Leaving The Dark Side

A story with code about 5 years learning

- Developing a C++ Based Medical Device, Successful Again -

Prepared for the ACCU 2016

by

Felix Petriconi

About me

Study of electrical engineering

Since 1993 working as a programmer

- At the university (Turbo Pascal, Ada, C++)
- Education of high gifted children (PovRay, Pascal, C++)
- 7 years as freelancer (C/C++, Perl)
- Since 2003 employed as programmer by MeVis Medical Solutions AG, Bremen, Germany (C++, x86 Assembler, Ruby)

About the team today

Department size: 34

3 Product Owner

4 SCRUM Teams

- 4-5 Developers
- 1 Test Engineer
- 1 Requirement/Usability Engineer
- ½ SCRUM Master

1 Test-Lab Team

- 5 Test Engineers

About our product

Reviewing workstation for mammography images

Manufactured for a single OEM customer

Medical device => regulated environment

In the market since 2002

About 7000 installations world wide

Market share in that segment > 50%

Our product

About the application

Deployed as standalone / client-server

OS: Windows 7 / Server 2008 R2

C++ / Qt application

2 million lines of code

About the technical challenges

Huge variety of hospital setups

Radiologists must be able to read about 120 patients / h

Up to 4 GB uncompressed pixel data for a single patient

Up to 400 patients per day

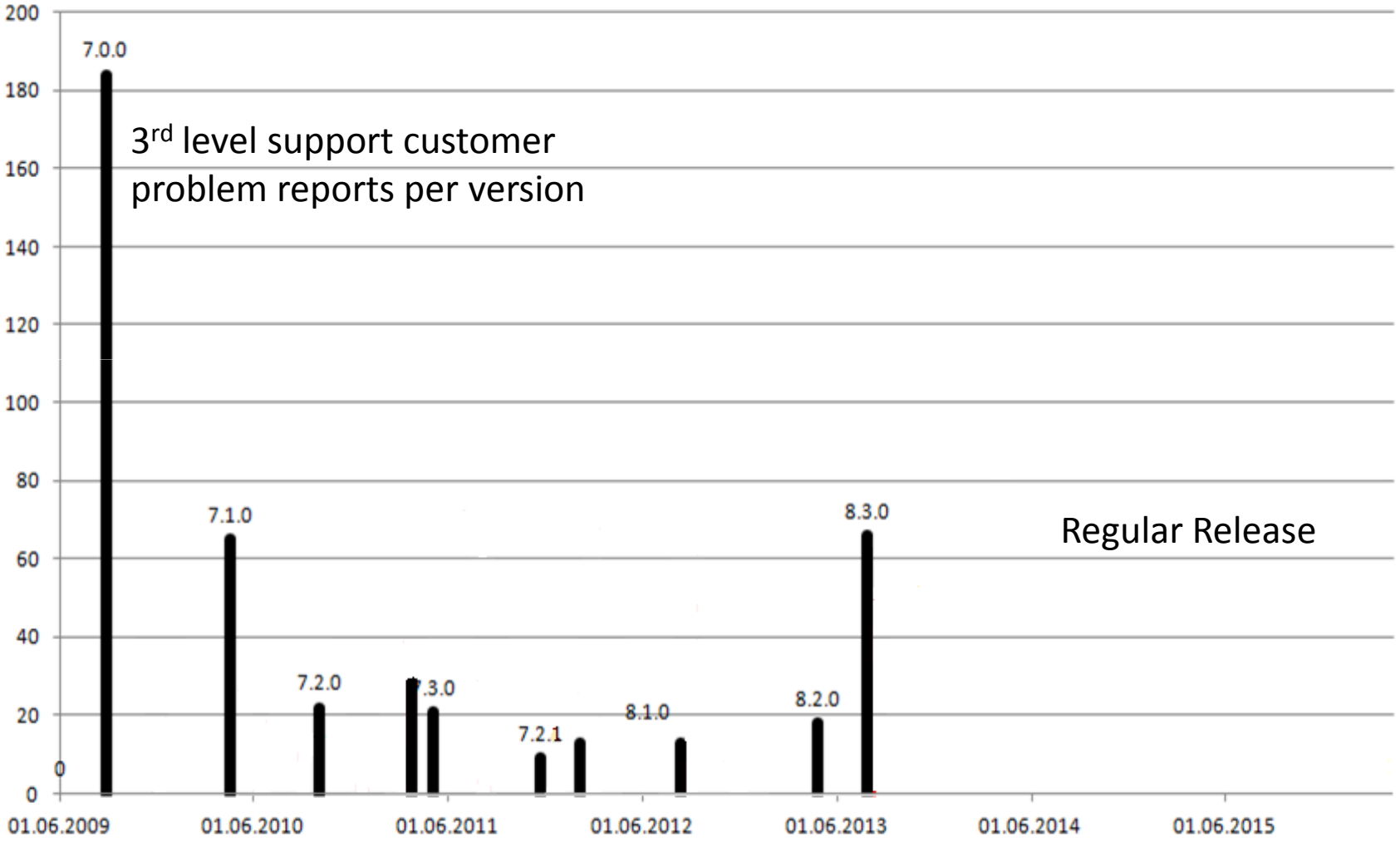
8-16 bit grayscale images (16 - 800MB) on 2 * 5MP 10 bit grayscale displays

Of the shelve workstations (no special HW possible)

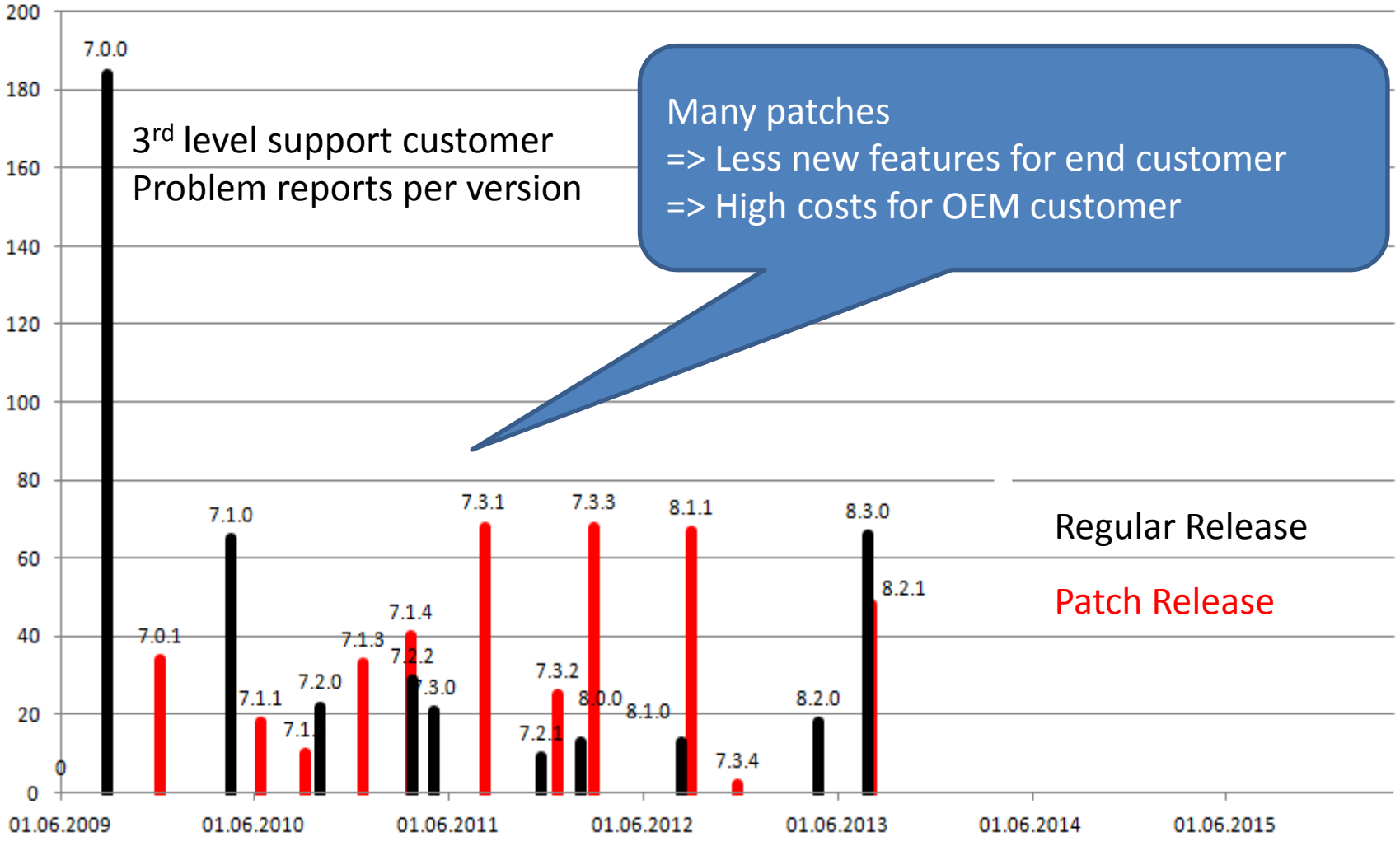
Server with up to 24 clients

Each case change, image change < 1s on every client

About our problem



About our problem



Some reasons

At 2011 about 10,000 requirements in a requirement management tool

Each requirement had to be traced to a test case

Only paper scripts existed to test the application

Each release test phase took up to 8-12 weeks

High number of bugs

Lack of training of the team

Our way out

Use people for intelligent work

Let machines perform dumb work

Change the development process from Waterfall to SCRUM (Problem: All regulatory documents are written with Waterfall process in mind)

Invest in engineering education

Invest in test automation

Engineering education

Developer training with educational videos

Creating a library of books

Regular conference visits for software engineering

Introduction of Community of Practice every week

- Regular Dev-Talks

ISTQB Training

- Base for all team members
- Advanced for all test engineers

Process investments

Buy in of our customer and upper management

Introduction of SCRUM (started self educated)

External SCRUM coaching over several weeks

Introducing MeVis 10% (“Crazy Fridays”)

Introducing of “Continuous Integration”

External SCRUM coaching follow-up after one year

New test strategy evolved

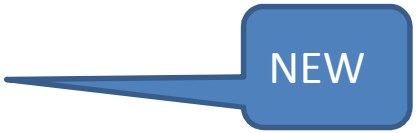
Less manual functional tests

- Primary focus now on exploratory tests

UnitTests

Automated UI tests

Behaviour tests

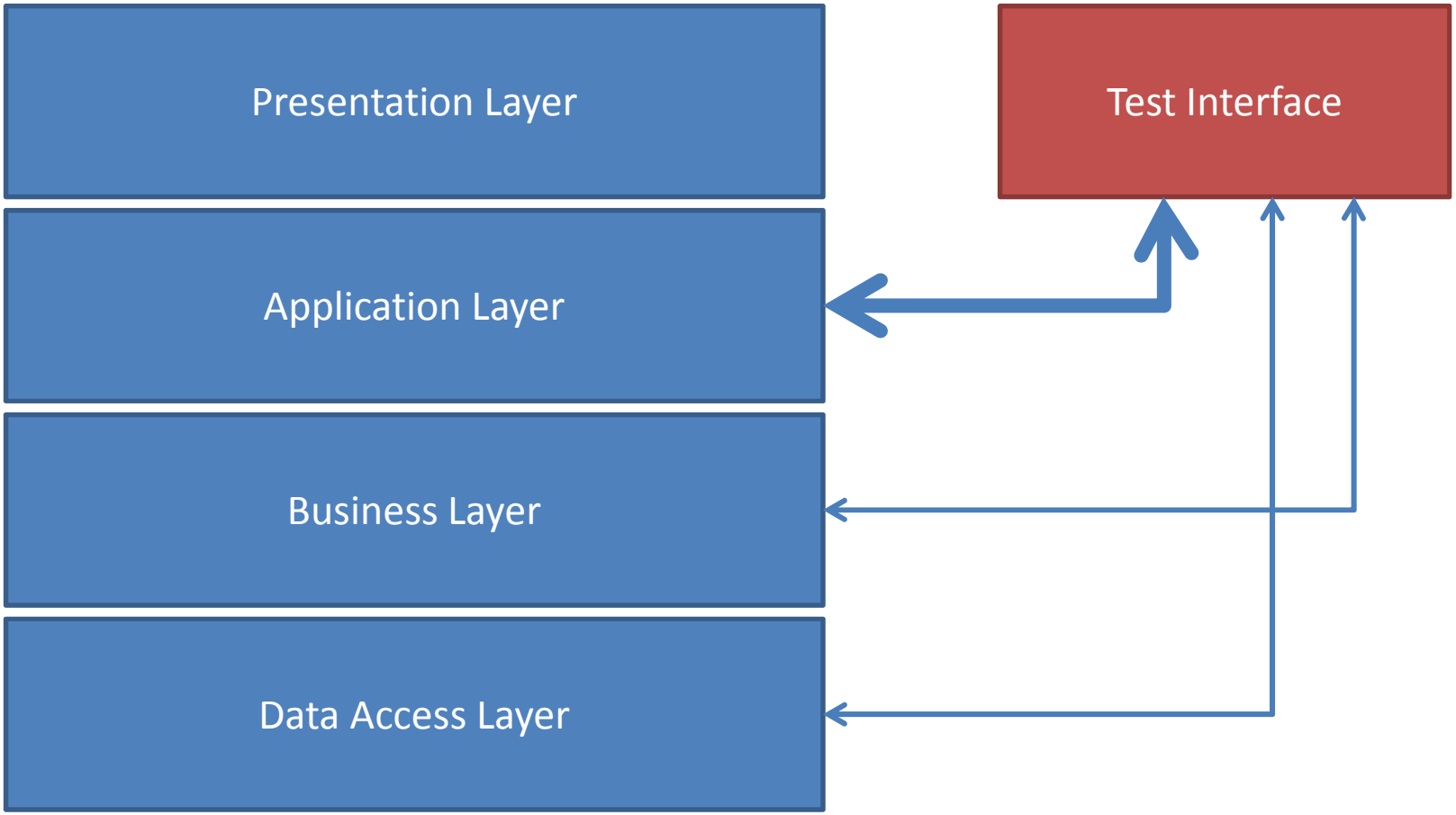


Behaviour tests

End to End Tests

Don’t test through the UI

Where to inject behaviour tests?



Behaviour tests

Started with Specification by Example with Cucumber

```
Given the login dialog is visible
When a registered user provides
  username and password
Then the user is logged in
And the administration module is
  available
```

Which Cucumber binding?

Native C++ binding (cukebins) could not be used, because our application runs with multiple processes on multiple machines

⇒ Nothing out of the box was available

⇒ Customization necessary

⇒ Cucumber with Ruby binding was the natural choice

Behaviour tests with Cucumber?

Started very promising

But the tool Cucumber was not capable of handling nested contexts inside a test

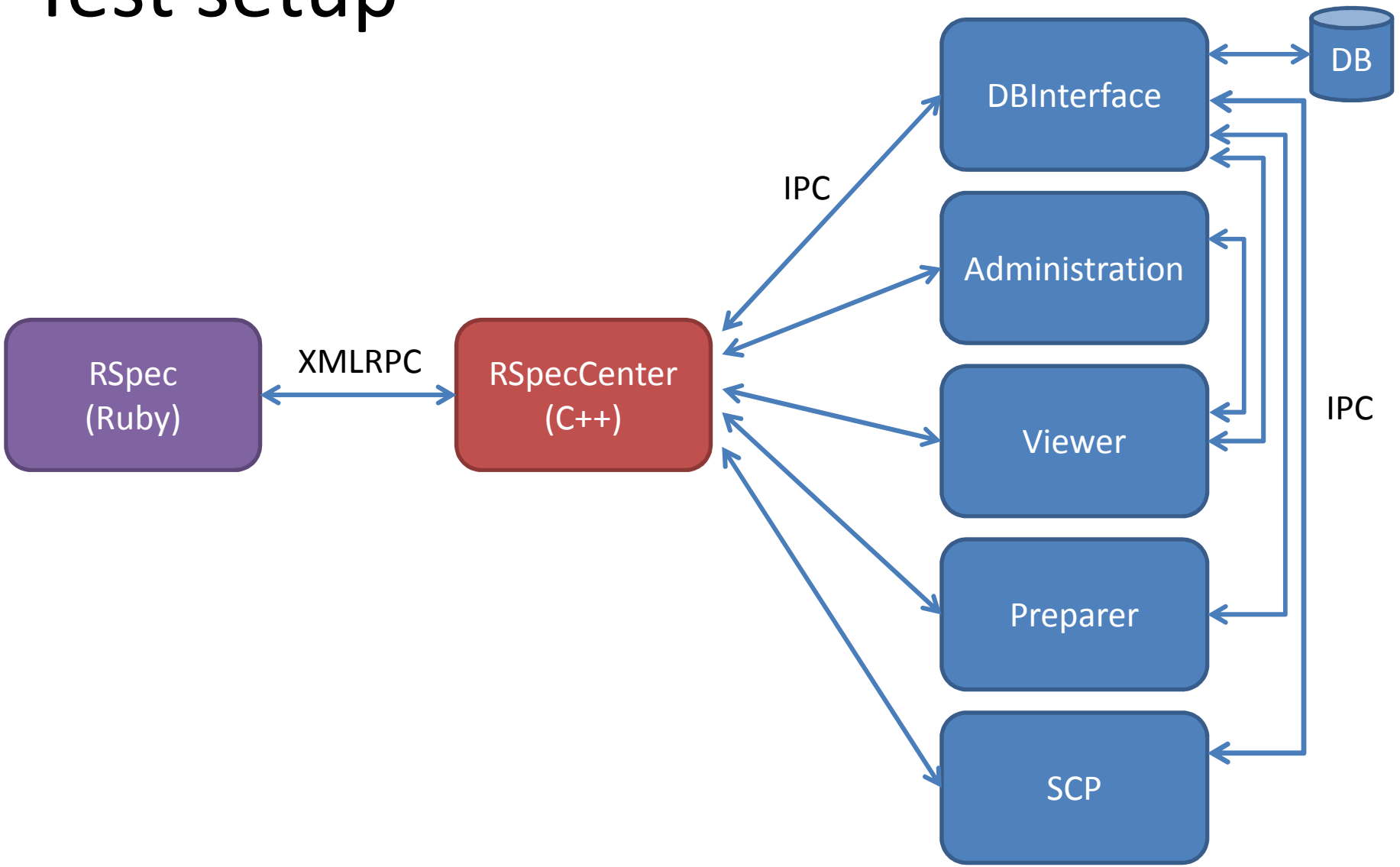
Required intensive collaboration with Product Owner

Examples were too complicated and could not serve as a specification because of the complexity of the domain

⇒New approach with **RSpec** (Predecessor of Cucumber)

⇒The remaining test infrastructure could be the same

Test setup



Let's write a simple test

```
describe 'Login mechanism' do
  context 'When the login dialog is available' do
    before (:all) do
      administration.waitUntilLoginIsVisible()
    end

    context 'And the user logs into the application' do
      before (:all) do
        administration.login("user1", "password4user1")
      end

      it 'Then th  administr ion module is a ilable for the user' do
        administr ion.waitUn lAdministrationI isible()
      end
    end
  end
end
```

Representative of Administration process

Test method in Administration process

Parameters of login method

Feaze the Ruby part …

For each process a representative Ruby object exists

Ruby's `method_missing` feature is used to "generate" methods on the fly. So there is no need to specify all possible test methods manually (more code in the bonus slides)

XMLRPC protocol

```
<methodCall>
  <methodName>rspeccommand</methodName>
  <params>
    <param><value><string>ADMISTRATION</string></value></param>
    <param><value><string>login</string></value></param>
    <param><value><i4>60</i4></value></p
    <param><value>
      <array><data>
        <value><string>user1</string>
        <value><string>password4user1</st
      </data></array>
    </value></param>
  </params>
</methodCall>
```

RPC name

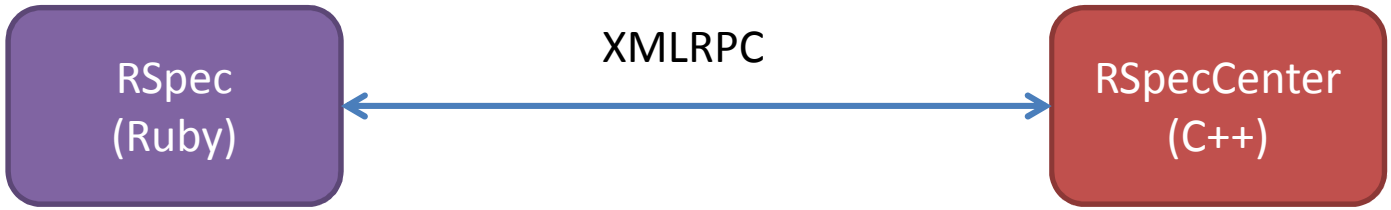
Process name

Test method name in the process

Command timeout /s

Array with all method parameters

RSpecCenter




```
<methodCall>
  <methodName>scrcukecommand</methodName>
  <params>
    <param><value><string>ADMISTRATION</string></value></param>
    <param><value><string>login</string></value></param>
    <param><value><i4>60</i4></value></param>
    <param><value>
      <array><data>
        <value><string>user1</string></value>
        <value><string>password4user1</string></value>
      </data></array>
    </value></param>
  </params>
</methodCall>
```

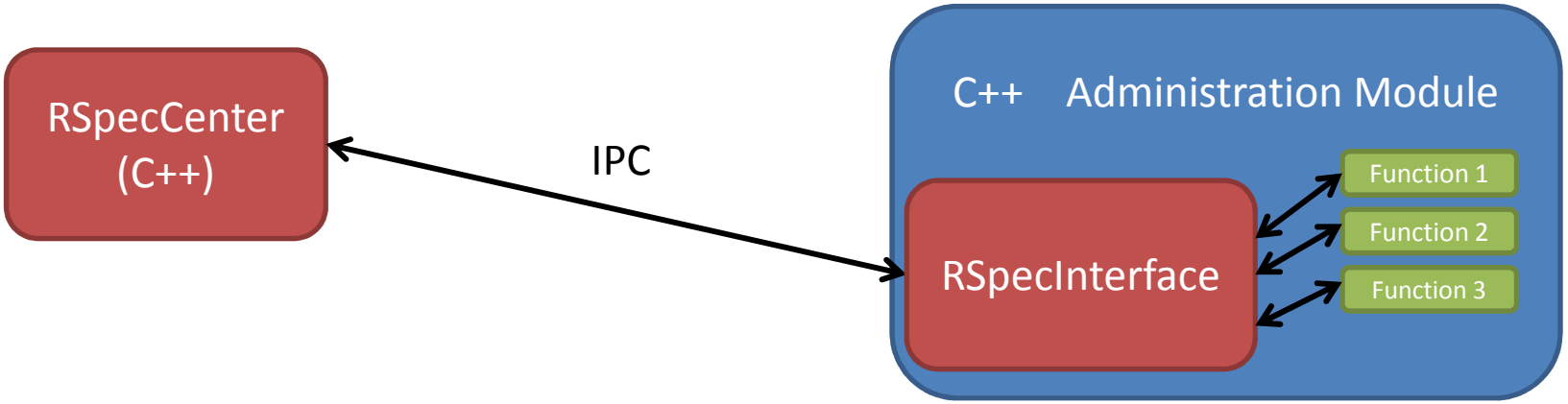
Target process lookup

Converts XMLRPC command payload to application specific binary IPC protocol

Limited list of supported types: string, int, double, bool, array, hash

Any nested combination of these types is possible

RSpecInterface



Each process has a RSpecInterface instance

It registers for a dedicated IPC callback

Starts to parse the binary stream and extracts method name

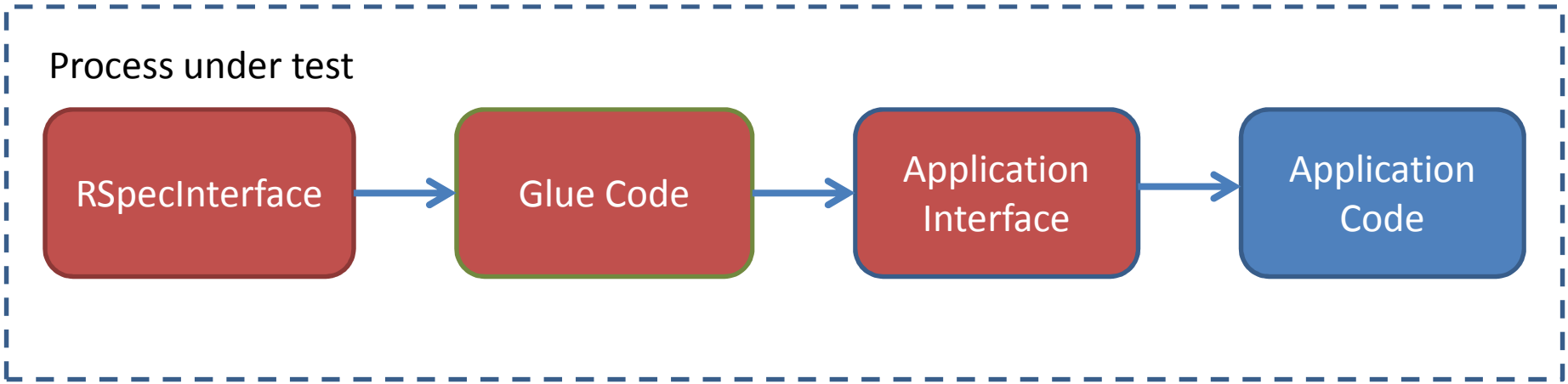
Lookup of registered test method

Calls method with remaining in-stream (Source) and returns new values in out-stream (Sink)

Application test interface

```
class AdminstrationInterface
{
public:
  void userLogin(const std::string& userName,
                 const std::string& password);
  void logout();

  CommandResult waitUntilLoginIsVisible();

  CommandResult waitUntilAdministrationIsVisible();

  static AdministrationInterface s_interface;
};
```

Execution chain inside application



Glue code

At the beginning written by hand, later created within the build process by a code generator

```
// defining the test function
void login(const Source& source, Sink& sink);

// registering the function and its name with a registrar
CommandRegistrar(login, "login");


// implementation of the test function
void login(const Source& source, Sink& sink)
{
  auto userName = createFromSource<std::string>(source);
  auto password = createFromSource<std::string>(source);

  s_interface.userLogin(userName, password);
}
```

When to proceed?

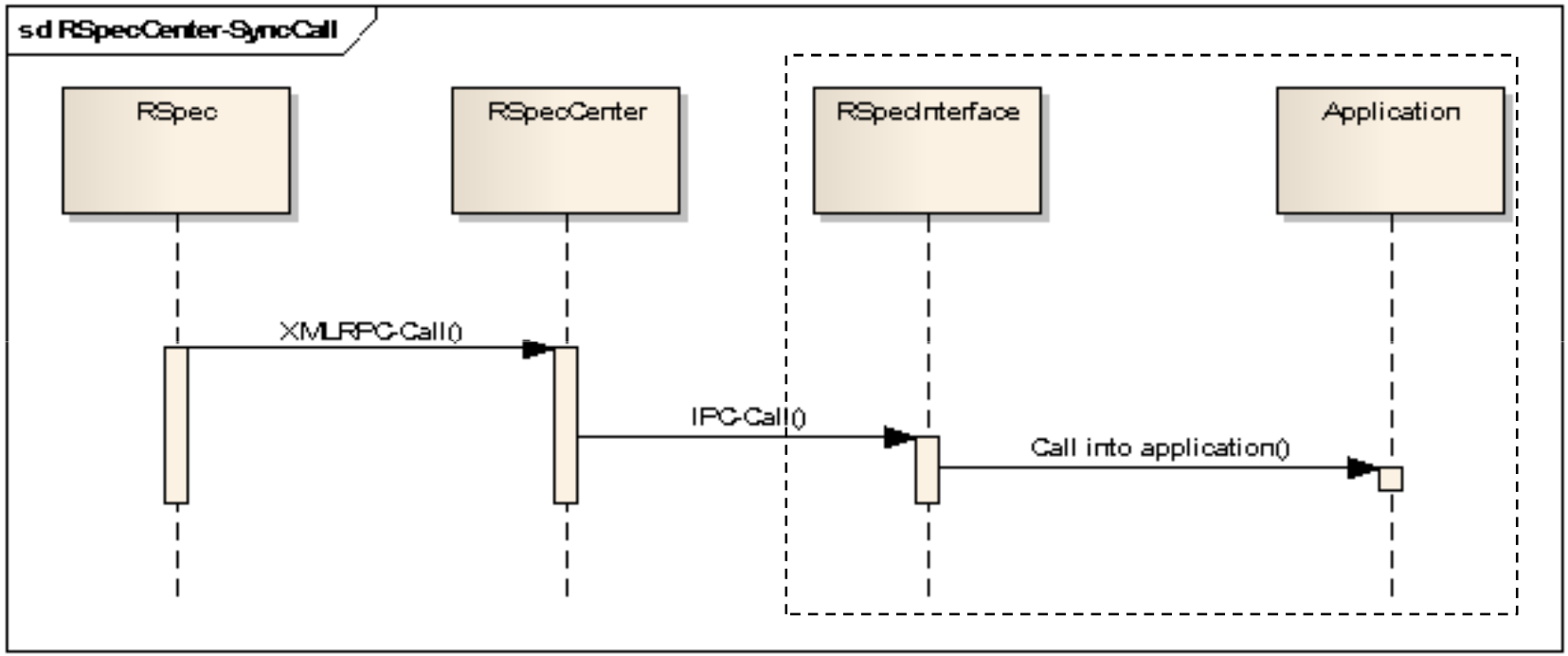
Many things in the application happen asynchronously

Add sleep call into the test script

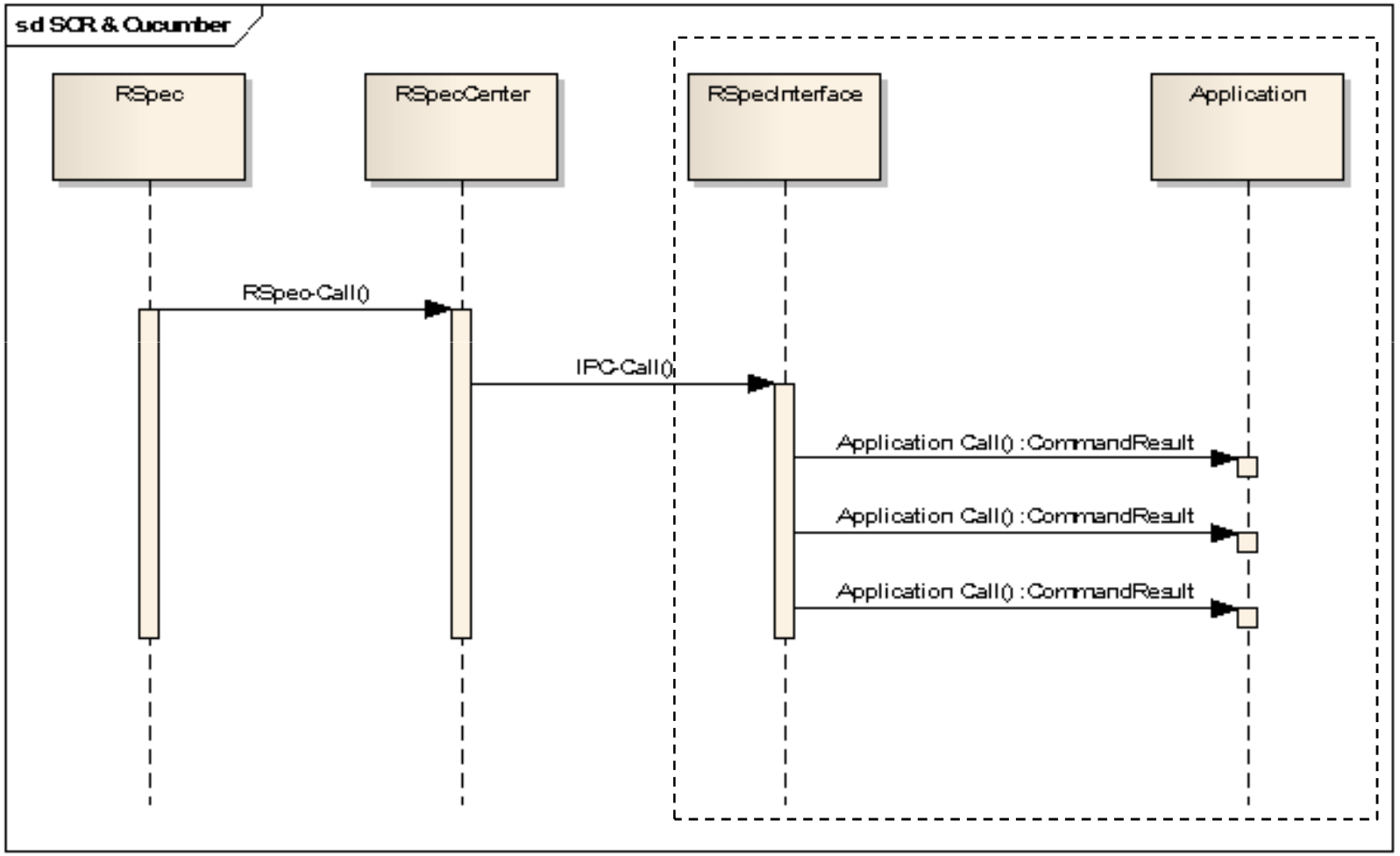
Callback from the application into the test could be an option, but would make the application depend on the test

RSpecInterface polls with short interval (50ms) until a certain condition is reached or the command timed out

Synchronous command

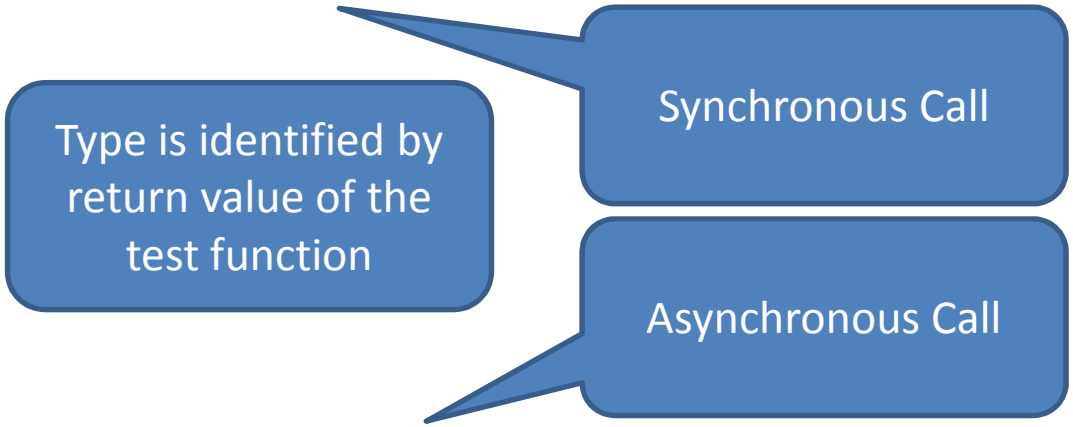


Asynchronous command



Test functions

```
void AdminstrationInterface::userLogin(const std::string& userName,
                                       const std::string& password);
```



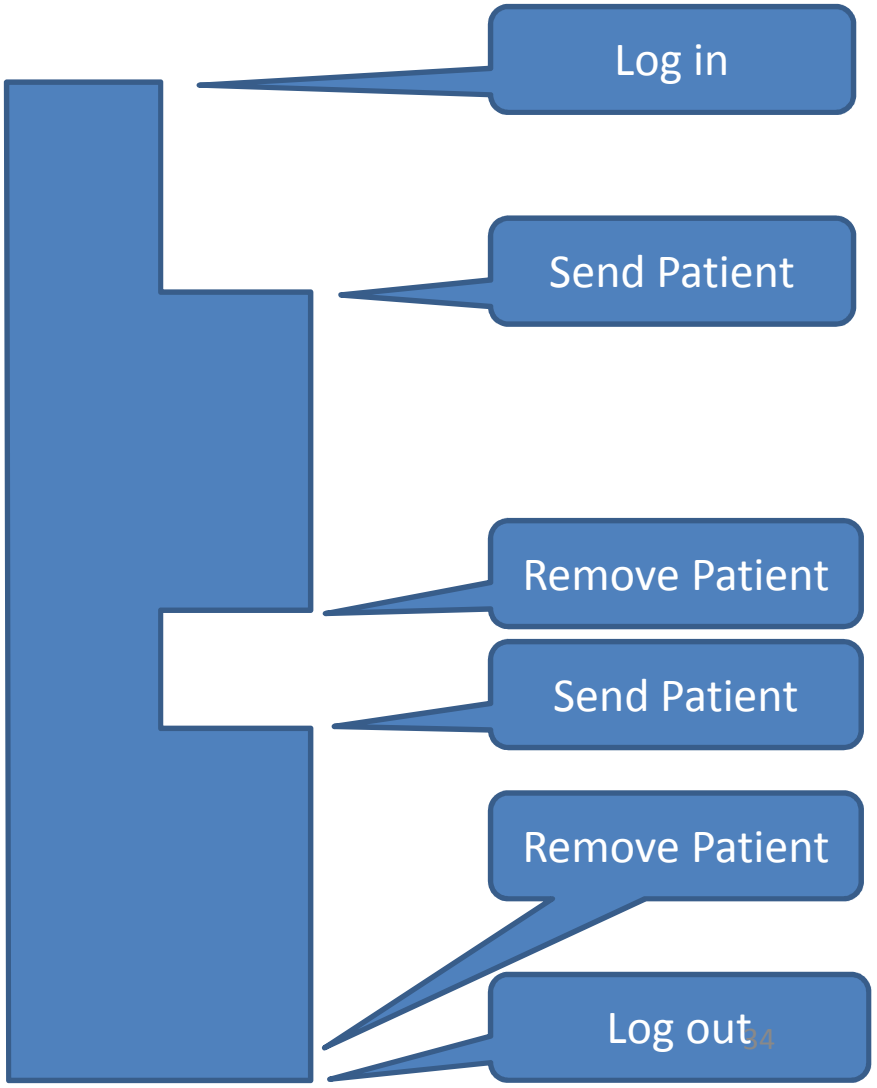

```
CommandResult
   AdminstrationInterface::waitUntilAdministrationIsVisible();

enum class CommandResult
{
  Success, // when the condition is fulfilled
  Failed,  // when the condition cannot be fulfilled (anymore)
  Pending  // when the condition is not yet fulfilled
};
```

Asynchronous test function

```
CommandResult
AdminstrationInterface::waitUntilAdministrationIsVisible()
{
  if (administrationModule().isVisible())
  {
    return CommandResult::Success;
  }
  return CommandResult::Pending;
}
```

Scoped test contexts in RSpec

```
describe 'foo' do
  before(:all) do
    login_scoped("name", "password")
  end
  describe '1st test scenario' do
    before(:all) do
      send_patient_scoped("TestPatient_A")
    end
    it 'bar 1' do
      # perform check
    end
  end
  describe '2nd test scenario' do
    before(:all) do
      send_patient_scoped("TestPatient_B")
    end
    it 'bar 2' do
      # perform check
    end
  end
end
```



“RAII” within RSpec

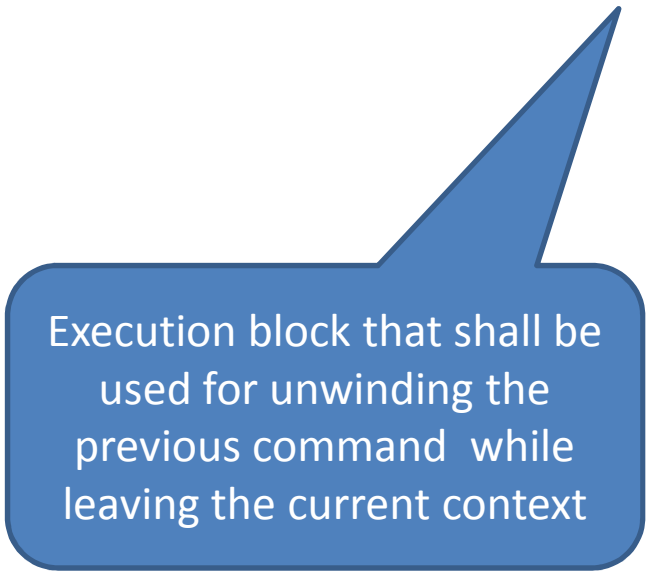
```
require 'cleaner'
require 'rspec'

module RSpec
  module Core
    class ExampleGroup
      class << self
        alias old_set_it_up set_it_u

        def set_it_up(*args)
          old_set_it_up(*args)

          hooks.register(:append, :before, :all)
            { cleaner.set_mark }
          hooks.register(:append, :after, :all)
            { cleaner.clean_up_till_last_mark }
        end
      end
    end
  end
end
```

Register ‘cleaner’ in RSpec hooks

The ‘cleaner’ implements a stack that can get execution blocks pushed at and those are separated per context with a special marker

Scope function example

```
def login_scoped(user, password) do
    administration.login(“user1”, “password4user1”)
    cleaner.push_action( {administration.logout()} )
end
```



Helpful additions

Log the complete I/O stream of the RSpecCenter

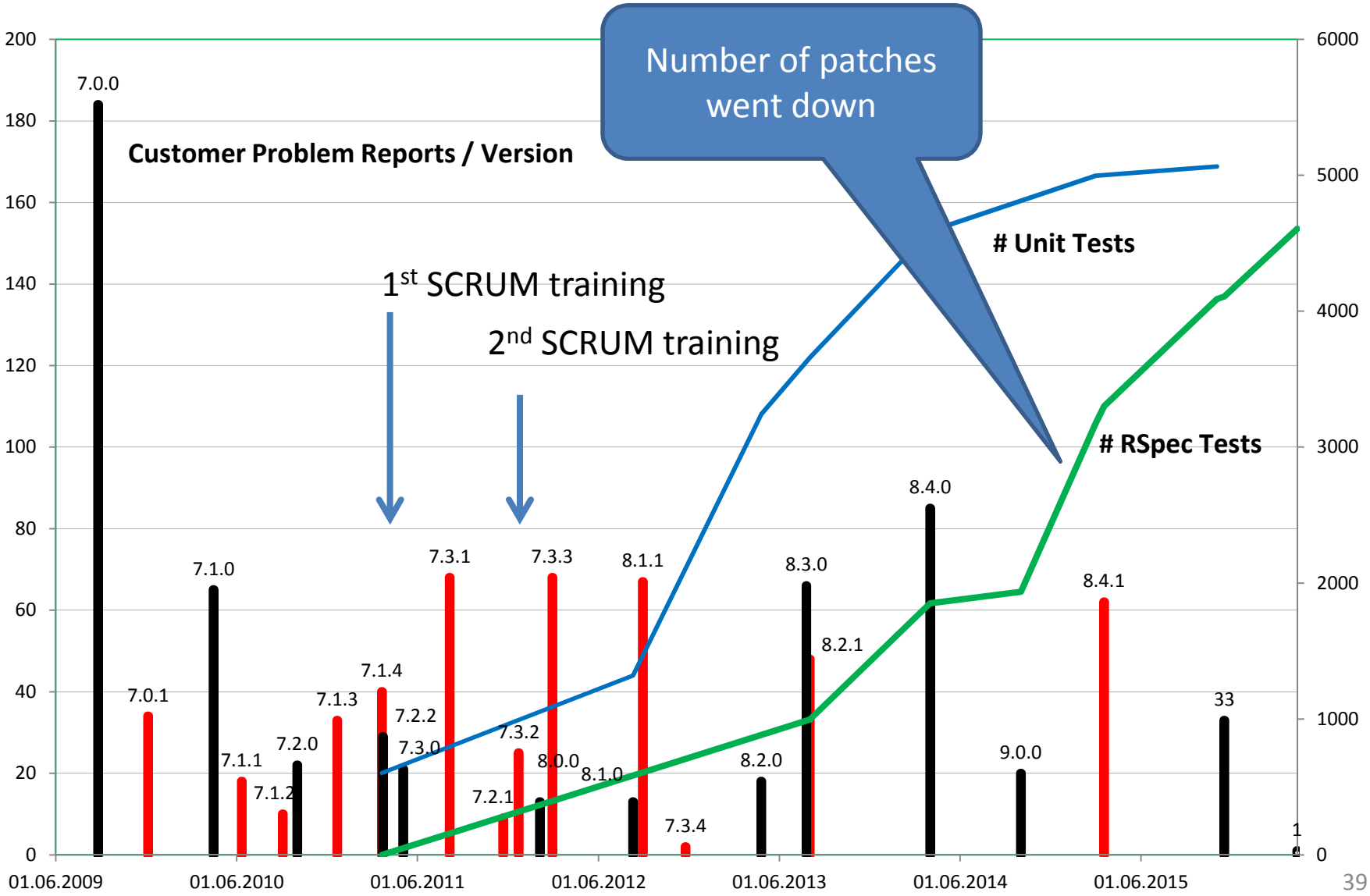
Log inside RSpecCenter execution time per command and generate statistics at the end of each test to find potential bottle-necks

Nice side effect

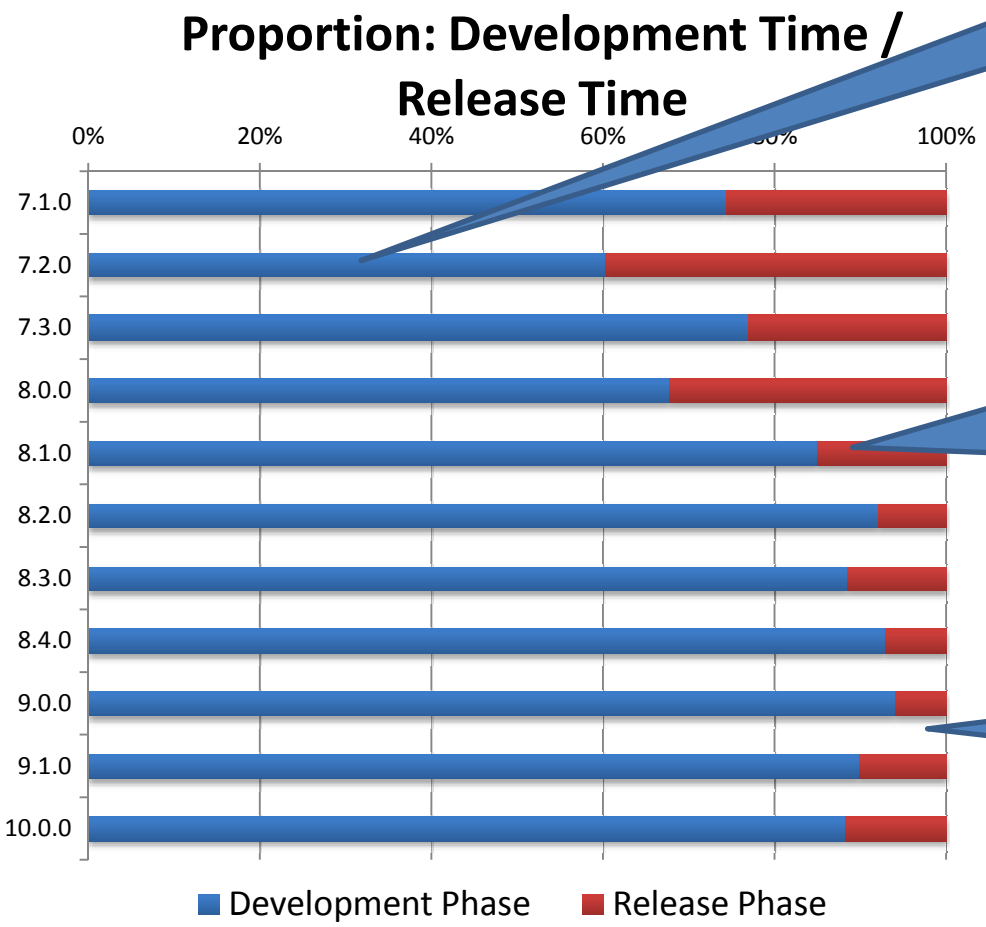
The application was stressed in a way that it was never done before

=> Many race conditions were identified and could be fixed

Results of quality improvements



Process improvements



In the development phase all PBIs are done and all bugs are fixed.

In the release phase are tests executed that we a required to do after code freeze and those that we choose

Decreasing trend of the proportion of dev. time / rel. time

Current test status

UnitTests are integrated into the build process (A failing UnitTest results in a failing library build)

We just write UnitTests any more for generic code. No Business rules are checked with UnitTests but with behaviour tests.

Complete continual test suite run takes 3h

Release test cycle takes 2 weeks (main focus is now on regulatory required and exploratory tests)

Overall lessons learned

Agile development is possible in a regulated environment

Train the whole team

Empower the team

Responsibility lies on everyone

It is possible to turn around a huge legacy code base

Practical lessons learned

Fix failing tests fast

Refactor not only production code, refactor tests code with the same passion

Test code ***IS*** production code

Acknowledgements

The presented work is the result of our whole team

Special thanks for support with the statistics goes to Christian Beck and Thomas Koschel

Reference

- Why Most UnitTesting is Waste and Segue by James O. Coplien, 2014
- Effective Programming with Components - Screen casts by Alexander Stepanov
- Clean Coders – Screen casts by Robert C. Martin
- Continuous Delivery; Jez Humble & David Farley; Addison Wesley, 2010
- Continuous Integration; Stephen M. Matyas, Nicholas Schneider, Mark Voit & Paul Duvall; Addison Wesley, 2007
- Cucumber
- RSpec
- GoogleTest

Feedback is always welcome!

Thank‘s for your attention!

Felix Petriconi

MeVis Medical Solutions AG

Caroline-Herschel-Str. 1

28359 Bremen

Germany

eMail: felix.petriconi@mevis.de

Twitter: @FelixPetriconi

GitHub:
https://github.com/FelixPetriconi

```
module SCR
  # This exception is thrown whenever a timeout occurred in the
  # RSpecCenter
  class TimeoutException < StandardError
  end

  # This exception is thrown whenever an error happens inside the
  # SCR application code
  class CommandFailedException < StandardError
  end

  # This exception is thrown whenever a command was tried to
  # execute that does not exist on SCR side
  class CommandNotFoundException < StandardError
  end

  # This class writes anything which is written to IO (like
  # stderr) to a given logger
  class IOToLog < IO
    def initialize(logger)
      @logger = logger
    end

    def write(text)
      #assume anything written to stderr is an error
      @logger.debug(text)
    end
  end

  class Interface
    def initialize(url)
      @server = XMLRPC::Client.new2(url)
      client_log = Logger.new("XmlRpcClient.log")
      @server.set_debug(IOToLog.new(client_log))
      @server.timeout=60*60*24
    end

    def split_timeout_from_args(args)
      ipc_timeout = 60
      args.each do |element|
        if element.is_a?(Hash) && element.has_key?(:ipc_timeout)
          ipc_timeout = element[:ipc_timeout]
          args.delete element
        end
      end
      ipc_timeout
    end
```

```
def command(process, command, *args)
  ipc_timeout = split_timeout_from_args *args

  xml_result = @server.call("scrcukecommand", process,
    command, ipc_timeout, *args)

  command_result = xml_result[0]
  xml_result.delete_at(0)

  # These are the possible result values from the RSpecCenter
  #enum CommandResultEnum
  #{
  #  CR_SUCCESS,
  #  CR_FAILED,
  #  CR_PENDING,
  #  CR_TIMED_OUT,
  #  CR_NO_COMMAND
  #};

  if command_result == 3
    raise TimeoutException, "The remote IPC command
      (#{command}) timer of #{ipc_timeout}s elapsed.", caller[0]
  end

  if command_result == 4
    raise CommandNotFoundException, "The remote IPC command
      (#{command}) was not found.", caller[0]
  end

  if command_result != 0
    raise CommandFailedException, "The remote IPC command
      #{command} failed: #{xml_result.first}", caller[0]
  end

  xml_result
end
```

```
  @@rpc_locator = nil

  def rpcLocator
    if @@rpc_locator.nil?
      @@rpc_locator = RPCLocator.new
      @@rpc_locator.interface =
      Interface.new("http://127.0.0.1:65501")
      @@rpc_locator.interface.reset_rpec_center
    end
    @@rpc_locator
  end
  module_function :rpcLocator

  def administration
    @@administration ||= SCR::Application.new('ADMINISTRATION',
      rpcLocator)
  end
  module_function :administration


  # This class implements the dependency injection pattern for
  # the XMLRPC interface
  class RPCLocator
    attr_accessor :interface
  end

  # Each process that shall be used inside a
  # RSpec test must have an instance of this class
  # All methods to be called into the process are
  # realized through method_missing.
  class Application
    attr_reader :name

    def initialize(ipc_module_name, locator)
      @name = ipc_module_name
      @rpc_locator = locator
    end

    def method_missing(sym, *args, &block)
      @rpc_locator.interface.command(@name, "#{sym}", args)
    end
  end
end
```