Introdução ao Qt e QtWidgets

Sandro S. Andrade

sandroandrade@kde.org

@andradesandro

Objetivos

- Apresentar as principais características e funcionalidades do Qt
- Apresentar as extensões ao modelo de objetos do C++ disponibilizadas pelo Qt
- Proporcionar vivências práticas sobre como o QtWidgets pode ser utilizado para construções de aplicações gráficas modernas

Agenda

- Visão geral do Qt
- Modelo de objetos do Qt
- Sinais e slots
- Layout e parentesco
- Fundamentos de QtWidgets
- Model-View

Agenda

- Containers Qt
- Banco de Dados
- Trabalhando com plug-ins

Visão Geral do Qt

- O Qt é um *toolkit* (conjunto de frameworks) para desenvolvimento de aplicações multiplataforma
- Disponível publicamente desde maio de 1995
- Possui mais de 1000 classes
- Possui licença dual (LGPLv3 e comercial):
  - http://www.qt.io

Visão Geral do Qt

- Porque usar o Qt ?
  - É uma tecnologia madura (20 anos de existência)
  - É um toolkit extremamente produtivo (mesmo com C++ e melhor ainda com QML/JS)
  - É um toolkit bastante completo

Visão Geral do Qt

- Porque usar o Qt ?
  - É efetivo no suporte ao desenvolvimento multiplataforma:
    - Linux/X11, Windows, OS X, Android, IOS, WinCE
  - Excelente documentação e comunidade bastante ativa
  - Excelente desempenho (aceleração via hardware no QML)

Visão Geral do Qt

- Porque usar o Qt ?
  - Diversas bibliotecas de terceiros baseadas no Qt (inqlude.org)
  - Open governance com licença dual:
    - LGPL
    - Comercial

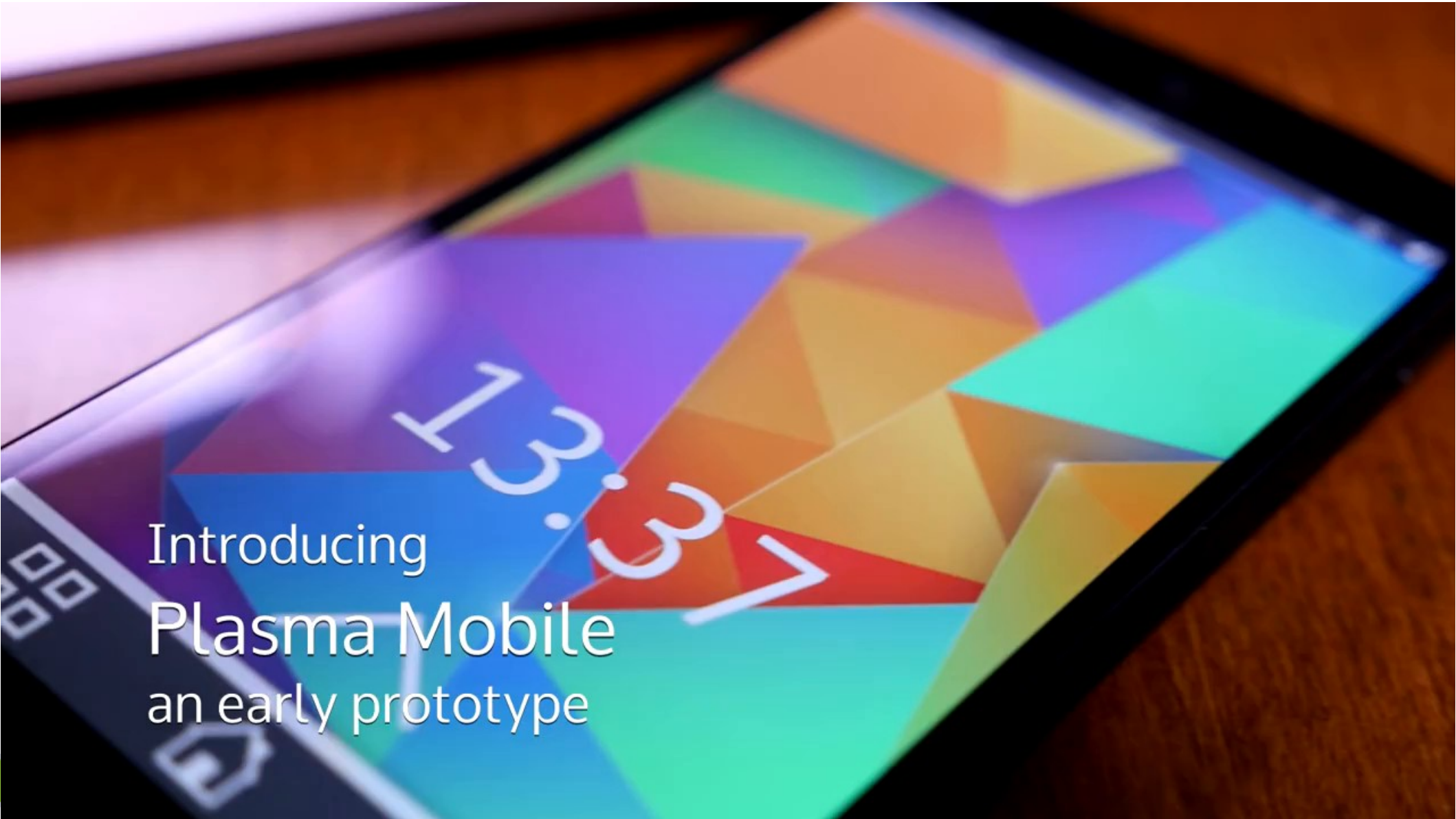
Visão Geral do Qt

- Quem usa o Qt ?

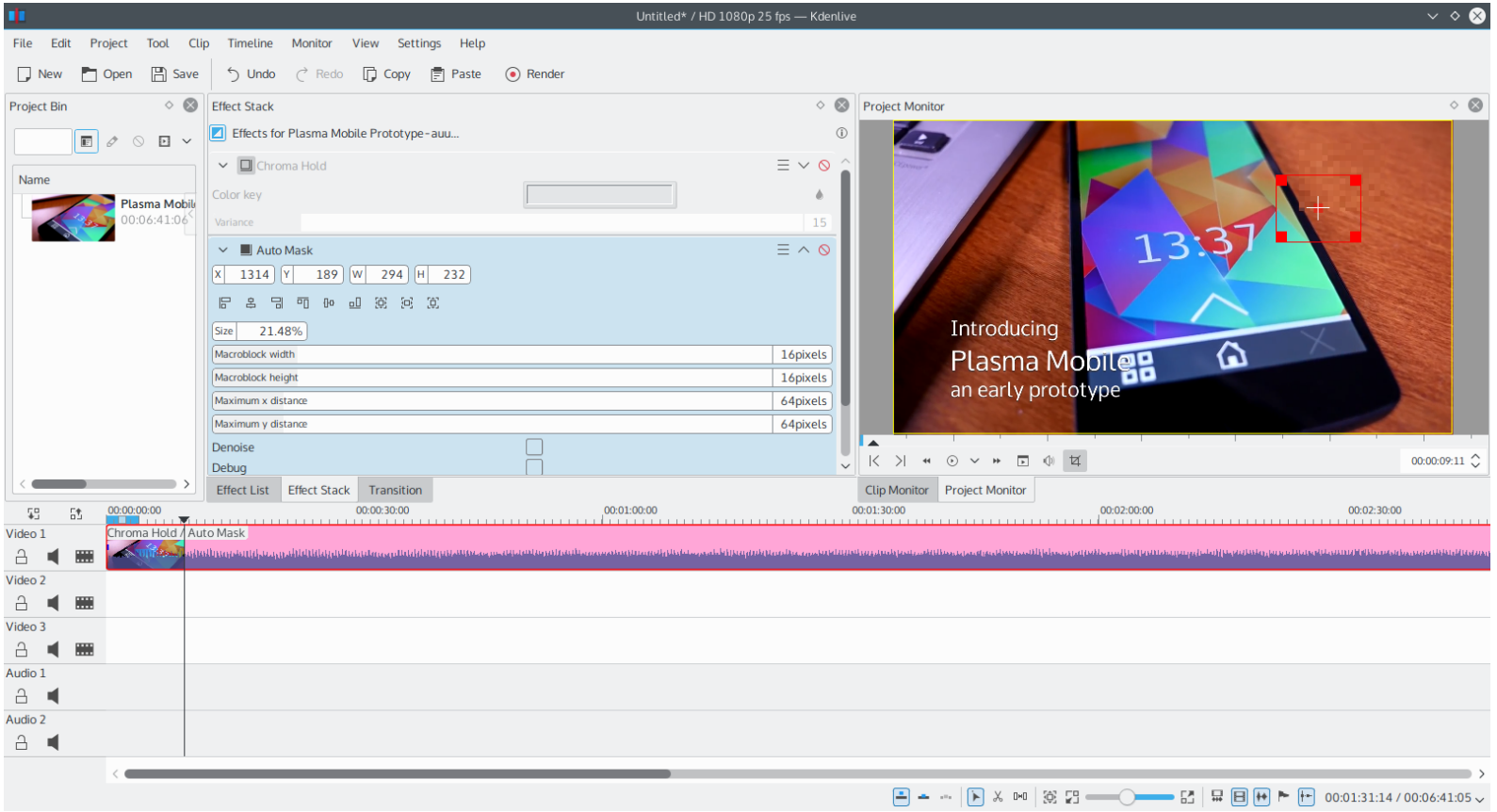
Visão Geral do Qt

Visão Geral do Qt

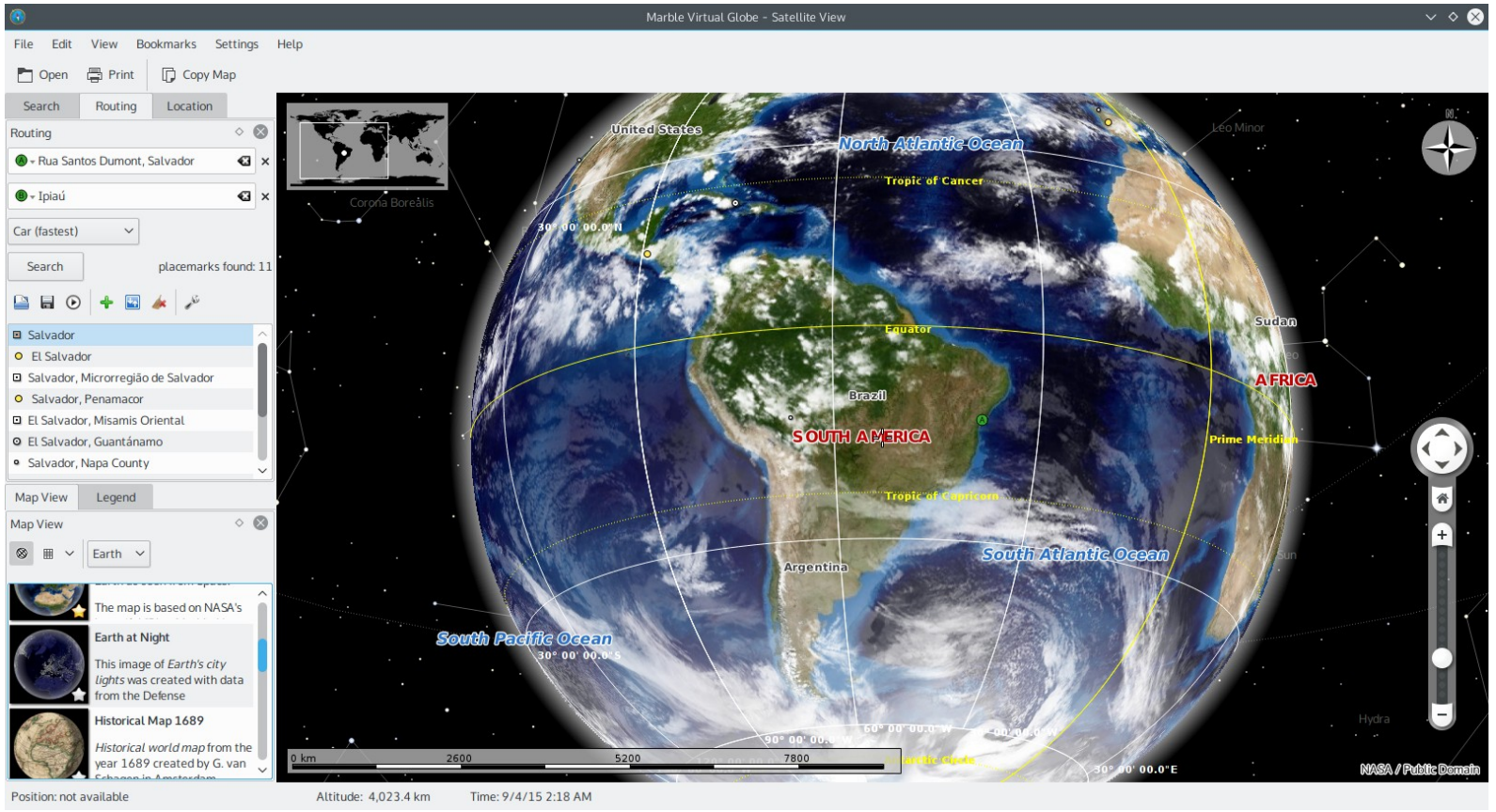
Visão Geral do Qt



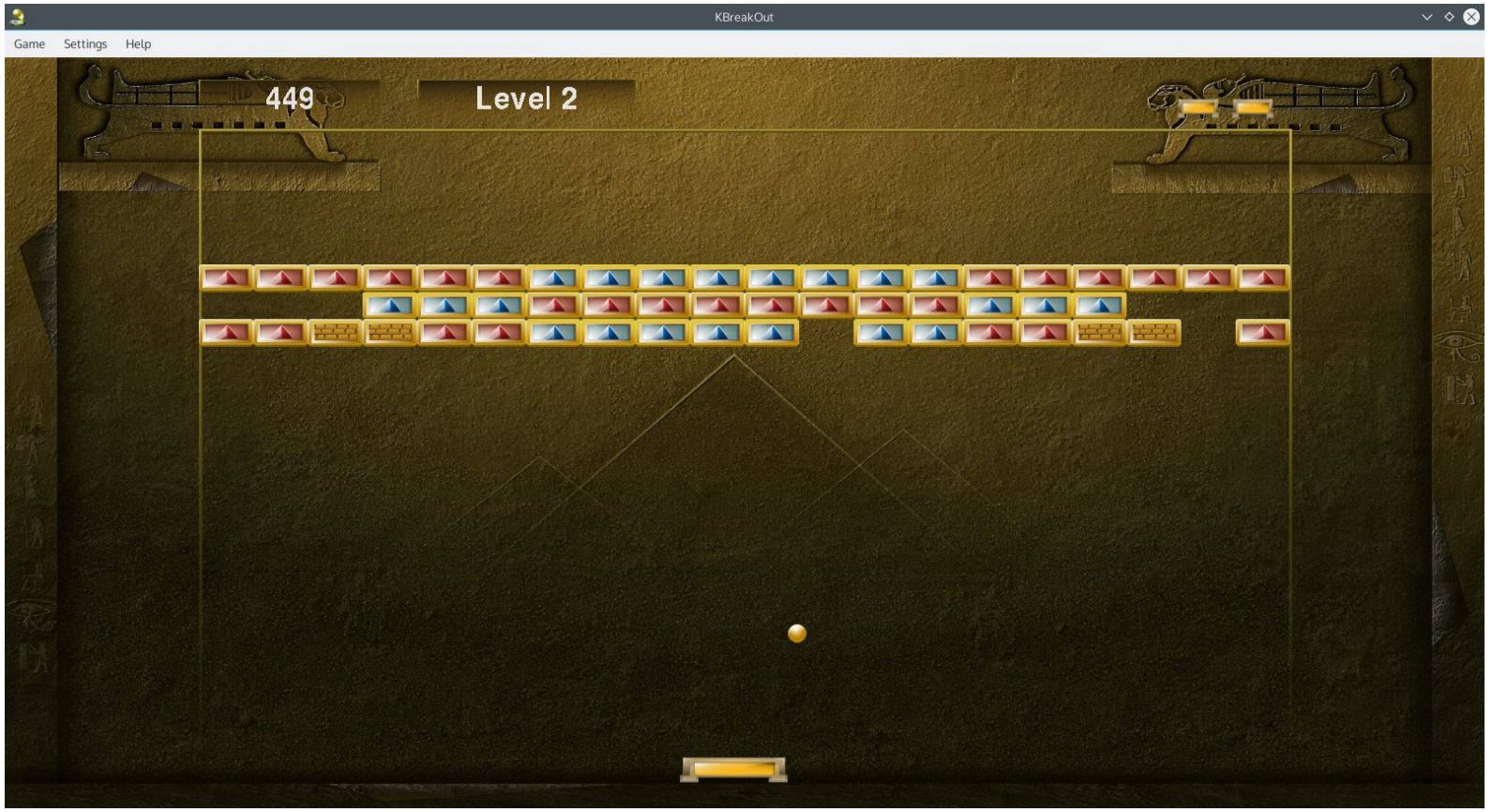
Visão Geral do Qt



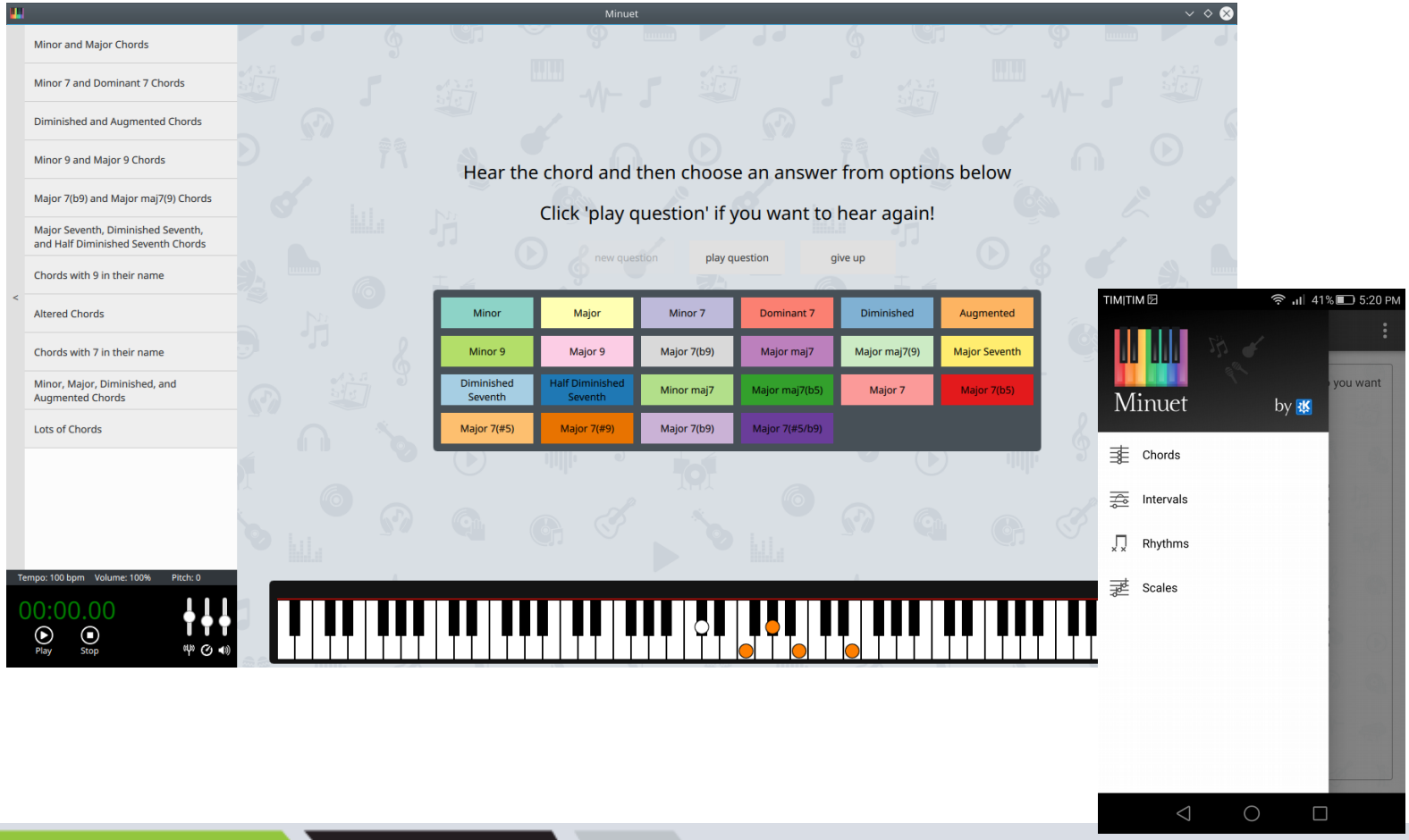
Visão Geral do Qt



Visão Geral do Qt



Visão Geral do Qt



Visão Geral do Qt

- Plataformas suportadas:
  - Desktop:
    - Windows, Linux/X11, OS X
  - Embarcadas:
    - Embedded Android, Embedded Linux, Windows Embedded, QNX, VxWorks
  - Mobile:
    - Android, iOS, BlackBerry 10, Sailfish OS
    - WinRT, Tizen (wip)

Visão Geral do Qt

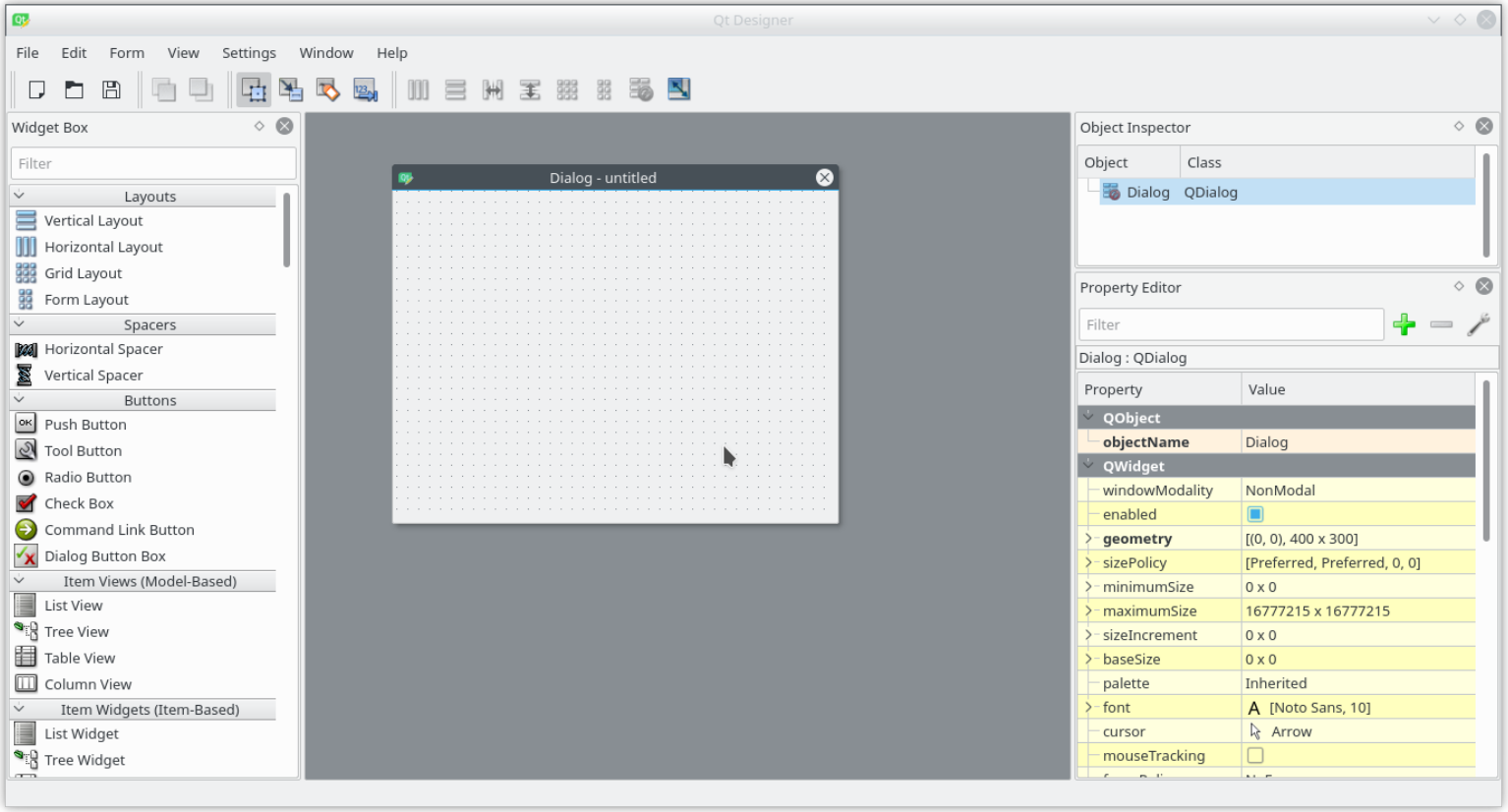
- Módulos Essenciais:
  - Qt Core
  - Qt GUI
  - Qt Multimedia e Qt Multimedia Widgets
  - Qt Network
  - Qt QML, Qt Quick, Qt Quick Controls, Qt Quick Layouts
  - Qt SQL
  - Qt Test
  - Qt Widgets

Visão Geral do Qt

- Módulos Add-on:
  - Qt 3D
  - Qt Bluetooth e Qt NFC
  - Qt Concurrent
  - Qt D-Bus
  - Qt Location e Qt Positioning
  - Qt Purchasing
  - Qt SerialPort
  - Qt WebView ... e muitos outros (inqlude.org)

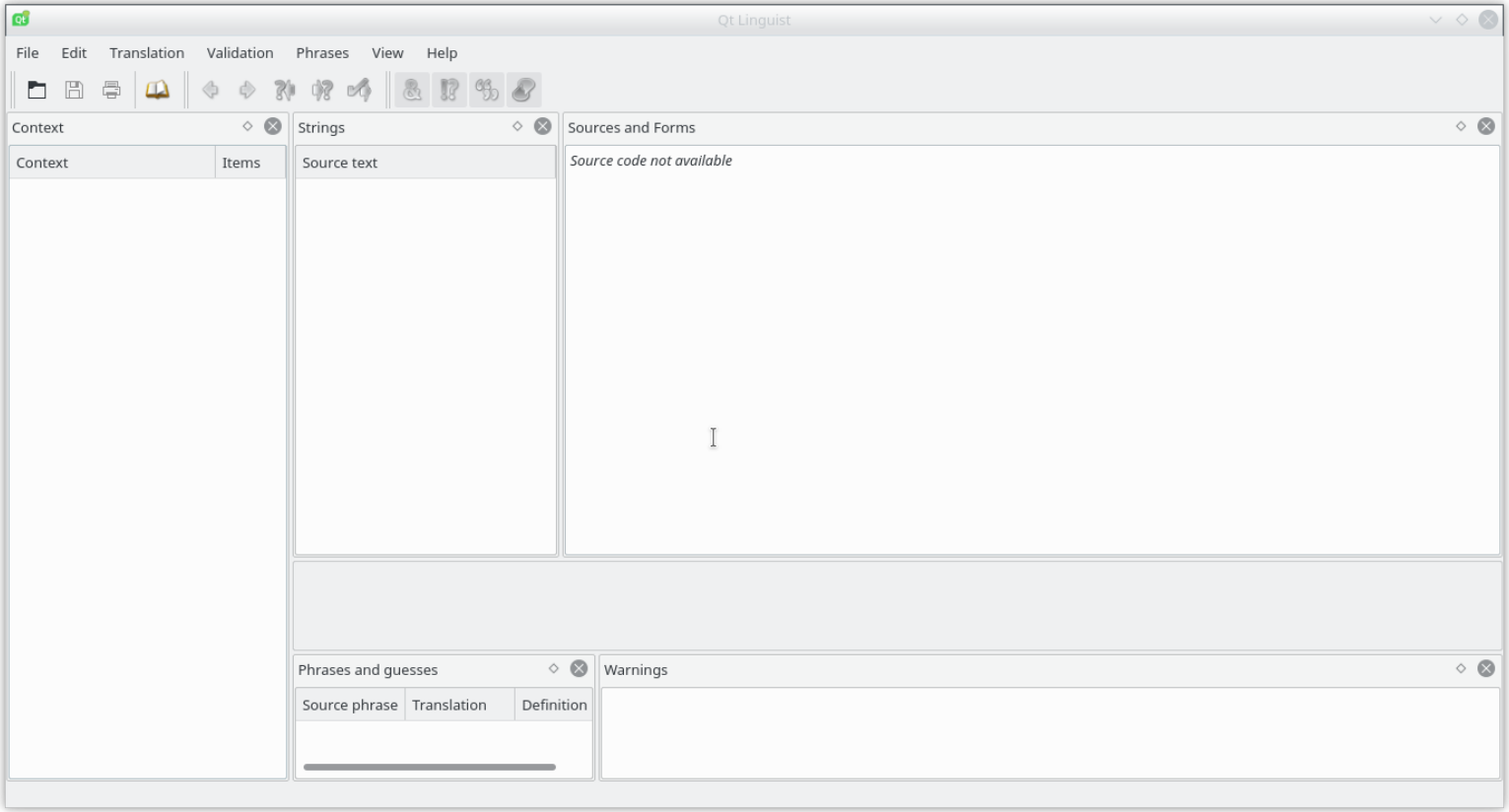
Visão Geral do Qt

- Ferramentas – Qt Design



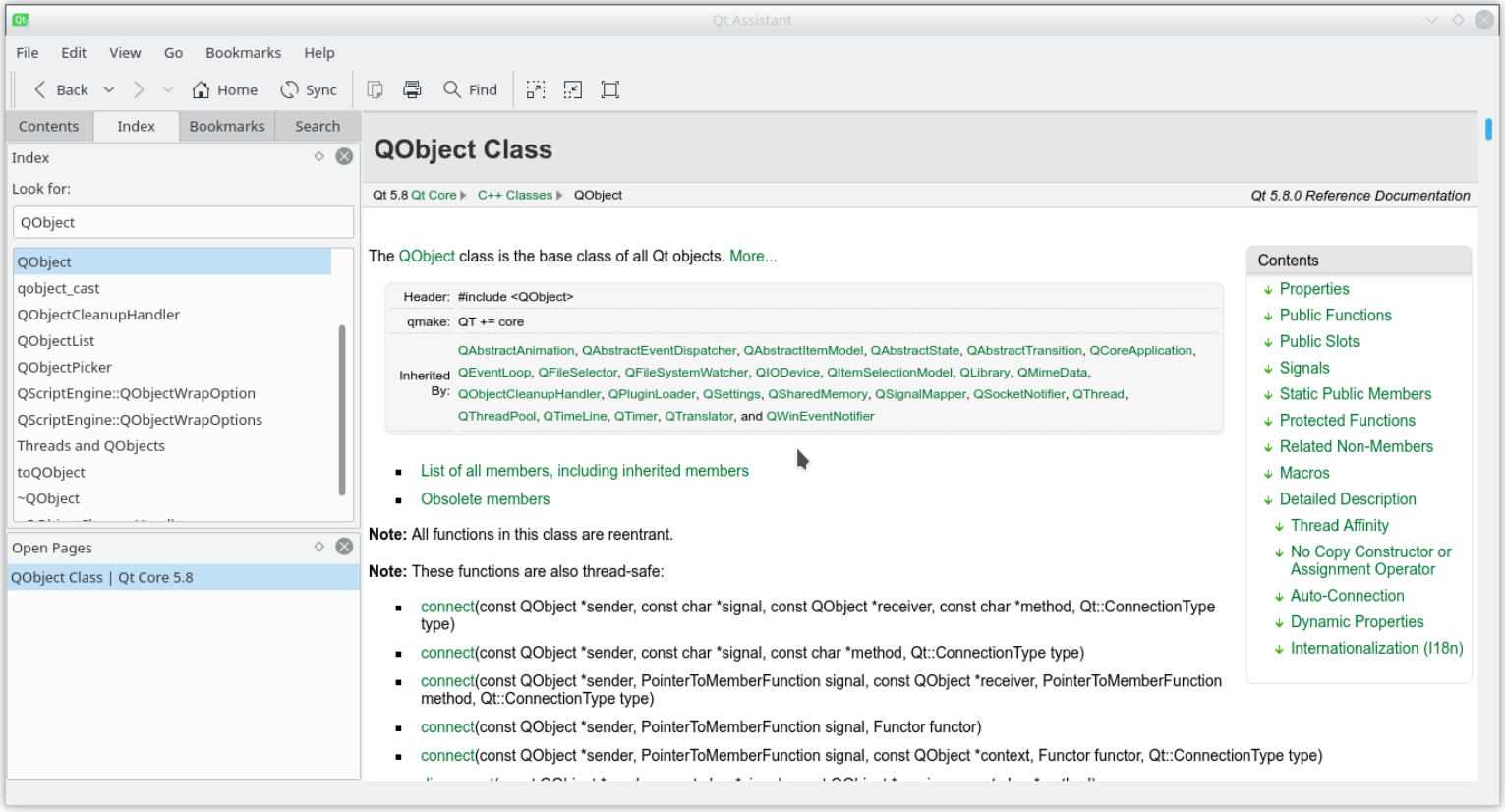
Visão Geral do Qt

- Ferramentas – Qt Linguist



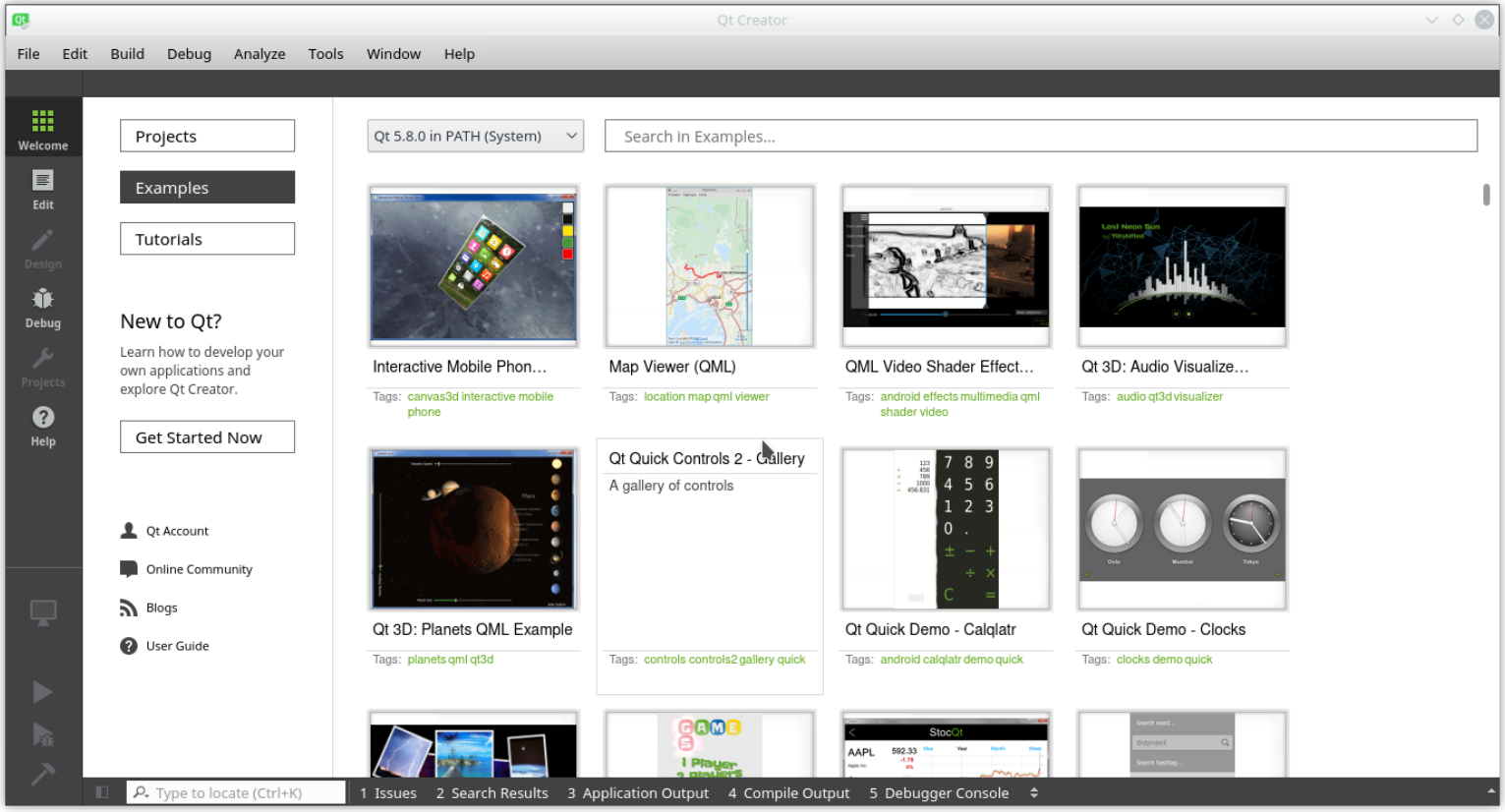
Visão Geral do Qt

- Ferramentas – Qt Assistant



Visão Geral do Qt

- Ferramentas – Qt Creator



Visão Geral do Qt

- Ferramentas – outras:
  - qmake
  - moc: meta-object compiler
  - uic: user interface compiler
  - rcc: resource compiler
  - qdbusxml2cpp: conversor de interfaces dbus para c++

Visão Geral do Qt

- O Qt oferece três tecnologias principais para desenvolvimento de GUI:
  - Qt Widgets: C++ (oficial), Python, C#, Go, Haskell, Ruby
  - Qt Quick: QML + JavaScript
  - Qt Webkit: HTML + CSS + JavaScript

Visão Geral do Qt

- Comparação entre as três tecnologias para GUI:

	Qt Widgets	Qt Quick	Qt Webkit
Linguagens utilizadas	C++	QML/JS	HTML/CSS/JS
Look'n'feel nativo	✔	✔	
Look'n'feel customizado		✔	(✔)
GUI fluidas e animadas		✔	
Suporte a touch screen		✔	
Widgets padrao da industria	✔		

Visão Geral do Qt

- Comparação entre as três tecnologias para GUI:

	Qt Widgets	Qt Quick	Qt Webkit
Model/View programming	✔	(✔)	
Rapid UI development		✔	(✔)
Aceleracao por hardware		✔	
Efeitos graficos		✔	
Processamento de rich text	✔	✔	
Integracao de conteudo web existente			✔

Lab 1

- Implementação acompanhada de um navegador web simples

O Loop de Eventos

- Função main típica de um programa em Qt

main.cpp

```
1. int main(int argc, char *argv[])
2. {
3.   QApplication app(argc, argv);
4.   QLabel *label = new QLabel("Hello Qt!");
5.   label->show();
6.   int r = app.exec();
7.   delete label;
8.   return r;
9. }
```

O Modelo de Objetos do Qt

- O Qt estende o modelo de objetos do C++ com as seguintes funcionalidades:
  - Signals / Slots: mecanismo desacoplado de comunicação
  - Object properties: atributos dinâmicos
  - Meta-Objects: para operações RTTI e de Introspecção
  - Eventos, filtros de eventos e timers
  - Tradução contextual de strings para internacionalização
  - Object trees (composite)

O Modelo de Objetos do Qt

- Signals e slots:
  - Representam um mecanismo central do Qt
  - Um signal é uma mensagem que está presente em uma classe como uma declaração de uma função-membro void. Signals não são invocados, mas emitidos (via emit) por um objeto da classe
  - Um slot é uma função-membro void e pode ser invocada normalmente

O Modelo de Objetos do Qt

- Um signal de um objeto pode ser conectado a slots de um ou mais outros objetos, desde que os parâmetros sejam compatíveis
  - Sintaxe de conexão (Qt4 – ainda válida):

```
1. connect(sender, SIGNAL(valueChanged(QString,QString)),
2.         receiver, SLOT(updateValue(QString)));
```

O Modelo de Objetos do Qt

- Um signal de um objeto pode ser conectado a slots de um ou mais outros objetos, desde que os parâmetros sejam compatíveis
  - Sintaxe de conexão (Qt5):

```
1. connect(sender, &Sender::valueChanged,
2.         receiver, &Receiver::updateValue);
```

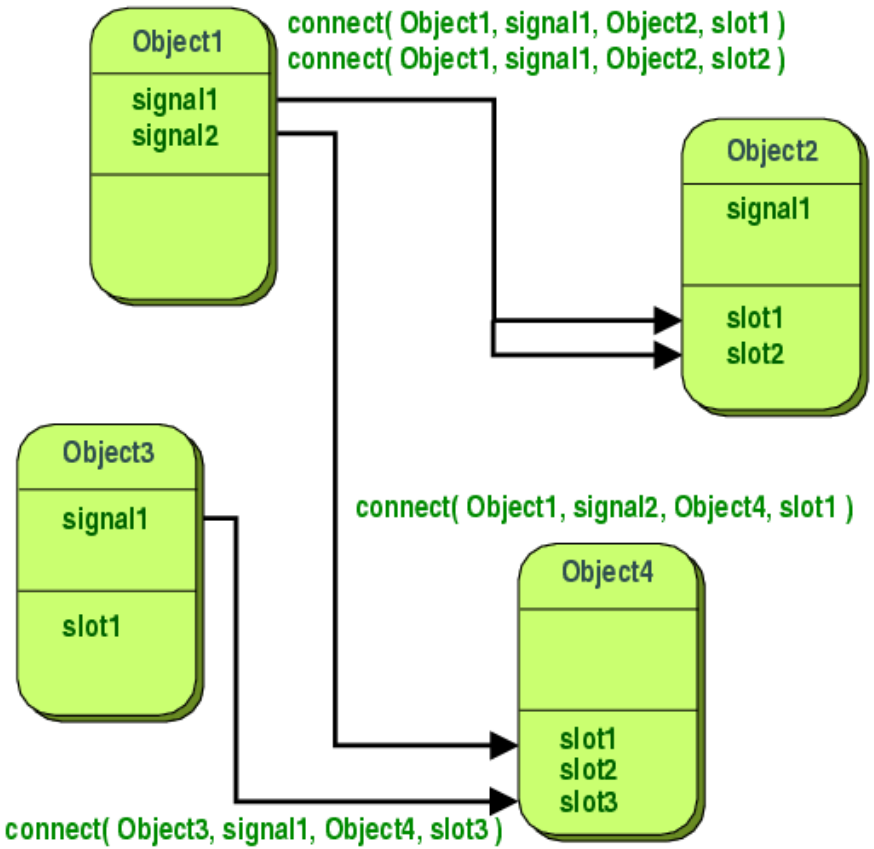
O Modelo de Objetos do Qt

- Um signal de um objeto pode ser conectado a slots de um ou mais outros objetos, desde que os parâmetros sejam compatíveis
  - Sintaxe de conexão (com as lambda expressions do c++11):

```
1. connect(sender, &Sender::valueChanged, [=](const QString &newValue) {
2.         receiver->updateValue("senderValue", newValue);
3.     } );
```

O Modelo de Objetos do Qt

- Possibilidade de conexão entre signals e slots



Lab 2

- Criando classes com signals e slots

counter.h/counter.cpp

```
1. #include <QObject>
2. class Counter : public QObject {
3.   Q_OBJECT
4. public:
5.   Counter() { m_value = 0; }
6.   int value() const { return m_value; }
7. public slots:                                    // public Q_SLOTS
8.   void setValue(int value) {
9.     if (value != m_value) {
10.      m_value = value;
11.      emit valueChanged(value);
12.    }
     }
13.signals:                                         // ou Q_SIGNALS
14.  void valueChanged(int newValue);
15.private:
16.  int m_value;
17.};
```

Lab 2

- Criando classes com signals e slots

main.cpp

```
1. Counter a, b;
2. QObject::connect(&a, &Counter::valueChanged,
3.                  &b, &Counter::setValue);
4.
5. a.setValue(12);     // a.value() == 12, b.value() == 12
6. qDebug() << “Valor de b: “ << b.value();
7. b.setValue(48);     // a.value() == 12, b.value() == 48
8. qDebug() << “Valor de a: “ << a.value();
```

Lab 2

- Considerações importantes:
  - Slots são funções comuns do C++: podem ser invocadas diretamente, sobrecarregadas, públicas ou privadas
  - Um signal pode ser conectado a vários slots
  - Mais de um signal pode ser conectado ao mesmo slot (o emissor pode ser descoberto com QObject::sender);
  - Um signal pode ser conectado a outro signal

Lab 2

- Considerações importantes:
  - Conexões podem ser removidas com QObject::disconnect
  - Um signal pode ter um número de parâmetros maior ou igual ao número de parâmetros do slot conectado
  - Signals e slots podem ser utilizados em qualquer classe derivada de QObject, não somente widgets

Lab 2

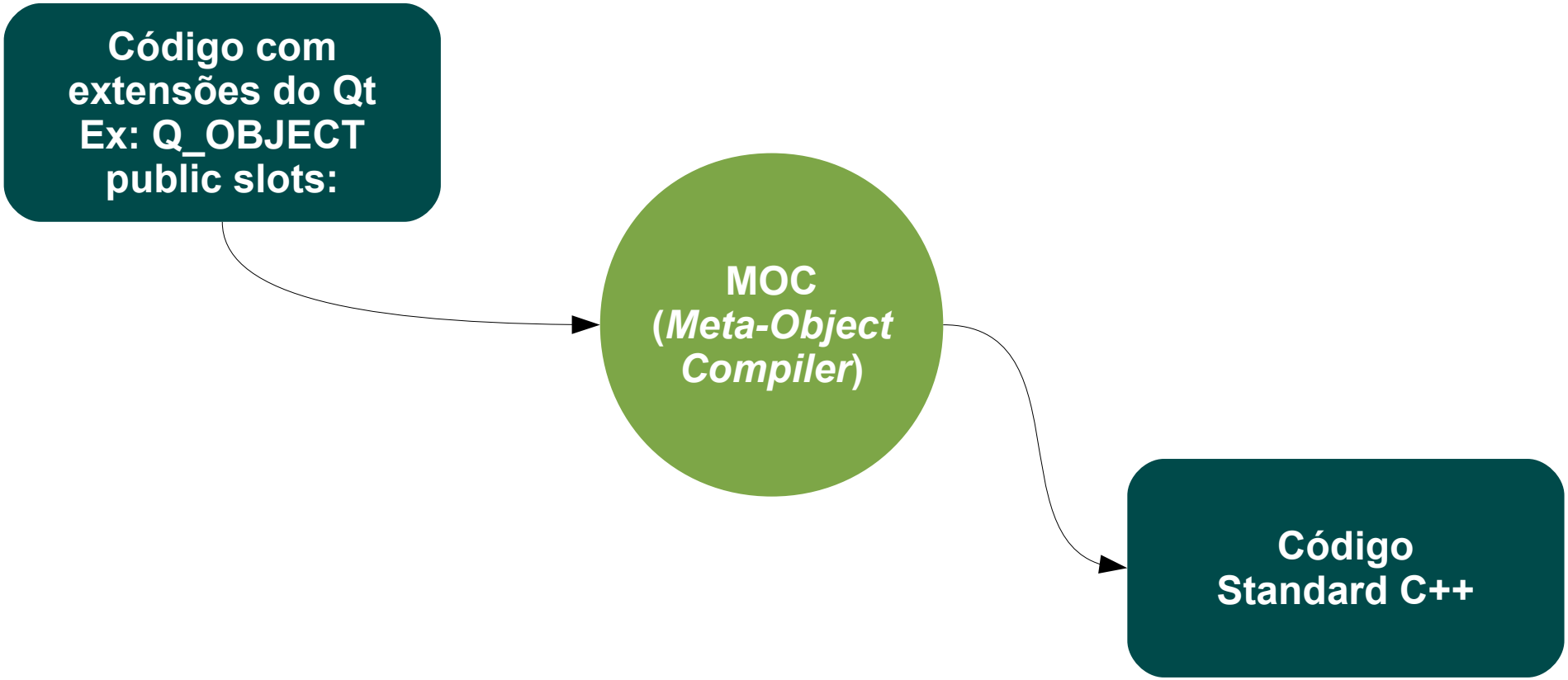
- Considerações importantes:
  - Conexões, em QDialogs, podem ser automaticamente realizadas (sem requerer o QObject::connect)
  - Se as palavras reservadas signals, slots e emit estiverem sendo utilizadas por outra biblioteca (ex. boost) pode-se desabilitá-las e usar as macros Q_SIGNALS, Q_SLOTS e Q_EMIT
  - O método connect possui um quinto parâmetro

O Meta-Object Compiler (MOC)

- Como signals e slots são implementados ?
  - As palavras reservadas signals, slots e emit (dentre outros recursos) não estão presentes no Standard C++
  - Essas extensões são tratadas pelo MOC
  - O qmake verifica quais classes, declaradas na variável HEADER, utilizam a macro Q_OBJECT e automaticamente inclui a invocação do moc no arquivos Makefile gerados

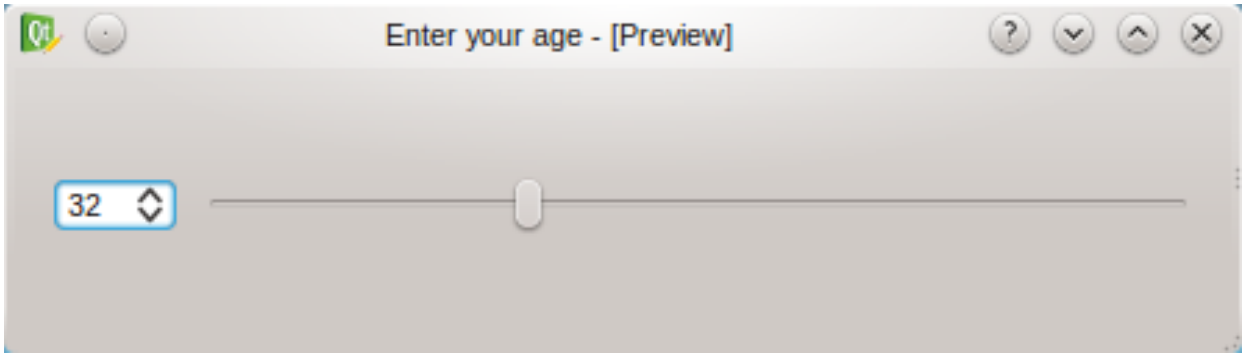
O Meta-Object Compiler (MOC)

- Como signals e slots são implementados ?



Layout e Parentesco

- Layouts gerenciam a geometria dos widgets de uma janela
- Um widget pode ter uma relação de parentesco com outro widget
- Exemplo:



Layout e Parentesco

- Tipos de layout:
  - QHBoxLayout
  - QVBoxLayout

QGridLayout

QFormLayout

- Layouts dinâmicos e em fluxo

Layout e Parentesco

- Exemplo de layout e parentesco:

```
1. #include <QApplication>
2. #include <QHBoxLayout>
3. #include <QSlider>
4. #include <QSpinBox>
5.
6. int main(int argc, char *argv[])
7. {
8.   QApplication app(argc, argv);
9.   QWidget *window = new QWidget;
10.  window->setWindowTitle
                ("Enter Your Age");
11.  QSpinBox *spinBox =
                     new QSpinBox;
12.  QSlider *slider =
        new QSlider(Qt::Horizontal);
13.  spinBox->setRange(0, 130);
14.  slider->setRange(0, 130);
15.  QObject::connect(spinBox,
     SIGNAL(valueChanged(int)),
     slider,
     SLOT(setValue(int)));
16.  QObject::connect(slider,
     SIGNAL(valueChanged(int)),
     spinBox,
     SLOT(setValue(int)));
17.  spinBox->setValue(35);
18.  QHBoxLayout *layout =
       new QHBoxLayout;
19.  layout->addWidget(spinBox);
20.  layout->addWidget(slider);
21.  window->setLayout(layout);
22.  window->show();
23.  int r = app.exec();
24.  delete window;
25.  return r;
26.}
```

Layout e Parentesco

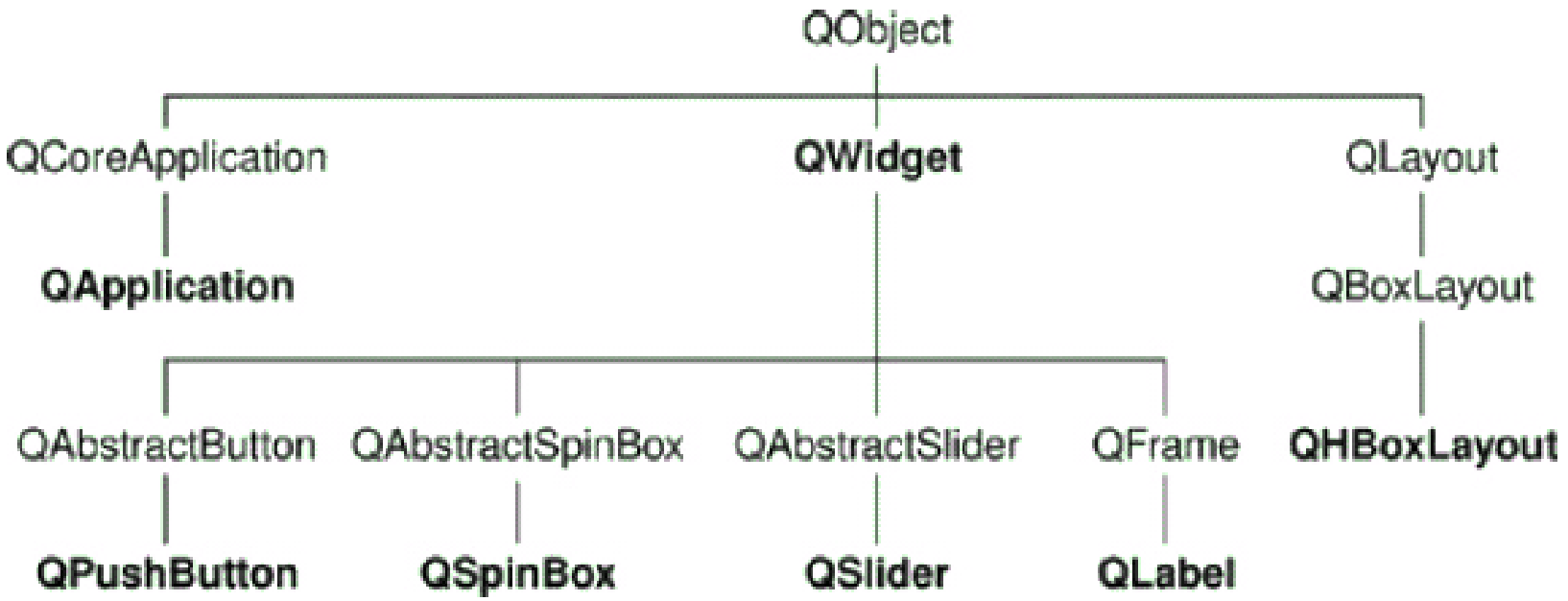
- Considerações sobre relações de parentesco:
  - O pai automaticamente assume a responsabilidade de liberação de memória de todos os filhos
  - Os filhos são liberados quando o pai sair do escopo
  - Os únicos objetos a serem deletados manualmente são os criados com new e que não possuem pai

Layout e Parentesco

- Considerações sobre relações de parentesco:
  - Se um filho é deletado antes do pai ele é automaticamente excluído da lista de filhos do pai
  - Widgets filhos são vistos dentro da área do pai
  - A execução do show() no pai automaticamente exibe todos os filhos

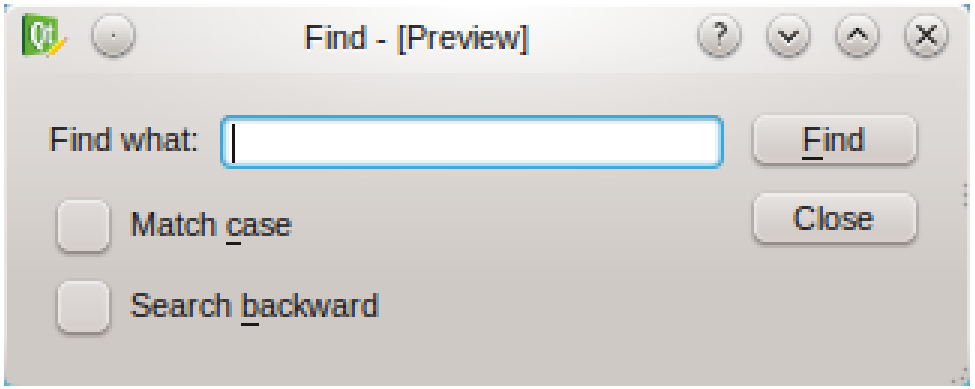
Layout e Parentesco

- Hierarquia de classes utilizadas:



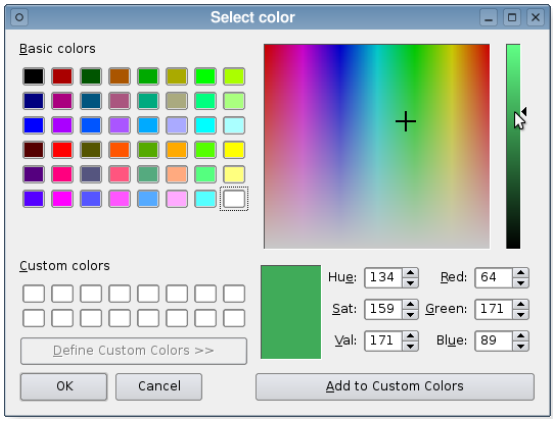
MainWindow e Dialogs

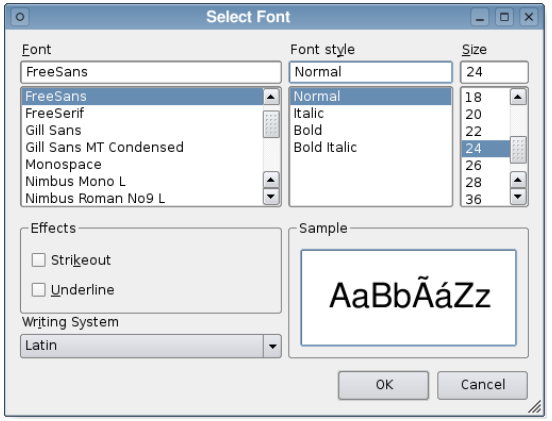
- Uma aplicação é geralmente formada por uma tela principal (QMainWindow) e por várias outras telas (QDialog)
- Essas telas podem ser criadas manualmente ou através do Qt Designer
- Exemplo (Dialog):

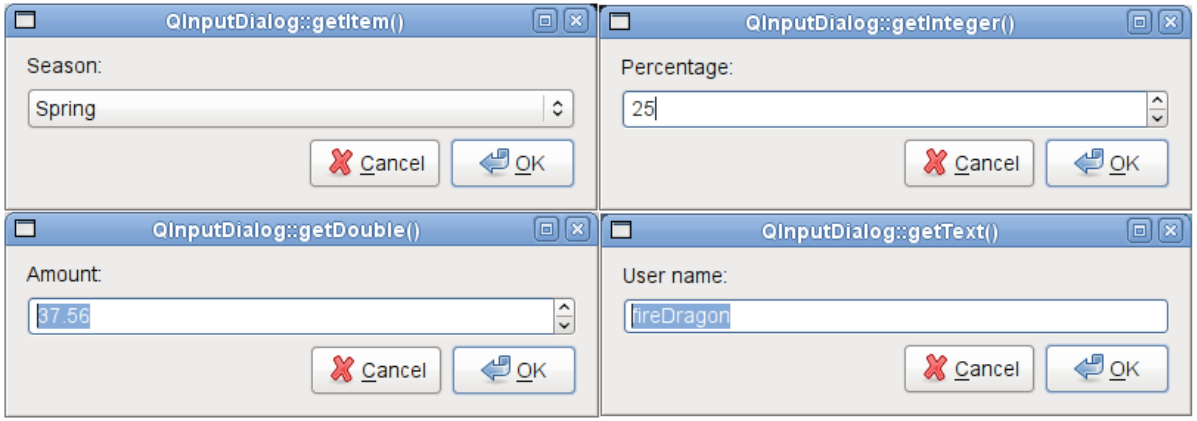


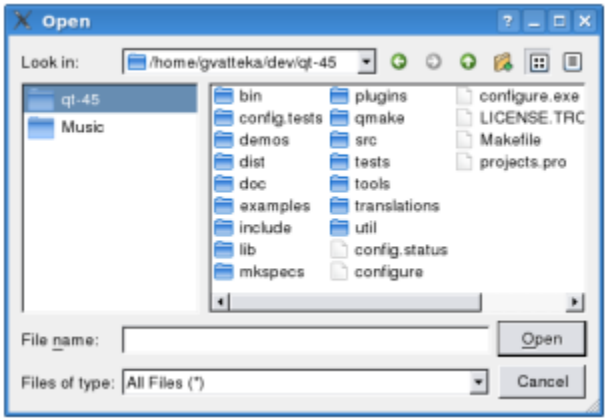
MainWindow e Dialogs

- Dialogs pré-existentes no Qt:



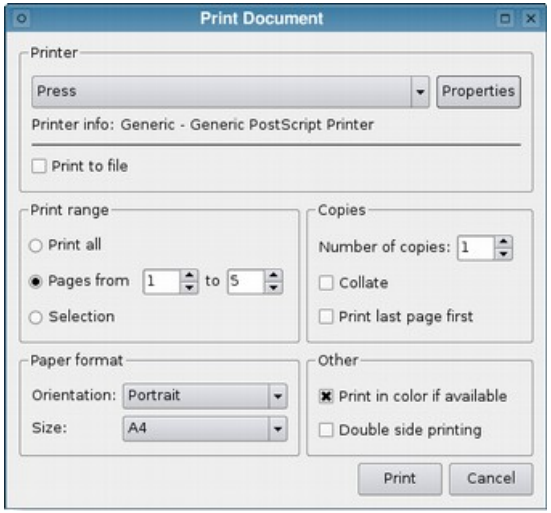


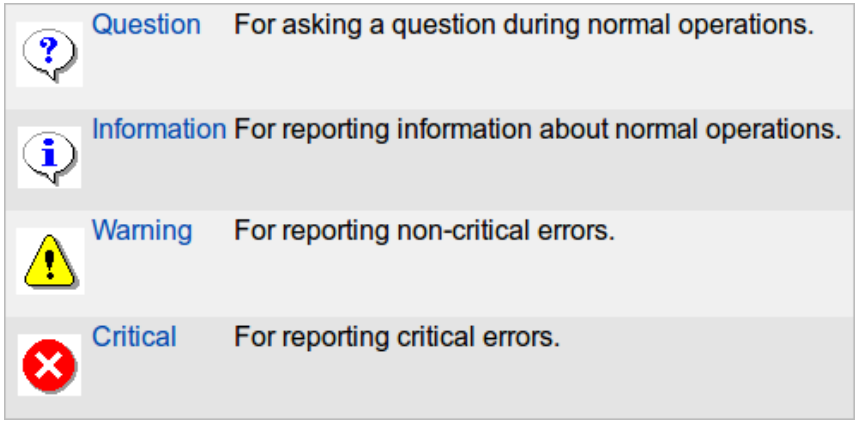




MainWindow e Dialogs

- Dialogs pré-existentes no Qt:



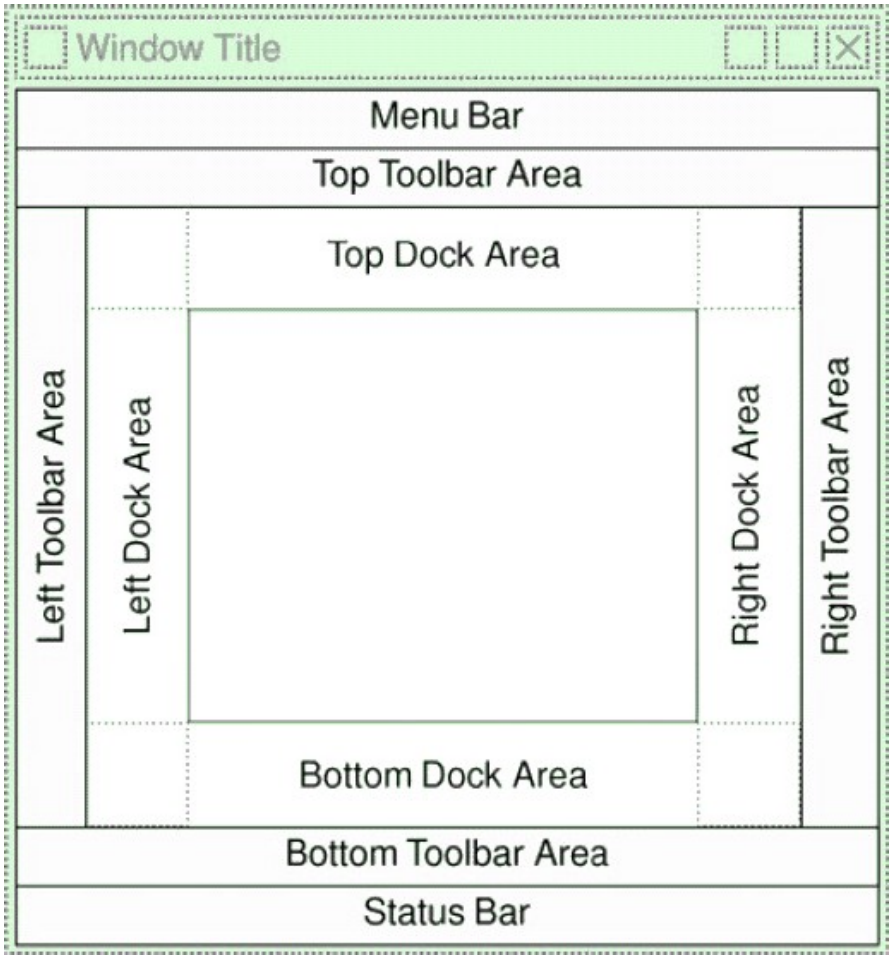


MainWindow e Dialogs

- Main Windows são widgets que podem conter menus, toolbars e barra de status
- Main Windows são criadas através da derivação da classe QMainWindow
- O Qt Designer facilita sobremaneira a criação de Main Windows e Dialogs

MainWindow e Dialogs

- Áreas definidas pela QMainWindow:



MainWindow e Dialogs

- Menus e toolbars baseiam-se no conceito de action:
  - Um action é um item que pode ser adicionado a qualquer número de menus e toolbars
- Criar menus e toolbars requer três passos:
  - Criação e configuração dos actions
  - Criação do menu e utilização dos actions
  - Criação das toolbars e utilização dos actions
- Todo action possui o signal triggered()

MainWindow e Dialogs

- Criando action, menus e toolbars:

```
1. // Criando um action
2. QAction *newAction = new QAction(tr("&New"), this);
3. newAction->setIcon(QIcon(":/images/new.png"));
4. newAction->setShortcut(tr("Ctrl+N"));
5. newAction->setStatusTip(tr("Create a new spreadsheet file"));
6. connect(newAction, &QAction::triggered, this, &MainWindow::newFile);
7.
8. // Inserindo actions em menus
9. QMenu *fileMenu = menuBar()->addMenu(tr("&File"));
10.fileMenu->addAction(newAction);
11.
12.// Inserindo actions em toolbars
13.QToolBar *fileToolBar = addToolBar(tr("&File"));
14.fileToolBar->addAction(newAction);
```

O Sistema de Resources do Qt

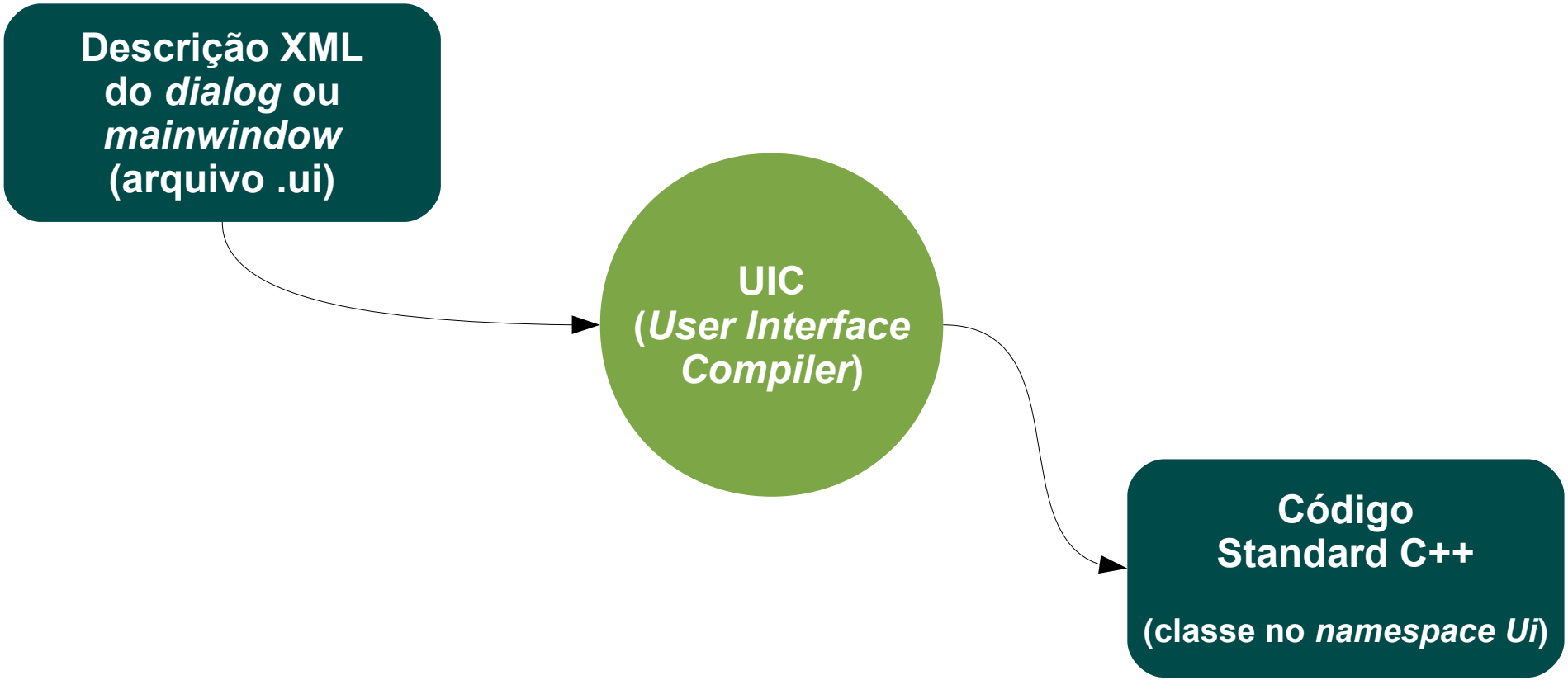
- As imagens utilizadas pela aplicação (ícones etc) podem ser carregadas de três formas:
  - Através da leitura, em tempo de execução, dos arquivos das imagens
  - Através da inclusão de imagens XPM no código-fonte
  - Através do uso do mecanismo de resources do Qt

O Sistema de Resources do Qt

- Resources:
  - O arquivo de recursos de uma aplicação Qt indica quais arquivos serão diretamente incluídos no executável a ser gerado. Dessa forma, ícones não são perdidos
  - Na verdade, qualquer tipo de arquivo (não somente imagens) pode ser incluído no executável
  - Arquivos de recursos são organizados em categorias
  - Recursos são utilizados com o prefixo ":/"

O User Interface Compiler (UIC)

- Entendo o Qt Design:



Model-View Framework

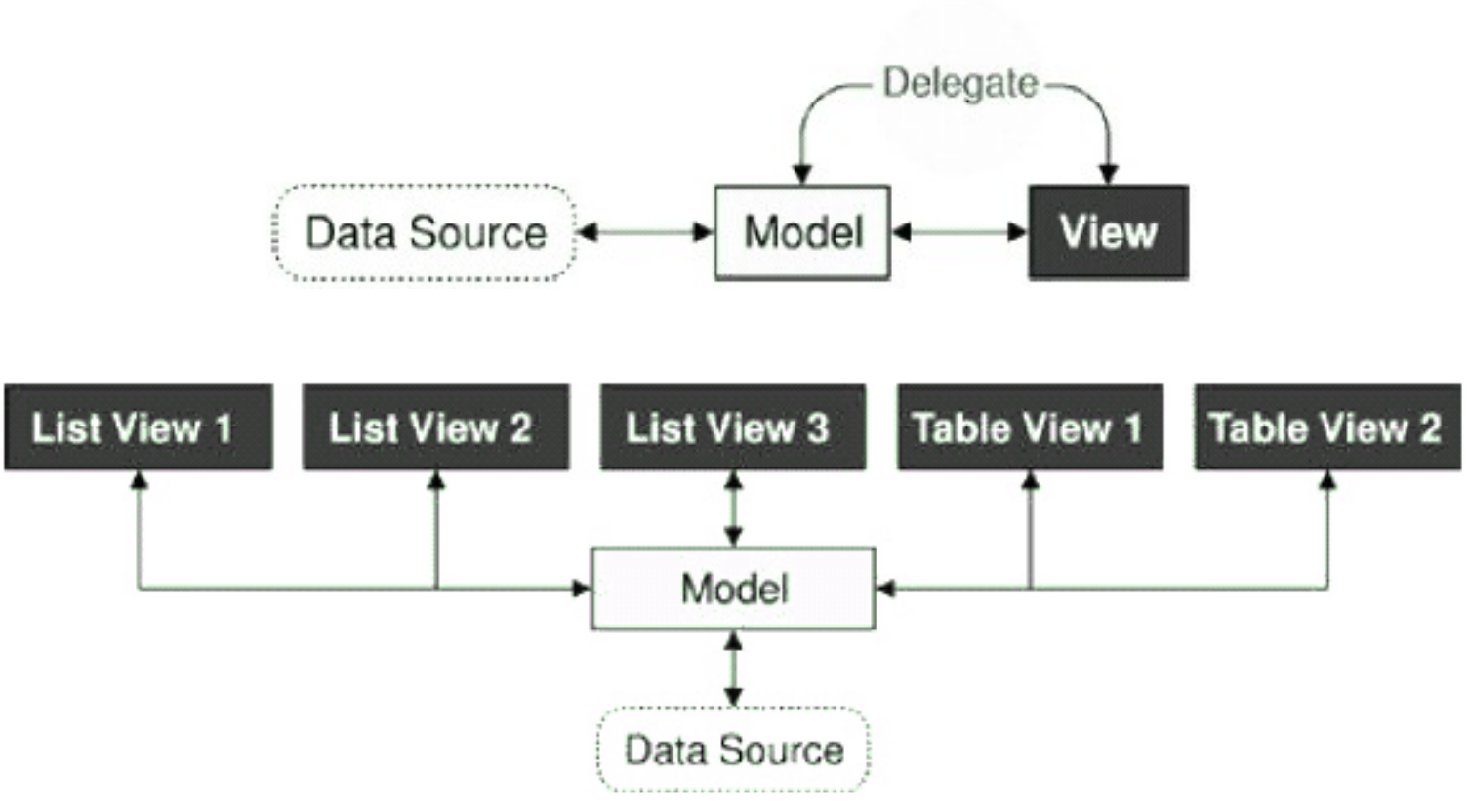
- Muitas aplicações permitem a busca, visualização e edição de itens individuais de um conjunto de dados
- Em versões anteriores do Qt, os widgets eram populados com todo o conteúdo do conjunto de dados
- Entretanto, esta abordagem não é escalável para grandes conjuntos e não resolve o problema da visualização múltipla do mesmo conjunto

Model-View Framework

- O Qt usa o padrão model-view-delegate, uma variação do MVC, para resolver esses problemas:
  - Model: representa o conjunto de dados e é responsável pela aquisição e atualização de dados na origem. Cada tipo de dados tem seu próprio modelo, porém a API provida às views é padrão
  - View: apresenta os dados para o usuário (com lazy-load)
  - Delegate: permite a configuração da forma de exibição e entrada de dados

Model-View Framework

- O Qt usa o padrão model-view-delegate, uma variação do MVC, para resolver esses problemas:



Model-View Framework

- Model view no Qt:
  - O delegate é usado para ter total controle sobre como os itens são apresentados e editados
  - O Qt disponibiliza um delegate padrão para cada tipo de view
  - Os modelos podem obter somente os dados que são realmente necessários
  - Cada item de dados em um modelo é representado por um index
  - O Qt traz um conjunto de modelos e outros podem ser criados

Model-View Framework

- Os widgets são divididos em dois grupos:
  - As classes QListWidget, QTableWidget e QTreeWidget devem ser utilizadas para apresentar poucos itens
  - Para grandes conjuntos devem ser utilizadas as classes QListView, QTableView e QTreeView, em conjunto com um modelo de dados (do Qt ou customizado)

Model-View Framework

- Modelos predefinidos do Qt:
  - QStringListModel: armazena uma lista de strings
  - QStandardItemModel: armazena dados hierárquicos arbitrários
  - QFileSystemModel: encapsula o sistema de arquivos local
  - QSqlQueryModel: encapsula um resultset SQL
  - QSqlTableModel: encapsula uma tabela SQL
  - QSqlRelationalTableModel: encapsula uma tabela SQL com chaves estrangeiras
  - QSortFilterProxyModel: ordena ou filtra outro modelo

Lab 3

- Visualizando o sistema de arquivos:

```
1. QFileSystemModel *model = new QFileSystemModel;
2. model->setRootPath(QDir::currentPath());
3.
4. QTreeView *tree = new QTreeView(splitter);
5. tree->setModel(model);
6. tree->setRootIndex(model->index(QDir::currentPath()));
7.
8. QListView *list = new QListView(splitter);
9. list->setModel(model);
10.list->setRootIndex(model->index(QDir::currentPath()));
```

Containers do Qt

- Classes template de propósito geral para armazenamento de outros objetos
- São implicitamente compartilhados, reentrantes, com bom desempenho, baixo consumo de memória e thread-safe
- Possuem dois tipos de iterators:
  - Java-style: fáceis de usar
  - STL-style: ligeiramente mais eficientes
- foreach

Containers do Qt

- São implicitamente compartilhados
- Podem ser serializados com o QDataStream
- Resultam em menos código executável que os correspondentes da Standard C++ Library
  - Containers sequenciais: QList, QLinkedList, QVector, QStack e QQueue
  - Containers associativos: QMap, QMultiMap, QHash, QMultiHash e QSet
  - Algoritmos genéricos. Ex: qSort(), qBinaryFind(), qSwap()

Containers do Qt

- Visão Geral:
  - QList:
    - Implementada usando arrays, acesso rápido por índices, expansão mínima de código no executável
  - QLinkedList:
    - Implementada como lista encadeada, melhor desempenho para inserções no meio, melhor semântica de iterators
  - QVector:
    - Implementado usando arrays, inserções no início e no meio são custosas
  - QStack:
    - Sub-classe conveniente de QVector com semântica LIFO

Containers do Qt

- Visão Geral:
  - QQueue:
    - Sub-classe conveniente de QList com semântica FIFO
  - QSet:
    - Representa conjuntos matemática com acessos rápidos
  - Q[Multi]Map:
    - Dicionário que mapeia chaves em valores, armazena os dados ordenados pela chave
  - Q[Multi]Hash:
    - Armazena os dados em ordem arbitrário, acesso mais rápido que o Q[Multi]Map

Containers do Qt

- Iterators:

Containers	Read-only iterator	Read-write iterator
QList<T>, QQueue<T>	QListIterator<T>	QMutableListIterator<T>
QLinkedList<T>	QLinkedListIterator<T>	QMutableLinkedListIterator<T>
QVector<T>, QStack<T>	QVectorIterator<T>	QMutableVectorIterator<T>
QSet<T>	QSetIterator<T>	QMutableSetIterator<T>
QMap<Key, T>, QMultiMap<Key, T>	QMapIterator<Key, T>	QMutableMapIterator<Key, T>
QHash<Key, T>, QMultiHash<Key, T>	QHashIterator<Key, T>	QMutableHashIterator<Key, T>

Containers do Qt

- Iterators (Java-style):

```
1. QList<QString> list;
2. list << "A" << "B" << "C" << "D";
3.
4. QListIterator<QString> i(list);
5. while (i.hasNext())
6.   qDebug() << i.next();
7.
8.
9. QListIterator<QString> i(list);
10.i.toBack();
11.while (i.hasPrevious())
12.  qDebug() << i.previous();
```

Containers do Qt

- Iterators com suporte a modificações:

```
1. QMutableListIterator<int> i(list);
2. while (i.hasNext()) {
3.   if (i.next() % 2 != 0)
4.     i.remove();
5. }
6.
7.
8. QMutableListIterator<int> i(list);
9. while (i.hasNext()) {
10.  if (i.next() > 128)
11.    i.setValue(128);
12.}
```

Containers do Qt

- Iterators (STL-style):

```
1. QList<QString> list;
2. list << "A" << "B" << "C" << "D";
3.
4. QList<QString>::iterator i;
5. for (i = list.begin(); i != list.end(); ++i)
6.   *i = (*i).toLower();
```

Containers do Qt

- foreach

```
1. QLinkedList<QString> list;
2. foreach (QString str, list) {
3.   if (str.isEmpty())
4.     break;
5.   qDebug() << str;
6. }
7.
8. QMap<QString, int> map;
9. foreach (QString str, map.keys())
10.  qDebug() << str << ":" << map.value(str);
11.
12.QMultiMap<QString, int> map;
13.foreach (QString str, map.uniqueKeys()) {
14.  foreach (int i, map.values(str))
15.    qDebug() << str << ":" << i;
16.}
```

Containers do Qt

- Complexidades

	Index lookup	Insertion	Prepending	Appending
QLinkedList<T>	O(*n*)	O(1)	O(1)	O(1)
QList<T>	O(1)	O(n)	Amort. O(1)	Amort. O(1)
QVector<T>	O(1)	O(n)	O(n)	Amort. O(1)

	Key lookup		Insertion	
	Average	Worst case	Average	Worst case
QMap<Key, T>	O(log *n*)	O(log *n*)	O(log *n*)	O(log *n*)
QMultiMap<Key, T>	O(log *n*)	O(log *n*)	O(log *n*)	O(log *n*)
QHash<Key, T>	Amort. O(1)	O(*n*)	Amort. O(1)	O(*n*)
QSet<Key>	Amort. O(1)	O(*n*)	Amort. O(1)	O(*n*)

Acessando Banco de Dados

- O módulo QtSql disponibiliza uma interface independente de plataforma e de banco de dados
- Essa interface é suportada por um conjunto de classes que usam a arquitetura Model View do Qt
- Uma conexão de banco é representada por uma instância da classe QSqlDatabase

Acessando Banco de Dados

- Drivers disponíveis:
  - QMYSQL: MySQL >= 4
  - QOCI: Oracle Call Interface
  - QODBC: Open Database Connectivity
  - QPSQL: PostgreSQL >= 7.3
  - QTDS: Sybase Adaptive Server
  - QDB2: IBM DB2 >= 7.1
  - QSQLITE2: SQLite v2
  - QSQLITE: SQLite >= 3
  - QIBASE: Borland Interbase

Acessando Banco de Dados

- O banco pode ser utilizado de duas formas:
  - Com a classe QSqlQuery é possível a execução direta de sentenças SQL
  - Com classes de mais alto nível como QSqlTableModel e QsqlRelationalTableModel
- As classes de mais alto nível podem ser anexadas a widgets para visualização automática dos dados do banco
- O Qt torna fácil a programação de recursos como master-detail e drill-down

Acessando Banco de Dados

- Criando a conexão default

```
1. inline bool createConnection()
2. {
3.     QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
4.     db.setDatabaseName("cddatabase.db");
5.     if (!db.open())
6.     {
7.         QMessageBox::warning(0, QObject::tr("Database Error"),
8.                              db.lastError().text());
9.         return false;
10.    }
11.    return true;
12.}
```

Acessando Banco de Dados

- Executando sentenças SQL

```
1.    QSqlQuery query;
2.    query.exec("SELECT title, year FROM cd WHERE year >= 1998");
3.    while (query.next())
4.    {
5.       QString title = query.value(0).toString();
6.       int year = query.value(1).toInt();
7.       qDebug() << title << ": " << year << endl;
8.    }
```

Acessando Banco de Dados

- QSqlQuery pode ser utilizado com comandos DDL e inserts

```
1. QSqlQuery query;
2.
3. query.exec("CREATE TABLE artist (id INTEGER PRIMARY KEY, "
4.             "name VARCHAR(40) NOT NULL, country VARCHAR(40))");
5.
6. query.exec("CREATE TABLE cd (id INTEGER PRIMARY KEY, "
7.             "title VARCHAR(40) NOT NULL, artistid INTEGER NOT NULL, "
8.             "year INTEGER NOT NULL, "
9.             "FOREIGN KEY (artistid) REFERENCES artist)");
10.
11.query.exec("CREATE TABLE track (id INTEGER PRIMARY KEY, "
12.            "title VARCHAR(40) NOT NULL, duration INTEGER NOT NULL, "
13.            "cdid INTEGER NOT NULL)");
```

Acessando Banco de Dados

- Consultando com QSqlTableModel

```
1.    QSqlTableModel model;
2.    model.setTable("cd");
3.    model.setFilter("year >= 1998");
4.    model.select();
5.    connect(model, SIGNAL(beforeInsert(QSqlRecord &)),
6.            this, SLOT(beforeInsertArtist(QSqlRecord &)));
7.    for (int i = 0; i < model.rowCount(); ++i)
8.    {
9.       QSqlRecord record = model.record(i);
10.      QString title = record.value("title").toString();
11.      int year = record.value("year").toInt();
12.      qDebug() << title << ": " << year << endl;
13.   }
```

Acessando Banco de Dados

- Ligando o model a uma view

```
1. model = new QSqlTableModel(this);
2. model->setTable("artist");
3. model->setSort(Artist_Name, Qt::AscendingOrder);
4. model->setHeaderData(Artist_Id, Qt::Horizontal, tr("Id"));
5. model->setHeaderData(Artist_Name, Qt::Horizontal, tr("Name"));
6. model->setHeaderData(Artist_Country, Qt::Horizontal, tr("Country"));
7. model->select();
8. artistTableView = new QTableView;
9. artistTableView->setModel(model);
10.artistTableView->setSelectionBehavior(QAbstractItemView::SelectRows);
11.artistTableView->resizeColumnsToContents();
```

Acessando Banco de Dados

- Incluindo registros:

```
1.    ...
2.    connect(model, SIGNAL(beforeInsert(QSqlRecord &)),
3.            this,  SLOT(beforeInsertArtist(QSqlRecord &)));
4.    ...
5.
6.    int row = model->rowCount();
7.    model->insertRow(row);
8.    QModelIndex index = model->index(row, Artist_Name);
9.    tableView->setCurrentIndex(index);
10.   tableView->edit(index);
11.   …
12.
13.void ArtistForm::beforeInsertArtist(QSqlRecord &record)
14.{
15.   record.setValue("id", generateId("artist"));
16.}
```

Acessando Banco de Dados

- Incluindo registros:

```
1. inline int generateId(const QString &table)
2. {
3.    QSqlQuery query;
4.    query.exec("SELECT MAX(id) FROM " + table);
5.    int id = 0;
6.    if (query.next())
7.       id = query.value(0).toInt() + 1;
8.    return id;
9. }
```

Acessando Banco de Dados

- Excluindo registros:

```
1.    tableView->setFocus();
2.    QModelIndex index = tableView->currentIndex();
3.    if (!index.isValid()) return;
4.    QSqlRecord record = model->record(index.row());
5.    QSqlTableModel cdModel;
6.    cdModel.setTable("cd");
7.    cdModel.setFilter("artistid = " + record.value("id").toString());
8.    cdModel.select();
9.    if (cdModel.rowCount() == 0) {
10.       model->removeRow(tableView->currentIndex().row());
11.   } else {
12.       QMessageBox::information(this,
13.                    tr("Delete Artist"),
14.                    tr("Cannot delete %1 because there are CDs associated "
15.                    "with this artist in the collection.")
16.                    .arg(record.value("name").toString()));
17.}
```

Lab4

- Uso da classe QSqlQuery
- Integração com o Model View
- Inclusão e Exclusão