UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE CIÊNCIAS EXATAS E DA NATUREZA
DEPARTAMENTO DE INFORMÁTICA

TRABALHO DE CONCLUSÃO DE CURSO

CLODOALDO BRASILINO LEITE NETO

GINGAINSTANCING: UMA FERRAMENTA PARA GERAÇÃO DE PRODUTOS DERIVADOS DO MIDDLEWARE DO SISTEMA BRASILEIRO DE TELEVISÃO DIGITAL

João Pessoa
2010

Clodoaldo Brasilino Leite Neto

GINGAINSTANCING: UMA FERRAMENTA PARA GERAÇÃO DE PRODUTOS DERIVADOS DO MIDDLEWARE DO SISTEMA BRASILEIRO DE TELEVISÃO DIGITAL

Monografia apresentada como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação, pelo curso de Ciência da Computação da Universidade Federal da Paraíba.

Orientadora: Profa. Dra. Tatiana Aires Tavares

João Pessoa
2010

Clodoaldo Brasilino Leite Neto

GINGAINSTANCING: UMA FERRAMENTA PARA GERAÇÃO DE PRODUTOS DERIVADOS DO MIDDLEWARE DO SISTEMA BRASILEIRO DE TELEVISÃO DIGITAL

Monografia apresentada como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação, pelo curso de Ciência da Computação da Universidade Federal da Paraíba.

Orientadora: Profa. Dra. Tatiana Aires Tavares

COMISSÃO EXAMINADORA

_____

Profa. Dra. Tatiana Aires Tavares
Universidade Federal da Paraíba

_____

Prof. Dr. Guido Lemos de Souza Filho
Universidade Federal da Paraíba

_____

Prof. Raoni Kulesza
Universidade Federal da Paraíba

João Pessoa, ____ de _____ de _____

RESUMO

Com a necessidade de uma ferramenta para automatização da geração de configurações necessárias pelo ambiente de execução do middleware Ginga, foi construído um software para conquistar essa automatização. Neste trabalho é gerenciada a engenharia de aplicação da linha de produto de software do middleware. O software construído foi elaborado através de frameworks de modelos e uma linguagem de descrição de arquitetura, a Fractal ADL. Ela permite que arquiteturas sejam descritas, tanto de referência quanto de aplicação. Com um desenvolvimento baseado em componentes, e o ambiente FlexCM de execução, com ligação dinâmica desses componentes, é permitido o uso de tal linguagem de descrição de arquitetura, com uma certa adaptação na arquitetura de referência para que possa ser modelada de acordo com um modelo de características. A proposta de interface dada para o software foi de um wizard construído em três etapas coesas, de maneira que etapas distintas trabalham em processos distintos da engenharia de aplicação.

ABSTRACT

With the need for a tool for automating the generation of configurations needed by the runtime environment of middleware Ginga, we built a software automation to win this. In this software is managed the application engineering of the middleware software product line. The software built was elaborated through modeling frameworks and an architecture description language, the Fractal ADL. It permits the architecture descriptions; both reference architecture and application architecture. With a component-based development, and the FlexCM runtime environment, with dynamic linkage of these components, it is allowed the use of such architecture description language, with a certain adaptation in the reference one so it can be modeled according to a feature model. The interface proposed to the software was a wizard built in three cohesive stages, so that these stages work in distinct processes of the application engineering.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

API: Application Programming Interface

CSP: Concurrent Specification Programming

DBC: Desenvolvimento Baseado em Componentes

DLL: Dynamic Link Library

EMF: Eclipse Modelling Framework

F4E: Fractal for Eclipse

FSM: Finite State Machines

FlexCM: Flexible Component Model

GingaCDN: Ginga Code Development Network

GMF: Graphical Modelling Framework

GUID: Globally Unique Identifier

LAViD: Laboratório de Aplicações de Vídeo Digital

LDA: Linguagem de Descrição de Arquitetura

LPS: Linha de Produtos de Software

MVC: Model-View-Controller

POP: Post Office Protocol

POP3: Post Office Protocol version 3

SBTVD: Sistema Brasileiro de Televisão Digital

SGBD: Sistema de Gerência de Banco de Dados

SPL: Software Product Line

TVD: Televisão Digital

UFPB: Universidade Federal da Paraíba

XML: Extensible Markup Language

SUMÁRIO

1. INTRODUÇÃO

O processo de produção do middleware brasileiro de TVD Ginga possui alguns desafios importantes, como a implantação de uma linha de produto de software em sua estrutura já implantada de desenvolvimento baseado em componentes (através do FlexCM – um modelo de componentes para sistemas embarcados adaptativos), afim de facilitar e tornar gerenciável sua customização em massa.

O projeto Ginga Code Development Network do Laboratório de Aplicações em Vídeo Digital da Universidade Federal da Paraíba – laboratório esse que é um dos colaboradores com o desenvolvimento do padrão de TVD brasileiro – trouxe consigo o desafio de construir uma rede colaborativa de desenvolvedores do middleware Ginga, de forma a facilitar e centralizar a gerência da produção do middleware.

A rede GingaCDN possui a necessidade de uma ferramenta de gerencia de ativos, controle de versão e derivação de produtos Ginga de forma automatizada e intuitiva para usuários do portal GingaCDN pois até o presente momento a geração de derivações do middleware é feita manualmente através de cópias dos arquivos binários das implementações e geração manual do registro de ativos e arquitetura do middleware para o FlexCM.

1.1. OBJETIVO

Este trabalho monográfico visa propor uma solução de software que resolverá a questão de geração de produtos derivados da linha de produto do middleware, sendo automatizado o processo de cópia e geração dos arquivos de configuração de registro de ativos e arquitetura do sistema e também propor uma interface amigável para os usuários da rede GingaCDN, de modo com que traga produtividade à eles.

2. FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, abordamos as necessidades teóricas que fundamentam a ferramenta. Abordamos o desenvolvimento baseado em componentes, a engenharia de linhas de produtos de software e linguagens de descrição de arquitetura.

2.1. DESENVOLVIMENTO BASEADO EM COMPONENTES

2.1.1 COMPONENTES

Os Componentes de Software, segundo [18], possuem propriedades características que são:

- Ser uma unidade independente de implantação;
- Ser uma unidade de composição terceirizada;
- Não possuir nenhum estado observável externamente.

Para satisfazer essas propriedades, temos sérias implicações na construção de componentes. Primeiramente, para fazer com que o componente seja independente de implantação, ele precisa ser separado do seu ambiente e de outros componentes. Também, o componente deve encapsular seus recursos constituintes. Por ser uma unidade de implantação, o componente nunca será implantado parcialmente, pois ele funciona como uma única peça.

Para motivos de esclarecimento neste contexto, dizemos que um componente terceiro é um que não se espera ter acesso aos detalhes de construção de todos os componentes envolvidos.

Para que um componente seja composto com outros por um terceiro, ele precisa ser, além de completo, pela definição do item anterior, também possuir especificações bem definidas de suas interfaces, ou seja, suas maneiras de interagir com o seu ambiente. A definição de interface de um componente será feita mais adiante.

Não possuir nenhum estado observável externamente significa que, para o ambiente ou componentes terceiros, um componente não é distinto à cópias deles individualmente.

Um componente pode ser carregado e ativado em um sistema particular. Podemos observar que, já que o componente não possui nenhum estado observável externamente, ele é caracteristicamente *stateless*. Por isso, notamos que os

componentes não precisam possuir múltiplas cópias em uma execução particular, já que nenhuma diferença aparente existirá entre as cópias. Logo, não faz sentido falar sobre número de cópias de um componente sobre uma certa execução.

Um componente também pode possuir atributos que são também uma característica de uma classe (também pode ser chamado de campo). A utilização de atributos na configuração de um componente pode fazer um caso particular em que duas instancias de um componente se comporte de maneira diferente, por causa dos atributos de inicialização utilizados nesse.

2.1.2 INTERFACES

Interfaces são maneiras pela qual componentes se conectam. Tecnicamente, uma interface é um conjunto de operações que podem ser invocadas por clientes. As semânticas de cada operação é especificada, e essa especificação assume dois papéis, já que servem tanto provedores implementando a interface quanto clientes utilizando-as. Como, na criação de componentes, provedores e clientes não estão cientes uns do outros, a especificação de interface se torna o mediador que permite as duas partes trabalharem juntas.

Um componente pode tanto prover diretamente uma interface ou implementar objetos que, se forem disponíveis aos clientes, provêem interfaces. Interfaces diretamente providas por um componente correspondem a interfaces procedurais de bibliotecas tradicionais. Pode haver interfaces providas indiretamente por objetos que são disponibilizadas pelo componente. Tais interfaces indiretamente implementadas correspondem a interfaces de objetos. Componentes podem ser especificados separadamente das interfaces e das suas especificações. Uma especificação de componentes nomeia todas as interfaces que um componente deve aderir e adiciona propriedades especificas de componente.

Também é possível focar na interação entre componentes utilizando certas interfaces. Isto é feito escrevendo especificações que restringem dados através de um conjunto de interfaces. Isto é. Interfaces aparecem como regras nos pontos finais de uma especificação de interação e nenhuma interface pode aparecer em mais de uma especificação de interação.

2.1.3 INTERFACES COMO CONTRATOS

Uma maneira útil de ver especificações de interfaces é como contratos entre um cliente de interface e um provedor de implementação dela. O contrato estabelece o que o cliente precisa fazer para usar a interface. Também mostra a maneira que o provedor tem que implementar para atender a maneira que os serviços são providos pela interface.

No nível de uma operação individual de uma interface, há um método de especificação contratual particularmente popular. Os dois lados do contrato devem ser capturados pela especificação de pré e pós-condições para a operação. O cliente deve estabelecer uma pré-condição antes de chamar a operação, e o provedor pode confiar na pré-condição sendo atendida quando a operação for chamada.O provedor deve estabelecer uma pós-condição antes de retornar ao cliente e cliente deve confiar na pós-condição sendo atendidas quando a chamada à operação retornar. Pré e pós-condições não são a única maneira de formular contratos (adicionar uma especificação a uma interface). Adicionalmente, com pré e pós-condições apenas não é possível especificar todos os aspectos de uma interface.

2.2. ENGENHARIA DE LINHAS DE PRODUTOS DE SOFTWARE

A engenharia de linhas de produtos de software definida por [16] pode ser entendida como um paradigma para desenvolver aplicações de software(sistemas de software intensivos e produtos de software) usando plataformas e customização em massa. Uma plataforma é qualquer base de tecnologias sob a qual outras tecnologias ou processos são construídos.

No caso em questão (que é uma plataforma de software), podemos ainda definir, segundo [14] de uma mais específica:

"Uma plataforma de software é um conjunto de subsistemas e interfaces de software que formam uma estrutura comum da qual um conjunto de produtos derivativos podem ser eficientemente desenvolvidos e produzidos."

Desenvolver aplicações usando plataformas significa planejar proativamente para reuso, construir partes reusáveis, e reusar o que foi construído para reuso. Construir aplicações para customização em massa significa empregar o conceito de variabilidade gerenciável, por exemplo os aspectos comuns e diferenças entre

aplicações (em termos de artefatos de requisitos, arquitetura, componentes e testes) da linha de produto devem ser modelados de uma maneira comum.

Variabilidade gerenciável tem um grande impacto na maneira que o software é desenvolvido, estendido, e mantido. Comumente, para aqueles que entendem como uma peça de software funciona, também é fácil trocá-la e adaptá-la para caber num novo propósito. Porém, tais mudanças também corrompem a estrutura original do software e embaraça aspectos de qualidade como manutenabilidade e legibilidade. Para ser capaz de se dar com adaptações de uma maneira gerenciável, eles devem ser realizados de uma maneira reproduzível. A abundância de possibilidades para onde uma peça de software pode se adaptar deve ser restrita a apenas onde faz sentido colocá-la. Mais que disciplinas de engenharia, engenharia de linha de produto de software se dá mais com maneiras de restringir variações de uma maneira gerenciável.

O paradigma de engenharia de linha de produto de software se separa em dois processos: engenharia de domínio e engenharia de aplicação. Mas antes de abordá-los, devemos falar primeiramente dos princípios de variabilidade. A figura 2.1 demonstra como é dado o fluxo do processo de produção e dos ativos em uma linha de produto de software.

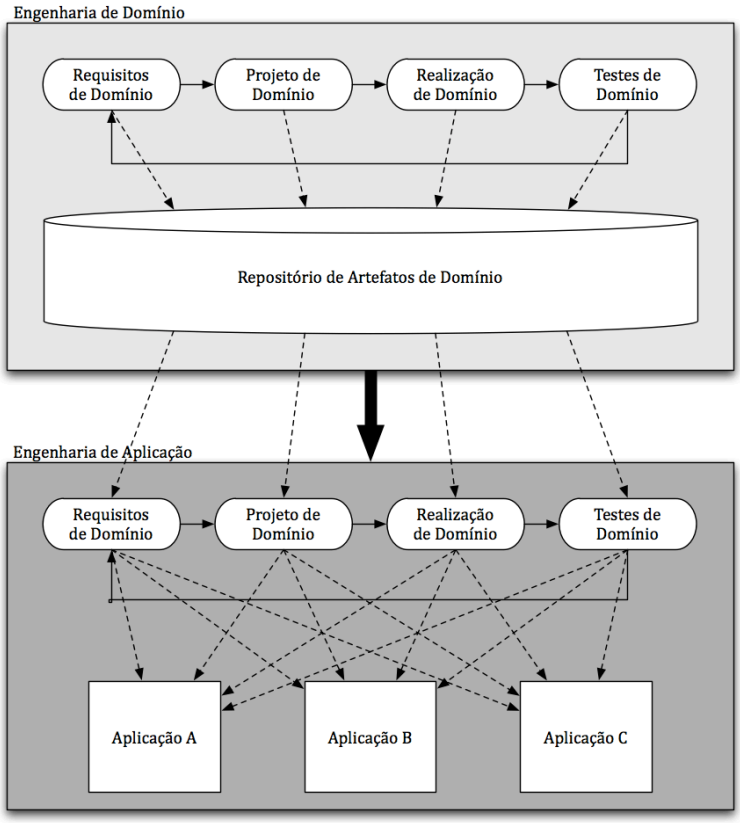

FIGURA 2.1 – Paradigma da Engenharia de LPS [16]

2.2.1 PRINCÍPIOS DE VARIABILIDADE

Refere-se à soma de todas as atividades interessadas na identificação e documentação da variabilidade como *definir variabilidade*. A variabilidade é definida durante a engenharia de domínio e explorada durante a engenharia de aplicação pelas amarrações às variantes apropriadas.

Definir e explorar variabilidade através dos diferentes estágios do ciclo de vida de uma linha de produto de software é suportado pelo conceito de *variabilidade gerenciada*. Este conceito basicamente engloba as seguintes questões:

- Apoio de atividades interessadas em definir variabilidades;
- Gerência de artefatos de variabilidade;
- Apoio de atividades interessadas em resolver variabilidades;
- Coleta, armazenamento, e gerenciamento de informação de rastreamento necessária para completar estas tarefas.

É necessário haver a definição de termos específicos de variabilidade em uma LPS. Alguns desses termos são:

- Sujeito de variabilidade: É um item variável do mundo real ou uma propriedade de tal item. Deve responder a pergunta *"O que varia?"*.
- Objeto de variabilidade: É uma instância particular de um sujeito de variabilidade. Deve responder a pergunta *"Como varia?"*.
- Ponto de variação: É a representação de um sujeito de variabilidade dentro dos artefatos de domínio enriquecido com informação contextual.
- Variante: É a representação de um objeto de variabilidade dentro dos artefatos de domínio.

Na LPS trataremos de pontos de variação e variantes, pois são os sujeitos e objetos pertencentes ao escopo dela.Com os pontos de variação e variâncias devidamente definidas, podemos definir dois tipos de variabilidade:

- Variabilidade no Tempo: É a existência de diferentes versões de um artefato que são válidas em tempos diferentes.
- Variabilidade no Espaço: É a existência de artefatos de diferentes formas em um mesmo tempo.

As diferenças entre variabilidade no tempo e no espaço são facilmente identificadas através de simples exemplos. Podemos imaginar uma linha de produto de cliente de mensagens eletrônicas (e-mails). Este cliente pode possuir variabilidade em seus métodos de envio e recebimento de mensagens. No caso do recebimento de mensagens, podemos exemplificar como variabilidade no tempo a utilização de um protocolo mais antigo POP, ou a escolha de uma versão mais nova desse protocolo, o POP3. Para exemplificar a variabilidade no espaço, esse protocolo que é uma evolução(do POP ao POP3) segue uma variância, mas também poderia ser utilizado o protocolo IMAP como outra variância, causando assim a variabilidade no espaço, pois são artefatos de formas diferentes.

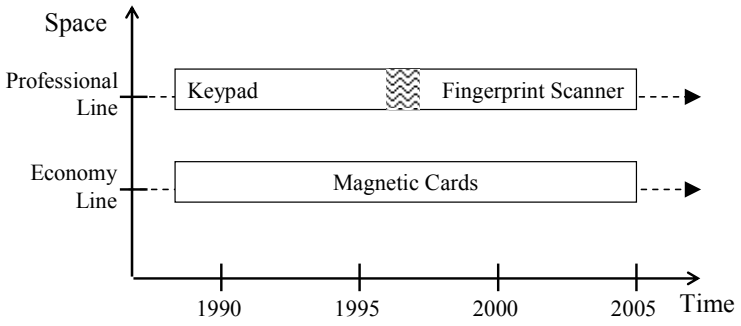

FIGURA 2.2 – Exemplo de variabilidade no tempo e no espaço [16]

Outro fator importante nas variabilidades é quais delas os stakeholders devem estar cientes. Quando uma variabilidade é dita *interna*, entende-se que essa variabilidade não é conhecida pelos clientes. Caso contrário, ela é dita *externa*.

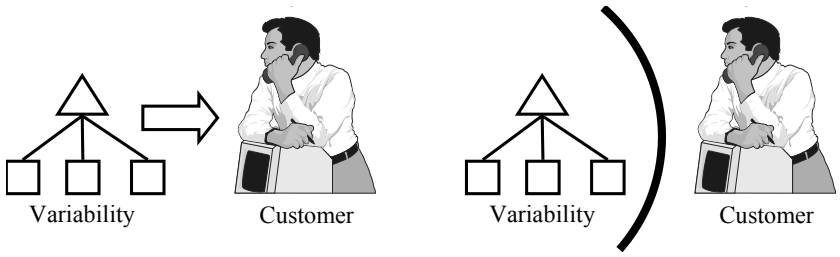

FIGURA 2.3 – Variabilidade externa e interna [16]

Após definida, a variabilidade possui algumas restrições. Essas restrições entre variantes e pontos de variação podem ser de dependência ou de exclusão, exibindo a interdependência ou incompatibilidade entre características de uma LPS. Essas dependências podem acontecer entre as seguintes combinações:

- Variâncias dependendo de/excluindo variâncias;
- Variâncias dependendo de/excluindo pontos de variação;
- Pontos de variação dependendo de/excluindo pontos de variação.

2.2.2 ENGENHARIA DE DOMÍNIO

Este processo é responsável por estabelecer a plataforma reusável e assim definir os aspectos comuns e a variabilidade da linha de produto. A plataforma consiste em todos os tipos de artefatos de software (requerimentos, projetos, realização, testes). Haver rastreabilidade entre esses artefatos facilita reuso sistemático e consistente.

O escopo deste trabalho leva em consideração o projeto e a realização de domínio de uma LPS, pois são assuntos que servem de base para a construção da ferramenta.

2.2.2.1 PROJETO DE DOMÍNIO

O objetivo principal do sub-processo projeto de domínio é produzir a arquitetura de referência, definindo a estrutura principal do software. O arquiteto determina como os requisitos, incluindo a variabilidade, são refletidos na arquitetura.

A mais importante atividade de projeto de sistemas únicos é definir uma arquitetura, que determina de que maneira o sistema será construído. Requisitos, incluindo sua variabilidade, devem ser mapeados em soluções técnicas para serem usados durante a realização do sistema. A arquitetura determina a estruturação do software em partes e seus relacionamentos e as regras comuns a serem aplicadas. Para apoiar isso, o arquiteto realizará as seguintes atividades de apoio:

- Abstração: Esta atividade agrupa informações do sistema em abstrações considerando apenas certos aspectos. Isto reduz a complexidade do projeto. Abstrações separadas lidam com diferentes aspectos dos sistemas.
- Modelagem: Esta atividade relata abstrações umas as outras de maneira a criar algum raciocínio entre elas.
- Simulação: Esta atividade "executa" certos modelos de maneira a medir certos aspectos do sistema. Freqüentemente há uma teoria de execução de software disponível que permite tradução dos resultados das medidas em propriedades do sistema.
- Prototipação: Esta atividade produz implementações rápidas, cobrindo aspectos importantes do sistema. O propósito é executar o protótipo para medir como o sistema atual se comporta.
- Validação: Em adição às atividades de projeto, o arquiteto tem um papel na validação dos resultados da realização. A validação considera se a arquitetura é obedecida pelo sub-processo da realização.

Há várias razões porque o relacionamento de rastreabilidade entre requerimentos e arquitetura não são um mapeamento simples um para um. Há até circunstâncias quando um requerimento comum é relacionado a um ativo de arquitetura variável e vice-versa. Entretanto, um bom arquiteto deixa o

relacionamento de rastreabilidade se tornar um mapeamento poucos para poucos, onde “poucos” é uma palavra deliberadamente vaga, e é certamente dependente das circunstâncias. No entanto, rastreabilidade é apenas usável quando é compreensível. Razões para adiantar de um simples mapeamento um para um são:

- Interação de requerimentos;
- Requerimentos da linha de produto, como flexibilidade ou adaptabilidade;
- Opções de tecnologia;
- Disponibilidade de recursos de desenvolvimento (pessoal, ferramentas, etc.);
- Preparação para o futuro (escalabilidade).

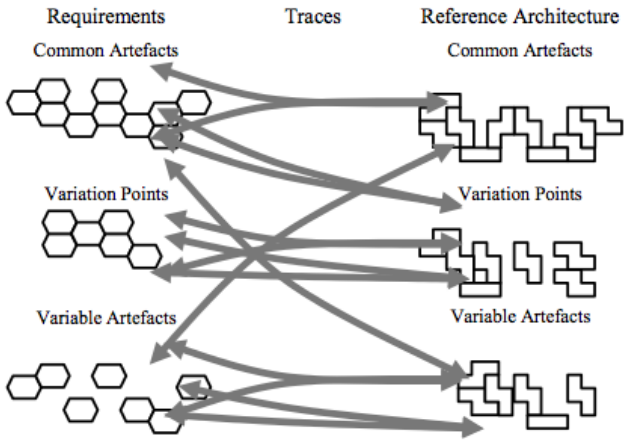

FIGURA 2.4 – Rastreabilidade entre requerimentos variáveis e ativos de arquitetura [16]

Variabilidade de requerimentos é uma importante fonte de variabilidade na arquitetura de referência. Variação em requerimentos freqüentemente resulta em variações no projeto ou na realização. O arquiteto analisa a comunalidade da variação primeiro, reduzindo a variação a um mínimo para facilitar a flexibilidade e evolução. Exceto por essa variabilidade externa, originada da variabilidade dos requisitos, o projeto também leva em conta variabilidade interna, que é introduzido pela solução técnica escolhida. Diferenças na qualidade dos requisitos pode levar em diferenças de dispositivos de hardware e software básico de funcionalidade tais como protocolos ou SGBDs. Isso resulta em variações em muitos lugares do software.

As ligações de rastreamento na figura 2.4 documentam a relação entre variabilidade de requisitos e variabilidade na arquitetura de referência assim como entre artefatos do domínio de requisitos e de projeto. As ligações trazem, por

exemplo, a estimativa de impacto de mudanças no curso de um processo de gerenciamento de mudanças.

Por causa das escolha de projeto iniciais em requisitos de qualidade, a variabilidade remanescente é freqüentemente distribuída, por exemplo porque vários subsistemas e camadas são afetadas. O arquiteto deve evitar duplicação da repetição de informação o máximo possível. Não é uma boa idéia ter os parâmetros de pontos de variação singulares distribuídos sobre vários lugares da aplicação. Uma possível solução é armazenar esses parâmetros em um lugar único, e deixar as outras partes da aplicação acessar este lugar para obter sua informação.

Um interesse principal da arquitetura é lhe dar com requisitos instáveis. Isto significa que é conhecido ou esperado que esses requisitos mudarão com o tempo. Em discussão com o gerenciamento de produtos e o engenheiro de requisitos, deve ser clarificado qual novo ou adaptado requisito pode ser esperado em curto ou longo prazo. A arquitetura deve apoiar futuras adoções desses requisitos o máximo possível. Assim como com requisitos normais, a esperada prioridade do novo requisito influencia quanto impacto o requisito possui na arquitetura, e quando a arquitetura deve tomar medidas em um estágio adiantado para lidar com ele. Por exemplo, a introdução de frameworks e o uso de seus plug-ins, as mudanças necessárias estão amarradas a locais específicos. Mesmo assim, mudanças tardias a arquitetura de referência não pode ser evitada completamente.

2.2.2.2 REALIZAÇÃO DE DOMÍNIO

A mais importante atividade de realização é construir um sistema funcionando de acordo com a arquitetura de referência. Esta atividade inclui o projeto e implementação detalhado de artefatos de software e compilação, ligação e configuração deles em código executável.

A arquitetura determina a decomposição de uma aplicação em artefatos de software, como componentes e interfaces. O projeto detalhado provê projetos para cada um deles, e, após validação, são implementados. Em vários casos, a realização de diferentes artefatos de software é feita por diferentes grupos de pessoas, cada um tomando conta de alguns artefatos relacionados. As seguintes atividades pertencem à realização:

- Projeto de interfaces;
- Projeto de componentes;
- Implementação de interfaces;
- Implementação de componentes;
- Compilação e ligação.

Interfaces podem ser usadas para implementar a variabilidade realizada em componentes. Uma interface provida pode ter funções que adaptam seleção de variantes internas. Isso significa que é possível para o ambiente adaptar-se a variante em tempo de execução. Uma interface requerida pode ser usada por um componente para questionar sobre informação relacionada a variabilidade no ambiente.

O nível de abstração determina o quão genérica ou específica é a informação mostrada pela interface. Se o nível de abstração for alto, a interface pode ser utilizada para vários propósitos, mas os desenvolvedores dos componentes requisitados estarão em dúvida sobre o que estará realmente acontecendo, e se a funcionalidade provida bate com a funcionalidade requerida. Se o nível de abstração for muito baixo, muita informação irrelevante é exposta na interface. Em alguns casos, um nível baixo de abstração não pode ser evitado.

Podem haver interfaces que são providas por vários componentes. Isto assegura por exemplo as interfaces relacionadas à aspectos. Tais interfaces devem ser muito genéricas. Caso contrário, ela não pode ser provida por cada componente. Os tipos nessas interfaces devem estar em um baixo nível de abstração.

A variabilidade de uma linha de produto eventualmente deve ser realizada em termos de componentes reusáveis. De maneira a obter o reuso, a realização do domínio desenvolve componentes de alta qualidade para prover a variabilidade requerida.

Componentes de domínio reusáveis podem ser usados em várias aplicações. Para seu projeto, isso significa que deve ser dada uma atenção especial para sua robustez. O contexto de utilização de um componente não é conhecido em tempo de desenvolvimento. Portanto, apenas pressuposições podem ser feitas que serão justificadas pelas interfaces requeridas e providas. Robustez significa que o

componente interage corretamente em várias circunstâncias, independentemente dos recursos utilizados e da ordem e tempo entre chamadas. Isto não significa que um componente deve ser projetado para executar sobre qualquer circunstância. Uma mensagem apropriada de erro deve ser retornada caso o componente não seja capaz de executar completamente uma requisição. Porém, isso significa que tal comportamento deve estar já declarado na interface.

As interfaces providas determinam a funcionalidade de um componente. Os métodos, tipos, e classes são todos providos da maneira que são declarados na interface. Nenhuma restrição adicional deve ser colocada neles, tais como ordem de chamada ou limite de re-entrâncias. Peças da interface provida tipicamente também ocorrem na interface requerida de um componente e assim podem usadas diretamente pelo componente. Um componente deve apenas utilizar funcionalidades externas que sejam disponibilizadas por interfaces.

Os componentes e interfaces são implementados em arquivos de programas. Esses arquivos são configurados e combinados em aplicações em vários passos:

- Compilação leva a arquivos objetos;
- Ligação leva a executáveis e bibliotecas dinâmicas;
- Carregamento leva vários executáveis e bibliotecas dinâmicas juntos em um mesmo sistema.

A realização define diferentes tempos de amarração da variabilidade. Mecanismos diferentes são utilizados para amarrar variantes antes, durante ou depois cada passo. Tais mecanismos devem prover os meios apropriados para localizar variantes e determinar quais variantes serão amarradas. A escolha do período de amarração e o mecanismo de suporte é independente da modelagem da variabilidade. Para o caso particular, veremos no próximo parágrafo o amarramento em tempo de carregamento, maneira que o FlexCM, que é tecnologia alvo da ferramenta (será explicado no capítulo 3), funciona.

Em tempo de carregamento, um arquivo de configuração contém uma lista de arquivos que serão carregado juntos. O conjunto de arquivos na seleção forma um sistema executável. O arquivo de configuração deve ter conteúdo variável de maneira a realizar diferentes variantes de um ponto de variação. O arquivo de configuração usa a *runtime* para localizar e iniciar todos os arquivos que devem ser carregados. Este mecanismo é útil para produzir sistemas que consistem de

configurações de componentes binários variáveis. Comumente regras de arquitetura são necessárias para prover cada executável com mecanismos de direito a ativar sua localização, inicialização e ligação ao restante do sistema.

A realização do domínio provê uma coleção coerente de artefatos de software reusáveis. Isso significa que:

- A realização do domínio não provê uma aplicação completa;
- Interfaces devem ser projetadas cuidadosamente com o apropriado nível de detalhe e de abstração para ser usado em várias aplicações;
- Gerenciamento de configurações é uma atividade mais importante na engenharia de domínio que em sistemas simples;
- Artefatos de software devem incorporar variabilidade;
- Os componentes e interfaces são mais robustos que os requeridos em um desenvolvimento de sistema simples.

2.2.3 ENGENHARIA DE APLICAÇÃO

O objetivo principal da engenharia de aplicação é derivar uma aplicação de LPS reusando o máximo possível os artefatos do domínio. Isso pode ser conseguindo explorando a comunalidade e a variabilidade da linha de produto estabelecida na engenharia de domínio.

O escopo deste trabalho leva em consideração o projeto e a realização de aplicação de uma LPS, pois são os assuntos que servem de base para a construção da ferramenta.

2.2.3.1 PROJETO DE APLICAÇÃO

O objetivo principal do sub-processo de projeto de aplicação é produzir a arquitetura da aplicação. A arquitetura da aplicação é uma especialização da arquitetura de referência desenvolvida no projeto de domínio. Os arquitetos de aplicação amarram a variabilidade da arquitetura de referência e introduzem mudanças específicas da aplicação de acordo com a especificação de requisitos da aplicação. A arquitetura de aplicação é passada para a realização da aplicação onde os componentes reusáveis e interfaces são montadas e onde componentes e interfaces específicas de aplicação são desenvolvidas.

O projeto de aplicação suporta decisões "*trade-off*" feitas na engenharia de requisitos da aplicação determinando o esforço estimado da realização. Decisões *"trade-off"* sobre requisitos específicos de aplicação são parte da negociação com os *stakeholders* no subprocesso de engenharia de requisitos de aplicação.

O arquiteto de aplicação provê *feedback* através de pedidos de alteração e adição de partes ao projeto que podem levar a uma melhora na arquitetura de referência. Além disso, o projeto de aplicação entrega artefatos de projeto, que devem receber manutenção para flexibilidade e reuso e incorporados à plataforma, para o projeto de domínio.

A realização de aplicação provê *feedback* em todos os tipos de erro de projeto que emergem durante a realização e devem ser resolvidos pelos arquitetos. Estes incluem, acima dos erros de projeto comuns, componentes e interfaces que se tornam não - reusáveis, e configurações que não funcionam corretamente.

Um arquiteto de aplicação possui responsabilidades similares a de um arquiteto tradicional. Como tal, abstração, modelagem, simulação e prototipação são atividades que são realizadas por ele. Porém, todas essas atividades devem ser realizadas apenas para as partes específicas do projeto da aplicação. A arquitetura de referência inclui várias decisões que podem ser reusadas na engenharia de aplicação. O arquiteto de aplicação inicia com a arquitetura de referência e a especializa em direção da arquitetura de aplicação. Os modelos da arquitetura de referência são especializados são especializados através de amarrações das variantes de acordo com as amarrações no modelo de variabilidade da aplicação e adicionando partes específicas da aplicação.

O arquiteto de aplicação introduz abstrações que são necessárias para a aplicação específica à mão, como por exemplo adicionar abstrações para aspectos específicos da aplicação que não são cobertos pelos artefatos do domínio. As abstrações adicionais são geralmente relacionadas a requisitos específicos da aplicação. Especialmente quando há requisitos de qualidade rigorosos, novas abstrações de aplicação devem ser introduzidos. As abstrações da arquitetura de aplicação devem estar em conformidade com as abstrações definidas na arquitetura de referência para obter uma arquitetura de aplicação consistente.

A qualidade da arquitetura de referência determina o quão fácil será amarrar as variantes aos pontos de variações da arquitetura de referência, como definido no modelo de variabilidade da aplicação. Isto depende de:

- A maneira que a customização em massa é incorporada;
- As abstrações usadas, determinando quais pontos de variação e variantes estarão disponíveis;
- A rastreabilidade entre variabilidade nos artefatos de requisito do domínio e a arquitetura de referência.

Para reduzir o trabalho da realização da aplicação o máximo possível, artefatos reusáveis do domínio devem ser utilizados sempre que possível. A arquitetura de referência determina interfaces e componentes comuns. Por amarrar os pontos de variação, o arquiteto de aplicação seleciona artefatos de projeto de domínio adicionais que podem ser reusados. Se nenhum artefato de domínio está disponível, o arquiteto de aplicação deve definir um artefato específico de aplicação durante a realização da aplicação.

Certas variantes devem ser realizadas para uma única aplicação. Estas variantes podem envolver novos componentes ou interfaces, mas algumas vezes pode envolver também peças maiores da estrutura, como configurações de componentes e interfaces, novos padrões e até novos *frameworks* de componentes.

Quando projetando peças específicas da aplicação, o arquiteto deve considerar o esforço adicional na realização da aplicação. Freqüentemente, não há pessoal ou tempo disponível para realizar muitos componentes do rascunho. O arquiteto deve avaliar cuidadosamente o que pode ser implementado especificamente para uma simples aplicação pensando nos recursos disponíveis. A quantidade de recursos disponíveis para a realização da aplicação deve ser negociada com a engenharia de requisitos da aplicação, os *stakeholders*, e o gerenciamento de produto. Para o desenvolvimento de aplicações normal, a quantidade de recursos pode ser escassa, o que significa que muito trabalho adicional não pode ser feito. Em aplicações de relevância estratégica, recursos adicionais podem ser alocados, como por exemplo em um produto líder de mercado, ou dependendo da importância do cliente.

Parte da variabilidade da linha de produto se dá com especificações de hardware da aplicação, como tamanho de memória e quantidade de periféricos de hardware ou poder de processamento. Variações específicas de hardware estão amarradas tanto

por selecionar componentes apropriados quanto por definir parâmetros de configuração de um ou mais componentes a valores apropriados.

O projeto de aplicação tem o mesmo papel que o projeto de software em sistemas singulares, sem LPS. A arquitetura da aplicação determina a estrutura global de uma aplicação particular e deve ser capaz de satisfazer os requisitos de aplicação. O arquiteto deve usar a arquitetura de referência, que provê um projeto para os vários dos requisitos de aplicação que o arquiteto deve satisfazer. Ainda mais, componentes e interfaces reusáveis, e configurações deles, são providas pela arquitetura de referência. Portanto, o arquiteto pode focar sua atenção em peças específicas da aplicação, salvando assim muito tempo.

Na discussão com o arquiteto de domínio, certas soluções e artefatos de aplicação podem se tornar candidatos para integração na plataforma. Comumente a integração aparece após a aplicação é finalizada e as propriedades dos artefatos desenvolvidos são validados. O arquiteto de aplicação tem a responsabilidade de prover o arquiteto de domínio com informação sobre tais artefatos.

2.2.3.2 REALIZAÇÃO DA APLICAÇÃO

O objetivo da realização da aplicação é desenvolver aplicações que podem ser testadas e levadas ao mercado após garantir uma qualidade suficiente. A realização da aplicação provê o projeto e implementação de componentes específicos da aplicação detalhadamente e configura-os com as variantes corretas dos ativos do domínio em aplicações. Os resultados principais da realização de aplicação são os componentes e interfaces específicos da aplicação, as variantes selecionadas dos componentes reusáveis e a configuração da aplicação.

O projeto de aplicação provê a arquitetura de aplicação que determina a configuração de componentes e interfaces, os quais são reusados da plataforma ou projetados especificamente para a aplicação considerada.

A realização da aplicação provê um *feedback* tendo em vista todos os tipos de erros de projeto que são descobertos durante a realização e quais devem ser corrigidos pelo projeto de aplicação. Um exemplo de tal erro é uma configuração que não funciona como deveria.

A realização provê uma aplicação completa pronta para testes. O teste de aplicação realiza testes unitários, de integração e de sistema baseados nos artefatos de requisitos, projeto e realização da aplicação.

Como *feedback*, a realização da aplicação recebe resultados dos testes incluindo aceitação ou rejeição da aplicação e relatórios de problemas descrevendo como os itens de testes falham. Este retorno é utilizado para melhorar a aplicação até que sua aceitação seja alcançada. Mais ainda, o teste relata defeitos em descrições de interface que são detectados no teste de aplicação já que eles dificultam o desenvolvimento de casos de teste.

A entrada para a realização da aplicação da realização do domínio consiste em componentes e interfaces reusáveis projetadas, implementadas e prontas para uso. De maneira a ser capaz de integrar esses artefatos em uma aplicação, a realização de domínio provê adicionalmente suporte de configuração.

A realização de aplicação provê um *feedback* através de requisições para adição e alteração de realizações. As requisições pertencem a uma funcionalidade ou qualidade que deve ser provida pela plataforma. Ainda mais, ela desenvolve componentes e interfaces específicas de aplicação que podem ser integradas aos artefatos de domínio.

Muitas interfaces entre os componentes de uma aplicação são interfaces de domínio reusáveis. Grande parte dos componentes da aplicação carregam-nas consigo já que são meios importantes de implementar pontos de variação. Diferentes variantes de um simples componente as vezes requerem e provêem as mesmas interfaces. Tais interfaces são realizadas durante a engenharia de domínio mas são fortemente utilizadas durante a engenharia de aplicação.

Componentes reusáveis de domínio as vezes possuem pontos de variação internos. A realização de domínio provê mecanismos para suportar tal seleção de variação. Variantes dentro de um componente podem ser selecionadas, por exemplo, por parâmetros. Guiados pela arquitetura e o modelo de variabilidade da aplicação, a realização seleciona as variantes adequadas dos componentes para ser parte da aplicação. Para cada componente reusável, a realização da aplicação determina a escolha correta de parâmetros do componente para amarrar a uma certa variante.

Componentes específicos de aplicação são realizados da mesma maneira que na engenharia de sistemas simples. Porém, em vários casos, as interfaces de domínio podem ser reusadas para componentes específicos de aplicação.

Interfaces e componentes específicos de aplicação são necessários sempre que não houver nenhum componente reusável do domínio disponível que se encaixe. Fazer componentes específicos de aplicação reusáveis não é do interesse do desenvolvedor que trabalha com uma simples aplicação apenas. Se o componente deve ser integrado nos artefatos de domínio, a realização de domínio o toma em um tempo apropriado, sem atrapalhar a realização da aplicação. Porém, um componente de aplicação pode parecer com componentes existentes do domínio. Neste caso, é uma boa idéia usar o projeto das variantes existentes como entrada do projeto da nova variante.

A última tarefa da realização de aplicação é a realização da configuração que é colocada na aplicação. Variantes de componentes devem ser compiladas, ligadas, e implantadas no hardware atual. Em todos esses passos a variabilidade pode ser amarrada, dependendo do mecanismo usado.

Uma variante de componente é realizada como uma coleção de arquivos. Os arquivos englobam fontes, cabeçalhos, e definições de parâmetros. Interfaces são normalmente realizadas em um ou vários cabeçalhos. Os cabeçalhos para interfaces requeridas são necessariamente capazes de compilar a variante do componente. Note que as interfaces providas não precisam incluir elas mesmas. Ligações combinam as variantes dos componentes à executáveis ou bibliotecas dinâmicas. A aplicação é composta de um ou vários executáveis e nenhuma ou várias bibliotecas dinâmicas.

Assim como na engenharia de sistemas simples, com o tempo, cada componente e interface existirão em várias versões. Novas versões são originadas de manutenção assim como da incorporação de novos requisitos. A última situação ocorre especialmente se um componente é usado em mais de uma aplicação, a qual é uma situação normal para ativos de plataformas. A seleção das variantes do componente também deve levar em consideração a versão a ser usada. Uma versão posterior é geralmente melhor tendo em vista a qualidade do componente. Assim, é possível que uma nova versão de um componente não pode ser introduzida numa aplicação já que não será capaz de interagir com outros componentes. Isso pode ser por causa de

mudanças na funcionalidade, mudanças no apoio de qualidade, e mudanças nas interfaces, por exemplo.

Durante os testes de integração e manutenção já em funcionamento, relatórios de problemas serão emitidos. A engenharia de aplicação é responsável por resolver os problemas relatados. Quando os problemas são relacionados a componentes e interfaces específicas da aplicação, é responsabilidade da realização da aplicação corrigi-los. Porém, quando o problema é relacionado aos artefatos de domínio, o problema deve ser repassado à engenharia de domínio para ser consertado.

A realização da aplicação provê uma aplicação que funciona e está pronta para testes. Esta aplicação é baseada na arquitetura de aplicação e reusa componentes e interfaces do domínio. Isto reduz o esforço da realização significativamente. Isto significa que:

- O desenvolvedor da aplicação seleciona as variantes dos componentes de domínio reusáveis;
- As variantes componentes de domínio reusáveis devem ser consistentes umas com as outras e ter conformidade com a arquitetura da aplicação;
- A aplicação é construída configurando componentes e interfaces tanto reusáveis do domínio quanto específicos da aplicação.

A realização da aplicação se dá com projeto em detalhes, implementação, e configuração de componentes para produzir uma aplicação executável. Interfaces são reusadas da plataforma sem mudanças, mas componentes as vezes possuem pontos de variação internos, que devem ser amarrados (por exemplo, por parâmetros). Componentes específicos da aplicação recém implementados e componentes reusáveis especializados são configurados e conectados pelas interfaces para montar a aplicação. Após montagem, a aplicação pode ser testada e implantada no *hardware* alvo.

2.3. LINGUAGENS DE DESCRIÇÃO DE ARQUITETURA

Uma ADL é uma linguagem que provê características para modelar uma arquitetura conceitual de um sistema de software, distinto da implementação do sistema. Essas linguagens provêem tanto uma sintaxe concreta quanto um *framework* conceitual para caracterizar arquiteturas. Este framework em geral reflete características do domínio para o qual a ADL foi projetado e/ou o estilo arquitetural. O *framework* tipicamente agrupa as teorias semânticas subseqüentes da ADL (CSP, Redes de Petri, FSM). Uma característica marcante das ADLs é que não possuem uma formalização unificada de suas linguagens ou padrões, pois geralmente são muito específicas para o domínio do problema.

Os blocos de construção de uma descrição arquitetural são: componentes, conectores e configurações arquiteturais. Uma ADL deve prover meios para sua especificação explícita; isso nos permite determinar quando, ou não, uma rotação particular é uma ADL. De maneira a inferir qualquer tipo de informação sobre uma arquitetura, no mínimo, também devem ser modeladas as interfaces constituintes dos componentes. Sem esta informação, uma descrição arquitetural se torna nada mais que uma coleção de identificadores (interconectadas), similar a um diagrama de "caixas e linhas" sem nenhuma semântica por baixo. Muitos outros aspectos de componentes, conectores e configurações são desejáveis, mas não essenciais: seus benefícios foram reconhecidos e comprovados no contexto do problema, mas sua ausência não descaracteriza de dada linguagem ser ADL.

Um componente numa arquitetura é uma unidade de computação. Componentes podem ser tão pequenos quanto um simples procedimento ou grande como uma aplicação inteira. Cada componente pode requerer seus próprios dados ou espaços de execução, ou pode dividi-las com outros componentes. Como dito anteriormente, interfaces de componentes explícitas são necessárias para ADLs.

Uma interface de um componente é um conjunto de pontos de interação entre ele e o mundo exterior. A interface especifica os serviços (mensagens, operações e variáveis) que um componente provê.

Os conectores são blocos de construção arquitetural utilizados para as interações entre os componentes do modelo e as regras que governam as interações. Ao contrário de componentes, os conectores podem não corresponder à compilação de unidades em um sistema implementado. Eles podem ser implementados como

dispositivos encaminhadores de mensagens compiláveis separáveis, mas também podem manifestar-se como variáveis compartilhadas, as entradas de tabela, *buffers*, instruções para um *linker*, estruturas de dados dinâmicas, seqüências de chamadas de procedimento incorporado no código, parâmetros de inicialização,protocolos cliente-servidor, *pipes*, ligações entre um banco de dados SQL e uma aplicação, e assim por diante.

Configurações arquiteturais, ou topologias, são grafos conectados de componentes e conectores que descrevem uma estrutura arquitetural. Esta informação é necessária para determinar quando componentes apropriados são conectados, suas interfaces combinam, conectores ativam comunicações apropriadas, e suas semânticas combinadas resultam no comportamento desejado. De acordo com modelos de componentes e conectores, descrições de configurações permitem avaliação de aspectos concorrentes e distribuídos de uma arquitetura, por exemplo, possíveis *deadlocks* e *starvation*, performance, confiança, segurança e assim sucessivamente. Descrições de configurações também permitem análise de arquiteturas para aderência a heurísticas de projeto e restrições de estilo arquitetural.

Cararcterísticas destacantes no nível de configuração arquitetural cai em três categorias gerais:

- Qualidade de descrição de configuração: inteligibilidade, composicionalidade, refinamento e rastreabilidade, e heterogeneidade;
- Qualidade do sistema descrito: heterogeneidade, escalabilidade, evolucionariedade, e dinamismo;
- Propriedades dos sistemas descritos: dinamismo, restrições e propriedades não funcionais.

Note que as três categorias não são totalmente ortogonais: heterogeneidade e dinamismo aparecem em duas categorias. Heterogeneidade pode ser manifestada em múltiplos formalismos empregados em descrições de configurações e múltiplas linguagens de programação em implementações de sistemas. Dinamismo antecipado é uma propriedade que o sistema pode ser arquitetado especificamente para acomodar a (esperada) mudança dinâmica; dinamismo não antecipado é uma qualidade que refere a uma adequação geral do sistema para uma mudança dinâmica.

As diferenças entre os dois pares de características são sutis, particularmente no caso do dinamismo. Enquanto mantemos tal categorização em mente, de maneira a manter a simplicidade conceitual do nosso *framework* e evitar confusão, procedemos por descrever características individuais; incluímos tanto as noções de heterogeneidade e dinamismo em apenas uma das duas categorias (qualidade de descrição da configuração e do sistema descrito, respectivamente).

2.4. MIDDLEWARE GINGA

O Ginga é a especificação oficial e padronizada de middleware para o Sistema Brasileiro de TV Digital. As aplicações Ginga podem ser divididas em dois conjuntos: o das aplicações declarativas escritas na linguagem de programação Java e o das aplicações procedurais escritas na linguagem de programação NCL Nesse sentido, o middleware Ginga possui dois ambientes para suportar aplicações interativas: Ginga-J e Ginga-NCL [9].

A arquitetura do middleware Ginga tem como objetivo integrar as duas soluções (Ginga-NCL e Ginga-J) utilizando por base a arquitetura definida na norma internacional ITU J.200. Esta divisão permite o desenvolvimento de aplicações seguindo dois paradigmas de programação diferentes. Dependendo das funcionalidades requeridas no projeto de cada aplicação, um paradigma será mais adequado do que o outro.

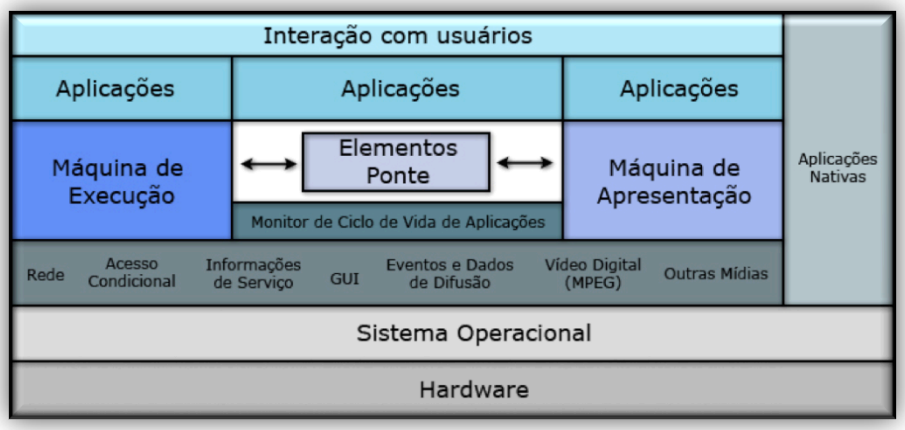

FIGURA 3.1 – Arquitetura estabelecida pelo padrão ITU J.200 [9]

O Ginga-NCL (ou Máquina de Apresentação) é um subsistema lógico do Sistema Ginga que processa documentos NCL. Um componente-chave do Ginga-NCL é o mecanismo de decodificação do conteúdo informativo (formatador NCL). Outros módulos importantes são o navegador XHTML, que inclui suporte a linguagem de estilo (CSS) e intérprete ECMAScript, e o mecanismo LUA, que é responsável pela interpretação dos scripts LUA.

O Ginga-J (ou Máquina de Execução) é um subsistema lógico do Sistema Ginga que processa aplicações procedurais (Xlets Java). Um componente-chave do ambiente imperativo é o mecanismo de execução do conteúdo procedural, que tem por base uma Máquina Virtual Java.

Esses dois subsistemas executam sobre um subsistema chamado Ginga-CC (ou Núcleo Comum) que funciona como uma plataforma para fornecimento de recursos do middleware, como sintonização, exibição de mídias, canal de retorno, tratamento de entradas, etc. O Ginga-CC roda em cima do sistema operacional Linux, que é o sistema operacional que irá controlar acesso às bibliotecas requeridas e operar o hardware do sistema.

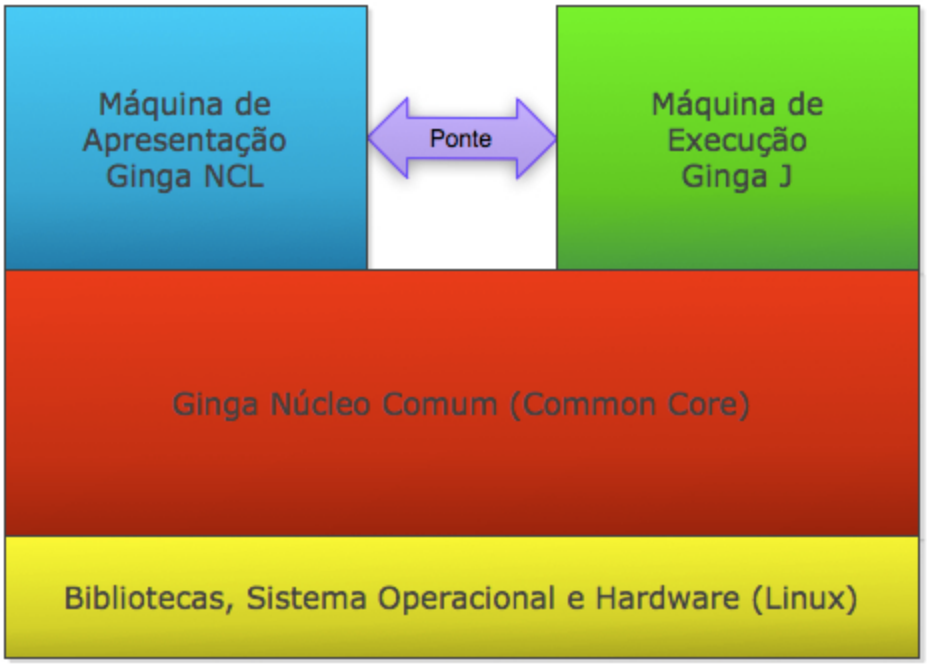

FIGURA 3.2 – Arquitetura do Middleware [10]

O Ginga-CC é composto por uma série de componentes bem especificados, desacoplados e coesos. O conjunto de componentes que compõem o Ginca-CC

são:*Return Channel, Stream Information, Controller, Security, Media Processing, Demultiplexer, Data Processing, Application Manager, Graphics Manager, Tuner, Persistence* e *Input Manager.* Esses componentes atualmente são implementados através da linguagem C++ e compilados para o sistema operacional Linux como bibliotecas de ligação dinâmica.

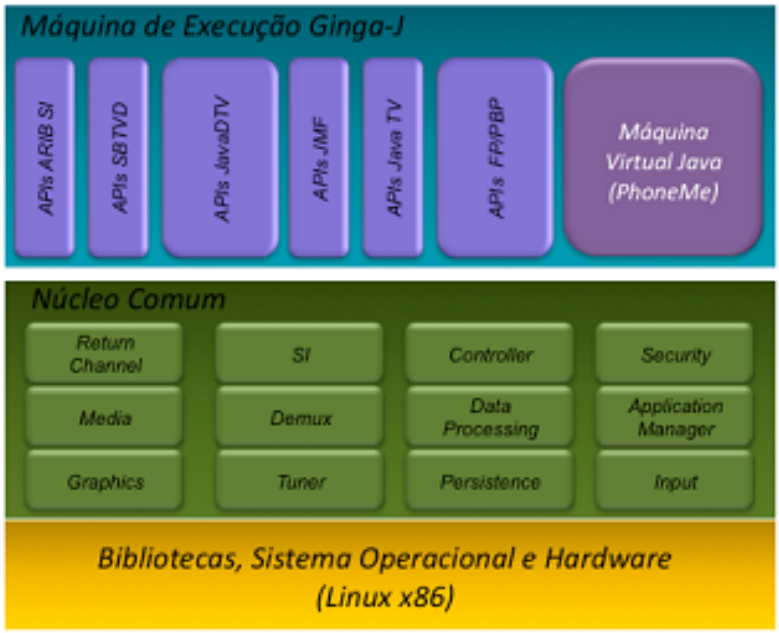

FIGURA 3.3 – Pilha da arquitetura de acordo com a máquina de execução [9]

2.4.1 FLEXCM

O Núcleo Comum visto anteriormente é executado através de um ambiente de execução de componentes, já que todos os recursos desenvolvidos no núcleo comum são componentizados.

Este ambiente de execução que é utilizado é o FlexCM – Flexible Component Model, um modelo de componentes para sistemas embarcados. O modelo de componentes FlexCM foi projetado para ser independente de plataforma de execução e de linguagem de programação [14]. A principal característica do modelo é a montagem automática de arquiteturas baseadas em componentes através da representação explícita das conexões entre componentes através de um arquivo de descrição arquitetural, escrito em XML.

O FlexCM enfatiza a modelagem de software através da decomposição do sistema em componentes funcionais com interfaces de interação bem definidas. Assim, um

modelo de componentes padroniza o esquema de instanciação, composição e o ciclo de vida dos componentes de um sistema e o ambiente de execução é um software responsável por gerenciar os componentes implementando e garantindo as especificações definidas pelo respectivo modelo de componentes.

O modelo FlexCM segue uma abordagem declarativa, onde os componentes declaram suas dependências explicitamente (interfaces requeridas) e o ambiente de execução carrega e provê as dependências através do padrão de injeção de dependências. Desta forma, o modelo FlexCM permite que seus componentes conheçam apenas as interfaces, enquanto as implementações são tratadas pelo ambiente de execução.

Além das interfaces requeridas, os componentes registrados no FlexCM também podem declarar parâmetros de configuração cujos valores também são injetados pelo ambiente de execução permitindo o desenvolvedor configurar facilmente o componente no produto final onde ele será instalado.

Além das vantagens comumente conhecidas do DBC como, por exemplo, modularidade, manutenabilidade e reuso, o modelo de componentes FlexCM oferece uma série de vantagens específicas para a implementação do Ginga:

- Conhecimento da arquitetura em nível de modelo;
- Facilidades na configuração dos componentes individuais;
- Facilidades na configuração do sistema como um todo.

Por fim, essas características trazem a possibilidade de gerenciar diferentes arquiteturas facilitando também a execução de testes de unidade e integração de diferentes porções da arquitetura, assim como a adição de uma maneira simples do Ginga a uma linha de produto de software, sem gerar grandes impactos no processo de produção do middleware.

2.4.2 A LINHA DE PRODUTO DE SOFTWARE GINGA

No contexto de TV Digital, um middleware basicamente padroniza o ciclo de vida da aplicação e as APIs providas para as aplicações interativas funcionarem sobre. Em geral, um middleware deve se dar com variabilidades como portabilidade para diferentes plataformas de hardware, diferentes padrões de difusão, apoio para

aplicações difundidas, etc. Essas variabilidades podem ser facilmente mapeadas em um modelo de variabilidades numa LPS.

Também pode-se observar que no caso do middleware Ginga, há uma estrutura consolidada de desenvolvimento baseado em componentes reusáveis, o que também fornece um bom embasamento para a implantação tardia de uma linha de produto de software, já que o processo de produção não foi iniciado como uma LPS.

Alguns pré-requisitos chave realizados são importantes para a viabilidade da implantação dessa linha, como programação orientada à objetos e desenvolvimento baseado em componentes. Como técnicas de amarração tardia dos componentes (*Late-binding*), ocorrendo durante à execução (*run-time binding*), estão sendo utilizadas, isto colabora com um ambiente favorável à implantação desta linha.

Outra característica interessante que faz com que facilite a implantação da LPS é que, por ser um padrão, dificilmente ocorre mudança nos requisitos chave do middleware. Logo, temos requisitos de domínio estáveis e consolidados, sendo provavelmente também bem analisados e calculados.

2.5. EXTENSÃO DA FRACTAL ADL PARA LINHAS DE PRODUTOS

No contexto de linhas de produto de software existem dois tipos de arquiteturas: a arquitetura de referência e a arquitetura do produto final. A arquitetura de referência descreve a arquitetura da linha de produto, sendo tida como não instanciada, pois não faz referência a componentes concretos. Por outro lado, a arquitetura do produto final reflete a arquitetura de um produto específico, a qual é conhecida como arquitetura instanciada, uma vez que faz referência às implementações específicas de componentes que serão utilizadas. Para uma determinada linha de produto, pode existir uma única arquitetura de LPS e diversas arquiteturas de produto, cada uma refletindo uma versão específica de um produto gerado a partir da linha.

Para a ferramenta proposta neste trabalho, foi bastante utilizada a linguagem Fractal ADL, que é uma linguagem de descrição de arquitetura desenvolvida para o modelo de componentes Fractal.

2.5.1 FRACTAL ADL

A Fractal ADL é uma linguagem aberta e extensível para definir arquiteturas de componentes para o modelo de componentes Fractal, o qual já é aberto e extensível, segundo [4] e [11]. Mais precisamente, a Fractal ADL é composta de um conjunto aberto e extensível de módulos ADL, onde cada módulo define uma sintaxe abstrata para um dado "aspecto" arquitetural (tais como interfaces, componentes, amarrações, atributos, ou relacionamentos de continência). Os usuários dela são então livres para definir seus próprios módulos para os seus próprios aspectos. Eles podem também definir sua própria sintaxe concreta para a linguagem.

A Fractal ADL utiliza a sintaxe padrão do XML para representar seus elementos da ADL. Isso traz várias facilidades na interpretação (já que vários frameworks realizam *parsing* de arquivos XML), leitura e traz também um modelo consolidado para linguagens declarativas de fácil personalização.

A partir de agora iniciaremos a apresentação de alguns dos módulos típicos da linguagem, de maneira a introduzir um embasamento sobre como é a construção de arquiteturas pela Fractal ADL.

No exemplo a seguir, da figura 4.1, podemos observar na definição acima que existe um componente de nome *ClientImpl* que possui uma interface provida ("server") de nome *r* e identificador *java.lang.Runnable*, bem como uma interface requerida ("client") de nome *s* e identificador *Service*. Além dessas informações, existe a especificação da implementação do componente pela classe *ClientImpl*(elemento *content*, atributo *class*). Percebe-se que o nome do componente é igual ao nome da classe que o implementa. Essa prática não é obrigatória, mas é freqüentemente usada.

```
<definition name="ClientImpl">
  <interface name="r" role="server" signature="java.lang.Runnable"/>
  <interface name="s" role="client" signature="Service"/>
  <content class="ClientImpl"/>
</definition>
```

FIGURA 4.1 – Componente Primitivo

Como pode ser analisado na definição acima, existe um componente de nome *ClientImpl* que possui uma interface provida ("server") de nome *r* e identificador *java.lang.Runnable*, uma interface requerida ("client") de nome *s* e identificador *Service*. Além dessas informações, existe a especificação da implementação do componente pela classe *ClientImpl* (elemento *content*, atributo *class*). Percebe-se que o nome do componente é igual ao nome da classe que o implementa. No caso do Ginga, estes nomes não serão iguais, devido que a identificação dos componentes será feita por GUIDs, que são números de 128 bits gerados de forma a garantir a sua unicidade global.

Além de componentes primitivos, pode-se construir componentes à partir de outros componentes já criados, os chamados componentes compostos. Na figura 4.2 podemos observar um exemplo de componente composto.

```
<definition name="HelloWorld">
  <interface name="r" role="server" signature="java.lang.Runnable"/>
  <component name="client">
    <interface name="r" role="server" signature="java.lang.Runnable"/>
    <interface name="s" role="client" signature="Service"/>
    <content class="ClientImpl"/>
  </component>
  <component name="server">
    <interface name="s" role="server" signature="Service"/>
    <content class="ServerImpl"/>
  </component>
  <binding client="this.r" server="client.r"/>
  <binding client="client.s" server="server.s"/>
</definition>
```

FIGURA 4.2 – Componente Composto

Na Figura 4.2 é declarado no início do arquivo xml um componente composto denominado *HelloWorld* e uma interface provida (“server”) e de identificador *java.lang.Runnable*. Logo em seguida, existe a especificação de dois componentes que fazem parte de *HelloWorld*. O primeiro é *Client* que apresenta uma interface provida de nome *r* e de identificador também igual a *java.lang.Runnable* e uma interface requerida de nome *s* e identificador *Service*. O segundo componente apresenta o nome *Server* e uma interface provida de nome *s* e identificador *Service*. É interessante enfatizar que *Client* e *Server* são componentes primitivos com as definições já vistas anteriormente. Por fim, nas últimas linhas é possível encontrar a especificação das ligações entre os componentes pelo uso do elemento *binding*. Existe a ligação da interface *r* do componente composto *HelloWorld* (expressado pelo “This”) com a interface *r* do componente *client*, bem como a ligação do componente *client* e *server* por meio da interface *s*. Com a possibilidade de expressar ligações através da ADL Fractal, a construção de arquiteturas de software baseadas em componente é alcançada, bastando apenas usar os elementos do tipo *binding* para interligar os componentes.

Um terceiro fator importante da Fractal ADL é a possibilidade de representação de parâmetros de configuração na sua descrição. Esses parâmetros são presentes nos componentes e são conhecidos como descritor controlador, que são identificados pela definição do descritor do controlador, uma interface controladora de atributos e parâmetros de configuração.

```
<definition name="ServerImpl">
 <interface name="s" role="server" signature="Service"/>
 <content class="ServerImpl"/>
 <attributes signature="ServiceAttributes">
  <attribute name="header" value="->"/>
  <attribute name="count" value="1"/>
 </attributes>
 <controller desc="primitive"/>
</definition>
```

FIGURA 4.3 – Controlador de Componentes

Na figura 4.3, temos declarado em preto recursos já conhecidos: uma definição de componente de nome *ServerImpl* com uma interface provida *s*, bem como uma classe de implementação chamada *ServerImpl*. Em vermelho, podemos notar a adição de um conjunto de atributos declarados através do controlador de atributos de nome *ServiceAttributes*, com os atributos *header* de valor *->* e *count* de valor *1* e também podemos notar o descritor controlador como *primitive*. O descritor controlador tem o papel de agrupar conjuntos de atributos de maneira a organizar e identificar a entrada de parâmetros ao componente.

Outro aspecto importante é o mecanismo de extensão da linguagem.Essa ADL permite que sejam herdados (como herança de classes em programação orientada a objetos) definições e componentes já declarados. Nessa herança, todos os componentes, interfaces e amarrações criados na definição herdada serão importados para a definição que herda. As figuras 4.4 e 4.5 mostram como é utilizado o mecanismo de herança.

```
<definition name="ClientType">
 <interface name="r" role="server" signature="java.lang.Runnable"/>
 <interface name="s" role="client" signature="Service"/>
</definition>

<definition name="ClientImpl" extends="ClientType">
 <content class="ClientImpl"/>
</definition>
```

FIGURA 4.4 – Mecanismo de extensão

Na Figura 4.4 acima é visto a definição do componente *ClientType* e o componente *ClientImpl*, o qual estende de *ClientType*. No exemplo, existe a *super definição* representada pela definição *ClientType* e a *sub-definição* representada por *ClientImpl*. Dessa forma, as interfaces providas e requeridas especificadas em *ClientType* são herdadas para *ClientImpl*. A mesma idéia é encontrada com componentes compostos, conforme pode ser analisado na Figura 4.5 a seguir.

```
<definition name="AbstractClientServer">
 <interface name="r" role="server" signature="java.lang.Runnable"/>
 <component name="client" definition="ClientType"/>
 <component name="server" definition="ServerType"/>
 <binding client="this.r" server="client.r"/>
 <binding client="client.s" server="server.s"/>
</definition>

<definition name="ClientServerImpl" extends="AbstractClientServer">
 <component name="client" definition="ClientImpl"/>
 <component name="server" definition="ServerImpl"/>
</definition>
```

FIGURA 4.5 – Mecanismo de extensão com reuso de definição

Na Figura 5 acima é definido um componente *ClientServerImpl* que herda a definição do componente *AbstractClientServer*. É interessante notar outro recurso da ADL fractal no exemplo encontrado na Figura 5 que é o de referência de definições através da inclusão do atributo *definition*. Com esse atributo, é possível referenciar definições localizadas externamente, ou seja, definições de componentes que não estão incorporadas no mesmo arquivo xml do componente composto. Por exemplo, no componente *AbstractClientServer* existe a referência para a definição do componente *ClientType*, a qual é sobrescrita pela referência *ClientImpl* em *ClientServerImpl*.

2.5.2 FRACTAL ADL PARA LPS

No contexto de linhas de produto de software existem dois tipos de arquiteturas: a arquitetura de referência e a arquitetura da aplicação. A arquitetura de referência descreve a arquitetura da linha de produto, sendo tida como não instanciada, pois não faz referência a componentes concretos. Por outro lado, a arquitetura do produto final reflete a arquitetura de uma aplicação específica, a qual é conhecida como arquitetura instanciada, uma vez que faz referência às implementações específicas de componentes que serão utilizadas. Para uma determinada linha de produto, pode existir uma única arquitetura de LPS e diversas arquiteturas de produto, cada uma refletindo uma versão específica de um produto gerado a partir da linha.

No que se refere à arquitetura de referência de linhas de produto foi identificada a necessidade de incluir dois novos conceitos para a Fractal ADL: componentes opcionais e grupos de componentes alternativos.Os grupos de componentes opcionais são componentes que podem ou não podem entrar na arquitetura do produto final, dependendo para a escolha de um componente as características selecionadas. Por exemplo, *zapper* é um termo usado para definir uma versão de um receptor de TV digital que não tem a capacidade de executar aplicações interativas transmitidas pela emissora de TV. Dessa maneira, se for preciso gerar um produto *zapper* numa linha de produto do middleware Ginga, não é necessário incluir o componente *ApplicationManager*, pois esse componente é responsável por gerenciar as aplicações interativas em execução no middleware. Dessa forma, percebe-se que o componente *ApplicationManager* é um exemplo de um componente opcional na arquitetura de referência. Na figura 4.6 é encontrado o elemento *component* usado para descrever esse componente na arquitetura de referência.

```
<component name="ApplicationManager" optional="true">
        <interface name="IApplicationDatabase"
                   role="server"
                   signature="AppManager.IApplicationDatabase"/>
</component>
```

FIGURA 4.6 – Componente *ApplicationManager*

Como pode ser visto na Figura 4.6 foi adicionado o atributo opcional para o elemento *component* original da Fractal ADL para indicar se o componente descrito é um componente opcional na arquitetura de referência. Esse atributo apresenta o valor "*true*" para o componente *ApplicationManager* com a finalidade de informar que esse componente é opcional na arquitetura da aplicação final.

O segundo conceito que precisou ser adicionado para a Fractal ADL foi o conceito de grupos de componentes alternativos. Com o objetivo de representar esse conceito, um novo atributo, chamado *alternative_group* foi criado para ser usado no elemento *component* da Fractal ADL. Com esse novo atributo é possível identificar um grupo de componentes alternativos na arquitetura. Conforme estudado na seção 4.1, a fractal ADL é capaz de representar componentes primitivos e compostos. Com relação aos componentes compostos pode-se dizer que eles são formados por outros componentes e apresentam informações referentes aos seus componentes internos e as amarrações entre esses componentes internos. Dessa forma, um componente composto pode ser utilizado na arquitetura de referência para encapsular um grupo de componentes. A figura 4.7 a seguir apresenta três componentes compostos alternativos na arquitetura de referência do Ginga.

```
<component name="Data1" alternative_group="data">
    <interface name="IControler" role="server" signature="Controller.IController"/>

    <component name="Controller" definition="Controller"/>
    <component name="DataProcessor" definition="DataProcessor"/>
    <component name="ApplicationManager" definition="ApplicationManager"/>

    <binding client="this.IController" server="Controller.IController"/>
        <binding client="Controller.IDataProcessor" server="DataProcessor.IDataProcessor"/>
        <binding client="Controller.IApplicationDatabase"
                 server="ApplicationManager.IApplicationDatabase"/>
</component>
<component name="Data2" alternative_group="data">
    <interface name="IControler" role="server" signature="Controller.IController"/>

    <component name="Controller" definition="Controller"/>
    <component name="DataProcessor" definition="DataProcessor"/>

    <binding client="this.IController" server="Controller.IController"/>
    <binding client="Controller.IDataProcessor" server="DataProcessor.IDataProcessor"/>
</component>
<component name="Data3" alternative_group="data" default="true">
    <interface name="IControler" role="server" signature="Controller.IController"/>

    <component name="Controller" definition="Controller"/>

    <binding client="this.IController" server="Controller.IController"/>
</component>
```

FIGURA 4.7 – Componentes alternativos

Na figura 7 são encontrados três componentes compostos pertencentes ao mesmo grupo alternativo de nome *data* definido pelo atributo *alternative_group*. Isso significa que apenas um dos componentes pode ser selecionado (baseado nas características do produto selecionadas) para fazer parte da arquitetura do produto final instanciada a partir dessa arquitetura de referência. Essa abordagem proporciona boa flexibilidade para o arquiteto da LPS na atividade de especificar os pontos de variação da arquitetura da linha de produto, uma vez que é possível ter qualquer número de componentes internos e qualquer número de declarações de ligações dentro de um componente composto. O arquiteto da LPS tem a flexibilidade de definir qualquer parte da arquitetura como opcional ou alternativo.

Nessa extensão foi possível representar aspectos específicos de arquiteturas de referência de linhas de produto através da inclusão de dois novos conceitos: componentes opcionais e grupos de componentes alternativos. As extensões não apresentaram um nível de dificuldade alto para serem incorporadas e conseguiram atender as necessidades de representação arquitetural para LPSs.

3. FERRAMENTA PROPOSTA

Neste momento apresentamos a ferramenta de derivação, que constitui as etapas de projeto e realização de aplicação da LPS do *middleware* Ginga, as quais são construídas através de um *wizard* de derivação, divido em três etapas: Seleção de características desejadas, configuração da arquitetura de aplicação (seleção de componentes, adição de atributos) e geração de arquivos de descrição de arquitetura e componentes para a plataforma FlexCM.

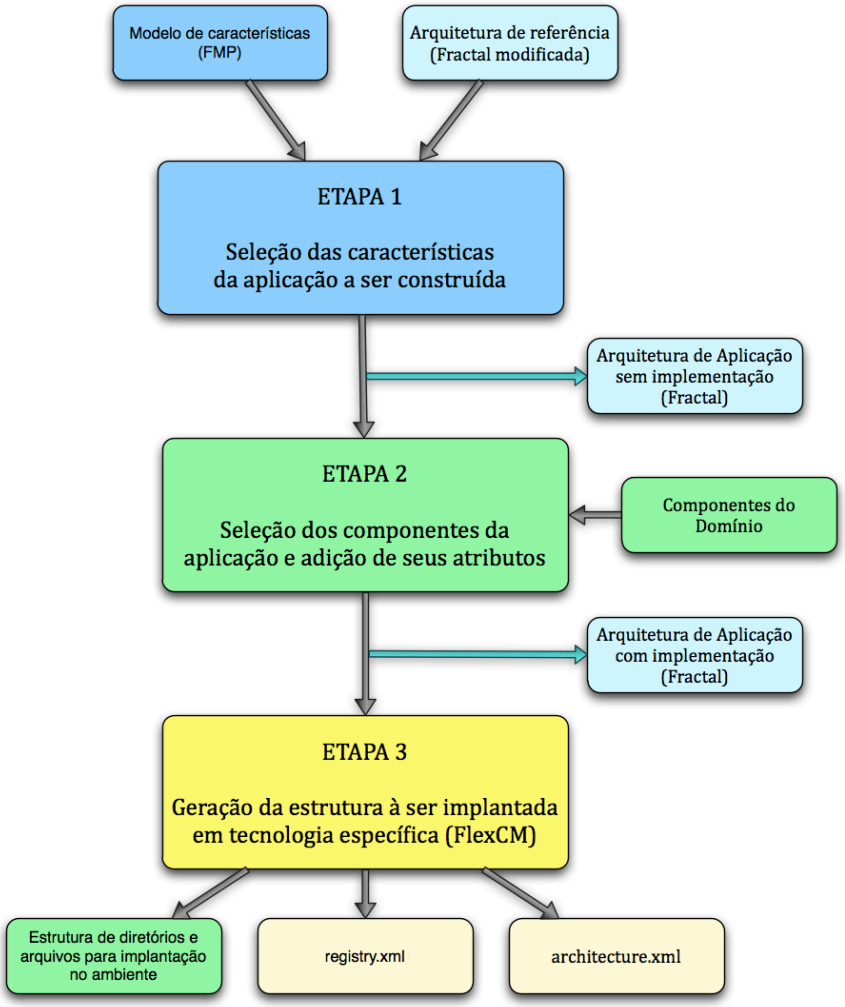

FIGURA 5.1 – Artefatos utilizados e gerados durante o processo de derivação

A figura 5.1 representa o fluxo de funcionamento conceitual da ferramenta de derivação. Na primeira etapa, são recebidos da engenharia de domínio a arquitetura de referência e o modelo de características do domínio, com todas as variantes que podem ser selecionadas ou não para a aplicação que está sendo gerada. Nesse momento, serão selecionadas as características desejadas e, a partir das seleções criadas, teremos a criação da arquitetura instanciada. Com essa arquitetura em mãos, temos o projeto de aplicação concebido.

A segunda etapa da derivação consiste em atrelar as implementações da realização de domínio na arquitetura instanciada criada anteriormente. Esse atrelamento consiste da seleção de implementações para os devidos componentes necessários para a realização da arquitetura instanciada, junto da adição de atributos que cada um dos componentes requisitar.

Ao final da segunda etapa, os componentes selecionados e a arquitetura instanciada serão configurados na tecnologia de execução utilizada. No caso, serão criados os arquivos de descrição de arquitetura e de componentes do FlexCM (*registry.xml* e *architecture.xml*) de maneira a tornar o software executável no ambiente de execução do *middleware*. Após gerados esses ativos, teremos a união de todos os recursos necessários para a aplicação numa estrutura de pastas de modo que a aplicação possa, de fato, ser implantada no ambiente de execução.

A ferramenta funciona como um *plug-in* para a plataforma Eclipse. Isso traz algumas vantagens em relação a produtividade, já que essa plataforma possui uma grande quantidade de ferramentas de apoio à construção de *plug-ins*, tanto da Eclipse Foundation quanto de terceiros, que é o caso de alguns frameworks utilizados na construção dessa ferramenta (incluindo na etapa de domínio, não apresentada no escopo deste trabalho): FMP (*Feature Modelling Plugin*), F4E (*Fractal for Eclipse*), EMF (*Eclipse Modelling Framework*) são exemplos de frameworks utilizados neste trabalho.

3.1. WIZARD DE DERIVAÇÃO

A proposta de interface sugerida nesta seção é de um *wizard*, que nada mais é que telas ordenadas que possuem quatro ações básicas: Avançar, Voltar, Cancelar e Finalizar. A primeira tela não possui o botão de Voltar, pois não possui nenhum lugar para retroceder, assim como a tela final não possui um botão de Avançar, tendo em seu lugar um botão para Finalizar.

O *wizard* de derivação possui três telas, uma referente a cada uma das etapas citadas na seção 5.1.

A primeira tela, referente a etapa 1, possui um campo para seleção da linha de produto em que a ferramenta vai trabalhar, e a segunda opção está relacionada a configuração desejada para aquela linha de produto, sendo que cada configuração desta representa as características selecionadas para a arquitetura de referência ser transformada em arquitetura de aplicação.

Essas configurações e características selecionadas são gerenciadas pelo engenheiro de domínio, que por ser parte da engenharia de domínio, não faz parte do escopo deste trabalho. A figura 5.2 representa esta primeira tela.

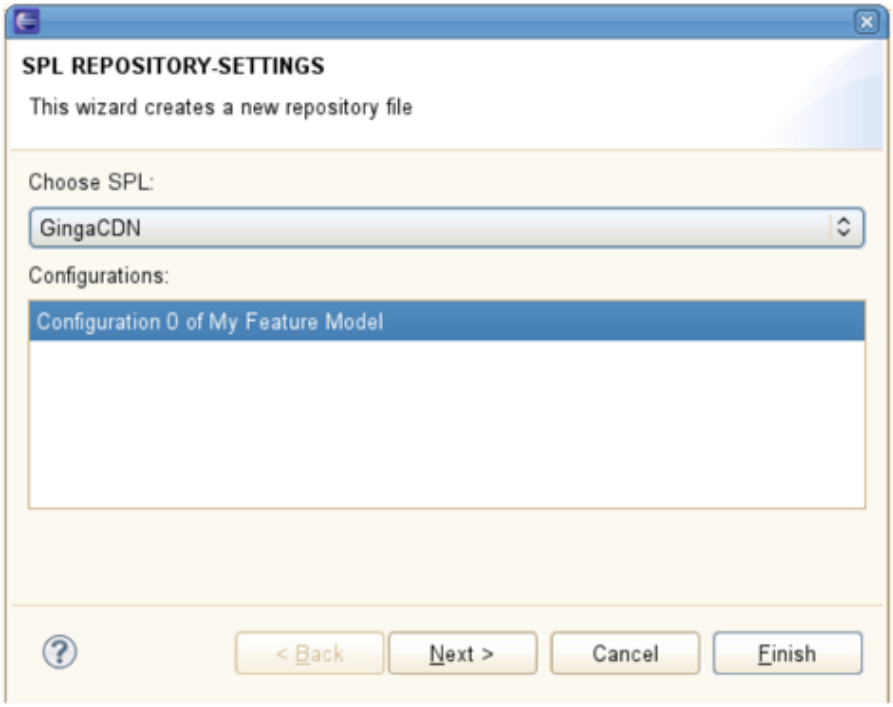

FIGURA 5.2 – Primeira tela do *wizard* de derivação

A partir desta seleção, de acordo com a figura 5.1, é criada a arquitetura de aplicação, mas ainda sem suas implementações e configurações adicionadas. Vale

ressaltar que, apesar de ser representada uma saída de uma arquitetura descrita em Fractal ADL na figura 5.1 após a etapa 1, ela não é serializada. Logo, este artefato apresentado na figura serve apenas para mostrar qual a saída da etapa 1, mesmo que ela ainda não seja uma saída da derivação.

Com a arquitetura de aplicação em mãos, é dado início à segunda etapa da derivação. Esta etapa consiste de selecionar as implementações dos componentes gerados pela realização de domínio para as interfaces disponibilizadas e a adição de atributos a esses componentes selecionados, caso necessitem. A figura 5.3 mostra a interface gerada para essa etapa.

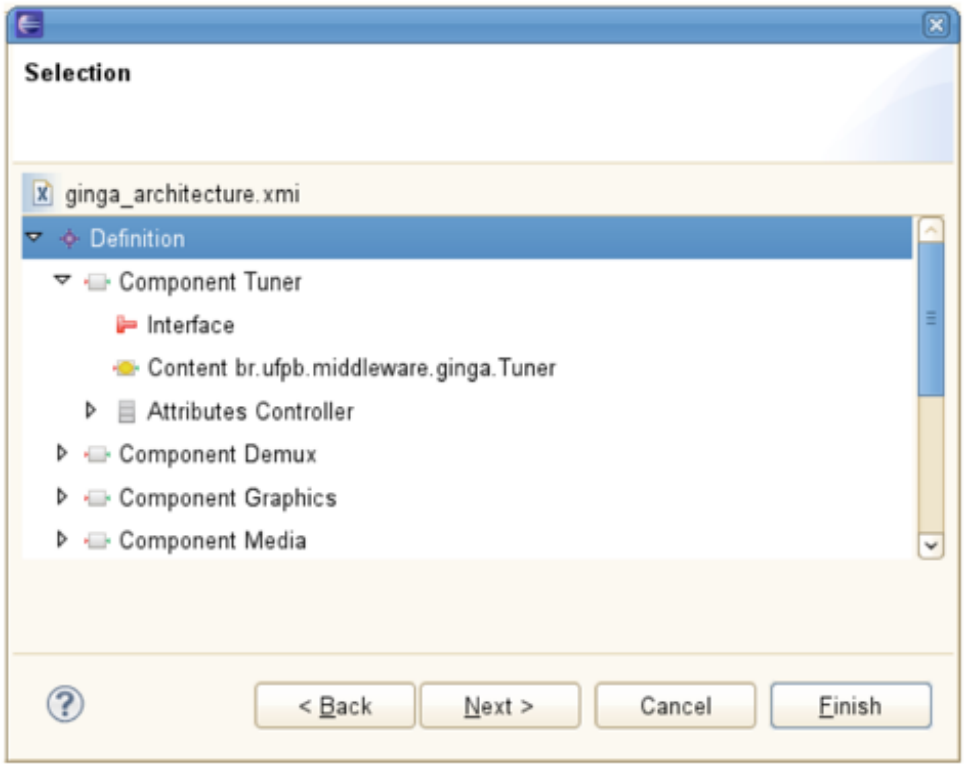

FIGURA 5.3 – Segunda tela do *wizard* de derivação

A arquitetura de aplicação descrita pela Fractal ADL é representada na tela através de uma árvore, pois segue a mesma maneira que o modelo é representado, que será abordado com detalhes na seção 5.3.

O nó *Content* representado é onde é adicionada a implementação do componente. O nó *Attributes Controller* é onde é adicionados os parâmetros necessários de funcionamento do componente. Este nó possui vários filhos *Attribute*, onde são adicionados o nome do atributo e o valor dele.

Assim como na primeira etapa, a segunda etapa também possui uma saída que não é saída da derivação, mas que diferente do caso anterior, possui um papel mais importante na personalização deste software. A saída desta etapa que é a arquitetura de aplicação com implementação, traz à tona a necessidade da etapa três: a criação de configuração da aplicação para a plataforma específica de execução. Nesta plataforma específica, podem ser colocados diferentes plataformas de execução, desde que funcionem com as implementações criadas e possuam suporte ao modelo de componentes da aplicação.

No caso desta ferramenta, a terceira etapa consta na geração de arquivos de configuração do ambiente de execução FlexCM (arquivos *registry.xml* e *architecture.xml*). Estes arquivos que são gerados serão unidos aos arquivos de implementação dos componentes e bibliotecas e no final temos a estrutura de diretórios e arquivos apresentada na figura 5.4.

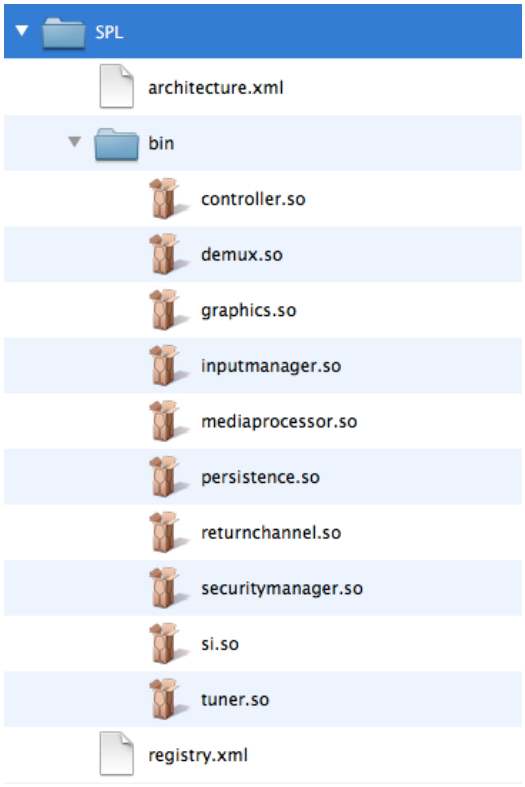

FIGURA 5.4 – Estrutura de implantação para o FlexCM

O caminha para onde esta estrutura é salva é escolhido na terceira etapa, através da janela apresentada na figura 5.5. Um simples campo de texto é preenchido com o caminho de destino.

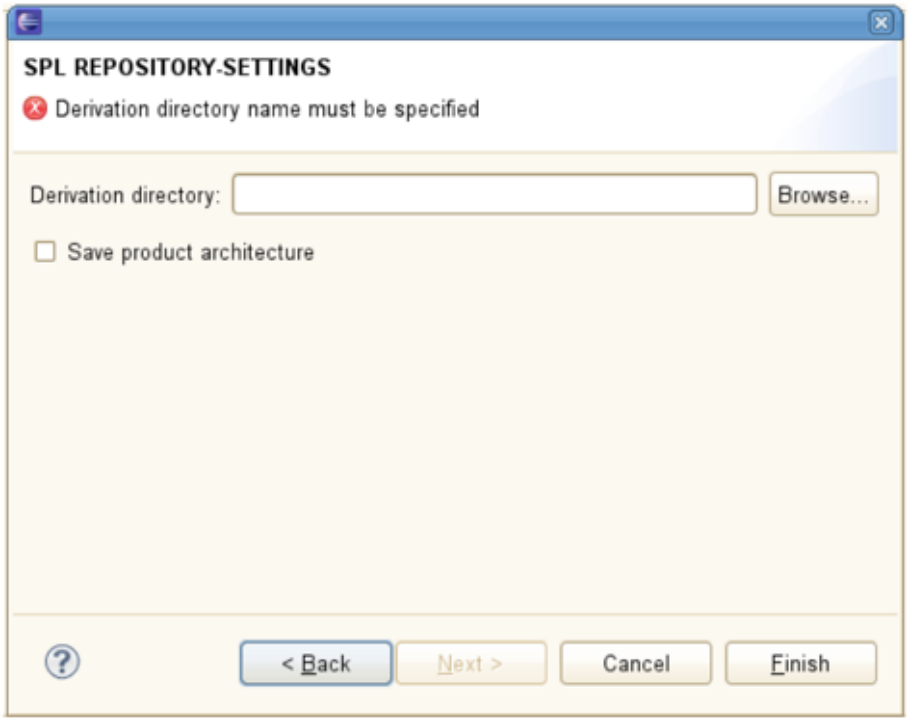

FIGURA 5.5 – Terceira tela do *wizard* de derivação

Há uma opção adicional onde a arquitetura do produto gerado pode ser salva, que faz com que, nas configurações do repositório, seja adicionada uma nova configuração com os atributos e implementações utilizados nesta derivação, de modo a facilitar na hora de derivar o produto caso seja distribuído para vários lugares diferentes.

3.2. PONTOS IMPORTANTES DA IMPLEMENTAÇÃO DA FERRAMENTA

Esta ferramenta foi implementada como um plug-in do Eclipse de maneira que alguns frameworks dessa plataforma pudessem auxiliar com tarefas de interface de usuário bem convenientes, como copiar e colar, alguns recursos visuais exclusivos da plataforma e fácil manipulação de alguns modelos utilizados na aplicação que serão abordados em breve.

A estrutura arquitetural básica do software foi construída através do padrão MVC (Model-View-Controller), que separa a lógica de negócio, o modelo de dados e a visualização de maneira à tornar o código mais legível e de mais manutenabilidade [7]. O controle gerencia o modelo e a visão, instanciando e atualizando, enquanto a visão apenas lê o controle para gerar visualização de seus dados.

Foram utilizados três frameworks para dar suporte a construção da ferramenta: um para construção da interface (Eclipse JFace) e outros dois para manipulação de modelos (EMF e F4E). Ao detalhar estes frameworks, mostraremos também como foram utilizados durante a construção desta ferramenta.

3.3. ECLIPSE JFACE

O JFace é um framework de construção de janelas bem semelhante ao popular Swing, da Sun Microsystems. Apesar de bem semelhantes, a opção por utilizar o JFace vem de que ele possui uma ótima integração com a plataforma Eclipse, de forma a facilitar o trabalho de construção da aplicação nessa plataforma.

O JFace possui um framework de *wizards* (já explicado anteriormente) que facilitam na construção de páginas de *wizard*. Através da extensão da classe Wizard geramos uma conversação constituída por várias extensões de WizardPage, que quando ordenadas corretamente, nos vem a idéia das etapas apresentada na figura 5.1 e realizada nas figuras 5.2, 5.3 e 5.5. Cada uma dessas três figuras é uma extensão da WizardPage. As hierarquias de classe de Wizard e WizardPage são mostradas na figura 5.7.

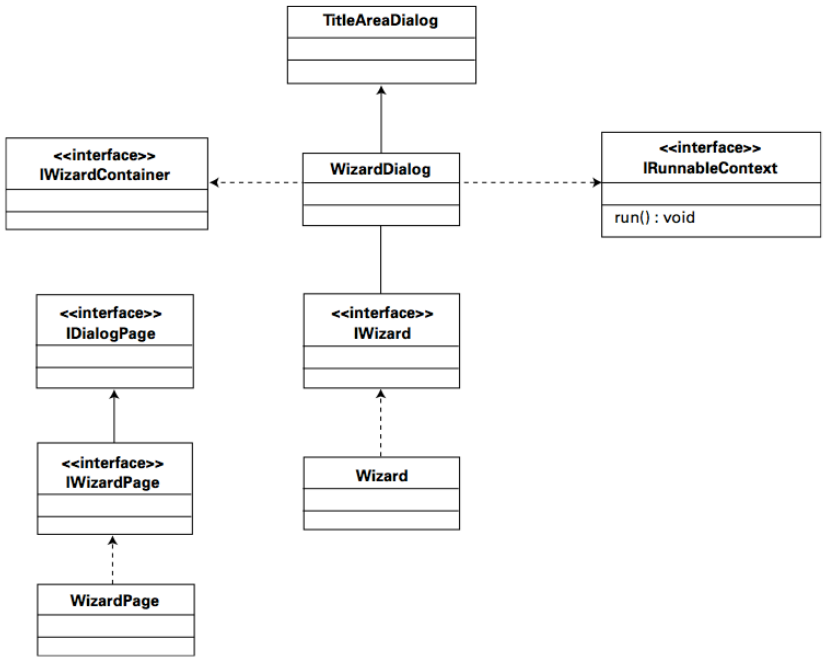

FIGURA 5.7 – Hierarquia de Classe de Wizard e WizardPage

Na figura 5.7 apresentamos as classes IDialogPage e TitleAreaDialog são simples páginas, que são inseridas dentro de um *widget*, um elemento de tela composto por

um *frame* e *canvas*, onde o *canvas* é a parte onde o framework renderiza as janelas e o *frame* a janela gerenciada pelo sistema operacional. Nessas extensões, Wizard e WizardPage adicionam os seus métodos para controle de paginação e cancelamento ou finalização da conversação.

Nas extensões realizadas pela nossa ferramenta, são apenas adicionados os elementos de tela ao *canvas* necessários para a visualização de cada etapa descrita nas figuras 5.2, 5.3 e 5.5, como *comboboxes*, *text fields*, estruturas de árvores, *checkboxes*, entre outros recursos.

3.4. ECLIPSE MODELLING FRAMEWORK (EMF)

O Eclipse Modelling Framework nada mais é que um, como seu próprio nome diz, *framework* para modelos no Eclipse. Ele é um *framework* com enfoque em MDD (Model Driven Development), que unifica Java, XML e UML [1], como mostrado na figura 5.8.

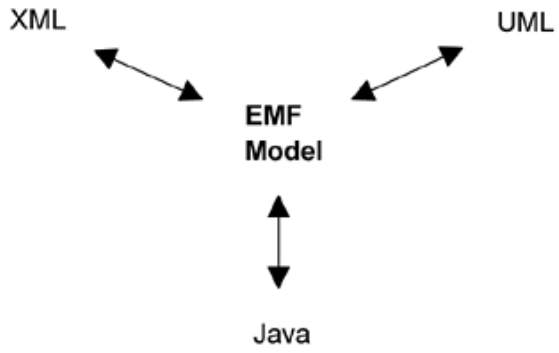

FIGURA 5.8 – Modelo EMF e serializações possíveis

Esse modelo do EMF é chamado de meta-modelo *ECore*. Esse meta-modelo, como a definição da palavra explica, é um modelo do modelo que será utilizado na aplicação, de maneira que, apos gerado o meta-modelo, poderemos serializar a instancia do modelo para objetos Java ou um arquivo XML qualquer facilmente. A partir disso, para serializar os arquivos de configuração do FlexCM (*registry.xml e architecture.xml*) foi criado um meta-modelo *ECore* do EMF.

3.5. FRACTAL FOR ECLIPSE (F4E)

Por utilizarmos o modelo de Fractal ADL para gerenciar as arquiteturas de referência e aplicação, a *The Fractal Project* desenvolveu uma ferramenta para gerência de modelos Fractal ADL de maneira visual, adicionando recursos de tela que facilitam na construção desses modelos, como *drag-and-drop* e recursos de recorte, cópia e colagem, assim como, apesar de não utilizado, oferece recursos gráficos, para que se possa construir visualmente uma arquitetura Fractal.

Essa ferramenta foi iniciada através de um meta-modelo *ECore*, como do EMF, mas criado como um plug-in utilizando o *framework* Eclipse GMF (*Graphical Modeling Framework*), que suporta a criação de modelos gráficos (formas geométricas, ligações com linhas, etc.) para uma interface mais amigável.

O que pode ser destacado do Fractal for Eclipse para o projeto é o aproveitamento do meta-modelo da Fractal ADL criado para o plug-in F4E para a modelagem de nossa arquitetura, já que ele pode ser completamente encaixado na estrutura arquitetural do *middleware* Ginga facilmente, através da extensão de sua definição.

4. CONSIDERAÇÕES FINAIS

O projeto GingaCDN necessita de uma ferramenta de derivação de produtos para a sua linha de produto do *middleware* do sistema brasileiro de televisão digital.

A partir desta necessidade foi construída a ferramenta apresentada neste trabalho. Esta ferramenta tem o intuito de resolver dois problemas: a questão da automatização da geração de código específico da plataforma de execução, junto com sua estrutura de implantação e uma proposta de interface amigável para a ferramenta.

Vale salientar que esta ferramenta possui mais recursos relacionados à engenharia de domínio, mas que não são parte do escopo deste trabalho.

Apesar da ferramenta ter sido construída para o *middleware* Ginga, partes de suas etapas podem ser aproveitadas para utilização em outros sistemas, alterando apenas os modelos de entrada apresentados na figura 5.1. Como as etapas de construção estão bem coesas, pode-se ainda trocar a terceira etapa apresentada na figura 5.1 de forma que implantações para outras tecnologias específicas podem ser adicionadas, desde que a arquitetura do sistema possa ser descrita através da linguagem de descrição de arquitetura Fractal ADL.

A interface proposta nos traz lembrança de instaladores de softwares comumente utilizados em grande parte das aplicações distribuídas a usuários não-especialistas em implantação de softwares. A idéia de construir no formato de *wizard* traz semelhanças com instaladores como o InstallShield®, Microsoft Installer®, entre outros. Porém, por ter sido construído como um *plug-in* do Eclipse, possui a necessidade desta plataforma ser instalada, que poderia ser removida caso os frameworks auxiliares provessem suporte aos recursos fornecidos fora da plataforma.

Alguns recursos adicionais podem ser adicionados a esta ferramenta, como a capacidade do engenheiro de aplicação realizar personalizações na escolha das características da arquitetura de referência, possibilitando a criação de novas configurações pelo arquiteto da aplicação.

Por ser bem desacoplado, pode-se substituir a terceira etapa da derivação, fazendo com que, caso mude a tecnologia específica de implantação, possamos alterar a geração da configuração para outras plataformas.

REFERÊNCIAS

[1] Budinsky, F. et al. Eclipse Modeling Framework: A Developer's Guide, Addison-Wesley, 2003.

[2] Clements, P., Northrop, L., Software Product Lines Practices and Patterns, Addison-Wesley, Boston, 2002.

[3] Eclipse Modeling – EMF – Home, Disponível em: <www.eclipse.org/modeling/emf>. Acesso em: Junho de 2010.

[4] Fractal – ADL Tutorial, Disponível em: <fractal.ow2.org/tutorials/adl/index.html>. Acessado em: Maio de 2010.

[5] Fractal – Documentation, Disponível em: <fractal.ow2.org/documentation.html>. Acessado em: Maio de 2010.

[6] Fractal for Eclipse Website, Disponível em: <fractal.ow2.org/f4e/>. Acessado em: Junho de 2010.

[7] Gamma, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[8] Ginga Digital TV Middleware Specification,Disponível em: <www.ginga.org.br>. Acesso em: Junho de 2010.

[9] Kulesza, R. et al. Ginga-J: Implementação de Referência do Ambiente Imperativo do Middleware Ginga. Simpósio Brasileiro de Sistemas Multimídia e Web, a ser publicado.

[10] Kulesza, R. Lívio, A. Relatório Técnico RT02 – Arquitetura do Middleware Ginga com definição de interfaces do Ginga-J e Ginga-CC relacionados componentizados. Projeto GingaCDN, Programa CTIC, Rede Nacional de Ensino e Pesquisa, Brasil.

[11] Leclercq, M., Ozcan, A. E., Quéma, V., Stefani, J. B. Supporting Heterogeneus Architecture Descriptions in an Extensible Toolset. 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 2007.

[12] Medvidovic, N. Taylor, R. N. A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering, 26(1), 2000.

[13] Meyer, M. Lehnerd, A. The power of Product Platforms. Free Press, 2007.

[14] Miranda Filho, S. et al. FLEXCM - A Component Model for Adaptive Embedded systems. In: 31ST Annual IEEE COMPSAC. IEEE Computer Society. Beijing, China, 2007.

[15] Miranda Filho, S. et al. Automating Software Product Line Development: A Repository-Based Approach. 36th EUROMICRO Conference on Software Engineering and Advanced Applications, to appear.

[16] Pohl, K. Böckle, G. Van der Linden, F. Software Product Line Engineering: Foundations, Principles and Patterns. Springer, 2005.

[17] Scarpino, M. Holder, S. Ng, S. Mihalkovic, L. SWT/JFace in Action. Manning, 2005.

[18] Szyperski, C., et al. Component Software – Beyond Object-Oriented Programming. 2nd ed. ACM Press, 2002.