A Multi-Agent System Assisting Software Developers

Ivar Jacobson
Vice President Process Strategy
Rational Software Corporation
ivar@rational.com

Stefan Bylund
Senior Process Architect
Jaczone AB
stefan.bylund@jaczone.com

The "State of the Art"

It has never been so complex to develop software as it is today. Software developers are intensive knowledge workers. They must daily apply their hard-earned development skills to the fullest. They must continuously learn more because new trends and technologies emerge while yesterday's become obsolete. Not only must they understand new trends and technologies, they must adopt them in a productive manner, and they must do it quickly. This is tough, even for the hardest working experts.

More specifically, relevant knowledge manifests itself in terms of best practices, guidelines, methodologies, and the like. In particular, this kind of knowledge manifests itself in terms of:

- Development standards such as the Unified Modeling Language [UML], the Unified Process [USDP,RUP,Kruchten,Road],
- Application platform and middle-ware standards such as [.NET,J2EE],
- Various lower-level implementation standards such as Java, C#, HTML, and XML.

This knowledge must then be continuously – and just-in-time – accessed, distilled, and interpreted by developers as they apply it to the system under construction.

We believe that many problems bedeviling today's projects stem from exactly this: the lack of the capabilities needed to apply relevant knowledge. Moreover, adding to the difficulties, developers may lack both training and experience, often due to the shortage of such people. Worse, the demand for skilled people is expected to increase by over 100 percent in the next six years.

Projects are always in a hurry, leaving developers limited time for learning. Therefore, relevant knowledge that takes too much time and effort to access, distil and interpret simply stays on the shelves. It remains unexploited.

The consequences are devastating. Exclusion of relevant knowledge yields lesser productivity, higher costs, and sub-standard quality. It may even yield project failure when development tasks become too challenging.

If nothing is done, the situation will just get worse: The technology is getting more complex; the business requirements are becoming harder than ever to fulfill; lead times become ever shorter; and quality requirements become ever higher.

There is no magic way out. For most software developers, all this just implies chaos. What can we do about it?

The Vision 20 Years Ago

In 1980 Ivar Jacobson wrote a visionary letter to his CEO, proposing that Ericsson leverage its component-based software development approach in three steps:

1. Develop the Ericsson software modeling language into a standard language.
2. Develop the Ericsson software development work process into a standard process.
3. On top of this, develop a support system that employs knowledge-based reasoning to help developers when working according to the standard development process and modeling language.

As for step 1, Ericsson had already (partly) succeeded by standardizing a language, SDL [ITU-T], as a world standard for telecom systems. SDL inspired the development of the Unified Modeling Language [UML], adopted by the Object Management Group as a standard in 1997. Many of the ideas in UML (e.g., sequence diagrams, collaboration

diagrams, activity diagrams, state charts) were already in use at Ericsson in the late 1960's!
As for step 2, the work process used at Ericsson was actually a forerunner of the Objectory process that has now evolved into the Rational Unified Process [USDP, RUP]. This process is, in turn, becoming a de facto standard. It has also inspired a standard for process modeling adopted by the OMG called the Software Process Engineering Metamodel [SPEM].
As for step 3, there existed already in the late 70s and early 80s knowledge-based systems, expert systems, artificial intelligence (AI) systems, etc. that were judged to be very promising new technologies. There was a lot of belief in what these new technologies could achieve, and many new companies were founded. Unfortunately, many of them failed, and in the late 80s most of them were gone. The implementation technologies available at that time just did not fulfil the high, and sometimes over-hyped, expectations. Today, two decades later, we believe that the time has come for knowledge-based systems to take a stab at the challenges of software development and the chaos these challenges bring. The standards and technologies required to meet these challenges are just now in place.

Introducing: The Software Developer Assistant (DA)

For this purpose, we introduce the notion of a developer assistant (DA). This assistant provides some of the expert knowledge that a developer needs, such as development guidelines, language specifications, and process descriptions. The assistant is able to reason with that knowledge to help an individual developer achieve results comparable to those of expert developers.

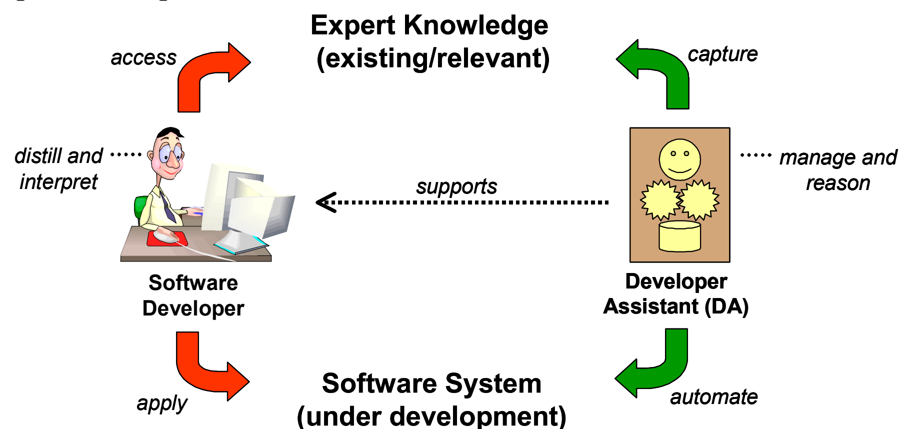

Figure 1. A developer assistant (DA) for software development.

Thus, instead of requiring that the software developer have all this knowledge, the DA would support the developer in three key ways: By capturing a significant part of the knowledge; by reasoning based on this knowledge; and by automating the application of this knowledge to the software system under development.
The fundamental requirements for making a DA truly support a software developer are:

- The DA must by its nature be considered as a part of an integrated development environment. The DA must harmonize, interact, and co-exist with other tools supporting the developer in for example modeling and coding tasks.

- To be able to help, the DA must understand what goals the developer has at a particular point in time. For example, such a goal might include what software artifacts the developer is about to produce, and the expected level of quality of the artifacts.
- The DA must be able to reason in the developer's particular context. This reasoning needs to be done just when needed, often in a dialogue with the developer. This capability calls for mechanisms within the DA to structure knowledge. An important coordinate condition in this knowledge application is that it not interfere with other tasks of the developer. It must not annoy him or her.
- The DA must be based on development standards such as the Unified Modeling Language and the Unified Process [UML, USDP, RUP]. Capturing DA knowledge is a significant effort as well as an investment. It would be hard to justify otherwise.
- It must be possible to specialize and configure the DA to suit the needs of a specific project or organization. These needs may, for example, depend on the type of software system being developed (banking, insurance, telecom, air-traffic control, etc.). They may also depend on the size and complexity of the project or organization. This requirement means that the knowledge provided by the DA may be used selectively. It means further that new knowledge may be added, specific to the needs of the project or organization.
- The knowledge provided by a DA must be easily updated, because change is natural when it comes to knowledge. It is not only specific projects and organizations that need to add and refine knowledge. It may also be the case that the underlying knowledge standards change. (See bullet above regarding standards; and see for example [UML] in particular). Simply put, knowledge must be structured for "plug & play."

We have reached a milestone in technological evolution where these kinds of requirements are fully realizable. We can in fact build a DA for software development that would be useful from day one. We can also grow through changes to operate in the long run. In this paper we elaborate on how these goals can be achieved in practice.

Integrating the Development Environment

The DA is not an isolated island. Alone it cannot help the developer accomplish all the tasks it takes to build software in general. On the contrary, the DA needs to be considered as a complementing part of a more complete environment.

This environment also contains other supporting technologies such as modeling and coding tools, configuration management tools, and the like [VisualStudio, RationalSuite].

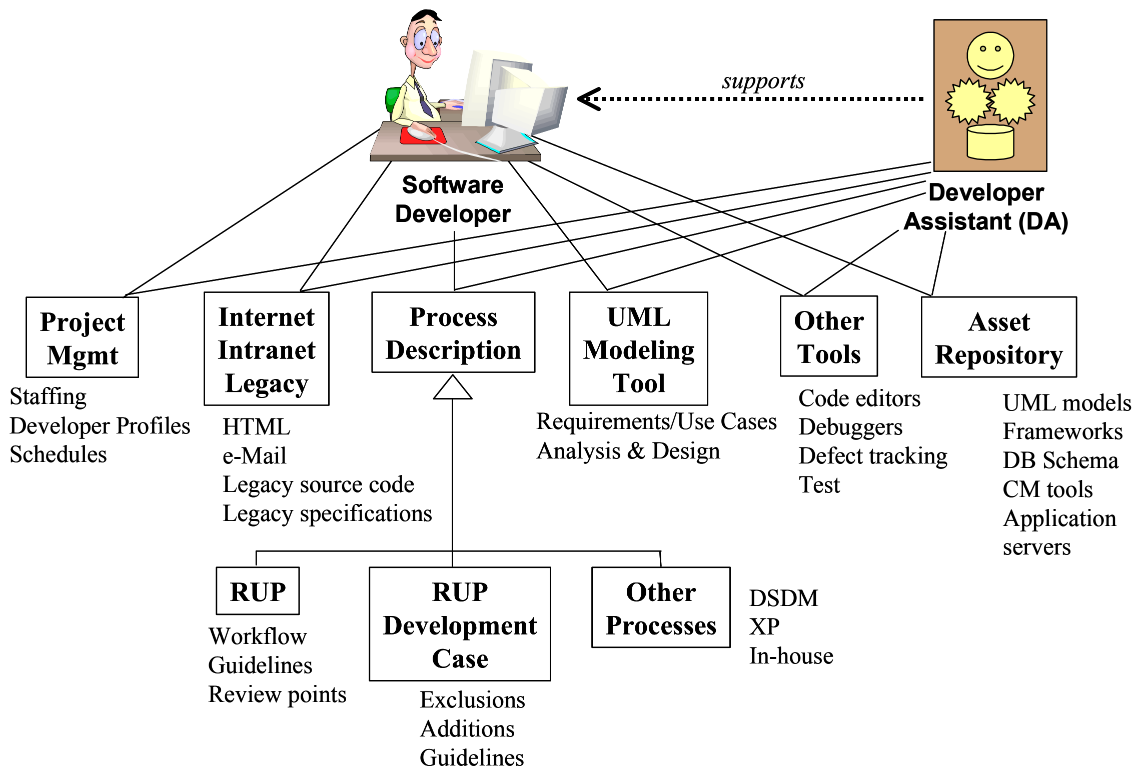

Figure 2. The DA in the development environment.

Existing tools help developers work in specific areas such as requirements capture, design, implementation, architecture, and configuration management. In these tools we often find sophisticated functionality to describe, maintain, and visualize software artifacts. We also find the ability to integrate with other tools. Existing tools will continue to perform functions such as these. Moreover, they should continue to improve for years to come.
Now, one could assume that current tool strategies would focus on supporting developers knowledge-wise as they work. Yet, there is still plenty of room for improvement. Despite some initiatives to "open up" knowledge (see, for example, "dynamic help" in [VisualStudio]), it is still often locked up in passive repositories, such as static help systems. Existing tools often only serve up specific "knowledge pages" such as tool manuals and development guidelines, based on explicit requests. The reading tools are in many respects just a read-only dumb terminal—developers can only browse information given that they know what they need. It is often difficult to actually apply the knowledge served up in this way to the problem at hand. Knowledge is simply not served to the developer in a useful way.
More is required to actually support the software developer in solving hard development problems. That is, more is required to be able to access knowledge in a productive manner, to set the knowledge in context, and to apply it to the problem at hand. This is the role of the DA.
To do all this, the DA needs to interact with the other development tools. For example, to evaluate knowledge, the DA needs to be able to detect the state of artifacts managed within the other tools. Also, as the developer applies knowledge, any automated support provided by the DA needs to refine these artifacts accordingly. Moreover, the DA may need to refer to specific knowledge (e.g., help topics) in existing online documentation within the other tools of the development environment, instead of having to redefine this

information within itself. This integration is illustrated further on (section *Making Knowledge Useful and Accessible*).

Understanding the Developer's Goals

The goals of a project are often defined in terms of milestones to be reached. Such a milestone concludes a phase or an iteration, and is more or less standardized in, for example, the Unified Process [USDP, RUP].

Based on these overall goals, we need to set up the goals for each individual developer participating in the project. This is – from a technical standpoint – naturally done in terms of roles, activities, and artifacts (R/A/A for short): developers adopt *roles* as they perform *activities* that result in *artifacts*.

- **Roles.** A role is adopted by a developer, or by a group of developers working together. It defines i) a set of related activities that are best performed by one individual (or a group of individuals), and ii) a set of related software artifacts that lie under the responsibility of the individual (or the group) adopting the role.
- **Activities.** An activity is performed by the developer, and defines a focused piece of development work; an activity is performed by using some existing artifacts as input, and produces as results new or refined artifacts as output.
- **Artifacts.** An artifact is created or maintained by a developer, and can be a software work product like the source code for a class, a use case (including its description), or any other model element [UML]. Such an artifact is a part of the software system under construction.

Thus a DA for software development needs to understand the R/A/A that are relevant to the software developers. It needs this understanding to truly grasp the developers' goals.

As an example of R/A/A, consider a developer adopting the *System Analyst* role and performing its related activities *Find Actors and Use Cases* and *Structure the Use-Case Model*; these activities result in a refined *Use-Case Model* artifact that captures requirements made on the software system [USDP, RUP].

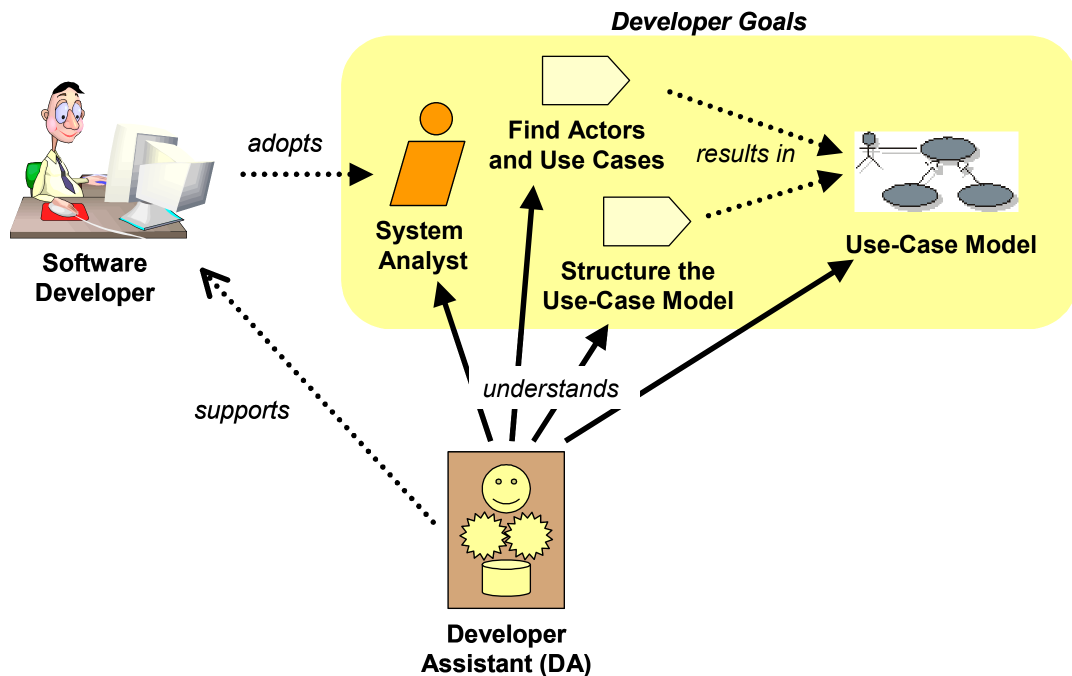

Figure 3. A DA must understand the roles, activities, and artifacts relevant to a developer, to be able to support the developer.

This structure in terms of R/A/A is applicable in any software development discipline. Additional examples are class design activities performed by designers that result in classes; and test activities performed by testers that result in test specifications and/or identified defects.
From this, we draw conclusions:

- **What knowledge to capture.** The most essential knowledge to be captured by the DA and dynamically provided to the developer is: how to adopt roles and perform activities in a productive and focused manner. More specifically, it is knowledge about how to develop ”good” artifacts (e.g., ”good” model abstractions and ”good” run-time components) that together make up the software system.
- **When to apply knowledge.** The DA needs to understand *when* to provide knowledge to the software developer. The activities are excellent instruments for making this determination because, given that the DA understands when the developer performs a particular activity, the DA can support this particular context. The DA does this by preparing parts of the knowledge significant to the activity or micro-activity in progress, and by immediately serving it to the developer "just in time."
- **What to apply knowledge to.** The DA needs to understand *what* to apply knowledge to and the answer is obvious: knowledge should be applied to software artifacts as they are developed in activities. The application of knowledge then refines existing artifacts in various ways.

Thus, a DA must understand the developer’s goals, that is, the roles, activities, and artifacts (R/A/A) relevant to the developer. As a result, we understand what knowledge to capture, when to apply this knowledge, and what the knowledge should be applied to. In the following section we discuss how one may manage knowledge technically, ”within” the DA.

Making Knowledge Useful and Accessible

As a vehicle for capturing, structuring, and applying knowledge, we employ a rule-based language. By using this language, the DA can serve knowledge in the right context (roles and activities), just-when-needed, and also apply it automatically to the software system (artifacts) under development.

When knowledge is to be captured, we consider goals in terms of R/A/A, and from these we derive structured knowledge in terms of rules, as shown in Figure 4.

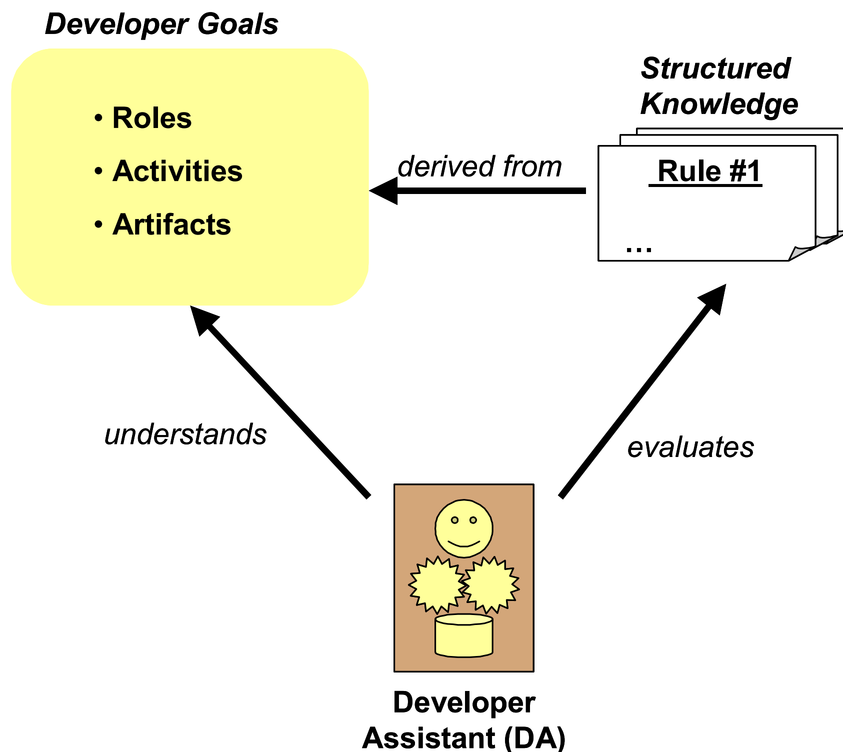

Figure 4. The DA evaluates structured knowledge in terms of rules derived from developer goals (R/A/A). (The rules are actually a part of the DA.)

In this rule-based language we have primarily used ideas from a few well-known formal languages, such as first-order predicate logics, basic collection and set operations, and the Object Constraint Language [OCL, UML]. The rules in this language basically detect interesting states of software artifacts and then propose new and better states of those artifacts. The overall purpose, of course, is to enhance the quality of artifacts under development, given the developer's goals.

Detecting Existing State

The first step in capturing knowledge is to decide what state of the knowledge domain it is interesting to reason about. One example of an interesting state is faults such as errors and inconsistencies in artifacts. Another example is a state of incompleteness such as an artifact that is missing or not fully described.

Proposing New States via Rules

The second step of capturing knowledge is to decide what to do with a detected state of a software artifact, thereby proposing a new state of the artifact. Such a new state can include, for example, that we create other (new) artifacts related to the first artifact, that we provide refinement opportunities in the artifact, or that we fix errors or inconsistencies in the artifact.
To specify this action, we employ a refined variant of if-then rules, sometimes also called productions, situation-action rules, or condition-action rules [AI]. Each such rule is then used as a mechanism to decompose and structure knowledge into small, manageable, and focused chunks that can be automated.

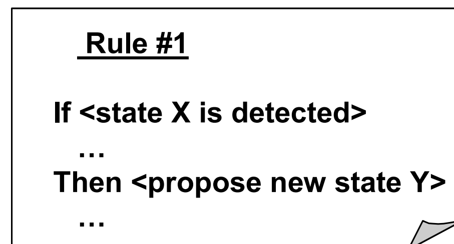

Figure 5. An if-then rule template.

The "then part" of rules can either completely automate the reaching of a new state, or they can help the developer reach the new state via a dialogue. Moreover, the then part of rules can sometimes propose several alternative new states, especially when there is no single recommendation on how to handle a problem.
To give more concrete examples of if-then rules, let's consider the *System Analyst* role and its related activities, *Find Actors and Use Cases* and *Structure the Use-Case Model.* This example results in a refined *Use-Case Model* artifact (as discussed above). From the *Find Actors and Use Cases* activity, we derive the following simple rule example:

Example Rule: Handle Missing Association

The following rule is based on the assumption that every concrete use case in the use-case model should be associated with an actor. Thus, if a concrete use case not associated with any actor was to be detected (see example state specification above), we can specify a rule that proposes new states of the use case.

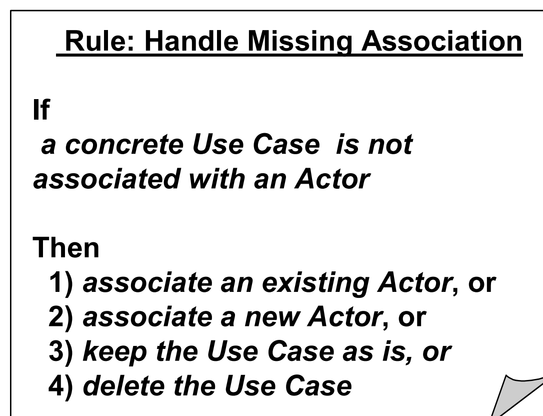

Figure 6. Example rule.

When this rule fires, there may be many reasons why an association is missing. However, the then part proposes a number of alternative recommendations for correcting the model.

Of course, for this rule to be really useful to a developer in a concrete situation, it needs to describe in more detail the detected (suspect) state of the model and the consequences of following the corrective recommendations. We still include the simple rule here to communicate the notion and essence of if-then rules in a real situation.

Some other rule types that we have found are:

- Rules to get started (initially) building software system models by detecting more or less empty models and then suggesting ways to start identifying and describing new elements within the model.
- Rules for model completion that identify missing model elements such as subsystems, classes, relationships, etc. and then propose corresponding model refinements.
- Rules for employing software patterns [GoF].
- Rules for evaluations and reviews of models that identify checkpoints such as formal inconsistencies, errors, or anti-patterns in the model, and then propose corresponding model corrections.

This approach to using rules can in general support all stages of development: first, the initial, more creative stages, then the more ongoing work, and finally the evaluations and reviews to assure the quality of the artifacts before delivery. This approach is important because almost all adoptions of roles and activities, such as in the Unified Process [USDP, RUP], go through these stages.
Now, imagine thousands or even tens of thousands of rules like these being packaged and served by the DA just-when-needed within the roles and activities that have been adopted by a software developer. Rules are evaluated by the DA to assist the developer in making the correct decisions at the right time by automated application of relevant expert knowledge. These rules make the developer move and take action if he or she finds it appropriate; the more and better rules are, the better a developer can progress.

Implementing the DA with Intelligent Software Agents

Background

Since the late 80s, we have seen a rocket-like development of software-based agents in a wide range of application areas [ISA, AI]. Many important academic institutes, such as MIT and Carnegie Mellon, have formed significant agent research groups. There is also standardization work related to agents, such as that within the Object Management Group. There are interest groups that base agent technologies on standard modeling languages such as the UML (see www.auml.org). Moreover, on the commercial side, we see agent technology from Microsoft Corporation (that is, Microsoft® Agent). There are Internet web sites, such as agentland.com, that list numerous agent-based companies and products.

Something that can serve as yet another concrete token of this general trend is an on-line bibliography such as [IAB] on software agents. As of today, we find 2480 (!) agent-related references, most of which were written in 1997-2001. The activity in this field is simply mind blazing.

Key Agent Characteristics

In general, we can discuss many characteristics of software-based agents. However, an industry-standard definition of “agent” has not yet emerged. Because of this lack, we try here to state what we think are the key characteristics of a software-based agent. The key characteristics are those that discriminate an agent from “ordinary” objects and components. These key characteristics, summarized in Figure 7, are the following:

Agent
knowledge autonomous behavior goal

Figure 7. Agent key characteristics.

- An agent is based on a significant set of *knowledge*. Given that knowledge, it is able to reason. Within the DA, agents primarily represent knowledge in terms of rules, as discussed above. Agents may in general also represent knowledge in other ways such as in terms of attributes, states, and the like.
- An agent has *autonomous behavior* in the sense that it is self-driven and acts on its own behalf without direct external intervention. Based on the result of its reasoning, it can decide when to act. When the agent draws conclusions, it may notify its environment (e.g. other agents or a developer). Given those conclusions, it may even take actions by itself. The environment, such as a developer, can then in turn decide what to do with the agent’s conclusions. Within the agent, this autonomous behavior is driven by rule machinery that pro-actively evaluates rules.
- An agent has a well-defined *goal* that is represented by the knowledge it possesses and by its behavior (that is, its ability to reason given that knowledge). An agent simply tries to reason, given its knowledge, to fulfill its goal.

This said, we may of course also discuss other characteristics (collaboration, adaptability, mobility, lifecycle, etc.) of agents if required, but we think these are more like consequences or aspects of the above. Given this limitation, we discuss what these key characteristics really mean when considering the DA as an agent system.

The DA is a Multi-Agent System

The basic idea is that the DA should be implemented as a collection of software-based agents, that is, as a multi-agent system. An agent is created to manage a role adopted by the developer, an activity being performed by the developer, or a software artifact that the developer is responsible for. Thus we have role agents, activity agents, and artifact agents as is shown in Figure 8.

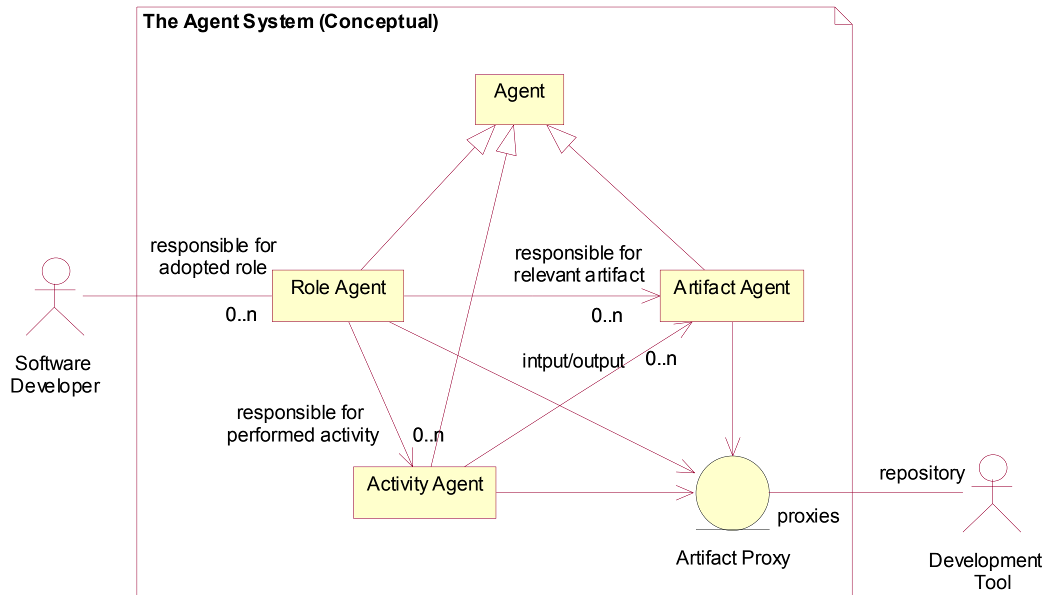

Figure 8. Agent types in the DA.

The rationale for this model is the following:

- The goal of a role agent is to help a developer adopt a role. It includes helping the developer be responsible for artifacts and for starting and performing activities when appropriate.
- The goal of an activity agent is to help a developer perform an activity. It includes providing directions and guidelines on i) what to do, ii) how to do it, and iii) when to do it, within the activity.
- The goal of an artifact agent is to help a developer maintain a correct state of the artifact. The artifact should be well formed according to its definition in terms of both syntactical as well as semantical aspects.
- When an activity is performed, a set of artifacts is input and output as results from the activity. An artifact agent would be created to handle each such artifact.
- Other development tools in the development environment serve as a repository for artifacts as seen from the DA's perspective (see section *Integrating the Development Environment* above).
- The Artifact Proxy entity in the figure represents "passive" artifact data acquired from the other development tools, as the original artifact physically lives in these external tools. The knowledge (rules) of the various agents may need to access this artifact data as rules are evaluated. Moreover, any change of this artifact proxy (as initiated by agents) is assumed to be reflected in the actual artifacts residing in the other development tools.

This agent system is then configurable in the sense that the developer can select what types of agents are to be used, and then adjust the knowledge (rules) of these agents if required.

The Agent System: A Concrete Scenario

The concrete scenario within the agent system (Figure 9) illustrates the adoption of a role.

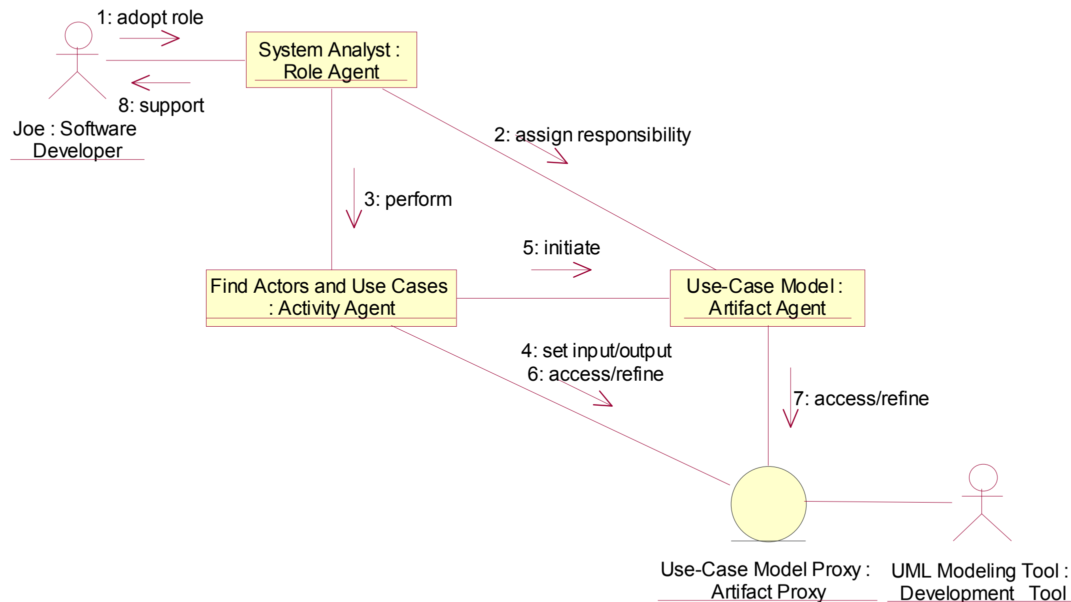

Figure 9. Concrete agent system scenario: role adoption.

This scenario can be described as follows:

> *Joe, a software developer, decides to adopt the System Analyst role (1: adopt role). A corresponding role agent is created.*
>
> *To start with, the role agent suggests that Joe should be responsible for the Use-Case Model artifact. Joe accepts this responsibility, and a corresponding artifact agent is created to assist Joe in shouldering this responsibility (2: assign responsibility).*
>
> *Then, the role agent suggests that Joe start performing the Find Actors and Use Cases activity, to develop the Use-Case Model. Joe also finds this to be a good idea, and a corresponding activity agent is created (3: perform). When this is done, the activity agent is set up to refer to the artifact proxy regarding input and output of the activity (4: set input/output), and the corresponding artifact agent is initiated to handle the output artifact (5: initiate); Joe of course confirms this.*
>
> *Now, the artifact and activity agents start to reason, given their specific knowledge in terms of rules. To evaluate their rules, they may access and refine the Use-Case Model artifact proxy (6, 7: access/refine). Then, additional support[1] from these agents is provided in a dialogue with Joe (8: Support).*
>
> *… [This is continued as follows (sketch): Based on the dialogue with Joe, more refinements (access/refine) are made on the Use-Case Model artifact. At some point in time, the role agent may propose that other activities be performed, such as to Structure the Use-Case Model; then, Joe maybe accepts this and corresponding activity agents*

[1] Refer to section *Making Knowledge Useful and Accessible* above for examples of support given via rules.

would be created. This way, new agents are created, giving support, and finally terminated based on the developer's goals.]

Of course, the developer may at any time during this scenario work with the artifacts directly in other development tools. However, the basic idea is that the DA is pro-active and continuously provides support (recommendations on how to solve relevant problems) to the developer, given the developer's goals. The developer may turn to the DA whenever he or she feels the need.

Supporting Micro-Activities with Agents

Each agent can support the developer via a fairly large number of rules. The agent presents the rules, and it is then often up to the developer to select a single rule and apply it in context. There are ways to make rules easier to use. One way, for example, is to categorize them according to steps within an activity agent. Another way is to prioritize them according to importance.
However, sometimes many rules appear simultaneously. For example, within an activity agent the many-rules approach typically fails to support a novice developer adequately. The novice may be more passive. That passivity may require the DA, in turn, to be more pro-active.
To handle this problem, an agent can group related rules into larger constellations of rules. The novice can then use the rules in sequence. Each such rule constellation can support more large-grained modeling tasks than individual rules can do per se. A constellation also gives a more pro-active support by guiding the developer through several rules in sequence. Thereby the developer is not required to decide what individual rules to use at each point in time.

There are many kinds of ”large-grained modeling tasks," or micro-activities if you wish, that we have been identifying as appropriate for this kind of support. For example, one is to identify all elements of a certain type (e.g., actors) in a model. Another is to merge or split complex model elements as in refactoring. A third is to review a certain model element by going through a set of checkpoints. All these micro-activities need to involve several more primitive rules that are appropriate to apply in sequence.

Components in the DA

To implement the DA as an agent system, a number of components are required to realize the different agents. See Figure 10.

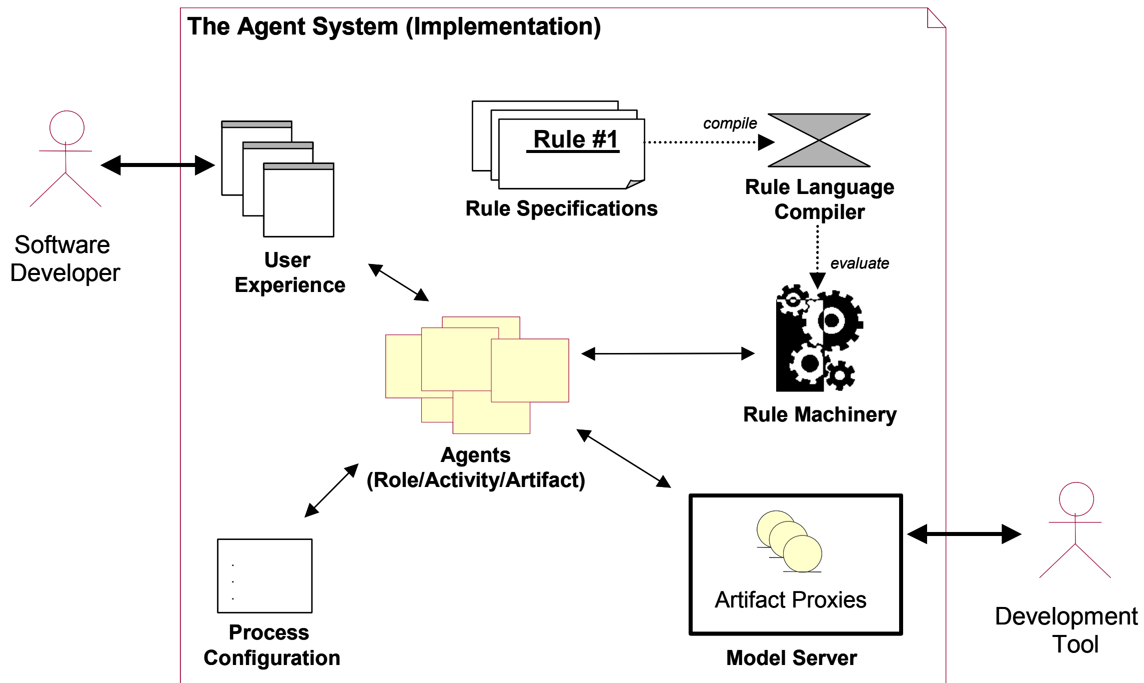

Figure 10. Implementing the agent system.

The components are the following:

- **Rule Language Compiler and Rule Machinery.** As rules are specified using a high-level rule language, they need to be compiled into a format that can be evaluated by rule machinery. The rule machinery is, in turn, incorporated in and used within each agent. Thus, an agent can evaluate the rules relevant to itself and can do so in its own context.
- **User Experience.** The developer experiences the agents through a user interface; all user input to and output from the agents is handled via this interface.
- **Model Server.** As noted above, agents and their rules need to access and refine the artifact proxies (when driven by a developer). This access is something that, in turn, needs to be reflected in the development tools hosting the artifacts. Given this, the purpose of the model server is i) to manage all artifact proxies relevant to the agents together as a complete model, ii) to keep proxies updated and consistent with the original artifacts hosted in the external development tools, and iii) to update the artifacts in these tools as requested by agents.
- **Process Configuration.** The development process supported by the agent system often needs to be configured to suit a specific developer and project. The degree of this adaptation depends, for example, on the software quality ambitions of the project. It also depends on the complexity and type of software system being developed. The process configuration is similar to a "development case" in the Unified Process [USDP, RUP]. However, its purpose here is to configure the DA to support a specific development process and to personalize it to a particular developer.

Achieving a Transparent Development Process

We are convinced by many examples that adoption of a development process such as the Unified Process yields nothing but success. But, the adoption must be sound and incremental. It must include adequate mentoring and training of the development team. It must be based on the use of appropriate development tools. There are, of course,

challenges with this approach. One is the place of "soft factors." The Unified Process puts its primary focus on methodology and how to master technology. That still leaves a wide range of soft factors to be taken into consideration. Not the least of these are: how to treat people as individuals, how to make them prosper, how to consider their competencies, how to deal with their attitudes, how to facilitate the means of communication, how to encourage collaboration, and the like.

In particular, there is a significant challenge with the description of such a process. The problem is that potential users can perceive it as too large and too generic. They can find it not really in context and on target. This difficulty in communication can make the process hard for developers and teams to digest. In the worst case it can decrease their inherent work lust and creativity. There is a risk in this event that the process hinders more than it helps. It becomes a burden. The consequence can be that developers discard the process and start inventing their own processes on the fly. The result is that the project easily gets out of control.

Now, the DA does not by itself handle all these process challenges. However, the DA has the capabilities to serve *process* on a silver plate. It does this by making the process itself more alive and by streamlining it towards supporting the individual developer and his or her team.

An interesting way to approach this issue is to think in terms of transparency, that is, that a development process should be transparent to the developer. By transparency we mean a developer should face a minimal initial threshold before being able to use the process. He should not have to assimilate a lot of knowledge before being able to follow the process. It should be easy to acquire the knowledge relevant at any specific point in time. It should be easy to put the knowledge in context and apply it to the problem at hand.

A simple analogy is driving a car: You don't ever think about applying a "driving process," do you? In particular, you do not need to be aware of everything in the car, such as your spark plugs, or the exact friction between the wheel rubber and the asphalt. Still, you may need to have some vague idea of all these things for the purpose of achieving your goal, that is, to drive from A to B. In a sense, you "just drive," more or less without having a specific process in mind.

In this sense, we find it obvious that the DA helps in achieving a transparent process because:

- The DA puts the individual developer in focus. It selectively presents and applies knowledge for the developer. It does this just in time, based on the developer's goals in terms of roles to adopt, activities to perform, and software artifacts to be responsible for. The DA empowers each individual developer.
- The DA promotes alignment and supports collaboration among developers working together as a team. It encourages the use of a development process agreed upon by all team members. It serves (via agents) as a medium for developer communication.
- The DA can adapt by detecting patterns in developer behavior. It then refines its support (rules) automatically.

Current DA Initiatives

Currently there are only a few initiatives that fall under the concept of a DA as outlined above. Two are worth mentioning here: Argo/UML (argouml.tigris.org) and Jaczone WayPointer™.

Jaczone WayPointer™ (www.jaczone.com) is an agent system primarily developed on top of products such as Rational Rose® and Rational Unified Process™. Although WayPointer is not a full DA, it is a useful first step. WayPointer is currently supporting developers that adopt roles and perform development activities within the areas of use-case modeling, analysis, and design.

A New Era Beckons

The software developer assistant (DA) empowers individual developers in their daily work assignments. It helps them work together as a team.

The DA accomplishes this empowerment by understanding the goals of the developer in terms of roles, activities, and software artifacts. It provides streamlined support in reaching those goals through an implementation consisting of intelligent software agents and well-oiled rule machinery.
As a result, the development process itself becomes transparent to the developer. The old days of initial process adoption thresholds, straitjackets, and bulkiness are over. Instead we see a new era of development processes supporting a just-when-needed approach served in the context of the developer and the development challenge at hand.

References

[GoF] Design Patterns, by E. Gamma, R. Helm, R. Johnson, J. Vlissides; Addison-Wesley, 1995.

[USDP] The Unified Software Development Process, by I. Jacobson, G. Booch, J. Rumbaugh; Addison-Wesley, 1999.

[RUP] Rational Unified Process, www.rational.com/rup; see also [Kruchten] and [Road].

[Kruchten] The Rational Unified Process, An Introduction; by Philippe Kruchten, 2nd ed., Addison-Wesley, 2000.

[Road] The Road to the Unified Software Development Process, by I. Jacobson, S. Bylund; Cambridge, 2000.

[UML] The Unified Modeling Language, by Object Management Group, www.omg.org/uml/

[J2EE] Java™ 2 Platform, Enterprise Edition; java.sun.com/j2ee/

[.NET] Microsoft® .NET, www.microsoft.com/net/

[VisualStudio] Microsoft Visual Studio® .NET, msdn.microsoft.com/vstudio/nextgen/default.asp

[RationalSuite] Rational Suite® Enterprise, www.rational.com/products/entstudio/index.jsp

[ITU-T] Specification and Description Language (SDL), ITU-T Recommendation Z.100, Mar. 1993.

[SPEM] The Software Process Engineering Metamodel (SPEM), www.omg.org

[OCL]	The Object Constraint Language: Precise Modeling with UML, by J. B. Warmer, A. G. Kleppe; Addison-Wesley, 1999. See also [UML].

[AI]	Artificial Intelligence: A Modern Approach, by S. Russell, P. Norvig; Prentice-Hall, 1995.

[ISA]	Intelligent Software Agents, by R. Murch, T. Johnson; Prentice-Hall, 1999.

[IAB]	Collection of Computer Science Bibliographies, http://liinwww.ira.uka.de/bibliography/Ai/software.agents.html

TRADEMARK NOTICE
Jaczone, Jaczone WayPointer, and WayPointer are trademarks of Jaczone AB in the United States and in other countries. Rational, Rational Suite, Rational Unified Process, and Rational Rose are trademarks or registered trademarks of Rational Software Corporation in the United States and in other countries. Microsoft is a registered trademark of Microsoft Corporation. All other names are used for identification purposes only and are trademarks or registered trademarks of their respective companies.