TROLLTECH®

DEV DAYS '07

Qt Styles and Style Sheets

Girish Ramakrishnan

About me

- Me
  - Girish Ramakrishnan
  - Software Developer + Release manager
- Qt Development
  - Qt Style Sheet architect
  - MinGW platform boss
  - Auto completion framework
  - Desktop integration
  - Part of widget team (message box, label)

Agenda

- My grand plan
  - Tell you everything you need to know about styling and theming Qt applications
- History and evolution of Qt's style API
- QStyle API Tour
- Qt Style Sheets
- Qt Style Sheet Future

Before I get into the history...

- How does a widget paint itself?

```
QPushButton::paintEvent(QPaintEvent *event)
{
    QPainter p(this);

    // draw lines, fill with color

}
```

- Problem definition : What needs to be done to paint a button that looks and feels native across platforms?

Qt 1 (24 Sep 1996)

- GUIStyle QWidget::style() const // enumeration
- QPushButton::paintEvent(QPaintEvent *event)

```
  {
    QPainter p(this);

    if (style() == MotifStyle) {
        // draw like motif
    } else if (style() == WindowsStyle) {
        // draw like Windows
    }
}
```

- Custom styling required subclassing every widget
  - Didn't work for containers like QFileDialog, QMessageBox

Qt 2 (26 Jun 1999)

- Introduced QStyle, a new class, that paints depending on the platform.
- class QStyle {

```
virtual void drawButton ( ... )
virtual QRect buttonRect ( .... )
virtual void drawButtonMask ( ... )
virtual void drawBevelButton ( ... )
QRect bevelButtonRect ( ... )
}
```

- QMotifStyle, QPlatinumStyle, ...
- New virtual functions could not be added because of Binary compatibility

Qt 3 (16 Oct 2001)

- Functions -> enums (Adding enums is BC)
- enum ControlElement {

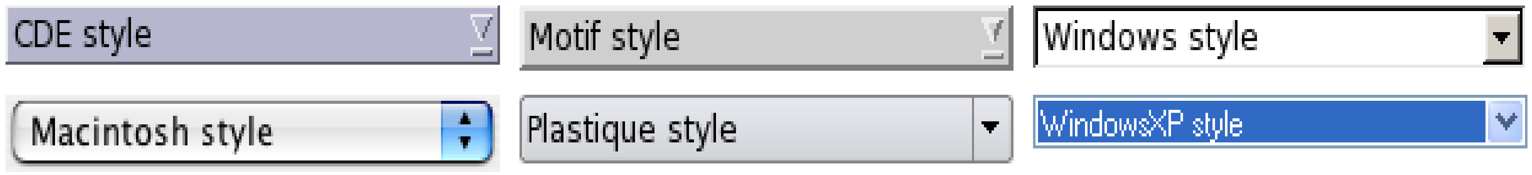
```
CE_PushButton
CE_PushButtonLabel
}
QStyle::drawControl(ControlElement, QPainter *, const QWidget *,...)
```

- The style used the widget pointer to determine widget options and features
  - Is the button flat?
  - Is the button the default button?

Qt 4 (28 Jun 2005)

- Qt was split - QtGui and Qt3Support
- Qt3 widgets still need to be styled
  - Accessing widget pointer creates dependancy!
  - What if you want to print the appearance of a widget?
- QStyleOption introduced to store the widget options and features
  - QStyleOption is a base class that stores properties that are common to all widgets (e.g, state, palette, fontMetrics)
  - QStyleOption is then subclassed for widget specific features. (e.g) QStyleOptionButton
- QStyle::drawControl(ControlElement, const QstyleOption *, QPainter *, const QWidget * = 0)

Qt Style API advantages

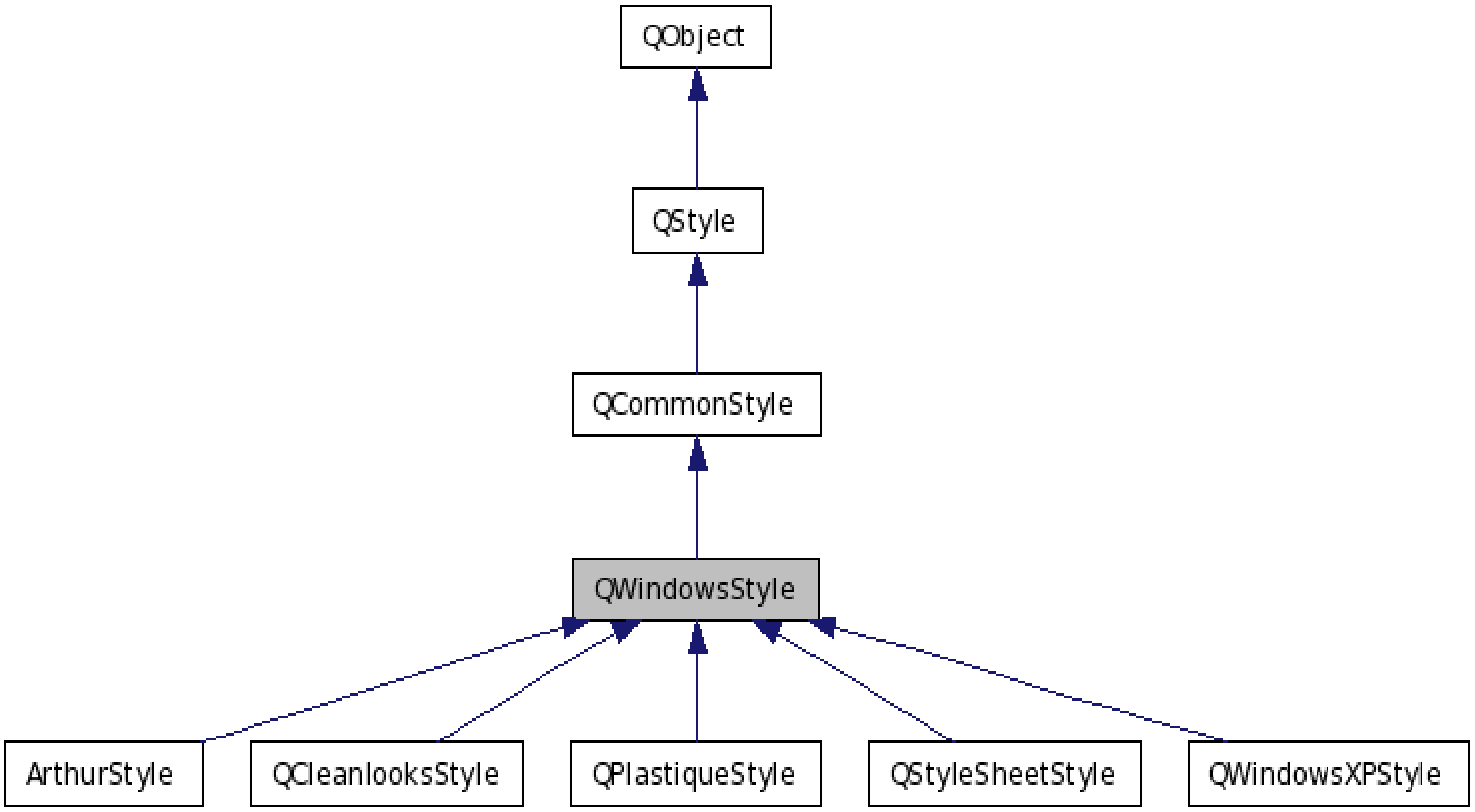


- Can draw widgets differently on different platforms.
- Look and feel of widgets can be changed without having to subclass every widget
- Implement QStyle plugins that can be used in any Qt application

QStyle API peek

- Draws a widget or a part of a widget
  - drawPrimitive(), drawControl(), drawComplexControl()
- Helps determine where parts of widgets are located
  - subControlRect(), hitTestComplexControl()
- Gives the widget a standard size
  - sizeFromContents()
- Provides “hints” for style specific behavior
  - styleHint()
- Provides standard icons
  - standardIcon()
- Helps the layout system to layout items

QStyle hierarchy



QStyle hierarchy

- QStyle is an Abstract Class with practically no implementation.
- QCommonStyle provides a good place to start implementing a style.
- QWindowsStyle provides an even better place to start implementing a style :-)

How does paintEvent() look these days?

```
QPushButton::paintEvent(QPaintEvent *event)
{
    QPainter p(this);
    QStyleOptionButton option;
    initStyleOption(&option);
    style()->drawControl(CE_PushButton, p, &option, this);
}
```

- The widget uses the style to do the actual painting.
- Lets dig into the details...

How does paintEvent() look these days?

```
QPushButton::paintEvent(QPaintEvent *event)
{
    QPainter p(this);
    QStyleOptionButton option;
    initStyleOption(&option);
    style()->drawControl(CE_PushButton, p, &option, this);
}
```

QStyleOption

- QStyleOption holds generic styling information
- Has the following public members
  - int version;
  - int type;
  - QStyle::State state; // user interface state
  - Qt::LayoutDirection direction; // RTL or LTR
  - QRect rect; // rectangle
  - QFontMetrics fontMetrics; // convenience metrics
  - QPalette palette; // widget palette
- Does not derive from Qobject (no qobject_cast!)
- initFrom(QWidget *) tries to initialize the above members using the widget

QStyleOptionButton

- Widget specific information in subclasses
- QStyleOptionButton is a subclass containing
  - ButtonFeatures features; // none, flat, hasmenu, default
  - QString text;
  - QIcon icon;
  - QSize iconSize;
- Adding new members to an existing structure is binary incompatible
- QStyleOptionButtonV2 will be created to add new members

qstyleoption_cast

- qstyleoption_cast inspects the version and Type members

```
class QStyleOptionProgressBar : public QStyleOption
    enum { Type = SO_ProgressBar };
    enum { Version = 1 };
    int minimum;
    int maximum;
    int progress;
    QString text;
    Qt::Alignment textAlignment;
    bool textVisible;
```

- if (const QStyleOptionButton *button = qstyleoption_cast<const QStyleOptionButton *>(opt)) {

```
  // use button here...
}
```

How does paintEvent() look these days?

```
QPushButton::paintEvent(QPaintEvent *event)
{
    QPainter p(this);
    QStyleOptionButton option;
    initStyleOption(&option);
    style()->drawControl(CE_PushButton, p, &option, this);
}
```

Initializing a QStyleOption

- Before 4.3, one needed to populate by hand

```
QStyleOptionButton option;
option.initFrom(this); // populates common fields
option.text = text();
option.icon = icon();
if (isDown()) option.state |= Qstyle::State_Sunken;
```

- 4.3 introduced initStyleOption() for most Qt widgets

```
QStyleOptionButton option;
initStyleOption(&options); // does the above
```

How does paintEvent() look these days?

```
QPushButton::paintEvent(QPaintEvent *event)
{
    QPainter p(this);
    QStyleOptionButton option;
    initStyleOption(&option);
    style()->drawControl(CE_PushButton, p, &option, this);
}
```

The style() pointer

- style() pointer is never 0. Every widget has a style() pointer
- On startup, QApplication::style() init'ed to the appropriate style subclass depending on the platform
  - Can be overriden using the -style <stylename> command line
- QWidget::style() returns QApplication::style()
  - Overriden using QWidget::setStyle()

How does paintEvent() look these days?

```
QPushButton::paintEvent(QPaintEvent *event)
{
        QPainter p(this);
        QStyleOptionButton option;
        initStyleOption(&option);
        style()->drawControl(CE_PushButton, p, &option, this);
}
```

QStyle API

```
void MyStyle::drawControl(ControlElement element, const QStyleOption *
    option, QPainter * painter, const QWidget * widget = 0 ) const {
     switch (element) {
     case CE_PushButton:
          if (const QstyleOptionButton *button = qstyleoption_cast<const
             QstyleOptionButton *>(option)) {
             // use button and painter to draw
          }
          }
          break;
     default: break;
     }
     SuperClass::drawControl(element, option, painter, widget);
}
```

QStyle API

```
void MyStyle::drawControl(ControlElement element, const QStyleOption *
    option, QPainter * painter, const QWidget * widget = 0 ) const {
      switch (element) {
      case CE_PushButton:
            if (const QstyleOptionButton *button = qstyleoption_cast<const
                QstyleOptionButton *>(option)) {
                // use button and painter to draw
            }
            }
            break;
      default: break;
      }
      SuperClass::drawControl(element, option, painter, widget);
}
```

QStyle API

```
void MyStyle::drawControl(ControlElement element, const QStyleOption *
    option, QPainter * painter, const QWidget * widget = 0 ) const {
    switch (element) {
    case CE_PushButton:
        if (const QstyleOptionButton *button = qstyleoption_cast<const
            QstyleOptionButton *>(option)) {
            // use button and painter to draw
        }
        }
        break;
    default: break;
    }
    SuperClass::drawControl(element, option, painter, widget);
}
```

QStyle API

```
void MyStyle::drawControl(ControlElement element, const QStyleOption *
      option, QPainter * painter, const QWidget * widget = 0 ) const {
        switch (element) {
        case CE_PushButton:
                if (const QstyleOptionButton *button = qstyleoption_cast<const
                    QstyleOptionButton *>(option)) {
                    // use button and painter to draw
                }
                }
                break;
        default: break;
        }
        SuperClass::drawControl(element, option, painter, widget);
        // Delegate unknown elements to our super class
}
```

QStyle tour—Drawing functions

“Primitive” elements include frames, indicators, and panels

```
drawPrimitive(PrimitiveElement elem, const QStyleOption *option,
            QPainter *painter, const QWidget *widget = 0) const
```

Single part controls include buttons, tabs, and progress bars

```
drawControl(ControlElement element, const QStyleOption *option,
          QPainter *painter, const QWidget *widget = 0) const
```

Mulit-part controls like scrollbars, title bars, and combo boxes

```
drawComplexControl(ComplexControl control,
                const QStyleOptionComplex *option,
                QPainter *painter,
                const QWidget *widget = 0) const
```

QStyle tour—Drawing helper functions

- Draw text or a pixmap in a given rectangle with a given alignment and palette
- Usually called inside QStyle functions

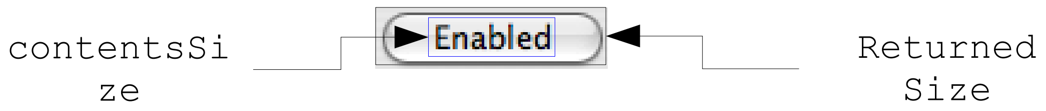
```
drawItemText(QPainter *painter, const QRect &rect, int alignment,
      const QPalette &pal, bool enabled,
                const QString &text,
                QPalette::ColorRole textRole = QPalette::NoRole) const
drawItemPixmap(QPainter *painter, const QRect &rect,
                  int alignment, const QPixmap &pixmap) const
```

Metrics and Sizes

- Metrics are a collection of little numbers that tweak various elements

```
int pixelMetric(PixelMetric metric, const QStyleOption *option = 0,
                const QWidget *widget = 0) const
```

- Many widgets have a content size, but the style might actually need a different size.



```
QSize sizeFromContents(ContentsType type,
                       const QStyleOption *option,
                       const QSize &contentsSize,
                       const QWidget *widget = 0) const
```

QStyle tour—Rectangle functions

- Return a rectangle for an element in a widget
- For Controls:

```
QRect subElementRect(SubElement element,
                     const QStyleOption *option,
                     const QWidget *widget = 0) const
```

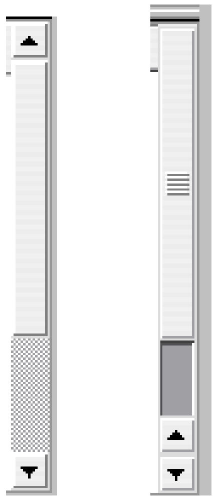
- For Complex Controls:

```
QRect subControlRect(ComplexControl control,
                     const QStyleOptionComplex *option,
                     SubControl subControl,
                     const QWidget *widget = 0) const
```

QStyle tour—Hit testing

- Typically used in mouse events in complex controls

```
SubControl hitTestComplexControl(ComplexControl control,
                                 const QStyleOptionComplex *option,
                                 const QPoint &pos,
                                 const QWidget *widget = 0) const
```



Usually implemented in terms of
`subControlRect()`

QStyle tour—Style hints

- Give widgets hints to how they should act
  - Usually use the int returned
  - You can get extra information by passing QStyleHintReturn subclass
    - Works very similar to QStyleOption (subclass, qstyleoption_cast<T>(), etc.)

```
int styleHint(StyleHint hint, const QStyleOption *option = 0,
              const QWidget *widget = 0,
              QStyleHintReturn *returnData = 0) const
```

QStyle Tour—Icon/Pixmap functions

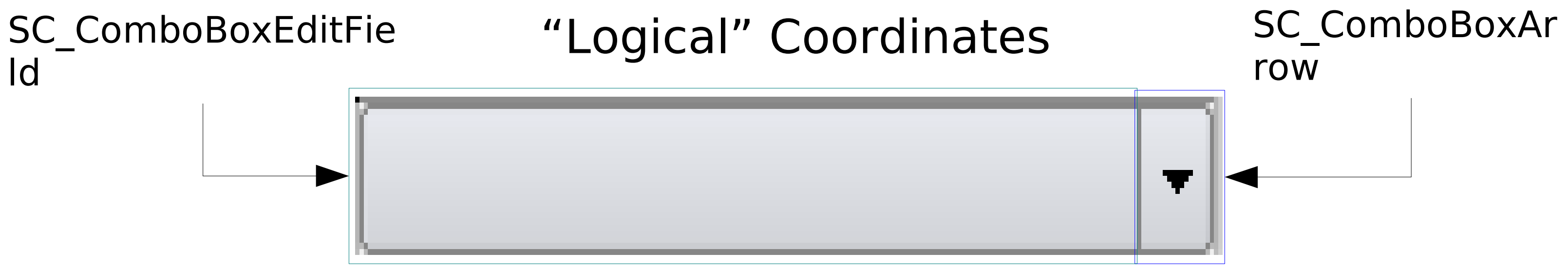
- QStyle can return typical pixmaps like file system images, message box icons, etc.

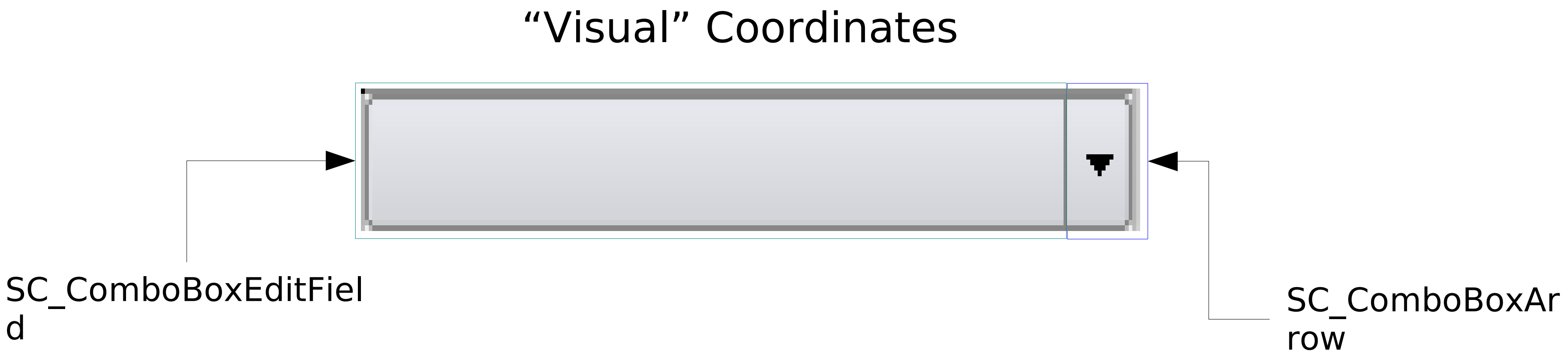
```
QPixmap standardPixmap(StandardPixmap standardPixmap,
                       const QStyleOption *option = 0,
                       const QWidget *widget = 0) const
QIcon standardIconImplementation(StandardPixmap standardIcon,
                               const QStyleOption *opt = 0,
                               const QWidget *widget = 0) const;
```

- Given a normal icon from QIcon, QStyle can generate an active or disabled icon

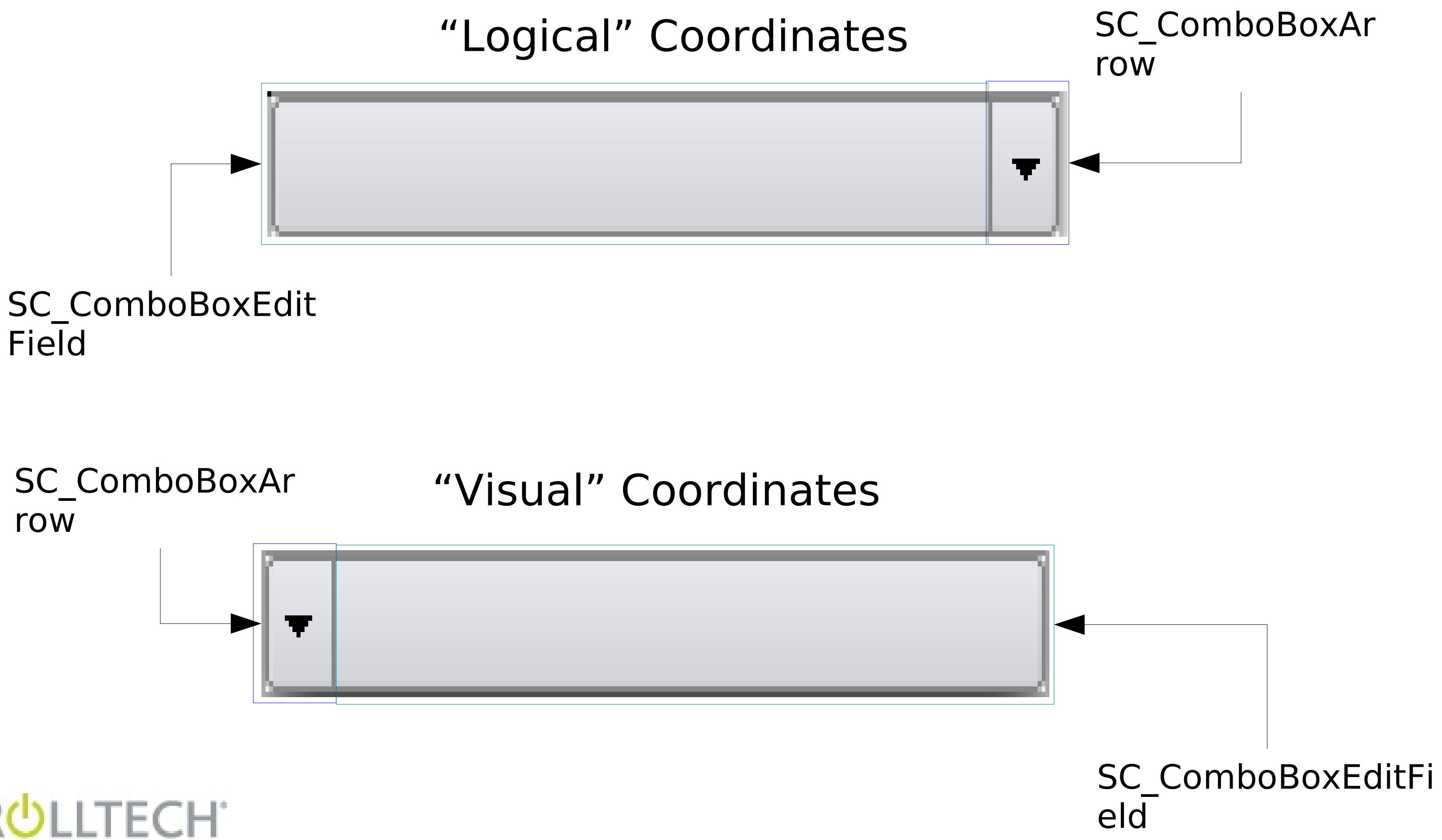
```
QPixmap generatedIconPixmap(QIcon::Mode iconMode,
                            const QPixmap &pixmap,
                            const QStyleOption *option) const
```

Logical vs. Visual in Left-to-Right





Logical vs. Visual in Right-to-Left



QStyle—static functions

- These functions are very handy for right-to-left languages (RTL)

```
QRect visualRect(Qt::LayoutDirection direction,
                 const QRect &boundingRect,
                 const QRect &logicalRect)
QPoint visualPos(Qt::LayoutDirection direction,
                 const QRect &boundingRect,
                 const QPoint &logicalPos)
QRect alignedRect(Qt::LayoutDirection direction,
                  Qt::Alignment alignment, const QSize &size,
                  const QRect &rectangle)
Qt::Alignment visualAlignment(Qt::LayoutDirection direction,
                              Qt::Alignment alignment)
```

Right-to-Left languages

- QWidget::layoutDirection() transferred to QStyleOption
- Styles shipped with Qt 4 always return visual rectangles in subControlRect() and subElementRect()
- Test with -reverse on the command line and use QStyle's static functions to help

Demo

- Lets look at some code!

QStyle Tips

- Try to find a style that already does most of what you want and derive from that
- If you just need stuff for a specific widget, maybe you just should re-implement QWidget::paintEvent()
- Don't depend on the QWidget pointer (it can be anything)
- Be careful with filling and contents propagation (especially in Qt 4.1)

The problem with QStyle

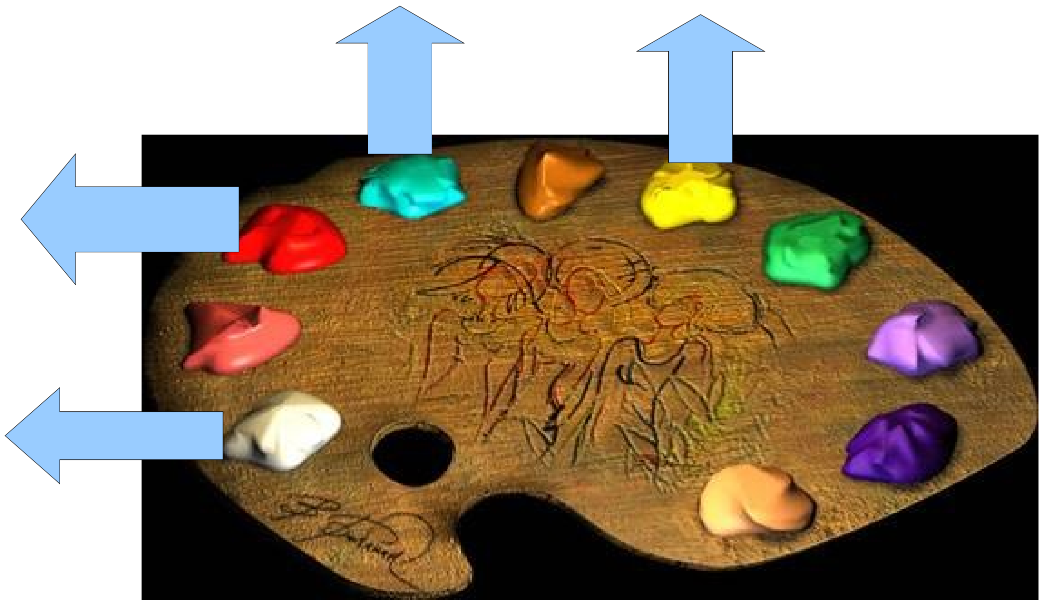
- QStyle is quite complex
  - QStyle is super flexible
  - We have to use it to style Qt for all platforms
- “Compile, test, compile test” cycle is annoying
  - Styling is mostly about tweaking pixels
- QStyle completely shuts off graphics designers
- Simple customizations (changing foreground, background) is quite complex
  - involves fiddling with the palette
  - What's the problem here?

What is QPalette?

- Let's start out by understanding what a QPalette is.
- Palette is a structure used to store “system colors”
  - The keyword here is “system colors”

QPalette roles

- Colors have roles



QPalette continued...

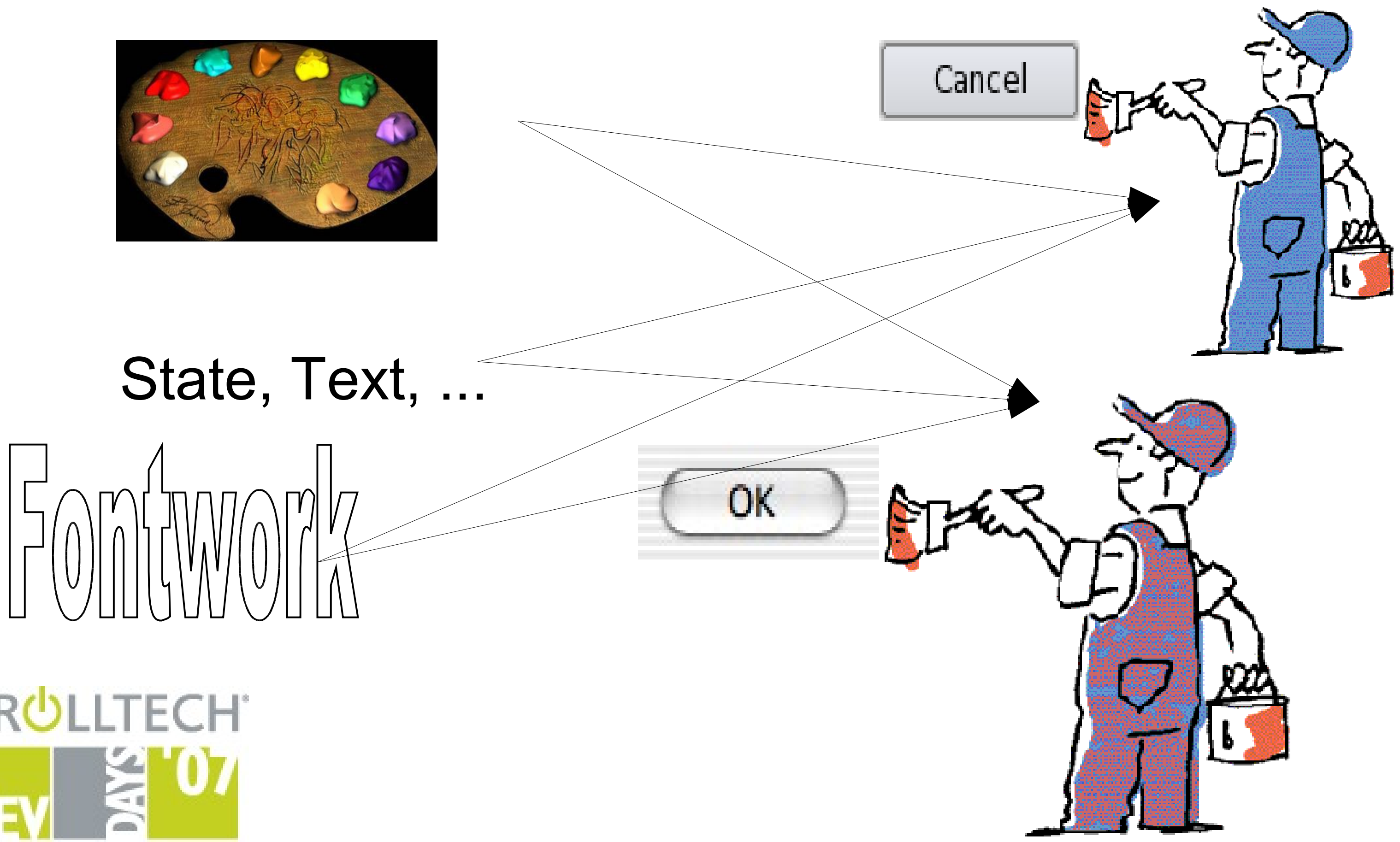
- Changing the palette is seen as a way to change the appearance (color) of a widget

```
QPalette p;
p.setColor(QPalette::Button, Qt::red);
button->setPalette(p);
```

- Setting color “sometimes” works
  - Why?

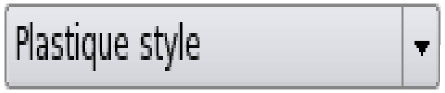
QStyle+QPalette

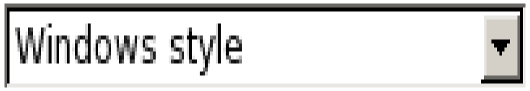
- QStyle completely controls widget rendering
- The palette is, but a, hint. It can be completely ignored.



QStyle+QPalette

- Inconsistent palette usage
  - ColorRole used is decided by the QStyle
  - Plastique uses Button role, Windows uses Base role





Mr. Plastique

Mr. Windows

History of Style Sheet

- Customers have been complaining about palette problems
- Customizing appearance seemed to be a very complex affair
- “Red PushButton” project
  - *“Customers want red color push buttons. Make it possible.”* - Matthias Ettrich

Qt Style Sheet (Qt 4.2)

- Simple customization like setting colors and backgrounds
- Style Guarantees
  - Red buttons will be red everywhere
- Complete customization of widget look
- Introduced in Qt 4.2 for form widgets

Qt Style Sheet (Qt 4.3)

- Qt 4.3 makes practically any widget stylable
- Support for gradients
- Styling using SVG
- Makes QPalette colors accessible from style sheets
- Support for background-attachment in a scroll area
- Syntax highlighter and validator in Designer
- Many advanced CSS3 properties

Style Sheets are ...

- “Style strings”
  - Similar to CSS in concept and syntax
  - Adapted to the world of widgets
- Scalable
  - CSS3 border-image
- Interactive UI development
  - Designer friendly. Compile free

Style Sheet Syntax

- A Style Rule

```
QPushButton { color: red; }
```

QPushButton – Selector
{ color: red } - Declaration
color – attribute/property

Style Sheet Syntax

- A Style Sheet is a set of rules

```
QPushButton, QCheckBox {
   background-color: magenta;
}

QRadioButton, QCheckBox {
   spacing: 8px;
   color: gray;
}
```

Selectors

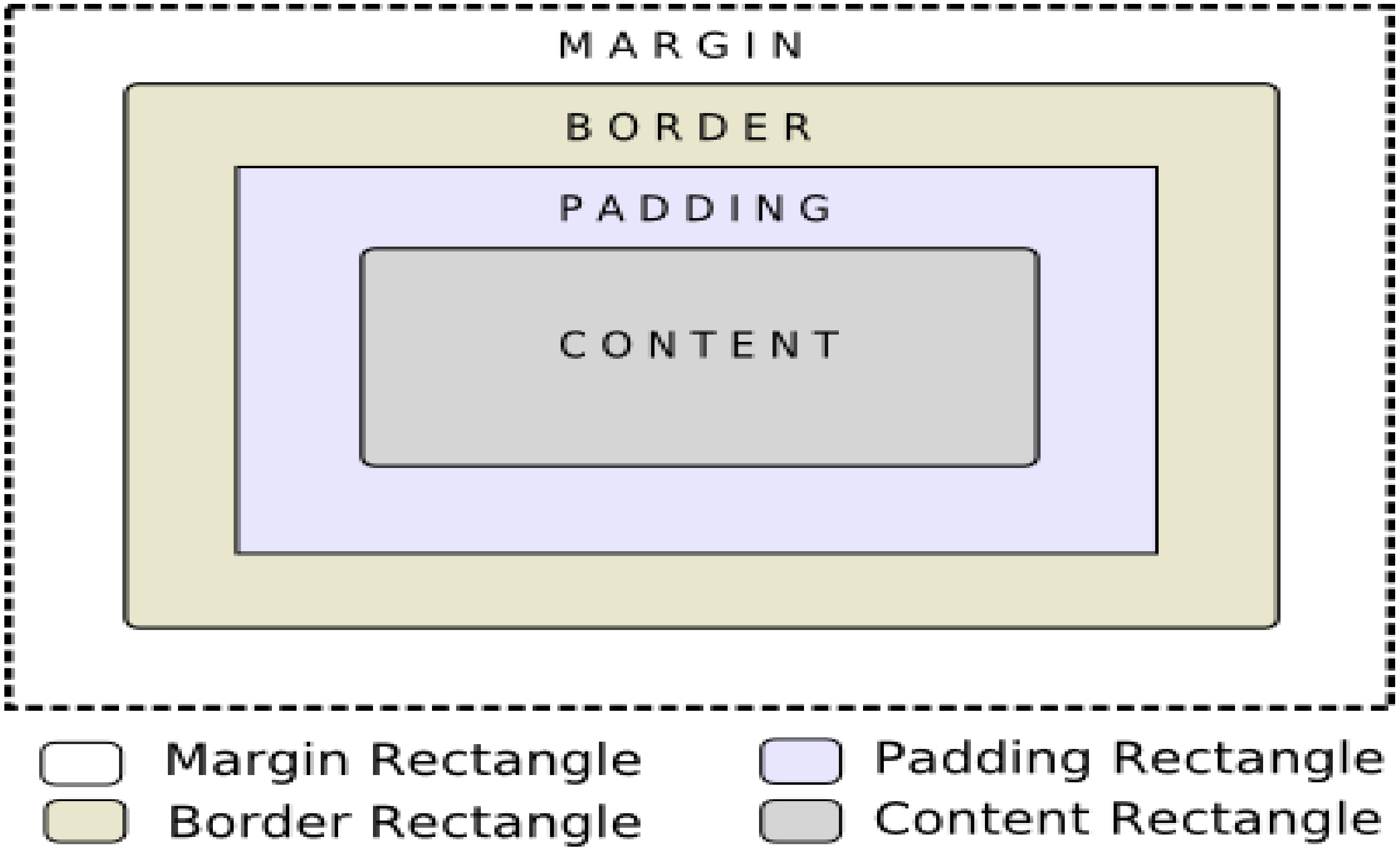
- Selectors decide whether a rule applies
  - *Type* QPushButton { color: red }
  - *Class* .QCheckBox { background-color: pink; }
  - *ID* #foodMenuView { alternating-color: gray; }
  - *Descendant* QDialog QRadioButton { spacing: 10px; }
  - *Attribute* QCheckBox[text=”falafel”]
    { font-size: 14pt; }

- Whole range of CSS2.1 selectors supported

Pseudo States

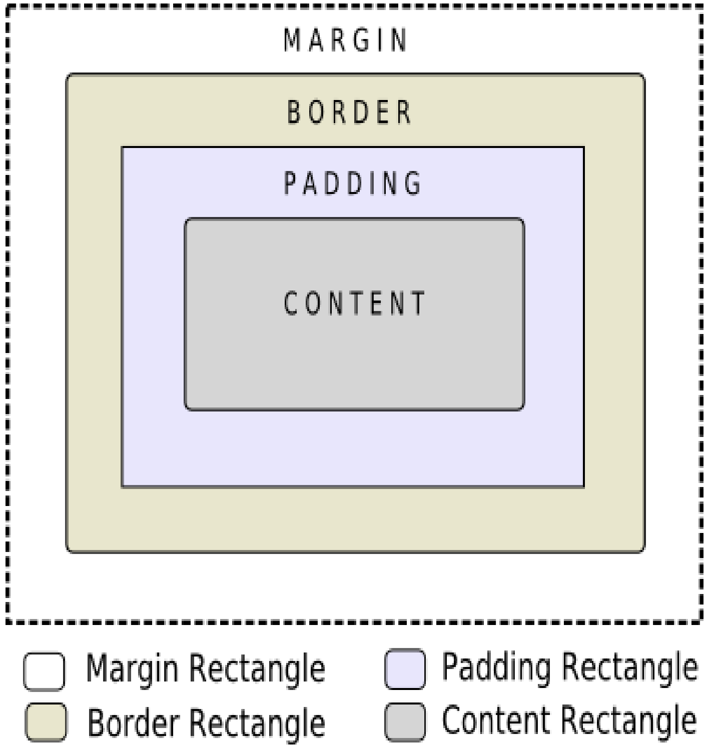
- Pseudo states limit the rule to the widget's state
  - `QPushButton:hover { color: red }`
  - `QCheckBox:pressed:checked { color: pink }`
  - :checked , :disabled , :enabled, :focus, :hover , :indeterminate , :off :on , :pressed, :unchecked (around 36)
  -
  - Use “!” for negation
    - QPushButton:!hover { color: white; }

The Box Model



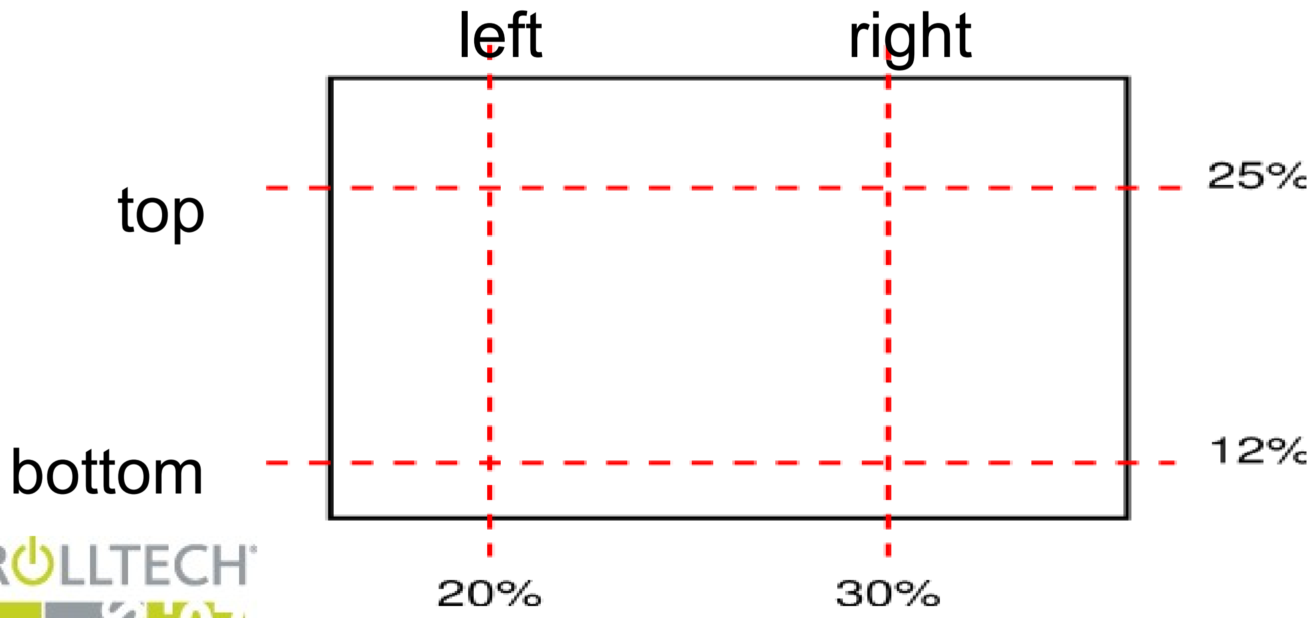
Qt Style Sheet Box Model Attributes

- margin
- border
  - border-width
  - border-style
  - border-color
- padding
- background
  - background-image
  - background-repeat
  - background-position
  - background-clip, background-origin
- min-width, min-height
  - width and height refer to contents rect



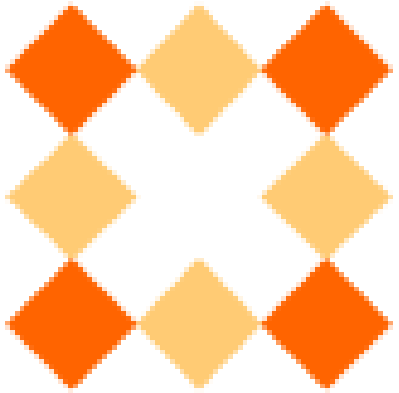
Border Image

- Borrowed idea from CSS3 to make borders scalable
- Image cut into 9 parts



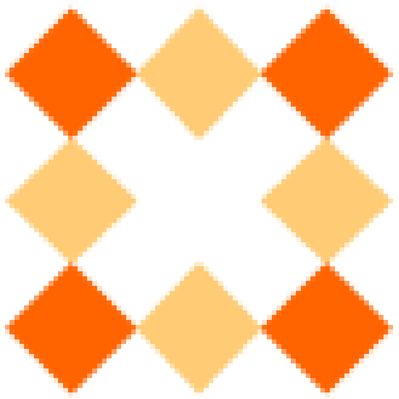
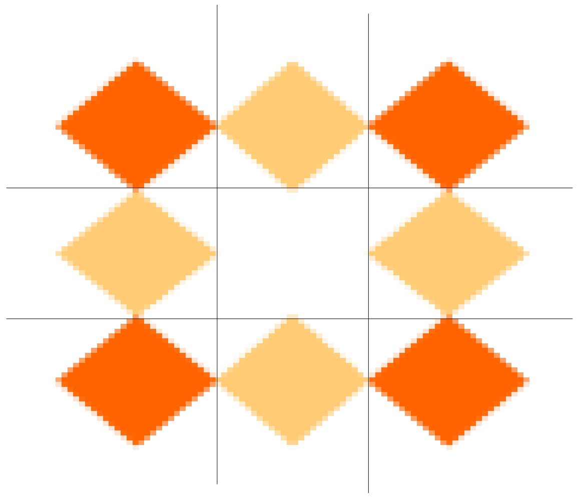
Border Image

- border-image: url(x.png) 3 3 3 3 tile stretch
  - border-image: url t r b l hstretch vstretch



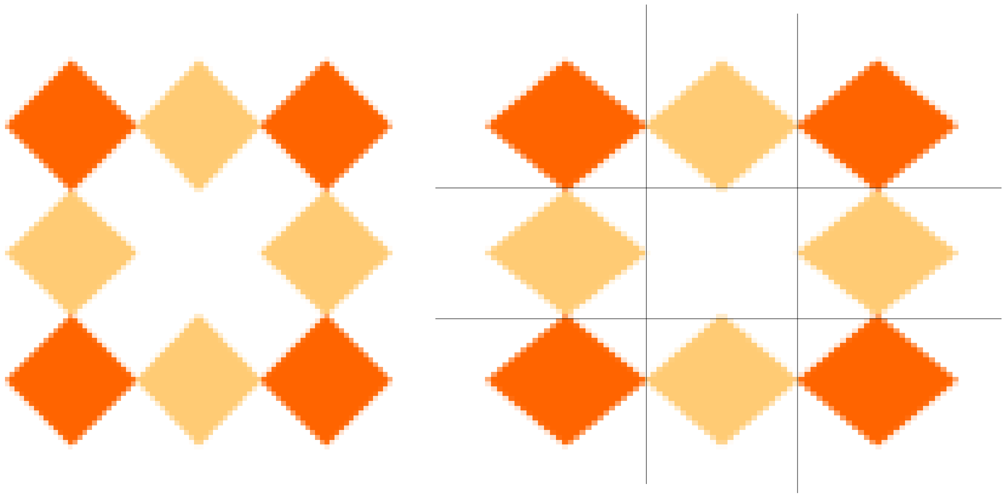
Border Image

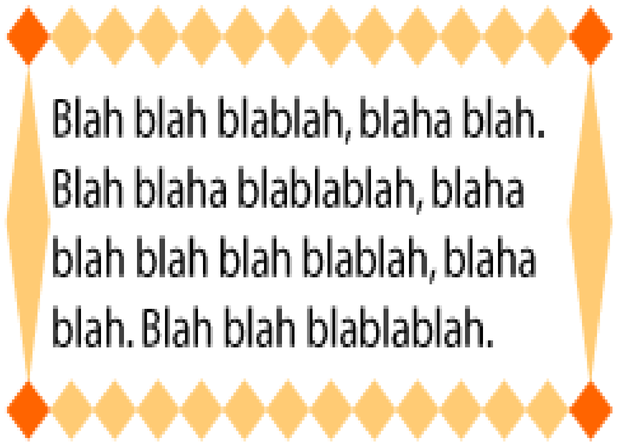
- `border-image: url(x.png) 3 3 3 3 tile stretch`

Border Image

- `border-image: url(x.png) 3 3 3 3 tile stretch;`

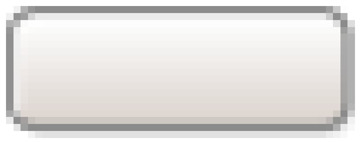




Blah blah blablah, blaha blah. Blah blaha blablablah, blaha blah blah blah blablah, blaha blah. Blah blah blablablah.

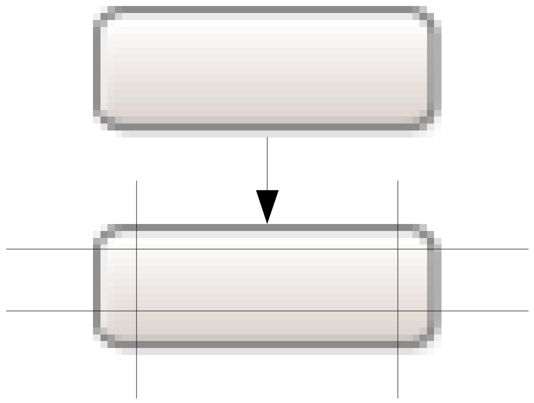
Border Image

- `border-image: url(btn.png) 5 5 5 5 stretch stretch;`



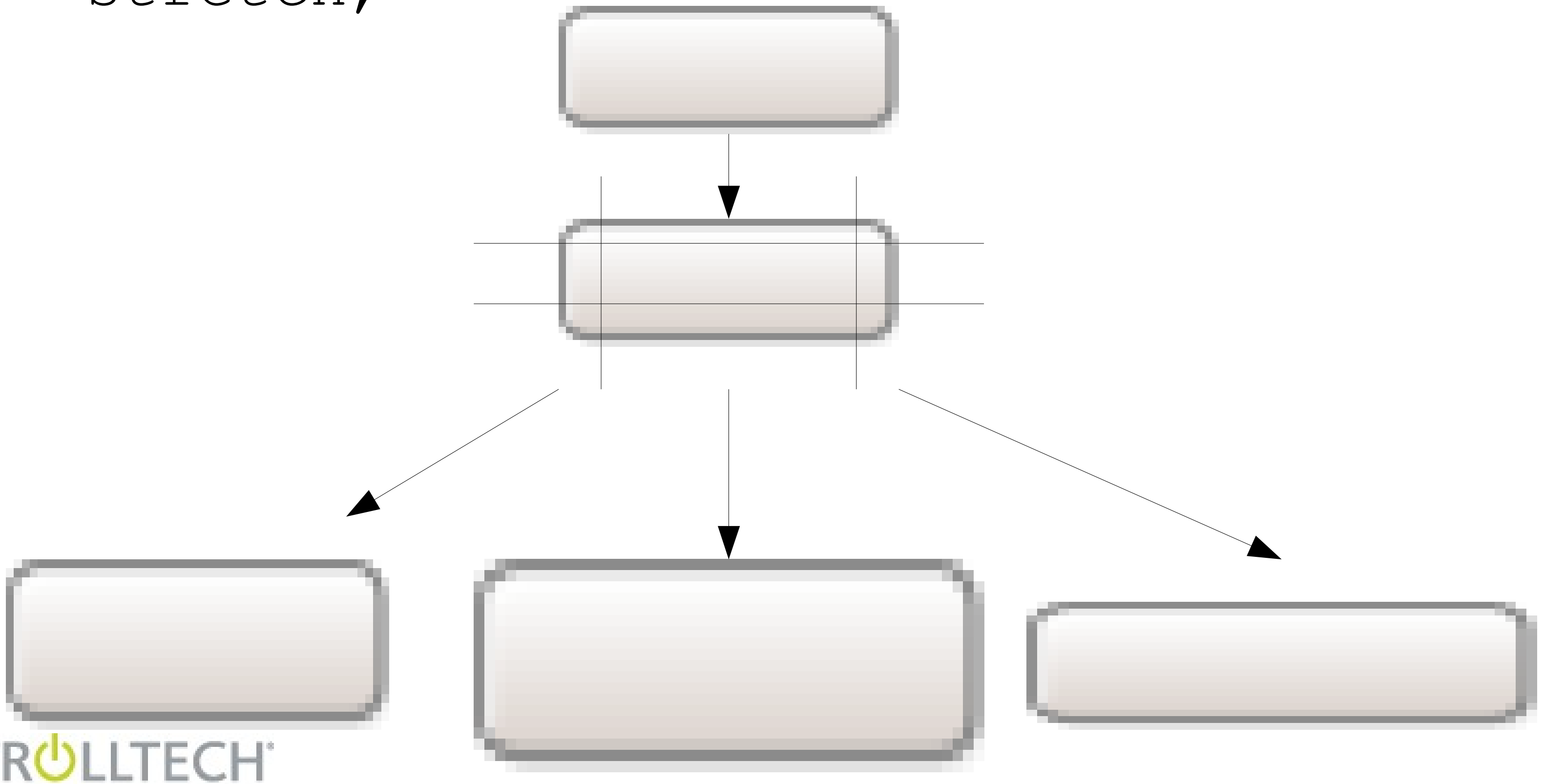
Border Image

- `border-image: url(btn.png) 5 5 5 5 stretch stretch;`



Border Image

- `border-image: url(btn.png) 5 5 5 5 stretch stretch;`



Rendering Model

- Set clip for entire rendering operation
  - border-radius
- Draw the background
  - First fill with background-color
  - Then draw the background-image
  - Depends on various properties like background-origin, background-clip, background-repeat, background-attachment
- Draw the border
  - Draw the border-image or the border
- Draw overlay image
  - image
  - Depends on image-alignment property

Style Sheet Computation

- Style Sheet can be set on QApplication or any QWidget

Consider,

Application Style Sheet

```
   QLineEdit { color: red; }
```

Dialog Style Sheet

```
     #myLineEdit { color: red; background:
   yellow }
```

Line Edit Style Sheet

```
     color: white;
```

Style Sheet Computation

- Selectors have weights
  - More specific selectors have more weight
    - `#myButton`
    - `QpushButton:enabled`
    - `QPushButton[flat=”false”]`
      - `(e.g) .QPushButton`
    - `QPushButton`

For example,

```
#myButton { color: red; }
.QPushButton { color: green }
```

Style Sheet Computation

- When rules have the same weight, the latter wins
- For example,

```
QPushButton { color: red; }
QAbstractButton { color: green; }
```

Style Sheet Computation

- Style rules are “merged”

  For example,

  ```
  #myButton { color: red; }
  QPushButton { color: pink; background:
   white; }
  ```

- Final style sheet

  ```
  { color: red; background: white }
  ```

Style Sheet Computation

- Sources of Style Sheet
  - Widget style sheet
  - Ancestor widgets' style sheet
  - Application style sheet

- When conflicts arise,
  - Widgets style sheet preferred over ancestor
  - Ancestor style sheet preferred over application style sheet

Style Sheet Computation

- Example,

Application Style Sheet

```
QLineEdit { color: red; background: blue; border: 2px solid green }
```

Dialog Style Sheet

```
#myLineEdit { color: red; background: yellow }
```

Line Edit Style Sheet

```
* { color: white; }
```

Final result,

color: white; background: yellow; border: 2px solid green

Sub controls

- Sub controls are “Parts” of complex widget
  - drop-down indicator of combo box
  - menu-indicator of push button
  - buttons of a spin box

- Follow CSS3 Pseudo Element syntax but has little to do conceptually
  - QPushButton::menu-indicator { image:url(indicator.png) }

Sub controls

- Sub controls are styled the exact same way as normal elements
  - Box model
  - *border-image* property
  - *image* property
  - Pseudo states

```
QPushButton::menu-indicator:hover {
  border: 2px solid red;
 image: url(indicator_hover.png)
}
```

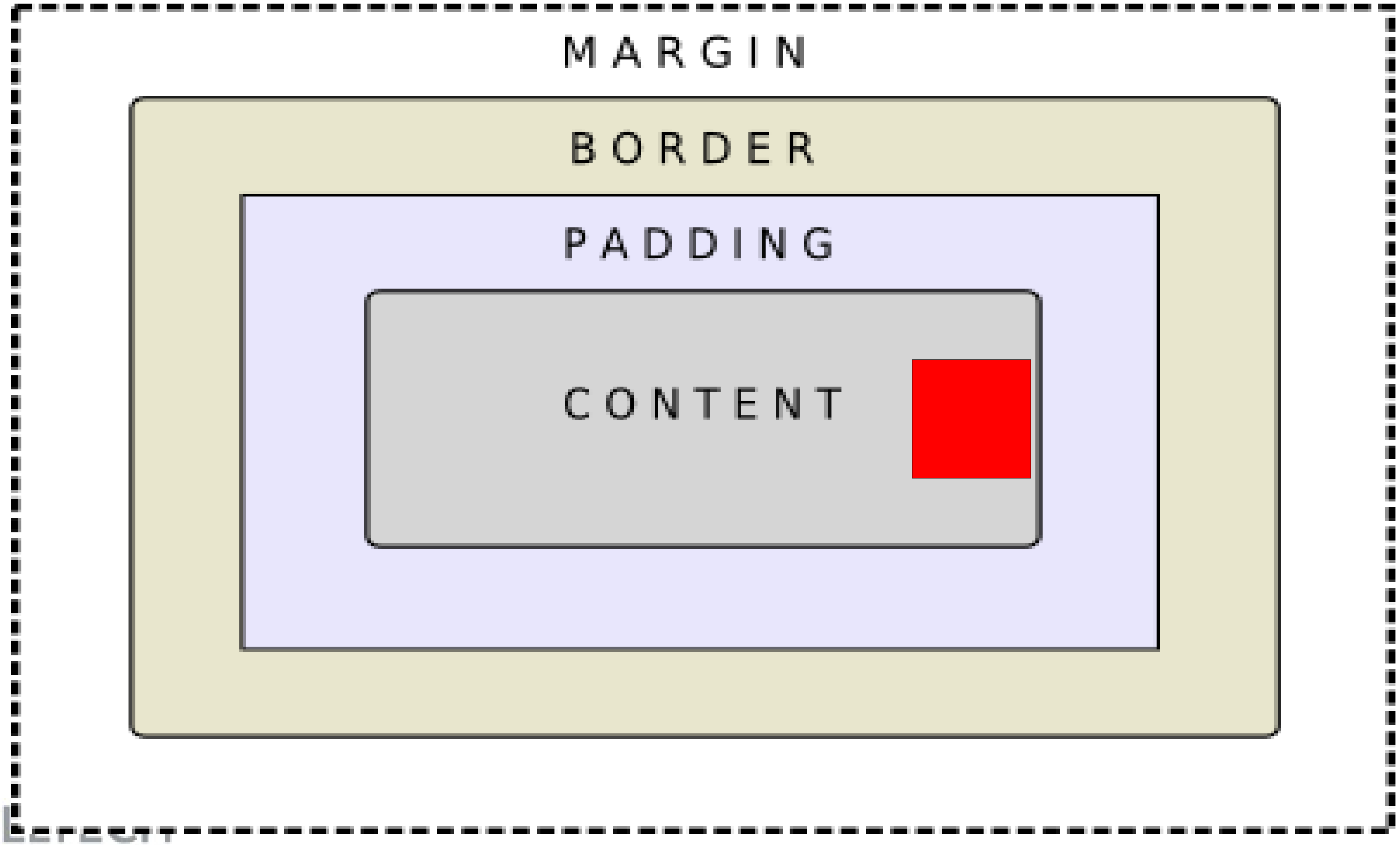
Sub controls (Geometry)

- Size
  - width, height
  - width and height of the element's *content*
- subcontrol-origin
  - The origin rect in the parent's box model
- subcontrol-position
  - The alignment within the above rect

Sub control positioning

```
QPushButton::menu-indicator {
    subcontrol-origin: content;
    subcontrol-position: right center;
    image: url(menu.png);
}
```

Sub control positioning



Positioning Modes

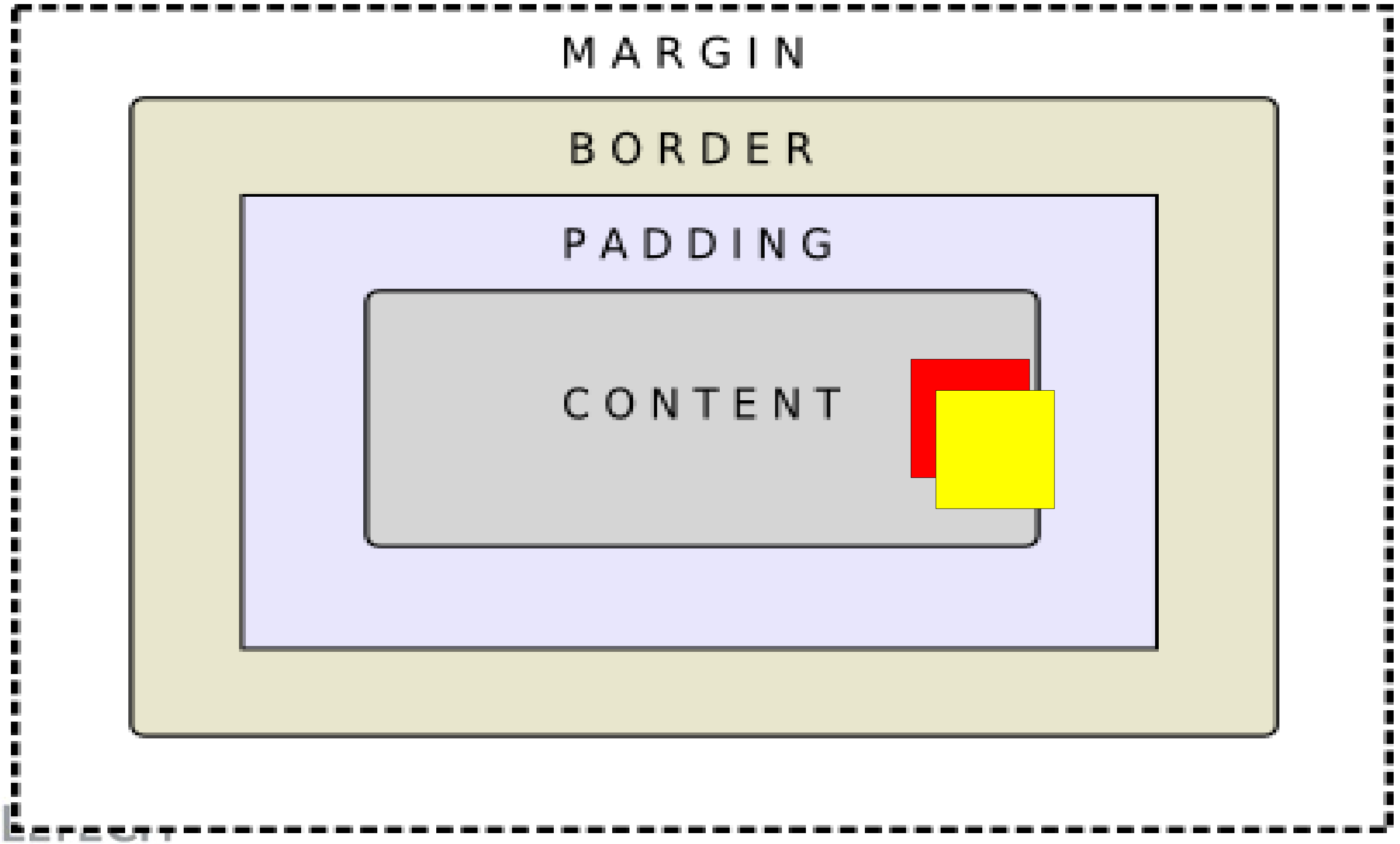
- Fine tune position using position modes
  - Relative Positioning
  - Absolute Positioning

Relative Positioning

- Use top, left, right, bottom to move a sub control from its current position

```
QPushButton::menu-indicator {
    subcontrol-origin: content;
    subcontrol-position: right center;
    image: url(menu.png);
    position: relative;
    top: 5px; left: 4px;
}
```

Relative Positioning

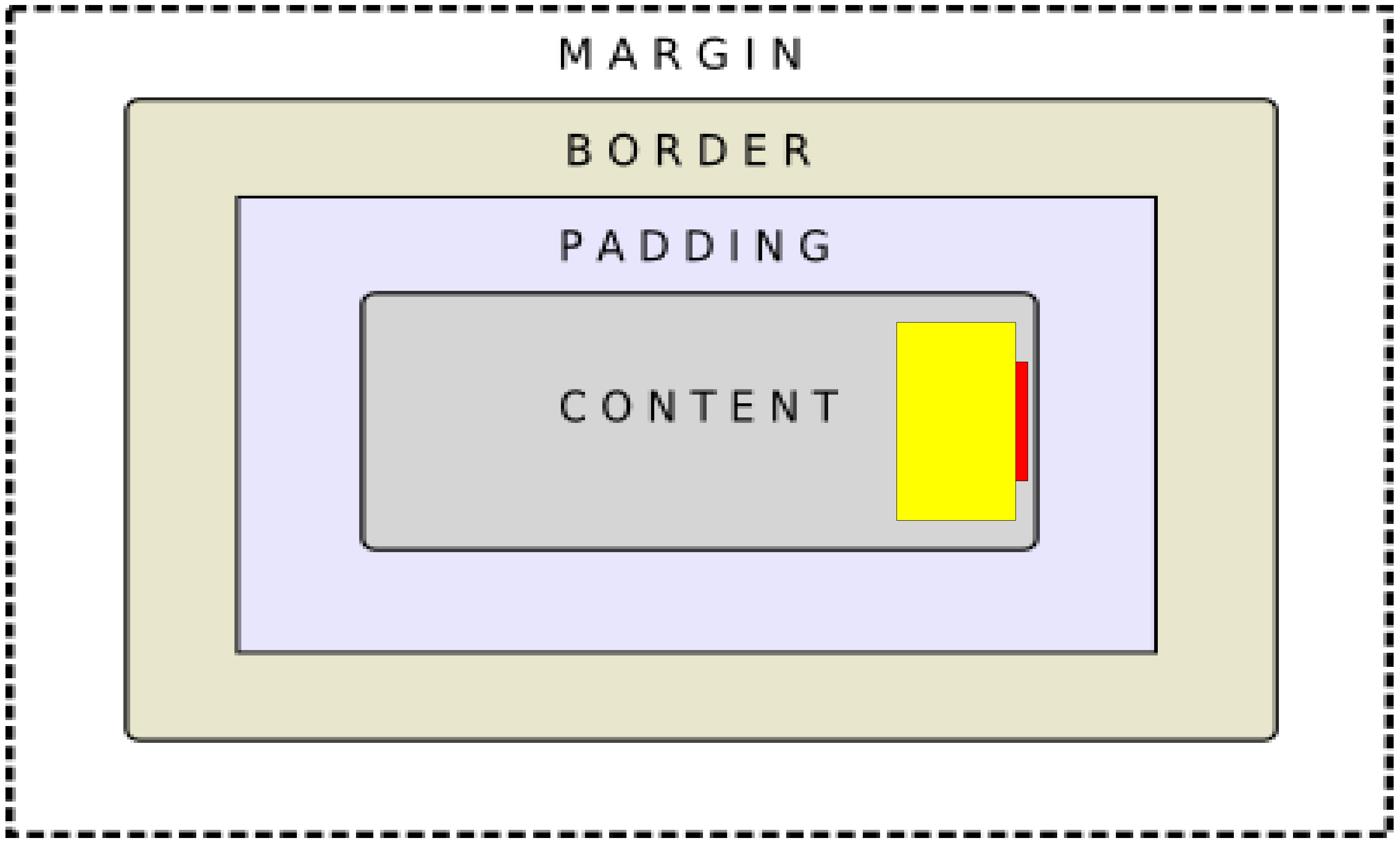


Absolute Positioning

- Absolute Positioning
  - Use top, left, right, bottom to define absolute positions within the origin rectangle

```
QPushButton::menu-indicator {
    subcontrol-origin: content;
    subcontrol-position: right center;
    image: url(menu.png);
    position: absolute;
    width: 20px;
    top: 5px; right: 2px; bottom: 5px;
}
```

Absolute Positioning

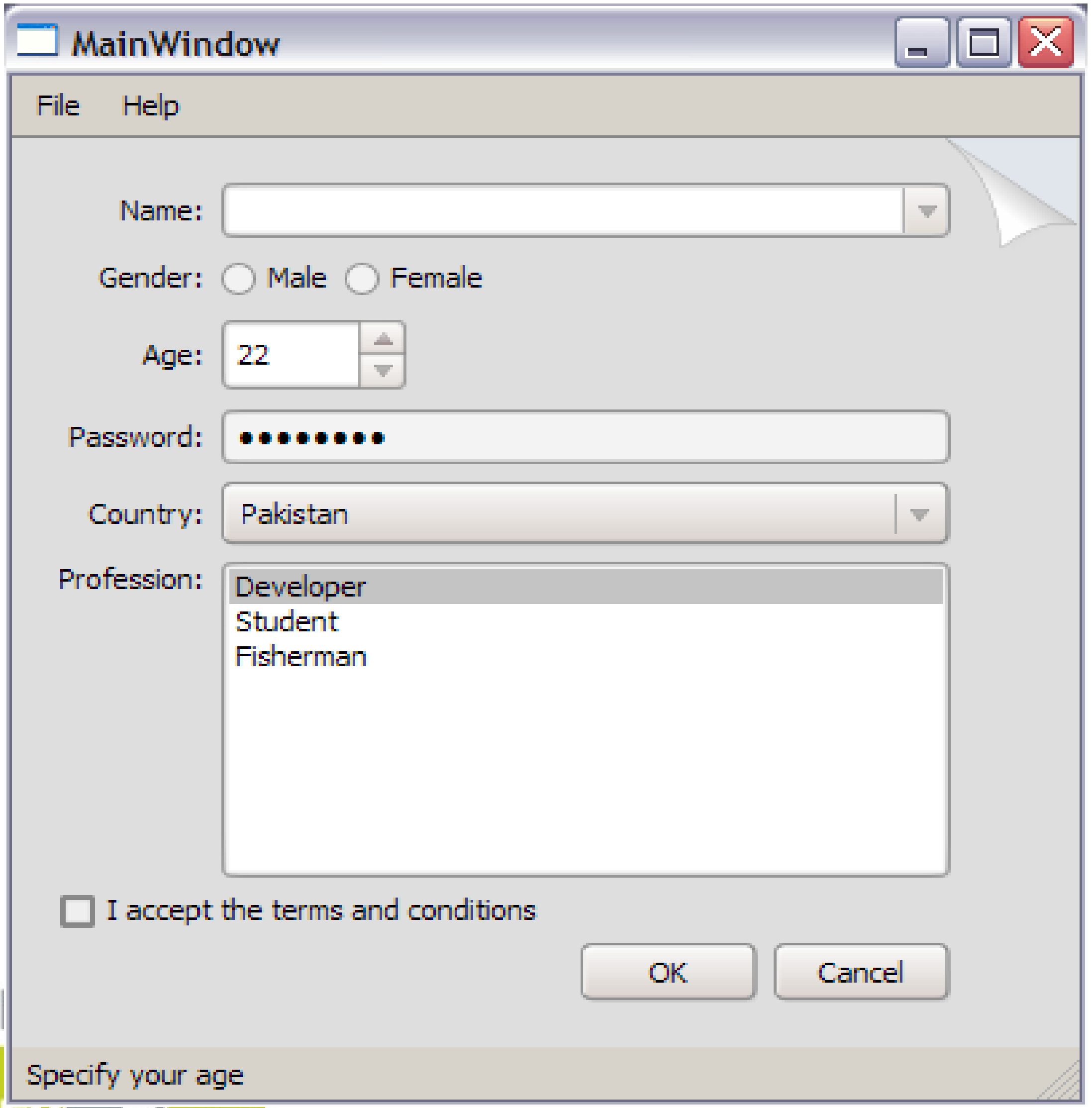


position:absolute; width: 20px; top: 5px; right: 2px; bottom: 5px;

Rendering Model

- Widget is defined by subcontrols drawn on top of each other
- Rendering
  - QComboBox { }
  - QComboBox::drop-down { }
  - QComboBox::down-arrow { }
- Subcontrols always have a parent and are positioned w.r.t parent
  - The drop-down is positioned w.r.t QComboBox
  - The down-arrow is positioned w.r.t drop-down
-

DEMO 1


MainWindow

File Help

Name:

Gender: ( ) Male ( ) Female

Age: 22

Password: ●●●●●●●●●

Country: Pakistan

Profession:
- Developer
- Student
- Fisherman

[ ] I accept the terms and conditions

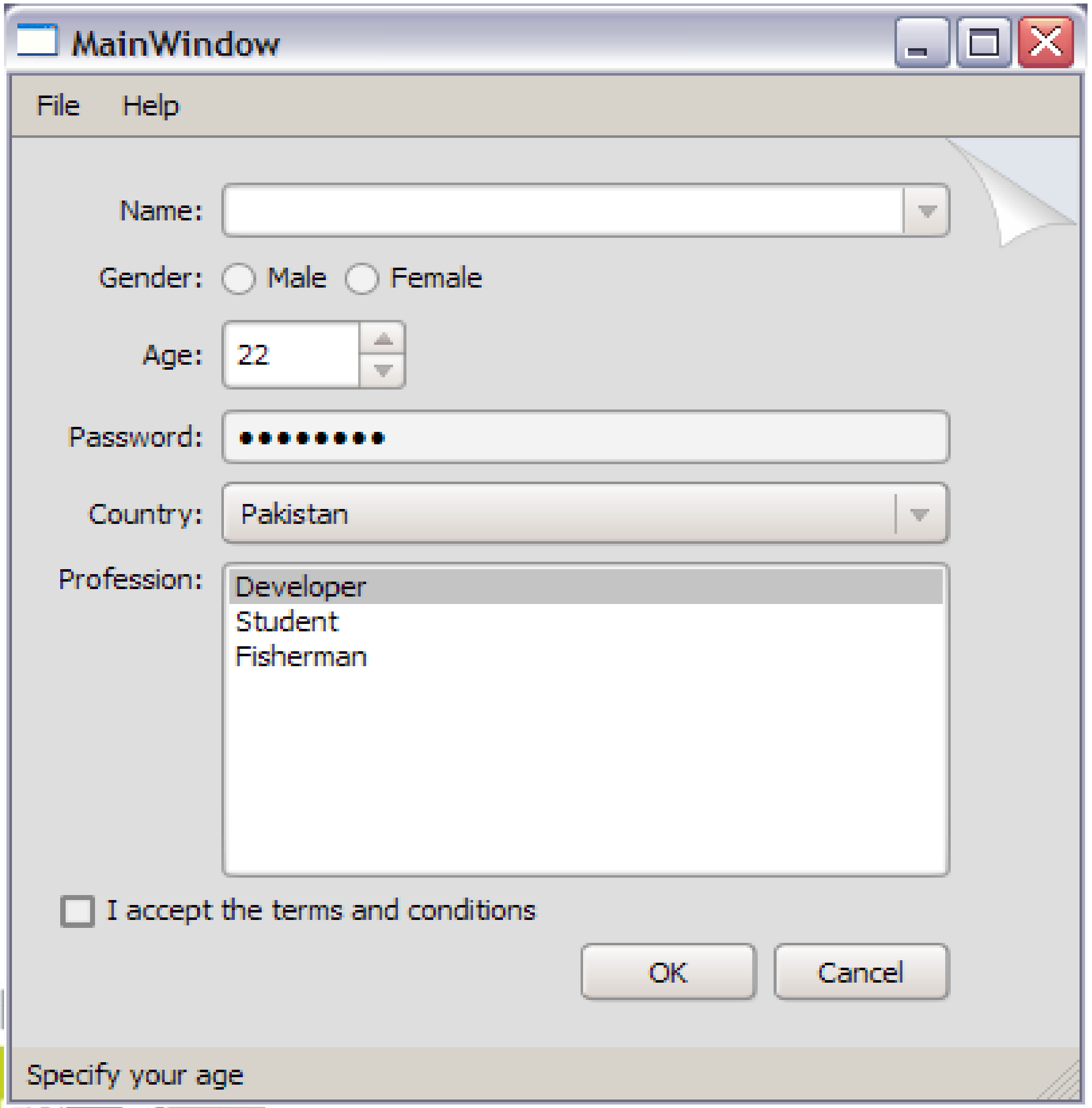
OK Cancel

Specify your age

Negative Margins

- A box's positive margin either pushes content away in the same direction as its margin, or pushes the box itself in the opposite direction of the margin.
- A box's negative margin either pulls content over it in the opposite direction as its margin, or pulls the box itself in the same direction as the margin.

DEMO (2)


MainWindow

File Help

Name:

Gender: ( ) Male ( ) Female

Age: 22

Password: ●●●●●●●●●

Country: Pakistan

Profession:
Developer
Student
Fisherman

☐ I accept the terms and conditions

OK Cancel

Specify your age

Advanced features

- You can set any Q_PROPERTY using style sheets
  - QlistView { qproperty-alternatingRowColors: true; }
- Dynamic properties can be used in the selector
  - QObject::setProperty can be used to create properties dynamically

Advanced features

- Can customize various style hints

  "activate-on-singleclick", "button-layout", "gridline-color", "lineedit-password-character", "messagebox-text-interaction-flags", "opacity", "show-decoration-selected"

Qt 4.3's well kept secret

- Qt 4.3's well kept secret

Future direction

- Customizable icons and icon size
  - Already in 4.4 snapshots!
- Make all widgets styleable
  - QDockWidget, QmdiSubWindow already in snapshots!
- Support Mac
- Support for RTL
  - We are trying to complete the above two for 4.4
- Support custom QStyle subclasses
  - Works only with Qt's built-in QStyles

More information

- Style Sheet Example
  - examples/widgets/stylesheet

- Style Sheet Documentation
  - http://doc.trolltech.com/4.3/stylesheet.html

- Labs
  - http://labs.trolltech.com/blogs/