PARADIGMAS PARA O PROCESSO DE PRODUÇÃO DE SOFTWARE E SUAS CONSEQUÊNCIAS SOBRE OS AMBIENTES DE DESENVOLVIMENTO

SUELI MENDES; ANA REGINA ROCHA
Universidade Federal do Rio de Janeiro - COPPE
C.P. 68511; CEP 21945 - Rio de Janeiro - Brasil

RESUMO

Este trabalho discute paradigmas para o processo de desenvolvimento de software. Tendo em conta estes paradigmas, são identificadas e discutidas, características para ambientes de desenvolvimento de software.

1. INTRODUÇÃO

Antes de discutirmos o que consideramos os paradigmas para ambientes de desenvolvimento de software é conveniente deter-nos um pouco no significado da existência de paradigmas em ciência.

A discussão em torno de paradigmas tem-se dado de diferentes formas ao longo da história da ciência. Pode-se, contudo, observar uma tendência no sentido de que a ciência mais em evidência em um determinado momento trace paradigmas para as demais ciências. Como exemplo podemos observar o que aconteceu ao se considerar a matemática como um paradigma para todas as disciplinas que desejassem ser consideradas como ciência: outras áreas do conhecimento, para serem consideradas ciência, passaram a incorporar métodos matemáticos às suas metodologias de trabalho.

Para exemplificar consideremos o trabalho de Kant

"Prolegômenos a toda Metafísica futura que queira se constituir como ciência" e o trabalho de Karl Popper [7]. Ambos colocam a Física como paradigma. Popper, com seu princípio de refutação, pretende demonstrar que nem o Marxismo nem a Psicanálise de Freud podem ser consideradas ciências.

Ao considerarmos o processo de desenvolvimento de software podemos dizer que este se realiza de acordo com um dos três paradigmas que descrevemos a seguir:

1. **Abordagem artesanal**, onde a construção de software é feita por processos artesanais, sem utilização de métodos. Desta forma, o produto gerado é visto como uma obra pessoal, quase como uma obra de arte. A regulamentação jurídica, atual, usando a lei de direito autoral é uma manifestação da aplicação deste paradigma. Um ambiente de desenvolvimento de software usado neste contexto, não chega a ser algo determinante para o produto mas, simplesmente, um instrumento que o "artista" usa quando lhe é conveniente. Neste trabalho não consideramos este paradigma.

2. **Abordagem formal**, onde se procura aplicar métodos matemáticos, mais especificamente da Lógica, ao processo de desenvolvimento de software, considerando o que é bom para matemática como adequado e útil para o desenvolvimento de software.

Este paradigma conduz a um interesse de pesquisa por tudo aquilo que represente métodos formais de desenvolvimento (linguagens formais para especificação, verificação formal de

programas,etc). Ambientes de desenvolvimento construídos segundo este paradigma deverão ter ferramentas que auxiliem na utilização destes métodos.

Parodiando Kant, poderíamos escrever os "Prolegômenos para todo o software futuro que queira se constituir como ciência", ficando em aberto a questão de se as conclusões seriam tão negativas para o software como o foram para a metafísica. Que a pretensão existe não há dúvida. Neste contexto podemos citar a introdução de H.P.Ershov no "International Symposium on Theoretical Programming", onde declara:

> "Theory of computing, mathematical foundation of computer science, theory of programs, theoretical programming - are not a complete list of identifiers used. However, it is more and more recognized that these identifiers are to a certain degree synonyms signifying that a new mathematical discipline has come into beeing" (Lecture Notes in Computer Science,5, Springer Verlag,1974).

3. Abordagem empírica, onde o processo de desenvolvimento de software é visto de forma similar ao processo de desenvolvimento de produtos de Engenharia. Seguindo este paradigma, a pesquisa será na direção de buscarem-se métodos e técnicas que aproximem, o mais possível, o processo de desenvolvimento de software das características de desenvolvimento de produtos em áreas tradicionais da Engenharia.

Uma das preocupações decorrentes da aplicação deste paradigma é, por exemplo, a busca de se conseguir uma maior visibilidade do produto desde as suas fases iniciais de desenvolvimento. A pesquisa em torno de protótipos rápidos se situa neste contexto. A referência a sistemas tolerantes a falhas é outro exemplo de aplicação deste paradigma.

Ambientes de desenvolvimento de software, construídos neste contexto, procuram fornecer métodos e ferramentas análogos aos das disciplinas de Engenharia.

O processo de desenvolvimento de software é, contudo, uma atividade ainda bastante nova e pouco madura. Neste sentido é, perfeitamente, natural que se busquem paradigmas que orientem este processo. Tem-se, então, a tendência a buscar estes paradigmas em outras áreas do conhecimento. O desejável, entretanto, é que sejam encontrados modelos próprios que reflitam as características originais deste processo. Estes modelos, seguramente, conterão elementos presentes nos três paradigmas descritos acima.

Na medida em que desenvolver software significa encontrar solução para problemas, que terminarão sendo resolvidos através de algoritmos, pode-se afirmar que desenvolver software está muito próximo do processo de resolução de problemas em matemática. Outras atividades, no entanto, (por exemplo, questões relacionadas a confiabilidade e eficiência das soluções encontradas) são próximas a atividades de áreas tradicionais da engenharia. Isto nos leva a afirmar que um ambiente ideal deve considerar todos estes aspectos.

2. CARACTERÍSTICAS PARA AMBIENTES DE DESENVOLVIMENTO DE SOFTWARE

Vários ambientes de desenvolvimento de software vem sendo propostos nos últimos anos. Muitos destes, não chegaram a ser realmente úteis. Consideramos que antes de se pretender construir um novo ambiente é importante definir que características são desejáveis para ambientes de desenvolvimento de software e como estas características se relacionam com os paradigmas identificados.

2.1 O ambiente deve fornecer apoio durante todo o processo de desenvolvimento do software.

Embora não haja um consenso sobre como dividir o ciclo de vida em fases (e, nem mesmo, se é desejável definir fases) [3], [4], esse consenso é facilmente obtido no que se refere à necessidade de apoio durante todo o desenvolvimento, independentemente de como ele se faça.

A experiência tem mostrado que é de pouca utilidade utilizarem-se métodos e ferramentas de auxílio ao desenvolvimento de software que dêm apoio a apenas parte do ciclo de vida (continuaremos a usar este termo apesar da polêmica em torno). Estas metodologias e ferramentas tendem a ser de difícil integração com outras, o que traz uma série de problemas ao longo do desenvolvimento.

Um ambiente de apoio ao desenvolvimento de software deve, portanto, auxiliar durante todo o processo de desenvolvimento.

A consideração sobre o desenvolvimento de software em

fases de um ciclo está mais diretamente relacionada ao enfoque empírico que descrevemos acima. No entanto, um ambiente que apoie todo o ciclo de vida pode conter ferramentas que o aproximem do enfoque formal, por exemplo um verificador formal de programas. No entanto, se acreditamos na possibilidade de provar formalmente a correção de um software, por mais complexo que ele seja, fases do ciclo de vida tradicional tais como testes e manutenção poderiam ser eliminadas. Assim sendo, o paradigma do enfoque formal modificaria de modo drástico o ciclo de vida do software.

2.2 O ambiente deve conter um ciclo de vida, métodos e ferramentas que apoiem o desenvolvimento

A dificuldade de construir um ambiente de desenvolvimento de software realmente útil e efetivo, está diretamente relacionada à dificuldade de entender o processo de desenvolvimento de software. Entender como se dá o desenvolvimento é essencial ao se propor a construção de um ambiente.

Um ambiente de apoio ao desenvolvimento de software, pode ser definido como composto de:

- um ciclo de vida, definindo as fases do processo de desenvolvimento e as atividades a serem realizadas em cada fase;
- métodos, usados para organizar o pensamento e o trabalho do usuário ao longo do processo de desenvolvimento, e,
- ferramentas, automatizadas ou não, que tornam possível utilizar os métodos.

Diferentes aplicações têm características diferentes. Assim sendo, seus processos de desenvolvimento também têm diferentes características. Neste sentido, características de aplicações devem determinar a escolha de um ou outro método. Um ambiente deve conter diferentes tipos de ferramentas: ferramentas para descrição do problema (linguagens de representação e editores), ferramentas para avaliação da qualidade e ferramentas para apoio à gerência (estimadores de custo, acompanhadores de projetos, etc).

2.3 O ambiente deve conter várias linguagens formando uma hierarquia, cada uma delas apropriada para descrever o produto em um determinado nível de detalhe

Durante o processo de desenvolvimento, são geradas diversas representações do software. Em cada uma delas, o produto é descrito com um determinado objetivo. Assim, tem-se representações cujo objetivo é possibilitar a interação com o usuário para fins de definição do produto a ser desenvolvido. Em outros momentos, as representações geradas têm outros objetivos (por exemplo a comunicação com o projetista ou com o implementador) até se chegar à representação final (o software-produto) cujo objetivo é atender aos requisitos do usuário que motivaram sua construção.

Estas representações são produzidas utilizando-se linguagens e a qualidade do produto final dependerá da qualidade e adequação destas linguagens. Existem, atualmente, várias linguagens propostas. Algumas abordagens pretendem ter linguagens que sejam adequadas para uso durante todo o

processo de desenvolvimento, como por exemplo as linguagens ADA [5] e GYPSY [1]. Outras abordagens propõem uma sequência de linguagens a serem usadas ao longo do desenvolvimento, como por exemplo o ambiente HDM [9].

Condideramos mais adequado ter um ambiente que contenha um conjunto de linguagens, cada uma apropriada para descrever o produto em um determinado nível de detalhe. Estas linguagens deverão formar uma hierarquia, onde as linguagens de nível mais alto serão usadas no início do desenvolvimento e terão como objetivo favorecer a interação com o usuário, seja por seu alto grau de comunicabilidade ou por possibilitarem a construção de protótipos. No primeiro caso, as pesquisas em torno do processamento de linguagens naturais, em Inteligência Artificial, poderiam levar ao nível ideal de comunicação (guardadas as devidas restrições). Veja-se, como exemplo, SAFE [2].

O principal problema do uso de diferentes linguagens está na sua integração. A grande motivação de se usar uma única linguagem, como no projeto ADA, reside certamente nesta questão. No entanto, se tivermos ferramentas para auxiliar nesta integração, o problema estaria em parte resolvido, o que nos remete ao item abaixo.

2.4 O ambiente deve fornecer auxílio semi-automatizado, para migrar de uma linguagem na hierarquia para outra.

Migrar de uma linguagem para outra na hierarquia não é uma tarefa trivial. O ambiente deve conter métodos que indiquem,

claramente, o processo e os passos a serem seguidos. Idealmente deve, pelo menos, fornecer um apoio semi-automatizado para a realização desta tarefa.

Este apoio deve possibilitar caminhar na hierarquia nas duas direções. Embora se tenha claro que desenvolvimento "top-down" é a melhor estratégia para se controlar a complexidade, sabe-se que o desenvolvimento de software dificilmente se dá de uma forma puramente "top-down". De forma similar à Engenharia é comum, ao longo do processo de desenvolvimento de software, a necessidade de modificações e estas exigem o uso da estatégia "bottom-up".

2.5 O ambiente deve conter linguagens que permitam descrever o mesmo problema sob diferentes pontos de vista permitindo, assim, detectar inconsistências e conflitos.

O ambiente pode, ainda, conter mais de uma linguagem para um mesmo nível na hierarquia. Estas linguagens possibilitariam descrever o produto sob mais de um ponto de vista (por exemplo, atividades e dados [8]), tornando possível a detecção de inconsistências e conflitos.

Esta característica é especialmente útil se existirem mecanismos automatizados de verificação da consistência entre as descrições.

2.6 O ambiente deve conter um método para avaliação da qualidade do produto (validação) e do processo de desenvolvimento (verificação).

Um ambiente para desenvolvimento de software deve conter

ferramentas para controle da qualidade tanto no que se refere aos produtos gerados (validação) quanto ao processo de desenvolvimento destes produtos (verificação). Estas ferramentas, por sua vez, devem estar baseadas em algum método para avaliação da qualidade que nos permita confiar nas avaliações realizadas.

2.7 O ambiente deve ser de uso amigável, guiando o usuário através de um determinado método e auxiliando-o no uso das ferramentas.

Sabe-se, na prática, que forçar os usuários a usar métodos novos e desconhecidos para eles é tarefa árdua e, em alguns casos, impossível. Um exemplo disso, é tentar forçar o analista e o programador sem maturidade matemática a usar métodos formais. No entanto, se desejamos melhorar a qualidade do software essa tarefa deve ser empreendida de alguma forma.

Acreditamos que um modo de atacar o problema é oferecer ambientes com métodos e ferramentas de mais fácil uso, que convençam o usuário mais recalcitrante da vantagem de usá-los e procurar, então, fazê-lo tentar o uso de instrumentos mais sofisticados. O problema de qualquer forma é de difícil solução. E a única maneira que divisamos de uma possível solução, é investir na educação da próxima geração formando engenheiros de software com novos hábitos.

2.8 O ambiente deve conter uma biblioteca de componentes de forma a permitir a re-utilização de especificações, projeto e

código.

A re-utilização de módulos é algo que traz tantas vantagens que, em geral, não está sujeita a grandes polêmicas. Pode-se, é claro, discutir se se deve levar isto como uma meta. Módulos podem ser construídos com o objetivo de serem re-utilizados em diferentes tipos de aplicações. Estes módulos tendem a ser demasiado gerais o que pode prejudicar a eficiência. Se este é um requisito importante para o produto, a questão da re-utilização pode ter discutida a sua aplicabilidade.

Outro aspecto a ser considerado é a conveniência de se re-utilizar, não apenas módulos, mas também especificações e projeto. Veja-se, neste caso, o sistema DRACO [6]. Para isso são necessárias bases de conhecimento sobre o domínio das aplicações o que nos remete a outra característica (incorporação de técnicas de Inteligência Artificial ao processo de desenvolvimento de software).

2.9 O ambiente deve permitir a realização de verificações formais e/ou a construção de protótipos rápidos.

Este item está de acordo com a nossa posição de que um ambiente de desenvolvimento de software deve ter seu modelo próprio sem se conformar exclusivamente a nenhum dos paradigmas citados. Isto, entretanto, não quer dizer ficar "em cima do muro" com relação às posições radicais. Acreditamos, realmente que existe espaço e, principalmente, diversidade de aplicações que podem acomodar instrumentos incorporados dos vários paradigmas.

2.10 O ambiente deve incorporar técnicas de Inteligência Artificial ao processo de desenvolvimento de software.

Consideramos, aqui, a possibilidade do casamento entre Engenharia de Software e Inteligência Artificial e não apenas da Engenharia de Software com Sistemas Especialistas [10].

A união entre Inteligência Artificial e Engenharia de Software pode ser bastante frutífera. De um lado Inteligência Artificial já está motivando novas linhas de pesquisa em Engenharia de Software, tanto do ponto de vista do próprio ciclo de vida quanto das metodologias a serem desenvolvidas e aplicadas, com vantagens para ambos os lados. Por outro lado, ferramentas de Inteligência Artificial vem sendo e deverão continuar sendo utilizadas em Engenharia de Software, o que nos parece ter um grande potencial para se conseguir sair do longo túnel, apelidado "crise de software".

Quanto ao problema da utilização de Sistemas Especialistas em Engenharia de Software, acreditamos que sua validade é, ainda, como pesquisa acadêmica e que é prematuro se partir, neste momento, para a construção de sistemas especialistas com possibilidades de lançamento no mercado.

2.11 O ambiente deve ser implementado de forma incremental, de modo que as primeiras ferramentas desenvolvidas possam ser usadas e criticadas por seus usuários possibilitando melhorar a qualidade do ambiente.

Desenvolvedores de ambientes costumam recomendar que estes sejam desenvolvidos de forma incremental. Concordamos com esta abordagem pois acreditamos ser melhor partir de algo

simples e concreto, mas que possa ser utilizado em prazo relativamente curto do que partir, logo de início, para grandes metas exigindo longos prazos.

3. CONCLUSÕES

Procuramos discutir, aqui, o efeito que a adoção de um determinado paradigma para a atividade de desenvolvimento de software pode ter na criação de ambientes.

Tentamos detectar posições extremadas que podem levar à exclusão de métodos e ferramentas em detrimento de outras ao se adotar um determinado paradigma.

Existem várias razões fortes para considerar que qualquer posição radical, neste sentido, pode ser prejudicial:

- métodos formais tem sido grandemente pesquisados, mas ainda não foram demonstrados, para software grandes e complexos, como sendo realmente exequíveis. A esperança reside no desenvolvimento de métodos automáticos ou semi-automáticos, possivelmente, derivados de técnicas de Inteligência Artificial (como provadores automáticos de teoremas);
- protótipos representam pesquisas recentes e a demonstração cabal de que o esforço dispendido para realizá-los é compensado em termos de custos do desenvolvimento, não foi ainda conseguido. Por outro lado, as transformações que os protótipos devem passar nos vários níveis desde a primeira especificação até o sistema totalmente codificado ainda não estão claramente definidos. Muito esforço de pesquisa e experimentação

está por ser feito;

- tanto a utilização de métodos formais como de construção de protótipos demandam esforços dos usuários que exigem persuação, demonstração de sua utilidade, etc. Assim, pode-se considerar que para projetistas de sistemas de tempo-real, por exemplo, a demonstração da utilidade da construção de protótipos é muito mais fácil do que convencê-lo a utilizar métodos formais baseados em Lógica Temporal.

Em resumo, o que propomos, aqui, é que se caminhe no sentido de encontrar um novo paradigma como modelo para a atividade de desenvolvimento de software. Este novo paradigma deve incorporar elementos de vários paradigmas encontrados ao longo da história da Ciência e da Tecnologia, uma vez que não acreditamos na total ruptura com o passado. E aí adotamos o paradigma do evolucionismo de Darwin, que tanto influenciou várias ciências, mas no nosso caso temperando-o com a possibilidade de saltos de vez em quando.

REFERÊNCIAS

[1] A.L.Ambler et al "GYPSY: a language for specification and implementation of verifiable programs"; In Proceedings of the ACM Conference on Language Design for Reliable Software; março 1977

[2] R.Balzer et al "Informality in program specifications"; IEEE Transactions on Software Engineering, vol-SE-4, n 3, Julho 1978

[3] B.I.Blum "The Life Cycle: a debate over alternate models", ACM SIGSOFT Software Engineering Notes, vol 7, n 4, outubro 1982

[4] G.R.Gladden "Stop the life cycle, I want to get off, ACM SIGSOFT Software Engineering Notes, vol 7, n 2, abril 1982

[5] Ichbiah et al " Ada Programming Course, ALSYS, La Cella Saint Cloud, França, dezembro 1980

[6] J.M.Neighbors Software Development using Components: Tese de Doutorado, Universidade da California, Irvine, 1981

[7] K.Popper The Logic of Scientific Discovery; Routledge and Kegan Paul, Londres, 1959

[8] D.T.Ross "Structured Analysis (SA): a language for communicating ideas"; IEEE Transactions on Software Engineering, vol SE-3, n 1, Janeiro 1977

[9] B.A.Silverberg "An overview of the SRI hierarchical development metodology"; in Software Engineering Environments; Hunke,H. ed.; North Holland; 1981

[10] W.T.Znaikerman "Expert Systems and software Engineering: ready for marriage?"; IEEE EXPERT; Inverno 1986