CoCoTEX

A Commmon LATEX Stylesheet Framework

Version 0.5.0
(2024/07/16)

Contents

1 Overview

1.1 Installation and Usage

1.1.1 Installation

The current build of the package can be obtained from GitHub:

```
git clone https://github.com/transpect/CoCoTeX.git
```

The actual source files can be found in the `src` sub-folder.

The most recent stable version that is in active use on the *xerif* servers can be found at `https://github.com/transpect/xerif-latex`. Note, however, that is version might be several minor versions behind the source code release and contains additional, xerif-specific, files that are not part of the official CoCoTEX build.

The package is installed via

```
latex cocotex.ins
```

This will create the `cocotex.cls` file, as well as some additional modules that follow the naming convention `coco-<module>.sty`. These modules will be explained in greater detail below in section 1.1.3 “Usage”.

The documentation of the framework’s source code can be created via

```
lualatex cocotex.dtx
```

Note 1: You *must* use `lualatex` in order to create the source code documentation!

Note 2: The source code documentation is a technical breakdown of the framework’s source code; it is not the same document as the more user-oriented Manual you are currently reading.

1.1.2 Dependencies

CoCoTEX requires a fairly recent LaTeX kernel. It is recommended to use the latest TeXlive build, but not older than texlive 2022, since it uses some newly added concepts like Hooks and Sockets..

The following packages are requied by the various CoCoTEX modules:

`coco-kernel*`	requires `kvoptions-patch`, `xkeyval`, and `etoolbox`
`coco-common*`	requires `coco-kernel`, `iftex`, `xcolor`, and `graphicx`
`coco-floats*`	requires `coco-common` and `rotating`, `grffile`, `footnote`, `adjustbox`, and `stfloats` and supports `tabularx`, `tabulary`, and `htmltabs`[1]
`coco-meta*`	requires `coco-common`
`coco-heading*`	requires `coco-meta`, and `bookmark`
`coco-notes*`	requires `footnote`, and `endnotes`
`coco-title*`	requires `coco-meta`
`coco-accessibility`	requires lualaTEX, `coco-kernel`, and `ltpdfa`[2]. Older LATEX kernel versions require `atbegshi`, `xparse`, `luatexbase-attr`, and `atveryend`
`coco-lists`	requires `coco-common`, `footnote`, and `endnotes`
`coco-frame`	requires `luatex85`, and `crop`
`coco-script`	requires `coco-kernel`, `babel`, `fontspec` (and therefore lualaTEX or XeLATEX), and `filecontents`

[1] The `htmltabs.sty` is included in CoCoTEX’s main GitHub Repository in the `externals/htmltabs/` folder

[2] `ltpfa` is included in the `externals/ltpdfa` folder in CoCoTeX’s GitHub repository. Note that CoCoTeX uses only the .lua files from that package

The CoCoTEX class file `cocotex.cls` includes most (namely those indicated with an asterisk in the list above) CoCoTEX modules and requires *additionally* the `index` and `hyperref` packages. Note that all those packages might have secondary dependencies.

CoCoTEX itself is designed to run with all LATEX engines, however, in partiular the `coco-script` and `coco-accessibility` modules require lualatex, therefore those modules are either not loaded (coco-script), or it is loaded but not activated (coco-accessibility) by default.

1.1.3 Usage

CoCoTEX follows a modular design. It comes with several `.sty` files that can be used independently from another. However, there is also a LATEX Document Class file `cocotex.cls` which can be used to load the whole framework at once.

Using cocotex.cls

The `cocotex.cls` serves as stand-in for the LATEX default document classes `article` and `book`. It is called with the usual LATEX commmand:

```
\documentclass[<options>]{cocotex}
```

The actual document type can be set with the `pubtype` option:

```
\documentclass[pubtype=<mono|article|collection|journal>]{cocotex}
```

The allowed values are:

`mono`	for monographs, i.e., books that are written by one or multiple authors as a whole,
`collection`	for books that are collections of contributions of multiple authors, and
`article`	for single journal articles,
`journal`	for journals, i.e., collections of multiple journal articles.

Using Single Modules

CoCoTEX is designed to be used modularly. That means you can use selected modules as packages together with LATEX's default or other third-party document classes. Modules are included like any other package, e.g.,

```
\RequirePackage[<options>]{coco-floats}
\RequirePackage[<options>]{coco-headings}
\RequirePackage[<options>]{coco-title}
```

1.2 Design Goals and Purpose

CoCoTEX is a programming framework for LATEX developers who need to build and maintain a number of (not too) different publisher-specific style sheets in partly or fully automatted typesetting processes. Its original purpose is to serve as a rendering backend for the typesetting tool *xerif*[3], but it is also usable as a standalone extension to plain LATEX.

The following features are the main design goals of the CoCoTEX framework:

- Handling of different document types in the same stylesheet:
  - journal articles
  - whole journals
  - chapters by different authors in proceedings and collections,
  - text collections and proceedings, and
  - monographs by (a) single author(s).

[3] see `https://www.le-tex.de/en/xerif.html`

- Handling of recurring complex elements that are difficult to set-up using standard-LaTeX, e. g.
  - headings of all levels with authors, subtitles, quotes, etc.;
  - a four-way distinction of material in a heading's title, its pendant in headers and footers, and their entry in the table(s) of contents, and in the PDF bookmarks; and
  - the possibility to provide classes of text components like headings and floats, similar to classes in HTML/CSS; and
  - the structured handling of meta-data, especially for titlepages.

The framework introduces some new concepts into LaTeX programming that are extensivley influenced by object-oriented design principles. The name CoCoTeX is derived from two of those concepts, namely **Co**ntainers and **Co**mponents. In the next sections, those and other concepts are exlained in more detail.

1.2.1 Basic Concepts

One design goal of the CoCoTeX Framework is to provide an easy and unified way to configure the typesetting of blocks of inter-connected data.

For instance, take *headings*: They always consist of a *Title*, but also may have some sort of *Numbering*, some have a *Subtitle*, some might have a dedicated *Author*, some are followed by a *Quote* or a *Motto*. They may re-appear (partly) in the head-line of a page, as well as in the table of contents, in some cases with slightly altered data.

Containers

Container

Such a bundle of structured information in the CoCoTeX framework is referred to as a **Container**. In the aforementioned example, the information pieces "Title", "Subtitle", "Author" etc., together form a unit "heading".

Containers can be derived from one another. For instance, the abstract concept "heading" might be extended to various levels, like "section", "chapter", "paragraph", or "part". Some of the abstract constituents, like Title or Number, are shared among all those derivations of the concept "heading", while others might not. E.g. "Author" is usualy used on "chapters", but rarely on "sections" or even deeper levels. The mechanism to pass certain properties or constituents from one Container to another is called **Inheritance**.

Inheritance

Components

Component

The inter-connected pieces of information that constitute a Container are called **Components** in the CoCoTeX framework. Most basic components are simple LaTeX macros that take one argument for the content that is to be stored inside that Component for the respective Container.

Some Components can be collections of other Components. An `Author` Component of a `Heading` Container, for example, can contain a first, a middle and a last name, an academic title, an affiliation, or an email address, among many other things. Those complex Components are called **Group Components**, a Group Component together with its (possible) Child Components is called a **Component Group**.

Group Components

Component Group

Some Components may occur multiple times in the same parent Container. A good example are multiple Authors that contibuted to the same chapter in a collection. Those Components are called **Counted Components**. Note that despite the name, it is not necessarily the case that those Counted Components are numbered or even ordered in any way. Rather, "Counted" refers to the way they are processed internally. Due to the way both concepts are implemented in CoCoTeX, Group Components are always also Counted Containers.

Counted Component

Usually, Counted Components are printed in such a way that all instances of the Component are concatenated in some way or another. The result is again stored inside a Component, called an **Collection Component**. Those particuar Components can be used standalone in spite of the single Group Components, and are therefore also called **Overrides**.

Collection Component

Overrides

As an example, take the Author Group Component. It consists of a `FirstName` and a `LastName` component. The Collection Component `FullName` is generated from those two Components, but the user could opt to give the `FullName` directly, as well, and therefore *override* what CoCoTEX would otherwise generate. The same holds for multiple Authors: There is a `AuthorNameList` Collection Component, that holds the a comma separated list of all the Author's `FullName` values. As with the `FullName` Override earlier, the `AuthorNameList` can as well be given directly, thus reducing the need to list each author seperately. Note, however, that the `LastName` Components might be used by other Collection Components (e.g., for a Citation advice), so it is likely, that the user needs to give more than one Override for a layout to work as intended.

Note that each Collection Component is always an Override, but not vice-versa: The version of a heading's title that is printed in the table of contents is usually generated from the `Title` Component of a `Heading` (Child) Container and stored in a Component `TocTitle`. The user can override the `TocTitle` directly by assigning it a value in the `Heading` Instance, therefore, `TocTitle` is an *Override*, but *not* a *Collection Component*, as it is not derived from a Group Component.

Properties

While headings of the same level are usually rendered in the same way for a given publisher style, the actual typeface may vary depending on the Components that are actually filled with content for a given heading: A numbered heading might look slightly different than one without a number; a heading with a subtitle might have different spacing than one without a subtitle, and so on.

Property

How Components are processed and ultimatly rendered is controlled by so-called **Properties**. Properties are mostly short pieces of LaTeX code that are usually set by the stylesheet developer according to a publisher's guidelines and requirements. One of the design goals of CoCoTEX is to keep the code behind those Properties as comprehensible and pointed as possible.

Types, Scope and Modular Inheritance

Type

Properties and Components can be seen as Container-specific Data **Types**. They are only defined within the scope of their parent Container and are usually not accessible from the outside. When a new Container is declared, it can **inherit** the Data Types from one or multiple other containers. A Container that is inherited *from* is called the **Parent Container**, the Container that inherits is the **Child Container**.

Inheritance

Parent Container

Child Container

For example, there might be an abstract Container named `Floats` that defines the Components `Caption` and `ListofCaption` for the float's caption and it's entry in the list of figures or tables, respectively, as well as a bunch of Properties that tell us how the Caption and the list-of entry is to be rendered. We can then declare two more Containers, `Figure` and `Table`, that both inherit both the Components, `Caption` and `ListofCaption`, as well as the Properties and add their own Components and Properties, like `Fig` for the image file, or `Table` for the tabular environment. Thanks to the inheritance mechanism, there is no need to define the `Caption` and `ListofCaption` Components again for the two Child containers.

Style Classes and Attributes

Style Classes

Instances of Containers can further be specialized by Attributes and Style Classes. **Style Classes** are comparable to the `class` term that is used in HTML or CSS, respectively. They are a way to further specify Instances of user-lvel Containers without the need to declare new Child Containers. Container instances of the same Style Class share Properties that diverge somewhat from the standard Property list of their respective Containers.

Attributes

Attributes are a way to alter the functionality of a Container in a pre-defined way. They are defined for a Container, but called per Instance. Usually, they are predefined keys and values (or switches without values) in an optional argument of a Container's LaTeX environment or macro.

Take, for instance, the `Figure` Container we defined earlier. We built in a functionality that allows us to exclude certain Instances from generating a list-of entry, but we need a way to tell LaTeX which instance of the Figure Container should make use of that function. This is where Attributes come

in handy: By simply adding a value-less switch to the optional argument of the `Figure` Container's environment, we can prevent the list-of entry to be generated. Another example for a valued attribute would be the `float-pos` parameter that tells us where the float is to be placed (top, bottom, here, single page).

An example for a Style Class would be a pre-defined set of widths that image files are allowed to be printed. Instead of defining new Figure containers for each allowed width, we can simply define a style class that overrides one single Property for the Container. It is noteworthy that Style Classes are usually activated by Attributes at the Container instance.

1.2.2 Concepts from Object-Oriented Programming

Containers are comparable to the concept of *classes* in object-oriented programming. A concrete heading in a document is an *instance* of that class. **Components** serve as *class variables*, **Properties** can be seen as *instance methods*. **Types** can include macros and control sequences that are somewhat comparable to *class methods*.

The **Inheritance** and **Type** mechanisms are comparable to *Mixins* in some object-oriented programming languages like Ruby.

1.2.3 Implementation of CoCoTEX Concepts in LaTeX Documents

In the CoCoTEX framework, *Containers* are realised in the document source as LATEX environments. *Simple Components* are LATEX commands that take one argument while Group Components are LATEX environments that hold the Commands for its consitutent Components:

```
\begin{<Container>}[<options>]
  \<Component1>{<Content1>}
  \<Component2>{<Content2>}
  \begin{<GroupComponent>}
    \<Component3>{<Content3>}
    \<Component4>{<Content4>}
  \end{<GroupComponent>}
\end{<Container>}
```

The basic idea is that the Content in the Argument of the Component commands within a Container are collected, processed and the output is printed at the end of the corresponding Container environment. Containers allow Components with the same name to be used and processed independently in different Containers.

Components are only allowed within their corresponding Container environments. Outside, Container sensitive Components may have different meaning or even throw an `Undefined control sequence` error.

An example for a Container instance is the following:

```
\begin{Figure}
  \Caption{A~nice image.}
\end{Figure}
```

utilizes a Component named `Caption` with the `Content` *"A nice image."* within a Container named `Figure`.

Properties consist of two parts, the property's name and its value. Some Properties provided by the CoCoTEX modules may have a fixed set of string values, while others are completely free to be set and used.

In this manual, the properties provided by the various modules are documented in the following way:

⚙<name> [<default value>]

A property with the name <name> is set by default to <default value>. The user may chose to set it to any of the <allowed values>.

<any> the user is completely free to set this property to any value she wants.

<dimen> It is expected the property to be a dimension. This may be a length or dimension register, a fix value-unit pair that is understood by TEX, or a macro that expands to a dimension/length.

<num> It is expected the property to be a numeric value. This may be a counter register, a fix value, or a macro that expands to a number.

<allowed values> without angles mean that those are fixes strings that have a special meaning. Those are explained in the descriptions below the property header.

The "data type" <empty> is used to indicate that the property is un-set or empty. This is the default for some of the properties provided by the CoCoTEX modules, but basicly all properties can be set to <empty>.

1.3 Overview: Modules

The following modules are included in CoCoTEX:

1.3.1 User-Level Modules

coco-headings.sty The headings module provides a new way to declare and use chapter, section and paragraph titles. It is described in greater detail in **??** "**??**".

coco-floats.sty The floats module provides some extended handling for floating objects like tables or figures. It is described in greater detail in **??** "**??**".

coco-title.sty The title module provides meta data handlers for title pages. It is described in greater detail in **??** "**??**".

coco-frame.sty

coco-notes.sty The notes module handles the easy switching between footnotes and endnotes, as well as the position where and in what way endnotes are printed. It is described in greater detail in **??** "**??**".

1.3.2 Backend Modules

coco-kernel.sty The kernel module is the heart of the CoCoTeX framwork. As such, it is a hard dependency for all other modules and loaded automatically.

coco-common.sty The common module is a collection helper macros and functions, that are not perse part of the CoCoTEX Framework, but utilised by multiple other modules. The common module is loaded automatically by some of the other modules, but not by all.

coco-meta.sty The meta module collects methods and concepts that are used by both the title and headings modules. It is therefore auto-loaded by both modules.

\ccDeclareContainer

2 Custom Containers

As we already discussed in chapter 1, Containers are representations of typographical elements that share a more or less fixed set of components that are supposed to be rendered in a similar way.

In this section, we discuss how to declare custom Containers.

\ccDeclareContainer A new, custom Container can be declared with the `\ccDeclareContainer` command:

```
\ccDeclareContainer{<name>}{<body>}
```

where `<name>` is the name of the Container, and `<body>` is a list of Container Type Declarations.

\ccDeclareType Data Types are declared with the `\ccDeclareType` command:

```
\ccDeclareType{<name>}{<body>}
```

where `<name>` is the name of the Data Type, and `<body>` a list of type-specific variable declarations. The most commonly used Data Types are *Properties*, *Attributes*, and *Components*, but essentially, they can be named anything.

\ccInherit Another common command inside the Container body is the `\ccInherit` command. It takes two arguments: `{#1}` is a comma-separated list of Data Types, and `2` is a comma-separated list of Container names. The newly defined Contianer will then inherit all Data Type declarations from the parent Containers. For instance,

```
\ccDeclareContainer{Parent 1}{%
  \ccDeclareType{Properties}{...}%
  \ccDeclareType{Components}{...}%
  \ccDeclareType{Junk}{...}%
}
\ccDeclareContainer{Parent 2}{%
  \ccDeclareType{Properties}{...}%
  \ccDeclareType{Components}{...}%
}
\ccDeclareContainer{Child}{%
  \ccInherit{Properties,Components}{Parent 1,Parent 2}%
}
```

means that the Container named `Child` will inherit the *Components* and *Properties* Data Types from both Containers `Parent 1` and `Parent 2`, respectively, but not the Data Type `Junk` from `Parent 1`.

Note that both `\ccDeclareType` and `\ccInherit` can only be used inside the body of a Container Declaration!

Index

In general, page entries in **bold face** refer to main sections that describe the index term in greater detail. Roman page numbers refer to pages where the entry is used.

General Index

Macro and Environment Index

In this index, the `cc(@)`-Prefixes are omitted when sorting the entries.