PARTE 3 – PRINCÍPIOS DE ENGENHARIA DE SOFTWARE

Nesta seção apresentamos os princípios básicos da engenharia de *software*. Estes princípios têm por objetivo o sucesso no desenvolvimento de *software* e se preocupam tanto com o processo quanto com o produto final. O processo certo ajudará a produzir o produto certo, mas por outro lado o produto desejado também tem influência na escolha do tipo de processo a ser utilizado.

Os princípios apresentados são genéricos o suficiente para serem aplicados através do processo de construção e gerenciamento do *software*. Princípios, contudo, não são suficientes para conduzir o desenvolvimento de um produto. Na realidade eles são declarações gerais e abstratas que descrevem as propriedades desejadas dos processos de desenvolvimento e dos produtos de *software*. Porém, para aplicar princípios a engenharia de *software* deveria estar equipada com métodos apropriados e técnicas específicas para ajudar na incorporação das propriedades desejadas nos processos de desenvolvimento e nos produtos em geral.

Primeiramente nós deveríamos saber distinguir métodos e técnicas. Métodos são linhas gerais que regulam previamente uma série de operações que se devem realizar em vista de um resultado determinado. São rigorosos, sistemáticos e disciplinados. Técnicas são maneiras ou habilidades especiais de se executar ou fazer algo. Pode-se dizer que são as habilidades de execução de métodos.

A união destes dois pontos forma a metodologia, cujo propósito é promover uma determinada forma para resolver um problema através da pré-seleção dos métodos e técnicas a serem utilizados. Uma metodologia é ainda caracterizada pelo estudo dos métodos em busca de técnicas mais apropriadas e específicas à solução de determinados problemas.

As ferramentas por sua vez, são desenvolvidas como mecanismos de apoio à aplicação de métodos, técnicas e metodologias. A figura a seguir mostra o relacionamento destes aspectos. Cada nível na figura está baseado nos níveis inferiores e, quanto mais acima, mais suscetível a mudanças com o passar do tempo. A figura 3.1 demonstra também que os princípios são a base de qualquer método, técnica, metodologia e ferramenta.

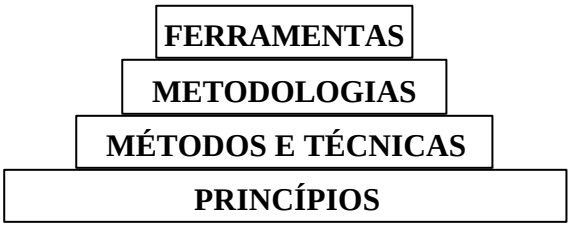

Figura 3.1. Relacionamento entre Princípios, Métodos, Técnicas, Metodologia e Ferramentas

Embora os princípios da engenharia de *software* geralmente demonstrem um forte relacionamento, eles serão descritos separadamente e em termos gerais.

Rigor e Formalismo

Como já dito anteriormente o desenvolvimento de *software* é uma atividade criativa. Em geral processos criativo não são precisos nem exatos e seguem a inspiração do momento de forma não estruturada. O rigor, por outro lado, é um complemento necessário à criatividade em todas as atividades de engenharia. É

através do rigor que podemos desenvolver produtos mais confiáveis, controlar custos e aumentar a confiança em relação ao produto.

Existem diversos níveis de rigor que podem ser alcançados. O mais alto grau de rigor que pode ser obtido é chamado de formalismo. Portanto, o formalismo é um requisito mais forte do que o rigor. Ele requer que os processos de desenvolvimento de *software* sejam conduzidos e avaliados por leis e princípios matemáticos.

Na engenharia, de modo geral, o processo de projeto procede como uma seqüência de passos bem-definidos e perfeitos e precisamente declarados. Em cada passo o engenheiro segue algum método ou aplica alguma técnica. Estes métodos e técnicas aplicados podem ser baseados em:

- combinações de resultados teóricos derivados de alguma modelagem formal da realidade;
- adaptações empíricas[1] que tratam dos fenômenos não tratados pelo modelo;
- modos práticos que dependem de experiências passadas;

A combinação destes fatores resulta em uma abordagem rigorosa e sistemática (metodologia) que pode ser facilmente explicada e aplicada diversas vezes. Não há necessidade, porém, de sermos sempre formais durante um projeto, contudo o engenheiro deve saber quando e como ser formal. O engenheiro deve estar apto para entender o nível de rigor e formalismo que deve ser obtido dependendo da criticalidade e dificuldade conceitual de uma determinada tarefa. Este nível pode também variar para diferentes partes de um mesmo sistema.

O mesmo acontece na engenharia de *software*. A descrição do que um programa faz pode ser dada de uma forma rigorosa através de uma linguagem natural ou ela pode ser feita formalmente através das descrições formais de uma linguagem de declarações lógicas.

Tradicionalmente existe apenas uma atividade no desenvolvimento de *software* que utiliza uma abordagem formal: a programação. Na realidade programas são objetos formais. Eles são escritos em linguagens de programação cuja sintaxe e semântica estão totalmente definidos. Os programas são também descrições formais que podem ser automaticamente manipuladas por compiladores que fazem uma checagem de corretude (a nível de sintaxe apenas), transformam o programa em um equivalente em outra linguagem (*assembly* ou linguagem de máquina), etc. Estas operações mecânicas, as quais são feitas possivelmente pela utilização de formalismo na programação, podem efetivamente aumentar a confiabilidade e a verificabilidade dos produtos de *software*.

O rigor e formalismo também beneficiam qualidades como manutenabilidade, reusabilidade, portabilidade, entendabilidade e interoperabilidade. Por exemplo, uma rigorosa ou mesmo formal documentação de *software* pode melhorar todas estas qualidades em relação a documentações informais, as quais são freqüentemente ambíguas, inconsistentes e incompletas.

O rigor e formalismo se aplicam também aos processos de desenvolvimento de *software*. A documentação rigorosa de um processo ajuda para que o mesmo possa ser reutilizado em projetos similares. Baseados nesta documentação os gerentes podem prever os passos através dos quais o novo projeto se envolverá, distribuindo e aplicando corretamente os recursos de forma apropriada e de acordo com a necessidade.

Da mesma forma uma documentação rigorosa de um processo de desenvolvimento pode auxiliar na manutenção de um produto existente. Se os vários passos do processo forem documentados, alguém poderá modificar um produto existente começando pelo nível apropriado e não diretamente pelo código final. E, finalmente, se o processo for especificado rigorosamente os gerentes podem monitorá-lo precisamente, com o objetivo de atingir o *timeliness* e aumentar a produtividade da equipe.

Separação de Preocupações (Divisão de Tarefas)

A separação de preocupações nos permite tratar individualmente de diferentes aspectos de um problema de forma a concentrar esforços separadamente. A divisão de tarefas é uma prática de senso comum que tentamos seguir em nosso dia-a-dia para diminuir as dificuldades encontradas. Este princípio deve ser aplicado também ao desenvolvimento de *software* de forma que possamos dominar a complexidade inerente dos problemas.

[1] Algo baseado em experiências práticas a na observação e que não possui base teórica.

Existem muitas decisões a serem tomadas durante o desenvolvimento de um produto. Algumas delas são atribuídas às características do produto tais como funcionalidades a serem oferecidas, confiabilidade esperada, eficiência a nível de velocidade e economia de recursos, relacionamento com o ambiente (*hardware* especial ou recursos específicos de um sistema), interface com o usuário, etc. Outras decisões são atribuídas ao processo de desenvolvimento como por exemplo ambiente de desenvolvimento, organização e estruturação da equipe, planejamento, controle dos procedimentos, estratégias de projeto, mecanismos para recuperação de erros, etc. Por fim, algumas outras decisões são atribuídas a fatores econômicos e financeiros.

Em geral algumas destas decisões (preocupações) podem ser resolvidas sem interferir em outras decisões. Neste caso é óbvio que elas estariam sendo tratadas separadamente. Porém, em alguns casos uma decisão implica na alteração de outras decisões ou definições. Por exemplo, inicialmente por questões de desempenho definiu-se que o produto teria como requisito de *hardware* um mínimo de 32 megabytes de memória RAM. Através de uma projeção de custos verificou-se posteriormente que este requisito tornaria muito caro o ambiente de funcionamento do sistema. Por esta razão, custo muito elevado, foi necessário ajustar os algoritmos do produto para que os requisitos de *hardware* não tornassem a utilização do produto muito cara. Quando diferentes decisões de projeto estiverem interconectadas, é praticamente impossível tratar de todos os problemas ao mesmo tempo e pelas mesmas pessoas.

A única forma de dominar a complexidade do projeto é separar as preocupações e decisões do projeto. Primeiramente alguém deve tentar isolar problemas que estão menos relacionados com outros. Depois, quando os problemas estiverem relacionados separadamente, os problemas relatados não devem ser considerados em todos os seus detalhes, mas apenas nos aspectos que estão relacionados com o principal problema em questão.

Existem diversas formas de aplicação da divisão de tarefas. Uma delas é a separação em termos de tempo, onde são definidos prazos e tarefas a serem realizadas nos prazos definidos. Esta aplicação caracteriza as bases de motivação dos ciclos de vida dos *software*: um modelo racional das seqüências de atividades que deveriam ser seguidas na produção de um *software*.

Outro tipo de aplicação da divisão de tarefas está definida em termos de qualidade, ou seja, quais qualidade serão tratadas de forma separada. Por exemplo, no caso do *software* nós podemos tratar separadamente a corretude e eficiência de um determinado programa. Alguém pode decidir primeiramente projetar um sistema no qual se espera que a corretude seja obtida *a priori*. Posteriormente o produto seria restruturado total ou parcialmente para melhorar sua eficiência e/ou desempenho.

Outro importante tipo de separação de preocupações permite diferentes "visões" do *software* para serem analisadas separadamente. Por exemplo, quando analisamos os requisitos de uma aplicação pode ser útil analisar separadamente os dados que fluem entre as atividades do sistema e o fluxo de controle que gerenciam as diferentes atividades do sistema. Ambas as visões nos auxiliam a entender o sistema em questão, mas nenhuma delas dá uma idéia do todo.

Ainda outro tipo de separação nos permite tratar com "partes" do mesmo sistema separadamente. Neste caso a abordagem é em relação ao tamanho das partes. Sistemas muito grandes são de difícil verificação e análise, portanto é importante analisar o sistema através de partes menores. Como as partes menores representam apenas parte do problema e problemas menores são, em geral, mais facilmente resolvidos esta abordagem garante que a separação de preocupações também se aplica a nível de modularização.

Num contexto mais amplo a separação de preocupações pode resultar na separação de responsabilidades no tratamento de problemas separadamente. Portanto, este princípio é a base para dividir o trabalho de solucionar problemas complexos em atribuições específicas de trabalho, possivelmente para diferentes pessoas com diferentes experiências e atribuições.

Modularização

Um sistema complexo pode ser dividido em partes mais simples chamadas módulos. Um sistema formado por módulos é dito um sistema modular e um sistema formado por um único e grande módulo é dito monolítico.

O principal benefício da modularização é que ela permite que o princípio da separação de preocupações seja aplicado em duas fases. Primeiramente quando tratarmos dos detalhes de cada módulo isoladamente (ignorando detalhes dos outros módulos) e, posteriormente, quando tratarmos das características globais de

todos os módulos incluindo seus relacionamentos, o que possibilita interligá-los para formar um sistema íntegro e coeso.

Se estas duas fases forem executadas na ordem mencionada, então dizemos que o sistema é projetado *bottom up* (ou de baixo para cima). Se forem executadas em ordem inversa será chamado de projeto *top down* (ou de cima para baixo).

A modularização é uma importante propriedade da maioria dos produtos e processos de engenharia. Na indústria automobilística, por exemplo, a construção de carros é feita pela montagem de partes que são projetadas e construídas separadamente. Além disso, estas partes são freqüentemente reutilizadas de um modelo para o outro, o que proporciona menores mudanças no futuro.

A modularização é, além de um princípio desejado de projeto, um mecanismo que acompanha totalmente a produção do *software*. Em particular existem três objetivos que a modularização tenta atingir na prática:

- capacidade de decompor um sistema complexo;
- capacidade de compor um sistema através de diferentes módulos;
- capacidade de entender o todo de um sistemas através de suas partes;

A capacidade de decomposição de um sistema está baseada na divisão do problema original de cima para baixo (abordagem *top-down*) em subproblemas ou problemas menores e então aplicar a decomposição para cada subproblema de forma recursiva. Este procedimento reflete claramente a frase-chavão do Latin *divide et impera* (ou divisão e conquista) a qual reflete a filosofia seguida pelos antigos romanos para dominar outras nações: primeiro dividí-los e isolá-los para então conquistá-los individualmente.

A capacidade de composição de um sistema está baseada em iniciar de baixo para cima (abordagem *bottom-up*) com componentes elementares e agrupando-os afim de compor módulos maiores, procedendo assim até finalizar o sistema. O exemplo da indústria automobilística pode ser novamente utilizado. Um carro é fabricado através do agrupamento de partes menores que vão formando o todo.

Na produção de *software* seria extremamente valioso se pudéssemos criar um novo *software* a partir de módulo já existentes e disponíveis em bibliotecas. Bastaria para isso compor os módulos de acordo com os requisitos do novo sistema. Para isso é necessário que tais módulos fossem projetados para serem reutilizáveis.

A capacidade de entender cada parte de um sistema separadamente com o objetivo de entender o sistema todo auxilia em futuras modificações do sistema. A natureza evolutiva dos *software* é tal que freqüentemente os engenheiros de *software* são solicitados para retomar trabalhos anteriores afim de modificá-los. Se o sistema só puder ser entendido na sua totalidade as modificações serão difíceis de serem aplicadas e o resultado não será confiável.

Para alcançar estes três objetivos da modularização os módulos devem ter bastante coesão e pouca interrelação. Um módulo tem bastante coesão se todos os seus elementos estiverem fortemente relacionados. Os elementos de um módulo (declarações, procedimentos e dados, p.e.) são agrupados no mesmo módulo por uma razão lógica e não apenas por acaso. Eles cooperam entre si para atingir um objetivo comum que é a funcionalidade do módulo.

A coesão é uma propriedade interna de um módulo enquanto que a interrelação caracteriza o relacionamento de um módulo com outro. Ela mede a interdependência de dois módulos, portanto, se os dois módulos dependem um do outro para funcionarem corretamente eles terão muita interrelação. O que se espera dos módulos é que eles tenham pouca interrelação, pois caso contrário eles serão difíceis de serem analisados, entendidos, modificados, testados ou mesmo reutilizados separadamente.

Uma estrutura modular com alta coesão e baixa interrelação funciona como uma caixa preta quando a estrutura geral de um sistema é descrita. Quando a funcionalidade de cada módulo é descrita e analisada ela reflete cada módulo separadamente.

Abstração

A abstração é um processo pelo qual identificamos os aspectos importantes de um fenômeno ignorando seus detalhes. Desta forma a abstração é um caso especial da separação de preocupações no qual separamos os aspectos importantes dos detalhes não importantes.

Aquilo que abstraímos ou consideramos como um detalhes que pode ser ignorado depende do propósito da abstração. Por exemplo, consideremos um relógio de pulso. Para o dono do relógio uma abstração útil é a simples observação dos efeitos de apertar seus botões para visualizar suas diferentes funcionalidades. Já para um técnico encarregado de consertar os relógios uma abstração útil é a existência de uma tampa ou proteção que pode ser aberta para substituição de pilhas. Podemos verificar que existem diferentes tipos e níveis de abstrações de uma mesma realidade, cada uma oferecendo uma visão da realidade e servindo a algum propósito específico.

A abstração acompanha todo e qualquer processo de implementação ou programação. As linguagens de programação que utilizamos nada mais são do que construções abstratas para representar ou interagir com o *hardware*. Elas nos proporcionam construções úteis e poderosas de forma que podemos escrever programas ignorando, na maioria dos casos, detalhes como o número de bits que são usados para representar números ou mesmo mecanismos de endereçamento. Isto permite que nos concentremos no problema a ser resolvido ao invés de nos preocuparmos em como instruir a máquina para resolvê-lo. Os programas que escrevemos são sempre abstrações.

A abstração é um importante princípio que se aplica tanto nos processos quanto nos produtos de *software*. Por exemplo, os comentários que aparecem a respeito de um procedimento ou função em um código fonte são abstrações que descrevem o propósito e os efeitos do mesmo. Quando a documentação do programa for analisada esta documentação oferece toda informação necessária para o entendimento das outras partes do programa que utilizam este procedimento.

Como exemplo de abstração a nível de processo podemos considerar a fase de análise de requisitos do *software*. Nesta fase podemos definir que o sistema terá como requisito mecanismos de criptografia e compressão de dados, porém não interessa neste momento saber como estes requisitos serão implementados a nível de linguagem de programação nem tampouco a nível de algoritmos. Posteriormente, até podemos estabelecer como requisitos os algorimtos e a linguagem de programação que implementarão estes mecanismos, mas mesmo assim os detalhes do código fonte continuarão desconhecidos ou abstraídos.

Antecipação de Mudanças

Os sistemas de *software* sofrem mudanças constantemente. Como já visto, as mudanças são necessárias ou para reparar o *software* (eliminação de erros) ou para suportar a evolução e os novos requisitos das aplicações. Esta é a razão pela qual definimos que a manutenabilidade é uma das principais qualidades de *software*.

A habilidade do *software* em poder evoluir não vem de graça. Ela requer um esforço especial afim de antecipar como e onde as mudanças provavelmente irão acontecer. Quando as prováveis mudanças são identificadas um cuidado especial deve ser tomado para que procedamos de forma que futuras mudanças sejam fáceis de serem aplicadas. Estes pontos serão discutidos com mais detalhes no capítulo 5 - *Projeto de Software*.

A antecipação de mudanças é talvez o único princípio que distingue o *software* de todos os outros tipos de produção industrial. Em muitos casos uma aplicação é desenvolvida enquanto seus requisitos ainda não foram totalmente entendidos. Desta forma, após ser liberado e baseado no *feedback* dos usuários a aplicação pode evoluir na medida que novos requisitos forem descobertos ou os requisitos antigos forem atualizados. Além disso, as aplicações são freqüentemente embutidas em um ambientes que possuem uma certa estrutura organizacional por exemplo. O próprio ambiente (ou estrutura organizacional) é afetado pela introdução da aplicação e isso pode gerar novos requisitos que não existiam anteriormente. A antecipação de mudanças é, sem dúvida, um princípio que podemos utilizar para atingir a evolutanabilidade.

A reusabilidade é outra qualidade de *software* que é frenquentemente afetada pela antecipação de mudanças. Como visto, um componente é reusável se ele for diretamente utilizado para a geração de um novo produto. Porém, um componente pode necessitar de uma pequena evolução (ajustes) antes ser reutilizado. Desta forma, a reusabilidade pode ser vista como uma pequena parte da evolutanabilidade. Neste caso, evolutanabilidade é definida a nível de componentes e não de *software*. Se pudermos antecipar

as mudanças de contexto no qual um componente de *software* pode ser embutido, nós podemos então projetar o componente de uma maneira que tal evolução poderá ser feita facilmente.

A antecipação de mudanças requer que ferramentas apropriadas estejam disponíveis para gerenciar as várias versões e revisões do *sofware* de uma maneira controlada. Isto se deve a necessidade de armazenar e recuperar documentação, códigos fontes, objetos, etc., de bases de dados que servem como repositórios centrais de componentes reutilizáveis. O acesso à base de dados deve ser controlado. Um sistema deve ser mantido consistente mesmo quando as mudanças forem aplicadas apenas em alguns de seus componentes.

Embora as atenções tenham sido voltadas mais para produto de *software* a antecipação de mudanças também afeta o gerenciamento do processo de produção. Por exemplo, os gerentes devem antecipar os efeitos de mudanças pessoais nos processos e, portanto, quando o ciclo de vida de uma aplicação for projetado é importante levar isso em consideração. Dependendo da antecipação de mudanças os gerentes devem estimar custos e projetar a estrutura organizacional que suportará a evolução do *software*. Assim, os gerentes poderão decidir se vale a pena investir tempo e esforço na produção de componentes reutilizáveis, na produção de acessórios de projetos para desenvolvimento de *software* ou ainda investir tempo e esforço em desenvolvimentos paralelos.

Generalização

O princípio da generalização pode ser analisado sob a seguinte situação. Toda vez que você for solicitado para resolver um determinado problema tente, primeiramente, se focar na descoberta de um problema mais geral que possa existir por trás do problema em questão. Pode acontecer que o problema generalizado não seja mais complexo que o problema original. Sendo mais geral é provável que a solução para a generalização do problema tenha um potencial maior de ser reutilizada. Pode também acontecer que uma solução genérica já exista através de alguma biblioteca de desenvolvimento. E ainda pode acontecer que generalizando um problema você acabe projetando um módulo que seja utilizado em vários pontos da aplicação ao invés de projetar e desenvolver diversas soluções especializadas que muitas vezes são utilizadas para resolver problemas semelhantes.

Soluções generalizadas, por outro lado, podem ser mais custosas em termos de velocidade de execução, requisitos de memória e/ou tempo de desenvolvimento do que soluções que são “feitas sob medida” para o problema original. Por estas razões é necessário avaliar as vantagens da generalização no que diz respeito a custos e eficiência, para que seja possível decidir se vale a pena resolver o problema generalizado ou o problema original.

Por exemplo, suponha que você seja solicitado para unir dois arquivos seqüenciais ordenados em um único arquivo seqüencial, também ordenado. Baseado nos requisitos do *software*, você sabe que os dois arquivos originais não contém nenhum registro com valores de chaves iguais. Obviamente, se você generalizar sua solução para aceitar arquivos que possuam chaves duplicadas você está oferecendo uma solução com mais possibilidades de ser reutilizada do que uma solução que só funcionaria para arquivos sem duplicação de chaves.

A generalização é um princípio fundamental se nossos objetivos como engenheiros de *software* for desenvolver ferramentas genéricas e pacotes de *software* para o mercado. O sucesso de aplicativos como planilhas eletrônicas, banco de dados e processadores de textos é que eles são genéricos o suficiente para resolver a maioria das necessidades práticas dos usuários. É obvio que desenvolver aplicativos que solucionem problemas específicos de cada usuário ou empresa é menos econômico do que adquirir soluções já existentes no mercado, ou seja, se um problema qualquer puder ser definido como parte de um problema maior já solucionado por um pacote genérico é mais conveniente adotar o pacote do que implementar uma solução especializada.

Incrementabilidade

A incrementabilidade é o princípio que busca a perfeição ou a obtenção dos objetivos através de passos que evoluem (ou são incrementados) ao longo do tempo. Cada passo é alcançado através do incremento do passo anterior.

Este princípio pode ser aplicado a muitas atividades de engenharia. Quando aplicado ao *software* ele significa que a aplicação desejada é produzida como uma conseqüência de um processo evolutivo.

Uma forma de aplicar a incrementabilidade consiste na identificação antecipada dos subconjuntos (partes) mais úteis de uma aplicação para que os mesmos possam ser desenvolvidos e liberados ao cliente com o objetivo de obter *feedback* também antecipadamente. Isto permite que a aplicação seja desenvolvida de uma maneira controlada mesmo quando os requisitos iniciais ainda não foram estabelecidos ou totalmente entendidos.

A motivação para o uso deste princípio se deve ao fato de que, na maioria dos casos, não existe uma maneira correta de obeter todos os requisitos antes da aplicação ser desenvolvida. Sendo assim, os requisitos emergem em conjunto quando a aplicação, ou parte dela, estiver disponível para experimentação prática. Consequentemente, quanto mais cedo os desenvolvedores receberem *feedback* dos usuários no que se refere às preocupações e utilidades da aplicação, mais facilmente será possível incorporar as mudanças requisitadas para o produto.

Um exemplo prático da utilização da incrementabilidade pode ser uma abordagem comum no desenvolvimento de sistemas, ou seja, desenvolver um produto inicialmente operacional para somente mais tarde torná-lo eficiente. Neste caso a versão inicial do sistema poderia enfatizar a interface com o usuário e/ou a confiabilidade para somente em futuras versões apresentar eficiência de execução e otimização dos recursos.

Quando uma aplicação é desenvolvida incrementalmente os estágios intermediários podem constituir protótipos do produto final ou, em outras palavras, os módulos operacionais seriam apenas aproximações (a nível funcional) do produto final.

A idéia de desenvolver produtos de *software* através de protótipos rápidos é frequentemente usada como forma de desenvolvimento progressivo de uma aplicação em paralelo ao entendimento de seus requisitos. O ciclo de vida de um *software* baseado no desenvolvimento prototipado é mais flexível e iterativo do que outros modelos, como o modelo cascata por exemplo (capítulo 4). A diferença entre eles terá efeitos não só nos aspectos técnicos dos projetos mas também nos aspectos gerenciais e organizacionais das equipes de desenvolvimento.

Desta forma, como já mencionado na antecipação de mudanças, software que evoluem através de protótipos requerem cuidados especiais no gerenciamento dos documentos, programas, testes, etc, desenvolvidos para as várias versões do software. Cada passo incremental significativo deve ser armazenado, a documentação deve poder ser facilmente recuperada, as mudanças devem ser aplicadas de forma controlada. Se isto não for feito de forma cuidadosa qualquer intenção de desenvolver um software de forma evolutiva pode facilmente se tornar em um processo indisciplinado de desenvolvimento de software e, conseqüentemente, todas as vantagens em potencial da incrementabilidade serão perdidas.