

Contents

1 Overview	2
1.1 Installation and Usage	2
1.1.1 Installation	2
1.1.2 Dependencies	2
1.1.3 Usage	3
1.2 Design Goals and Purpose	3
1.3 Overview: Modules	4
1.3.1 User-Level Modules That Use Containers	4
1.3.2 User-Level Modules That Do Not Use Containers	4
1.3.3 Back End Modules	5
2 CoCoTeX's Design Principles	6
2.1 Containers and Data Types	6
2.1.1 Containers	6
2.1.2 Data Types	6
2.1.3 Inheritance	6
2.2 Components	7
2.2.1 General Concept	7
2.2.2 How to Use Components	8
2.3 Properties	9
2.3.1 General Concept	9
2.3.2 How to Use Properties	9
2.3.3 Style Classes and Attributes	11
2.4 Accessibility Features	12

Part I CoCoTeX's Container-Based Modules

Part II Additional Modules

Part III CoCoTeX's Backend

3 Custom Containers	16
Index	18
General Index	18
Macro and Environment Index	19

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 this 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 required by the various CoCoTeX modules:

<code>coco-kernel*</code>	requires <code>kvoptions-patch</code> , <code>xkeyval</code> , and <code>etoolbox</code>
<code>coco-common*</code>	requires <code>coco-kernel</code> , <code>iftex</code> , <code>xcolor</code> , and <code>graphicx</code>
<code>coco-floats*</code>	requires <code>coco-common</code> and <code>rotating</code> , <code>grffile</code> , <code>footnote</code> , <code>adjustbox</code> , and <code>stfloats</code> and supports <code>tabularx</code> , <code>tabulary</code> , and <code>htmltabs</code> ¹
<code>coco-meta*</code>	requires <code>coco-common</code>
<code>coco-heading*</code>	requires <code>coco-meta</code> , and <code>bookmark</code>
<code>coco-notes*</code>	requires <code>footnote</code> , and <code>endnotes</code>
<code>coco-title*</code>	requires <code>coco-meta</code>
<code>coco-accessibility</code>	requires <code>luaLaTeX</code> , <code>coco-kernel</code> , and <code>ltpdfa</code> ² . Older LaTeX kernel versions require <code>atbegshi</code> , <code>xparse</code> , <code>luatexbase-attr</code> , and <code>atveryend</code>
<code>coco-lists</code>	requires <code>coco-common</code> , <code>footnote</code> , and <code>endnotes</code>
<code>coco-frame</code>	requires <code>luatex85</code> , and <code>crop</code>

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

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

`coco-script` requires `coco-kernel`, `babel`, `fontspec` (and therefore `luaLATEX` or `XeLATEX`), and `filecontents`

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

CoCoT_EX itself is designed to run with all L^AT_EX engines, however, in particular 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

CoCoT_EX follows a modular design. It comes with several `.sty` files that can be used independently from another. However, there is also a L^AT_EX 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 L^AT_EX default document classes `article` and `book`. It is called with the usual L^AT_EX command:

```
\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

CoCoT_EX is designed to be used modularly. That means you can use selected modules as packages together with L^AT_EX'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

CoCoT_EX is a programming framework for L^AT_EX developers who need to build and maintain a number of (not too) different publisher-specific style sheets in partly or fully automated 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 L^AT_EX.

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

- Handling of different document types in the same stylesheet:
 - journal articles

³see <https://www.le-tex.de/en/xerif.html>

- whole journals
- chapters by different authors in proceedings and collections,
- text collections and proceedings, and
- monographs by (a) single author(s).
- 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 and accessible PDFs.

To achieve those goals, the framework introduces some concepts into \LaTeX programming that are extensively influenced by object-oriented design principles. The name CoCo \TeX is derived from two of those concepts, namely **C**ontainers and **C**omponents.

The most recent versions of the \LaTeX kernel (Texlive 2024 and later) has seen some quite similar concepts being introduced after they existed before in the form of the `xtemplate` package. The core mechanics of CoCo \TeX and the relationship between CoCo \TeX and \LaTeX Templates are explained in greater detail in 2 “CoCo \TeX 's Design Principles”.

1.3 Overview: Modules

As mentioned earlier, CoCo \TeX is modular. The following modules are included in CoCo \TeX :

1.3.1 User-Level Modules That Use Containers

<code>coco-headings.sty</code>	The <code>headings</code> module provides a new way to declare and use chapter, section and paragraph titles. It is described in greater detail in ?? “??”.
<code>coco-floats.sty</code>	The <code>floats</code> module provides some extended handling for floating objects like tables or figures. It is described in greater detail in ?? “??”.
<code>coco-title.sty</code>	The <code>title</code> module provides meta data handlers for title pages. It is described in greater detail in ?? “??”.
<code>coco-lists.sty</code>	The <code>lists</code> module provides support for list environments. It is described in greater detail in ?? “??”.

1.3.2 User-Level Modules That Do Not Use Containers

<code>coco-frame.sty</code>	provides some helper tools to make the type area visible and add crop marks for printing. It is explained in more detail in ?? “??”.
<code>coco-notes.sty</code>	The <code>notes</code> module handles the easy switching between footnotes and end-notes, as well as the position where and in what way endnotes are printed. It is described in greater detail in ?? “??”.
<code>coco-script.sty</code>	This module provides support for non-latin scripts utilizing Google's Noto fonts. It is described in greater detail in ?? “??”.
<code>coco-accessibility.sty</code>	The <code>accessibility</code> module provides support to generate PDFs that conform to the PDF/UA standard and interfaces for the <code>ltpdfa</code> package. It will be described in greater detail in ?? “??” with some remarks in 2.4 “Accessibility Features”.

1.3.3 Back End Modules

<code>coco-kernel.sty</code>	The <code>kernel</code> module is the heart of the CoCoTeX framework. As such, it is a hard dependency for all other modules and loaded automatically. The kernel module is explained in greater detail in 3 “Custom Containers”.
<code>coco-common.sty</code>	The <code>common</code> module is a collection helper macros and functions, that are not per-se 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. It is explained in greater detail in ?? “??”.
<code>coco-meta.sty</code>	The <code>meta</code> module collects methods and concepts that are used by both the <code>title</code> and <code>headings</code> modules. It is therefore auto-loaded by both modules. It is explained in greater detail in ?? “??” and ?? “??”, respectively.

2 CoCoT_EX's Design Principles



In this chapter, we discuss the major design principles of the CoCoT_EX framework, how they are represented in the remainder of the Manual, and introduce some user-level macros to influence the behaviour of those design concepts.

2.1 Containers and Data Types

One design goal of the CoCoT_EX 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. Another part of the idea behind a heading is that it is always rendered in the same way, for instance, the Title is always bold (or italic or normal) and in a certain font size; if there is an author, it always preceeds (or succeeds) the title in this or that font; and so on.

2.1.1 Containers

Container Such a bundle of structured constituents and the instructions how they are presented are called **Containers** in the CoCoT_EX framework. In this manual, whenever a pre-defined Container is mentioned, it is colored orange and preceeded by the Symbol , for instance  **Heading**. By convention, Container names are (almost) always Capitalized.

Instance One particular manifestation of an abstract Container in a document is called an **Instance** of that Container. For instance, the (particular) heading with the (particular) title “Introduction” is an Instance of the abstract concept (or “Container”) “Heading”.

The general concept of Containers and Instances is quite similar to the notion of *classes* and *objects* in Object Oriented Programming, whereas classes are representations of abstract concepts and objects are concrete instantiations of that abstract class.

2.1.2 Data Types

Type As mentioned above, a Container in CoCoT_EX is defined by a specific set of (possible) constituents and a specific set of (conditional) rendering instructions. Those two are fundamentally different in nature, but they share that they define what a Container actually is. CoCoT_EX refers to those different sets of Container-defining building blocks as (Data) **Types**. The aforementioned set of structured constituents of a Container form one Type, which we call *Components*, while the set of instructions that tell the engine how those Components are to be processed and rendered form another Type, which we call *Properties*.

2.1.3 Inheritance




Containers can be derived from one another by passing all or some Data Types from one Container to 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 usually used only on the “chapter” level, but rarely on “section” or even deeper heading levels. The

Inheritance mechanism to pass certain Types from one Container to another is called **Inheritance**. A Container that is inherited *from* is called the **Parent Container**, the Container that receives the Type from the Parent is called the **Child Container**.

For example, there might be an abstract Container named `Floats` that comprises of the constituents `Caption` and `ListofCaption` for the float's caption and its entry in the list of figures (or tables), respectively, as well as a bunch of instructions that tell us how the list-of entry is generated from the `Caption`. We can then declare two more Containers, say, `Figure` and `Table`, that both inherit the constituents, as well as the generator instructions from the abstract Container. They then can add their own constituents and instructions, like `Fig` for the image file of the `Figure` Container, or `tabular` for the content body of the `Table` Container. Thanks to the inheritance mechanism, there is no need to define the `Caption` Component and the `ListofCaption` Generator for the two Child containers separately.

2.2 Components

2.2.1 General Concept

As mentioned above, the inter-connected pieces of information that constitute a Container are called **Components** in the CoCoTeX framework. In this manual, Components are marked by the symbol  and colored blue. By convention, Component names are Capitalized or use CamelCase when they are more complex, e.g.,  `Author` or  `TocTitle`.

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

Some Components may occur multiple times in the same parent Container. A good example are multiple Authors that contributed to the same article in a journal. Those Components are called **Counted Components**. Despite the name, it is not necessarily the case that those Components are enumerated, the name "Counted" merely refers to the way they are processed internally. Due to the way both concepts are implemented in CoCoTeX, Group Components are always also Counted Components.

Usually, Counted Components are printed in such a way that all instances of the Component that occur in the same Container Instance are concatenated. The result is stored inside a special Component that is called a **Collection Component**. Those particular Components can be used standalone in spite of the Group Components, and are therefore also called **Overrides**.

As an example, take an `Author` Group Component, which 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, and therefore *override* what CoCoTeX would otherwise generate from the `FirstName` and `LastName` Components. The same holds for multiple Authors: There is a Collection Component `AuthorNameList` that holds the comma separated list of all the Author's `FullName` values in the same Container Instance. As with the `FullName` Override earlier, the `AuthorNameList` can as well be given directly, thus eliminating the need to list each author separately.

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.

The `coco-meta` module introduces another type of Components, the **Labeled Component**. It is a simple Component with a `Name`, that automatically defines an additional Component `NameLabel`. Usu-

ally, Labeled Components are expanded together with the label's value first, then a separator, and then the simple Component's value.

2.2.2 How to Use Components

Most Components in Container Instances and Group Components are \LaTeX macros that take one Argument, e.g.

```
\begin{Figure}
  \Caption{One nice figure.}
  \Fig{\includegraphics{example.eps}}
\end{Figure}
```

In this example, the Container  **Figure** is instantiated with two Components,  **Caption** and  **Fig**. The Argument of the Component Macros is the Component's *value*.









`\ccComponent` Another way to pass Content to a Component is to use the `\ccComponent` macro:






```
\begin{Figure}
  \ccComponent{Caption}{One nice figure.}
  \ccComponent{Fig}{\includegraphics{example.eps}}
\end{Figure}
```

It takes two arguments, the first being the name of the Component, the second being the Content that is assigned to the component in that particular Container Instance.

Container Groups are usually \LaTeX environments inside the Container instance environment, for instance:

```
\begin{Heading}{Section}
  \ccComponent{Title}{Section Title}
  \begin{Author}
    \FirstName{Jane}
    \LastName{Doe}
  \end{Author}
  \begin{Author}
    \FirstName{John}
    \LastName{Doe}
  \end{Author}
\end{Heading}
```

In this example, the Container  **Section** (for the special syntax of  **Heading**'s Child Container Instances, see ?? "??") is instantiated with three Components, two of which are Instances of Counted Components. The value *Section Title* is assigned to the simple Component  **Title**. The Counted Component  **Author** is instantiated twice, once with the values *Jane* and *Doe* assigned to the Components  **FirstName** and  **LastName**, respectively, and second, with the values John and Doe assigned to two different instances of the Components  **FirstName** and  **LastName**. As seen in this example, Group Components can repeat multiple times.

When processed, the Values of the Components  **FirstName** and  **LastName** are concatenated and the result is stored in a Collection Component  **FullName** inside each instance of the Group Component  **Author**, so the following example is equivalent to above's example, assuming that the instructions on how to build the Collection Component  **FullName** results in the same strings:

```
\begin{Heading}{Section}
  \ccComponent{Title}{Section Title}
  \begin{Author}
    \FullName{Jane Doe}
  \end{Author}
  \begin{Author}
  \end{Author}
```



```
\FullName{John Doe}
\end{Author}
\end{Heading}
```

The two instances of the Group Component ➡ **Author** are further concatenated to a Container-level Collection Component ➡ **AuthorNameList**. Assuming, the instructions that tell the engine how to build this Collection Component does so by joining the two ➡ **FullName** values with an “and”, the following line is equivalent to both previous examples:

```
\begin{Heading}{Section}
\ccComponent{Title}{Section Title}
\ccComponent{AuthorNameList}{Jane Doe and John Doe}
\end{Heading}
```

2.3 Properties

2.3.1 General Concept

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 the Components of a Container Instance are processed and ultimately rendered is controlled by so-called **Properties**. Properties are mostly short pieces of L^AT_EX code that are usually set by the stylesheet developer according to a publisher’s guidelines and requirements. One of the design goals of CoCoT_EX is to keep the code behind those Properties as comprehensible and pointed as possible. In this manual, pre-defined Properties are indicated by violet text color and the symbol ⚙, e.g., ⚙ **title-format**. By convention, Property names are always lower case and words are separated by hyphen.

2.3.2 How to Use Properties

As mentioned in 2.1.2 “Data Types”, Properties are Types. Most CoCoT_EX modules pre-define Properties. In order to change the behaviour of pre-defined Properties, the Type needs to be appended. This is done with the macro `\ccAddToType`, which takes three arguments:

```
{#1} is the name of the Type
{#2} is the name of the Container whose Type list should be altered
{#3} is a list of instructions
```

For instance

```
\ccAddToType{Properties}{Heading}{\foo}
```

add `\foo` to the end of the **Heading** Container’s Properties list.

`\ccAddToProperties` For Property lists in particular there exists a shorter macro that takes two arguments:

```
{#1} is the name of the Container
{#2} is a list of instructions
```

The line

```
\ccAddToProperties{Heading}{\foo}
```

is equivalent to above’s example.

In the following, we discuss the macros that can be used in the list of instruction in the last argument of `\ccAddToType{Properties}` and `\ccAddToProperties`, respectively.

`\ccSetProperty` The most basic way to set the value of a Property is the macro, that takes two arguments:

- `{#1}` the name of the Property to set
- `{#2}` the new value of the Property

Returning to our examples from 2.2.2 “How to Use Components”, the instructions that tell the engine how to put together the values of `\ccFirst` and `\ccLast` to generate the value of the `\ccFullName` Component, might look something like that:

```
\ccAddToProperties{Heading}{%
  \ccSetProperty
    {author-full-name-format}
    {\ccUseComp{FirstName}\ccWhenComp{MidName}{ \ccUseComp{MidName}} \ccUseComp{
      LastName}}
}
```

Note that the definition of the Property `author-full-name-format` contains a conditional `\ccWhenComp` that takes a Component name as first argument `{#1}` and processes its second argument `{#2}` if and only if the Component has a value assigned to in a given Container instance: If the `\ccMid` Component is non-empty, it is written in between the `\ccFirst` and `\ccLast` values, which are otherwise put next to each other with a simple space as separator.

`\ccUseComp` The second important macro that is used in this example is which takes as its only argument `{#1}` the name of a Component and is expanded to the value of that Component.

`\ccIfComp` A macro similar to `\ccWhenComp` is, which takes three arguments: `{#1}` is the name of the Component, `{#2}` is the branch that is expanded when the Component is assigned a value to in the processed Container, and `{#3}` is the code that is expanded when the Component is *not* assigned a value to within the currently processed Container or Group Component instance.

`\ccUnlessComp` Another variant is which also takes two arguments: `{#1}` for the Component’s name, and `{#2}` that is only expanded when the Component is *not* assigned a value in the current instance.

`\ccGetComp` The macro is a combination of `\ccWhenComp` and `\ccUseComp`. It takes one argument `{#1}`, the name of the Component. The two lines in the following example are equivalent:

```
\ccWhenComp{Subtitle}{\ccUseComp{Subtitle}}
\ccGetComp{Subtitle}
```

`\ccUseProperty` Often, Properties make use of other Properties. In the definition of a Property this is done with the macro, which takes as its only argument `{#1}`, the name of the Property. In the following example, this is used to pass the font commands for the main title down to the facility that expands the value of the `\ccTitle` Component:

```
\ccSetProperty{title-face}{\itshape\bfseries}
\ccSetProperty{main-title-format}{%
  \bgroup
    \ccUseProperty{title-face}
    \ccUseComp{Title}
  \egroup}
```

This will internally be resolved to

```
\ccSetProperty{main-title-format}{%
  \bgroup
    \itshape\bfseries
    \ccUseComp{Title}
  \egroup}
```

This mechanism allows Properties to be relatively clear and slender: We can change the font of the `\Title` by changing the `title-face` Property without the need to take care about the expansion and composition of the actual title. And, via Inheritance, we can change the `title-font` once for all Child Containers, while each Child Container might define the `main-title-format` Property slightly differently.

`\ccAppToProp` To add code to a Property, the macros and can be used. This allows to add code to the beginning and end of an existing Property, respectively, without changing its original definition. Both macros take two arguments:

`{#1}` is the name of the Property
`{#2}` is the code that should be added to the beginning or end of the previous definition, respectively.

```
\ccSetProperty{title-face}{\bfseries}
\ccPreToProp{title-face}{\itshape}
\ccAppToProp{title-face}{\large}
```

is equivalent to

```
\ccSetProperty{title-face}{\itshape\bfseries\large}
```

— TODO:

```
\ccSetProperty{<name>}{<val>}
\ccSetPropVal{<name>}{<val>}
\ccSetPropertyX{<name>}{<val>}
```

With `\ccSetPropVal`, the second argument is expanded *once* prior to the assignment. Use this if you want to assign the current value of a L^AT_EX control sequence (e.g., a length register) to a property.

Properties consist of two parts, the property’s name and its value. Some Properties provided by the CoCoT_EX 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 T_EX, 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 CoCoT_EX modules, but basicly all properties can be set to `<empty>`.

2.3.3 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. In this documentation, pre-defined Attributes are colored green and preceded by the symbol $\text{\textcircled{A}}$, for instance $\text{\textcircled{A}float-pos}$.

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.

2.4 Accessibility Features

TODO

Part I: CoCoT_EX's Container-Based Modules

Part II:

Additional Modules

Part III: CoCoT_EX's Backend

3 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 Container 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!

`\ccIfProp` Sometimes, it is necessary to check whether a Property yields any Content, at all. This can be done with the macro `\ccIfProp`, which takes three arguments:

- `{#1}` the name of the Property
- `{#2}` code that is executed when the Property exists and yields a non-empty string
- `{#3}` code that is executed when either the Property does not exist or if yields an empty string

An example could be:

```
\ccSetProperty{}{}
```

`\ccIfPropVal` Some Properties have pre-defined values. To check for a given value, the macro can be used. It has four arguments:

- `{#1}` the name of the Property

- `{#2}` the string to compare the Property's expansion against
- `{#3}` code that is executed when the Property's expansion matches `{#2}`
- `{#4}` code that is expanded when either the Property does not exist or its expansion value does not match .

2 WARNING! Since Properties are LaTeX macros that are always defined `\long`, the comparison value `{#2}` is also stored in a temporary macro using `\long\def`. Therefore, `\ccIfPropVal` cannot be used in contexts that fully expand the Property list, in particular `\ccApplyCollection`!

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

Numbers		Instance	6
\2	17		
A		Kernel	5
Accessibility	4		
Attributes	12	L	
C		Labeled Component	7
\ccAddToProperties	9	Lists	4
\ccAddToType	9		
\ccAppToProp	11	M	
\ccGetComp	10	Meta	5
\ccIfComp	10	Module	
\ccIfProp	16	Accessibility	4
\ccIfPropVal	16	Common	5
\ccPreToProp	11	Floats	4
\ccSetProperty	10	Frame	4
\ccUnlessComp	10	Headings	4
\ccUseComp	10	Kernel	5
\ccUseProperty	10	Lists	4
Child Container	7	Meta	5
Collection Component	7	Notes	4
Common	5	Title	4
Component	7		
Component Group	7	N	
Container	6, 16	Notes	4
Counted Component	7		
D		O	
Dependencies	2	Overrides	7
F			
Frame	4	P	
G		Parent Container	7
Group Components	7	Property	9
H		pubtype	see Class Options
Headings	4		
I		S	
Inheritance	7	Style Classes	11
Installation	2		
		T	
		Title	4
		Type	6
		U	
		Usage	3

Macro and Environment Index

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

A		I	
\ccApplyCollection	17	\ccInherit	16, 16
C		U	
\ccComponent	8	\ccUseComp	10
D		W	
\ccDeclareContainer	16	\ccWhenComp	10, 10, 10
\ccDeclareType	16, 16		