Workshop Brasileiro de Métodos Ágeis

Conferência Brasileira sobre Métodos Ágeis
de Desenvolvimento de Software
Agile Brazil 2010
PUC - RS

24 de Junho de 2010

Mensagem do Coordenador de Programa do WBMA

Neste ano, fiquei muito satisfeito ao organizar o primeiro workshop acadêmico dedicado ao desenvolvimento de métodos ágeis no Brasil. Primeiro, gostaria de agradecer todo o apoio que eu tive dos organizadores do AgileBrazil. Em seguida, gostaria de ressaltar a importância do WBMA, pois já na nossa primeira edição tivemos 33 submissões completas. Isto indica que já havia na comunidade acadêmica uma necessidade por um evento dedicado, seguindo a mesma linha de eventos tradicionais como as conferências Agile e XP.

Os membros do comitê de programa foram essenciais na escolha dos melhores artigos, das 33 submissões completas apenas 11 artigos foram aceitos. Logo, o WBMA já nasce como um workshop grande, com taxa de aceitação de três para um. Neste ano ainda teremos como palestrante convidado no WBMA o pesquisador Philippe Kruchten, professor da UBC em Vancouver (Canadá).

Finalmente, gostaria de agradecer a todos os autores, membros do comitê e revisores externos, sem os quais não poderiamos já nesta primeira edição oferecer um evento de qualidade.

Alfredo Goldman

IME - USP

Comitê de Programa

Alexandre Vasconcelos
Alfredo Goldman (coordenador)
Ana Cristina Rouiller
Cleidson de Souza
Clovis Fernandes
Cristine Gusmão
Fabio Kon
Frank Maurer
Jutta Eckstein
Marcos Chaim
Maria Istela Cagnin
Márcio Delamaro
Rafael Prikladnicki
Raul Sidnei Wazlawick
Rodolfo Resende
Sérgio Soares
Tore Dyba
Valter Vieira de Camargo

Revisores Externos

Alexandre Freire
André Abe Vicente
Cláudia Melo
Célio Santana
Dairton Bassi
Danilo Sato
Débora Paiva
Edson Murakami
Eduardo Katayama
Ellen Francine Barbosa
Felipe Besson
Fernando Castor
Fernando Kamei
Gustavo Alves
Hugo Corbucci
Luiz Carlos Miyadaira Ribeiro
Marcelo Fantinato
Marcelo Morandini
Marcio Cornelio
Mariana Bravo
Paulino Viana
Paulo Cheque
Renata Moreira
Rohit Gheyi
Tiago Massoni
Vinicius Humberto Serapilha Durelli
Viviane Malheiros

Índice

Sessão 1

Testes no contexto de métodos ágeis: técnicas, padrões e apoio automatizado

ClassMock: A Testing Tool for Reflective Classes Which Consume Code Annotations

Eduardo Guerra[1], Fábio Silveira[2], Clovis Fernandes[1]

[1]Aeronautical Institute of Technology (ITA) , Praça Marechal Eduardo Gomes, 50
Vila das Acácias - CEP 12.228-900 – São José dos Campos – SP, Brazil

[2] Federal University of São Paulo (UNIFESP), Rua Talim, 330
CEP 12231-280 - São José dos Campos – SP, Brazil

guerraem@gmail.com, fsilveira@unifesp.br, clovistf@uol.com.br

Abstract. *In an automated test, a method can be called with different parameters to verify its behavior within distinct contexts. Classes which use reflection and metadata in its processing logic can have distinct behaviors based on the structure of the class received as a parameter. This fact can complicate the tests since a different class structure is necessary for each test. This paper presents the ClassMock, which is an open source tool that can be used to generate classes at runtime to be used in such tests. As a result, using ClassMock the test code becomes less verbose and code duplication is avoided in the test class.*

1. Introduction

Automated tests perform an important role in agile methodologies (Beck 2000), since they provide a fast feedback to the implemented functionality and give confidence and safety for the developers to refactor the source code. This fact facilitates changes in the source code, reducing its cost even in advance phases of the application development. Test Driven Development is a software development technique, widely used with other agile methods, where the unit tests are coded before the application code (Beck 2002). This technique helps to ensure a good test coverage, besides other benefits like better defects detection and a lower complexity in the source code (Canfora *et al.* 2009).

In this context, the quality and readability of the test code are important, since it should be modified when a new functionality should be added or an existent one should be modified (Guerra and Fernandes 2007). Based on that, it is important to avoid code duplication and unnecessary verbosity on the test classes, aiming a better test code design that is easier to understand and to modify.

In regular classes, the behavior can vary according to environment configurations, methods invocation and data passed as parameters. In order to verify each class behavior, the test should be able to manipulate those elements to simulate the desired scenario. In classes which uses reflection (Maes 1987), the class structure and elements of the instance received as a parameter have influence in the behavior. Those classes can also read code annotations (JSR175 2003) (Schwarz 2004), which is a metadata definition form that is part of the class structure.

In the test code of those classes, in order to simulate different situations, it is necessary to create for each test, a new class with a specific structure. Differences between the structure of each class are usually small, such as a change on a method's name or a modification in an annotation parameter. The definition of an entire class in each test makes the test code verbose and with a compromised readability, since differences between classes can be hard to detect.

This paper presents ClassMock, which is a tool whose main objective is to help in the generation of classes with a desired structure at runtime, making the test code less verbose. Additionally, the facilities that it provides to manipulate the class structure can help to avoid duplication, since code which generates it can be shared among different test methods. Finally, this paper presents the use of ClassMock as a base to other test tools, as well as the experience in its use to the tests of a metadata-based framework.

Section 2 presents more details about reflection and code annotations. Section 3 explains how tests to this kind of class are usually created, highlighting the main problems of this approach. Further, Section 4 presents the ClassMock tool and how it can be used in a test class, followed by Section 5, which shows how it can be used with mock objects to test dynamic proxies. Section 6 presents the experience in the use of the proposed tool in the test of the framework SystemGlue. Next, Section 7 presents the role of ClassMock in the creation of JQuati, which is a test tool for aspect's pointcut accuracy. At last, Section 8 highlights the main contributions of this research and propose some future works.

2. Reflection and Code Annotations

The notion of computational reflection was first introduced by Smith (1982) in the context of procedural languages. Reflection can be defined as the process by which a computer program can observe and modify its own structure and behavior (Maes 1987). A programming language can be classified as reflective if it provides meta-information about its programs to themselves and a reflective architecture that separate the meta-information from the program itself (Doucet et al. 2003).

Introspection is the subset of reflection that allow a computer program to obtain information about itself. A prerequisite to introspection is reification, that is a data structure to store the reflected meta-information about program's structure and properties. In this context, introspection is the capability to query the structure and modify information in the base program (Doucet et al. 2003).

The Java programming language is an example of a language with introspection capabilities. The Reflection API (Forman & Forman 2005) allows the access to the program meta-information. For example, an instance of the meta class `java.lang.Class` for a given class provide information about its superclass, interfaces, methods and attributes. The meta class `java.lang.reflect.Method` provides information about its return type, parameters and thrown exceptions. This meta-information can be used by a program to instantiate classes and invoke methods.

Metadata is an overloaded term in computer science and can be interpreted differently according to the context. In the context of object-oriented programming, metadata is information about the program structure itself such as classes, methods and

attributes. A class, for example, has intrinsic metadata which can be retrieved using an introspection API.

When a class uses reflection to access the class elements and execute its responsibilities, sometimes the class intrinsic information is not enough. If its behavior should differ for different classes, methods or attributes, it is necessary to add a more specific meta-information to enable differentiation. For some domains, it is possible to use marking interfaces, like `Serializable` in Java Platform, or naming conventions (Chen 2006), like in Ruby on Rails (Ruby *et al.* 2009). But those strategies can be used only for a limited amount of information and are not suitable for situations that need more information.

An alternative to define this additional metadata that is becoming popular in the software community is the use of code annotations, which is supported by some programming languages, such as Java (JSR175 2003) and C# (Miller 2003). Using this technique the developer can add custom metadata elements directly into the class source code, keeping this definition less verbose and closer to it. The use of code annotations is a popular practice in recent software development, as can be evidenced by its usage on highly used frameworks and APIs such as Hibernate (Bauer and King 2006), Struts (Brown *et al.* 2008), Spring (Walls and Breidenbach, 2007) and EJB 3 (JSR220 2006).

The use of code annotations is also called attribute-oriented programing (Schwarz 2004), which can be defined as a program-level marking technique used to mark program elements, such as classes, methods and attributes, with annotations to indicate that they maintain application-specific or domain-specific semantics (Schwarz 2004) (Rouvoy e Merle 2006). This technique introduces a declarative approach inside an imperative programming language.

3. Testing Classes with Reflection and Metadata

As presented in the Section 1, in order to test different possibilities in a class which uses reflection, it is necessary to vary the class passed as a parameter. Those tests usually uses classes with a similar structure varying some details which can influence in the behavior. For instance, if the class under test (CUT) uses naming conventions, classes passed as parameters should have different names for similar methods or attributes. A further example, in the tests of classes which processes code annotations, the difference between the parameter class can be a change in the annotations or in their attributes. This work define as a *mock class*, a fake class created to define a test scenario for classes whose behavior depends on the structure and metadata of the instance received as parameter.

Those classes defined specifically for the tests can be defined inside the test class or externally. The external definition has the drawback to keep the mock class apart from the test code, which can prejudice the test readability. For instance, it is similar to regular tests to define method parameters in a different file. The other alternative is to define the mock classes as an inner class in the test code. This approach also has its drawbacks, such as to make the test code verbose and to duplicate code in the similar classes definitions.

Figure 1 presents two test methods, based on JUnit 4 (Massol and Husted 2003), of the class `PropertyMapFactory`, whose method `getPropertyMap()` generates a map from an instance properties by the introspection of its structure. An inner class named `Example` is defined in both tests to generate the desired class structure. The class structure is the same, except for the `@Ignore` annotation in the `getProp2()` method in the second test. The code snippet where the `Example` class is instantiated and its properties are set is exactly the same in both of tests. However, it can not be reused since in each test `Example` is referring to a different class.

```
@Test
public void mapCreation(){

   class Example{
      private String prop1;
      private int prop2;
      public String getProp1() {return prop1;}
      public void setProp1(String prop1) {this.prop1 = prop1;}
      public int getProp2() {return prop2;}
      public void setProp2(int prop2) {this.prop2 = prop2;}
    }

    Example example = new Example();
    example.setProp1("test");
    example.setProp2(13);

    Map<String, String> map = PropertyMapFactory.getPropertyMap(example);

    Assert.assertEquals("test", map.get("prop1"));
    Assert.assertEquals("13", map.get("prop2"));
}

@Test
public void mapCreationWithIgnore(){

   class Example{
      private String prop1;
      private int prop2;
      public String getProp1() {return prop1;}
      public void setProp1(String prop1) {this.prop1 = prop1;}
      @Ignore public int getProp2() {return prop2;}
      public void setProp2(int prop2) {this.prop2 = prop2;}
    }

    Example example = new Example();
    example.setProp1("test");
    example.setProp2(13);

    Map<String, String> map = PropertyMapFactory.getPropertyMap(example);

    Assert.assertEquals("test", map.get("prop1"));
    Assert.assertNull(map.get("prop2"));
}
```

Figure 1. Two tests of a class which uses reflection.

This example illustrates that in the tests of this kind of class, the main difference between the scenarios is in the class structure. Subtle differences in those classes, such as in names or in annotation attributes, can be hard to perceive since the verbosity in the class definitions hinders the test code readability. As a consequence, similar code can be hard to be reused because it deals with distinct classes. The authors did not find a specific tool which helps this kind of test, and the existent alternative is to create each mock classes individually.

4. Testing with ClassMock

ClassMock is a tool which generates mock classes to be used in the test of classes based on reflection and code annotation's processing. It can be used to test an individual class or an entire component. The programmatic class definition enable the reuse of code which generates part of its structure. For instance, it can be used to define part of the mock class in the methods that execute before the test methods. ClassMock is open-source and can be downloaded at <http://classmock.sf.net>. Despite the tool support is only for the Java language, a similar approach can be used to create similar tools for other languages.

`ClassMock` is the main class of the tool and each instance of it can be used to generate one mock class. It has methods to add properties (attributes with getters and setters), methods, attributes and annotations. The generated methods can be empty or abstract depending on the `ClassMock`'s method invoked. It uses the concept of fluent interface (Fowler 2008), in which each method called returns the class instance. The class `ClassMockUtils` provides some static methods that can be used to invoke methods, create new instances, set properties and get properties on the generated mock classes. Figure 2 presents the same tests of Figure 1 but now using `ClassMock`.

```
public class PropertyMapFactoryTest {

   private ClassMock mockClass;

   @Before
   public void createMockClass() {
      mockClass = new ClassMock("ExampleClassMock");
      mockClass.addProperty("prop1", String.class).addProperty("prop2",int.class);
   }

   @Test
   public void mapCreation() {
      Object instance = createMockClassInstance();
      Map<String, String> map = PropertyMapFactory.getPropertyMap(instance);

      Assert.assertEquals("test", map.get("prop1"));
      Assert.assertEquals("13", map.get("prop2"));
   }

   @Test
   public void mapCreationWithIgnore() {
      mockClass.addAnnotation("prop2", Ignore.class, Location.GETTER);
      Object instance = createMockClassInstance();
      Map<String, String> map = PropertyMapFactory.getPropertyMap(instance);

      Assert.assertEquals("test", map.get("prop1"));
      Assert.assertNull(map.get("prop2"));
   }

   private Object createMockClassInstance() {
      Object instance = newInstance(mockClass);
      set(instance, "prop1", "test");
      set(instance, "prop2", 13);
      return instance;
   }
}
```

Figure 2. Using ClassMock to test the PropertyMapFactory

In this test class, the main mock class definition occurs in the method `createMockClass()`, which executes before each test method. This definition can be complemented in each method, as occurred in `mapCreationWithIgnore()` in

which the `@Ignore` annotation was added in the getter method of `prop2` property. Besides the code reuse, this definition makes clear what changes among the class definition for each test method. The code snippet which deals with the generated classes can also be reused by using the static methods `set()` and `newInstance()` of the `ClassMockUtils` class.

Despite the increased code reuse and the reduced verbosity, the great drawback of this approach is that it makes difficult the use of the test code as documentation. The programmatic class definition does not provide direct examples of the classes as the approach presented in Section 3. So, it is important to consider the role of the test code in the project before using this technique.

5. ClassMock Usage With Mock Objects

A mock object can be considered a fake object which is used to replace class dependencies, in order to isolate the CUT from them (Mackinnon *et al.* 2001) (Freeman *et al.* 2004). A mock object does not only simulates the desired behavior for the test, but also verifies if the method invocations performed on it are correct. Examples of behavior simulation on a method invocation are: returning a value or throwing an exception, and instances of verifications can involve the number of method invocations, their parameter values or the sequence in which they are invoked. The mock object must have the same contract of the one expected as a dependence, which is achieved by implementing the same interface or extending the class.

The creation of mock objects for a large number of tests can be a hard task, since a new class implementing the dependence behavior simulation and the verifications should be created for each test. For that reason, a common practice is to use a framework for an automated generation of the mock object, such as JMock (2010) and Mockito (2010). This approach, besides providing a faster test development, also enable the developer to express the test scenario inside the test method, improving its readability.

It is important to distinguish between the mock objects and the mock classes, which is a term introduced by this paper. On one hand, the mock objects simulates a desired behavior and executes verifications for a given contract which the CUT depends on. On the other hand, a mock class generates a new contract to enable the test to simulate different structures for classes whose behavior depends on it.

Consistent with that, the generation of mock objects based on the contract of mock classes can be used, for instance, for testing dynamic proxies. A dynamic proxy is a proxy which can assume the wrapped object interface dynamically (Forman and Forman 2005). The mock class can be used to generate a class or an interface to be wrapped by the dynamic proxy, whose behavior can be based on its structure or in annotations present on it. In addition, the mock object can be created based on the mock class contract, verifying how the invocation is delegated to it by the dynamic proxy.

Figure 3 presents the source code of a test class which uses ClassMock in conjunction with JMock. The CUT is the `AuthorizationProxy`, which is a dynamic proxy that authorizes or not the invocation of the class method based on the user roles and on the method's `@AuthorizedRoles` annotation. In the method

`createMock()`, the mock class called `DummyInterface` is created with a method called `execute()` annotated with `@AutorizedRoles("admin")`. A mock object is created based on this interface.

The test method simulate the access of the `execute()` method by users with different roles. The mock object is used to verify if the method was called or not based on the scenario. This simple example aims to illustrate the use of mock objects and mock classes together. In a real example, the security annotations could be varied among the test methods to simulate a different scenarios.

```
@RunWith(JMock.class)
public class AuthorizationProxyTest {

    Mockery context = new JUnit4Mockery(){{
        setImposteriser(ClassImposteriser.INSTANCE);
    }};

    private Object mock;

    @Before
    public void createMock(){
        ClassMock classMock  = new ClassMock("DummyInterface",true);
        classMock.addAbstractMethod(void.class,"execute")
          .addMethodAnnotation("execute", AuthorizedRoles.class,new String[]{"admin"});
        Class interf = classMock.createClass();

        mock = context.mock(interf);
    }

    @Test
    public void authorizedMethod() throws Throwable {
        Object proxy = AuthorizationProxy.createProxy(mock, new User("john","admin"));
        context.checking(new Expectations() {{
            invoke(one(mock),"execute");
        }});
        invoke(proxy,"execute");
    }

    @Test(expected=AuthorizationException.class)
    public void unauthorizedMethod() throws Throwable {
        Object proxy = AuthorizationProxy.createProxy(mock, new User("john","oper"));
        context.checking(new Expectations() {{
            invoke(never(mock),"execute");
        }});

        invoke(proxy,"execute");
    }

}
```

Figure 3. ClassMock with mock objects using JMock.

6. Using ClassMock for Framework Tests

This section presents the creation of automated tests for a real framework called SystemGlue. This framework is open-source (SystemGlue 2010) and the test code mentioned in this section are available with the main download file. The application of ClassMock in the test of this framework was used to validate the proposed approach. In this case study, the tests were created after the framework implementation.

SystemGlue is a framework whose main objective is to reduce the coupling between the application main functionalities and concerns related to the integration with other systems. It can be used to generate different integration profiles for the same

application to be deployed in different environments. SystemGlue uses dynamic proxies to intercept method invocations and reflection to invoke the respective methods to integrate with other systems. The developers should configure code annotations and/or XML documents to define which methods should be invoked before and after the application functionalities.

```
@RunWith(JMock.class)
public class SystemGlueTest {

    Mockery context = new JUnit4Mockery(){{
        setImposteriser(ClassImposteriser.INSTANCE);
    }};
    private Object mock;
    private ClassMock baseClass;
    private OtherClass mockCalled;
    private Object mockBase;
    private Object proxyBase;

    @Before
    public void createBaseClass(){
        ProxyFactory.setProxyImplementation(ProxyFactory.REFLECTION);
        baseClass  = new ClassMock("ApplicationClass",true);
        baseClass.addAbstractMethod(void.class,"executeWithParam",int.class);
        mockCalled = context.mock(OtherClass.class);
        MockFinder.addMock(OtherClass.class, mockCalled);
    }

    @Test
    public void ruleExecution() throws Throwable{

        //Add parameter name in parameter
        baseClass.addMethodParamAnnotation(0, "executeWithParam",Param.class);
        baseClass.addMethodParamAnnotationProperty(0, "executeWithParam",Param.class,
                                                   "value", "num");
        //Add before annotation with rule
        addAnnotation(ExecuteBefore.class, OtherClass.class, "otherExecution",
                      "executeWithParam");
        baseClass.addMethodAnnotationProperty("executeWithParam", ExecuteBefore.class,
                                              "rule", "num < 10");
       //Add after annotation with rule
        addAnnotation(ExecuteAfter.class, OtherClass.class, "toBeExecuted",
                      "executeWithParam");
        baseClass.addMethodAnnotationProperty("executeWithParam", ExecuteAfter.class,
                                              "rule", "num > 10");
        createMock();
        final Sequence sequence = context.sequence("sequence");
        context.checking(new Expectations() {{
           never(mockCalled).otherExecution();
           invoke(one (mockBase),"executeWithParam", 15);inSequence(sequence);
            one(mockCalled).toBeExecuted();inSequence(sequence);
        }});
        invoke(proxyBase,"executeWithParam",15);
    }

    private void addAnnotation(Class an, Class clazz, String call, String inMethod) {
        baseClass.addMethodAnnotation(inMethod, an)
                .addMethodAnnotationProperty(inMethod, an, "clazz", clazz)
                .addMethodAnnotationProperty(inMethod, an, "method", call)
                .addMethodAnnotationProperty(inMethod, an, "finder", MockFinder.class);
    }

    private void createMock() {
        Class base = baseClass.createClass();
        mockBase = context.mock(base);
        proxyBase = ProxyFactory.createProxy(mockBase);
    }
}
```

Figure 4. Using ClassMock for a test of the SystemGlue framework.

Figure 4 presents one of the tests performed on SystemGlue by using ClassMock and JMock, with the respective auxiliar methods. This test address the functionality to execute conditionally methods based on rules defined in the annotations.

A mock class is created in order to simulate the application class with the annotations. The base of the mock class is created in the `createBaseClass()` method, which is executed before each test. This class is complemented in the test methods to define a structure necessary to that test scenario. The annotation `@Param` is inserted in the method parameter to configure its name to be used in the rules. The auxiliar method `addAnnotation()` is used to simplify the framework annotations definition. In addition, an annotation property to define the invocation rule is also defined.

Two mock objects are created in this test: one for the mock class and another for the class to be invoked by the framework. A sequence is defined for JMock to verify if the methods are invoked in the correct order. On one hand, according to the scenario, the method `otherExecution()` should never be invoked, since the condition defined in the annotation is not satisfied. On the other hand, the method `toBeExecuted()` should be invoked since the condition is satisfied.

The annotations of the mock class methods varies along the other tests to simulate different configurations. Since the framework should deal with a large number of possibilities related to parameter mapping and annotation's configuration, ClassMock was useful to provide a simple way to vary only the relevant pieces of the class structure for each test.

The testing of frameworks are very important since it is reused in many functionalities, even in more than one application. Reporting the experience of using ClassMock for those tests it is important to highlight that the tool configuration for the first test to run take some time, but the subsequent tests were not hard to create. Authors consider SystemGlue a hard component to test, because its functionality is more abstract than usual and it should deal with a lot of possibilities. In this context, ClassMock was useful since it enables the generalization of the base class structure and provides ways to specify only the important pieces for each test method scenario.

7. ClassMock Usage for Pointcut Accuracy Tests in JQuati

ClassMock can be used as a component for other test frameworks which needs to generate classes at runtime. An example is JQuati (Santana *et al.* 2009) whose objective is to simplify the creation of pointcut accuracy tests for aspects (Kiczales *et al.* 1997). This section presents JQuati usage and how ClassMock is used in its internal structure.

To illustrate the use of JQuati, a simple example presented in (Santana *et al.* 2009) is used. Figure 5 presents the aspect under test, whose advice “shootingSound” should be executed to generate a sound effect before any method named “shoot”.

Figure 6 presents the test created with JQuati to verify the pointcut accuracy. The `@RunWith` annotation from JUnit must be used to configure the `JQuati` class as its test runner. The `@ExecutionContextCreation` determine the classes and the methods which should be generated to test the pointcut accuracy. Those classes are

created and stored in an instance of `ClassExecutionContext`, which is injected in the test class by the test runner.

In each test method, the annotations `@MustExecute` and `@MustNotExecute` should be used to define which advices are expected or not to be executed in such a test. The instance `ClassExecutionContext` can be used to invoke methods on the generated classes to perform the verifications.

```
public aspect SoundEffectsTrigger {

        @AdviceName("shootingSound")
        before():execution(* *.shoot(..)) {
                // make sound effect
        }

}
```

Figure 5. Example of aspect under test

```
@ExecutionContextCreation(
        methods={"void shoot (Integer, Integer)"},
        className={"Gun", "Pistol", "RocketLauncher"})
@RunWith(JQuati.class)
public class SoundEffectsTriggerTests {

        @ExecutionContextElement
        ClassExecutionContext cec;

        @Before
        public void setup(){
           cec.instanciateClasses( ) ;
        }

        @Test
        @MustExecute ("shootingSound")
        public void shouldCauseAShootingSound() throws ClassExecutionException{
           cec.executeMethodOnAll("shoot",0,0);
        }

}
```

Figure 6. Example of test using JQuati (Santana *et al.* 2009)

Figure 7 presents the architecture of JQuati. The main class, named `JQuati`, is the test runner which orchestrates the invocation of the other components. It is responsible to read the annotations from the test class, create the invocation context instance and inject it on the test class.

The class `ClassExecutionContext` uses ClassMock to generate the classes specified by the annotations. The generated classes are dynamically loaded and weaved by the existent aspects at load-time. The simple specification using the annotations and the class generation are important to enable a simple simulation of contexts involving naming patterns and parameter configurations in which the advices should or not be invoked.

The component `AdviceInspector` is responsible for verifying if each advice are executed or not. The approach is similar to AdviceTracer (Delamare *et al.* 2009), which is a very similar tool to the advice inspector and can define and verify expectations regarding the execution of certain advices. The major JQuati differential is

that it supports the generation of the classes using ClassMock to simulate the joint points, while using AdviceTracer the developer should write the classes for the test manually.

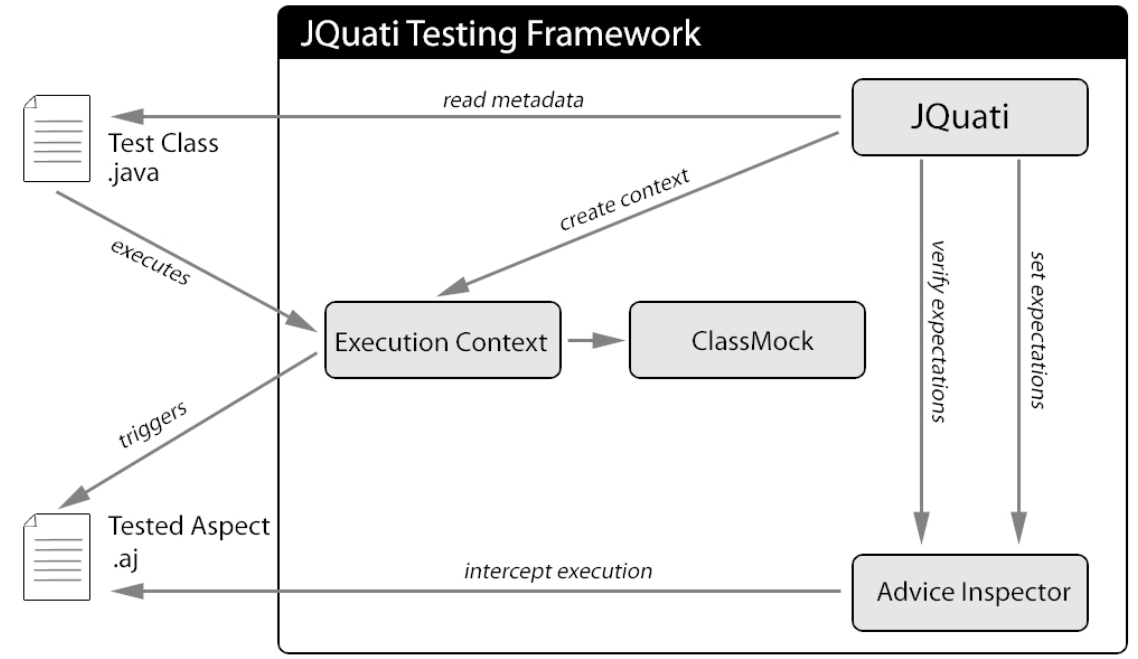


Figure 7. JQuati Architecture (Santana *et al.* 2009)

JQuati illustrates how ClassMock can be used to simulate the execution context for dynamic environments. In aspect-orientation, the pointcut usually depends on class names and method signatures. In this context, ClassMock was used to simulate different situations to verify whether the pointcut definition are defined according to the requirements.

ClassMock also can be used to simulate execution environments for other kinds of software. For instance, in (Wada and Junior 2009), it was used in a tool to define metadata for the framework SwingBean (2009), which aims to generate user interfaces at runtime, like forms and tables based on metadata and in the class structure. Since SwingBean needs a Java bean class to execute, ClassMock was used to generate such class at runtime to enable the user interface to be previewed. Based on this experience, it is possible to state that ClassMock can also be used to simulate execution environments not exclusively for testing tools.

8. Conclusion

This work addressed the automated tests of classes which uses reflection or consume code annotations. Those classes usually are used in frameworks or in flexible solutions which can be reused in different functionalities and contexts. Accordingly, this fact highlights the importance of the test for those kind of classes. However, due to their high abstraction level and generality, they comprise a hard task to test.

The present work introduced ClassMock, which is a tool whose main objective is to simplify the test creation for classes which uses reflection or consume code annotations. It enables a programmatic definition of runtime generated mock classes to be used in the tests, allowing pieces of the class definition to be reused among them. The proposed tool was employed in the tests of a real framework and used to compose other tool for aspect's pointcut accuracy tests.

The following can be considered the major contributions of the present work:

- The introduction of the term 'mock class', which is a technique already used in those tests, which has not yet been differentiated from the concept of 'mock objects'.
- The creation of the framework ClassMock, which enables the definition and the generation of classes at runtime.
- The identification of practices using ClassMock which enable the class definition code to be reused and its usage with mock objects.
- The ClassMock usage as a component to simulate execution environments in other tools.

The main future work in this subject is the use of ClassMock in further projects and in different contexts. For instance, it was used in SystemGlue to create tests after the codification and, in the next step, it can be used with Test Driven Development approach. An important improvement in ClassMock refers to the simplification of the class definition. An idea that would be studied is the definition of the method signatures and annotations using Strings.

Future works could also address an experimental evaluation of the testing creation using ClassMock. In this experiment, distinct developers should develop tests for a framework with and without the usage of the proposed approach. Source code measurements, questionnaires and the development time should be used to assess the benefits and drawbacks of this approach.

References

Bauer, Christian; King, Gavin. Java Persistence with Hibernate. Manning Publications, 2006.

Beck, K. “Extreme Programming Explained”, Addison Wesley Longman, 2000.

Beck, K. “Test-Driven Development by Example”, Addison Wesley, 2002.

Brown, Donald; Davis, Chad Michael; Stanlick, Scott. Struts 2 in Action. Manning Publications, 2008.

Canfora, G.; Viseggio, C.; Garcia, F.; Piattini, M. “Measuring the impact of testing on code structure in Test Driven Development”. Tenth International Conference on Agile Processes and eXtreme Programming in Software Engineering (XP 2009), 2009.

ClassMock. “ClassMock - Test Tool for Metadata-Based and Reflection-Based Components”. Available at http://classmock.sf.net acessed in 2010-03-20.

Chen, Nicholas. Convention over Configuration. 2006. Available at http://softwareengineering.vazexqi.com/files/pattern.html acessed in 2009-12-17.

Delamare, R.; Baudry, B.; Ghosh, S.; Traon, Y. L. “A test-driven approach to developing pointcut descriptors in aspectj,” in ICST ’09: Proceedings of the 2009 International Conference on Software Testing Verification and Validation. Washington, DC, USA: IEEE Computer Society, 2009, pp. 376–385.

Doucet, Frederic; Shukla, Sandeep; Gupta, Rajesh. Introspection in System-Level Language Frameworks: Meta-Level vs. Integrated. In: SOURCE DESIGN, AUTOMATION, AND TEST IN EUROPE, 1., 2003. Proceedings... p. 382-387.

Forman, Ira ; Forman, Nate. Java Reflection in Action. [S. l.]: Manning Publications, 2005.

Fowler, Martin. Fluent Interface. Last Update 23 June 2008. Available at http://martinfowler.com/bliki/FluentInterface.html acessed in 2010-03-22.

Freeman, S. and Mackinnon, T. and Pryce, N. and Walnes, J. (2004) “Mock roles, not objects”, In: OOPSLA ´04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, New York, NY, USA, ACM, p. 236-246.

Guerra, E.; Fernandes, C. “Refactoring Test Code Safely”. In: ICSEA, 2007, Tahiti. ICSEA'07, 2007.

JMock. jMock - A Lightweight Mock Object Library for Java. Available at http://www.jmock.org accessed in 2010-03-22.

JSR175. JSR 175: A Metadata Facility for the Java Programming Language. 2003. Available at http://www.jcp.org/en/jsr/detail?id=175 acessed in 2009-12-17.

JSR220. JSR 220: Enterprise JavaBeans 3.0. 2006. Available at http://www.jcp.org/en/jsr/detail?id=220 acessed in 2009-12-17.

Kiczales, Gregor; Lamping, John; Menhdhekar, Anurag; Maeda, Chris; Lopes, Cristina; Loingtier, Jean-Marc; Irwin, John. Aspect-oriented programming. In EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 1997. Proceedings... p. 220–242.

Maes, Pattie. Concepts and Experiments in Computacional Reflection. In THE INTERNATIONAL CONFERENCE ON OBJECT ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES AND APPLICATIONS – OOPSLA 1987. Proceedings... [S.n.t.] p. 147-169.

Mackinnon, T. and Freeman, S. and Craig, P. (2001) “Endo-testing: unit testing with mock objects”, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, p. 287-301.

Massol, Vincent; Husted, Ted. JUnit in Action. Manning Publications, 2003.

Mockito. Mockito - Simpler & Better Mocking. Available at http://mockito.org accessed in 2010-03-22.

Rouvoy, Romain; Merle, Philippe. Leveraging Component-Oriented Programming with Attribute-Oriented Programming. In PROCEEDINGS OF THE 11TH ECOOP INTERNATIONAL WORKSHOP ON COMPONENT-ORIENTED PROGRAMMING (WCOP 2006), Nantes, France, July 2006.

Ruby, Sam; Thomas, Dave; Hansson, David; Heinemeier, David. Agile Web Development with Rails. Pragmatic Bookshelf, Third Edition, 2009.

Santana, E. C. ; Tanaka, S. H. ; Guerra, E. M. ; Fernandes, C. T. ; Silveira, F. . Towards a Practical Approach to Testing Pointcut Descriptors With JQuati. In: III Latin American Workshop on Aspect-Oriented Software Development - LA-WASP´2009, 2009, Fortaleza.

Schwarz, Don. Peeking Inside the Box: Attribute-Oriented Programming with Java 1.5. [S.n.t.] 2004. Available at http://missingmanuals.com/pub/a/onjava/2004/06/30/insidebox1.html accessed in 2009-12-17.

Smith, Brian. Reflection and Semantics in a Procedural Language, PhD thesis, Massachusetts Institute of Technology, 1982

SwingBean. Framework SwingBean. Available at http://swingbean.sf.net/ accessed in 2009-12-17.

SystemGlue. SystemGlue. Available at http://systemglue.sf.net/ accessed in 2010-05-14.

Wada, Marcos; Junior, Salomão. Estudo Comparativo de Ferramentas de Apoio ao Uso de Frameworks Baseados em Metadados. Aeronautical Institute of Technology, Technical Report, 2009.

Walls, Craig; Breidenbach, Ryan. Spring in Action. Manning Publications; 2nd edition, 2007.

ATMM uma ferramenta para gerenciamento de métricas de teste no contexto de métodos ágeis

André Abe Vicente[1], **Márcio Eduardo Delamaro**[1]

[1]Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo (ICMC)
Caixa Postal 668 – 13560-970 – Sao Carlos – SP – Brasil

{avicente,delamaro}@icmc.usp.br

Abstract. *From the characterization of the testing activity applied in agile development methods it was proposed a metrics set adoption to facilitate constant tracking and improvement of this. Some of these test tracking metrics were implemented in the Agile Testing Metrics Management tool (ATMM). To validate the tool it is presented a case study of a software project that used agile methods. The case study analyzed the adherence and maturity of the testing activity, as well as tracking through testing metrics and software quality metrics related to cohesion and coupling.*

Resumo. *A partir da caracterização da atividade de teste de software aplicada dentro de métodos de desenvolvimento ágil foi proposta a adoção de um conjunto de métricas para facilitar o seu acompanhamento e melhoria constante da mesma. Algumas dessas métricas de acompanhamento de testes foram implementadas na ferramenta Agile Testing Metrics Management (ATMM). Para validar a ferramenta é apresentado um estudo de caso com um projeto de software que utilizou métodos ágeis. O estudo de caso procurou analisar a aderência e maturidade da atividade de teste, além do acompanhamento por meio de métricas de teste e qualidade do software utilizando métricas relacionadas à coesão e acoplamento.*

1. Introdução

Métodos ágeis foram desenvolvidos para beneficiarem a entrega rápida de código que agregue valor ao cliente por meio do desenvolvimento em pequenos ciclos. Para atingir esse objetivo, esses métodos são focados na contínua interação entre desenvolvedores e clientes, que garantem que o software atenda as necessidades de mudança dos requisitos do cliente [Paetsch 2003]. Os métodos ágeis mudam o foco de documentação do projeto, fortemente utilizado em métodos tradicionais, para técnicas focadas no desenvolvimento de código fonte e testes. Todo esse dinamismo dentro de métodos ágeis tem provocado um grande impacto na forma de se conduzir um projeto de software fortemente sensível a mudanças.

A utilização de métodos de desenvolvimento ágil tem crescido sensivelmente. Segundo [Sato 2007] esses métodos vêm sendo adotados em diversos contextos, em pequenas, médias e grandes empresas e até agências governamentais e universidades [Begel and Nagappan 2007, Dybå and Dingsøyr 2009]. Em projetos que utilizam métodos ágeis, a atividade de teste de software vem sendo considerada uma atividade primordial, com o objetivo de evitar que a qualidade do produto e a condução do projeto não sejam afetados por processos menos formais de documentação e projeto em relação aos

métodos tradicionais [Simons 2005]. Os resultados dos testes servem como uma forma de *feedback* instantâneo, no qual o desenvolvedor pode detectar em pouco tempo se o método desenvolvido ainda precisa ser refatorado.

Para apoiar o *feedback* constante da equipe em projetos ágeis são utilizadas diversas práticas como reuniões diárias, reuniões de revisão e retrospectivas. Outra prática que apoia o *feedback* constante e a melhoria contínua é area de trabalho informativa que deve fornecer instrumentos que forneçam dados sobre o andamento do projeto. Esses dados serão coletados por meio de métricas de software que devem medir o progresso, apontar melhorias e dificuldades durante todo o projeto [Crispin and Gregory 2009]. Nesse contexto, é importante que a equipe de desenvolvimento e teste utilize métricas para acompanhar essa atividade de teste, focando principalmente na melhoria contínua do processo, das práticas e ferramentas de teste utilizadas. Por fim, essa equipe deve conduzir a atividade utilizando métricas para avaliar e estabelecer metas de qualidade para os artefatos de teste produzidos. A partir de um código de teste de qualidade também será possível medir de forma eficiente a qualidade do software produzido.

Alguns trabalhos [Sato 2007, Williams et al. 2004, Hartmann and Dymond 2006, Crispin and Gregory 2009] propuseram abordagens para escolha e utilização de métricas de software em projetos ágeis, além disso alguns desses trabalhos também criaram métricas específicas para projetos ágeis. No entanto, a utilização de métricas para acompanhamento da atividade de teste foi pouco explorada, havendo a necessidade de descrever melhor os objetivos de se utilizar cada métrica de teste e diversos aspectos que influenciam a utilização dessas métricas. Além disso, há uma carência de ferramentas que automatizem a coleta dessas métricas e forneçam a possibilidade de gerenciamento e análise das métricas durante as iterações de desenvolvimento.

Nesse sentido, este artigo propõe a adoção de um conjunto de métricas que podem ser utilizadas para o acompanhamento da atividade de teste. O principal objetivo da utilização dessas métricas é a melhoria contínua do processo de teste e também dos artefatos de teste produzidos. As informações podem detectar problemas ou evoluções e metas de qualidade para os casos de teste produzidos.

O restante do artigo é estruturado conforme discutido a seguir. Na Seção 2 são descritas as práticas ligadas à atividade de teste no contexto de métodos ágeis. Na Seção 3 é apresentado um conjunto de métricas de teste que podem ser adotadas por uma equipe de desenvolvimento ágil e também são descritos aspectos gerais da ferramenta ATMM. Na Seção 4 é apresentado o estudo de caso com a ferramenta Kalibro e na Seção 5 são apresentadas as contribuições em relação a utilização de métricas de acompanhamento de teste em projetos ágeis e também possíveis trabalhos futuros.

2. Teste de Software em Métodos Ágeis

A atividade de teste assume papéis essenciais dentro do processo de desenvolvimento de software utilizando métodos ágeis. O teste apoia a comunicação entre desenvolvedores e clientes, fornece *feedback* sobre quais as funcionalidades do sistema estão funcionando e apoia a manutenção do sistema em ciclos iterativos e curtos. As mudanças tendem a ser mais seguras se a equipe tiver um bom conjunto de testes para o sistema. Diversas abordagens, estratégias e práticas de teste foram criadas ou adaptadas para o contexto de projetos ágeis, apoiando a integração contínua e o desenvolvimento de soluções simples e que atendam as necessidades do cliente.

Diferentemente de métodos tradicionais nos quais os testes ocorrem mais tarde dentro do processo de desenvolvimento, os testes ágeis devem ocorrer de forma frequente,

procurando detectar defeitos o mais cedo possível dentro de ciclos de desenvolvimento iterativos e curtos, com um constante *feedback* do cliente. Em projetos ágeis, o código é considerado completo apenas se passar por todos testes de unidade. Além disso, no fim de cada iteração, todos os testes de aceitação (*business testing)* que foram criados durante a fase de planejamento serão executados por usuários e clientes.

Entre as práticas de teste utilizadas em métodos ágeis pode-se citar a estratégia de desenvolvimento dirigido a testes (TDD) [Beck 2002] que pode ser utilizada para explorar, projetar, desenvolver e testar o software, não devendo ser tratada apenas como uma atividade de testes. Outras práticas de teste também são utilizadas em projetos ágeis: teste de integração contínuo, teste de aceitação com o cliente e teste de regressão associado a prática de refatoração. Para complementar essas práticas a equipe pode utilizar o teste exploratório, teste da interface gráfica (GUI) e teste de requisitos não-funcionais que podem envolver por exemplo, requisitos de desempenho, carga ou *stress*. Na Tabela 1 é apresentado um comparativo em termos de práticas para retrospectiva e melhoria durante o projeto, além das práticas de VV&T aplicadas dentro de projetos ágeis.

Tabela 1. Comparativo entre Métodos Ágeis (Reuniões, Retrospectiva, Melhoria e Técnicas e Práticas de VV&T) [Vicente et al. 2009]

	Retrospectiva e Melhoria	Técnicas e Práticas de VV&T
XP	- Ciclos semanais e trimestrais	- Integração contínua e testes - *Test-Driven Development* - Testes de aceitação associados a histórias do usuário
Scrum	- Reuniões diárias (*Daily Scrum*), - Reunião de revisão (*Sprint Review*) - Retrospectiva	- *Builds* diários e testes (todos os tipos) - Testes durante o *sprint* e Teste de sistema
FDD	- Relato e visibilidade dos resultados - Inspeções de projeto e código	- Inspeção do projeto e do código - Teste de unidade
Crystal	- Monitoramento do progresso - *Workshops* de reflexão	- Teste de regressão de funcionalidades automatizados
ASD	- Revisões de qualidade	- Teste faz parte da construção concorrente de componentes - Inspeções de código - Revisões para melhorar a qualidade do produto (Testes de Aceitação e Sistema)
DSDM	- *Workshops* com o cliente	- Testes integrados durante todo o ciclo de vida - Todo componente de software é testado pelos desenvolvedores e usuários assim que são desenvolvidos

3. Métricas de Software

Métricas de software são padrões quantitativos de medidas de vários aspectos do projeto de software. A medição dessas métricas em um projeto de software pode apoiar estimativas, o controle de qualidade, a produtividade da equipe e o controle do projeto [Pressman 2006]. Além disso, um conjunto de métricas bem projetado pode ser utilizado para medir a qualidade de produtos de trabalho, dar suporte à tomada de decisão dos gerentes de projeto e aumentar o retorno de investimento do produto [Kulik 2000]. Em métodos de desenvolvimento ágil, a utilização de métricas apoia a medição contínua do produto e do processo, permitindo que os ciclos de desenvolvimento sejam constantemente inspecionados, adaptados e melhorados [Hartmann and Dymond 2006].

Em [Hartmann and Dymond 2006] é definido um conjunto de sugestões para que uma equipe de desenvolvimento ágil defina boas métricas para sua equipe. De forma geral, as métricas devem reforçar princípios ágeis de colaboração com o cliente, entrega de valor, simplicidade, se preocupando com as tendências demonstradas. A quantidade

de métricas também deve sempre ser minimizada, fácil de ser coletada e demonstrar os fatores que as influenciam, evitando manipulações. Métricas ágeis fornecem um guia para a equipe ágil, medindo o progresso do projeto e procurando apontar quando a equipe está desviando dos objetivos da equipe, ou fornecendo o *feedback* de que a equipe está no caminho correto. Métricas podem nos alertar a respeito de problemas, mas analisadas de forma isolada geralmente não fornecem valor algum [Crispin and Gregory 2009]. Um exemplo desse cenário é a diminuição da cobertura de código de um projeto, que pode gerar preocupações se for utilizada de forma isolada. Uma das razões da diminuição da cobertura de código, por exemplo, poderia ser pela remoção de código que não estava sendo utilizado e estava coberto por testes. Se removermos esse código e os seus respectivos testes, a cobertura irá diminuir. Se utilizadas da forma correta, métricas podem estimular o time, que também deve focar na qualidade, não somente em números.

No contexto ágil, métricas de qualidade de código e de projeto oferecem conselhos objetivos, por exemplo, na identificação de áreas da aplicação candidatas à refatoração, assim como métricas de cobertura de código fornecem um guia necessário para o TDD e também a refatoração [Knoernschild 2006]. Alguns trabalhos relacionados propõem métricas para projetos ágeis, um deles é um *framework* de avaliação de projetos em XP (*XP Evaluation Framework*) que sugere diversas métricas para projetos XP [Williams et al. 2004]. Em [Sato 2007] um conjunto de métricas organizacionais e de acompanhamento é aplicada em projetos acadêmicos e governamentais. Diversos trabalhos avaliam projetos ágeis por meio de métricas de programas orientados a objeto [Sato et al. 2007, Ambu et al. 2006]. Também há trabalhos que propõe a automação de métricas de software, como por exemplo a ferramenta GERT *(Empirical Reliability Estimation and Testing Feedback Tool*) uma ferramenta de apoio ao TDD por meio do acompanhamento métricas de teste, que apoiam os ciclos de *feedback* criados pelos testes contínuos [Davidsson et al. 2004].

No entanto, ainda verifica-se a necessidade de mais estudos a respeito da utilização de métricas ágeis para o acompanhamento da atividade de teste, os benefícios e dificuldades em utilizar essas métricas para a melhoria contínua da condução e dos artefatos produzidos na atividade de teste. Além disso, existe a necessidade em se produzir ferramentas que coletem as métricas de teste e facilitem o gerenciamento de iterações de desenvolvimento relacionados a evolução dessas métricas.

3.1. Métricas de acompanhamento para Testes Ágeis

As métricas de teste de software propostas neste trabalho foram criadas a partir de uma lista de verificação proposta por [Hartmann and Dymond 2006]. Essa lista é baseada na abordagem GQM (*Goal Question Metric*) e na abordagem Lean. O objetivo dessas métricas é facilitar o acompanhamento da atividade de teste, visando a melhoria contínua dos testes e consequentemente a qualidade do código fonte, com menos defeitos, com um bom *design* e mais fácil de ser mantido. Algumas das métricas foram propostas nesse trabalho, outras métricas foram adotadas de outros trabalhos e descritas segundo a lista de verificação. A origem das métricas de acompanhamento de teste deste trabalho são resumidas na Tabela 2 e uma das métricas é descrita na Tabela 3.

3.2. Ferramenta ATMM

A ferramenta *Agile Testing Metrics Management* (ATMM) complementa a proposta deste trabalho, tendo sido desenvolvida como parte de suas contribuições. Tem como objetivo apoiar a atividade de teste de software no contexto ágil, especificamente na fase de teste de unidade, em que os testes são gerenciados em cada iteração de desenvolvimento. O

Tabela 2. Origem das métricas de acompanhamento de teste adotadas no trabalho

Métrica	Origem
M1. Cobertura de Código	XP-EF [Williams et al. 2004] e [Nagappan et al. 2005]
M2. Fator de Teste	Extraída de [Sato 2007]
M3. Quantidade de Casos de Teste e Assertivas	XP-EF [Williams et al. 2004] e [Nagappan et al. 2005]
M4. Porcentagem de Assertivas de Teste de Unidade Passando e Falhando	Proposta neste trabalho
M5. Quantidade de Testes de Aceitação por Funcionalidades	[Nagappan 2004]
M6. Porcentagem de assertivas de teste de aceitação passando e falhando	Proposta neste trabalho
M7. Funcionalidades Testadas e Entregues (*Running Tested Features* ou RTF)	Extraída de [Sato 2007]
M8. Tempo de Execução de Testes	Proposta neste trabalho
M9. Quantidade de Defeitos Encontrados	XP-EF [Williams et al. 2004]

objetivo principal da ferramenta, além de gerenciar essas iterações, é exibir métricas relacionadas ao código que está sendo testado e aos casos de teste desenvolvidos utilizando a ferramenta JUnit (bastante utilizada em projetos ágeis). Além disso, a ferramenta exibe informações de cobertura a partir de informações extraídas da ferramenta *JaBUTi* (*Java Bytecode Understanding and Testing*)[1] que utiliza critérios estruturais de fluxo de controle e fluxo de dados [Vincenzi 2004].

A maioria das métricas da ferramenta ATMM é coletada de forma automática (automatizada) e outras dependem da informação do usuário que está utilizando a ferramenta. Algumas das métricas coletadas pela ferramenta são relacionadas diretamente ao acompanhamento da atividade de teste, são elas: fator de teste, qtde de casos de teste, qtde de assertivas, casos de teste com sucesso/falha/ignorados e tempo total de execução dos casos de teste. O restante das métricas foram implementadas por serem facilmente coletadas e também contribuírem para a avaliação das iterações de desenvolvimento, como por exemplo a quantidade de classes de teste.

A ferramenta ATMM faz interface com a API Java Parser [2] e a ferramenta *JaBUTi*. A arquitetura geral da ferramenta é descrita a seguir: **(i) Coleta de informações do código fonte e testes:** a ferramenta utiliza a API *Java Parser* para coletar informações sobre o código fonte e os casos de teste no formato da ferramenta JUnit, desenvolvidos em Java. **(ii) Execução dos casos de teste e coleta de informações de cobertura:** utilizando a ferramenta *JaBUTi*, a ferramenta ATMM executa os casos de teste, coleta informações sobre essa execução dos testes JUnit e também informações de cobertura de código. **(iii) Gerenciamento das iterações:** as métricas coletadas são armazenadas a cada iteração de desenvolvimento.

O primeiro passo da ferramenta ATMM é a criação do projeto e a seleção de informações gerais do projeto. Nesse passo o testador no início do projeto deverá inserir informações que envolvem características gerais do projeto, a sua descrição, as estratégias ou técnicas de teste que serão utilizadas no projeto, além de outras práticas ágeis que o projeto utiliza, como programação em pares ou refatoração. Após a criação do projeto, o segundo passo é inserir informações sobre cada iteração de desenvolvimento e testes. Ao fim de cada iteração o testador deverá empacotar o código fonte e código de teste em diretórios específicos e esses caminhos deverão ser informados a ferramenta. O terceiro

[1] JaBUTi -(Project site) - http://ccsl.ime.usp.br/pt-br/project/jabuti

[2] AST Java Parser (API) - Disponível em: `http://code.google.com/p/javaparser`

Tabela 3. Métrica - Porcentagem de Assertivas de Teste de Unidade Passando e Falhando

- **Classificação:** Objetiva, Quantitativa e Acompanhamento
- **Objetivo:** Verificar qual a porcentagem de assertivas dos casos de teste de unidade/integração que estão passando ou falhando.
- **Pergunta:** Qual a porcentagem de assertivas de testes de unidade/integração que estão passando/falhando? Base da Medição: A porcentagem de assertivas passando será dada pelo:
- **Base da medição:**

  Assertivas Passando % = (N^o de assertivas passando / N^o total de assertivas)

  ou

  Assertivas Falhando % = (N^o de assertivas falhando / N^o total de assertivas)
- **Suposições:** A equipe está desenvolvendo código fonte para determinadas funcionalidades do sistema que possuem testes de unidade/integração associados. Para cada caso de teste de unidade devem ser criadas assertivas suficientes para verificar o comportamento (entrada/saída esperada) de uma unidade ou a integração dela com outras unidades.
- **Tendência esperada:** Todo o código de produção deve passar pelos casos de teste e consequentemente suas assertivas.
- **Quando utilizar:** Essa porcentagem deve ser utilizada para acompanhamento dos testes de unidade, podendo verificar a existência de defeitos em novas funcionalidades ou em funcionalidades alteradas durante a iteração.
- **Quando parar de utilizar:** Apenas quando o projeto terminar, pois sempre a equipe de desenvolvimento deve acompanhar os resultados da execução dos testes de unidade/integração para verificar a existência de defeitos no código de produção.
- **Formas de manipular:** Podem ser criadas diversas assertivas que não melhorem a qualidade dos testes simplesmente para aumentar a porcentagem de assertivas corretas.
- **Cuidados e observações:** Mesmo se todas as assertivas estão passando, isso não garante que o teste de unidade e consequentemente a funcionalidade funcionam de acordo com o desejo do cliente. Assertivas falhando podem ser consequência de um defeito no código desenvolvido como também defeito na assertiva.

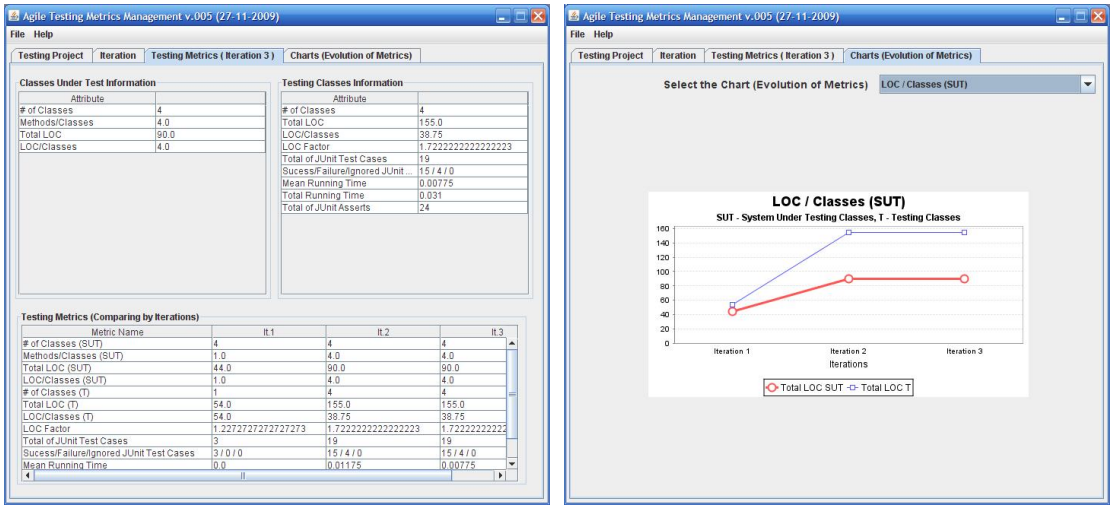
passo (Figura 1(a)) irá exibir as métricas relacionadas ao código que está sendo testado, o código de testes e também uma tabela comparativa que mostra os valores das métricas em cada iteração. Esses dados podem ser exibidos na forma de gráficos comparativos conforme ilustrado na Figura 1(b).

Ao final de cada iteração conforme os resultados das métricas apresentadas pela ferramenta, o testador poderá desenvolver mais casos de teste ou refatorar os casos de teste existentes. A equipe pode decidir se essa tarefa será realizada antes da retrospectiva da iteração ou será realizada na próxima iteração. Por fim, o resultado das métricas deverá ser discutido durante a reunião de retrospectiva da iteração, e a equipe de desenvolvimento e teste poderá estabelecer novas metas de qualidade do código-fonte e dos casos de teste para a próxima iteração baseando-se nas métricas obtidas nas iterações anteriores. O *tracker* deverá atualizar as informações a respeito das métricas na área de trabalho informativa com gráficos ou tabelas que mostrem a evolução das métricas durante as iterações. Caso a equipe de desenvolvimento, teste e os gerentes do projeto acharem necessário, as métricas poderão ser avaliadas diariamente e não somente no final de cada iteração.

4. Estudo de Caso

Essa seção apresenta o método utilizado no estudo de caso, as métricas avaliadas, uma descrição geral do projeto Kalibro, a análise dos resultados em termos das métricas de acompanhamento de teste e métricas de qualidade ligadas a coesão e acoplamento.

O estudo de caso consistiu na avaliação da maturidade e aderência a atividade de teste do projeto Kalibro e a coleta das métricas de acompanhamento de teste propostas neste trabalho. Foi analisada a evolução dessas métricas sob a hipótese inicial de que a utilização de métricas de teste pode colaborar na detecção de problemas durante a condução dessa atividade, além de monitorar a evolução da qualidade dos casos de teste, que reflete na qualidade do código fonte produzido.



(a) Métricas (Tabelas) (b) Métricas (Gráficos)

Figura 1. Ferramenta ATMM - Exibição comparativa das métricas

A análise do projeto Kalibro apresentou algumas limitações, como a análise das métricas de teste após o projeto ter sido desenvolvido e também por se tratar de um projeto acadêmico, com um estudo de caso sem um ambiente controlado como em um experimento, sendo difícil a generalização dos resultados obtidos. Apesar dessas limitações e de não ter sido estabelecido toda a formalidade necessária para um estudo de caso segundo a engenharia de software experimental, este estudo apresenta diversos resultados a respeito da utilização métricas para o acompanhamento da atividade de teste em projetos ágeis. Esses resultados podem servir como incentivo e guia para o uso de métricas de software em projetos ágeis para a melhoria contínua da atividade de teste.

4.1. Método e Métricas avaliadas

Para coletar as métricas de acompanhamento de testes de forma automatizada, foram utilizadas as seguintes fontes de informação: (i) Ferramenta ATMM: a partir da ferramenta ATMM foram coletadas algumas das métricas de teste propostas nesse trabalho, além de métricas relacionadas ao código-fonte de cada projeto. (ii) *Scripts* da ferramenta *JaBUTi*: foram obtidas informações a respeito da cobertura de código conforme critérios estruturais de fluxo de controle e fluxo de dados a partir de um conjunto de *scripts*. (iii) Ferramenta Kalibro: foram coletadas algumas métricas da ferramenta Kalibro para verificar a qualidade do código produzido no decorrer das iterações. Essas métricas são relacionadas a qualidade dos métodos produzidos, e também métricas de coesão e acoplamento. (iv) Questionário: outras métricas quantitativas e subjetivas foram coletadas a partir de um questionário baseado no *Extreme Programming Framework* (XP-EF) [Williams et al. 2004][3].

4.2. Caracterização do Projeto

Nessa Seção serão apresentadas informações gerais do projeto Kalibro[4], que foi analisada no estudo de caso deste artigo. A ferramenta Kalibro foi projetada para ser incorporada a qualquer ferramenta de métricas de código fonte, extendendo essas ferramentas para

[3]Questionário disponível em: http://andvicente.wordpress.com/msc-project/

[4]Ferramenta Kalibro (Projeto Mezuro) - Disponível em: http://softwarelivre.org/mezuro/kalibro/

fornecer um fácil entendimento na avaliação de qualidade do software analisado. A Kalibro permite que um usuário experiente em métricas de software especifique intervalos de aceitação para cada métrica fornecida pela ferramenta base de métricas e permite a criação de métricas customizadas a partir das métricas nativas da ferramenta base. Além disso, a ferramenta permite a configuração de categorias e pesos de cada métrica, e resultados agregados de todo o código fonte ou resultados detalhados por classe.

Informações gerais do projeto Kalibro foram coletadas a partir de um questionário subjetivo que tem como objetivo coletar dados gerais sobre projetos que utilizam métodos ágeis e relacionar essas informações com as métricas de acompanhamento de teste e métricas de qualidade. Foram propostos neste trabalho duas novas categorias: a categoria de *"aderência a prática de teste de software"* e a categoria de *"maturidade da atividade de teste de software"*. Essas duas novas categorias foram baseadas no trabalho [Krebs 2002] e parte do questionário de métricas de aderência proposto por [Sato 2007].

A aderência a prática de teste é calculada de 0 a 4, conforme as práticas das seguintes categorias:

- **Atividades de teste de unidade automatizados [0- Nenhuma das atividades e 4- Segue todas as atividades]**: (1) testes de unidade automatizados existem para o código de produção, (2) uma ferramenta é utilizada para medir a cobertura de código, (3) há uma forma automatizada de executar todo o conjunto de casos de teste para todo o programa, (4) todos os casos de teste de unidade são executados e passam quando uma tarefa é finalizada e antes de integrarem o código, (5) quando estão sendo consertados os defeitos do software, testes de unidade são utilizados para capturar o defeito antes de ser reparado, (6) testes de unidade são refatorados, (7) testes de unidade são rápidos o bastante para serem executados com frequência.
- **Atividades de teste de aceitação (*Business Testing*) [0- Nenhuma importância e 4- Total importância]**: (1) testes de aceitação são utilizados para verificar uma funcionalidade do sistema e requisitos do cliente, (2) o cliente fornece o critério de aceitação, (3) o cliente usa os testes de aceitação para determinar o que foi terminado no fim de uma iteração, (4) o teste de aceitação é automatizado, (5) uma história não é considerada finalizada até que os testes de aceitação passem, (6) testes de aceitação são executados automaticamente toda noite, (7) um ambiente compatível com o ambiente do usuário final é utilizado para o teste.
- **Atividades do desenvolvimento dirigido a testes (TDD) [0- Não aplicado e 4- Totalmente aplicado]**: (1) código é desenvolvido somente após um teste de unidade (que falha) tenha sido escrito, (2) melhoria do código por meio de refatorações, (3) uso de padrões para criação de testes, buscando a testabilidade e qualidade dos testes, (4) os testes guiam o *design* do código-fonte, (5) todo código de produção é desenvolvido utilizando TDD.

A maturidade da atividade de teste de software é dividida em quatro categorias e elas são avaliadas da seguinte maneira:

- **Teste de unidade e TDD:** possui seis níveis, que vão desde o projeto que não possui nenhum teste formal (0) até o nível em que a equipe se preocupa com padrões para os testes, que incluem a preocupação com a testabilidade (5).
- **Teste de Aceitação e Teste de Sistema:** possue apenas dois níveis, o de teste de aceitação e o de teste de sistema. O nível é calculado conforme a porcentagem de opções selecionadas (de 0 a 5). Se forem selecionadas 50% das opções por exemplo, será atribuído um valor 2,5 e se não forem utilizados testes de aceitação e testes de sistema será atribuído o nível 0.

- **Aspectos de automatização do teste:** possui seis níveis que também serão calculados conforme a porcentagem de opções selecionadas (de 0 a 5). Se forem utilizados apenas testes manuais será atribuído o nível 0.
- **Processo de teste e melhoria contínua:** possui quatro níveis e o nível é calculado conforme a porcentagem de opções selecionadas (de 0 a 5). Se não forem utilizadas nenhuma das práticas sugeridas, será atribuído o nível 0.

O projeto Kalibro possui licença de software livre e foi desenvolvido por acadêmicos de doutorado, mestrado e graduação utilizando programação extrema (XP) e também a linguagem Java (J2SE). Para auxiliar a atividade de teste são utilizadas as seguintes práticas: testes de unidade, testes de GUI / usabilidade, métricas de código e testes. Em relação a ferramentas, o projeto utilizou a plataforma Eclipse como ambiente de desenvolvimento, e a ferramenta Xplanner e Noosfero para gerenciamento de projeto. Além disso, foram utilizadas as ferramentas JUnit (teste de unidade), Coverage (cobertura de casos de teste) e Fest (teste GUI) para auxiliar a atividade de teste de software.

4.3. Análise dos Resultados

Nessa seção será apresentada uma análise dos resultados das métricas de acompanhamento de testes, que foram coletados pelos *scripts* da ferramenta *JaBUTi* (cobertura de código), pela ferramenta ATMM (métricas do código fonte e acompanhamento de testes) e também pela ferramenta Kalibro (métricas de qualidade do código produzido). As métricas foram coletadas a partir do código fonte e os testes armazenados nos repositórios de código dos projetos, sendo que no projeto Kalibro foram analisadas quatro iterações (4 meses de projeto)

Foram medidos o nível de aderência à prática de teste de software e também a maturidade do projeto em relação à atividade de teste. O nível de aderência (de 0 à 4) foi coletado por valores objetivos do questionário. O nível de maturidade de cada prática de testes (de 0 à 5) também foi coletado conforme as opções selecionadas no questionário. Esses dados são apresentados na Tabela 4

Tabela 4. Kalibro - Níveis de aderência e maturidade da atividade de testes

Aderência à atividade de teste (0 - 4)		
Testes de Unidade Automatizados	3	
Testes de Aceitação com o Cliente (*Business Testing*)	1	
Desenvolvimento Dirigido a Testes (TDD)	2	
Maturidade da atividade de teste (0 - 5)		
Testes de Unidade e TDD	5	Máx (TDD e Padrões)
Testes de Aceitação e Testes de Sistema	2,5	1 de 2 (50%)
Aspectos de Automatização dos Testes	2	2 de 5 (40%)
Processo de Testes e Melhoria Contínua	3,75	3 de 4 (75%)
Total de Práticas de Teste	2,1	3 de 7 (42%)

Em termos da aderência à atividade de teste, foi constatado uma maior aderência relação aos testes de unidade automatizados e uma menor aderência a testes de aceitação e ao TDD. A maturidade da atividade de teste de unidade e TDD é grande no projeto Kalibro e também há uma grande preocupação com o processo de testes e melhoria contínua.

4.3.1. Evolução da quantidade de código e testes produzidos

A evolução da quantidade de classes, e código fonte e código de testes produzidos no projeto Kalibro é descrita por meio da Tabela 5 e da Figura 2. Constata-se que não houveram

grandes alterações no fator de teste na iteração 1 à iteração 3, no entanto, na iteração 4 foi produzido uma grande quantidade de código de testes (aumento de 66,26%), aumentando o fator de teste para 0,71 (Figura 2(b)).

Tabela 5. Kalibro - Métricas de Acompanhamento de Testes (Quantidade de Código e Testes)

Métrica	Iteração 1	Iteração 2	Iteração 3	Iteração 4
# de Classes SUT	49	47	49	54
# de Classes T	12	16	14	27
Métodos / Classes (SUT)	5.89	6.82	7.26	7.46
LOC SUT	2924	3098	3249	3649
LOC T	1184	1484	1580	2627
Fator de Teste (LOC SUT / LOC T)	0.40	0.47	0.48	0.71
LOC / Classes (SUT)	59.67	65.91	66.30	67.57
LOC / Classes (T)	98.66	92.75	112.85	97.29

(a) Total LOC (SUT e T)

(b) Fator de Teste (LOC SUT / LOC T)

Figura 2. Kalibro - Quantidade de linhas de código (LOC) e Fator de Teste

Adicionalmente também foi avaliada a qualidade do código produzido, utilizando algumas das métricas fornecidas pela ferramenta Kalibro. A própria ferramenta sugere os intervalos para que o resultados das métricas sejam considerados bons, ruins ou regulares [Oliveira Filho 2009][5]: **(i)** AMLOC (*Average Lines per Method* - Número médio de linhas por método), **(ii)** MMLOC (*Max Method LOC* - Número máximo de linhas em um método), **(iii)** LCOM4 (*Lack of Cohesion in Methods* - Ausência de coesão em métodos), **(iv)** ACC (*Afferent Connections per Class* - Conexões aferentes de uma classe) , **(v)** CBO (*Coupling Between Objects* — Ligações entre objetos).

Os valores médios das métricas AMLOC e MMLOC mostram que o projeto Kalibro possui métodos pequenos que facilitam o seu entendimento, no entanto ainda apresenta alguns métodos com uma quantidade de linhas maior do que a sugerida. Segundo a prática da refatoração, sempre que possível métodos com uma grande quantidade de linhas devem ser refatorados [Fowler 1999]. Em relação as métricas LCOM4, ACC e CBO que medem o grau de coesão e acoplamento, sugerem que a ferramenta Kalibro tem classes com um bom grau de coesão entre seus métodos segundo a métrica LCOM4 e um baixo acoplamento entre suas classes segundo as métricas ACC e CBO .

[5]Apesar de ser utilizada pela ferramenta Kalibro, a configuração dos intervalos está sendo avaliada e ainda não possui validação científica.

Tabela 6. Kalibro - Qualidade do código fonte (AMLOC, MMLOC, LCOM4, ACC, COF)

Métrica	Iteração 1	Iteração 2	Iteração 3	Iteração 4
AMLOC	11.55 (bom)	7.69 (bom)	8.06 (bom)	7.20 (bom)
MMLOC	157.0 (ruim)	86.0 (ruim)	86.0 (ruim)	68.0 (ruim)
LCOM4	2.65 (regular)	1.48 (bom)	1.56 (bom)	1.72 (bom)
ACC	3.16 (regular)	0.98 (bom)	1.10 (bom)	1.25 (bom)
CBO	2,18 (regular)	0.92 (bom)	1.04 (bom)	1.28 (bom)

4.3.2. Evolução do teste de unidade (Casos de Teste, Assertivas e Tempo de Execução)

No projeto Kalibro os testes de unidade foram criados utilizando a ferramenta JUnit e o *framework* Fest (para testes da GUI). Por meio da Tabela 7 é possível constatar uma evolução constante da quantidade de casos de teste e da quantidade de assertivas, sendo que da iteração 3 para iteração 4 houve um aumento de 70% na quantidade de casos de teste e 41,50% na quantidade de assertivas. A partir da relação entre esses dados de evolução dos casos de teste e assertivas e o aumento do valor do fator de teste, pode-se presumir que houve uma evolução da cobertura de código durante as iterações, principalmente da iteração 3 para iteração 4. O tempo total e o tempo médio de execução dos casos de teste de unidade mantiveram valores aceitáveis, permitindo que sejam executados com frequência, a cada alteração no código fonte.

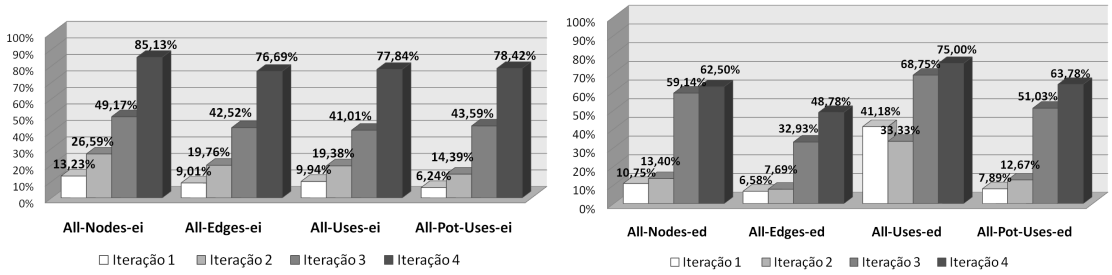
Tabela 7. Kalibro - Métricas de Acompanhamento de Testes (Testes de Unidade)

Métrica	Iteração 1	Iteração 2	Iteração 3	Iteração 4
Qtde casos de teste	60	87	103	175
Qtde assertivas	231	317	378	535
Casos de teste com Sucesso / Erro / Falha	19 / 40 / 1	85 / 1 / 1	102 / 1 / 0	173 / 1 / 1
Tempo Total	0,2s	19s	22s	287s
Tempo Médio de execução	0,016s	1,187s	1,571s	10,62s

4.3.3. Evolução da Cobertura de Código

Um dos benefícios do TDD, segundo [Beck 2002] é que a equipe consegue atingir 100% de cobertura dos testes. No entanto, alguns tipos de código são inerentemente difíceis de testar utilizando TDD (como por exemplo teste de GUI) [George and Williams 2004]. A equipe deve encontrar meios de testar esse tipo de código, utilizando um *framework*, ferramenta ou até mesmo executando testes manuais [Martin 2007]. Apesar de alguns estudos experimentais demonstrarem que a utilização do TDD fornece códigos próximos do 100% para cobertura de linhas e desvios do código [George and Williams 2004], [Madeyski 2010] afirma que o impacto da prática do TDD em termos de cobertura de código é ainda não conclusivo. Programadores que utilizam o TDD para escrever teste de unidade, podem ser beneficiados por medidas que indiquem quando o software foi testado de forma eficiente [Madeyski 2010]. Nesse contexto é apresentado na Figura 3 a evolução da cobertura do código do projeto Kalibro utilizando critérios de teste estruturais da ferramenta *JaBUTi*.

Observa-se que no projeto Kalibro houve uma grande evolução na cobertura do



(a) Critérios estruturais EI - *Execution Independent* (b) Critérios estruturais ED - *Execution Dependent*

Figura 3. Kalibro - Cobertura de Testes utilizando Critérios de Fluxo de Controle e Fluxo de Dados

código, sendo que a equipe constantemente se dedica a desenvolver novos casos de teste para melhoria da cobertura. A preocupação com os testes pode ser constatada tanto pela evolução da cobertura dos testes, como nos diversos comentários nas revisões do repositório de código, que informam a adição de mais testes, além dos documentos das versões entregues que mostram a cobertura alcançada nos pacotes de classes do projeto. Outro aspecto interessante do projeto foi a utilização do *framework* Fest para testar as classes da interface gráfica da ferramenta. Esses testes contribuíram para o aumento da cobertura do código de todo o projeto.

5. Conclusões e Trabalhos Futuros

Os principais resultados obtidos a partir da avaliação das informações coletadas pelo questionário e pelas demais métricas avaliadas no estudo de caso foram:

- **Aderência e maturidade da atividade de testes:** apesar do projeto Kalibro possuir um nível alto em relação aos testes de unidade automatizados, os testes de aceitação e a estratégia TDD ainda não são utilizados totalmente. Em relação a maturidade da atividade de teste, foi constatado que os testes de unidade já possuem uma grande maturidade e também há uma preocupação com o processo de testes e melhoria contínua.
- **Evolução do código fonte,testes e cobertura do código fonte:** analisando a evolução do código e dos testes no projeto Kalibro, foi possível verificar iterações que deram uma maior ênfase a quantidade de testes criados. Essas iterações trouxeram um aumento visível da cobertura de código. Além disso, a utilização de um *framework* para o teste de classes de interface gráfica no projeto Kalibro, contribuiu para o aumento da porcentagem de cobertura.
- **Qualidade do código:** além da equipe de desenvolvimento se preocupar em gerar códigos com o mínimo de defeitos, a ferramenta Kalibro possui métodos pequenos que facilitam o seu entendimento, tem classes com um bom grau de coesão entre seus métodos e um baixo acoplamento entre suas classes.

A partir dos resultados obtidos no estudo de caso foi possível analisar a evolução do projeto Kalibro durante suas iterações de desenvolvimento, que evidenciou uma preocupação com a melhoria dos casos de teste. Por meio do estudo de caso, pode-se confirmar com suas devidas limitações a hipótese de que a utilização de métricas de teste pode colaborar na detecção de problemas durante a condução dessa atividade, além de monitorar a evolução da qualidade dos casos de teste, que reflete na qualidade do código fonte produzido. Esta hipótese foi confirmada pelo projeto Kalibro que apresentou bons resultados em termos de métricas de acompanhamento de teste de unidade que resultaram em uma alta cobertura de requisitos de teste, com poucos defeitos nos testes de unidade

e que possivelmente se refletiu no bom desempenho das métricas ligadas a qualidade do código fonte.

A coleta de métricas de acompanhamento da atividade de teste fornece um *feedback* constante sobre os conjunto de casos de teste que são desenvolvidos durante as iterações do projeto. A partir desse cenário, é possível que a equipe de desenvolvimento e os gerentes de projeto utilizem as informações coletadas em reuniões diárias e retrospectivas de iteração para a melhoria contínua do processo de teste e também dos artefatos de teste produzidos. As informações podem detectar problemas ou evoluções em termos da qualidade dos casos de teste produzidos. Além disso, a equipe pode estabelecer uma metodologia na qual são estabelecidas metas de qualidade para as métricas de acompanhamento de teste antes do início de uma iteração. As métricas podem ser gerenciadas em um intervalo de tempo pré-estabelecido (diariamente ou por iterações) e podem ser estabelecidos intervalos de referência para que um determinado valor de uma métrica seja considerado bom, regular ou ruim.

Também foi constatada a importância de utilizar-se ferramentas de apoio como a ferramenta ATMM. Utilizando essas ferramentas é possível coletar as métricas rapidamente, utilizando uma interpretação única a respeito de cada métrica, com saídas que possam ser facilmente interpretadas por desenvolvedores e gerentes de projeto.

Para dar continuidade as atividades apresentadas neste artigo, poderão ser realizados os seguintes trabalhos futuros: validação mais aprofundada das métricas utilizando métodos de validação teórica e experimental, definir valores de referências para as métricas, além de compor novas métricas de acompanhamento de teste e conduzir o estudo de caso em disciplinas acadêmicas e projetos ágeis da indústria durante o seu desenvolvimento. Além disso, seria interesante desenvolver algumas melhorias a ferramenta ATMM para dar suporte a métricas de outras linguagens, métricas de teste de aceitação, possibilidade de coletar o código diretamente do repositório de código e integração com a ferramenta Kalibro para que seja possível configurar valores de referência para as métricas.

Agradecimentos

Este trabalho contou com apoio financeiro do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq).

Referências

Ambu, W., Concas, G., Marchesi, M., and Pinna, S. (2006). Studying the evolution of quality metrics in an agile/distributed project. In *Extreme Programming and Agile Processes in Software Engineering, XP 2006, Oulu, Finland*, pages 85–93.

Beck, K. (2002). *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Begel, A. and Nagappan, N. (2007). Usage and perceptions of agile software development in an industrial context: An exploratory study. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, pages 255–264. IEEE.

Crispin, L. and Gregory, J. (2009). *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley Professional.

Davidsson, M., Zheng, J., Nagappan, N., Williams, L., and Vouk, M. (2004). Gert: an empirical reliability estimation and testing feedback tool. In *15th International Symposium on Software Reliability Engineering (ISSRE 2004)*, pages 269–280. IEEE.

Dybå, T. and Dingsøyr, T. (2009). What do we know about agile software development? *IEEE Software*, 26:6–9.

Fowler, M. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

George, B. and Williams, L. (2004). A structured experiment of test-driven development. *Information and Software Technology*, 46(5):337 – 342.

Hartmann, D. and Dymond, R. (2006). Appropriate agile measurement: Using metrics and diagnostics to deliver business value. In *Proceedings of Agile Conference 2006 (AGILE '06). Minnesota, USA*, pages 126–134, Washington, DC, USA. IEEE.

Knoernschild, K. (2006). Using metrics to help drive agile software. Agile Journal. Disponível em: http://www.agilejournal.com/articles/columns/the-agile-developer/56-using-metrics-to-help-drive-agile-software. Acesso em: 15/01/2010.

Krebs, W. (2002). Turning the knobs: A coaching pattern for xp through agile metrics. In *Extreme Programming and Agile Methods - XP/Agile Universe 2002, Chicago, IL, USA*, pages 60–69. Springer.

Kulik, P. (2000). A practical approach to software metrics. *IT Professional*, 2(1):38–42.

Madeyski, L. (2010). The impact of test-first programming on branch coverage and mutation score indicator of unit tests: An experiment. *Information & Software Technology*, 52(2):169–184.

Martin, R. C. (2007). Professionalism and test-driven development. *IEEE Software*, 24(3):32–36.

Nagappan, N. (2004). Toward a software testing and reliability early warning metric suite. In *26th International Conference on Software Engineering (ICSE 2004), Edinburgh, United Kingdom*, pages 60–62. IEEE Computer Society.

Nagappan, N., Williams, L., Osborne, J., Vouk, M., and Abrahamsson, P. (2005). Providing test quality feedback using static source code and automatic test suite metrics. In *16th nternational Symposium on Software Reliability Engineering*, pages 85–94. IEEE.

Oliveira Filho, C. M. d. (2009). Kalibro: Uma ferramenta de configuração e interpretação de métricas de código-fonte. Trabalho de Conclusão de Curso, IME/USP.

Paetsch, F. (2003). *Requirements Engineering in Agile Software Development*. PhD thesis, University of Calgary, Calgary, Alberta.

Pressman, R. S. (2006). *Engenharia de Software*. McGraw-Hill, São Paulo, 6 edition.

Sato, D., Goldman, A., and Kon, F. (2007). Tracking the evolution of object-oriented quality metrics on agile projects. In *Proceedings of Agile Processes in Software Engineering and Extreme Programming (XP 2007)*, pages 84–92. Springer.

Sato, D. T. (2007). Uso eficaz de métricas em métodos Ágeis de desenvolvimento de software. Master's thesis, IME–USP, São Paulo, SP.

Simons, A. J. H. (2005). Testing with guarantees and the failure of regression testing in extreme programming. In *Proceedings of Extreme Programming and Agile Processes in Software Engineering (XP 2005)*, pages 118–126. Springer.

Vicente, A. A., Delamaro, M. E., and Maldonado, J. C. (2009). Uma Revisão Sistemática sobre a Atividade de Teste de Software em Métodos Ágeis. In *XXXV Conferencia Latinoamericana de Informática (XXXV CLEI)*, Pelotas - RS, Brasil. CLEI.

Vincenzi, A. M. R. (2004). *Orientação a Objetos: Definição, Implementação e Análise de Recursos de Teste e Validação*. PhD thesis, ICMC-USP, São Carlos, SP.

Williams, L., Krebs, W., Layman, L., Antón, A. I., and Abrahamsson, P. (2004). Toward a framework for evaluating extreme programming. In *Empirical Assessment in Software Engineering (EASE), Edinburgh, Scotland*, pages 11–20.

Sessão 2

Ferramentas e técnicas para métodos ágeis

Software Process Improvement in Agile Software Development

Célio Santana[2, 3], Mariana Cerviño[1], Danilo Caetano[2], Cristine Gusmão[2]

1 Departamento de Sistemas Computacionais, Escola Politécnica de Pernambuco (POLI) Universidade de Pernambuco (UPE) Rua Benfica, 445, Madalena – 50.751-460 Recife, PE – Brasil

2 Curso de Sistemas de Informação - Faculdades Integradas Barro Melo Av. Transamazônica, 405, Jardim Brasil – 53.300-240 Olinda, PE – Brasil

3 Centro de Informática – Universidade Federal de Pernambuco Caixa Postal 7.851 – 50.732-97 – Recife – PE – Brasil

{maricervino, celio.santana, danilocaetano1987, cristinegusmao} @gmail.com

Abstract. *Organizations are increasingly adopting agile methodologies and traditional methods in their software development process. This emerging scenario brings new challenges and opportunities for process improvement software. The Nokia Test is an effective method to assess how different teams are using Scrum seeking their "adherence" to the procedure proposed by this agile method. Quality and productivity are the key indicators when using traditional maturity model of software process improvement CMMI. The aim of this study is analyze the relationship between the traditional methods of software process improvements that emphasizes continuous improvement of the organizational development. But the agile approach of software development is iterative, encourage improvements in software development increasing the individual effectiveness into the team.*

Keywords: Scrum, Agile software development, Nokia Test, CMMI, Software Process Improvement.

1 Introduction

Software process improvement (SPI) has become a practical tool for companies where the quality of the software is of high value [Jarvinem 1994]. In a technical report with results from thirteen organizations, and with the number of post-release defect reports used as a measure, Herbsleb and

others practitioners (1994) showed that due to software process improvement (SPI), the products and business value (especially return on investment – ROI) was improved.

It is generally considered that a well documented and a repeatable process is essential for developing software products of high quality [Stamelos & Sfetsos 2007]. There is also evidence that the use of standards and process assessment models has a positive impact on the quality of the final software product [Kitchenham 1996].

The software engineering community has gradually moved from product quality-centered corrective methods to process quality-centered, preventive methods, thus shifting the emphasis from product quality improvement to process quality improvement. Inspections at the end of the production line have long been replaced by design walkthroughs and built-in quality assurance techniques throughout the development life cycle [Georgiadou 2004].

In traditional SPI methods and approaches the aspect of organizational improvement has usually been placed in a central role, due to the fact that the planning and control of the SPI initiatives are managed by the organizational stakeholders [Basili & Caldiera 1994, SEI 2006]. The reported positive effects of SPI methods and approaches include reducing indicators such as time to market, risks and costs, and increasing the productivity and quality in software development organizations [Krasner 1999, Van Soligen & Berghout 1999].

However, various negative effects have also been encountered, e.g. regarding the cost-effectiveness of SPI initiatives, their actual effectiveness in improving the software development practices of organizations, the volume of the effort needed to implement SPI initiatives and the low speed at which visible and concrete results are achieved [Krasner 1999, Goldensen & Herbsleb 1995, Dybå 2000].

In fact, it has been reported that around two-thirds of SPI traditional initiatives fail to achieve their intended goals [Debou 1999]. From the mid-1990s onwards, agile software development principles and methodologies have been increasingly challenging the traditional view of software development, and therefore, provides a highly untraditional approach to SPI, in which the process improvement knowledge of software developers and software development teams is acknowledged and valued [Salo 2007].

We have long known that skilled people are the most crucial resource in software development. Bach (1994) stated that “Everyone knows the best way to improve software productivity and quality is to focus on people.” However, it took more than 10 years for the agile manifesto and agile methods (extreme programming, scrum, crystal, and many others) to truly place the emphasis on people and their interaction [Stamelos & Sfetsos 2007].

This paper presents a state of the art review about SPI in agile software development, this work is part of two *MSc* dissertations and in this light after this introductory section, the Section 2

explores software process improvement, the Section 3 explores the Software Process Improvement in Agile Software Development, the Section 4 will present the conclusions of the paper.

2 Software Process Improvement

A software process can be defined as the sequence of steps required to develop or maintain software aims at providing the technical and management framework for applying methods, tools, and people to the software task [Humphrey 1995]. However, even the most exquisitely defined and managed process may still not meet the context specific needs and objectives of software development organizations and customers regarding, for example, performance, stability, compliance and capability [Florac et al. 2000].

Thus, SPI aims providing software development organizations with mechanisms for evaluating their existing processes, identifying possibilities for improving as well as implementing and evaluating the impact of improvements [Florac et al. 2000].

Traditionally, the ultimate goal of SPI in organizations is to provide a Return on Investment (ROI) for the organization through the improvement activities yielding more money than is spent on them [Rico 2004]. ROI has been reported for various SPI achievements, such as improved efficiency of the development process and reduction of total software costs, increased quality of theend product, higher predictability of cost and schedule, and increased level of reuse (Krasner 1999).

The focus on quality in SPI is based on the fundamental ideology that quality-driven development is likely to yield not only better quality but also lower cost and improvement of competitive position [Deming 1990].

One of the characteristics of SPI, as traditionally defined, is its emphasis on the continuous improvement of organizational software development processes in terms of performance, stability, compliance, and capability, for instance. Often the existing SPI methods and approaches seem to enhance the underlying business goals and needs in the improvement of organizational software development processes [Salo 2007]. Traditionally, SPI initiatives are also strictly controlled and managed by the organizational stakeholders (Boehm & Turner 2003).

Salo (2007) identified six different elements in the context of traditional SPI: (1) The organizational models of continuous improvement, (2) standard processes and assessments, (3) tailoring, (4) deployment, (5) measurement, and (6) the utilization of knowledge and learning in SPI.

The focus of this work is research related to (1) The organizational models of continuous improvement, (2) standard processes and assessments because these subjects were investigated widely in previous researches. The others subjects were not studied or adopted in software industry enough to present a substantial content about the subject, once that researches are just found in Salo (2007) work.

3 Software Process Improvement in Agile Context

Salo (2007) states that agile software development provides new possibilities for conducting SPI, which may well provide grounds for meeting some of the central challenges of traditional SPI. agile software development provides a highly untraditional approach to SPI, in which the process improvement knowledge of software developers and software development teams is acknowledged and valued.

This difference comes from the idea of the traditional goal of a software process is to provide high predictability, stability, and repeatability using highly managed and quantitatively monitored software development processes. On the other hand, agile principles highlight the need for the software process to be flexible, to be able to rapidly respond to the constant changes and context specific needs of software development [Salo 2007].

As a result, traditional software development emphasizes up-front contract negotiations where the requirements, cost and schedule of the product development are fixed and the end product will be delivered at the end of the project lifecycle. In this mode of software development, traditionally, extensive documentation and quantitative monitoring of the product development process plays a central role. The principles and practices of agile software development, in turn, address the constant changes. [Salo 2007].

The following subsections will present what of the aspects identified in researches based in the six elements is software process improvement stated by Salo (2007) and is related with agile software development.

3.1 Agile Software Process Improvement Based on Team Behavior (Salo 2007)

Salo (2007) states when considering the relationship between agile software development and SPI, there are three principles, in particular, of the agile manifesto that deserve attention: the valuing of individuals and interactions over processes and tools, the principle that encourages regular reflection by software development teams in order to become more effective, and the self-organization of software development teams. Taking regular improvement within project teams as one of the twelve principles of agile software development highlights the importance of continuous improvement also in the agile software development context. In order to welcome changes throughout the agile software development project, whether they concern product requirements or technical aspects, the software process with its practices, methods, and tools must be able to adapt to the specific context while also to respond to the changes when needed.

Salo (2007) also states that traditionally, SPI has been approached in a top-down manner, in which the organizational level has played a major role in defining the goals of SPI and planning, managing, and controlling the SPI initiatives. In the agile software development context, on the contrary, the experience and knowledge of software developers and the self-organization of software developers in improving and adapting their daily working practices have been clearly placed in a central role.

In the agile approach, the role of management is to organize and co-ordinate rather than plan, execute, or control. Furthermore, the contextual needs for improving and adapting processes throughout the entire development process are emphasized, rather than the organizational goals in the regular SPI activities of development teams [Salo 2007].

The strong difference between the traditional and agile SPI could be seen when Salo (2007) compares both approaches considering the SPI element organizational models of continuous improvement, these differences are shown in Table 1. The agile view is entirely based on team behavior.

Table 1: Organizational SPI Models [Lycett at al 2003].

Characteristic	Plan-Driven View	Agile View
SPI Approach	Top-down	Bottom-up
Primary Goal	Organisational procedures for improving the organisational software process(es)	Adapting the process to the contextual needs of individual project teams
		Improving the effectiveness of individual project teams
SPI Control	Organisational control of SPI	Self-organisation of teams
Knowledge Transfer	Explicit knowledge: external knowledge capture and inert knowledge transfer to support a learning paradigm (Lycett et al. 2003)	Face-to-face communication Tacit knowledge: Establishing and updating project knowledge in the participants' heads rather than in documents. (Boehm & Turner 2005)
Basis for Improvement	Organisational Goals Measurements	Contextual needs Experience and Learning of Software Developers Regular team reflections

3.2 Agile SPI Based on tools automation - Process Centered Software Engineering Environments

Gruhn (2002) states that software process denotes the set of all activities which are carried out in the context of a concrete software development project. It usually covers aspects of software development, quality management, configuration management and project management. The description of a software process is called a software process model. A software process model does not only describe the activities which have to be executed, but also the tools to be used, the types of objects and documents to be created/manipulated and the roles of persons involved in software development. One of the key questions for supporting software processes is how software process models and software

engineering environments are related or – in other words – how a supporting infrastructure can be derived from the information given in a software process model.

Software engineering environments (SEEs) are meant to support software development. Process-centered software engineering environments (PCSEEs) give up the notion of a predefined process model. Process models used in this way should define which software development activities are to be executed when and by whom, they should identify tools to be used and the format of documents to be created and/or manipulated.

Then, the process model is interpreted at runtime to identify which process behavior has to be supported. In concrete this means, that software developers are reminded of activities which have to be carried out, automatic activities are executed without human interaction and consistency between documents is enforced up to a certain level [Gruhn 2002].

Actually PCSEEs is closely related to the purpose of software process improvement. The findings gathered during process modeling and the results of process model analysis usually indicate how actual processes can be improved in order to contribute to improved process productivity and product quality [Gruhn 2002].

Four elements of PCSEE are important to understanding tools automation in this context these elements are [Gruhn 2002]:

Workflow management: The idea of workflow management is that process models are used as basis of real processes, i.e. real processes are governed on the basis of the underlying process models. A workflow engine checks process models and the actual process state in order to identify which activities can be executed and lets potential participants know about it.

Automation: The process modeling purpose of automation is closely related to the purpose of workflow management. While workflow management tries to automate the coordination between activities, the immediate automation of single activities is another purpose of process modeling. In the context of software processes, activities like checking in and out of configuration units, building releases, evaluating test set coverage etc. usually can be automated. If automation is a major goal, then at least the preconditions for activities to be automated have to be described in detail.

Monitoring: Another purpose of process modeling is to measure the deviation between what processes are supposed to look like and how they actually behave. Then, traces of real world processes can be used to identify what the deviations are. Usually, process monitoring is not a self-contained purpose, but only a first step towards SPIor process automation.

Deriving support for tool integration: Based on the assumption that not all parts of software processes are suited for workflow management support and that different parts of software processes deserve different kinds of support and based on the experience that different software processes require different tools, the integration of which results in process specific challenges, a modest – even though worthwhile – purpose of process modeling is to identify the requirements for tool integration in the context of a concrete process model.

The first value of Agile Manifesto (2001): *Individuals and interactions over processes and tools*, lead the agile practitioners to give little importance to using tools. But, seven years later, one of the agile manifesto signatories, Brian Marick (2008) stated that the first agile value should be revised once that tools help teams to deal with non creativity tasks.

Many agile teams are adopting tools to automate their process. Beck (1999) states that teams must have one computer testing and integrating source code all time. Adopting the four elements of PCSEE purposed by Ghrun (2002) and seeking their use in agile in teams

Workflow management: Many Agile teams are using issues tracking tools to control the stories flow inside the teams [Abrahamsson et al 2002]. In this scenario, these tools are acting as workflow management tools.

Automation: Agile teams are using continuous integration servers and tests robots to maintain the high quality of the final product [Abrahamsson et al 2002]. When automating these parts, the team releases its members for doing creativity tasks while automated tasks ensure that there will not be occasional failures in the process.

Monitoring: Spreadsheets, Charts, Wiki are commons tools for monitoring the progress of the projects. But many teams are including process metrics like builds broken or tests cases wrote per story [Jackobsen 2009]. This is an example of tools monitoring not the project but the process as well.

Deriving support for tool integration: Agile teams usually use different tools for performing different tasks [Abrahamsson et al 2002]. The integration of all these tools becomes painful for agile teams challenging them. No one of these elements where formally studied in agile SPI perspective.

3.3 Agile Assessment Methods – Nokia Test

Following the Salo's (2007) definition for SPIelements, this subsection refers about the element (2) processes and assessments. An urgent need has been recognized for a set of guidelines for agility-compatible standard process maturity assessments and also for a set of standards for the acknowledgement of agile methods by lead assessors (Boehm & Turner 2005).

The agile assessment method has been suggested as providing a lightweight approach for assessment to identify and adopt the most suitable agile methods amongst the existing organizational practices [Pikkarainen & Passoja 2005]. Furthermore, techniques have been suggested for increasing the agility level of a software development team by assessing the current agility level against the defined agility goals [Lappo & Andrew 2004].

Thus, the current discussion of process assessments in the agile context does not so much address the certification or define the maturity of the organizational software development processes, but rather evaluates the purpose of adopting agile practices (Salo 2007).

In this context was emerged the Nokia Test [Vodde 2006]. In 2005, Bas Vodde was coaching teams at Nokia Networks in Finland and developed the first Nokia Test focused on agile practices. He had hundreds of teams and wanted a simple way to determine if each team was doing the basics.

It does not provide the secret sauce for hyper performing teams. However, it is the first line of the recipe for high performance. These tests were applied in scrum teams at OpenView Venture Partners and to their portfolio companies as the venture group does not expect good performance from Scrum teams without passing the Nokia Test. They are also very interested in predictability of release dates which is impossible without passing grades on the test [Sutherland 2008].

The test consists in eight questions about how the teams are adopting Scrum. The Nokia Test is in two parts. The first part consists of three questions asking if the team is doing iterative development. The questions are [Sutherland 2008]:

- Iterations must be timeboxed to less than 4 weeks?
- Software features must be tested and working at the end of each iteration?
- The iteration must start before specification is complete?

The next part of the test checks whether the team is doing Scrum, in view of the Nokia, understanding, the remaining five questions are:

- You know who the product owner is?
- There is a product backlog prioritized by business value?
- The product backlog has estimates created by the team?
- The team generates burndown charts and knows their velocity?
- There are no project managers (or anyone else) disrupting the work of the team?

In 2008 Jeff Sutherland developed a Nokia Test scoring system. Each person on the team takes a sheet of paper and prepares to score eight questions on a scale of 1-10 and teams must average their score. In 2009, a team question was added to the Nokia test about teams. Sutherland created four categories for Nokia Tests results and linked these averages with following revenues [Sutherland 2008]:

- ScrumBut (Average < 8,0) - revenue up 0-35%
- Pretty Good Scrum (8,0 <= Average > 9,0) – revenue up 150% - 200%
- Good Scrum (9,0 <= Average > 10,0) - revenue up 300%
- Great Scrum (Average = 10) - annual revenue up 400%

Sutherland (2008) stated that in the initial evaluations the teams are average 4.0, but in six months their average up to 7,0 and the velocity is about 300% of initial velocity. The Return of Investment (ROI) is about 11000% on first year since that is relatively easy and cheap to change scrum process for better. Answering the Nokia Test provide to the team some idea about how much it is really doing Scrum.

3.4 Agile Assessment Methods – Comparative Agility

Kenny Rubin & Mike Cohn (2007) launched the Comparative Agility Survey. The Comparative Agility™ assessment is based on a simple, but powerful concept, determine how good is one company compared to it competitors. Rubin & Cohn assume agile teams and organizations strive always to be better than their competition and their past selves.
Comparative Agility™ assessments present the results of a set of survey responses in comparison to some other set of responses. Using Comparative Agility it is possible to compare a team, project or organization to:

- The total set of collected responses; responses from organizations in the same industry;
- Responses from similar types of projects (such as commercial software, websites, and so on);

- Responses from projects with similar lengths of experience at becoming agile.

The approximately 100 questions of a Comparative Agility™ assessment are organized into seven dimensions and thirty-two characteristics. The seven dimensions represent broad classifications of changes to be expected of a team or organization as it becomes more agile. The seven dimensions are:

- teamwork
- requirements
- planning
- technical practices
- quality
- culture
- knowledge creation

Each dimension is made up of three to six characteristics and a set of questions is asked to assess a team's score on each characteristic. Questions are answered on a fivepoint Likert scale of:

- True
- More true than false
- Neither true nor false
- More false than true
- False

Through a combination of Dimensions, Characteristics, and individual Questions, a team or organization can see how they compare to other organizations, or to themselves. The score ranges start in -2 (worst scenario) until 2 (Best Scenario). The average Scenario shows the 0 (zero) grade. The study involving this survey was presented in Cohn (2009).

3.5 Agile and Standards Process

The compatibility of agile software development approaches with the existing standard process models is one SPI issue that has been addressed in agile literature [Salo 2007]. One central problem has been posed as follows: How do you merge agile, lightweight processes with standard industrial processes without either killing agility or undermining the years you have spent defining and refining your systems and software engineering process assets [Boehm & Turner 2005]?

Some agile proponents have argued that people willing to spend money on CMM® certification are less interested in the agile value proposition, while those needing agility for business reasons are less interested in getting CMM or ISO 9000 certification [Williams & Cockburn 2003]. Nevertheless, mature software organizations especially are concerned about how the adoption of agile processes will affect their assessment ratings [Boehm & Turner 2005].

It has been argued that the synergy [Paulk 2001] and philosophical compatibility [Reifer 2003] of XP and CMM® have been agreed upon among most of the leaders in the field. However, shortages also between the ISO and CMM requirements and agile methodologies have been reported, along with a lack of practices to support the commitment of management to the defined software development process, and also regarding the setting up and staffing of an independent quality assurance group [Vriens 2003].

In addition, the degree of documentation and the infrastructure required by current process standards for lower-level certification are issues of concern [Boehm & Turner 2005].

From 2003 onwards, however, the IEEE Standards Association has conducted agile standardization work in the IEEE 1648 working group[1] to establish and manage the expectations to an agile customer should when working together with an agile team.

4 Conclusion

SPIwas extensively studied in traditional software development, on the subject of SPI in the context of agile software development there are several avenues of investigation to be explored in future research. Even researches related to SPI on agile environments are studied in another point of view and this is a complicating factor for studying agile SPI in the light of traditional and established methods in traditional SPI.

Looking for the six elements purposed by Salo (2007), there are several avenues of investigation to be explored in future research either by lack of direction from the subject to process improvement or the lack of research itself. These lacks of research will be explored in the two MSc dissertations that were initiated by this literature review, but also may help other researchers to identify what kind of research could be performed when software process improvement is related to agile.

[1] http://standards.ieee.org/board/nes/projects/1648.pdf

References

[Abrahamsson et al 2002] Abrahamsson, P., Salo, O., Ronkainen, J., Warsta, J.: Agile Software Development Methods Available in: http://www.pss-europe.com/P478.pdf, last access in 3/22/2010. (2002)

[Agile Manifesto 2001] Agile Alliance: Agile Manifesto for Software Development: Available in www.agilemanifesto.org, last access in 03/22/2010.

[Bach 1994] Bach, J.: The Immaturity of CMM, American Programmer Magazine Vol 3, pp 7--8 (1994)

[Basili & Caldiera 1994] Basili, V. R., Caldiera, G.: Experience Factory. In: Encyclopedia of Software Engineering. Marciniak, J. J. (ed.). John Wiley & Sons, Inc. 469--476 (1994)

[Beck 1999] Beck, K.: Extreme Programming Explained – Embrace Change. Addison-Wesley. (1999)

[Boehm & Turner 2003] Boehm, B. & Turner, R.: Using Risk to Balance Agile and Plan- Driven Methods. Computer, Vol. 36, 6 (6), June, pp. 57-66, (2003)

[Boehm & Turner 2005] Boehm, B. & Turner, R.: Management Challenges to Implementing Agile Processes in Traditional Development Organizations. IEEE Software, Vol. 22(5), September-October, pp. 30-39, (2005)

[Cohn 2009] Cohn, M.: Succeeding with Agile: Software Development using Scrum.Addison Wesley Longman, Inc. 504 p. (2009)

[Deming 1990] Deming, W. E.: Out of the Crisis. 10 Printing ed. Massachussetts Institute of Technology, Center for Advanced Engineering Study. Cambridge (1990)

[Debou 1999] Debou, C.: Goal-Based Software Process Improvement Planning. In: Better software practice for business benefit: Principles and experience. Messnarz, R. & Tully, C. (ed.) IEEE Computer Society, pp 107--150, Los Alamitos, CA. (1999)

[Dybå 2000] Dybå, T.: An Instrument for Measuring the Key Factors of Success in Software Process Improvement. Empirical Software Engineering, Vol. 5, 357-390 (2000)

[Florac et al 2000] Florac, W. A., Carleton, A. D. & Barnard, J. R.: Statistical Process Control: Analyzing a Space Shuttle Onboard Software Process. IEEE Software, Vol. 17, 4 (4). July, August, pp. 97-106 (2000)

[Georgiadou 2004] Goergiadou E.: Software Process and Product Quality Assurance: A historical Perspective. Cybernetics and Systems Analysis. 11(4), pp 19--27 (2004)

[Goldensen & Herbsleb 1995] Goldensen, D. R., Herbsleb, J. D.: After the Appraisal: A Systematic Survey of Process Improvement, Its Benefits, and Factors that Influence Success. CMU/SEI-95-TR-009. Software Engineering Institute. Pittsburgh. (1995)

[Gruhn 2002] Gruhn, V.: Process-Centered Software Engineering Environments A Brief History and Future Challenges. In: Annals of Software Engineering, 14, pp 363--382,(2002)

[Herbsleb et al 1994] Herbsleb, J., Carleton, A., Rozum, J., Siegel, J., Zubrow, D.: Benefits of CMM-based software process improvement: Initial results. Technical report, CMU/SEI- 94-TR-13 (1994)

[Humphrey 1995] Humphrey, W. S.: A Discipline for Software Engineering. Addison Wesley Longman, Inc. 242 p. (1995)

[Jackobsen 2009] Jackobsen, C.: Scrum and CMMI: from Good to Great - are you ready - ready to be done – done. In Proceedings of Agile Conference 2009. (2009)

[Jarvinem 1994] Järvinen, J.: On comparing process assessment results: BOOTSTRAP and CMM. In Software Quality Management, SQM94, pp 247--261, Edinburgh (1994)

[Kitchenham 1996] Kitchenham, B., Pfleeger, S. L.: Software quality: The elusive target. In IEEE Software, 13(1), pp 12--21. (1996)

[Krasner 1999] Krasner, H.: The Payoff for Software Process Improvement: What it is and How to Get it. In: Elements of Software Process Assessment & Improvement. El Emam, K.& Madhavji, N. H. (ed.). IEEE Computer Society, PP 113-130, Los Alamitos, California(1999)

[Lappo & Andrew 2004] Lappo, P. & Andrew, H. C. T.: Assessing Agility. In: The proceedings of the Extreme Programming and Agile Processes in Software Engineering. June, 2004. Garmisch-Partenkirchen, Germany. Eckstein, J. & Baumeister, H. (ed.). Springer. pp. 331--338. (2004)

[Lycett at al 2003] Lycett, M.; Macredie, R. D.; Patel, C.; Paul, R. J. Migrating Agile Methods to Standardized Development Practice. In: Computer, pp. 79-85. IEEE Computer Society. Jun. 2003

[Marick 2008] Marick, B.: Seven Years Later: What the Agile Manifesto Left Out. In:Agile Development Practices (2008)

[Paulk 2001] Paulk, M. C.: Extreme Programming from a CMM Perspective. Software, Vol. 18, 6 (6), Nov. Dec, pp. 19--26. (2001)

[Pikkarainen & Passoja 2005] Pikkarainen, M. & Passoja, U.: An Approach for Assessing Suitability of Agile Solutions: A Case Study. In: The proceedings of the Sixth International Conference on

Extreme Programming and Agile Processes in Software Engineering. Sheffield University, UK. pp. 171--179. (2005).

[Reifer 2003] Reifer, D. J.: XP and the CMM. IEEE Software, Vol. 20, 3 (3), May/June, pp. 14--15. (2003)

[Rico 2004] Rico, D. F.: ROI of Software Process Improvement: Metrics for Project Managers and Software Engineers. J. Ross Publishing. Florida, U.S.A. (2004)

[Rubin & Cohn] Rubin, K., Cohn, M.: Comparative Agility Survey. Available in http://www.comparativeagility.com/, last visit in 03/22/2010.

[Salo 2007] Salo, O.: Enabling Software Process Improvement in Agile Software Development Teams and Organisations, PhD Thesis, Oulu, (2007)

[SEI 2006] Software Engineering Institute.: Capability Maturity Model® Integration (CMMISM), Version 1.2. Carnegie Mellon Software Engineering Institute. (2006)

[Stamelos & Sfetsos 2007] Stamelos, I., Sfetsos, P.: Agile Software Development Quality Assurance. Information Science Reference, (2007)

[Sutherland 2008] Sutherland, J.: Money for nothing: And your change for free – Agile Contracts. In Proceedings of Agile Development Conference (2008)

[Van Soligen & Berghout 1999] Van Solingen, R., Berghout, E.: The Goal/Question/Metric Method: A Practical Guide for Quality Improvement of Software Development. The McGraw-Hill Companies. (1999)

[Vodde 2006] Vodde, B.: Nokia Networks and Agile Development. In Proceedings of EuroMicro Conference. (2006)

[Vriens 2003] Vriens, C.: Certifying for CMM Level 2 and ISO9001 with XP@Scrum. In: The proceedings of the Agile Development Conference (ADC'03). September, 2003. IEEE Computer Society. pp. 120--124. (2003)

[Williams & Cockburn 2003] Williams, L. & Cockburn, A.: Agile Software Development: It's about Feedback and Change. IEEE Computer Society, Vol. 36, 6 (6), June, pp. 39-43. (2003)

XP Tracking Tool: Uma Ferramenta de Acompanhamento de Projetos Ágeis

Carla Adriana Barvinski[1], Maria Istela Cagnin[2], Nilson E. S. Lima Filho[1], Leandro Veronezi [1]

[1]Faculdade de Ciências Exatas e Tecnologia – Universidade Federal da Grande Dourados (UFGD)
Caixa Postal 533 – 79.804-970 – Dourados – MS – Brasil

[2]Faculdade de Computação – Universidade Federal do Mato Grosso do Sul (UFMS)
Caixa Postal 15.064 – 91.501-970 – Campo Grande – MS – Brasil

carlabarvinski@gmail.com, istela@facom.ufms.br,
nilson.lima.filho@hotmail.com, leandroveronezi@gmail.com

Abstract. *The methodology eXtreme Programming (XP) concentrates its efforts on coding and testing activities, with no emphasis on formal and bureaucratic. In XP, the game planning plays an important role enabling the implementation of the project within the principles adopted by the methodology. In project planning, the use of metrics for tracking, monitoring and control of the project was essential because it provided information for making decisions and evaluations. This paper presents a tool, called XP Tracking Tool, specific for implementation and monitoring of metrics in XP projects. By focusing specifically on metrics XP, it embodies the essential tasks of a team of extreme programming, encouraging and strengthening the principles and practices of the methodology, which provides time measurements, information, graphics and documentation about projects.*

1. Introdução

As métricas são importantes instrumentos de acompanhamento, monitoramento, planejamento e administração de um processo ou produto de software. Composta por medidas e implementada por meio de medições, elas instrumentalizam a obtenção de dados estatísticos tanto sobre o produto quanto sobre o processo de software, subsidiando e embasando ações gerenciais [PRESSMAN, 2006; BOURGAULT, *et al.*, 2002].

No contexto ágil, em que um processo de desenvolvimento de software sofre radicais adaptações com fortes supressões nas atividades de documentação e produção de artefatos com o objetivo de liberar *releases* o quanto antes, as métricas têm um papel essencial pois, enquanto as práticas e princípios ágeis estruturam as atividades de desenvolvimento, cabem às métricas viabilizar o monitoramento dos resultados produzidos, embasando as orientações e decisões da gerência de projetos ágeis.

Há várias ferramentas CASE *(Computer Aided Software Engineering)*, como por exemplo, *XPlanner[1] VersionOne[2]*, *Pivotal Tracker[3]*, voltadas para a gerência de projetos

1 URL: *http://www.xplanner.org*

2 URL: *http://www.versionone.com*

3 URL: *http://www.pivotaltracker.com*

ágeis, todavia o enfoque primordial de tais ferramentas está na gerência de projetos, levando em consideração apenas as métricas mais convencionais, como é o caso de estimativas de esforço, velocidade do time e horas ideais.

Em desenvolvimento ágil, uma métrica torna-se dispensável quando o propósito a que ela atendia foi superado [BECK, 2004], seja por adequação do time ao desenvolvimento das tarefas ou em decorrência do aprimoramento do processo [SOMMERVILLE, 2007]. Em consequência disso, a quantidade de métricas disponibilizada por uma ferramenta é um aspecto relevante para a gerência de projetos ágeis.

Este artigo apresenta uma ferramenta, denominada XP Tracking Tool, específica para aplicação e monitoramento de métricas em projetos XP. Esta ferramenta é de fácil utilização, sendo ideal para atividades práticas do ambiente acadêmico nas disciplinas de Engenharia de Software e Gerência de Projetos e para o acompanhamento de pequenos projetos XP. Ela fornece recursos essenciais para o gerenciamento do projeto, tais como cadastro e pontuação de estórias, definição de iterações, atividades, tarefas, e registro de pareamento. A partir dos dados do projeto são extraídas automaticamente medidas e geradas as métricas de cálculo do fator de erro nas estimativas [SATO, 2007], de gráfico de Burn Down, de prazo e de cálculo do tempo ideal estimado, adaptada de Sato (2007). A ferramenta permite o acompanhamento de vários projetos simultaneamente e a definição de vários níveis de usuários; fornece documentação HTML sobre o projeto, bem como relatórios e gráficos de acompanhamento do projeto; possibilitando também que o cliente registre suas estórias via Web.

Na Seção 2 deste artigo aborda-se a metodologia ágil *eXtreme Programming* (XP), na Seção 3 discorre-se sobre métricas ágeis, na Seção 4 faz-se breve descrição de ferramentas para gerência de projetos ágeis, na Seção 5 apresenta-se a Ferramenta *XP Tracking Tool* e na Seção 6 discutem-se as conclusões e indicam-se sugestões de trabalhos futuros.

2. e*Xtreme Programming*

eXtreme Programming é uma metodologia de desenvolvimento ágil adequada para equipes com poucos desenvolvedores cujos projetos sejam de pequeno a médio porte, em que os requisitos são vagos e se alteram com frequência [BECK, 2004].

XP concentra seus esforços nas atividades de codificação e testes, sem ênfase em processos formais e burocráticos. Além disso, utiliza o paradigma de desenvolvimento orientado a objetos com o qual busca elaborar soluções simples, desenvolvendo software de forma iterativa e incremental, entregando versões funcionais de software constantemente.

Esta metodologia também se caracteriza pelo modo peculiar de participação do cliente em um projeto de software. Isto é, em XP as responsabilidades do projeto são divididas entre o cliente e a equipe, havendo uma separação do papel de cada um, o que evita conflitos e assegura que cabe ao cliente tomar as decisões de negócio e que compete a equipe de desenvolvimento responder pelas decisões técnicas [BECK, 2004].

Toda abordagem de desenvolvimento do XP é baseada em doze práticas que se fundamentam em quatro valores básicos que são: a comunicação entre os envolvidos no projeto, o *feedback* rápido entre cliente e equipe de desenvolvimento, a simplicidade na implementação e a coragem para adotar ações inovadoras e vencer os desafios do desenvolvimento de software. O conjunto formado por valores e práticas XP atua de

forma harmônica e coesa, assegurando ao cliente um alto retorno do investimento [TELES, 2004].

O papel das doze práticas é orientar como serão executadas as atividades básicas de desenvolvimento, de forma a manter a conformidade com os quatro valores. As práticas estão divididas em três grupos abrangendo: práticas organizacionais que são relacionadas ao planejamento; práticas voltadas à equipe; e práticas individuais de codificação e testes [BECK, 2004]. As práticas de XP são: jogo do planejamento, entregas frequentes, metáfora, projeto simples, testes, refatoração, programação em pares, propriedade coletiva, integração contínua, semana de 40 horas, cliente presente e padrões de codificação.

Destas práticas destaca-se o jogo do planejamento, cujo objetivo é planejar globalmente o projeto, elaborando um esboço da ideia principal e detalhando as funcionalidades com mais importância. A meta do jogo do planejamento é assegurar que a equipe esteja implementando funcionalidades que realmente proporcionem valor ao negócio do cliente. O jogo do planejamento é elemento-chave em XP. Durante a execução do projeto, o planejamento é refinado várias vezes, em períodos cada vez mais curtos, motivados por necessidade de adaptação a mudanças, alterações nos requisitos ou aprimoramento de processos [BECK, 2004]. As métricas fornecem estimativas cujos indicadores são considerados no refinamento do planejamento. Tais métricas (Seção 3) são de interesse deste trabalho e algumas delas são cobertas pela ferramenta proposta (Seção 5).

Tanto os princípios quanto as práticas são rigidamente exercitados pela metodologia XP, o que torna triviais atividades como a revisão permanente do código, o desenvolvimento de soluções o mais simples possíveis, o teste e integração em vários momentos em um mesmo dia e a execução de iterações curtas como forma de minimizar os erros [BECK, 2004].

Equipes de desenvolvedores XP tendem a ser pequenas, com um mínimo de três até quinze pessoas, desempenhando os papéis de treinador, gerente, rastreador, cliente e programador. É papel do rastreador acompanhar um projeto XP, o que envolve coletar e manter dados sobre o projeto, alimentar as métricas por ele definidas, comparar os dados colhidos com os estimados e divulgar a toda equipe as informações sobre o andamento geral do projeto [BECK, 2004]. Assim, é importante o apoio de ferramentas computacionais nestas atividades não somente para fornecer dados a partir de métricas sobre o andamento de projetos ágeis em tempo real, mas também disponibilizar histórico contendo registros importantes relativos à equipe de projeto, à velocidade de desenvolvimento, pareamento, entre outros.

3. Métricas Ágeis

As métricas constituem importante instrumento no acompanhamento, monitoramento e controle de projetos de software [BOURGAULT *et al.*, 2002]. Elas estão relacionadas à medição de indicadores qualitativos e quantitativos do tamanho e complexidade de um produto ou processo [PRESSMAN, 2006]. Os indicadores obtidos referem-se a experiências passadas e permitem aplicar os desempenhos observados tanto na projeção de previsões de desempenho futuro quanto na reformulação do processo de desenvolvimento ou na revisão do comportamento de times de desenvolvimento.

Dessa forma, métricas podem ser usadas para acompanhar o progresso do

processo de desenvolvimento de um projeto e/ou para obtenção de indicadores da qualidade do produto, de forma a permitir que gerentes e profissionais técnicos tenham a devida compreensão sobre o processo de engenharia de software e sobre o produto. Para Pressman (2006), a medição do processo de software e do produto por ele produzido é a única maneira real de determinar se está havendo melhorias ou não no processo de desenvolvimento e na qualidade final do produto.

As etapas, fases, atividades e tarefas de um processo de desenvolvimento de software adaptado às metodologias ágeis estão intrinsecamente vinculados aos princípios preconizados pelo Manifesto Ágil que estabelece [BECK *et al.*, 2001]: **indivíduos e interações** ao invés de processos e ferramentas; **software funcionando** antes que documentação completa; **colaboração com o cliente** mais que negociação com contratos; **adaptação às mudanças** é preferível a seguir um plano.

Estes princípios têm profundos impactos no comportamento de uma equipe de desenvolvimento de software, seja na sua organização interna, na interação com o cliente, no desenvolvimento das atividades, modificando permanentemente a sua forma de atuação. Por isso, as metodologias ágeis requerem uma re-contextualização das métricas a serem rastreadas [SCHWABER, 1995], havendo um consenso geral de que essas devem focar nos objetivos de negócios [BARNETT, 2009; GRIFFITHS, 2007; BECK, 2004], rompendo total ou parcialmente com algumas medições tradicionais classificadas como métricas orientadas a tamanho, orientadas à função, de produtividade cuja aplicação tendem a influenciar o comportamento das equipes [SATO, 2007; HARTMAN e DYMOND, 2006; DEMMERe BENEFIELD, 2007].

Vários autores citam a inconveniência de se medir projetos ágeis pelas abordagens tradicionais [BECK, 2004; GRIFFITHS, 2007; HARTMANN e DYMOND, 2006; BARNETT, 2009], pois na opinião deles as métricas convencionais tendem a desencadear comportamentos inadequados em resposta aos estímulos que elas conferem. Nesse sentido, para auxiliar na seleção adequada de métricas ágeis, há na literatura orientações [HARTMAN e DYMOND, 2006; SATO, 2007] que apontam para fatores importantes que um *tracker* deve considerar quando estiver escolhendo uma métrica para sua equipe.

Na realidade, as métricas são utilizadas pelas metodologias ágeis como um sensor, com o objetivo de acompanhar, mensurar, planejar, detectar anomalias, estimular correções e liberar valor de negócio [BECK, 2004; HARTMANN e DYMOND, 2006, BARNETT, 2007] durante um "ciclo constante de inspeção, adaptação e melhoria" [SATO, 2007]. Por isso, em projetos ágeis, as medições são realizadas continuamente de modo a fornecer indicadores do andamento do projeto em tempo real ao invés de *a posteriori*. Dessa forma, é possível propiciar ao líder de equipe a detecção imediata de dificuldades, possibilitando que o time de desenvolvimento faça a correção de problemas potenciais na medida em que estes surjam, além de fornecer dados concretos do andamento do projeto: cronograma, velocidade, qualidade, satisfação do cliente.

No que se refere a quantidade de métricas a serem rastreadas em um projeto de desenvolvimento ágil de software, observa-se divergência entre os autores. Beck (2004) sugere uma quantidade máxima de quatro métricas por projeto em um dado instante, enquanto que Barnett (2007) detectou em pesquisa de campo, a preferência pelo rastreamento de métricas que possibilitem o acompanhamento de um projeto em cinco grandes dimensões: risco, qualidade, cronograma, esforço/orçamento e conformidade. Independente deste aspecto, ambos recomendam o uso de poucas métricas simultâneas

por projeto e as descartam assim que se tornam obsoletas. Isso deve ser feito quando se detecta que os índices apresentados estão próximos ao ideal, o que demonstra a estabilização das atividades e indica que a métrica perdeu a finalidade, devendo portanto ser descartada, com exceção de métricas voltadas a testes de unidade, que devem permanecer durante toda duração do projeto [BECK, 2004].

Dadas essas considerações, pode-se deduzir que durante a execução de um projeto de software, incluindo os ágeis, pode-se fazer necessário o uso de variadas métricas. Definir quais métricas aplicar a determinado projeto é uma tarefa desafiadora, que requer experiência e tato. Segundo Griffiths (2007), se faltar clareza na utilização das métricas, as medições coletadas não serão importantes para o bom andamento do projeto e ocasionarão a perda de foco das medidas.

Neste contexto, de acordo com Sato (2007) é recomendável o estudo e a aplicação de abordagens, como *Goal Question Metric* (GQM) [BASILI *et al.*, 1996] e *Lean* [POPPENDIECK, 2003], que disponibilizam orientações seguras para a seleção das métricas mais adequadas a determinado objetivo/projeto. Sato (2007) sugere ainda o aproveitamento dos resultados de Reuniões de Retrospectivas [COCKBURN, 2001] como fonte de seleção de métricas.

Na metodologia *XP*, a unidade central do processo é representada pelo cartão de estórias (*story card)*, que são representações dos requisitos do sistema e também o ponto de partida para medidas de gerenciamento de projetos e qualidade de negócio. Sato (2007) sugere várias medições, as quais apresentam objetivos pontuais que podem ser utilizados na composição de uma métrica com objetivo mais amplo de acompanhamento e/ou controle do processo de desenvolvimento ou do produto de *software.* Sato (2007) também indica um conjunto de métricas consideradas as mais adequadas para acompanhar projetos XP.

4. Ferramentas para Acompanhamento de Projetos Ágeis

Pressman (2006) classifica ferramentas de gerência de projetos em genéricas e específicas. As primeiras atendem um conjunto amplo de atividades englobando listas de verificação, guias de planejamento, planilhas inteligentes, entre outros. Enquanto que ferramentas específicas podem focalizar diferentes áreas, tais como: estimativas de esforço e custo; elaboração de cronogramas; gestão de riscos e métricas de projeto e processo. Segundo Pressman (2006), uma ferramenta de métricas de projeto e de processo deve dar suporte "a definição, coleta, avaliação e relato de medidas e métricas de software", devendo disponibilizar mecanismos de coleta e de avaliação que conduzam ao cálculo de métricas de software.

No âmbito de gerência de projetos ágeis, há várias ferramentas CASE disponíveis, como por exemplo, *XPlanner*, *VersionOne*, *Pivotal Tracker*, *FireScrum*[4].

A VersionOne é uma solução comercial de código fechado, com interface Web. Proporciona um planejamento inicial simplificado, onde são definidas as iterações do projeto. Os prazos e entregas são definidos dentro de cada iteração. Tal ferramenta permite visualizar o *status* das iterações, suas entregas e gráficos *Burn down*[5].

XPlanner é uma ferramenta estritamente voltada ao gerenciamento de projetos ágeis XP, de código livre e gratuita, suporta multi-projetos em ambiente colaborativo e

4 www.firescrum.com.

5 Representação gráfica das atividades realizadas *versus* tempo estimado.

distribuído. Disponibiliza interface Web, dá o suporte para os artefatos e atividades típicas de projetos XP, iterações, estórias de usuário e tarefas, disponibiliza medidas e estimativas da iteração.

A *Pivotal Tracker* é uma solução Web, gratuita, colaborativa, em que os cartões de estória são a base do planejamento de projeto. As entregas são priorizadas em *backlogs*. A ferramenta agrupa as estórias automaticamente em iterações e faz estimativas a partir dos pontos cadastrados pelo usuário, os quais fornecem a base para geração dos gráficos de *Burn-down (*progresso do *release)*, *Burn-up (*progresso da iteração*)*, velocidade e *story type breakdown* (classificação das estórias por tipo e pontos na iteração). Os dados cadastrados pelo usuário são armazenados no *Amazon Simple Storage Service*, bem como documentação adicional que se julgue interessante anexar ao projeto.

A *FireScrum* é uma solução de código livre e gratuito, que tem seu foco no gerenciamento de projetos elaborados segundo a metodologia Scrum. Ela é constituída de um conjunto de aplicações integradas que dão suporte ao desenvolvimento de projetos desenvolvidos por uma equipe geograficamente distribuída. Devido a sua vinculação com a metodologia SCRUM, a *FireScrum* não foi considerada para análise durante a fase pré-concepção da XP *Tracking tool*.

5. Ferramenta *XP Tracking Tool*

Nesta seção são apresentadas as principais características da ferramenta XP Tracking Tool: visão geral, medidas e métricas implementadas, principais funcionalidades, os módulos da ferramenta, aspectos da ferramenta que apoiam e fortalecem a prática da metodologia ágil XP e a avaliação da ferramenta proposta.

A ferramenta *XP Tracking Tool* objetiva apoiar a coleta de métricas de gerenciamento de processo aplicadas a metodologia de desenvolvimento ágil XP, a fim de auxiliar no acompanhamento de projetos ágeis que seguem tal metodologia, mais especificamente, no melhoramento do planejamento da equipe, identificando elementos para aperfeiçoamento de suas práticas XP.

Para isso, foram implementadas na versão inicial da XP Tracking Tool as métricas de "Tempo Ideal Estimado" proposta por Sato (2007), "Gráfico de Burn Down" descrita em Schwaber (2004), e duas métricas adaptadas do trabalho de Sato (2007), intituladas "Métrica de cálculo do Tempo Ideal Estimado" e métrica de prazo. A escolha das métricas seguiu as orientações apresentadas por Beck (2004), que sugere que elas sejam de fácil coleta, que respondam a uma questão específica e reforcem os princípios ágeis. Considerou-se também que elas auxiliariam na visualização do andamento do projeto e oportunizariam a equipe de desenvolvimento rever suas estimativas, bem como possibilitariam a equipe certificar-se quanto a utilização correta das práticas propostas pela metodologia.

Salienta-se que a coleta das métricas presentes na ferramenta é feita automaticamente a partir de sua base de dados. As métricas implementadas pela *XP Tracking Tool* são apresentadas na Tabela 1.

Na *XP Tracking Tool*, o controle de acesso à ferramenta é feito por meio de autenticação de nome de usuário, senha e da aplicação de regras específicas para cada tipo de usuário, o que significa que grupos de usuários têm acesso a diferentes funcionalidades: o administrador tem acesso irrestrito; o programador acessa apenas os projetos para qual foi designado podendo, neste contexto, manter estórias e tarefas; o

cliente acessa apenas os projetos para o qual foi designado, podendo inserir e modificar estórias e testes de aceitação.

Tabela 1 – Métricas implementadas pela ferramenta *XP Tracking Tool*

Métricas	Descrição
Métrica de cálculo do fator de erro nas estimativas [SATO, 2007]	O valor desta métrica é um número positivo. Ele resulta da divisão do **tempo estimado** pelo **tempo real gasto** para conclusão das tarefas, quanto mais próximo de 1 for o valor, mais em conformidade com o planejado está o projeto.
Métrica de Gráfico de *Burn Down* [SCHWABER, 2004]	Esta métrica é formada por dados apresentados em gráficos que mostram uma linha decrescente do andamento do projeto e das *releases.*
Métrica de prazo (adaptado de Sato(2007))	Esta métrica apresenta o resultado da divisão do **número de dias decorridos** pelo número de **dias totais estimados** multiplicado por 100, que fornece a porcentagem do tempo estimado já decorrido.
Métrica de cálculo do Tempo Ideal Estimado (adaptado de Sato(2007))	Esta métrica estima o tempo necessário para a conclusão do projeto, baseado no **fator de erro** das estimativas da equipe, seu valor é obtido através da multiplicação do tempo estimado para a conclusão pelo **fator de erro**, é importante para a equipe ver quão distante está de suas metas.

Todo usuário ao acessar o sistema pode visualizar o andamento dos projetos aos quais tem acesso, além de poder imprimir diferentes relatórios em formato HTML (*HyperText Markup Language),* organizados por *release* ou por integração. Esses relatórios abrangem informações acerca do projeto, quais sejam: nome e dados específicos do projeto, como datas de início e fim, equipe; conteúdo de todos os cartões de estória e seus respectivos cartões de tarefa e testes de aceitação; entre outros. Também são disponibilizados dois gráficos de *Burn Down*, um que representa dados sobre o andamento da *release* e outro que ilustra a evolução do projeto como um todo.

As medidas (Seção 5.1) extraem dados sobre projetos que permitem a geração automática de estimativas, métricas e gráficos. Os gráficos gerados e as estórias podem ser exportados em arquivos nos formatos PDF (*Portable Document Format*), *png* (*Portable Network Graphics*), *jpg* (*Joint Photographic Experts Group*) ou impressos para documentar o projeto.

5.1. Medidas coletadas

Para a implementação das métricas, a ferramenta requer a coleta de dezessete diferentes medidas sobre o projeto. Algumas delas são utilizadas para o cálculo das métricas suportadas pela ferramenta e as demais apoiam na documentação do projeto. Na Tabela 2 são apresentadas as medidas consideradas na implementação da ferramenta proposta. Buscou-se implementar uma quantidade significativa de medidas visando a expansão do número de métricas em versão futura do software.

Tabela 2 – Medidas coletas pela ferramenta *XP Tracking Tool*

Medidas	Descrição
Número de Clientes	Representa a quantidade de usuários do tipo cliente, designados para o projeto
Número de Estórias	Estabelece a quantidade de estórias de um projeto
Número de Estórias canceladas	Registra a quantidade de estórias que estão com seu *status* definido como canceladas
Número de Estórias em andamento	Contém a quantidade de estórias que estão com seu *status* definido como em Andamento

Número de Estórias esperando	É a quantidade de estórias que estão com seu status definido como esperando
Número de Estórias estimadas	É a quantidade de estórias que estão com seu *status* definido como estimadas
Número de Estórias prontas	É a quantidade de estórias que estão com seu *status* definido como pronta
Número de Integrações	É a quantidade de integrações feitas
Número de Integrantes	É a quantidade de pessoas cadastradas na ferramenta e que estão envolvidas no projeto
Número de Programadores	É a quantidade de usuários do tipo programador, *design*ados para o projeto
Número de *Release*s	É a quantidade de *releases* que foram concluídos
Número de Testes de Aceitação	É a quantidade de testes de aceitação escritos pelo cliente
Número de Testes de Aceitação Passando	É a quantidade de testes de aceitação que tiveram seu *status* alterado pelo cliente para concluído
Tempo Decorrido	É a quantidade de dias sem os fins de semana que decorreram desde o início do projeto
Tempo estimado para a conclusão	É a quantidade de dias estimados pelos programadores para concluir todas as tarefas ainda não realizadas
Tempo Total	Representa o tempo total em dias estimado para executar todas as tarefas do sistema, admitindo a prática de 40 horas semanais e 8 horas diárias

5.2. Módulos da XP *Tracking Tool*

A XP *Tracking Tool* foi desenvolvida em Java e utiliza o SGBD (Sistema de Gerenciamento de Banco de Dados) MySQL. É composta por dois módulos, um denominado módulo principal e o outro de configuração de base de dados. Sua arquitetura está ilustrada na Figura 1.

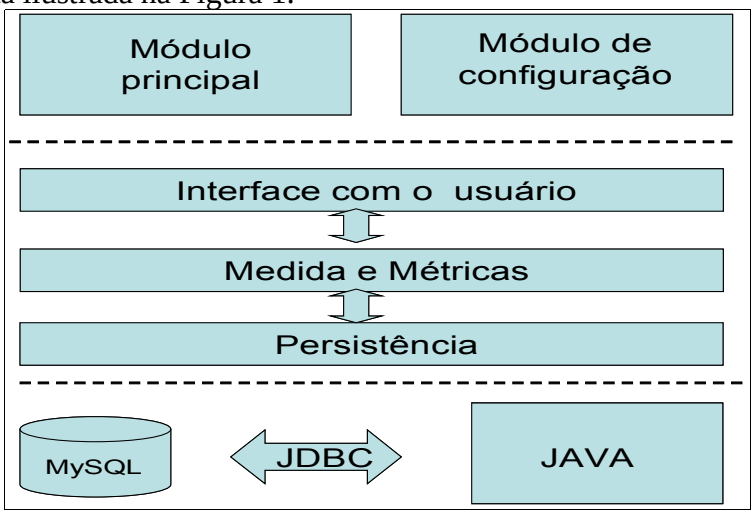


Figura 1 – Arquitetura da Ferramenta XP *Tracking tool*.

No módulo principal estão presentes as funcionalidades essenciais da ferramenta, a seguir relacionadas:

- Permite ao administrador cadastrar e acompanhar vários projetos simultaneamente;
- Permite ao cliente efetuar o cadastro dos cartões de estórias e testes de aceitação, necessários para a validação dos requisitos da estória;

- Permite ao administrador ou programador definir a prioridade de cada cartão de estória – alta, média ou baixa;
- Possibilita a todos os usuários acompanhar/monitorar o andamento *status* de implementação das estórias de um projeto a que tenham acesso. O *status* de uma estória pode ser: em espera, em andamento, concluído ou cancelado;
- Permite ao administrador definição de integrações e *releases;* bem como a seleção de estórias para cada *release;*
- Possibilita ao programador efetuar a marcação de tempo para cada estória – consiste em informar o tempo estimado para realização de cada tarefa necessária a implementação de uma estória e o tempo real gasto na sua execução;
- Permite ao administrador o acompanhar as combinações de pares (programação pareada), formados durante a execução do projeto;
- Centraliza e organiza as informações gerenciais e de processo relacionadas a um projeto XP, como por exemplo, estórias, *release*, membros do time de desenvolvimento, tarefas;
- Gera automaticamente, a partir da inserção de medidas sobre o projeto (Seção 5.1), as métricas descritas na Tabela 1 disponibilizando-as a todos os usuários da ferramenta;
- Gera gráficos de *Burndown* relativo a *releases* do projeto e a iterações de uma *release* específica, possibilitando sua impressão ou exportação, permitindo a visualização das medidas inseridas, métricas e gráficos gerados pela ferramenta. Todos os usuários podem visualizar os gráficos.
- Disponibiliza a todos os usuários, em formato HTML, documentação de todos os dados cadastrados e gerados sobre o projeto.
- Permite ao administrador o cadastro de diferentes tipos de usuário: cliente, programador e administrador. Cada usuário possui permissões de acesso distintas na ferramenta.

O módulo principal é organizado em projetos que contém estórias. As estórias são apresentadas em uma interface que recorda um painel de indicadores e são o ponto de partida para definir as atividades de desenvolvimento: atribuição de tarefas, pareamento, definição de release e testes de integração. O usuário deve criar ou selecionar uma estória para iniciar sua interação com a ferramenta. Algumas métricas implementadas e todas as medidas coletadas são apresentadas em um painel específico na tela de gerência de projeto, conforme ilustrado na Figura 2. Visualiza-se, na parte destacada, as métricas: a) de prazo, nomeada como tempo total, b) de cálculo do fator de erro nas estimativas, identificada no painel como fator de erro nas estimativas, c)de cálculo do Tempo Ideal Estimado, intitulada tempo ideal para conclusão.

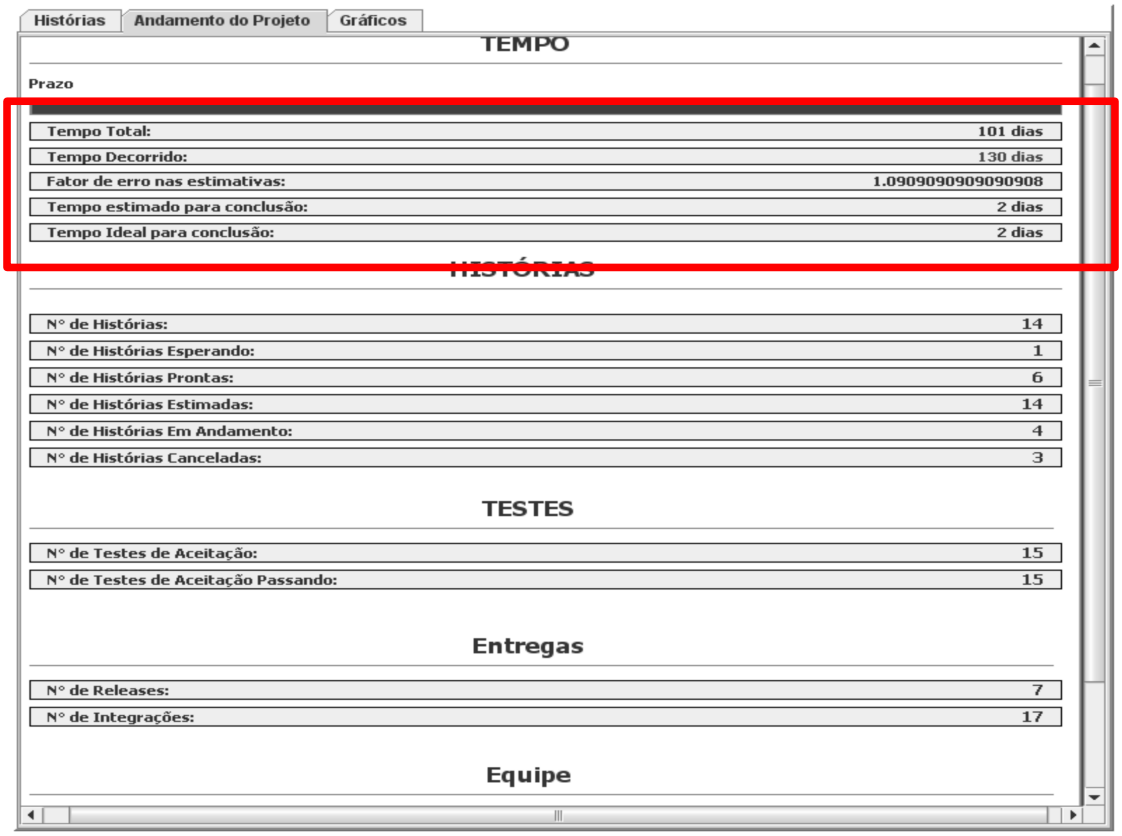


Figura 2 – Painel de visualização das medidas e métricas

A quantidade de dias por integração e integrações por release é apresentada na forma de gráfico, de acordo com o exibido na Figura 3, e as estórias são agrupadas por status e visualizadas em um painel específico na tela de gerência de projeto, conforme mostrado na Figura 4.

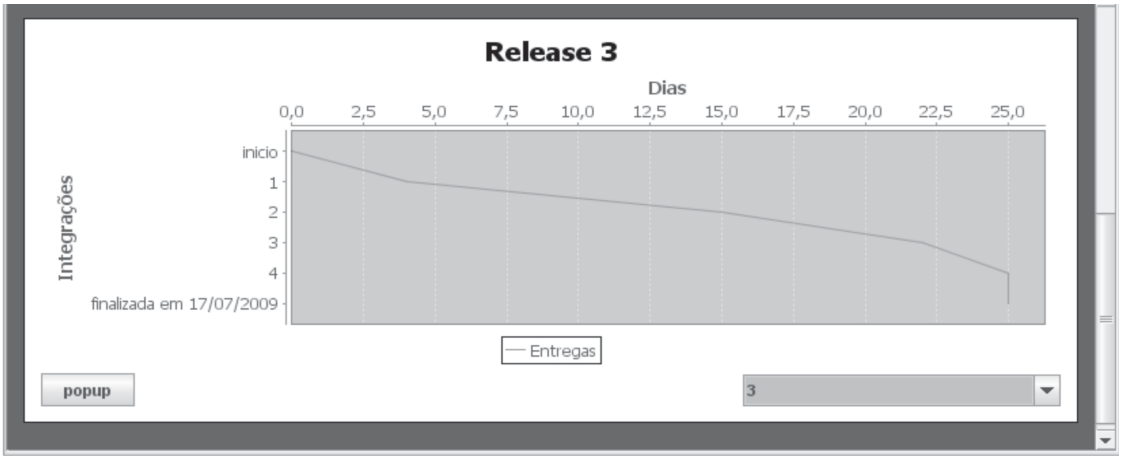


Figura 3 – Painel de visualização dos gráficos de Burn Down

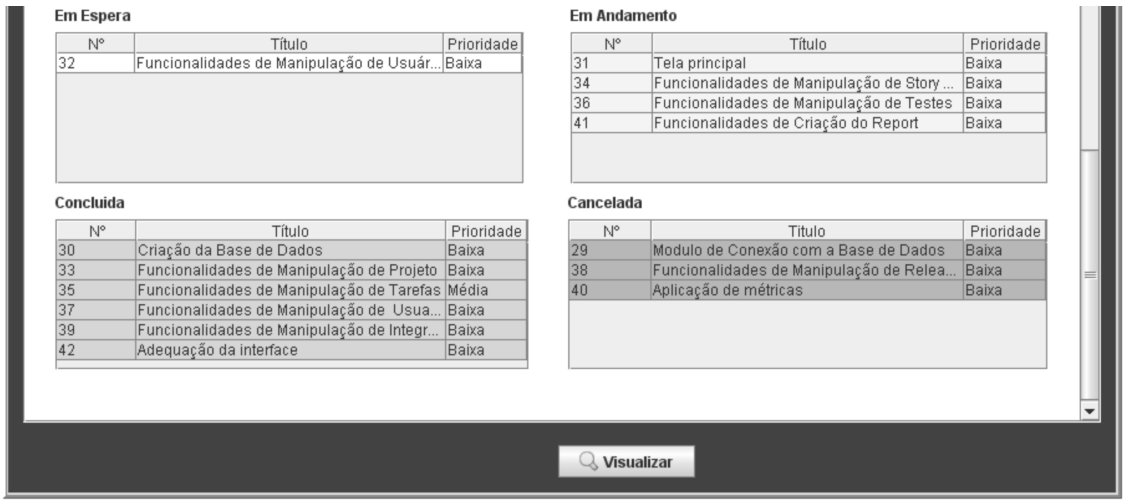


Figura 4 – Painel de visualização de estórias cadastradas

O módulo de configuração de base de dados é utilizado para a configuração da conexão com a base de dados. A *XP Tracking Tool* permite diversas formas de configuração da conexão com a base de dados: acesso local, em rede (em um servidor) ou *on-line* (em um servidor acessado via Web). Para isso, é necessário indicar o local da base de dados da ferramenta, como sendo um endereço na rede local, ou um servidor *Web* ou, ainda, o próprio computador; informar o nome de usuário e a senha da base de dados; bem como criar a base de dados a partir de um *script* próprio, caso ela não exista.

5.3. Práticas e Artefatos de XP presentes na XP *Tracking Tool*

Algumas características básicas da metodologia *XP*, materializados na forma de práticas ou artefatos, estão incorporadas na ferramenta *XP Tracking Tool* de forma explícita ou implícita, conforme sumarizado na Tabela 3.

Tabela 3 - Práticas e Artefatos XP na ferramenta XP Tracking Tool

Práticas e Artefatos XP (descrição)	**ARTEFATO**	**PRÁTICA**	**EXPLÍCITO**	**IMPLÍCITO**
Cartões de História	X		X	
Cartões de Tarefa	X		X	
Testes de Aceitação	X		X	
Programação Pareada		X	X	
Semana de 40 Horas		X		X
Releases		X	X	
Interações		X	X	

São abordados de forma explícita na ferramenta os artefatos específicos de projetos *XP*, tais como, cartões de estórias e seus respectivos cartões de tarefas e testes de aceitação, uma vez que requerem cadastramento. As práticas programação pareada e integração contínua, também recebem o mesmo tratamento - na ferramenta é possível indicar se houve ou não pareamento na realização de uma tarefa. As integrações e *releases*, elementos fundamentais para a integração contínua, devem ser cadastradas.
A semana de 40 horas se faz presente de forma implícita, aparecendo nos cálculos das métricas, uma vez que é considerada uma semana de 40 horas de trabalho.

5.4. Avaliação

Nessa seção são descritos os resultados obtidos com o uso da XP *Tracking Tool* bem

como os pressupostos em que estes se apoiam, uma vez que não foram conduzidos estudos de casos em ambiente real até o momento da escrita deste artigo.

Em comparativo com as ferramentas *VersionOne, Xplanner, Pivotal Tracker* (Seção 4) verifica-se que a XP *Tracking tool* detém as mesmas propriedades destas soluções, reservadas as proporções, porém com o adicional de incluir recurso para o acompanhamento do pareamento e possibilitar a configuração e acesso de diferentes níveis de usuários ao sistema. Cada tipo de usuário pode acessar a XP *Tracking Tool* para alimentar ou visualizar dados de acordo com suas atividades no processo de desenvolvimento. Os tipos de usuários são pré-definidos e englobam gerente, *tracker*, programador e cliente. Na Tabela 4 são assinalados os aspectos em comum existentes nas ferramentas *Pivotal Tracker, XPlanner, VersionOne* e na XP *Tracking Tool*.

Tabela 4 – Comparativo de f*erramentas CASE para gerência de projetos XP*

Característica/Ferramenta	A	B	C	D	E	F	G	H	I
Pivotal Tracker	X	-	X	X	X	X	-	-	-
XPlanner	X	X	X	X	X	X	-	-	X
VersionOne	-	-	X	X	X	X	-		X
XP Tracking Tool	X	X	X	X	X	X	X	X	-

LEGENDA: **A.** Gratuito **B.** *Opensource* **C.** Gráficos **D.** Priorização	**E**. Marcação de horas **F.** Estimativas **G.** Programação pareada **H.** Níveis de acesso	**I.** Suporta diferentes metodologias

Os demais benefícios proporcionados pela XP *Tracking Tool* são o fortalecimento das práticas XP de pareamento, desenvolvimento iterativo, interações entre a equipe e semana de 40 horas.

No quesito pareamento, a ferramenta força o cadastramento do par que irá implementar determinada estória, assim possibilita ao gerente de projetos acompanhar a diversifição dos pares e interferir quando detectar distúrbios de preferência por pares ou ainda orientar quando perceber dificuldades de adoção da prática pela equipe. A relação pares, estórias, tarefas podem sem acompanhadas por relatórios. Em versões futuras planeja-se o aperfeiçoamento desse instrumento de acompanhamento da prática.

O fato da ferramenta requerer a associação de cada estória a uma iteração e o monitoramento de resultados ocorrer sobre o *release* correlacionado, condiciona a prática do desenvolvimento iterativo para que a ferramenta apresente resultados efetivos.

A visualização dos gráficos de *Burn Down* referente a um projeto é permitida a todos os membros que formam a equipe alocada ao projeto. Deste modo, a ferramenta disponibiliza um recurso extra para a divulgação de informações e dados relativos ao projeto facilitando a comunicação e interação da equipe e fortalecendo a prática correspondente.

Cada métrica contribui para o fortalecimento de práticas ágeis enquanto proporciona informações significativas sobre o projeto. A métrica “Cálculo de Fator de Erro” permite a equipe visualizar a variação entre suas estimativas e os prazos reais de desenvolvimento, ou seja, ela favorece uma análise por parte da equipe sobre sua

capacidade em estimar com precisão a velocidade de desenvolvimento, estimulando-a ao aprimoramento das estimativas. A métrica "Gráficos de *Burn Down"* fornece uma medição visual do projeto, uma vez que os gráficos podem ser impressos e colocados em um mural ou anexados a documentação favorecendo a divulgação de informações sobre o projeto. A métrica "Prazo" auxilia a equipe visualizar como as estimativas feitas por ela se comportam diante do andamento do projeto, mostrando o prazo prometido pelo desenvolvedor ao cliente. A métrica "Cálculo do Tempo Ideal Estimado" indica o tempo ideal necessário para concluir o projeto baseando-se no fator de erro e com isso decidir pelo ajuste na velocidade de desenvolvimento ou na redução do escopo do projeto.

6. Conclusão

As métricas fornecem subsídios para a tomada de decisões e avaliações e podem ser aplicadas durante ou depois do término de um projeto. No contexto de desenvolvimento ágil, caracterizado pelo pouco rigor na aplicação de formalidades em processos de desenvolvimento de software, elas desempenham papel fundamental, pois seus indicadores asseguram a efetividade das práticas ágeis utilizadas ou não para apoiar o desenvolvimento.

A adoção de uma ferramenta para o cálculo de métricas é um importante recurso para a medição, acompanhamento e monitoramento de projetos, não somente devido à agilidade na obtenção de indicadores como também pelos fatores adicionais tais como documentação e o histórico de projetos desenvolvidos.

A ferramenta *XP Tracking Tool* proporciona recursos para o acompanhamento, monitoramento e controle das atividades fundamentais de um projeto XP quais sejam, cartões de estórias, *releases*, integrações contínuas e testes; fortalece as práticas e princípios preconizados pela metodologia, tais como, o pareamento e o desenvolvimento iterativo e incremental. Disponibiliza quatro métricas significativas para XP e dezesseis importantes medidas de projetos ágeis. Adicionalmente, permite que todos os membros da equipe de desenvolvimento acompanhem as métricas de andamento do projeto a qualquer momento, tornando-se mais um instrumento de divulgação de informações sobre a evolução do mesmo, aspecto que é fator essencial em XP.

Como trabalhos futuros podem ser elencados: efetuar uma avaliação mais acurada da ferramenta por meio de estudos de caso em ambiente real; ampliar o número de métricas XP implementadas pela ferramenta agregando métricas de qualidade, team morale e progresso do projeto; incorporar características de outras metodologias ágeis; adaptar a arquitetura da XP Tracking Tool de forma a permitir sua integração com outras ferramentas que favoreçam a coleta automática de dados.

7. Referências

BARNETT, L. (2007) *Agile Projects Must Measure Business Value. Agile Journal*. http://www.agilejournal.com/articles.

BARNETT, L. (2009) *An Agile Approach to Project Management Leveraging Agile Practices to Improve PMO Effectiveness.* *http://www.agilejournal.com/articles*, fevereiro, 2010.

BASILI, V. CALDIERA, G. and ROMBACH, D. (1996) *The goal question metric approach*, *Encyclopedia of Software Engineerin*, p.528-532

BECK, K. (2004) Programação Extrema (XP) explicada - acolha as mudanças. Porto

Alegre: Bookman.
BECK, K. BEEDLE, M. BENNEKUM, A. COCKBURN, A. CUNNINGHAM,W. FOWLER, M. GRENNING, J. HIGHSMITH, J. HUNT, A. JEFFRIES, R. KERN, J. MARICK, B. MARTIN, R. MELLOR, S. SCHWABER, K. SUTHERLAND, J. and THOMAS, D. (2001) *Agile Manifesto for Agile Software Development*, http://www.agilemanifesto.org/, Outubro, 2009.
BOURGAULT, M.(2002) LEFEBVRE, E. LEFEBVRE, L.A. PELLERIN, R. ELIA, E. *Discussion of Metrics for Distributed Project Management: Preliminary Findings Proceedings of the 35th Hawaii International Conference on System Sciences.*
COCKBURN, A. (2001) Crystal Clear: a Human Powered Methodology for Small Teams, Editora Addison Wesley Longman.
DEMMER, P *and* BENEFIELD, G. (2007) SCRUM *Primer: An Introduction to Agile Project Management with Scrum*. Version 1.04. *www.goodagile.com/scrumprimer/scrumprimer.pdf* .Acessado em fevereiro, 2010.
GRIFFITHS, M. (2007) *Most Software Development Metrics Are Misleading And counterproductive*, http://www.agilejournal.com/articles, fevereiro, 2010.
HARTMANN, D. *and* DYMOND, R. (2006) *Appropriate Agile Measurement: Using Metrics and Diagnostics to Deliver Business Value*.
AGILE '06: *Proceedings of the conference on AGILE* 2006, IEEE *Computer Society*.
HELDMAN, K. (2006) Gerência de projetos guia para o exame oficial do PMI. Editora Campus.
POPPENDIECK. M. (2003) *Lean Software Development: An Agile Toolkit*, New York: Addison-Wesley.
PRESSMAN, R. (2006) Engenharia de Software. 6 ed. São Paulo: MacGraw-Hill.
SATO, D. (2007) Uso Eficaz de Métricas no Desenvolvimento Ágil de Software, Universidade de São Paulo.
SCHWABER, K. (1995) SCRUM *Development Process. Proceedings of the 10th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications* (OOPSLA), p.117-134 .
SCHWABER, K. (2004) Agile Project Management with Scrum. Microsoft Press.
SOMMERVILLE, I. (2007) Engenharia de Software. 8 ed. São Paulo: Pearson Addison Wesley.
TELES, V. (2004) *Extreme Programming*: aprenda como encantar seus usuários desenvolvendo *software* com agilidade e alta qualidade, 1. ed. São Paulo: Novatec.

Sessão 3

Estudos conceituais com métodos ágeis

Desenvolvimento de jogos é (quase) ágil

Fábio Petrillo e Marcelo Pimenta[1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

fabio@petrillo.com, mpimenta@inf.ufrgs.br

Abstract. *Software developers recognize game development as a very complex and multidisciplinary activity. However, the success of games as one of most profitable areas in entertainment domain could not be incidentally. The goal of this paper is to investigate if (and how) principles and practices from Agile Methods have been adopted in game development, by means bibliographic review and mainly through postmortem analysis.*

Resumo. *Os desenvolvedores de software reconhecem que o desenvolvimento de jogos é uma atividade complexa e multidisciplinar. No entanto, o fato de ser uma das mais bem sucedidas do setor de entretenimento, leva a crer que tal pujança e lucratividade não são obras do acaso. O objetivo deste artigo é investigar se (e como) alguns princípios e práticas ágeis têm sido aplicados no desenvolvimento de jogos , a partir da análise da literatura sobre desenvolvimento de jogos e principalmente através da análise de postmortems.*

1. Introdução

A criação de jogos eletrônicos é uma atividade incrivelmente complexa [Gershenfeld et al. 2003] e uma tarefa mais dura do que se pode imaginar inicialmente. Hoje, criar um jogo é uma experiência muito diferente do que era no passado e, certamente, mais difícil, tendo explodido em complexidade nos últimos anos [Blow 2004].

O incremento da complexidade, aliado à natureza multidisciplinar do processo de desenvolvimento de jogos (arte, som, jogabilidade, sistemas de controle, inteligência artificial, fatores humanos, entre muitos outros) interagindo com o desenvolvimento tradicional de software, cria um cenário que aumenta ainda mais essa complexidade, a ponto de [Callele et al. 2005] recomendarem uma metodologia de engenharia de software especializada para o domínio de jogos digitais.

A indústria de jogos pode beneficiar-se tremendamente ao adquirir os conhecimentos da engenharia de software, permitindo que os desenvolvedores utilizem práticas sólidas e comprovadas. De fato, segundo [Flynt and Salem 2004], o claro entendimento das ferramentas disponíveis e como aplicá-las pode potencializar os resultados no desenvolvimento de jogos.

Na indústria tradicional de software, inúmeras obras tratam das boas práticas de engenharia de software [Pressman 2006, Sommerville 2001, Tsui and Karam 2007]. Contudo, será que essas práticas também são encontradas no desenvolvimento de jogos eletrônicos? Quais são as práticas mais destacadas? Com que frequência essas

práticas são encontradas nos projetos de jogos? Que práticas são encontradas em ambas as indústrias? Existiriam boas práticas encontradas somente no desenvolvimento de jogos? Nossa intenção neste trabalho é discutir estas questões.

A comunidade de desenvolvimento de jogos conta com uma vasta literatura especializada, principalmente no que tange a questões tecnológicas. Entretanto, poucas são as obras de engenharia de software dedicadas à indústria de jogos eletrônicos, destacando-se, principalmente, as obras de [Bethke 2003] e [Flynt and Salem 2004]. É interessante ressaltar que essas duas são eminentemente propagadoras da visão filosófica do processo em cascata de desenvolvimento (mais detalhes adiante na seção 2) , em especial a obra de [Bethke 2003], uma das mais citadas pela comunidade de jogos, demonstrando a cultura arraigada aos processos prescritivos e não-iterativos. Nela, [Bethke 2003] defende explicitamente a adoção de um subconjunto "confortável" do Processo Unificado (UP - Unified Software Development Process) como processo de desenvolvimento de jogos eletrônicos [Bethke 2003], pelo simples fato de ser um padrão da indústria de software.

O objetivo deste artigo é investigar se (e como) alguns princípios e práticas ágeis têm sido aplicados no desenvolvimento de jogos. Esta investigação pode ajudar a desmistificar a impressão de que a adoção de práticas ágeis pelos desenvolvedores de jogos é um processo difícil. Na verdade, se muitas das práticas ágeis já estiverem sendo (mesmo que parcialmente) adotadas, acreditamos que muitos desenvolvedores podem tentar compreender melhor os fundamentos do desenvolvimento ágil de software e mais facilmente encontrar meios de colocá-lo em ação no seu cotidiano.

O desenvolvimento ágil de software é um conjunto de metodologias para o desenvolvimento de software que promove adaptação, fortalecimento do trabalho em equipe, auto-organização, entregas rápidas de alta qualidade e adoção de boas práticas, alinhando o desenvolvimento com as necessidades das empresas. Acima de tudo, a proposta do desenvolvimento ágil é aumentar a capacidade de criar e responder às mudanças, reconhecendo que está nas pessoas o elemento primário para guiar um projeto ao sucesso [Highsmith and Cockburn 2001, Marchesi et al. 2002].

O artigo está estruturado da seguinte forma: após esta introdução, a seção 2 apresenta um resumo do processo de desenvolvimento de jogos. A seção 3 discorre sobre as boas práticas de engenharia de software encontradas no processo de desenvolvimento de jogos eletrônicos, a partir da análise da literatura sobre desenvolvimento de jogos e principalmente através da análise de postmortems. A seção 4 discute os resultados destas análises. Finalmente, algumas considerações finais são tecidas como conclusão.

2. O processo tradicional de desenvolvimento de jogos

Ao longo dos anos, muito tem sido escrito (ver p.ex. [Bethke 2003], [Flynt and Salem 2004], [Crawford 1984]) sobre o processo de desenvolvimento de jogos. Segundo [Flood 2003], é possível identificar um ciclo comum de desenvolvimento em várias equipes: o modelo em cascata [Pressman 2006] adaptado à produção de jogos.

Em especial, duas etapas deste modelo podem ser apresentadas em destaque: a) a definição das regras do jogo e b) a produção do **Documento do Jogo** ou **Documento de Projeto**.

As regras de um jogo determinam o comportamento e a interação entre os agentes

que habitam o seu ambiente e na prática são encaradas como requisitos. De fato, o projeto de um jogo nada mais é do que a criação de um conjunto de regras [Cook 2001], normalmente definidas em um processo informal, envolvendo todos os membros da equipe de desenvolvimento [Schaefer 2000].

O Documento de Projeto (*Design Document*) é a principal, muitas vezes a única, documentação de um jogo. Seu objetivo é descrever e detalhar a mecânica do jogo, isto é, o que o jogador é capaz de fazer dentro do ambiente do jogo, como ele é capaz de fazê-lo e como isso pode levar a uma experiência satisfatória. O documento também costuma incluir os principais componentes da história e descreve os cenários no qual o jogo é ambientado, dando suporte à descrição das ações do jogador. Muitos desenvolvedores se referem a ele como especificação funcional, utilizando-o como base para o projeto de arquitetura [Rouse 2001].

A criação de um Documento de Projeto sólido é vista, tradicionalmente, como o passo mais importante no desenvolvimento de um jogo. As dificuldades na criação desse documento são provocadas, principalmente, pela natureza da tarefa e pelas ferramentas usadas na sua concepção, não sendo possível, por exemplo, documentar a "diversão" [McShaffry 2003]. A equipe de desenvolvimento pode usar sua experiência e intuição na definição da mecânica do jogo, mas a qualidade do entretenimento, proporcionado pelo jogo, em geral só poderá ser avaliada nos estágios de teste.

A modelagem de jogos não evoluiu nos últimos anos e continua baseada em técnicas narrativas, como roteiros e *storyboards*, emprestadas de outros meios de entretenimento, como o cinema [Kreimeier 2002]. A idealização de um jogo também pode contar com um nível de abstração maior, organizado no Documento de Conceito ou Proposta (*Concept Document* ou *Proposal*). Esse documento pode analisar aspectos como mercado, orçamento, cronograma, tecnologias, estilo de arte, perfil do grupo de desenvolvimento e alguma descrição em alto nível da jogabilidade. No entanto, a elaboração o Documento de Conceito não é comum a todos os projetos [Rouse 2001].

O processo de desenvolvimento mais usado na produção de jogos é baseado no modelo em cascata [Rucker 2002]. Esse processo é composto de fases que são executadas seqüencialmente, na qual cada uma gera um produto e é independente das demais. As características inerentes à produção de jogos exigem algumas adaptações ao processo clássico, que podem ser visualizadas na figura 1.

Algumas atividades (como p.ex. especificação, concepção, construção, teste de qualidade) são encontradas no desenvolvimento de software tradicional e não descritas aqui mas algumas são específicas para o desenvolvimento de jogos, e por isto as descrevemos sucintamente:

- **Definição do Roteiro e Estilo de Arte:** o roteiro descreve o fluxo do jogo, isto é, como o jogador irá alcançar seu objetivo e como este será expresso ao longo dos vários cenários do jogo. A definição da arte especifica os estilos que serão usados, as ferramentas para criação e os modelos escolhidos para validar o estilo. Ainda que esses estágios sejam normalmente mostrados como atividades separadas, a forte conexão entre o roteiro e o tipo de arte a ser empregada no jogo permite a sua descrição como um item único.
- **Teste de Jogabilidade:** o produto do estágio de Teste de Qualidade é uma versão

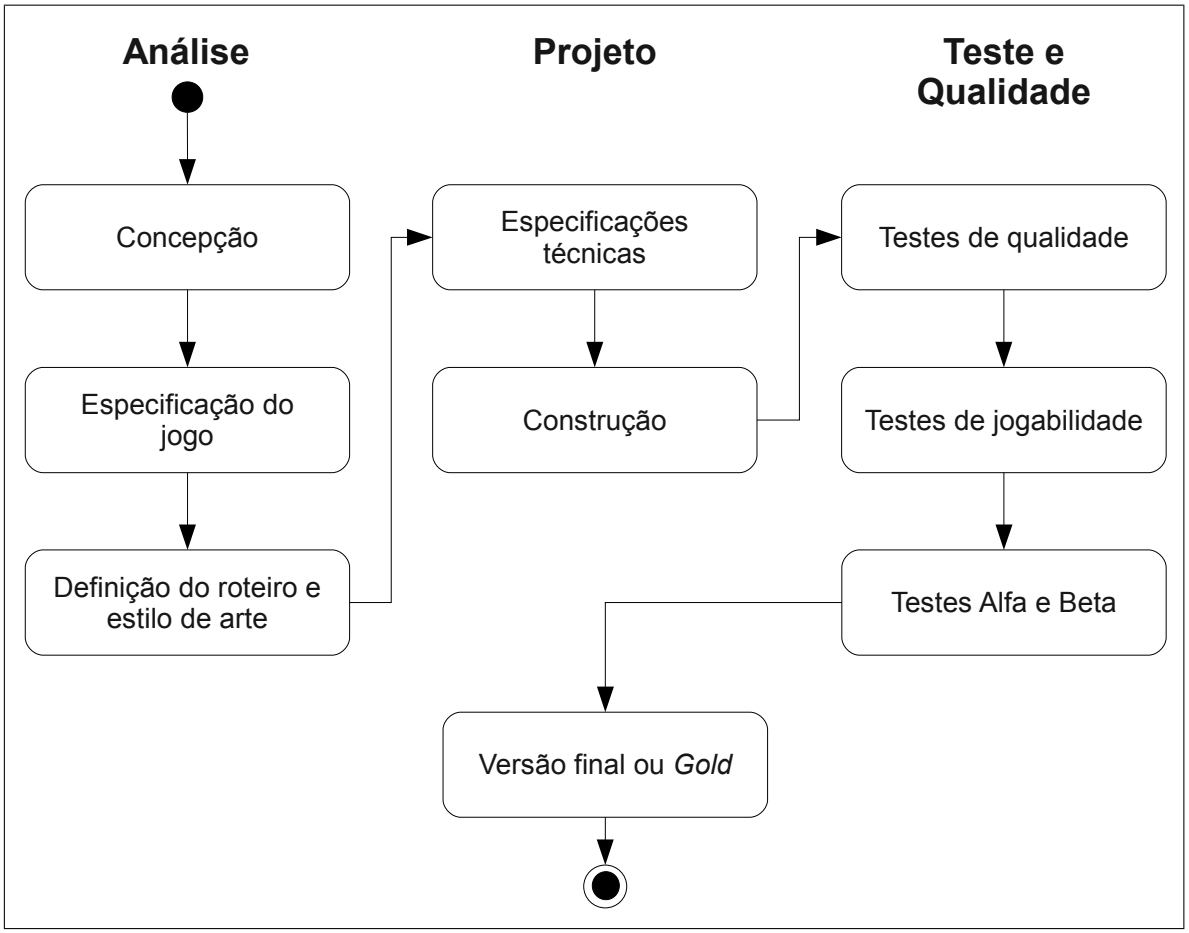


Figura 1. Processo em cascata adaptado ao desenvolvimento de jogos

pronta para testes de jogabilidade, na qual são demonstradas as características do jogo para grupos de usuários. O objetivo das seções de demonstração é validar ou criticar a mecânica do jogo, indicando as mudanças cosméticas ou mesmo estruturais do projeto. Nesse estágio, é possível a adoção de um ciclo com a atividade de Construção. Dessa forma, as sugestões e críticas dos usuários podem ser encaminhadas diretamente aos desenvolvedores.

- **Teste Alfa:** o produto do estágio de Teste de Jogabilidade é um jogo que atende (e possivelmente extrapola) as especificações do Documento de Projeto. Nesse momento, o jogo está pronto para testes com uma audiência maior, composta por usuários selecionados que têm algum conhecimento do projeto. Embora algumas das alterações sugeridas pelos usuários nesse processo possam ser substanciais, espera-se que nenhuma mudança significativa deva ser realizada no projeto.
- **Teste Beta:** nessa atividade o jogo está pronto para testes com uma audiência maior, que detém pouco ou nenhum conhecimento sobre o projeto. O objetivo desse estágio é medir a receptividade dos usuários ao jogo, além de detectar eventuais problemas que possam ocorrer.
- **Versão final ou *Gold*:** no momento em que as principais alterações sugeridas pelos usuários no estágio de Teste Beta foram incorporadas ao jogo, o produto está pronto para ser entregue ao público em geral.

Alguns autores argumentam que o modelo em cascata, apesar de servir como de-

nominador comum entre os ciclos existentes, **não pode ser aplicado integralmente**. As características inerentes a um jogo, como a dinâmica do *design* ou dificuldades no planejamento da jogabilidade, tornam difícil - ou mesmo impossível - especificar completamente um jogo sem escrever nenhum código, isto é, sem ter nenhuma versão do sistema no qual o projeto possa ser testado. Isso resulta em ciclos de desenvolvimento que envolvem laços entre os estágios de Especificação do Jogo e os processos de teste. [Rucker 2002] cita um ciclo de desenvolvimento incremental, o *Staged Delivery Model*, como uma alternativa ao modelo em cascata.

3. Boas práticas na indústria de jogos eletrônicos

Nesta seção descreveremos como foi realizado um levantamento de boas práticas referenciadas na literatura especializada para o desenvolvimento de jogos. Com as práticas mapeadas, analisaremos sua adoção no cotidiano do desenvolvimento de jogos, através da análise de *postmortems*.

A Análise de *Postmortems* (PMA) é um enfoque de Engenharia de Software Empírica. A Engenharia de Software Empirica visa investigar e coletar dados relevantes para, posteriormente, poder generalizar os resultados, aprendendo com erros e acertos detectados, além de possibilitar um futuro reuso deste conhecimento.

Experimentos controlados, os estudos de caso e surveys são enfoques mais frequentemente utilizados e mais conhecidos que a Análise de *Postmortems* (PMA) [Shull et al. 2008, Wohlin et al. 2003, Shull et al. 2004]. A principal razão para a PMA não ser adotada em projetos de software nos diferentes domínios de aplicação não é sua dificuldade de operacionalização, nem os recursos envolvidos ou muito menos seu custo, mas sim a ausência de *postmortems*.

Tradicionalmente, as equipes de TI não tem hábito de criar *postmortems*, embora existam obviamente exceções (ver por exemplo [Stalhane et al. 2003]) e fortes recomendações para fazê-lo [Birk et al. 2002]. Felizmente uma rara exceção é o domínio dos jogos.

No desenvolvimento de jogos, a elaboração de postmortems é um hábito difundido. Esse hábito propicia a análise das práticas mais bem sucedidas, assim como a coleta dos principais problemas enfrentados, através da avaliação de depoimentos espontâneos e que contém os aspectos mais relevantes apontados por quem realmente participou do processo, minimizando os problemas relatos por [Shull et al. 2008] na utilização de entrevistas. Exemplos importantes da adoção dessa prática são encontrados no *site* especializado em jogos Gamasutra[1] [Myllyaho et al. 2004].

Os *postmortems* do Gamasutra seguem a estrutura proposta pelo *Open Letter Template* [Myllyaho et al. 2004], trazendo uma descrição resumida do projeto e as informações mais importantes sobre o desenvolvimento do jogo. Em seguida, o relato é dividido em duas seções:

- **O que deu certo:** relata as boas práticas utilizadas no projeto, soluções, melhoramentos, decisões de projeto e gerenciamento que foram acertadas, podendo, em muitos casos, servir de modelo para futuros projetos.

[1] `http://www.gamasutra.com`

- **O que deu errado:** relata dificuldades, enganos e decisões de projeto que resultaram em problemas durante o desenvolvimento do jogo, tanto na parte técnica quando na área gerencial.

O relato é encerrado com uma mensagem final do relator e é reservada uma seção na qual tipicamente é feita uma ficha do jogo, com algumas informações como editor e desenvolvedor; número de desenvolvedores em tempo integral, tempo parcial e contratados; tempo de desenvolvimento e data da liberação; plataforma do jogo, além do hardware e software utilizados no desenvolvimento.

Para realizar a análise de postmortens e a coleta das boas práticas, foi necessário selecionar postmortems, analisá-los e compilar os resultados, chegando a um conjunto de práticas que são aprovadas ou desaprovadas por esses desenvolvedores. Realizamos o mesmo procedimento (análise de literatura e análise focada de post-mortens) para a realização de um diagnóstico dos maiores problemas e dificuldades encontrados no desenvolvimento de jogos (ver [Petrillo et al. 2009]).

Entre os vários postmortems existentes, selecionamos 20 *postmortems* publicados no *site* Gamasutra, e que estão listados na tabelas 1 e 3. Os 20 *postmortems* analisados foram selecionados de foram aleatória, tendo como único critério a seleção de projetos finalizados, isto é, que resultaram em um jogo completo e entregue ao mercado, não tendo sido analisado nenhum relato de projeto cancelado ou encerrado sem um produto.

Na fase de preparação foram estudadas as principais práticas ágeis [Cockburn 2000, Ambler 2004, Beck 1999, Poppendieck and Poppendieck 2003] e boas práticas aplicadas na indústria de jogos por [Schofield 2007, Gibson 2007]. Desse estudo, 12 boas práticas foram elencadas para análise.

Os 20 postmortems foram lidos e as sentenças que citavam as práticas foram destacadas. Durante esse fase, mais uma prática foi encontrada ("Crença no sucesso do projeto"), que foi acrescida ao rol analisado, totalizando 13 práticas, que são as listadas nas tabelas 1 e 2 e ainda na figura 2. Finalmente, uma tabela para a coleta dos dados foi organizada (tabela 1), sendo os projetos arranjados nas linhas e as boas práticas nas colunas.

4. Discussão

Ao analisarmos os resultados obtidos e presentes na tabela da seção anterior, vemos que as boas práticas mais encontradas foram **equipe qualificada, motivada ou coesa** e a **crença no sucesso do projeto**, com **90% (18 em 20)** dos projetos relatando tais práticas. Em seguida, destacaram-se o **estímulo à criatividade** e o **foco no produto**, com 80% dos projetos citando-as (16 em 20). A figura 2 apresenta o histograma de ocorrência das boas práticas analisadas em ordem decrescente, no qual pode-se fazer uma comparação gráfica desses resultados.

4.1. Análise de aderência das boas práticas já adotadas em projetos de jogos aos métodos ágeis

A indústria de jogos, inúmeras vezes de forma *ad hoc*, vem empregando boas práticas no desenvolvimento de software, como foi apresentado na seção 3. De forma mais organizada, podemos agrupar as práticas ágeis analisadas, segundo as práticas ágeis do Scrum, do XP e da Modelagem ágil, formando a tabela 2.

Tabela 1. Ocorrência de boas práticas nos projetos

Jogo	Equipe Qualificada, Motivada ou Coesa	Crença no Sucesso do Projeto	Estímulo à Criatividade	Foco no Produto	Controle de Versão	Uso de Ferramentas Simples ou Produtivas	Boas Práticas de Programação	Modelagem Ágil	Processo Definido	Controle de Qualidade	Retorno Rápido ou Iterativo	Boas Práticas de Gerenciamento	Integração Contínua	Total	% de boas práticas encontradas no projeto
Beam Runner Hyper Cross	Sim	Sim	Sim	Sim	Sim	Sim	Sim	Sim	Sim	Não	Não	Não	Não	9	69%
Gabriel Knights	Sim	Não	Não	Não	Sim	Não	Não	Não	Não	Sim	Não	Não	Não	3	23%
Black & White	Sim	Sim	Sim	Sim	Não	Não	Não	Sim	Não	Sim	Não	Não	Não	6	46%
Rangers Lead the Way	Não	Não	Não	Não	Não	Sim	Sim	Não	Não	Não	Não	Não	Não	2	15%
Wild 9	Sim	Sim	Sim	Sim	Sim	Sim	Não	Sim	Não	Não	Não	Não	Não	7	54%
Trade Empires	Sim	Sim	Sim	Sim	Sim	Sim	Sim	Sim	Sim	Sim	Sim	Sim	Não	12	92%
Rainbow Six	Sim	Sim	Sim	Sim	Sim	Não	Não	Não	Não	Não	Não	Não	Não	5	38%
The X-Files	Sim	Sim	Sim	Sim	Sim	Não	Não	Não	Não	Não	Não	Não	Não	5	38%
Draconus	Sim	Sim	Sim	Não	Não	Não	Sim	Não	Não	Não	Não	Não	Não	4	31%
Cel Damage	Sim	Sim	Não	Sim	Sim	Sim	Sim	Não	Sim	Sim	Sim	Sim	Sim	11	85%
Comand and Conquer: Tiberian Sun	Sim	Sim	Sim	Sim	Não	Sim	Sim	Sim	Sim	Não	Não	Não	Não	8	62%
Asheron's Call	Não	Sim	Sim	Sim	Sim	Sim	Sim	Não	Sim	Sim	Não	Não	Sim	9	69%
Age of Empires II: The Age of Kings	Sim	Sim	Sim	Não	Sim	Sim	Não	Não	Não	Sim	Não	Sim	Não	7	54%
Diablo II	Sim	Sim	Sim	Sim	Sim	Não	Sim	Não	Sim	Sim	Sim	Não	Não	9	69%
Operation Flashpoint	Sim	Sim	Sim	Sim	Sim	Sim	Sim	Sim	Sim	Não	Sim	Não	Não	10	77%
Hidden Evil	Sim	Sim	Sim	Sim	Sim	Não	Não	Não	Não	Sim	Não	Não	Não	6	46%
Resident Evil 2	Sim	Sim	Não	Sim	Não	Sim	Não	Sim	Sim	Não	Sim	Sim	Não	8	62%
Vampire: The Masquerade	Sim	Sim	Sim	Sim	Não	Sim	Sim	Sim	Sim	Não	Não	Não	Não	8	62%
Unreal Tournament	Sim	Sim	Sim	Sim	Não	Não	Sim	Sim	Não	Não	Sim	Sim	Não	8	62%
Tropico	Sim	Sim	Sim	Sim	Sim	Sim	Não	Não	Não	Não	Não	Não	Não	6	46%
Ocorrências	18	18	16	16	13	12	11	9	9	8	6	5	2	**143**	**7,2**
%	90%	90%	80%	80%	65%	60%	55%	45%	45%	40%	30%	25%	10%	**0,55**	**55%**

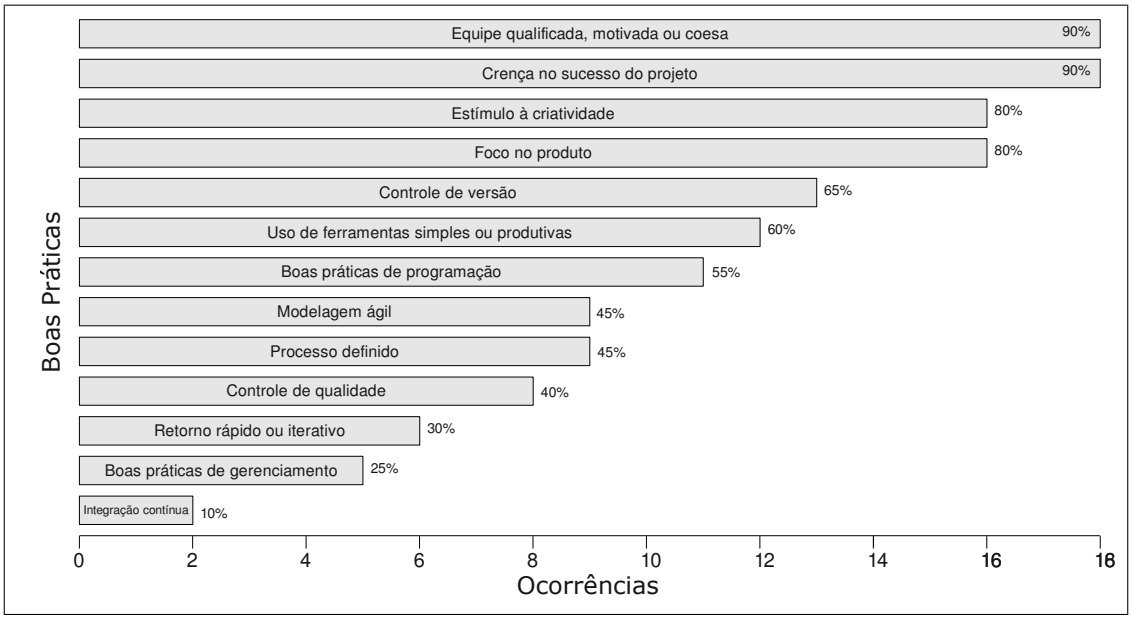


Figura 2. Ocorrências de boas práticas

Boa prática já adotada	**É aderente ao método...**
Equipe qualificada, motivada ou coesa	MA, Scrum, XP
Crença no sucesso do projeto	Scrum, XP
Estímulo à criatividade	Scrum, MA
Foco no produto	XP, Scrum
Controle de versão	XP
Uso de ferramentas simples ou produtivas	MA, XP
Boas práticas de programação	XP
Modelagem Ágil	XP, MA
Processo definido	Scrum, XP
Controle de qualidade	XP
Retorno rápido e iterativo	XP, Scrum
Boas práticas de gerenciamento	XP, Scrum
Integração contínua	XP

Tabela 2. Aderência das boas práticas já adotadas em jogos aos métodos ágeis

Mesmo de modo instintivo, as equipes de desenvolvimento de jogos vêm adotando essas práticas. Nesse cenário, a implantação de métodos ágeis, como o Scrum e XP, pode ocorrer naturalmente, visto que as equipes já utilizam, em suas rotinas, vários princípios da agilidade. Assim, através da tabela 2 é possível justificar a adoção dos métodos ágeis analisados neste trabalho.

Uma nova análise pode ser proposta ao cotejarmos os resultados apresentados por [Petrillo et al. 2009], que apresenta os problemas encontrados, com os resultados da análise de postmortems da seção 3, que contém as boas práticas levantadas nos *postmortems*. Ao avaliarmos, por exemplo, o projeto *Cel Damage*, que adotou **85%** das boas práticas, observamos a **menor incidência de problemas**, com somente **20%** dos problemas relatados. Existiria uma correlação linear entre o número de problemas encontrados e as boas práticas adotadas nos projetos de jogos?

Para essa análise, foi elaborada a tabela 3, que é formada pelos projetos de jogos analisados, os percentuais de problemas encontrados e os percentuais de boas práticas adotadas.

Projeto	**% de problemas encontrados no projeto**	**% de boas práticas encontradas no projeto**
Cel Damage	20,0%	84,6%
Trade Empires	26,7%	92,3%
Diablo II	33,3%	69,2%
Command and Conquer	33,3%	61,5%
Tropico	26,7%	46,2%
Age of Empires II	33,3%	53,8%
Resident Evil 2	40,0%	61,5%
Beam Runner	46,7%	69,2%
Asheron's Call	46,7%	69,2%
Hidden Evil	33,3%	46,2%
Vampire	46,7%	61,5%
Unreal	46,7%	61,5%
Wild 9	53,3%	53,8%
The X-Files	40,0%	38,5%
Operation Flashpoint	80,0%	76,9%
Black & White	60,0%	46,2%
Rainbow Six	73,3%	38,5%
Draconus	80,0%	30,8%
Rangers Lead the Way	46,7%	15,4%
Gabriel Knights	86,7%	23,1%

Tabela 3. Comparativo entre o problemas e práticas encontradas por projeto

A partir do levantamento feito, através da análise de *postmortems* com o foco em aspectos da engenharia de software, vemos que as boas práticas mais citadas foram a **equipe qualificada, motivada ou coesa** e a **crença no sucesso do projeto** com **90% (18 em 20)** dos projetos relatando essas duas práticas. Em seguida, destacaram-se o **estímulo à criatividade** e o **foco no produto**, com 80% dos projetos citando-as (16 em 20). Ainda nesse contexto, as outras duas práticas mais encontradas foram o **controle de versão**, com 65% (13 em 20) e o **uso de ferramentas simples ou produtivas**, com 60%.

As práticas gerenciais são uma carência clara na indústria de jogos. Não chega a 50% o número de projetos nos quais se identifica a adoção de um **processo definido** de trabalho e somente 25% (5 em 20) dos projetos adotaram **boas práticas de gerenciamento**. Além disso, somente 40% dos projetos utilizaram práticas de controle de qualidade e foi de 10% (2 em 20) a ocorrência constatada na aplicação da **integração contínua** nos projetos.

5. Conclusão

A maior contribuição deste trabalho foi ter ajudado a organizar o universo das práticas ágeis para o domínio de jogos, a partir da análise dos problemas e das boas práticas encontrados na indústria de jogos, sendo um ponto de partida para o uso dos métodos ágeis no processo de desenvolvimento de jogos eletrônicos.

Se pensarmos que todos os principais problemas da indústria tradicional de software são encontrados também na indústria jogos (ver [Petrillo et al. 2009], é possível constatar que estão correlacionados e que as soluções que podem ser adotadas na indústria tradicional de software deveriam tambem ser investigadas para a indústria de jogos.

Mesmo de forma *ad hoc* e isolada, as equipes de desenvolvimento de jogos vêm adotando um conjunto de práticas ágeis. Nesse cenário, a implantação de métodos ágeis, como o Scrum e o XP, pode ocorrer naturalmente, visto que as equipes já utilizam, em suas rotinas, vários princípios da agilidade.

Nossa reflexão sobre os resultados descritos neste artigo nos levam a crer que os desenvolvedores de jogos muitas vezes **não** adotaram **deliberadamente** estas boas práticas por serem **Ágeis**. Talvez, como muitos desenvolvedores de software, eles achem que - por serem informais - não correspondam às noções de rigor e sistematização geralmente associadas à Engenharia de Software.

Entretanto, elas são práticas estudadas e aplicadas cada vez mais pela comunidade de Engenharia de Software, embora nem sempre reconhecidas e divulgadas como formas disciplinadas de resolver os problemas crônicos do desenvolvimento. Como vimos, muitas das práticas ágeis já estão sendo adotadas (mesmo que parcialmente), e é nossa convicção que desenvolvedores - se estiverem dispostos a conhecer melhor os fundamentos do desenvolvimento ágil de software - podem facilmente incorporá-lo às suas atividades.

Uma vez que ainda são poucos os estudos acadêmicos sobre a utilização de métodos ágeis no desenvolvimento de jogos, este trabalho abre perspectivas para novas pesquisas. Acreditamos por exemplo que a modelagem (de preferência ágil) de elementos complexos como a jogabilidade e a diversão merece mais investigação.

Os estudos realizados por [Xu and Rajlich 2006], [Kasperavicius et al. 2008] e [Gibson 2007] nos levam a acreditar que a adoção de práticas ágeis no processo de desenvolvimento de jogos pode atingir resultados promissores. As práticas ágeis parecem contemplar melhor as características da indústria de jogos, como a multidisciplinaridade e as dificuldades na modelagem de aspectos como experiência do usuário (*user experience*), a diversão e prazer, visto que processos empíricos são mais adaptativos e reagem melhor a mudanças durante a implementação do projeto. Esperamos que este artigo seja uma contribuição para aumentar a adoção de práticas ágeis no processo de desenvolvimento de jogos.

Referências

Ambler, S. W. (2004). *Modelagem Ágil*. Bookman, São Paulo.

Beck, K. (1999). *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, Reading, MA, 1st edition.

Bethke, E. (2003). *Game Development and Production*. Wordware Publishing, Plano.

Birk, A., Dingsoyr, T., and Stalhane, T. (2002). Postmortem: never leave a project without it. *IEEE Software*, 19.

Blow, J. (2004). Game development: Harder than you think. *ACM Press Queue*, 1(10):28–37.

Callele, D., Neufeld, E., and Schneider, K. (2005). Requirements engineering and the creative process in the video game industry. In *13th IEEE International Conference on Requirements Engineering*.

Cockburn, A. (2000). *Agile Software Development*. Addison Wesley Longman, Reading, MA, 1st edition.

Cook, D. (2001). Evolutionary design - a practical process for creating great game designs. *GameDev.net*.

Crawford, C. (1984). *The Art of Computer Game Design*. Mcgraw-Hill Osborne Media.

Flood, K. (2003). Game unified process. *GameDev.net*.

Flynt, J. P. and Salem, O. (2004). *Software Engineering for Game Developers*. Software Engineering Series. Course Technology PTR, 1st edition.

Gershenfeld, A., Loparco, M., and Barajas, C. (2003). *Game Plan: the insider's guide to breaking in and succeeding in the computer and vieo game business*. St. Martin' s Griffin Press, New York.

Gibson, A. (2007). Agile game development and fun. Technical report, University of Colorado Department of Computer Science.

Highsmith, J. and Cockburn, A. (2001). Agile software development: The business of innovation. *IEEE Computer*, 34:120–122.

Kasperavicius, L. C. C., Bezerra, L. N. M., Silva, L., and Silveira, I. F. (2008). Ensino de desenvolvimento de jogos digitais baseado em metodologias Ágeis: o projeto primeira habilitação. In *Anais do XXVIII Congresso da SBC - Workshop sobre Educação em Computação*, pages 89 – 98, Belém do Pará.

Kreimeier, B. (2002). The case for game design patterns. Gamasutra - The Art & Business of Making Games, http://www.gamasutra.com/features/20020313/kreimeier_01.htm.

Marchesi, M., Succi, G., Wells, D., and Williams, L. (2002). *Extreme Programming Perspectives*. Addison Wesley.

McShaffry, M. (2003). *Game Coding Complete*. Paraglyph Press, Scottsdale.

Myllyaho, M., Salo, O., Kaariainen, J., and Koskela, J. (2004). A review of small and large post-mortem analysis methods. In *Proceedings of the ICSSEA*, Paris.

Petrillo, F., Pimenta, M., Trindade, F., and Dietrich, C. (2009). What went wrong? a survey of problems in game development. *ACM Computer in Entertainment*, CIE: 7(1).

Poppendieck, M. and Poppendieck, T. (2003). Principles of lean thinking.

Pressman, R. S. (2006). *Engenharia de Software*. McGraw-Hill, São Paulo, 6th edition.

Rouse, R. (2001). *Game Design: theory & practice*. Wordware Publishing, Inc.

Rucker, R. (2002). *Software Engineering and Computer Games*. Addison Wesley.

Schaefer, E. (2000). Postmortem: Diablo ii. *Gamasutra - The Art & Business of Making Games*.

Schofield, B. (2007). Embracing fun: Why extreme programming is great for game development. Gamasutra: The Art & Business of Making Games.

Shull, F., Mendonça, M., Basili, V., Carver, J., Maldonado, J. C., Fabbri, S., Travassos, G. H., and Ferreira, M. C. (2004). Knowledge-sharing issues in experimental software engineering. *Empirical Software Engineering*, 9(1-2):111–137.

Shull, F., Singer, J., and Sjøberg, D. I. (2008). *Guide to Advanced Empirical Software Engineering*. Springer-Verlag, London.

Sommerville, I. (2001). *Software Engineering*. International computer science series. Addison-Wesley, London, 6th edition.

Stalhane, T., Dingsoyr, T., Hanssen, G. K., and Moe, N. B. (2003). *Post Mortem? An Assessment of Two Approaches*, chapter Empirical Methods and Studies in Software Engineering, pages 129–141. Springer Berlin / Heidelberg.

Tsui, F. and Karam, O. (2007). *Essentials of software engineering*. Jones and Barlett Publishers, São Paulo, 6th ed edition.

Wohlin, C., Höst, M., and Henningsson, K. (2003). *Empirical Methods and Studies in Software Engineering*, chapter Empirical Research Methods in Software Engineering, pages 7–23. Springer Berlin / Heidelberg.

Xu, S. and Rajlich, V. (2006). Empirical validation of test-driven pair programming in game development. *Computer and Information Science, 5th IEEE/ACIS International Conference on*, 0:500–505.

Perspectiva Social, Comunicação e Cooperação nos Processos de Desenvolvimento de Software

Amaury Soares Pires de Carvalho[1], Ciro Carneiro Coelho[1]

[1] Pós-Graduação em Engenharia de Software - Faculdade Sete de Setembro (Fa7)
amauryspires@gmail.com, ciro@fa7.edu.br

Abstract. *This work presents a study on the social aspects contained in both predictive and adaptive software development processes. It points human factors that influence and are influenced by these processes. It also analyzes practices that favors cooperation based on the Theory of Games and indicates characteristics that improve cooperation between team members in software development based on Axelrod's cooperation factors.*

Resumo. *O presente trabalho investigou os aspectos sociais contidos nos processos preditivos e adaptativos de desenvolvimento de software. São apontados os fatores humanos que influenciam e são influenciados pelos processos. São analisadas, com base na Teoria dos Jogos, práticas que favorecem a cooperação entre membros das equipes de desenvolvimento com, bem como identificando, com base nos fatores de cooperação de Axelrod, caracterísitcas que propiciam uma maior cooperação nas equipes de desenvolvimento.*

1. Introdução

O crescente aumento na complexidade dos produtos de *softwares* ocasionou uma forte demanda por recursos humanos devidamente capacitados para sua produção, bem como por processos que possam suportar e potencializar tal força produtiva.

Dessa forma durante muito tempo o desenho de novos processos de desenvolvimento que permitissem a produção em massa de *software* eficiente e de qualidade foi uma preocupação importante na área de engenharia de *software*.

Na última década, porém, o foco vem mudando do processo para o indivíduo, como evidenciam os trabalhos de Beck (1999), o movimento liderado pela *Agile Alliance* com o Manifesto Ágil, e a proposta do quinto valor do manifesto ágil (MARTIN, 2008), *Craftsmanship over Execution,* e o conseqüente *Manifesto for Software Craftsmanship* (ISCC, 2009).

Desenvolvimento de *software*, segundo McBreen (2001), não é uma atividade realizada de forma solitária por indivíduos que escrevem programas para uso próprio. Ao invés disso, trata-se de uma tarefa intelectual social que requer muita comunicação e colaboração. A construção de sistemas, portanto, requer trabalho em equipe.

Num contexto onde o indivíduo e suas interações em equipe emergem como um fator relevante no cenário do desenvolvimento de *software* mundial, o estudo das características cooperativas e sociais presentes nos processos de desenvolvimento torna-se relevante para elucidar deficiências produtivas em equipes de software, uma vez que essas características são fortemente influenciadas pelo tipo de processo utilizado.

Highsmith *et al.* (2001) defende que "processos ágeis são desenhados para capitalizar o potencial único de cada indivíduo e cada equipe", porém tais métodos "não são para qualquer um". Pessoas com grande foco em processo e não colaborativas terão

dificuldade em aderir às idéias do desenvolvimento ágil, as quais funcionam melhor em culturas organizacionais voltadas para pessoas e que valorizam a colaboração.

Em contrapartida, os processos tradicionais, de acordo com Cockburn (2006), costumam apoiar-se em aspectos comportamentais pouco confiáveis das pessoas, tais como exigir que estas sejam lógicas e precisas, que não cometam erros, ou que pesquisem e usem soluções prontas – quando em geral preferem criar, ser conservadoras nas ações e escravas do hábito.

Nesse contexto, Cockburn (2006) propõe que o desenvolvimento de *software* seja visto sob o prisma de um processo onde os indivíduos interagem entre si segundos as regras de um jogo, onde a invenção, a comunicação e a cooperação atuam como peças importantes nesse quadro.

Este trabalho faz uma análise social dos processos de desenvolvimento de software sob o prisma da Teoria dos Jogos (DAVIS, 1997) e dos fatores de cooperação de Axelrod (1996; 2007). Algumas práticas são analisadas quanto à sua capacidade de promover cooperação entre os membros da equipe de desenvolvimento e são apontados os fatores sociais que influenciam os processos estudados, analisando pontos que favorecem uma maior cooperação entre os membros das equipes de desenvolvimento.

2. Teoria dos jogos

Teoria dos Jogos, segundo Morgenstern (*apud* DAVIS, 1997), é uma disciplina que vem ganhando grande interesse por parte do público em geral devido suas aplicações nas mais variadas áreas. Embora inicialmente a teoria focasse apenas o estudo do campo de exatas, a sua estrutura matemática atual consegue explicar com sucesso fenômenos sociais e seus problemas, tais como conflitos que levam a competição ou colaboração entre indivíduos.

A importância do estudo de problemas teóricos relacionados a jogos, segundo Davis (1997), é que:

> Num jogo existem outros indivíduos tomando decisões baseadas em seus próprios desejos. [...] Num jogo cada jogador deve perceber se seus objetivos estão alinhados ou não aos dos demais e decidir cooperar ou competir com todos ou [somente] alguns deles.

Conforme Fiani (2006), "a teoria dos jogos ajuda a entender teoricamente o processo de decisão de agentes que interagem entre si, a partir da compreensão da lógica da situação em que estão envolvidos". Para o autor (2006), toda situação que envolva interações entre agentes racionais que se comportam estrategicamente pode ser considerada e analisada como um jogo.

Um ponto importante da teoria é que esta diferencia basicamente dois tipos de jogos: os de soma-zero e de soma-não-zero. Jogos como xadrez ou damas são soma-zero devido o fato da vitória de um jogador significar a derrota do adversário, fazendo com que a soma do ganho positivo de um e a perda negativa do outro resulte em zero.

Embora o estudo de tais jogos tenha seu valor, jogos de soma-zero diferenciam-se de problemas reais, pois o primeiro permite encontrar soluções deterministas, enquanto o segundo em geral traz consigo múltiplas, indiretas ou simplesmente nenhuma solução universalmente aceitável (DAVIS, 1997). Nesse contexto, jogos de soma-não-zero são mais comuns no nosso dia-a-dia, além de serem mais interessantes e difíceis de analisar. Tais jogos caracterizam-se por não haver necessariamente relação direta entre o ganho de um jogador e a perda do adversário. Na verdade, não é necessário nem mesmo existirem perdedores para haver ganhadores.

Os jogos podem ainda ser caracterizados como cooperativos ou não-cooperativos. De acordo com Fiani (2006), um jogo é dito não-cooperativo quando os jogadores não podem estabelecer compromissos garantidos em relação ao jogo. Caso contrário, se os jogadores podem estabelecer compromissos relacionados ao jogo, e esses compromissos possuem garantias efetivas, diz-se que o jogo é cooperativo. Nessa definição, jogos de soma-não-zero são por natureza caracteristicamente cooperativos, pois permitem que tais compromissos sejam efetuados, mesmo que na prática os jogadores não o façam.

O nível de comunicação entre os jogadores tem papel preponderante em jogos de soma-não-zero, o que não acontece nos de soma-zero devido estes serem intrinsecamente competitivos. Em geral, quanto mais cooperativo o jogo, mais crucial para o resultado final deste torna-se a habilidade de comunicar-se (DAVIS, 1997). A comunicação é importante em jogos cooperativos devido a exigência de que cada indivíduo conheça os desejos e pretensões dos demais para que se possa tomar a decisão que melhor alinhe suas expectativas às dos envolvidos.

Embora notavelmente identificados com jogos de soma-não-zero, problemas do mundo real são difíceis de serem diretamente estudados devido não poderem ser analisados em ambientes cientificamente controlados (DAVIS, 1997). Então, uma forma de fazê-lo é substituindo-os por modelos de jogos mais simplificados que possam ser simulados quantas vezes se fizer necessário.

Um dos modelos mais estudados dos jogos de soma-não-zero é o problema do "Dilema do Prisioneiro", originalmente formulado por A. W. Tucker (DAVIS, 1997):

> Dois suspeitos [cúmplices] de cometerem um crime foram presos pela polícia e colocados em celas separadas. Cada suspeito deve confessar ou manter-se em silencio, e cada qual sabe as possíveis conseqüências de sua ação. Essas são: (1) Se um suspeito confessar e o outro não, o que confessou fornece evidencias ao Estado e é libertado enquanto o outro é preso por vinte anos. (2) Se ambos confessarem vão juntos para a prisão por cinco anos. (3) Se ambos ficarem em silencio, vão para a prisão por um ano por porte ilegal de armas. [...] Cada jogador tem duas escolhas básicas: agir "cooperativamente" ou "não cooperativamente". Quando todos os jogadores cooperam, obtém melhores resultados de quando não o fazem.

Jogos de soma-não-zero são intrincados, altamente sujeitos a interferências causadas pelo ambiente e comunicação envolvidos. Quando os interesses de múltiplos indivíduos entram em jogo o problema cresce de forma surpreendente. A interação entre indivíduos inseridos em um mesmo jogo expõe os jogadores a uma vasta possibilidade de situações, afetando-os individualmente e ao grupo como um todo.

Em geral sistemas profissionais não são escritos por desenvolvedores solos, mas sim por equipes de pessoas com objetivos em comum. Dessa forma, é de suma importância considerar-se tal aspecto comportamental dos jogos de soma-não-zero entre múltiplos indivíduos quando da análise dos processos de desenvolvimento de *software*.

3. Desenvolvimento de software: um jogo de comunicação e cooperação

Desenvolvimento de *software*, segundo Cockburn (2006), é uma atividade de cognição e expressão realizada por pessoas que pensam e comunicam-se à medida que trabalham contra limites econômicos, de tempo e de recursos, condicionadas às suas culturas e sensíveis aos indivíduos envolvidos no processo.

Assim, Cockburn propõe que a construção de um *software* por uma equipe pode ser visto como um jogo de invenção e cooperação onde os indivíduos às vezes cooperam e às vezes competem entre si.

Na teoria dos jogos, tal tipo de jogo é classificado na categoria de soma-não-zero – posto ser cooperativo, podendo assim gerar relações ganha-ganha –, com múltiplos indivíduos. Assim, no desenvolvimento de software, como em todos os jogos de soma-não-zero, a comunicação tem um papel preponderante. Logo, a qualidade dos participantes como um time pode ser medida através de sua capacidade de comunicação e cooperação durante o jogo. Para que isso aconteça, os indivíduos devem agir como especialistas em comunicação (COCKBURN, 2006).

No entanto, alguns desenvolvedores não gostam de desenvolvimento colaborativo, preferindo trabalhar sozinhos e mostrar o produto aos usuários somente quando a aplicação estiver pronta. Outros não lidam bem com ciclos incrementais de requisitos e preferem uma especificação formal e completa no inicio do projeto. Há ainda os que não abraçam o aprendizado contínuo, não aceitando facilmente novas idéias. Todos esses problemas minam a efetividade do desenvolvimento colaborativo.

Cockburn (2006) sugere estratégias que uma equipe deve seguir no jogo de desenvolver sistemas, tais como: a) manter a equipe original intacta o máximo de tempo possível; b) jogadores que são aprendizes em uma partida devem tornar-se lideres no jogo seguinte; c) utilizar um modelo de aprendizagem para treinar novatos na equipe; e, d) jogar de forma simples na primeira partida, reservando tempo para melhorar o jogo nas partidas seguintes (o que significaé realizado, no campo de *software,* pelo desenvolvimento iterativo e refatoração).

4. Processos de software analisados como dilema do prisioneiro

Desenvolvimento de *software* é uma tarefa intelectual social que requer muita comunicação, colaboração e, portanto, trabalho em equipe. Para que isso de fato ocorra, os indivíduos envolvidos no processo devem trabalhar de forma coesa, permitindo que estes evoluam de mero grupo de pessoas para o *status* de equipe.

Uma forma de analisar os padrões comportamentais apresentados por grupos de indivíduos expostos a determinados ambientes e situações é considerá-los imersos em um jogo, pois, segundo a Teoria dos Jogos, toda situação que envolva interações sociais pode ser analisada como tal.

Blumen (1995) afirma que, sob o ponto de vista da Teoria dos Jogos, toda interação social contida no desenvolvimento de *software* explica-se como dilemas do prisioneiro, sendo freqüentemente dilemas iterados. Como exemplo, o autor cita a relação comercial entre desenvolvedor e cliente, que se baseia na promessa de entrega de um bom produto pelo primeiro em troca do devido pagamento pelo segundo.

Lieberman (2001) cita que o dilema do prisioneiro emerge também na relação cliente *versus* desenvolvedor durante a negociação do escopo de projetos de *software*, quando se abre a possibilidade de apoio mútuo (cooperação) ou disputa (competição), esta última na forma de ditadura, extorsão ou impasse entre eles.

Semelhantemente, Hazzan & Dubinsky (2005) lembram que o dilema do prisioneiro também ocorre dentro da equipe de desenvolvimento, conforme demonstra o cenário de cooperação/competição entre dois desenvolvedores em um mesmo projeto (Quadro 1). Para os autores "a competição entre membros da equipe é a fonte de alguns dos problemas que caracterizam os processos de desenvolvimento de *software*".

Os autores afirmam que, num ambiente de desenvolvimento de sistemas, a cooperação e competição expressam-se em diferentes formas, tais como compartilhamento ou restrição de informação, o uso ou não dos padrões de codificação

da companhia, código de simples ou complexo entendimento, entre outras situações. A cooperação é vital em processos de engenharia de *software*, sendo importante mesmo quando é apenas parcial durante os projetos, como expresso nos quadrantes coopera-compete demonstrado no cenário do Quadro 1.

Hazzan & Dubinsky (2005) afirmam que os membros da equipe são freqüentemente solicitados a cooperarem entre si, mas, quando há incerteza se haverá reciprocidade por parte dos demais membros do projeto a tendência natural é a competição, resultando no pior cenário (quadrante compete-compete do Quadro 1).

		Desenvolvedor B	
		Coopera	Compete
Desenvolvedor A	Coopera	O projeto é completado no tempo previsto. Contribuição pessoal de A e B são iguais, compartilhando o bônus pelo sucesso do projeto.	Cooperação de A leva o projeto a ser completado no tempo previsto. Porém, desde que A dedica parte do tempo analisando e corrigindo o código escrito por B, enquanto B continua trabalhando em sua própria parte, a contribuição de A para o projeto é avaliada como menor que B. Como resultado, B aparenta ser responsável pela produtividade e sucesso do projeto.
	Compete	Semelhante ao apresentado em "A coopera" *versus* "B compete", porém A apresenta melhor produtividade.	Desde que ambos exibem comportamento competitivo, não completam o projeto no tempo previsto, cancelando-se o projeto.

Quadro 1. Dilema do prisioneiro em equipes de desenvolvimento de software.

Nesse contexto, de acordo com os autores (2005), a análise da interação da equipe – mais especificamente os processos que estes aplicam no seu dia a dia – na forma de cenários de competição e cooperação modelados como dilema do prisioneiro pode auxiliar na solução de conflitos existentes dentro de projetos de *software*.

4.1 O modelo ganha-ganha de negociação de conflitos

Egyed & Boehm (1998) propõem um modelo que visa identificar soluções para situações que envolvam dilema do prisioneiro, mais especificamente a conciliação de interesses conflitantes através de negociação, objetivando estabelecer relações de ganha-ganha entre os envolvidos.

O modelo, segundo os autores, possui quatro artefatos principais, conforme apresentado na Figura 1: condições de ganho, problemas, opções e acordos.

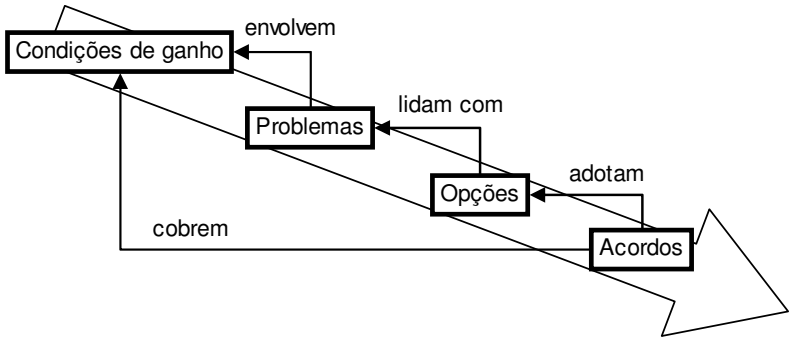


Figura 1. Artefatos do modelo ganha-ganha de negociação de conflitos.

De acordo com o modelo, "condições de ganho" capturam os interesses – na forma de objetivos e preocupações – de cada um dos envolvidos no dilema. Em seguida, caso não sejam controversos, estes são logo cobertos por um "acordo". Em caso contrário um "problema" é estabelecido para registrar o impasse entre os mesmos.

Na existência de um "problema", "opções" são registradas na forma de ações e alternativas que possam eliminar ou pelo menos mitigar a situação. Após a negociação

do “problema” – através da exposição aos envolvidos das “opções” – um “acordo” é estabelecido registrando qual opção foi utilizada para resolver o “problema”.

Dessa forma, o modelo expõe metodicamente as questões levantadas no dilema vivido pelos envolvidos e como os acordos foram estabelecidos, provendo uma documentação que pode ser utilizada no futuro para promover lições aprendidas além de tornar transparentes as decisões tomadas.

4.2 Práticas ágeis mapeadas como dilema do prisioneiro

Extreme Programming (XP), segundo Astels *et al.* (2002), “é um jogo produtivo” e “rápido”, no qual “os elementos importantes [...] são seus princípios, suas práticas e seus jogadores”, e onde “ganhar [...] significa entregar um produto”. Adicionalmente, Beck (2004) afirma que XP é um jogo que objetiva construir relações ganha-ganha entre as partes, mais notadamente a área de negócios e desenvolvimento.

XP possui práticas organizacionais (ex: desenvolvimento iterativo, cliente presente), práticas de equipe (ex: propriedade coletiva, ritmo sustentável) e práticas individuais (ex: projeto simples, refatoração) (ASTELS *et al.*, 2002).

Serão analisadas aqui, na forma de cenários de impasse mapeados como dilema do prisioneiro, uma prática de equipe, representada pela “programação em pares”, uma individual, constituída pelo “desenvolvimento baseado em testes”, e uma organizacional, o “desenvolvimento iterativo”.

O primeiro cenário (Quadro 2), expõe problemas e opções disponíveis quando membros do projeto discordam na adoção da “programação em pares”.

“Programação em pares” é, segundo Beck (2004), “um diálogo entre duas pessoas tentando programar simultaneamente (e analisar, projetar e testar) e entender em conjunto como programar melhor”, sendo também “uma conversação em muitos níveis, assistida por um computador e focada nele”.

De acordo com Williams (2000), muitos rejeitam a idéia de programação em pares porque assumem que o custo homem-hora irá duplicar quando aloca-se dois desenvolvedores na mesma atividade. Porém, ainda segundo o autor, o que ocorre de fato é a redução no tempo total de desenvolvimento, diminuindo os custos.

PRÁTICA:	**Programação em pares**	
DILEMA DO PRISIONEIRO:		
Cooperação:	Todos os membros formam pares, alternando-se periodicamente, para conduzir em conjunto suas atividades de desenvolvimento conforme sugerido pelo processo (ASTELS *et al.*, 2002; BECK, 2004).	
Competição:	Um ou mais membros não seguem a recomendação de trabalhar em par, ou o fazem aquém do recomendado pelo processo (ASTELS *et al.*, 2002; BECK, 2004).	
MODELO DE NEGOCIAÇÃO GANHA-GANHA:		
	Cooperador	Competidor
Condições de ganho:	Ganha quando compartilha suas atividades de desenvolvimento com um parceiro.	Ganha quando analisa, desenvolve e testa de forma individual, sem interferências de terceiros.
Problemas:	Quando a programação em pares não é feita perde-se em oportunidade de comunicação, coesão da equipe, maior compromisso entre os indivíduos, produtividade, melhor relação custo-benefício e qualidade de código (BECK, 2004). A programação *solo* restringe a visão do desenvolvedor, tendendo este a se ater apenas às partes e não ao todo do problema (ASTELS *et al.*, 2002).	Programação em pares é uma prática difícil e cansativa (SHARP; ROBINSON, 2006). Falta de espaço físico, monopólio do teclado, conflitos de propriedade ou prioridade entre atividades, ou simples desavenças pessoais entre indivíduos podem impactar na produtividade quando estes codificam em par (ASTELS *et al.*, 2002; BECK, 2004). Tal prática aumenta o custo do desenvolvimento ao alocar dois recursos a uma única atividade.
Opções:	Pode-se usar o papel “treinador” – *coach* –	Pode-se estabelecer um horário para a prática

	como um comunicador para divulgar o processo como um todo para a gerência e a equipe, instigando-os a desenvolver suas habilidades técnicas e de abstração de problemas, auxiliando na adoção de outras práticas, além de estar disponível como parceiro de desenvolvimento, em particular para desenvolvedores iniciantes na prática de programação em pares (BECK, 2004).	ao invés de tê-la em tempo integral. Outras práticas como "jogo do planejamento", "padrões de codificação", "ritmo sustentável" e "projeto simples" minimizam desavenças na equipe. Além disso, deve-se evitar pares entre indivíduos que não se dão bem no campo pessoal, e deve-se rearranjar o espaço físico de forma a propiciar a prática (BECK, 2004; WILLIAMS, 2000).
Acordos:	Adotar o papel "treinador" na equipe, incentivar o uso das outras práticas individuais e de equipe, estabelecer horário para a programação em par e redispor o espaço físico do local de trabalho da equipe, o que está de acordo com o princípio de adaptação local da XP pela gerencia como proposto por Beck (2004).	

Quadro 2. Prática "programação em pares" mapeada como dilema do prisioneiro.

Em contrapartida, Astels *et al.* (2002) lembra que a programação em pares não afeta a velocidade em si do desenvolvimento, isto é, "produz-se em par pelo menos tanto quanto individualmente", porém, a qualidade final do produto é superior.

Beck (2004) afirma que "a programação em pares pode ser desconfortável no princípio", devendo-se ser rígido em aplicá-la. A visão de uma prática intensa e cansativa é compartilhada também por Sharp & Robinson (2006), que desaconselham sua aplicação em tempo integral. Além disso, Astels *et al.* (2002) diz que "muitos desenvolvedores sentem que a programação em pares limita sua independência".

Para Kniberg (2007) essa aversão à prática se deve principalmente à falta de experiência real na mesma e que, quando é dada uma oportunidade para tal, em geral, esta visão é modificada. Mesmo assim, segundo o autor, certos indivíduos simplesmente não se adaptam a ela, visão também compartilhada por Beck (2004).

Williams *et al.* (2000) cita que, numa experiência realizada com alunos da universidade de Utah em 1999, identificou-se que cerca de 90% dos envolvidos preferiram a programação em par do que a *solo* e que nela viam-se mais confiantes do resultado entregue, mas que sentiram a prática mais eficaz nos momentos de análise, projeto e testes do que de codificação propriamente dita. O autor cita também que as principais preocupações apresentadas pelos alunos eram a quantidade de comunicação adicional requerida, a adaptação aos hábitos, estilo de programação e ego do parceiro – tanto excesso, por não deixar-se ajudar, quanto falta, quando não ajuda o parceiro – além das possíveis desavenças sobre aspectos de implementação.

Assim, a prática "programação em pares" expõe um forte viés social ao quebrar o paradigma do desenvolvedor *solo*. A análise ganha-ganha constante no Quadro 2 demonstra que impasses em sua adoção na equipe podem ser mitigadas seguindo-se práticas e conselhos simples constantes no próprio processo XP.

Já a prática "desenvolver o *software* iterativamente", demonstrada no Quadro 3, discute o cenário onde a cultura tradicional de desenvolvimento, representado por defensores do modelo cascata, confronta-se com a proposta de desenvolvimento iterativo e incremental, gerando impasse dentro da equipe.

Kruchten (2003) descreve que o modelo iterativo "quebra o ciclo de desenvolvimento em uma sucessão de iterações", além disso, cada iteração "parece pequenos projetos" envolvendo atividades de captura de requisitos, projeto, implementação e garantia da qualidade. Além disso, o autor defende que a importância do modelo iterativo incremental está em mitigar riscos de projeto, em acomodar mais facilmente mudanças, promover reuso e também permitir a equipe aprender sobre os requisitos à medida que desenvolve o sistema.

Defensores do modelo tradicional, por sua vez, relatam a dificuldade gerencial de estimar o esforço total do projeto (KRUCHTEN, 2003), bem como de substituir princípios de engenharia há muito tempo estabelecidos na industria de *software* e baseadas em técnicas exaustivamente testadas por outras industrias por novos conceitos que exigem treinamento por parte dos desenvolvedores (LARMAN *et al*., 2002).

Além disso, o tamanho e quantidade de iterações a serem aplicados em cada projeto também podem motivar conflitos na equipe, no que tange o confronto da visão mais preditiva de longas iterações versus mais adaptativa de pequenas e rápidas entregas (LARMAN *et al*., 2002).

Novamente o impasse na equipe, quando analisado sob o ponto de vista de dilema do prisioneiro mostrado no Quadro 3, pode ser solucionado com a situação ganha-ganha de treinamento e adaptação do desenvolvimento iterativo, principalmente em relação à duração das iterações, adequando-o ao projeto e estilo cultural vigentes.

PRÁTICA:	**Desenvolver o *software* iterativamente.**	
DILEMA DO PRISIONEIRO:		
Cooperação:	Todos os membros da equipe desenvolvem de forma iterativa e incremental de acordo como recomendado pelo processo (KRUCHTEN, 2003).	
Competição:	Um ou mais membros não desenvolvem iterativamente – isto é, adotam o modelo tradicional cascata – ou o fazem aquém do sugerido pelo processo (LARMAN *et al.*, 2002).	
MODELO DE NEGOCIAÇÃO GANHA-GANHA:		
	Cooperador	Competidor
Condições de ganho:	Ganha quando os sistemas são desenvolvidos de forma iterativa e incremental.	Ganha quando os sistemas são desenvolvidos de forma menos iterativa e mais tradicional.
Problemas:	Não desenvolver iterativamente significa enfrentar problemas de acomodação de mudanças e mitigação de riscos, originado pela impossibilidade de congelamento de requisitos – cliente e projeto mudam com o tempo – o que impacta no modelo preliminar que foi proposto no inicio do projeto (KRUCHTEN, 2003).	Desenvolver iterativamente vai contra as práticas de engenharia tradicionais e dificultam a estimativa de esforço do projeto, principalmente quando contratos de preço fixo são exigidos (LARMAN *et al*., 2002).
Opções:	O desenvolvimento iterativo presume mini-cascatas dentro de cada iteração, com porções de elicitação de requisitos, análise e projeto, implementação, integração e teste (LARMAN *et al*., 2002; KRUCHTEN, 2003). Pode-se promover e valorizar o uso da prática através de treinamento da equipe e demais interessados no projeto, principalmente com o estabelecimento de "mentores" e uso do papel "engenheiro de processo" (LARMAN *et al*., 2002; KRUCHTEN, 2003)	Pode-se reduzir o número de iterações, aumentando proporcionalmente o tamanho destas, de acordo com o tipo de projeto enfrentado e o tamanho da equipe alocada, fazendo com que a prática se aproxime do modelo tradicional cascata facilitando assim sua adoção (LARMAN *et al*., 2002).
Acordos:	Promoção de treinamento na prática para os envolvidos no projeto. Além disso, considerar o tamanho e tipo de projeto, bem como o tamanho da equipe, na decisão do número de iterações necessárias, o que está de acordo com o principio de adaptação do processo RUP à empresa e projeto (KRUCHTEN, 2003; LARMAN *et al*., 2002).	

Quadro 3. Prática "desenvolver software iterativamente" mapeada como dilema do prisioneiro.

O terceiro cenário analisado como dilema do prisioneiro, trata da adoção pela equipe da prática individual "desenvolvimento guiado por testes" (Quadro 4).

"Desenvolvimento guiado por testes" é, conforme Beck (2004), uma atividade feita pelos desenvolvedores para aumentar sua confiança no código. Nela, testes são escritos antes de codificar o programa, para então serem guardados e executados posteriormente sempre quando o programa for modificado.

Segundo Beck (2004), testar o que se faz vai contra a natureza humana, por isso deve-se buscar motivos que tornem a prática benéfica para o praticante. O principal motivo em curto prazo para escrever testes antes do código é que estes aumentam a confiança do desenvolvedor sobre o que está sendo escrito.

Além disso, quando a análise ganha-ganha é feita na situação de impasse de adoção da prática, conforme mostrado no Quadro 4, observa-se novamente que o processo XP possui dispositivos reguladores sociais intrínsecos que auxiliam na mitigação de tais ocorrências, evitando-se apoiar-se somente em normas para isso.

PRÁTICA:	**Desenvolvimento guiado por testes**	
DILEMA DO PRISIONEIRO:		
Cooperação:	Todos os membros escrevem os testes antes do código conforme sugerido pelo processo (ASTELS *et al*., 2002; BECK, 2004).	
Competição:	Um ou mais membros não seguem a recomendação de escrever testes antes do código, ou os escrevem aquém do recomendado pelo processo (ASTELS *et al*., 2002; BECK, 2004).	
MODELO DE NEGOCIAÇÃO GANHA-GANHA:		
	Cooperador	Competidor
Condições de ganho:	Ganha quando equipe escreve testes automatizados.	Ganha em não escrever testes automatizados.
Problemas:	Não escrever testes reduz a confiança no código. Além disso, ganhos de produtividade em curto prazo por não ter que construir testes automatizados resultam em longo prazo em atrasos por retrabalho causado por excesso de depuração e análise de código desnecessário (ASTELS *et al*., 2002; BECK, 2004). Outro ponto é que manter um testador que só age no final do ciclo de desenvolvimento retarda a identificação de problemas, encarecendo o processo (ASTELS *et al*., 2002).	Escrever testes diminui a produtividade da equipe. Além disso, a maioria do código existente foi escrita sem testes automatizados, de onde se pode inferir que estes últimos não são essenciais para o desenvolvimento de *software* (BECK, 2004). Outro ponto é que testes devem ser escritos por outra equipe, composta por especialistas em testes, os quais verificarão e validarão o sistema exaustivamente após o mesmo tiver sido escrito, gerando um extenso e detalhado relatório de problemas a serem resolvidos pelos desenvolvedores (ASTELS *et al*., 2002; BECK, 2004).
Opções:	Pode-se utilizar "programação em pares" para promover a prática de testes, pois a primeira diminui a probabilidade de que ambos os desenvolvedores esqueçam simultaneamente de seguir a segunda devido ao efeito "pressão dos pares" (ASTELS *et al*., 2002; BECK, 2004; KNIBERG, 2007).	Estabelecer claramente para a equipe de desenvolvedores o nível de erro tolerável em determinado período de tempo ajuda a equacionar o número de testes automatizado que deve ser escrito (BECK, 2004).
Acordos:	Adoção da "programação em pares" buscando unir programadores com níveis diferentes de experiência em teste, estabelecimento do nível de teste que será escrito pela equipe ajustado às exigências de cada projeto e sugestão de foco primeiro na escrita dos testes mais simples e urgentes às equipes iniciantes (ASTELS *et al*., 2002; BECK, 2004; KNIBERG, 2007).	

Quadro 4. Prática "desenvolvimento guiado por testes" mapeada como dilema do prisioneiro.

5. Cooperação e os processos de desenvolvimento de software

Cooperação é um aspecto social que desempenha importante papel no âmbito das relações humanas por permitir agregar indivíduos em torno de um fim comum, otimizando a utilização de recursos e maximizando os benefícios advindos dessa

interação. Nesse contexto, tal aspecto merece, portanto, análise teórica dos fatores que a fazem emergir, manter-se e disseminar-se em uma dada sociedade.

Com base na literatura sobre o assunto publicada por Axelrod (1997; 2006), o corrente estudo extraiu 17 condições (Quadro 5), que, de acordo com a freqüência de ocorrência em um sistema social, incentivariam a cooperação entre indivíduos.

Para melhor organização didática, propõe-se ainda a divisão de tais fatores em quatro grupos distintos, a saber: comportamento, contexto, estratégia e território.

Grupo	Condição	Descrição
Comportamento	Admiração	Admira qualidades dos indivíduos do grupo, não cultua a inveja.
	Arrependimento	Não retalia a resposta a uma defecção prévia do próprio jogador.
	Bondade	Inicia a interação cooperando, nunca ataca primeiro.
	Generosidade	Permite que parte das defecções ocorra sem retaliação principalmente devido à possibilidade de ruídos de comunicação.
	Gratidão	Coopera quando recebe cooperação.
	Distinção	Tem capacidade de reconhecer características que diferenciam um indivíduo ou situação de outra.
	Irritabilidade	Retalia a defecção sofrida.
	Memória	Recorda as interações passadas.
Contexto	Futuro	Futuro vantajoso em relação ao presente.
	Periodicidade	Traz alta probabilidade de novas interações.
	Recompensa	Métodos e normas que tornam a cooperação mais atrativa que a defecção.
Estratégia	Clareza	Tem clareza de comportamento, usa estratégias simples.
	Maleabilidade	Mantém estratégias lucrativas, modifica as de baixo valor.
	Reciprocidade	Adota estratégias baseadas na reciprocidade aplicando os fatores gratidão e irritabilidade, mesmo que em medidas diferenciadas.
Território	Compatibilidade	Ocorre compatibilidade cultural entre os grupos.
	Comunicação	Existência abundante de comunicação entre as partes.
	Liberdade	Multiplicidade de estratégias permitidas.

Quadro 5. Resumo dos fatores de cooperação de Axelrod.

No grupo comportamento tem-se características inerentes ao indivíduo propriamente dito. O grupo contexto traz fatores oriundos da situação em que o indivíduo está imerso. Já no grupo estratégia lista-se os fatores relativos ao *modus operandi* do indivíduo perante os demais membros da sociedade. Por fim, o grupo território traz características da região onde o indivíduo está incluso.

De acordo com Axelrod (1997; 2006), a freqüência com que tais fatores ocorrem em uma dada sociedade diz se há incentivo à cooperação – muita ocorrência – ou à competição – pouco vestígio – entre seus indivíduos.

No campo do desenvolvimento de *software*, equipes trabalham cooperativamente com o objetivo de produzir sistemas. Porém, de acordo com Highsmith (*apud* PRESSMAN, 2006), colaboração não é algo fácil de se obter, uma vez

que trata-se, sobretudo, de uma questão de confiança entre os membros da equipe. Para que tal nível de confiança ocorra, faz-se necessária ampla e irrestrita comunicação entre os membros da equipe (COCKBURN, 2006).

Nesse contexto, buscou-se no corrente estudo identificar evidências da existência dos fatores de cooperação propostos por Axelrod (1997; 2006) em dois processos de desenvolvimento, RUP (KRUCHTEN, 2003) e XP (BECK, 2004), representando, respectivamente, processos preditivos e adaptativos.

5.1 Evidências dos fatores de cooperação na literatura sobre o processo RUP

Para que a relação entre os fatores de cooperação e o processo RUP seja determinada de forma mensurável, a condução de uma pesquisa de campo faz-se necessária, não sendo este objetivo do corrente trabalho. Porém, podem-se identificar evidências iniciais dessa relação através do estudo das características intrínsecas do processo descritas de forma empírica na literatura pesquisada.

Dessa forma, o Quadro 6 demonstra a análise dos fatores sociais de cooperação relacionados ao processo RUP, indicando, através da pontuação caracteristicamente competitiva da maioria dos itens, sua natureza metodológica preditiva na medida em que requer a presença de uma autoridade central fiscalizadora que garanta a colaboração entre os indivíduos, além de normas com força de lei que apóiem tal agente.

Tal autoridade central é representada pelo papel "gerente de projeto", sendo que as normas constantes no processo apóiam-se predominantemente em mecanismos como "dominância", "dissuasão" e "lei" para que motivem a colaboração dos indivíduos.

PROCESSO: *Rational Unified Process (RUP)*					
FATORES PARA COOPERAÇÃO:					
		Compete ◄		► Coopera	
		Reduz	Indifere	Incentiva	Comentário
Comportamento	Admiração	X			A especialização de papéis leva à pulverização das atividades que por sua vez promove o distanciamento funcional resultando em inveja e descontentamento quando há papeis mal alocados ou insatisfação devido a gargalos motivados por centralização excessiva de tarefas.
	Arrependimento			X	Grande cerimônia, fluxos bem definidos e incentivo a um gerenciamento iterativo e incremental permite que problemas sejam identificados e corrigidos no decorrer do projeto.
	Bondade			X	Papéis e responsabilidades bem definidas levam cada membro a cooperar quando uma ação sua é requerida.
	Generosidade	X			Papéis e responsabilidades bem definidas fazem cada membro esperar que seu parceiro cumpra à risca sua parte no projeto.
	Gratidão		X		Papéis e responsabilidades bem definidas levam cada membro a cooperar indiferentemente da reação dos demais indivíduos.
	Distinção			X	Papéis e responsabilidades bem definidas deixam claros a cada membro o que os demais fazem e a fase atual do projeto.
	Irritabilidade	X			Não cooperação na equipe é retaliada de forma indireta através do acionamento do papel gerente de projeto.
	Memória			X	Por manter artefatos detalhados ao longo do projeto, a rastreabilidade de quem fez o quê e quando é garantida.
Contexto	Futuro		X		O processo foca-se apenas na execução do projeto e entrega do produto requisitado, sendo indiferente às questões relativas a qualquer futura solicitação das partes interessadas.
	Periodicidade			X	As interações freqüentes entre os mesmos membros são garantidas pela abordagem iterativa e incremental.
	Recompensa			X	Por ter fluxos bem definidos o processo apóia-se em normatização

					(diretrizes) para garantir a cooperação entre os membros da equipe.
Estratégia	Clareza		X		Papéis e responsabilidades bem definidas trazem clareza de comportamento no nível individual, contra balanceada, porém, por uma estratégia complexa no nível de equipe.
	Maleabilidade	X			Mesmo instanciado em cada projeto, os fluxos de atividades bem definidos apóiam-se fortemente em regras que inibem comportamentos e ações não previamente estabelecidos.
	Reciprocidade	X			Por ser indiferente ao fator "gratidão" e tornar indireta a "irritabilidade", o processo reduz a ocorrência de "reciprocidade" dentro da equipe.
Território	Compatibilidade	X			A especialização de papéis causa o aumento tanto da diversidade cultural quanto dos grupos antagônicos no projeto, devido à redução dos aspectos culturais comuns.
	Comunicação	X			Por basear-se em artefatos como principal forma de comunicação, reduz o número de canais comunicativos disponíveis, desencorajando comunicados não documentados.
	Liberdade	X			Sendo um processo preditivo, adere melhor em organizações centradas em documentação onde a liberdade de ação é culturalmente limitada.

Quadro 6. Análise dos fatores de cooperação no processo RUP.

5.2 Evidências dos fatores de cooperação na literatura sobre o processo XP

Nos mesmos moldes da análise realizada na seção 6.3.1 para o RUP, a literatura pesquisada sobre o XP pode ser explorada de forma a encontrar indicações empíricas de características cooperativas no processo.

O Quadro 7 demonstra a análise dos fatores sociais de cooperação relacionados ao processo XP, indicando, através da pontuação caracteristicamente cooperativa da maioria dos itens, sua natureza metodológica adaptativa na medida em que dispensa a existência de uma autoridade central para garantir o comportamento colaborativo entre os indivíduos, apoiando-se em normas sociais motivadoras de cooperação por "pressão dos pares". Tais normas sociais valem-se principalmente de mecanismos como "aprovação social", "participação" e "reputação" para que motivem a pressão dos pares.

PROCESSO: *Extreme Programming (XP)*					
FATORES PARA COOPERAÇÃO:					
		Compete ◄——| Coopera		
		Reduz	Indifere	Incentiva	Comentário
Comportamento	Admiração			X	Valorização do indivíduo ao invés do processo (AMBLER, 2004) além de papéis flexíveis e aderentes aos talentos disponíveis incentivam e expõem as qualidades da equipe.
	Arrependimento			X	O estilo iterativo e incremental e o incentivo a valores como "*feedback*" e "coragem", bem como "humildade", promovem na equipe confiança para correção de equívocos.
	Bondade			X	Os envolvidos são levados a comprometerem-se com o projeto, já que os desenvolvedores são incentivados a candidatarem-se a tarefas e a auxiliarem uns aos outros.
	Generosidade			X	Valores como "comunicação" e "*feedback*", bem como as práticas "propriedade coletiva", "refatoração" e "programação em pares" estimulam a parceria no grupo permitindo a solução de mal entendidos.
	Gratidão			X	Práticas como "jogo do planejamento" e "programação em pares" promovem a camaradagem dentro da equipe.
	Distinção	X			O pouco registro das situações vivenciadas nos projetos e interações ocorridas reduz o nível de distinção na equipe, restringindo-a aos envolvidos em cada ocorrência.
	Irritabilidade			X	"*Feedback*" e "coragem" incentivam a exposição de problemas.

					"Programação em pares" e "testes" promovem retaliação direta a comportamentos não cooperativos.
	Memória	X			Foco na comunicação face-a-face ao invés de formas documentadas reduz a memória de ocorrências, restringindo-a às pessoas diretamente envolvidas nessas situações.
Contexto	Futuro			X	Constante preocupação com o envolvimento do cliente e com os futuros trabalhos resultantes do corrente projeto.
	Periodicidade			X	O estilo iterativo e incremental, bem como as práticas "entregas freqüentes" e "programação em pares", garantem vários ciclos de interações entre os envolvidos.
	Recompensa			X	"Padrões de codificação" e "programação em pares" estabelecem regras e meios de auto policiamento da equipe.
Estratégia	Clareza		X		Papéis e responsabilidades amplas inibem a clareza de comportamento no nível individual, contra balanceada, porém, por uma estratégia simplificada no nível de equipe.
	Maleabilidade			X	O processo pode ser aplicado parcialmente e em conjunto com práticas de outros processos.
	Reciprocidade			X	"Propriedade coletiva" estimula o senso de unidade e auto preservação no grupo. "Reciprocidade" promovida através dos fatores "gratidão" e "irritabilidade" supracitados.
Território	Compatibilidade			X	Papéis generalistas reduzem a diversidade cultural na equipe, porém aumentam os aspectos culturais comuns entre os indivíduos. Práticas como "propriedade coletiva" e "padrões de codificação" incentivam o senso de unidade na equipe.
	Comunicação			X	Incentiva uma comunicação direta face-a-face, aberta e honesta entre os envolvidos no projeto.
	Liberdade			X	O processo estimula e é estimulado por ambientes culturais organizacionais onde há liberdade de adoção de novas idéias e confiança nos profissionais.

Quadro 7. Análise dos fatores de cooperação no processo XP.

6. Conclusão

Desenvolvimento de software não é uma atividade solitária, mas uma tarefa intelectual social que requer comunicação, colaboração e trabalho em equipe. Este trabalho apresentou um estudo dos aspectos sociais facilitadores dessa cooperação, analisando algumas práticas ágeis sob o prisma da Teoria dos Jogos. Foi realizada também uma comparação entre processos preditivos e adaptativos com base nos fatores de cooperação de Axelrod.

Concluiu-se que processos preditivos e adaptativos são metodologias que se apóiam, respectivamente, em práticas de engenharia de software e engenharia social. Processos adaptativos capitalizam o potencial individual e coesão da equipe, mas requerem uma cultura organizacional centrada em pessoas. Processos preditivos, por sua vez, se adequam melhor a pessoas centradas em processo e não colaborativas, bem como a uma cultura organizacional mais tradicional.

Processos preditivos pressupõem a existência de normas com força de lei acompanhadas por uma autoridade central fiscalizadora bem estabelecida e reconhecida pela equipe – no caso, o papel gerente de projeto – para que o processo seja seguido, ocorra cooperação e eventuais conflitos sejam dirimidos.

Processos adaptativos, por sua vez, não dão grande ênfase a uma autoridade central mediadora de conflitos, pois delegam tal papel a normas sociais que agem nos indivíduos através da "pressão dos pares". Hazzan & Dubinsky (2005) afirmam que a análise do método XP como dilema do prisioneiro ilustra claramente como este promove a confiança nos membros da equipe e paulatinamente leva-os a situações cooperativas de ganha-ganha. Essa afirmação é corroborada pelo presente estudo, como pode-se perceber pela análise realizada na seção 5.2. O uso conjunto dos mecanismos

sociais providos pelas práticas propostas pelo processo e o estabelecimento de um ambiente físico propício para que negociações possam ocorrer de forma justa entre as partes, permite aos membros da equipe dirimir conflitos sem interferências externas.

7. Referências

ASTELS, D.; MILLER, G.; NOVAK, M. Extreme Programming. Campus, 2002.

AXELROD, R.M. The evolution of cooperation, revised edition. Basic Books Inc Publishers. 2006.

AXELROD, R.M. The complexity of cooperation: agent-based models of competition and collaboration. Princeton University Press. 1997.

BECK, K. Programação extrema explicada: acolha as mudanças. Bookman. 2004.

BLUMEN, J. The prisoner's dilemma in software development. Ethical Spectacle, Vol. I, No. 9, September, 1995.

COCKBURN, A. Agile Software Development, second edition. Addison-Wesley Professional, 2006.

DAVIS, M. D. Game Theory: a nontechnical introduction. Dover Publications. 1997.

EGYED, A.; BOEHM, B. Telecooperation Experience with the WinWin System. 15th IFIP World Computer Congress, 1998.

FIANI, R. Teoria dos jogos: com aplicações em economia, administração e ciências sociais, 2.ed.rev e atual. Elsevier. 2006.

HAZZAN, O.; DUBINSKY, Y. Social Perspective of Software Development Methods: The Case of the Prisoner Dilemma and Extreme Programming, 2005.

HIGHSMITH, J.; COCKBURN, A. Agile Software Development: The People Factor. Computer, Los Vaqueros Cir., p. 131–133, nov. 2001.

ISCC. Manifesto for Software Craftsmanship. International Software Craftsmanship Conference, 2009.

KNIBERG, H. Scrum and XP from the trenches. InfoQ, 2007.

KRUCHTEN, P. Introdução ao RUP: rational unified process, nova edição revisada. Ciência Moderna. 2003.

LARMAN, C. Agile and Iterative Development: A Manager's Guide, 11th printing. Addison Wesley. 2004.

LIEBERMAN, B. Project Scope Management: Effectively Negotiating Change. The Rational Edge, November 2001.

MARTIN, R.C. Quintessence: The fifth element for the Agile Manifesto. 2008.

McBREEN, P. Software Craftsmanship: The New Imperative. Addison Wesley. 2001.

PRESSMAN, R.S. Engenharia de Software, 6ª edição. McGraw-Hill. 2006.

SHARP, H.; ROBINSON, H. Collaboration in mature XP teams. Centre for Research in Computing, The Open University, PPIG Newsletter, september 2006.

WILLIAMS, L; KESSLER, R; CUNNINGHAM, W; JEFFRIES, R. Strengthening the case for pair-programming. University of Utah, 2000.

Sessão 4

Estudos empíricos com métodos ágeis

Experiência Acadêmica de uma Fábrica de *Software* utilizando *Scrum* no Desenvolvimento de *Software*

Alinne C. Corrêa dos Santos[1], João Paulo F. Sette[1], Adauto T. de Almeida Filho[1], Igor Cavalcanti Ramos[1], Luciano Soares de Souza[1], Luis Alberto L. Lima[1], Rafael de Araujo Bacelar[1], Rebecca C. Linhares de Carvalho[1,] Fábio Q. B. da Silva[1]

[1] Centro de Informática – Universidade Federal de Pernambuco (CIN – UFPE)
CEP 50732-970 – Recife – PE – Brasil

{accs,jpfs,ataf,icr2,lss2,lall,rab5,rclc,fabio}@cin.ufpe.br

Abstract. *In recent years, has been seen a growing movement around the use of agile methodologies in distributed software development environment. However, according to the literature, this scenario has been little discussed and the major Software factories doesn't have current process models , which can accommodate this peculiarity. This paper will present the results from a software factory's academic experience on Distributed Development based on the Scrum process, for developing the FireScrum tool.*

Keyword: Distributed Software Development, Scrum, FireScrum, Desktop Agent

Resumo. *Nos últimos anos, tem-se observado uma crescente movimentação em torno da utilização de metodologias ágeis em ambientes de Desenvolvimento Distribuído de Software. Entretanto, de acordo com as literaturas, este cenário foi pouco discutido e a maioria das fábricas de Software, atuais, não possuem processos modelados que possam contemplar esta peculiaridade. Neste artigo serão apresentados os resultados da experiência acadêmica de uma fábrica de Software, baseada no processo de Desenvolvimento Distribuído de Software embasada na metodologia ágil Scrum para o desenvolvimento da ferramenta FireScrum.*

Palavra-chave: Distributed Software Development, Scrum, FireScrum, Desktop Agent

1. Introdução

Nos últimos anos, pode-se perceber um grande avanço em direção à globalização dos negócios, em particular, aos que estão relacionados com um intenso investimento na tecnologia de desenvolvimento de s*oftware*. Dada esta evolução, sabe-se que o s*oftware* tem se tornado um componente vital para quase todos os negócios. Neste sentido, para as organizações que buscam sucesso, é clara a necessidade do uso da Tecnologia da Informação (TI) de forma mais estratégica, o que têm estimulado o Desenvolvimento Distribuído de *Software* (DDS) em escala mundial.

Atualmente, é notável que o número de empresas que estão distribuindo seus processos de desenvolvimento de *software* ao redor do mundo é cada vez mais significativo. Assim, essas empresas visam obter vantagens competitivas associadas ao custo, qualidade e flexibilidade no desenvolvimento de s*oftware*, buscando um aumento de produtividade, assim como, diminuição de riscos (SENGUPTA, 2006).

Em busca dessas vantagens no ambiente de negócio, um número significativo de empresas passou a viver, a partir do ano 2000, uma tendência para o desenvolvimento ágil de aplicações. Este processo ocorre devido ao ritmo acelerado de mudanças e inovações na tecnologia da informação (BOEHM, 2006). A essência desse movimento é a definição do novo enfoque de desenvolvimento de *software*, calcado na agilidade, na flexibilidade, nas habilidades de comunicação e na capacidade de oferecer novos produtos e serviços de valor ao mercado, em curtos períodos de tempo (HIGHSMITH, 2004).

Em paralelo a essa discussão, a adoção de abordagens ágeis tem sido crescente na indústria de software, apresentando resultados positivos em termos de prazo, custo e qualidade (VERSIONONE, 2008). Tais abordagens visam à desburocratização das atividades (SCHWABER, 2004), e empregam princípios como ciclos iterativos e entregas rápidas de software funcionando (BECK et al., 2001). A essência desse movimento é calcada na agilidade, flexibilidade e habilidade de comunicação (HIGHSMITH, 2004). Dentre as metodologias ágeis propostas, o *Scrum* tem se apresentado como uma das mais adotadas. Observa-se em pesquisas que 70% das empresas que utilizam metodologias ágeis, adotam o *Scrum* (VERSIONONE, 2008).

É importante ressaltar que a combinação de metodologias ágeis com DDS está crescendo no mercado. Soares et al. (2007), em seu estudo, apresenta a utilização do método ágil, *Scrum*, em um processo de desenvolvimento de software *open source*, com fortes características de DDS. Luna et al. (2008) também ressalta essa combinação no desenvolvimento, de forma distribuída e apoiada no Scrum, de uma biblioteca multimídia, para o compartilhamento de objetos de aprendizagem, utilizados pelos projetos de telemedicina e telesaúde do Núcleo de Telesaúde da UFPE. Portanto, é notável que a utilização de DDS com Scrum, pode ser considerado um processo simples e objetivo, focado nas necessidades do cliente, e não, na documentação do projeto, apresentando resultados satisfatórios de acordo com os estudos de Soares et al. (2007) e Luna et al. (2008).

Motivado por este cenário, o artigo apresenta um experimento acadêmico, que consiste na vivência de uma fábrica de s*oftware* na disciplina de Engenharia de *Software* do curso de pós-graduação em Ciência da Computação, baseado no método *Scrum*, em um processo de Desenvolvimento Distribuído de *Software.*

O artigo está organizado em cinco seções. A seção 2 apresenta o contexto relacionado ao Desenvolvimento Distribuído de *Software;* A seção 3 aborda a metodologia ágil, *Scrum;* A Seção 4 exibe o processo de desenvolvimento da ferramenta *FireScrum*, detalhando o módulo *Desktop Agen*; A seção 5 evidencia os resultados obtidos e as ferramentas utilizadas, bem como, os problemas identificados e suas respectivas soluções. Por fim, a seção 6 apresenta as considerações finais e perspectivas de trabalhos futuros.

2. Desenvolvimento Distribuído de *Software*

Motivados, principalmente, pela globalização, muitas empresas nas últimas décadas passaram a distribuir seus processos de desenvolvimento em lugares diferentes, levando suas equipes a trabalharem de forma distribuída. Segundo Zanoni (2002), com a distribuição geográfica de recursos e investimentos, surge uma nova tendência de desenvolvimento de software, em que usuários e equipes de desenvolvimento estão em locais físicos diferentes, às vezes, com culturas diferentes. O Desenvolvimento Distribuído de Software, incluindo a terceirização, subcontratação e parcerias, tornou-se uma realidade empresarial comum.

Para Audy e Prikladnicki (2007), o desenvolvimento distribuído de software ganha cada vez mais força, motivado por três fatores ligados ao ambiente de negócios: a globalização; o crescimento da importância dos sistemas de informação nas empresas e os processos de terceirização, que geram ambiente propício a esse cenário de desenvolvimento. Portanto, é justificável que as empresas de software, em sua grande maioria, estão adotando a prática de DDS para construir seus softwares, assim conseguem recrutar trabalhadores qualificados e de baixo custo em todo mundo (LIANG, 2008). Times distribuídos podem aumentar o aprendizado e a criatividade da equipe, já que os membros têm a chance de interagir com uma grande variedade de culturas, experiências e pontos de vista. Um time formado por uma gama de conhecimentos diferentes é capaz de realizar um variado número de ações (BAROFF, 2002).

O DDS tem sido caracterizado, principalmente, pela colaboração e cooperação entre departamentos de organizações e pela criação de grupos de pessoas que trabalham em conjunto, porém, estão localizados em cidades ou países diferentes, distantes temporal e fisicamente. Assim, as principais características que diferenciam o desenvolvimento co-localizado do desenvolvimento distribuído são: distância (a distância entre os desenvolvedores e entre os desenvolvedores e clientes), diferenças de fuso horário e cultural (incluindo a língua, tradições, costumes, comportamentos e normas). A literatura reconhece não só a distância física, mas também os seguintes tipos de distância: temporal, cultural, organizacional (diferentes culturas organizacionais envolvidas), e a distância entre os *stakeholders* (a quantidade de pessoas interessadas no projeto com diferentes objetivos em mente).

A engenharia de software global tornou-se parte da estratégia de crescimento das empresas, pois podem estar mais próximas dos mercados locais e entender melhor as necessidades regionais. Além disso, alguns países não têm recursos suficientes para a demanda de TI de produtos e serviços de software (EBERT et al., 2008). Logo, aproveitar os recursos globais para o desenvolvimento de software tornou-se quase uma regra para grandes empresas. Alguns fatores (Figura 2.3) são citados como principais motivos do crescimento do DDS, segundo os estudos Prikladnicki (2003) e Freitas (2005):

- Necessidade de recursos globais para serem utilizados a qualquer hora, inclusive profissionais qualificados em áreas especializadas;
- Incentivos fiscais para o investimento de pesquisas em informática;
- Disponibilidade de mão-de-obra especializada e de custos reduzidos em países em desenvolvimento;
- Vantagem de estar perto do mercado local, incluindo o conhecimento, os clientes e as condições locais;

- Rápida formação de organizações e equipes virtuais para explorar as oportunidades locais;
- Grande pressão para o desenvolvimento *time-to-market* (velocidade no trabalho, tempo entre a concepção e a comercialização do produto) utilizando as vantagens do fuso horário diferente, no desenvolvimento conhecido como *follow-the-sun* (24 horas contínuas);
- Necessidade de integrar recursos resultantes de aquisições e fusões organizacionais.

É possível perceber, com as razões mencionadas, que, embora as empresas queiram reduzir os custos no desenvolvimento e aproveitar as oportunidades dos mercados locais, elas também querem ter equipes cada vez mais capacitadas, o que nem sempre é plausível encontrar em um único local. Porém, com pessoas capacitadas em lugares diferentes, é difícil e financeiramente inviável deslocar todas para um mesmo ambiente físico, o que estimula o desenvolvimento distribuído.

O ambiente global apresenta grande impacto na forma como os produtos são concebidos, desenvolvidos, testados e entregue aos clientes. Além disso, Herbsleb (2001) destaca que trabalhar com DDS é um dos maiores desafios que o atual ambiente de negócios apresenta, do ponto de vista do processo de desenvolvimento de software.

Segundo MacGregor et al. (2005), são de conhecimento dos profissionais da área, as dificuldades e baixas taxas de sucesso de projetos de software com equipes co-localizadas. Novas variáveis como: distância, comunicação virtual, diferenças de fusos horários e culturais, não contribuem para que essas taxas de sucesso melhorem. Enquanto a abordagem clássica de desenvolvimento de software com equipes co-localizadas permite a resolução de problemas no corredor, durante um café ou ao redor de uma mesa, equipes distribuídas são formadas por pessoas culturalmente, etnicamente e funcionalmente diversificadas. Pessoas que trabalham em diferentes horários e locais nem sempre são, facilmente, acessíveis para uma conversa sobre como criar uma interface ou resolver um *bug* que impede um teste de prosseguir (EBERT et al., 2008).

Para Liang (2008), o tempo e a distância em um projeto que trabalha de maneira distribuída é um dos grandes desafios para o sucesso, pois significa mais infraestrutura e maior coordenação para estabelecer uma comunicação eficaz, dentro do projeto. Além disso, a distância física e as diferenças culturais entre os membros são, na sua maioria, muito grandes e a comunicação pode ser ineficiente, já que, as pessoas que não estejam fisicamente ao seu lado, são mais fáceis de ser ignoradas.

Uma solução óbvia, para os inúmeros desafios que a distribuição intensifica, seria evitar o desenvolvimento distribuído de software. No entanto, para organizações que estão sempre em busca de vantagens competitivas em escala global, os benefícios potenciais de desenvolvimento distribuído podem ser muito atraentes para, simplesmente, serem ignorados (MAK, 2007).

3. Metódos Ágeis e Scrum

O termo “Metodologias Ágeis tornou-se popular em 2001, quando especialistas em processos de software estabeleceram princípios comuns, sendo criada a Aliança Ágil e o estabelecido o “Manifesto Ágil (BECK, et al., 2001).

Alinhado aos princípios ágeis, o *Scrum* foi criado, inicialmente, como um framework para gerenciamento de projetos na indústria convencional e publicado no

artigo *The New Product Development Game* (TAKEUCHI, 1986). O primeiro desenvolvimento de software com *Scrum* foi realizado em 1993 por Jeff Sutherland na Easel Corporation (Sutherland, 2004), e junto com Ken Schwaber formalizaram o *Scrum* (Sutherland, 2007), como processo de desenvolvimento na OOPSLA (SCHWABER, 1997).

O *Scrum* congrega atividades de monitoramento e feedback, através de reuniões rápidas e diárias com toda a equipe, visando à identificação e correção de quaisquer deficiências e/ou impedimentos, no processo de desenvolvimento (SCHWABER, 2004). A proposta é baseada em um ciclo iterativo e incremental, onde cada iteração é planejada de acordo com a prioridade definida pelo cliente.

O método baseia-se nos seguintes princípios: equipes pequenas; requisitos que são pouco estáveis ou desconhecidos; e iterações curtas. Este método apresenta três papéis principais e independentes (BEEDLE, 2004). *Product Onwer* (PO), que estabelece objetivos do produto, define e prioriza as funcionalidades, ou itens de *backlog* e participa, ativamente, do desenvolvimento, validando o produto de cada *sprint*; *Scrum Master* (SM), cujo papel é facilitar o trabalho do time, removendo os impedimentos levantados pelo time e apoiando o mesmo no uso do *Scrum*; Time, responsável pelo desenvolvimento dos itens de *backlog*, define como transformar o *product backlog* em incremento de funcionalidades, gerenciando seu próprio trabalho. São responsáveis, coletivamente, pelo sucesso da iteração e, conseqüentemente, pelo projeto como um todo.

O *Scrum* não requer ou fornece qualquer técnica ou método específico para a fase de desenvolvimento de *software*, apenas estabelece conjuntos de regras e práticas gerenciais que devem ser adotadas para o sucesso de um projeto. As práticas gerenciais do *Scrum* estão divididas em artefatos e cerimônias. Os artefatos são compostos por: *Product Backlog, Sprint Backlog e Burndown;* as cerimônias são compostas por: *Daily Scrum, Sprint Planning Meeting, Sprint Backlog* e *Sprint Review Meeting*.

3.1. Ciclo do Scrum

O início do projeto de *Scrum* ocorre a partir da existência de uma visão do produto que será desenvolvido (SCHWABER, 2004). O ciclo de vida de projetos *Scrum* é definido em iterações que podem durar de duas a quatro semanas, conhecidas como *sprint*. Ao final de cada *sprint* é entregue um incremento do produto tendo passado por todo o processo de desenvolvimento, auditoria e teste.

Segundo Schwaber (2004), cada *Sprint* inicia-se com uma reunião de planejamento (*Sprint Planning Meeting*), na qual o *PO* e o time decidem, em conjunto, o que deverá ser implementado (*Selected Product Backlog*). A reunião é dividida em duas partes. Na primeira parte, *Sprint Planning* 1, o *PO* apresenta os requisitos de maior valor e prioriza aqueles que devem ser implementados. O time, então, define colaborativamente, o que poderá ser desenvolvido na próxima *Sprint*, considerando sua capacidade de produção. Na segunda parte, *Sprint Planning* 2, o time planeja seu trabalho, definindo o *Sprint Backlog*, que são as tarefas necessárias para implementar as funcionalidades selecionadas no *Product Backlog*.

Durante a execução das *Sprints*, o time faz, diariamente, uma reunião de quinze minutos para acompanhar o progresso do trabalho. Nessa reunião diária (*Daily Scrum Meeting*), cada membro do time responde a três perguntas básicas: O que eu fiz no

projeto desde a última reunião? O que irei fazer até a próxima reunião? Quais são os impedimentos?

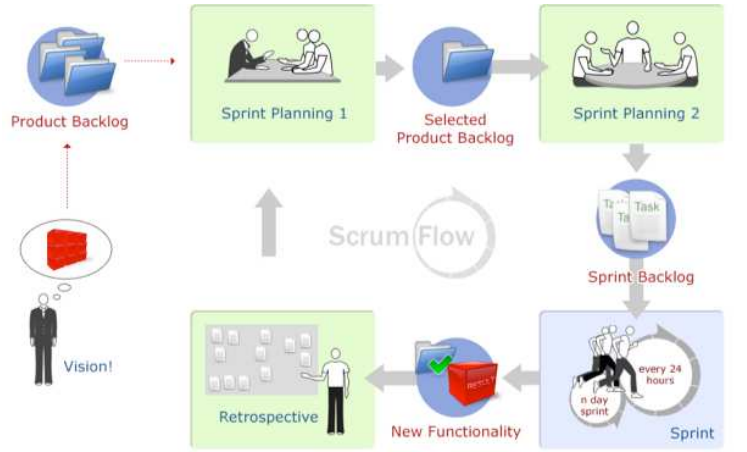


Figura 1. Visão geral do processo do *Scrum* (fonte: adaptada de (GLOGER, 2007)).

No final da *Sprint* é realizada a reunião de revisão (*Sprint Review Meeting*) para que o time apresente o resultado alcançado na iteração ao *Product Owner*. Neste momento, as funcionalidades são inspecionadas e adaptações do projeto podem ser realizadas. Em seguida, o *ScrumMaster* conduz a reunião de retrospectiva (*Sprint Retrospective Meeting*), que ocorre no final de uma *Sprint* para identificar o que funcionou bem, o que pode ser melhorado e que ações serão tomadas para melhorar o processo.

3.2. Scrum Distribuído

O *Scrum*, assim como outras metodologias ágeis, enfatiza a interação do time através de comunicação constante e *face-to-face*. No entanto, o desenvolvimento por equipes remotas tem apresentado um crescimento contínuo, particularmente visível desde a última década (DAMIAN & MOITRA, 2006). Entre as razões para a adoção por times distribuídos, pode-se destacar facilidade de envolvimento de especialistas em áreas chave, flexibilidade para criação e desativação de equipes de trabalho e a redução de custos de locomoção. Neste contexto, o modelo de *Scrum* de *Scrums* distribuído é recomendado pela ScrumAlliance (www.scrumalliance.com), que divide o trabalho em partes (times *Scrum)* isoladas, eliminando a maioria das dependências (SUTHERLAND, et al., 2007).

4. Estudo de Caso: Ferramenta FireScrum

4.1. FireScrum

O *FireScrum* é uma ferramenta open source que reúne um conjunto de aplicações integradas para suportar equipes que utilizam o *Scrum* como base para o desenvolvimento de seus projetos, sendo, especialmente, útil para equipes que trabalham remotamente.

A ferramenta é um produto idealizado por Eric Cavalcanti, que nasceu das atividades desenvolvidas durante o programa de mestrado de engenharia de software do CESAR.EDU, no Centro de Estudos e Sistemas Avançados do Recife (C.E.S.A.R).

O *FireScrum* foi desenvolvido utilizando conceitos da Web 2.0 e de *Rich Internet Applications* (RIA), apresentando foco em usabilidade para garantir a simplicidade do seu uso. É uma aplicação Web, o que significa que a mesma é acessível, remotamente, através de um browser, podendo ser utilizada em um ambiente de Internet ou Intranet, apresentando uma arquitetura modular e extensível.

A primeira versão do *FireScrum*, somente era composta pelo módulo *Task Board*, que foi desenvolvido por uma equipe de alunos do Mestrado Acadêmico do Centro de Informática (CIN) da Universidade Federal de Pernambuco (UFPE). Diante deste cenário, com a finalidade de continuar a desenvolver e melhorar a ferramenta, cerca de sessenta alunos da disciplina de Engenharia de *Software* do curso de pós-graduação em Ciência da Computação do CIN da UFPE constituíram um grande time para desenvolver os novos módulos para o *FireScrum*.

O projeto da ferramenta foi composto por seis novos módulos: *Planning Pooker, Test, Task Board, Core, Bug Tracking e Desktop Agent.* Esses módulos foram desenvolvidos durante os quatro meses da disciplina e por seis grupos distintos compostos pelos os alunos da disciplina, os quais foram formados de forma aleatória pelos responsáveis que conduziram a Fábrica de Software. Esses grupos interagiram por meio da troca de emails, ferramenta oro-aro[1], compartilhamento de conhecimentos e ferramentas, durante as aulas da disciplina e reuniões entre os *SM*. Cada grupo tinha, em média, cerca de dez pessoas, as quais se encontravam geograficamente dispersas. Dentre os módulos desenvolvidos, na próxima subseção será detalhada a realidade do time responsável pelo desenvolvimento do módulo *Desktop Agent*, abordando a experiência do time, procedimentos adotados para a execução das *sprints*, ferramentas utilizadas, as dificuldades enfrentadas, bem como os problemas identificados e suas respectivas soluções.

4.1. Módulo *Desktop Agent*

O *Dektop Agent* é um dos módulos que compõem a ferramenta *FireScrum*, desenvolvido seguindo os princípios ágeis do *Scrum*. Neste módulo, foi desenvolvido um aplicativo que é executado no próprio desktop, facilitado o acesso às funcionalidades do *FireScrum*, uma vez que, quando executado, estará disponível através da bandeja do Sistema Operacional. O usuário terá acesso às funcionalidades através de um menu suspenso. Dentre as funcionalidades destacam-se: visualizar tarefas e tarefas do usuário, editar tarefas, receber alertas do sistema, reportar *bugs* e fornecer um chat entre os integrantes do time.

O desenvolvimento do módulo foi constituído por seis *sprints* e compostas por histórias. O status delas foi definido por cores distintas. Inicialmente, foram definidas doze histórias pelo *PO* e, dentro de cada s*print*, eram realizadas reuniões do *SM* com *PO*, com a finalidade de acordar quais histórias agregavam maior valor de negócio. Cada s*print* foi dividida em duas semanas e, ao final do prazo, um conjunto do novo produto era apresentado ao *PO* e aos demais times.

As priorizações das histórias de cada *sprint* foram realizadas de acordo com o *Business Value* (BV) fornecido pelo PO. Juntamente com o BV, o PO definia o objetivo

[1] Ferramenta online que se trata de um meio de comunicação, interação e disponibilidade de conhecimentos entre alunos e professores.

da sprint, onde sugeria quais itens eram mais importantes para serem feitos na *sprint*. De acordo com os itens priorizados pelo PO, o time jogava o planning poker para realizar a estimativa, em seguida, verificar quais dos itens priorizados eram possíveis de serem realizados na *sprint*.

Em seguida, com as histórias acordadas com o *PO*, o time dividia em tarefas com média de execução de, aproximadamente, dois dias. As tarefas, bem como, o controle das s*prints* e o b*urndown* eram disponibilizados no *Google Docs*, onde, o *PO* e os demais times podiam acompanhar o andamento da execução dos mesmos.

Durante as *sprints*, o time utilizava o s*kype* para simular o *Daily Scrum Meeting*, assim, permitindo que todos os membros do time ficassem cientes do andamento das atividades e dos impedimentos encontrados. Em paralelo ao *Daily Scrum Meeting*, aconteciam as reuniões presenciais do time (uma vez por semana). Assim, como a comunicação do time é imprescindível durante a s*print*, a comunicação entre o *SM*, os demais membros do time e o *PO* era realizada por email ou por meio da ferramenta oro-aro, seja para aprovação de documentos, para o esclarecimento de dúvidas ou alinhamento do andamento da s*print*.

Antecedendo o final da s*print*, precisamente, às vésperas, era realizada uma reunião de *review* interna e de retrospectiva. A reunião de *review* interna contava somente com a participação do time e tinha por finalidade discutir o que foi desenvolvido, verificando se foi atendida a proposta da história, bem como, obter sugestões de melhorias.

O final da s*print* acontecia às segundas-feiras, com a presença de todos os seis times, objetivando apresentar ao *PO* o produto e o desempenho da equipe, por meio do *burndown*, até o presente momento. Cada time era representado por um de seus membros durante a apresentação. A análise do esforço planejado e realizado ocorria durante a reunião de retrospectiva da *sprint*, onde os membros do time faziam levantamento dos pontos positivos e pontos a serem melhorados que ocorreram na *sprint*.

5. Resultados

Os resultados do desenvolvimento do módulo *Desktop Agent* foram obtidos por meio da execução de seis s*prints*. Ao término de cada s*print* foi apresentado uma parte do módulo geral ao *PO*. Os registros da obtenção desses resultados serão apresentados por s*prints* e são de extrema importância, pois evidenciam a experiência da equipe, as ferramentas utilizadas, as dificuldades enfrentadas, os problemas identificados e suas respectivas soluções.

Na s*print* 1 foi acordado com *PO* o desenvolvimento de uma única história, a qual tratava da criação de um ícone que aparecesse no *systray*, apresentando os menus que, futuramente, seriam as funcionalidades dessa aplicação. Foi escolhida apenas uma, pois constatou-se a necessidade de aprofundamento no estudo das tecnologias a serem utilizadas como o *AdobeAir*; bem como ambientação com a metodologia adotada (*Scrum*).

Nesta *sprint* foram realizadas trinta e oito tarefas, dentre as quais podem ser citadas: ambientação do time às tecnologias a serem utilizadas durante o desenvolvimento do módulo; estudos referentes à forma de comunicação do aplicativo *Desktop Agent* com o servidor; escolha da melhor tecnologia (AMF ou *WebService*); estudo referente à tecnologia *Adobe Air* para obter o resultado proposto pela história

desta s*print* e, por último, definição do processo de testes referente à criação e execução de casos de teste. O fato de ser a primeira *sprint*, a inexperiência do time e a falta de afinidade com essa metodologia ocasionou alguns problemas que foram solucionados. Dentre esses problemas pode-se enfatizar a precisão referente ao tempo, onde as reuniões demoravam cerca de uma hora e a escolha da ferramenta para a realização do primeiro *Daily Scrum Meeting*. Nas reuniões realizadas posteriormente esses problemas foram solucionados. Definiu-se o *skype* como ferramenta padrão para as reuniões, pois, os membros encontravam-se fisicamente dispersos, e notava-se a necessidade de realizar vídeo conferência. Outros chats testados (*Gtalk e MSN*) apresentaram instabilidades.

Na s*print* 2 foi acordado com o *PO*, o desenvolvimento de três histórias: 1 - Deixar usuário e senha cadastrados no *login*: tem por objetivo armazenar as informações do usuário no momento do *login*, bem como permitir o *login* automático; 2 - Visualizar minhas tarefas: permite que o usuário tenha acesso às tarefas alocadas a ele na *sprint* corrente, referente a um determinado produto; 3 - Visualizar todas as tarefas da *sprint*: acesso à listagem de todas as tarefas da s*print* corrente.

Nessa *sprint* foram realizadas cinqüenta e uma tarefas, dentre as quais podem ser citadas: a criação de protótipos das telas de acordo com os requisitos levantados, sendo estes validados com o *PO* e, conseqüentemente, utilizados pelos membros do time, responsáveis pelo desenvolvimento das histórias citadas acima; a criação de casos de teste para cada história, onde foi utilizada a ferramenta *open source*, *Test Link*, que permitiu agilidade no processo de criação, execução, e controle dos testes. Essa ferramenta de teste foi selecionada para uso, devido à afinidade e experiência dos membros do grupo com a mesma. Dentro do processo de execução dos testes, quando um *bug* era identificado, abria-se uma *Change Request* (CR), utilizando o *Mantis*. Este *software* é um sistema de análise de erros, *open source*, baseado em tecnologia Web, e permite especificar os responsáveis pela declaração de *bugs*, bem como, seus respectivos analisadores. Esse sistema foi utilizado por todos os módulos do *FireScrum*.

Ao longo dessa s*print* foram identificados problemas como: pouco tempo dos participantes; indisponibilidade dos integrantes com maiores habilidades em programação; afastamento de um dos desenvolvedores do time e, o fato de ser o primeiro contato dos programadores com as tecnologias utilizadas e com a arquitetura do *FireScrum*. Estes problemas demandaram maior trabalho e esforço do time e foram solucionados quando os mesmos passaram a se dedicar, exclusivamente, ao adiantamento das tarefas.

Na s*print* 3 foi acordado com o *PO* o desenvolvimento de três tarefas: 1 - Editar minhas tarefas: o usuário pode alterar as informações referentes às tarefas alocadas a ele na s*print* corrente; 2 - Alocar-me nas tarefas: permite ao usuário se alocar e desalocar em quaisquer tarefas da *sprint*, mesmo se essas estiverem alocadas a outros usuários; 3 - Reportar b*ugs*: permite ao usuário reportar *bugs* encontrados por meio do *Desktop Agent*, não havendo a necessidade de acessar o *FireScrum* para reporta-los.

Nessa s*print* foram realizadas setenta e três tarefas e criadas mais treze tarefas, referentes às correções de *bugs* encontrados durante a s*print* anterior e *rework* de CR. Ao longo dessa *Sprint* foram identificados problemas referentes à dependência da realização de tarefas de outros módulos para o desenvolvimento das histórias. Um caso a citar, foi a necessidade de *merge* dos times dos módulos *Core* e *Bug Tracking*, para possibilitar o desenvolvimento das histórias dessa s*print*.

Na s*print* 4 foi acordado com o *PO* o desenvolvimento de três histórias: 1 - Exibir contatos: tem como objetivo listar os usuários do *FireScrum* por produto, para que o usuário logado no *Desktop Agent* possa iniciar o chat com cada contato, sendo estes classificados em *online* e *offline*, os quais foram controlados utilizando o servidor multimídia *Red5*; 2 - Trocar o produto que o usuário está logado: permite ao usuário realizar suas atividades em diversos produtos em que esteja alocado, bastando somente a troca de produto; 3 - Preparar a arquitetura do *chat*: objetivando a criação de diagramas de seqüencia e a interação da comunicação entre as partes envolvidas no *chat*.

Nessa s*print* foram realizadas cinqüenta e oito tarefas e criadas mais dezesseis. Tornando-se um grande desafio por ser o primeiro contato com a tecnologia *Red* 5. Esta tecnologia foi utilizada para fazer o controle de quais usuários estariam ou não logados no *FireScrum*. O servidor Red5 é um servidor de código aberto dedicado à interação entre o *Adobe Flash Player* e um servidor gratuito orientado à conexão utilizando RTMP (*Real Time Message Protocol*). Ao longo dessa *sprint* foram identificadas poucas dificuldades, pois o time se encontrava mais maduro e auto-gerenciável. No entanto, a adoção da nova tecnologia demandou tempo de estudo e implementação de acordo com as necessidades do módulo *Desktop Agent*. Além disso, na negociação inicial com o *PO* foram definidas três histórias, onde as tarefas já estavam sendo realizadas, porém houve a necessidade de abortar uma das histórias, implicando em uma reorganização do time para não afetar a conclusão das demais tarefas. O desempenho do time nessa sprint pode ser visualizado por meio da Figura 2.

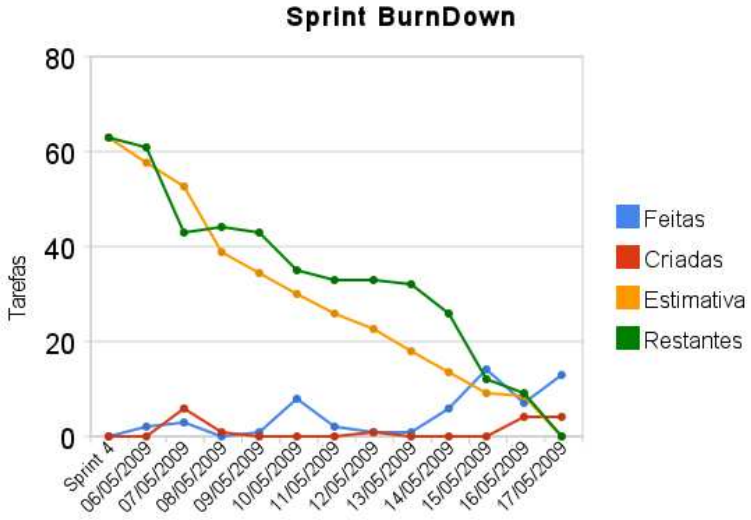


Figura 2. BurnDown sprint 4.

Na s*print* 5 foi acordado com o *PO* o desenvolvimento de quatro histórias: 1 - Abrir uma conversa de bate papo com algum contato: objetivando escolher outro usuário que esteja logado pelo *Desktop Agent* para iniciar o chat; 2 - Adicionar funcionalidades na tela de reportar *bug*: selecionar em qual história e a quem, o *bug* estava associado, o *deadline* do *bug* e o seu tempo de resolução; 3 - Ver os contatos separados por produto: objetivando visualizar os contatos de acordo com os produtos existentes; 4 - Configurar o endereço de acesso aos servidores: permitindo ao usuário a opção de colocar os endereços do servidor *FiresScrum* e *Red5* que foram utilizados.

Nessa penúltima s*print* foram realizadas quarenta e seis tarefas, nas quais o time obteve muita dificuldade, ocasionando no abortamento de uma história. Essa

história era referente à melhoria da qualidade do código e foi abortada em função dos seguintes problemas: o módulo *Desktop Agent* inerentemente dependia, significativamente, de funcionalidades de outros módulos e a página do *SourceForge* se encontrava indisponível. Portanto, dois dias de trabalho foram perdidos, conseqüentemente, causaram atrasos na execução das tarefas.

Para finalizar não ocorreu oficialmente a *sprint* 6, pois ao finalizar a s*print* anterior, todos os times de forma organizada fariam os *merges*, para, em seguida, serem realizados testes e, conseqüentemente, os *bugs* identificados serem corrigidos. Nessa s*print* o time do referido módulo foi o último a realizar os *merges*, permitindo a conclusão de algumas tarefas que foram abortadas na s*print* passada. Além disso, foram realizadas correções de CRs e após a concretização do *merge*, algumas modificações foram discutidas com o Comitê de Controle de Mudanças para serem implementadas. Após a finalização das 6 *sprints* ,o produto final, módulo Desktop *Agent*, foi apresentado ao *PO* e aos professores da disciplina.

6. Considerações Finais e Trabalhos Futuros

Este artigo apresentou a experiência de uma equipe formada por estudantes e profissionais de TI na concepção de uma Fábrica de *Software*, utilizando o método ágil *Scrum* como base para o desenvolvimento do produto para um cliente, simulando um ambiente real com prazos, cronogramas, entregas e acordos entre os membros da fábrica e o cliente.

A principal contribuição deste artigo está em uma detalhada descrição da vivência de uma fábrica de software visando os aspectos positivos e negativos no desenvolvimento do software utilizando a metodologia *Scrum*.

O módulo, Desktop *Agent*, componente desenvolvido para a ferramenta *FireScrum* apresenta resultados, até o momento, considerados satisfatórios. Acredita-se que os ganhos conseguidos com o módulo, podem ser, efetivamente, superiores com o amadurecimento dos processos e da visão do time. As dificuldades encontradas na sua implantação mostraram-se superáveis e inerentes a qualquer mudança significativa na ferramenta.

Os resultados alcançados até o presente momento propiciam expectativas de trabalhos futuros, onde se pretende dar continuidade ao projeto *FireScrum* a fim de realizar alguns ajustes e refiná-los, visando sempre a otimização do projeto, bem com sua melhoria contínua e difusão para empresas ou modelos de negócio.

Referências

AUDY, J.; PRIKLADNICKI, R. Desenvolvimento Distribuído de Software: Desenvolvimento de software com equipes distribuídas. Rio de Janeiro: Elsevier, 2007.

BAROFF, L. E. Distributed teaming on JPL projects. IEE Aerospace Conference, vol 7, p. 3461- 3466, 2002.

BEEDLE, M. "*Agile Development with Scrum*". 3 ed, Addison Wesley Press, 2004.

BOEHM, B. "*A View of 20th and 21st Century Software Engineering*". ICSE 2006.

DAMIAN, D.; MOITRA, D. "Guest Editors' Introduction: Global Software Development: How far Have We Come?".IEEE Software, 2006. 23(5), pp.1719.

EBERT, C. et al. Managing Risks in Global Software Engineering: Principles and Practices. Proc. IEEE International Conference on Global Software Engineering ICGSE 2008, p. 131-140, 2008.

FREITAS, A. V. APSEE-Global: a Model of Processes Management of Distributed Software Processes. Faculdade de Informática – UFRS – RS- Brazil, 2005.

GLOGER, B. "*The Zen of Scrum*". Disponível em: http://www.glogerconsulting.com. Acesso: jan. de 2010.

HERBSLEB, J. D.; MOITRA, D. "*Global Software Development*". IEEE Software Magazine, IEEE Computer Society, EUA, 2001.

HIGHSMITH, J. "*Agile Project Management – Creating Innovative Products*". AddisonWesley Press, 2004.

LIANG, H. "*Distributed Software Development*". Leading Edge Forum. CSC Experience e Results, 2008.

LUNA, A. et al. Desenvolvimento Distribuído de uma Aplicação de Telessaúde com a Metodologia Ágil SCRUM. X Congresso Brasileiro de Informática em Saúde. São Paulo, 2008.

MACGREGOR, E. et al. "*The impact of intercultural factors on global software development*". Proc. Canadian Conference on Electrical and Computer Engineering, p. 920-926, 2005.

PRIKLADNICKI, R. MuNDDoS - Um modelo de referência para desenvolvimento distribuído de software. Dissertação de Mestrado, PUC/RS, Porto Alegre, RS, Brasil, 2003.

SCHWABER, K. et al. "*Scrum Development Process*", OOPSLA Business Object Design and Implementation Workshop, Eds. London: Springer. 1997.

SCHWABER, K. "*Agile Project Management With Scrum*". Microsoft, 2004.

SENGUPTA, B.; CHANDRA, S.; SINHA, V. "*A Research Agenda for Distributed Sofwtare Development*". In: 28th ICSE, Xangai, China.

SOARES, F. et al. Adoção de Scrum em uma Fábrica de Desenvolvimento Distribuído de Software. I Workshop de Desenvolvimento Distribuído de Software. João Pessoa, 2007.

SUTHERLAND, J.; VIKTOROV, A.; BLOUNT, J.; PUNTIKOV, N. *"Distributed Scrum: Agile Project Management with Outsourced Development Team,"* HICSS'40, Hawaii International Conference on Software Systems, Big Island, Hawaii, 2007.

TAKEUCHI, H.; NONAKA, I. *"The New New Product Development Game".* Harvard Business Review, 1986. Janeiro-Fevereiro, pp137-146.

VERSIONONE. "*The State of Agile Development Survey Results*". Disponível em: http://www.versionone.com/pdf/3rdAnnualStateOfAgile_FullDataReport.pdf. Acesso em: maio de 2010.

ZANONI, R. Modelo de Gerência de Projeto Baseado no PMI para ambientes de Desenvolvimento de Software Fisicamente Distribuído. Dissertação de Mestrado, PUC/RS, Porto Alegre, RS, Brasil, 2002.

Agilidade na UFABC - Implantação da Metodologia *Scrum* na Divisão de Desenvolvimento

Maurício Eduardo Szabo[1], Christiane Marie Schweitzer[2]

[1] Núcleo de Tecnologia da Informação (NTI) - Universidade Federal do ABC (UFABC) - Santo André - SP - Brasil

[2] Centro de Matemática, Computação e Cognição (CMCC) – Universidade Federal do ABC (UFABC) – Santo André – SP - Brasil

mauricio.szabo@gmail.com, chris@ufabc.edu.br

Abstract. *The adoption of agile software development methodologies breaks rigid paradigms in institutions, bringing innovation, flexibility and new challenges to software development process. This paper presents Scrum implementation for software development in Federal University of ABC (UFABC), describing used Scrum practices, their evolution, and pointing successes, errors and solutions.*

Resumo. *A implantação de metodologias ágeis de desenvolvimento de software quebra paradigmas antes estruturados e rígidos dentro de uma instituição, trazendo inovação, desafios, e flexibilidade a todo o processo de desenvolvimento. Este artigo tem por objetivo apresentar a implantação da metodologia Scrum no desenvolvimento de sistemas na Universidade Federal do ABC (UFABC), identificando processos da metodologia implantados, a evolução, acertos, erros, e soluções de contorno.*

1. Introdução

Criada em 2005 com o intuito de ser uma "Universidade de Ponta para o Século XXI", a Universidade Federal do ABC (UFABC) oferece cursos nas áreas tecnológica e científica. A UFABC apresenta uma estrutura curricular diferenciada, de forma a atender as novas demandas que o profissional se depara tanto em aspectos sociais, quanto tecnológicos e econômicos que caracterizam o mundo moderno. Sobretudo, esta proposta pedagógica visa tornar os acadêmicos capazes de enfrentar novos problemas, com confiança nas suas potencialidades e demonstrando capacidade de investigação e inovação. A partir destes princípios norteadores, a proposta pedagógica da UFABC visa:

- Ampliar o currículo básico em extensão e profundidade no que diz respeito à Informática, Computação Científica, às Ciências Naturais, às Ciências de Engenharia e à Matemática
- Estruturar o currículo profissional, de modo a atender as demandas das tecnologias modernas e emergentes e incorporar disciplinas que permitam uma inserção mais rápida dos formandos na sociedade moderna

- Incorporar disciplinas como a História da Ciência, História da Tecnologia e História do Pensamento Contemporâneo com o intuito de desenvolver a capacidade crítica no exercício da atividade profissional e da cidadania
- Estimular e desenvolver nos estudantes as habilidades de descobrir, inventar e sistematizar, características respectivamente das Ciências Naturais, das Engenharias e das Matemáticas
- Individualizar, ainda que parcialmente, o currículo de modo que o aluno possa desenhar sua formação profissionalizante de acordo com sua vocação e suas aspirações.

Objetivando a operacionalização da proposta pedagógica, a estrutura curricular divide-se em duas etapas: (i) ciclos iniciais de três anos, conduzindo ao Bacharelado em Ciência e Tecnologia (BC&T), (ii) ciclo complementar de um ano para licenciatura ou bacharelado específico (Física, Química, Matemática, Computação, Biologia) ou ciclo profissional de dois anos para Engenharia.

Desta forma, através do ciclo básico o aluno tem condições de moldar seu perfil profissional de acordo com suas aptidões e preferências através de um conjunto disciplinas tanto do BC&T quanto já direcionadas ao ciclo complementar ou profissional.

Além disso, procurando evitar uma separação muito grande entre áreas de conhecimento, a UFABC não se organiza em um modelo de divisão departamental. A eliminação de departamentos é um avanço que permite uma interlocução permanente entre os docentes e discentes trabalhando em uma forma interdisciplinar. Sendo assim organizou-se a UFABC em três grandes centros, quais sejam: Centro de Ciências Naturais e Humanas, Centro de Matemática, Computação e Cognição e Centro de Engenharia, Modelagem e Ciências Sociais Aplicadas.

A estrutura curricular do curso de BC&T da UFABC está organizada em torno de seis eixos principais: Estrutura da Matéria, Processos de Transformação, Energia, Comunicação e Informação, Representação e Simulação, e por fim Humanidades.

Para viabilizar esta proposta inovadora a UFABC recebeu seus primeiros servidores, técnicos administrativos, concursados em 2006. Inicialmente, na área de Tecnologia da Informação (TI), foram contratados dois Analistas de TI e dois Técnicos de Laboratório -Área Computação, e mais tarde (início de 2007) mais um Analista de TI e um Técnico. Até início de 2009, nenhum outro funcionário da área de TI foi contratado o que impedia a criação de uma divisão de desenvolvimento de software durante os primeiros anos de exercício da UFABC. É importante ressaltar que as demandas de soluções automatizadas de software para os setores administrativos e acadêmicos eram contínuas e crescentes.

Assim, em meados de 2009, com a contratação dos novos servidores da área de TI, foi possível a implantação de uma divisão de desenvolvimento, criando uma equipe coesa e dedicada ao estudo de metodologias para dar início ao desenvolvimento de aplicações para os setores da UFABC, de forma rápida e eficiente, a fim de atender as demandas engavetadas.

Para isto, a equipe deparou-se com diversos desafios: como vencer o atraso de quase dois anos e meio, ainda assim garantindo uma qualidade de serviço? Como criar

uma divisão de desenvolvimento de software para uma Universidade de Ponta para o Século XXI? A resposta parecia apontar para o pragmatismo e estudo de metodologias ágeis. Desta forma, o presente artigo tem por objetivo apresentar a experiência da UFABC no processo de investigação e implantação de metodologias ágeis em seus processos de desenvolvimento, identificando seus erros e acertos.

2. Metodologias

Desde o artigo de Royce [Royce, 1970] até os tempos recentes, metodologias têm passado por diversas transformações ou modificações. Com o intuito de qualificar o "nível de maturidade" que uma empresa ou instituição possui no desenvolvimento de software, foram criados modelos de capacitação e qualidade, tais como o o ISO/IEC 9126 e o CMM/CMMI [COUTO, 2007]. Porém, conforme cita Schwaber [Schwaber, 2004], modelos como o CMM, se não forem corretamente implantados, podem aumentar a burocracia interna de uma instituição, o que implica naturalmente em atrasos na entrega dos projetos. Como uma resposta às metodologias mais tradicionais, foi publicado em 2001 o Manifesto Ágil [MANIFESTO, 2001], e em seu site oficial, é possível encontrar a seguinte frase:

*"Estamos descobrindo melhores maneiras de desenvolver software **fazendo-as** e ajudando outros a fazê-lo." (Grifo meu)*

O grifo acima demonstra claramente o pragmatismo inerente às metodologias ágeis, um pragmatismo que parece em conformidade com a proposta inovadora da UFABC, em ser uma instituição que não se guia pelos padrões de outras universidades (principalmente na estrutura de cursos). Este pragmatismo e adaptabilidade também está presente nas metodologias *Scrum* e *Extreme Programming*, nas quais o processo de desenvolvimento (e gerência) está constantemente em alteração e evolução, de acordo com a experiência da equipe como um todo. Além disso, metodologias ágeis estão constantemente preocupando-se em entregar, após um período curto de tempo, *software* que já pode ser implantado e utilizado, e no futuro estender as suas funcionalidades, para que se evite um levantamento prévio de requisitos muito extenso que provavelmente será completamente alterado.

Antes de decidir qual seria o caminho da implantação de uma metodologia ágil, foi feito um levantamento das principais práticas das metodologias *Extreme Programming* [Wells 2009] e *Scrum* [Schwaber 2004].

2.1. Extreme Programming [Wells, 2009]

A metodologia *Extreme Programming* foca-se não apenas na gerência do desenvolvimento, mas também na codificação do software. Esta metodologia possui diversas definições sobre como um código deve ser escrito, quais as melhores práticas de escrevê-lo, e define também algumas reuniões que devem ser feitas. As principais práticas do *Extreme Programming* são:

1. Durante o levantamento de requisitos, são escritas *user stories*. Semelhantes às definições formais de requisitos, *user stories* diferenciam-se principalmente pela facilidade de lê-las e pela vantagem de serem escritas na linguagem do usuário.
2. Todos os dias, antes do início de qualquer atividade, faz-se uma *stand-up meeting*. Este tipo de reunião é semelhante à reunião apresentada pela

metodologia *Scrum*, sendo uma reunião executada em pé (para evitar demoras) e cujo único objetivo é promover comunicação entre os desenvolvedores.

3. Toda a codificação do software é feita em pares. Dois programadores trabalham juntos, no mesmo computador, para evitar cansaço excessivo de apenas uma pessoa, e também para identificar erros mais cedo (seja em lógica ou em codificação propriamente dita).
4. Todo código deve estar coberto por testes automatizados, de preferência escrevendo o teste antes do código.
5. Quando um erro for encontrado no software, é necessário escrever um teste para que o erro não recorra em versões futuras Assim como em outras metodologias ágeis, as pessoas que solicitaram o desenvolvimento do sistema devem estar sempre disponíveis e presentes.

Adicionalmente, *Extreme Programming* promove certos conceitos como *planning poker,* uma forma de estimar prazos usando cartas numeradas, alguns conceitos como *story points*, uma medida de complexidade de uma tarefa, usada na hora de estimar um prazo, e outros conceitos que, estando fora do escopo deste artigo, não serão a seguir plenamente descritos.

2.2. Scrum

A metodologia *Scrum* é mais voltada para a gerência do desenvolvimento do software ao invés das práticas. Criada por Ken Schwaber e Jeff Sutherland, o propósito é ser uma metodologia simples, de poucas regras, porém muita mudança de paradigma. *Scrum* define mais reuniões e planejamento do que a metodologia *Extreme Programming*, não focando na codificação do software propriamente dito. Esta metodologia também diferencia claramente as pessoas interessadas em um projeto e as comprometidas no mesmo. Para as pessoas comprometidas, há três papéis:

1. ***Product Owner:*** na metodologia *Scrum*, é responsabilidade do *Product Owner* controlar quais funcionalidades o sistema possuirá, quais são as prioridades, acompanhar e guiar o desenvolvimento do sistema e informar ao *Scrum Master* todas as alterações que ocorreram nos requisitos. É sua responsabilidade também manter atualizado o *Product Backlog*, uma lista de funcionalidades que o sistema deve possuir, cada uma com suas prioridades.
2. ***Scrum Master:*** semelhante ao gerente de projetos, o *Scrum Master* deve reportar-se constantemente ao *Product Owner,* apresentar as dificuldades que a equipe possui e também as alterações feitas nos planos (caso haja alguma). Também suas responsabilidades estão em marcar as reuniões previstas pelo *Scrum*, bem como garantir que todos estejam seguindo as regras evitando assim interrupções no desenvolvimento.
3. ***Team:*** Todos os desenvolvedores do software estão presentes neste papel. A responsabilidade da Equipe é se auto-organizar, definir melhores práticas, reportar-se um ao outro durante o desenvolvimento do sistema, e também apresentar ao *Product Owner* o resultado do trabalho, quando for apropriado.

Scrum pede algumas reuniões, tanto para comunicação entre os membros da Equipe quanto para o planejamento do desenvolvimento. Há também reuniões para apresentar o

que foi desenvolvido e para melhorar o próprio modelo de trabalho. As reuniões são as que seguem:

1. ***Sprint Meeting:*** toda a codificação é dividida em períodos de trinta dias corridos chamados *sprints* (Figura 1, Iteration). No início de cada *sprint*, é feita uma reunião com o *Product Owner* para priorizar os itens mais importantes do *Product Backlog*.
2. ***Scrum Daily:*** durante um *sprint*, diariamente e sempre no mesmo horário é feita uma reunião de, no máximo, trinta minutos no qual o *Scrum Master* pergunta aos membros da Equipe o que cada um fez entre o *Scrum Daily* anterior e o atual, o que cada um pretende fazer no dia, e se há alguma dificuldade que impede a continuidade do trabalho (Figura 1, 24-hour inspection).
3. ***Sprint Review:*** após o término do *sprint*, é feita uma reunião para que a equipe apresente ao *Product Owner* tudo o que foi desenvolvido durante o período.
4. ***Sprint Retrospective:*** logo depois do *Sprint Review,* é feita uma reunião na qual a Equipe, junto com o *Scrum Master*, discute melhores maneiras para melhorar o processo de desenvolvimento.

Este ciclo é repetido até o desenvolvimento completo do sistema. A figura 1 mostra o ciclo de desenvolvimento do *Scrum*:

Figura 1: Ciclo de desenvolvimento da metodologia *Scrum* [Schwaber, 2004]

3. Idéia e Filosofia

Após o estudo das metodologias ágeis, foi feito um pequeno levantamento do que se esperava de uma divisão de desenvolvimento, para priorizar aspectos da implantação de uma metodologia ágil sobre a outra. Foram consultados diversos artigos, porém um deles foi mais incisivo sobre a idéia da divisão, o artigo "Provocações", do professor Xéxeo [Xéxeo, 2009]. Entre todos os tópicos do artigo, vale citar os seguintes:

1. Pensamos em computadores e algoritmos, PORÉM nossos egressos trabalham com pessoas e processos.
2. *Pensamos em preparar pessoas especializadas capazes de aplicações como se detivessem um segredo do universo, programadores míticos, PORÉM qualquer um pode gerar aplicações, basta um pouco de vontade, uma planilha eletrônica ou outros super-meta-aplicativos fáceis de usar, como o Access.*

3. *Pensamos em linguagens de programação do tipo "bondage and discipline", PORÉM linguagens libertárias, como Python e Ruby nos provam que podemos ser mais produtivos de forma totalmente diferente.*
4. *Pensamos em equipes de trabalho pré-organizadas, PORÉM o desenvolvimento de software livre e mesmo nas empresas é cada vez mais auto-organizado.*

Após a leitura de outros artigos sobre o assunto, todos os membros da divisão de desenvolvimento sentaram-se para definir um conjunto de idéias sobre qual seriam as responsabilidades da divisão, bem como um conjunto de preceitos sob os quais quaisquer decisões futuras se baseariam, e os dois principais (e que mais estão relacionados com a implantação da metodologia) foram:

Não tratar a divisão como uma "fábrica de software": este conceito foi extremamente importante para indicar quais seriam as práticas a serem adotadas. Por exemplo, a prática do *pair programming*, como pede o *Extreme Programming*, exige uma rigidez que num primeiro momento não era desejada. Além disso, seria necessário atentar-se para uma metodologia que permitisse que todos os envolvidos, caso desejassem, pudessem pesquisar assuntos diversos e implantá-los. Tal preceito foi baseado principalmente no primeiro e quarto itens, do artigo acima.

Desenvolvimento de sistemas deselitizado: permitir que futuros usuários e pessoas interessadas no desenvolvimento acompanhem todo o projeto, desde seu início, sem a necessidade de regras formais. É responsabilidade da Equipe ouviras idéias do usuário e evitar jargões ou diagramas complexos. Por exemplo, caso fosse necessário apresentar um diagrama para um usuário, o ideal seria que o diagrama fosse "traduzido" em uma linguagem mais simples para ser apresentado. Tal pensamento foi principalmente baseado no segundo item do artigo acima.

Além destes preceitos, havia também a necessidade de manter claro para todos os envolvidos em desenvolvimento de *software*, tanto da divisão de desenvolvimento como o restante da universidade, qual o auxílio que o setor de TI da universidade pode oferecer no desenvolvimento de sistemas, e até onde estes sistemas poderiam auxiliar tanto no controle interno das atividades como na tomada de decisões. Para tal, a equipe comprometeu-se a evitar determinadas pré-concepções comuns entre os desenvolvedores, como a afirmação de que o usuário nunca sabe o que deseja. Portanto, seria atribuição, também, da equipe de desenvolvimento auxiliar o demandante de um sistema a organizar as idéias e pensamentos sobre o que um sistema deveria fazer, ao invés de reclamar da falta de definições ou da ausência de informações. Além disso, foram definidas as seguintes idéias, que dizem especificamente a respeito do *software* a ser desenvolvido:

Não "frear" a universidade: o *software* não pode ser um fator decisivo para evitar a inovação prevista, e sim ser dinâmico para acomodá-la.

Apresentar falhas, mas não impor condições: se for identificado algo que não daria certo no desenvolvimento de um aplicativo (tais como exceções constantes de regras), apresentar o problema, propor soluções e confiar que os comprometidos com o projeto definirão a melhor forma de solucioná-lo.

Solucionar apenas os problemas que foram apresentados: quando uma dificuldade for apresentada, ver se existe uma solução que atende previamente o problema. Se for necessário um desenvolvimento, fazer um levantamento de requisitos e

codificar apenas o que foi pedido, atentando-se para as prioridades. Nos métodos ágeis, é comum a utilização do termo *YAGNI* (do inglês *You Ain't Gonna Need It*, ou "você não vai precisar disto").

Aprender a conviver com as mudanças, ao invés de reclamar delas: um fato presente em metodologias ágeis e no próprio projeto pedagógico da UFABC é a mudança constante. Aceitar que as mudanças são naturais, e garantir um *software* que se adapte a elas. No caso, se um *software* tornar-se instável por mudanças excessivas, é necessário pesquisar melhores tecnologias ou práticas (possivelmente de testes automatizados) para manter o produto final flexível.

Software como "arte" e "ferramenta": uma das principais idéias que o desenvolvimento precisa manter clara é a importância de manter-se um código limpo, claro, e sem construções "obscuras" nem "soluções paliativas", principalmente para facilitar a manutenção futura. Uma frase que ainda é muito citada na divisão é: "jamais escreva um código que você terá vergonha de mostrar para alguém". Entretanto, é necessário também entender que o produto final do desenvolvimento, embora tenha que atender uma série de critérios, é apenas uma ferramenta para outra pessoa-portanto, se o *software* não atender as demandas e necessidades do outro, ele não terá utilidade, independente de estar estável, bem codificado e testado.

4. Implantação de Scrum na UFABC

No primeiro semestre de 2009, novos funcionários (Técnicos e Analistas) oficialmente entraram em exercício, e conseqüentemente foi possível a criação de um setor de desenvolvimento de sistemas. Com o apoio da chefia, os pedidos de projetos foram anotados, porém nenhum projeto começou a ser desenvolvido até que todos os funcionários recebessem treinamento na tecnologia adotada na UFABC (linguagem de programação *Ruby* e framework *Rails*, além de práticas de teste automatizado). Durante um mês, foram feitos treinamentos e, em paralelo, foram escolhidos aspectos das metodologias que seriam adotadas. Decidiu-se que seria implantado *Scrum*, e algumas práticas do *Extreme Programming* referentes aos testes automatizados.

Logo no primeiro projeto, foi definido que atentaríamos para algumas regras do *Scrum*. Num primeiro momento, entretanto, as reuniões do *Scrum Daily* eram executadas a cada três dias, e as outras reuniões previstas pela metodologia não foram adotadas. Do *Extreme Programming*, era desejado que se obedecesse à prática de escrever os testes antes do código (*Test-Driven Development*), evitando-se deixar qualquer fragmento do código-fonte sem estar coberto por algum teste, porém devido à pouca experiência da equipe com testes automatizados, nos primeiros projetos isto foi apenas parcialmente atendido. Outros aspectos do *Scrum*, tais como o papel do *Product Owner* e *Scrum Master* não estavam claros para a equipe de desenvolvimento, demonstrando que a maior parte da metodologia *Scrum* não havia sido implantada. Porém, conforme os projetos foram sendo desenvolvidos, mais aspectos de ambas as metodologias foram sendo implantados, até consolidarem-se nas práticas que temos hoje, apresentadas mais à frente.

4.1. Projetos de Implantação

Este estudo de caso está dividido cronologicamente por projeto. É feita uma breve descrição dos problemas enfrentados em cada um (relativos apenas à metodologia) e são apresentadas as soluções adotadas e implantadas.

4.1.1. Primeiro Projeto - Melhoria na Definição de Requisitos

Necessidade: automatizar o processo de avaliação em estágio probatório.

Demandante: setor de Recursos Humanos.

Problemas enfrentados: falha na definição das funcionalidades do sistema (chamadas de *user-stories* nos métodos ágeis). Na metodologia *Scrum*, as descrições de cada funcionalidade devem ser escritas na linguagem mais próxima possível do usuário. Tais descrições são chamadas de *user-stories*, e consistem em uma linguagem sucintal, tal como "o sistema deve permitir um cadastro de usuários". Porém, no projeto desenvolvido, as tarefas foram descritas em uma linguagem que facilitaria a transformação das mesmas em testes de aceitação, tais como "na tela de cadastros, deve haver o campo 'nome', 'endereco', 'telefone'...". Quando estas descrições de funcionalidade foram repassadas ao *Product Owner*, ele não conseguiu identificar as falhas, e alguns problemas emergiram devido à mudança de rumo no desenvolvimento do sistema no meio de um *sprint*.

Soluções implantadas: melhores maneiras de descrever as funcionalidades. Foi estudado o formato de *user-stories*, melhorado o formato do *Product Backlog* e também foram estudadas *wireframes,* formas de apresentar um esboço das telas para o usuário.

4.1.2. Segundo Projeto - Revisão do Papel do *Scrum Master*

Necessidade: criar um formulário de inscrição em concurso público.

Demandante: setor de Recursos Humanos.

Envolvidos no projeto: setor de Finanças.

Problemas enfrentados: falha grave de comunicação entre os dois setores envolvidos no desenvolvimento, que resultou em um atrito entre um dos funcionários do desenvolvimento com o de outro setor. A maior falha, porém, consistiu em não se obedecer às regras do *Scrum* no que diz respeito ao papel do *Scrum Master*, no qual ele é o único responsável em manter o canal de comunicação entre a equipe e todos os agentes externos a ela.

Solução implantada: revisão do papel do *Scrum Master*. A partir deste projeto qualquer contato com a Equipe deveria passar pelo *Scrum Master*, para evitar atritos futuros.

Melhoria implantada: adotação do *Google Spreadsheets* como forma de manter o *Product Backlog* disponível para todas as pessoas interessadas no projeto ou comprometidas com ele.

4.1.3. Terceiro Projeto - Adoção de *Test-Driven Development*

Necessidade: desenvolvimento de um sistema de matrícula em disciplinas.

Demandante: Pró-reitoria de Graduação.

Problema enfrentado: repentina alteração no sistema de matrículas, próximo do prazo de entrega. No último período de matrícula em disciplinas, cada disciplina era separada em um determinado grupo de horários, cada um contendo os dias da semana e horários de início e fim de cada aula. Este tipo de separação se dava devido ao fato que a matrícula era feita de forma manual, então garantia-se que os grupos jamais se chocariam uns com os outros. Porém durante o desenvolvimento do sistema, foi feita uma alteração brusca, onde a idéia de grupo de horários foi abolida e cada disciplina recebeu seus próprios horários, independentemente das outras. Tal mudança foi efetuada faltando poucas semanas para a matrícula dos alunos, e representou uma alteração em aproximadamente um terço do sistema.

Melhoria implantada: seguir rigorosamente a prática de *Test-Driven Development.* Durante o desenvolvimento deste sistema, foi adotado *Test-Driven Development*, num primeiro momento apenas como meio de estudo. Graças a esta prática a alteração foi efetuada em apenas seis horas corridas, e o prazo de entrega foi atendido. Os projetos seguintes, então, passaram a adotar rigorosamente a prática de *Test-Driven Development*, e também foi adotada a prática de escrita do modelo entidade-relacionamento, como forma de documentação da base de dados.

4.1.4. Quarto Projeto - Apresentação ao Usuário das Regras do *Scrum*

Necessidade: criação de um questionário socioeconômico para os alunos, integrado com o sistema de matrícula.

Demandante: setor de Planejamento.

Problemas enfrentados: dificuldades para o usuário se reportar a apenas uma pessoa. Neste projeto, o *Product Owner* estava se dirigindo a vários membros da Equipe, quebrando a regra do *Scrum* que pede que, durante um *Sprint*, somente o *Scrum Master* faça a intermediação entre a Equipe de trabalho e quaisquer outros envolvidos no desenvolvimento.

Soluções implantadas: limitar o contato do usuário ao *Scrum Master*. Como o *Product Owner* evitava se reportar apenas ao *Scrum Master,* mesmo após alguns pedidos, a Equipe foi instruída a encaminhar todos os e-mails que chegavam pedindo informações sobre o projeto ao *Scrum Master*, indicando ao remetente que estavam repassando a correspondência para a pessoa competente.

Melhorias implantadas: otimização das reuniões e instalação de um quadro-branco. Com o objetivo de manter a Equipe a par do que cada um de seus membros fazia, foi definido que a reunião *Scrum Daily* seria executada a cada vinte e quatro horas. Também foi implantada a reunião *Sprint Retrospective*. Além disso, houve a instalação de um quadro-branco que facilitou a comunicação entre todos os desenvolvedores. Com a instalação do quadro, algumas práticas começaram a emergir, tais como escrever fragmentos e código que outros poderiam aproveitar, ou mesmo deixar o modelo entidade-relacionamento (ou um fragmento dele, referente àquele *sprint*) desenhado apenas no quadro, e depois passado a limpo.

4.1.5. Quinto Projeto - Melhoria na Comunicação

Necessidade: desenvolvimento de um sistema acadêmico, para atender as demandas específicas da universidade.

Demandante: Pró-Reitoria de Graduação.

Problemas enfrentados: indefinições nos requisitos. Durante o desenvolvimento deste sistema, havia muitas indefinições que precisavam de uma resposta. Infelizmente, o *Product Owner* não se apresentava muito presente no desenvolvimento, e embora esta mesma situação tivesse ocorrido nos outros sistemas, devido à complexidade deste projeto o problema foi exposto. Além disso, depois do primeiro *Sprint,* ficou claro que os envolvidos com o projeto não sabiam quanto do sistema estava desenvolvido, ou mesmo em que estado ele se encontrava.

Soluções implantadas: implantação das reuniões *Sprint Meeting* e *Sprint Review*, e desta forma todas as reuniões previstas pelo *Scrum* passaram a ser marcadas por projeto. Foram feitas também tentativas de aproximar mais o *Product Owner* do desenvolvimento, e também e-mails com questionamentos passaram a ser mandados com cópia para outras pessoas que possuíam conhecimento dos processos. Por fim, foi instalada uma Wiki para a equipe de desenvolvimento e os envolvidos em cada projeto, contendo manuais dos sistemas, documentações, e manuais de boas-práticas.

4.1.6. Sexto Projeto - Esboço do *Q/A*

Necessidade: criar um formulário para os docentes com perguntas sobre quais disciplinas cada um está apto a ministrar.

Demandante: comitê de Alocação Didática.

Envolvidos no projeto: Pró-Reitoria de Graduação.

Problemas enfrentados: queda na qualidade do código-fonte, além de erros introduzidos no código devido à falta de testes de aceitação. Neste projeto, devido ao prazo apertado, muitas das boas-práticas e convenções de sintaxe acabaram sendo desobedecidas, bem como o uso de *Test-Driven Development*. Adicionalmente, a ausência dos testes de aceitação (*Acceptance Tests* no *Extreme Programming*, também chamados de testes funcionais ou testes de caixa-preta) começaram a expor alguns *bugs*.

Soluções implantadas: estudos de bibliotecas de teste que pudessem facilitar a criação de testes de aceitação, e revisão de algumas políticas de teste. Também, foi criado um processo denominado *Q/A* (do inglês *Quality Assurance*), que consiste em revisar cada tarefa, a fim de determinar se ela segue os padrões de qualidade do código-fonte.

4.1.7. Sétimo Projeto - Maior Participação do *Product Owner*

Necessidade: desenvolver um formulário para cadastramento de reagentes controlados.

Demandante: setor de Reagentes e Descartes.

Envolvidos no projeto: laboratórios de Química.

Soluções implantadas: formalizações no processo de *Q/*A. Este projeto, por ser bem simples, não apresentou problemas no desenvolvimento. Mesmo assim, melhorias foram implantadas, tais como formalizações no processo de *Q/A*, que consistiu em incluir na *wiki* uma lista de passos a serem vistoriados no momento de análise do código, bem como padrões de qualidade a serem seguidos. Durante este projeto também foi escrito um resumo de cada projeto desenvolvido, bem como um guia completo de utilização do sistema de controle de versão (para consulta pelos novos servidores). Durante o projeto, o *Product Owner* apresentou-se bem presente, comparecendo ao

setor uma vez por semana para acompanhar o desenvolvimento e respondendo dúvidas e questionamentos quase imediatamente.

4.2. Evolução dos Processos

Muitas alterações foram feitas desde o primeiro projeto da UFABC até o final do sétimo projeto. As principais mudanças desde o início da implantação até a situação hoje, são as seguintes:

1. Num primeiro momento, acreditava-se que as reuniões diárias pedidas pelo *Scrum* incorporariam rigidez desnecessária ao desenvolvimento, o que provou ser falso: tais reuniões facilitam o sincronismo da equipe. Em sistemas complexos, esta prática é indispensável para que todos os membros estejam trabalhando no mesmo ritmo.
2. A prática de escrever testes automatizados, especialmente antes do próprio código-fonte, é a maior (ou talvez, a única) garantia de que o *software* desenvolvido está e permanecerá estável. No processo de desenvolvimento ágil, onde não há divisão entre desenvolvedores, testadores, ou *designers*, o teste automatizado torna-se simplesmente indispensável, e a cobertura de código sempre deve*tá* ser muito alta (nos projetos desenvolvidos, a cobertura é sempre maior que noventa e cinco porcento das linhas de código).
3. No início da implantação, fora o *Scrum Daily,* nenhuma outra reunião era efetuada a não ser um levantamento preliminar de requisitos. Durante o desenvolvimento dos diversos projetos, notou-se que a regra do *Scrum* de que cada *Sprint* deve contar com uma reunião de planejamento (*Sprint Meeting*), com toda a equipe de desenvolvimento, permite uma percepção melhor do problema a ser resolvido e das práticas a serem adotadas.
4. Após a implantação da reunião *Sprint Retrospective* e instalação do quadro-branco, diversas mudanças emergiram e a qualidade do desenvolvimento no geral subiu consideravelmente. Tais melhoras se justificam especialmente pela maior comunicação entre a Equipe.
5. A prática de apresentar um esboço das telas ao usuário (*wireframes*) consolidou-se, em projetos futuros, como uma forma de capturar falhas no levantamento de requisitos. Em projetos futuros, ao apresentar as *wireframes* ao usuário notavam-se situações que não haviam sido pensadas, e que eram prontamente adicionadas ao *Product Backlog*.

4.3. Situação Hoje

Todo o desenvolvimento na Universidade Federal do ABC, hoje, atende as maiores práticas da metodologia Scrum. Os processos no desenvolvimento são:

1. Pedido do sistema, e um breve resumo com a montagem do *Product Backlog*. O *Product Backlog* deve ficar sempre visível a qualquer pessoa interessada.
2. Definição da Equipe que trabalhará no projeto, marcação do *Sprint Meeting*, priorização das tarefas no *Product Backlog*.
3. Durante o *Sprint Meeting*, são definidos até mesmo esboços de tela (chamados *wireframes*) para facilitar o entendimento tanto dos desenvolvedores quanto dos

demandantes do sistema. Normalmente, no esboço do *wireframe*, são identificadas novas tarefas que são incluídas no *Product Backlog*.

4. Durante o *Sprint*, às 9h 30min, todos os desenvolvedores devem atender ao *Scrum Daily*.
5. Cada desenvolvedor pode escolher qualquer tarefa, e deve marcar seu nome ou iniciais nela. A partir deste momento, a tarefa é considerada "Em andamento".
6. Uma tarefa é considerada "Aberta" quando não há ninguém trabalhando nela porém foi selecionada para compor o *Sprint*, "Em andamento" quando está sendo codificada por uma pessoa, "Em *Q/A*" quando a tarefa foi concluída porém ainda precisa passar pela vistoria, ou "Fechada" quando cluída e revisada.
7. Usa-se sempre o sistema de controle de versão durante o desenvolvimento. Além disso, não se pode subir a cópia de trabalho atual no sistema de controle de versão se houver testes falhando.
8. Todo o código deve estar coberto por testes. Escreve-se o teste antes de escrever o código.
9. O Modelo Entidade-Relacionamento é normalmente montado junto com a codificação, e seu esboço fica no quadro-branco. Após o término do *Sprint*, o modelo é passado a limpo e compõe a documentação auxiliar do *software*.
10. Toda definição de classe ou método deve conter um comentário antes, definindo o que aquele código faz (num formato semelhante ao *Javadoc*). Esta documentação "em código" também compõe a documentação do *software*.
11. O *Product Owner* é livre para entrar em contato em qualquer momento com o *Scrum Master* para pedir informações, e também pode participar do *Scrum Daily*.
12. Após o término do *Sprint*, faz-se uma reunião para demonstrar o que foi desenvolvido (*Sprint Review*). Caso o *Product Owner* deseje, o software pode ser publicado.
13. Logo após o *Sprint* Review, faz-se a reunião *Sprint Retrospective*, e analisa-se o que deve ser melhorado. Normalmente, há dois ou três dias para pesquisar uma melhora, e então planeja-se outro *Sprint Meeting*.

Como é possível ver, houve uma grande evolução no processo de desenvolvimento, embora ainda existam muitos aspectos que possam evoluir. Outros projetos, feitos em paralelo, também auxiliaram muitas tomadas de decisão que permitiram uma evolução constante do processo e da metodologia.

5. Considerações Finais

Desde o início, a divisão de desenvolvimento de software foi montada sob uma ótica pragmática e orientada de acordo com a própria filosofia da universidade. Este fator, principalmente, auxiliou muito a implantação de *Scrum* e o início do desenvolvimento de sistemas que garantissem padrões de qualidade e atendessem a universidade. Outro fator extremamente auxiliador foi o apoio da chefia do Núcleo de Tecnologia da Informação, auxiliando sempre que possível para que as regras de *Scrum* fossem bem atendidas e não houvesse informações cruzadas.

É possível também perceber que, desde o início até o momento presente, houve uma grande evolução na metodologia, o que resultou em grandes avanços na qualidade do *software* entregue. Também, todos os projetos desenvolvidos tiveram seus prazos cumpridos, apesar das dificuldades inerentes de cada um.

Por fim, *Scrum* é uma metodologia que sempre está em evolução. Na divisão de desenvolvimento, praticamente todos os aspectos inicialmente implantados foram revisados, e alguns estão passando por sua segunda revisão. A cada *Sprint*, seja ele no mesmo projeto ou em projetos diferentes, há uma nova mudança a ser implantada, ou algum aspecto a ser revisado-e isto torna a metodologia dinâmica, havendo sempre uma alteração com o intuito de atender demandas cada vez mais específicas.

Na experiência da UFABC, as poucas regras da metodologia devem ser rigidamente seguidas, pelo menos em um primeiro momento. A cada *Sprint*, uma nova alteração deverá ser implantada, mas nunca quebrando qualquer regra. Apenas depois que todos os desenvolvedores estiverem confiantes na metodologia, até mesmo os aspectos do próprio *Scrum* podem ser repensados, dessa forma garantindo-se que cada membro da equipe já esteja pensando de acordo com o paradigma ágil.

Scrum representa uma mudança de paradigma, na qual o processo de desenvolvimento de sistemas deixa de ter práticas bem definidas e controláveis e começa a ter passos flexíveis, mutáveis e auto-organizáveis. Esta metodologia, junto com outras metodologias ágeis, tem como natureza estar sempre sendo adaptada pela própria Equipe, e não deve nunca cair em estagnação. Sua natureza é ser, sempre, uma "metodologia sem fim".

6. Agradecimentos

Agradecemos à UFABC, pela oportunidade de inovação na criação de uma equipe de desenvolvimento, ao NTI, pela confiança depositada na implantação de uma metodologia ágil como solução à demanda de desenvolvimento, e principalmente à Equipe que compõe a divisão de Desenvolvimento de Sistemas, por manter-se sempre comunicativa, desta forma permitindo o bom funcionamento de uma metodologia ágil.

Referências Bibliográficas

Couto, Ana Brasil (2007) "CMMI -Integração dos Modelos de Capacitação e Maturidade de Sistemas", ed. Ciência Moderna.

Portela, Carlos S., Oliveira, Sandro R. B. (2009) "Uma Proposta de Adaptação do Processo de Gerenciamento do CTIC-UFPA adotando Práticas Ágeis", apresentado no III Workshop de TI das IFES, Belém - PA

Royce, Winston W. (2003) "Managing the Development of Large Software Systems", disponível em: http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf.

Schwaber, Ken (2004) "Agile Project Management with Scrum", Microsoft Press. Xéxeo, Geraldo B. (1999), "Estamos Ficando Ultrapassados", disponível em: http://xexeo.wordpress.com/2009/10/24/estamos-ficando-ultrapassados/

Wells, J. D. (2009) "Extreme Programming", http://www.extremeprogramming.org

Adoção de métodos ágeis em uma Instituição Pública de grande porte - um estudo de caso

Claudia de O. Melo[1], Gisele R. M. Ferreira[2]

[1]Instituto de Matemática e Estatística
Universidade de São Paulo (USP) - São Paulo, SP - Brasil

[2]Banco Central do Brasil
Departamento de Informática - Brasília - DF - Brasil

claudia@ime.usp.br, gisele.ferreira@bcb.gov.br

Abstract. *An increasing need of software quality and the pressure for faster software delivery are the main reasons that lead companies to adopt agile software development methodologies. This paper describes a case study of a Brazilian government organization that adopted agile methods after a long time of using traditional software development techniques. This article describes the organizational context that motivated and supported this change, describes two pilot projects and discusses the observed results from technical and management perspectives. The results after 18 months were satisfactory and decisive to encourage new experiences with agile methods within the organization.*

Resumo. *A busca crescente por qualidade de software e a pressão por entregas rápidas são os principais motivos que levam as empresas a adotarem métodos ágeis de desenvolvimento de software. Este artigo descreve um estudo de caso de uma Instituição Pública Brasileira de grande porte que optou por adotar métodos ágeis após anos de experiência com métodos tradicionais de desenvolvimento. O artigo detalha o contexto organizacional que motivou e apoiou o processo de adoção, descreve dois projetos pilotos e discute os resultados observados, sob perspectivas técnicas e gerenciais. Os resultados após 18 meses de implantação foram considerados satisfatórios e decisivos para encorajar novas experiências com métodos ágeis dentro da organização.*

1. Introdução

Métodos ágeis de desenvolvimento de software vêm ganhando crescente popularidade desde o início da década de 2000 e, de acordo com Parsons et al. (2007), em algumas circunstâncias podem oferecer melhores resultados para projetos de desenvolvimento de software quando comparados às abordagens mais tradicionais. Eles são regidos pelo Manifesto ágil (2001), conjunto de valores e princípios criados por 17 desenvolvedores experientes, consultores e líderes da comunidade de desenvolvimento de software. Segundo Dybå (2000) e Nerur et al. (2005) estes métodos podem ser vistos como uma reação aos métodos tradicionais (também conhecidos como dirigidos por planos) que enfatizam o planejamento e a predição de soluções para cada problema do desenvolvimento.

De acordo com Ambler (2006) e VersionOne (2009), dentre os métodos ágeis existentes, a Programação Extrema de Kent Beck (2001), ou XP, e o Scrum de Schwaber (2004) são os mais conhecidos e adotados na indústria. XP propõe um conjunto de valores, princípios e práticas que visam a garantir o sucesso no desenvolvimento de software, em face a requisitos vagos e com alto grau de incerteza. XP promete produzir software de alta qualidade com alta produtividade, o que atrai a atenção das empresas que demandam cada vez mais velocidade e qualidade em seus produtos. Já o Scrum é uma metodologia ágil para gestão e gerenciamento de projetos, muitas vezes associada a outros métodos e processos de desenvolvimento de software.

Dybå e Dingsøyr (2008) apontam que diversos trabalhos sobre métodos ágeis já foram publicados, no entanto pouco se sabe sobre como esses métodos são realizados na prática e que efeitos geram. Dentre os estudos de caso levantados nesta pesquisa sobre implantação de métodos ágeis, apenas o reportado por Freire et al. (2005) tratava da adoção por empresas brasileiras. Além disso, nenhum trabalho foi encontrado sobre a implantação de métodos ágeis em órgãos públicos brasileiros.

Este trabalho tem como objetivo analisar empiricamente a adoção de métodos ágeis e seu impacto no aprendizado, qualidade do código-fonte, produtividade e satisfação do cliente. Para isso foi conduzido um estudo de caso em uma instituição pública brasileira de grande porte. Foram levantados dados quantitativos e qualitativos para responder às questões de pesquisa. Os ganhos e limitações da adoção serão discutidos e contrastados com alguns resultados da literatura. Espera-se também gerar uma contribuição para outras empresas brasileiras que desejem adotar métodos ágeis, sejam elas públicas ou privadas.

O artigo está organizado da seguinte forma: a Seção 2 apresenta uma revisão da literatura acerca da adoção de métodos ágeis e os principais resultados obtidos. A Seção 3 descreve os objetivos de pesquisa e detalha a metodologia usada. A Seção 4 descreve o estudo de caso realizado em uma instituição pública brasileira de grande porte, enquanto a Seção 5 descreve e analisa os primeiros efeitos da adoção de métodos ágeis na instituição. A Seção 6 discute os resultados, suas implicações e limitações. A Seção 7 conclui o trabalho e aponta alguns trabalhos futuros.

2. Revisão da literatura

Um estudo apresentado por Svensson e Höst (2005) descreveu os resultados da adoção de XP em uma organização considerada de grande porte (emprega 1500 pessoas, 250 em desenvolvimento de software). A avaliação se deu por meio de um projeto piloto que durou oito meses. Três pessoas foram escolhidas para representar os diferentes pontos de vista da organização, duas líderes de projeto e uma representante de cliente. Como resultado, a organização percebeu que a adoção de métodos ágeis teve um efeito positivo sobre a colaboração entre os membros da empresa. Os autores aconselham não subestimar o esforço necessário para introduzir e adaptar XP em uma organização.

Ilieva et al. (2004) descrevem um estudo de caso que compara um projeto XP com um projeto tradicional da organização. Os projetos eram similares em tamanho, esforço e tecnologia adotada. O time do piloto foi composto de 4 pessoas e o projeto foi organizado para realizar uma entrega com 3 iterações. Como resultado final foi

observado o aumento da produtividade em 41,23%, redução de esforço (homem/hora) de 11,45% e redução de defeitos em 13,33%.

Freire et al. (2005) descrevem a experiência de introdução de XP em uma *start-up* brasileira. Eles discutem as adaptações feitas nas práticas de XP e como os aspectos culturais e econômicos brasileiros afetam a implantação do método. Todas as práticas foram implantadas de uma só vez e 4 sistemas foram desenvolvidos ao longo de 12 *releases* com iterações de 2 semanas. Os autores relatam a dificuldade de se implantar métodos ágeis em times heterogêneos, o que pode ser resolvido com paciência e 'paixão brasileira'. No entanto, o artigo não apresenta uma avaliação dos times envolvidos acerca das dificuldades e benefícios percebidos com a implantação.

Loftus e Ratcliffe (2005) apresentam um estudo realizado com estudantes de pósgraduação para verificar, dentre diversos objetivos, se XP promove o aprendizado de novas tecnologias. Como resultado, a maior parte dos estudantes relatou ter adquirido muito conhecimento em um período curto de tempo e que o uso de XP auxilia a introdução de novas tecnologias. Os autores recomendam que os estudantes sejam apresentados às metodologias tradicionais em projetos de grupo antes de aprender XP em sua forma completa. Por outro lado, os resultados de McAvoy e Butler (2007) mostraram que o aprendizado do time pode não ser efetivo em XP, possivelmente em função da pressão social sobre um indivíduo em estar sempre em conformidade com a 'visão do grupo' (fenômeno conhecido como o Paradoxo de Abilene). Portanto, é necessário manter um certo nível de conflito no time para que haja maior comunicação, contestação e, por fim, aprendizado.

3. Projeto do estudo de caso

Este trabalho tem como objetivo explorar a adoção de métodos ágeis (MAs) e seu impacto em uma organização por meio de quatro questões de pesquisa (QP):

QP1. MAs aceleram o aprendizado de novas tecnologias, conceitos e padrões?

QP2. MAs aumentam a qualidade do código do sistema?

QP3. MAs aumentam a produtividade do time?

QP4. MAs aumentam a satisfação do cliente?

Para descrever e explorar os efeitos da adoção de métodos ágeis foi usada a metodologia de pesquisa de estudo de caso. Segundo Zelkowitz e Wallace (1998), estudo de caso é um estudo observacional em que o pesquisador monitora projetos em profundidade e coleta dados ao longo do tempo. As unidades de análise utilizadas foram os dois projetos piloto de métodos ágeis da organização.

Método de coleta de dados. Os dados foram coletados de várias maneiras e fontes, incluindo observação, entrevistas, questionários online e bases de dados organizacionais. Os questionários tinham o objetivo de coletar dados qualitativos e quantitativos a respeito do time, sua experiência prévia, seu nível de aprendizado e opiniões sobre produtividade, qualidade e satisfação. Dados de observação e entrevistas complementaram os questionários sob o ponto de vista qualitativo. As pesquisadoras atuaram como observadoras participantes em parte dos projetos. Dados quantitativos sobre a produtividade e a qualidade foram coletados ao final de cada projeto piloto para possibilitar a comparação entre os projetos anteriores (tradicionais) e os pilotos ágeis. Bases de dados organizacionais continham as medições dos projetos passados.

De acordo com Svensson e Höst (2005), a maior parte das pesquisas sobre introdução e uso de métodos ágeis em organizações é baseada na opinião dos times que aplicaram os métodos. Dois questionários foram criados, um para a equipe técnica, outro para os clientes dos projetos estudados, ambos com três seções. A primeira seção continha perguntas demográficas sobre o perfil do participante e sua experiência prévia em desenvolvimento de software e métodos ágeis. A segunda seção tratava das experiências durante os pilotos executados, como o nível de dificuldade no aprendizado das práticas, dificuldades enfrentadas e fatores que favoreceram o aprendizado. Para medir o nível de aprendizado, uma escala Likert de 4 pontos foi usada para refletir o nível de percepção de cada participante sobre seu próprio aprendizado. Para colher a opinião sobre as dificuldades enfrentadas no aprendizado, assim como os fatores positivos no processo, foram feitas perguntas abertas. A terceira seção era sobre a percepção sobre o aumento ou diminuição do aprendizado, produtividade, qualidade e satisfação do cliente após o uso de métodos ágeis. As percepções foram medidas com uma escala Likert de 5 pontos, projetada para ser compatível com a usada no questionário online de Ambler (2008) e, assim, permitir comparação entre os resultados.

O questionário online[1] foi aplicado após a finalização dos pilotos e ficou disponível durante uma semana para os 24 participantes dos projetos e 20 pessoas responderam, representando 83,3% do total. O restante não trabalhava mais na organização ou estava em período de férias ou licença.

Validade. As principais ameaças aos estudos de caso aplicáveis a este estudo de caso são mencionadas por Yin (2009). Dentre elas, destaca-se a confiabilidade dos dados coletados e dos resultados obtidos. Para Seaman (1999), o uso de várias fontes de dados e métodos de coleta permite a triangulação, uma técnica para confirmar se os resultados de diversas fontes e de diversos métodos convergem. Dessa forma é possível aumentar a validade interna do estudo e aumentar a força das conclusões. Nesta pesquisa houve triangulação de dados, de observador e de metodologia. A triangulação de dados se deu pelo uso de bases organizacionais, questionários e entrevistas para coletar dados, enquanto a de observador se deu pelo uso de dois observadores. Por fim, a triangulação de métodos ocorreu pelo uso de métodos de coleta quantitativos e qualitativos.

4. Estudo de caso

4.1. A organização em estudo

A empresa em estudo é uma instituição pública que atua no sistema financeiro. Ela emprega cerca de 5000 pessoas, 700 delas na área de informática. A preocupação com processos padronizados e documentados foi inserida no contexto de desenvolvimento de software em 2002. Uma infraestrutura de apoio à implantação de um processo derivado do RUP, de Kruchten (2000), foi montada com disciplinas e fases, além da especialização de papéis. O departamento de informática foi dividido em subáreas por especialidade, criando áreas de relacionamento com o negócio (com a maior parte dos funcionários), a área de testes, a de projeto de software, além de outras áreas de apoio como a de gestão do relacionamento com as fábricas e a de controle da qualidade, padronização e reuso.

[1] Os questionários e as respostas estão disponíveis em www.ime.usp.br/~claudia/wbma

Por mais de 8 anos, toda a organização utilizou (e ainda utiliza) esse processo no desenvolvimento. Houve uma aposta na terceirização como chave para o aumento de escala de produção, porém mantendo o foco nas competências principais dentro da instituição, seguindo as tendências do mercado apontadas por Paisittanand e Olson (2006). O modelo de terceirização adotado garantia que o conhecimento dos requisitos de negócio e a definição da solução arquitetural fosse propriedade da organização, enquanto a implementação do sistema ficava a cargo das fábricas de software.

A formalização de um processo de desenvolvimento corporativo, com especializações em papéis, e a adoção de fábricas de software representaram um aumento de escala a um curto prazo, porém trouxeram problemas novos relatados pela organização. Havia dificuldade de comunicação derivada da alta especialização em papéis. Os prazos de entrega eram longos e faltava objetividade na definição do escopo de sistemas. Os funcionários estavam desmotivados por trabalhar grande parte do tempo em tarefas burocráticas de gerenciamento das fábricas ou em atividades muito especializadas. Além disso, criou-se uma dependência das fábricas para todas as atividades relacionadas à implementação. Neste contexto, a adoção de métodos ágeis apareceu como uma proposta de solução para os diversos problemas apresentados, inclusive como uma alternativa para a solução do problema de escala de produção.

Infraestrutura tecnológica. O ambiente de desenvolvimento da organização foi projetado para aumentar a produtividade dos times, pois adotava um servidor de aplicação leve - Jetty (2010), um banco de dados em memória - Hsqldb (2010), uma ferramenta de gerência do build - Maven (2010) e, por fim, uma ferramenta para integração contínua - Hudson (2010). A arquitetura de referência estava bem documentada e disseminada. Ela baseava-se em Java, em alguns frameworks de aumento de produtividade como Wicket (2010), Hibernate 3 (2010) e Spring (2010) e dispunha de um conjunto de componentes de infraestrutura e negócio.

Além disso, o entendimento sobre qualidade interna de sistemas já era um conceito compreendido. A organização fazia uso de ferramentas de análise estática de código como PMD (2010) e CheckStyle (2010) com regras já adaptadas à arquitetura de referência e também praticava inspeção de código por meio do grupo de controle da qualidade. Testes de unidade, mesmo que após a implementação, já haviam sido experimentados em projetos recentes de desenvolvimento. Seus desafios iniciais já haviam sido superados e a organização já estava convencida dos seus benefícios.

4.2. Planejamento e execução da implantação

Em 2007 foi a primeira vez que métodos ágeis foram discutidos em âmbito organizacional. Porém, a idéia apenas ganhou força um ano depois, quando um grupo de trabalho foi montado para planejar as etapas da implantação. O grupo foi formado por pessoas que defendiam a adoção de métodos ágeis e pessoas responsáveis pela manutenção dos processos de desenvolvimento corporativo da organização. O planejamento e a execução foram cuidadosos e seguiram algumas das melhores práticas sugeridas por Griffiths (2003), como a obtenção de apoio gerencial, a escolha do método e das práticas, a realização de pilotos, além de educação e suporte ao time.

O apoio da chefia do departamento foi obtido no início do planejamento. Foi decidido que os primeiros projetos adotariam todas as práticas das metodologias XP e

Scrum para fins de aprendizado. Além disso, como já mencionado, a organização já tinha uma infraestrutura tecnológica adequada disponível e se sentia preparada para iniciar novas experiências.

Foram planejados dois pilotos iniciais. Também foram discutidas as características dos sistemas a serem construídos nos pilotos e dos times de desenvolvimento alocados. Antes do início dos pilotos, as pessoas envolvidas buscaram estudar XP e Scrum. Ainda dentro do planejamento de implantação, o trabalho de preparação do time de desenvolvimento não durou mais do que uma semana. Foram repassados conceitos, valores e princípios ágeis para um nivelamento inicial. A partir daí, todo o aprendizado técnico e do método aconteceria na prática, orientado por pessoas mais experientes da própria organização. A contratação de um treinamento externo de práticas de XP foi previsto para o primeiro projeto piloto e aconteceria quando a maturidade adquirida pelo time já favorecesse um melhor aproveitamento.

De acordo com o planejado, o 1º projeto piloto foi iniciado em outubro de 2008, com duração prevista de 6 meses. O treinamento externo aconteceu após 3 meses do inicio do projeto e, como esperado, surgiu um efeito muito positivo. Em seguida foi preparado o 2º projeto piloto, nos mesmos moldes anteriores, e planejado para 10 meses.

Como os resultados observados após o 1º projeto piloto haviam sido satisfatórios, o plano de implantação de métodos ágeis foi estendido com um novo ciclo de desenvolvimento do 1º projeto piloto e planejado a subcontratação da evolução do sistema em uma fábrica de software como mostra a Figura 1. Todo processo de implantação foi acompanho pela equipe de qualidade e processos de desenvolvimento organizacional que, ao fim do 2º projeto piloto, montou um planejamento para institucionalização dos métodos em toda a organização.

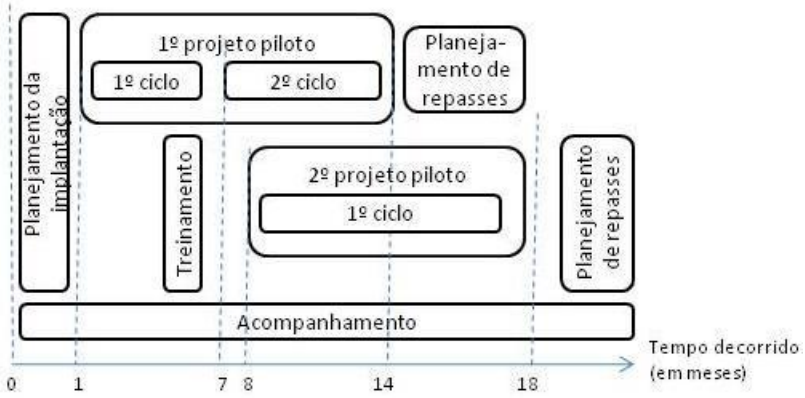


Figura 1 – Implantação de métodos ágeis na organização

4.3. Projetos pilotos

Para os projetos pilotos foram escolhidos sistemas que não eram críticos sob o ponto de vista de negócio e nem possuíam restrições de prazo severas. Estas escolhas visavam minimizar os riscos que a experimentação de novos métodos poderia trazer para o negócio da organização.

Os projetos pilotos foram organizados em ciclos, *releases* e iterações, de acordo com as recomendações de Scrum e XP. Os ciclos representavam marcos formais com a entrega de uma versão completa do sistema para uso em produção. *Releases*

representavam marcos intermediários com entrega de funcionalidades para fins de homologação ou, eventualmente, de produção. As iterações representavam pequenos marcos de planejamento e controle do desenvolvimento. Funcionalidades da aplicação eram colocadas para validação pelos clientes ao final das iterações. A Tabela 1 caracteriza os projetos piloto executados. Vale ressaltar que houve troca de membros nos projetos, por esse motivo o número de Tamanho médio do time não equivale ao número total de participantes de cada projeto (e de respondentes aos questionários). Nenhum participante do Projeto 1 trabalhou no Projeto 2 e vice-versa.

Tabela 1 - Informações gerais sobre os projetos pilotos

Características	Projeto 1	Projeto 2
Tamanho (médio) do time	7 (6 desenvolvedores e 1 gerente)	6 (5 desenvolvedores e 1 coach com papel gerencial)
Duração do projeto	1 ano e 2 meses	10 meses
Domínio	Cálculo de dívidas de contratos	Controle de ações educacionais
Time coalocado	Time e cliente na mesma sala	Time e cliente separados por 1 andar.
Tamanho da iteração	1º ao 4º mês: 2 semanas 5º mês em diante: 1 semana	1 semana
Número de entregas (em homologação)	10	3
Número de entregas (em produção)	6	1
Práticas adotadas	Todas de XP e Scrum	Todas de XP e Scrum

As equipes eram compostas essencialmente por pessoas com perfil de desenvolvedor, como mostra a Tabela 2, mas 50% delas vinham exercendo atividades de analista de requisitos, analista de qualidade, gerentes de projetos e testadores antes da experiência ágil.

Tabela 2 - Perfis dos membros dos projetos pilotos

Perfis do projeto	Time do Projeto 1	Time do Projeto 2
Desenvolvedor	4	6
Gerente	2	0
Especialista em testes	0	2
Projetista	3	0
Analista de requisitos	0	1
Analista de qualidade	1	1
Total	**10**	**10**

A experiência técnica dos membros da equipe também era bem heterogênea como mostra a Tabela 3. 40% do time possuíam menos de dois anos de experiência com desenvolvimento de software orientado a objetos, 10% sequer já havia trabalhado com desenvolvimento OO. 90% do time conheciam pouco ou quase nada da recente arquitetura de referência da organização, seja pela alta especialização dos papéis já mencionada, seja pelo pouco tempo de trabalho na instituição. Com relação aos métodos ágeis, não mais de 15% das pessoas possuíam experiência profissional com alguma prática ágil. Durante o acompanhamento dos pilotos foi possível observar que algumas

pessoas possuíam conhecimento teórico do assunto e outras sequer sabiam os desafios trazidos pelo novo paradigma de desenvolvimento.

Tabela 3- Experiência das equipes antes dos pilotos

Tempo	Desenvolvimento de software	Orientação a objetos	Arquitetura de referência	Na instituição	Métodos ágeis
0	10%	10%	60%	0%	85%
De 1 a 6 meses	5%	0%	30%	35%	5%
7 meses a 2 anos	25%	30%	5%	25%	10%
De 2 a 5 anos	30%	35%	5%	15%	0%
De 6 a 9 anos	25%	15%	0%	25%	0%
10 ou + anos	5%	10%	0%	0%	0%
Total	100%	100%	100%	100%	100%

5. Primeiros efeitos da adoção

5.1 Efeitos sobre o aprendizado

Para analisar o ganho em aprendizado, primeiro foi levantado o nível de facilidade/dificuldade em aprender as práticas ágeis, particularmente as de XP – mais visíveis ao corpo técnico. Segundo a percepção da equipe técnica, as práticas de pares e de equipe são facilmente compreensíveis e aplicáveis como pode ser observado pela Figura 2. Dentre as práticas de pares, exceto a programação em pares, que foi a mais facilmente absorvida pela equipe, as demais apresentaram em média o mesmo grau de dificuldade. Dentre as práticas de equipe, apenas a "Metáfora" apresentou alguma dificuldade de compreensão e aplicação.

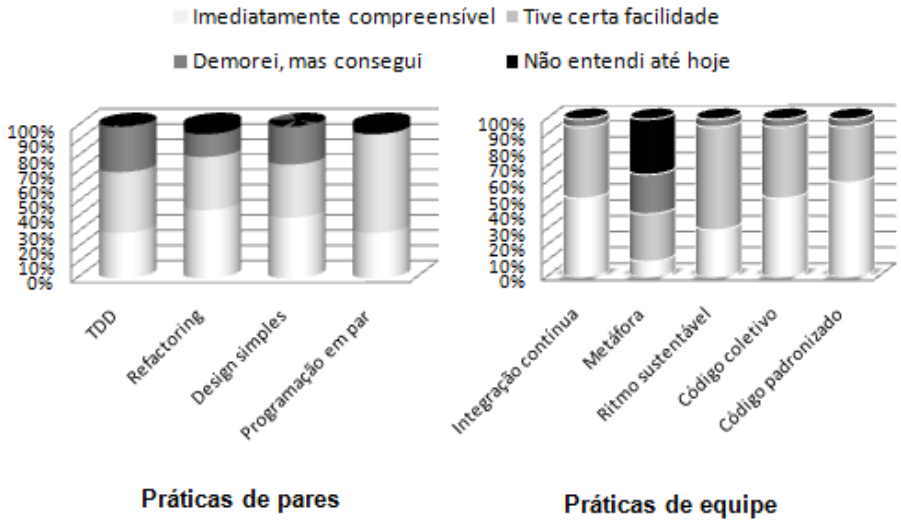


Figura 2 - Aprendizado de Práticas de pares e equipe

O aprendizado das práticas organizacionais foi avaliado segundo as perspectivas da equipe técnica e dos clientes como pode ser observado na Figura 3. A equipe técnica sentiu mais dificuldades na aplicação da prática "Time completo". Em ambas as

perspectivas, a aplicação da prática de "Testes do cliente" apresentou certa dificuldade. As análises qualitativas também indicaram dificuldades maiores na adoção das práticas de testes em relação às demais. Os principais pontos destacados estão relacionados à técnica de testes de unidade, legibilidade e manutenibilidade dos testes dos clientes. Em ambos os projetos pilotos, a introdução da prática de testes dos clientes foi implantada após certo tempo de projeto (cerca de 4 meses), quando a equipe se sentiu madura o suficiente para compreender e aplicar tal prática.

Quanto ao emprego da prática de "*Releases* curtas", embora facilmente compreendida, não apresentou a mesma facilidade na sua aplicação. Ambos os projetos pilotos relataram ter enfrentado dificuldades em convencer os clientes a realizar publicações parciais do sistema em produção. Percebeu-se que a organização não estava acostumada a receber soluções ainda não completamente finalizadas, mesmo que agregassem valor ao negócio. Além disso, em função da cultura de desenvolvimento de sistemas existente na organização, ambos os projetos relataram dificuldades em realizar um projeto simples que não tivesse a intenção de antecipar mudanças futuras.

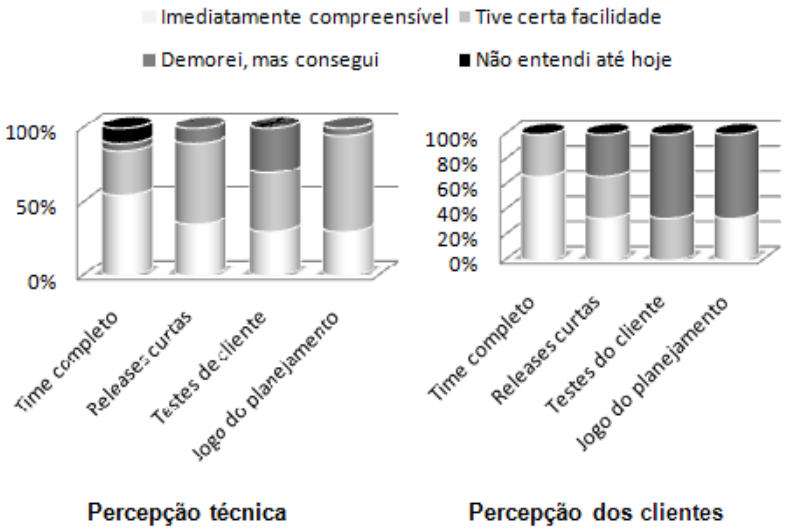


Figura 3 - Aprendizado de práticas organizacionais

Em uma escala de 1 (Definitivamente não) a 5 (Definitivamente sim), a Tabela 4 mostra a percepção do time e dos clientes sobre a Questão de Pesquisa 1 (QP1). As entrevistas e respostas abertas do questionário confirmaram a opinião das pessoas sobre a facilidade de aprender métodos ágeis e a rapidez em acumular novos conhecimentos.

Tabela 4 - Percepções sobre o aprendizado do time

Percepção (média / desvio padrão)	**Percepção técnica**	**% de respostas entre a escala 4 e 5**	**Percepção dos clientes**	**% de respostas entre a escala 4 e 5**
QP1. Acelera o aprendizado de novas tecnologias, conceitos e padrões	4,45 / 0,69	90%	5 / 0	100%

Esses resultados confirmam os achados de Loftus e Ratcliffe (2005). Não houve evidência de que o aprendizado foi prejudicado em função do pensamento de grupo, conforme mencionado por (McAvoy e Butler 2007).

5.2. Efeitos da implantação sobre a qualidade do código

A qualidade do código de um sistema pode ser avaliada por diversas métricas. Das métricas usadas na instituição, três foram selecionadas para efeitos de comparação entre os sistemas desenvolvidos com o processo tradicional e os pilotos ágeis. O critério de seleção foi unicamente baseado na disponibilidade e facilidade de acesso aos dados pelos pesquisadores. As métricas escolhidas foram 1) Porcentagem de aderência às regras de análise estática (implementadas pelas ferramentas já mencionadas PMD e Checkstyle), 2) Porcentagem de cobertura de código e 3) Porcentagem de testes de unidade executados com sucesso. Dezesseis projetos tradicionais, feitos com a mesma tecnologia dos projetos piloto, foram usados para o cálculo da média e desvio padrão de projetos tradicionais. A Tabela 5 apresenta, para cada métrica selecionada, a média e desvio padrão obtidos nos projetos tradicionais ($Qualidade_{TRAD}$) e a qualidade obtida nos pilotos ágeis ($Qualidade_{AGIL}$).

Tabela 5 – Comparações de qualidade do código

Atributos de qualidade	**$Qualidade_{TRAD}$ (média/desvio)**	**$Qualidade_{AGIL}$ Projeto 1**	**$Qualidade_{AGIL}$ Projeto 2**
Aderência às regras de análise estática (PMD/ CheckStyle)	98,8%/0,03	100%	99,8%
Cobertura de código	78,25%/0,25	93,4%	92,8%
Testes de unidade executados com sucesso	98,10%/0,06	100%	100%

Os números médios de qualidade dos projetos tradicionais confirmam que a preocupação com qualidade de código e o uso de testes de unidade já estavam presentes na instituição. Dentre as métricas de qualidade analisadas, a métrica de cobertura de código foi a que apresentou um aumento significativo, aproximadamente 19% de aumento da cobertura.

Mais importante do que a cobertura de código por testes de unidade é a qualidade dos testes escritos pelos clientes e equipe de desenvolvimento. Como não é escopo deste trabalho medir a qualidade dos testes de forma quantitativa e objetiva, foi considerada a percepção técnica do time. A Tabela 6 mostra, em uma escala de 1 (Definitivamente diminuem) a 5 (Definitivamente aumenta), a percepção da equipe técnica sobre a Questão de Pesquisa 2 (QP2). Os resultados foram mais otimistas que os descritos por Ambler (2008), em que 77% das respostas ficarem entre a escala 4 e 5.

Tabela 6 - Percepções sobre a qualidade do código-fonte do sistema

Percepção (média / desvio padrão)	**Percepção técnica**	**% de respostas entre a escala 4 e 5**
QP2. Aumentam a qualidade do código do sistema	4,6 / 0,60	95%

5.3 Efeitos da implantação sobre a produtividade

A métrica de produtividade usada na organização é a razão entre tamanho do sistema em pontos de função e a quantidade de horas gastas no projeto. Essa medição em geral é realizada ao final do projeto ou em algum marco especificado pela gerência. Uma equipe especializada em contagem de pontos de função realiza a medição do tamanho do projeto e as horas são apontadas por toda a equipe em um sistema corporativo. A Tabela 7 apresenta o ganho de produtividade dos pilotos ágeis ($Ganho_{AGIL}$) em relação à

produtividade média dos projetos tradicionais da organização[2]. No piloto 1, houve um ganho de produtividade de 8,21%. Já no piloto 2 foi constatado um ganho de produtividade de 30,89% em relação à média da organização. O Projeto 1 era mais complexo (envolvia diversos cálculos com alta precisão) do que o Projeto 2. Além disso, foi o primeiro piloto de ágeis, o que sugere uma curva maior de aprendizado e adaptação. O projeto 2 pôde aprender com os erros do projeto 1. Por fim, a produtividade obtida pelo projeto 2 foi considerada satisfatória, entretanto a organização acredita que o ganho pode ser aumentado.

Tabela 7 - Comparações de produtividade

$Ganho_{AGIL\ Projeto\ 1}$ (%)	$Ganho_{AGIL\ Projeto\ 2}$ (%)
8,21%	30,89%

Em uma escala de 1 (Definitivamente diminuem) a 5 (Definitivamente aumenta) a Tabela 8 apresenta a média e o desvio padrão da opinião do time e do cliente sobre Questão de Pesquisa 3 (QP3). Para a equipe técnica, em média a produtividade aumenta, sendo que 90% dos participantes acreditam que ela aumenta ou definitivamente aumenta. Os resultados foram próximos aos descritos por Ambler (2008), onde 82% das respostas ficaram entre 4 e 5. Todos os clientes acreditam que a produtividade aumenta muito.

Tabela 8 - Percepções sobre a produtividade do time

Percepção (média / desvio padrão)	Percepção técnica	% de respostas entre a escala 4 e 5	Percepção dos clientes	% de respostas entre a escala 4 e 5
QP3. Aumentam a produtividade do time	4,35 / 0,67	90%	5 / 0	100%

A análise qualitativa confirma essa percepção do time e do cliente sobre o aumento da produtividade. Em entrevistas realizadas com alguns membros ao final do projeto, sempre que questionados sobre a percepção de aumento da produtividade, os entrevistados afirmavam que houve um aumento. No entanto, a produtividade não foi um tema muito comentado nos questionários online, principalmente pela equipe técnica, ao contrário da qualidade e da satisfação do cliente. Isso leva à conclusão de que o aumento da produtividade não é tão perceptível para os participantes quanto os demais ganhos. Isso também pode ser observado na média da percepção técnica apresentada na Tabela 9, a menor dentre as médias de aprendizado, qualidade e satisfação do cliente.

5.4 Efeitos sobre a satisfação do cliente

Os clientes de ambos os projetos pilotos já haviam participado em projetos anteriores de desenvolvimento de software e puderam então fazer um comparativo da experiência ágil e experiências anteriores. A Tabela 9 apresenta a análise quantitativa das opiniões do time e do cliente sobre a Questão de Pesquisa 4 (QP4). 90% dos respondentes acham que o uso de métodos ágeis aumenta ou aumenta muito a satisfação do cliente. Os resultados foram mais otimistas que os descritos por Ambler (2008), onde 78% das respostas ficarem entre 4 e 5. Os clientes avaliaram, em unanimidade, que sua satisfação aumentou muito.

[2] os dados reais não foram divulgados por serem usados em negociação contratual entre a instituição e seus subcontratados.

A satisfação também pôde ser observada durante o desenrolar dos projetos pilotos, à medida que os clientes foram incorporando as práticas ágeis às suas rotinas de trabalho diário. Além disso, a satisfação foi avaliada pelo questionário online e por entrevistas. Dentre os benefícios destacados pelos clientes, os principais foram: 1) redução significativa do prazo de entrega do sistema; 2) possibilidade de alteração do escopo ao longo do desenvolvimento e assim obter um software mais aderente às necessidades do negócio; 3) aumento do entendimento sobre os custos de implementação de uma funcionalidade, conscientizando o cliente sobre a importância da priorização das demandas e 4) Maior segurança com relação à qualidade dos sistemas por possuírem testes automatizados, escritos e revisados por eles.

Tabela 9 - Percepções sobre satisfação dos clientes

Percepção (média / desvio padrão)	**Percepção técnica**	**% de respostas entre a escala 4 e 5**	**Percepção dos clientes**	**% de respostas entre a escala 4 e 5**
QP4. Aumentam a satisfação do cliente	4,5 / 0,69	90%	5 / 0	100%

5. Discussão

A implantação de métodos ágeis em uma organização é um processo lento e complexo. Em organizações públicas, onde os processos burocráticos prezam pelo maior controle, em detrimento dos resultados mais rápidos, é particularmente mais complicado. A simples realização de alguns projetos pilotos não é suficiente para tornar as práticas, valores e princípios ágeis de fato implantados.

Este estudo de caso permitiu observar que as principais dificuldades enfrentadas na implantação de métodos ágeis não estão relacionadas ao aprendizado das práticas ágeis e sim com a necessidade de mudança da cultura organizacional. Enquanto apenas o projeto de desenvolvimento de software pensar e agir de forma ágil e o restante da organização mantiver os vícios e culturas derivadas dos processos tradicionais não será possível usufruir realmente dos benefícios ágeis.

O fato da instituição já ter um ambiente tecnológico favorável permitiu a rápida adoção e o aprendizado de práticas como Integração contínua, *Releases* curtas, Projetos Simples, Código padronizado, Código coletivo e TDD. Isso contribuiu para os resultados positivos observados na Questão de pesquisa 1. Além disso, a obtenção de patrocínio gerencial, a escolha adequada de projetos pilotos e a preparação do time e da infraestrutura tecnológica foram destacados pelos participantes como pontos muito importantes na adoção de métodos ágeis.

Os primeiros resultados obtidos após 18 meses de implantação de métodos ágeis motivaram a organização a explorar novas possibilidades de trabalho com fábricas de software. O primeiro projeto piloto foi subcontratado como manutenção evolutiva em uma fábrica em outro estado. Segundo relatos dos clientes, os testes de aceitação e documentação do código fonte foram suficientes para que a fábrica pudesse compreender o sistema e o processo de negocio. A equipe técnica destacou ainda que o projeto simples, os testes automatizados e o código padronizado foram práticas que favoreceram a subcontratação da evolução do software de forma mais eficiente que a tradicional documentação do sistema.

As limitações dos resultados obtidos neste estudo são relacionadas aos métodos de coleta de dados e à precisão do dado em si. Segundo Sue e Ritter (2007), a condução de questionários online pode levantar informações incompletas e estimular opiniões que só foram geradas, porque foram solicitadas, não pelo fato do participante ter certa opinião. Além disso, o número de projetos (dois) e de participantes (20) representa apenas uma pequena amostra da instituição. As médias de qualidade e produtividade tradicionais representam todo o universo de sistemas e profissionais da empresa e não são necessariamente um critério de comparação justo com os pilotos. Por isso, para aumentar a validade interna do estudo, foram usados dados qualitativos que permitem a confirmação dos resultados quantitativos. Além disso, resultados de outros estudos foram citados para permitir mais uma análise dos resultados deste estudo de caso.

6. Conclusão

Os métodos ágeis para desenvolvimento de software são uma alternativa ao desenvolvimento tradicional, dirigido por planos. Para adotar métodos ágeis em uma organização, são necessários diversos passos de planejamento e uma execução cuidadosa. Neste artigo foram apresentados os resultados empíricos obtidos no estudo de caso de adoção de métodos ágeis em uma instituição pública brasileira de grande porte. Os resultados mostraram que a adoção teve um efeito positivo no aprendizado de novas tecnologias e na satisfação dos clientes e um discreto aumento na qualidade do código e na produtividade dos times estudados. O trabalho gera uma contribuição tanto para as empresas (públicas e privadas) que desejam adotar métodos ágeis, quanto para a academia, pois relata e analisa os resultados empíricos observados no estudo de caso.

Além dos benefícios percebidos nos dois projetos pilotos sob estudo, a adoção de métodos ágeis pode vir a ser uma opção de aumento de escala de produção em organizações públicas frente à contratação de fábricas de software. A adoção de métodos ágeis, além de reduz os prazos de entregas, reduz também os problemas de comunicação e a necessidade de negociações contratuais com empresas subcontratadas. Como trabalho futuro, pretende-se investigar o impacto da subcontratação de manutenções evolutivas de sistemas desenvolvidos com métodos ágeis.

Agradecimentos

As autoras agradecem o apoio financeiro recebido da Fapesp (processo 09/10338-3) e ao Banco Central do Brasil[3] pela autorização e colaboração na condução do estudo de caso.

Referências

Ambler, S. (2006) Agile adoption rate survey. http://www.ambysoft.com/surveys/agileMarch2006.html.

Ambler, S. (2008) Agile adoption rate survey. http://www.ambysoft.com/surveys/agileFebruary2008.html.

Beck, K. (2000) “Extreme Programming Explained - Embrace Change”. Addison-Wesley.

Chekstyle (2010). Disponível em: http://checkstyle.sourceforge.net/. Abril de 2010.

Dybå, T e Dingsøyr, T. (2008) “Empirical studies of agile software development: A systematic review”. In: *Information and Software Technology*, 50(9-10):833–859.

[3] As opiniões aqui expressas não necessariamente espelham o pensamento do Banco Central do Brasil.

Dybå, T. (2000) “Improvisation in small software organizations”. In: *IEEE Software*, 17(5):82–87.

Freire, A., Kon, F. e Torteli, C. (2005) “XP south of the equator: An experience implementing XP in Brazil”. In *Proceedings of the 6th International Conference on Extreme Programming and Agile Processes in Software Engineering* (XP2005), p. 10–18. Springer.

Griffiths, M. (2003) “Crossing the Agile Chasm: DSDM as an Enterprise Friendly Wrapper for Agile Development”, Quadrus Development White Paper, www.dsdm.org/knowledgebase/download/91/crossing_the_agile_chasm.pdf

Hibernate (2010) Relational persistence for Java. Disponível em http://www.hibernate.org/.

HSQLDB (2010). HyperSol data base. Disponível em http://hsqldb.org/. Abril de 2010.

Hudson (2010). Hudson CI. Disponível em: http://hudson-ci.org/. Abril de 2010.

Ilieva, S., Ivanov, P., and Stefanova, E. (2004). Analyses of an Agile Methodology Implementation. In *Proceedings of the 30th EUROMICRO Conference*. IEEE, Washington, DC, 326-333.

Jeffries, R. (2001) "What is XP?", http://xprogramming.com/xpmag/whatisXP. Novembro.

Jetty (2010). Jetty WebServices. Disponível em: http://jetty.codehaus.org/jetty/.

Kent Beck et al. (2001) “Manifesto for agile software development”. http://agilemanifesto.org/.

Kruchten, P. (2000) “The Rational Unified Process - An Introduction”. Addison Wesley.

Loftus, C. and Ratcliffe, M. (2005) “Extreme programming promotes extreme learning?”. In: *SIGCSE Bull.* 37, 3 (September), p.311-315.

Maven (2010). Disponível em: http://maven.apache.org/. Abril de 2010.

Nerur, S., Mahapatra, R. e Mangalaraj, G. (2005) “Challenges of migrating to agile methodologies”. In: *Communications of the ACM*, 48(5):72–78.

Paisittanand, S. e Olson, D. L. (2006) “A simulation study of IT outsourcing in the credit card business”, In: *European Journal of Operational Research*, v. 175, Issue 2, p.1248-1261.

Parsons, D., Ryu, H. e Lal, R. (2007) “The impact of methods and techniques on outcomes from agile software development projects”. In: *Organizational Dynamics of Technology-Based Innovation: Diversifying the Research Agenda*, v.235 of *IFIP*, p. 235–249. Springer Boston.

PMD (2010). Disponível em: http://pmd.sourceforge.net/. Abril de 2010.

Schwaber, K. (2004) “Agile Project Management with Scrum”, Microsoft Press, 163pp.

Seaman, C. B. (1999) “Qualitative Methods in Empirical Studies of Software Engineering”, In: *IEEE Trans. Softw. Eng.* 25, 4, 557-572.

Spring (2010) Disponível em http://www.springsource.org/ Abril de 2010.

Sue, V. M. e Ritter, L. A. (2007). “Conducting online surveys”. Thousand Oaks, CA: Sage.

Svensson, H. e Höst, M. (2005) “Views from an organization on how agile development affects its collaboration with a software development team”, In: *Lecture Notes in Computer Science*, vol. 3547, Springer Verlag, Berlin, p. 487–501.

VersionOne (2009). “State of Agile Development” 4th annual survey. http://www.versionone.com/pdf/2009_State_of_Agile_Development_Survey_Results.pdf

Wicket (2010). Apache Wicket. Disponível em http://wicket.apache.org/. Abril de 2010.

Yin, R. K. (2009) Case Study Research: Design and Methods. 4 ed. Thousand Oaks, CA: Sage.

Zelkowitz, M.V. e Wallace, D. (1998) “Experimental models for validating Technology”. IEEE Computer, 31(5): 23-31.

Estudo de Caso da Utilização de Scrum no Desenvolvimento Distribuído de Software

Virgínia C. Chalegre[1], Wylliams B. Santos[1], Leandro O. de Souza[1], Hernan J. Muñoz[1], Silvio Romero de Lemos Meira[1]

[1]Centro de Informática – Universidade Federal de Pernambuco (UFPE)
Caixa Postal 7851, Cidade Universitária – 50.732-970 – Recife – PE – Brasil
{vcc, wbs, los2, hjm, srlm}@cin.ufpe.br

Abstract. *The use of agile software development has grown recently by the benefits brought with incremental deliveries of the product which add value to the customer's business, more interaction among the customer and team members and therefore greater customer satisfaction. Another widely used form of development is the distributed development, aiming to reduce the delivery time and costs. When used jointly, agile methods can help alleviate some of the problems of distributed teams such as the lack of management and communication. Thus, this paper describes the experiences in adopting the two forms development jointly with the aim of raising the problems and report possible solutions that have been taken and can be applied in other distributed teams.*

Keywords: Agile Methodologies, Scrum, FireScrum, Distributed development

Resumo. *O uso de metodologias ágeis de desenvolvimento de software vem crescendo nos últimos tempos graças aos benefícios trazidos com entregas incrementais do produto que agregam valor ao negócio do cliente, maior interação com o cliente e os membros da equipe e consequentemente maior satisfação do cliente. Outra forma de desenvolvimento muito utilizado é o desenvolvimento distribuído, com o objetivo de reduzir o tempo de entrega e os custos. Quando utilizadas de forma conjunta, as metodologias ágeis podem ajudar a amenizar alguns dos problemas de equipes distribuídas como a falta de gerenciamento e comunicação. Desse modo, este trabalho descreve as experiências ao adotar as duas formas de desenvolvimento em conjunto, com o objetivo de levantar os problemas enfrentados e relatar possíveis soluções que foram tomadas e podem ser aplicadas por outras equipes distribuídas.*

Palavras-chave: *Metodologias ágeis, Scrum, FireScrum, Desenvolvimento Distribuído*

1. Introdução

Diante do cenário em que condições de mercado e necessidades dos usuários mudam constantemente, surgiram os métodos ágeis de desenvolvimento de software, com a proposta de tornar as equipes mais dinâmicas e o desenvolvimento mais rápido. Uma dessas abordagens é a metodologia Scrum, que enfatiza o uso de um conjunto de

padrões, permitindo que o time se envolva, participando ativamente do processo de desenvolvimento de software e que haja uma comunicação intensa entre as pessoas.

Outra grande tendência do mercado de software é o desenvolvimento distribuído caracterizado pela construção de software por equipes dispersas geograficamente. Essa configuração de desenvolvimento traz algumas vantagens competitivas, buscando soluções globais, como a redução do tempo de entrega do projeto, através de times paralelos trabalhando além das limitações de fuso-horário [Herbsleb 2001]. O baixo custo de manter uma estrutura física para comportar muitas pessoas e a carência de pessoas qualificadas numa mesma localidade são outras vantagens descritas do desenvolvimento distribuído de software [Sutherland 2008]. Entretanto, para que esse ambiente seja realmente produtivo, esforços adicionais na gestão do projeto são necessários para acompanhar o trabalho dos times, sendo importante dar ênfase à comunicação e interação entre os membros da equipe, já que segundo Sutherland [Sutherland 2008] esse é um ponto crucial em projetos com times distribuídos.

A utilização de metodologias ágeis como o Scrum tem sido uma forma adotada por muitas empresas para contornar os problemas enfrentados por times distribuídos, uma vez que as metodologias ágeis focam numa maior comunicação entre os membros da equipe e com o próprio cliente [Sutherland 2008]. Entretanto a própria utilização de métodos ágeis de forma distribuída pode ser vista como um problema, já que todas as formas de comunicação proposta devem ocorrer face a face e não de forma distribuída, com o objetivo de criar um senso de equipe e prover a autoconfiança e liderança dos membros.

Atualmente existem ferramentas que visam auxiliar as empresas no gerenciamento de projetos ágeis, dentre elas vale ressaltar o FireScrum, ferramenta open source desenvolvida inicialmente durante o programa de Mestrado de Engenharia de Software do CESAR.EDU. Após o desenvolvimento inicial, surgiu a necessidade de adicionar novos módulos e funcionalidades importantes com o objetivo de dar mais apoio a equipes distribuídas.

Um dos módulos acrescentado foi o Planning Poker. A implementação desse módulo foi realizada por uma equipe distribuída que aplicou a própria metodologia Scrum em todo seu desenvolvimento. Sua implementação desse módulo é contextualizada na Seção 5, que se refere ao estudo de caso.

Baseado nesse estudo de caso, esse trabalho tem como objetivo apresentar os problemas encontrados durante o desenvolvimento, como esses problemas foram resolvidos e como as soluções encontradas podem ser adotadas por outras equipes distribuídas. Assim como, fazer um paralelo entre as práticas do uso Scrum em equipes no mesmo espaço físico e os problemas dessas práticas quando aplicado o Scrum de forma distribuída.

Este trabalho está organizado da seguinte maneira: a Seção 2 apresenta detalhes sobre Desenvolvimento Distribuído de Software; a Seção 3 descreve uma visão geral do Scrum; a Seção 4 apresenta a o projeto de desenvolvimento da ferramenta FireScrum e suas características; na Seção 5 é apresentado o estudo de caso desse artigo, o processo de desenvolvimento do módulo Planning Poker do FireScrum; na Seção 6 são apresentadas os problemas encontrados e as soluções adotadas, e por fim na Seção 7 as considerações finais referentes ao estudo de caso.

2. Desenvolvimento Distribuído de Software

O Desenvolvimento Distribuído de Software (DDS) surgiu como uma alternativa para solucionar o problema da falta de mão-de-obra especializada das empresas. Como afirma Karolak [Karolak 1998], a demanda por aplicações cresce exponencialmente e a oferta de profissionais não tem sido suficiente para suprir a necessidade de tantos projetos.

O DDS busca acelerar o desenvolvimento e com isso reduzir o tempo de entrega, trazendo assim, uma maior satisfação para o cliente. Essa redução do tempo de desenvolvimento pode ser obtida através de equipes trabalhando em paralelo, assim como equipes trabalhando além do fuso-horário. Além disso, as empresas podem formar parcerias com o objetivo de explorar mercados maiores, além da facilidade em selecionar profissionais com competências específicas em qualquer espaço geográfico [Herbsleb 2001].

Mas, para que esse ambiente seja realmente produtivo, esforços adicionais na gestão do projeto são necessários para sincronizar o trabalho dos times, fazendo uso de tecnologias de comunicação e informação para gerenciar interdependências. O uso de sistemas de controle de versão, o uso compartilhado de ferramentas de comunicação e a integração dos resultados gerados são grandes desafios. Além disso, a intercomunicação cultural e linguística pode ser um dos grandes entraves na comunicação dos times distribuídos [Sutherland 2008].

3. Visão Geral do Scrum

O *Scrum* foi criado em 1996 por Ken Schwaber e Jeff Sutherland e destaca-se dos demais métodos ágeis pela maior ênfase dada ao gerenciamento do projeto. Reúne atividades de monitoramento e *feedback*, em geral, reuniões rápidas e diárias com toda a equipe, visando à identificação e correção de quaisquer deficiências e/ou impedimentos no processo de desenvolvimento [Schwaber 2004].

O *Scrum* é uma metodologia cujas práticas são aplicadas em um processo iterativo e incremental. Assume-se que os projetos no qual o *Scrum* se insere são complexos e difíceis de prever tudo que irá acontecer. Por essa razão, ele oferece um conjunto de práticas que torna tudo isso visível [Schwaber 2004]. Atualmente, é uma das metodologias mais utilizadas pela sua estrutura simples, de fácil aprendizado, com papéis claramente definidos [Schwaber 2004].

Na Figura 1, notamos que o *Scrum* inicia-se com uma visão do produto que será desenvolvido, contendo as características definidas pelo cliente, premissas e restrições. Em seguida, o *Product Backlog* é criado contendo a lista de todos os requisitos conhecidos. O *Product Backlog* é então priorizado e dividido em *releases*. Cada *release* contém um conjunto de requisitos, denominado *Sprint Backlog*, que será desenvolvido em uma iteração, denominada de *Sprint* [Marçal et al 2007]. Na execução da *Sprint*, diariamente a equipe faz reuniões de 15 minutos (*Daily Scrum Meeting*) para acompanhar o andamento do projeto. Ao final da *Sprint*, é realizada uma reunião (*Sprint Review Meeting*) de modo que o time apresente o resultado alcançado ao representante do cliente (*Product Owner*). Em seguida, o responsável por resolver os impedimentos do time (*Scrum Master*) conduz a reunião de lições aprendidas (*Sprint Retrospective*

Meeting) com o objetivo de que a equipe proponha melhorias de processo e/ou produto para a próxima *Sprint* [Marçal et al 2007].

Como em outras metodologias ágeis, o Scrum se fundamenta na interação constante e eficiente dos membros de um time. Por outro lado, com o crescimento contínuo do desenvolvimento de software por equipes remotas , fortemente visível desde a última década [Damian & Moitra 2006], torna-se essencial o uso de ferramentas de apoio que minimizem os impedimentos gerados na aplicação do Scrum como metodologia de desenvolvimento em equipes distribuídas. Assim, embora recomendada pela ScrumAlliance (www.scrumalliance.com), o uso de ferramentas de gerenciamento tradicional de projetos pode resultar no comprometimento de algum princípios do Scrum, como o apelo visual e simplista defendido pela metodologia, o uso de artefatos não automatizados como cartões e murais, podem representar um desafio adoção da metodologia por equipes remotas [Cavalcanti 2009]. Neste contexto uma boa iniciativa pode ser o desenvolvimento de ferramentas de apoio ao Scrum em equipes distribuídas [Cristal 2008].

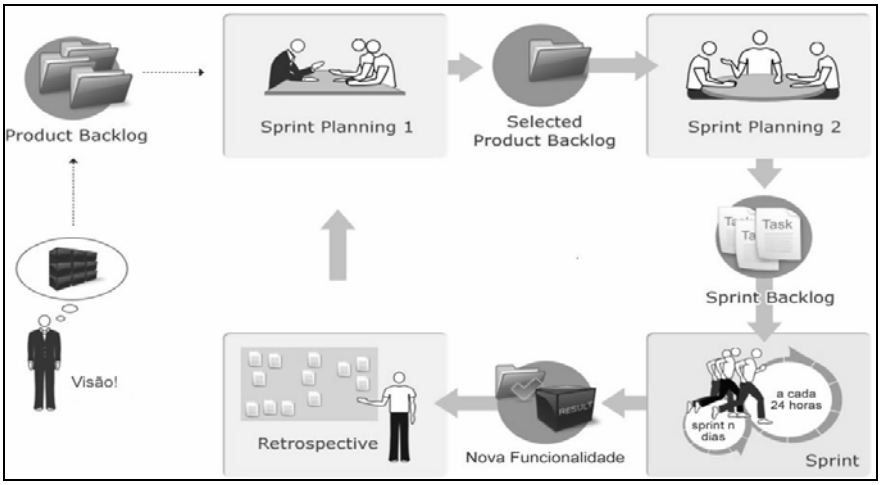


Figura 1: Visão geral do processo do Scrum (adaptada de [Gloger 2007])

4. O Projeto FireScrum

O FireScrum é uma ferramenta open source e foi desenvolvida para dar suporte ao gerenciamento de times *Scrums* distribuídos, evitando a perda de histórico das atividades, propiciando o levantamento de métricas sem demandar esforço excessivo do time e principalmente, facilitando sua adoção sem burocratizar o processo. O aplicativo utiliza a tecnologia RIA (*Rich Internet Application*) e tem como princípios: interfaces focadas em usabilidade e praticidade.

A ideia do FireScrum surgiu a partir de pesquisas realizadas como fundamentação para uma dissertação de mestrado de Engenharia de Software do CESAR.EDU, motivada pelas lacunas existentes em ferramentas similares de apoio a metodologia Scrum [Cavalcanti 2009]. A ferramenta é agora um produto INES (Instituto Nacional de Ciências e Tecnologia para Engenharia de Software) disponível no Sourceforge [SourceForge 2010].

Durante o desenvolvimento da dissertação, as funcionalidades foram implementadas por 60 pós-graduandos da disciplina de Engenharia de Software [IN953, 2009] do programa de pós-graduação do Centro de Informática (CIn) da Universidade Federal de Pernambuco (UFPE), a partir da qual foram desenvolvidos cinco novos módulos, totalizando seis com o já criado módulo core, como ilustrado na Figura 2.

Durante o desenvolvimento do projeto os 60 integrantes da disciplina foram distribuídos em seis times responsáveis, cada um, por um módulo do sistema, onde cada time funcionava como um fábrica de desenvolvimento independente com um Scrum Master cada. Os Scrum Masters de cada time de desenvolvimento se reuniam para reuniões entre "Scrums Masters". A proposta da disciplina foi simular um ambiente próximo da realidade da indústria de software, proporcionando aos alunos o exercício da engenharia de software (Conceitos, Métodos, Práticas e Ferramentas), permitindo aos estudantes adquirirem uma experiência prática de vivencia de problemas e imprevistos que acontecem em uma fabrica de software e que precisam ser resolvidos.

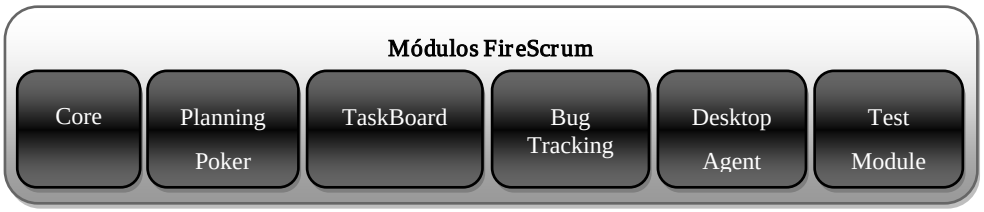


Figura 2: Módulos que compõem o FireScrum

5. Estudo de Caso – Módulo Planning Poker

A fábrica composta por nove alunos do Mestrado ficou incumbida de desenvolver o módulo Planning Poker da ferramenta FireScrum, utilizando a metodologia Scrum, de forma distribuída, já que os integrantes estavam dispersos entre quatro cidades diferentes. Essa abordagem possibilitou identificar problemas e propor soluções que foram integradas à ferramenta, o que possibilita melhor suporte ao Scrum em ambiente distribuído.

O processo utilizado foi: A cada início de *Sprint*, eram acordadas, com o *Product Owner*, as estórias que seriam desenvolvidas, assim como suas priorizações, a partir daí, eram feitas reuniões de definição de funcionalidades, arquitetura e atividades necessárias para a implementação. Essas atividades ficavam disponíveis em uma planilha compartilhada via web, com a informação do responsável, status e tempo estimado, fazendo com que todos pudessem analisar e atualizar o andamento das suas atividades. Ao término de cada *Sprint*, reuniões de *Sprint Review* e *Retrospective* eram realizadas, para que fossem identificados os erros e acertos, assim como os pontos de melhoria para o próximo sprint. Uma vez por semana, era feita a reunião *Scrum* de *Scrums*, em que os seis *Scrum Masters* se encontravam com os responsáveis da disciplina para resolverem os impedimentos das suas respectivas fábricas.

Uma prática muito comum no *Scrum*, durante o *Sprint Planning 1*, é o *planning poker*, uma técnica para se realizar a estimativa de estórias e tarefas. É uma atividade importante, pois é nesse momento que todo o time se reúne para entrar em um consenso sobre as estimativas das estórias, o que normalmente gera muitas discussões. Com isso,

todo o time passa a ter um entendimento maior do que será realmente necessário fazer, além de facilitar o entendimento de como as atividades deverão ser desenvolvidas.

O objetivo do módulo do *Planning Poker* é simular sessões de estimativas, de modo que os participantes possam discutir e argumentar como se estivessem reunidos presencialmente. Para isso, foi criado um ambiente com recursos multimídia, como áudio, vídeo e chat, de forma a facilitar a comunicação do time.

Para as estimativas, o usuário define a sequência de cartas a serem usadas, exemplo, a sequência de Fibonacci, assim como é feito na reunião presencial. A reunião é inicializada pelo *Scrum Master*, na qual todas as estórias referentes ao produto são disponibilizadas. Para que a estimativa seja iniciada, o time se conecta à sessão criada, o *Scrum Master* seleciona uma estória, cada membro do time a visualiza e escolhe uma carta. Com o objetivo de simular o ambiente real, a imagem dos usuários conectados aparecem na tela e as suas respectivas cartas só são mostradas depois que a votação é encerrada. O chat pode ser usado para a discussão/argumentação, bem como os recursos de áudio e vídeo, que são controlados pelo *Scrum Master*, já que ele é o moderador da *Sprint Planning*. Em resumo, estas são as funcionalidades do módulo Planning Poker, representadas nas Figuras 3 e 4.

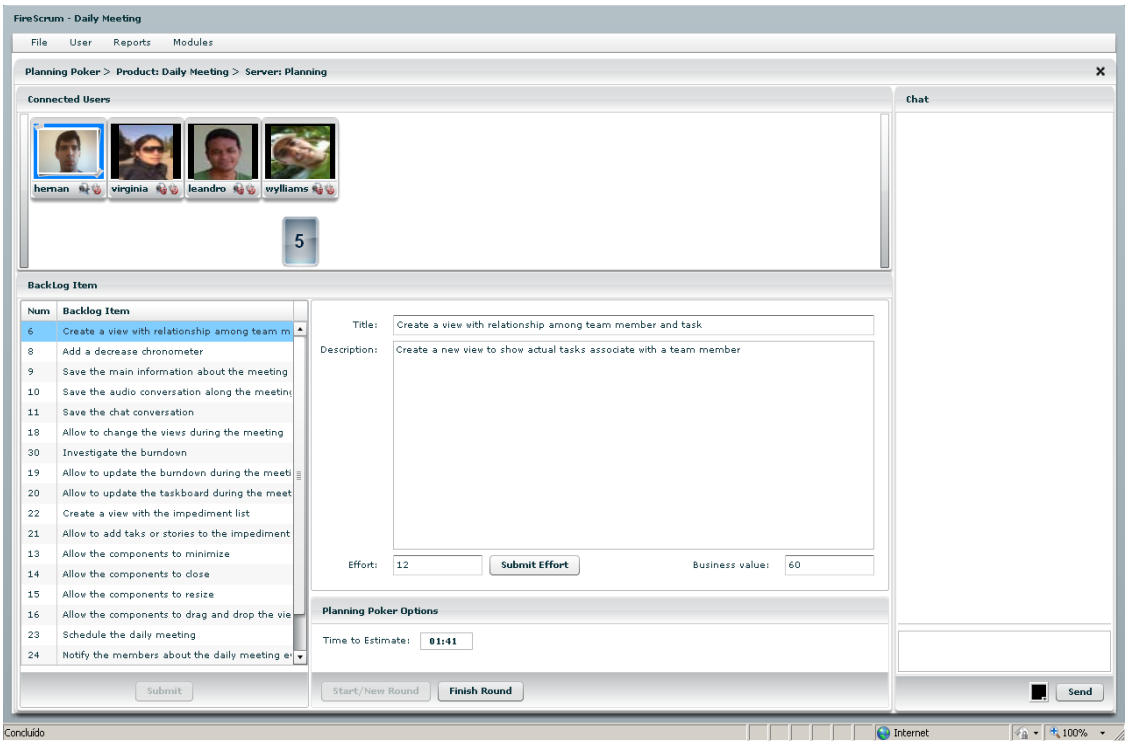


Figura 3: Tela do Scrum Master

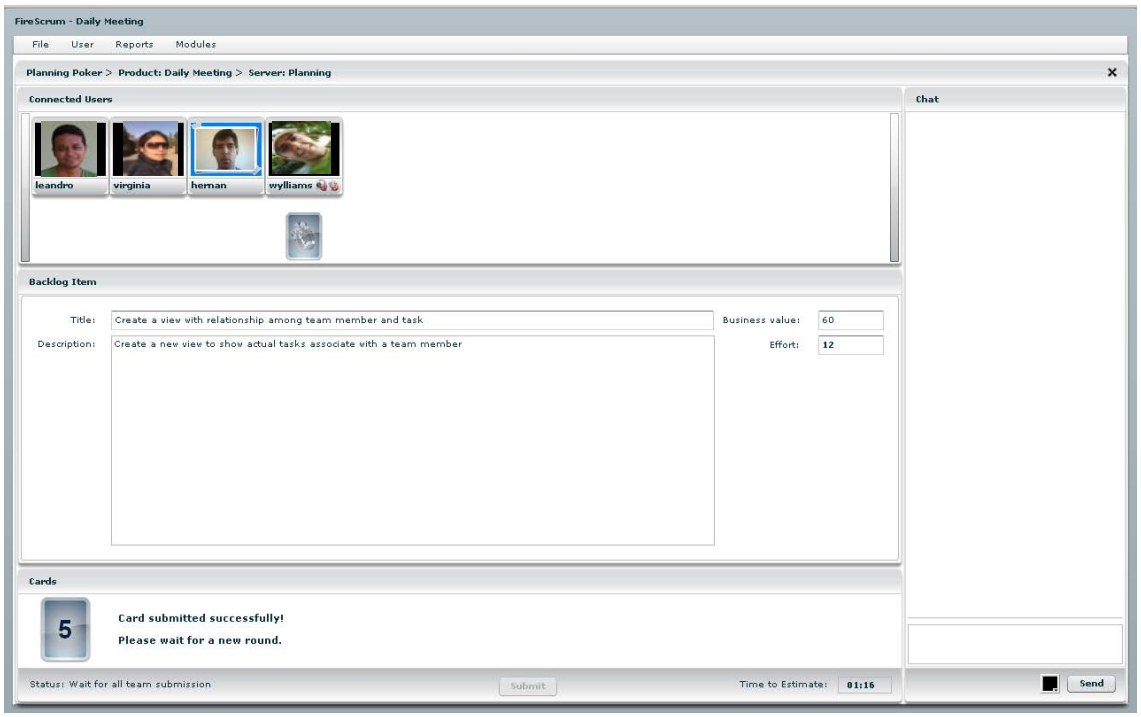


Figura 4: Tela do Time

6. Problemas Enfrentados e Soluções Encontradas

São apresentados a seguir, os problemas enfrentados, assim como as soluções encontradas pela fábrica durante o desenvolvimento do módulo. No total, foram realizadas cinco *Sprints*, de duas semanas cada uma.

Na primeira *sprint*, foram iniciados os planejamentos, no entanto o time não tinha total proficiência com a metodologia e tecnologias adotadas na implementação do projeto. O primeiro grande desafio da fábrica foi o nivelamento técnico de seus integrantes nas práticas do scrum, assim como nas tecnologias utilizadas no projeto. Para tal, workshops foram elaborados e o estudo das tecnologias utilizadas foi incluso no sprint.

Questões de comprometimento e interação entre os integrantes da equipe são pontos fortes na metodologia Scrum, no entanto a fábrica enfrentou a problemática relacionada ao não comprometimento de duas, dentre os dez integrantes. Contudo, reestimativas das atividades tiveram que ser realizadas com base no esforço de oito membros. Isso é demonstrado pelo gráfico de *burndown* apresentado na Figura 5, na qual a linha mais clara, que representa as atividades pendentes, se distancia muito da linha mais escura, que representa um prazo ideal para a realização das atividades.

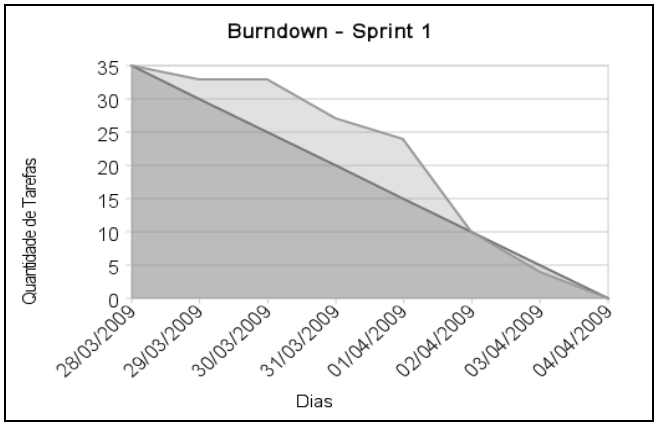


Figura 5: Burndown da *Sprint* 1

Um aspecto bastante relevante a ser destacado nos gráficos de burndown é a aproximação da linha de estimativa e a linha real. A previsão de esforço para cada atividade, ao longo das *sprints* foi se tornando mais precisa. Isso mostra o ganho de experiência do time, tanto nos aspectos técnicos, quanto no autogerenciamento, conforme Figura 6.

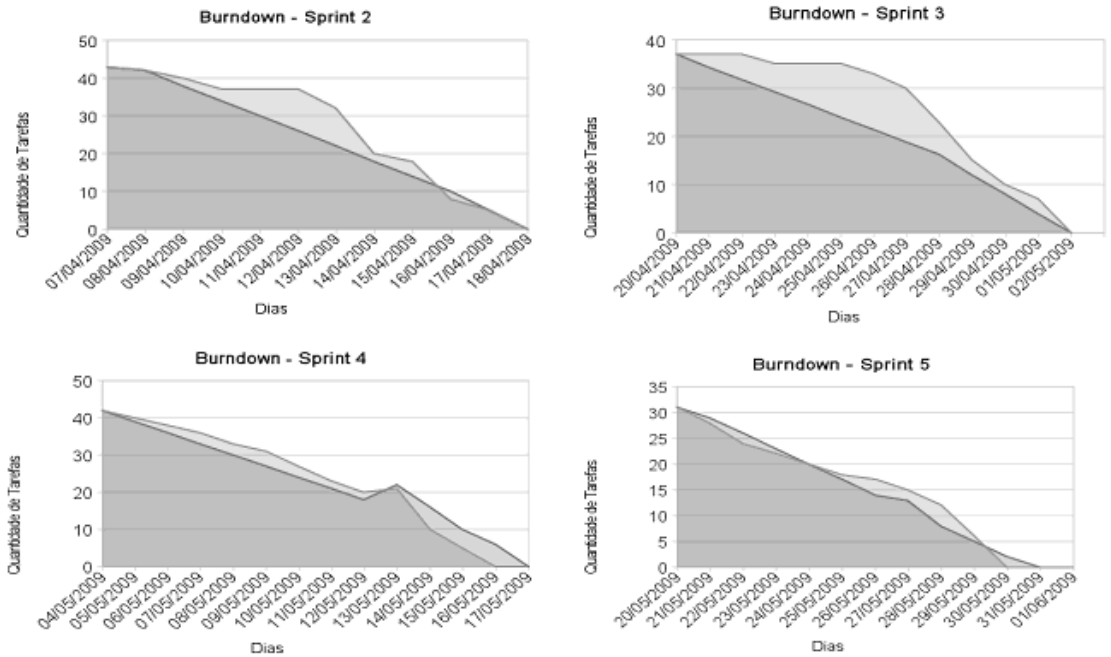


Figura 6: Burndown das *Sprints 2, 3, 4 e5*

Podemos observar na tabela a seguir, os principais problemas enfrentados, assim como as principais soluções adotadas para minimizar os conflitos relacionados às práticas Scrum, quanto utilizados em um projeto de forma distribuída.

Problemas	Scrum	Soluções
Equipe geograficamente distribuída	Um dos aspectos definidos no Scrum é a unidade do time de desenvolvimento. Aspecto estritamente relacionado a presença física do time com atividades de monitoramento e feedback, em geral, reuniões rápidas e diárias com toda a equipe, de forma a identificar e corrigir possíveis deficiências e/ou impedimentos no processo de desenvolvimento [Schwaber 2004].	O desafio de proporcionar a efetiva interação entre os membros do time geograficamente distribuído foi obtido através do uso sistemático de listas de discussões e ferramentas de comunicação via web. Estas abordagens aliadas ao comprometimento e pro atividade do time, aspecto importante na metodologia Scrum, possibilitaram uma boa interação entre os membros do time.
Reuniões diárias	O Scrum prevê reuniões diárias presenciais de quinze minutos entre o time para que todos respondam às três perguntas básicas.	Esta característica do Scrum torna-se complexa quando aplicada em um time geograficamente distribuído. No entanto, este problema foi solucionado através do uso de ferramentas de comunicação e conferência. Porém, o controle por parte do Scrum Master nesta atividade foi de vital importância intervindo e gerenciado a conferencia.
Visibilidade	Todos os membros do time vêem o quadro de atividades, onde podem visualizar e manipular o quadro de atividades de forma prática e rápida.	Uma planilha compartilhada com todas as atividades planejadas foi utilizada para que cada membro cada membro realizasse a atualização de status de suas atividades, mantendo assim a transparência entre todos os integrantes.
Atribuições do Scrum Master	Atribuição de auxiliador da equipe e Product Owner, além de resolução de impedimentos.	Com o time geograficamente distribuído aumentam as atribuições e responsabilidades do Scrum Master que, neste contexto, além de ter parte de seu tempo dedicado ao gerenciamento dos impedimentos reportados pelo time, também precisa estar atento a eficiente interação entre os membros.
Acoplamento dos componentes do sistema e das atividades	Embora exista no desenvolvimento tradicional com Scrum, o acoplamento das atividades pode ser facilmente solucionado quando se está em constante contato com o responsável pela tarefa dependente, pois as equipes trabalham de forma integrada.	A prática de distribuir os requisitos do sistema de forma que proporcionasse o mínimo de dependências entre as atividades dos membros, minimizando os impedimentos por atividades relacionadas.

Tabela 1: Soluções adotadas para o desenvolvimento distribuído com Scrum

Como foram relatados na Tabela 1, alguns problemas foram enfrentados ao longo do desenvolvimento distribuído como podem ser vistos na coluna 1. Muitos desses problemas possivelmente não existiriam numa equipe Scrum tradicional como descrito na coluna 2, porém a coluna 3 explica as ações adotadas pela equipe para contornar esses problemas. Desse modo acreditamos que essas ações poderiam ser aplicadas por outras equipes distribuídas e que através de um maior apoio ferramental, principalmente durante as reuniões diárias, ajudariam a amenizar os problemas enfrentados por equipes distribuídas.

7. Considerações Finais

Com a necessidade do mercado por sistemas mais complexos em um curto espaço de tempo, alinhado com os requisitos que mudam a todo instante, além da necessidade de captação recursos humanos de diferentes localidades, levaram as empresas a adotarem essa combinação de desenvolvimento ágil e distribuído com o objetivo de reduzir o tempo de entrega e custo, trazendo muitos benefícios, mas em contrapartida, enfrentando uma série de problemas.

Esse trabalho descreveu a experiência obtida ao desenvolver um módulo de uma ferramenta open source dentro desse cenário distribuído. Durante o seu desenvolvimento, problemas foram encontrados, relatados e foram amenizados ao longo do desenvolvimento das sprints.

O principal objetivo desse trabalho foi destacar os problemas encontrados, levantando eventuais soluções que amenizariam tais problemáticas e como essas soluções poderiam ser aplicadas por outras equipes distribuídas. A partir dos problemas e soluções, foi possível fazer um paralelo entre as práticas do uso Scrum em equipes no mesmo espaço físico e os problemas dessas práticas quando aplicado o Scrum de forma distribuída. Dentre os problemas podemos destacar o gerenciamento da equipe e comunicação entre os membros distribuídos. Em consequência desses problemas, a equipe sentiu falta de um apoio ferramental, o que motivou o desenvolvimento da própria ferramenta FireScrum.

Sendo assim, esse trabalho apresenta como contribuição, possíveis soluções que podem ser adotadas por equipes distribuídas que adotam as metodologias ágeis, assim como os principais problemas enfrentados pelo Scrum tradicional quando aplicado de forma distribuída. Desse modo, como o desenvolvimento distribuído e o Scrum são fortes tendências do mercado de software, o relato dessa experiência pode colaborar tanto com fábricas de software que já estão ou planejam estar alinhadas com essa tendência, quanto com a área acadêmica que tenha um foco mais prático, almejando desafios iguais ao do mercado real.

A partir dos problemas encontrados durante o estudo de caso, algumas melhorias foram identificadas para tornar o FireScrum mais colaborativo e dar mais apoio as equipes distribuídas. Desse modo, como trabalho futuro, é importante que o FireScrum forneça mais apoio a etapas importantes na interação e comunicação entre os membros da equipe como ambiente para discussão, interação entre os integrantes e o cliente. O mesmo pode dar apoio as reuniões diárias, sprint review e retrospectiva.

Referências

Carmel, E. (1999) Global Software Teams: Collaboration Across Borders and Time Zones. Prentice-Hall, EUA.

Cavalcanti, E. O.; Maciel, T. M. M.; Alburquerque, J. O. (2009). “Ferramenta Open-Source para Apoio ao Uso do Scrum por Equipes Distribuídas”, III Workshop de Desenvolvimento Distribuído de Software (III WDDS).

Cristal M., Wildt D., Prikladnicki, R. (2008) “Usage of Scrum Practices Within a global Company”, "Usage of SCRUM Practices within a Global Company", 2008 IEEE.
International Conference on Global Software Engineering, pp. 222226.

Damian, D., Moitra, D. (2006). “Guest Editors' Introduction: Global Software Development: How far Have We Come?”, IEEE Software, 23(5), pp.1719.
Gloger, B. (2007), “The Zen of Scrum”, http://www.glogerconsulting.de.

Herbsleb, J. D.; Moitra, D. (2001). Guest editors’ introduction: global software development. IEEE Software.

IN953 (2009). Software Engineering: Building Open Source Software Factories. Disponível em http://www.cin.ufpe.br/~in953/.

Karolak, D. W. (1998) Global Software Development – Managing Virtual Teams and Environments. IEEE Computer Society, EUA.

Marçal, A. S. C.; Freitas, B. C. C.; Soares, F. S. F.; Maciel, T. M. M.; Belchior, A. D. (2007) “Estendendo o SCRUM segundo as Áreas de Processo de Gerenciamento de Projetos do CMMI”, In: CLEI Eletronic Journal.

Schwaber, K. (2004), Agile Project Management with Scrum, Microsoft Press.

SourceForge (2010) “SourceForge.net”http://sourceforge.net/projects/firescrum/develop , Acessado em Janeiro/2010.

Sutherland, Jeff. (2008) Fully Distributed Scrum: The Secret Sauce for Hyperproductive Offshored Development Teams. Agile 2008. IEEE Software.

Conducting an Architecture Group in a Multi-team Agile Environment

Mauricio José de Oliveira De Diana[1], **Fabio Kon**[1], **Marco Aurélio Gerosa**[1]

[1]Department of Computer Science – University of São Paulo (USP)
São Paulo – SP – Brazil

{mdediana,kon,gerosa}@ime.usp.br

Abstract. *An important agile principle is that the best designs and architectures are emergent. It is completely feasible in low-complexity systems and / or in a single development team context. However, when systems grow, development is usually split among many teams. When it happens, implementing agile principles that lead to emergent architecture becomes harder. Although there is discussion on scaling agile in general, it is not clear how to deal specifically with software architecture in a multi-team scenario. This article presents an experience report on how a medium-sized company created an architecture group to tackle this problem, and how this group adopted the same agile principles and practices that development teams use in its arrangement and operation.*

1. Introduction

In Agile Methods, the design of an application starts as simple as possible and incrementally evolves from that moment on. Practices such as automated testing, continuous integration, and refactoring together with simplicity principles such as "do the simplest thing that could possibly work" and "You Aren't Going to Need It" (YAGNI) [Fowler 2001] are the foundation of emergent design, an important agile principle [Beck et al. 2001].

In a single-team scenario, an emergent architecture is feasible. However, when development involves a bigger system composed of sub-systems built by several teams, systems architectures become more complex. In such a scenario, the simplest solution for one team may not be so simple for other teams. In this case, the lack of a big picture may jeopardize the whole system conceptual and architectural integrity [Brooks 1975, Bass et al. 2003]. When this vision does not exist, it is hard for those independent teams to find adequate solutions to their architectural issues just by talking to each other and expecting that the whole system architecture will emerge.

In this situation, the first approach that may come to mind is to name a chief architect or create an architecture board, responsible for defining and enforcing the whole system's architecture. The main drawback of this approach is that it creates the necessary conditions to the rise of an anti-pattern observed by Phillipe Kruchten, Ivory Tower architects – architects disconnected from the development day-to-day reality [Kruchten 2008]. This anti-pattern can hurt teams' self-organization, teams' motivation, and architecture emergence, all cornerstones of Agile Methods.

We addressed the problem of handling architecture in a multi-team agile environment in a medium-sized company by creating an architecture group composed of team representatives. To manage the group, we adopted the same agile principles and practices

that work for development. In this article, we show how the group is organized, what it delivers, how it operates, and the lessons we have learned. We believe this approach may be useful for other multi-team organizations facing the challenge of balancing software architecture concerns and agility.

Section 2 describes the scenario which led to the creation of the architecture group. Section 3 explains the group's organization and lists its objectives. Section 4 describes the expected group's deliverables, while section 5 describes the group's practices and operation. Section 6 presents alternative approaches on architecture and multi-team agile. Section 7 concludes this article reporting the main lessons learned.

2. Scenario

Locaweb is a 10-year-old Brazilian web hosting provider, the leader in its market with 22% market share and an annual growth of about 40% over the last few years. The company has around 100 developers in its IT department, who produce applications such as control panels and account administration tools for the company's customers and operational support systems such as provisioning and billing. For many years, the company used an ad hoc software development method or, in better terms, no method at all. Developers were allocated and deallocated to tasks as needed. There was no notion of teams nor projects.

The lack of methodology was not a crucial issue while the company was still small. But as the company grew, so did the complexity of its products and back-end systems. Slipped schedules and quality problems in the IT department became more frequent. Systems maintenance costs rose steadily due to bad design and workarounds. Big Balls of Mud [Foote and Yoder 1999] were common. This situation directly impacted business growth. Good ideas, from a business point of view, were discarded due to their high implementation costs.

By the end of 2007, the company restructured the IT department to fix this situation. After many presentations and discussions, the company decided to adopt agile methodologies, and Scrum was chosen for its ability to organize the development from a managerial point of view, with little dependence on technical issues. It would be harder to implement a method such as Extreme Programming (XP) in the early stages, as most of the codebase was composed of old web scripting technologies such as ASP, which would make essential XP practices, such as automated testing, very hard to adopt.

The department was divided into 13 teams, each one with up to 9 developers and a development manager. These teams and the systems they build are loosely coupled, each team being responsible for everything regarding their respective systems, including the software architecture – we stuck to XP's practice of Whole Team as much as possible [Beck 2005]. Most of these systems are constantly evolving as the company's product portfolio and the business rules change. Although the teams must work as independent as possible, some coordination between them is necessary in crucial situations such as integration with central provisioning and billing systems, which is a must for virtually every system. At first, to decide on issues affecting the whole department, a department board composed of development managers, the product management director, and the CTO was created.

Agile practices usually need to be adapted to the context where they are being

used; what perfectly works for one team may not work for another. Therefore, during the following year, each team experimented with different practices and learned what worked best for them. Each team shared the lessons learned with each other, with the department board serving as the formal channel to exchange experiences. Team independence and autonomy grew considerably as the process improved. Teams were truly becoming Whole Teams, with each responsible for system development from inception to deployment, and from maintenance to operation. In other words, they had total control of their own work.

That environment helped to resolve many problems for each team, including architectural ones. Although individual subsystems improved, the architecture of the whole system, composed of each subsystem and their interactions and interfaces, was still undefined and unmanaged. For instance, most systems communicated with others, usually through RPC. But there were no constraints on what protocols the systems should use. Consequently, a team could build a system using W3C Web Service standards and another team would use an XML-RPC based protocol they created. Any system that needed to talk to both would have to implement both communication mechanisms. Such duplication is wasteful, since implementation costs rise and no real value is added. Furthermore, this approach is error-prone as each new protocol increases system complexity. Each team embraced the XP value of Simplicity [Beck 2005], however what would be the simplest solution for a team often imposed complexity upon the others.

In a scenario with only a few teams, it is relatively easy for each to know each other's particularities and to work out architectural issues as they arise. Naturally, the agile principle that states "the best architectures, requirements, and designs emerge from self-organizing teams" [Beck et al. 2001] would be the most appropriate approach. But with 13 teams in place, the company noticed that communication, and consequently decision making, became cumbersome. It was even more difficult because the teams did not share the same technical background. The company presents a wide range of platforms and programming languages, and teams specializes in one or two of them (e.g., Java or .NET). Self-organization inside each team worked well, but the same could not be said at the department level.

3. The Architecture Group

The company needed to find an approach to balance architectural issues and team agility. The first attempt was to centralize cross-system architectural decisions via the department board. However, the board was composed of managers who, despite having some technical background, had little technical responsibility on their teams. They usually played the role of product owner or project manager. The most technically skilled member on many teams were not necessarily the manager.

In light of this, the company created the Architecture Group, composed of each team's most skilled members, appointed by their respective development managers, and a leader responsible for facilitation, not decision making. The first author of this article has been the architecture group leader since its inception. Due to the difficulties with large scale self-organization, we believe this approach provides an effective way to handle architectural concerns in an environment with many teams building interdependent systems. We believe this approach stands valid while the architecture group size is at most twenty members. If the group grows beyond that, communication problems will likely arise, and

a scale-up strategy may be necessary [Lindvall et al. 2002].

According to Bass et al. [Bass et al. 2003], "the software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them." This definition is very important for the group to understand the relationship between each subsystem architecture and the whole system architecture, and consequently understand how the group actions must take place. The group also sees value in Martin Fowler's more relaxed definition [Fowler 2003] that states that architecture is the set of "things that people perceive as hard to change." Such a broad definition helps the group tackle important issues that may fall outside more formal definitions. Linked to Fowler's definition, another important concept is the difference between an emergent and an evolutionary architecture [Ford 2009]. Since the process of creating the architecture involves addressing what will be hard to change later, to depend on emergence to get there is risky. But it does not mean that all architectural decisions must be made and implemented from the beginning of a project. If a team constantly pays attention to the architecture, it can be implemented incrementally, evolving over time as needed.

The group is mainly concerned with three topics, namely architectural management, development quality, and technical debt. We explain them in the next subsections.

3.1. Architectural management

The architecture group cares about the whole system architecture. From the group's point of view, the subsystems built by each team are the software elements from the Bass et al. definition. This vision bears strong resemblance to the Architecture Team pattern [Coplien and Harrison 2004] defined by James Coplien and Neil Harrison, where "the architecture team's task is to create a high-level partitioning. Much architectural work remains to be completed at lower levels." In this respect, the architecture group influences each team at the boundaries, including issues such as systems integration mechanisms, operational support systems, and quality attributes that must be met by any system. The group does not make, review, nor approve decisions on specific systems architectures. Each team is still responsible for its systems architectures. In reference to Fowler's classification [Fowler 2003], the group is closer to the way the Architectus Oryzus works (whose "most noticeable part of the work is the intense collaboration") than to the Architectus Reloadus (who "is the person who makes all the important decisions"). Fowler states that "an architect's value is inversely proportional to the number of decisions he or she makes." This attitude is aligned with the agile concept of an empowered team, where the developers know the best course of action in each context (something confirmed by our observations).

It is crucial to recognize that it is difficult to make good decisions when detached from day-to-day reality. This was the main reason for creating an architecture group composed of members from all teams instead of creating a separate architecture team, a Community of Practice as advised by Craig Larman and Bas Vode [Larman and Vode 2010]. In describing what a Core Architecture Team is, Scott Ambler says that creating such a group "helps to increase the chance that each subteam learns and follows the architecture as well as increases the chance that the core architecture team will not ignore portions of the system" [Ambler 2002].

3.2. Development quality

Even though the architecture group is mainly concerned with architectural issues, it has also a stake in each team's development processes. As such, its second goal is to define development quality standards. In a distributed systems environment, the lack of quality in a specific system can leak and affect other systems. For example, if a system is poorly tested, it will present more defects, which in turn decreases its reliability. Therefore, the group is concerned with quality in all subsystems development. An effective way to improve quality is the systematic use of software development best practices and testing tools at all levels where validation is necessary. Therefore, the group demands actions from teams regarding testing coverage, code standards, and continuous integration, for example. A software development quality group may be a more suitable way to handle development quality issues than the architecture group. At the time of this writing the department is investigating creating such a group, but until it is completely functional, the architecture group undertakes this responsibility.

3.3. Technical debts

At the time of this writing, many teams struggle with technical debts in their systems. Technical debt is a metaphor to show the costs involved in bad design and/or implementation [Fowler 2004]. The idea is that whenever developers neglect the design of their system, they incur in technical debt. In financial debts, there is interest payment. The same happens with technical debts – the longer it takes for the developers to correct the results of bad decision making from the past, the more interest they will have to pay, and the more expensive the correction will become.

Although paying down technical debts is each team's responsibility, it is common for legacy systems to impede or even prevent the appropriate architectural decisions. Because of this, the group set a temporary goal of supporting teams in paying down technical debts. The group can be especially helpful in situations where specific architectural debts affect many systems. For example, high coupling between systems by different teams makes it difficult for them to maintain and evolve those systems. In such cases, the group helps coordinate efforts to cleanly separate the systems, usually by designing strategies for legacy migration that concerns more than one system. In addition, this help can take the form of informal consultancy services that members offer to each other.

Upon recognizing that much technical debt was the direct result of lack of knowledge and experience, the department implemented many training and knowledge sharing initiatives. By helping its developers to become more knowledgeable, the company expects them to both attack existing debts and to build quality into new systems from the ground up in order to avoid new debts. Although the architecture group is not responsible for training, it strongly supports many such initiatives.

4. Architecture Group Deliverables

The architecture group does not decide on subsystems architecture nor produce code. Instead, its members discuss what they see as important issues affecting the whole system architecture, analyze ways to address those issues, and request teams to take actions on them. The group basically delivers these requests in three different forms: standards, goals, and definition of shared infrastructure.

4.1. Standards

Standards definition is the preferred way for the group to manage the whole system architecture. The group uses the same principles behind standardization groups such as IETF and the Java Community Process, i.e., it states what should be done, but not how. In other words, the group does not make recommendations regarding implementation details, it just defines the interfaces between the subsystems.

The group asserts the quality of its standards by observing each group's adherence to them. One of the group's key principles is that if developers do not see value in what the group is advocating they will just bypass it, especially in an environment where autonomy is so intense. This is similar to when de facto standards become more popular in an industry than their formal counterparts. Consequently, whenever a standard is not broadly adopted, the group understands that it failed to create something of value. With that in mind, the group tries to make all the standards as lightweight as possible and then let them evolve naturally. For example, when discussing a substitute for the in-house systems integration mechanism, the group adopted RESTful web services, the simplest solution it could find that would meet the requirements.

4.2. Goals

Another important mechanism put in place by the architecture group is quarterly goals. Every three months the group members discuss the most important issues affecting their teams, looking for commonalities. Based on that, the group defines goals to be achieved by all teams with the aim of solving two different problems. The first one is related to architectural integrity. Goals are a good way to know that every team has worked on a common issue. For example, if teams require information for capacity planning, a possible goal may be that every system must have an attached monitoring component to gather operational information. The second problem is related to backlog prioritization. Our experience shows that although product owners see value in architectural development, when it is completely up to them, they may discard architectural work to make room for user stories contemplating new features, which are more valuable from the customer point of view. Furthermore, they rarely have the technical knowledge to evaluate architectural activities, so it is hard for them to prioritize them. The quarterly goals make the architecture group a project stakeholder as any other to each team.

The group maintains an evolving architecture manifesto which lists the requirements every subsystem must conform to. At the end of each quarter, the goals related to quality attributes are added to the manifesto. By doing that, the architecture group is periodically incrementing and standardizing what is expected from all subsystems architecture. Creating a manifesto incrementally instead of trying to completely define it from the first moment is a pragmatic way to assume what is feasible at the moment and to always focus on the priorities.

The quarterly goals are composed of a few specific items, usually sharing a common theme. For example, the goals for Q1 2009 had the theme "build the basic infrastructure needed to support development (continuous integration servers, code coverage tools, etc.) and operations (monitoring and statistics)", and were:

- All projects that are not considered legacy systems must have continuous integration in place.

- All projects not considered legacy must have their testing coverage measured.
- All production systems infrastructure must be monitored by an alert system (e.g., Nagios).
- Statistics about all systems not considered legacy must be harvested and displayed by a graphical information system (e.g., Cacti).
- Each team must prepare a lightweight plan describing its actions regarding their legacy systems.

4.3. Definition of shared infrastructure

Many systems share common needs, especially infrastructural ones. For example, a single sign-on solution is desired so the customers, who access many distinct web applications, do not need to login again every time they switch from one to another. Another example is the definition and creation of a system integration environment where each team can test their systems by running their automated test suite using other subsystems. In these situations, the group defines which team should be responsible for this particular concern. Usually a member from the team which will get the most from it steps in as volunteer to do the job. The team may be helped by developers from other teams while implementing the solution, but it will be the only responsible for the maintenance of the system after its deployment.

5. Conducting the Architecture Group

The architecture group uses agile principles and practices that work for development teams, i.e., concepts such as iterations, backlog prioritization, planning, and review meetings. Since its actions happen across many teams and are perceived in the long term, the group adopts these concepts in another scale – for example, instead of 1 or 2-week iterations, monthly ones; instead of daily meetings, bi-weekly ones. Following are some adaptations that the group has made to other practices as well.

5.1. Self-organization

In an empowered team, all technical decisions are collectively made by the team members. The same happens in the architecture group, its leader basically has coordination and facilitation responsibilities, not decision ones. Decisions are not made by majority either, since the group believes that in many situations one or two group members have special knowledge that others may not have, so all members have to discuss until rough consensus is reached. It is the same principle behind Planning Poker [Cohn 2005]. In very rare situations when consensus is not reached the leader makes a decision so the group does not get stuck in endless discussions. But the group believes these situations actually are "bad smells" (similar to code smells) – after some time, it discusses "what happened that we couldn't agree on something even after discussion?". We have noticed that it can be for many different reasons, lack of knowledge on the subject being the most common.

5.2. Demand management

The group has a backlog open to the whole company in an internal wiki. Anyone, group member or not, can add items to it. The backlog is prioritized by the group's leader after listening to other members and stakeholders ranging from the CTO to product owners. A constant worry is to always be aligned to the business, the group should attack the most relevant issues from the company's point of view. So the group considers any demand coming from the outside a very positive sign that it is on the right track.

5.3. Planning and review

The group runs planning and review meetings, occurring every other week, with different purposes. At the beginning of the month there is a 3-hour meeting, mandatory to all group members, when work-in-progress is reviewed and new work is started. After two weeks, the group meets again in an 1-hour review meeting, just to follow-up ongoing work. Presence in this meeting is optional. Since all members have about 10% of their time dedicated to the architecture group, this frequency is sufficient to keep them and their teams informed. These meetings are also a good moment for teams to share what they have been working on regarding architecture specifically, or general issues affecting architecture. In a way, it works as an informal Scrum of Scrums, but focused on the engineering side of ongoing projects.

5.4. Working groups

The group quickly learned that it is almost impossible to keep long technical discussions productive with around 20 people talking. Moreover, many of the discussed topics need profound analysis and research. To deal with this situation, the group found inspiration in the way IETF manages RFCs [Bradner 1998]. Working groups are typically created to address a specific problem or to produce one or more specific deliverables (a guideline, standards specification, etc.). Working groups are generally expected to be short-lived in nature. After the tasks are completed, the group is disbanded. However, if a working group produces a Proposed or Draft Standard, it frequently becomes dormant rather than disband. When the deliverable produced by the working group is a draft, it is presented at the next review or planning meeting to public appreciation.

Not everything needs a working group to be worked out. Actually, the group solves most issues during the planning meeting. Working groups are created when the group notices that a particular issue will not be solved with a quick discussion or when it takes too long to reach an agreement. The group leader and up to six volunteers comprise the working groups. Normally, someone volunteers for a working group due to two reasons: their team may have a stake in the target issue or they can be particular fond of the subject. For example, it would be expected that a working group formed to address user access policies has members from the core team, the hosting team, and people interested in security. The usual working group's mode of operation is to meet once a week to discuss the issue and divide research and/or development between its members until their next meeting.

6. Related Work on Architecture and Multi-team Agile

While agile methods were seeing broader adoption in industry, their first limitations started to show up. How to apply agile methods away from their "sweet spot" (single collocated teams, greenfield projects, etc) became a common concern for practitioners and academics, and we have seen the growth in literature on how to scale agile as a consequence. Scaling agile usually involves a wide range of concerns, we limit this section to works which deal with architecture in multi-team agile environments.

Dean Leffingwell's approach [Leffingwell 2007] is to organize systems in components, and have the components assigned to teams. From that, an architecture team, formed by senior architects or technical leaders from the teams, defines the initial architecture in the beginning of the project. A prototyping team may also be formed to test the

architecture. The process follows important agile concepts such as working in time-boxed iterations and focusing primarily on implementation rather than just modeling. After the initial phase, work is assigned to development teams. They start by building what Leffingwell named the architectural runaway, which "contains existing or planned infrastructure sufficient to allow incorporation of current and anticipated requirements without excessive refactoring." It is expected that the runaway get extended to accommodate new features. However, development teams must not necessarily be responsible for extending the runaway. If architectural expertise is mainly concentrated in the architecture team, this team may work in advance and prepare the runaway so development teams find the needed infrastructure ready when they start to implement a new feature. Leffingwell's approach seems more suited to deal with scenarios where systems are more tight coupled, teams work closer to each other and there are few ongoing strongly related projects at a time, a scenario different from ours.

Jutta Eckstein, following Frederick Brooks' observations, recognizes that a system's conceptual integrity is lost when a team becomes too large [Eckstein 2004]. To avoid that, she suggests the assignment of a lead architect. This person is responsible for making final architectural decisions, remembering the rationale behind those decisions, and, most importantly, spreading the system's architectural ideas so people have a better understanding of its architecture. Besides the presence of the technical lead, Eckstein suggests what she calls architecture as a service. The term means that an architecture team's work should be guided by, instead of guide, the development teams' work. The architecture team build only what is requested by the development teams that, by their turn, only ask for work related to their respective customers. By doing that, an organization sticks to the YAGNI principle, keeps its architecture simple, and ease the burden on developers to understand the architecture. Eckstein's approach seems a good fit in a more homogeneous and stable environment, where the lead architect can keep enough details of the systems architectures in his head. In our case, the number and diversity of systems, platforms, and technologies are such that it is difficult to have only one person looking at them all. Regarding architecture as a service, we follow Eckstein's advice, in a different way. One could see the group's goals as the group guiding the development teams, but the group's members come from each team exactly for the same reason Eckstein suggests her approach: to keep the group working in issues those teams see as valuable.

Roland Faber presents a case study describing how he has organized an architecture team to act as a service provider [Faber 2010], similar to Eckstein's approach. In his organization, architects are responsible for nonfunctional requirements, while developers are responsible for functionality. The architecture process is divided in two phases, preparation and support. During preparation, an architect specifies system qualities, creates an overview of the architecture, creates prototypes and a system skeleton to be used by developers to test and integrate their work early. In the support phase, an architect is collocated with the development team, and helps the developers implement the application, coding with them. Faber notes this is important because it builds trust between developers and architects and provides valuable feedback for architects about their architectural decisions and frameworks. Faber's approach particularly aims at the architecture of each separated system, but with overall issues such as reuse in sight. In our approach, the main concern of the architecture group is the whole system architecture, and each team is responsible for its system's architecture. Another difference is that instead of working with a sepa-

rated architecture team that collocates its members in teams, we followed the other way around, and formed the group with development team members.

7. Conclusion

The architecture group has been working for almost a year by the time of this writing. It already has presented considerable results, maybe the most important being an unplanned one: the increasing of architecture awareness by all company developers. It is an important achievement, since architecture was seen as unworthy two years ago. Besides, so far many mechanisms have been put in place to improve the quality of both customer products and back-end systems, such as the system integration standard, and an extensive use of testing and continuous integration.

Up to now, we have had spotted difficulties of operating an architecture group that others following this approach can face:

- Sometimes, members have a hard time dealing with conflicts between activities in their teams and in the architecture group, specially regarding agenda. For example, a team planning meeting may be scheduled to occur at the same time as a working group meeting. There is no general rule to deal with those situations, but usually one takes the team activities since most architecture group work is aimed at the long-term, and thus can wait.
- There are very few controversial issues where consensus is unreachable. In these cases, enforcing a decision is better than just skipping the issue.
- In a heterogeneous environment, there are situations when the technical lead must conduct discussions, but he does not have sufficient knowledge about the topic or the details involved to do so. When this is the case, the lead has to admit one's lack of confidence, ask for help and study as much as possible about the problem in hand.

In this article, we presented the practices that have been working for the architecture group. They may be useful for other organizations interested in adopting an architecture group following the approach proposed in this article. Table 1 provides a summary of the practices:

Table 1. Summary of the architecture group's practices.

Practice	Responsible	Objective	Details
Fill the architecture group with members of each team	Managers	Connect the group to the day-to-day reality of the organization	Section 3
Create standards	Architecture group	Address technical issues affecting multiple teams while preserving each team's autonomy	Section 4.1
Create goals	Architecture group	Make multiple teams consistently apply specific practices and standards and / or help balancing architectural related development and new features in teams' backlogs	Section 4.2
Define shared infrastructure	Architecture group	Avoid duplicate efforts when multiple teams share the same infrastructural needs	Section 4.3
Decide on technical issues collectively	Architecture group	Find the better solutions by summing the knowledge and points of view of members coming from different backgrounds and specialties	Section 5.1
Create and prioritize a backlog	Anyone add items, the architecture group leader prioritizes	Organize the architecture group's work and make it visible	Section 5.2
Run planning and review meetings	Architecture group leader organizes, architecture group members and any other interest party attends	Organize the architecture group's work and keep track of work being done at the moment	Section 5.3
Form working groups	Architecture group	Avoid long discussions when current information on technical or business is not sufficient	Section 5.4

When the number of people involved in a system goes much beyond the suggested agile team size, it becomes very difficult to keep the whole system architectural integrity. Our approach recognizes the need to balance architectural integrity with team self-organization and autonomy. Although having a central coordinating group in an agile environment may seem contradictory at first, it makes sense if the group not only respects, but also uses, the agile values, principles and practices guiding the whole organization.

8. Acknowledgements

We thank Locaweb for supporting this work and to Andrew de Andrade for revising preliminary versions of this article. Fabio Kon and Marco Gerosa receive individual grants from CNPq.

References

Ambler, S. W. (2002). *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. John Wiley & Sons.

Bass, L., Clements, P., and Kazman, R. (2003). *Software Architecture in Practice*. Addison-Wesley Longman.

Beck, K. (2005). *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D. (2001). Agile manifesto. `http://agilemanifesto.org/`.

Bradner, S. (1998). IETF working group guidelines and procedures, IETF RFC 2418. `http://www.rfc-editor.org/rfc/rfc2418.txt`.

Brooks, F. (1975). *The Mythical Man-Month*. Addison-Wesley.

Cohn, M. (2005). *Agile Estimating and Planning*. Prentice-Hall.

Coplien, J. O. and Harrison, N. B. (2004). *Organizational Patterns of Agile Software Development*. Prentice-Hall.

Eckstein, J. (2004). *Agile Software Development in the Large: Diving Into the Deep*. Dorset House Publishing.

Faber, R. (2010). Architects as service providers. *IEEE Software*, 27(2):33–40.

Foote, B. and Yoder, J. (1999). Big ball of mud. In *Pattern Languages of Program Design 4*. Addison-Wesley Longman.

Ford, N. (2009). Evolutionary architecture and emergent design: Investigating architecture and design. `http://www.ibm.com/developerworks/java/library/j-eaed1/index.html`.

Fowler, M. (2001). Is design dead? In *Extreme Programming Examined*. Addison-Wesley Longman.

Fowler, M. (2003). Who needs an architect? *IEEE Software*, 20(5):11–13.

Fowler, M. (2004). Technical debt. http://martinfowler.com/bliki/TechnicalDebt.html.

Kruchten, P. (2008). What do software architects really do? *The Journal of Systems & Software*, 81(12):2413–2416.

Larman, C. and Vode, B. (2010). *Practices for Scaling Lean & Agile Development*.

Leffingwell, D. (2007). *Scaling Software Agility: Best Practices for Large Enterprises*. Addison-Wesley Professional.

Lindvall, M., Basili, V. R., Boehm, B. W., Costa, P., Dangle, K., Shull, F., Tesoriero, R., Williams, L. A., and Zelkowitz, M. V. (2002). Empirical findings in agile methods. *Lecture Notes in Computer Science*, pages 197–207.