# Tab2Pdf Design Document

EECS 2311
V. Tzerpos
Wednesday, April 7, 2015

**Group 1**
Atto, Brody
Cirillo, Marco
Patel, Deep
Ragavendran, Varsha
Sitiugin, Glib
Sitkovets, Anton

# Overview

The program operates by being given a plain text guitar tab file as input from the user and converting it to a pretty-printed PDF. A Graphical User Interface is used to select the tab, as well as where to save the PDF. When the conversion process is complete, the final PDF is opened for the user to view.

# High-Level Design

As per project requirements, we use the iTextPDF library to convert a user's tab files to a formatted PDF version of the sheet music. The user of this application interacts only with our graphical user interface, which abstracts away the inner workings of the system. The result is a user-friendly application that is quick and easy to use.

The software is designed similarly to the way a programming language compiler converts code from human readable instructions (`int x = 3 + 2;`) into machine-readable instructions (`add eax, ebx`). There are two phases involved when converting from tab to PDF. The first phase consists of transforming the symbols of the tab into Java Objects that we can work with. The second phase takes these Objects and draws them into the PDF document.

> *Technical Note:* the first phase is not unlike converting code into an Abstract Syntax Tree (AST), a technique used in compilers. Failure to create an AST in compilers can be due to a syntax error (such as the code `int 3;` producing `error: not a statement` in Java).

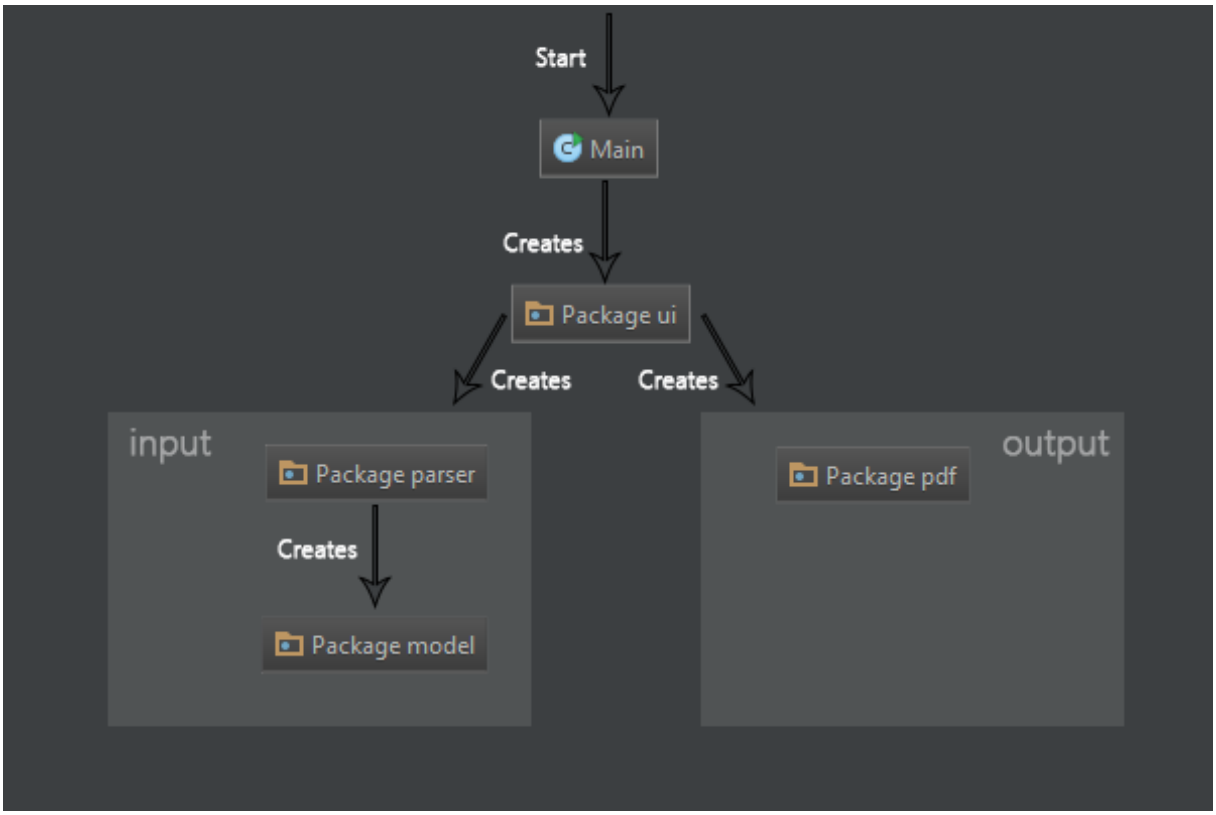
Objects that represent tab notation implement a common Interface, `ITabNotation`, which we use to guarantee us the Object can perform certain tasks. We use `IParser` Objects to create these `ITabNotation` objects out of the guitar tab. A `PDFCreator` uses these `ITabNotation` Object to create the final PDF.

> *Technical Note:* `ITabNotation` Objects are reflexive and symmetric. calling `toString()` on `ITabNotation` must yield the **exact** string that the `IParser` created it with.

This architecture lends itself to apply certain design patterns. Example of some of the design patterns used include:

- Iterator Pattern, to traverse collections of `ITabNotation` Objects
- Factory Pattern: our `IParser` Objects are factories that create `ITabNotation` Objects
- Visitor Pattern: our `PDFCreator` visits each `ITabNotation` Object to construct the PDF.
- Decorator Pattern: Some models decorate other models. An example of this is the `HammerOn` and `PullOff` classes, which decorate a `Note` with a hammer-on or pull-off.

# High-Level Class Diagram




The class diagram shows how objects in the system interact with each other. The program begins from the `Main.java` file, from which it constructs the UI. The UI handles user interaction, including handling input (parsing the tab and creating objects for each symbol), and output (exporting the tab to a PDF document).

# Maintenance Scenarios

The code is designed to be very maintainable, by being organized in to logical packages (see *Package Overview* below) that implement certain features. We tried to ensure high cohesion and low coupling when designing our packages and classes. A few typical maintenance scenarios will be described below.

## Adding a New Symbol

If a new tab symbol needs to be added, all a developer has to do is implement a new `ITabNotation` class to model the symbol, and a new `IParser` class to parse the symbol out of a tab. All that is needed to do to wire up the new parser is to add it to the `List<IParser>` in the `TabParser`. This list represents all parsers that the `TabParser`

may use when parsing a tab file.

> *Technical Note*: the new `IParser` will return the new `ITabNotation` model that represents the new symbol.

This functionality can be expanded in the future by exposing an API that allows the `TabParser` to add new parsers programmatically (ie: if someone wanted to use Tab2Pdf as a library in another project)

## Adding a New Tab to the GUI

To add a new tab to the GUI (beside the `Editor` and `Preview` tabs), a developer would have to first construct their new GUI tab as a `JPanel` object. Then, they can add their new `JPanel` as a tab by fetching the exposed `JTabbedPane` in `MainJFrame.getTabbedPane()` and calling `JTabbedPane.addTab()` on the object.

## Adding More Output Options

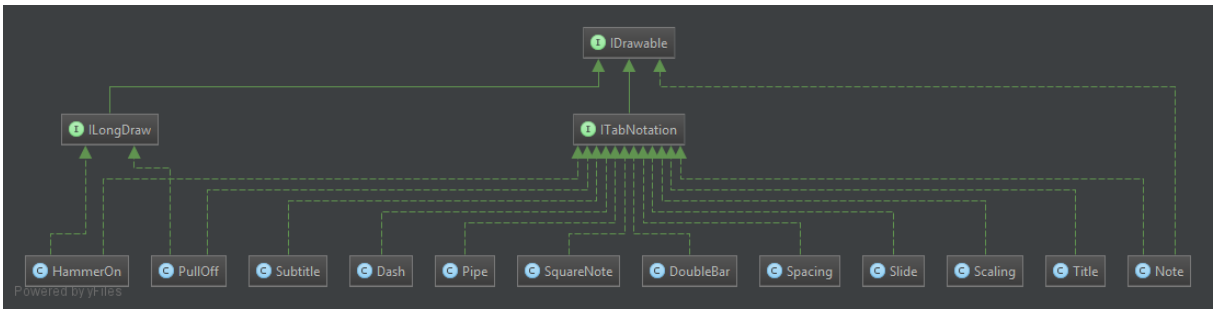
If a developer wishes to output something other than a PDF from Tab2PDF, they can do so by creating a Java class that can take the `Tab` object as an input. One of the benefits to our design is that we parse the text into the tab file into an object representation (similar to an *Abstract Syntax Tree* in compiler design). This allows us to transform the output into any format, since we only need to concern ourselves with converting from an object format, to the desired output format (as opposed to having a separate parser for each format we wish to output to). It is recommended that running this new Java class is done on a separate thread so as to not freeze up the GUI, so the output code should be wrapped in a `Runnable` object.

# Package Overview

## ca.yorku.cse2311.tab2pdf

The main package of this application. It consists of `Main`. This class is the entry point of the application. It is responsible for creating and showing the GUI.

# ca.yorku.cse2311.tab2pdf.model

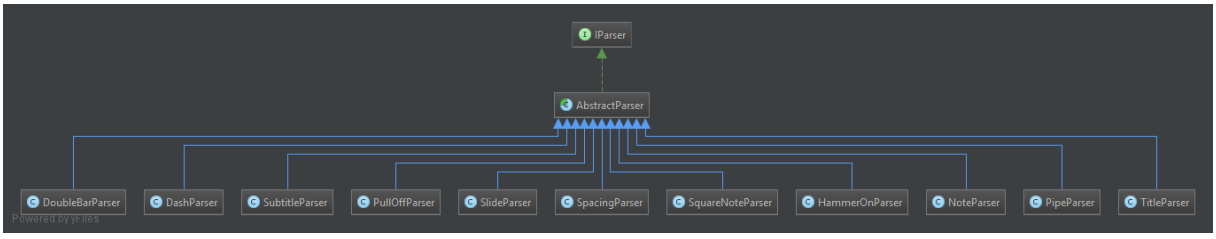



This package holds the business models of this application. All models implement `ITabNotation`, which describes a set of methods we expect all tab models to have, such as the ability to `draw()` to the PDF, the `size()` of the drawn symbol, and `toString()`, which must return the exact String that was used to create the object. This means models are reflexive and symmetric. "3" should parse into a `Note` with value 3, and calling `toString()` on this `Note` should yield "3".

> *Technical Note:* This is accomplished programmatically, rather than by storing the String that was used to create the object.

Special notes, such as `HammerOn`s and `PullOff`s (which use the *decorator pattern* around a `Note` object) implement `ILongDraw` so they are able to draw across musical bar boundaries.

# ca.yorku.cse2311.tab2pdf.parser




This package holds parsers capable of creating our models. All parsers implement `IParser`, which requires parsers to have a `getPattern()` method which returns the `Pattern` (Java's Regular Expression class) the parser will use internally to create the model object, a `canParse()` method, signifying the parser can indeed construct a model out of a specified String token, and `parse()`, which actually parses the token into the model.

`TabParser` holds all `IParser` objects, and is concerned with parsing the entirety of the

tab using these objects. If a new symbol is needed, one must write a model, implementing `ITabNotation`; a parser, implementing `IParser`; and finally they must add the parser to `TabParser`, making it aware of the new symbol.

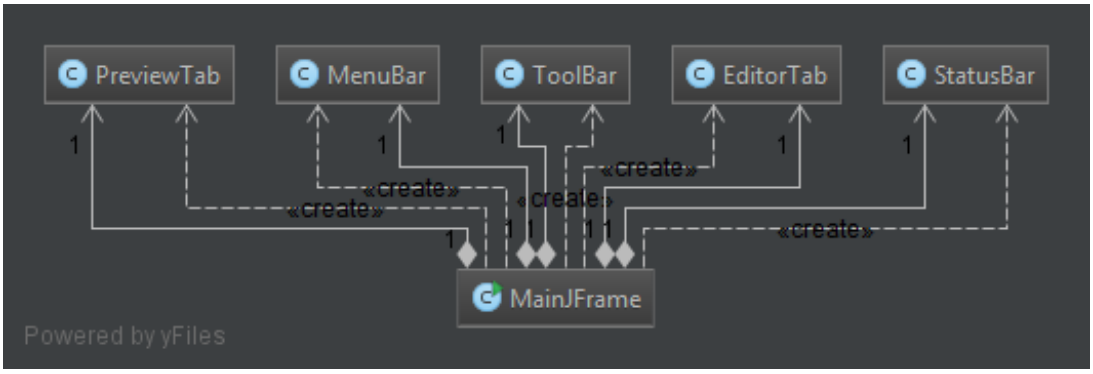
# ca.yorku.cse2311.tab2pdf.pdf

This package holds PDF-related utilities. `PdfHelper` is a wrapper around the iText PDF library, which allows drawing to a PDF document. `PdfCreator` is a `Runnable` that allows us to create the PDF on a separate thread (so as to not freeze up the GUI if the PDF takes a while to create).

# ca.yorku.cse2311.tab2pdf.ui

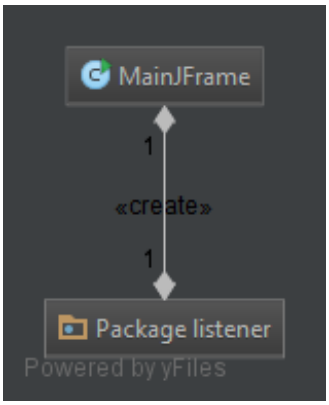
This package holds the GUI of the program, in one file `MainJFrame`. The `MainJFrame` is made up of individual components, such as the `Toolbar`, the `Menubar`, or the `EditorTab`. Each component has a reference back to its parent (`MainJFrame`) to allow the component to use its parents exposed methods. Additionally, the `MainJFrame` has that represent user actions, such as clicking the `Save Tab` button, which calls the `SaveFileListener`, which ultimately saves the tab file the user is working on. Listeners, like components, also take a reference back to its parent, to use its exposed methods.

## Components




This diagram shows the two-way relationship between components and the `MainJFrame`. The `MainJFrame` creates each component, which in turn has a reference the the `MainJFrame` that created it.

## Listeners




This diagram shows the two-way relationship between listeners and the `MainJFrame`. The `MainJFrame` creates each listener, which in turn has a reference the the `MainJFrame` that created it.

## ca.yorku.cse2311.tab2pdf.util

This package holds small utilities that contain helper methods to aid in development. `FileUtils` contains utilities to help with `File`s, such as reading a `File` and returning a `List<String>` consisting of each line in the `File`.