Universitat Autònoma de Barcelona

escola d'enginyeria

# 5303: XML-BASED INFORMATION MANAGEMENT ENVIRONMENT FOR PE TECHNOLOGIES

Memòria del Projecte Fi de Carrera
d'Enginyeria en Informàtica
realitzat per
***Bernat Labarías Comaposada***
i dirigit per
***Dr. Lluís Antoni Terés Terés***
Bellaterra, 22 de gener de 2013

El sotasignat, Dr. Lluís Antoni Terés Terés
Professor/a de l'Escola Tècnica Superior d'Enginyeria de la UAB,

**CERTIFICA:**

Que el treball a què correspon aquesta memòria ha estat realitzat sota la seva direcció per en Bernat Labarías Comaposada

I per tal que consti firma la present.

Bellaterra, Gener de 2013

Signat: Lluís Antoni Terés i Terés

## Abstract

Printed electronics is an emerging concept in electronics manufacturing and it is in very early development stage. The technology is not stable, design kits are not developed, and flows and Computer Aided Design (CAD) tools are not fixed yet. The European project TDK4PE addresses all this issues and this PFC has been realized on this context.

The goal is to develop an XML-based information system for the collection and management of information from the technology and cell libraries developed in TDK4PE. This system will ease the treatment of that information for a later generation of specific Design Kits (DK) and the corresponding documentation.

This work proposes a web application to generate technology files and design kits in a formatted way; it also proposes a structure for them and a database implementation for storing the needed information. The application will allow its users to redefine the structure of those files, as well as export and import XML files, between other formats.

## *Resum*

Printed electronics *és un concepte emergent en la fabricació de circuits electrònics, però encara es troba en les primeres etapes de desenvolupament. La tecnologia no està estabilitzada, els kits de disseny no estan desenvolupats i els fluxes i eines CAD de disseny encara no s'han fixat. El projecte europeu TDK4PE està abordant tots aquests temes i es en aquest context que s'ha realitzat el present PFC.*

*L'objectiu es desenvolupar un sistema d'informació basat en XML que faciliti la recol·lecció d'informació provinent de la tecnologia i les biblioteques de cel·les desenvolupades en el TDK4PE per tal de facilitar el seu tractament per una posterior generació de kits de disseny específics i la documentació corresponent.*

*Per tal d'assolir aquests objectius s'ha desenvolupat una aplicació web que permetrà generar fitxers tecnològics i kits de disseny amb un format determinat; s'ha proposat també un exemple per aquesta estructura i una implementació de base de dades per a poder guardar la informació necessària. L'aplicació permetrà regenerar aquesta estructura, així com exportar i importar fitxers XML juntament amb d'altres formats.*

## *Agraïments*

*Primerament agraïr a en Lluís la paciència i els consells aportats en aquest projecte; la seua visió aguda m'ha proporcionat un esquelet sòlid sobre el qual aquest projecte ha anat fent passos ferms i sense la qual hagués estat molt difícil la seua realització. En segon lloc agraeixo l'ajuda, la paciència i les grans idees del meu company de carrera i actual integrant del grup ICAS del CNM, en Cesc. Ha estat una gran soport en els moments durs del projecte. També agraeixo la seua aportació en les tasques de desenvolupament i síntesi de figures, entre d'altres coses, a en Jofre, la seua visió artística i toc mestre amb l'Inkscape m'han proporcionat una gran ajuda. Agraeixo a tots els companys de carrera que m'han donat consells executius, com ara l'Arturo o l'Albert; la seua experiencia m'ha anat molt bé. Finalment agraïr a la resta de companys del CNM que han aportat alguna cosa al projecte i a la gent del TDK4PE que m'ha recolzat, encara que sigui mínimament. A la meua mare i la meua germana, sobretot per la paciencia que han tingut amb mí.*

# Acronym List

**ADC** Analog-to-Digital Converter

**API** Application Programming Interface

**CAD** Computer-Aided Design

**DRC** Design Rule Check

**EDA** Electronic Design Automation

**ERC** Electrical Rule Check

**FOLAE** Flexible Organic Large Area Electronics

**GPL** General Public License

**HTTP** HyperText Transfer Protocol

**IC** Integrated Circuit

**ICAS** Integrated Circuits And Systems group

**LVS** Layout Versus Schematics

**MIT** Massachusetts Institute of technology

**MVC** Model-View-Controller

**NRE** Non-Recurring Engineering

**OE** Organic Electronics

**OOP** Object Oriented Programming

**ORM** Object-Relational Mapping

**OTA** Operational Transconductance Amplifier

**OTFT** Organic Thin Film Transistor

**PDK** Process Design Kit

**PE** Printed Electronics

**PHP** Personal Home Page (programming language)

**Si** Silicon

**TDK** Technology & Design Kit

**XML** eXtensible Markup Language

**XSD** XML Schema Document

**XSL** eXtensible Stylesheet Language

**XTR** eXTraction Rules

# Contents

# List of figures

# List of tables

# List of listings

# 1 Introduction and Background

The *Institut de Microelectrònica de Barcelona* (IMB-CNM CSIC) and the UAB's CAIAC group together with other partners (ENEA, Flexink, InfiniScale, Phoenix, Sensing Tex, Technische Universität Chemnitz, Universidade do Algarve and 3D-Micromac) are participating in a European project TDK4PE which is mainly devoted to emerging Flexible Organic Large Area Electronics (FOLAE) technologies development and application by addressing three complementary activity areas.

1. Specific Printed Electronics (PE) technologies;
2. Physical design kits (PDK) and Electronic Design Automation (EDA) tools to facilitate the design for such technologies;
3. Application demonstrators' development using above tools and technologies.

This PFC will focus mainly on the development of web-based environment for the compilation and management of the mandatory information for the technology. This technology information will be stored in an eXtensible Markup Language (XML) file which will be used by the future CAD tools for PE design, so that a standard open format is being used and that makes the further deployments possible.

## 1.1 Printed Electronics

During the past years this technology appeared under several names, such as Printed Electronics (PE), Organic Electronics (OE) or Flexible Organic Large Area Electronics (FOLAE). All of these names refer to a combination of new processes and materials for the fabrication of devices and electronic systems. These new processes adapt existing printing techniques to electronic fabrication [1][2].

### 1.1.1 What is Printed Electronics?

Printed electronics is a term that defines the printing of circuits on media such as paper, plastic and textiles, but also on a large number of potential media. The recent rapid development of PE technology is motivated by the promise of low-cost, high-volume, high-throughput production of electronic components or devices which are lightweight and small, thin and flexible, inexpensive and disposable. PE is not a substitute for conventional silicon-based electronics, but it opens a new world of low-cost printed circuits based on conductive, semi-conductive and dielectric printed materials aiming at high-volume market segments where the high performance of conventional electronics is no required, as can be seen in Figure 1.1

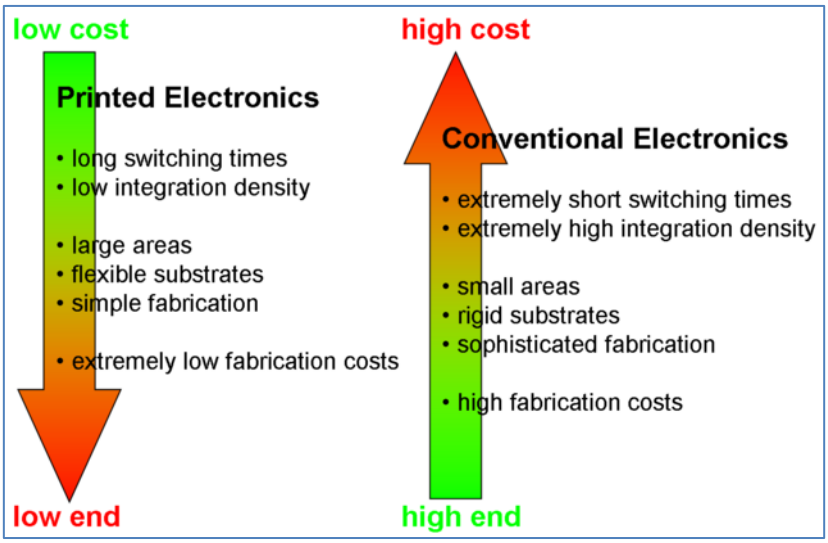



**Figure 1.1. PE review**

The basic elements of printing technologies, illustrated on Figure 1.2 are:

- Printing engines.
- Materials or inks for printing.
- Substrates to be printed on.

Interactions between different parameters of those elements for each specific technology suite (inks + substrate + printer) shall be identified and characterized in order to provide the related technology information.

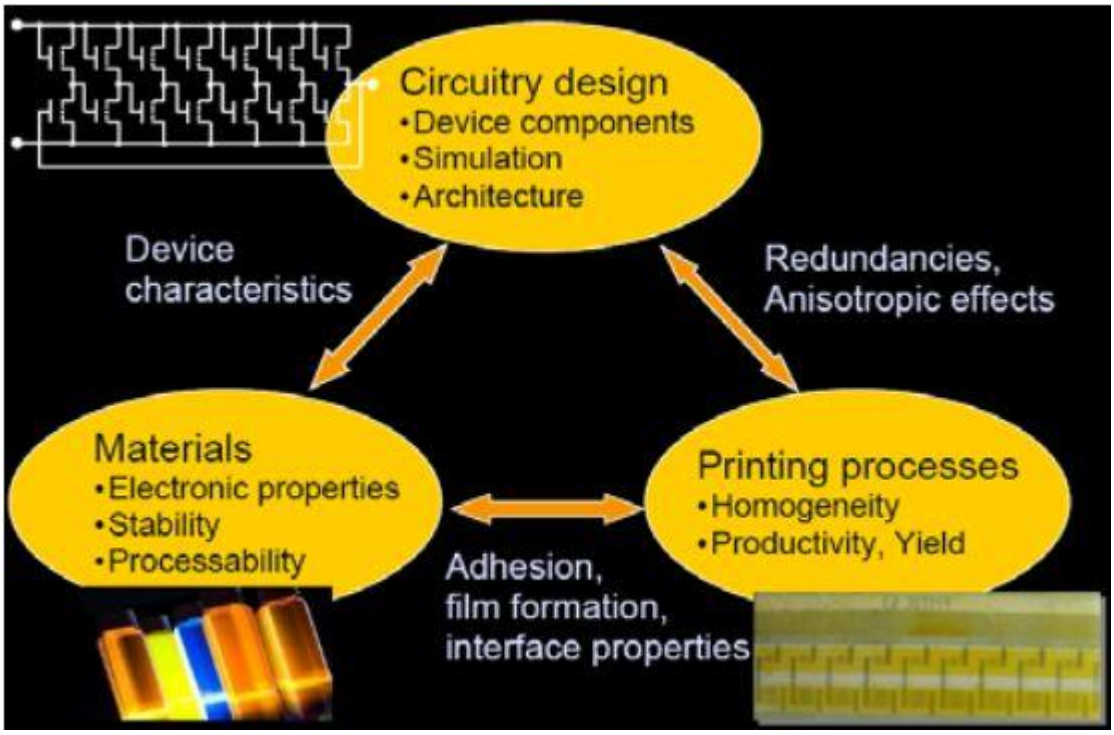



**Figure 1.2. Relationship of basic printing technologies [BASF S.E.]**

This section will present an overview of the current state of the art about flexible organic large area electronics with a short review of its fundamental areas: materials (inks & substrates), printing systems (mainly focused on inkjet) and circuit design (from devices to Electronic Design Automation (EDA) flows and tools); and the relationships between those three components of the printed electronics equation [Figure 1.2].

## Materials

In the field of applied materials a lot of research has been done about physical and chemical properties of the printed materials and devices (both active and passive) in order to obtain optimal organic semiconductor materials and devices performance and stability. Organic Thin Film Transistors (OTFTs) are the basic building blocks for flexible integrated circuits and displays. A schematic structure is shown on Figure 1.3.

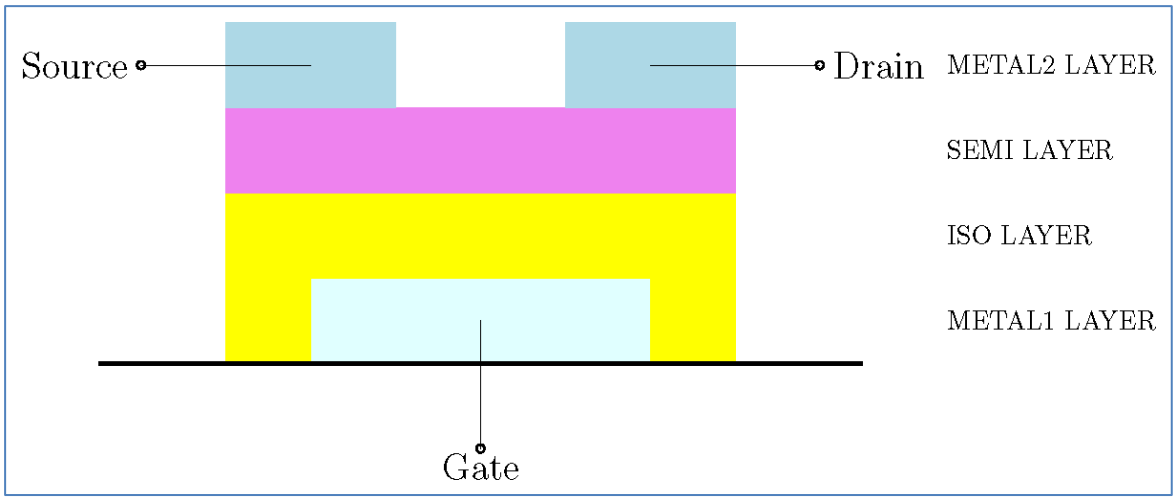



**Figure 1.3. Schematic view of an OTFT**

To make OTFTs, materials ranging from conductors (for contact electrodes), semiconductors (for active channel materials), to insulators (for gate dielectric layers) are needed. There are two types of organic semiconductor based on the type of majority charge carriers: p-type (holes as major carriers) and n-type (electrons as major charge carriers). By characterizing and understanding the conditions that lead to different printed line morphologies, electronic devices performance can be improved, thereby increasing yield. For example, in a bottom-gate OTFT configuration, the gate area must be as narrow and as uniformly flat as possible; and the source and drain contacts need soft edges with a relatively small and controlled separation although their width is less critical.

### Printing processes

This section shows different printing methods used in PE [3]:

**Screen Printing:** There is a mesh with several openings that is put into contact with the substrate. Then the ink is pressed through the openings of the screen and is transferred to the substrate. Show on Figure 1.4-E

**Offset Printing:** This is the most widespread printing technique. It's based on bringing printing plate into contact with ink. This printing plate has regions ink-accepting and ink-repelling areas, and from there to the substrate via and intermediate blanket cylinder. Illustrated on Figure 1.4-B

**Gravure Printing:** This is a very high-throughput technique, based on a metal cylinder with an etched set of patterns, which are filled with ink and then brought into contact with the substrate. Pad printing is a much lower-throughput technique, related to gravure and based on a flexible silicon stamp that picks ink from a cliché and transfers it to the substrate. These techniques are shown on Figure 1.4-A and 1.3-D

**Flexography:** This technique is based on a flexible cylinder with a pattern of protruding regions. These regions are soaked with ink and transferred to the substrate. Shown on Figure 1.4-C

**Inkjet Printing:** Inkjet printing is the most widespread technique in the home and desktop printing markets. It is based on an ejector, most times based on a piezoelectric transducer, which ejects drops to the substrate. This is illustrated on Figure 1.4-F

Not only printing method is important for a successful system manufacture. As seen on Figure 1.4, inks and substrates also play an important role to the final result. Each of those three elements must be compatible with the printing system, and the ink must wet the substrate, but no run off before drying. Also the printing method must be capable of using the ink [2][4].

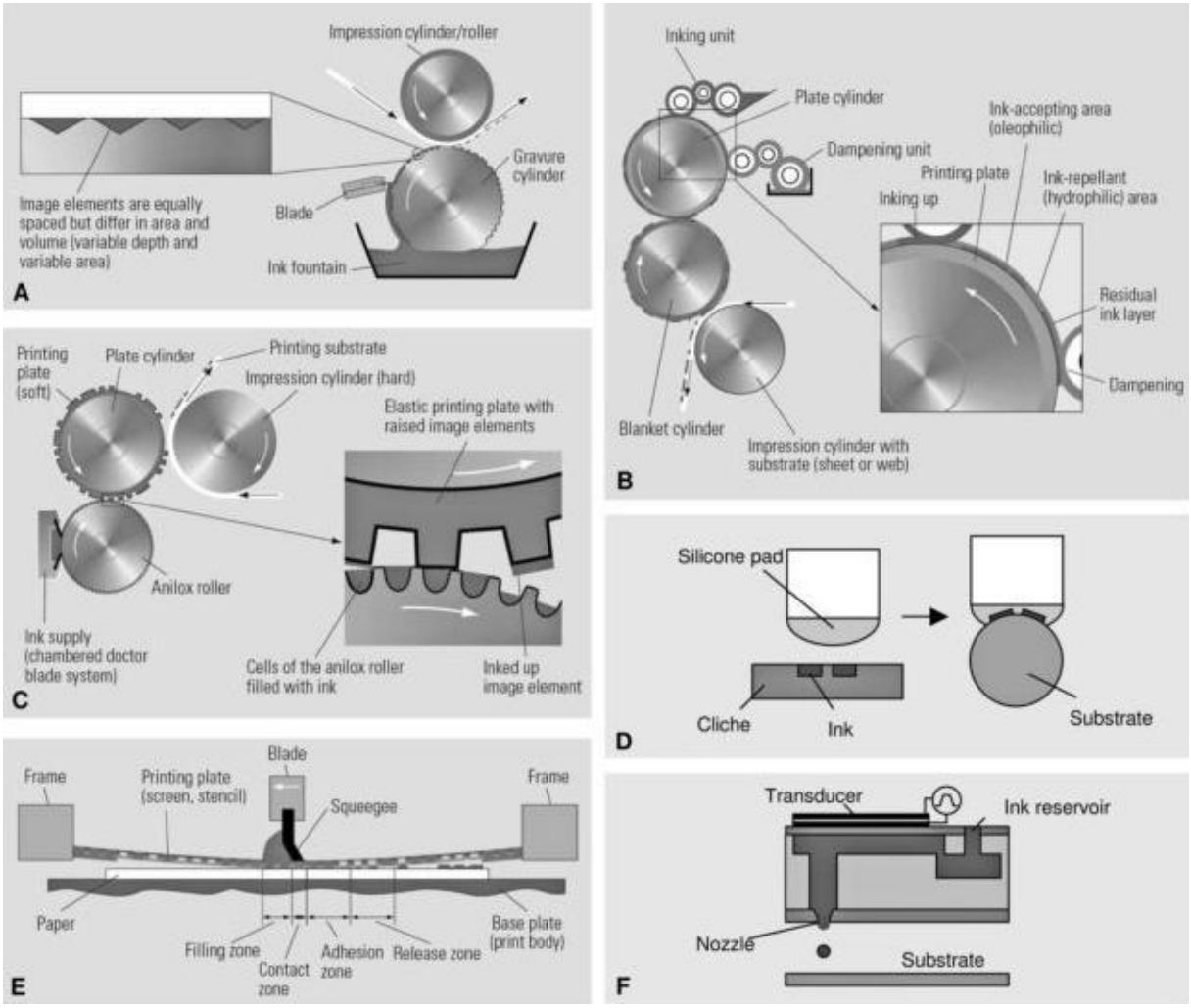



**Figure 1.4. Several printing methods. (A) gravure, (B) offset, (C) flexography, (D) pad printing, (E) screen printing, (F) inkjet printing.**

The inkjet printing technology is the most promising alternative to photolithographic and masking patterning technologies enabling fabrication of cost-effective electronics. Direct patterning by inkjet printing can be integrated in many existing electronics production lines at remarkably low investment costs and by enabling the usage of soluble function materials for active and passive layers of electronic devices and systems. The inkjet printing of microelectronic devices has been widely investigated. A Roll-to-Roll (R2R) inkjet process is the next necessary step needed to scale-up the use of digital printing technology in production lines.

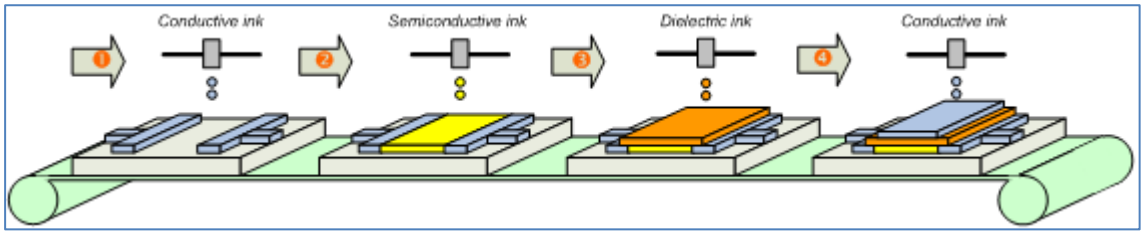



**Figure 1.5. Envisioned inkjet R2R printing manufacturing flow.**

Several challenging steps have to be addressed in R2R inkjet production workflow as sketched above:

- Layer registration of different printing layers is a key challenge since it affects directly device performance and systems, devices yield and reproducibility.
- Printing resolution has to be improved to minimum of 50 µm in order to achieve minimum required device performances.
- Multi-head inkjet printer system has to be developed in order to enable separately-successively-simultaneously printing semi-conductive, conductive and insulating inks. This can increase printing process speed.
- Drying/curing systems have to be developed accordingly to the chose materials/solvents.
- The solvents used in different inks need to be compatible when printing subsequent layers and yet be environmentally safe for use in large-scale production.

## Circuitry design

According to semiconductors industry we can divide ASIC circuit design flow into two parts: technology independent (or front-end) and technology dependent (or back-end). For inkjet based FOLAE systems, these two parts suggest two different approaches for system design and implementation. With regard to the front-end, at the current level of technology, a top-down design approach would be valid, and it will result in library cell requirements for cell-based design. This front-end can be considered quite well covered by current silicon microelectronics EDA tools and methodologies and therefore it can be almost directly reused. In contrast, for the back-end, technology dependence for library cells and design kits construction requires a bottom-up approach according to the new fabrication processes involved resulting in a design kit for full-custom design.

Today, no dedicated software kit exists for PE. Current state of the art is to design a layout with existing tools like *CleWin*, *GLADE*, *Cadence*… A conversion from the layout to

the actual printer files is performed either manually, by simple home-made software tools or provided by machine manufacturers. Compared to the electronics world or even MEMS, PE technology and design kits are in its infancy, or even worst they are not yet born.

### 1.1.2 Pros and Cons

As a review of the previous sections of this chapter we can state a list of advantages and disadvantages of PE respect conventional Microelectronics.

Its main advantages are:

- Low fabrication costs: As the Non-Recurring Engineering (NRE) costs are reduced or almost negligible, we don't need to produce large series of each prototype to amortize costs and it's quite easier to afford the production costs. Large fabrications are no longer needed and also they have become simpler. Building a "PE-house" instead of a foundry is another point where costs are notably lowered (~1,000 times cheaper).
- Adaptability to substrate: One of the principal aims of PE is making possible to embed digital microsytems in organic or flexible and large surfaces, so that the requirement of silicon substrate is avoided. A new wave for microelectronics is opened as one of the claims for PE is to be capable of including digital circuits on flexible/organic substrates such as textile, plastics…
- Large area printing: This is another possibility that PE brings us. Large area designs integration was not possible with conventional electronics and also it's opposite with one of its main features. With PE we can adapt the throughput of our design by changing the printing devices.

But it also has disadvantages:

- Lower resolution: The circuits are going to be implemented by printers with its own headers that will deposit the ink onto the surface. Those headers have a higher size than the drives use to build masks on conventional micro-electronics, so the shapes that will be possible to build shall be bigger. Another point are the fluidic effects derived from inks, due to that effects the final shape that is going to be printed may vary from the desired one. That's another part where we lose resolution. [6]
- Lower performance: Since the components of the circuits are going to be printed, and not manufactured under exhaustive and expensive technologic processes, its durability and performance is lowered. Throughput is highly increased by lowering its fabrication cost and thus the performance, but the level is enough for a wide range of applications.

## 1.2 Design flows and EDA tools

### 1.2.1 PE design flows

PE technologies aim to complement expensive Integrated Circuit (IC) foundries with a cheaper printing house concept providing a lot of different "PE – foundries" with specific technological processes to address different applications. In order to facilitate and diffuse the fabrication offers, printing houses and application have to be linked through a design kit, which can be seen as the personalization of the EDA tools for a given technology that abstract fabrication details and enables technological access to a large number of designers and end users.

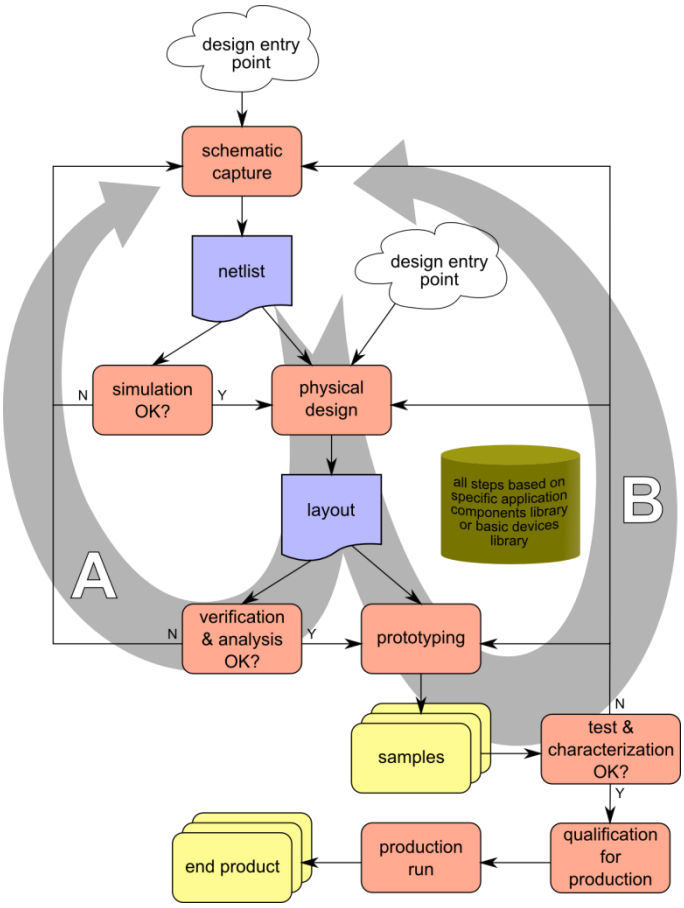



**Figure 1.6. Basic device and component level design flow.**

Design flow used for microelectronics design can be seen on Figure 1.6. Figure shows two design entry points, that is, that flow contemplates geometric layout or schematics. This is an interactive flow, after each step, the design is tested, and if the tests are not correct, a new iteration is started. Once the results are satisfactory, the flow advances to the next level. This flow has been modified to offer two possibilities:

- A strongly simulation based flow (loop A) for expensive prototyping technologies like microelectronics.
- A direct way for prototyping for inexpensive technologies like PCB design (loop B).

Loop A from Figure 1.6 is recommended for traditional silicon-based microelectronics, because it claims a strong simulation and verification of the design before prototyping, due to its high cost. It provides also a good starting point for developing a PE compatible design flow (loop B) where the prototyping is less problematic due to its low cost.

As for the evolution of the abstraction levels for the design implementation, a timeline is illustrated onFigure 1.7. The figure shows the evolution, from full custom design to Hardware Description Languages (HDLs), passing through P-cells or standard cells.

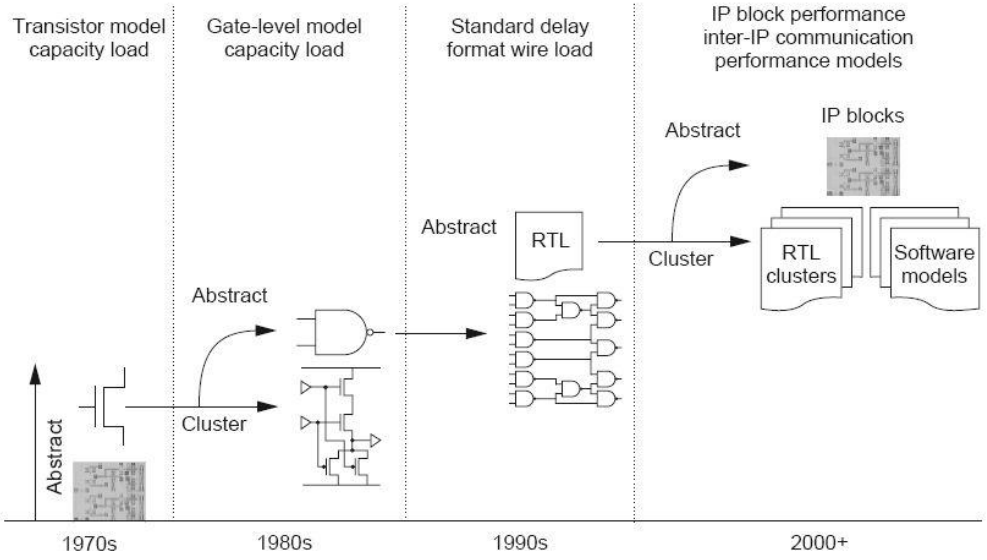



**Figure 1.7. Raising design level abstraction.**

Many of the current research of PE covers the fabrication and characterization of electrical devices (both active and passive) [7][8] or improvement to current printing techniques to allow more resolution [9]. But research does not pay much attention to the definition of design flows or tools to aid the designer.

The development flow for PE could be the one illustrated on Figure 1.8 which shows different low levels to be addressed in order to develop, characterize and fix a specific technology for a specific application domain. These levels are shown in a bottom-up approach. The figure covers the physical level, where the materials, inks, substrates and processes are applied characterized and modeled; the device level where basic building blocks and devices are designed, prototyped, characterized, and modeled and the components level, where more complex cells and macro cells are developed, characterized, modeled, and characterized.

Based on the physical level on Figure 1.8, the physical and backend flow (see Figure 1.9) can be extracted. This flow shows the environment needed for basic operations at this level:

- An appropriate layout editor.
- A basic Design Rule Checking (DRC) checker.
- A layout to printer tool, performing basic compensations and format conversions.
- A print-out emulation tool to generate 2D/3D models from a printing file before sending it to the printer.

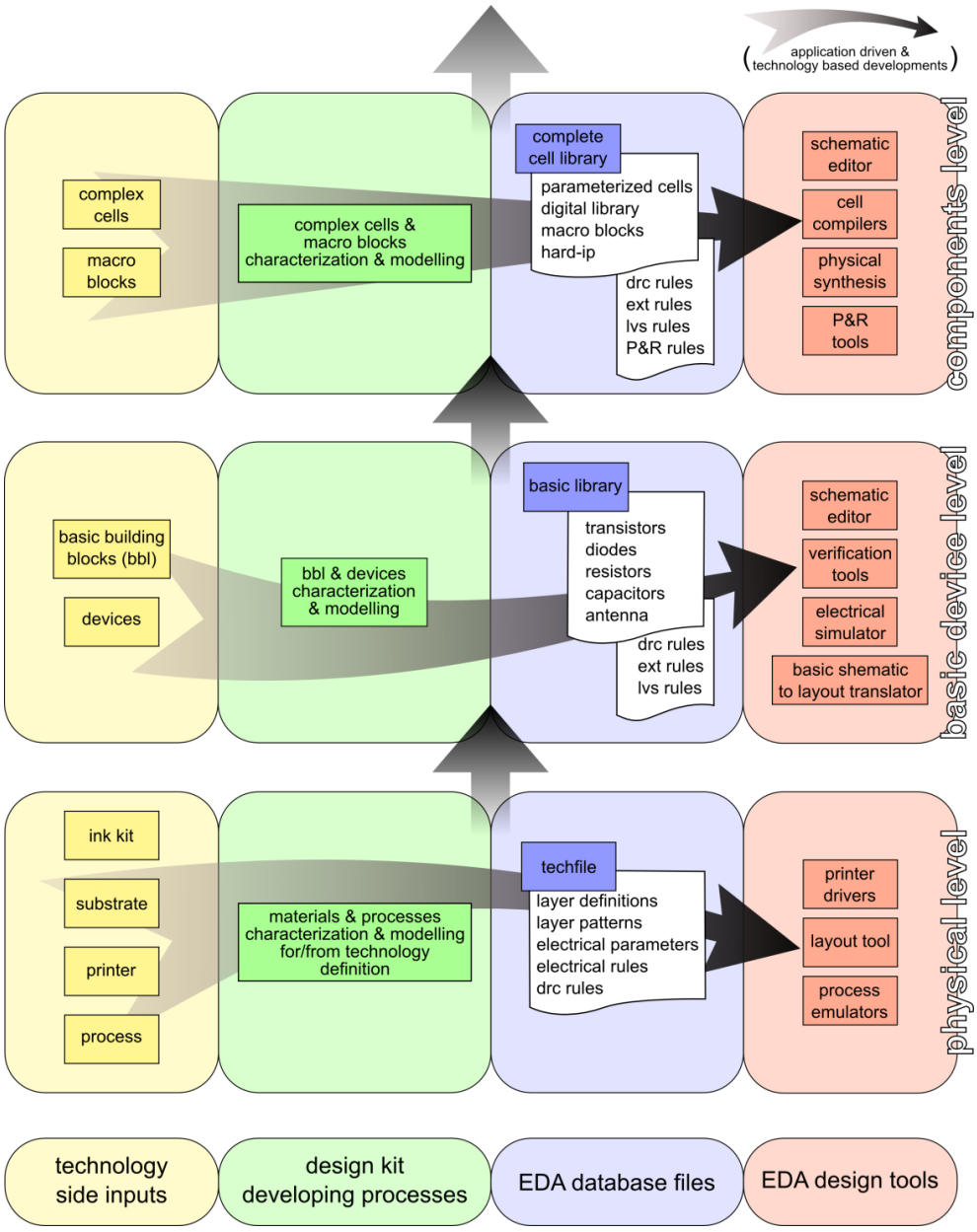



**Figure 1.8. PE and design kits methodology and development flow.**

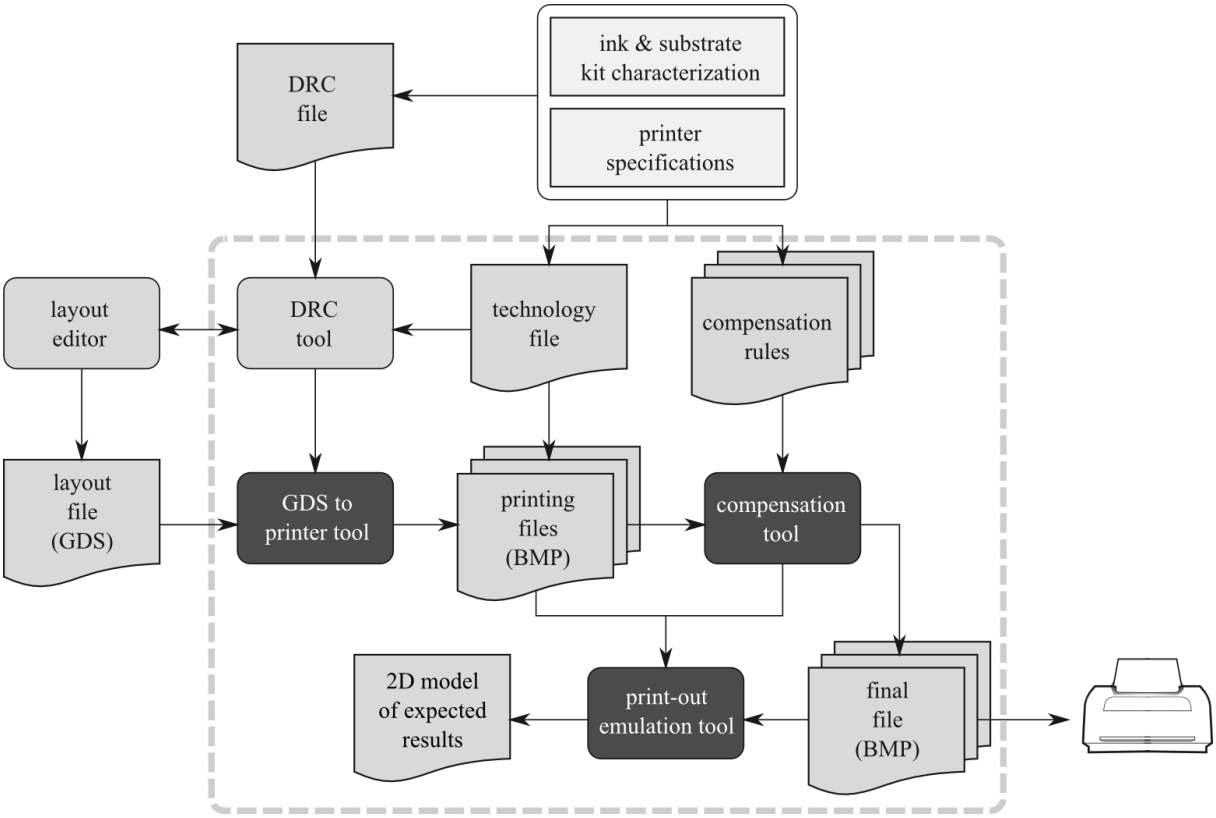



**Figure 1.9. Layout to Printing backend processes.**

### 1.2.2 EDA tools

This section shows information about different available CAD tools, oriented to layout IC full-custom design, which have been tested by Integrated Circuits And Systems (ICAS) team. One of these should be adapted to become the final tool for the main project TDK4PE. Given the state of development of this main project, the partners involved, and the goals to be achieved, the desired characteristics of the tool should be the following:

- Simplicity to use, to install, and maintain.
- Cheap, free if possible.
- DRC tool and Extract tool (XTR).
- Arbitrary geometry.
- Import tool from different formats (GDS2, DXF) and export tool to standard format (GDS2, CIF).
- Possibility of generating scripts for setting up purposes.
- Parameterized cells.

Next tables show an overview of different tested CAD tools with the characteristics shown above.

## Commercial tools

| Name | Complex | Price | DRC/XTR | Geometry | Import | Scriptable | P-Cells |
|---|---|---|---|---|---|---|---|
| Cadence | High | 1,800 + 1,400 | Yes | Arbitrary | DEF, LEF, EDIF, CIF GDS2, VHDL, Verilog, Spice, OpenAccess | Yes | Yes |
| Mentor | High | Free + 1,000 | Yes | Arbitrary | - | Yes | Yes |
| Synopsys | High | 480 + 800 | Yes | Arbitrary | - | Yes | Yes |
| Tanner | Medium | 900 + 1,250 | Yes | Arbitrary | CIF, EDIF, GDS2, VHDL, Verilog | Yes | Yes |
| CleWin/ Mask Engineer | Medium | 1,200 + 1,050 | Yes | Arbitrary | GDS2, CIF | Yes | Yes |

**Table 1.1. Commercial CAD tools overview.**

The prices shown in Table 1.1 are specified in Euros and provided by *Europractice*. They refer to the cost of a unique user license + maintenance for one year for the category of research center. The prices for more than one user or for the commercial category may vary very much.

## Open/free tools

| Name | Complex | Prize | DRC/XTR | Geometry | Import | Scriptable | P-Cells |
|---|---|---|---|---|---|---|---|
| Toped | Not much | Free | DRC | Manhattan | GDS2, OASIS, DXF, CIF | Yes | Yes |
| Glade | Yes | Free | Yes | Arbitrary | GDS2, OASIS, DXF, Verilog, LEF and DEF | Yes | Yes |
| Electric | Not | Free | DRC limited | Manhattan | GDS2, OASIS, DXF, Verilog, LEF and DEF | Yes | No |

**Table 1.2. Free/open CAD tools overview.**

After that review it seems that the most suitable software could be Glade, it gathers almost every desired characteristics for free. The schematics low complexity of the designs that will be performed in TDK4PE doesn't make necessary that the tool integrates some schematic editor or a simulator. This project is addressing Glade, every time that a real application example is shown. The main project TDK4PE is also taking Cadence and CleWin into account. Cadence will probably be adapted to PE technologies, since it has been widely used in enterprise and academic domain. CleWin has been developed by *Phoenix Software,* one of the partners of TDK4PE project, so it will be referenced for future developments since it is heavily linked with the main project.

## 1.3 About XML & PHP frameworks

### 1.3.1 XML Programming interfaces

This section is not addressing the current use of XML files, but the existence of API for XML. Since this project provides an XML-based environment, it is assumed that XML format is widely-used by current computer applications. Internet has also promoted the XML use; sets of rules have been created for that purpose [10]. The design goals of XML include "it shall be easy to write programs which process XML documents." [11] Despite this, the XML specifications contain almost no information about how programmers might go about doing such processing. A variety of Application Programming Interfaces (APIs) for accessing XML have been developed and used, and some have been standardized. Existing APIs for XML processing can be identified within the following categories.

- Stream-oriented APIs accessible from programming language, for example *SAX* and *StAX*.
- Tree-traversal APIs accessible from programming language, for example *DOM*.
- XML data binding, this provides an automated translation between XML documents and programming language objects.
- Declarative transformation languages, for example *XSLT* and *Xquery*. [12]

Stream-oriented APIs require less memory and, for certain tasks which are based on a linear traversal of an XML document, are faster and simpler than other alternatives. Tree-traversal and data-binding APIs typically require the use of much more memory, but are often found more convenient for use by programmers; some include declarative retrieval of document components via the use of *XPath* expressions.

*XSLT* is designed for declarative description of XML document transformations, and has been widely implemented both in server-side packages and Web browsers. *XQuery* overlaps *XSLT* in its functionality, but is designed more for searching of large XML databases.

#### *SAX* (Simple API for XML)

*SAX* is an event-based sequential access parser. It was originally a Java-only API; nowadays there are versions for several programming languages. It provides a mechanism for reading data from an XML document that is an alternative to that provided by Document Object Model (*DOM*). Where the *DOM* operates on the document as a whole, *SAX* parsers operate on each piece of the XML document sequentially. Its main advantage over *DOM*-style parsers is the memory use, as they parse XML sequentially they only report each parsing event as it happens, and normally discard almost all information after that. *SAX* parsers are in general faster processing documents than *DOM* parsers. [13]

#### *DOM* (Document Object Model)

The *DOM* is a cross-platform and language-independent convention for representing and interacting with objects in *HTML*, *XHTML*, and *XML* documents. *DOM* parsers usually load the entire XML file and build a tree object with the content of that file. Objects in the *DOM* tree may be addressed and manipulated by using methods on the objects and this

is specified on its API. *DOM* has been standardized by the *w3c* [14] and that fact has promoted its development and many libraries have been published that implement *DOM*. Examples of these libraries can be *libxml2*, *MSXML*, *Xerces*, *JAXP*; the class used on this project for the XML generation *DOMDocument* is a PHP Class that executes functions from the *libxml2* functions. *DOM* parsers require more use memory, but besides that they can validate quickly a loaded document since they can access quickly in any part of that document.

### XML data binding

XML data binding is the binding of XML documents to objects designed especially for the data in those documents. This allows applications (usually data-centric) to manipulate data that has been serialized as XML in a way that is more natural than using the *DOM*. Based on this mapping, the product can then create objects from XML documents ("unmarshalling") or serialize objects as XML ("marshalling"). The main limitation that these parsers present are the loss of information, for example the structure, sibling order and comments and processing instructions are not binded to memory [15]. Some implementations are *JAXB* or *JSON*.

### XSLT and transformation languages

As its own name states, transformation languages transform XML documents into other XML documents, or other objects such as *HTML* for web pages, plain text or into XSL formatting objects which can be converted to *PDF* or *PostScript*. As like as *DOM*, *XSLT* has also been standardized [16]. Typically, input documents are XML files, the original document is not changed; rather, a new document is created based on an existing one. The flow of these transformations is applying an XSLT code to an XML input, and XSLT processor generates a resulting document with a new format. Some implementations are *libxslt*, *Saxon*, *Xalan*.

For this project, an implementation of a *DOM* parser, the PHP class *DOMDocument* has been used for the generation and parsing of the XML and XSD files. Some XSLT transformations have been applied also, for the output of the files.

### 1.3.2 PHP frameworks

This section will present an overview of the current state of art for PHP frameworks. After showing an option list we are going to analyze their features to ensure a good choice for developing the web application for this PFC. First of all we define frameworks.

Frameworks in general include support programs, compilers, code libraries, APIs, and other tools. Most of times different coding software provides support for these frameworks, so it helps the task of linking different components inside a project. A PHP framework defines a conceptual structure, most times by modules, for supporting and facilitating the task of developing a new PHP application, in this case. The following are key distinguishing features that separate them from normal libraries [17].

- Inversion of control: In a framework, unlike in libraries or normal application, the overall program's flow of control is not dictated by the caller, but by the framework [18].
- Default behavior: A framework has a default useful behavior; this default behavior must not be a series of no-operation.
- Extensibility: A framework can be extended by the user usually by selective overriding or specialized by user code to provide specific functionality.
- Non-modifiable framework code: The framework code, in general, is not allowed to be modified, excepting extensibility. Users can extend the framework, but not modify its code.

Due to its primary goal, that is facilitating the development of applications, PHP frameworks have exploded in popularity over the past few years. Nowadays we have lots of different PHP frameworks to choose, the problem for us is to decide which framework inside all options is right for us. First of all we show a list of the most used frameworks, choosing one of these is a good point for us, because a great community using same framework means having a lot of documentation.

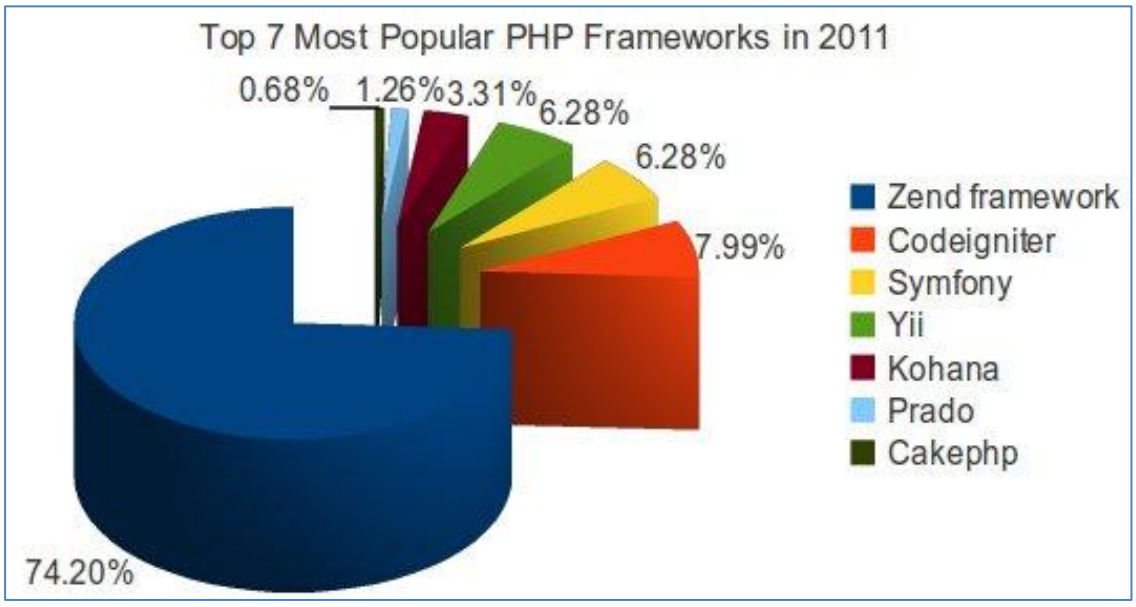



**Figure 1.10. Top 7 most popular PHP frameworks (source Google keyword) [19]**

The data from Figure 1.12. is taken from Google keyword tool that provides the amount of searches for that PHP framework in one month time frame worldwide. The higher the searches for that framework, the more popular will be the PHP framework. After looking at that graphic we get the following list, our election will be one of these so we make a review for each one of them.

### Zend framework

The *Zend Framework* is for more experienced developers and those building enterprise-level applications. The framework is highly modular, meaning you can use as little or as much of the *Zend* code as you need. Several libraries can easily be extracted for stand-alone use. With *Zend*, you're not tied to the Model-View-controller (MVC) pattern (though it is certainly an option), and there is a lot of built-in support for tasks such as integration with existing web services, localization and extensive unit testing.

*Zend* requires PHP 5.1.4+ but has multiple configuration options and excellent documentation. It's a framework with corporate backing and a strong community. You'll get long-term stability and fairly quick integration of new features, not to mention security updates tested by a team of professionals. While the *Zend* framework may be a large overhead for small projects, if you're building large-scale applications it is typically considered the framework of choice.

Zend is based on simplicity, object-oriented best practices, corporate friendly licensing, and a rigorously tested agile codebase. *Zend* Framework is focused on building more secure, reliable, and modern Web 2.0 applications & web services, and consuming widely available APIs from leading vendors like Google, Amazon, Yahoo!, Flickr, as well as API providers and cataloguers like StrikeIron and ProgrammableWeb. It has a powerful engine for searches used by other frameworks also [20].

### Codeigniter

*CodeIgniter* is (as of v2.0) a PHP5.2+ MVC framework with a small footprint and great documentation. Often called a "beginner" framework because of its relative ease of use and short learning curve, *CodeIgniter* is nonetheless flexible and powerful. The framework has a large community surrounding it, so it's fairly easy to find an existing library.

*CodeIgniter* is best suited for content sites and small-to-medium web applications. You'll likely see a performance drop with very large numbers, but there is some built-in caching to help with that. If you're new to PHP frameworks or want something that can be picked up and learned in a few hours, *CodeIgniter* is one of the top frameworks to consider.

*Codeigniter* is a powerful PHP framework with a very small footprint, built for PHP coders who need a simple and elegant toolkit to create full-featured web applications [21].

### Symfony

*Symfony* is one of the oldest PHP frameworks, and is designed for enterprise-level web applications. For all its power and performance, however, *Symfony* has a small footprint and is easy to configure on a variety of PHP hosting environments. Since it's been around for so long, you'll find a lot of tutorials and books available on framework use, a perk for new users.

*Symfony* uses command line generators for quick project setups and automatic code generation, which may be new to some developers. However, they allow you to get up and running with your code in very short order. If you feel intimidated by this, don't worry. The Symfony's web site has a large collection of tutorials, showcase applications and screencasts to get you going.

*Symfony* provides architecture, components and tools for developers to build complex web applications faster. Choosing *Symfony* allows you to release your applications earlier, host and scale them without problem, and maintain them over time with no surprise. *Symfony* is based on experience. It does not reinvent the wheel: it uses most of the best practices of web development and integrates some great third-party libraries [22].

### Yii

*Yii* is a highly modular, high-performance PHP5 framework designed specifically for developing web applications. *Yii* uses a lot of command line generators and tools to get you up and running quickly; therefore, it's best used by people that aren't intimidated by a terminal window. All those generators mean more commands to memorize, but once you do, you'll find that they greatly decrease the time it takes to set up and configure your application.

It's easy to extend *Yii* and add in your favorite third-party libraries. *Yii* also supports templating, themes and widgets, and includes tools for testing, libraries for internationalization and web service integration. *Yii*'s performance is good and the automatic code generation makes it a great framework for rapid development. The framework is well-documented, but since so much of the work is done at the command line, it can be intimidating to some users.

*Yii* is a high-performance component-based PHP framework best for developing large-scale Web applications. *Yii* comes with a full stack of features, including MVC, DAO/ActiveRecord, I18N/L10N, caching, *jQuery*-based *AJAX* support, authentication and role-based access control, scaffolding, input validation, widgets, events, theming, Web services, and so on. Written in strict Object-Oriented Programming (OOP), *Yii* is easy to use and is extremely flexible and extensible [23].

These are characteristics of the frameworks explained by their own developers, to improve a description for each one, a list of technical characteristics that can be useful for us is shown [24][25].

After seeing all of those characteristics we are ready to make a choice for developing this project. *Zend* is a reliable and powerful framework, developed by the same team that developed PHP3. It has powerful libraries for search engines and transactions for purchasing with real money, but since we don't need these features for our application and also *Zend* does not provide form validation we discard this framework.

The rest of options are *Symfony*, *CodeIgniter* and *Yii*. All of them satisfy our technical needs for this project. One of the main features of *Yii* framework is that it is oriented to high-performance, big projects, and since the application we are developing is going to be simpler and will not perform operations with hard computing resources demanding, maybe it does not make *Yii* the most suitable framework for this project.

Last options are *Symfony* and *CodeIgniter*. In that case the decision is easy. *Symfony* has a great integration with Doctrine Object Relational Mapping (ORM), which is configurable with XML files and since that project is oriented to an XML environment, we will be able to reuse some XML generators/parsers for setting-up our application. *CodeIgniter* does not have integration with any ORM.

The final decision for the development of our application is taking *Symfony* v2.0.11 as our framework, which front page is shown at Figure 1.11. The following is a review of the reasons of this choice:

- Well-known framework, used by many companies during past years. Great community and documentation.
- Easy to install. Fast development.
- Plenty disposition of useful components oriented to develop the main features for web-based applications.
- Integration on many operating systems. Massachusetts Institute of Technology (MIT) license, similar to General Public License (GPL).
- Less grounding time.
- XML configurable.


**Figure 1.11. Symfony2 framework main page.**

| Name | Ajax | MVC (arch.) | ORM | Testing | Security | Template | Scaffolding | Form Valid. |
|---|---|---|---|---|---|---|---|---|
| *Zend* | Toolkit-independent | Yes (push/pull) | Table and row data gateway and *Doctrine* 2.0 | Unit tests | ACL-based | Yes | Yes | No |
| *CodeIgniter* | Any | Yes (push) | Third party only | Yes | Yes | Yes | Yes | Validation, security |
| *Symfony* | *Prototype*, *script.aculo.us* | Yes (push) | *Propel*, *Doctrine* (YAML/XML) | Yes | Yes | PHP, *twig* | Yes | Yes |
| *Yii* | *jQuery*, *jQuery UI*, own plug-ins | Yes (push/pull) | Data Access Objects (DAO), Active Record Pattern, Plugins (*Doctrine* 2.0) | *PHPUnit*, selenium | ACL-based, RBAC-based | PHP-based | Yes | Yes |

**Table 1.3. Framework technical characteristics**

## 1.4 Global project context and goals

### 1.4.1 TDK4PE

The goal of **Technology & Design Kit for Printed Electronics (TDK4PE)** is to set a fundamental change in the way PE are designed and manufactured in Europe, with the aim of reducing costs and time-to-market by more than an order of magnitude for more complex designs than ever before by addressing thousands of transistors on a substrate. The key is to develop a methodology to enable application-specific PE circuit implementation. This will be achieved by adopting a methodology similar to the silicon micro-electronics: using a Technology and Design Kit to abstract physics to a point where engineers could address physical design with enough reliability and great freedom for creativity.

### 1.4.2 Project goals and needs

TDK4PE project addresses the idea of how to abstract the Printed Electronics technology process details and identify the basic knowledge, methods and tools needed to design circuits for those emerging technologies without having to bear the entire burden of technology details regarding equipment, inks and substrates. This idea was already developed in the 80's for the micro-electronics industry by developing their related Process Design Kit (PDK) containing all technology related information for low end full-custom design, and Design Kits (DK) including libraries for cell-based semicustom design. Now a similar situation could be considered, but the maturity stage of PE technologies, their instability and variability and their expected low cost for short of large productions poses a lot of additional challenges that shall be specifically faced taking into account all the previous experience on silicon based micro and nano-electronics.

Based on this experience the main milestones of this project to obtain a useful TDK are:

- Formal collection and management of PE basic process technology information: physical and electrical parameters & models, geometric layout design rules and tolerances, fundamental design parameters considering its variability and their effects on performance.
- TDK including some basic abstraction levels like physical-electrical full-custom designs and cell library based design to make transparent to application designers most of the details about the final electronic printing technology.
- PE oriented design tools and flows customized with the previously developed TDK, covering the design flow from circuit descriptions (HDLs, schematics or parameterized cells) down to geometrical layout and fabrication related files. This includes device models and technology files for usual design processes like design entry (schematics and layout levels), simulations (functional, electrical and physical) and verification (Layout Versus Schematic (LVS), E/DRC and XTR).
- Parameterized or specific backend tools to move the circuit geometries or layout (like GDS2) to specific printing system format (like general bit-map or specific printer language), while applying the needed compensations and formatting operations.

In order to reach such targets, a methodology for technology definition and characterization will be defined. This methodology will be refined along the project under

an open parameterized scheme based on standards to assure its further reusability on other PE processes.

Figure 1.12 shows how top-down approach from applications domain and bottom-up approach from technology domain are meeting-in-the-middle in terms of TDK customized over EDA tools. Such design flow definition and tools customization by TDKs is summarized in Figure 1.13.

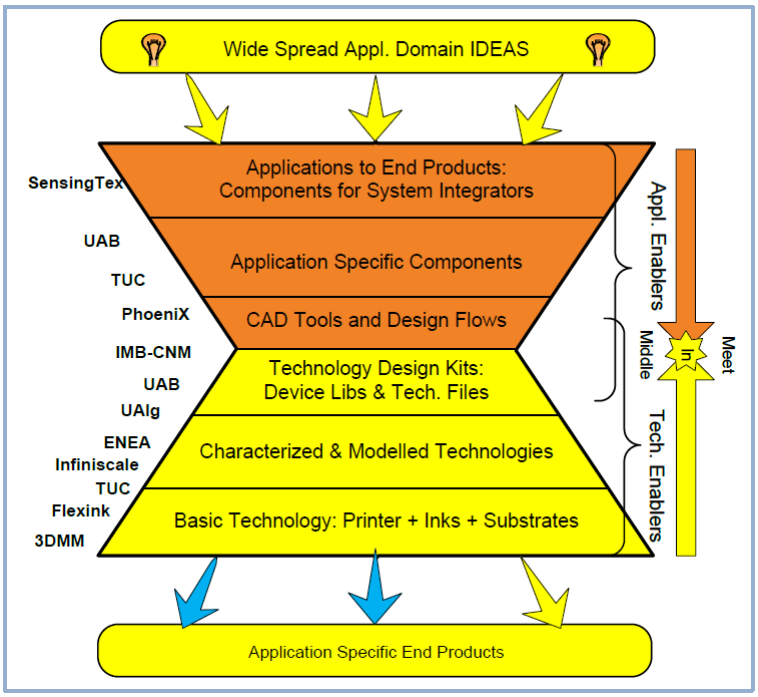



**Figure 1.12. Project tasks and assignments.**

The project focuses on the first time development of generic knowledge, methods and tools that can be used in both proprietary and open CAD tools. In the project this will be applied to a specific application for a specific technology to demonstrate its power. The inkjet printing technology has been chosen for the project partners as it is one of the most promising and challenging alternative to masking patterning technologies.

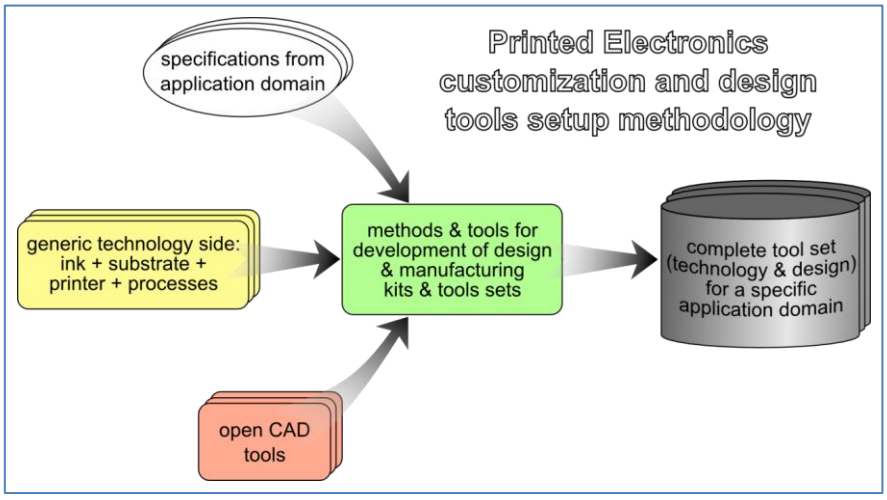



**Figure 1.13. CAD tools setup methodology.**

To reach those objectives TDK4PE will address challenges on different topics (technology design and application) at different abstraction levels from the baseline technology up to application driven libraries development. A methodology for development, characterization and modeling at each abstraction level shall be defined, applied and refined in order to get all the needed information that will later on become the technology and design kit database from where different TDKs to address different CAD tools could be derived. More precisely the target objectives are grouped and detailed in the following blocks:

- Technology processes (printing systems + function inks + flexible substrates):
    - Identify the target inkjet technology (and its missing steps) for the specific application. The inkjet printing technology has been chosen by the project partners as it is one of the most promising and challenge digital printing alternative to masking patterning technologies.
    - Baseline technology processes (different layers, isolation connection structures, interconnection-transmission lines) development and characterization.
    - Extract and collect the related technology files and models.
    - Build the post-processing steps to efficiently adapt geometrical EDA description to machinery position and jet parameter controls.
- Libraries of basic devices and building blocks:
    - Basic passive and active devices like resistors, capacitor, diodes, thin film transistors, antennas, etc.
    - Digital cells to cover both combination and sequential functional circuits.
    - Fundamental analog cells like Operational Transconductance Amplifier (OTA), OpAmp, Analog to Digital Converter (ADC) and DAC converters, etc.
    - Basic I/O blocks.
    - Different representations (schematic, layout, p-cells…) and models for all the above devices and components ready for its use on different TDKs.
    - Macro blocks including parameters to allow individual personalization for each fabricated circuit in digital printing (i.e. device ID being personalized for each device copy at printing time).
- Design flow and tools:
    - Layout to printer compensation, conversion tools, and formatting.
    - Backend simulation and validation tools to allow physical design at full-custom level.
- Cell-based design tools for design entry, simulation and parameterized and assisted layout synthesis tools considering process variations to allow tolerant design.

- Application domain:
    - Incremental demonstrators (RF structures & antenna, 4-8 bit flash ADC, redundant & fault tolerant ADC large structure) oriented to its usage as components for the final application and useful for project evolution measurements and follow-up.
    - Final system demonstrator including a complete "printed logic unit" for a specific sensor-actuator system.
    - A fault tolerant version of the demonstrator aiming to approach the target 10,000 OTFTs system.

Those objectives are addressing the whole value chain for PE and the compact consortium build around this project is able to cover not only each one of the steps, but also all the critical links in between as shown in Figure 1.12.

### 1.4.3 How does this project connect to TDK4PE

This section explains how this project connects with TDK4Pe. This project will focus mainly on the management and deployment of the technology information files.

As it has been exposed on the previous sections of this chapter, big amounts of information concerning the PE technology side have been generated. This information may refer terms from baseline technology (inks, substrates, printer…) to design level (layers, materials, colors…), device level, etc (Figure 1.8)

On the other side we have a set of tools which need some of this information. All this information has to be integrated and be consistent for achieving one of the final goals of TDK4PE, which is to build a set of tools for PE design. Here lays the motivation of this project, we want to gather all of this information, link both sides (technology and application domains), store it in a proper way (inside a database) and it is also required to generate PDKs with this information in order to be used by the EDA applications. In big terms, the application we are going to develop will be capable of reading files with big amounts of information, store it in order to allow its users to manage that information, and generate files for the set-up of PE CAD tools. This application could be classified as one of the tools referred on the green square of Figure 1.13**.**

Once finished, this project will provide an environment which will lead to an easy generation of information/technology files, very useful for TDK4PE flows. This environment is mainly intended to generate and manage information needed for the future PE tools, but it also will be capable of regenerating the structure of the information.

For instance, we could need to define a new technology for a new inkjet printer. Once we have defined it, we could set up a structure for the information and specify that structure with some files. The specific format is described on chapter 3. These files will be the seed for the storage and manipulation the technology information. Finally the proposed application will provide some means to exploit the saved information (for example, customizing an EDA tool).

# 2 Project objectives

## 2.1 Main goals

As it was stated on the introduction, this project will focus mainly on the development and characterization of a web-based environment for the compilation and management of the necessary information for the technology and design kits (TDK) regarding PE project TDK4PE (see Figure 2.1). This technology information will be used to generate files intended for PDKs to customize CAD tools used for PE design, or for documentation purposes.

These are the main goals of the project:

- Environment definition.
- Information management application development.

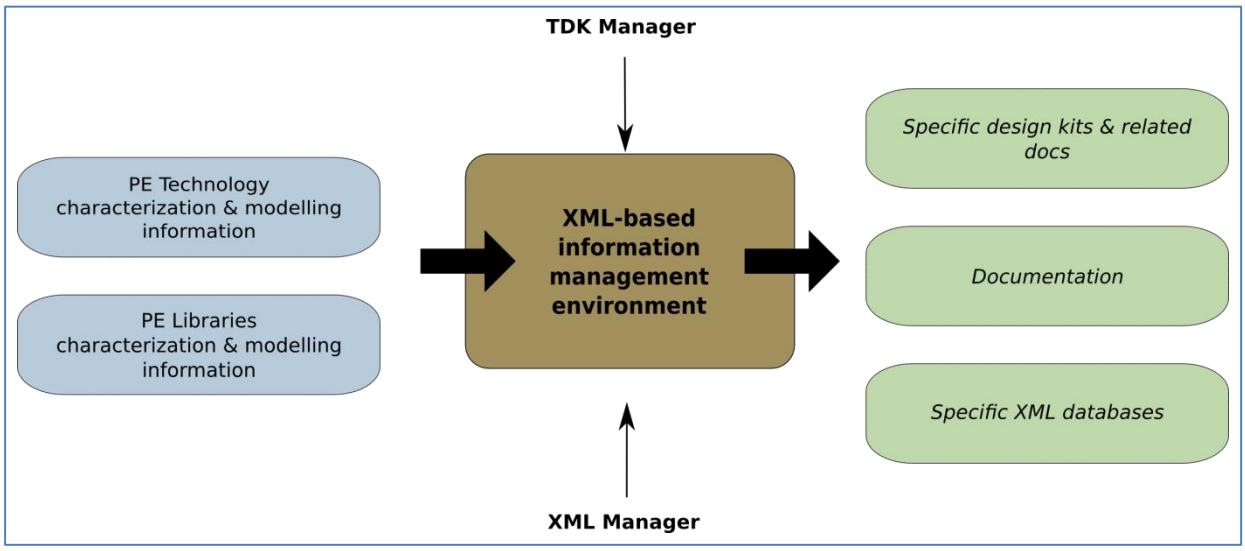



**Figure 2.1. Project input/output**

## 2.2 Detailed goals

This section explains each of the goals for this project with detail. First we explain the goals derived from the environment definition:

- Styles, formats and specifications defining and dealing with the storage, description and validation of the information: The information that's going to be managed, is supposed to be formatted, so we need to define its structure for the input and the storage. XML-XSD is a good way to afford the definition/validation, therefore it will be the format used for the import/export feature. For the storage we need to define a database structure that fits the mandatory information and its relationships.
- Handling, exchange of information and formats development: As we don't exactly know what will be the needed information or also it may vary from time, we need to adapt the format to the environment so it will include a format updater tool.

- Define TDK information file with all its requirements, the chosen format is XML: Specification for the main XML file will be exposed. It will not be definitive, but an initial structure will be defined to make tests with the environment.

And these are the goals directly related to the information management framework that we are going to develop:

- Define technology file structure within an XSD file: The proposed initial structure will be defined with an XSD file and that will be stored on the server side. This file will also be used for the possible regeneration of the TDK file structure.
- Create an export tool from XSD file by using some forms, so that we will be able to obtain an XML file which will be read by the future CAD tool or used to generate the related PDK files. The application will be able to generate XML files compliant with the XSD provided by its administrator. This will be possible due to different forms which will be generated from the XSD file.
- Create an import tool which will read an XML file and validate itself with the XSD file: We also will have the functionality of validating XML files. The user will be able to upload that files and the application will validate themselves with the current structure of the information, defined by an XSD file.
- Create a document generation utility, destined to the development of the specific design kit: The application also may gather sets of XML files related to some specific parts of the TDK.

## 2.3 Project costs

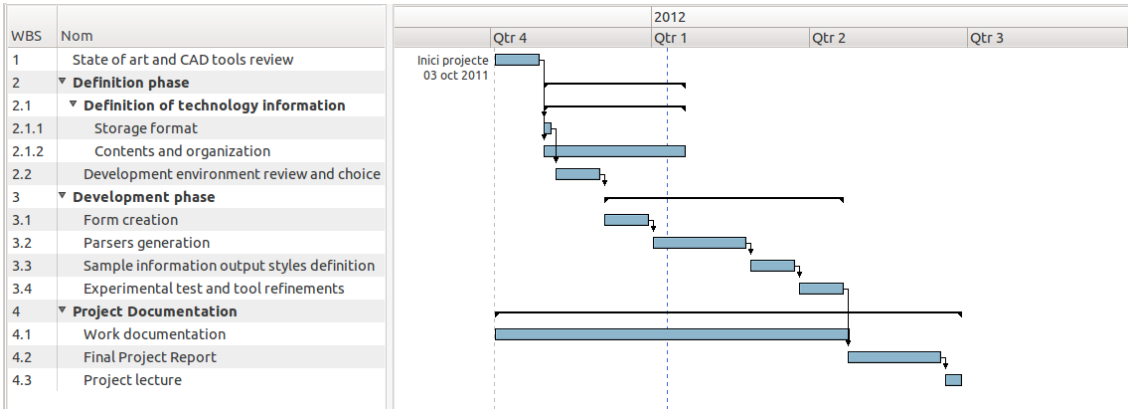
The costs of the different tasks to be developed will be estimated. The following basic factors must be taken into account.

- Human resources: 6 person/month work (20€/hour) ≈ 3,500€.
- Software: To be determined, not more than 1,500€ as it will probably be under GPL license.
- Hardware: Computing equipment for development and hosting ≈ 1,000€
- Overheads: At IMB-CNM 23,5% of personnel costs ≈ 3,750€
- Other: None
- **Total estimated costs around 9,750€.**

## 2.4 Work planning

In order to address the above exposed objectives, this project has been divided in some sub-tasks:

1. Definition phase
    a. Define technology information.
    b. Development environment review.
2. Development phase
    a. Form creation.
    b. Parsers development.
    c. Sample information output styles definition.
    d. Experimental tests.
3. Project documentation
    a. Work documentation.
    b. Final project report.
    c. Project lecture.



| WBS | Nom |
|---|---|
| 1 | State of art and CAD tools review |
| 2 | **Definition phase** |
| 2.1 | **Definition of technology information** |
| 2.1.1 | Storage format |
| 2.1.2 | Contents and organization |
| 2.2 | Development environment review and choice |
| 3 | **Development phase** |
| 3.1 | Form creation |
| 3.2 | Parsers generation |
| 3.3 | Sample information output styles definition |
| 3.4 | Experimental test and tool refinements |
| 4 | **Project Documentation** |
| 4.1 | Work documentation |
| 4.2 | Final Project Report |
| 4.3 | Project lecture |



**Figure 2.2. Project tasks and planning.**

# 3 Specifications and Application structure

## 3.1 Application

We are developing a web-based environment application, so it will be easier to deploy any further change of the functionality, without caring about obsolete versions of the application, because it will be stored on a web server and any user will connect to it and use the same updated version.

The basic functionality of the application is to generate TDK information files in an XML format. Every user of the application will be able to generate any kind of XML file related to TDK, at the beginning only the generation of the technology files information will be possible, but as we refine and improve the application, more XML formats will be available for all users.

The main feature of the application is the dynamic generation of the XML files. It will be possible for the user administrators to regenerate the XML structure. Once one user is authenticated as an administrator, he will have access to a restricted area where he will be able to upload a zip file containing a set of XSD files that will define the whole XML structures and database schema (one XSD file for each XML). When this action is performed a link to the XML generation will appear in the front page. So that this application will provide some dynamic XML generation by reading its structure from an XSD file.

The application is divided into two big parts, as can be seen on Figure 3.1, the frontend and the backend. The frontend shows the list of XML files we can generate and it provides many forms to collect information. The backend gathers all the information related to the admin part, contains the XSD parser which will detect the XMLs the application will be able to export, and generates the database schema needed to store the information related to XML files.

## 3.2 XML-XSD file formats

In order to use a standard and easy-readable format we chose XML-XSD for the whole application functionality and TDK information specification. The application can export a set of XML files and each one is defined by its related XSD file. We have chosen the XML format because it is a standard format with defined specifications that allows us to organize information in an easy way and there are lots of parsers defined. Another feature that will be very useful for us is the validation, as we need to ensure the generated files are XSD compliant, we may need to validate them. XML files claim to be one of the easiest information files to understand because they are organized by tags with its description on the name of the tag. For more information concerning XML format see Annex B.

Before going in depth into this proposal we would try to clarify some fundamental concepts around the different meanings and abstraction levels of TDK information to be collected along this project.

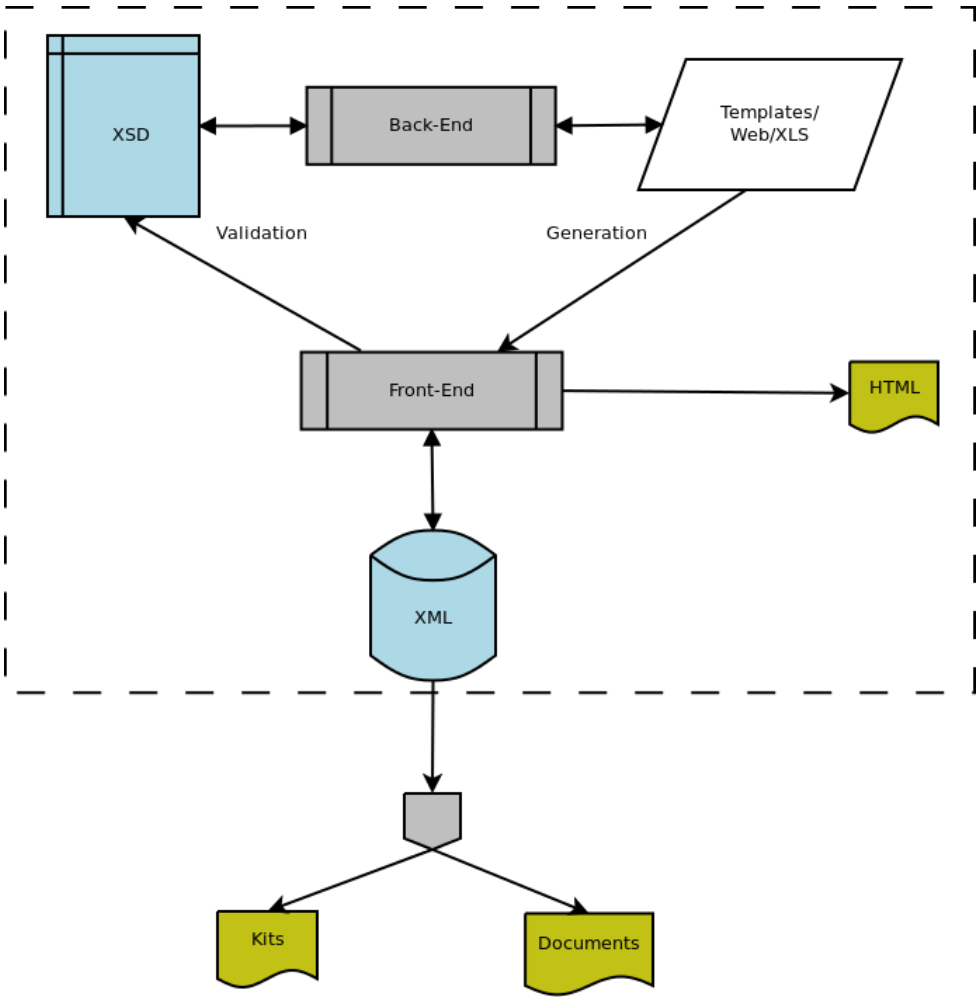



**Figure 3.1. Application flow.**

One of the most important concepts at this point is the meaning of "TDK Information", which deals with the information we need to know at different abstraction levels in order to address the right design activities at each one of them, being able to reach the fabrication steps for a specific technology. TDK Information is not addressing any specific design description at any one of the abstraction levels, but the needed information to allow the generation of technology and design kits to enable and drive the design activities over specific EDA tools.

Concerning the abstraction levels we can make an early distinction on three different levels.

- **Printing level:** information dealing with what is needed to describe geometries and parameters for its specific printing. This information is fully dependent on the specific technology (printer + inks + substrate + processing steps).
- **Physical level design:** information to enable the generation of PDKs for design at physical level (geometrical layout, electrical parameters, DRC, Electrical Rules Check (ERC), XTR).
- **Higher level design:** it refers to all the information to allow cell-based designs and will collect the different libraries:
    - Basic devices, parametric cells and regular structures building blocks library
    - I/O pads, Digital cells and Analogue blocks libraries

As we have introduced on the above points those three levels will have some additional sub-levels, and it could happen that additional ones would be needed to make a seamless interface at information level between each two of those principal abstraction levels.

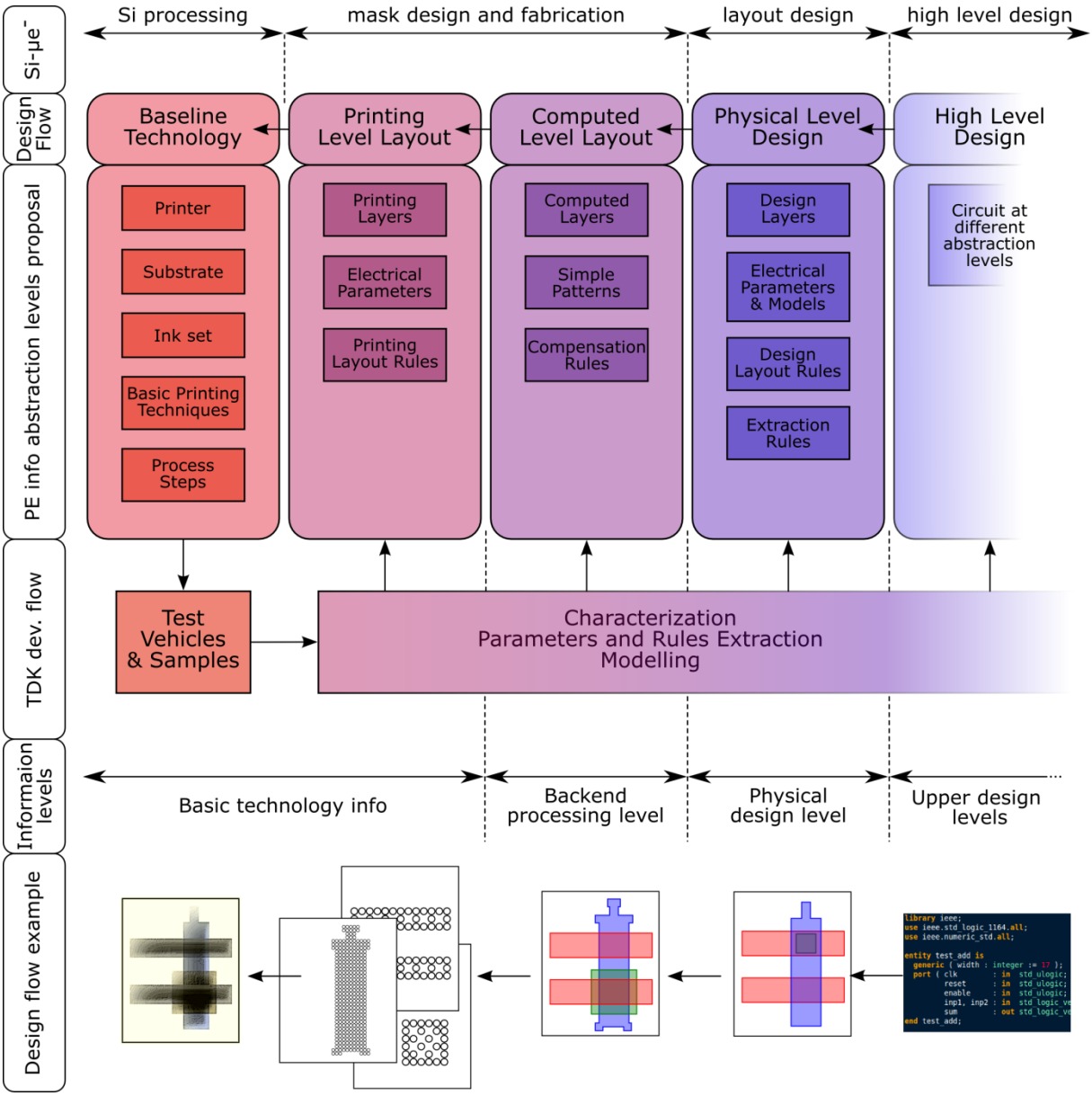



**Figure 3.2. PE design abstraction levels.**

In Figure 3.2 we can see some of those abstraction levels showing different flows and concepts:

- Silicon based microelectronics equivalent phases to PE ones.
- The normal top-down design flow, from right to left in the Figure 3.2, to reach the technology and fabrication phase from physical or high level design.
- The generic TDK information at each abstraction level.
- The TDK bottom-up, from left to right in the Figure 3.2**,** development flow.
- Few simple design related examples, not to be described within the "TDK Information", but to show the kind of information needed and managed at each design level.

### 3.2.1 PDK level: partial and earlier example

Some simple XML description examples trying to show how this language could collect the related "TDK Information" at each abstraction level are provided later on the specific sections. This project is going to focus on the "Physical level design" to expose its working. As the application is ready to adapt to new formats, it will be able to accept any structure as long as it's defined in a XSD file. To make the explanation easier we only consider "Design Layers" from now on.

| **Design Layer** | | |
|---|---|---|
| **PARAMETER** | **TYPE** | **DESCRIPTION** |
| NAME | String | Name of the Layer. |
| SHORT_NAME | String | Short for the Layer. |
| PURPOSE | String | Purpose of the Layer. |
| GDS_NUM | Integer | Layer identifier on a GDS file. |
| GDS_DTYP | Integer | Identifier used to state printing order. |
| COLOR | String | Color representation for layer. |
| SEL | Boolean | Selectability of the layer. |
| VIS | Boolean | Visibility of the layer |
| FILLSTYLE | String | Name for the fill style definition. |
| LINESTYLE | String | Name for the line style definition. |

**Table 3.1. Design Layer parameters.**

```
<?xml version="1.0" encoding="UTF-8"/>
<xs:element name="dlayer">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="name"/>
            <xs:element ref="short_name"/>
            <xs:element ref="purpose"/>
            <xs:element ref="gds_num"/>
            <xs:element ref="gds_dtyp"/>
            <xs:element ref="color"/>
            <xs:element ref="sel"/>
            <xs:element ref="vis"/>
            <xs:element ref="fillstyle"/>
            <xs:element ref="linestyle"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

**Listing 3.1. Design Layer described in a XSD file.**

This file will define the structure of our entities by each element definition. Above we can see some parameters for the "Layer" type with its name, but this file also defines the types for every one of them. This is one of the XSD files that can be part of the .zip file the administrator uploads when he wants to regenerate the application schema. (For more information concerning the regeneration of the structure see chapter 0.)

```
<xs:element name="purpose" type="xs:string"/>
<xs:element name="gds_num" type="xs:integer"/>
<xs:element name="sel" type="xs:boolean"/>
```

**Listing 3.2. Types definition.**

Listing 3.2 shows how the types are specified in the same document. When an XSD file describes an element, its type must be defined too, so that the content of the future XML related to that file can be validated. We will use that type specified in the XSD file to determine the type of the data we are going to store, so that our database is fully consistent with the definitions. For instance we can consider that "xs:string" are strings "xs:integer" are integers, etc…

```
<?xml version="1.0" encoding="UTF-8"?>
<dlayerSet>
  <dlayer>
     <name>Metal1</name>
     <short_name>MET1</short_name>
     <purpose>drawing</purpose>
     <gds_num>100</gds_num>
     <gds_dtyp>10</gds_dtyp>
     <color>red</color>
     <sel>1</sel>
     <vis>1</vis>
     <fillstyle>grain</fillstyle>
     <linestyle>plain</linestyle>
  </dlayer>
…
</dlayerSet>
```

**Listing 3.3. Dlayer example XML file.**

Listing 3.3 contains the kind of file our application will export for the defined layer named "Dlayer.xml", but as we are validating that file with the XSD, we could have lots of different files related to a Layer, even though a file with a set of Layers on it. On the next section we are going to explain how the data is stored and what we do with each XML file definition.

## 3.3 Contents specification and storage

It's very important to collect the information for the TDK specifications in a proper way. These specifications may contain; apart from Layer definitions, distinct design rules or any other information that could be useful in any abstraction level as we can see in Figure 3.2. But, how do we know what kind of information is that, and how to store it? The answer for this question remains inside the XSD file. These files define the structure of the XML files and the type of data related to each element; so that, we can extract all that information within this file. Chapter 0 will explain deeply how this information is extracted.

It will be necessary that information concerning the data types is included on the XSD file. This is not an issue because XSD specification states that any field has its own type definition on the same file. It is even possible to enclose a set of fields within another. In that case we would consider the latter as a future database entity. Each field that appears on an XSD file will describe a variable that will be stored on a database. In order to generate those entities forming the database we need to generate some XML files, which will lead to the PHP classes from the ORM we used in our framework (Doctrine2).

At the end we will have one class for each XSD file, and will be capable of storing the contents for our final XML files with the TDK information.

Now we are going to explain how the XSD files uploaded by admin must be organized in order to generate correctly the database schema. We pick up an example of a supposed Layer for a PE technology with two main characteristics that shall be mandatory for a future CAD design tool. This Layer will be the same exposed in previous section but we consider that two of its properties "Color" and "Fillstyle" have its own characteristics.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:element name="dlayer">
    <xs:complexType>
        <xs:sequence>
            …
            <xs:element ref="color"/>
            …
            <xs:element ref="fillstyle"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

**Listing 3.4. References inside XSD file.**

Suppose we want to link some characteristics to any set of elements; for example the "Color" property is defined as RGB so we need at least three variables to describe it, one for each primary parameter, but we want that color to be related to a Layer. We can use the database relations for making that possible, for instance we may have a primary entity called "Layer" with two related entities called "Color" and "Fillstyle".

We have taken use of the annotation property of the XSD files to indicate when we are going to need some external info for the current file. Now when we parse an element which has an annotation node, we know that another file shall be included with an element labeled *extern:<name>* with the name defined inside that node. **The related entity must have a property labeled "name"**, if not, the application won't know how to refer to this relationship. That's the main requirement XSD files must follow in order to generate a consistent database. Assuming that we have uploaded a ZIP file containing an XSD entry as the Layer definition with the characteristics exposed above and two more entries with the "Color" and "Fillstyle" definition, the following database schema is going to be generated.

```
<xs:element name="color" type="xs:NCName">
     <xs:annotation>
          <xs:appinfo> extern:Color </xs:appinfo>
     </xs:annotation>
</xs:element>
<xs:element name="fillstyle" type="xs:integer">
     <xs:annotation>
          <xs:appinfo> extern:Fillstyle </xs:appinfo>
     </xs:annotation>
</xs:element>
```

**Listing 3.5. References definition.**

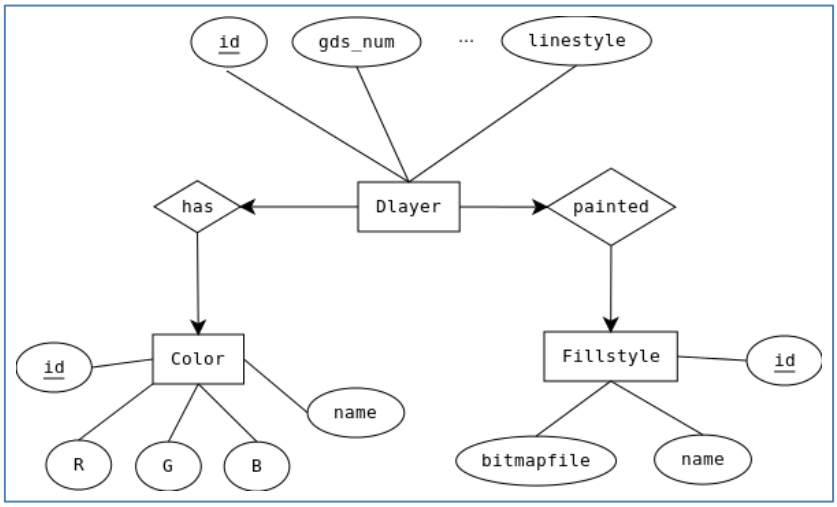



**Figure 3.3. Entity/Relationship diagram for a "Dlayer" entity.**

# 4 Back-End application

This chapter describes the process used to regenerate the XML structure by changing the database schema related to the exported files. This part of the application is launched when an authenticated user logs in and uploads a set of XSD files compressed in a ZIP file. Each XML file the application is able to generate, is related to a database entity. When we want to change the structure of any of the XML files, we have to regenerate the database schema, and we do this by parsing the XSD file and creating some classes which will lead to the final generation of the database.

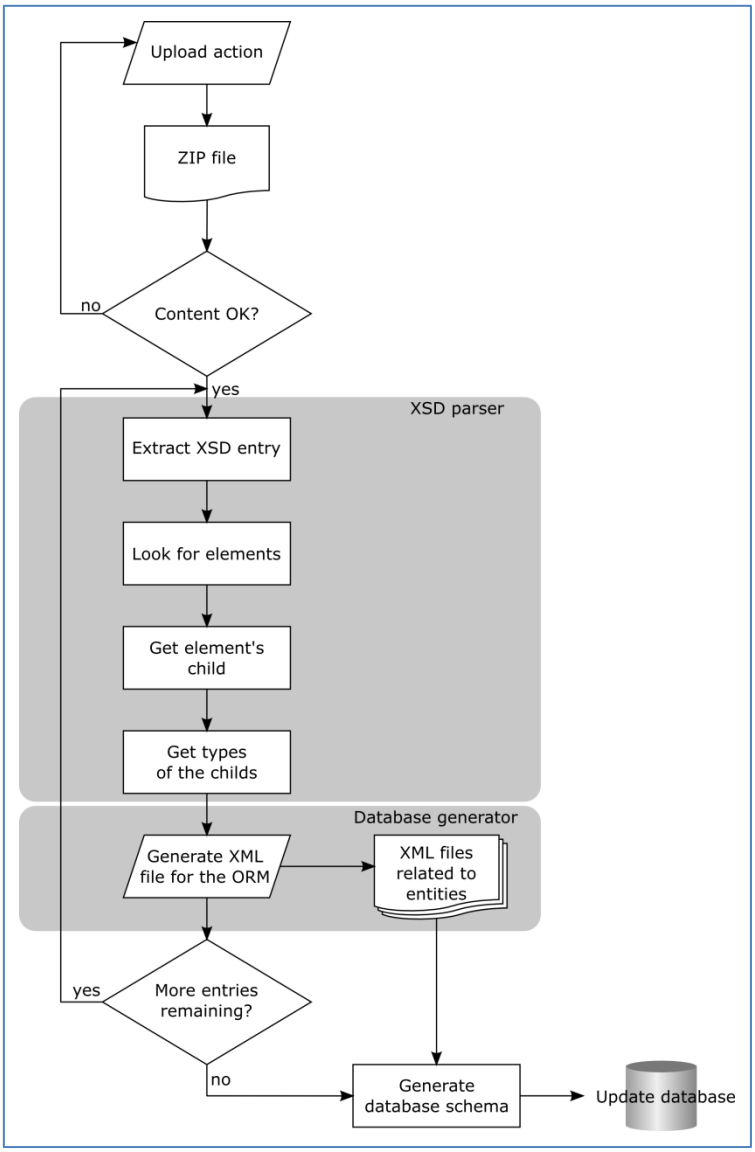



**Figure 4.1. Internal operation flowchart.**

Next sections describe the internal operation of each block.

## 4.1 XSD Parser

The XSD parser is called when the administrator clicks on the upload link on the application and sends a ZIP file to the server. Following the Model-View-Controller (MVC) design pattern (Annex A), we attach that link to a controller's action that is responsible of calling the parser. This parser will generate XML files, which will become entity descriptors for the ORM. First of all, the consistency of the file is checked, the following conditions are checked:

- Administrator sends an upload request.
- The uploaded file has *.zip extension.
- The ZIP entries refer to an XSD file and ZIP file has not any directory inside.
- XSD files are built in a proper way.

When all of these conditions are met we treat each entry as a distinct file and we store it on the server directory. First of all the parser looks for elements that have more than two children, those would be potential entities of our database. We consider that any element that has more that two related elements needs to be stored on our database, so an XML file with the necessary information for that element/entity will be generated. Section 4.2 will explain how we turn that XML files into database entities.

For building the database we need some information, like the name of the entity, its different fields and types, any possible extern related entity and the possibility of repetition of any field. To get all of this we use the PHP class *DOMDocument()* which has some useful methods to parse the XSD file like *getElement()*, *getChildren()*, *getData()* or *getName()*. We will use all of these methods to get all the mandatory data for generating our entities related to the future XML file.

The parser will collect information for two basic classes shown on Figure 4.2.

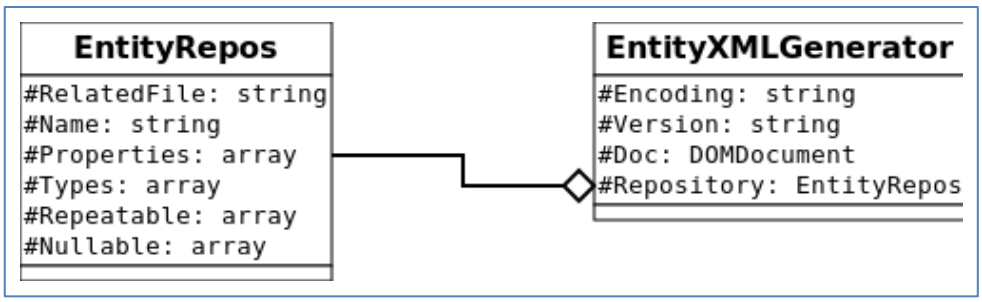



**Figure 4.2. PHP classes generated by the parser.**

The first class *EntityRepos* (appearing in Figure 4.3) is used to gather the necessary information for each entity that is going to be generated. Its properties are the following:

- *RelatedFile*: This variable contains the XSD file name.
- *Name*: Name of the future entity, parsed from the main element of the XSD file.
- *Properties*: Set of element names which are children of the main current element.
- *Types*: Set of types for each property. Property name is used as key for the array.

- *Repeatable*: Array. Property name is used as key for the array elements. If we have an element of this array set, then its key shows the name of the property that can be repeated in the XML file.
- *Nullable*: Array. Property name is used as key for the array elements. If we have an element of this array set, then its key shows the name of the property that can be *null*.

The second class *EntitityXMLGeneration* is used to generate the XML-mapped file to the database, and will perform the base generation of the database schema. The XML files that this class is going to generate will describe it entirely. Its properties are the following:

- *Encoding*: This variable is a string. It contains the encoding used for the XML file.
- *Version*: This variable is also a string. It contains the version of XML currently used by the application.
- *Doc*: *DOMDocument* object. Used for the generation of the XML file due to its useful XML-oriented methods.
- *Repository*: *EntityRepos* object. Passed as argument to the constructor of this class. For each *EntityRepos* instance, a single *EntityXMLGenerator* object is created.

## 4.2 Data Base Generator

This section refers to database generation using an ORM, its main aim consists on simplify the translation between database rows and the PHP object model. In this project the *Doctrine2* ORM has been used for a simple reason, its configuration files can be defined in XML format so we can use the same XML generator that we will use on the generation of the output files [29].

The data base generator, as can be seen on Figure 4.1, is called after the parser finished its work and all the XSD files have been read, at this point we have collected all the information related to the structure of the files in many *EntityRepos* instances. After this, for each *EntityRepos* object that has been created before, an instance of *EntityXMLGenerator* is generated, as can be seen on Figure 4.3. This new instance is dealing with the task of generating an XML file which will map to the database entity, as one of its properties is an *EntityRepos* object it's got the whole information needed to generate an entity.

The *Doctrine2* ORM is presented with many console commands, which perform database actions, that can be very helpful for this process. One of this is the *doctrine:schema:create*. It basically consists in reading the mapped files to the entities and generating the database schema with its information. Once many XML files mapped to the entities have already been generated, which have been extracted with the parser, we can run that command and it will generate automatically the PHP mapped classes to the entities and the whole database schema. This ORM has another useful feature of generating the PHP classes mapped to the entities, so that we can collect instances of those entities with PHP and call ORM functions to update the database without caring about its architecture.

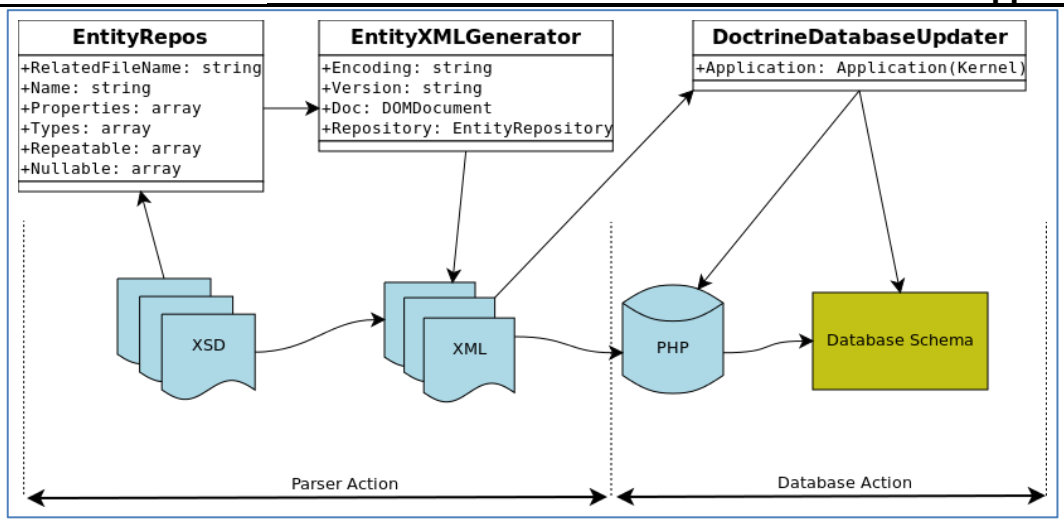



**Figure 4.3. Database regeneration operations and involved PHP classes.**

Figure 4.3 shows the whole file conversion that occurs before the database is created. The *DoctrineDatabaseUpdater* is a class used to execute the necessary actions to update the database. Its unique property is named application. This property is just an *Application* instance which consists in a *Symfony2* class that allows us to call *Doctrine2* commands automatically from code (without running a console and typing it).

Listing 4.1 shows the XML mapped file for the example "Dlayer" XSD parsed, in this case, two more XML files mapped to two more entities are needed because we have defined two extern keys.

At this point the framework is ready to store and validate XML files related to the XSD files that have been parsed before. The database with the necessary entities and fields for the XSD has been generated and we can validate XML files with the XSD that have been uploaded. Despite the information concerning the types of the properties has been extracted, we need to store it somewhere because the forms related to the XML files generation will need to know about the types of the fields, and these cannot be loaded nor from database neither from the PHP classes mapped to entities that have been generated.

Another bundle with its related entity, (see annex 9.1), is being used for that purpose. Chapter 5 explains how the information of the data structure is extracted and stored for a future use related to the generation of the XML files.

```
<?xml version="1.0" encoding="UTF-8">
<doctrine-mapping xmlns="…" xmlns:xsi="…">
     <entity name="ecmi\FrontendBundle\Entity\Dlayer" table="Dlayer">
          <id name="id" type="integer" column="id">
               <generator strategy="AUTO"/>
          </id>
          <field     name="purpose"     column="purpose"     type="string"
     length="100"/>
          <field name="gds_num" column="gds_num" type="integer"/>
          <field name="gds_num" column="gds_num" type="integer"/>
          <field name="gds_dtyp" column="gds_dtyp" type="integer"/>
          <many-to-one field="color" target-entity="Color"/>
          <field name="sel" column="sel" type="boolean"/>
          <field name="vis" column="vis" type="boolean"/>
          <many-to-one field="fillstyle" target-entity="Fillstyle"/>
          <field    name="linestyle"    column="linestyle"    type="string"
     length="100"/>
     </entity>
<doctrine-mapping>
```

**Listing 4.1. XML mapped file to database.**

# 5 Front-End application

This chapter is about all the functionalities of input of information and XML file generation. For generating the XML files according to the uploaded XSD structure we need to store the structure of the files a part from generating the database schema. That information is going to be stored in two database entities called *Config* and *Types*. These two entities are generated after the parser finished its work and they collect the information about the XSD related to each entity and the whole data types of the entity properties. These two entities, due to the use of an ORM, are mapped with its related PHP classes too. If the administrator doesn't call the parser, then the previously generated information is used.

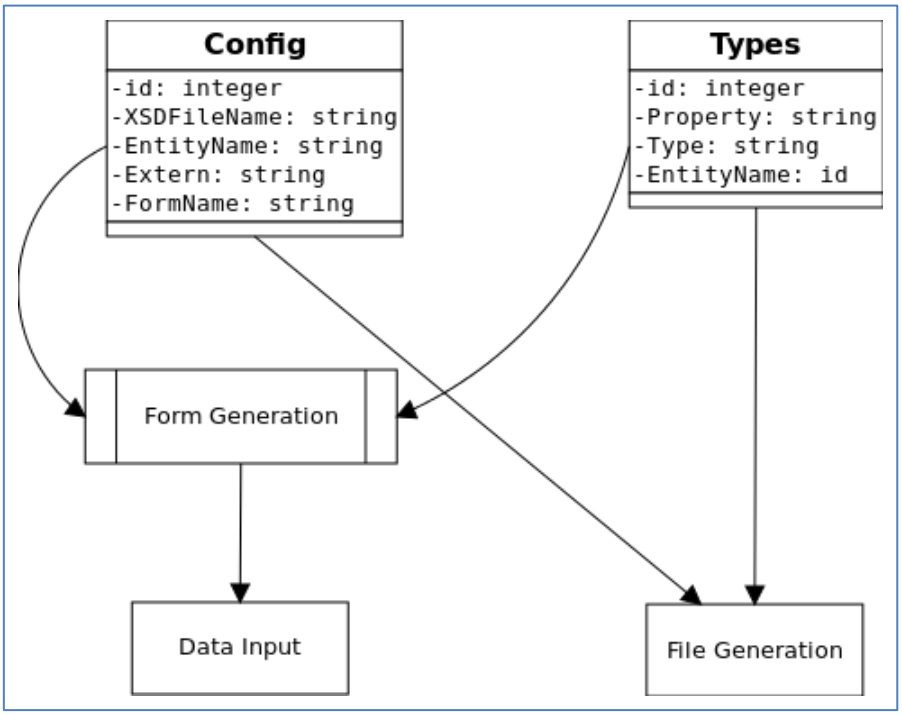



**Figure 5.1. PHP Classes containing information related to the structure of the entities.**

Figure 5.1 above show how the exposed classes make possible the generation and input of information. The *Config* class is called after the database generation, as like as the EntityXMLGeneration exposed before, for each instance of *EntityRepos*. Its properties are the following:

- *Id*: Integer that stores the primary key of the entity.
- *XSDFileName*: String with the name of the XSD file where the current entity was extracted.
- *EntityName*: String with the name of the current entity.
- *Extern*: String with the names of the extern fields, separated with commas.

- *FormName*: String with the name of the form. For internal application purposes.

The *Types* class, as its name indicates, refers to the types of the properties of the extracted entities. This class is generated after all the *Config* instances are created, for each property that has been parsed. Its properties are the following:

- *Id*: Integer that stores the primary key of the entity.
- *Property*: String with the name of the property
- *Type*: String with the name of the type of the current property.
- *EntityName*: Integer which points to the primary key of *Config.*

These two classes are the main responsible of the proper building of forms for the input of data and the correct generation of XML files. Next sections describe how these tasks are performed.

## 5.1 Data Input

This is one of the most important tasks performed by the application. With the use of forms, it allows the user to enter information according to the structures already defined by the XSD files. The forms are generated dynamically with the *Symfony2* form component which allows an easy generation. This component consists on a PHP class called *AbstractType* which has the needed methods for generating forms properly and fast. This class has a constructor that gets as parameter an array of properties and another one of types so that it's possible to generate a form with the previously loaded information.

Once the class has been instanced (with every *Config* object), it runs a method called *add(name, type)* which receives as parameter, the name of the field and its type and it generates an input type for the form.

```php
<?php
use Symfony\Component\Form\AbstractType;

class LayerType extends AbstractType
{
        // …
        public function buildForm(FormBuilder $builder, array $options)
        {
                $builder->add('name', 'text');
        }
        //…
}
```

**Listing 5.1. Example class for building an HTML form with Symfony2.**

This is a simple example of its working. This class is generating an input text type with the label 'name'. It can be called with a great bunch of types and also declare our own types and the builder will perform the necessary actions to generate properly the layout of the form.

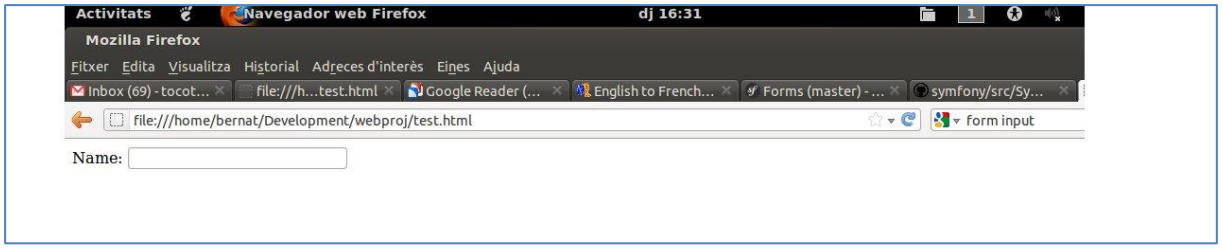



**Figure 5.2. Layout result from applying previous class.**

The application loads all the *Config* entries on the database and for each one of them it calls a form generator and creates a link for that on the front page, so that at the end we are having a link to a form for every XSD file that was loaded on the ZIP file, because a *Config* entry is generated for each one of them. When the user wants to enter information, he just has to click on the link with the name of the desired entity and submit that information to the database. The application will load that information and it will be collected with the possible previous entities that might be entered to the application.

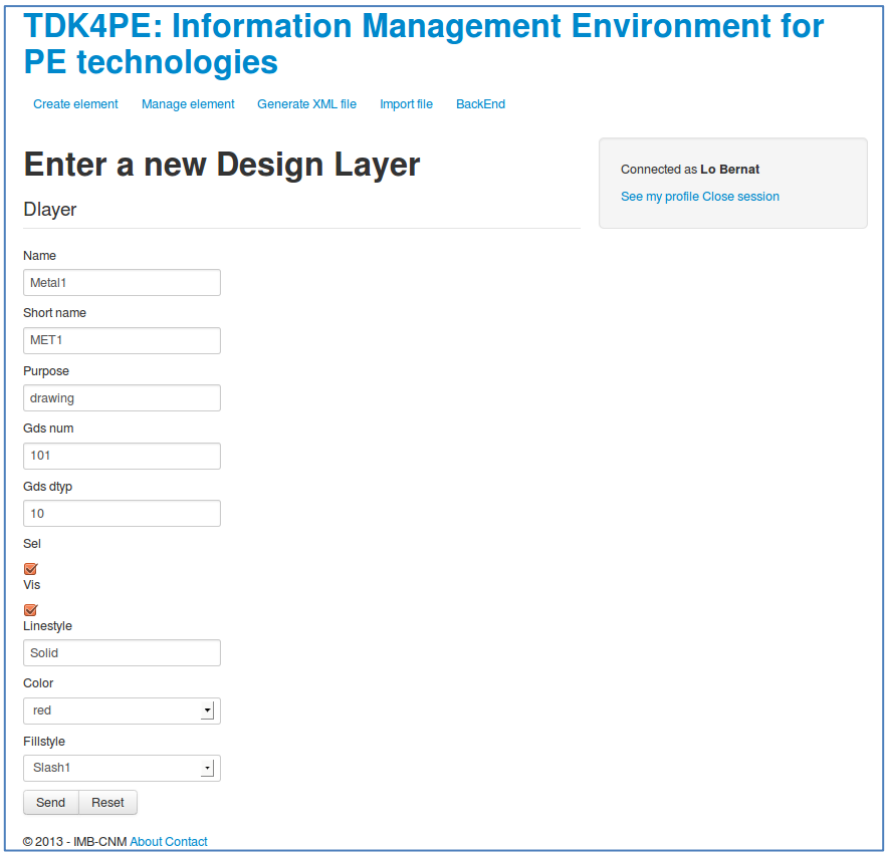



**Figure 5.3. Example form for the "Dlayer" entity.**

Figure 5.3 in the last page shows the HTML form presentation of the Design Layer with some information we want to load to the database. Related entities are shown and the application allows us to upload a register for any of them, since these are registers of the current entity. Once uploaded, we will see this result upon the database.

```
mysql> select * from Dlayer;
+----+----------+-------------+--------+------------+---------+---------+----------+-----+-----+-----------+
| id | color_id | fillstyle_id | name   | short_name | purpose | gds_num | gds_dtyp | sel | vis | linestyle |
+----+----------+-------------+--------+------------+---------+---------+----------+-----+-----+-----------+
|  1 |        1 |           1 | Metal1 | MET1       | drawing |     101 |       10 |   1 |   1 | Solid     |
+----+----------+-------------+--------+------------+---------+---------+----------+-----+-----+-----------+
1 row in set (0.00 sec)
```

**Figure 5.4. Database register.**

## 5.2 File Generation

This section describes the process executed when the user clicks on the generate utility. For this process both classes *Config* and *Types* are used again along the previously shown class *DOMDocument*. The first step is to load the whole Dlayer entries stored on database. After this, for each one of these entries, we load its type and finally the entire structure of the current entity and all the properties are added to the *DOMDocument* instance.

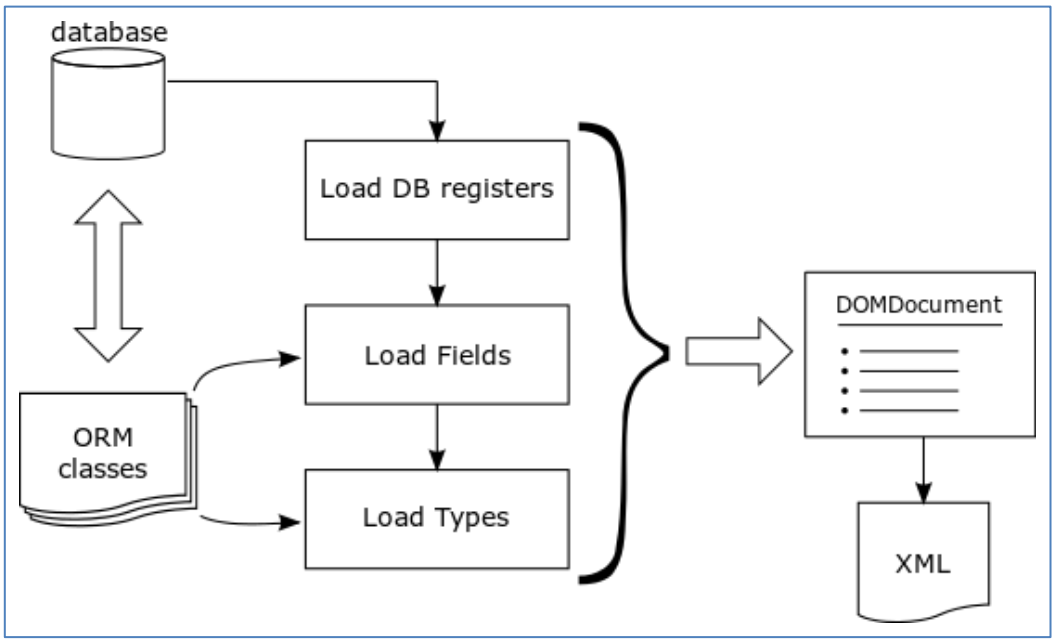



**Figure 5.5. XML file generation flowchart.**

This process ensures that the XML validates against uploaded XSD file, because we have stored its valid structure on the database and the generation runs according to this. When this process is executed, the whole "Dlayer" registers that have been stored to the database are read and written into an XML file.

```
<?xml version="1.0" encoding="UTF-8"?>
<dlayerSet xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
ecmi.local/uploads/documents/03_dlayer.xsd">
  <dlayer>
    <name>Metal1</name>
    <short_name>MET1</short_name>
    <purpose>drawing</purpose>
    <gds_num>101</gds_num>
    <gds_dtyp>10</gds_dtyp>
    <color>red</color>
    <sel>1</sel>
    <vis>1</vis>
    <fillstyle>Slash1</fillstyle>
    <linestyle>Solid</linestyle>
  </dlayer>
</dlayerSet>
```

**Listing 5.2. Example of generated file.**

An important point that has to be considered is that, all the registers from database are included in the XML generated file. This XML is exported by the application into the *$LOCAL_PATH/web/output/xml/* where it will be accessible. *$LOCAL_PATH* is instanced with the application installation path.

## 5.3 File validation and import

The application also allows users to import information in the described format by uploading some files, for this feature two formats are allowed CSV and XML files. Once the user wants to upload some files he just has to click on the link in the front page and choose the entity contained in the file. For instance we would want to upload a CSV file containing some "Color" entity, we just have to click on Import File and select the desired entity and choose available formats.

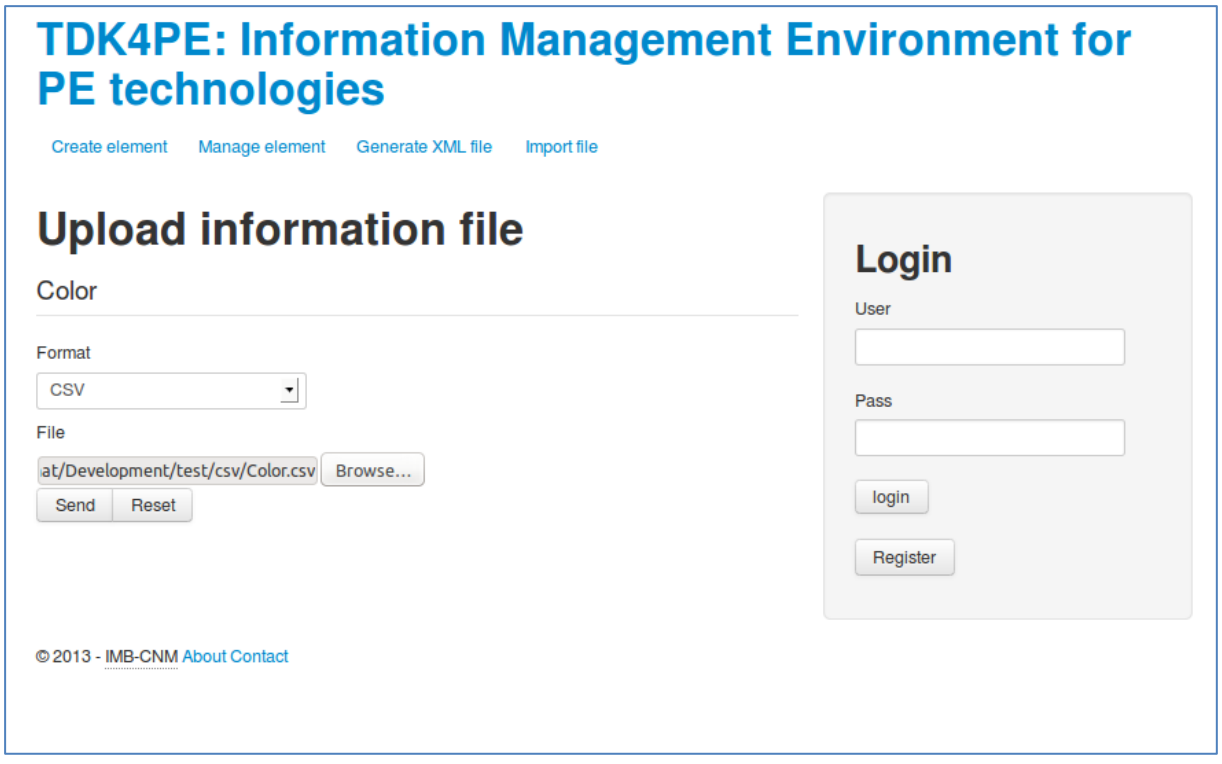



**Figure 5.6. Page for importing files.**

When we have selected the desired file to upload, the user just has to click on Send button and the import process starts. First of all the structure of the file is validated, this validation depends on the type of the file:

- If the imported file is a CSV, the application counts the number of parameters contained in the file and compares this number with the number of parameters stored on Types Class. If these numbers are equal then the file validates. It is very important the values for the parameters **follow the same order than XSD file**.
- If the imported file is an XML, then the validation is easier due to the use of PHP Class method *DomDocument::validateSchema(file)*. This class is instanced with the XML uploaded file and then we can validate it by calling that method which returns *true* if the file validates.

Once the structure is validated, the application proceeds to validate its content for consistency purposes, for instance, two different instances with same or almost the same values have not to be accepted. So, the application builds a query with all the data contained in the imported file and checks for existence in the database. A PHP object is build for each entity contained in the uploaded file. If any object exists with these parameters then its importation is allowed and the application loads the final validation page.

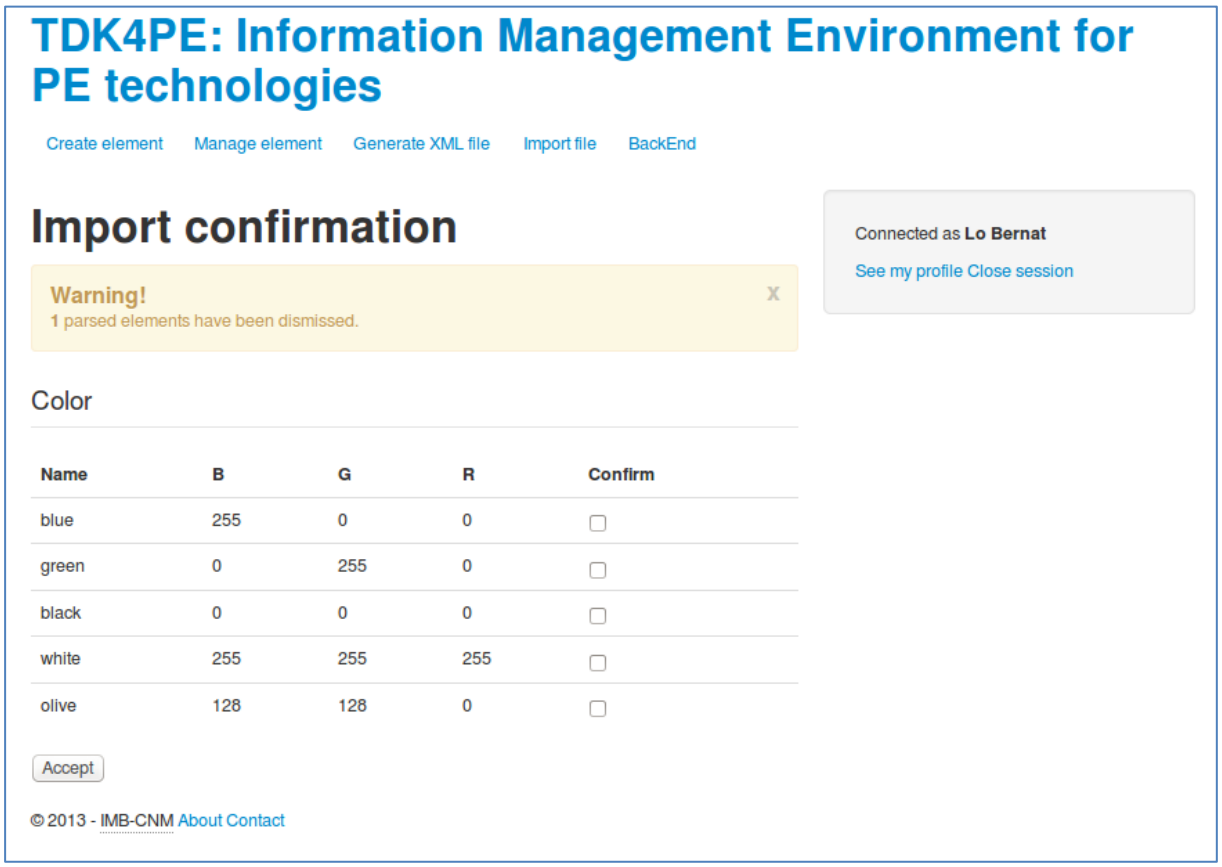



**Figure 5.7. Final validation for import feature.**

When the structure and content of the imported files have been validated, the application redirects the user to the page shown at Figure 5.7**.** On this page the user just has to select the desired entities to be import. The values of these entities are the parsed values from the imported file. Once the user has selected the entities he wants to import, he just has to click on Accept button and the entities will be stored into the database.

# 6 Output

## 6.1 Applicable functionality

The main functionality of the application will be focused on work around XML files. At this point, we should have been able to generate various XML files in the server directory. As it has been stated on Annex B, one of the primary goals of the XML files is the exchange of information and so will be the main use of these generated files.

This project is about the generation and storage of information related to the TDK of PE specification, this functionality is gathered on this web application and ruled by the input of XSD files. Once we have generated these files, we will be able to download them and they will be ready for a further use, one of these uses may be:

- Transfer design layers information between different CAD tools.
- Export TDK information, related with EXT, DRC, etc. to other applications.
- Generate custom TDK files for different design flows and CAD tools.
- As the application has flexible generation, we can adapt quickly any further change in the TDK to the XML related files.

Next sections demonstrates an example of information collecting

## 6.2 Results and practical example

As a result for this project we take its XML output and one of its applications which could be CAD customize. As an example we can take the particular case of Glade customizing, so first of all we need to know which parameters are included on its technology file. As a simple example we consider a simple "Design Layer" with its parameters "Color" and "Fillstyle". Let's suppose a TDK manager has uploaded some "Design Layers" to the database and he has generated its related XML files which could be the following.

```
<?xml version="1.0" encoding="UTF-8"?>
<dlayerSet xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
ecmi.local/uploads/documents/03_dlayer.xsd">
  <dlayer>
    <name>Metal1</name>
    <short_name>MET1</short_name>
    <purpose>drawing</purpose>
    <gds_num>101</gds_num>
    <gds_dtyp>10</gds_dtyp>
    <color>red</color>
    <sel>1</sel>
    <vis>1</vis>
    <fillstyle>Slash1</fillstyle>
    <linestyle>Solid</linestyle>
</dlayer>
<dlayer>
    <name>Metal2</name>
    <short_name>MET2</short_name>
    <purpose>drawing</purpose>
    <gds_num>102</gds_num>
    <gds_dtyp>20</gds_dtyp>
    <color>blue</color>
    <sel>1</sel>
    <vis>1</vis>
    <fillstyle>Slash1</fillstyle>
    <linestyle>Solid</linestyle>
    </dlayer>
</dlayerSet>
```

**Listing 6.1. Dlayer loaded to database and exported as XML file.**

As we can see, these "Design Layers" have some external properties attached, like "Color" and "Fillstyle". Those properties appear in the following XML files.

```
<?xml version="1.0" encoding="UTF-8"?>
<colorSet xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
ecmi.local/uploads/documents/01_color.xsd">
  <color>
    <name>red</name>
    <B>0</B>
    <G>0</G>
    <R>255</R>
  </color>
    <color>
    <name>green</name>
    <B>0</B>
    <G>255</G>
    <R>0</R>
  </color>
    <color>
    <name>blue</name>
    <B>255</B>
    <G>0</G>
    <R>0</R>
  </color>
    <color>
    <name>yellow</name>
    <B>255</B>
    <G>255</G>
    <R>0</R>
  </color>
</colorSet>
```

**Listing 6.2. Color loaded to database and exported as XML file.**

```
<?xml version="1.0" encoding="UTF-8"?>
<fillstyleSet xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
ecmi.local/uploads/documents/02_fillStyle.xsd">
  <fillstyle>
    <name>Slash1</name>
    <bitmapFile>slash1.bmp</bitmapFile>
  </fillstyle>
</fillstyleSet>
```

**Listing 6.3. Fillstyle loaded to database and exported as XML file.**

The next step is to merge all this information inside a unique XML file. After this, we to apply transformations to the merged XML file in order to obtain the desired format for Glade *.tch* files. Since we have that information collected inside an XML file, it will be very easy for us to apply these transformations due to a language for transforming XML files called XSL (Extensible Stylesheet Language). These language define transformations between XML files into any other kind of files (PDF, XHTML, etc…). We define an XSL file with the conversions needed to turn an XML file into a Glade *.tch* file.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xpath-default-namespace="http://test.com" version="1.0">
        <xsl:output method="text" encoding="UTF-8" omit-xml-declaration="yes" indent="no" />
        <xsl:template match="/">
        <xsl:for-each select="dlayerSet/dlayer">
        <xsl:variable name="select">
                <xsl:choose>
                        <xsl:when test="sel = 1">
                                <xsl:text>t</xsl:text>
                        </xsl:when>
                        <xsl:otherwise>
                                <xsl:text>f</xsl:text>
                        </xsl:otherwise>
                </xsl:choose>
        </xsl:variable>
        <xsl:variable name="visible">
                <xsl:choose>
                        <xsl:when test="vis = 1">
                                <xsl:text>t</xsl:text>
                        </xsl:when>
                        <xsl:otherwise>
                                <xsl:text>f</xsl:text>
                        </xsl:otherwise>
                </xsl:choose>
        </xsl:variable>
                <xsl:value-of select="concat('LAYER ',
                name,
                '&#x9;drawing&#x9;',
                gds_num,
                '&#x9;',
                gds_dtyp,
                '&#x9;(',
                color/R,
                ', ',
                color/G,
                ', ',
                color/B,
                ', 255)&#x9;',
                $select,
                '&#x9;',
                $visible,
                '&#x9;',
                fillsyle/name,
                '&#xA;')" />
        </xsl:for-each>
        </xsl:template>
</xsl:stylesheet>
```

**Listing 6.4. XSL file for generating Glade *.tch* files.**

And finally we just have to apply that XSL transformation to the XML file resulting of merging the XML files appeared on Listing 6.1, Listing 6.2, and Listing 6.3. The result obtained can be seen at Listing 6.5.

```
// Generated Technology file for Printed
//
//      Name    Purpose         gds_num gds_dtyp        RGBA            sel?    vis?    fillstyle       linestyle
LAYER   Metal1      drawing         101       10       (255,0,0,255)     t       t       Slash1   Solid ;
LAYER   Metal2    drawing         102       20       (0,0,255,255)       t       t       Slash1   Solid ;
```

**Listing 6.5. Generated .tch file.**

Listing 6.5 consists in a correct *.tch* file, ready to be loaded with the Glade tool. Figure 6.1 and Figure 6.2 show the actions performed in order to load this technology file.

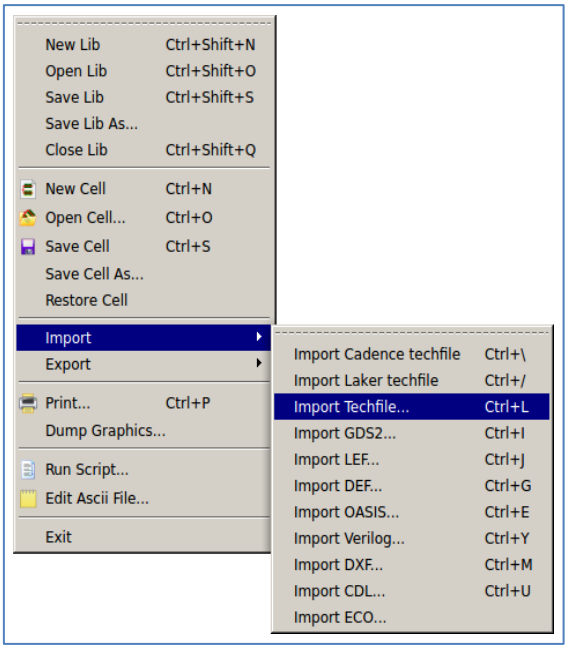



**Figure 6.1. Selecting techfile to import.**

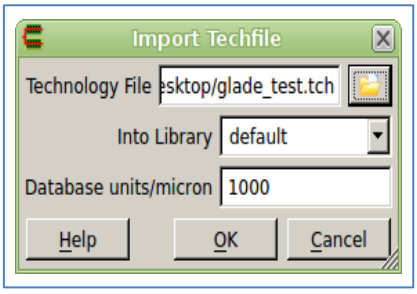



**Figure 6.2. Importing techfile.**

Figure 6.3 shows a layout design, performed with the developed example technology file shown at Listing 6.5.

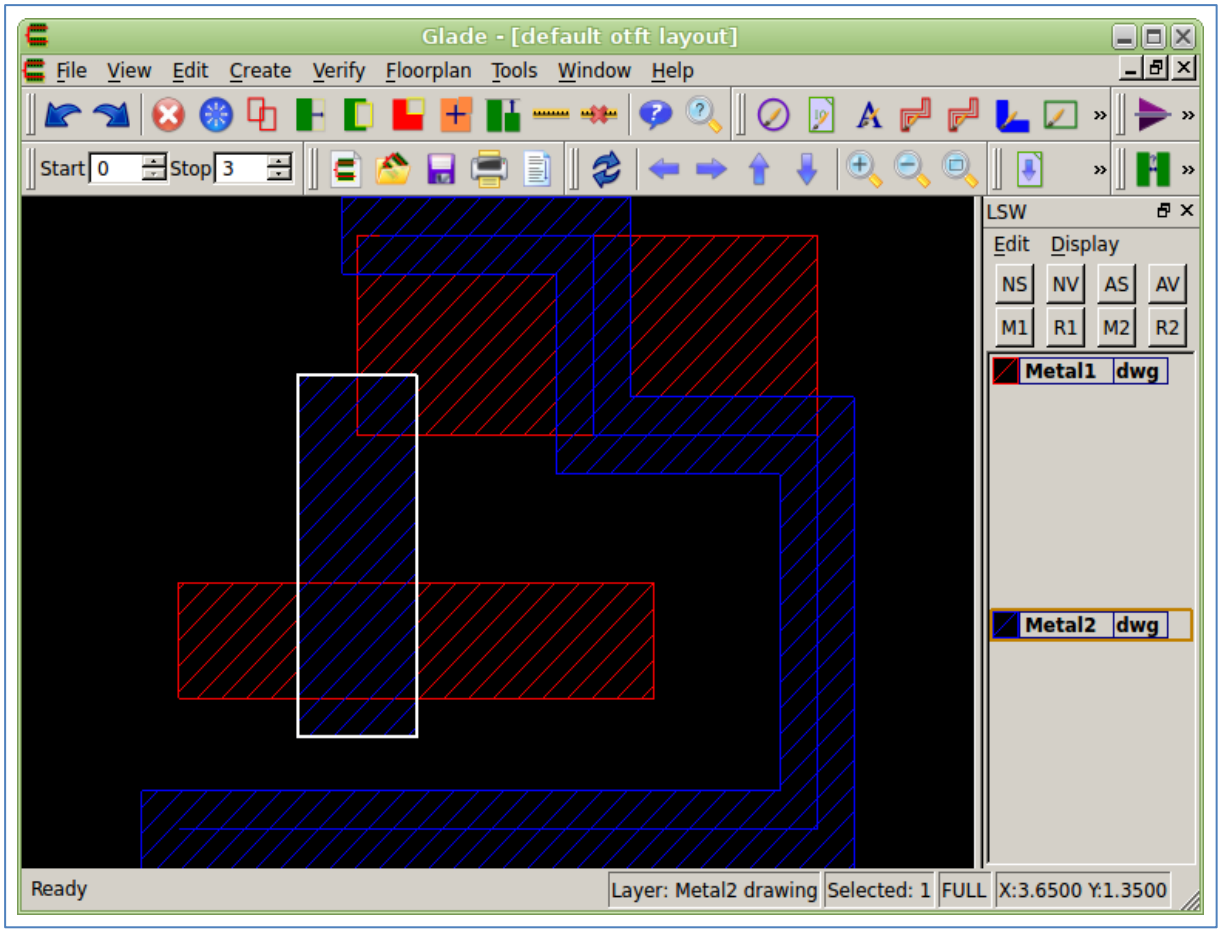



**Figure 6.3. Glade layout example with imported layers.**

# 7 Conclusions and future work

## 7.1 Conclusions

PE or OLAE are new promising processes and techniques to develop electronic systems, reducing the overall cost and providing new features, as flexibility or integration with existing printing processes. As this technology is yet in early development stages, the design flow followed is not clearly defined. Specific tools to aid in the system design either free or commercial do not exist yet, and most of the designs are done with traditional microelectronics tools or general purpose programs.

Since that technology and flows are not already fixed, this application is very useful yet it allows regenerating its whole schema due to the XSD upload functionality. So that, we will be able to export test files until the final schema for the technology files is defined, and after that we'll still be able to generate different related files with this application. Even though we will be able to generate none exactly related to TDK files which may be useful for some technology information deployment such as documentation purposes.

The goals achieved for this project are the following.

1. Environment definition
    a. Styles, formats and specifications defining for dealing with the storage: The style and format for the storage on the entities have been defined as XSD files with some particularities. (see chapter 3.3)
    b. Handling, exchange of information and formats development: Formats are up to change so a regenerate utility has been developed. (see chapter 4.2)
    c. Define TDK information file with requirements: Output files related to some PE abstraction level have been generated. (see chapter 6.2)
2. Information management application development
    a. Define technology file structure within an XSD file: Some files have been generated to show the structure proposal. (see chapter 3.2.1)
    b. Create an export tool: Forms have been generated for loading information in the database, and also some files have been exported. (see chapter 5.1, and chapter 5.2)
    c. Create an import tool and validate: The application also has an import/validation tool. (see chapter 5.3)

## 7.2 Future work

The formats and structures are not completely defined for all PE processes yet, so the developed application has to be ready for being updated once necessary. The application is focused on the management of information, but the output can be updated to new needs derived from the TDK4PE project. Since the information is stored in a formatted way and collected inside XML files, it will be very easy to adapt that information thanks to XSLT files, for instance. A part from output updates, some features that could be added to the website could be the following.

- Generating more formats of files.
- Adapt the database to new needs for the technology information evolutions.
- Link the information database to a repository used by TDK-managers.
- Redefine parser, in order to generate more specific XML files.
- Backend menu for redefining database.
- Create an export utility for XSD files.
- Develop a migration utility for the database.
- Perform more exhaustive database tests in production environment.

# 8 Bibliography

[1] Jordi Mujal Colell. Technology Caracteriztion Oriented to Printed Electronics Design: Touch Panel Case. Master's thesis, Universitat Autònoma de Barcelona, 2009.

[2] Eloi Ramón. Design of Printed Organic HF Rectification Structures in RFID Front-ends. Master's thesis, Universitat Autònoma de Barcelona, 2010.

[3] Hagen Klauk. Organic Electronics. Materials, Manufacturing and Applications. Wiley-VCH, 2007.

[4] A. Hodgson. The role of paper in the future of printed electronics. In 2nd International Workshop on Collaborating over Paper and Digital Documents (CoPADD), London, November, 2007.

[5] E. Ramón, F. Vila, E. Gonzalo, J. Pallarés, L. Terés, J. Carrabina. Inkjet Printed Electronics at UAB-CNM: Technology and Design Flow Developments. CAIAC (UAB) and IMB-CNM, 2011.

[6] F. Vila, J. Pallarés, L. Terés. Layout to Bitmap: A layout design compensator and bitmap converter for Printed Electronics. ICAS Group, IMB-CNM, 2011.

[7] Elkin G. Díaz. Electrical Characterization of Inkjet Printed Structures. Master's thesis, Universitat Autònoma de Barcelona, 2010.

[8] T. Sekitani, Y. Noguchi, U. Zschieschang, H. Klauk and T. Someya. Organic transistors manufactured using inkjet technology with subfemtoliter accuracy. Proceedings of the National Academy of Sciences, 105(13):4976, 2008.

[9] GC Schmidt, M. Bellmann, B. Meier, M. Hambsch, K. Reuter, H. KEmpa, and AC Hübler. Modified mass printing technique for the realization of source/drain electrodes with high resolution. Organic Electronics, 2010.

[10] "XML Media types" http://tools.ietf.org/html/rfc3023 [January 2013]

[11] "XML goals" http://www.w3.org/TR/REC-xml/#sec-origin-goals [January 2013]

[12] "XML APIs" http://en.wikipedia.org/wiki/XML#Programming_interfaces [January 2013]

[13] "SAX Parser" http://www.saxproject.org/ [January 2013]

[14] "DOM Specifications" http://www.w3.org/DOM/ [January 2013]

[15] "XML data binding" http://www.rpbourret.com/xml/XMLDataBinding.htm [January 2013]

[16] "XML transformations" http://www.w3.org/standards/xml/transformation [January 2013]

[17] "Software framework" http://en.wikipedia.org/wiki/Software_framework [January 2013]

[18] Dirk Riehle. "Framework design: A role modeling approach." Dissertation for the degree of Doctor of technical sciences. Universität Hamburg, 2000.

[19] "Most used PHP frameworks" http://www.php-developer.org/most-used-php-framework-the-popular-top-7-list-in-year-2011/ [visited January 2013]

[20] "Zend framework" http://framework.zend.com/ [visited January 2013]

[21] "CodeIgniter" http://ellislab.com/codeigniter [visited January 2013]

[22] "Symfony" http://symfony.com/ [visited January 2013]

[23] "Yii framework" http://www.yiiframework.com/ [visited January 2013]

[24] "Comparison of web application frameworks" http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks#PHP_2 [visited January 2013]

[25] "PHP frameworks" http://phpframeworks.com/ [visited January 2013]

[26] "Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)" http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html [visited January 2013]

[27] "Learn Symfony" http://symfony.com/doc/current/index.html [visited January 2013]

[28] "The bundle system" http://symfony.com/doc/current/book/page_creation.html#the-bundle-system [visited January 2013]

[29] "Doctrine project ORM mapping" http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/basic-mapping.html [visited January 2013]

[30] Javier Eguiluz "Desarrollo web ágil con Symfony2" easybook v3.1, 2011

# 9 Annex A

## 9.1 Symfony2: MVC and architecture

One of the main goals of a framework is to ensure Separation of Concerns. This keeps your code organized and allows your application to evolve easily over time by avoiding the mixing of database calls, HTML tags, and business logic in the same script. As of MVC architecture [26], we have three main components. If we look at Figure 9.2 we can add three modules to understand quickly the Symfony2 architecture [27].

- Front controller: A Front Controller is a short PHP script that lives in the web directory of your project. Typically, all requests are handled by executing the same front controller, whose job is to bootstrap the *Symfony2* application.
- Kernel (model): The Kernel is the core of Symfony2. The Kernel object handles HyperText Transfer Protocol (HTTP) requests using all the bundles and libraries registered to it.
- Controller: A controller is a PHP function that houses all the logic necessary to return a Response object that represents a particular page. Typically, a route is mapped to a controller, which then uses information from the request to process information, perform actions, and ultimately construct and return a Response object.
- Routing: Symfony2 routes the request to the code that handles it by trying to match the requested URL against some configured patterns.
- Action: An action is a PHP function or method that executes, for example, when a given route is matched. The term action is synonymous with controller, though a controller may also refer to an entire PHP class that includes several actions.
- View: Files used to render the response generated by any controller. Views are template files that are able to generate *HTML*, *CSS*, *XML*, *CSV*, *LaTeX*, etc.

The directory structure of a *Symfony2* application is rather flexible, but the directory structure of the Standard Edition distribution reflects the typical and recommended structure of a *Symfony2* application.

- *app/*: The application configuration;
- *src/*: The project's PHP code;
- *vendor/*: The third-party dependencies;
- *web/*: The web root directory.

Everything is organized inside a bundle [28] in *Symfony2*, including both the core framework functionality and the code written for your application. Bundles are first-class citizens in *Symfony2*. This gives you the flexibility to use pre-built features packaged in third-party bundles or to distribute your own bundles.

## 9.2 Symfony2 and HTTP basics

*Symfony2* provides the developer a bunch of classes that make the fact of dealing with the full stack of HTTP protocol an easy task. Before presenting deeply the *Symfony2* we introduce a little the HTTP basics.

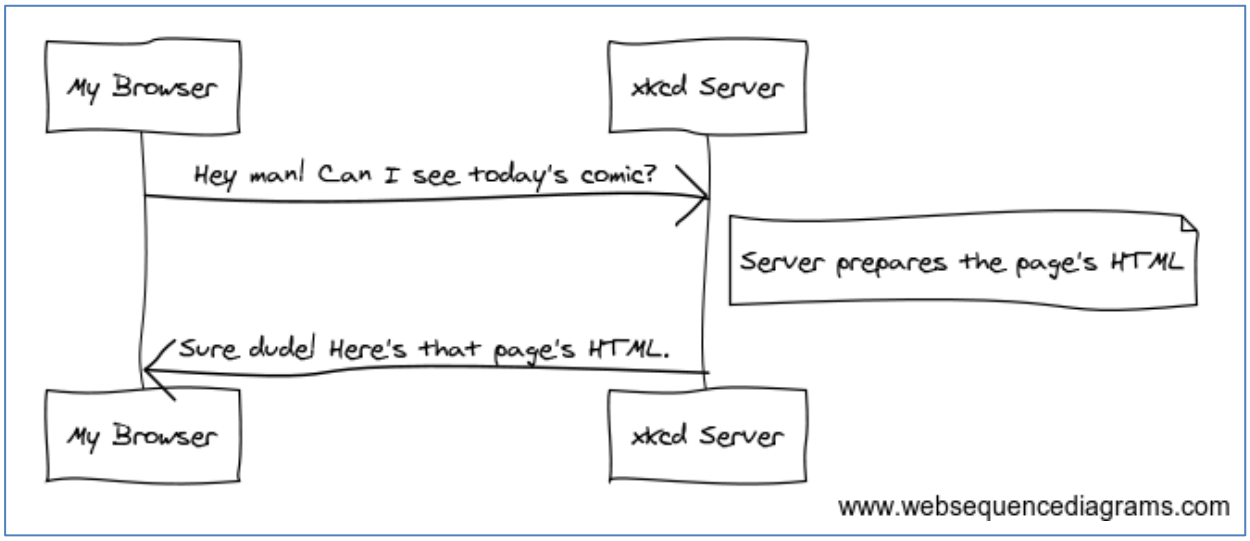



**Figure 9.1. HTTP basic actions diagram.**

Every conversation on the web starts with a request. The request is a text message created by a client (e.g. a browser, an *android* app, etc) in a special format known as HTTP. The client sends that request to a server, and then waits for the response. In HTTP-speak, this HTTP request would actually look something like this.

```
GET / HTTP/1.1
Host: xkcd.com
Accept: text/html
User-Agent: Mozilla/5.0 (Macintosh)
```

**Listing 9.1. HTTP request to xkcd server.**

Once a server has received the request, it knows exactly which resource the client needs via the Uniform Resource Identifier) and what the client wants to do with that resource (via the method). And it sends a response.

```
HTTP/1.1 200 OK
Date: Sat, 02 Apr 2011 21:05:05 GMT
Server: lighttpd/1.4.19
Content-Type: text/html
<html>
  <!-- HTML for the xkcd comic -->
</html>
```

**Listing 9.2. Server response.**

So how do you interact with the "request" and create a "response" when using PHP? In reality, PHP abstracts you a bit from the whole process, but it's not very likely to work at this level. *Symfony2* provides an alternative to the raw PHP approach via two classes that allow you to interact with the HTTP request and response in an easier way. The *Request* class is a simple object-oriented representation of the HTTP request message. With it, you have all the request information at your fingertips.

```php
<?php
use Symfony\Component\HttpFoundation\Request;

$request = Request::createFromGlobals();

// the URI being requested (e.g. /about) minus any query parameters
$request->getPathInfo();

// retrieve GET and POST variables respectively
$request->query->get('foo');
$request->request->get('bar', 'default value if bar does not exist');

// retrieve SERVER variables
$request->server->get('HTTP_HOST');

// retrieves an instance of UploadedFile identified by foo
$request->files->get('foo');

// retrieve a COOKIE value
$request->cookies->get('PHPSESSID');

// retrieve an HTTP request header, with normalized, lowercase keys
$request->headers->get('host');
$request->headers->get('content_type');

$request->getMethod();          // GET, POST, PUT, DELETE, HEAD
$request->getLanguages();       // an array of languages the client accepts
```

**Listing 9.3. HTTP actions with Symfony2 Request class.**

The Request class does a lot of work in the background that you'll never need to worry about. For example, the *isSecure()* method checks the three different values in PHP that

can indicate whether or not the user is connecting via a secured connection (i.e. https). *Symfony2* also provides a Response class: a simple PHP representation of an *HTTP* response message. This allows your application to use an object-oriented interface to construct the response that needs to be returned to the client

```php
<?php
use Symfony\Component\HttpFoundation\Response;
$response = new Response();

$response->setContent('<html><body><h1>Hello world!</h1></body></html>');
$response->setStatusCode(200);
$response->headers->set('Content-Type', 'text/html');

// prints the HTTP headers followed by the content
$response->send();
```

**Listing 9.4. HTTP actions with Symfony2 Response class.**

Like *HTTP* itself, the *Request* and *Response* objects are pretty simple. The hard part of building an application is writing what comes in between. In other words, the real work comes in writing the code that interprets the request information and creates the response. Your application probably does many things, like sending emails, handling form submissions, saving things to a database, rendering *HTML* pages and protecting content with security. How can you manage all of this and still keep your code organized and maintainable? *Symfony2* was created to solve these problems so that you don't have to.

Traditionally, applications were built so that each "page" of a site was its own physical file, for instance:

- index.php
- contact.php
- blog.php

There are several problems with this approach, including the inflexibility of the URLs (what if you wanted to change blog.php to news.php without breaking all of your links?) and the fact that each file must manually include some set of core files so that security, database connections and the "look" of the site can remain consistent. *Symfony2* use a solution based in a front controller: a single PHP file that handles every request coming into your application. So:

| | |
|---|---|
| /index.php | **executes** index.php |
| /index.php/contact | **executes** index.php |
| /index.php/blog | **executes** index.php |

**Table 9.1. Symfony2 *Controller* actions.**

Every request is handled exactly the same way. Instead of individual URLs executing different PHP files, the front controller is always executed, and the routing of different URLs to different parts of your application is done internally. This solves both problems with the original approach. But inside your front controller, how do you know which page should be rendered and how can you render each in a sane way? One way or another, you'll need to check the incoming URI and execute different parts of your code depending on that value. Solving this problem can be difficult.

```php
<?php
// index.php

$request = Request::createFromGlobals();
$path = $request->getPathInfo(); // the URI path being requested

if (in_array($path, array('', '/')) {
    $response = new Response('Welcome to the homepage.');
} elseif ($path == '/contact') {
    $response = new Response('Contact us');
} else {
    $response = new Response('Page not found.', 404);
}
$response->send();
```

**Listing 9.5. Symfony2 different responses depending on requested URL.**

Fortunately it's exactly what *Symfony2* is designed to do. *Symfony2* follows the same simple pattern for every request.

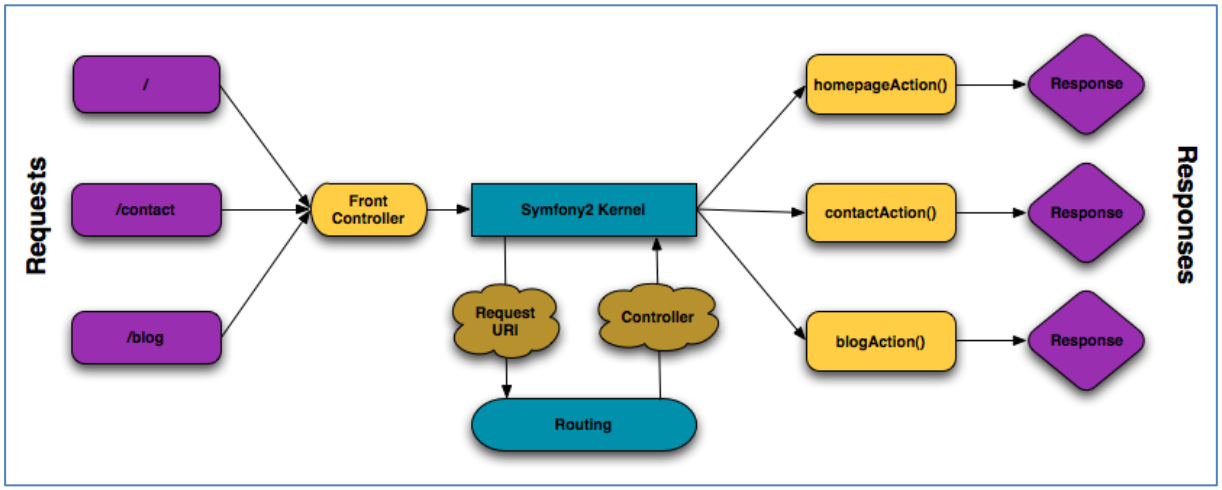



**Figure 9.2. Symfony2 workflow.**

Incoming requests are interpreted by the routing and passed to controller functions that return Response objects. Each "page" of your site is defined in a routing configuration file that maps different URLs to different PHP functions. The job of each PHP function, called a Controller, is to use information from the request - along with many other tools Symfony2 makes available - to create and return a Response object. In other words, the

controller is where your code goes: it's where you interpret the request and create a response. So that, Symfony2 simplified flow could be reviewed as this:

- Each request executes a front controller file;
- The routing system determines which PHP function should be executed based on information from the request and routing configuration you've created;
- The correct PHP function is executed, where your code creates and returns the appropriate Response object.

Without diving into too much detail, let's see this process in action. Suppose you want to add a /contact page to your *Symfony2* application. First, start by adding an entry for /contact to your routing configuration file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/routing-1.0.xsd"

    <route id="contact" pattern="/contact">
        <default key="_controller">AcmeBlogBundle:Main:contact</default>
    </route>
</routes>
```

**Listing 9.6. XML routing file for Symfony2 applications.**

As can be seen, XML format has been used for this configuration file. This is an important *Symfony2* feature for this project as we'll be working and generating XML files.

When someone visits the /contact page, this route is matched, and the specified controller is executed. The *AcmeDemoBundle:Main:contact* string is a short syntax that points to a specific PHP method *contactAction()* inside a class called MainController

```
<?php
class MainController
{
    public function contactAction()
    {
        return new Response('<h1>Contact us!</h1>');
    }
}
```

**Listing 9.7. Action defined inside a Symfony2 Controller.**

Symfony2 goes with a collection of over twenty independent libraries that can be used inside any PHP project. These libraries, called the Symfony2 Components, contain something useful for almost any situation, regardless of how your project is developed. To name a few:

- *HttpFoundation* - Contains the Request and Response classes, as well as other classes for handling sessions and file uploads;
- *Routing* - Powerful and fast routing system that allows you to map a specific URI (e.g. /contact) to some information about how that request should be handled (e.g. execute the contactAction() method);

- *Form* - A full-featured and flexible framework for creating forms and handling form submissions;
- *Validator* - A system for creating rules about data and then validating whether or not user-submitted data follows those rules;
- *ClassLoader* - An autoloading library that allows PHP classes to be used without needing to manually require the files containing those classes;
- *Templating* - A toolkit for rendering templates, handling template inheritance (i.e. a template is decorated with a layout) and performing other common template tasks;
- *Security* - A powerful library for handling all types of security inside an application;
- *Translation* - A framework for translating strings in your application.

Each and every one of these components is decoupled and can be used in any PHP project, regardless of whether or not you use the Symfony2 framework. Every part is made to be used if needed and replaced when necessary.

# 10 Annex B

## 10.1 XML

Acronym for extensible markup language, it's a mark language developed by the w3c, it defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. This language allows users to define the grammar of specific languages to structure big documents. Differently from other languages XML gives support to data bases, being useful when many applications have to communicate between themselves or to integrate information. XML hasn't been created just for internet applications, it has been proposed as a standard for the exchange of structured information between different platforms. It can be used on data bases, text editors, worksheets and almost every imaginable thing. XML is an easy technology which surrounds itself with other ones that complement it and make it much bigger and with greater possibilities. It has a very important role nowadays because it makes possible to share information in a secure, reliable and easy way.

### 10.1.1 Key terminology

The material in this section is based on the XML Specification. This is not an exhaustive list of all the constructs which appear in XML; it provides an introduction to the key constructs most often encountered in day-to-day use. By definition, an XML document is a string of characters. Almost every legal Unicode character may appear in an XML document.

- Processor and Application: The processor analyzes the markup and passes structured information to an application. The specification places requirements on what an XML processor must do and not do, but the application in outside its scope, the processor is often referred to colloquially as an XML parser.
- Markup and Content: The characters which make an XML document are divided into markup and content. Markup and content may be distinguished by the application of simple syntactic rules. All strings which constitute markup either begin with the character '<' and end with a '>', or begin with the character '&' and end with a ';'. Strings of characters which are not markup are content.
- Tag: A markup construct that begins with '<' and ends with '>'. Tags come in three forms.
    1. Start-tags; for example: `<section>`
    2. End-tags; for example `</section>`
    3. Empty-element tags; for example: `<line-break />`
- Element: A logical document component either begins with a start-tag and ends with a matching end-tag or consists only of an empty-element tag. The characters between the start and end-tags, if any, are the element's content, and may contain markup, including other elements, which are called child elements.

  ```
  <Greeting>Hello, world.</Greeting>
  ```

- Attribute: A markup construct consisting of a name/value pair that exists within a start-tag or empty-element tag. In the example below the element *img* has two attributes, *src* and *alt*.

  ```
  <img src="figure.jpg" alt="Example figures" />
  ```

- XML Declaration: XML documents may begin by declaring some information about themselves, as in the following example.

  ```
  <?xml version="1.0" encoding="UTF-8" ?>
  ```

The XML specification defines an XML document as a text that is well-formed, i.e. it satisfies a list of syntax rules provided in the specification. The list is fairly lengthy; some key points are that it contains only properly encoded legal Unicode characters; none of the special syntax characters such as '<' and '&' appear except when performing their markup-delineation roles; the begin, end, and empty-element tags that delimit the elements are correctly nested, with none missing and none overlapping; the element tags are case-sensitive; the beginning and end tags must match exactly. Tag names cannot contain any special characters and cannot start with '–', '.', or a numeric digit; and finally there is a single "root" element that contains all the other elements.

In addition to being well-formed, an XML document may be valid. This means that it contains a reference to a Documents Type Definition (DTD), and that its elements and attributes are declared in that DTD and follow the grammatical rules for them that the DTD specifies. XML processors are classified as validating or non-validating depending on whether or not they check XML documents for validity. A DTD is an example of a schema or grammar. Since the initial publication of XML 1.0, there has been substantial work in the area of schema languages for XML. Such schema languages typically constrain the set of elements that may be used in a document, which attributes may be applied to them, the order in which they may appear, and the allowable parent/child relationships.

A newer schema language, described by the W3C as the successor of DTDs, is XML Schema, often referred to by the initialism for XML Schema instances, XSD (XML Schema Definition). XSDs are far more powerful than DTDs in describing XML languages. They use a rich datatyping system and allow for more detailed constraints on an XML document's logical structure. XSDs also use an XML-based format, which makes it possible to use ordinary XML tools to help process them.

### 10.1.2 Example

Below we can see an XSD file defining the structure of a set of XML files.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Address">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Recipient" type="xs:string" />
        <xs:element name="House" type="xs:string" />
        <xs:element name="Street" type="xs:string" />
        <xs:element name="Town" type="xs:string" />
        <xs:element name="County" type="xs:string" minOccurs="0" />
        <xs:element name="PostCode" type="xs:string" />
        <xs:element name="Country">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:enumeration value="IN" />
              <xs:enumeration value="DE" />
              <xs:enumeration value="ES" />
              <xs:enumeration value="UK" />
              <xs:enumeration value="US" />
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

**Listing 10.1. XSD file example.**

- Each element tag, allows the XML to have a tag with the name indicated on the "name" attribute.
- Each element of this name can appear only one time, unless the attribute *maxOccurs* is set to "unbounded". The tag can also become optional in the XML if the attribute *minOccurs* is set to zero.
- A *complexType* is a definition of an element made by the user.
- A *sequence* is a set of elements that can occur in the XML file.
- An *enumeration* is a set of values that can apply to an element.
- Also the type must be compliant with the content of the XML file.

And this is an XML file that validates with this XSD.

```
<?xml version="1.0" encoding="utf-8"?>
<Address xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="SimpleAddress.xsd">
  <Recipient>Mr. Walter C. Brown</Recipient>
  <House>49</House>
  <Street>Featherstone Street</Street>
  <Town>LONDON</Town>
  <PostCode>EC1Y 8SY</PostCode>
  <Country>UK</Country>
</Address>
```

**Listing 10.2. XML generated from XSD specifications.**