# COS10009 Introduction to Programming

## Design Report

### Overview

The program that was created in Ruby is called \_Routine. To put it simply, this program allows the user to set up schedules that are filled with tasks, which can then be displayed visually for the user to see.

The program aims to assist the user with time management. The user will be able to insert tasks that are to be completed in that day, which will then be displayed for the user to see. By allowing the user to view their current schedule in a visual way such as a pie chart, the user will be able to see how busy their schedule is. With this information, the user will be able to allocate appropriate timeslots for their tasks without overloading one part of their schedule, while leaving another part empty.

The program also aims to help reduce procrastination of the user. By allowing the user to set up their own schedule, the user will feel more responsible and will strive to follow the schedule. The visual representation of the tasks also helps with this, by letting the user know what tasks they should be doing in that specific time frame.

### Program Design

\_Routine makes use of classes. These classes contain their own attributes and methods which are used throughout the program. These classes work together to create a functioning task management system. Listed below are the classes used in the program and their attributes and methods:

* Task
  + title - Name of the task
  + desc - Extra info about the task
  + timeStart - Beginning time (24-hour system)
  + timeEnd - End time of this task (24-hour system)
  + isScheduled - Boolean. If timeStart and timeEnd is *nil*, *false*, else, *true*
  + timeDuration - Difference between timeEnd and timeStart in minutes
* Tasks
  + date - The date of creation of an instance of this class
  + tasks - An array of Task instances
  + generate\_tasks - Creates a new Task instance with a given title, desc, timeStart and timeEnd, and appends it to tasks.
* Template
  + name - Name of the template
  + tasks - An instance of Tasks
* Templates
  + routine - A hash associating the day of the week to a Template instance
  + templates - An array of Template instances
  + generate\_templates - Creates a new Template instance with a given name and tasks and append it to templates.

There are also classes created for each GUI “page” and element, such “Main”, “MenuBar’, “TemplatesList”, “RoutineList” and many more. These GUI elements contain their own frames, labels and buttons and is created for the ease of transition between pages. With this, the program is able to separate different bits of functionality into different “pages” for an easier usage of the program.

Next, the program also utilizes FXRuby to create a graphical user interface for the ease of navigation for the user. By using the FXHorizontalFrame and FXVerticalFrame that came with the FXRuby module, the GUI elements are grouped and organized appropriately. Functionality of the program are accessed through FXButton and FXMenuCommand instances, which run a block of code that is connected to it based on certain event triggers, such as SEL\_COMMAND. Text is also displayed to the user with use of FXLabel instances, to label certain parts of the GUI.

To display the user’s tasks in a visual way, FXRuby provides a drawing context in which lines and shapes can be drawn into. By using the drawArc function together with the timeStart, timeEnd and timeDuration of the Task instances, coloured arcs that show the beginning and end time of the tasks can be drawn onto the drawing context. By also drawing numbers around the circumference of the circle, this can simulate a clock and shows the user what tasks are set for what time.

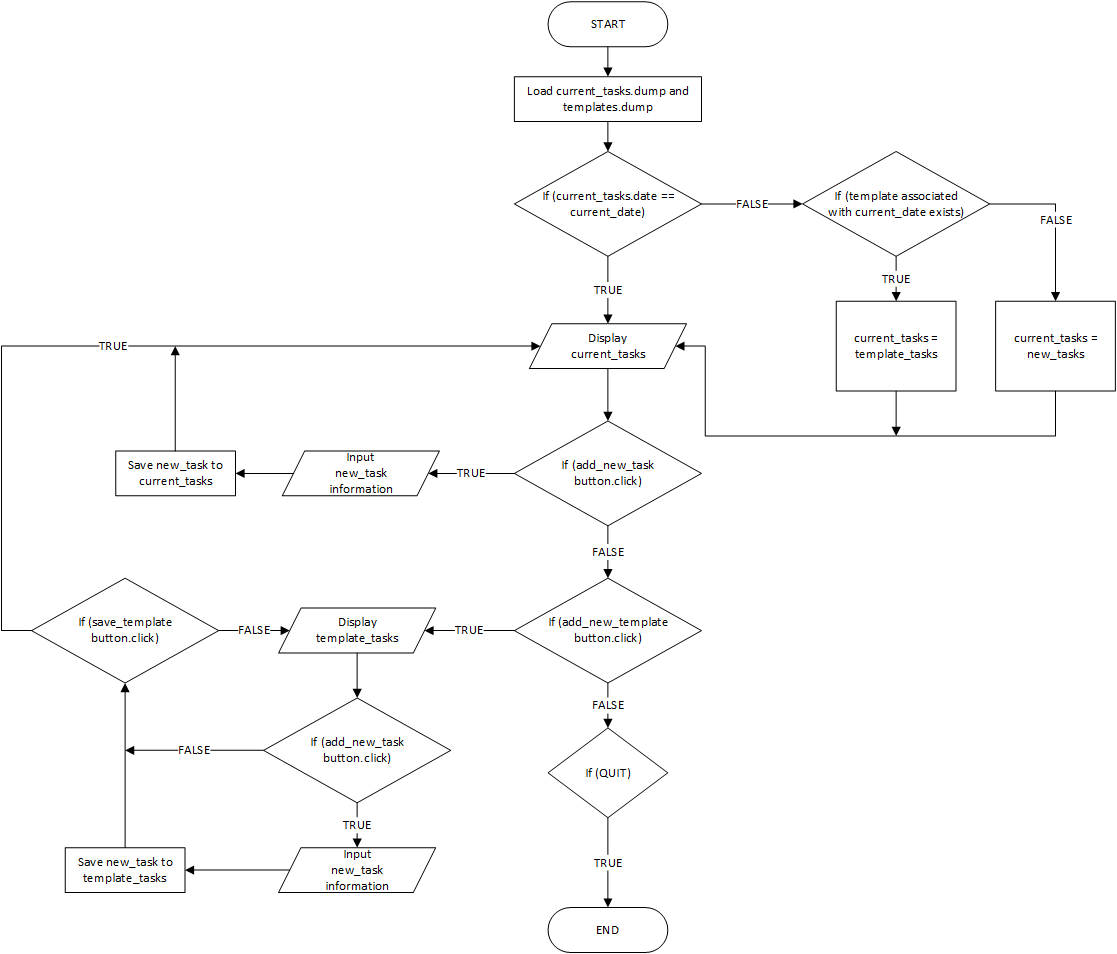
But with just this, overlapping will occur which will cause the arcs to cover each other. To prevent this, an algorithm to calculate the amount of times an arc overlaps with other arcs is created. Each task is given an overlap index which is calculated using the algorithm. How the algorithm works, is that the task is checked against other tasks with the same overlap index, and if there is an overlap, increase the index by 1 and check again. This runs recursively until the task does not overlap with anything on that index. When all the indexes are calculated, the arcs are then drawn on different “levels” based on the overlap index.

Throughout the day, the user will most likely close and reopen the program multiple times. This means that a system should be set up to save the tasks so that it may be loaded again in the future. This is done with the help of the Marshal module. The Marshal module contains various functions that read and writes data to and from “.dump” files. It does this by serializing the data and then writing to the text file. This allows the program to save instances of classes to text files to be reused in the future. When loading the file, the text within the text file is deserialized and then returned into a variable for usage in the program.

The program automatically resets the tasks the first time the program is run on a new day. This is done by checking the date attribute of the Tasks instance, with a date variable that is set each time the program is run. If it founds out that it is a different date, it will create a new Tasks instance to replace the saved one. This is done like so because the program is meant to help users schedule their tasks for the day, as they are most likely to have different tasks every day.

The program also features a template system, from which the user can load a set of tasks that they have created beforehand. Maybe the user has a set of tasks that are required to be done togeer. The template system also allows the user to set which day of the week uses which template. This allows the program to load the associated template at the beginning of the associated day. This means that if a user has a routine set of tasks each day, they can set it so that those tasks show up for those days without having to add them in manually. This allows the user to schedule tasks around their routine.

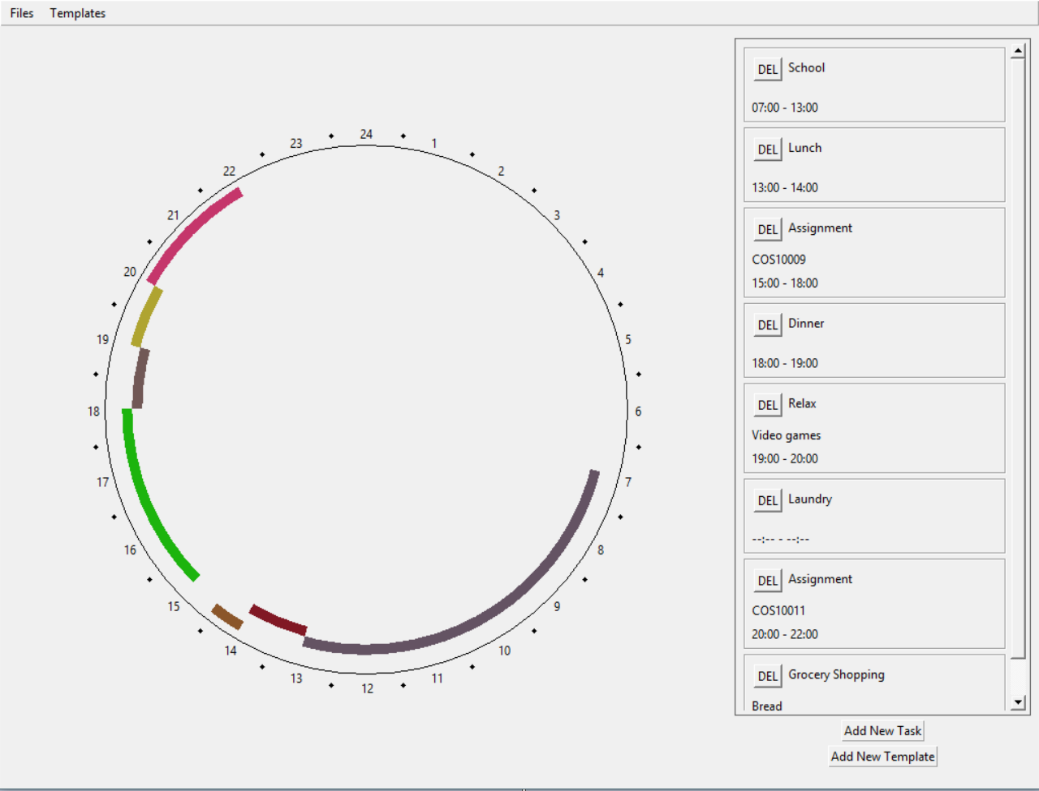
### Flowchart



### User Manual

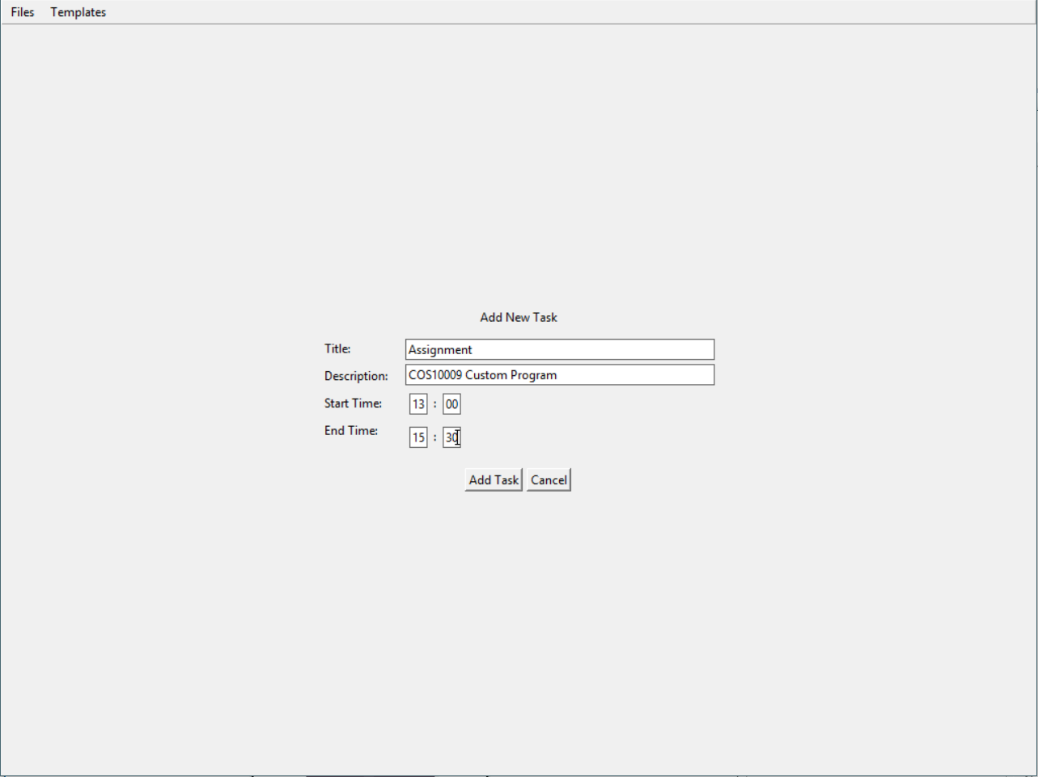
When the program is first run, new files will be created, which are called ‘current\_tasks.dump” and “templates.dump”. “current\_tasks.dump” stores the tasks for the current day, while “templates.dump” stores a list of templates that have been created together with the hash for associating days with templates. Deleting these files will cause the program to lose the data for the current tasks and the created templates, and new files will be created.

#### Main Task Display Menu



The current ongoing tasks are displayed in this menu, in the form of a donut chart on the left side and also a list on the right side. Within each block of tasks in the list is the title, description, start time and end time of each task. There is also a DEL button that is used to delete the task from the current list. On the left side is the donut chart which displays the time slots for each task, although “unscheduled” tasks are not displayed here, as they do not have a start time and an end time. Beneath the list of tasks are two buttons. The “Add New Task” button links to the Add Tasks Menu, while the “Add New Template” menu links to the Template Task Display Menu.

#### Add Tasks Menu



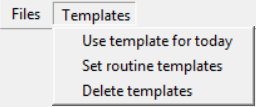
In this menu, there are text fields that receive input from the user. Only the “Title’ field is strictly required, while “Start Time” and “End Time” are only required if one of them contain values, else, they can be empty. There are no restrictions when it comes to entering a value for the “Title” and “Description” fields. For the “Start Time” and “End Time” fields, since the program uses the 24-hour system, only values from 0000 until 2400 are allowed. Beneath the fields are two buttons. The “Add Task” button takes the values from the fields and creates a new Task instance using those values, and then saves them into the list of tasks, then sends the user to the previous menu. The “Cancel” button just ignores the values and returns the user to the previous menu.

#### Template Task Display Menu



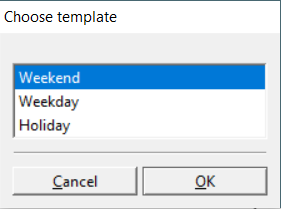
This menu looks similar to the Main Task Display Menu, but with a few extra features. Aside from the donut chart, the list of tasks and the “Add New Task” button, there is an extra “Template Name” field at the bottom, and some more buttons. The value within the “Template Name” field will be used during the creation of a Template instance. The “Add Template” button takes the value from the “Template Name” field and the list of tasks and creates a new Template instance, which is then saved into the “templates.dump”, after which the user is sent to the Main Task Display Menu. The “Cancel” button just sends the user to the Main Task Display Menu.

#### Menubar



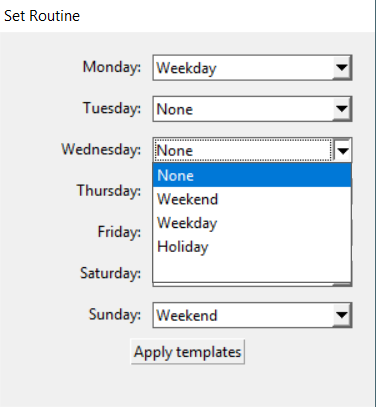
The menubar contains two tabs, the “Files” tab and the “Templates” tab. Within the “Files” tab is a menu command, “Clear current tasks”. This command clears the current list of tasks and resets the “current\_tasks.dump” with an empty Tasks instance. Under the “Templates” tab are three commands. The “Use template for today” command sends the user to the Choose Template Menu, where the result will be used to load that template into the current list of tasks and replace the “current\_tasks.dump” file. The “Set routine templates” sends the user to the Set Routine Template Menu, where additional inputs will be displayed. The “Delete templates” command sends the user to the Choose Template Menu, where the result will be used to delete the chosen template from the list of templates, and updates the “templates.dump” file.

#### Choose Template Menu



This menu displays all the available templates within “templates.dump”. The user will be able to choose one from the list, which will then be returned afterwards for further processing. There are buttons below the list of templates. The “OK” button accepts the current selection and returns it to the caller for further processing, after which it will close itself. The “Cancel” button just closes itself while ignoring the selection.

#### Set Routine Template Menu



This menu displays several list boxes, with the days of the week displayed in front of them. Within the list boxes are options for the templates that exist within “templates.dump”. Choosing a template will assign that template to that specific day of the week, so that the template will be loaded automatically on the start of that day. The user can also choose “None” which will cause the program to load an empty list of tasks on that day. The “Apply templates” button saves the selections and assigns the templates to the respective days within the “routine” hash, and then saved into “templates.dump”, after which it closes itself.

## Source Code

#### schedule.rb

FILE\_CURRENT\_TASKS = "current\_tasks.dump"

FILE\_TEMPLATES = "templates.dump"

# Stores the date of creation and a list of tasks

class Tasks

attr\_accessor :date, :tasks

def initialize

@date = Time.new

@tasks = []

end

def generate\_task(title, desc, timeStart, timeEnd)

taskNew = Task.new(title, desc, timeStart, timeEnd)

@tasks.push(taskNew)

end

end

# Stores the information about the task

class Task

attr\_accessor :title, :desc, :timeStart, :timeEnd, :timeDuration, :isScheduled

def initialize(title, desc, timeStart, timeEnd)

@title = title

@desc = desc

@timeStart = timeStart

@timeEnd = timeEnd

@isScheduled = false

if (@timeStart != nil && @timeEnd != nil)

@isScheduled = true

@timeDuration = (timeEnd - timeStart) / 60

end

end

end

# Stores a list of templates and templates associated with each day

class Templates

attr\_accessor :routine, :templates

def initialize

@routine = {"1" => nil, "2" => nil, "3" => nil, "4" => nil, "5" => nil, "6" => nil, "7" => nil}

@templates = []

end

def generate\_template(name, tasks)

template = Template.new(name, tasks)

@templates.push(template)

end

end

# Stores the name of the template and the list of tasks associated with this template

class Template

attr\_accessor :name, :tasks

def initialize(name, tasks)

@name = name

@tasks = tasks

end

end

# Serializes and stores the data in a text file

def dump\_file(file, content)

data = Marshal.dump(content)

File.open(file, "w") do |f|

f.write(data)

end

end

# Reads the file and deserializes the data from the text file

def load\_file(file)

File.open(file, "r") do |f|

data = f.read()

content = Marshal.load(data)

end

end

#### main.rb

require './schedule.rb'

require 'fox16'

include Fox

# Window in which the GUI is displayed in

class Main < FXMainWindow

attr\_accessor :dateToday, :objTasksCurrent, :templates

def initialize(app)

super(app, "\_Routine", :width => 1024, :height => 768)

@@font = FXFont.new(app, "segoe ui", 9) # Font used in the text

@@font.create

@dateToday = Time.new()

check\_existing\_tasks

check\_existing\_templates

check\_date

MenuBar.new(self)

TasksDisplayMain.new(self, @objTasksCurrent)

end

def self.font

@@font

end

def create

super

show(PLACEMENT\_SCREEN)

end

# Checks for an existing current\_tasks.dump, and creates a new one if it does not exist

def check\_existing\_tasks

begin

@objTasksCurrent = load\_file(FILE\_CURRENT\_TASKS)

rescue

@objTasksCurrent = Tasks.new

dump\_file(FILE\_CURRENT\_TASKS, @objTasksCurrent)

end

end

# Checks for an existing templates.dump, and creates a new one if it does not exist

def check\_existing\_templates

begin

@templates = load\_file(FILE\_TEMPLATES)

rescue

@templates = Templates.new

dump\_file(FILE\_TEMPLATES, @templates)

end

end

# Checks the date within current\_tasks.dump, and resets the file if it is different to today's date

def check\_date

dateFile = @objTasksCurrent.date

isSameDay = dateFile.year == @dateToday.year && dateFile.month == @dateToday.month && dateFile.day == @dateToday.day

if (!isSameDay)

if (@templates.routine[@dateToday.wday.to\_s] == nil) # Checks if there is a templte associated with the day

@objTasksCurrent = Tasks.new

else

@objTasksCurrent = @templates.routine[@dateToday.wday.to\_s].tasks

@objTasksCurrent.date = Time.new

end

dump\_file(FILE\_CURRENT\_TASKS, @objTasksCurrent)

end

end

# A method to update current\_tasks.dump during program runtime

def update\_tasks(newTasks)

@objTasksCurrent = newTasks

dump\_file(FILE\_CURRENT\_TASKS, @objTasksCurrent)

end

end

# Menubar that contains various commands

class MenuBar < FXMenuBar

attr\_accessor :parent

def initialize(parent)

super(parent, :opts => LAYOUT\_FILL\_X | FRAME\_RAISED)

@parent = parent

menuFilesPane = FXMenuPane.new(self)

menuFilesTitle = FXMenuTitle.new(self, "Files", :popupMenu => menuFilesPane)

menuFilesClear = FXMenuCommand.new(menuFilesPane, "Clear current tasks")

menuFilesClear.connect(SEL\_COMMAND) do

# Replaces current\_tasks.dump with a new Tasks instance

@parent.update\_tasks(Tasks.new)

gui\_recalc(@parent.objTasksCurrent)

end

menuTemplatesPane = FXMenuPane.new(self)

menuTemplatesTitle = FXMenuTitle.new(self, "Templates", :popupMenu => menuTemplatesPane)

listTemplates = TemplatesList.new(self, @parent.templates.templates)

listRoutine = RoutineList.new(self, @parent.templates)

menuTemplatesUse = FXMenuCommand.new(menuTemplatesPane, "Use template for today")

menuTemplatesUse.connect(SEL\_COMMAND) do

i = listTemplates.execute

# Replaces current\_tasks.dump with the tasks within the selected template

if (i >= 0 && i < @parent.templates.templates.length)

chosenTasks = @parent.templates.templates[i].tasks

chosenTasks.date = Time.new

@parent.update\_tasks(chosenTasks)

gui\_recalc(chosenTasks)

end

end

menuTemplatesRoutine = FXMenuCommand.new(menuTemplatesPane, "Set routine templates")

menuTemplatesRoutine.connect(SEL\_COMMAND) do

listRoutine.execute

end

menuTemplatesDelete = FXMenuCommand.new(menuTemplatesPane, "Delete templates")

menuTemplatesDelete.connect(SEL\_COMMAND) do

i = listTemplates.execute

# Deletes the selected task and updates the current\_tasks.dump

if (i >= 0 && i < @parent.templates.templates.length)

@parent.templates.templates.delete\_at(i)

dump\_file(FILE\_TEMPLATES, @parent.templates)

gui\_recalc(@parent.objTasksCurrent)

end

end

end

# Redraws the GUI with the updated list of tasks

def gui\_recalc(objTasks)

@parent.children.each do |child|

@parent.removeChild(child)

end

MenuBar.new(@parent).create

TasksDisplayMain.new(@parent, objTasks).create

@parent.recalc

end

end

# A seperate window for choosing a template, which will be returned to the caller

class TemplatesList < FXChoiceBox

def initialize(parent, arrTemplates)

templateNames = []

for i in 0..arrTemplates.length-1

templateNames << arrTemplates[i].name

end

super(parent, "Choose template", "", nil, templateNames)

end

end

# A window containing listboxes for choosing templates for specific days

class RoutineList < FXDialogBox

def initialize(parent, objTemplates)

super(parent, "Set Routine", :width => 300, :height => 300)

@parent = parent

weekdays = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"]

templateNames = ["None"]

objTemplates.templates.each do |template|

templateNames << template.name

end

hfrInputZone = FXVerticalFrame.new(self, :opts => LAYOUT\_FILL)

listboxes = []

# Creating the listboxes

weekdays.each do |day|

vfrRow = FXHorizontalFrame.new(hfrInputZone, :opts => LAYOUT\_FILL\_X)

FXLabel.new(vfrRow, day + ": ", :opts => LAYOUT\_FIX\_WIDTH | JUSTIFY\_RIGHT, :width => 100)

listbox = FXListBox.new(vfrRow, :opts => LAYOUT\_FILL\_X | FRAME\_LINE)

listbox.numVisible = 5

listbox.fillItems(templateNames)

listboxes << listbox

end

# Assigns the template to the routine hash in Templates in the templates.dump

btnApply = FXButton.new(hfrInputZone, "Apply templates", :opts => LAYOUT\_CENTER\_X | FRAME\_LINE)

btnApply.connect(SEL\_COMMAND) do |sender, selector, data|

for i in 0..listboxes.length-1

if (listboxes[i].currentItem > 0)

templateIndex = listboxes[i].currentItem - 1

objTemplates.routine[(i+1).to\_s] = objTemplates.templates[templateIndex]

else

objTemplates.routine[(i+1).to\_s] = nil

end

end

dump\_file(FILE\_TEMPLATES, objTemplates)

puts "RUNNING"

getApp().stopModal(self)

self.hide

end

end

end

# Displays the tasks in donut form and in a list

class TasksDisplay < FXHorizontalFrame

def initialize(parent, objTasks)

super(parent, :opts => LAYOUT\_FILL)

@parent = parent

@objTasks = objTasks

# Large vertical frame pies

@vfrScheduled = FXVerticalFrame.new(self, :opts => LAYOUT\_FILL)

TaskPie.new(@vfrScheduled, @objTasks.tasks)

# Skinny vertical frame for listing tasks and buttons

@vfrList = FXVerticalFrame.new(self, :opts => LAYOUT\_FIX\_HEIGHT | LAYOUT\_FIX\_WIDTH, :width => 300, :height => @parent.height - 50)

hfrScrollWindowBorder = FXHorizontalFrame.new(@vfrList, :opts => LAYOUT\_FILL | FRAME\_LINE)

scrWindow = FXScrollWindow.new(hfrScrollWindowBorder, :opts => LAYOUT\_FILL)

vfrTasks = FXVerticalFrame.new(scrWindow, :opts => LAYOUT\_FILL)

@objTasks.tasks.each do |task|

TaskBlock.new(vfrTasks, task, self)

end

end

end

# Main task display menu, displaying tasks from the current\_tasks.dump

class TasksDisplayMain < TasksDisplay

def initialize(parent, objTasks)

super(parent, objTasks)

@parent = parent

@objTasks = objTasks

btnAddTask = FXButton.new(@vfrList, "Add New Task", :opts => LAYOUT\_CENTER\_X | FRAME\_RAISED)

btnAddTask.connect(SEL\_COMMAND) do

@parent.removeChild(self)

TaskCreateMenu.new(@parent, @objTasks, TasksDisplayMain, true).create

@parent.recalc

end

btnAddTemplate = FXButton.new(@vfrList, "Add New Template", :opts => LAYOUT\_CENTER\_X | FRAME\_RAISED)

btnAddTemplate.connect(SEL\_COMMAND) do

@parent.children.each do |child|

@parent.removeChild(child)

end

newTemplateTasks = Tasks.new()

MenuBar.new(@parent).create

TasksDisplayTemplate.new(@parent, newTemplateTasks).create

@parent.recalc

end

end

end

# Display menu for new template creation

class TasksDisplayTemplate < TasksDisplay

def initialize(parent, objTasks)

super(parent, objTasks)

@parent = parent

@objTasks = objTasks

btnAddTask = FXButton.new(@vfrList, "Add New Task", :opts => LAYOUT\_CENTER\_X | FRAME\_RAISED)

btnAddTask.connect(SEL\_COMMAND) do

@parent.removeChild(self)

TaskCreateMenu.new(@parent, @objTasks, TasksDisplayTemplate, false).create

@parent.recalc

end

lbTemplateName = FXLabel.new(@vfrScheduled, "Template Name:", :opts => LAYOUT\_CENTER\_X)

@inTemplateName = FXTextField.new(@vfrScheduled, 25, :opts => LAYOUT\_CENTER\_X | FRAME\_LINE)

btnAddTemplate = FXButton.new(@vfrScheduled, "Add Template", :opts => LAYOUT\_CENTER\_X | FRAME\_LINE)

btnAddTemplate.connect(SEL\_COMMAND) do

if (@inTemplateName.text != "")

@parent.templates.generate\_template(@inTemplateName.text, @objTasks)

dump\_file(FILE\_TEMPLATES, @parent.templates)

to\_TasksDisplayMain

end

end

btnCancel = FXButton.new(@vfrScheduled, "Cancel", :opts => LAYOUT\_CENTER\_X | FRAME\_LINE)

btnCancel.connect(SEL\_COMMAND) do

to\_TasksDisplayMain

end

end

# Return to the main task display menu

def to\_TasksDisplayMain

@parent.children.each do |child|

@parent.removeChild(child)

end

current\_tasks = load\_file(FILE\_CURRENT\_TASKS)

MenuBar.new(@parent).create

TasksDisplayMain.new(@parent, current\_tasks).create

@parent.recalc

end

end

# Formatting task information to be displayed in the display menus

class TaskBlock < FXVerticalFrame

def initialize(parent, task, owner)

super(parent, :opts => LAYOUT\_FILL\_X | FRAME\_THICK)

@task = task

@main = owner.parent

hfrTitle = FXHorizontalFrame.new(self, :opts => LAYOUT\_FILL\_X)

btnDelete = FXButton.new(hfrTitle, "DEL")

btnDelete.connect(SEL\_COMMAND) do

tasksCurrent = @main.objTasksCurrent

tasksCurrent.tasks.delete(@task)

@main.update\_tasks(tasksCurrent)

@main.children.each do |child|

@main.removeChild(child)

end

MenuBar.new(@main).create

TasksDisplayMain.new(@main, tasksCurrent).create

@main.recalc

end

lbTitle = FXLabel.new(hfrTitle, @task.title)

lbDesc = FXLabel.new(self, @task.desc)

if (@task.isScheduled)

timeStart = format\_time(@task.timeStart)

timeEnd = format\_time(@task.timeEnd)

timeRange = timeStart + " - " + timeEnd

else

timeRange = "--:-- - --:--"

end

lbTime = FXLabel.new(self, timeRange)

end

# Formats the time into a readable format

def format\_time(time)

timeString = time.hour.to\_s + ":" + time.min.to\_s

timeHour = (time.hour.to\_s.length == 1) ? "0" + time.hour.to\_s : time.hour.to\_s

timeMinute = (time.min.to\_s.length == 1) ? "0" + time.min.to\_s : time.min.to\_s

timeStr = timeHour + ":" + timeMinute

return timeStr

end

end

# Donut chart for displaying tasks

class TaskPie < FXCanvas

def initialize(parent, tasks)

super(parent, :opts => LAYOUT\_FILL)

@parent = parent

@tasks = []

tasks.each do |task|

if (task.isScheduled)

@tasks.push(task)

end

end

@listIndexOverlap = calc\_overlap

@dia = 500

self.connect(SEL\_PAINT) do

dc = FXDCWindow.new(self)

dc.foreground = @parent.backColor

dc.fillRectangle(0, 0, self.width, self.height)

dc.foreground = FXRGB(0, 0, 0)

diaRing = @dia + 15

dc.drawArc((self.width / 2) - (diaRing / 2), (self.height / 2) - (diaRing / 2), diaRing, diaRing, 0, 23040)

# Draws the arc based on the overlap index of the task

if @tasks.length > 0

for targetIndex in 0..(@listIndexOverlap).max

for i in 0..@tasks.length - 1

if (@listIndexOverlap[i] == targetIndex)

task = @tasks[i]

draw\_arc(dc, task.timeStart, task.timeEnd, @listIndexOverlap[i])

end

end

end

end

dc.font = Main.font

draw\_hours(dc)

dc.end

end

end

# Loops through all the tasks to calculate the overlap index of the tasks

def calc\_overlap

@listIndexOverlap = Array.new(@tasks.length, 0)

for iTarget in 0..@tasks.length-1

targetTask = @tasks[iTarget]

if (targetTask.isScheduled)

check\_overlap(iTarget)

end

end

return @listIndexOverlap

end

# Logic to calculate the "overlap-ness" of the tasks

def check\_overlap(iTarget)

targetTask = @tasks[iTarget]

for iRef in 0..@tasks.length-1

refTask = @tasks[iRef]

if (refTask != targetTask && refTask.isScheduled)

# If the task overlaps with other tasks on the same index, index + 1 and then run the function again until it does not overlap

if (((targetTask.timeStart >= refTask.timeStart && targetTask.timeStart <= refTask.timeEnd) || (targetTask.timeEnd >= refTask.timeStart && targetTask.timeEnd <= refTask.timeEnd) || (targetTask.timeStart <= refTask.timeStart && targetTask.timeEnd >= refTask.timeEnd)) && (@listIndexOverlap[iTarget] == @listIndexOverlap[iRef]))

@listIndexOverlap[iTarget] += 1

check\_overlap(iTarget)

end

end

end

end

# Draws the arc based on the start time, end time and overlap index of the task

def draw\_arc(dc, timeStart, timeEnd, indexOverlap)

weight = 10

diameter = @dia - (weight \* indexOverlap \* 2)

x = (self.width / 2) - (diameter / 2)

y = (self.height / 2) - (diameter / 2)

start = ((timeStart.hour \* 60) + timeStart.min) \* 16

extent = ((timeEnd - timeStart) / 60) \* 16

dc.foreground = FXRGB(rand(210), rand(210), rand(210))

dc.fillArc(x, y, diameter, diameter, 5760 - start, -extent)

dc.foreground = @parent.backColor

dc.fillArc(x + weight, y + weight, diameter - (weight \* 2), diameter - (weight \* 2), 0, 23040)

end

# Draws the clock hour

def draw\_hours(dc)

centerX = self.width / 2

centerY = self.height / 2

hour = 1

angle = -75

angleEnd = angle + 360

radius = @dia / 2 + 20

dc.foreground = FXRGB(0, 0, 0)

# Draws the hour numbers

while (angle < angleEnd)

radians = angle \* Math::PI / 180

offsetX = Math.cos(radians) \* radius

offsetY = Math.sin(radians) \* radius

dc.drawText(centerX + offsetX - 4, centerY + offsetY + (dc.font.fontHeight / 2), hour.to\_s)

hour += 1

angle += 15

end

# Draws the 30 minute mark

angle = -82.5

angleEnd = angle + 360

while (angle < angleEnd)

radians = angle \* Math::PI / 180

offsetX = Math.cos(radians) \* radius

offsetY = Math.sin(radians) \* radius

dc.fillArc(centerX + offsetX, centerY + offsetY, 5, 5, 0, 23040)

angle += 15

end

end

end

# Displays text fields to enter information about the new task

class TaskCreateMenu < FXVerticalFrame

def initialize(parent, objTasks, tasksDisplay, isSave)

super(parent, :opts => LAYOUT\_CENTER\_X | LAYOUT\_CENTER\_Y)

@parent = parent

@objTasks = objTasks

@tasksDisplay = tasksDisplay

@isSave = isSave

FXLabel.new(self, "Add New Task", :opts => LAYOUT\_CENTER\_X | LAYOUT\_TOP)

hfrInputZone = FXHorizontalFrame.new(self, :opts => LAYOUT\_FILL)

vfrLabels = FXVerticalFrame.new(hfrInputZone, :vSpacing => 8)

FXLabel.new(vfrLabels, "Title: ")

FXLabel.new(vfrLabels, "Description: ")

FXLabel.new(vfrLabels, "Start Time: ")

FXLabel.new(vfrLabels, "End Time: ")

vfrInputs = FXVerticalFrame.new(hfrInputZone)

@inTaskTitle = FXTextField.new(vfrInputs, 50, :opts => FRAME\_LINE)

@inTaskDesc = FXTextField.new(vfrInputs, 50, :opts => FRAME\_LINE)

hfrTaskStart = FXHorizontalFrame.new(vfrInputs)

@inTaskStartH = FXTextField.new(hfrTaskStart, 2, :opts => TEXTFIELD\_INTEGER | TEXTFIELD\_LIMITED | FRAME\_LINE)

FXLabel.new(hfrTaskStart, ":")

@inTaskStartM = FXTextField.new(hfrTaskStart, 2, :opts => TEXTFIELD\_INTEGER | TEXTFIELD\_LIMITED | FRAME\_LINE)

hfrTaskEnd = FXHorizontalFrame.new(vfrInputs)

@inTaskEndH = FXTextField.new(hfrTaskEnd, 2, :opts => TEXTFIELD\_INTEGER | TEXTFIELD\_LIMITED | FRAME\_LINE)

FXLabel.new(hfrTaskEnd, ":")

@inTaskEndM = FXTextField.new(hfrTaskEnd, 2, :opts => TEXTFIELD\_INTEGER | TEXTFIELD\_LIMITED | FRAME\_LINE)

hfrButtons = FXHorizontalFrame.new(self, :opts => LAYOUT\_CENTER\_X)

btAdd = FXButton.new(hfrButtons, "Add Task")

btAdd.connect(SEL\_COMMAND) do

valid = check\_input

# Formats the time inputs and creates a new task then saves it

if valid

taskStart = taskEnd = nil

isFilled = @inTaskStartH.text != "" && @inTaskStartM.text != "" && @inTaskEndH.text != "" && @inTaskEndM.text != ""

if (isFilled)

taskStart = generate\_time(@inTaskStartH.text.to\_i, @inTaskStartM.text.to\_i)

taskEnd = generate\_time(@inTaskEndH.text.to\_i, @inTaskEndM.text.to\_i)

end

@objTasks.generate\_task(@inTaskTitle.text, @inTaskDesc.text, taskStart, taskEnd)

if (@isSave)

dump\_file(FILE\_CURRENT\_TASKS, @objTasks)

end

removeChild(self)

@tasksDisplay.new(@parent, @objTasks).create

@parent.recalc

end

end

btCancel = FXButton.new(hfrButtons, "Cancel")

btCancel.connect(SEL\_COMMAND) do

removeChild(self)

@tasksDisplay.new(@parent, @objTasks).create

@parent.recalc

end

end

# Validation of the text fields

def check\_input

# Checks for an empty task title

if (@inTaskTitle.text == "")

return false

end

# Checks if task time is given, and if given, checks all inputs are given

isEmpty = @inTaskStartH.text == "" && @inTaskStartM.text == "" && @inTaskEndH.text == "" && @inTaskEndM.text == ""

isFilled = @inTaskStartH.text != "" && @inTaskStartM.text != "" && @inTaskEndH.text != "" && @inTaskEndM.text != ""

if !(isEmpty || isFilled)

return false

end

# Checks if a valid number is given for the hours and minutes

invalidHours = @inTaskStartH.text.to\_i < 0 || @inTaskStartH.text.to\_i > 24 || @inTaskEndH.text.to\_i < 0 || @inTaskEndH.text.to\_i > 24

invalidMinutes = @inTaskStartM.text.to\_i < 0 || @inTaskStartM.text.to\_i > 59 || @inTaskEndM.text.to\_i < 0 || @inTaskEndM.text.to\_i > 59

if (isFilled)

if (invalidHours || invalidMinutes)

return false

end

end

taskStart = generate\_time(@inTaskStartH.text.to\_i, @inTaskStartM.text.to\_i)

taskEnd = generate\_time(@inTaskEndH.text.to\_i, @inTaskEndM.text.to\_i)

if (taskStart > taskEnd)

return false

end

return true

end

def generate\_time(h, m)

dateToday = @parent.dateToday

return Time.new(dateToday.year, dateToday.month, dateToday.day, h, m)

end

end

if \_\_FILE\_\_ == $0

FXApp.new do |app|

Main.new(app)

app.create

app.run

end

end