer	Lueilwitz Group	✓	Anteprima Versioni Altre immagini Edit Delete

Figure 11.2: The old Products administrative view, as managed by ActiveScaffold

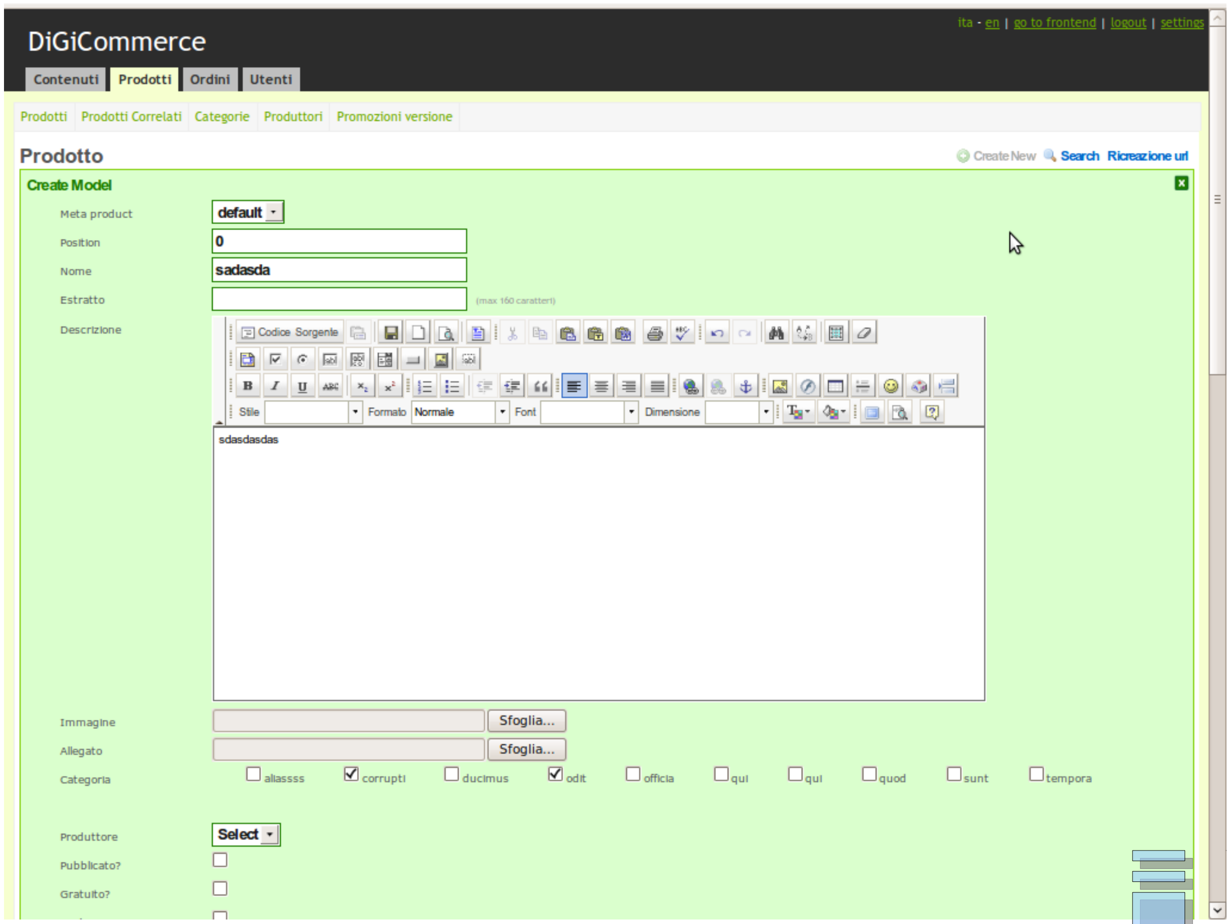


		Pos	Pubblicato	Nome	Descrizione	Immagine prede	Categoria	Gratuito?	In Eviden:	Actions
1	☐	18	✓	Schoen Group	Facere deserunt libero autem ut placeat. Voluptate dolo		officia	✓	✓	
2	☐	17	✓	Ward-Kemmer	Quam praesentium error sit asperiores eligendi rerum.		officia	☐	☐	
3	☐	16	☐	Kirlin LLC	Dolores omnis et sequi. Vitae ea id est et dolorem expe		qui	☐	☐	
4	☐	15	✓	Hodkiewicz and Sons	Est quia nulla omnis labore facilis pariatur. Aut dolorem		tempora	☐	☐	
5	☐	13	✓	Turcotte Inc	Libero quis est odio sit magnam. Reprehenderit debitis		quod	☐	✓	
6	☐	12	✓	Cassin-Wisozk	Quod aut ipsa ab. Quia doloribus amet labore dolor dele		qui	☐	✓	
7	☐	11	✓	Reinger, Kutch and Flatley	Quasi repellendus consequatur iure. Sunt et aut iusto m		officia	☐	✓	
8	☐	9	✓	Crist-Bosco	Quo harum laboriosam voluptatum est voluptatem perfe		odit	✓	☐	
9	☐	10	✓	Lueilwitz Group	Pariatur qui ut officiis eum est veniam esse sunt. Animi		odit	☐	✓	
10	☐	8	✓	Feeney, Hyatt and Wiegand	Praesentium excepturi exercitationem qui repellendus. I		aliassss	☐	✓	
11	☐	7	✓	Ledner-Auer	Porro laborum iste ab excepturi pariatur debitis. Necess		qui	☐	☐	
12	☐	6	✓	Hickle-Larkin	Exercitationem maxime sed et in et. Repellat quisquam		ducimus	☐	✓	
13	☐	4	✓	Kreiger, Paucek and Kling	Nulla rerum dolor unde eum. Corrupti ipsum suscipit in		odit	☐	✓	
14	☐	5	✓	Windler-O'Keefe	Fugiat quo ut ut placeat pariatur quibusdam. Fugiat neq		officia	☐	✓	
15	☐	3	✓	Jacobson-Wiegand	Corporis sunt repudiandae dignissimos labore. Alias et		qui	☐	✓	
16	☐	2	✓	Breitenberg, Smitham and Rau	Error laboriosam et modi. Facere consequatur quasi te		quod	☐	✓	
17	☐	0	✓	Kozey LLC	Porro maxime autem ut. Placeat omnis quia recusanda		odit	☐	☐	
18	☐	1	✓	Murray-Cruickshank	Qui autem accusantium architecto vero aut laborum. N		qui	☐	☐	
19	☐	14	☐	Lemke, Tremblay and Grant	Quia qui natus sint. Exercitationem dolor non natus. Au		odit	☐	☐	
20	☐	19	☐	Considine, Dibbert and Stiedemann	Et est laboriosam vero voluptatum accusamus id sint. T		ducimus	☐	☐	

Page 1 of 1

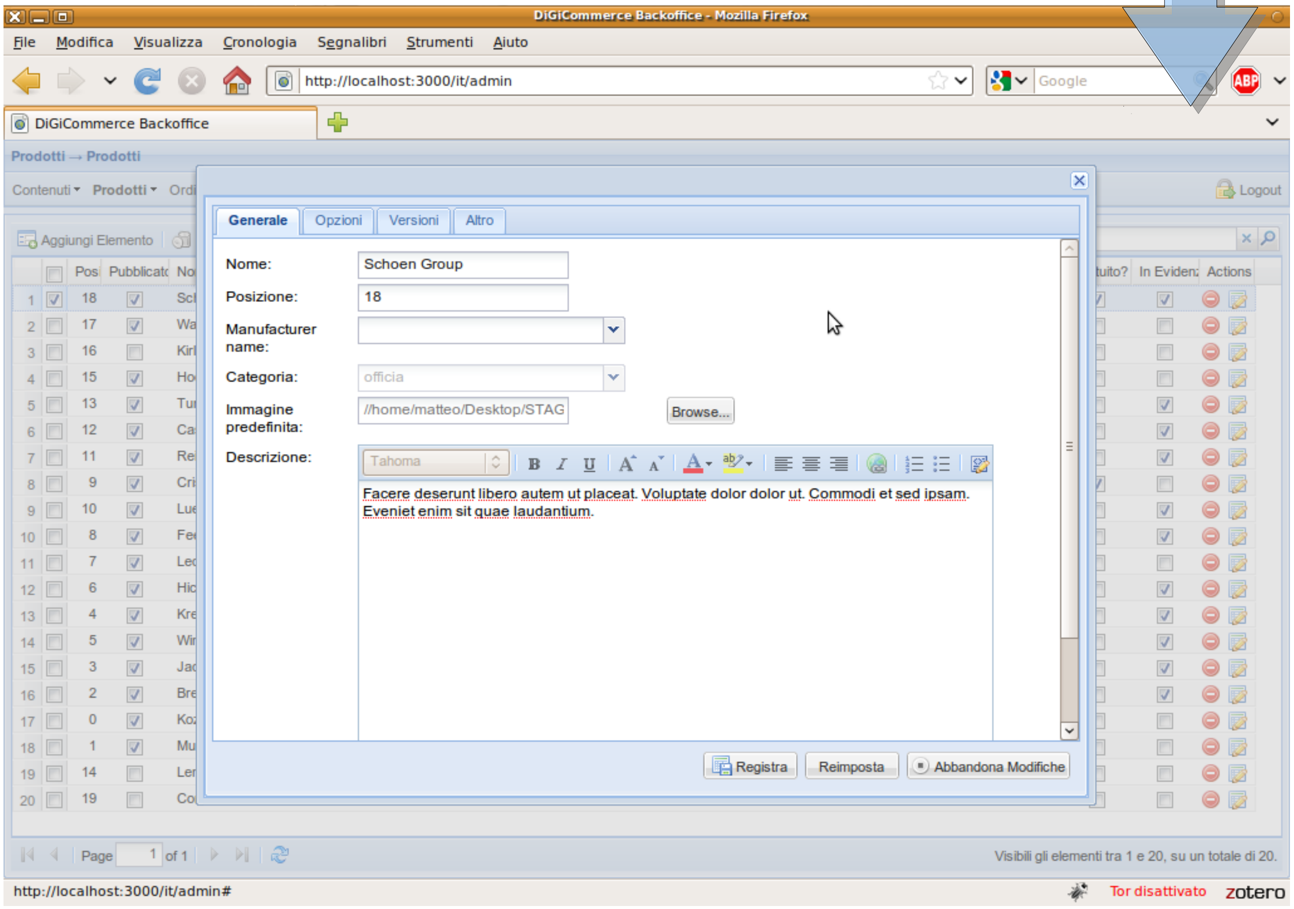
Visibili gli elementi tra 1 e 20, su un totale di 20.

Figure 11.3: The product view



Meta product	default
Position	0
Nome	sadasda
Estratto	(max 160 caratteri)
Normale	
Descrizione	sdasdasdas
Immagine	
Allegato	
Categoria	☐ allassss ☑ corrupti ☐ ducimus ☑ odit ☐ officia ☐ qui ☐ qui ☐ quod ☐ sunt ☐ tempora
Produttore	Select
Pubblicato?	☐
Gratuito?	☐

Figure 11.4: The old form for creating a new Product, as outputted by ActiveScaffold



Nome:	Schoen Group
Posizione:	18
Manufacturer name:	
Categoria:	officia
Immagine predefinita:	//home/matteo/Desktop/STAG
Descrizione:	Facere deserunt libero autem ut placeat. Voluptate dolor dolor ut. Commodi et sed ipsam. Eveniet enim sit quae laudantium.

Visibili gli elementi tra 1 e 20, su un totale di 20.

Figure 11.5: The single-item editor in a separate window, with an HTML editor and tabbed contents

12. Conclusions

I would definitely recommend Ruby on Rails for new projects. Its high manageability through time, cleanness and ease of development fully outweigh my only complain with it: it's slow. However, for small-to-medium sized websites, and given the hardware available nowadays, this is largely negligible.

Being able to have a fully functional mock-up within a couple of hours is certainly a hands-down winner in my opinion. Moreover, the wonderful community support and the incredible number of dedicated plug-ins that's around is enough to make your head spin in delight. Just a word of warning, though: when you decide including a gem in your work, check that the bus factor[19] isn't too low (even then, often replacing plug-ins when they become obsolete isn't much work, and that contributes to the wow-factor).

On the other hand, I'd advice **against** using ExtJS. Even if aesthetically very pleasing, it has some heavy downsides, and namely:

I'd advice against using ExtJS, [it] has some heavy downsides

1. Debugging is **very** hard, even by using Firebug (which is the unique tool that works well enough for this task).

2. The same things may be achieved also via directly generating the pages server-side, with much less overhead, much less hassle for the developer, and much more control (debugging on your machine – may it be your local host or your server – is one thing; debugging client-side errors on customer PCs outside your control is difficult).

3. Accessibility features for visually or movement impaired people aren't up on the standard. Future ExtJS versions may address this by supporting WAI-ARIA[20], as reported by the *roadmap*[21].

4. ExtJS integrates badly with Rails' way to manage field names (the `'[]'` problem we discussed before, providing a solution; however, there are more complex cases where it isn't enough) and with some other features provided by routing and controllers. I've had to fiddle around quite a little bit to get `Readers` to behave in a truly RESTful manner.

5. Many useful widgets are missing, and are left to the developer to implement by hand. Such examples, as we've seen, are the multi-selection HTML element `<select>` and check-boxes inside grids, just to name two. Extending widgets is hard, you've got a gazillions methods to redefine to have everything working fine – and many times you don't know *which* they are.

6. Many times the documentation, even if quite extensive, isn't enough. I've often had to read hundreds of lines of code, backtracking inside private methods and properties, just to end up extending widgets defining the same private properties to achieve a behavior comparable to the pre-defined ones[22].

19 http://en.wikipedia.org/wiki/Bus_factor
20 See http://www.w3.org/WAI/intro/aria
21 http://www.extjs.com/products/extjs/roadmap.php
22 And don't believe it's just me being a lazy developer. One, because reading others' code can hardly be called laziness. And two, if you take a look to many plugins written by prominent ExtJS community members, you'll see they use the same techniques. For a concrete example of this in action, see http://rowactions.extjs.eu/source.php?file=js/Ext.ux.grid.RowActions.js , lines 226-231.

7. Too much glue to write to have everything working fine together in a nice way. There's not enough return of investment.

Therefore, I would stay with a more classical generator which keeps away from AJAX and JavaScript where possible. Other frameworks like jQuery and Prototype (and even ExtJS Core, which is ExtJS without all the widgets) offer the base functionality you need to have visually pleasing artifacts and manipulating the DOM, but I'd delegate the server side to do the most of the work.

Personally, I would have further automatized the creation of some code by defining some proxy classes for ExtJS widgets in Ruby, and used them at run-time to output the right code on-the-fly. However, it was an explicit request of the stakeholder not to do so, purportedly for reasons of easiness of customization in the future.

Due to pressing time constraints I didn't use the meta-configuration capabilities of the ExtJS data Store. However, whoever will come after me may want to seriously consider changing Gallows' logic to rely on it.

See “*Automatic configuration using metaData*” at http://www.extjs.com/deploy/dev/docs/?class=Ext.data.JsonReader.

As a final note, JSON is a good format to go with, it's much more lighter than XML as a data format. Its employment in rising technologies like CouchDB[23], its easiness of parsing and its improving support inside Rails makes it a nice choice for data exchange.

To sum it up: I'm glad I took a shot at this project. I've got a much clearer vision now. I believe my qualities as a programmer and a designer have improved as a whole, and I'd like to thank Diginess for the opportunity they offered me.

I'm happy to say that Gallows, and the administrative interface I integrated with Digicommerce should – to the best of my knowledge – be already in production.

In the following part of this thesis I'll treat shortly some of the challenges you'll encounter when interfacing with others on the workplace, and about some of the solutions you may adopt to come out of them successfully.

Stay tuned.

23 http://couchdb.apache.org/

PART II: WORKING WITH PEOPLE

1. Rationale

What makes a good programmer? Someone says it's keeping the bugs-per-line-of-code ratio low while staying on schedule, someone else says it's all about being fast adapting to new technologies emerging in the arena, others affirm the secret is in being able to fully understand the domain of analysis, and being able to read between the lines of the specifications, as passed down by the designer, in order to find the best way to implement things.

Well, there's something in all that, but I found more. A lot more. The secret? Building good relationships, having a critical mindset and, most importantly, being able to push your point without trying to win an argument, thus transforming every conversation in an ongoing battle.

The time in which the myth of the solitary programmer, alone in a small cubicle 12h per day with a mug of coffee casting assembly-language spells on his PDP-11, are at their end. Nowadays, you must confront every few minutes with your colleagues, your boss, the customers (or your project manager on their behalf), your sales director, and so on and on and on.

This is – and don't do that face – '*social*'. No, it should not trigger a sensation of disgust. It is, for whom is able to master this art, the best way to be successful **and** to ensure that what you're building: a) respects the specs – Agile methods' practitioners push exactly on this when they talk of getting continuous feedback during development -, b) is of good quality and c) makes you proud of having been part of the whole.

[T]he benefit of sharing with others largely outweighs [the] risks

The last point is somewhat contradictory: you're asked to give up exclusive fatherhood (or motherhood, depending on your gender), and learn to share your experience, your criticism and your ideas with others. This means you've got much more to lose; what if the others learn from you and then decide they can do without you? What if you're not up to the job? The point is that the benefit of sharing with others largely outweighs these risks: you've as much to learn as to teach, and the whole is bigger than the sum of the parts[24]. By the way, don't think that just acting friendly and using others' experience for your goals without giving a fair exchange will work. Building trust is the key.

There are well-known and precise techniques to reach this result. It is out of the scope of this thesis to encompass all of them; however, I'll try to give a quick summary of some of the things I found most useful for your daily work, asking you to refer to [6] and [9] for further informations.

24 Just please don't hold me accountable for propagandizing holistic theories.

2. Managing relationships

In my personal experience as a worker, I've found that building a strong trust relationship is fundamental. When trouble arises (and it does, rest assured), I've often needed an effective method to resolve conflicts identifying the problem at hand and a good solution which works well for all the parts involved.

when things go hard, we attack or we flee

Millions of years of genetic inheritance have built us to react mainly in two ways, when things go hard: attack or flee. The heart starts pumping blood into muscles and away from the brain, arguably the most in need for oxygen for cooling down and rationalize what's happening [6:pp. 4-5].

And there's the effort. Calm down, and ask yourself: "*What am I trying to achieve? What's my goal into this discussion?*". Chances are that you'll find already a way to tackle the situation.

2.1. Don't hide

Avoiding facing your problems, be it in your personal life or on the workplace, isn't a good way to resolve conflicts. The point is, you probably won't be satisfied by the solution, since you weren't involved in it. You'll keep building frustration or anger, because you didn't express your views. And others won't benefit from your POV, which may be potentially fundamental for a correct analysis of the problem. Usually, people avoid jumping into a discussion either because they think they will pay the price of expressing their opinions (and be held accountable for them if things go awry), or because they fear they make a fool of themselves.

For the first class of people, my advice is: don't panic. If a decision is *shared* across people, everyone must also share the same amount of credit and of responsibility, too. This is why reaching conclusions in a group is a process which requires everyone's approval. This way, nobody can then raise his finger to say: "*I did know it would have ended like this!*". Why didn't you tell your opinion before, then?

And for the second class: trust yourself. You're a human being as much as the others. People can be wrong. How can you know if you did make a mistake, until you don't confront with what others would do? Many times, you'll see that *both* you and the others will be wrong. That's fine. The goal is exactly this: from direct confrontation stems a better solution which probably *will be better than all those thought by each one in advance*. You're contributing to the pool of meaning. [6:pp. 22-23]

from direct confrontation stems a better solution, better than all those thought by each one in advance

This, of course, doesn't imply speaking before thinking. We will shortly speak about *the method* which distinguish someone trying to confront successfully with others from someone who's just trying to win an argument, or isn't interested in other views.

2.2. Don't attack

The second form of reaction to being under pressure is that of attacking. This may be masked under different clothes: sarcasm, controlling, labeling and even violence. [6:pp. 51-54]

Resist the temptation, strong as it may be. You're actively (instead of passively, which happens when you hide) preventing others to participate to the pool of meaning. Take a deep breath, and change your mindset: “*May I do something more to have other people at ease, so they can contribute to the conversation?*”

“Let's try to find a solution together”

It isn't sufficient to just do an angry step back, eyes glazing retribution for whoever opens his mouth. You should make the conversation safe for others to participate. Like: “*Okay, listen: sorry if I've raised my voice. I sincerely want to hear your opinion, because I think it may help the project. Let's try to find a solution together which will benefit the most, shall we? I promise I'll sit down until we don't have found it.*” Suddenly, you've made it *safe* for the other person to share their views.

You should *feel* what you say, not just say it. The last example contains most of the steps you should take, and we will go through them in the next paragraph.

2.3. Managing critical conversations

Let's give a short method for managing crucial conversations. A quick summary of the steps you should take to confront with others is:

1. *Believe in what you're saying.* Because lies don't work. Everyone, consciously or not, will be able to spot if you really do *mean* what you're saying. If you start a conversation just to lie, then stop here, because you've the wrong goal.

2. *Make the conversation safe!* In order to have others contributing to the pool of meaning, you should work to make it safe for them to express their ideas, or else they'd do exactly what you'd do: attack or escape. Stand back, tell them you're sorry if you did something wrong, and create the right environment for them to intervene. This is the single most important step: if you manage this, people will be able to talk about anything, even their most buried feelings, and it'll be fit for the space you've built for them. Also remember than a conversation has to *be kept* safe for all its duration.

   Make the conversation safe!

3. *Commit to a shared goal.* A good way is to work by contrast. Say exactly the outcome you would not want, and what you're not trying to do (for example, hurt others' feelings). Then state what's you're goal, what you're trying to accomplish. Do this in that order. It'll help to make the conversation even safer, and to set up a common direction for everyone involved.

4. *Allow everyone to tell their opinions*. And you too. Also remember: if you made some mistake (say: you're trying to push for some cuts to wages, but you approved last month a new budget for renewing the office furniture), don't get angry, or evade the question. Admit your error – or you'll lose all the trust the others have in you. This is the worst it may happen, since it takes years to build and once destroyed, it won't grow as easily. Then work to find a solution, tell them why you did something, and demonstrate your good intentions.

5. *Find a common solution*. Sometimes, this isn't possible. Try to find the solution that, by democracy, will satisfy the most. Explain *exactly* to the others why you think it's the best one at hand. Give them time to think about it, don't rush things. Three words: listen, listen and listen.

6. *Thank others for their contributions.* Make them feel important not just because an happy customer/colleague/friend/lover is better than an unhappy one, but because you honestly believe they did their best for the conversation's outcome. Believe me, they were; even if they did it wrong, they always set an example for others – admittedly, in some desperate case it's about how not to do it, but still.

If you want to continue improving and going deeper into these matters, rest assured it'll be time well spent in training. Please refer to [6] for further informations.

3. A note about critical thinking

Even if critical thinking is such an important skill, it isn't often taught in schools, probably hoping it'll be absorbed by osmosis by the students. Myself, I'm pretty new to the subject. I wanted nevertheless spend a couple of words of advice: learn how to process your mental processes. It's like mental – I mean, meta! – programming your brain.

> *Critical thinking is that mode of thinking – about any subject, content or problem – in which the thinker improves the quality of his or her thinking by skillfully taking charge of the structures inherent in thinking and imposing intellectual standards upon them.* (Paul, Fisher and Nosich, 1993, p. 4) [3:pp. 4-5]

Many times, "thinking" is taken for granted. Even if you concentrate upon *what*, you don't pay much attention about *how*. Critical thinking is a needed instrument in your toolkit.

There are different way of reasoning, hard-to-spot fallacies you want to learn to avoid doing, evaluating others views in search for good and bad reasons, and so on. This directly translates into being able, as a programmer, to read a spec file and highlight those passages which aren't directly required for demonstrate the completeness of the system, wrong assumptions and long shots, but also as an individual when you confront with others which are trying to push their points.

Even if we don't have the time and space here to dwell into all the aspect of this subject, it will come handy to know you can refer to [3] for a quick introduction. However, there is something that must be said, since programmers often forget about it. **Read a lot** and also **write a lot**.

Read a lot and also write a lot.

Reading a lot doesn't just mean reading technical papers. A well-written thriller or sci-fi book will work all the same, if not better. That's because you need to be able to understand *stories*. One, because you've specs to read, and those stories tend to be one million times more complicated than a Tolkien book (believe me...). And two, because you need it as a *tool for your trade*. You should treasure every expression and form which tickles your imagination, and that makes you jump on your chair wondering: "*Gosh, I was thinking roughly the same thing, only this is the way I would have expressed it if I knew the right words*".

And don't stop just about the general plot. Do some meta-analysis. Use *critical thinking*. Ask yourself: why the author did say things this way and not in another? Try to make an example of another way to say the same thing. Is your way more effective? (If so, we're going somewhere). Why is there a ascending climax between these two sections? Why *those* words and not *other* words? Even if chosen by instinct, they should give you informations about how the author do her reasoning. Don't take it all for granted!

Then, *train yourself on writing* [9:pp. 69-76]. You probably thought that, finally leaving high school for a CS course, you finally were dispensed from writing essays. Let me say that clearly: **you were dead wrong**. Myself, I've not yet finished getting my degrees and I've already started writing dozens-of-pages-long documents (mostly, requirement analysis for developing new software). Remember: writing for a compiler means you've got to pay attention to the *syntax*, but after it compiles, it doesn't have to be also *beautiful*. Human beings aren't compilers. Tah.

You should train weekly, if not daily, on writing essays in a good form. It is essential using all the narrative skills and rhetorical methods you learned in all those years as an undergraduate. You must be able to transmit emotions to the reader; that's the only way to interest him or her. If you can't do that, don't expect that a bank will finance your business plan, or that other people will join you in a venture.

Writing is a muscle. The more you write, the more you'll be able to write.

Joel Spolsky tells us: "*[...] so many people don't like to write. Staring at a blank screen is horribly frustrating. Personally, I overcame my fear of writing by taking a class in college that required a 3-5 page essay once a week. Writing is a muscle. The more you write, the more you'll be able to write. If you need to write specs and you can't, start a journal, create a weblog, take a creative writing class, or just write a nice letter to every relative and college roommate you've blown off for the last 4 years. Anything that involves putting words down on paper will improve your spec writing skills.*" [9:p. 51]

Bad communication, and the inability to write well can have disastrous effects on your work – just think about how many e-mails you've to send as a professional every day. Now think about the individual at the other end of the pipe laughing at you for your shaky grasp of the grammar, or judging you just on the basis of what you've written. Scary, huh? That's *exactly* how it works.

Training is the *only* way to improve.

4. Working as a programmer

Being a programmer resembles a little being a painter [5]. You're a craftsman which, by careful analysis and training, is able to build something unique. As in the Renaissance, you do train on the workplace, like in an artisan's *bottega*. The final product is something that can never be replicated from scratch by someone else, except by mechanical copy; every individual has their way to work on a problem and solve it. That's probably what builds the distinction between craftsmanship and industry – incidentally, the term “computer industry” isn't one to be proud of.

Therefore, you should aim not only to mere functionality but, in the same vein an artisan does, also for *beautifulness* of your work.

4.1. The environment

Contrary to what some personnel managers will tell you, a programmer needs silence and a certain isolation to do her work. This doesn't mean a programmer shouldn't interact with others every time it's needed; it just means that if you've an open space, programmers will ask questions and advice one each others every few minutes.

Empirically, a programmer needs more or less fifteen minutes of concentration before entering what Spolsky calls “*the zone*”, and thus becoming fully productive. Each time a colleague asks something, a phone rings, someone yells, or the boss starts some disco music, this concentration is disrupted. All the variable names you keep in your short-memory, along with their values, do vanish.

That's why, if possible, separating *physically* programmers by a door is preferable. When searching for the signature of a method takes 20 seconds on Google, but asking it to your mate involves getting up from your chair, crossing a room, and knocking to a door, you prefer Google.

If your mate is two meters away, it costs you 5 seconds to ask him, ***and*** (because not every one of us is a programming bible) 20 seconds to search it on Google. To *him*, it didn't cost 25 seconds, nor five. It cost 15 minutes. [9:pp. 25-26]

4.2. Organizing your work

Before starting my internship, I was requested a tentative Gantt chart and schedule for planning my work. I was doubtful. How could I estimate the time I should spend on each task, if we're talking about unknown technologies, and an integration with a closed-source product I've never seen before?

That's why overall planning is best left to whoever knows better about it, e.g. your project manager. However, experience plays a role in order to assess how much time a specific task will require as a part of a larger project; expect in your work as a programmer to be continuously queried about it by product managers and others.

Anyway, Gantt charts are mostly academical exercises, or they are useful when you're trying to track what a lot of different people are doing, with pre/post-requisites for each different task that have to be respected. Usually, it is quite straightforward to assess requisites for your own tasks, so these charts are just overkill when you've just to manage yourself.

Trying different methods, I've found that the best way to organize your internship is to:

1. write a clear **descriptive** spec of about five-ten pages, giving some stories about how the system should behave for an end-user;

2. have it approved, eventually cut or extended by your tutor;

3. have your tutor give you a rough plan;

4. prepare a **simple spreadsheet** with all tasks and for each of these, the following columns:

   a) the task priority;

   b) the number of hours spent on the task (initially 0);

   c) the number of hours allocated for the task originally – your initial estimation;

   d) the number of hours you estimate the task will take – your current estimation.

5. Remember to keep a margin for debugging and for testing. Usually, this can take up to 60% of the time. Debugging itself can usually be amortized inside each single task, like testing if you follow TDD. In this case, keep each task a little bit longer than what you'd do if it was just writing a feature, and keep a 10% margin at the end of the project for further integration testing.

Remember to keep a margin for debugging and testing

6. Writing documentation requires time. Write it as you go, where possible (especially make an effort of documenting code, or it'll easily slip out of control in just the two months you'll work as an intern). Code shouldn't be commented just for the sake of writing something; you should choose carefully *what* to document:

   a) Classes and methods want a description, even they are private. Shortly describe what a method does (not *how*), what the parameters mean, what it returns, eventually what exceptions it could raise.

   b) Inside methods, use clear variable names, and don't be afraid of typing if they are long. Fortunately Rails will help you in this. Document particularly difficult steps and kludges; if you keep your method shorts and well re-factored – for example, staying under twelve lines, on average, is an achievable goal in Rails; for other languages your mileage may vary – you shouldn't need to explain anything more.

Take care to update the spreadsheet daily

You should choose fine-grained tasks, and take care to update the spreadsheet daily. This will help you a lot in assessing where you're spending the most time, what to cut and what to keep when time runs low, and so on. This method is strongly advised by people knee-deep in project-management and software engineering such as Joel Spolsky [9:pp. 77-88].

4.2.1. Documentation

In the last years I've come to appreciate the quality of good documentation. I've been involved frequently in projects with a very high turn-over, in which no prior programmer left any documentation at all of what they wrote. The system was concocted upon personal taste and without any serious analysis. Needless to say, this quickly brought these projects down on their knees, with obscure and seldom used functions which you wouldn't risk removing because you did not know if another part of the program would break. Unfortunately, introducing new features did mean just increasing complexity, and reading hundreds of lines of code just for a couple of lines of a change.

You should write documentation as you go

Having no tests at all (see next paragraph) made things worse; you could not just remove something, run the test suite, see what broke and fix it again. Anyway, the biggest problem was still the documentation, especially in a dynamic language like Ruby: function signatures don't help you with type information. How do you know what a method returns, without documenting it or reading the full source code (if available)? How do you know exactly what parameters a variadic function takes – and they're really common, in Ruby and JavaScript?

But let's take a step back. How do you know what you're trying to build, the task you still need to tackle, and what you should **not** do, without a clear spec? Writing a specification is an art form, and we already discussed the importance of writing well back at page 58. You should make it nice for others to read, or else nobody will. See [9] for more on writing good specs.

More importantly, you should write documentation as you go, because going back later and adding some documentation inevitably makes you lose some details, forget some things, and results plain boring... therefore, nine times out of ten, you stop after five minutes since you started.

4.2.2. Testing

I made a mistake with Gallows, which is not having followed the test-first policy of TDD.

Even if testing a generator is inherently more difficult than testing another piece of code, I should have known better than postponing tests at the end just because the stakeholder demanded new features to be implemented. This simply doesn't work well.

Write your integration tests *before* the rest

Have care to implement your tests before or contextually to the rest of your code; enable continuous integration testing. You need to know exactly when something breaks, and why. The only way is to write a lot of functional and integration tests.

My two cents go into the direction: write your integration tests in a high level language (like, using Cucumber[25]), and have them in place *before* the rest of the code. Proceed to implement stubs and methods, and check the tests run as expected. Then complete the methods, and implement along with them also the unit tests.

It works. Really does. What you spend now writing tests will pay at least the double in debugging time for any sensibly-sized project. Checking that a new feature, maybe coming from an external contributor, doesn't break anything already in production, should just be the matter of launching the test suite (ideally, one command and a mug of coffee while it runs).

25 http://cukes.info/

5. Conclusions

All in all, the most important thing to keep in mind when facing an internship is that it is a method for each of the two stakeholders to know themselves. Both introspectively and one each other. It isn't just you, that must be overseen and evaluated. You may very well decide to leave that job at the end of your time as an intern, or to keep it. Just keep in mind that:

- if you're good, you'll certainly get a job relevant to your interest somewhere. Thus, don't take an employment if you aren't satisfied by the current position and conditions offered. Even if you're forced to accept the job (say, because you need some money to reach the end of the month, like everyone), actively research another working place.

- everyone is entitled to their own priority of values, of course; however, my opinion is that, once surpassed the minimum wage threshold – the one that allows you to live without anxiety – getting on well with other colleagues, a stimulating environment and . If you don't like the majority of the people you work with, and **especially** if you don't like the management (maybe because they act unethically – even criminally –, they take decisions in ignorance, they don't value your work or they are fossilized in their ideas), just leave. The damage you, a worker, can do this way is almost always much greater to that they can perpetrate to you; if they're not aware of this before you leave, then they *deserved it*. People aren't interchangeable, like tools or other objects.

- in the same way, pay due respect to whoever pay your wages, directly (your boss) or indirectly (your colleagues). Being aware how much you're worth mustn't become an excuse for greed, malevolence or pedantic behavior. Pride is very different from sufficiency.

Given these notes and your experience, brace yourself and jump into the job market. It'll be difficult, tiresome and very intimidating at first, but also an immense source of satisfaction.

A good programmer will be able to experience the same feeling as an artist when finishing a painting, or as an artisan who knows her craft. Look at your hands, and simply imagine: what may I give them to build, now?

Appendix A. Very small glossary of terms

I included here a very small glossary, because I think that most terms are already explained in the text when used, or as a programmer you're already familiar with them.

AJAX

Asynchronous JavaScript And XML: a group of technologies employed to allow the client-side dispatching of messages to a server, and awaiting asynchronously for the result. This allows the dynamic part of the web to interact with a more event-driven style of programming.

JSON

JavaScript Object Notation is a lightweight data-interchange format, which is easily parseable and human-readable. It's mime-type is normally `application/json`.

POV

Abbreviation for Point Of View.

RESTful

REST stands for REpresentational State Transfer and is a style of software architecture; among other things, it uses mandates clear and predictable paths for accessing resources via HTTP verbs.

XHR

Abbreviation for XmlHttpRequest, a JavaScript method's name that's the inheritance of ancient times – it isn't limited just to XML requests. It is the function that received the most hype inside the AJAX family of technologies.

Appendix B. Bibliography

1. Arlow, J. e Neustadt, I. *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design*. Addison-Wesley Professional, 2005.
2. Brooks, F.P. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*. Addison-Wesley Professional, 1995.
3. Fisher, A. *Critical Thinking*. Cambridge University Press, 2001.
4. Flanagan, D. e David, F. *JavaScript: The Definitive Guide*. O'Reilly Media, 2006.
5. Graham, P. *Hackers & Painters: Big Ideas from the Computer Age: Essays on the Art of Programming*. O'Reilly Media, Inc., 2004.
6. Patterson, K., Grenny, J., McMillan, R., Switzler, A., e Covey, S.R. *Crucial Conversations: Tools for Talking When Stakes are High*. McGraw-Hill, 2002.
7. Rappin, N. *Professional Ruby on Rails*. Wrox, 2008.
8. Sommerville, I. *Software Engineering*. Addison Wesley, 2004.
9. Spolsky, J. *Joel on Software: And on Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and Managers, and to Those Who, Whether by Good Fortune or Ill Luck, Work with Them in Some Capacity*. Apress, 2004.
10. Thomas, D. e Fowler, C. *Programming Ruby: The Pragmatic Programmers' Guide, Second Edition*. Pragmatic Bookshelf, 2004.

Appendix C. Credits and tools used

This thesis has been typeset on an IBM Thinkpad G40 laptop, running *Ubuntu GNU/Linux 9.10* and *OpenOffice.org 3.1.1*. For downloading Ubuntu for your computer, head to http://ubuntu.com/.

The UML diagrams have been generated using the wonderful *BOUML editor* by Bruno Pagès (http://bouml.free.fr/). Other images have been edited, cropped or created using *The Gimp* and *Inkscape*. The conceptual maps at the beginning of this thesis are made employing *Labyrinth* (http://www.gnome.org/~dscorgie/labyrinth.html). The bibliography has been generated by using *Zotero* (http://www.zotero.org/).

The project itself was developed using both *GNU Emacs 2.23* (http://gnu.org/software/emacs/) and *NetBeans 6.8m1* for debugging (http://netbeans.org/). *GNOME Planner* was used to prepare some Gantt charts about the progress of the work-plan (http://live.gnome.org/Planner). Syntax highlighting in this document courtesy of Highligt by Andre Simon (http://www.andre-simon.de/). For debugging the JavaScript and thus the ExtJS-related code, the excellent *Firebug* Firefox extension by Parakey, Inc. (http://getfirebug.com/) was used.

Ruby 1.8.7 is released under the Ruby license (http://ruby-lang.org/). The *Ruby logo* is a work of Yukihiro Matsumoto and is licensed under the CC-by-sa 2.5. *Ruby on Rails 2.3.2* is released under the MIT license (http://rubyonrails.org/) and its name is a registered trademark of David Heinemeier Hansson. The *Rails logo* was created by Kevin Milden and is distributed under the BY-ND Creative Commons License.

ExtJS 3.0 is released both under the GPL and under a commercial restrictive license (http://extjs.com/). The *Silk icon set* is released under the CC-by 2.5 by Mark James (http://famfamfam.com/). I used two plug-ins by Jozef Sakalos: *RowActions* and *GridSearch*.

If you feel you deserve to figure in these credits, please e-mail me at matteo@member.fsf.org and I'll take care to put out the needed errata.