In an earlier talk I introduced documents as components, just the way you make object oriented programs in e.g. Java.

That concept holds more truth than meets the eye.

TeX and \LaTeX have an own programming language supporting:

- Composition
- Inheritance
- Overwrite
- The language is extensible. There is e.g. a version that allows the inclusion of execution of scripting language Lua code inside the source files. Very useful for preparing exams.

An of course produces good looking documents.
Almost unlimited composition

Use case examples:

- A main document that inputs child documents that include grand children that include picture or tables.
- Which allows a quick compiler pass when only working with one part
- Which allow parts of the documents created by other people or tools etc
- Which can input (Reusable) parts from other places or from the source
- Which allow grow or shrink if needed

\[^{1}\text{in case of pictures etc}\]

Input and friends

- You can input a file with
  \[
  \text{\input{filename}}
  \]
  This can be nested almost unlimited. Note that extension is not needed and .tex is assumed.
- You can input a file with
  \[
  \text{\include{chapter}}
  \]
  which you use for chapters.
  - Cannot be nested, but does not err, just complains when file is missing and
  - keeps your chapter numbers intact when you skip some parts with the use of
  \[
  \text{\includeonly{chapter}}
  \]

Packages and configuration

- Using predefined \LaTeX packages, which is the first and foremost form of (code) reuse is done with
  \[
  \text{\usepackage{latexpackage, package2, \texttt{\rightarrow} package3}}
  \]
  which allows multiple packages in one invocation.
- But when the packages accepts and needs configuration parameters it works like
  \[
  \text{\usepackage[utf8]{inputenc}}
  \text{\usepackage[a4paper, includemp, textwidth=140mm, textheight=260mm, includeheadfoot, bindingoffset=1.2cm]{geometry}}
  \]
  - Both forms assume .sty as the extension.
  - .sty files have a different interpretation of some characters (notably the @ character).
Search and ye shall find...

- By default \LaTeX{} searches in the working directory (pwd). Where your pdflatex filename
- \LaTeX{} understands the concept of a search path.
- The standard installation uses well defined source tree.
- TDS or \TeX{} Directory Structure
- To improve compilation speed, the TDS is hashed.
- Finds Any file (\input, \include, \usepackage and even \includegraphics) in TDS.
- You can add your own files to the TDS, but do it in a TDS conforming way: \LaTeX{} files under TDSROOT/texmf/tex/latex/ like TDSROOT/texmf/tex/latex/sebivenlo.
- Add local extension(s) under ~/texmf if TDS conforming. Small differences here per OS/\LaTeX{} installation.

The \LaTeX{} engine

- \LaTeX{} is a macro language define on top of \TeX{}.
- \TeX{} was developed by Professor Donald Knuth and first released 1978
- \LaTeX{} \LaTeX{} was originally written in the early 1980s by Leslie Lamport at SRI International.
- Both \TeX{} and \LaTeX{} allow the definition of (small) reusable parts of code, called macros.
- It is the standard documentation format for certain publishers, certainly in the scientific and mathematical area. This include linguistics.

How to define a macro

- The standard \LaTeX{} way is to use
  \begin{verbatim}
  \newcommand\[2]{\begin{commandname}\begin{parameter}
\textbf{#1} and \textbf{#2}\end{parameter}\end{commandname}}
  \end{verbatim}
- a macro defined in this way is 'final'. \LaTeX{} will complain if you try to define an already defined macro.
- You can however overwrite is using
  \begin{verbatim}
  \renewcommand\[2]{\begin{commandname}\begin{anotherbody}\begin{parameter}
\textbf{#2} and \textbf{#1} in \textbf{a different order}\end{parameter}\end{anotherbody}\end{commandname}}
  \end{verbatim}
- You can also use the \TeX{} primitive
  \begin{verbatim}
  \def\begin{getSimpleName}This is a Simple \textbf{\begin{initial}}\end{initial}\end{verbatim}
- which does not protect against redefinition,
- can be handy for your own simple purposes.
- Using \def{} with parameters is a bit more tricky.
Overwritable defaults

- Very convenient is being able to define macros with defaults or using default definitions
- To do that use the
  \providecommand{\myCommand}{myCommand will have ← this as default value unless there is a previous providecommand for ← myCommand}

- This is extremely handy, because:
  1. macros can be sure that a definition they rely on is available
  2. the user can “overwrite” that definition by using \providecommand before inputting the defining file or package. Example will follow.

Strange things

- \TeX{} has been designed when memory space was at a premium and processors where slow (a 10MHz clock frequency was called turbo in those days, the all famous Aplii II ran at 1023 KiloHerz) and 64 kilobytes were expensive.
- This means characters are commands, or at least categorised into several classes, the catcode. It is also ASCII based.
- I.e. the alpha characters abedef...zABCDEF...Z are in one class. Digits in another.

Strange things, cont’d

- Amongst others it avoids the need of parenthesis in macro definitions
- but: One of the implications is that you cannot use digits in macro names.
- and also that some characters have an unfamiliar class and behaviour.
- e.g. you cannot simply use the backslash or the underscore characters.
- It may not come as a surprise if you know that in XML like languages such as HTML the ⟨, ⟩ and & also have a special meaning.
Naming conventions, of private and public

- There are no name spaces 😊
- The LaTeX naming convention is almost: public LaTeX macros are spell-ed in **lower case** only with a few exceptions².
- User macros may and probably should, use CamelCase. Starting with upper case is fine.
- Macros names that are only used inside a .sty file use @ characters in their names. This prevents these macros being invoked accidentally.

² we will be using one of the exceptions in an example

---

Living in trees

Living in trees can be a life saver in some areas
- Your file system is a hierarchical database
- With directories as keys
- and file contents as value

You should also know that the importance of having a stable **base** if you live in trees, even if it is not at the root.

---

Where o where... 

... can I find my file?

- Many projects use a directory structure by convention.
- Such projects often also use some kind of source code repository
- Working with a repository also means that the contents is checked out into some sandbox.
  - typically in different parent directories on different machines.
  - You do not want someone else to tell you how to set up your computer, do you?
- Meaning that you should **never** use absolute paths for directory structures that must be "portable".
- However, it is good to know that relative paths are very useful, once you know how **high** or **deep** your sandbox sub-tree is.
Where are my figures...

- Nowadays I always checkout the trunk of my repository into my sandboxes as *trunk*. This allows me
  - To have other things besides trunk next to it. Like a tag or a branch of some repository or some related but unversioned stuff (for e.g. teachers: grade lists per year, student repo sandboxes etc)
  - Use a “stable” view on files, to input parts of the files highlighted code in documents. It is nice to have a fixed view, instead of a moving target because someone changed the code.
- Then you can use a relative path to that resource.

Always include base

- The various ways of input (**\input, **\include, **\includegraphics** etc) use the file name to ask the file system for the files contents. This filename can contain directories, including relative directories.
- The file name can also be composed, e.g. by using a macro:
  \begin{verbatim}
  \def{\FigBase}{./figures}
  \end{verbatim}
  if you want you figures piggybacked into the source directory of the chapter the figure belongs to.
- Most of the time I use
  \begin{verbatim}
  \def{\FigBase}{../figures}
  \end{verbatim}
  because I like to store my figures in one directory. A matter of taste and habit.
- Use
  \begin{verbatim}
  \includegraphics[...]{\FigBase/\rightarrow drawing.pdf}
  \end{verbatim}

Some documents grow up

- Too big to handle. (certainly with some office packages).
- No problem at all for a single pass compiler that **\TeX** really is.
- But sometimes the compile simply takes to long or your proofread only takes a few pages instead of the whole caboodle.
- Enter partial compilation
### Use \includeonly and a trick

- Although \include cannot be nested, you can \input from within an included file. So this breaks not much.
- The power in include is that you can make it conditional:
  ```latex
  \usepackage{chapters}
  \includeonly{chapter4, chapter5}
  ```
  to only input chapters 4 and 5.
- This will even keep the (starting) page and chapter numbers correct, if you do that after you kept the aux files produced by a previous run.
- You have to change your main file to change the includeonly statement.

### Use \includeonly and a trick

- It is smarter to pick it up from an optional file like:
  ```latex
  \IfFileExists{includeonly.tex}
  \{
  \def\Logo{\includegraphics[width=\FigBase/logo.pdf]\}
  \}
  ```
  Note the two empty brace pairs. They can be used for if (file available) and else (file not there) and must be present.
- then “the” or any of the (co)authors can set or remove that optional includeonly.tex file in the working directory (sandbox) like (s)he sees fit and thus optimise the local work flow.
- You can always keep that line in your main file, as long as you stick to the optional file name.

### Provide defaults and overrides

- You can also test the existence of files and then take action:
  ```latex
  \def\Logo{\includegraphics[width=\FigBase/logo.pdf]}
  \IfFileExists{./logo.pdf}
  \{\def\Logo{\includegraphics[width=\FigBase/logo.pdf]}
  \}
  ```
  to define a default logo, but override it when one exists in the work directory. Nice for cover pages and headers or footers.
Dry matter ahead: the subfile package

- One package that I find extremely handy is the subfile package.
- Use case is multiple files that may have their own “raison d’etre”, but can also be combined into a larger main document. Think project plan, quality plan, analysis, you get the hang.
- This helps a lot if you have multiple authors, the kind of thing I would like to do in the mobile platform project later in this 4th semester.
- You can use one “preamble file”, defining all your macros and styles and what not.
- Using subfile you then can compile the parts and the whole separately using the same files without touching any of these with an editor. Reuse really DRY.

Subfile main file

\documentclass[times, 12pt]{article}
\% to process this from the current directory
\providecommand{docroot}{.}
\usepackage{subfiles}
\input{\docroot/preamble}
\begin{document}

Note the use of \providecommand to set a base (docroot).
The rest of this main document mainly inputs the sub files like so:

\subfile{sections/collectiveCoaching}

Subfile sub file

\providecommand{docroot{../}}
\documentclass[{docroot/eplop12}]{\fontysPatterns}\%
\subfiles
\begin{document}
\subsection{Collective feedback}\label{sec:collectiveFeedback}

Note the use of \providecommand to set a base (docroot), now pointing one directory up.
This file can be compiled with the same settings as the main file, producing a separate pdf file with only the required content.
**Is \LaTeX\ sick?**

- You might think \LaTeX\ suffers of file Diarrhoea.
- or at least it produces a lot of files.
- For instance when I compile these slides (8 sub files), it creates the one pdf file I want and 13 other files I did not wittingly ask for. (8 .aux files plus .log, .nav, .out, .snm and .toc).
- That is what you get with a single pass compiler design ☹.
- You can of course teach you version control client to ignore these files.
- You can also tell \LaTeX\ to store its junk elsewhere, for instance in an `out`-sub directory. However, find your pdf file there too:
  
  ```latex
  pdfflatex –output–directory=out sheets_en.tex
  ```
  
  - Do not forget to tell subversion to ignore the out directory.

**Give me back my underwear**

Sorry, underscore.

- The underscore character `_` is special. It is meant to produce subscripts in math mode and has little use elsewhere.
- If you want to use it without altering you installation, you must type it as `\_` which is especially annoying if you have to type this code in in \LaTeX\ which is `\textbackslash_` etc and then two curly braces you cannot see☺
- There is a trick to use something else in math mode to do the sub-scripting and give the underscore its normal human meaning, the substitute for forbidden characters like spaces in file and directory names.
- You could use the underscore package. However, read its fine print. (3 pages).

**Installation of the SEBI packages**

- You will be using the sebi venlo macros in prj2 for your requirements document
- put style files in your local tex installation (next slides)
- put sebidefault.tex in 99_texp (already done for prj2 groups)
Installation Linux

- Assuming your distribution uses TeXLive as L\TeX\ installation
- either put the sebivenlo macros in 
  `/usr/share/texlive/texmf/tex/latex/sebivenlo`
  - run `sudo texhash` afterwards. Remember to run `sudo texhash` after each addition like coming from the repository.
- or put them in your personal tex tree under `~/texmf/tex/latex/sebivenlo`
  Running `sudo texhash` not needed.
- Ready to rock.

Installation Mac OS-X

- Assuming your distribution uses MacTex as L\TeX\ installation
- either put the sebivenlo macros in 
  `/usr/local/texlive/texmf-local/tex/latex/sebivenlo`
  - run `sudo texhash` afterwards. Remember to run `sudo texhash` after each addition like coming from the repository.
- or put them in your personal tex tree under `~/Library/texmf/tex/latex/sebivenlo`
  Running `sudo texhash` not needed.
- Ready to rock.

Installation on Windows (tested on Windows 8)

- Assuming your distribution uses MikTeX as L\TeX\ installation
- Make a directory called `LocalTexFiles` on your data drive, say `D:` resulting in this example in `D:\LocalTexFiles`
- Add this directory to the MikTeX ROOTS in the settings.
- Underneath that directory create the required text tree `\text\latex\sebivenlo`
- Run “Refresh FNDB” from the programme `Program Files/MikTex2.x/Maintinance/Settings` or some such. This is the equivalent of running texhash for the other two OS-es.
- see http: //docs.miktex.org/manual/localadditions.html
- Ready to rock. (I hope).
Macros in sebivenlorequirements etc

- We provide three simple `.sty` files to be included into your document.
- In your prj2 project you also find `31_requirements` which contains a main file and a few examples to test your installation.
- Work your way up from those examples.
  - Maybe put user stories and use cases in different sub directories
  - and the collection of those in separate chapters you `
include` into your main.

\[In your prj2 project you find a `99_tex/sebidefault.tex` file that does it for you. Simply input it\]

Usage

Start your document as in the example.

```latex
\def\Application{Library}
\def\Customer{City Library of Venlo}
% include the standard pakgages and style
\input{../99_tex/sebidefault}
% Give the beast a name.
\title{Requirements example}
% Who donit?
\author{Group Anonymous \TeX{}icians}
% show a toc.
\begin{document}
% show a toc.
```

In particular the line `\input{../99_tex/sebidefault}` imports the required settings and some other useful packages. Modify this file to update style etc.

What macros are provided?

- In short:
  1. one multi line macro to create CRC cards. See MOD1 if you want to know what those are.
  2. One macro to input files if they exist and output a warning (in the pdf file) if the file does not yet exist
  3. two environments, one for user stories and one for use cases and a set of one argument helper macros
Usage input helper

▶ The generic name we choose to identify an element of a requirement is **Aspect**.
▶ To input any kind of file there is the macro

\begin{verbatim}
\texttt{\textbackslash InputAspect\{filename\}\{Comment\}. e.g:}
\%
Two parameter: 1: file and 2: ← hint
\section{Lending}
\texttt{InputAspect\{us-lending-by-rfid\}{← Lending by RFID tag}}
\%
The comment is not used anywhere in the document lest the file exists.
\%
If the file does not exist, you get a warning on the console and a box in your document where the aspect was supposed to be placed.
\end{verbatim}

User Story environment

▶ To create a user story start the environment with

\begin{verbatim}
\texttt{\textbackslash begin\{UserStory\} to which you append 5 mandatory arguments wrapped in curly braces.}
\%
\begin{UserStory}
\{US-1\}% code
\{Members can handle lendings via ← an rfid scanning device.\}% description
\{library/lending\}% package
\{1.0\}% Version
\{ODE\}% author
\end{UserStory}
\%
To close the environment use
\end{verbatim}

Usage User Story cont’d

▶ After that you can fill in the user story using the helper macros:

**Story** Describing text of the story

**RefinedBy** Identifies the Use cases that further detail the user story.
Body of User Story

The "Story" body consists of two parts, Story and Refined by.

\begin{Story}
Members of the library have a \leftarrow membership card. It contains an \leftarrow rfid tag, so that it can communicate \leftarrow with the library scanning device. Items can be put on the \leftarrow device, as well as the \leftarrow membership card. A touch screen monitor \leftarrow displays possible actions from \leftarrow which the user can make choices: make a \leftarrow lending or return books to the library. Optional is a printed \leftarrow ticket.
\end{Story}

\begin{RefinedBy}{UC-1.1, UC-1.2, UC-1.3}

Usage User Story cont’d

This results in:

<table>
<thead>
<tr>
<th>User Story Code</th>
<th>Package</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>US-1</td>
<td>Library/lending</td>
<td></td>
</tr>
<tr>
<td></td>
<td>library-lending.fr.tex</td>
<td></td>
</tr>
<tr>
<td>Story</td>
<td>Members of the library have a membership card. It contains an rfid tag, so that it can communicate with the library scanning device. Items can be put on the device, as well as the membership card. A touch screen monitor displays possible actions from which the user can make choices: make a lending or return books to the library. Optional is a printed ticket.</td>
<td></td>
</tr>
<tr>
<td>Refined by</td>
<td>UC-1.1, UC-1.2, UC-1.3</td>
<td></td>
</tr>
<tr>
<td>Version</td>
<td>1.0</td>
<td>Author ODE</td>
</tr>
</tbody>
</table>

Use Case environment

- To create a use case start the environment with \begin{UseCase} to which you append 5 mandatory arguments wrapped in curly braces.

\begin{UseCase}{UC-1.1} Member \leftarrow lends books from the library}{\leftarrow library}{1.0}{ODE}

- To close the environment use \end{UseCase}
Usage Use Case cont’d

- After that you can fill in the user story using the helper macros, each taking one mandatory argument:

  **Actors** and/or roles.

  **Description** or short name/synopsis.

Preconditions what must be true/valid before you start this

**Scenario** The scenario, possibly in steps: use `enumerate*.`

**Extension or sub scenarios**

**Exceptions** That might occur

**Result** The final effect at the successful end the user story.

---

Use Case Scenario text

As example the scenario part in source:

```latex
\begin{enumerate}
\item System displays: Please remove some items.
\item Member selects: 'yes'.
\item System asks if user would like to have a ticket printed.
\item System prints a ticket and closes the session.
\end{enumerate}
```

---

Usage User Story cont’d

This results in:

<table>
<thead>
<tr>
<th>Use case</th>
<th>Code</th>
<th>Description</th>
<th>Preconditions</th>
<th>Scenario</th>
</tr>
</thead>
</table>
| Member lends books from the library | uc-lend-books.tex | Member’s membership card is valid, card and books have been placed on the scanning device. | Make a lending | 1. System scans card.  
2. System asks to make a choice: lend or return books.  
3. Member selects: 'make lending'.  
4. System scans books and presents a list and asks for confirmation that the list is correct.  
5. Member acknowledges that the list is correct.  
6. System reports that the lending has been made, and reports the due date for the items.  
7. System asks if user would like to have a ticket printed.  
8. Member selects: 'yes'.  
9. System prints a ticket and closes the session. |

**Extensions**

- System reports that member has too many books in his/her possession.
  1. System displays: Please remove some items.
  2. Member removes items

**Exceptions**

- List is not correct. Use case proceeds at step 5.

**Result** Books successfully lent out.

**Version** 1.0
Simple CRC card

There is also a macro called \CRC in sebicrccards.sty.

- The macro has 5 arguments:
  1. Title
  2. The reference id, used to be in \LaTeX \ref{not implemented yet
  3. Description
  4. Responsibility of the class
  5. Collaborators with this class.

CRC card example source

\CRC{RFIDScanner}{rfid-scanner}{The \rfid scanner reads information from\rfid
chips in books\and member cards}({The \rfid scanner powers the \rfid chip by induction, which opens a
communication channel with the chip.\Then the scanner requests data
from the chip. The data read is used by the application.})
\end{itemize*}

Usage User Story cont’d

This results in:

<table>
<thead>
<tr>
<th>RFIDScanner</th>
<th>Responsibility</th>
<th>Collaborators</th>
</tr>
</thead>
</table>
| The \rfid scanner reads information from \rfid chips in books and member cards | The \rfid scanner powers the \rfid chip by induction, which opens a communication channel with the chip. Then the scanner requests data from the chip. The data read is used by the application. | • RFIDChip
• Book
• Member Card |
Any questions?

See http://wikibooks.org/wiki/LaTeX and http://www.texample.net/ and various other TeX and LaTeX web sites.

Questions

Ask them now or ...