NEW APPROACHES IN SOFTWARE DEVELOPMENT

Cyril KLIMEŠ, Jaroslav PROCHÁZKA
Department of Informatics and Computers, Faculty of Natural Science,
University of Ostrava, 30. dubna 22, 701 03 Ostrava, Czech Republic,
E-mail: cyril.klimes@osu.cz, jaroslav.prochazka@osu.cz

SUMMARY

In the last few years we can meet modern approaches and concepts like agile methodologies, extreme programming and model-driven development in software development. These new approaches use methods and technologies that allow us to deliver valuable software products matching all requirements in time as well as decrease the cost of system operation and maintenance. Paper deals with some of these approaches. Technical managers, developers and also management should have at least brief knowledge of them because they can bring time and budget savings mainly in maintenance phase.

Keywords: *software development, agile approaches, model driven approaches, rule based systems*

1. INDRODUCTION

As hardware becomes more and more powerful in time, it becomes also smaller and cheaper. Consequently, software drivers and applications using this hardware need to be changed too. As software changes its extent, complexity and therefore becomes more complicated. Developers deal with extending users requirements. Development process becomes more complex and user or market requirements change often or are vague and misty. Since requirements on software ability and quality change and market and legislative environment where software is used changes too, it is necessary to change also software development process and technologies.

Programming languages have their evolution as well as the whole information technology field. Present most spread languages (e.g. Java, C++, C#) radically differ from their historical ancestors. The main difference is in the rising level of abstraction. In early programming years, the 1st generation languages (1GL) were used. 1GL languages worked on the lowest level of abstraction, i.e. with machine code. The evolution continued across symbolical languages (2nd generation) up to specialized domain languages (database SQL, publication PostScript). All these were mainly structured languages.

Many rules, principles and recommendations for structural programming were methodically devised. Since these were only recommendations, they had no influence on error occurrences in programs. However, most of these recommendations include usage of objects. Consequently, object oriented programming started to spread widely.

2. METHODS AND APPROACHES EVOLUTION

Also methodologies have their evolution as programming languages have. Methodologies reflect current situation; that means they adapt software development to contemporary requirements. We can say that methodologies are guides that say what to do during software development process, when it should be done sand how it should be done. They also help us not to forget anything and say what to produce and also what not to produce ("we don't need it now") in a particular phase of development.

In the beginning of the 50th, when the first computers were programmed, we could not speak of methodologies or engineering approaches to software development yet. because nothing such these did not exist. Software engineering as a new discipline came in the 70th arose new discipline called software engineering. Due to falling prices and hence better availability, computers became much more common than before. Consequently, the first problems with delayed or never completed projects, along with software maintenance problems appeared. These problems resulted in software crisis.

To reduce costs and development time, structured methodologies were defined as a first approach to work around the crisis. These divided development activities into phases, which allowed developers to concentrate only on steps and issues important for one particular phase (e.g. YSM, SSADM).

Due to popularity of object-oriented programming (OOP), the second attempt to solve still ongoing crisis was an inception of object-oriented methodologies. OOP success brought object principles also into software development methods. Object oriented methodologies don't include only objects and object oriented principles, but also the best practices from structured methods. Among others, e.g. Unified Process, Rational Unified Process (now IBM) or original Czech methodology OOMT (VSE Praha) belong into well known and widely used object-oriented methodologies.

Since one can often read about models and modelling in this article, it is necessary to mention unified modelling language – UML, which is nowadays widely used in many software projects. We do not judge if it is right or not, but only outline its role: UML serves for modelling, visualization, specification and documentation of object-oriented software systems components.

3. AGILE METHODOLOGIES

Now we finally describe new concepts and methodologies mentioned above. Selected group of these new methods is generally called 'agile'. Rigorous methodologies have developed from requirements valid in times in which they were defined, which often mean they are older than 15 or even 20 years. But times change and "the only today's guarantee is a change". Changes can occur in market competition, in market itself, or in legislative. If we develop enterprise system using rigorous methodology with an extensive analysis and design, the development process can take one or two years. It can happen that an enterprise changes its intention in the meantime.

Nowadays, a customer still wants quality software, but he rejects to wait for it for a long time. Moreover, a customer often doesn't know what to expect from the future system and he cannot imagine its functions. He only wants to do his business and software should automate and support his enterprise processes. He often realizes what his true requirements are when he uses an application that doesn't really meet them. The objectives of agile methodologies are to develop software more quickly and effectively (which are today's main requirements), and to develop a system that customer really needs and which is useful for him. Small development teams and a tight cooperation with customer are typical for agile methodologies.

Before year 2000, several respected IT professionals (e.g. Kent Beck, Martin Fowler or Alistar Cockburn) started to work on new approaches to software development. Although they didn't work together, they found that their work had many common features. When they met in 2001 in conference in Utah, they formulated so called agile manifesto [1]. This manifesto is not a modern matter of several enthusiasts. All the involved worked on many projects (from small to large ones, successful and also unsuccessful ones) and all of them became respected professionals. Moreover, all methodologies came through its evolution (though sometimes short) and were tested on suitable projects. These approaches are based on one rule, which says that the only way how to verify the system is to implement it and hand it over to the modified according to customer's remarks and further requirements. Agile manifesto authors prefer (from [1]):

- Individuals and interactions over processes and tools,
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan.

Authors emphasise the following points:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Business people and developers must work together daily throughout the project.
- Simplicity – the art of maximizing the amount of work not done – is essential.

Such development process is often test-driven (e.g. XP). It means that tests are written prior to writing code. A test-suite exists and is run every time when source code is changed. Application is always thoroughly tested and it means fully functional (without system errors). We can describe these methodologies as the ones that prefer software development over creation of sometimes extensive and unnecessary documentation, use short development iterations, emphasize cooperation with customer and his active participation, produce simple solution, and support continuous learning thanks to tight feedback. Agile methodologies are called e.g. these:

- Extreme Programming (XP, Kent Beck, 2000),
- Scrum (Ken Swaber, 1995),
- Crystal methodologies family (Alistar Cockburn, 2000),
- Agile Modelling (AM, Scott Ambler, 2002),
- Feature-Driven Development (Jeff De Luca, Peter Coad),
- Adaptive Software Development (Jim Highsmith, 2000).

We stress the main common principles of agile approaches. These are tight, every-day cooperation between users and development team, simplicity, early and continuous delivery of valuable software, test-driven development and changing requirements during development (user's advantage). As advantages, we can state these: welcome changes (precondition); fulfilling user requirements; quicker development; users have just the system they need (no special functions without value, etc.). But there are disadvantages as well: the main one is primarily suitability for green-field engineering (obvious from its role), it is not suitable for maintenance and evolution. The next disadvantage is also important: users have to communicate and cooperate with development team every day; it means they are responsible for success of the project too. As the last one, we mention developer's skills and behaviour.

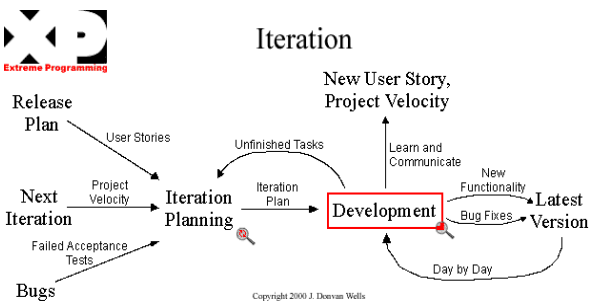

Fig. 1 XP iteration development (source [4])

Developers need to write readable code according to standards, with valuable comments, and need to follow the rule of simplicity in design and code. More about agile methodologies can be found e.g. in [2] and [3].

3.1. MDD (Model-Driven Development)

Agile methodologies are the first often mentioned and discussed concept. The second one is model-driven development. Typical representative is OMG's initiative called MDA – Model-Driven Architecture. While the core of agile methodologies is a quick development of functional software, (documentation is mainly well structured and commented source code), MDA emphasises consistent documentation in the model and also code form. Developers often only create models, without seeing resulting application.

Development process using MDA consists of several phases. MDA is not based on classical programming, but more on model transformations. The first model is a domain model without any specific implementation rules. This model is transformed into platform specific model and then into application source code. Thank to this model-code binding is whole project documentation in consistent state. Changes in models are projected into source code and vice versa. Maintenance of such system is easier and consequently less expensive. Another advantage is the form of models. They are not "just pictures", but executables that are parts of the system.

One of disadvantages is that developer doesn't see any results (running application). He can work for weeks or even months without being sure that the result will be satisfactory.

MDA transformations use best practices and well-tested standards. Standardised modelling language UML is used for modelling. For transformations, design patterns are used. They exist for every field of SW development and abstractly describe best practices, well-tried procedures and developer's experiences.

Developing using MDA has several phases; the most interesting one is a transformation from platform independent model to platform specific model:

- Platform independent (analytical) model PIM is extended by mappings, which define the use of general transformation rules.
- For platform independent model (or for its part), an implementation platform is chosen
- Based on mappings (and also according to chosen platform), transformations to be performed are given.

OMG says that MDA brings economical as well as non-economical advantages. A lot of case studies were created for this advantage and merit verification. One of them should confirm (or reject) this claim by developing server application. Two experienced teams developed the same application based on J2EE architecture. One team had MDA-based tools at disposal, while the second one worked using classical code-driven approach with IDE (Integrated Development Environment) tools. According to case study results, MDA team developed application about 35% faster than second team (MDA team worked for 330 hours, the second on for 507.5 hours). However, these numbers cannot act as an absolute measure, as the values may radically differ for different projects.

The main principles of MDD are modelling and transformation. Models are mapped and transformed according to patterns and templates. Now we list some MDD advantages. This approach shortens development cycle; includes best practices through patterns; is platform independent (patterns for J2EE, .NET, WS, CORBA); there exist a lot of MDD tools on market; is OMG's standard (based on UML, MOF, XML); includes automated testing. MDD is also suitable for maintenance (not only for green-field engineering). Our opinion is that MDD developer should be skilled and patient modeller with sense for details. He can model for weeks without any result in the functional application form, which can be frustrating. This issue we present as a disadvantage. Another one is a possibility of vendor lock-in; there have to exist patterns for our platform and architecture. But it seems that advantages bring more than disadvantages can take.

3.2. Agile MDA

There exists also agile form of MDA in development approaches that combines ostensibly opposite ideas – agility of agile approaches and modelling. Agile MDA is based on presumption that MDA's source code and executable models are the same. This implies use of agile principles (testing, short iterations, etc.) also on executable models, not only on source code. This approach (complete models, lastly code) can resemble rigorous methods. But there is a difference between models. MDA models are executable and thus different from rigorous method models that are "simple pictures".

3.3. Agile Modelling - AM

Till now, we explained some of the new concepts and ideas. Another one we discuss is Agile Modelling – AM. Author of AM is very well known IT specialist Scott W. Ambler. We have already listed AM in agile methodologies together with XP or Crystal. But AM is not a new complex methodology, it is more an extension of existing ones (see Fig. 2). As author says, AM is practically based method (not an academic theory) for an effective software systems modelling and documentation. AM is a suite of guides and procedures presented by principles and values for everyday professional use.

A frequent problem of developers using strict, rigorous methodologies is creating a lot of documentation, often unnecessary or never used. A

typical example of such problem is creating of all UML models. Of course, documentation is needed, but with reasonable amount. AM approaches try to outline how to agile model and how to create only needed documentation. Apart from other methodologies AM do not say, what models to use and what to create. It fully depends on developer's decision. AM does not restrict only one set of models (e.g. only UML models). As shown on figure 2 depicts, it is possible to use AM together with RUP (Rational Unified Process) or EUP (Enterprise Unified Process) methodologies as well as with agile ones (XP, FDD, etc.). Besides, AM uses some XP principles and originally was called Extreme Modelling – XM.

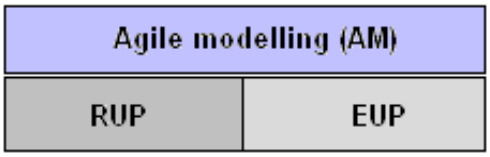

Fig. 2 AM and its scope

4. OTHER DEVELOPMENT APPROACHES

No matter that they can radically differ, all the above mentioned methods and approaches have one common point. An application should be always hand-coded (except MDD) by team of developers. A little bit different approach is presented specifically by MDD and its variants that we have already mentioned. Using MDD, we do not need to hand-code because the final version of application is generated from executable visual model. This brings us to another development approach that is also already used nowadays. This is automatic code generation.

4.1. Code generation

Code generation is not an innovation; it is a technique used in everyday work, although we often don't know about it. We can name generators in the form of compilers, documentation generators (e.g. Javadoc) or HTML generators. Use of generators brings several advantages, the main are:

- Increasing productivity of labour – thanks to automatic generation.
- Increasing code consistency – code is generated using still the same template or pattern, so code has the same quality in every part (the quality of code depends on the quality of template).
- Reducing the count of errors – there are no typing errors or forgotten mistakes after copy and paste in the generated code, etc.

Generators are most often used for user interface generation (JSP pages, Swing GUI) and for database scheme generation. Concerning application logics, this approach is not so spread, because it is not so easy to describe the logics using meta-data and rules. The following figure depicts needed generator artefacts.

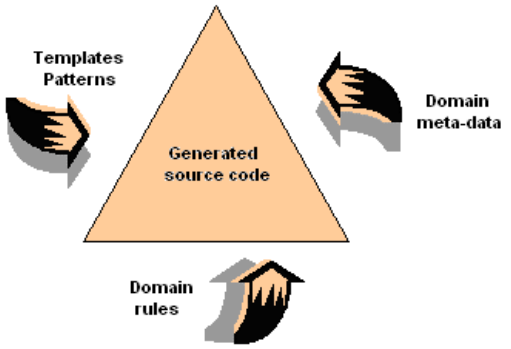

Fig. 3 Code generator components.

There are several ways how to generate source code. Some of them are listed below [5]:

- Templates + filtering – the easiest way, we filter off part of model text specification (e.g. XML/XMI file) and then apply template.
- Templates + meta-model – extends the previous pattern, if clear architecture to implementation platform mapping exist, meta-model is created and templates are applied.
- Frame processing – source code is generated as a result of frame evaluation; frames are typically functions.
- Direct transcription – we examine text specification of a model (e.g. XMI file). If a given expression is found, it is written into output file (source code): `if found modelElement then write("CodeElement").`

As said above, the easiest way is application of template and filtering pattern. Every visual model can be stored as text file using XML (or XMI). The following example shows text form of a part of a simple visual model.

```
<UML:Class xmi.id='802f4_1105_1'>
  <UML:ModelElement.name >
    Třída
  </UML:ModelElement.name>
  <UML:ModelElement.visibility
xmi.value='public' />
  <UML:Classifier.feature >
    <UML:Attribute xmi.id='802f4_1105_2'>
      <UML:ModelElement.name >
        atribut1
      </UML:ModelElement.name>
     <UML:ModelElement.visibility
          xmi.value='private'/>
    </UML:Attribute>
  </UML:Classifier.feature>
</UML:Class>
```

This XMI text is simplified using filtering and the following XSLT template is consecutively applied:

```
<xsl:template match="class">
  public class <xsl:value-of select="name"/>
{
```

```
  <xsl:apply-templates select="attribute">
}
</xsl:template>
<xsl:template match="attribute">
  private <xsl:value-of select="@type"/>
  <xsl:value-of select="@name"/>;
</xsl:template>
```

The result can then look like this:

```
Public class Třída {
  private int atribut1;
}
```

We can emphasize one rule: everything repeatable can be automated. When developing code generator, we should obey the following rules. The best approach is to develop a part of application source code (not a generator, it is obviously hand-coded) by hand. Such procedure helps developer to understand used technology or framework. Hand-coded source code can be then used for generated source code control. Testing suites for generated source code should also be written. They check correctness of functionality as well as weird constructions and behaviour. Automated robust source code controls and static code control are also good thing as well as marking generated code in source code files.

4.2. Rule based systems

Quite different approach to software development, which is really worth to be mentioned, is using so called business rule engines – BRE. The convergence of market to such systems is predicted by Gartner group in one of its studies. BRE systems do not describe processes (we do not bind process by programming code), because these can change. Instead of describing processes, we describe general rules valid in the whole domain. When developing application, we define basic concepts, their relations, and rules for their manipulation. After analysis, thes system is not hand-coded, but the source code is automatically generated according to rules or BRE directly interprets these rules. Rules in such systems are in formalized natural language form that is understandable for domain workers involved (common employees as well as managers). Thanks to this, it is not necessary for these people to talk developer's language while developing application. Users work with well known concepts like bill, project, milestone, contract, manager, etc. and common rules like: "All unpaid bills should be listed in debt-book".

When enterprise processes change, we do not need to alter the system, because the domain rules are still valid. When rules change too, e.g. because of legislation change, we only update given rule or add a new one. Rule systems have these advantages: they communicate with user using his language; their development is not so time consuming as classical development (and, hence, not so expensive); implementation of changes is also cheaper. Rule based systems are sometimes called "personal assistants" rather than "software". Such system is really more an assistant than a common program. What should rule based systems include can be found in [7].

5. CONCLUSION

New approaches have a common feature, which is quick application development together with tight cooperation with customer. However, each of these approaches use different way to reach this. Because of their philosophy, some of agile methodologies can be used only for green-field engineering. However, most of mentioned approaches can be profitably used for system change implementation.

One can expect such methodology's evolution because of customer's requirements (SW quickly with quality). It is important to emphasize that agile methods are not a universal solution for existing problems. Their usefulness shows up in rather smaller teams and small and middle projects with unclear (vague) or often changing requirements. For complex and large-scale systems, it is often essential to use rigorous methods (milestones are planed and accepted by customer, etc.).

For some parts of an application, code generation is used more and more. Code generation can be used in rigorous methodologies, as well as in agile ones.

One of possible development approaches in the future is a rule-based system, which reduces time needed for development and also simplifies changes implementation.

How one can see, an evolution of methodologies and software development approaches is an interesting topic and we can be optimistic about its contribution to fulfilling customers needs and lowering projects and maintenance costs.

REFERENCES

[1] Agile manifesto web. Available on: www.agilemanifesto.org

[2] Kadlec, V.: Agilní programování, Computer Press, Brno 2004, ISBN 80-251-0342-0, (in Czech)

[3] Ambler, S. W.: Agile modeling. John Wiley & Sons, 2002.

[4] Extreme programming web. Available on: www.extremeprogramming.org

[5] Voelter, M.: A catalog of patterns for program generation. Document available on: www.voelter.de

[6] Herrington, J.: Code generation in action, Manning 2004, ISBN 1930110979.

[7] Business Rules Group web. Available on: www.businessrulesgroup.org

BIOGRAPHIES

Cyril Klimeš graduated (Ing.) in 1976 at the Faculty of electrical engineering at Brno university of technology. He got his CSc. degree in 1985 at VŠB-Technical university of Ostrava and in 1991 he defended his habilitation at the same university. Now he works at the department of Informatics and computers of the Faculty of Science at University of Ostrava where he is department manager. He worked also in OKD and he has managed computer company OASA Computers, s.r.o. He is author and co-author of tens of publications and co-author of 3 patents. His scientific research is focusing on computers architecture, operation systems, information system development and e-learning.

Jaroslav Procházka was born on the 15. 7. 1978. In 2003 he graduated (MSc.) at the department of Informatics and Computers of the Faculty of Science at University of Ostrava. Now he study doctoral program (Ph.D.) also at University of Ostrava with specialization on the field of process automation and software development. Since 2003 he is working as a tutor with the Department of Informatics and Computers and since 2005 he is working as analyst and Java developer in Crux IT. His scientific research is focusing on process automation and new development methods as well as software evolution (implementing changes and enhancements).