ONE YEAR ANNIVERSARY

the NeRd Talks

One Year Anniversary

NeRd Talks Vol12 - QtPlugins

Relatore: Stefano Aru

What is a plugin

In computing, a plug-in (or plugin, add-in, addin, add-on, or addon) is a software component that adds a specific feature to an existing computer program. When a program supports plug-ins, it enables customization.
[Wikipedia]

What is a QtPlugin

QtPlugins are implemented as shared library loaded at runtime.

Qt offer a set of functionalities to load and use plugins at run time.

Why plugins?

Plugins allow us to upgrade and customize an application one step at the time.

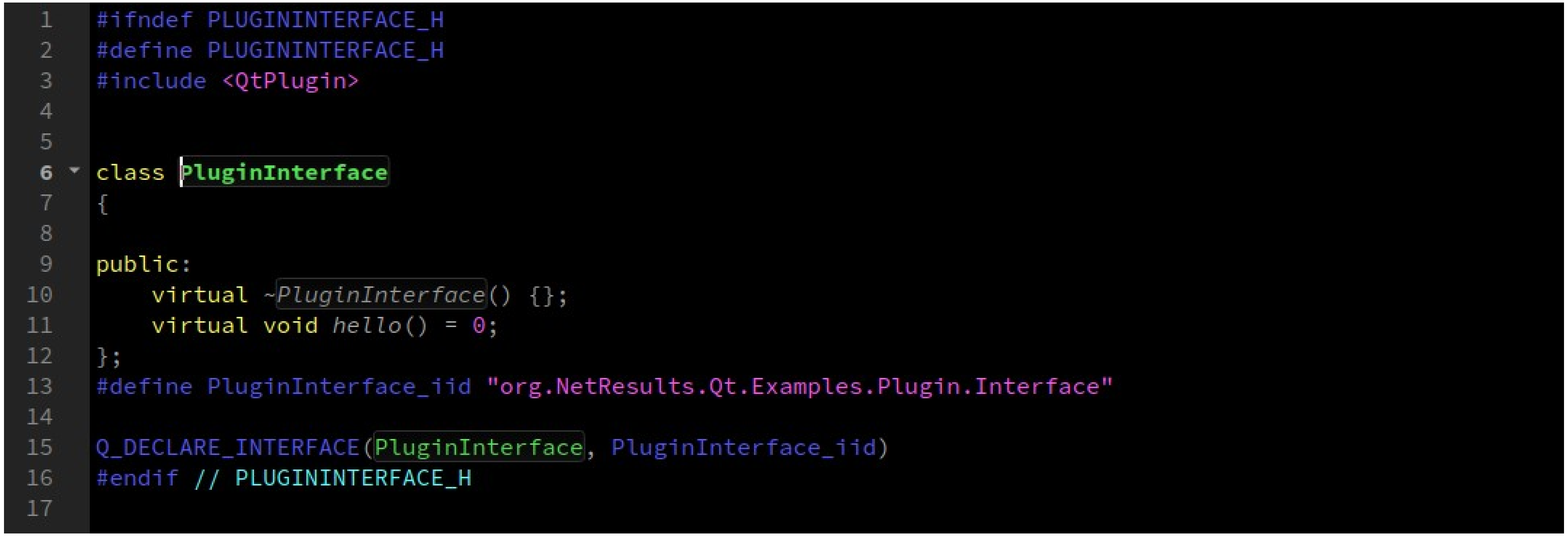
How do I make a QtPlugin?

- Plugin interface definition

- Plugin project creation

- Plugin implementation

QtPlugin interface definition



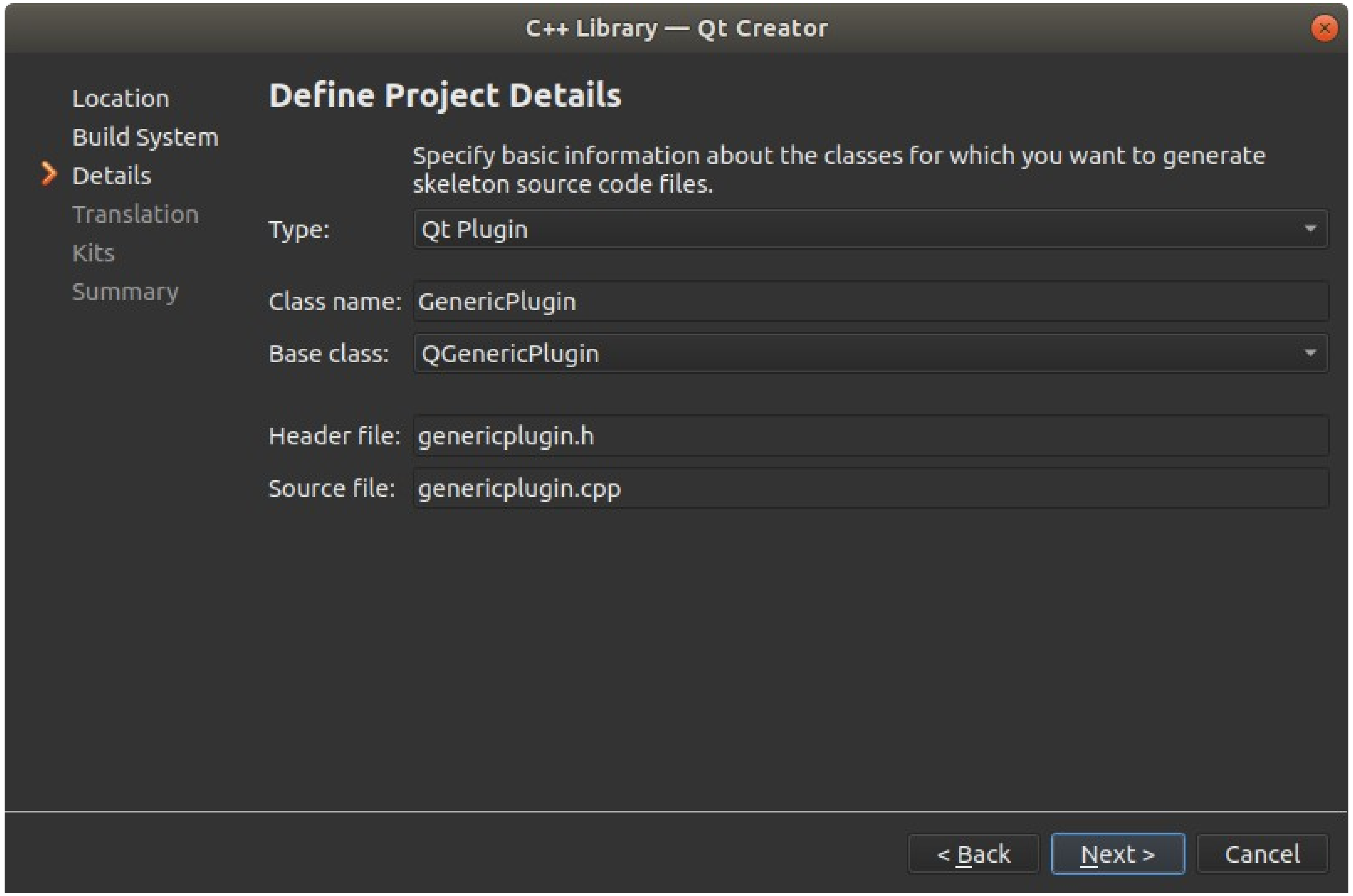
```cpp
#ifndef PLUGININTERFACE_H
#define PLUGININTERFACE_H
#include <QtPlugin>


class PluginInterface
{

public:
    virtual ~PluginInterface() {};
    virtual void hello() = 0;
};
#define PluginInterface_iid "org.NetResults.Qt.Examples.Plugin.Interface"

Q_DECLARE_INTERFACE(PluginInterface, PluginInterface_iid)
#endif // PLUGININTERFACE_H

```

QtPlugin interface definition

Q_DECLARE_INTERFACE:
  This macro allows Qt to register the new interface to be referred later

virtual void hello() = 0;
  Our virtual method to implement

Plugin project creation



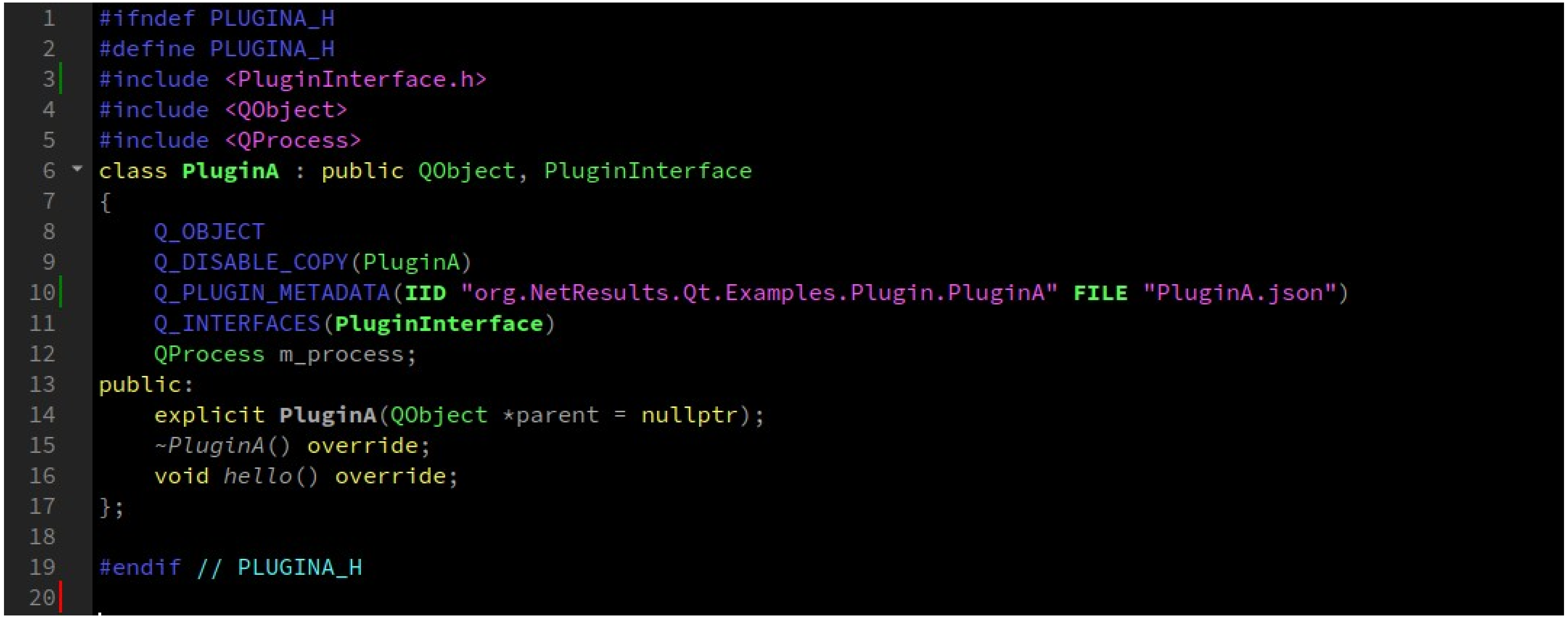
C++ Library — Qt Creator

Define Project Details

Specify basic information about the classes for which you want to generate skeleton source code files.

Type:	Qt Plugin
Class name:	GenericPlugin
Base class:	QGenericPlugin
Header file:	genericplugin.h
Source file:	genericplugin.cpp

QtPlugin definition



```
#ifndef PLUGINA_H
#define PLUGINA_H
#include <PluginInterface.h>
#include <QObject>
#include <QProcess>
class PluginA : public QObject, PluginInterface
{
    Q_OBJECT
    Q_DISABLE_COPY(PluginA)
    Q_PLUGIN_METADATA(IID "org.NetResults.Qt.Examples.Plugin.PluginA" FILE "PluginA.json")
    Q_INTERFACES(PluginInterface)
    QProcess m_process;
public:
    explicit PluginA(QObject *parent = nullptr);
    ~PluginA() override;
    void hello() override;
};

#endif // PLUGINA_H

```

QtPlugin definition

Q_PLUGIN_METADATA:

This macro allows us to assign the id to our Plugin, we can also specify additional metadata passing a json file

Q_INTERFACES:

This macro tells Qt which interfaces the class Implements, the interface must be previously registered with Q_DECLARE_INTERFACE

QtPlugin definition

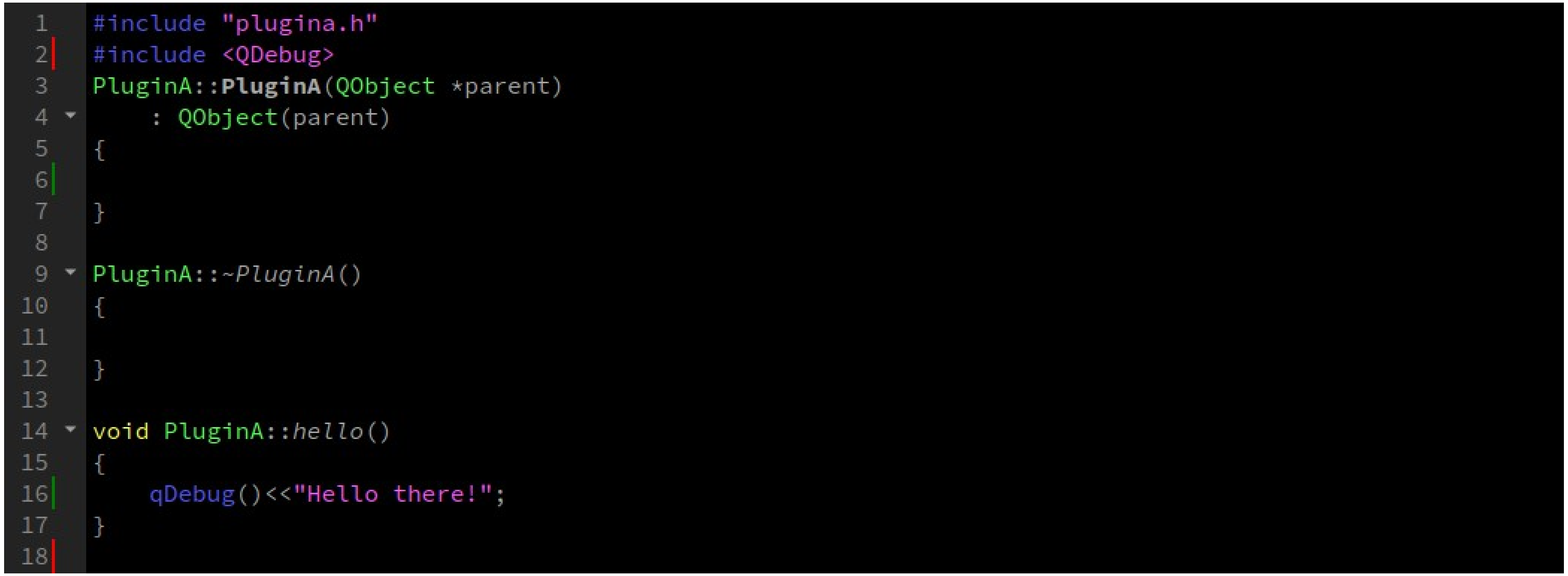
void hello() override;

The interface method we want to implement

class PluginA : public QObject, PluginInterface

Our plugin must inherits from QObject

QtPlugin implementation



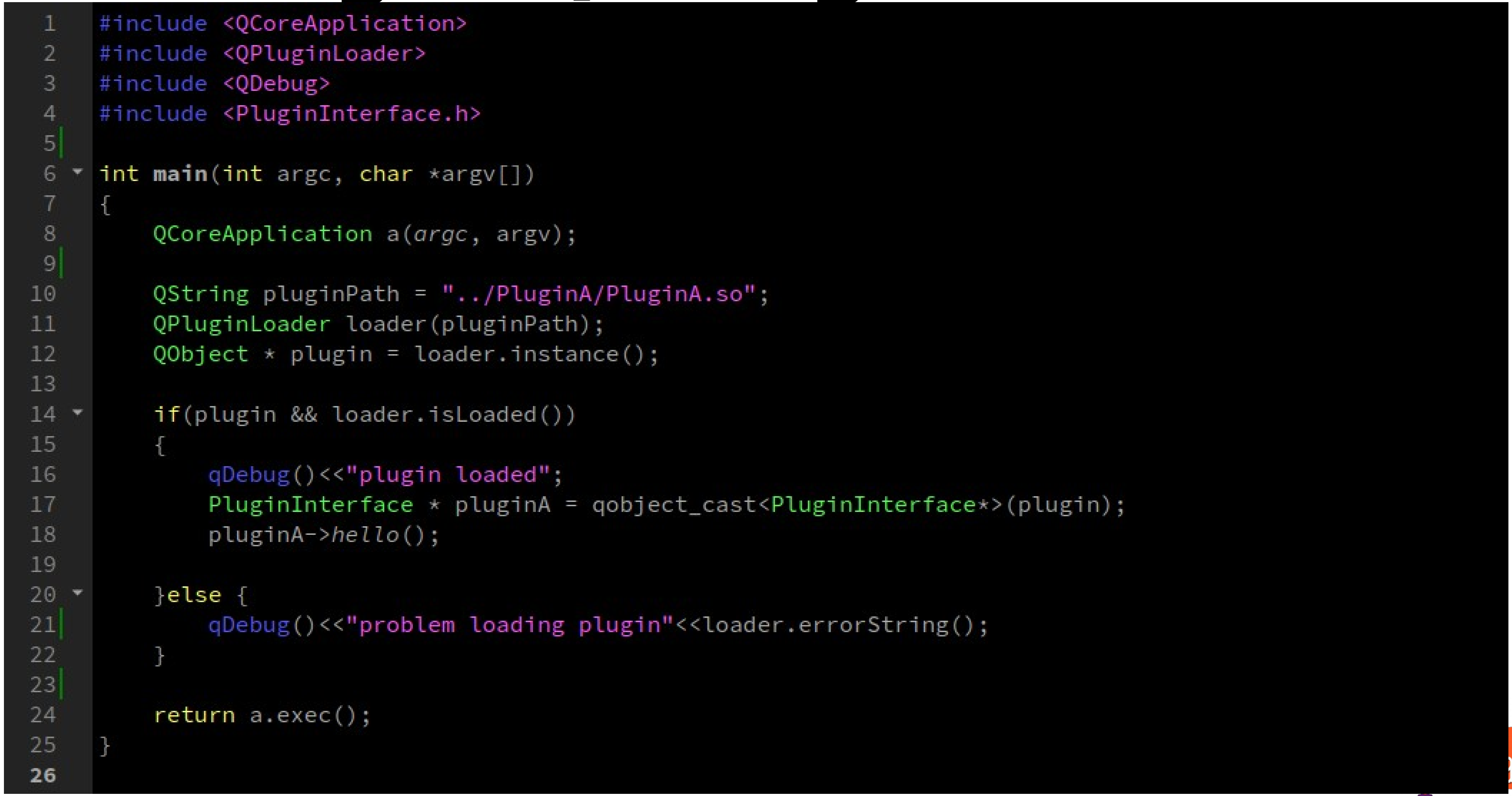
```
#include "plugina.h"
#include <QDebug>
PluginA::PluginA(QObject *parent)
    : QObject(parent)
{

}

PluginA::~PluginA()
{

}

void PluginA::hello()
{
    qDebug()<<"Hello there!";
}

```

QtPlugin implementation

void PluginA::hello()

The virtual method implementation

Loading a QtPlugin



```cpp
#include <QCoreApplication>
#include <QPluginLoader>
#include <QDebug>
#include <PluginInterface.h>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    QString pluginPath = "../PluginA/PluginA.so";
    QPluginLoader loader(pluginPath);
    QObject * plugin = loader.instance();

    if(plugin && loader.isLoaded())
    {
        qDebug()<<"plugin loaded";
        PluginInterface * pluginA = qobject_cast<PluginInterface*>(plugin);
        pluginA->hello();

    }else {
        qDebug()<<"problem loading plugin"<<loader.errorString();
    }

    return a.exec();
}
```

Loading a QtPlugin

QPluginLoader

Is the class responsible to load plugins at runtime

loader.instance()

Returns the instance to our plugin

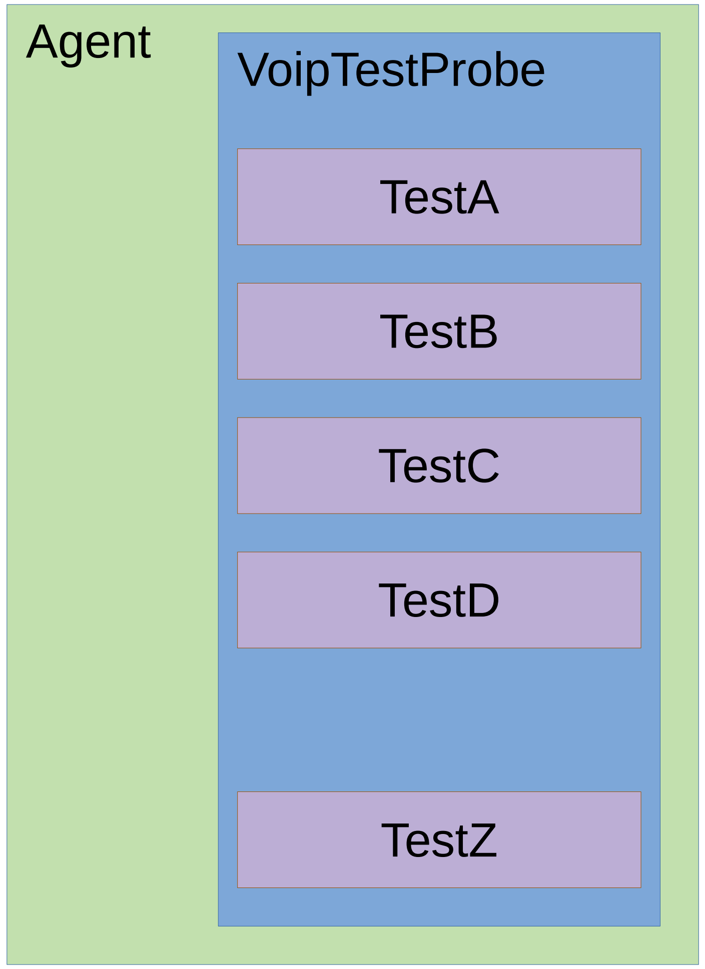
The instance will be always the same unless the loader is unloaded and then loaded again

iQAC Agent and QtPlugins

Starting scenario:

In the Agent software the test creation and configuration was demanded to a single class

iQAC Agent and QtPlugins



iQAC Agent and QtPlugins

Problems:

- Adding a new test meant to add even more code to this class

- This class has grown up to over 9000 lines of code with a huge switch

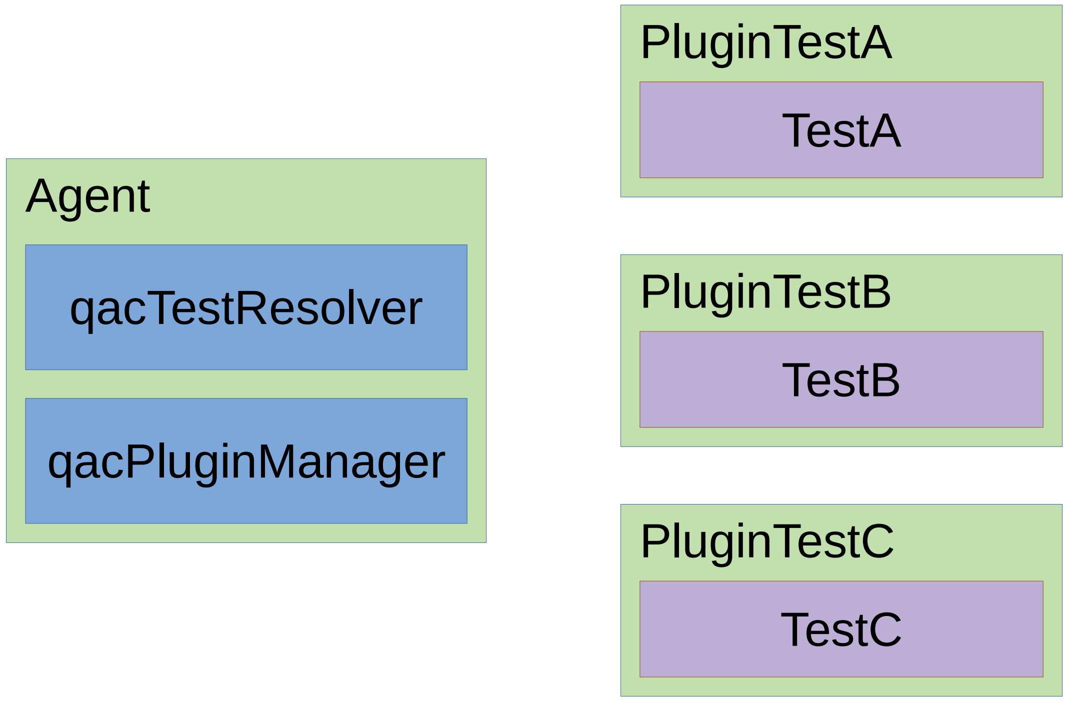
iQAC Agent and QtPlugins

We aimed to translate this logic into plugins

- Improved the maintainability

- The delivery process for a new test can be much more simple

iQAC Agent and QtPlugins



iQAC Agent and QtPlugins

- Metadata are used to track capabilites of each Test plugin

- Two simple components replaces the structure of the previously massive class, spreading the logic over all the plugins

Plugins pros

- Granularity

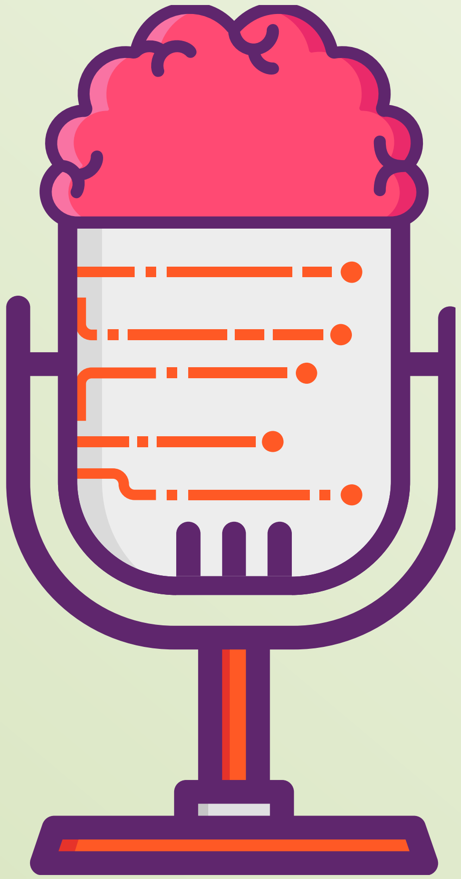
- Partial upgrades

Plugins cons

- Data contracts

- They must be plugged

Question time



Thanks for your attention!