Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciências da Computação

LEONARDO DE ANDRADE PRATES

Diretrizes para Desenvolvimento de Software para Sistemas Embarcados

Belo Horizonte
2012

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciências da Computação
Especialização em Informática: Ênfase: Engenharia de Software

Diretrizes para Desenvolvimento de Software para Sistemas Embarcados

por

LEONARDO DE ANDRADE PRATES

Monografia de Final de Curso

Prof. Dr. Marco Tulio de Oliveira Valente
Orientador

Belo Horizonte
2012

LEONARDO DE ANDRADE PRATES

Diretrizes para Desenvolvimento de Software para Sistemas Embarcados

Monografia apresentada ao Curso de Especialização em Informática do Departamento de Ciências Exatas da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do grau de Especialista em Informática.

Área de concentração: Engenharia de Software

Orientador: Prof. Dr. Marco Tulio de Oliveira Valente

Belo Horizonte
2012

Folha de aprovação

AGRADECIMENTOS

Agradeço primeiramente a Deus.

À minha esposa e filha pelo apoio e compreensão nos momentos que estive ausente.

Ao meu amigo João Luiz Neves pelo incentivo na busca e aprimoramento do conhecimento.

Ao professor Marco Tulio, pela paciência e valorosa atenção que a mim concedeu durante todo o desenvolvimento deste trabalho.

“O segredo da criatividade é saber esconder as fontes”

Albert Einstein (1879-1955)

RESUMO

Os sistemas embarcados estão cada vez mais presentes no cotidiano das pessoas. Ele está presente de brinquedos a sistemas de suporte a vida. As características peculiares do sistema em que o software está inserido requerem cuidados que não são abordados pela computação de propósito geral e nem pelas escolas de engenharia; existe, portanto, uma lacuna de conhecimento entre os profissionais de computação e engenharia. O objetivo deste trabalho é apresentar tais peculiaridades e demonstrar porque é importante uma abordagem sistemática de engenharia de software. Serão abordados conceitos fundamentais de métodos e processos de engenharia de software e sua aplicabilidade ao software embarcado e os aditivos necessários para garantir a correta abrangência nesse tipo de sistema. Ao final será apresentada uma proposta de processo de desenvolvimento de sistema embarcado, com foco em software, e a documentação mínima que deve ser gerada para garantir a formalização dessa proposta.

Palavras-chave: Software embarcado, sistemas embarcados, sistemas embutidos, engenharia de software, *embedded system*.

ABSTRACT

The embedded systems are more and more present in daily life. It is present from toys to life support systems. The peculiar characteristics of the system where the software is embedded require care that are not covered by general purpose computing and neither by engineering schools, therefore, there is a knowledge gap between computer professionals and engineering. The aim of this paper is to present such peculiarities and demonstrate why it is important a systematic approach of software engineering. It will examine the fundamental concepts of methods and processes of software engineering and its applicability to embedded software and the necessary additives to ensure proper coverage in this type of system. At the end, we will present a proposal for embedded system development process, with software focus, and the minimum documentation that must be generated to ensure the proposal formalization.

Keywords*:* Embedded software, embedded system, software engineering.

LISTA DE FIGURAS

LISTA DE SIGLAS

ABS	*Anti-block system*
API	*Application programming interface*
CBSE	*Component-based Software Engineering*
CPU	*Central Processing Unit*
FOB	*Free on board*
OO	Orientação a objetos
PC	*Personal computer*
PWM	*Pulse width modulation*
USB	*Universal serial bus*
XP	*Extreme Programming*
YAGNI	*You ain't gonna need it*

SUMÁRIO

1 INTRODUÇÃO

O desenvolvimento de software para *personal computers* (PC) apesar de algumas variações de sistema operacional é feito em arquitetura padronizada (x86) e a interação com o hardware e o ambiente é completamente abstraída através de *application programming interfaces* (APIs), fornecidas pelos desenvolvedores dos periféricos. Esse é o principal ponto de diferença para o sistema embarcado, que como o termo "sistema" sugere é formado pela integração de hardware, software e interação com o ambiente. Para ser desenvolvido não é possível ter uma visão apenas de software ou hardware, são necessários conhecimentos de ambas as disciplinas e uma visão de engenharia de sistemas para a correta operação do todo.

Nesse contexto é fácil perceber a lacuna de conhecimento existente: engenheiros são treinados para desenvolver hardware e cientistas da computação treinados para desenvolver software.

Historicamente, engenheiros com formação em eletroeletrônica tem assumido o papel de desenvolvedores desse tipo de sistema, pois aparentemente existe uma maior facilidade em se aprender a programar do que aprender a desenvolver hardware. A questão é que esse aprendizado de software é feito geralmente em uma única cadeira de programação básica nos cursos de engenharia.

A assertiva de que aprender a programar é menos complexo do que aprender a desenvolver hardware não é totalmente incorreta. Realmente aprender a desenvolver hardware exige um conjunto de disciplinas específicas que não são vistas em cursos de ciência da computação e nem mesmo são oferecidas em cursos de pós-graduação.

No geral, em empresas de pequeno porte, os profissionais desenvolvem todo o sistema e, portanto, são quase que integralmente engenheiros. Isso não exclui os profissionais de ciência da computação desse mercado relativamente novo e promissor, mas os restringe a empresas de médio a grande porte, que possuem equipes bem definidas de hardware, software e sistemas. Ainda assim, essas equipes de software precisam de profissionais de engenharia especializados em desenvolver APIs especificas de interação com o hardware, para tornar o trabalho da equipe de software um pouco mais parecido ao encontrado no ambiente PC, ou seja, com a devida abstração dessa camada.

Para os profissionais de computação interessados em conhecer alguns conceitos básicos de hardware para sistemas embarcados, o livro do Simon (2005), pode ser um guia valioso.

1.1 Objetivos

O objetivo desse trabalho é apresentar, principalmente para os engenheiros de hardware, as ferramentas de engenharia de software para desenvolver software com qualidade, com a mesma sistemática de engenharia a qual já estão acostumados.

Devido a extrema agilidade do mundo atual, uma das grandes preocupações desses profissionais é que a documentação adicional a ser gerada possa atrasar o desenvolvimento. O que se pretende demonstrar é um conjunto mínimo de documentos e boas práticas a serem seguidas para garantir a qualidade do sistema. E demonstrar que a sistematização do desenvolvimento integrado de software e hardware pode inclusive acelerar o desenvolvimento ou no mínimo mantê-lo dentro do prazo, pela minimização dos erros a serem corrigidos.

A expressão “qualidade do sistema” no parágrafo anterior não é por acaso. Uma vez que um sistema embarcado é inerentemente composto de hardware e software o processo de engenharia de software deve levar em conta as características de acoplamento dessas duas vertentes.

1.2 Metodologia

A metodologia adotada será a exposição dos conceitos básicos de sistemas embarcados, engenharia de software e em que pontos as teorias clássicas se aplicam ou não ao problema. Ao final da exposição será proposto o processo ou composição deles que melhor se adequa ao desenvolvimento de sistemas embarcados e quais aditivos serão necessários para garantir a completude da abordagem.

2 SISTEMAS EMBARCADOS

Não existe uma definição universalmente aceita para sistemas embarcados e cada autor tem uma forma de abordá-la. De acordo com Noergaard (2005) sistemas embarcados são sistemas compostos de hardware e software com severas restrições, que desempenham uma tarefa específica e devem ter alta confiabilidade e qualidade.

Nesse contexto, celulares, *tablets* e outros dispositivos que utilizam sistemas operacionais de alto nível (Android, Linux, Windows CE) não devem ser considerados sistemas embarcados, pois permitem, através de aplicativos, adição de infinitas funcionalidades e completa abstração da camada de hardware para os desenvolvedores. Esses dispositivos são equipados com processadores que possuem capacidade de processamento, armazenamento e memória volátil muito superior aos computadores pessoais de 10 anos atrás. Portanto, apesar das controvérsias sobre o assunto, o foco desse trabalho é a abordagem de sistemas embarcados com microcontroladores de baixo custo, com fortes restrições de memória e velocidade de processamento; onde o mais avançado, com arquitetura ARM, possui *clock* de 72MHz, 512KBytes de memória de programa e 58KBytes de memória volátil. Um grande avanço, num mercado que até pouco tempo atrás utilizava arquiteturas PIC com menos de 1KByte de memória de programa, 64Bytes de memória volátil e *clock* de 4MHz.

Do ponto de vista dos profissionais da computação, acostumados com arquitetura x86, com centenas de megabytes de memória, mesmo a arquitetura mais avançada é um tanto modesta. Contudo, para os desenvolvedores de sistemas embarcados esse aumento de disponibilidade de memória significou um aumento considerável de novas funcionalidades que os projetos passaram a demandar.

E é justamente nesse aumento de funcionalidades que está o problema atual do desenvolvimento de software embarcado. Fazer software com poucas funcionalidades não exige, a princípio, mecanismos muito severos de controle de processo, devido a baixa complexidade. Mas esse aumento repentino da complexidade, criado pelo barateamento dos microcontroladores está gerando para os desenvolvedores dificuldades de garantir que o software é confiável o suficiente para ser comercializado. Como mencionado anteriormente, o profissional que desenvolve esse tipo de software não possui o conhecimento e ferramentas necessárias para ter a situação controlada dentro de um processo de engenharia.

2.1 Tendências

Segundo pesquisa realizada pelo IDC (MORALES, RAU, PALMA, VENKATESAN, PULSKAMP e DUGAR, 2011), cerca de 19% de todos os eletrônicos vendidos hoje no mundo são sistemas embarcados e este número deve crescer para 33% até 2015. Isso representa um consumo de 14,5 bilhões de processadores por ano, mais de dois por habitante no mundo.

Os sistemas embarcados estão presentes em:

a) Eletrônicos de consumo – câmeras fotográficas, aparelhos de DVD, vídeo games portáteis, calculadoras;
b) Linha branca – microondas, máquinas de lavar, sistemas de segurança;
c) Escritório – aparelhos de fax, scanners, impressoras, telefones;
d) Comércio – máquinas de cartão de crédito, máquinas registradoras, leitor de código de barras;
e) Automóveis – sistemas de injeção, tacógrafos, freios ABS, suspensão ativa, cambio automático, alarme, central de vidros elétricos. Um único veículo do seguimento *premium* pode conter setenta centrais eletrônicas distribuídas (BROY, 2006).

Pela diversidade de aplicações é fácil explicar os números da pesquisa e observar o quão promissor é esse mercado e o número de profissionais especializados demandados. A China, um dos maiores desenvolvedores desse tipo de sistema no mundo, já identificou a necessidade de profissionalização e possui vários cursos de especialização em sistemas embarcados com foco em engenharia de software e sistemas (YIMEI, 2007). No Brasil, entretanto, existem poucos grupos de pesquisa sobre o assunto.

2.2 Aspectos específicos

Abaixo estão listados alguns aspectos peculiares de sistemas embarcados, que os tornam especiais em relação aos computadores de uso geral (MARWEDEL, 2003):

a) Desempenham uma única função – ao contrário de computadores capazes de executar e instalar vários programas, os sistemas embarcados executam um único programa de maneira contínua.
b) Restrição de memória – a memória de armazenamento, de programa e volátil disponível para realizar as funcionalidades é pequena. O código, as estruturas de dados e os algoritmos de compreensão devem ser otimizados para permitir a implantação de todos os requisitos.
c) Restrição de custo – geralmente são vendidos em grande escala e os recursos de hardware e software são planejados para atender somente os requisitos imediatos para reduzir os custos, limitando a capacidade de expansibilidade e manutenibilidade, prática que não deve ser adotada (KOOPMAN, 2010).
d) Restrição temporal – neste aspecto pode ser dividido em tempo sensível e tempo crítico. Os de tempo crítico são intolerantes a atrasos e devem ser executados no tempo exato. O *airbag* é um bom exemplo: um pouco antes ou um pouco depois do tempo exato, ele não irá cumprir sua função.
e) Restrição de consumo de energia – devem gastar o mínimo possível de energia para garantir o uso eficiente e racional dos recursos, reduzir custos e tamanho.
f) Reativo ao ambiente – muitas vezes é projetado para interagir com o ambiente através de sensoriamento e atuadores, funcionando de forma autônoma sem nenhum tipo de interação com um ator humano. E essa interação pode ter restrições temporais como mencionado no item d.
g) Tolerância a falhas – em um computador quando um software falha, ele simplesmente para de responder ou no máximo dá uma mensagem de erro ao usuário. O usuário deve reagir para resolver a situação. Caso o software apresente problemas depois de lançado no mercado, pode ser aplicado um *patch* para correção ou envio de um novo executável com a nova versão. No sistema embarcado muitas vezes não existe nenhum tipo de interação com usuário, e nem métodos de atualização e eles podem estar distribuídos por todo o mundo em milhões de unidades.
h) Ferramentas de desenvolvimento dedicadas – um software feito para arquitetura x86 é desenvolvido e testado diretamente no ambiente onde será usado. Nos sistemas embarcados o software desenvolvido irá funcionar em uma arquitetura completamente diferente. Nesse caso são necessárias ferramentas de *cross compile* para gerar o código na arquitetura de destino e interfaces específicas

para gravar e testar o software no hardware onde será usado. Sem a plataforma final, não é possível fazer boa parte dos testes, principalmente quando o sistema é altamente reativo ao ambiente, uma vez que quase sempre não é possível simular tal situação;

i) Ligados em tempo integral – uma estratégia muito comum em computadores é o de reiniciar quando o sistema se torna instável. Muitos sistemas embarcados funcionam em tempo integral, às vezes por décadas, e não podem passar pelo processo de reinicialização. Imagine ter que reiniciar uma central de *anti block system* (ABS) ou de injeção eletrônica na mesma frequência com que se reinicia um computador pessoal. A pessoa estaria em uma rodovia e teria que parar de vez em quando porque o carro está com comportamento "lento". Isso às vezes até acontece, mas o veículo possui estratégias de reconhecimento de falha no sensoriamento e passa a trabalhar sobre mapas padrão definidos em fábrica, para garantir o funcionamento adequado até que a origem da falha seja corrigida. A falha, inclusive, é indicada no painel e armazenada em *log* para identificação posterior;

Pelas características expostas é fácil observar que requisitos não funcionais em sistemas embarcados são de extrema importância e devem ser mensuráveis para garantir que foram corretamente atendidos (SOMMERVILLE, 2007).

Os sistemas embarcados devem ser confiáveis, robustos, eficientes e ter baixo custo de desenvolvimento.

3 ENGENHARIA DE SOFTWARE

O conceito de engenharia de software surgiu em 1968 durante a crise de software. Ela se iniciou pela evolução do computador que ano após ano aumentava seu poder de processamento e diminuía seu preço (SOMMERVILLE, 2007), algo semelhante ao que está ocorrendo com os microcontroladores e os sistemas embarcados. O software se tornava cada vez mais complexo e os desenvolvedores não sabiam lidar com essa nova realidade. Os projetos invariavelmente custavam muito, o prazo era maior que o estimado e não funcionava a contento.

Os projetos de engenharia de modalidades tradicionais (civil, elétrica, mecânica) possuíam níveis de acerto elevados graças a uma sistemática de desenvolvimento bem definida. A ideia era sistematizar também o desenvolvimento de software para se obter resultados similares.

Conforme Sommerville (2007), a engenharia de software está relacionada com todos os aspectos de produção de software, desde os estágios iniciais até sua manutenção. A engenharia de software permite adotar uma abordagem sistemática e organizada durante o desenvolvimento de software. Isto gera produtos de alta qualidade com custo adequado.

Essa sistemática permitiu uma melhora significativa no desenvolvimento de software, contudo os *stakeholders* perceberam que o software ao contrário do hardware era flexível e permitia mudanças. Essa flexibilidade criou uma visão equivocada de que em software tudo é possível, rápido e barato. Os métodos tradicionais utilizados até então não eram adequados a essa nova realidade de mudanças contínuas surgindo então a proposta das metodologias ágeis com o manifesto ágil (2001).

3.1 Métodos e processos de engenharia de software

Nesse ponto é importante diferenciar os conceitos de métodos e processos de engenharia de software. Métodos de software fornecem a técnica de como fazer: as regras, notações e modelos que devem ser seguidos para se desenvolver software de forma estruturada e uniforme. Processo de software é o conjunto de atividades que devem ser seguidas para se obter estabilidade, organização e controle (PRESSMAN, 1995).

Conforme Schwartz (1975) e Sommerville (2007) independente do processo de software adotado, todos apresentam as seguintes atividades de uma forma geral: especificação, desenvolvimento, validação e evolução. Obviamente dentro de cada atividade existe um conjunto de subatividades.

3.1.1 Especificação de software

Nessa etapa do processo todos os requisitos e restrições do software devem ser elicitados. É uma etapa muito importante, pois os erros introduzidos nessa etapa são os mais caros de serem corrigidos.

Deve conter tudo que o software deve ou não fazer, ser consistente, preciso, mensurável e rastreável. É o contrato que será seguido para desenvolver o software e tudo deve ser acordado nessa etapa.

3.1.2 Desenvolvimento de software

É o processo de transformação dos requisitos em software operacional. Envolve a criação de modelos abstratos que depois são convertidos para uma linguagem de programação, no caso de métodos tradicionais; ou um conjunto de práticas, em métodos ágeis.

3.1.3 Validação de software

Consiste em uma série de atividades de teste que tem por objetivo garantir que o software atende a todos os requisitos. Os requisitos devem ser mensuráveis para garantir que foram plenamente atendidos.

3.1.4 Evolução de software

O software possui vida relativamente longa e é natural que novas funcionalidades se tornem necessárias para continuar útil. A evolução é a parte mais desafiadora da engenharia de software. Geralmente é conduzida por equipes menos experientes (os mais experientes preferem o desenvolvimento de novos sistemas), e a

qualidade do software tende a cair segundo as leis de Lehman (SOMMERVILLE, 2007).

3.2 Métodos tradicionais

As metodologias tradicionais são focadas no desenvolvimento orientado a documentação. Elas pregam que nada pode ser feito sem que tudo esteja devidamente documentado e toda e qualquer mudança deve ser documentada antes de ser realizada no software.

O modelo tradicional mais conhecido é o modelo cascata (ROYCE, 1970). Esse modelo trata as atividades do processo de forma sequencial e independente (Figura 1). Ao fim de cada etapa, a documentação associada deve ser formalmente aprovada e assinada para passar para a etapa seguinte.

As principais críticas a esse modelo são:

- A entrega de software funcional só ocorre na etapa de integração e, caso algo esteja comprometido será identificado em uma etapa bastante avançada e sua correção será dispendiosa;
- Os custos de criar, manter e gerenciar a documentação são altos;
- O processo é rígido e pouco receptivo a mudanças. Está, portanto, restrito a software que possua requisitos muito bem definidos no momento da especificação e com baixa probabilidade de mudanças. Sommerville (2007) sugere que seja utilizado em grandes projetos com equipes distribuídas.

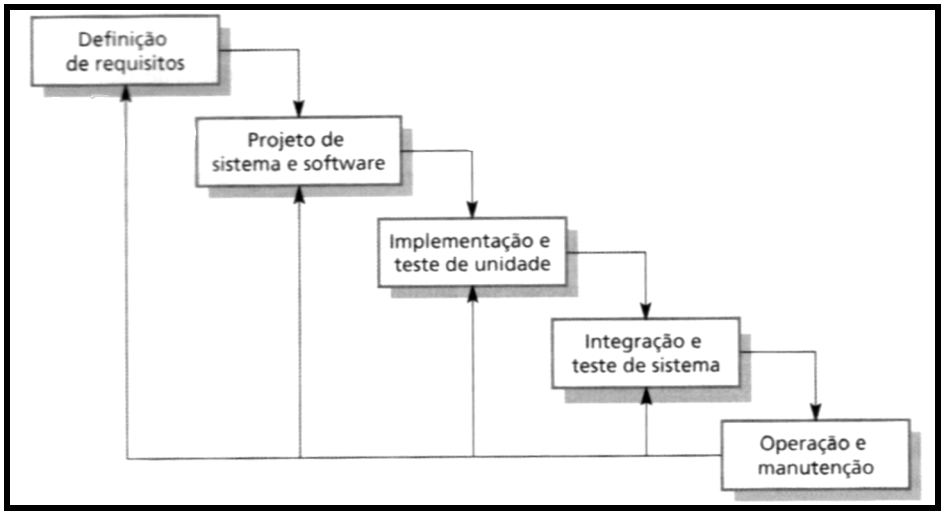

FIG. 1 Fases do modelo cascata. Fonte: SOMMERVILLE (2007).

3.3 Desenvolvimento rápido de software

Um modelo de processo considerado mais adequado ao desenvolvimento de software é o desenvolvimento evolucionário (Figura 2). O desenvolvimento evolucionário baseia-se na entrega de funcionalidades de maneira incremental e contínua, submetendo o software para que o cliente dê seu *feedback* sobre o resultado do incremento e priorize as novas funcionalidades que serão adicionadas, incorporando assim as funcionalidades mais importantes primeiro. Isso permite que as especificações sejam geradas ao longo do desenvolvimento, de forma que não precisam ser totalmente compreendidas no início. O software construído dessa forma tem mais chances de sucesso, uma vez que a cada incremento as atividades de processo são aplicadas e o software começa a ser integrado e testado desde o início. Caso ocorra um defeito durante a integração é muito mais fácil corrigir do que a integração *big bang* (de uma única vez) que ocorre no sistema cascata.

As críticas a esse modelo são basicamente as seguintes:

- Como o software não é totalmente definido no início é mais difícil mensurar seus custos e prazos, ou seja, apresenta dificuldades de gerenciamento e preço;
- A mudança tende a corromper a estrutura, conforme enunciado pelas leis de Lehman, e as mudanças tornam-se caras;

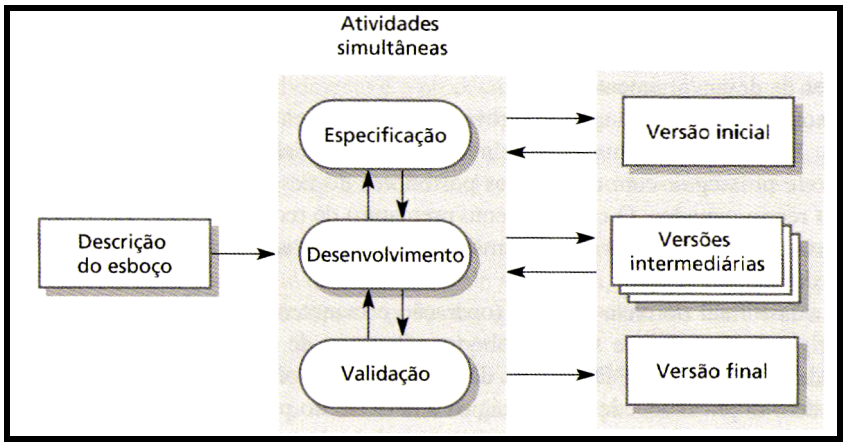

FIG. 2 Desenvolvimento evolucionário. Fonte: SOMMERVILLE (2007).

Uma abordagem recente é a engenharia de software baseada em componentes (CBSE). Essa abordagem se baseia no desenvolvimento focado no reuso de componentes prontos (pagos ou não), que devem ser integrados ao software para adicionar funcionalidades de forma rápida. Não é uma ideia recente, os desenvolvedores geralmente procuram software semelhante dentro da própria organização ou comunidades de software livre para facilitar o desenvolvimento, uma vez que o desenvolvimento de software é uma tarefa repetitiva por natureza. O que a CBSE tenta é trazer uma sistemática de como realizar essa integração de maneira formal.

As principais desvantagens desse modelo são as seguintes:

- O fornecedor do componente pode não manter a funcionalidade entre versões;
- O fornecedor do componente pode abandonar o suporte;
- O código fonte geralmente não está disponível, impedindo adaptações e correções pontuais;
- O componente pode não possuir a funcionalidade necessária de forma completa, e com isso os requisitos podem ter que ser adaptados a essa realidade;
- Defeitos em software obtido de comunidades de código livre podem ser demorados de corrigir, pois dependem dos membros dessas comunidades que nem sempre tem essa atividade como compromisso;

A dependência de software que não está diretamente sobre o controle das organizações pode ser uma das causas da CBSE ter seu principal uso em sistemas de informação (SOMMERVILLE, 2007) em detrimento a outros tipos de sistema.

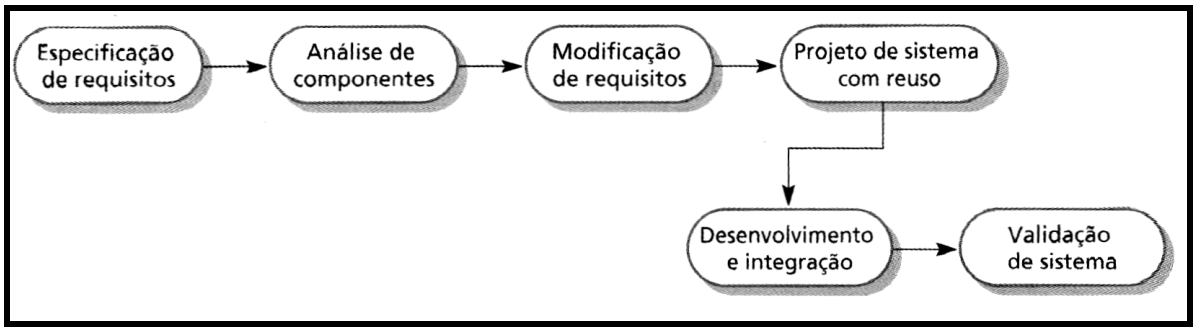

FIG. 3 Engenharia de software baseada em componentes. Fonte: SOMMERVILLE (2007).

3.4 Métodos ágeis

Os métodos ágeis tem sua origem baseada no desenvolvimento evolucionário. A formalização iniciou-se com a criação da Aliança Ágil e estabelecimento do manifesto ágil (AGILE, 2001).

As metodologias ágeis compartilham uma série de características que são a base fundamental dessa nova abordagem de desenvolvimento de software:

- Envolvimento do cliente – o cliente deve estar envolvido em todas as etapas de desenvolvimento e deve ser a pessoa que representa o interesse de todos os *stakeholders*. O cliente deve priorizar e validar cada incremento do software;
- Entrega incremental – o software é desenvolvido de maneira incremental, agregando as funcionalidades mais importantes primeiro, de forma a se obter software funcional e útil já nas primeiras iterações;
- Foco nas pessoas – o foco deve ser na equipe de desenvolvimento, nas suas características e habilidades individuais, buscando a melhor forma de explorá-las. E não forçá-las a se adequar a um processo onde elas não renderão 100% do que são capazes de oferecer. As pessoas que formam a equipe devem ser altamente especializadas e ter boa relação com os demais membros da equipe;
- Aceite as mudanças – esse é o ponto central das metodologias ágeis. No mundo atual tudo muda rápido e os produtos devem se adequar com igual velocidade a essas mudanças. Como a mudança inevitavelmente irá ocorrer os métodos ágeis

pregam que o desenvolvimento não deve ser baseado em estimativas de expansão ou do que será necessário no futuro e sim devem ser simples e fazer exatamente o que é necessário agora *You Ain't Gonna Need It* (YAGNI);

- Simplicidade – o software deve ser simples, nada de estruturas complexas que impedem que outros entendam o que foi feito para dar impressão de habilidade. O foco deve ser o mínimo necessário para realizar o requisito de forma correta. A cada iteração o software deve ser refatorado para retirar a complexidade e permitir uma estrutura enxuta e robusta para receber novas funcionalidades sem a degradação inerente da mudança.
- Uso de metáforas – deve-se usar uma linguagem natural ao cliente, para que ele entenda o que está sendo feito, afinal ele estará envolvido o tempo todo no desenvolvimento;

Nem todos os estudiosos consideram essa abordagem suficiente, alguns consideram que deve haver uma mistura do paradigma ágil com o tradicional, para criar uma metodologia que não fique nos extremos (DEMARCO e BOEHM, 2002).

Alguns pontos considerados problemáticos na metodologia ágil são os seguintes:

- O cliente nem sempre tem disponibilidade, interesse ou capacidade para estar envolvido no desenvolvimento;
- Os desenvolvedores no desenvolvimento ágil devem formar uma equipe coesa e membros com má relação interpessoal não serão capazes de formar tais equipes, mesmo que altamente especializados;
- A priorização pode ser complexa, caso os *stakeholders* tenham necessidades adversas. Por essa razão, o ideal é que exista um único *stakeholder* capaz de representar os interesses dos demais com seu aval;
- A refatoração necessária para manter a simplicidade pode não ser feita em caso de pressão por prazos de entrega e o software pode se corromper;

As metodologias ágeis mais conhecidas e utilizadas são o Extreme Programming (XP) e o Scrum.

3.4.1 Extreme programming

A *Extreme Programming* (XP) é uma metodologia ágil de desenvolvimento de software para equipes pequenas e médias que desenvolvem software baseado em requisitos vagos e que se modificam rapidamente (BECK, 2000).

No XP os requisitos são elicitados como histórias de usuário. O usuário conta com suas palavras como ele enxerga que o software deve se comportar ou como uma determinada atividade é feita. Essas histórias de usuário não viram requisitos formais, são convertidos diretamente em testes e as histórias de usuário descartadas. Por essa razão, o XP é considerado um método baseado em testes (*test-first*), ou seja, o código é escrito de forma que represente com sucesso o teste para o qual foi escrito.

Os programadores trabalham em pares, onde um escreve o código e outro faz inspeção em "tempo real" para garantir a qualidade do código. Esses pares sempre são trocados, bem como a pessoa que está escrevendo o código e sempre estão trabalhando em funcionalidades diferentes. Isso é feito para garantir a propriedade coletiva do código de forma que todos saibam tudo sobre o software. Caso o projeto perca um profissional essa perda é minimizada.

As práticas do XP, desenvolvidas por Beck (BECK, 2000), são as seguintes:

- Planejamento incremental – as histórias de usuário que serão transformadas em testes e posteriormente em tarefas são priorizadas pelo tempo disponível e importância;
- Pequenas entregas – o software deve ser útil desde a primeira entrega e vai crescendo a partir das funcionalidades mais importantes, de forma que elas são testadas e validadas desde o início;
- Projeto simples – segue-se a teoria YAGNI, de fazer somente o necessário e nada mais;
- Desenvolvimento *test-first* – o teste é feito antes de o código ser implementado;
- Refatoração – o código deve ser constantemente aprimorado para que a estrutura não se corrompa. E qualquer integrante da equipe pode fazer a alteração que julgar necessária para a melhoria do código;
- Propriedade coletiva – o código é de todos, e todos devem participar da criação e refatoração de qualquer parte do código;

- Integração contínua – quando uma nova funcionalidade é concluída deve ser integrada e os testes da nova funcionalidade devem ser executados, bem como os testes regressivos (repetição dos testes já realizados);
- Ritmo sustentável – não se deve fazer horas extras com o intuito de acelerar o projeto, a produtividade tende a cair;
- Participação do cliente – o cliente deve estar disponível para apoiar a equipe e esclarecer dúvidas;

O XP traz alguns conceitos próprios, mas boa parte de suas práticas são comuns ao desenvolvimento ágil.

3.4.2 Scrum

O Scrum é um processo iterativo e incremental para gerenciamento de projetos e desenvolvimento ágil de software. É focado no gerenciamento e, por isso, geralmente é implementado com apoio de um processo técnico, por exemplo, o XP.

No Scrum a equipe de desenvolvimento consiste de cinco a oito profissionais, que devem possuir conhecimentos para abranger todas as necessidades do desenvolvimento; da arquitetura à qualidade do software, ou seja, são equipes multidisciplinares e altamente especializadas. No caso de sistemas embarcados, esse conhecimento deve ser expandido e a equipe deve possuir conhecimentos em eletrônica, software, mecânica e design.

Na equipe existem duas figuras importantes: o *scrum master* e o *product owner.* A responsabilidade do *scrum master* é ajudar todos os integrantes da equipe a entender e aplicar o Scrum de forma eficiente. Ele deve remover qualquer obstáculo identificado pela equipe. O *product owner* representa os interesses do negócio e do cliente.

A equipe trabalha em ciclos, chamados *sprints* onde um número de itens são realizados e testados. O software ganha funcionalidades e utilidade em semanas. Um item na terminologia do Scrum é uma unidade de trabalho de tamanho suficiente para ser realizado em um único *sprint*. Esses são decompostos em uma ou mais tarefas menores.

Uma reunião de planejamento ocorre no início do projeto onde o *product owner*, juntamente com a equipe, transforma as funcionalidades do software em itens que podem ser realizados durante uma série de *sprints*.

Antes de cada *sprint*, a equipe seleciona os itens de maior prioridade ou os de maior risco e trabalham juntos para realizá-los.

Todos os dias, durante um *sprint* ocorre uma reunião de no máximo 15 minutos (geralmente de pé) onde todos informam os progressos e obstáculos encontrados no dia anterior e os planos para o dia atual.

No final de um *sprint* a equipe demonstra a nova funcionalidade para o *product owner* e demais interessados. Eles discutem como as lições aprendidas no *sprint* podem ser incorporadas nos próximos *sprints* para melhorar a produtividade. São analisadas também métricas de desempenho, estabelecidas antes de cada *sprint*, com o objetivo de se medir o desempenho real em relação ao previsto.

Os *sprints* ocorrem até que o projeto finalize ou o *product owner* decida que as funcionalidades são suficientes para uma primeira entrega.

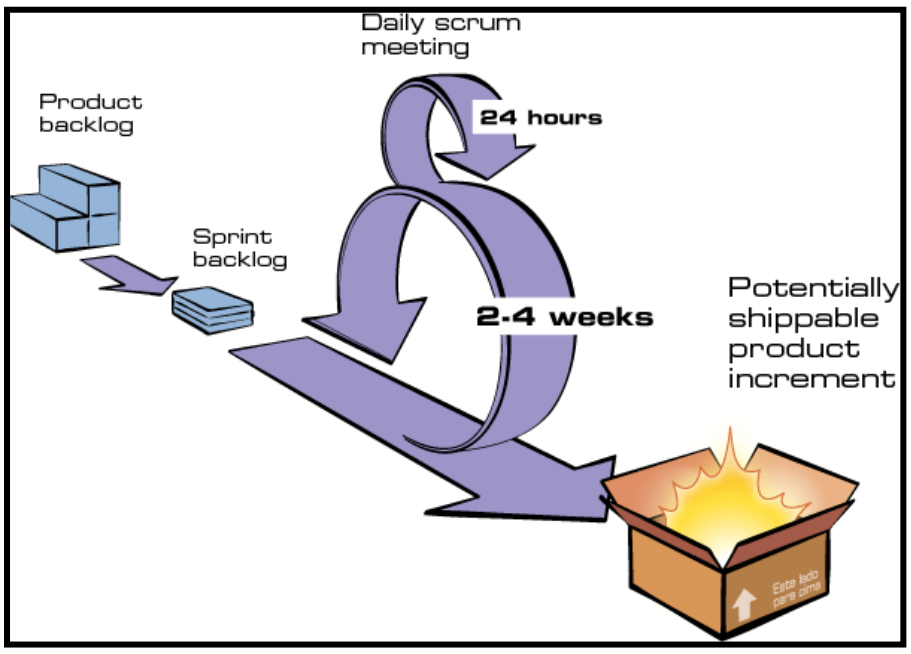

FIG. 4 Ciclo Scrum. Fonte: Autor desconhecido.

3.5 Engenharia de software e sistemas embarcados

Não existe na literatura muitas referências de aplicação da engenharia de software em sistemas embarcados. A abordagem do assunto, geralmente, tem foco no software para PC. Um trabalho que demonstra a aplicabilidade da engenharia de software ao sistema embarcado com foco em metodologias ágeis foi conduzido por (DAHLBY, 2004). A parte mais relevante desse trabalho é a verificação da aderência das práticas ágeis ao desenvolvimento de sistemas embarcados. O autor possui grande experiência em desenvolvimento desse tipo de sistema e faz uma abordagem interessante da aplicabilidade das práticas ágeis, explicando com detalhes os benefícios ou não de seu uso. São três os principais benefícios esperados para o projeto quando se aplica uma metodologia ágil: flexibilidade, produtividade e qualidade. O autor detalha de forma muito apropriada como esses objetivos podem ser atingidos com as práticas ágeis. Ainda segundo o autor, de maneira geral as práticas ágeis trazem melhoria ao projeto, principalmente as focadas em refatoração e testes.

Outro trabalho sobre o assunto foi conduzido por (RICCOBENE, SCANDURRA, ROSTI e BOCCHIO, 2007). O foco desse trabalho é desenvolver uma metodologia de desenvolvimento dirigida por modelos, baseada no UML2.0, perfis UML e SystemC. O processo desenvolvido foi chamado de *Unified Process for Embedded Systems* (UPES) e o desenvolvimento é baseado em plataforma.

A abordagem é iterativa e utiliza o UML, com os perfis adequados para cada parte do sistema, para o desenvolvimento em conjunto de hardware e software. Alguns problemas observados nessa abordagem são os seguintes:

- Apesar de a UML ter a proposta de ser uma linguagem unificada, ela exige um conjunto de perfis específicos para cada aplicação, o que expande sua abrangência, mas em contrapartida exige conhecimentos específicos que nem todos os desenvolvedores possuem.
- O modelo proposto usa o SystemC que é um subconjunto da linguagem C++ baseada em eventos. O uso de SystemC também é um fator limitador, pois nem todos conhecem a linguagem e pode não existir compilador para todas as arquiteturas.
- O desenvolvimento baseado em plataforma se concentra na existência de uma plataforma de hardware genérica, que possui vários periféricos e

dispositivos com a finalidade de atender às necessidades de um grande número de organizações. Essa abordagem é muito simplista e é utilizada apenas em sistemas onde o valor agregado é alto e os volumes são baixos, o que não justifica o desenvolvimento de hardware customizado. Outro empecilho é que a licença de uso do software e modelos provavelmente é fornecida para uso apenas com o hardware do fornecedor, impedindo a construção de um hardware customizado baseado na plataforma sem infringir direitos autorais;

4 BOAS PRÁTICAS NO DESENVOLVIMENTO DE SOFTWARE EMBARCADO

A seguir são apresentadas algumas boas práticas no que diz respeito à linguagem de programação utilizada em sistemas embarcados e a forma de estruturar o software, para se obter programas mais confiáveis e modulares.

4.1 Linguagem de programação

A grande maioria do software embarcado hoje é escrita em linguagem C. O motivo dessa escolha é que a linguagem C permite acesso direto e explícito a qualquer elemento ou endereço de memória (através de ponteiros), permitindo um controle total por parte do programador. Esses recursos são muito importantes para interação com o hardware que é feito através de registradores do microcontrolador. Os registradores são endereços de memória onde o microcontrolador busca as informações de como deve operar os periféricos, e é através deles que o software interage com o mundo externo (os sensores e atuadores são ligados ao microcontrolador através de circuitos específicos de condicionamento).

Além do acesso irrestrito que oferece aos programadores existem outras razões para a linguagem C ser utilizada até hoje:

- Todos os microcontroladores do mercado, de 8 a 32 bits, possuem compiladores C;
- Os compiladores geram código altamente eficiente, compacto e confiável, são maduros e estáveis;
- Domínio da linguagem pelos programadores;

É uma linguagem que não oferece todos os recursos linguísticos necessários para uma abordagem moderna de engenharia de software, que considera que a linguagem deve ser orientada a objetos (OO) para se aplicar os conceitos de abstração, composição, herança, encapsulamento e polimorfismo.

C++ oferece as mesmas vantagens de controle que o C e ainda possui todas as características de uma linguagem orientada a objetos, mas não é amplamente utilizada em sistemas embarcados.

As principais razões são:

- Nem todos os microcontroladores possuem compiladores C++;
- Os programadores não conhecem o paradigma de OO e preferem não arriscar um novo tipo de abordagem que possa colocar em risco o projeto;
- A OO se baseia na alocação dinâmica de objetos e a memória volátil pode se tornar altamente fragmentada. A rotina de alocação pode em algum momento não encontrar espaço para uma nova alocação do tamanho necessário. Essa característica afugenta parte dos desenvolvedores de software embarcado;
- C++ não possui o coletor de lixo automático e, portanto fica a cargo do programador liberar a memória alocada. Apesar de parecer óbvio é um dos principais defeitos encontrados no uso de C++. O defeito só é identificado muito tempo depois, geralmente já em campo, quando várias posições de memória alocadas não foram liberadas, até não existir mais memória disponível e o software se comportar de maneira inesperada;

A linguagem C++ se utilizada com cuidado pode trazer vários benefícios aos programadores com a abordagem orientada a objetos (BIGONHA, M., 2011), perdendo, contudo, a portabilidade para microcontroladores menores que não oferecem o suporte a C++.

A linguagem JAVA é considerada uma linguagem mais moderna e segura, por uma série de razões (BIGONHA, R., 2011), contudo não existem compiladores disponíveis no mercado para programação de software embarcado. Aparentemente pelo fato de ser uma linguagem interpretada, portanto, mais lenta, e necessitar de uma máquina virtual para funcionar. A pouca quantidade de memória e a necessidade de alto desempenho limita o uso de JAVA em sistemas embarcados. Certamente sofreria também todos os problemas enumerados para o uso de C++ com exceção do coletor de lixo que em JAVA é automático.

O barateamento dos microcontroladores e aumento progressivo da capacidade de memória pode viabilizar o uso de JAVA no futuro.

4.2 Boas práticas em C

A seguir apresenta-se uma lista de recomendações para o bom uso da linguagem C:

- Sempre inicialize as variáveis antes de utilizá-las;
- Trate todos os *warnings* do compilador, eles significam que o código, apesar de sintaticamente correto, pode gerar problemas;
- Use sempre pré e pós-verificação em todas as funções, para garantir a integridade dos dados de entrada e saída;
- Evite o uso de variáveis globais. Use uma nomenclatura específica para esse tipo de variável para facilitar sua identificação no programa (estilo de código);
- Use o modificador *volatile* em variáveis onde houver risco de acesso por interrupções ou mais de uma tarefa do sistema operacional. O objetivo é evitar que o compilador faça otimizações e consequente leitura de valores incorretos;
- Evite o uso de alocação dinâmica. A alocação dinâmica pode gerar fragmentação da memória e até risco de ausência de memória. Diretivas de programação segura para ambiente automotivo como o MISRA C (MISRA, 2004), proíbem qualquer uso desse tipo de recurso;
- Evite sempre que possível o uso do comando *goto;*
- Não use números no código, dê preferencia para macros e sempre comente a sua finalidade;
- Modularize o código separando as funcionalidades em vários arquivos de aplicação (*. c) e exporte somente o necessário através dos arquivos de interface (*. h);
- Não exporte variáveis (variáveis globais), garanta acesso a elas através de funções;
- Prefira passagem por valor a passagem por referência;

4.3 Programação modular

Dividir o software em módulos ajuda a reduzir a complexidade. Os módulos devem ser pequenos, conter operações relacionadas (alta coesão), baixa dependência (baixo acoplamento), esconder detalhes de implementação, não serem complexos e permitir composição para formar sistemas maiores.

Alguns princípios devem ser observados para se garantir a modularidade (BIGONHA, M., 2011):

- Unidade linguística – a linguagem deve oferecer todos os recursos linguísticos necessários para realizar as abstrações (somente as orientadas a objeto possuem essa característica);
- Baixa conectividade – o módulo deve ser o mais independente possível de forma que a propagação de alterações seja minimizada, os erros afetem poucos módulos e o grau de reusabilidade seja maior;
- Interface pequena – a comunicação entre módulos deve ser mínima;
- Interface explícita – a comunicação entre os módulos deve ser óbvia para se identificar com facilidade a propagação em caso de mudanças;
- Ocultação da informação – o módulo deve ser conhecido somente através de suas interfaces;

Em geral, programas modulares possuem menos chance de defeitos porque o risco de interação com fragmentos de código espalhados é menor. Eles tendem, portanto a serem mais fáceis de entender, manter e depurar.

A linguagem C não oferece abstração para criar tipos abstratos de dados ou mecanismos próprios de encapsulamento. Isso, contudo, não impede a aplicação dos conceitos de programação modular, apenas restringe alguns deles que não podem ser aplicados em sua totalidade.

4.4 Programação em camadas

O software deve ser escrito em camadas (de aplicação, de sistema operacional e de hardware).

A camada de hardware deve prover as interfaces necessárias para se trabalhar com o hardware, de forma a abstrair os detalhes de sua implementação. Dessa forma, mudanças no hardware ficam restritas a essa camada, aumentando a portabilidade.

Todos os eventos externos são tratados nessa camada. Assim, é possível, mesmo antes de se ter o hardware real (que geralmente demora muitos meses para ficar pronto), criar uma camada de hardware virtual para que as demais camadas possam ser desenvolvidas e testadas, de forma a realizar um desenvolvimento concorrente.

Isso permite também que os testes automatizados possam ser realizados nas demais camadas facilitando o trabalho da equipe de testes. O teste automatizado diretamente na arquitetura de destino pode ser difícil ou mesmo impossível.

5 PROCESSO DE DESENVOLVIMENTO DE SISTEMA EMBARCADO

Muitas empresas consideram que o desenvolvimento de sistema embarcado não necessita de um processo formalizado, pois muitas vezes o hardware é simples e o software é de pequeno porte. Mas as características especiais desse tipo de sistema, mesmo que aplicadas a sistemas não críticos (sistemas críticos lidam com a vida e segurança das pessoas) sugere que deve existir um processo de desenvolvimento formalizado (o que não significa pesado). Conforme Sommerville (2007), para sistemas de pequeno e médio porte (menos de 500 mil linhas), o método evolucionário de desenvolvimento de software é a escolha mais adequada e esse também é o caso do software embarcado. Uma característica importante extraída do modelo espiral proposto por Boehm (1998) e que deve ser aplicada ao desenvolvimento de sistemas embarcados é o gerenciamento de riscos. O objetivo do gerenciamento de riscos, no processo proposto, não é prever eventos que ainda não ocorreram (o que contraria a agilidade), e sim, utilizar o aprendizado adquirido em projetos anteriores para se obter maior assertividade no projeto atual e saber como trazê-lo para o caminho correto. Essa atividade dá o respaldo necessário para a integração com o gerenciamento de projetos de forma que existam planos de escape caso algo dê errado.

Os métodos ágeis são sem dúvidas um avanço na engenharia de software e possuem ideias inovadoras para agilizar o desenvolvimento de software. Contudo não existem resultados na literatura, com massa de dados suficiente, que indiquem que o desenvolvimento ágil, aplicado em sua totalidade, é uma abordagem totalmente eficiente para esse tipo de sistema. Existem vários relatos de utilização com sucesso, mas somente partes das práticas ágeis são utilizadas e outras, mais polêmicas, são pouco usadas, como por exemplo, a programação em pares (SALO e ABRAHAMSSON, 2007).

Apesar de o modelo evolucionário ser a principal referência, o modelo para sistemas embarcados deve possuir características do modelo tradicional e deve possuir características das metodologias ágeis, principalmente nas etapas de desenvolvimento e teste. No XP, por exemplo, o foco em testes é muito interessante e essa prática deve ser adotada para se realizar testes mais eficazes. A forma de selecionar e dividir os requisitos a serem realizados na próxima iteração podem ser baseados nas técnicas do Scrum.

No desenvolvimento de hardware o processo deve ser estritamente tradicional, seguindo um modelo cascata. Isso obriga que as funcionalidades de software que dependam diretamente de integração com o hardware sejam bem definidas desde o início do projeto, pois é caro e demorado modificar o hardware (alguns componentes por serem importados demoram até 30 semanas para serem fabricados e adquiridos). Com isso, apesar do modelo evolucionário aceitar que os requisitos evoluam com o desenvolvimento, isso é válido apenas para as funcionalidades que não dependam de alterações de hardware. Por exemplo, ao se definir que a comunicação com o mundo externo será feita via *universal serial bus* (USB), a camada de aplicação que será usada sobre a USB pode ser decidida em momento oportuno, de forma evolutiva, mas a mudança para a comunicação RS485 só é possível alterando o hardware. Isso demonstra que a especificação de requisitos nos sistemas embarcados deve ter partes baseadas no modelo tradicional e parte no modelo evolutivo, dependendo do tipo de funcionalidade.

O hardware, uma vez conhecido os requisitos, pode ser tratado como um projeto independente. Ele está amarrado ao software somente na fase de especificação e particionamento de requisitos e nos testes de sistema. O software ao contrário está amarrado ao hardware durante todo o desenvolvimento, pois mesmo que toda a dependência de hardware seja realizada como uma camada abstrata, algumas modificações de requisitos exigem alteração nessa camada. Se, por exemplo, um requisito especificar que deve existir um sinal de *pulse width modulation* (PWM) em um pino de saída do hardware que siga a fórmula $t=16000/k$, onde k é um parâmetro passado pela camada de aplicação para a camada de hardware e t o tempo, caso se decida mudar essa fórmula mudanças devem ser feitas nos registradores do microcontrolador para que seu *timer* gere o sinal de acordo com a nova regra.

Conforme Sommerville (2007) para sistemas críticos, o software deve seguir um processo tradicional de desenvolvimento. Existem inclusive algumas abordagens consideradas pesadas e altamente confiáveis para o desenvolvimento desse tipo de sistema, como os padrões MIL-STD-498 (Padrão de desenvolvimento de software para sistemas militares americano) e o NASA-STD-2100-91 (Padrão de desenvolvimento de software da NASA), que podem ser usados como referência para o desenvolvimento de sistemas críticos. Como esses sistemas possuem ainda mais restrições e boa parte dos sistemas embarcados não são conceitualmente críticos, a

abordagem de tratamento neste trabalho é mais geral para atender a um maior número de projetos.

Nos tópicos seguintes serão apresentadas as atividades do processo e a documentação que deve ser gerada em cada atividade. As atividades e documentos propostos são uma sugestão e cada organização tem a liberdade de desenvolver sua própria metodologia conforme suas necessidades.

5.1 Plano de desenvolvimento de sistema

O plano de desenvolvimento é um documento que descreve os passos que devem ser seguidos para se desenvolver o sistema. Ele descreve as ações a serem tomadas e os documentos que devem ser criados.

A boa engenharia é aquela que usa uma abordagem sistemática para o desenvolvimento e essa mesma visão deve ser aplicada ao desenvolvimento de software e sistemas.

O plano de desenvolvimento, quando não existe, deve ser escrito e todos os desenvolvedores e gerentes da organização devem participar da elaboração. O resultado deve ser um plano de desenvolvimento que é consenso entre os interessados para não se tornar um documento não utilizado. Todos devem enxergar o plano como uma forma de sistematizar o desenvolvimento e obter melhores produtos.

O plano deve conter os seguintes elementos mínimos:

- Os passos a serem seguidos para desenvolver o produto – o modelo de processo utilizado;
- Quais documentos que devem ser criados – inclusive o formato padrão (*templates*) que deve ser utilizado para cada um deles e quais devem ser guardados de forma eletrônica ou impressa (caso houver). Nesse quesito entram também a documentação que não necessariamente é preenchida. Os artefatos resultantes das atividades como: esquemas elétricos, desenho de placas de circuito impresso, códigos fonte do software, etc.;
- Como o gerenciamento de riscos será tratado – em quais etapas serão feitas essas avaliações para verificar se o projeto está no caminho;

- Medição de sucesso – é necessário estabelecer critérios de sucesso mensuráveis;

5.2 Modelo de processo

O modelo de processo para o desenvolvimento de sistemas embarcados, proposto é apresentado na Figura 5. Ele é uma mistura do processo evolucionário com a metodologia ágil XP. Nos tópicos seguintes serão tratados os detalhes de cada atividade do processo e as práticas da metodologia XP que são consideradas importantes de serem aplicadas ao processo.

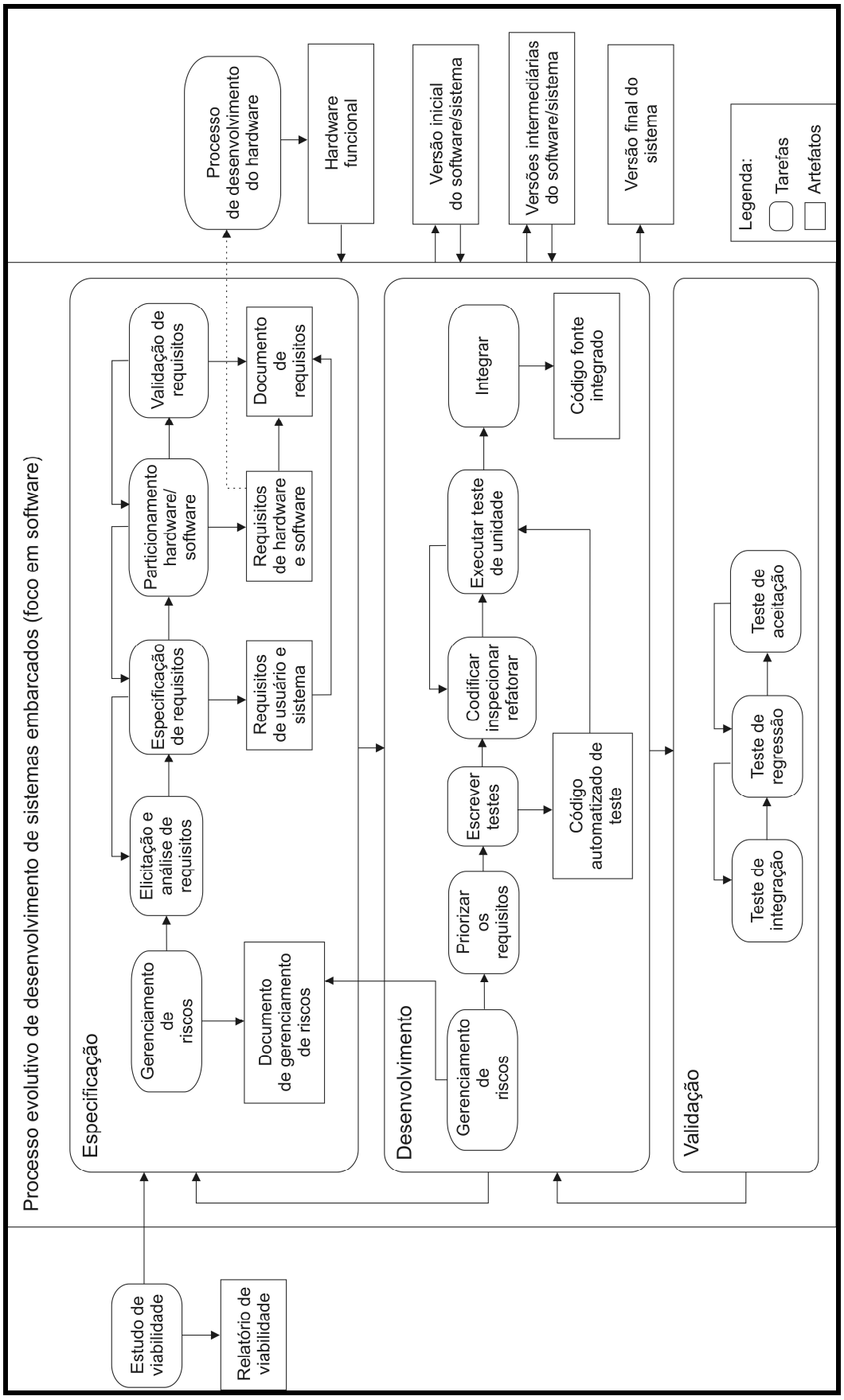

FIG. 5 Modelo de processo evolutivo de desenvolvimento de sistemas embarcados, com foco em desenvolvimento de software.

É importante observar que o desenvolvimento do hardware, uma vez que seus requisitos são conhecidos, é um projeto independente que segue uma metodologia tradicional. Finalizado o projeto de hardware, que pode demorar meses depois do projeto do sistema ter iniciado, ele entra no processo como ferramenta para testes de sistema. O ideal é que o hardware esteja disponível já no início, mas como isso pode não ser possível, é necessário desenvolver parte do software no PC e, tão logo seja possível, iniciar o ciclo de desenvolvimento diretamente no hardware. Contudo, alguns testes automatizados podem ser mais fáceis de realizar no próprio PC, usando a camada virtual de hardware criada no início do desenvolvimento do software.

5.3 Estudo de viabilidade

O estudo de viabilidade tem a finalidade de identificar se o projeto proposto seja por clientes externos ou internos é viável. O relatório de viabilidade é o artefato de saída dessa fase do processo e deve conter os seguintes itens:

- Viabilidade temporal – um cronograma preliminar baseado nas funcionalidades que o sistema deve oferecer é feito para verificar se o tempo disponível para execução do projeto, dentro da janela de oportunidade, é suficiente considerando uma margem adequada de desvio (baseado no histórico da empresa em outros projetos);
- Viabilidade técnica – deve-se verificar se na organização existem profissionais especializados capazes de desenvolver as funcionalidades e caso não exista, se existem empresas ou profissionais disponíveis no mercado para terceirizar as atividades, ou até mesmo, desenvolver as competências;
- Viabilidade econômica – deve-se verificar se o resultado que a venda do produto irá gerar é adequado para suprir os lucros esperados e os investimentos necessários no desenvolvimento do projeto com ferramentas, pessoal, custos fixos, etc.;

O relatório de viabilidade deve ser claro ao recomendar ou não o desenvolvimento do projeto, esclarecendo as razões para essa decisão.

5.4 Especificação de requisitos

Nos sistemas embarcados o software tem uma ligação íntima com o hardware onde irá operar. A partir da especificação completa do sistema é que será possível definir o projeto de hardware e software. Algumas funcionalidades podem ser realizadas em hardware ou software e essa decisão afeta diretamente o custo e o desempenho.

Suponha um requisito que exige a assinatura de um arquivo digitalmente utilizando o algoritmo RSA1024. Existem microcontroladores que possuem o periférico de criptografia que implementam o algoritmo especificado via hardware e entregam o resultado sem necessidade de interferência da *central processing unit* (CPU), apenas sinalizando quando o resultado da operação está concluído, liberando-a para outras atividades. Em hardware, a operação além de não ocupar a CPU, ocorre em um tempo muito curto, tipicamente em menos de 1ms no microcontrolador MAXQ1103 da Maxim IC rodando a 25MHz. O mesmo algoritmo sendo realizado por software em um microcontrolador ARM7 LPC2366 da NXP rodando a 72MHz gasta cerca de 3 segundos e a CPU ficará totalmente dedicada fazendo os cálculos durante esse tempo. Obviamente a solução por hardware é mais interessante, mas outros fatores devem ser levados em consideração nesse caso: a diferença de custo entre os microcontroladores (cerca de $6,00 FOB em cada unidade), se o resultado da operação tem características de tempo real ou se pode ser entregue quando a CPU estiver ociosa, entre outras.

Essa divisão de funcionalidades entre hardware e software é conhecida como particionamento. Maiores detalhes das técnicas de particionamento podem ser encontradas em (GAJSKI, VAHID, NARAYAN e GONJ, 1994).

O artefato de saída dessa etapa é o documento de requisitos. Ele deve descrever tudo que o sistema deve fazer (e não fazer), o que será implementado em hardware ou software, e qualquer restrição que deve atender.

Sem um documento de requisitos bem escrito é muito fácil omitir ou esquecer características importantes, ficar confuso com que o sistema deve fazer ou criar um sistema que faz a coisa errada. Os requisitos devem ser consistentes, precisos, mensuráveis e rastreáveis. Erros decorrentes de requisitos incorretos ou mal compreendidos são os mais caros e difíceis de corrigir (SOMMERVILLE, 2007).

Os requisitos que devem ser atendidos pelo hardware devem ser totalmente conhecidos e compreendidos no início do desenvolvimento, os requisitos de software podem seguir uma característica evolutiva.

5.4.1 Tipos de requisitos

Existem três tipos de requisitos:

- Funcionais – são funções, comportamentos ou características que devem ser providas pelo sistema. Podem ser realizados por hardware ou software. Exemplo: caso o botão de liga / desliga seja pressionado por 1 segundo ou mais, o sistema deve ir para modo de espera; por mais de 5 segundos desligar;
- Não funcionais – são atributos do sistema como um todo. Desempenho, segurança, usabilidade, etc.. Exemplo: O tempo de reposta a um evento de teclado deve ser menor que 100ms;
- Restrições – são limitações impostas ao sistema. Exemplo: deve atender a norma ISO16844;

5.4.2 Características fundamentais de bons requisitos

Geralmente os bons requisitos devem atender a todas as características abaixo:

- Mensuráveis – deve-se possuir formas de medir se o requisito foi atendido, de preferência com valores numéricos. Exemplo: em caso de colisão o *airbag* deve acionar em 10ms;
- Tolerância – baseado na característica anterior os valores não devem ser simples, mas uma faixa de valores aceitáveis. Exemplo: em caso de colisão o *airbag* deve acionar em 10 +/- 1ms;
- Consistentes – os requisitos devem ser escritos de forma uniforme, não ambígua e a nomenclatura deve ser a mesma durante todo o documento. Exemplo: se um requisito vai ser implementado com o periférico "*timer 1"* do microcontrolador ele não deve depois ser chamado de temporizador 1;

- Uso das palavras “pode” e “deve” – deve-se separar as características mandatórias e desejáveis. Dessa forma caso uma funcionalidade seja apenas desejada, ela pode ser descartada ou postergada caso seja complexa;
- Rastreabilidade – todos os requisitos devem ter um identificador único para que seja referenciado, por exemplo, na criação da matriz de relacionamento com os testes.

5.4.3 Requisitos em sistemas embarcados

Conforme indicado pelo modelo da Figura 5, a etapa de especificação deve seguir passos mais formais do que praticado nas metodologias ágeis. Aqui o documento de requisitos deve ser completo e atualizado, indicando o que o sistema deve fazer e as mudanças ocorridas (o que se traduz em tempo e custo não previsto). A formalização mais rígida nessa etapa não significa que a prática do “*embrace the change”* da metologia ágil não se aplica a esse modelo, significa apenas que essa é uma etapa suficientemente importante para que tudo seja documentado a rigor e que se existirem mudanças elas devem ser analisadas e validadas.

As atividades na fase de especificação são as seguintes:

- Elicitação e análise de requisitos – levantamento das funcionalidades do sistema;
- Especificação de requisitos – tradução das funcionalidades levantadas na etapa de elicitação e análise, em requisitos de usuário e de sistema;
- Particionamento hardware / software – atividade de decidir quais funcionalidades serão implementadas em hardware e quais serão implementadas em software. Durante o processo iterativo de especificação se mudanças ou novas funcionalidades, que só possam ser implementadas em hardware, surgirem, devem ser desencorajadas pelo alto custo e tempo envolvidos. As implementadas em software certamente irão mudar ao longo do processo de desenvolvimento evolucionário e o desenvolvimento deve seguir as práticas das metodologias ágeis para se adaptar com rapidez a essas mudanças;

- Validação de requisitos – atividade para se verificar se os requisitos podem ser atendidos e se estão coerentes com a realidade. O objetivo é certificar-se de que os requisitos que serão implementados estão corretos e possuem as boas características mencionadas no subitem 5.4.2.

A saída da etapa de especificação é o documento de requisitos completo, com requisitos de usuário, de sistema, de hardware, de software e o histórico das mudanças e novos requisitos solicitados. Deve conter também um *checklist* do que já foi implementado e para cada teste de aceitação feito pelo cliente um documento de aceitação deve ser assinado.

Um *template* interessante que pode ser usado como referência para a especificação de requisitos em sistemas embarcados é o proposto por (MARTINS, SOUZA, OLIVEIRA e PEIXOTO, 2010) chamado *Template* para Especificação de Requisitos de Ambiente em Sistemas Embarcados (TERASE).

O trabalho é uma referência importante, uma vez que o *template* nele definido é específico para a obtenção de requisitos não funcionais em sistemas embarcados, tratando a relação do software embarcado com o hardware onde irá operar. Definição dos pinos, periféricos e registradores, bem como seus nomes e relações são especificados em cartões de requisitos. O trabalho apresenta dados práticos de utilização e medições de satisfação de seus usuários, permitindo assim a definição de sua praticidade e abrangência.

Como sua proposta é limitada aos requisitos não funcionais, para ser abrangente deve ser aplicado juntamente com outro *template* para levantamento de requisitos funcionais e restrições ou expandido para suportá-los.

5.5 Desenvolvimento

No desenvolvimento a agilidade deve predominar e conforme a Figura 5 as práticas predominantes nessa etapa são baseadas no XP. Não se encoraja aqui a criação de documentos formais de arquitetura, desenho e documentação de código. Esses documentos atrasam o desenvolvimento e não agregam valor. No XP o foco é o código. Ele deve ser simples e constantemente aprimorado, de forma que as documentações

adicionais não sejam necessárias. Abaixo estão relacionadas as práticas do XP que são consideradas importantes e as adequações necessárias para que sejam realistas.

- Planejamento incremental – apesar de o XP considerar que o cliente (que pode ser interno) deve ser parte da equipe de desenvolvimento, a realidade não permite esse envolvimento "extremo". Uma abordagem mais realista, extraída do Scrum é que ele participe, com uma frequência razoável (a cada uma ou duas semanas), do planejamento das funcionalidades que deverão fazer parte da próxima versão, de forma que as funcionalidades mais importantes sejam testadas primeiro;
- Pequenas versões – uma vez que uma funcionalidade esteja pronta deve ser integrada imediatamente para ser testada;
- Metáfora – conversar com o cliente usando linguagem natural, a complexidade deve ficar restrita ao desenvolvedor. Isso exige do desenvolvedor boas habilidades de comunicação e conhecimento no domínio da aplicação;
- Projeto simples – o projeto deve atender somente o necessário no momento (principio YAGNI) e nada mais. Nada de prever o que poderá ser necessário no futuro e que certamente será diferente do previsto, gerando perda de tempo;
- Escrita de teste antes de código – esse é o ponto mais importante do XP adotado aqui. Isso garante que nenhuma funcionalidade deixará de ser testada e que o código implementa corretamente o requisito. Observe que isso não garante que o requisito está correto, essa verificação deve ser feita na atividade de validação de requisitos na fase de especificação.
- Refatoração – o código deve ser constantemente reescrito para se manter a simplicidade de forma que a estrutura não se corrompa com o tempo;
- Propriedade coletiva – todos os desenvolvedores devem trabalhar em todas as partes do código. Assim todos conhecem o código e não se formam ilhas de conhecimento e a perda de um profissional não gera grande impacto;
- Especialização – no XP os profissionais devem ser capacitados para serem capazes de desenvolver qualquer parte do código. Na prática isso nem sempre é possível, devido a dificuldade de encontrar esse tipo de profissional. As habilidades que os integrantes de uma equipe XP devem possuir, torna a formação da equipe uma tarefa complicada e apesar da metodologia tratar

explicitamente a perda de profissionais com a propriedade coletiva de código é melhor não perdê-los, pois a reposição não será fácil;

- Ritmo sustentável – não fazer horas extras, o rendimento dos profissionais cai consideravelmente quando estão trabalhando pensando nas horas de lazer perdidas;

O desenvolvimento em pares é sem duvida a abordagem mais polêmica do XP. Realmente é difícil convencer a alta gerência de uma empresa que são necessários dois profissionais por computador, onde apenas um está gerando artefato útil.

5.5.1 Priorização dos requisitos

A priorização de requisitos é feita pelo cliente, em períodos de uma ou duas semanas, baseado nas funcionalidades que ele julgar mais importantes. Caso o cliente não esteja disponível para priorizar, os desenvolvedores devem priorizar os de maior risco (mais complexos ou com pouco conhecimento sobre o domínio). O objetivo de priorizar os requisitos de maior risco é garantir que exista tempo suficiente para tratá-los sem prejudicar os prazos do projeto.

5.5.2 Escrever testes

Os testes são escritos antes de qualquer implementação de código, essa abordagem é baseada no XP. O objetivo é escrever testes que garantam a maior abrangência possível: passem por todos os condicionais, apliquem valores de fronteira, gerem comportamentos excepcionais, estouro de *buffers*, estouro de variáveis. A escrita de testes completos faz a diferença para o sucesso do sistema e deve ser tratado como a etapa mais importante do desenvolvimento.

A saída dessa atividade é o código automatizado de teste para implementar os testes unitários da funcionalidade sendo desenvolvida.

5.5.3 Codificação

O XP se concentra no código como a principal forma de documentação e encoraja que ele seja simples e constantemente refatorado para que sua estrutura não se degrade frente a mudanças que são inevitáveis. O código, no XP, tem a ideia de propriedade coletiva. Essa característica torna essencial o uso de um estilo de código.

O estilo de código se refere ao formato geral e a abordagem estilística do código. O estilo de código deve estar documentado no plano de desenvolvimento da empresa.

Um estilo de código consistente torna a leitura de programa escrito por outro programador mais fácil. Quem está lendo pode se preocupar apenas com o que o código faz.

O documento de estilo de código deve conter os seguintes elementos:

- Os elementos requeridos no código fonte, na forma de *template;*
- Regras para comentar o código;
- Regras de uso da linguagem;
- Convenção de como nomear as variáveis e funções;

A manutenção da simplicidade através da refatoração é uma atividade implícita dentro da atividade de codificação e deve ser feita sempre que uma possibilidade de melhoria for visualizada através de inspeções constantes do código. O XP considera que a inspeção seja feita em *tempo real* através do uso da programação em pares, mas como visto anteriormente é uma prática difícil de ser aplicada. Dessa forma a inspeção deve ser feita a medida que os desenvolvedores trocam de funcionalidade com intuito de manter a propriedade coletiva do código.

5.5.4 Teste de unidade

O código escrito deve ser testado de forma automática a partir do código de teste escrito para ele. Aconselha-se que o teste de unidade seja escrito por outra pessoa, de forma que o teste não seja escrito com o objetivo de facilitar a implementação (e possivelmente seja incompleto). O teste deve ser realizado pelo desenvolvedor, de forma que caso algo dê errado o código é imediatamente corrigido até ser aprovado no

teste, impedindo que sejam implementadas novas funcionalidades antes que a atual seja aprovada.

Caso os testes não sejam abrangentes, o código escrito também não será, pois ele é escrito para atender aos testes.

5.5.5 Integração

O código ao passar no teste de unidade deve ser imediatamente integrado. O resultado é o código fonte integrado, que passará para a fase de validação.

5.6 Validação

A fase de validação tem o objetivo de certificar que o software atende a especificação e entrega o que o cliente espera.

Nos processos tradicionais essa fase contém a tarefa de inspeção de código, que é considerada a forma mais eficiente de descoberta de erros (SOMMERVILLE, 2007). A inspeção no processo proposto é tratada como parte da atividade de codificação e feita por todos os desenvolvedores.

O processo proposto na Figura 5 traz explicitados três tipos de testes que devem ser executados para validar o sistema.

5.6.1 Teste de integração

O teste de integração é feito sobre o software integrado com o objetivo de se verificar se a nova funcionalidade funciona quando integrada com as demais já existentes, ou seja, verifica se nenhuma propriedade emergente prejudicou o software, o que pode significar que a estrutura foi corrompida e o software não está coeso e desacoplado como deveria.

Caso algum problema seja identificado um processo de inspeção e depuração deve ser realizado para se identificar o ponto de falha.

5.6.2 Teste de regressão

O teste de regressão é repetição de todos os testes executados anteriormente durante o desenvolvimento para verificar se a nova funcionalidade não prejudicou o funcionamento das existentes, ou seja, no teste de integração é verificado se a nova funcionalidade é prejudicada pela adição em conjunto com as antigas, aqui é verificado se a nova funcionalidade prejudica as já existentes.

O objetivo é garantir que todo o sistema funciona em conjunto. Esse tipo de teste deve ser automatizado, pois são em geral complexos e demorados.

5.6.3 Teste de aceitação

É o teste no qual o cliente valida o sistema com a nova funcionalidade integrada. Nos testes anteriores os desenvolvedores verificam se o sistema atende aos requisitos da forma que eles interpretaram como o sistema deve se comportar. No teste de aceitação o cliente informa se está funcionando da forma esperada por ele. Por essa razão a especificação deve ser a mais fiel possível de forma a evitar retrabalho.

A importância de um documento de requisitos é mais evidente nessa atividade, pois caso o cliente afirme que a funcionalidade é diferente da solicitada é fácil demonstrar que ele está errado e cobrar ou renegociar prazos da alteração.

5.7 Versões de software / sistema

As versões de software / sistema são construídas como incrementos de funcionalidade, até que o sistema final esteja completo. O sistema é considerado completo quando todas as funcionalidades presentes no documento de requisitos foram realizadas em hardware e software e validadas pelo cliente. No plano de desenvolvimento deve existir um documento de aceitação impresso e assinado para cada funcionalidade incremental atendida. Deve existir também um documento de saída de projeto, onde o cliente confirma que tudo está de acordo com o especificado e o projeto está entregue.

A partir desse momento o sistema entra na etapa de uso e manutenção. Os defeitos de projeto devem ser corrigidos (por um tempo acordado no contrato) e as

novas funcionalidades que surgirem por necessidade de mercado devem ser negociadas como um novo projeto de melhorias.

6 CONCLUSÃO

O modelo de processo proposto nesse trabalho é fruto da experiência de desenvolvimento de sistemas embarcados e o conhecimento teórico adquirido no curso de especialização de informática com ênfase em engenharia de software.

Acredito que seja um modelo realista que pode ser implementado em empresas com equipes de desenvolvimento de pequeno a médio porte.

O modelo foi baseado principalmente nas práticas das metodologias ágeis (XP), pois cada vez mais o mercado está pressionando por soluções diferenciadas com janelas de oportunidades cada vez menores. Uma abordagem tradicional certamente não traria os resultados esperados em tempo hábil, devido a série de artefatos formais que devem ser gerados antes de qualquer geração de software útil.

Essa proposta não foi testada em um projeto real e serve como um ponto de partida de como começar a usar a engenharia de software em projetos de sistemas embarcados, com o mínimo de documentos gerados. Ao invés de documentos o foco é nas práticas a serem seguidas.

Grande parte das empresas não usa nenhum processo descrito em um plano de desenvolvimento e os engenheiros de hardware, que geralmente projetam esse tipo de sistema, não têm conhecimento sobre a teoria da engenharia de software. Certamente para essas empresas o processo sugerido pode trazer melhorias significativas. Com o uso e maior contato com a engenharia de software, essas empresas podem adequar o processo a sua realidade, de forma a criar um processo próprio de desenvolvimento.

Uma proposta de continuidade desse trabalho é aplicar o processo ao desenvolvimento de um sistema real e medir os resultados obtidos através de métricas a serem definidas. Medir os níveis de satisfação dos desenvolvedores com o processo, com os artefatos, com os *templates*, enfim, verificar se o processo trouxe benefícios para o desenvolvimento e como melhorar o processo. Partindo do pressuposto que cada empresa tem uma realidade específica, ele deve ser aplicado a um conjunto de empresas e desenvolvedores. Os resultados devem ser usados para tornar o modelo genérico suficiente para atender o maior número de empresas.

Obviamente essa tarefa não é simples. As empresas certamente teriam relutância de mudar seus costumes e permitir um teste com uma nova abordagem (que traz risco ao negócio) bem como divulgar os resultados em um trabalho acadêmico. Mas certamente seria muito proveitoso para a engenharia de software que é uma disciplina

recente, principalmente em relação a abordagem ágil. Essa abordagem focada em sistemas embarcados é ainda mais restrita e está concentrada em apenas algumas empresas de desenvolvimento pelo mundo.

REFERÊNCIAS

AGILE Manifesto. C2001. Disponível em < http://agilemanifesto.org/>. Acesso em: 8 abr. 2012.

BECK, K. *Extreme programming explained:* Embrace Change. Addison-Wesley, 2000. 190 p.

BIGONHA, Mariza da S. *Programação modular*: Notas de aula. Belo Horizonte: UFMG, 2011.

BIGONHA, Roberto da S. *Ambientes de programação:* Notas de aula. Belo Horizonte: UFMG, 2011.

BOEHM, B. A spiral model of software development and enhancement. *IEEE Computer.* v. 21, n. 5, p. 61-72, maio 1988.

BROY, Manfred. *Challenges in Automotive Software Engineering.* 2006. 10 f. Paper – Institut fur Informatik – Technische Universitat Munchen, Munchen – Germany, 2006.

DAHLBY, Doug. *Applying Agile Methods to Embedded Systems Development.* 2004. 23 f. Paper – ArrayComm Inc – Chicago, 2004.

DEMARCO, T.; BOEHM, B. The agile methods fray. *IEEE Computer Science*. v. 35, n. 6, p. 90-91, ago. 2002.

GAJSKI, D. D.; VAHID F.; NARAYAN, S.; GONG, J. *Specification and design of embedded systems*. 1. Ed. Prentice Hall, 1994. 468 p.

KOOPMAN, Philip. *Better Embedded System Software.* Drumnadrochit Education, 2010. 369 p.

MARTINS, Luiz E. G.; SOUZA Jr., Roberto; OLIVEIRA Jr., Hermano P.; PEIXOTO, Cecília S. A. *Terase: Template para Especificação de requisitos de Ambiente em Sistemas Embarcados.* 2010. 12 f. Paper – Faculdade de Ciências Exatas e da Natureza – Universidade Metodista de Piracicaba (UNIMEP) – Piracicaba, 2010.

MARWEDEL, Peter. *Embedded System Design*. 1. Ed. Dortmund: Kluwer Academic Publishers, 2003. 237 p.

MISRA. *MISRA-C:2004:* Guidelines for the use of the C language in critical systems. Nuneaton, 2004. 111 p.

MORALES, M.; RAU, S.; PALMA, M.J.; VENKATESAN, M.; PULSKAMP, F.; DUGAR, A.. *Worldwide Intelligent Systems 2011-2015 Forecast: The Next Big Opportunity*, International Data Corporation – IDC, Setembro de 2011.

NOERGAARD, Tammy. *Embedded System Arquiteture:* A comprehensive guide for engineers and programmers. Burlington: Elsevier, 2005. 640 p.

PRESSMAN, Roger S. *Engenharia de software.* 3. Ed. São Paulo: Makron Books, 1995. 530 p.

RICCOBENE, Elvinia; SCANDURRA, Patrizia; ROSTI, Alberto; BOCCHIO, Sara. *Designing a Unified Process for Embedded Systems.* 2007. 12 f. Paper – Dip. Di Tecnologie dell'Informazione – Universita di Milano Chicago, AST Lab R&I C.d.Colleoni Agrate Brianza – STMicroelectronics – Italy, 2007.

ROYCE, W. W. Managing the development of large software systems: concepts and techniques. In: IEEE Westcon, 26, 1970, Los Angeles. *Proceedings IEEE Westcon*. IEEE, 1970, p. 1-9.

SALO, O.; ABRAHAMSSON, P. Agile methods in European embedded software development organizations: a survey on the actual use and usefulness of Extreme Programming and Scrum. *IET Software.* Finlândia. v. 2, n. 1, p. 58-64, set. 2007.

SCHWARTZ, J. I. *Construction of software*. In: Practical Strategies for Developing Large Systems. Menlo Park: Addison-Wesley, 1. ed. 1975.

SIMON, David E. *An Embedded Software Primer*. 12. Ed. Delhi: Pearson Education, 2005. 225 p.

YIMEI, Kang. *A graduate program on embedded software engineering in China.* 2007. 8 f. Paper – Software College – BeiHang University – Beijing, 2007.