Cross-platform software development in practice. Object-Oriented approach.

Vitaly Repin
Maemo Devices, Nokia

Maemo

March 25, 2010

Outline

Abstract

- The basic foundations of your software are changing all the time. The real challenge: *The only constant is change.*
- How to be prepared for the changes? Is it possible to predict the future? Yes, to some extent...
- Don't be afraid! There are technologies around which will help you. But *there is no silver bullet.* Use your brain and common sense to win!
- Development of portable software is engineering work. This is not a piece of cake but software engineer *CAN DO* this work and enjoy it!
- Qt and C++ are your friends. But use them wisely. *THERE IS NO SILVER BULLET.*

Abstract

- The basic foundations of your software are changing all the time. The real challenge: *The only constant is change.*
- How to be prepared for the changes? Is it possible to predict the future? Yes, to some extent...
- Don't be afraid! There are technologies around which will help you. But *there is no silver bullet.* Use your brain and common sense to win!
- Development of portable software is engineering work. This is not a piece of cake but software engineer *CAN DO* this work and enjoy it!
- Qt and C++ are your friends. But use them wisely. *THERE IS NO SILVER BULLET.*

Abstract

- The basic foundations of your software are changing all the time. The real challenge: *The only constant is change.*
- How to be prepared for the changes? Is it possible to predict the future? Yes, to some extent...
- Don't be afraid! There are technologies around which will help you. But *there is no silver bullet.* Use your brain and common sense to win!
- Development of portable software is engineering work. This is not a piece of cake but software engineer *CAN DO* this work and enjoy it!
- Qt and C++ are your friends. But use them wisely. *THERE IS NO SILVER BULLET.*

Abstract

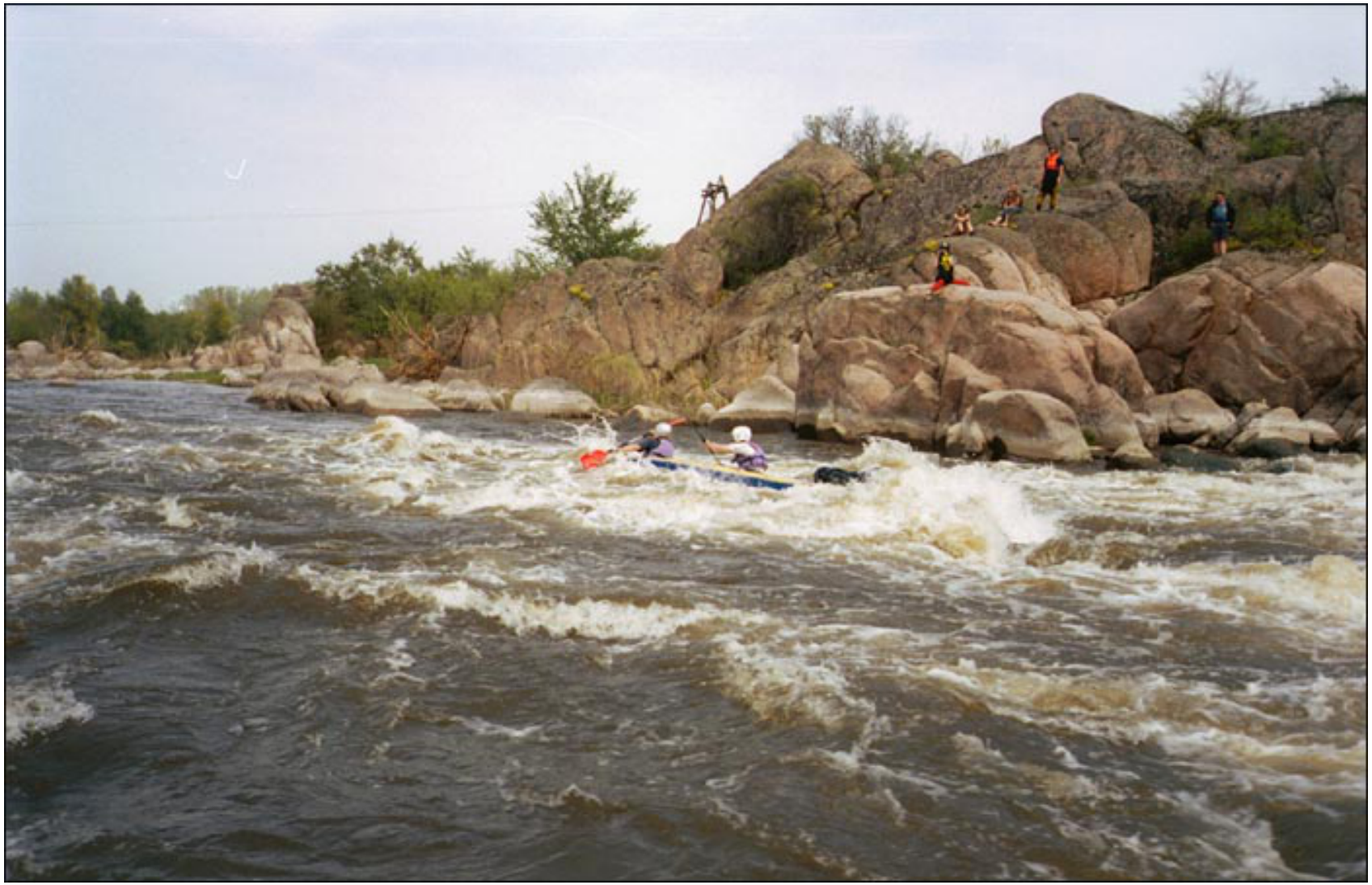
- The basic foundations of your software are changing all the time. The real challenge: *The only constant is change.*
- How to be prepared for the changes? Is it possible to predict the future? Yes, to some extent...
- Don't be afraid! There are technologies around which will help you. But *there is no silver bullet.* Use your brain and common sense to win!
- Development of portable software is engineering work. This is not a piece of cake but software engineer *CAN DO* this work and enjoy it!
- Qt and C++ are your friends. But use them wisely. *THERE IS NO SILVER BULLET.*

Abstract

- The basic foundations of your software are changing all the time. The real challenge: *The only constant is change.*
- How to be prepared for the changes? Is it possible to predict the future? Yes, to some extent...
- Don't be afraid! There are technologies around which will help you. But *there is no silver bullet.* Use your brain and common sense to win!
- Development of portable software is engineering work. This is not a piece of cake but software engineer *CAN DO* this work and enjoy it!
- Qt and C++ are your friends. But use them wisely. *THERE IS NO SILVER BULLET.*

Outline

What is happening there?



Definition

"In computing, cross-platform, or multi-platform, is an attribute conferred to computer software or computing methods and concepts that are implemented and inter-operate on multiple computer platforms. Cross-platform software may be divided into two types; one requires individual building or compilation for each platform that it supports, and the other one can be directly run on any platform without special preparation."

Wikipedia

Example

"For example, a cross-platform application may run on Microsoft Windows on the x86 architecture, Linux on the x86 architecture and Mac OS X on either the PowerPC or x86 based Apple Macintosh systems. A cross-platform application may run on as many as all existing platforms, or on as few as two platforms."

Wikipedia

Outline

What are they doing there?

Why cross-platform?

- Write once, run anywhere
- The technologies around are changing very rapidly
- The basic foundations of your software are changing all the time. The real challenge: *The only constant is change.* *Yes, Diogenes Laertius is very actual in the XXI century!*
- € *Money talks:* it is all about the cost of application maintenance and development of the new features
- € *Quality:* it is cheaper (in terms of bugs) to port the existing code to the new platform than to write new application from scratch with the same functionality
- € *Development speed:* it is faster to port the existing code to the new platform than to write new application with the same functionality

Why cross-platform?

- Write once, run anywhere
- The technologies around are changing very rapidly
- The basic foundations of your software are changing all the time. The real challenge: *The only constant is change. Yes, Diogenes Laertius is very actual in the XXI century!*
- € *Money talks:* it is all about the cost of application maintenance and development of the new features
- € *Quality:* it is cheaper (in terms of bugs) to port the existing code to the new platform than to write new application from scratch with the same functionality
- € *Development speed:* it is faster to port the existing code to the new platform than to write new application with the same functionality

Why cross-platform?

- Write once, run anywhere
- The technologies around are changing very rapidly
- The basic foundations of your software are changing all the time. The real challenge: *The only constant is change.* *Yes, Diogenes Laertius is very actual in the XXI century!*
- € *Money talks:* it is all about the cost of application maintenance and development of the new features
- € *Quality:* it is cheaper (in terms of bugs) to port the existing code to the new platform than to write new application from scratch with the same functionality
- € *Development speed:* it is faster to port the existing code to the new platform than to write new application with the same functionality

Why cross-platform?

- Write once, run anywhere
- The technologies around are changing very rapidly
- The basic foundations of your software are changing all the time. The real challenge: *The only constant is change.* *Yes, Diogenes Laertius is very actual in the XXI century!*
- € *Money talks:* it is all about the cost of application maintenance and development of the new features
- € *Quality:* it is cheaper (in terms of bugs) to port the existing code to the new platform than to write new application from scratch with the same functionality
- € *Development speed:* it is faster to port the existing code to the new platform than to write new application with the same functionality

Why cross-platform?

- Write once, run anywhere
- The technologies around are changing very rapidly
- The basic foundations of your software are changing all the time. The real challenge: *The only constant is change.* *Yes, Diogenes Laertius is very actual in the XXI century!*
- € *Money talks:* it is all about the cost of application maintenance and development of the new features
- € *Quality:* it is cheaper (in terms of bugs) to port the existing code to the new platform than to write new application from scratch with the same functionality
- € *Development speed:* it is faster to port the existing code to the new platform than to write new application with the same functionality

Why cross-platform?

- Write once, run anywhere
- The technologies around are changing very rapidly
- The basic foundations of your software are changing all the time. The real challenge: *The only constant is change.* *Yes, Diogenes Laertius is very actual in the XXI century!*
- € *Money talks:* it is all about the cost of application maintenance and development of the new features
- € *Quality:* it is cheaper (in terms of bugs) to port the existing code to the new platform than to write new application from scratch with the same functionality
- € *Development speed:* it is faster to port the existing code to the new platform than to write new application with the same functionality

Why cross-platform?

- Write once, run anywhere
- The technologies around are changing very rapidly
- The basic foundations of your software are changing all the time. The real challenge: *The only constant is change.* *Yes, Diogenes Laertius is very actual in the XXI century!*
- € *Money talks:* it is all about the cost of application maintenance and development of the new features
- € *Quality:* it is cheaper (in terms of bugs) to port the existing code to the new platform than to write new application from scratch with the same functionality
- € *Development speed:* it is faster to port the existing code to the new platform than to write new application with the same functionality

You probably DON'T need to develop cross-platform application if...

- Lifecycle of your application is really short
- The software you develop makes sense for the only and very unique hardware in the earth
- € *Business decision:* to develop only for the selected platform. No future is planned

You probably DON'T need to develop cross-platform application if...

- Lifecycle of your application is really short
- The software you develop makes sense for the only and very unique hardware in the earth
- € *Business decision:* to develop only for the selected platform. No future is planned

You probably DON'T need to develop cross-platform application if...

- Lifecycle of your application is really short
- The software you develop makes sense for the only and very unique hardware in the earth
- € *Business decision:* to develop only for the selected platform. No future is planned

You probably DO need to develop cross-platform application if...

- One of your business requirements is to support more than one computer platform
- The lifecycle of your application lasts longer than the technologies it is based on. You know that in 1 year from now the primary computer platform for your application will be different from the current one. Your application shall be prepared and investments of your company shall be protected by the engineering decisions you are making

You probably DO need to develop cross-platform application if...

- One of your business requirements is to support more than one computer platform
- The lifecycle of your application lasts longer than the technologies it is based on. You know that in 1 year from now the primary computer platform for your application will be different from the current one. Your application shall be prepared and investments of your company shall be protected by the engineering decisions you are making

Outline

OK. Let's hope these boys know where they are...

The tools

Variety of tools is available:

- Java
- Scripting languages: perl, python, ruby, ...
- “Classical” compilers: C, C++, Pascal, Haskell (GHC), ...

Let us focus on the third approach and discuss the C++ way in depth.

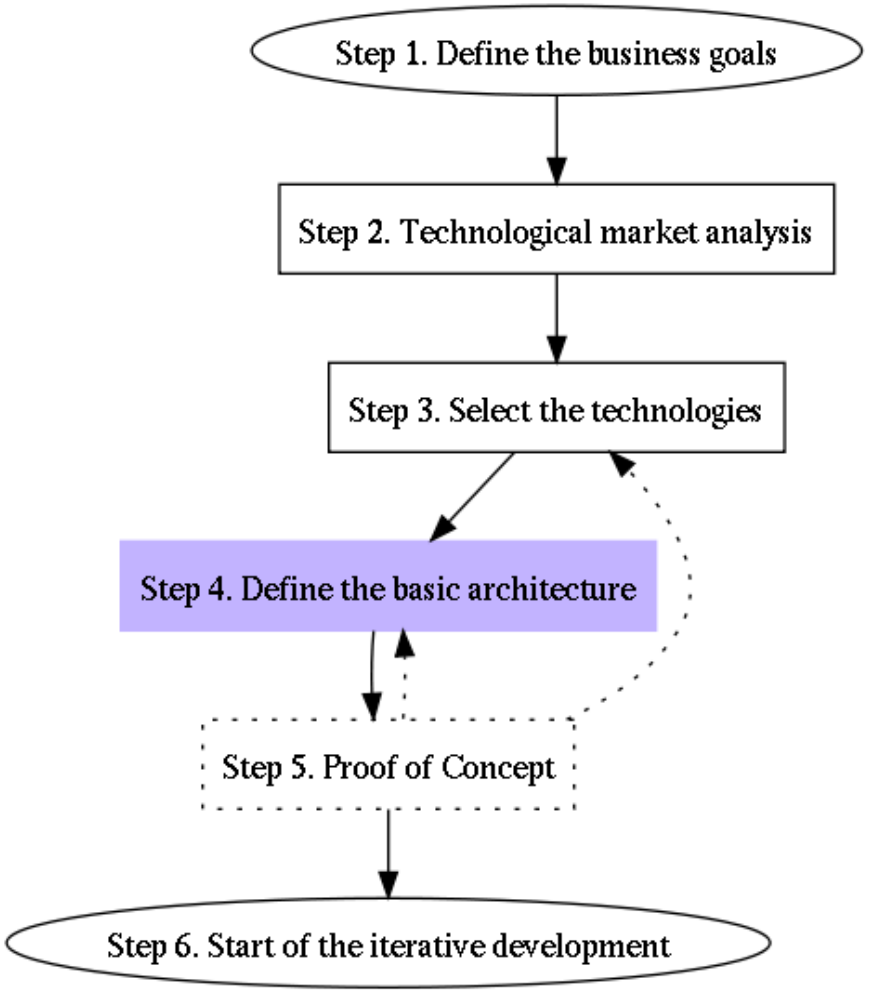
The tools

Variety of tools is available:

- Java
- Scripting languages: perl, python, ruby, ...
- “Classical” compilers: C, C++, Pascal, Haskell (GHC), ...

Let us focus on the third approach and discuss the C++ way in depth.

The typical steps



Step 1. € Define the business goals

- What is the business logic of your application?
- Define the (high-level) content, most important business requirements
- What features of the product you develop are unique compared with the competing products?
- What are the portability requirements? Is the business ready to invest resources into the portability?

Step 2. Technological market analysis

- What is available in the market (including open-source free "market") RIGHT NOW?
- What technologies are available to build the product?

The main question which shall be answered in this step — *What can be reused in order to build the product without reinventing the wheel?*

The most dangerous risk here is to miss the existing technologies or products and reinvent the wheel. The end-result can be total loss of the money invested. And your own life time which is even more important!

Step 2. Technological market analysis

- What is available in the market (including open-source free "market") RIGHT NOW?
- What technologies are available to build the product?

The main question which shall be answered in this step — *What can be reused in order to build the product without reinventing the wheel?*

The most dangerous risk here is to miss the existing technologies or products and reinvent the wheel. The end-result can be total loss of the money invested. And your own life time which is even more important!

Step 3. Select the technologies

The helpful questions:

- What are the APIs my application will use?
- What tools (compilers, debuggers, documentation tools, toolkits etc) shall be used?
- How portable are the technologies my application will be based on?

Step 3. Select the technologies

The helpful questions:

- What are the APIs my application will use?
- What tools (compilers, debuggers, documentation tools, toolkits etc) shall be used?
- How portable are the technologies my application will be based on?

Step 3. Select the technologies

The helpful questions:

- What are the APIs my application will use?
- What tools (compilers, debuggers, documentation tools, toolkits etc) shall be used?
- How portable are the technologies my application will be based on?

Too far view can be misleading. But in the same time the distance is needed to see the “big” picture

Technological landscape analysis

This phase is extremely important (and extremely difficult and risky) if your aim is to make an application which is prepared for the FUTURE. And this is the usual story in the modern word...

- What technologies are used?
- What APIs are used?
- What is going to be changed in the forecastable future? E.g., UI tends to change more often then backends. Typically. But everything depends on the project!
- Try to be Cassandra for the destiny of the technologies and APIs your application uses. What is their future? Use all the available information - insider's knowledge, expert's opinions, keep an eye on the researching projects. You shall be aware of the university activities in order to be prepared for the technology switch.

Try to forecast. To predict the future.

Technological landscape analysis

This phase is extremely important (and extremely difficult and risky) if your aim is to make an application which is prepared for the FUTURE. And this is the usual story in the modern word...

- What technologies are used?
- What APIs are used?
- What is going to be changed in the forecastable future? E.g., UI tends to change more often then backends. Typically. But everything depends on the project!
- Try to be Cassandra for the destiny of the technologies and APIs your application uses. What is their future? Use all the available information - insider's knowledge, expert's opinions, keep an eye on the researching projects. You shall be aware of the university activities in order to be prepared for the technology switch.

Try to forecast. To predict the future.

Technological landscape analysis

This phase is extremely important (and extremely difficult and risky) if your aim is to make an application which is prepared for the FUTURE. And this is the usual story in the modern word...

- What technologies are used?
- What APIs are used?
- What is going to be changed in the forecastable future? E.g., UI tends to change more often then backends. Typically. But everything depends on the project!
- Try to be Cassandra for the destiny of the technologies and APIs your application uses. What is their future? Use all the available information - insider's knowledge, expert's opinions, keep an eye on the researching projects. You shall be aware of the university activities in order to be prepared for the technology switch.

Try to forecast. To predict the future.

Technological landscape analysis

This phase is extremely important (and extremely difficult and risky) if your aim is to make an application which is prepared for the FUTURE. And this is the usual story in the modern word...

- What technologies are used?
- What APIs are used?
- What is going to be changed in the forecastable future? E.g., UI tends to change more often then backends. Typically. But everything depends on the project!
- Try to be Cassandra for the destiny of the technologies and APIs your application uses. What is their future? Use all the available information - insider's knowledge, expert's opinions, keep an eye on the researching projects. You shall be aware of the university activities in order to be prepared for the technology switch.

Try to forecast. To predict the future.

Cross-platform enablers

- Divide the (future) application into the packages.
- Put the portability requirements towards every package. E.g., “package A shall be portable to every system which has gcc and glibc”, “package B shall be portable to every system which has Qt core libraries”, “package C will be reimplemented for every system. APIs shall be carefuly designed”.
- Keep in mind - stricter portability requirements mean more efforts to implement specific component.

Cross-platform enablers

- Divide the (future) application into the packages.
- Put the portability requirements towards every package. E.g., “package A shall be portable to every system which has gcc and glibc”, “package B shall be portable to every system which has Qt core libraries”, “package C will be reimplemented for every system. APIs shall be carefuly designed”.
- Keep in mind - stricter portability requirements mean more efforts to implement specific component.

Cross-platform enablers

- Divide the (future) application into the packages.
- Put the portability requirements towards every package. E.g., “package A shall be portable to every system which has gcc and glibc”, “package B shall be portable to every system which has Qt core libraries”, “package C will be reimplemented for every system. APIs shall be carefuly designed”.
- Keep in mind - stricter portability requirements mean more efforts to implement specific component.

Analisys

- Do a classical object-oriented decomposition keeping portability requirements in mind
- Carefully analyze the dependencies your application has. What applications layers can be changed in future? Try to be Cassandra.

The risk — to make the architecture more complex than needed. To reinvent glibc.

Analisys

- Do a classical object-oriented decomposition keeping portability requirements in mind
- Carefully analyze the dependencies your application has. What applications layers can be changed in future? Try to be Cassandra.

The risk — to make the architecture more complex than needed. To reinvent glibc.

Step 4. Define the basic architecture

- Isolate the dependencies through internal APIs (set of classes)
- Document in the paper the most critical architectural decisions (including portability requirements towards each component)
- Agree on the basic architecture with your team and other experts you can reach. This is important step as it allows to find teh design problems in the very early phase. Even before the development starts! Never sit in the "architectural ivory tower"!
- € If needed, make a prototype. It shall be FAST prototyping. It can not take "1 man month".

Step 5. (Optional) Make Proof-of-Concept.

- Prototype your application. Try the most risky design ideas!
- The goal is to check that the creative ideas suggested in the previous step really work well.

Step 6. Start of the (iterative) application development

- Start the development based on the architecture agreed with the team
- Architecture is not a dogma. It could be changed if the development shows that it does not suit the business needs
- Regulary check that your application is really portable. Just build it for 2 platforms at least. And run the unit tests.

Step 6. Start of the (iterative) application development

- Start the development based on the architecture agreed with the team
- Architecture is not a dogma. It could be changed if the development shows that it does not suit the business needs
- Regulary check that your application is really portable. Just build it for 2 platforms at least. And run the unit tests.

Step 6. Start of the (iterative) application development

- Start the development based on the architecture agreed with the team
- Architecture is not a dogma. It could be changed if the development shows that it does not suit the business needs
- Regulary check that your application is really portable. Just build it for 2 platforms at least. And run the unit tests.

Outline

Don't be afraid! Just do!

Qt Messaging framework. Introduction.

- Cross-platform (Linux, Mac™, Maemo™, MS Windows™) C++ middleware to build email clients, and more generally software that interacts with messaging servers.
- Based on Qt
- Platform-dependent stuff is isolated in the corresponding classes which can have different implementation for different platforms.
- Extensible by protocol and content storage plugins. E.g., plugin to access your messages box in social network, can be easily developed
- Open-sourced: http://qt.nokia.com/doc/status/qdoc-output/public-messagingframework/html/

Qt Messaging framework. Introduction.

- Cross-platform (Linux, Mac™, Maemo™, MS Windows™) C++ middleware to build email clients, and more generally software that interacts with messaging servers.
- Based on Qt
- Platform-dependent stuff is isolated in the corresponding classes which can have different implementation for different platforms.
- Extensible by protocol and content storage plugins. E.g., plugin to access your messages box in social network, can be easily developed
- Open-sourced: http://qt.nokia.com/doc/status/qdoc-output/public-messagingframework/html/

The business goals

- Messaging subsystem engine. The reference message type is “e-mail”
- Shall make it possible to integrate the new 3-party (!) protocols into the subsystem
- Easy to use, object–oriented
- Shall make it possible to store the messages in the different ways (databases, files of different formats and so on)
- *Portability requirements: the same as for Qt*

The business goals

- Messaging subsystem engine. The reference message type is “e-mail”
- Shall make it possible to integrate the new 3-party (!) protocols into the subsystem
- Easy to use, object–oriented
- Shall make it possible to store the messages in the different ways (databases, files of different formats and so on)
- *Portability requirements: the same as for Qt*

The business goals

- Messaging subsystem engine. The reference message type is “e-mail”
- Shall make it possible to integrate the new 3-party (!) protocols into the subsystem
- Easy to use, object–oriented
- Shall make it possible to store the messages in the different ways (databases, files of different formats and so on)
- *Portability requirements: the same as for Qt*

The business goals

- Messaging subsystem engine. The reference message type is “e-mail”
- Shall make it possible to integrate the new 3-party (!) protocols into the subsystem
- Easy to use, object–oriented
- Shall make it possible to store the messages in the different ways (databases, files of different formats and so on)
- *Portability requirements: the same as for Qt*

The business goals

- Messaging subsystem engine. The reference message type is “e-mail”
- Shall make it possible to integrate the new 3-party (!) protocols into the subsystem
- Easy to use, object–oriented
- Shall make it possible to store the messages in the different ways (databases, files of different formats and so on)
- *Portability requirements: the same as for Qt*

Technological market analysis

Historical retrospective (several years ago)

- Qt is in place and contains a lot of reusable classes to implement the backend
- No e-mail protocol backend implementation available for reuse
- Relational database can be used to store email headers and implement effective local search through the messages databases

Select the technologies

- APIs: networking (including SSL), SQL (sqlite), file access, synchronization primitives. Use through Qt classes whenever it is possible.
- Documentation system: qdoc3 (as in Qt)
- Portability: all the major desktop platforms. Designed with being cross-platform in mind. Shall be easily ported to future computer platforms

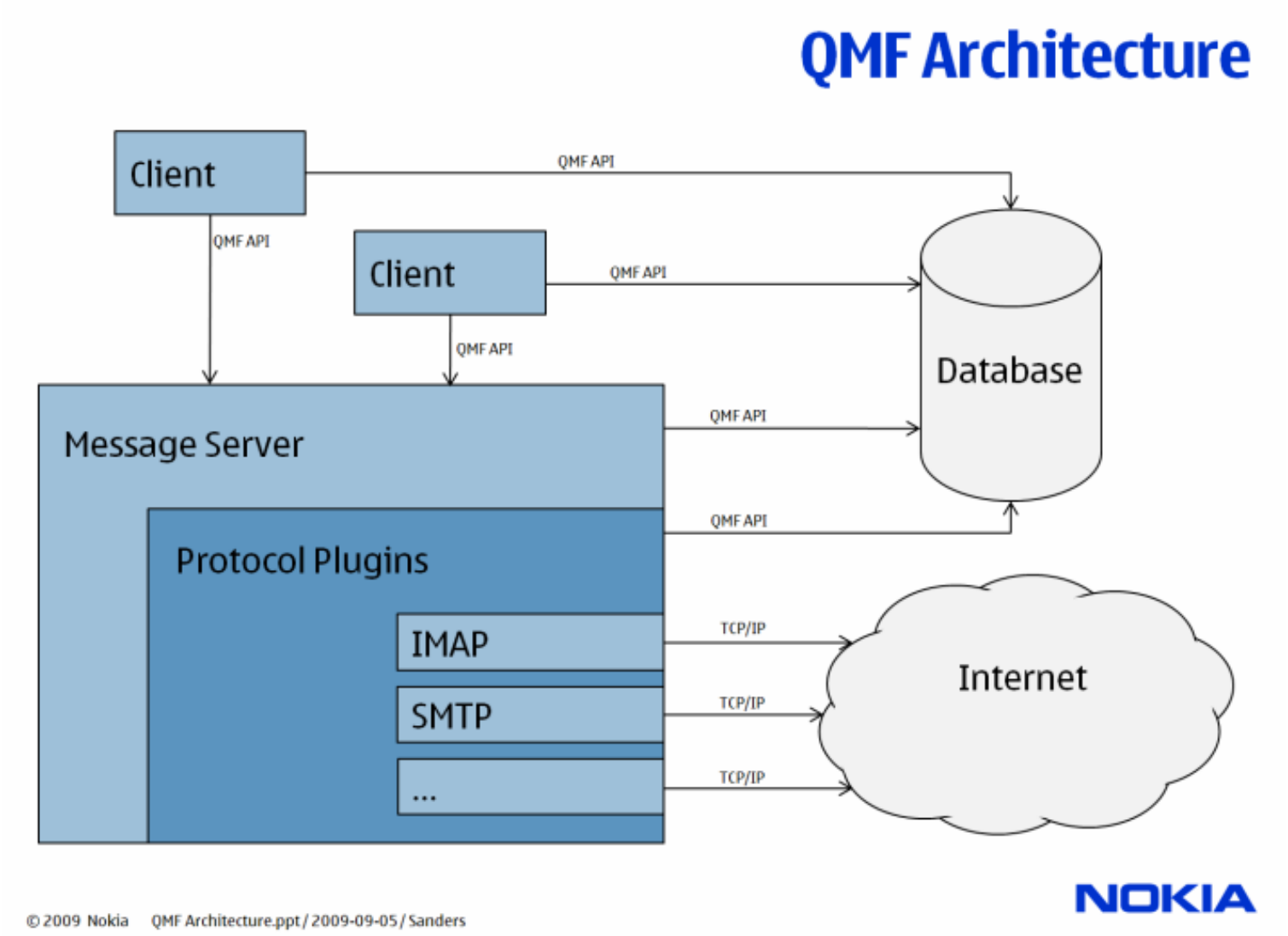
Select the technologies

- APIs: networking (including SSL), SQL (sqlite), file access, synchronization primitives. Use through Qt classes whenever it is possible.
- Documentation system: qdoc3 (as in Qt)
- Portability: all the major desktop platforms. Designed with being cross-platform in mind. Shall be easily ported to future computer platforms

Select the technologies

- APIs: networking (including SSL), SQL (sqlite), file access, synchronization primitives. Use through Qt classes whenever it is possible.
- Documentation system: qdoc3 (as in Qt)
- Portability: all the major desktop platforms. Designed with being cross-platform in mind. Shall be easily ported to future computer platforms

Define the basic architecture



Define the basic architecture

- Run-time executable: messageserver
- Mechanism to support 3-party protocols: plugins loadable into the messageserver in runtime
- Clear separation between UI and backend
- Use the C++ PIMPL idiom to encapsulate the platform-dependent stuff. Make porting efforts easier for the maintainers
- Use the Model/view paradigm to separate data and representation

Define the basic architecture

- Run-time executable: messageserver
- Mechanism to support 3-party protocols: plugins loadable into the messageserver in runtime
- Clear separation between UI and backend
- Use the C++ PIMPL idiom to encapsulate the platform-dependent stuff. Make porting efforts easier for the maintainers
- Use the Model/view paradigm to separate data and representation

Define the basic architecture

- Run-time executable: messageserver
- Mechanism to support 3-party protocols: plugins loadable into the messageserver in runtime
- Clear separation between UI and backend
- Use the C++ PIMPL idiom to encapsulate the platform-dependent stuff. Make porting efforts easier for the maintainers
- Use the Model/view paradigm to separate data and representation

Define the basic architecture

- Run-time executable: messageserver
- Mechanism to support 3-party protocols: plugins loadable into the messageserver in runtime
- Clear separation between UI and backend
- Use the C++ PIMPL idiom to encapsulate the platform-dependent stuff. Make porting efforts easier for the maintainers
- Use the Model/view paradigm to separate data and representation

Define the basic architecture

- Run-time executable: messageserver
- Mechanism to support 3-party protocols: plugins loadable into the messageserver in runtime
- Clear separation between UI and backend
- Use the C++ PIMPL idiom to encapsulate the platform-dependent stuff. Make porting efforts easier for the maintainers
- Use the Model/view paradigm to separate data and representation

Iterative application development

- The architecture was changing with the project evolution but the major architecture ideas are still valid
- New capabilities were added to the plugins APIs during the QMF evolution
- It was possible to port QMF to desktop and mobile platforms with different software architectures.

Iterative application development

- The architecture was changing with the project evolution but the major architecture ideas are still valid
- New capabilities were added to the plugins APIs during the QMF evolution
- It was possible to port QMF to desktop and mobile platforms with different software architectures.

Iterative application development

- The architecture was changing with the project evolution but the major architecture ideas are still valid
- New capabilities were added to the plugins APIs during the QMF evolution
- It was possible to port QMF to desktop and mobile platforms with different software architectures.

Thanks a lot for your attention!

Thank you!

Questions?

vitaly.repin@nokia.com