Template for SEW lecture documents

This template is intended to simplify crafting well-presented learning resources for our software engineering students. The examples here are meant to be read along with the document's source code, e.g. in the web app or using Tinymist's preview. Typst code snippets will only be presented in rare circumstances.

I recommend using semantic line breaks when writing, since it makes versioning easier:

> When writing text with a compatible markup language, add a line break after each substantial unit of thought.[1]

Attention

Among the tools in this template is the `colorbox`, an opinionated wrapper around `showybox`[2]. You can pass a named `color: ...` argument (which sets all relevant showybox colors), as well as all arguments accepted by showybox itself, to customize it.

Color boxes are great to **summarize** and **focus attention** on key concepts.

I don't prescribe a color philosophy, but red is useful to call out Don'ts, e.g.:

Don't overdo color boxes. In moderation, interrupting the text helps keep attention, but: **if everything is highlighted, nothing is!**

I like to add **emphasis** so that the bold parts form a **shortened message** on their own; useful for getting points across even when students only **skim** the document.

Code

Code snippets are highlighted using `zebraw`[3], with a bit of styling applied. You can also select a subset of lines, as these two examples show:

```
1 public class Main {
2   public static void main(String[] args){
3     System.out.println("Hello World!");
4   }
5 }
```

```
2   public static void main(String[] args){
3     System.out.println("Hello World!");
4   }
```

`zebraw` specifies ranges in $[\text{lower}, \text{upper})$ form (i.e. half-open as usual in programming). To make this a bit more convenient, the `lines()` function accepts strings like `"2-4, 6, 9-11"` that produces the appropriate ranges—but note that multiple disjoint ranges are not supported until pull request typst-zebraw#32 lands. Only strings like `"2-4"` will work for now.

*Based on earlier work by Jane Doe (comment out otherwise)

[1] https://github.com/sembr/specification

[2] https://typst.app/universe/package/showybox

[3] https://typst.app/universe/package/zebraw

Notes in code

This template adds some handling for putting notes onto code, powered by `pinit`[4]: write `PINn` somewhere (replacing `n` by a number), and it will form an anchor for your notes. You should just take care of two things:

- notes must appear on the same page as the anchors, so wrapping the code block in `block(breakable: false)` is recommended, and
- the `PINn` is part of the code when syntax highlighting happens, so avoid making it part of another token: `int foo;` and `intPIN1 PIN2foo;` look differently.

```
1 public class Main {          ← Boring
2   public static void main(String[] args) {   ← Boilerplate
3     System.out.println("Hello World!");   ← What it's all about (if your note is longer than
4   }                                        a line, you can specify a width to avoid
5                                            overflowing the page!)
6   int foo;
7 }
```

not great: not a keyword

This color is used for upper-case identifiers (e.g. constants)—also wrong!

The `pin-code-from` function works a bit differently from `pinit`'s `pinit-point-from`, in that its `pin` etc. parameters accept a pair of *numbers* instead of there being separate `pin-dx` and `pin-dy` numbers accepting *lengths*. The distances are specified in terms of the monospace font grid: 1 in x direction is equal to ~4.8pt, for example. The `pin` and `offset` arrays can further contain an alignment as a third parameter. For example, `top+left` would make the arrow start or end at that corner of the letter. More details can be found in the manual.

The pinning functionality is tuned for this template: changing the font or other parts of the raw block geometry will result in the pins not fitting anymore—beware!

Wrapping text around figures

This is not really a feature of this template, just a tutorial on using `meander`[5] for what I found useful in my documents.

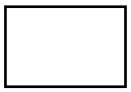

Figure 1: A rectangle. The figure caption wraps thanks to the fixed width.

The default mode of Meander is managing a whole page; I found myself usually wanting to manage small groups of paragraphs that contain a figure—like Figure 1. To do so, use `placement: box` when calling `meander.reflow()`. This will cause Meander to actually reserve the necessary space, so that subsequent regular layout only continues after Meander's content (this is not necessary when Meander manages a whole page anyway).

Another useful parameter is `placed(boundary: contour.margin(..), ..)`. The `boundary` defines how text should avoid the placed figure, and `contour.margin()` is a simple such boundary that adds a bit of space around your figure. Finally, I like to put my figures into fixed-width `block()`s. It means that the text may be slightly narrower than necessary to fit the figure, but it makes the layout more robust when writing long figure captions.

Wrapping text over pagebreaks

One final trick with `meander` is using multiple containers when the wrapping content overflows a page. This is an example of that: the paragraph starts on this page, but flows

[4] https://typst.app/universe/package/pinit
[5] https://typst.app/universe/package/meander

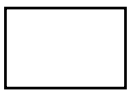

Figure 2: Another rectangle, at the to of the page.

down onto the next one. We also want Figure 2 to wrap around that paragraph, appearing at the top of the new page.

We can't just *not* make the first paragraph part of the `meander.reflow()` call, since then the figure wouldn't be at the top of the page, but we also can't have all content in a single Meander `container()`.

However, meander allows multiple containers with explicit pagebreaks in between, and the content will flow between these! It's not fully automatic—you have to specify the height of the first container, i.e. how much space is left on the page—but it can achieve this layout.

License

The `licenses` dictionary contains links to various creative commons licenses, displayed as the corresponding icon (powered by `ccicons`[6]): 🅭🅯🅭 or 🅭. If you want to specify a license, you would usually use it when calling the template, then it will be shown in the footer by default.

This document itself is CC-BY, but under normal circumstances (replacing the text with your own material) that license will not apply to you. The scaffolding alone (calling `set document()` etc.) is too trivial to entail copyright.

Customization

Some customization knobs are provided, but feel free to fork this template (MIT licensed) if you need more freedom.

Font

The template uses Not Sans by default. Changing the font is supported, but the code note feature may be impacted: the `pinit-code-from()` function configures line spacing that makes multiline notes line up with code lines. The used measurements would need to change, which is only supported via forking. Likewise, changing the raw font via `show raw: set text(..)` will mess up the monospace grid measurements that `pin-code-from()` is based on.

Header & footer

Both header and footer are divided into three equal-width parts; some of them have default values which you can find in the manual. The template further overrides some of these defaults, so that the outcome is this:

- Header: course/audience description; short version of the title; date-based version number
- Footer: copyright note including author, year and license; page number; institution

You are of course free to use any other header/footer content you like.

[6] https://typst.app/universe/package/ccicons