Mobile Edition

February 2017 – Based on Qt 5.8

Contents

Objevtives

- Qt on mobile platforms
- Mobile user experience
- Native API access
- Mobile APIs
  - Sensors
  - Bluetooth
  - In-app purchasing

- Any questions at any point – please do not hesitate to ask!

Qt in Mobile Platforms

Qt in Mobile Platforms

- Objective is to support Qt-based app development in mobile platforms
  - Not to provide all mobile APIs as cross-platform APIs
  - Qt-based frameworks for mobile game and app development exist, for example V-Play https://v-play.net
  - Mobile offering may be extended with partner contributions or acquisitions in the future

- Qt mobile APIs
  - Sensors
  - Positioning
  - Location
  - NFC
  - Bluetooth
  - In-app purchasing

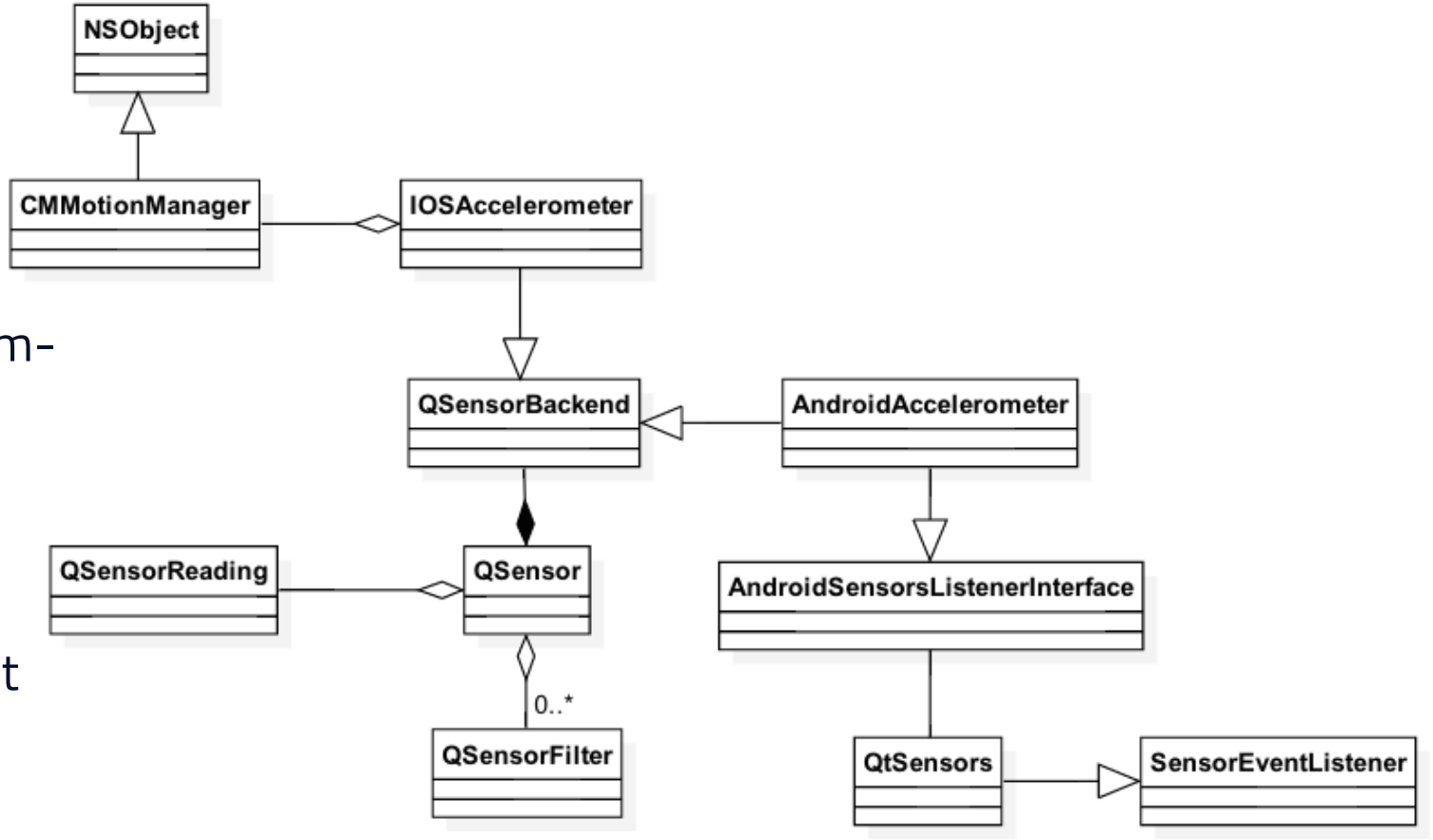
Platform-Dependent Variations

- Macro
- Line-by-line variation
- Function implementation
  - Function signature kept the same
  - Implementation differs
- Class implementation
  - Header/interface kept the same
  - The whole implementation differs
  - Separate source files:
    myclass.h, myclass_unix.cpp, myclass_win.cpp
- Plug-in
  - Usually requires a unified way to implement plug-ins

```
orientation: (Qt.platform.os === "osx" ||
    Qt.platform.os === "ios") ? -90 : 0
```

Mobile APIs – Architectures

- Backend with platform-dependent code
  - Sensors, positioning, location
  - Similarities to P-IMPL
- Pre-processor directives and platform-dependent source code
  - Bluetooth
  - NFC
  - In-app purchasing
- Calling non-existent functionality just does not do anything



Mobile API Usage – Sensors

```
// start the sensor
QSensor sensor("QAccelerometer"); sensor.start();

// later
QSensorReading *reading = sensor.reading();
qreal x = reading->property("x").value<qreal>();
qreal y = reading->value(1).value<qreal>();
```

```
import QtSensors 5.0

Text {
    TiltSensor {
        id: tilt; active: false
    }
    onSomeSignal: {
        tilt.active = (tiltStart.text === "Start");
        text: "X Rotation: " + tilt.xRotation
        text: "Y Rotation: " + tilt.yRotation
        text: "Proximity: " + (proxi.active ? (proxi.reading.near ? "Near" : "Far") : "Unknown")
```


Mobile API Usage – In-App Purchasing

- Supports Android Google Play and iOS App Store
  - Makes it easy to monetize your application (new features, new items etc.)
  - Makes it easier to use the credit card information on your platform
- Publish your application (possible using alpha or beta testing) and add new product in Google Play
  - Product Id
  - Product Type – Only Managed products supported by Qt
  - Price information
  - Activate the product
- For iOS, use iTunes Connect to register your application and products
  - Qt supports only Consumable and Non-Consumable types
  - Add the name, description, screen shot, and pricing information of your product
- Both C++ and QML types available

Qt Positioning

- In QML, the `Position` element provides the user device's current position
  - Coordinate, speed, time stamp
- Using the `PositionSource` element,
  - The source (backend) may be specified (satellite/non-satellite, socket)
  - Update interval may be set
  - Supported and preferred positioning methods may be read
  - `Position` type property `position` may be read

```
import QtPositioning 5.2

PositionSource {
    id: src
    updateInterval: 1000
    active: true
     onPositionChanged: {
        var coord = src.position.coordinate;
        console.log("Coordinate:", coord.longitude, coord.latitude);
```


Qt Positioning – Location and Address

- Location
  - Coordinate
  - Address
  - Bounding box
- Typical use case:
  - Location query with `GeocodeModel`

```
GeocodeModel {
    id: model
    plugin: mapPlugin
}

onSomeSignalHandler: {
     model.query("Hatanpään …, Tampere, Finland"); // Address, string, coordinate
     model.update();
```


Location Backend Abstraction – Plugin

- Nokia (“here”), Open Street Maps (“osm”), and MapBox (“mapbox”) supported
- Each plugin has plugin-specific configuration parameters
- Possible to set required features

```
Plugin {
    id: mapPlugin
    name: "osm"
    // preferred: [ "osm", "here" ]
    required: Plugin.RoutingFeature

    PluginParameter { name: "name"; value: "value" } }a
```

Location – Map

- Displays the map
- Supports all Qt gestures (pinch, pan, swipe)
- Can show annotations (from the model) `MapRectangle, MapQuickItem, MapItemView`
- Can show routes `MapRoute`
- Map type (style: satellite, street, terrain; mobile, night)

```
Map {
    id: map
    plugin: mapPlugin

    center {
        latitude: positionSrc.position.coordinate.latitude
        longitude: positionSrc.position.coordinate.longitude
    }
    zoomLevel: 8.0 // >= 0
```


Location – Places & Routes

- Use the corresponding model to query for places or routes
  - `PlaceSearchModel { searchTerm: “gasoline”; searchArea: currentCoordinate }`
  - `RouteModel { query: myQuery  } // queryObject.addWaypoint(coordinate)`
- Use any view to show the model content as usual
  - For route data, there is already a delegate type `MapRoute`

```
Map {
    RouteQuery { id: routeQuery }
    RouteModel { id: routeModel; plugin: mapPlugin; query: routeQuery }
    MapItemView { model: routeModel; delegate: routeDelegate }

    Component { id: routeDelegate
        MapRoute {
            route: routeData
            line.color: "blue"
            line.width: 5
            opacity: 0.8
```


Mobile API Usage – In-App Purchasing

- Register your products

```
Store {
    Product {
        id: myCoolProduct
        identifier: "consumableProduct"
        type: Product.Consumable
        // Product.Unlockable
```

- Implement UI elements to purchase products
  - Native functionality will handle the purchasing process (ask passwords, if needed etc.)

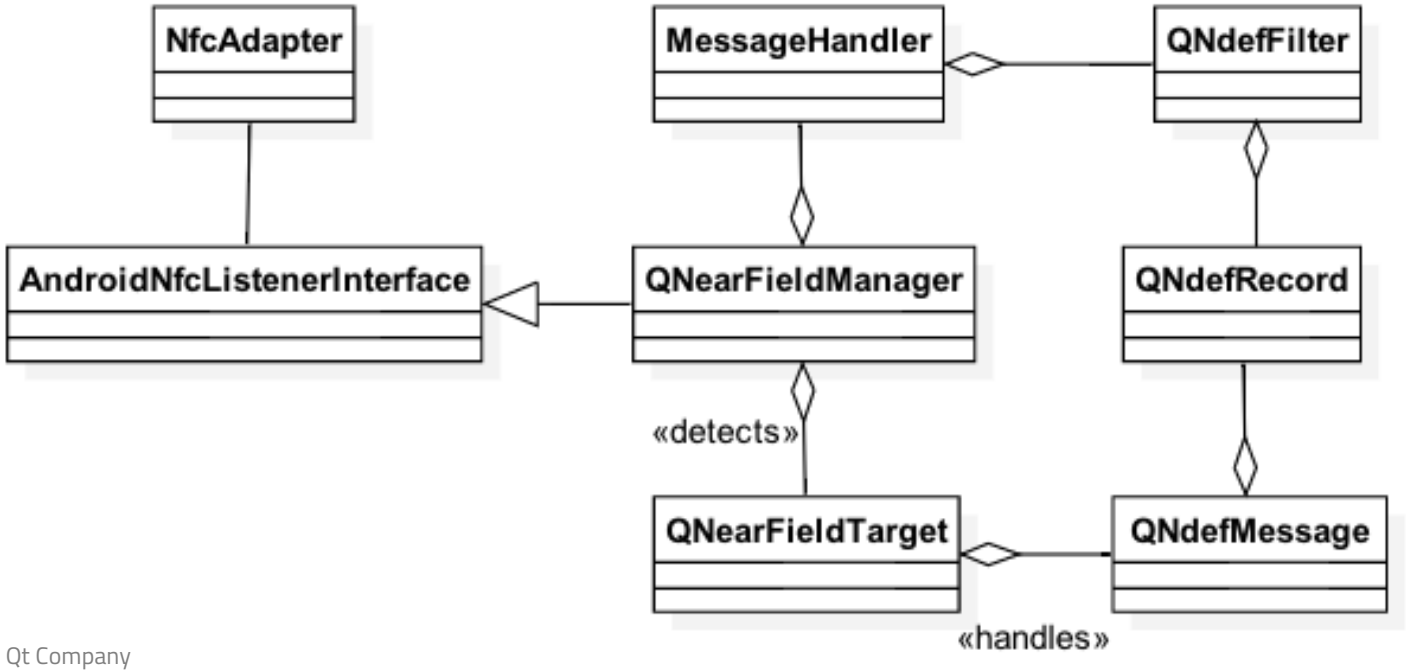
```
MouseArea {
    enabled: !myCoolProduct.purchasing &&
       myCoolProduct.status ===
                  Product.Registered
    onClicked: {
        myCoolProduct.purchasing = true;
        myCoolProduct.purchase();
```

Cross-Platform APIs: Connectivity

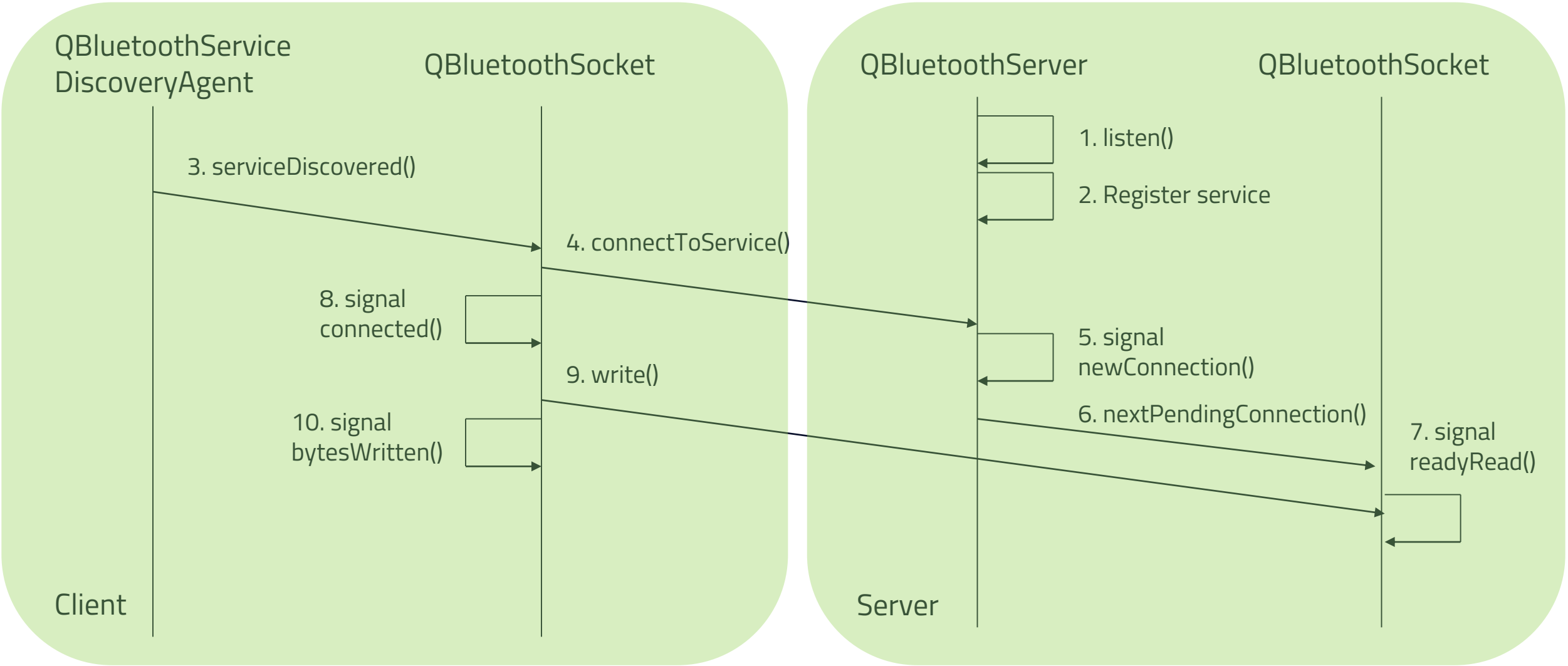
- NFC
  - NFC tag detection
  - Reading and writing NDEF messages, NDEF message handler registration
  - File and message sharing
- Bluetooth
  - Local device information
  - Device and service discoveries
  - OBEX object push profile (not on Android)
  - Serial port profile via RFCOMM

NFC

```
m_manager = new QNearFieldManager(this);
connect(m_manager, SIGNAL(targetDetected(QNearFieldTarget*)), this,
                   SLOT(targetDetected(QNearFieldTarget*)));
connect(m_manager, SIGNAL(targetLost(QNearFieldTarget*)), this,
                   SLOT(targetLost(QNearFieldTarget*)));
m_manager->startTargetDetection();
```



Bluetooth



QBluetoothServiceInfo

- Bluetooth service attributes
  - Protocol, port, description, name, provider, service class Uuid

- Defined as name and sequence pairs
  - Set with convenience functions
    - `serviceInfo.setServiceUuid(QBluetoothUuid(serviceUuid));`
  - Or with generic setAttribute() function

```
QBluetoothServiceInfo::Sequence publicBrowse;
publicBrowse << QVariant::fromValue(QBluetoothUuid(QBluetoothUuid::PublicBrowseGroup));
serviceInfo.setAttribute(QBluetoothServiceInfo::BrowseGroupList, publicBrowse);
```

Bluetooth

```
rfcommServer = new QBluetoothServer(QBluetoothServiceInfo::RfcommProtocol, this);
connect(rfcommServer, SIGNAL(newConnection()), this, SLOT(clientConnected()));
bool result = rfcommServer->listen(localAdapter);

// Set service attributes
serviceInfo.registerService(localAdapter);
```

```
socket = new QBluetoothSocket(QBluetoothServiceInfo::RfcommProtocol);
socket->connectToService(remoteService);
connect(socket, SIGNAL(readyRead()), this, SLOT(readSocket()));
connect(socket, SIGNAL(connected()), this, SLOT(connected()));
connect(socket, SIGNAL(disconnected()), this, SIGNAL(disconnected()));
```

Bluetooth QML Client

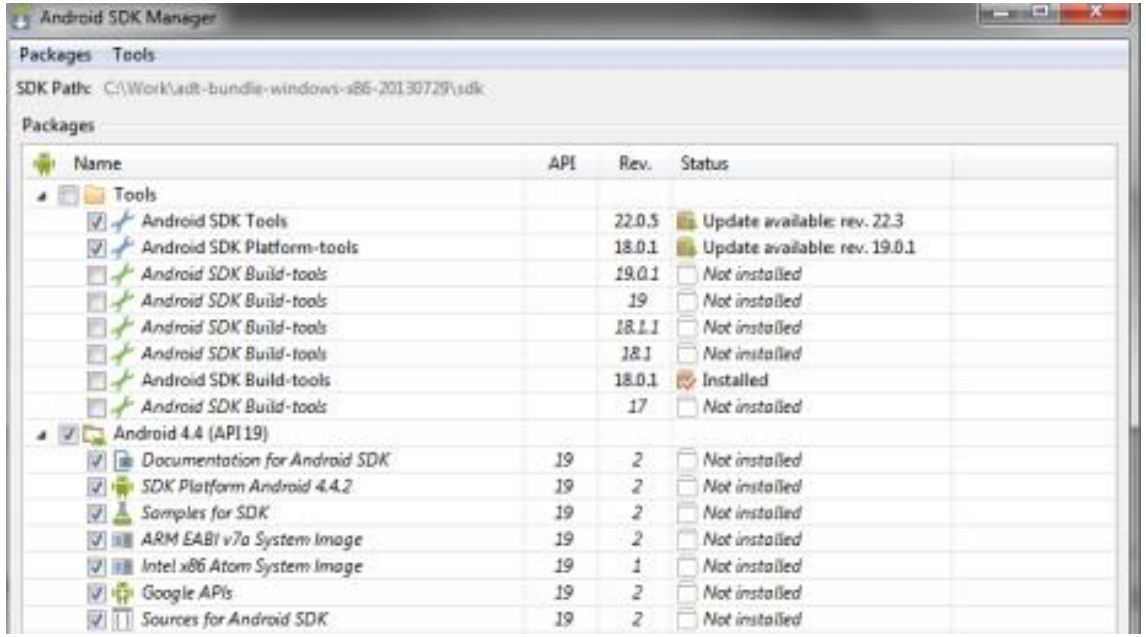
- Device and service discovery
  - `BluetoothDiscoveryModel`
  - Set the mode (full, minimal, device) and UUID to limit services
  - Connect to `deviceDiscovered()` or `serviceDiscovered()` signals
- Bluetooth service information
  - `BluetoothService`
  - Returned by the service discovery
  - Device address and name, service name, protocol, and description
  - Service UUID
- Device and service connection and communication
  - `BluetoothSocket`
  - Set the service
  - Check the state (unconnected, connecting, connected, listening, closing)
  - Read and write data (property `stringData`)

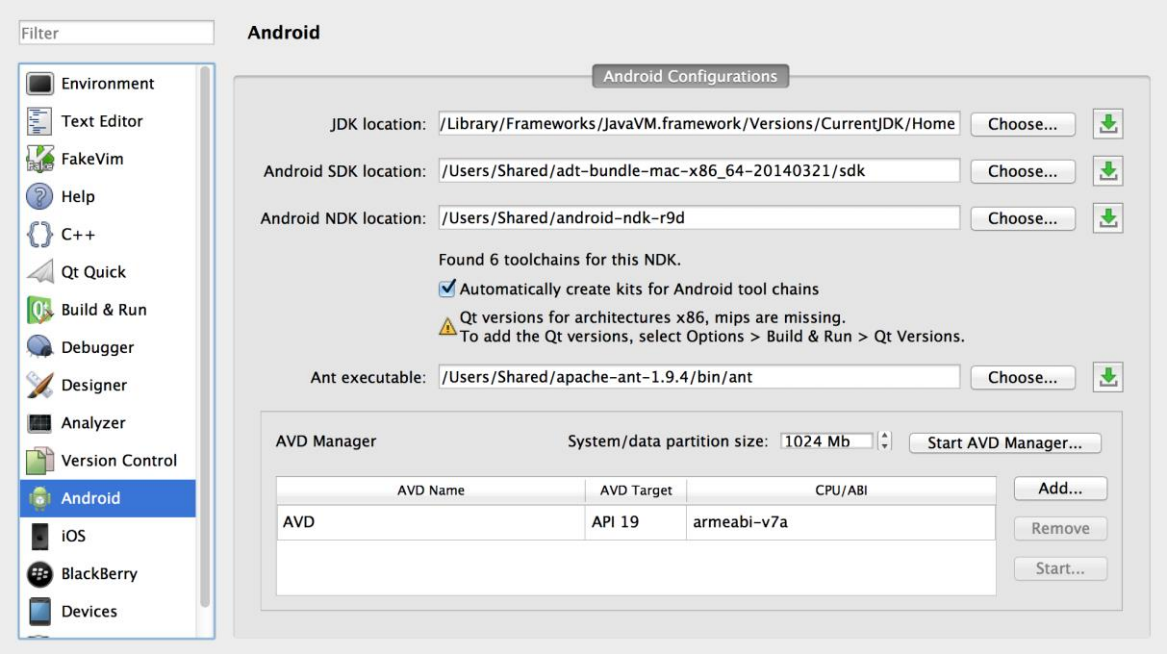
Bluetooth QML Client

```
BluetoothDiscoveryModel {
    id: btModel
    running: true
    discoveryMode: BluetoothDiscoveryModel.MinimalServiceDiscovery
    onServiceDiscovered: {
        socket.setService(service)
    }
    uuidFilter: "e8e10f95-1a70-4b27-9ccf-02010264e9c8"
}
BluetoothSocket {
    id: socket
    connected: true
    onSocketStateChanged: {
        console.log("Connected to server")
    }
    onStringDataChanged: {
        console.log("Received data: “, socket.stringData);
    }
}
```


Android

Qt in Android – Tooling



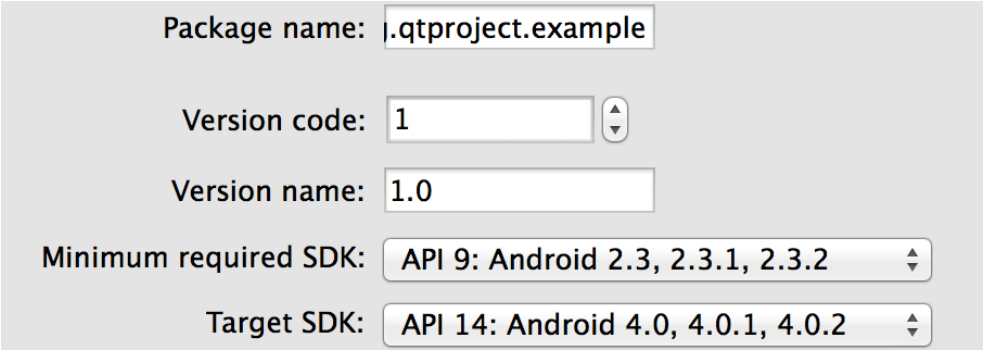


Qt in Android – Application Building Blocks

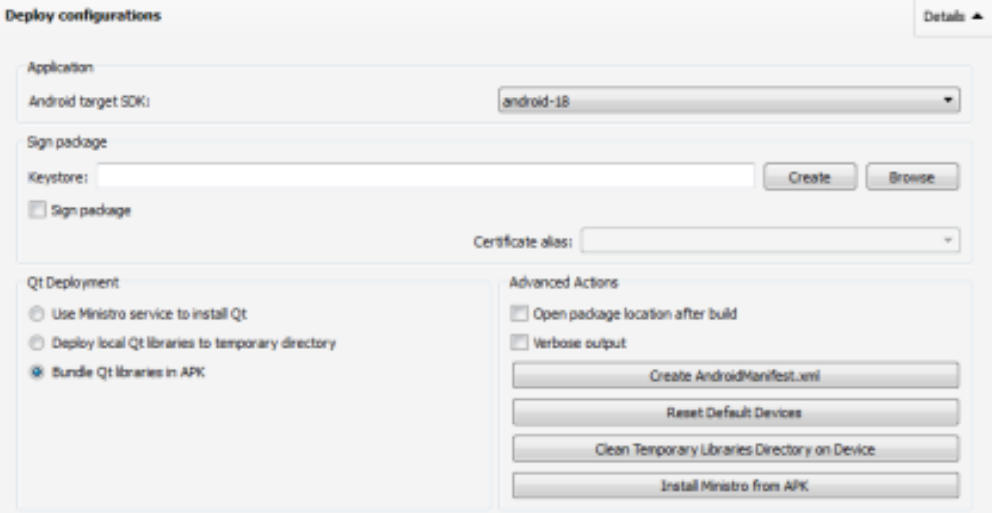
- All modules supported except Qt WebEngine, Qt NFC, Qt SerialPort and platform specific ones
- Android resources
  - Accessible using schema `assets://`

Building, Package Creation, Signing, Deployment

- Almost everything supported in QtCreator (Projects mode: Run Settings)
- Signing, icons, permissions, version management, library deployment
- Permissions will be selected automatically based on Qt headers



- Three deployment options
  - Projects > Run > Deploy configurations
  - Using Ministro, temporary directory for Qt libs, APK
- Publishing in Google Play
  - Signed APK can be created in QtCreator
  - Login to Goole Play and upload your app



Native Apps Android – Consists of Application Components

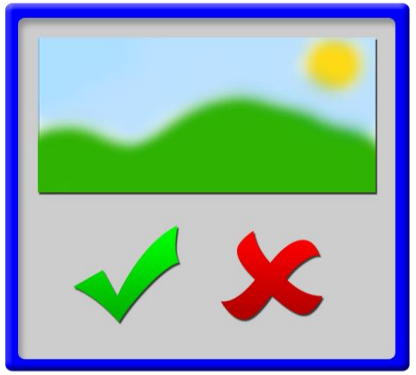
Activity	One UI in the application Typically full screen Compare to a page / view
Broadcast Receiver	Response to an event or a notification Can launch an application Incoming phone call, network connection established
Service	Background task without the UI Keeps running while user navigates to another application Music Player, Location Tracker
Content Provider	Shares content between apps (processes) Contacts

APK #1/Process

APK #2/Process

Android Activity

onCreate()
onStart()
onResume()
onPause()
onStop()
onDestroy()
onTouchEvent()
onUserInteraction()
etc.

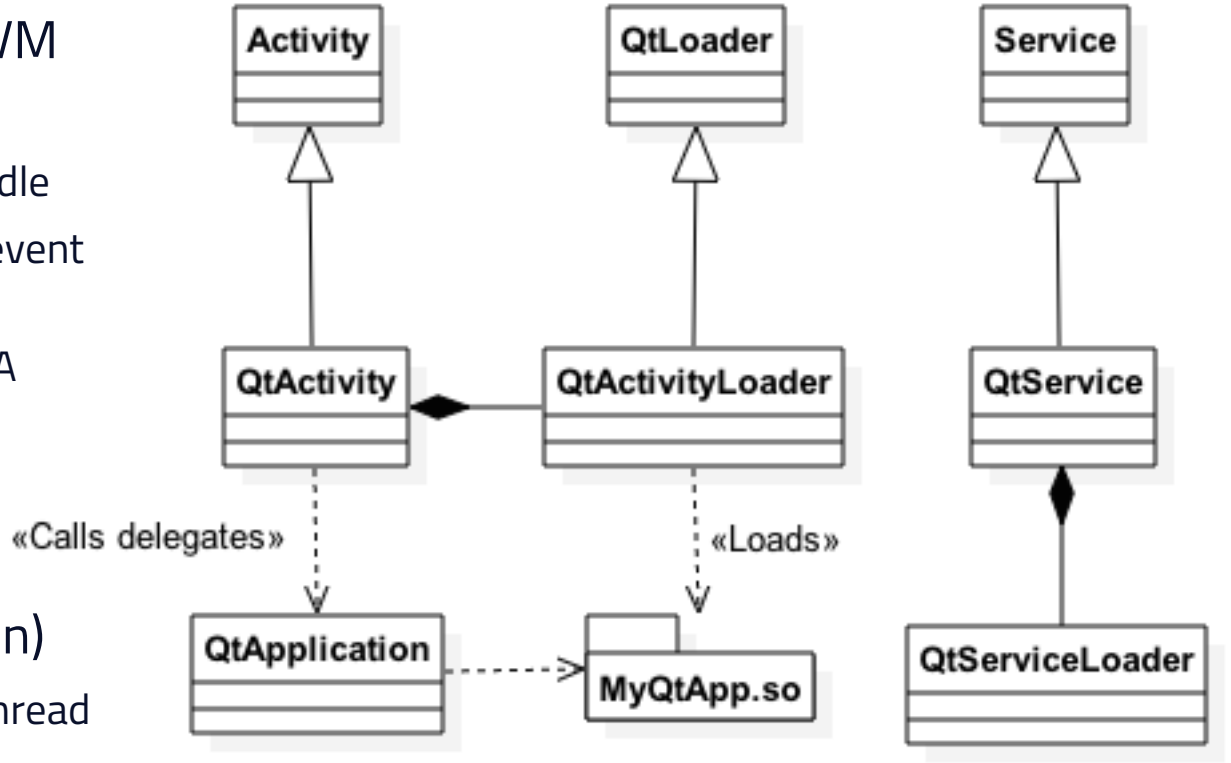


Activity

Looper
<<main thread>>

Qt Android Apps

- Native process with a GUI thread, running a Dalvik VM
  - Launces an activity
  - Activity uses the loader to load Qt libs using Ministro or bundle
  - Creates `QtApplication`, which works as a delegate for event handling
  - Events handled natively in the activity => propagated to QPA plugin using the application delegate
- In C++, main integration takes place in **androidjnimain.cpp** (part of the Andoird QPA plug-in)
  - Creates a new thread and starts handling Qt events in the thread
  - Uses JNI to access Android native functions (application state, screen orientation, create Drawables, show status bar etc.)



Native APIs Android

- Provided by Qt Android Extras module
- `QAndoirdJniEnvironment` – attaches your thread to Dalvik VM
- `QAndroidActivityResultReceiver` – implement to get notifications about `QtActivity` start activity functions
- `QAndroidJniObject` – provides APIs to call Java methods
  - Call static and non-static methods, set and get object fields
  - You need to provide a signature of a method "`(Arguments)ReturnType`"
  - All object types are returned as `QAndroidJniObject`

Arguments and Return Types in Signatures

jboolean	Z		void	V
jbyte	B		custom	L<fully qualified name>
jchar	C		jobject	Ljava/lang/Object;
jshort	S		jclass	Ljava/lang/Class;
jint	I		jstring	Ljava/lang/String;
jlong	J		jthrowable	Ljava/lang/Throwable;
jfloat	F		jobjectArray	[Ljava/lang/Object;
jdouble	D		jarray	[<*type*>

Java Class

```
package io.qt.training;
// Create a source folder by using the fully qualified class name

class Dummy {
    public static int simpleMethod(String string) {
        String anotherString = new String("Hello");
        if (string.startsWith(anotherString))
            return checkUsingNativeCode(string);
        else
            return 0;
    }
    private static native int checkUsingNativeCode(String string);
}
```


JNI – Calling Java Methods from C++

```
QAndroidJniObject string = QAndroidJniObject::fromString("Hello World");
QAndroidJniObject example = QAndroidJniObject::callStaticObjectMethod(
    ”io/qt/training/Dummy", ”simpleMethod"
    ”(Ljava/lang/String;)I"
     string.object<jstring>());
```

JNI – Calling C++ Methods from Java

```
static void callFromJava(JNIEnv *env, jobject thiz, jstring string)
{
    QString aString(env->GetStringUTFChars(string, 0));
}

void registerNativeMethods()
{
    JNINativeMethod methods[] {{"checkUsingNativeCode", "(Ljava/lang/String)I",
                                reinterpret_cast<void *>(callFromJava)}};
    QAndroidJniObject javaClass("io/qt/training/Dummy");
    QAndroidJniEnvironment env;
    jclass objectClass = env->GetObjectClass(javaClass.object<jobject>());
    env->RegisterNatives(objectClass, methods, sizeof(methods) / sizeof(methods[0]));
    env->DeleteLocalRef(objectClass);
}
```

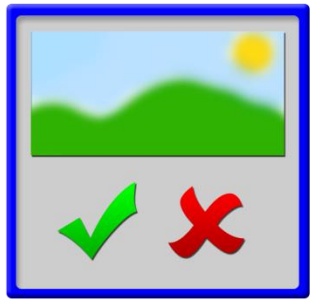
Custom Activity

- Subclass `QtActivity` Java class or implement new classes
- Implement Java functions to
  - Create new activities
  - Read data from content providers (calendar, camera, phone book)
  - Send intents for broadcast receivers (or receive broadcast message yourself)
  - Start services (possibly running in the background)
- Implement corresponding C++ functions and expose to QML, if needed
- Use Qt Android Extras module classes to implement the glue between C++ and Java

Intents

› Message to communicate with other UI components possibly in other threads or processes

Intent Filter



Activity

```
Context.startActivity() or
Activity.startActivityForResult()

 onAcitivityResult()
```

```
getIntent()
Activity.setResult()
```

QtAndroid Namespace

- Useful functions for Android development
  - `QNadroidJniObject QtAndroid::androidActivity();`
  - `QNadroidJniObject QtAndroid::androidContext();`
  - `QNadroidJniObject QtAndroid::androidService();`

  - `void QtAndroid::runOnAndroidThread(const Runnable &runnable);`

iOS

Qt in iOS – Tooling

- Xcode and Xcode command line tools are enough

- Apple developer id
- Register your device
- Developer/distribution provisioning profile
  - Associates together device id, application bundle id, and developer certificate
- Developer/distribution certificate

Qt in iOS – Application Building Blocks

- iOS fonts and icons
  - application icon must be deployed in the bundle and defined in the info.plist file using `CFBundleIconFiles` key

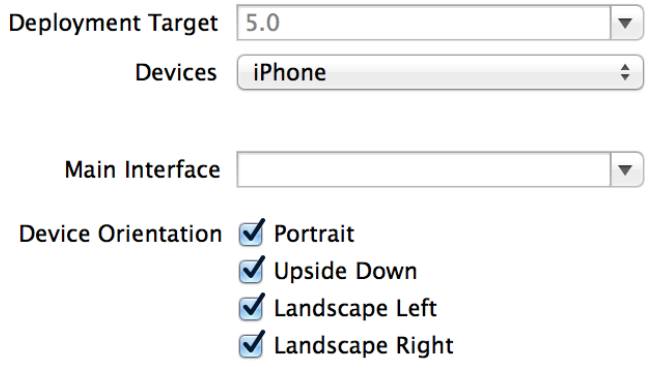
```
fontFiles.files = fonts/*.ttf
fontFiles.path = fonts
QMAKE_BUNDLE_DATA += fontFiles
```

- iOS assets

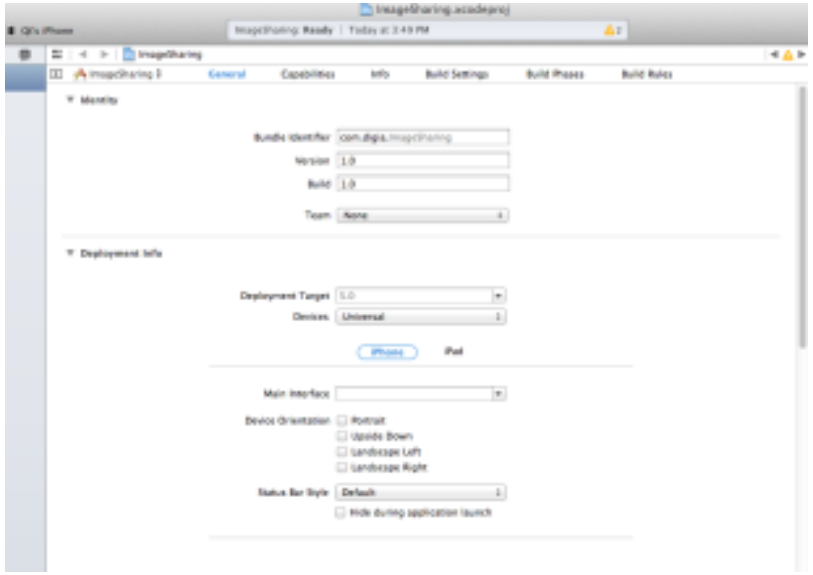
```
assets_catalogs.files =
$$files($$PWD/*.xcassets)
QMAKE_BUNDLE_DATA += assets_catalogs
```

Building, Package Creation, Signing, Deployment

- You may build and deploy your application to iOS in QtCreator
- Qt iOS project contains Xcode project file (`.xcodeproj`)
- You may open the project in Xcode and build it there
  - Package settings in Xcode
  - Icons, orientations, application bundle, version number, target devices, libraries, capabilities (In-App Purchase)

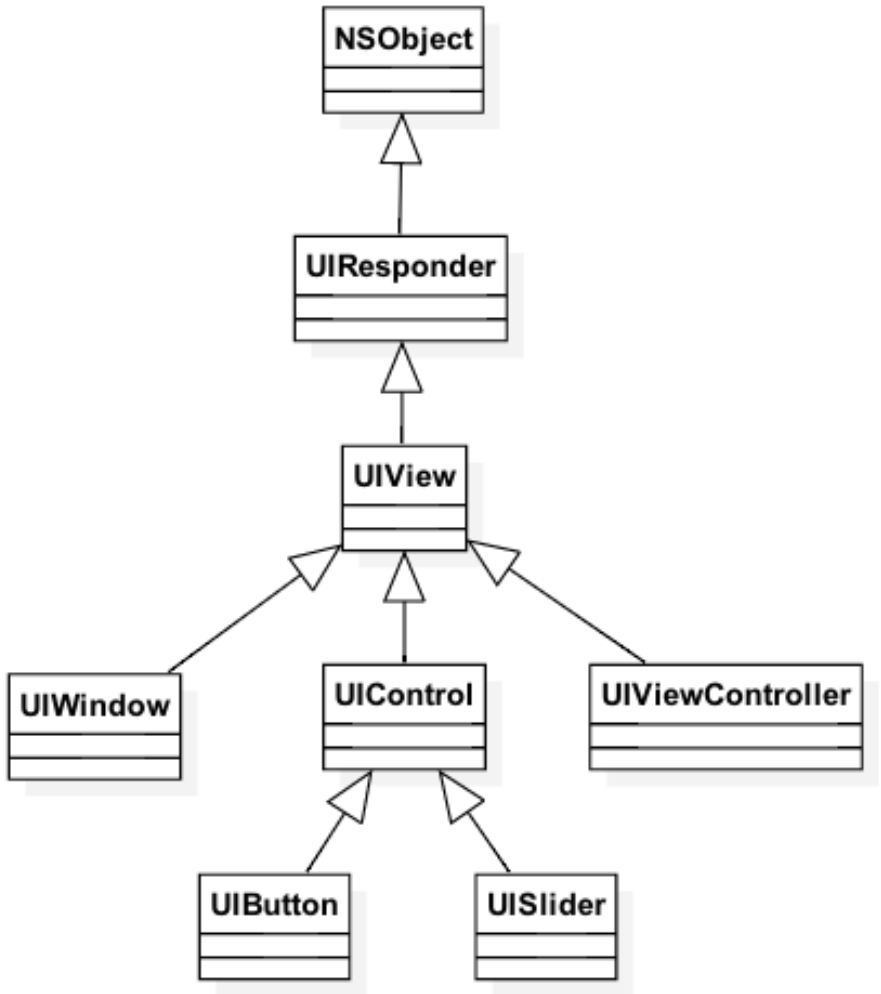


- Publish using iToons Connect



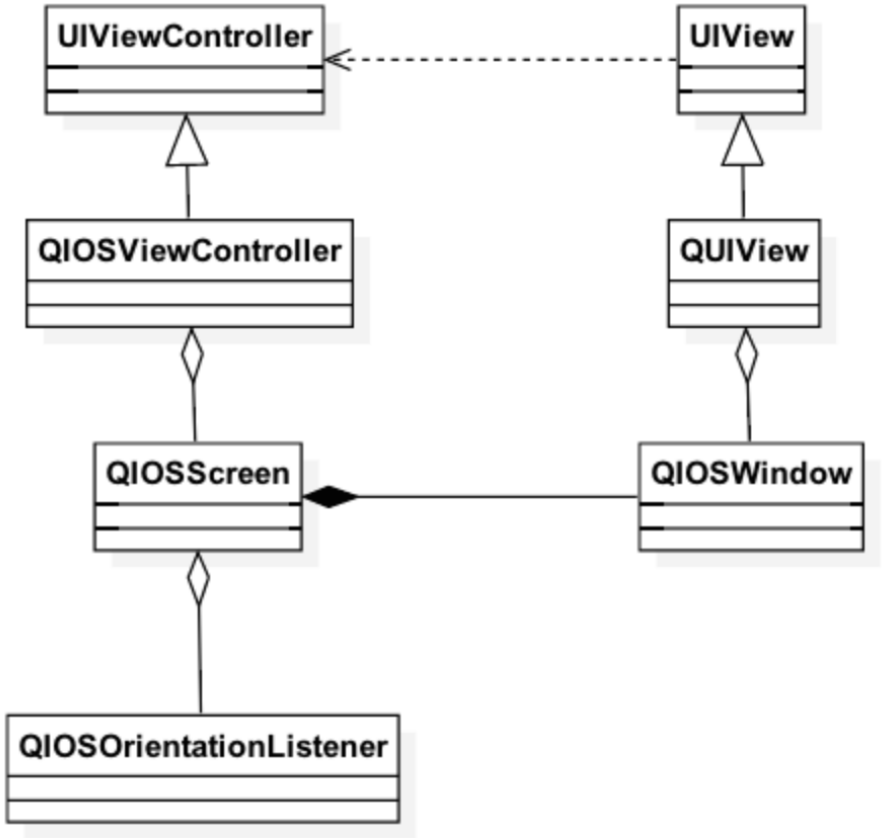
Native Apps iOS

- MVC pattern
- Controller
  - Manages the view
  - Works as a delegate for changes in the model
  - Provides an interface to access iOS frameworks (libraries)
    - Address book, event kit, message UI, telephony etc.
  - Application framework handles the navigation between views using controllers
- View is a rectangle area on a window
  - Similar to `QWidget` inside `QWindow`



Qt iOS Apps

- Based on Cocoa's Model-View Controller Framework, provided by UiKit framework
  - UIController manages the memory and navigation between windows (views) – does not exist in Qt
  - UI view corresponds to Qt window
  - UI controls are sub-windows / views
- `QWindow` mapped to `UIViewController`
  - Allows navigation to address book, calendar, messaging



Native APIs iOS

- Possible to mix C++ and Objective-C
- Use `.mm` suffix for your source code files
- Add sources into `OBJECTIVE_SOURCES` variable in the **`.pro`** file
- You may use `QMAKE_INFO_PLIST` variable to refer to your **`info.plist`** file

- Get the native `UiView` from the **`.qpa`** plugin

```
UIView *view = static_cast<UIView *> (QGuiApplication::platformNativeInterface() ->
nativeResourceForWindow("uiview", window()));
```

Native APIs iOS

- Use `UiView` to get the `UiViewController`

```
UIViewController *qtCtrl = [[view window] rootViewController];
```

- Set the delegate
  - Handles the callback after data have been picked or chosen
- Create a new controller
  - Picker – pick an image
  - Chooser – choose a file
- Navigate to a new view using the controller

Application Lifecycle

Application Lifecycle and Persistent Data

- Both in Android and iOS, apps are suspended, when the user switches to another app
  - Furthermore, a suspended application may be removed from the memory at any time
- Native states are mapped to `Qt::ApplicationStates` enumeration

Application States

- `Qt::ApplicationActive` – foreground top-most focused application
- `Qt::ApplicationInactive` – application is visible, but not in the foreground
  - Pause or stop video playback, games, animations, and sensors – i.e. stop using CPU
- `Qt::ApplicationSuspended` – application is about to suspend and not visible
  - User presses Home Screen button and navigates to another application
  - Save app state into a persistent storage, if needed (`QSettings`)
  - In this state, app may be removed from the memory at any time
- `Qt::ApplicationHidden` – background application
  - Music player

Thank you