Look and Feel

Qt gives you the option to provide custom styles for most of the built-in widgets using a syntax that's almost identical to HTML cascading stylesheets (*CSS*). A stylesheet is a list of *style rules*, each *style rule* is associated with one or more types of Qt widget. Here's one example:

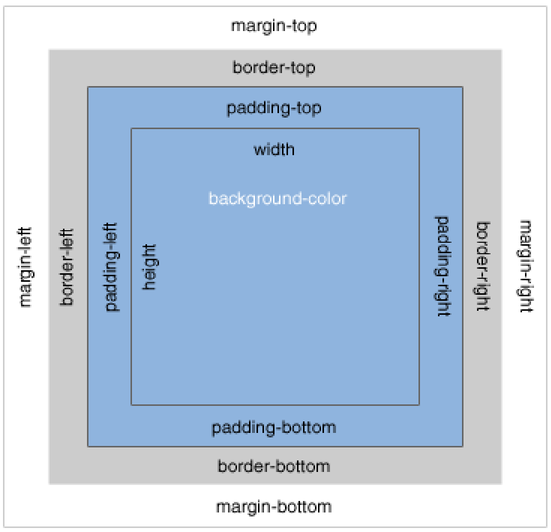
```
/* This is a comment */
QPushButton {
      min-width: 100px;
      color: #ff0000;
      font-size: 18px;
}
```

This style rule associates three *property values* with QPushButton, the minimum width of the button, the font color and the font size of the text inside the button. Usually a stylesheet has a long list of such style rules. *Side note:* if this looks like C code to you, that's because the syntax of CSS is borrowed heavily from C.

A full discussion of style sheets is beyond this little worksheet.
I strongly suggest studying the stylesheet syntax reference on the Qt website.

http://doc.qt.io/qt-5/stylesheet-syntax.html

Not all widgets obey the same properties. Some widgets can't be styled at all. And some have unique properties that only apply to them. Despite the variety in styling properties, many widgets *do* obey the CSS Box Model, which gives a standard set of properties for specifying size, padding, margins and borders.



Properties of Widgets that obey the CSS Box model

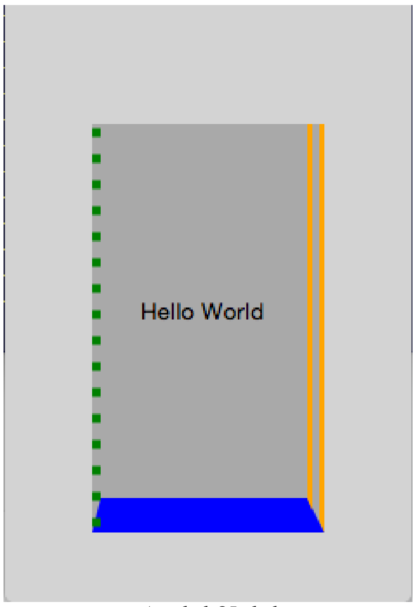
`QLabel` is one widget that follows the box model. Here's an example that sets a variety of the box model properties. Most of these properties give you shortcuts to specify a number of them at once - for example, `border-width` `border-style` and `border-color` can all be wrapped up into a single `border` property to style all borders with one statement.

```
QLabel {
       background-color: darkgrey;
       border-left: 5px dotted green; /* shortcut sets width style color all at once */
       border-right: 10px double orange;
       border-bottom: 20px solid blue;
       padding: 100px 20px; /* sets the top-down and left-right padding all at once */
       margin: 50px 30px 20px 30px; /* sets margin top right down left all at once */
     }
```

If you've ever done CSS on the web, you know that styling can be a little fidgety. Sometimes the results aren't exactly as you would expect, and often the results vary depending on your operating system.

Qt CSS also has a similar bit of occasional weirdness, but I find it far more reliable than using CSS in the browser. Just using the basic style properties of Qt's built-in widgets you can get quite far in creating a specific look and feel for your apps. Add to that the possibility of using images as backgrounds, border fills, and icons, and you can make your app look almost any way you want without ever having to build a custom widget class.

By default Qt will use stylesheets that try to match the look and feel of your operating system.



A styled QLabel

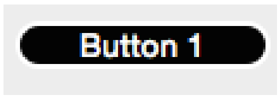
I strongly suggest you study the Qt stylesheet reference page.
This reference includes a full list of the widgets that can be styled and their properties.

http://doc.qt.io/qt-5/stylesheet-reference.html

StyleSheets + PyQt

`setStyleSheet()` is the method you use to set the style of a specific widget in your code. For example, if you want one of your QPushButtons to have a black background, white text, a 2 pixel white border, and rounded corners with a radius of 10 pixels - you could write some code like this.

```
button = QPushButton("Button 1")
button.setStyleSheet("""
    QPushButton {
        background-color: black;
        color: white;
        border: 2px solid white;
        border-radius: 10px;
    }
""")
```

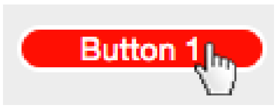


Styled Button

As soon as you start adding custom styles to a widget all the default operating system styles are thrown out the window. After setting the stylesheet above your button won't change its appearance when the user clicks it anymore. You have to implement these styles yourself.

To make a widget change appearance based on user interactions CSS gives you multiple styleable states called *pseudo-states*. For example QPushButton has the pseudo-states *hover* and *pressed*. You can specify the style of a pseudo-state using a colon (:) like so...

```
QPushButton:pressed {
        background-color: #ff0000;
}
```



The same button with the pressed pseudo-style set to have a red background

Widget Hierarchies

When you set a stylesheet directly on one of your widgets, those styles will apply to all widgets *inside* of that widget. Usually this is what you want. If you have a bunch of buttons and other UI objects inside a container widget, you probably want them all to look the same for the sake of visual coherence.

Sometimes, however, you want to be able to style one specific widget without affecting the style of any others. In Qt you do this by giving the widget you want to style a unique *object name*.

```
button = QPushButton("Pushme!")
button.setObjectName("specialButton")
```

Then in your stylesheets you can directly style that widget by using the id selector (#).

```
containerWidget.setStyleSheet("""
        QPushButton#specialButton {
                background-color: red;
        }
""")
```

Here's an example program that pulls together a number of these ideas.

BasicStyles.py

```
from PyQt5.QtWidgets import *
from random import randint, choice
import sys

```

```python
class MyWindow(QWidget):
    def __init__(self):
        super(MyWindow, self).__init__()
        self.initGUI()

    def initGUI(self):
        self.setObjectName("main")
        self.setStyleSheet("""
        QWidget#main {
            background-color: qlineargradient(x1:0, y2:0, x2:0, y2:1,
                                stop: 0 #bca6c2, stop: 1 #c9ab68);
            padding: 20px;
        }
        """)

        button1 = QPushButton("Button 1")
        button1.setStyleSheet("""
            QPushButton {
                background-color: black;
                color: white;
                border: 2px solid white;
                height: 50px;
                min-width: 100px;
                max-width: 100px;
                border-radius: 20px;
            }
            QPushButton:pressed {
                background-color: red;
            }
            QPushButton:hover {
                border: 5px solid #33AAFF;
            }
        """)

        button1.clicked.connect(self.buttonClick)
        self.button2 = QPushButton("Button 2")

        layout = QHBoxLayout()
        layout.addWidget(button1)
        layout.addWidget(self.button2)
        self.setLayout(layout)
        self.setWindowTitle("Some Styled Widgets")
        self.show()

    def buttonClick(self):
        newstyle = """
            QPushButton {
                background-color: white;
                font-size: %dpx;
                padding: %dpx;
                margin: %dpx;
                border: %dpx solid grey;
                font-family: %s;
            }
        """ % (randint(6, 48),
                randint(0, 20),
                randint(0, 20),
                randint(0, 20),
                choice(["Times New Roman","Georgia", "Monaco"]))

        self.button2.setStyleSheet(newstyle)


if __name__ == "__main__":
    app = QApplication(sys.argv)
    win = MyWindow()
    sys.exit(app.exec_())
```

A few things to note in this program...

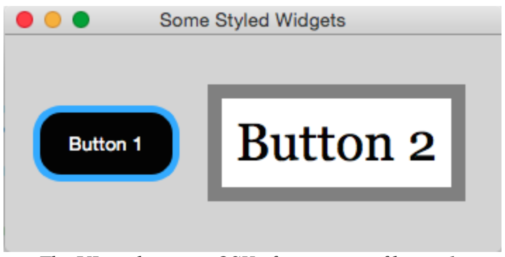
Line 11 gives the main widget a unique id "main" so it can be styled without affecting its child widgets.

Line 12-18 sets the stylesheet on the main widget, using the id specifier (#) to give the name of the widget this style should apply to. The background color is set to a gradient, using Qt's special `qlineargradient(x1, y1, x2, y2, stop, stop, stop ...)` function inside the stylesheet. This function takes a set of (x,y) coordinates from 0-1 to indicate the direction of the gradient, followed by a series of color stops and their positions along a line from 0-1.

For more info on specifying gradients see the documentation: http://doc.qt.io/qt-5/qlineargradient.html

Line 20-36 sets the stylesheet on button1, including styles for hover and pressed.

Line 48-61 generates a new randomized stylesheet for `button2`. Here I'm using Python's string format operator (%) to fill randomly generated values into the placeholders inside the string (`%d` and `%s`). For a review and tutorial of string formatting syntax in Python see: https://pyformat.info/



The UI as shown on OSX after a press of button1. Here button1 is shown in the hover state

Application-Wide StyleSheets

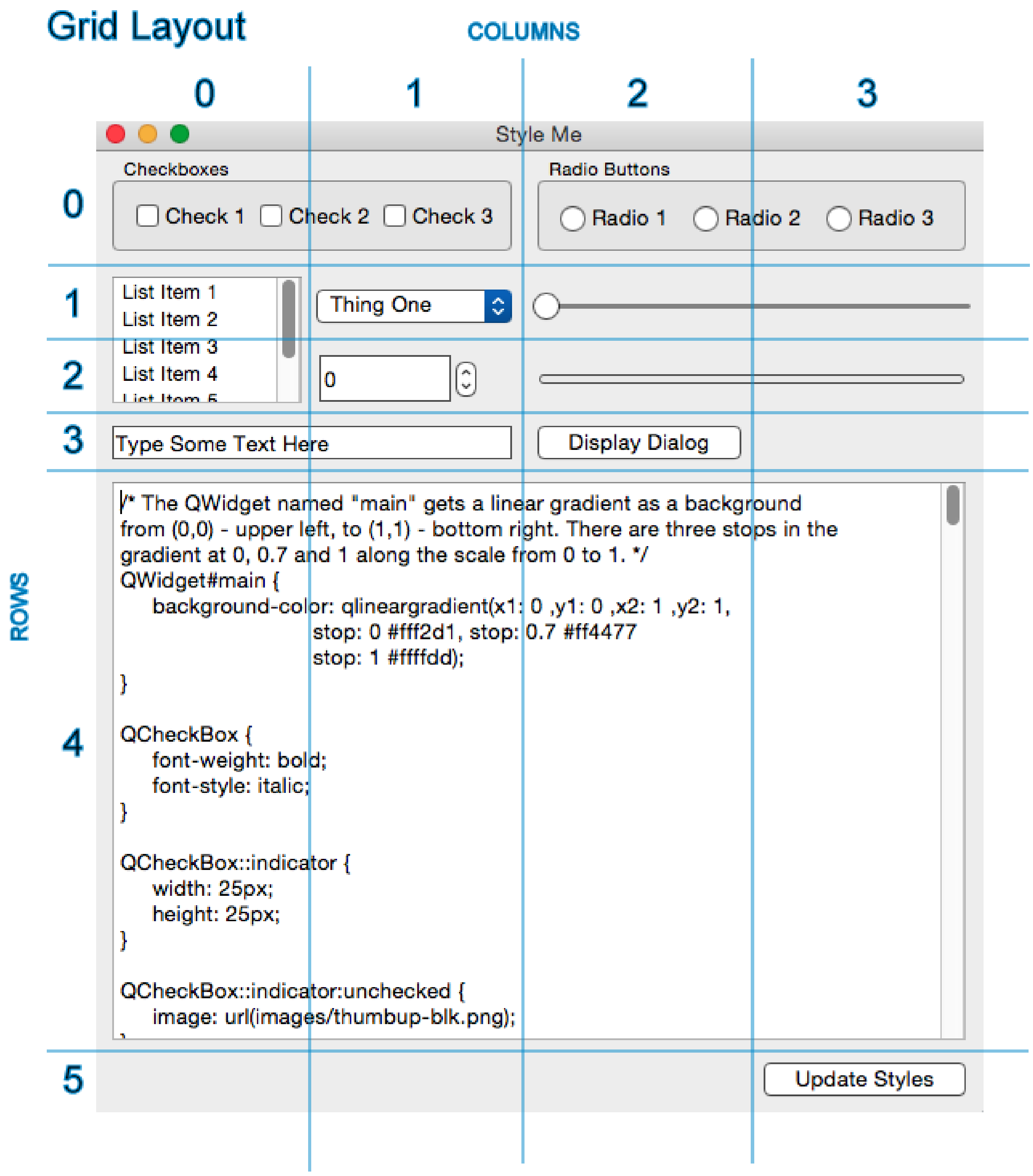
To apply a stylesheet to all the widgets in your interface you need to set it on the QApplication instance at the very top of the heirarchy of your UI. This is the very same object you create in the last lines of every PyQt program. Because this is such an important object, Qt gives you a global reference to it. The reference is stored in the variable qApp in the module PyQt5.QtWidgets.

Here's an application illustrating such a global stylesheet. The interface gives you a set of common widgets in a grid layout, and a large text editing box where you can edit the global stylesheet. The "Update Styles" button lets you apply your changes. You can find all the .css and image files for this program on the class github repository in the "Look and Feel" folder.

Note: if all your stylesheets break after clicking update, it's probably because you have a syntax error somewhere!

`StyleMe.py`

```python
from PyQt5.QtWidgets import * # qApp is in here!
from PyQt5.QtCore import Qt
import sys

class MyWindow(QMainWindow):
    def __init__(self):
        super(MyWindow, self).__init__()
        self.initGUI()

    def initGUI(self):
        self.dropdown = QComboBox()
        self.dropdown.addItem("Thing One")
        self.dropdown.addItem("Thing Two")
        self.dropdown.addItem("Thing Three")
        spinbox = QSpinBox()
        spinbox.setMaximumSize(100, 60)

        radios_layout = QHBoxLayout()
        radios_layout.addWidget(QRadioButton("Radio 1"))
        radios_layout.addWidget(QRadioButton("Radio 2"))
        radios_layout.addWidget(QRadioButton("Radio 3"))
        radiobuttons = QGroupBox("Radio Buttons")
        radiobuttons.setLayout(radios_layout)
        radiobuttons.setMaximumSize(300, 70)

        checkbox_layout = QHBoxLayout()
        checkbox_layout.addWidget(QCheckBox("Check 1"))
        checkbox_layout.addWidget(QCheckBox("Check 2"))
        checkbox_layout.addWidget(QCheckBox("Check 3"))
        checkboxes = QGroupBox("Checkboxes")
        checkboxes.setLayout(checkbox_layout)
        checkboxes.setMaximumSize(300, 70)


        self.lineedit = QLineEdit("Type Some Text Here")
        self.listwidget = QListWidget()
        self.listwidget.addItem("List Item 1")
        self.listwidget.addItem("List Item 2")
        self.listwidget.addItem("List Item 3")
        self.listwidget.addItem("List Item 4")
        self.listwidget.addItem("List Item 5")
        self.listwidget.setSelectionMode(QAbstractItemView.MultiSelection)

        progress = QProgressBar()
        slider = QSlider()
        slider.setOrientation(Qt.Horizontal)
        slider.valueChanged.connect(progress.setValue)
        self.textbox = QPlainTextEdit()
        self.textbox.setTabStopWidth(20)
        fp = open("css/default.css")
        self.textbox.setPlainText(fp.read())
        fp.close()

        dialogbutton = QPushButton("Display Dialog")
        dialogbutton.clicked.connect(self.openDialog)
        updatebutton = QPushButton("Update Styles")
        updatebutton.clicked.connect(self.updateStyle)

        grid = QGridLayout()
        grid.addWidget(checkboxes,      0, 0, 1, 2) # span 1 row, 2 columns
        grid.addWidget(radiobuttons,    0, 2, 1, 2) # span 1 row, 2 columns
        grid.addWidget(self.listwidget, 1, 0, 2, 1) # span 2 rows, 1 column
        grid.addWidget(self.dropdown,   1, 1)
        grid.addWidget(spinbox,         2, 1)
        grid.addWidget(slider,          1, 2, 1, 2)
        grid.addWidget(progress,        2, 2, 1, 2)
        grid.addWidget(self.lineedit,   3, 0, 1, 2) # span 1 row, 2 columns
        grid.addWidget(dialogbutton,    3, 2)
        grid.setRowStretch(4, 1)        # row 4 will stretch with the interface
        grid.addWidget(self.textbox,    4, 0, 1, 4) # span 1 row, 4 columns
        grid.addWidget(updatebutton,    5, 3)

        mainwidget = QWidget()
        mainwidget.setLayout(grid)
        mainwidget.setObjectName("main")
        self.setCentralWidget(mainwidget)
        self.setWindowTitle("Style Me")
        self.setGeometry(20,20, 500, 600)
        self.show()

    def updateStyle(self):
        styleSheet = self.textbox.toPlainText()
        qApp.setStyleSheet(styleSheet)

    def openDialog(self):
        thetext = "Selected List Items: "
        for item in self.listwidget.selectedItems():
            thetext = thetext + "\n     " + item.text()
        thetext = thetext + "\nSelected ComboBox Item: " + "\n     " + self.dropdown.currentText()
        thetext = thetext + "\nEntered Text: " + "\n      " + self.lineedit.text()
        message = QMessageBox()
        message.setText(thetext)
        message.exec_()

if __name__ == "__main__":
    app = QApplication(sys.argv)
    win = MyWindow()
    sys.exit(app.exec_())
```



```
/* The QWidget named "main" gets a linear gradient as a background
from (0,0) - upper left, to (1,1) - bottom right. There are three stops in the
gradient at 0, 0.7 and 1 along the scale from 0 to 1. */
QWidget#main {
    background-color: qlineargradient(x1: 0 ,y1: 0 ,x2: 1 ,y2: 1,
                                stop: 0 #fff2d1, stop: 0.7 #ff4477
                                stop: 1 #ffffdd);
}

QCheckBox {
    font-weight: bold;
    font-style: italic;
}

QCheckBox::indicator {
    width: 25px;
    height: 25px;
}

QCheckBox::indicator:unchecked {
    image: url(images/thumbup-blk.png);
```

The interface of the above program with the grid layout indicated

A few things to note in this program...

There's a lot of widgets in this program, but I won't go into detail on all of them. You can do your own research by looking up each widget and its functionality online. One striking thing about this interface is that it uses a QGridLayout to align all the widgets nicely. A grid layout treats your interface as a table of rows and columns. Using the function `addWidget(row, column, rowspan, columnspan)` you can set the row and column where you want your widget to appear. You can also optionally set a rowspan and columnspan for widgets that should span over multiple rows and columns. *Note that rows and columns start counting at 0!*

Lines 18-24 use a GroupBox to group together three radio buttons. First the radio buttons are added to a vertical layout, and then the layout is set on the GroupBox. Besides giving you a nice little title and styleable border around your radio buttons, a GroupBox is necessary to get radio buttons behaving the way you expect them to (where only one can be selected at a time). An identical approach is used to create a visual group of three checkboxes in ***lines 26-32***.

Lines 50-52 fill the big text editor widget with a default stylesheet. Rather than clutter up my program with a big multi-line string I've moved this stylesheet into css/default.css - note that both the css directory and the images directory mentioned inside default.css are relative to the directory where you run the python command from.

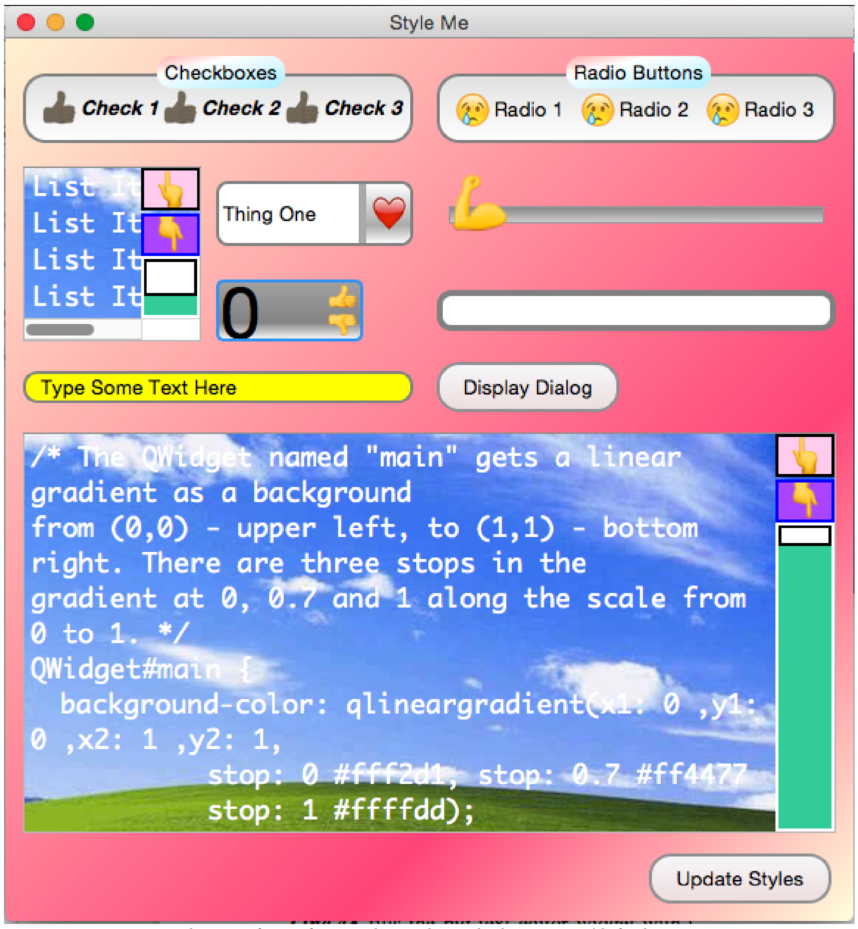
Lines 59-71 fill the grid layout with widgets. Take some time to compare these lines to the grid overlay in the image above.

Lines 81-83 define the function that updates the global stylesheet on qApp from whatever text is in the text edit field.

Lines 87-88 give an example of how you can iterate through the selected items of a QListWidget

Lines 91-93 display a modal QMessageBox window to the user. QMessageBox is a special class of model window based on QDialog. Zetcode has a nice tutorial on dialog windows if you'd like to use them http://zetcode.com/gui/pyqt5/dialogs/

For further inspiration, check out the Qt website's list of Stylesheet Examples
http://doc.qt.io/qt-5/stylesheet-examples.html



```
/* The QWidget named "main" gets a linear
gradient as a background
from (0,0) - upper left, to (1,1) - bottom
right. There are three stops in the
gradient at 0, 0.7 and 1 along the scale from
0 to 1. */
QWidget#main {
  background-color: qlineargradient(x1: 0 ,y1:
0 ,x2: 1 ,y2: 1,
            stop: 0 #fff2d1, stop: 0.7 #ff4477
            stop: 1 #ffffdd);
```

The interface after applying the stylesheet in css/default.css

Exercise

Take your time studying the program above and playing around with the style sheets. PyQt will look for files in the directory where you type the python command. If you're having trouble getting your images to show up, make sure that you're typing the python command inside the directory where your program file is.

Once you're satisfied, start working on your own interface look and feel. Use the software in your interface research as a starting point and try to replicate the look and feel of that software. If you're investigating a large piece of software with a diverse set of interfaces (say.. facebook) ...just try focusing on reproducing one part of the interface.