UNIVERSIDADE FEDERAL DE SANTA CATARINA

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

Elias Alexandrino de Souza Júnior

Fabiano Chiqueti

SISTEMA GERADOR DE INSTALADORES DE PROGRAMAS NO FORMATO JAR

Trabalho de Conclusão de Curso submetido à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Bacharel em Ciências da Computação.

Orientador: Vitório Bruno Mazzola

Florianópolis-SC, maio de 2005.

SISTEMA GERADOR DE INSTALADORES DE PROGRAMAS NO FORMATO JAR

Elias Alexandrino de Souza Júnior

Fabiano Chiqueti

Prof. Orientador: Vitório Bruno Mazzola

Membro da Banca: Prof. Mario Dantas

Membro da Banca: Prof. Rosvelter C. da Costa

Florianópolis. Maio de 2005

"Me dê o zero e o um que eu tiro o tudo e o nada lá de dentro."

Júlio Felipe Szeremeta

RESUMO

A preparação do produto final da produção de software para ser entregue ao cliente é muitas vezes desconsiderada ou até esquecida no início de um projeto, levando-se em consideração a sua simplicidade quando comparada ao restante do processo de desenvolvimento. Entretanto, esta preparação nem sempre se da de maneira trivial.

No que diz respeito a aplicações desenvolvidas em Java, a entrega de inúmeros arquivos *.class* é no mínimo pouco prática. O empacotamento desses arquivos em um único arquivo .zip ou .jar não resolve totalmente o problema, pois para aplicações *desktop* é necessária uma maneira de executar o programa localmente.

Este trabalho se propõe ao desenvolvimento de uma ferramenta que possa solucionar este problema, gerando um gerador de instaladores de programas feitos em Java.

ABSTRACT

The preparation of the result of the production of software to be delivers to the customer is often not considered or forgotten in the beginning a project, taking itself in consideration the its simplicity when compared with the remain it development process. However, this preparation is not so trivial.

About the applications developed in Java, the delivery of innumerable archives class is not so practical. The wrapping up of these archives in an only archive zip or jar does not entirely solve the problem, therefore for desktop applications is necessary a way to execute the program in the local machine.

This work considers to the development of a tool that can solve this problem, resulting in a generator of installers of Java programs.

AGRADECIMENTOS

Agradecemos a Deus, a dona Fátima e seu Elias (pais do Elias Jr.) ao Saulo e à Isabella (irmãos do Elias), à tia Seni (mãe do Chiqueti) e ao seu Erondino (homenagem póstuma ao pai do Chiqueti), à Tânia e o Gabriel (esposa e filho do Elias ("produzido" durante o TCC)) à Nini, Gabriel e Davi (esposa e filhos do Chiqueti (sendo Davi também "produzido" durante o TCC)) pelo apoio e paciência (e que paciência...). Não podemos nos esquecer do nosso orientador Mazzola e dos membros da banca professores Mario Dantas e Rosevelter Coelho da Costa. Ah, também agradecemos a Beth da secretaria e a todos os amigos *mandrakes* que se não nos ajudaram, pelo menos não nos atrapalharam...

SUMÁRIO

CAPÍTULO 4

CAPÍTULO 5

CAPÍTULO 6

ÍNDICE DE FIGURAS

CAPÍTULO 1

1. INTRODUÇÃO

O desenvolvimento de software sempre foi uma atividade em ascensão e expansão no cenário mundial. A necessidade da implementação da mais diversa gama de novas soluções, migração de sistemas velhos para sistemas novos, manutenção e extensão de soluções já existentes é somente uma das causas deste crescimento. Uma conseqüência deste cenário é o surgimento de novas tecnologias e entre elas aparece o Java.

Entretanto, a indústria de ferramentas de desenvolvimento de software vem lucrando espantosamente em cima desta necessidade de mercado, o que muitas vezes acaba colocando empecilhos à pesquisa acadêmica nesta área. Todavia, o software livre e versões *trial* vem sendo uma opção bem atrativa tanto para o meio acadêmico quanto para a própria indústria de produção de software.

Como qualquer projeto, uma aplicação em Java necessita ser entregue ao cliente em um formato que facilite ao máximo a sua instalação e utilização. Quanto à utilização, seria assunto para engenharia de usabilidade, o que não está diretamente ligado a este trabalho. Aqui trataremos da instalação de programas Java.

Este trabalho se propõe ao desenvolvimento de uma ferramenta que gere um arquivo no formato jar tomando como entrada arquivos compilados de

unidades Java (.class) e que através deste arquivo de extensão jar seja possível executar o programa desenvolvido, sem maiores complicações. Em uma segunda instância poderá se chegar a um software gerador de instalador, similar a programas citados no próximo capítulo.

O trabalho tem a seguinte divisão: no capítulo 2 um embasamento teórico sobre a tecnologia utilizada e seus aspectos referentes ao trabalho. O capítulo 3 apresenta uma pesquisa sobre o estado da arte, mostra algumas tecnologias já existentes e enfatiza suas principais características. Já no capítulo 4, é apresentada a definição do problema e sua proposta de solução. Logo após, no capítulo 5, é abordada a metodologia empregada na solução.

O trabalho segue com os ciclos de desenvolvimento no capítulo 6 e no capítulo 7 temos as conclusões e sugestões para trabalhos futuros.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

Apresentar os aplicativos existentes para geração de instaladores de programas desenvolvidos na linguagem Java, mostrando suas vantagens e desvantagens. Além disso, será proposta e desenvolvida uma nova solução.

1.1.2 Objetivos Específicos

- Pesquisar metodologias de desenvolvimento de software OO bem como modelos de ciclo de vida de software para um melhor entendimento do processo de produção de software.

- Desenvolver uma ferramenta de geração de instaladores de programas no formato jar, dentro de preceitos de engenharia de software, para que se obtenha um produto de qualidade que possa ser efetivamente aproveitado.

CAPÍTULO 2

2. A TECNOLOGIA JAVA

A linguagem Java foi criada pela Sun Microsistems no começo dos anos 90. Além de ser uma linguagem de programação orientada a objetos, multi-plataforma e acessível a qualquer interessado, têm sido utilizadas cada vez mais, tanto no mundo acadêmico quanto em aplicações comerciais e o número de “adeptos” tem crescido espantosamente [DEITEL].

Atualmente é amplamente utilizada em aplicações web, mas também podem ser construídos programas para desktop. Além da vasta utilização, muitos frameworks, componentes e tecnologias têm sido desenvolvidas “em Java e para Java”, o que vem formando uma “cultura Java”.

Portanto, Java é a cara do desenvolvimento de software neste começo de século e, ao que tudo indica, continuará no ápice ainda por muito tempo.

2.1 Arquivos de extensão jar.

Um arquivo de extensão jar é uma versão mais sofisticada do popular arquivo de extensão zip (arquivo compactado). Além de compactar quantos arquivos forem necessários mantendo a estrutura de diretórios, um arquivo de

extensão jar pode servir como um similar ao executável no Windows, porém multi-plataforma [JAVA].

Para tornar um arquivo jar um programa “executável”, basta inserir neste jar todos os arquivos compilados de um programa desenvolvido em Java (.class). Também deve ser inserido um diretório META-INF contendo um arquivo de manifesto (manifest.mf). Definindo devidamente neste arquivo de manifesto as bibliotecas que o programa utiliza (extensões), classe principal (main), etc, o jar se tornará então uma aplicação “executável” [JAVA].

Presente no kit de desenvolvimento do Java (SDK) desde a versão 1.1, este formato é muito utilizado para empacotamento de bibliotecas de classes, de componentes e até mesmo de frameworks. Portanto, já está mais do que agregado à cultura Java, sendo uma tecnologia padrão neste meio [JAVA].

2.2 Arquivo de manifesto

Como dito anteriormente, o manifesto é um arquivo que se localiza dentro do diretório META-INF de um arquivo de extensão jar [JAVA]. Este arquivo possui várias seções, cujas principais e relacionadas com este trabalho são:

Manifest-Version: Define a versão do arquivo de manifesto [JAVA].

Created-By: Define a versão e o distribuidor da implementação Java na qual o referido manifesto é gerado. Este atributo é gerado pela ferramenta que gera o manifesto [JAVA].

***Signature-Version*:** Define a versão da assinatura* do arquivo jar. O valor deve ser uma string de um número de assinatura válido [JAVA].

Class-Path: O valor deste atributo especifica a(s) URL(s) relativas às extensões ou bibliotecas requisitadas por este arquivo jar, que são geralmente estão na forma de outros arquivos de extensão jar. Estas URLs são separadas entre umas das outro pelo caractere espaço. O "*Class loader*" da aplicação usa este valor para construir um serch-path interno [JAVA].

Main-Class: O valor deste atributo define o caminho relativo da classe principal (main) da aplicação, ou seja, a classe por onde a aplicação deve começar a ser executada. Este valor não deve conter a extensão .class da classe principal, mas apenas seu nome completo, incluindo o de seu pacote [JAVA].

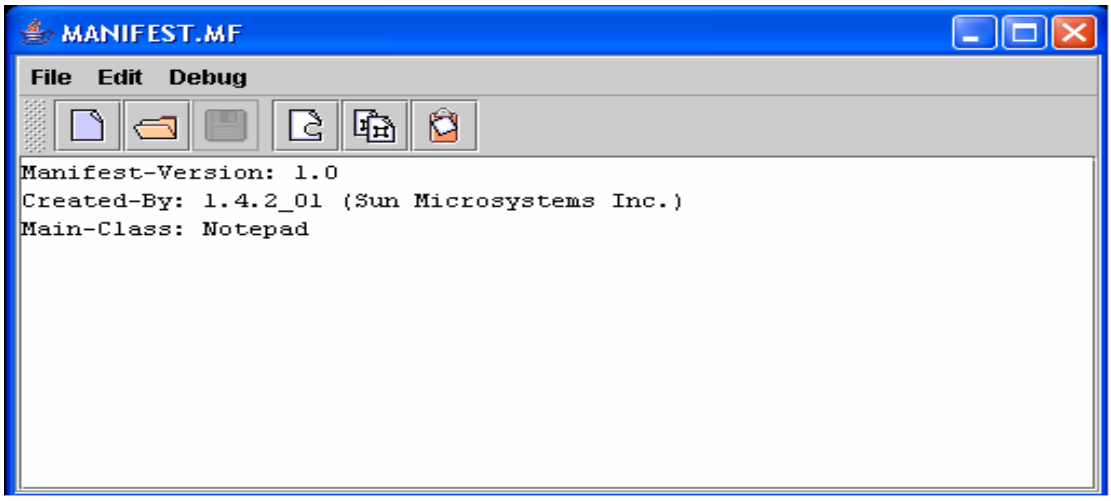

Figura 1 - Arquivo de manifesto da aplicação Notepad, mostrado na própria aplicação.

CAPÍTULO 3

3. ESTADO DA ARTE

Este trabalho não tem a pretensão de inovar, mas de criar uma ferramenta que agregue funcionalidades existentes em outros softwares. Algumas soluções do mercado foram analisadas e alguns programadores e analistas que trabalham com Java foram consultados para que este trabalho pudesse ajustar seu foco.

A seguir serão mostradas algumas tecnologias que se assemelham ou têm algo em comum com a proposta inicial deste trabalho.

3.1 Jar

Conforme informações contidas no site da Sun [JAVA], o SDK já vem com uma excelente ferramenta para criação de arquivos de extensão jar. A utilização pura e simples desta ferramenta é feita pelo do prompt do sistema operacional através do comando **jar *parâmetro(s) jar-file arquivos(s)* :**

Jar – comando de chamada do Jar.

Parâmetro(s) **–** Lista de parâmetros que indicam opções de compactação como se será criação ou atualização, o que deve aparecer na tela como retorno, etc.

Jar-file **–** Nome do arquivo de extensão jar onde serão compactados os arquivos. Se for omitido, será a.jar por padrão.

Arquivos(s) **–** Lista de 0 ou mais arquivos que serão inseridos no *Jar-file.*

A grande desvantagem deste programa é que, se a lista de arquivos a serem inseridos no jar for muito extensa, sua utilização se torna impraticável em tempo hábil. Além disso, a edição do arquivo de manifesto deve ser manual. Por padrão, ao ser criado um arquivo de extensão jar já é inserido automaticamente um manifesto, porém sem os parâmetros essenciais a sua execução como *Class-path* e *Main-class,* mas com apenas as seções *Manifest-Version* e *Created-by*.

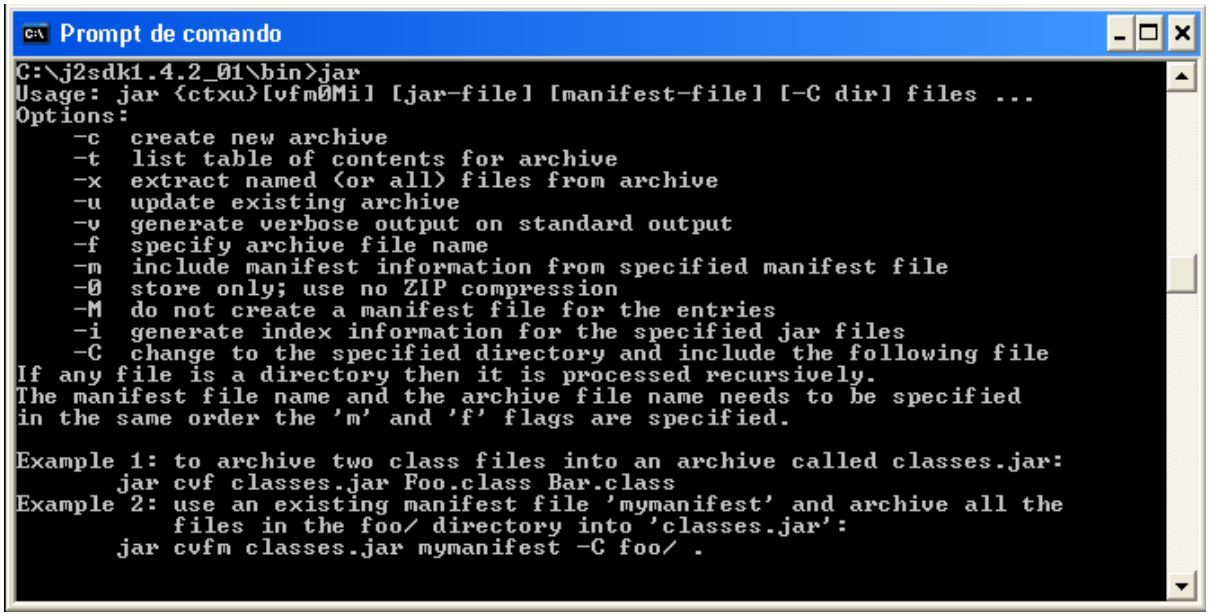

Figura 2 - Jar no prompt do MS-DOS.

3.2 Installanywhere

O Installanywhere é uma poderosa ferramenta produzida pela Zero G que gera um instalador em várias plataformas e tem uma interface mais do que amigável [ZERO G]. Além disso, o produto final poderá instalar ou não a jvm (Java virtual machine) antes de instalar o software propriamente, ou seja, pode tanto gerar uma aplicação menor para um sistema que possui a jvm quanto um instalador mais completo, que dispensa uma instalação prévia da jvm.

Além dessas, o installanywhere possui inúmeras outras funcionalidades. A única desvantagem desta ferramenta é o fato de ser paga (não gratuita), tendo uma versão *trial* para 30 dias.

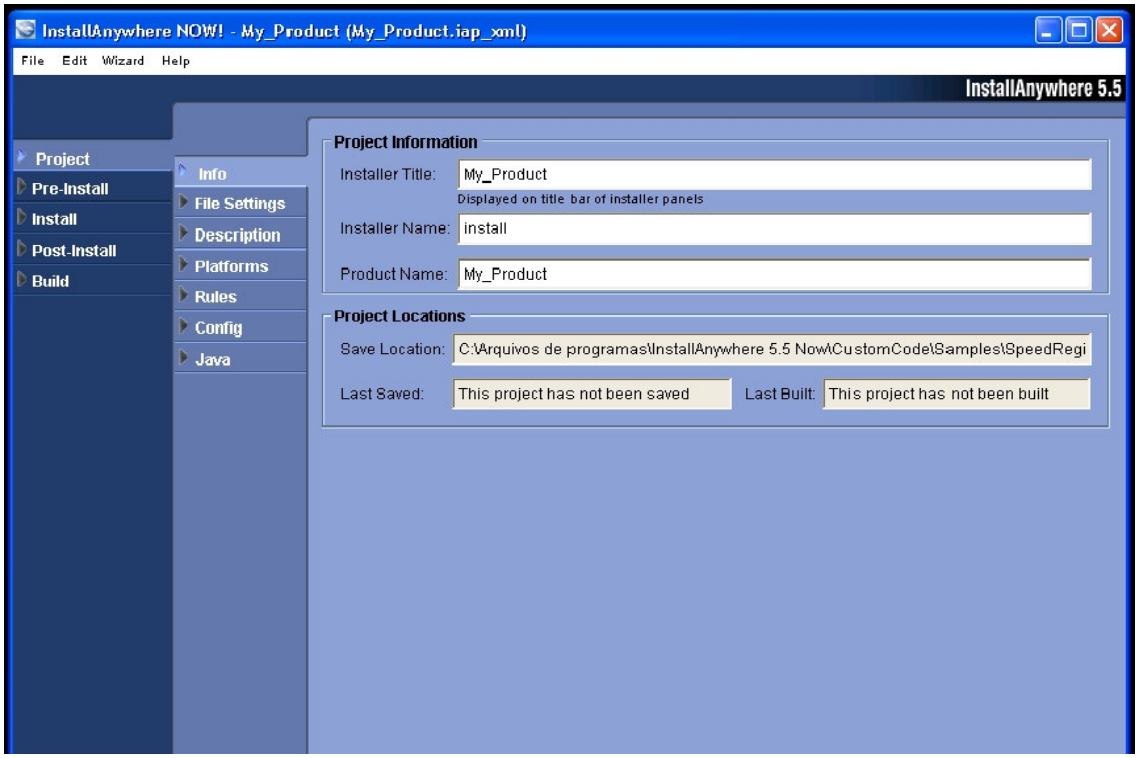

Figura 3 - Janela principal do Installanywhere

3.3 Outros instaladores

Não foram encontrados outros instaladores tão específicos para programas Java como o installanywhere. Alguns deles eram de uso gratuito e outros não, mas nenhum deles mostrou-se eficiente para geração de instaladores de programas feitos em Java.

3.4 As IDE's e seus plug-ins

Existem também IDE's para desenvolvimento em Java, como o Eclipse e o Netbeans, que possuem um plug-in que gera um jar. Entretanto, estes plug-ins não geram um instalador do programa, mas apenas um arquivo jar executável. Alem disso, é mais conveniente editar manualmente o arquivo de manifesto do que passar todos os parâmetros necessários a edição do mesmo através de um wisard, pois pelo menos no caso do Eclipse não há tratamento para o uso de bibliotecas ou extensões.

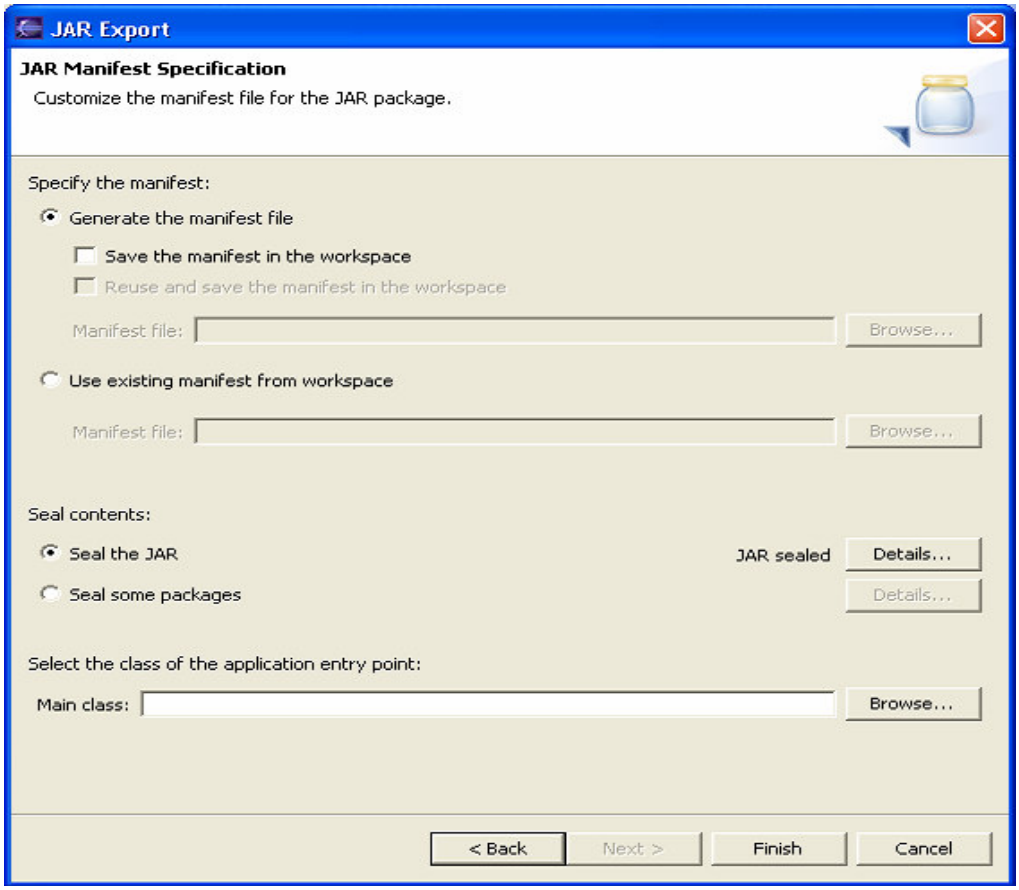

Figura 4 - Wisard do eclipse de exportação para arquivo jar

Logo, estas ferramentas também estão longe de ser uma solução ideal para obtenção de um produto final para ser entregue a um cliente.

3.5 Jarbuilder

O Jarbuilder é um software ainda em desenvolvimento e *freeware* que toma como entrada arquivos de qualquer extensão gera um jar como saída [JARBUILDER]. Através deste software o usuário pode escrever automaticamente o arquivo de manifesto, anexar um arquivo já editado, pedir ao programa que este gere o arquivo de manifesto automaticamente ou

simplesmente não colocar nenhum arquivo de manifesto (isso quando o usuário deseja apenas compactar uma biblioteca de classes ou arquivos quaisquer).

Este programa apresenta também uma funcionalidade bem interessante: dentre todos os arquivos de entrada, encontra automaticamente a classe principal (main) do programa a ser compactado. Além disso, possui uma interface gráfica bem amigável.

Entretanto, o Jarbuilder não funciona quando o programa a ser colocado no formato jar utiliza bibliotecas de classes.

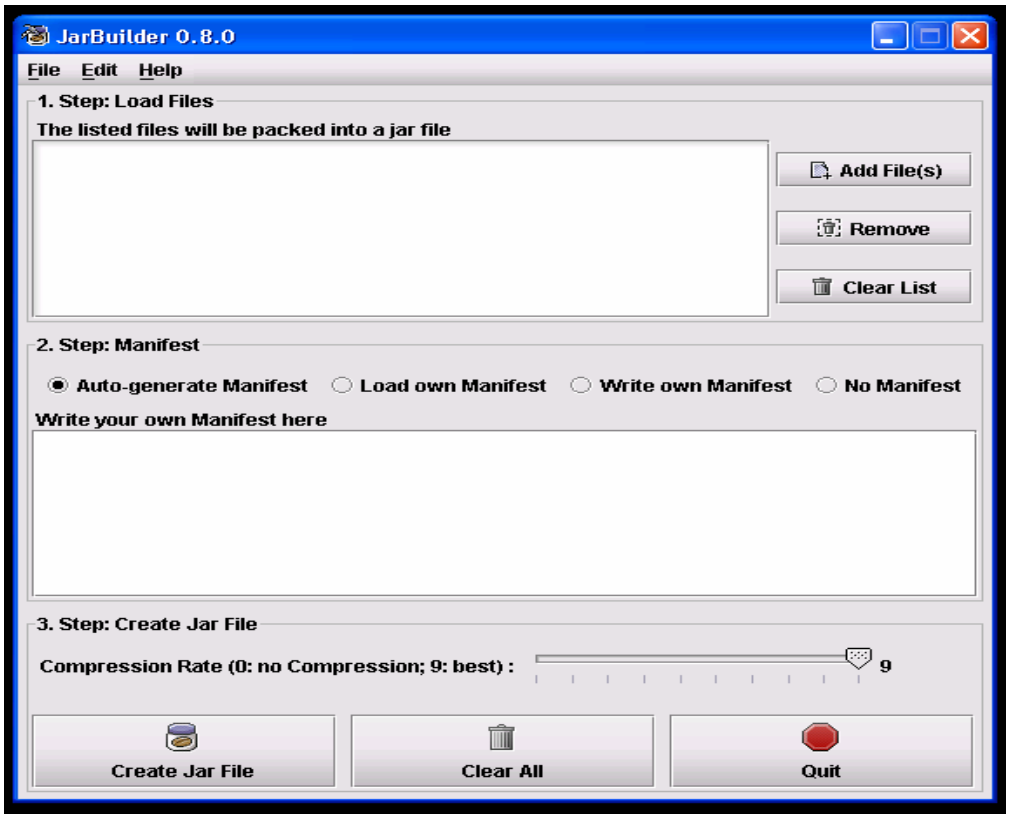

Figura 5 - Janela principal do Jarbuilder.

3.6 Conclusões da pesquisa do estado da arte.

A carência do mercado em relação a um gerador de instalador de programas feitos em Java se deve a reduzida utilização desta linguagem para aplicações desktop. Uma justificativa para este fato é a lentidão de uma aplicação para desktop em relação a uma solução implementada em C++ ou Delphi, por exemplo.

Todavia, é importante ressaltar que a Sun Microsistems tem lançado constantemente novas versões da JVM e estas cada vez mais otimizadas, o que faz com que programas Java sejam executados de maneira cada vez mais rápida [JAVA]. Aliás, a performance dos computadores está aumentando cada vez mais, o que fará com que uma diferença de performance entre programas Java e outros seja cada vez menor, tornando-se imperceptível para o usuário.

No que diz respeito às atuais opções existe uma boa solução no mercado: o Installanywhere. Entretanto, este software é pago. Existem soluções gratuitas disponíveis: o Jarbuilder, que não funciona corretamente, pois é apenas um protótipo e o Jar nativo do SDK, que também apresenta limitações. Os plug-ins das IDE's também possuem deficiências. O levantamento do problema e sua solução serão abordados nos próximos capítulos.

CAPÍTULO 4

4. DEFINIÇÃO DO PROBLEMA E SOLUÇÃO

4.1. Especificação do problema

Como citado anteriormente, existe uma grande dificuldade no meio acadêmico e nas pequenas empresas de desenvolvimento de software: o custo na aquisição de ferramentas de desenvolvimento.

Algumas corporações que constroem estas ferramentas de desenvolvimento disponibilizam versões gratuitas de softwares para estudante, como é o caso da borland com o Together. Mas isso não livra as pequenas empresas do problema. Além disso, existem muitas ferramentas que não possuem esta versão gratuita, mas uma versão *trial*, ou então não possui versão gratuita alguma.

Há também bons softwares gratuitos, como as ide's Eclipse (que foi utilizado no desenvolvimento desde projeto) e Net Beans, gerenciador de banco de dados PostGree SQL, My SQL e Firebird e até mesmo sistemas operacionais, como o as várias distribuições gratuitas do linux. O problema da substituição de aplicações pagas por software livre está no fato de que geralmente as ferramentas gratuitas são de utilização mais difícil ou então

menos conhecida, o que acaba gerando um custo em treinamento da equipe, sem contar com uma queda da produtividade no período de adaptação.

Pode-se ainda citar a pirataria, à qual muito acadêmicos e mesmo pequenas empresas recorrem.

Fazendo-se ainda considerações em relação às soluções citadas no capítulo anterior e sua relação com uma equipe de desenvolvimento: Quanto ao installanywhere, não existem críticas de usabilidade tão pouco de performance ou robustez. Evidentemente, é necessário um estudo desta ferramenta para que seja possível a sua utilização, o que pode gerar no máximo uma temporária diminuição na produtividade de uma equipe de desenvolvimento, o que pode ser resolvido com um pequeno remanejamento de pessoal. Mas o grande problema deste produto é o seu custo, ou ainda o fato da existência de um custo.

Quanto ao jar nativo do SDK, não existem críticas quanto ao custo, mas ao contrário, sua usabilidade torna-se inábil quando se deseja gerar um jar executável de aplicações de médio e grande porte.

No que diz respeito ao Jarbuilder não funciona corretamente.

Ainda existem várias IDE's que trazem como plug-in um gerador de jars, mas a maioria não edita efetivamente o arquivo de manifesto.

Portanto, existe uma carência de uma solução gratuita e de qualidade.

4.2. Especificação da solução

Este trabalho tem por finalidade o desenvolvimento de uma ferramenta para futuro uso gratuito, que seja eficiente (robusta e funcional) e tenha razoável usabilidade. Não tem a pretensão de concorrência com o Installanywhere, pois isso demandaria uma equipe maior e mais tempo para desenvolvimento, porém deve ser uma boa alternativa na geração de um único arquivo para instalação ou mesmo para ser executado.

A alternativa seria quase como que uma nova versão do Jarbuilder, porém que efetivamente funcione para qualquer programa Java, mesmo os que utilizem bibliotecas ou extensões. Com base nessas conclusões, apresenta-se a seguir um levantamento de requisitos inicial.

4.3 Requisitos Funcionais

Os requisitos funcionais do projeto foram levantados baseados na experiência dos autores do trabalho no desenvolvimento e posterior utilização de aplicações Java para desktop.

R1	Tomar como entrada quaisquer tipos de arquivos

R2	Criar automaticamente um arquivo de formato jar, com nome e localização (diretório) definidos pelo usuário.
R3	Ter a capacidade de apresentar interface no formato texto.
R4	Ter a capacidade de apresentar interface no formato gráfico (janela ou formulário).
R5	Ter a capacidade de ser executado por linha de comando.
R6	Ter a capacidade de ser executado em qualquer sistema operacional (multi-plataforma).
R7	Sobrescrever arquivos jar já existentes.
R8	Editar e inserir automaticamente o arquivo de manifesto no arquivo jar, definindo atributos que possibilitem a utilização de bibliotecas de classes e a classe main do programa tomado como entrada.
R9	Possibilitar a inserção de um arquivo de manifesto já existente e a não inserção de arquivo de manifesto no arquivo jar de saída.
R10	Avisar o usuário de qualquer erro de execução.

Uma primeira versão deste sistema é um gerador de programas no formato jar. Isso facilitaria a uma equipe de desenvolvimento a entrega de um produto final ao seu cliente. Um programa a ser entregue ao cliente nada mais seria do que um único arquivo, de execução trivial, produto do software desenvolvido.

4.4. Requisitos não funcionais.

Evidentemente, o trabalho é desenvolvido em Java e por isso deverá ser compatível com toda e qualquer plataforma com a JVM. O tempo de resposta (processamento) deve ser hábil, mesmo para a obtenção de arquivos jar resultantes de aplicações de grande porte, não desconsiderando a robustez.

Além de uma boa performance da aplicação, esforço intelectual e físico do usuário deve ser minimizado, maximizando assim a usabilidade intuitiva da aplicação. Um manual de utilização deverá ser algo dispensável. É claro que o software não terá a necessidade de ser de uso intuitivo para leigos, sendo que seu público alvo será estudantes e profissionais desenvolvedores de software em Java. Poderá ser algo similar ao próprio jar nativo do SDK, porém mais funcional.

CAPÍTULO 5

5. METODOLOGIA EMPREGADA

Este trabalho trata-se de um projeto orientado a objetos. Para seu desenvolvimento, foi adotada uma adaptação da metodologia Fusion (COLEMAN, 1996), utilizando um modelo espiral da seguinte forma: O projeto é desenvolvido em ciclos de desenvolvimento. Cada ciclo deverá solucionar um ou mais requisitos funcionais do sistema (requisitos descritos no ítem 4.3 do capítulo anterior) e deverá resultar em um protótipo que será então testado. Maiores informações e descrições de cada ciclo podem ser encontradas no próximo capítulo.

5.1. Metodologias de desenvolvimento de software OO

Diversos métodos de análise e projeto orientado a objetos têm sido propostos nos últimos anos, entre eles:

- CRC (Class Responsability Collaborator, Beecke e Cunningham, 1989)
- OOA (Object Oriented Analysis, Coad e Yourdon, 1990)
- Booch (Booch, 1991)
- OMT (Object Modeling Technique, Rumbaugh, 1991)
- Objectory (Jacobson, 1992)
- Fusion (Coleman e outros, 1994)

O objetivo desses métodos é tornar a análise compatível com a codificação do software, o que não seria efetivo se métodos baseados em outros paradigmas fossem aplicados.

5.1.2. A Metodologia Fusion

Fusion é um método de desenvolvimento de software orientado a objeto de segunda geração. Foi concebido em 1992 por um grupo de pesquisadores da HP. Segundo COLEMAN, Fusion integra e estende os principais dispositivos das abordagens orientadas a objeto de mais sucesso no mercado: OMT, Booch, CRL e Objetory. Fusion é composto de três fases: análise, projeto e implementação; cada fase é descrita, bem como seus modelos e notações. SILVA diz que esta metodologia se caracteriza por ser uma fusão de características de outras, tendo modelos diferentes nas fazes de análise e projeto. Outra característica do Fusion, também segundo SILVA, é a inexistência da noção de método de classe durante a fase de análise, sendo denominada operação de sistema.

5.2. Modelos de ciclo de vida

Pressman (1995) aborda quatro paradigmas ou modelos de ciclo de vida, denominados: modelo de ciclo de vida clássico, ou em cascata; modelo de prototipação; modelo espiral; e técnicas de quarta geração.

Um modelo de ciclo de vida define como as etapas da construção de software (análise, projeto, implementação, testes e manutenção) se sucedem, em que consiste cada uma e como harmonizam suas atividades [SILVA].

Modelos de ciclo de vida são estruturas que definem as macro-atividades de um processo, sua precedência e dependência [BORGES].

5.2.1. O Modelo Espiral

Segundo SILVA, o modelo espiral inclui entre as etapas do ciclo tradicional (análise, projeto, implementação, teste e manutenção) as atividades abaixo:

- Planejamento da próxima etapa
- Determinação de objetivos, alternativas e limitações.
- Avaliação das alternativas, identificação e solução de riscos.
- Prototipação (simulações e avaliações)

Portanto, embora longe da perfeição, o modelo espiral preenche lacunas de modelos de ciclo vida de software anteriores, como o *code and fix*, cascata e em V.

5.3 Padrões de Projeto

Um padrão descreve uma solução para um problema que ocorre freqüentemente durante o desenvolvimento de software, podendo ser considerado como um par “problema/solução” [BUSCHMANN]. Projetistas familiarizados com certos padrões podem aplicá-los imediatamente a problemas de projeto, sem ter que redescobri-los [GAMMA].

Um padrão é um conjunto de informações instrutivas que possui um nome e que capta a estrutura essencial e o raciocínio de uma família de soluções comprovadamente bem sucedidas para um problema repetido que ocorre sob um determinado contexto e um conjunto de repercussões [APPLETON].

Padrões de software podem se referir a diferentes níveis de abstração no desenvolvimento de sistemas orientados a objetos. Assim, existem padrões arquiteturais, em que os níveis de abstração são bem altos, padrões de análise, padrões de projeto, padrões de código, entre outros. Além disto, eles atuam como blocos construtivos a partir dos quais projetos mais complexos podem ser construídos [GAMMA].

CAPÍTULO 6

6. CICLOS DE DESENVOLVIMENTO

Primeiramente, na descrição dos ciclos de desenvolvimento existe a necessidade da convenção de uma nomenclatura para melhor compreensão dos procedimentos e exemplos. A aplicação desenvolvida neste trabalho será chamada **Gerajar** e um programa que esta aplicação esteja transformando para o formato jar será referido como **programa objeto**. O arquivo jar que o Gerajar resultar a partir de um programa objeto será denominado **arquivo jar resultante**.

A duração de cada ciclo depende do tempo de desenvolvimento. Sendo o projeto desenvolvido em uma simplificação da metodologia *fusion* usando um modelo de ciclo de vida espiral, cada ciclo de desenvolvimento terá uma fase de levantamento e/ou revisão de requisitos, análise, projeto, implementação e testes.

Nos tópicos seguintes, serão descritos os ciclos de desenvolvimento do trabalho.

6.1 Primeiro ciclo de desenvolvimento

6.1.1 Redefinição dos requisitos

Com este é o primeiro ciclo de desenvolvimento, não houve redefinição de requisitos. Neste ciclo foram solucionados os requisitos seguintes:

R1	Tomar como entrada quaisquer tipos de arquivos
R2	Criar automaticamente um arquivo de formato jar, com nome e localização (diretório) definidos pelo usuário.
R3	Ter a capacidade de apresentar interface no formato texto.

6.1.2 .Análise

Da fase de análise, resultou o seguinte diagrama:

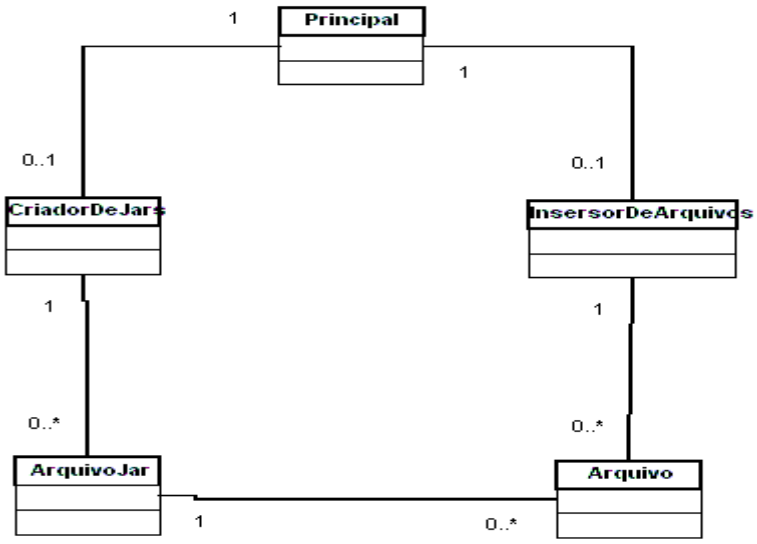

Figura 6 – Diagrama de Análise 1

A entidade principal representa um solucionador do problema (inserir arquivos em um jar). Esta entidade terá uma outra entidade para criar um arquivo jar resultante e uma terceira para inserir arquivos provenientes do programa objeto no arquivo jar resultante. Este arquivo jar resultante poderá compactar vários arquivos, podendo também estar vazio.

6.1.3. Design (Projeto)

Nesta etapa foram criadas classes de interface para tomada de dados e display de resultados. Também foi delegado a uma classe denominada GerenciadorDeJars a criação e inserção de jars, através das entidades já idealizadas na análise.

O diagrama a seguir resume o projeto no primeiro ciclo.

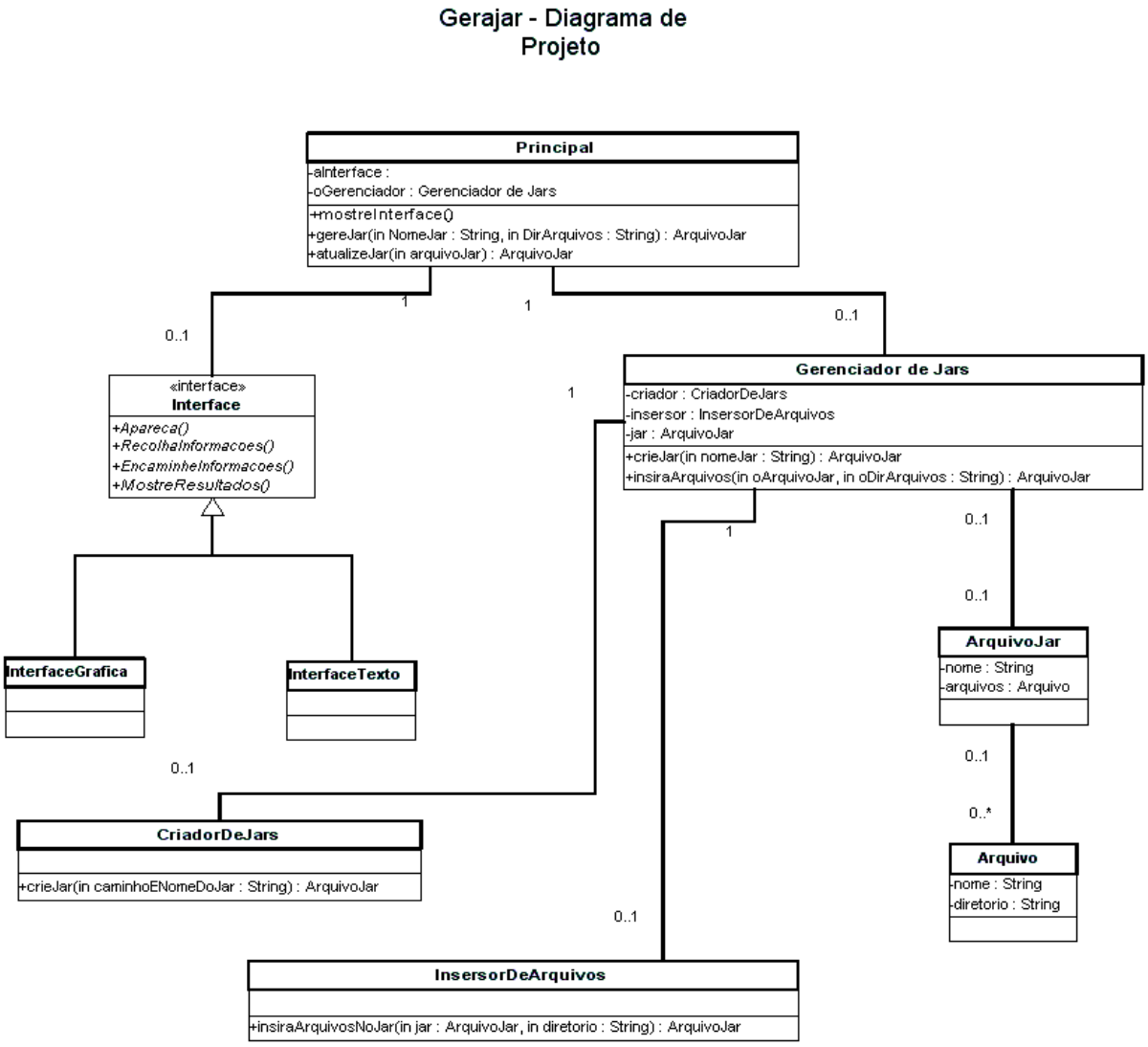

Figura 7 – Diagrama de Projeto 1

6.1.4. Implementação

Neste primeiro ciclo, a interface gráfica ficou apenas no papel, sendo implementada somente a interface texto.

As bibliotecas nativas utilizadas, além dos pacotes Java.io, foram classes do pacote Java.util.jar: A classe Java.util.jar. JarOutputStream, representa um arquivo Jar no qual poderemos inserir objetos da classe Java.util.jar.JarEntry, que por sua vez representa os arquivos que estaremos inserindo no arquivo jar resultante.

Para possibilitar a inserção dos arquivos no jar, foi necessária a conversão de arquivos para bytes, para então escrever cada arquivo como um JarEntry. Para isso foi criada uma classe chamada ByteadorDeArquivos, que toma como entrada um objeto da classe File e retorna um Array de Bytes.

Foi utilizada a classe lkj.es.LeitorDeTeclado para uma implementação de interface texto, pois mesmo a última versão do SDK do Java não apresenta uma classe nativa para ler dados do teclado.

6.1.5. Testes

O protótipo resultante deste primeiro ciclo foi um simples programa que pedia o nome do arquivo jar a ser gerado (destino), a pasta raiz das classes, o nome completo da classe main e a lista de bibliotecas que o programa objeto utilizaria. Com isso, notou-se que o número de informações requisitadas ao usuário era muito grande. Logo, concluiu-se que o Gerajar deveria procurar automaticamente as bibliotecas utilizadas e a classe principal do programa objeto.

Neste primeiro ciclo, não foi possível obter um programa que efetivamente gerasse um jar que pudesse ser executado, pois o arquivo de manifesto não foi editado corretamente. Entretanto, a inserção tanto de arquivos de extensão class quanto de arquivos de quaisquer extensões no arquivo jar destino e a compactação funcionaram perfeitamente. Também funcionou de maneira satisfatória a definição do nome e da localização do arquivo jar a ser criada. Todos os testes foram feitos através da interface texto que, dentro do proposto, também teve o funcionamento validado. Portanto, foram atingidos os objetivos do ciclo.

6.1.6. Objetivos e problemas para o próximo ciclo

No próximo ciclo o problema do arquivo de manifesto deve ser solucionado, bem como a implementação da procura automática da classe main e das bibliotecas utilizadas pelo programa objeto. Serão também implementados os tratamentos de exceções e validação dos dados fornecidos pelo usuário.

6.2. Segundo ciclo de desenvolvimento

6.2.1. Redefinição dos requisitos.

Neste ciclo acrescentaram-se alguns requisitos e descartaram-se outros, com o intuito de objetivar melhor o trabalho.

Primeiramente, foi repensada a possibilidade de uma interface gráfica e da operação por linha de comando. O foco do trabalho foi direcionado para as funcionalidades do Gerajar, deixando a interface com usuário com uma atenção secundária. Como se faz necessária uma interação com usuário, mantivemos uma interface texto. Entretanto, como estes usuários serão supostamente avançados, uma vez que serão ou programadores Java ou estudantes da área de informática, as perfumarias de uma interface gráfica são dispensáveis.

Futuramente, poderá ser implementada tanto uma interface gráfica quanto uma chamada do Gerajar por linha de comando, usando simples parâmetros (semelhante ao jar nativo do SDK).

O arquivo de manifesto será sempre criado automaticamente, logo a possibilidade de uma edição manual foi também descartada. Esta decisão de projeto foi tomada devido a seguinte justificativa: posteriormente um usuário interessado em editar o manifesto poderá ou gerar um novo jar resultante ou editar o manifesto e incluí-lo manualmente ao arquivo jar resultante, usando qualquer compactador (winzip, winrar, etc) ou mesmo até o próprio programa jar nativo do SDK.

Nos testes do primeiro ciclo, sentiu-se a necessidade de o Gerajar copiar as bibliotecas usadas pelo programa objeto para o mesmo diretório em que for gerado o arquivo jar resultante. Logo isso também foi tratado neste ciclo.

Além do já especificado, o usuário só deverá informar a pasta raiz onde se localizam todos os arquivos do programa objeto (inclusive as bibliotecas que ele utiliza), o nome e, opcionalmente, diretório onde deve ser colocado o arquivo jar resultante.

Portanto, os requisitos já redefinidos no início deste ciclo ficaram conforme a grade abaixo:

R1	Tomar como entrada quaisquer tipos de arquivos (já solucionado no primeiro ciclo)
R2	Criar automaticamente um arquivo de formato jar, com nome e localização (diretório) definidos pelo usuário. (já solucionado no primeiro ciclo)
R3	Ter a capacidade de apresentar interface no formato texto. (já solucionado no primeiro ciclo)
R4	Ter a capacidade de ser executado em qualquer sistema operacional (já solucionado no primeiro ciclo, uma vez que o programa é desenvolvido em Java)
R5	Sobrescrever arquivos jar já existentes (já solucionado no primeiro ciclo, pois se o arquivo jar resultante já existe é sobrescrito).

R6	Editar e inserir automaticamente o arquivo de manifesto no arquivo jar, definindo atributos que possibilitem a utilização de bibliotecas de classes e a classe main do programa tomado como entrada, precisando o usuário definir apenas a pasta raiz do programa objeto.
R7	Avisar o usuário de qualquer erro de execução.
R8	O usuário deverá fornecer apenas o nome do arquivo jar resultante e com a pasta raiz dos arquivos que compuserem o programa objeto.

Os requisitos R6, R7 e R8 são solucionados neste ciclo, conforme descrito nos tópicos a seguir.

6.2.2. Análise

Neste ciclo foi objetivada a inserção e edição de um manifesto de forma automática. Não somente como resultado do ciclo anterior omitindo entidades não conceituais, mas também fazendo os devidos ajustes de acordo com a evolução do trabalho, a fase de análise do segundo ciclo resultou no diagrama conceitual a seguir:

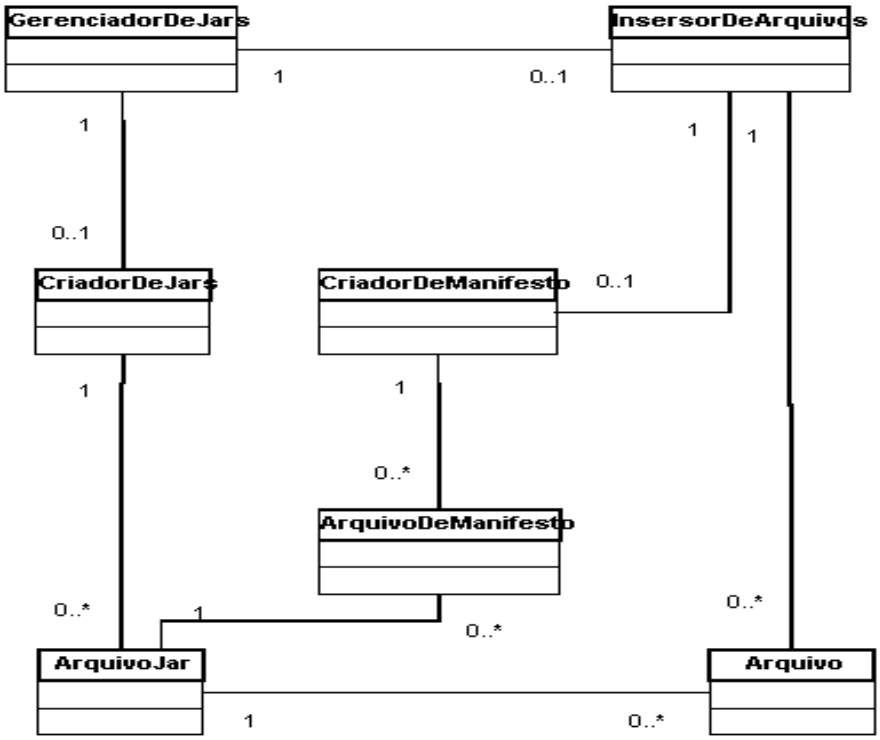

Figura 8 – Diagrama de Análise do segundo ciclo.

O diagrama acima mostra um GerenciadorDeJars que através de seu CriadorDeJars obviamente criará um ArquivoJar resultante. Neste ArquivoJar, serão inseridos Arquivos diversos através do InsersorDeArquivos. Esta entidade também possui um CriadorDeManifesto que por sua vez criará um ArquivoDeManifesto, que também será inserido no ArquivoJar. A partir daí, partiu-se para o design de um novo protótipo.

6.2.3. Projeto (Design)

Nesta fase do segundo ciclo foram feitas alterações bem significativas, pois foram modificados, eliminados e acrescentados vários métodos em várias classes bem como a inserção e remoção de classes. Alguns relacionamentos foram modificados e alguns padrões de projeto foram aplicados.

6.2.3.1. Padrões de projeto aplicados.

Os padrões de projeto aplicados neste ciclo foram *factory* e *sigleton*. A seguir, temos uma breve explanação sobre estes padrões.

6.2.3.2. *Factory*

Uma classe *factory* é responsável por fabricar instâncias de objetos que possuam uma superclasse comum. No caso do Gerajar, a entidade FabricaDeInterfaces é responsável por produzir instâncias de InterfaceComUsuario. No caso, pode ser qualquer subclasse desta entidade (ou implementação desta entidade, pois a InterfaceComUsuario na verdade é uma Interface) [LARMAN].

6.2.3.3. *Singleton*

A classe é dita *singleton* quando só pode existir apenas uma instância (objeto) desta classe. As classes FabricaDeInterface, GerenciadorDeJars, InterfaceComUsuario são singleton, pois qualquer uma delas não teria porque possuir mais de uma instância [LARMAN].

6.2.3.4. O diagrama de classes de projeto do ciclo dois.

Pode ser visualizado a seguir um diagrama de classes geral e, em seguida, os diagramas de classes dos principais módulos que chamamos de pacotes (componetes).

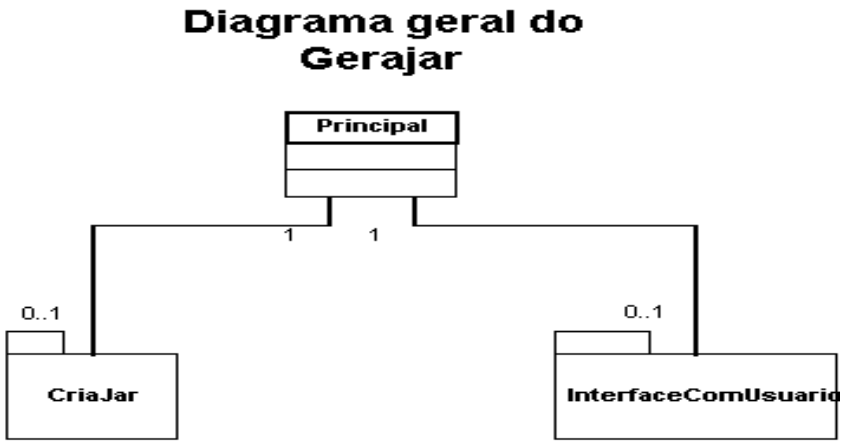

Figura 9 - Diagrama geral do gerajar

Conforme o diagrama acima, pode-se identificar a modularização do Gerajar em duas camadas: a camada de apresentação, definida pelo pacote de interface com o usuário e a camada de domínio representada pelo pacote responsável pela criação do arquivo jar resultante propriamente dita. Evidentemente não existirá uma camada de persistência, pois no projeto do Gerajar não foi identificada nenhuma necessidade de armazenamento de dados (banco de dados).

Quanto à classe Principal, trata-se apenas a classe que implementa o método Main, fazendo a integração das camadas sem maiores complicações.

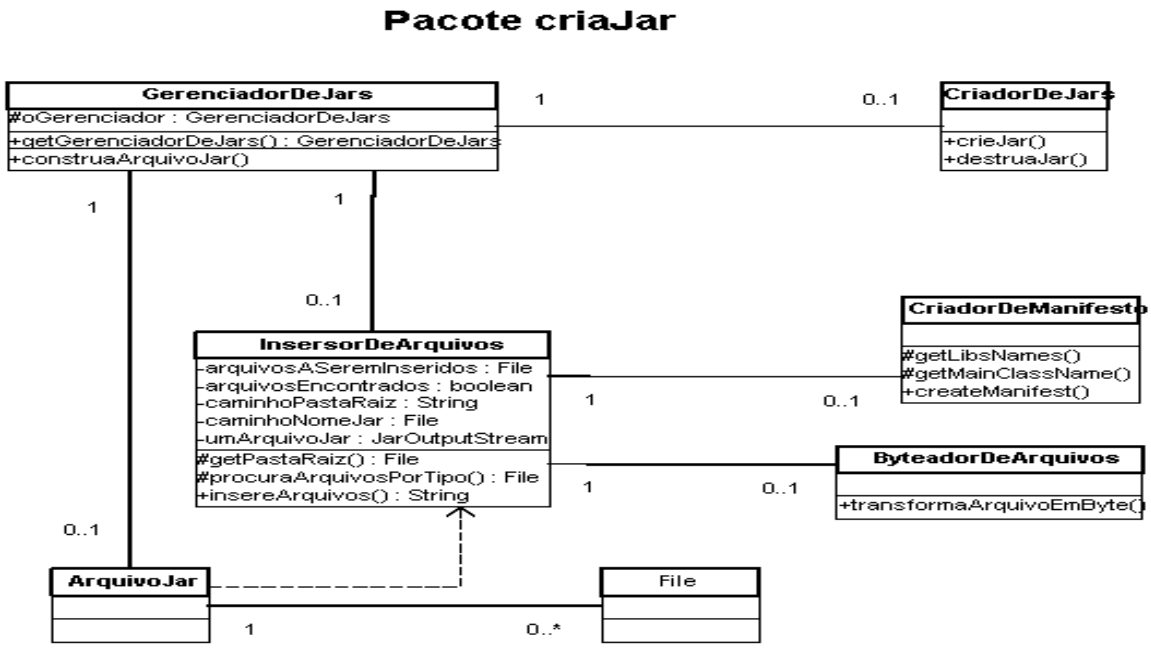

Figura 10 - Pacote responsável pela criação de jars.

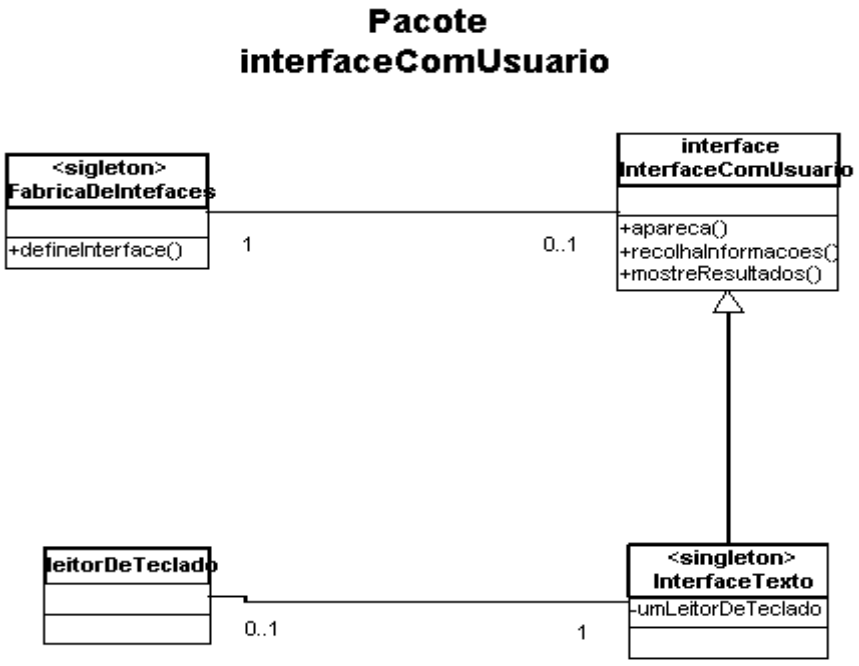

Figura 11 - Pacote responsável pela interface com usuário.

Observa-se que a classe interface gráfica ficou fora do projeto, entretanto, uma futura inclusão de uma interface gráfica no Gerajar é trivial. Não foram feitos descrições dos métodos porque estes são relativamente triviais.

6.2.4. Implementação

Na fase de implementação deste ciclo, foram protegidos alguns blocos e também inseridos desvios condicionais com o intuito de, obviamente, dar uma devida robustez à aplicação. Na ocorrência de erros, os métodos retornam

null como resposta, avisando a classe que chamou aquele método que algo ocorreu de errado. Estes erros são então transmitidos à camada de apresentação (interface com usuário) e são mostrados ao usuário de maneira inteligível e dentro de contexto. Por exemplo, se o usuário forneceu um diretório inválido como entrada, o programa apresentará uma mensagem como “Diretório Inválido” e não algo parecido com “Ocorreu erro interno”.

Na implementação de padrões, foram adotadas técnicas triviais e comumente utilizadas [LARMAN]:

6.2.4.1.Implementação do padrão *singleton*:

Para implementar o padrão *singleton*, basta colocar na classe o método construtor como privado, colocar um atributo privado x de classe (estático) como sendo um objeto da classe e criar um método que proceda da seguinte forma: se x não existir (for igual a null), chama-se o construtor da classe e coloca a instância em x e retorne; caso contrário (se x já existir), apenas retorne x. Isso assegura que existirá apenas uma instância de determinada classe.

6.2.4.2.Implementação do padrão *factory*:

No caso, foi implementado o padrão *factory* apenas na classe FabricaDeInterfaces. Não seria necessário se não tivesse uma pretensão futura da implementação de outros tipos de interface com o usuário além do modo texto. Logo, a FabricaDeInterfaces (que também é *singleton*) retornará qualquer objeto que tenha por superclasse a InterfaceComUsuario. Foi implementado apenas o modo texto, mas sendo assim existe a possibilidade da implementação de uma GUI (interface gráfica com usuário) se futuramente se desejar.

Detalhes de implementação podem ser facilmente compreendidos visualizando o código fonte.

6.2.4.3. Outras observações à cerca da implementação no segundo ciclo.

Muitos projetos de médio e grande porte contam com artefatos de descrição de métodos. Porém, por se tratar de uma pequena aplicação, a implementação foi feita de maneira direta, e uma descrição dos métodos seria o próprio código fonte. Comentários foram editados no código e a geração de documentação pelo javadoc também foi feita.

Entretanto, é importante frisar que os métodos das classes do Gerajar não foram implementados sem planejamento ou de maneira caótica, senão não se encontraria neste relatório tantas descrições de implementações de padrões de projeto. A omissão de uma descrição documentada dos métodos neste ciclo também pode ser justificada pela clareza do código fonte. Foram feitas quantas correções necessárias e todo cuidado foi tomado para que boas práticas de programação fossem aplicadas para uma edição legível do código, segundo os mais básicos conceitos de engenharia de software, tais como parametrização adequada, redução do número de variáveis globais (atributos de classes), etc.

6.2.5.Testes

Ao final da fase de implementação, já foi obtido um protótipo bem mais próximo do idealizado. Entretanto, um problema ainda foi constatado: a necessidade de as bibliotecas utilizadas pelo programa objeto estarem no mesmo diretório onde se encontrava o arquivo jar resultante.

O manifesto desta vez foi editado com sucesso, pois a classe CriadorDeManifesto confirmou a funcionalidade de procurar a classe main e de procurar as bibliotecas dentro do diretório raiz do programa objeto. Entretanto, se as bibliotecas estiverem em outro diretório, o programa não funciona corretamente.

6.2.6. Propostas e pendências para o próximo ciclo de desenvolvimento

Após os testes, foram enumeradas duas funcionalidades a serem inseridas no próximo ciclo: a capacidade de copiar as bibliotecas utilizadas pelo programa objeto para dentro do mesmo diretório do arquivo jar resultante e a compactação destas bibliotecas e do arquivo jar resultante em um único arquivo jar. Este arquivo jar teria uma execução da seguinte forma: ao executá-lo, ele pediria ao usuário o local onde deveria extrair o arquivo jar resultante e as bibliotecas. Depois disso, faria a transferência de arquivos. Poderia ou não executar o arquivo jar resultante após este procedimento.

Serão redefinidos os requisitos funcionais do Gerajar no começo do próximo ciclo.

6.3. Terceiro ciclo de desenvolvimento

Este provavelmente será o último ciclo de desenvolvimento do Gerajar, sendo que grande parte dos requisitos já foi atendida no ciclo anterior e neste ciclo objetiva-se apenas mais uma funcionalidade e um detalhe de implementação.

6.3.1. Redefinição dos requisitos:

Para este ciclo os requisitos ficam segundo descrito na grade abaixo:

R1	Tomar como entrada quaisquer tipos de arquivos (já solucionado no primeiro ciclo)
R2	Criar automaticamente um arquivo de formato jar, com nome e localização (diretório) definidos pelo usuário. (já solucionado no primeiro ciclo)
R3	Ter a capacidade de apresentar interface no formato texto. (já solucionado no primeiro ciclo)
R4	Ter a capacidade de ser executado em qualquer sistema operacional (já solucionado no primeiro ciclo, uma vez que o programa é desenvolvido em Java e utiliza somente bibliotecas nativas)
R5	Sobrescrever arquivos jar já existentes (já solucionado no primeiro ciclo, pois se o arquivo jar resultante já existe ele é sobrescrito).
R6	Editar e inserir automaticamente o arquivo de manifesto no arquivo jar, definindo atributos que possibilitem a utilização de bibliotecas de classes e a classe main do programa tomado como entrada, precisando o usuário definir apenas a pasta raiz do programa objeto (já solucionado no

	segundo ciclo).
R7	Avisar o usuário de qualquer erro de execução (já solucionado no segundo ciclo).
R8	O usuário deverá fornecer apenas o nome do arquivo jar resultante e com a pasta raiz dos arquivos que compuserem os programa objeto (já solucionado no segundo ciclo)..
R9	Gerar um único arquivo jar final contendo o arquivo jar resultante e todas as bibliotecas utilizadas pelo programa objeto.
R10	Ao executar o arquivo jar final (citado no requisito R9) o arquivo jar resultante e as bibliotecas devem ser extraídos para o diretório fornecido pelo usuário (por padrão o diretório em que se encontra o arquivo jar final).

Os requisitos R9 e R10 são solucionados neste ciclo, conforme descrito nos tópicos a seguir.

6.3.2. Análise

Na análise deste ciclo, tivemos a inserção de novas entidades, estas responsáveis pelas alterações pertinentes ao ciclo que na verdade se resume ao empacotamento das bibliotecas utilizadas pelo programa objeto, uma edição

de manifesto e um método main para esta novo pacote que obviamente será outro arquivo jar.

O diagrama conceitual a seguir mostra as novas mudanças na estrutura de classes do Gerajar.

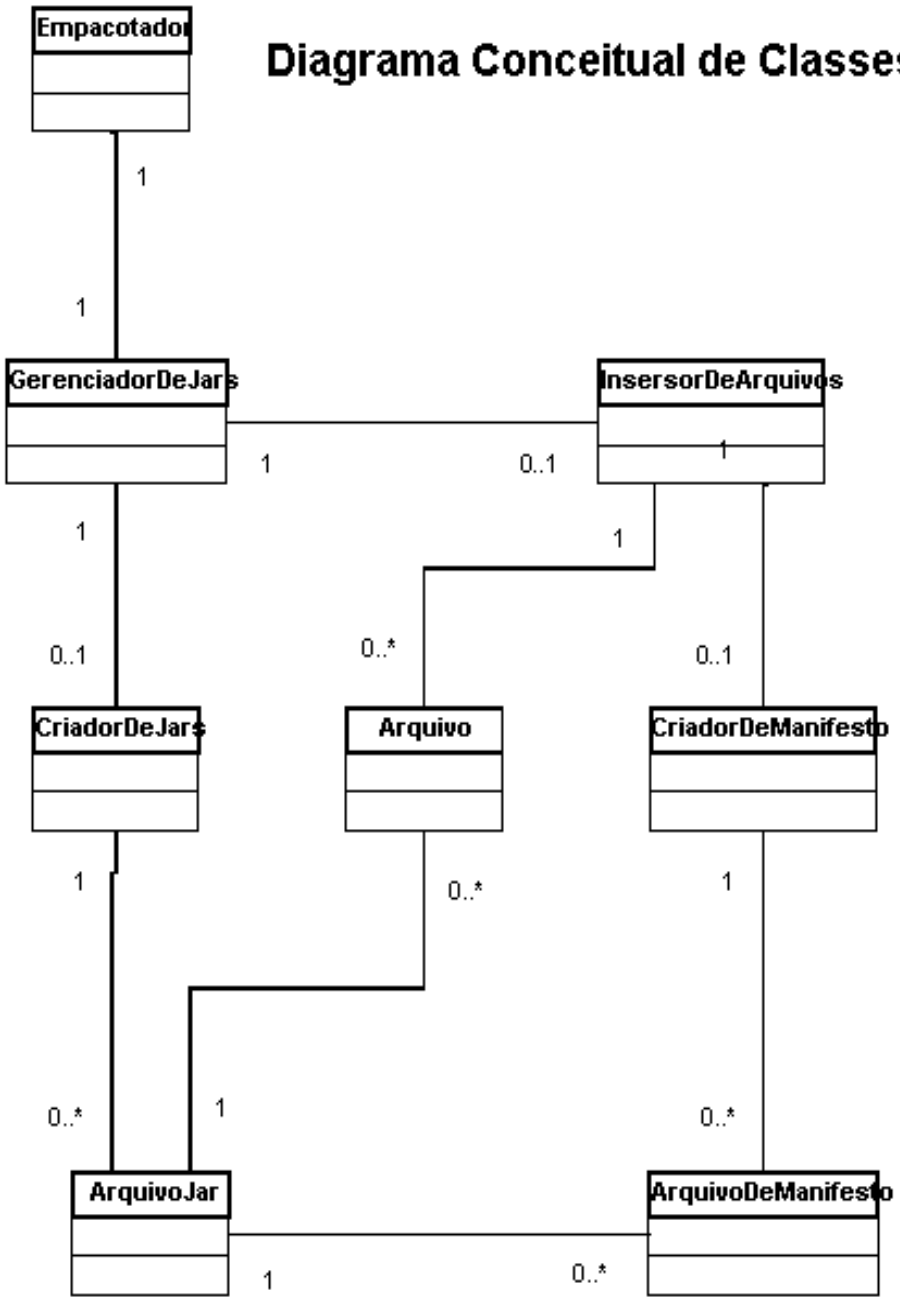

Figura 12 – Mudanças no diagrama conceitual de classes

Nota-se que agora a classe GerenciadorDeJars pertence a uma outra classe denominada Empacotador, e também servirá para a construção do arquivo jar final, que empacotará o arquivo jar resultante juntamente com as bibliotecas utilizadas pelo programa objeto.

6.3.3. Projeto

Mais uma vez as descrições de métodos foram omitidas, por se tratar de métodos triviais e de fácil compreensão através da leitura dos códigos fonte. Isso ocorreu devido a pequena escala do Gerajar, pois se trata de uma aplicação de pequeno porte.

O design de um novo protótipo é baseado, evidentemente, nas modificações feitas conceitualmente. A definição de métodos para novas classes bem como mudança nos relacionamentos entre classe é retratada nos diagramas a seguir.

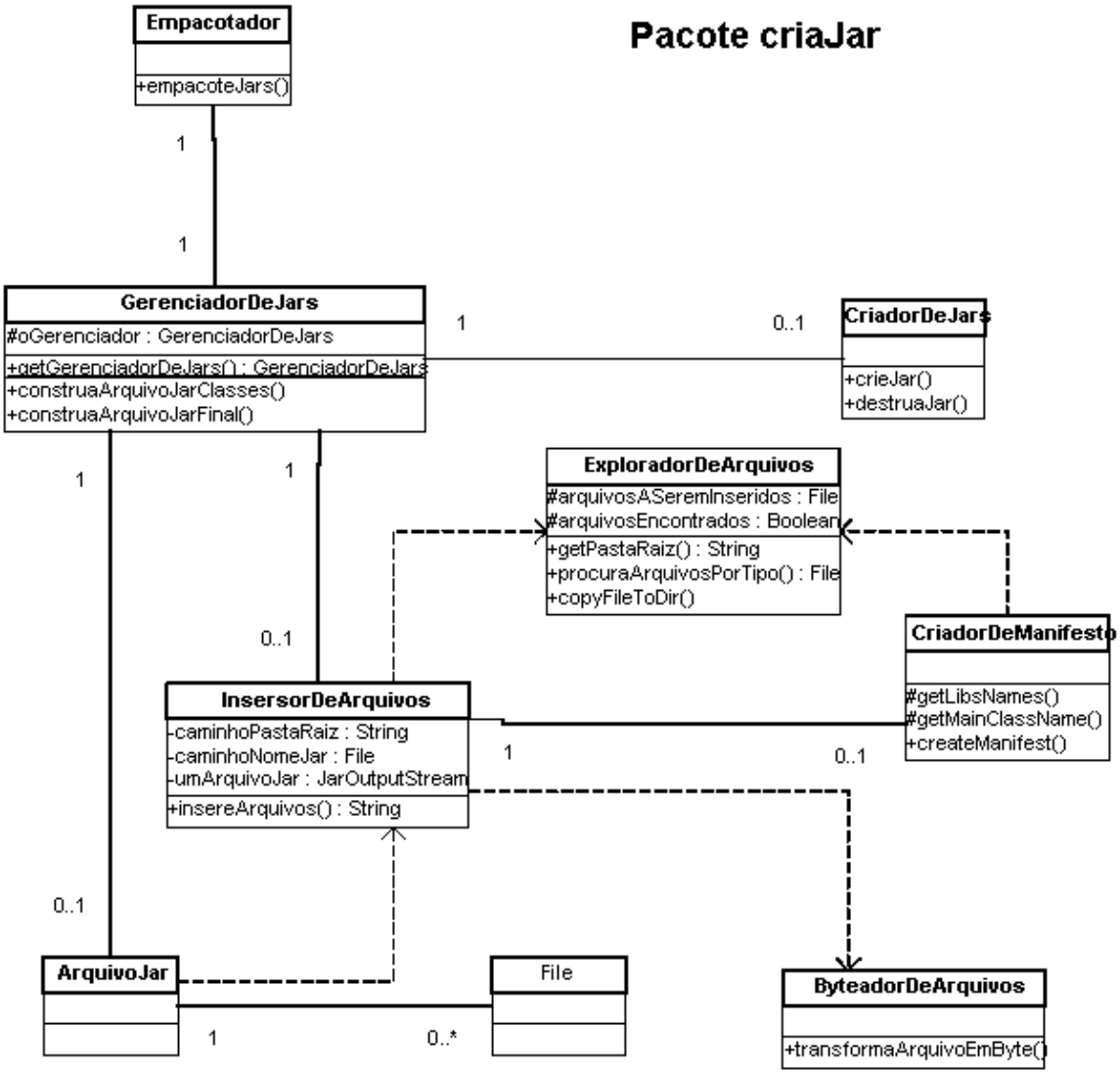

Figura 13 - Diagrama de classes do pacote criaJar no terceiro ciclo.

As mudanças significativas ficaram concentradas no pacote criaJar. Neste pacote ocorreu a inclusão da classe Empacotadora.

Para uma melhor coesão e também para diminuir o acoplamento, foi desmembrada da classe InsersorDeArquivos a classe ExploradorDeArquivos. Outra motivação para este desmembramento foi a utilização do método procuraArquivosPorTipo por parte da classe CriadorDeManifesto tanto para encontrar as bibliotecas utilizadas pelo programa objeto quanto na procura da

classe main e na cópia da bibliotecas utilizadas pelo programa objeto para o mesmo diretório do arquivo jar resultante, através do método copyFileToDir. Também ocorreram algumas mudanças nos relacionamentos entre classes.

Na interface com o usuário muito pouca coisa mudou. Abaixo o diagrama geral do Gerajar, idêntico ao do ciclo anterior.

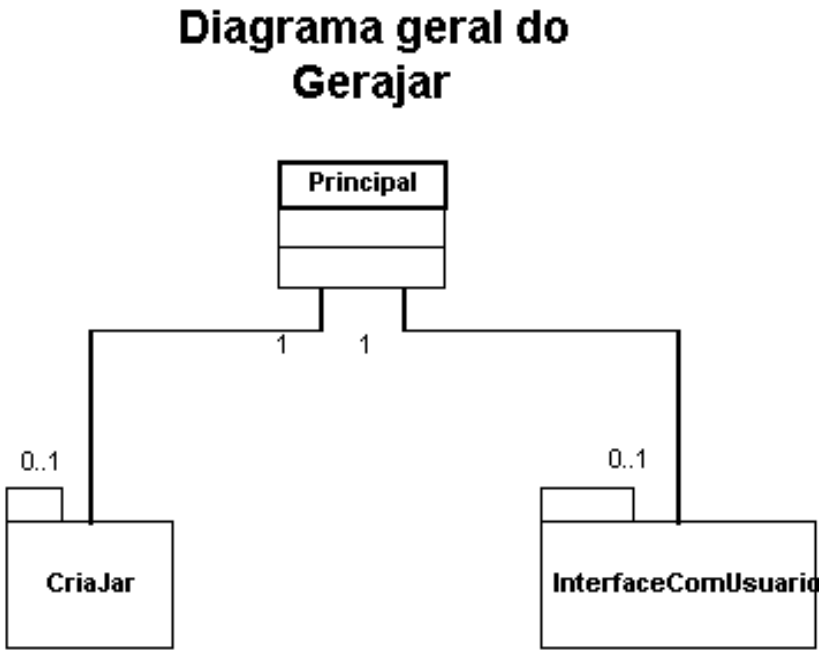

Figura 14 – Diagrama geral do terceiro ciclo.

6.3.4. Implementação

Além da implementação da classe denominada Empacotador, foi também implementado dentro do gerenciador de jars um método responsável por copiar os arquivos das bibliotecas utilizadas pelo programa objeto.

A classe principal também precisou de ajustes, pois agora não mais referencia o GerenciadorDeJars, mas o Empacotador e este, por sua vez, manda as mensagens para o GerenciadorDeJars.

A implementação de todos estes métodos é trivial e nenhuma biblioteca atípica da api do Java foi necessária. Maiores detalhes são mostrados na documentação gerada pelo javadoc e no próprio código fonte.

Já nesta fase, surgiu a idéia da implementação da opção entre gerar um arquivo jar final, que empacotaria todas as bibliotecas e também os arquivo jar resultante ou apenas gerar o arquivo jar resultante. Esse ajuste foi bem simples, uma vez que tratou apenas de fazer um desvio condicional juntando o código antigo com o novo código do método main da classe principal.

Para a implementação desta nova funcionalidade foi necessário também um pequeno ajuste na classe interface texto, uma vez que esta classe tomará um parâmetro a mais na coleta de informações. Isto foi trivial, pois a classe interface texto retorna um array de strings como resposta do método recolhaInformacoes.

Portanto, ainda na implementação foi feita uma espécie de mini-ciclo, alterando sutilmente os requisitos, a estrutura conceitual e o design do projeto.

6.3.5. Testes

Durante os testes foi confirmada a idéia de que o desenvolvimento se concluiria em apenas três ciclos, sem levar em consideração a pequena alteração feita no projeto feita durante a implementação neste terceiro ciclo.

O resultado da resolução dos requisitos a que se propunha no início do ciclo foi satisfatório, resultando em uma versão já bem funcional do Gerajar. Entretanto, ainda foi necessário algum ajuste, mas tudo dentro de condições aceitáveis e previsíveis.

Para um teste mais realista, foram tomados como programas objeto antigos projetos de diversas disciplinas cursadas durante a graduação. O Gerajar compactou e gerou arquivos de manifesto corretamente para a todas as aplicações. Apenas foram encontrados problemas com aplicações que não possuíam as bibliotecas dentro da mesma pasta raiz informada durante a execução do Gerajar, o que era previsível.

CAPÍTULO 7

7. CONSIDERAÇÕES FINAIS SOBRE O GERAJAR

7.1. Objetivos atingidos

O Gerajar inicialmente se propunha ser apenas um gerador de programas no formato jar. Porém, acabou por tornar-se um simples, mas eficiente gerador de instalador de programas também no formato jar, pois com as alterações do último ciclo lhe faltou apenas ter a opção de instalar a JVM (maquina virtual Java) para ser um instalador propriamente dito digno de competição com versões comerciais.

A correção (atributo de estar correto [SILVA]) do Gerajar foi refinada no decorrer dos ciclos, convergindo para o atendimento de todos os requisitos, estes revisados e redefinidos no início de cada ciclo de desenvolvimento. O que assegurou esta correção foi justamente esta redefinição dos requisitos em conjunto com a objetividade de atacar determinados requisitos em cada ciclo. Por sorte, alguns requisitos foram satisfeitos automaticamente, ou seja, sem ser objetivo de determinado ciclo, o que dá ainda mais mérito à metodologia empregada no decorrer do trabalho.

O fato de não possuir uma GUI (interface gráfica com usuário) não prejudicou a usabilidade nem mesmo tornou a sua interface com usuário

menos amigável. A interface texto do Gerajar é auto-explicativa e não requer qualquer experiência anterior com ferramentas do gênero (talvez não seja possível tal experiência, pois existem poucas ferramentas geradoras de instaladores de programas Java, e nenhuma similar ao Gerajar).

Pode existir uma remota possibilidade de a robustez do Gerajar apresentar alguma deficiência, entretanto, os teste feitos com diversos programas objetos mostraram uma consistência bem regular do sistema. Antagônica, porém tão importante quanto à robustez, à eficiência e a rapidez do software também se mostraram satisfatórias. Durante os testes não foi utilizado nenhum programa objeto de grande porte, o que talvez também não fosse possível, pois na pesquisa não foram encontrados programas de grande porte feitos em Java para desktop; o que se encontrou foram programas de grande porte para web, principalmente construídos sob a arquitetura J2EE. Logo, o Gerajar provavelmente tem estabilidade e eficiência suficientes para o que se propõe.

A estendibilidade, ou seja, a capacidade do software poder adaptar-se a novos requisitos ou mesmo a modificações dos requisitos [SILVA] foi preocupação constante durante o desenvolvimento do Gerajar, devido tanto a uma previsão de modificações dos requisitos quanto à preocupação de uma futura continuidade do trabalho. Disso resultaram módulos autônomos e simplicidade do projeto. Por ser uma aplicação pequena, a estendibilidade já é maximizada, mas nem por isso foi negligenciada na construção do Gerajar a aplicação de padrões de projeto, boas práticas de programação, nem tão

pouco a confecção de artefatos. Tudo isso resultou em razoáveis estendibilidade e reusabilidade.

A preocupação com a portabilidade (capacidade do software de ser executado em diferentes sistemas operacionais [LARMAN]) também esteve presente na construção do Gerajar. Como todos os artefatos utilizados foram nativos do Java, não foi encontrado nenhum problema em executar a aplicação tanto no Windows quanto no Linux. Outras plataformas não foram testadas.

7.2. Sugestões para trabalhos futuros

O Gerajar não tem a pretensão de ser um trabalho completo e sem carência de melhorias, apesar de ter conseguido atingir a grande maioria dos objetivos (requisitos) inicialmente estabelecidos. Após o último ciclo de desenvolvimento, foram vislumbradas algumas perspectivas em torno de futuros trabalhos em cima desta aplicação, com o intuito tanto de melhorar o trabalho quanto de disponibilizá-lo a usuários interessados.

A distribuição gratuita do Gerajar poderia gerar uma proposta de manutenção (pois os usuários descobririam defeitos que escaparam à última fase de testes). Uma opção seria o projeto piloto de implantação em alguma disciplina de graduação ou de curso técnico em uma área relacionada a desenvolvimento de software, na qual o Gerajar fosse disponibilizado aos alunos da disciplina para entrega de trabalhos.

Seria interessante tanto para a evolução quanto para a distribuição gratuita do Gerajar algum tipo de divulgação deste trabalho na web, principalmente para a comunidade acadêmica de áreas relacionadas à produção de software.

Apesar da razoável usabilidade do Gerajar, seria interessante o desenvolvimento de uma interface gráfica para usuários menos experientes, de forma que o Gerajar não precisasse ser executado via prompt, facilitando ainda mais a sua utilização.

Outra funcionalidade poderia ser incluída: a execução do Gerajar por linha de comando passando parâmetros, semelhante à utilização do jar nativo do SDK.

7.3 Conclusão

A engenharia de software é uma ciência que trata da construção de artefatos de software de maneira sistemática e otimizada, surgindo da necessidade de manutenção e evolução de sistemas de grande porte.

Contudo, existem algumas linhas de pensamento que desaconselham a sua utilização em pequenas aplicações, alegando que o levantamento de requisitos, análise e projeto são perda de tempo, pois a aplicação é simples demais para tanto.

Obviamente, programas como trabalhos de disciplinas de graduação de cursos da área de informática ou mesmo uma pequena aplicação

desenvolvida para uso pessoal e que jamais será atualizada por outra pessoa senão o próprio usuário dispensa qualquer artefato ou prática relacionada à engenharia de software.

Contudo, até que ponto se pode considerar uma aplicação pequena demais para ser desenvolvida de uma maneira mais metódica? Esta aplicação permanecerá sempre pequena ou existe uma possibilidade de evolução para um sistema mais complexo? Esta aplicação será atualizada por outros desenvolvedores? Se for atualizada por uma equipe, esta equipe terá condições ou disponibilidade de tempo para estudar o código fonte e então ter idéia de como funciona a aplicação? A resposta destes questionamentos é trivial, pois não só a experiência deste como de outros trabalhos analisados durante o desenvolvimento deste, aponta para a prática dos preceitos fundamentais de engenharia de software.

Desde que se objetive um produto de qualidade, a aplicação de uma metodologia para o desenvolvimento de um projeto de software não é questionável. O que deve ser decidido é qual metodologia é a mais adequada ao tamanho da equipe, disponibilidade de tempo e de ferramentas, conhecimento e experiência da equipe em relação à metodologia, entre outros.

Portanto, durante o desenvolvimento deste trabalho, a preocupação com a construção deste dentro dos preceitos da engenharia de software esteve sempre presente, o que contrariou tanto algumas teorias de que um projeto pequeno dispensa o uso destes preceitos, quanto a própria realidade da falta de metodologia na produção de software em muitas empresas.

8. REFERÊNCIAS BIBLIOGRÁFICAS

[APPLETON] APPLETON, Brad. ***Patterns and Software****: Essential Concepts and Terminology*. Disponível em http://www.enteract.com/~bradappdocpatterns-intro.html>. Acesso em maio, 2005.

[BORGES] BORGES, Ligia da Motta Silveira; FALBO, Ricardo de Almeida. **Gerência de Conhecimento sobre Processos de Software.** Disponível em <http://www.inf.ufes.br/~falbo/download/pub/BorgesWqs2001.pdf>. Acesso em maio, 2005.

[BRAGA] BRAGA, Rosana T. V; GERMANO, Fernão S. R.; MASIERO, Paulo C.; MALDONADO, José C. **Introdução aos Padrões de Software**. Disponível em <http://sugarloafplop2005.icmc.usp.br/NotasDidaticasPadroes.pdf>. Acesso em maio, 2005.

[BUSCHMANNN] BUSCHMANN, F. ***A System of Patterns*****.** Wiley, 1996

[CAPIONE] CAPIONE, Mary, WALRATH, Kathy, HUML, Alison. **The Java(TM) Tutorial:** A Short Course on the Basics (3rd Edition), Paperback, 1999.

[COLEMAN] COLEMAN, D. **Desenvolvimento Orientado a Objetos:** O Método FUSION. Editora Campus, 1996.

[CORNEL] CORNELL, Gary, HORSTMANN, Cay S. **Core Java** 2: Fundamentals, Vol. 1, Makron Books, 2000.

[DEITEL] DEITEL, Harvey M.; DEITEL, Paul J. **Java como Programar** - 3ª edição. Editora Bookman, 2001.

[GAMMA] GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. ***Design Patterns:*** *Elements of Reusable Object-Oriented Software*. Reading-MA, Addison-Wesley, 1995.

[LARMAN] LARMAN, Craing. **Utilizando UML e padrões:** Uma introdução à análise e ao projeto orientados a objeto. Editora Bookman, 2000.

[JARBUILDER] JARBUILDER. Disponível em <http://jarbuild.sourceforge.net/ >. Acesso em outubro, 2004

[JAVA] JAVA. Disponível em http://java.sun.com. Acesso em maio, de 2005.

[PRESSMAN] PRESSMAN, R. **Engenharia de Software**. Editora Makron Books, 1995.

[SILVA] SILVA, Ricardo Pereira. Disponível em <http://www.inf.ufsc.br/~ricardo/>. Acesso em abril, 2005

[ZEROG] ZEROG. Disponível em <http://www.zerog.com/products_ia.shtml >. Acesso em março, 2005

DESENVOLVENDO DE UMA SOLUÇÃO PARA INSTALAÇÃO DE PROGRAMAS EM JAVA PARA DESKTOP

Elias Alexandrino de Souza Júnior e Fabiano Chiqueti, graduandos do curso de Ciências da Computação da Universidade Federal de Santa Catarina.

PALAVRAS-CHAVE: jar, java, instalador, programação, engenharia de software.

RESUMO: A preparação do produto final da produção de software para ser entregue ao cliente é muitas vezes desconsiderada ou até esquecida no início de um projeto, levando-se em consideração a sua simplicidade quando comparada ao restante do processo de desenvolvimento. Entretanto, esta preparação nem sempre se da de maneira trivial. No que diz respeito a aplicações desenvolvidas em Java, a entrega de inúmeros arquivos *.class* é no mínimo pouco prática. O empacotamento desses arquivos em um único arquivo .zip ou .jar não resolve totalmente o problema, pois para aplicações *desktop* é necessária uma maneira de executar o programa localmente. Este texto propõe uma ferramenta geradora de instaladores de programas feitos em Java que possa solucionar este problema.

ABSTRACT: *The preparation of the result of the production of software to be delivers to the customer is often not considered or forgotten in the beginning a project, taking itself in consideration the its simplicity when compared with the remain it development process. However, this preparation is not so trivial. About the applications developed in Java, the delivery of innumerable archives class is not so practical. The wrapping up of these archives in an only archive zip or jar does not entirely solve the problem, therefore for desktop applications is necessary a way to execute the program in the local machine. This article considers a generator of installers of Java programs tool, which can solve this problem.*

1. INTRODUÇÃO

O desenvolvimento de software sempre foi uma atividade em ascensão e expansão no cenário mundial. A necessidade da implementação da mais diversa gama de novas soluções, migração de sistemas velhos para sistemas novos, manutenção e extensão de soluções já existentes é somente uma das causas deste crescimento. Uma conseqüência deste cenário é o surgimento de novas tecnologias e entre elas aparece o Java.

Como qualquer projeto, uma aplicação em Java necessita ser entregue ao cliente em um formato que facilite ao máximo a sua instalação e utilização. O Gerajar é uma ferramenta que se propunha a gerar um arquivo no formato jar tomando como entrada arquivos compilados de unidades Java (.class) e que através deste arquivo de extensão jar fosse possível executar o programa desenvolvido, sem maiores complicações. Em uma segunda instância, o Gerajar tornar-se-ia um software gerador de instalador, similar a programas como INSTALLANYWHERE, INSTALLSHIELD e outros, porém bem mais simplificada.

2. JUSTIFICATIVA PARA O DESENVOLVIMENTO DO GERAJAR

Conforme uma pesquisa feita sobre o estado da arte no que diz respeito a programas geradores de instaladores próprios para projetos desenvolvidos em Java, foi constatado que existem boas soluções no mercado, porém todas pagas, deixando a carência de uma aplicação de qualidade e de utilização gratuita.

A aplicação desenvolvida não tem a pretensão de concorrência com as soluções à venda no mercado, pois isso demandaria uma equipe maior e mais tempo

para desenvolvimento, porém deve ser uma boa alternativa na geração de um único arquivo para instalação ou mesmo para ser executado.

3. A TECNOLOGIA JAVA APLICADA NO DESENVOLVIMENTO DO GERAJAR.

Nos primeiros ciclos de desenvolvimento, o Gerajar se propôs apenas a ser um gerador de arquivos de extensão jar executáveis, empacotando as classes do programa objeto, ou seja, da aplicação a qual o Gerajar estiver tomando como entrada.

Um arquivo de extensão jar é uma versão mais sofisticada do popular arquivo de extensão zip (arquivo compactado). Além de compactar quantos arquivos for necessário mantendo a estrutura de diretórios, um arquivo de extensão jar pode servir como um similar ao executável no Windows, porém multi-plataforma [JAVA].

Para tornar um arquivo jar um programa “executável”, basta inserir neste jar todos os arquivos compilados de um programa desenvolvido em Java (.class). Também deve ser inserido um diretório META-INF contendo um arquivo de manifesto (manifest.mf). Definindo devidamente neste arquivo de manifesto as bibliotecas que o programa utiliza (extensões), classe principal (main), etc, o jar se tornará então uma aplicação “executável” [JAVA].

Presente no kit de desenvolvimento do Java (SDK) desde a versão 1.1, este formato é muito utilizado para empacotamento de bibliotecas de classes, de componentes e até mesmo de frameworks. Portanto, já está mais do que agregado à cultura Java, sendo uma tecnologia padrão neste meio [JAVA].

As bibliotecas nativas utilizadas, além dos pacotes Java.io, foram classes do pacote Java.util.jar, próprio para manipulação de arquivos de extensão .jar: A classe Java.util.jar. JarOutputStream, representa um arquivo Jar no qual poderemos inserir

objetos da classe Java.util.jar.JarEntry, que por sua vez representa os arquivos que estaremos inserindo no arquivo jar resultante, podendo ser estes arquivos de qualquer extensão além de .class, .java, etc.

4. METODOLOGIA EMPREGADA

Este trabalho trata-se de um projeto orientado a objetos. Para seu desenvolvimento, foi adotada uma adaptação da metodologia Fusion (COLEMAN, 1996), utilizando um modelo espiral da seguinte forma: O projeto é desenvolvido em ciclos de desenvolvimento. Cada ciclo deverá solucionar um ou mais requisitos funcionais do sistema e deverá resultar em um protótipo que será então testado.

4.1. A Metodologia Fusion - Justificativa

Fusion é um método de desenvolvimento de software orientado a objeto de segunda geração. Foi concebido em 1992 por um grupo de pesquisadores da HP. Segundo COLEMAN, Fusion integra e estende os principais dispositivos das abordagens orientadas a objeto de mais sucesso no mercado: OMT, Booch, CRL e Objetory. Fusion é composto de três fases: análise, projeto e implementação; cada fase é descrita, bem como seus modelos e notações.

Mas porque utilizar uma metodologia já ultrapassada como a Fusion? Atualmente, as metodologias utilizadas para desenvolvimento de software em Java, principalmente na arquitetura J2EE, utilizam casos de uso (*use cases)* para o desenvolvimento, fragmentando em pequenos pedaços. No caso do Gerajar, apenas um caso de uso foi definido, que é a geração do jar executável ou, nas versões finais, do instalador. Entretanto, esse caso de uso poderia ser fragmentado em pedaços que

nada mais seriam do que os próprios métodos das classes do sistema. Fazendo isso, caímos na velha metodologia Fusion, com as operações da fase de análise e os métodos da fase de *design.*

4.2. O Modelo Espiral - Justificativa

Segundo SILVA, o modelo espiral inclui entre as etapas do ciclo tradicional (análise, projeto, implementação, teste e manutenção) as atividades abaixo:

- Planejamento da próxima etapa
- Determinação de objetivos, alternativas e limitações.
- Avaliação das alternativas, identificação e solução de riscos.
- Prototipação (simulações e avaliações)

Portanto, embora longe da perfeição, o modelo espiral preenche lacunas de modelos de ciclo vida de software anteriores, como o *code and fix*, cascata e em V.

A escolha deste modelo de ciclo de vida foi feita por sugestão do primeiro orientador do projeto, professor Leandro Komosinsky. A prototipação mostrou-se como a melhor forma de testar e reavaliar os requisitos, determinando o planejamento para cada ciclo posterior. Apesar da escolha não ter sido fundamentada em experiência própria ou teórica, foi bem sugerida e o projeto adaptou-se bem a ela.

4.3 Padrões de Projeto

Um padrão descreve uma solução para um problema que ocorre freqüentemente durante o desenvolvimento de software, podendo ser considerado como um par “problema/solução” [BUSCHMANN]. Projetistas familiarizados com

certos padrões podem aplicá-los imediatamente a problemas de projeto, sem ter que redescobri-los [GAMMA].

Em algumas situações do projeto, foi bem a calhar a aplicação de padrões como o *Singleton, factory,* e outros, tanto como forma de prevenção de possíveis erros de execução ou prevenção de erros em utilizações posteriores por outras aplicações.

5. OBJETIVOS ATINGIDOS

O Gerajar inicialmente se propunha ser apenas um gerador de programas no formato jar. Porém, acabou por tornar-se um simples, mas eficiente gerador de instalador de programas também no formato jar, pois com as alterações do último ciclo lhe faltou apenas ter a opção de instalar a JVM (maquina virtual Java) para ser um instalador propriamente dito.

O resultado da resolução dos requisitos a que se propunha no início do projeto foi satisfatório, resultando em uma versão já bem funcional do Gerajar. Entretanto, ainda foi necessário algum ajuste, mas tudo dentro de condições aceitáveis e previsíveis.

Para um teste mais realista, foram tomados como programas objeto antigos projetos de diversas disciplinas cursadas durante a graduação. O Gerajar compactou e gerou arquivos de manifesto corretamente para a todas as aplicações. Apenas foram encontrados problemas com aplicações que não possuíam as bibliotecas dentro da mesma pasta raiz informada durante a execução do Gerajar, o que era previsível.

Pode existir uma remota possibilidade de a robustez do Gerajar apresentar alguma deficiência, entretanto, os testes feitos com diversos programas objetos mostraram uma consistência bem regular do sistema. Antagônica, porém tão

importante quanto à robustez, à eficiência e a rapidez do software também se mostraram satisfatórias. Durante os testes não foi utilizado nenhum programa objeto de grande porte, o que talvez também não fosse possível, pois na pesquisa não foram encontrados programas de grande porte feitos em Java para desktop; o que se encontrou foram programas de grande porte para web, principalmente construídos sob a arquitetura J2EE. Logo, o Gerajar parece ter estabilidade e eficiência suficientes para o que se propõe.

A estendibilidade, ou seja, a capacidade do software poder adaptar-se a novos requisitos ou mesmo a modificações dos requisitos [SILVA] foi preocupação constante durante o desenvolvimento do Gerajar, devido tanto a uma previsão de modificações dos requisitos quanto à preocupação de uma futura continuidade do trabalho. Disso resultaram módulos autônomos e simplicidade do projeto. Por ser uma aplicação pequena, a estendibilidade já é maximizada, mas nem por isso foi negligenciada na construção do Gerajar a aplicação de padrões de projeto, boas práticas de programação, nem tão pouco a confecção de artefatos.

A preocupação com a portabilidade (capacidade do software de ser executado em diferentes sistemas operacionais [LARMAN]) também esteve presente na construção do Gerajar. Como todos os artefatos utilizados foram nativos do Java, não foi encontrado nenhum problema em executar a aplicação tanto no Windows quanto no Linux. Outras plataformas não foram testadas.

6. SUGESTÕES PARA TRABALHOS FUTUROS

O Gerajar não tem a pretensão de ser um trabalho completo e sem carência de melhorias, apesar de ter conseguido atingir a grande maioria dos objetivos

(requisitos) inicialmente estabelecidos. Após o último ciclo de desenvolvimento, foram vislumbradas algumas perspectivas em torno de futuros trabalhos em cima desta aplicação, com o intuito tanto de melhorar o trabalho quanto de disponibilizá-lo a usuários interessados.

A distribuição gratuita do Gerajar poderia gerar uma proposta de manutenção (pois os usuários descobririam defeitos que escaparam à última fase de testes). Uma opção seria o projeto piloto de implantação em alguma disciplina de graduação ou de curso técnico em uma área relacionada a desenvolvimento de software, na qual o Gerajar fosse disponibilizado aos alunos da disciplina para entrega de trabalhos.

Seria interessante tanto para a evolução quanto para a distribuição gratuita do Gerajar algum tipo de divulgação deste trabalho na web, principalmente para a comunidade acadêmica de áreas relacionadas à produção de software.

Apesar da razoável usabilidade do Gerajar, seria interessante o desenvolvimento de uma interface gráfica para usuários menos experientes, de forma que o Gerajar não precisasse ser executado via prompt, facilitando ainda mais a sua utilização.

Outra funcionalidade poderia ser incluida: a execução do Gerajar por linha de comando passando parâmetros, semelhante à utilização do jar nativo do SDK.

7. CONCLUSÃO

A engenharia de software é uma ciência que trata da construção de artefatos de software de maneira sistemática e otimizada, surgindo da necessidade de manutenção e evolução de sistemas de grande porte.

Contudo, existem algumas linhas de pensamento que desaconselham a sua utilização em pequenas aplicações, alegando que o levantamento de requisitos, análise e projeto são perda de tempo, pois a aplicação é simples demais para tanto.

Obviamente, programas como trabalhos de disciplinas de graduação de cursos da área de informática ou mesmo uma pequena aplicação desenvolvida para uso pessoal e que jamais será atualizada por outra pessoa senão o próprio usuário dispensa qualquer artefato ou prática relacionada à engenharia de software.

Contudo, até que ponto se pode considerar uma aplicação pequena demais para ser desenvolvida de uma maneira mais metódica? Esta aplicação permanecerá sempre pequena ou existe uma possibilidade de evolução para um sistema mais complexo? Esta aplicação será atualizada por outros desenvolvedores? Se for atualizada por uma equipe, esta equipe terá condições ou disponibilidade de tempo para estudar o código fonte e então ter idéia de como funciona a aplicação? A resposta destes questionamentos é trivial, pois não só a experiência deste como de outros trabalhos analisados durante o desenvolvimento deste, aponta para a prática dos preceitos fundamentais de engenharia de software.

Desde que se objetive um produto de qualidade, a aplicação de uma metodologia para o desenvolvimento de um projeto de software não é questionável. O que deve ser decidido é qual metodologia é a mais adequada ao tamanho da equipe, disponibilidade de tempo e de ferramentas, conhecimento e experiência da equipe em relação à metodologia, entre outros.

Portanto, durante o desenvolvimento deste trabalho, a preocupação com a construção deste dentro dos preceitos da engenharia de software esteve sempre presente, o que contrariou tanto algumas teorias de que um projeto pequeno dispensa o uso destes preceitos, quanto à própria realidade da falta de metodologia na produção de software em muitas empresas.