Integração de Frameworks de Software

Alvaro Cesar Pereira Barbosa
alvaro@inf.puc-rio.br
Carlos José Pereira de Lucena
lucena@inf.puc-rio.br

PUC-Rio.Inf.MCC 02/2000 Janeiro, 2000

Resumo

Um Sistema de Gerência de Bancos de Dados Heterogêneos (SGBDH) é uma camada de software que tem por objetivo viabilizar o acesso aos dados de sistemas de bancos de dados autônomos, heterogêneos e distribuídos, necessitando de flexibilidade para adaptar-se a diferentes ambientes e tipos de sistemas componentes. No DI/PUC-Rio tem sido desenvolvido o Projeto HEROS, um SGBDH orientado a objetos, tendo sua arquitetura definida usando a técnica de frameworks. Para o avanço do projeto, faz-se necessário a pesquisa buscando uma melhor alternativa para a integração dos frameworks componentes do Framework HEROS^{fw}. Este trabalho apresenta um estudo sobre patterns para integração de componentes e sobre integração de frameworks de software.

Palavras-Chave: Frameworks, Patterns, Bancos de Dados, Sistema de Gerência de Bancos de Dados Heterogêneos, Interoperabilidade.

Abstract

Heterogeneous Database Management System (HDBMS) has been proposed as one of the solutions to the integration of autonomous, heterogeneous and distributed databases. The HDBMS needs flexibility to adapt itself to different environments and component systems. In the Computer Science Department at PUC-RIO, Brazil, it has been developed the HEROS project, an object oriented HDBMS whose architecture was defined using the frameworks techniques. In order to the advancement of this project, it is necessary a research about a better alternative to the components framework integration of the HEROS^{fw} Framework. This work presents a study about some patterns to the components integration and also about software frameworks integration.

Keywords: Frameworks, Patterns, Databases, Heterogeneous Databases Management Systems, Interoperability.

1 - Introdução

O ambiente computacional atual contem sistemas de hardware e software distribuídos, heterogêneos e autônomos, havendo uma necessidade de infra-estrutura tecnológica para suportar a cooperação dos serviços fornecidos através de hardware e software, de forma rápida, flexível e de menor custo. De uma forma geral, os requisitos para construir sistemas que combinem recursos e serviços heterogêneos podem ser encontrados desde baixo nível de interconectividade até um alto nível de interoperabilidade. Interconectividade simplesmente suporta comunicação de sistemas enquanto interoperabilidade permite, adicionalmente, que os sistemas cooperem entre si para a execução conjunta de tarefas [Ucho99].

Os Sistemas de Gerência de Bancos de Dados Heterogêneos (SGBDHs) surgiram da necessidade de se integrar sistemas de bancos de dados já existentes, autônomos, diferentes e distribuídos, através de redes de processamento. Estes bancos de dados já existentes são chamados de sistemas de bancos de dados locais ou componentes. Mais recentemente, SGBDHs não só estão sendo propostos para a integração de bancos de dados mas, de uma forma mais abrangente, para a integração de fontes de dados heterogêneas. De acordo com o tipo de integração existente entre os componentes de um SGBDH, estes podem ser classificados em sistemas com acoplamento forte ou com acoplamento fraco [SL90].

Um SGBDH deve ter a facilidade e a flexibilidade de poder integrar dados armazenados em diversas fontes, em ambientes distintos tanto de software quanto de hardware, atender o surgimento de novas formas de comunicação etc., sem a necessidade de se reestruturar os sistemas atualmente em operação. Faz-se, então, indispensável, se projetar e implementar SGBDHs que sejam flexíveis a ponto de poder integrar fontes de dados heterogêneas, sem que se tenha que implementar sistemas chamados "*heavy weight*" [BT95]. Sistemas "*heavy weight*" são projetados para atender a várias situações diferentes, fazendo com que, para uma certa aplicação, parte do código implementado se torne desnecessário o que poderá ocasionar aumento do custo do produto e de manutenção, maiores recursos de software e hardware e também problemas de performance. Já sistemas "*light weight*" são projetados para que somente o código necessário a uma aplicação específica seja disponível, tornando-se, então, sistemas enxutos.

Em outro aspecto, é prática comum se construir sistemas a partir de outros já existentes. A questão é que, apesar disso, quando sistemas são projetados, não é levada em consideração a possibilidade de sua reutilização no futuro nem a possibilidade de integração com outros sistemas. Assim sendo, novas técnicas devem ser pesquisadas e aplicadas na área de banco de dados [SSU96], de forma a baixar os custos e o tempo para se implementar e manter sistemas de gerência de bancos de dados, além de permitir que os mesmos sejam flexíveis o suficiente para poderem atender aos diferentes ambientes de hardware e de software, cada vez mais comuns. Dentre estas técnicas podemos citar: arquitetura de software, componentes, frameworks e patterns.

1.1 - Motivação

No DI/PUC-Rio, tem sido desenvolvido o Projeto HEROS (HeteRogeneous Object System), um SGBDH orientado a objetos. No HEROS, foram definidos seu modelo de dados integrador que é orientado a objetos, os mecanismos e sistemática para integração de esquemas, modelo e gerência de transação global e tratamento de heterogeneidades [Ucho99] [UML96]. Uma análise comparativa entre o HEROS e alguns projetos de SGBDHs é encontrada em [Silv94] [USM94].

No estágio atual, a arquitetura do HEROS foi definida usando a técnica de frameworks, com o objetivo de proporcionar uma alternativa mais flexível e extensível para a criação e utilização de SGBDHs. A técnica de frameworks oferece uma infra-estrutura adequada para o atendimento deste objetivo, pois um framework é um (sub)sistema de software de um domínio particular podendo ser ajustado para aplicações específicas. O Framework HEROS^{fw} é composto por outros seis frameworks: Interface Usuário, Regra, Consulta, Esquema, Transação e Comunicação [Ucho99]. Para o avanço do projeto, faz-se necessário a pesquisa com o objetivo de buscar melhores alternativas para a integração (interoperabilidade) dos frameworks componentes do HEROS^{fw} (framelets).

Esta monografia tem por objetivo apresentar um estudo sobre integração de frameworks de software. Serão abordadas técnicas de Patterns e Frameworks, que servirão como base para uma futura proposta de melhor integração dos frameworks componentes do HEROS^{fw}. A estrutura da monografia é a seguinte: inicialmente é apresentada uma introdução a SGBDHs, um resumo do HEROS^{fw} e os conceitos fundamentais de arquitetura de software, componentes, frameworks e patterns. Na seção 2 são apresentados alguns patterns para integração de componentes. Na seção 3 são discutidos os problemas, causas e possíveis soluções para a integração de frameworks. Na seção 4 são apresentadas as conclusões e trabalhos futuros.

1.2 - Sistemas de Bancos de Dados Heterogêneos

Um Sistema de Gerência de Bancos de Dados Heterogêneos (SGBDH) é uma camada de software que tem por objetivo viabilizar o acesso aos dados de sistemas de bancos de dados autônomos, heterogêneos e distribuídos, necessitando de flexibilidade para adaptar-se a diferentes ambientes e tipos de sistemas componentes. A criação de um SGBDH é uma tarefa complicada. Além do fato dos componentes serem autônomos e terem sido projetados independentemente, também existem as heterogeneidades causadas por diferenças no nível dos sistemas de bancos de dados, incluindo diferentes modelos de dados e linguagens de consulta, além de outros fatores como controle de transações, concorrência, falhas, recuperação, segurança e integridade, bem como as heterogeneidades dos sistemas não baseados em banco de dados.

Recentemente tem sido proposta a utilização da técnica de orientação a objetos para facilitar a construção de SGBDHs. A aplicação dos conceitos de orientação a objetos em arquitetura de sistemas, fornece um modelo natural para sistemas distribuídos, autônomos e heterogêneos. De acordo com esta arquitetura, chamada Arquitetura de Objetos Distribuídos, os recursos dos vários sistemas são modelados como objetos

enquanto os serviços são modelados como métodos. Nos sistemas de banco de dados, os recursos são as informações armazenadas no banco de dados e as facilidades fornecidas são métodos eficientes para recuperar e atualizar tais informações.

Um maneira de se modelar um sistema distribuído heterogêneo é através de uma coleção de objetos distribuídos que interagem entre si. Cada componente do sistema define uma interface de serviços e provê uma implementação destes serviços. Um cliente interage com o sistema heterogêneo emitindo requisições expressas em uma determinada linguagem. A gerência de objetos distribuídos é responsável por traduzir a requisição do cliente em termos de serviços disponíveis, envia a requisição para o sistema apropriado e, posteriormente, fornece a resposta na mesma linguagem usada pelo cliente.

O impacto da orientação a objetos na arquitetura de sistemas distribuídos heterogêneos se deve, também, aos esforços de padronização que estão sendo feitos. Embora a orientação a objetos seja uma solução para a interoperabilidade, a diversidade de modelos propostos vem complicando o problema. A longo prazo, estando os sistemas de banco de dados de acordo com um padrão adotado, facilitará a tarefa de integração. A curto prazo, os novos SGBDHs devem levar em consideração os padrões para a definição de suas interfaces.

Com relação à arquitetura, a tradicional de três níveis usada para descrever um sistema de banco de dados centralizado foi estendida para cinco níveis para atender à arquitetura de um SGBDH [PBE95]. O nível conceitual de cada banco de dados componente é chamado de *esquema local*, que é expresso no modelo de dados nativo do componente. Assim, os diferentes bancos de dados componentes podem estar expressos em diferentes modelos de dados. Mas, conforme dito anteriormente, tais componentes não precisam ser, necessariamente, sistemas de bancos de dados. Para facilitar o acesso, a maioria dos SGBDHs transforma os esquemas locais em um modelo de dados comum. O esquema derivado desta transformação é chamado de *esquema componente*. Cada componente participa do SBDH através da exportação de seu esquema componente, chamado *esquema de exportação*. Pode ocorrer de alguns SGBDHs não diferenciarem entre esquema componente e esquema de exportação, como no caso do HEROS. Neste caso, a arquitetura do esquema passa a ter apenas quatro níveis [USM94].

Um *esquema global* (ou federado) é criado pela integração de múltiplos esquemas de exportação. Finalmente, para customização ou por razões de controle de acesso, um *esquema externo* é criado para atender às necessidades de um grupo específico de usuários ou de aplicações (visões). Diferentes tipos de SGBDHs são criados por diferentes formas de integração dos esquemas de exportação. Um ambiente *não federado* assume não integrar os esquemas de exportação. Um exemplo de um ambiente não federado seria o acesso a todos os bancos de dados componentes através de uma linguagem de consulta. Um modelo *federado* [SL90] assume a integração dos esquemas de exportação para criar um esquema global. Sistemas de bancos de dados federados podem ser classificados com base na distribuição da integração. São *centralizados* se suportam apenas um único esquema federado. No caso de *descentralizados* cada componente constrói seu próprio esquema federado através da integração de seus esquemas locais com esquemas de exportação de algum outro componente.

Transformar esquemas locais em um Modelo de Dados Comum (MDC) é essencial tanto para os ambientes federados quanto não federados. O MDC deve ser rico o suficiente para capturar as semânticas expressas ou implícitas nos esquemas locais, e simples o suficiente para facilitar a criação de um esquema global no modelo federado e poder permitir consultas a diversos bancos de dados em um ambiente não federado. Muitos SGBDHs usam o modelo de dados orientado a objetos como MDC, como no caso do HEROS. A transformação de esquema alivia os problemas que ocorrem devido ao fato do uso de diferentes modelos de dados. Infelizmente, os mesmos conceitos podem estar representados em diferentes bancos de dados e, além disso, devido à heterogeneidade, estes conceitos podem estar representados diferentemente. Assim, existem alguns tipos de conflitos que podem ocorrer entre os esquemas componentes, que independem do tipo de MDC que se está utilizando, como o caso de c*onflito de identidade*, *conflito de esquema* (c*onflito de nome* e c*onflito estrutural*), c*onflito semântico* e c*onflito de dados* [PBE95].

A construção e a utilização de SGBDHs deve ser a mais flexível possível, para poder atender a aplicações específicas, com maior rapidez, confiabilidade e flexibilidade, oferecendo melhor performance, qualidade e, possivelmente, menor custo [MPLB98]. Estes são objetivos do Framework HEROS^{fw}.

1.3 – Framework HEROS^{fw}

O framework HEROS^{fw} possibilita a instanciação de SGBDHs para permitir que um conjunto de sistemas de bancos de dados heterogêneos sejam integrados em uma federação, de forma flexível, para que consultas e atualizações sejam realizadas de forma transparente à localização dos dados, aos caminhos de acesso e a qualquer heterogeneidade existente. O framework HEROS^{fw} é composto por seis outros frameworks: Interface Usuário, Regra, Consulta, Esquema, Transação e Comunicação [Ucho99].

O framework Interface Usuário é responsável pela comunicação do HEROS com o usuário final. Esta comunicação ocorre tanto na criação quanto no uso de federações. O framework Esquema é responsável por gerenciar os esquemas do HEROS, que usa uma arquitetura de esquemas de quatro níveis, conforme citado anteriormente. O framework Consulta é responsável por derivar, a partir da consulta global recebida e do esquema global, as sub-consultas que devem ser submetidas aos diversos sistemas de bancos de dados componentes. Uma outra responsabilidade do framework Consulta é produzir um plano de execução alternativo otimizado, com base no plano de execução e nos dados adicionais existentes sobre os sistemas de bancos de dados componentes do HEROS. O framework Transação submete consultas ou solicitações de execução de procedimentos globais aos respectivos sistemas locais, implementando operações de sistemas de gerência de banco de dados necessárias para a composição dos resultados parciais e armazenamento temporário dos resultados. O framework Comunicação é responsável pela comunicação física entre a federação e os sistemas componentes, detectando e reportando qualquer problema de conexão ao framework Transação. O framework Regra é responsável por prover o HEROS de um comportamento ativo, isto é, ter a capacidade de reagir automaticamente à ocorrência de determinados eventos.

Em [Ucho99] são apresentadas em maiores detalhes todas as características do framework HEROS^{fw} bem como os procedimentos para a criação e utilização de uma federação.

1.4 – Arquitetura de Software

Pode-se dizer que arquitetura de software teve início quando programas passaram a ser divididos em módulos, sendo os programadores os responsáveis pela composição dos módulos e pela funcionalidade do sistema como um todo [SG96]. Com a necessidade de se construir, cada vez mais, sistemas grandes e complexos, o problema não mais se restringe a escolher algoritmos e estruturas de dados e a se implementar módulos arbitrariamente, mas sim projetar e especificar a estrutura do sistema. Os problemas estruturais incluem a organização de um sistema como a composição de componentes, estruturas de controle global, protocolos de comunicação, acesso aos dados, sincronização, a funcionalidade e composição dos elementos de projeto e a distribuição física.

Arquitetura de software envolve a descrição de elementos dos quais os sistemas são construídos, a interação entre estes elementos, os padrões para guiar a composição e as restrições destes padrões. A arquitetura de um sistema de software define o sistema em termos dos componentes computacionais e das interações entre estes componentes. O projeto de um sistema pode ser construído em níveis. Em cada nível são encontrados componentes primitivos e compostos, regras de composição para a construção de componentes não primitivos e as regras de comportamento para fornecer as semânticas do sistema. O software também tem seus níveis de projeto, como o nível de arquitetura (associação dos componentes), nível de código (algoritmos e estruturas de dados) e nível de execução (mapeamento de memória, alocação de registradores, ...).

Segundo a definição de [BMR+96] arquitetura de software é a descrição dos subsistemas e componentes de um sistema de software e os relacionamentos entre eles. Subsistemas e componentes são tipicamente especificados em diferentes visões para mostrar as propriedades funcionais e não funcionais relevantes de um sistema de software. Um bom projeto de arquitetura é sempre um fator determinante para o sucesso de um software. A questão é como escolher uma arquitetura apropriada para um problema específico. Uma base arquitetural para a composição de componentes reutilizáveis auxiliaria muitos dos problemas de construção de software. Uma das propostas, mais amplamente aceita na área de orientação a objetos, é a de se usar frameworks para domínios específicos, que poderiam ser instanciados para produzir novos produtos no domínio.

1.5 - Componentes

Além dos problemas estruturais vistos anteriormente, os sistemas de software necessitam cada vez mais de serem abertos e distribuídos. Tais sistemas são abertos não apenas com relação a conexão a redes e a interoperabilidade com plataformas de hardware e software heterogêneos, mas, principalmente, em termos de atender à evolução e alteração dos requisitos. Sistemas abertos devem garantir esta propriedade

em três modos: **topologia** (redes configuráveis), **plataforma** (heterogeneidade de software e hardware) e **evolução** (alteração dos requisitos) [ND95]. Tais sistemas devem ter a propriedade de trocar componentes e, dinamicamente, conectá-los com aplicações em execução. Ligação dinâmica significa que o software pode ser configurado de acordo com as necessidades dos usuários, ou que aplicações em execução podem, dinamicamente, colaborar para trocar informação.

Em ambientes tradicionais de desenvolvimento de software, as várias fases para se construir uma aplicação são bem definidas e distintas: primeiro se escreve uma coleção de módulos, depois estes são compilados e por fim tem-se a fase de *link-edição*. Para se ter ambientes de composição mais flexíveis, este bem estabelecido esquema de desenvolvimento de software tem de ser revisto: mais informação da fase de compilação deve estar disponível em tempo de execução, e não somente aquelas da *link-edição*. Isso não quer dizer que o futuro dos componentes de software será, necessariamente, em linguagens totalmente interpretadas, mas que sistemas abertos e flexíveis necessitam lidar com componentes em várias formas possíveis, variando de código fonte a código de máquina, através de diversas representações intermediárias, parcialmente compiladas e otimizadas. Por exemplo, em sistemas orientados a objetos, a compilação estática de um método é algumas vezes complicada, pois deveriam ser previstos os contextos onde ele seria chamado. Na verdade, se o método é compilado em tempo de execução, então mais informação será conhecida sobre o contexto.

Sempre que componentes são integrados para executar uma tarefa comum, existe um contrato implícito entre eles sobre os termos de colaboração. Para ser capaz de verificar a corretude de uma configuração, os contratos necessitam ser explícitos para serem comparados no caso de discrepância. Um exemplo de contrato entre componentes é o esquema de um sistema de banco de dados que especifica as condições sobre as quais um banco de dados pode ser acessado e quais condições devem ser respeitadas pelos programas que executam transações sobre ele. Um componente tem que ser projetado para ser usado em uma combinação com outros componentes e não de forma isolada [Gogu86] [Pree97]. Componentes têm de ser projetados para reutilização e ser parte de uma arquitetura (*framework*). Funções, macros, procedimentos, *templates* e módulos são exemplos de componentes. Componentes em frameworks podem padronizar interfaces e código genérico para vários tipos de abstrações de software, além de poderem, também, ser de outros tipos de entidades e não somente software, como especificações, documentações, dados de teste e aplicações exemplo.

Segundo a definição de [BMR+96] um componente é uma parte encapsulada de um sistema de software, tem uma interface e serve como um bloco para a construção da estrutura de um sistema. Ao nível de linguagem de programação, componentes podem ser representados como módulos, classes, objetos ou um conjunto de funções relacionadas. Mas componentes podem ser de outra natureza, como o pattern Broker que dependendo da sua implementação, pode ser uma biblioteca interligada ou um processo separado. Um relacionamento define uma conexão entre componentes, podendo ser estático ou dinâmico. Relacionamentos estáticos são mostrados diretamente no código fonte e estão relacionados à localização dos componentes dentro da arquitetura. Relacionamentos dinâmicos tratam de conexões temporais e interação dinâmica entre componentes, não sendo facilmente visíveis da estrutura estática do código fonte.

1.6 – Frameworks e Patterns

Dentro da comunidade de orientação a objetos, arquitetura de software recebeu uma forma particular de ser representada, através de frameworks. A técnica de *framework* oferece toda uma infra-estrutura adequada para permitir flexibilidade, pois sendo um sub-sistema de software de um domínio específico, pode ser ajustado para aplicações individuais. Um *framework* consiste em uma grande estrutura (arquitetura) que pode ser reutilizada, para a construção de um novo sistema, o que representa, no contexto de SGBDHs, uma nova federação [Ucho99]. Um framework define uma arquitetura para uma família de sistemas, fornecendo partes predefinidas para sua construção e quais partes que devem ser adaptadas para uma função específica. Em um ambiente orientado a objetos, um framework é composto de classes abstratas e concretas e a sua instanciação consiste de composição de herança de classes.

[Pree96] apresenta um *framework* como sendo formado por ***frozen spots*** e ***hot spots***. Frozen spots definem a arquitetura global de um sistema, seus componentes básicos e os relacionamentos entre eles, ficando imutáveis em qualquer instanciação do framework. Já os *hot spots* são partes específicas para cada sistema, sendo projetados para serem genéricos e poderem ser adaptados de acordo com as necessidades da aplicação. Assim, um framework reutiliza código pois permite a construção de uma aplicação a partir de uma biblioteca de componentes, que podem ser facilmente compostos uns com os outros pois eles utilizam interfaces comuns.

Segundo [FS97] um framework é uma aplicação reusável, semi completa que pode ser especializada para produzir aplicações customizadas. Frameworks garantem modularidade por encapsular detalhes de implementação voláteis junto com interfaces estáveis. As interfaces estáveis fornecidas pelos frameworks garantem reusabilidade por definir componentes genéricos que podem ser reaplicados para criar novas aplicações. Um framework garante extensibilidade por fornecer métodos *hook* explícitos que permitem às aplicações estenderem suas interfaces estáveis. Métodos *hook* separam sistematicamente as interfaces e os comportamentos de um domínio de aplicação das variações requeridas pela instanciação da aplicação em um contexto particular.

[MB97] define um framework orientado a objetos como sendo um conjunto de classes que embute um projeto abstrato de soluções para uma família de problemas relacionados. Em outras palavras, um framework é um projeto abstrato e uma implementação para uma aplicação em um dado domínio de problema. No processo de desenvolvimento de software baseado em frameworks são definidas as seguintes fases: a) Fase do desenvolvimento do framework, b) Fase do uso do framework (também denominada fase de instanciação do framework ou ainda fase do desenvolvimento da aplicação) e, c) Fase de manutenção e evolução do framework.

O uso de um framework representa um investimento que deve ser avaliado considerando o retorno esperado. Também deve ser avaliado o desenvolvimento de um novo ou o uso e adaptação de um já existente. Na verdade, frameworks devem sempre ser usados. Em uma era de desregulamentação e competição global, entretanto, torna-se demasiadamente caro e extremamente demorado desenvolver aplicações inteiramente *in-house* [FS97]. Aplicações com vida longa certamente necessitarão mudanças nos requisitos com o tempo, que podem ser mais facilmente providenciadas com o uso de

um framework. Também aplicações de vida curta, que tipicamente são construídas com maiores restrições de tempo, poderiam ser facilitadas com o uso de um já existente. Para o uso de frameworks devem ser considerados, entre outros, o tempo de investimento e aprendizado sobre ele, pois o desenvolvimento é uma atividade custosa e demorada. Um outro aspecto é que não devem ser usados frameworks ainda instáveis para várias aplicações, mas, por outro lado, eles não evoluem para a estabilidade sem serem aplicados em diferentes tipos de aplicações [ND95].

Patterns [BMR+96] [GHJV95] têm se popularizado na área de orientação a objetos como uma forma de reutilizar informação de projeto, descrevendo um problema a ser resolvido, uma solução e o contexto na qual esta solução funciona. Patterns documentam frameworks [John92] e ajudam a garantir o correto uso de sua funcionalidade. Aplicações que usam frameworks têm que estar em conformidade com o projeto e modelo de colaboração dos framework*s* e assim, então, frameworks também geram patterns nas aplicações que o usam. Na verdade, a partir do momento que frameworks são implementados várias vezes, eles passam a representar um tipo de pattern.

Quando usados em conjunto com patterns, bibliotecas de classes e componentes, frameworks de aplicações orientados a objetos [BMMB98] podem significativamente aumentar a qualidade do software e reduzir o esforço de desenvolvimento. Frameworks geralmente falham se não forem reconhecidos e resolvidos os seguintes desafios: esforço de desenvolvimento, curva de aprendizado, facilidade de integração e manutenção, validação e remoção de erros, eficiência e falta de padrões [FS97]. A visão original de reuso de software foi baseada em componentes, mas frameworks não são componentes de software como eles foram originalmente previstos. Frameworks são mais customizáveis que componentes e têm interfaces mais complexas [John97].

Frameworks e componentes são tecnologias diferentes, mas que se cooperam [John97]. Primeiro, framework fornece um contexto reusável para componentes, fornecendo, ainda, um modo padrão para que componentes possam manusear erros, trocar dados e invocar operações um para outro. Mas qualquer tipo de framework fornece interfaces padrão que possibilitam o reuso dos componentes existentes. Um segundo modo no qual frameworks e componentes trabalham juntos é que framework torna mais fácil desenvolver novos componentes.

Patterns têm recentemente se tornado um modo popular para reusar informação de projeto na comunidade de orientação a objetos. Um pattern descreve um problema a ser resolvido, uma solução e o contexto no qual a solução trabalha. Desde que frameworks têm sido implementados diversas vezes, eles representam um tipo de pattern também. Mas frameworks são mais que somente classes: eles são código também. Patterns em [GHJV95] são diretamente relacionados a frameworks em outro modo: eles foram descobertos examinando um número de frameworks e foram escolhidos como sendo representativo de software orientado a objetos reusável. Um único framework, usualmente, contem muitos patterns, ou seja, tais patterns são menores que frameworks. Patterns são os elementos micro arquiteturais de frameworks [John97]. Da perspectiva de frameworks de aplicação, patterns podem ser vistos como seus blocos construtores. Da perspectiva de patterns, um framework de aplicação pode ser visto como um pattern para um sistema de software completo, em determinado domínio de aplicação.

2 - Patterns

Quando um especialista trabalha sobre um problema em particular, normalmente não cria uma nova solução para cada caso, que seja diferente das soluções anteriores. Na verdade, sempre se recai em problemas similares que já tenham sido resolvidos, cujas soluções podem ser utilizadas para resolver novos problemas. Esta é a grande motivação do uso de patterns. Um pattern endereça um problema de projeto recorrente que está relacionado a situações específicas de projeto e apresenta uma solução para isso. Patterns documentam experiências de projetos existentes e bem sucedidos, não sendo criados artificialmente. Eles proporcionam um meio de se reusar experiência ganha no desenvolvimento de projetos anteriores e podem ser aplicados em diferentes áreas do conhecimento [BMR+96].

O mesmo ocorre no desenvolvimento de um sistema de software: um pattern trata um problema específico, recorrente no projeto ou na implementação. Um pattern descreve um problema a ser resolvido, uma solução e o contexto no qual esta solução funciona. A um pattern também é associado um nome, de uma ou duas palavras, que deve ser representativo da sua aplicabilidade. Patterns podem ser usados para construir arquiteturas de software com propriedades específicas e, por outro lado, muitos patterns são encontrados em arquiteturas de software.

São diversas as propriedades de patterns para arquitetura de software [BMR+96]:

- Um pattern aplica-se a um problema de projeto recorrente que manifesta em uma situação específica de projeto e apresenta uma solução;
- Patterns documentam experiências de projeto existentes e bem sucedidas;
- Patterns identificam e especificam abstrações que estão acima do nível de classes, instâncias ou de componentes;
- Patterns fornecem um vocabulário e um entendimento comum para princípios de projeto;
- Patterns são uma das soluções para a documentação de arquitetura de software;
- Patterns permitem a construção de software com propriedades definidas;
- Patterns auxiliam a construção de arquiteturas de software complexas e heterogêneas;
- Patterns auxiliam a administrar a complexidade de software.

2.1 – Classificação

[BMR+96] classifica os patterns em três categorias: patterns arquiteturais, design patterns (patterns de projetos) e idiomas. Neste trabalho serão abordados apenas os patterns arquiteturais e design patterns. Os idiomas, que são patterns de baixo nível específicos para uma linguagem de programação, não serão estudados neste trabalho. [GHJV95] classifica os design patterns de acordo com o objetivo, podendo ser de criação, de estrutura e de comportamento e ainda de acordo com o escopo, se são aplicados a classes e/ou a objetos. O enfoque de [GHJV95] é para programas orientados a objetos, sendo sempre referenciados se são aplicados a classes e/ou objetos. Já o

enfoque de [BMR+96] é o de patterns no contexto de arquitetura de software, envolvendo patterns arquiteturais, design patterns e idiomas, sendo aplicados a componentes de uma forma geral.

2.1.1 - Patterns Arquiteturais

Arquiteturas de software viáveis são construídas, de uma forma geral, a partir de princípios estruturais. Estes princípios são denominados patterns arquiteturais. Assim, um pattern arquitetural expressa um esquema fundamental de estrutura organizacional para sistemas de software, fornecendo um conjunto de subsistemas predefinidos, especificando suas responsabilidades e incluindo regras e orientações para organizar os relacionamentos entre eles.

Patterns arquiteturais são modelos para a construção de arquitetura de software. Eles especificam as propriedades estruturais do sistema como um todo para uma determinada aplicação e têm impacto sobre a arquitetura de seus subsistemas. A seleção de um pattern arquitetural é, assim, uma decisão de projeto fundamental quando do desenvolvimento de um sistema de software.

2.1.2 - Design Patterns

Os subsistemas de uma arquitetura de software, bem como os relacionamentos entre eles, usualmente consistem de diversas unidades arquiteturais menores chamadas de design patterns (patterns de projeto). Assim, um design pattern fornece um esquema para refinar os subsistemas ou os componentes de um sistema de software, bem como os relacionamentos entre eles. Design pattern descreve uma estrutura recorrente de componentes que se comunicam e que resolvem um problema geral de projeto dentro de um contexto particular. Muito embora design patterns sejam menores na escala que patterns arquiteturais, eles são independentes de uma linguagem de programação particular ou de um paradigma de programação.

Os patterns arquiteturais podem ser usados no início do projeto, os design patterns durante o processo de refinamento do projeto e os idiomas durante a fase de implementação. A maioria dos patterns para arquitetura de software permitem que problemas possam ser resolvidos com patterns menores. Patterns, normalmente, não existem isolados: podem ser combinados em estruturas mais complexas no mesmo nível de abstração. Isto acontece quando o problema original inclui mais forças que possam ser balanceadas por um único pattern. Neste caso, aplicando-se diversos patterns pode-se resolver o problema, pois cada um deles resolve um subconjunto particular de forças.

Na definição de [GHJV95], os design patterns de criação estão relacionados com a construção de objetos, os design patterns de estrutura com a composição de classes e/ou objetos e os design patterns de comportamento com o modo que classes e/ou objetos interagem e distribuem responsabilidades. Patterns de classes lidam com relacionamentos entre classes e suas subclasses estabelecidos através de herança e são estáticos, pois são definidos em tempo de compilação. Já os patterns de objetos lidam com relacionamentos de objetos, que podem ser trocados em tempo de execução e são mais dinâmicos.

2.2 – Alguns Patterns para Integração

Nesta seção serão apresentados alguns patterns arquiteturais e design patterns, a partir das referências [BMR+96] e [GHJV95] que, a nosso ver, podem auxiliar no processo de integração dos frameworks e na reestruturação do framework HEROS^{fw}.

2.2.1 - Adapter

Converte a interface de uma classe para uma outra interface, permitindo que classes trabalhem em conjunto, o que não seria possível devido a incompatibilidade de interfaces [GHJV95]. Adapter é também conhecido como *Wrapper*. A motivação para uso do Adapter é que muitas vezes uma classe não é reusável somente porque sua interface não combina com a interface do domínio específico que a aplicação requer. Uma das questões a ser considerada quando do uso do pattern Adapter é a quantidade de adaptação necessária que ele deve fazer. Na classificação de [GHJV95] Adapter é um design pattern de estrutura.

O uso do pattern Adapter se dá quando:

- Se quer usar uma classe existente e sua interface não combina com as necessidades atuais da aplicação;
- Se quer criar uma classe reusável que coopere com classes que não tenham interfaces compatíveis;
- Se necessita usar diversas subclasses existentes, mas se torna impraticável adaptar todas as suas interfaces. Neste caso, um objeto Adapter pode adaptar a interface da superclasse.

O pattern Bridge tem uma estrutura similar ao pattern Adapter, mas o Bridge tem uma intenção diferente: ele separa uma interface de sua implementação para que eles possam ser alterados fácil e independentemente. Um Adapter se propõe a trocar a interface de um objeto existente.

2.2.2 - Bridge

O pattern Bridge tem como finalidade separar uma abstração de sua implementação de forma que as duas possam ser modificadas independentemente [GHJV95]. Quando uma abstração pode ter uma das diversas possíveis implementações, o modo usual de acomodá-las é através do uso de herança. Uma classe abstrata define a interface para a abstração e subclasses concretas a implementam em diferentes modos. Mas este modelo nem sempre é flexível o suficiente. Herança liga uma implementação a uma abstração permanentemente o que provoca dificuldades para se modificar, estender e reusar abstrações e implementações independentemente. O Bridge resolve este problema colocando a abstração e implementação em hierarquias de classes separadas, permitindo a elas variar independentemente. Na classificação de [GHJV95] Bridge é um design pattern de estrutura.

O uso do pattern Bridge se dá quando:

- Se quer evitar uma ligação permanente entre uma abstração e sua implementação, por exemplo quando a implementação deve ser selecionada em tempo de execução;
- Tanto as abstrações quanto suas implementações devem ser extensíveis através de subclasses, de forma independente;
- Trocas na implementação de uma abstração não devem ter impacto sobre os clientes, ou seja, o código não deve ser recompilado;
- Se quer compartilhar uma implementação entre múltiplos objetos, mas de forma transparente para o cliente.

O uso do Bridge permite separar interface e implementação, permitindo que a implementação de uma abstração possa ser configurada em tempo de execução. Permite ainda aumentar a extensibilidade e trata de forma transparente detalhes de implementação de seus clientes. Já o pattern Adapter permite que classes não relacionadas trabalhem em conjunto e é geralmente aplicado em sistemas após serem reprojetados, enquanto o Bridge, por sua vez, é usado durante o projeto para permitir que abstrações e suas implementações variem independentemente.

2.2.3 - Broker

O pattern Broker estrutura sistemas de software distribuídos, separando os componentes que interagem entre si através de chamada de serviços remotos [BMR+96]. Um componente Broker é responsável por coordenar a comunicação, incluindo o envio de requisições, a transmissão de resultados e de exceções. O Broker reduz a complexidade envolvida no desenvolvimento de aplicações distribuídas, fazendo esta distribuição transparente para o desenvolvedor. Na classificação de [BMR+96] Broker é um pattern arquitetural.

O uso de Broker se dá quando:
- Componentes devem ser capazes de acessar serviços fornecidos por outros, através da chamada de serviços remotos com transparência de localização;
- Se necessita trocar, adicionar ou remover componentes em tempo de execução;
- A arquitetura deve esconder detalhes do sistema e da implementação dos usuários de componentes e serviços.

Construir um sistema de software complexo como um conjunto de componentes separados que interoperam, resulta em maior flexibilidade e facilidade de manutenção e alteração, comparado a uma aplicação monolítica. Particionando funcionalidades em componentes independentes, o sistema torna-se potencialmente distribuído e extensível, podendo garantir portabilidade e interoperabilidade. Uma aplicação que usa um objeto deve somente ver a interface oferecida pelo objeto, não sendo necessário saber nada sobre detalhes de sua implementação ou sua localização física.

Quando dois Brokers interoperam, Bridges podem ser usados como componentes opcionais para esconder detalhes de implementação. Por exemplo, se dois Brokers devem se comunicar, independentemente de pertencerem a diferentes redes e/ou sistemas operacionais, um Bridge constrói uma camada que encapsula todos estes detalhes específicos do sistema. Broker é uma estrutura de larga escala, não sendo, geralmente, usado em aplicações simples, mas sim naquelas que servem como uma

plataforma para um conjunto de famílias de aplicações, enviando e monitorando solicitações, sem observar quem enviou ou mesmo o conteúdo da solicitação. Por sua vez, se comparado ao pattern Mediator, este último encapsula a semântica das aplicações, checando o conteúdo das solicitações e a sua origem, para decidir então o que fazer.

2.2.4 - Composite

O pattern Composite compõe objetos em estruturas de árvore que representam uma hierarquia *parte-de* [GHJV95]. Composite permite clientes tratar tanto objetos individuais como composição de objetos da mesma maneira. Clientes usam uma interface para interagir com objetos na estrutura composta. Se o recipiente é um objeto simples, então a requisição é manuseada diretamente e se a estrutura for composta, então a requisição é enviada para seus componentes filhos, possivelmente executando operações adicionais. O pattern Composite define hierarquias de classes consistindo de objetos primitivos e de objetos compostos. Objetos primitivos podem ser compostos em objetos mais complexos, estes com outros, e assim sucessivamente. Clientes podem tratar estruturas compostas e objetos individuais da mesma maneira, simplificando o código e tornando mais fácil adicionar novos componentes. Na classificação de [GHJV95] Composite é um design pattern de estrutura.

O uso de Composite se dá quando:

- Se quer representar hierarquias de objetos *parte-de*;
- Se quer que clientes sejam capazes de ignorar as diferenças entre composições de objetos e objetos individuais, ou seja, clientes tratarão todos os objetos na estrutura composta da mesma maneira.

Alguns cuidados devem ser tomados no uso da implementação do Composite, de acordo com a aplicação, como por exemplo manter referencias explícitas dos componentes com seus pais, o compartilhamento de componentes, a ordenação dos filhos e qual a melhor estrutura de dados para o armazenamento de componentes.

O design pattern Whole-Part [BMR+96] também auxilia na agregação de componentes que juntos formam uma unidade semântica, encapsulando os componentes constituintes, organizando suas colaborações e fornecendo uma interface comum para sua funcionalidade. Mas, diferente do Composite, o acesso direto às partes não é possível.

2.2.5 - Facade

O pattern Facade fornece uma interface unificada para um conjunto de interfaces de um subsistema, fornecendo uma interface de alto nível que facilita o uso do sub-sistema [GHJV95]. A estruturação de um sistema em subsistemas auxilia a redução da complexidade. Como um objetivo de um projeto é minimizar a comunicação e dependências entre os subsistemas, a introdução de Facade fornece uma única e simplificada interface para maior facilidade do subsistema. Na classificação de [GHJV95] Facade é um design pattern de estrutura.

O uso do pattern Facade se dá quando:

- Se quer fornecer uma interface simples para um subsistema complexo;
- Existem muitas dependências entre clientes e as classes de implementação de uma abstração;
- Quando os subsistemas estão dispostos em níveis hierárquicos diferentes.

A comunicação com os subsistemas é realizada através do Facade que transmite as requisições para o subsistema objeto apropriado, facilitando o uso dos subsistemas. Pode ser necessário que o Facade transforme sua interface de acordo com a interface do subsistema. Com o uso de Facade, não é necessário o acesso aos subsistemas objetos diretamente. Facade auxilia estruturar um sistema em camadas e também as dependências entre objetos, podendo também simplificar o porte de sistemas para outras plataformas.

O design pattern Mediator é similar a Facade pois eles abstraem funcionalidade de classes existentes. Entretanto, o objetivo do Mediator é abstrair arbitrariamente a comunicação entre os subsistemas objetos, centralizando funcionalidade que não pertence a nenhum deles. Já o Facade simplesmente abstrai a interface para os subsistemas objetos, facilitando o uso, mas sem definir novas funcionalidades.

2.2.6 - Layers

O pattern *Layers* auxilia na estruturação de aplicações que podem ser decompostas em grupos de sub-tarefas onde cada grupo de sub-tarefas pertence a um particular nível de abstração [BMR+96]. O modelo em camadas é considerado melhor prática que implementar um bloco monolítico. Um exemplo bem conhecido de arquitetura em camadas é o modelo de sete camadas para protocolos de rede de computadores. Na classificação de [BMR+96] Layers é um pattern arquitetural.

O uso do pattern Layers se dá quando:

- Alterações no código fonte, sempre que possível, não devem alastrar pelo sistema, devendo estar confinado a um componente e não afetar os outros;
- Componentes devem ser trocados por implementações alternativas, sem afetar o restante do sistema;
- Responsabilidades coerentes devem ser agrupadas para auxiliar no entendimento e manutenção;
- Componentes complexos devem ser melhor decompostos.

Alguns cuidados devem ser tomados na divisão em camadas, como por exemplo definir os critérios de abstração para agrupar tarefas em camadas, especificar a interface para cada nível, especificar a comunicação entre camadas adjacentes e definir uma estratégia de tratamento de erro.

2.2.7 - Mediator

O pattern Mediator define um objeto que encapsula um conjunto de objetos que interagem [GHJV95]. Mediator promove acoplamento fraco por manter os objetos referindo-se entre si explicitamente, permitindo variar sua interação independentemente. Um mediador é responsável por controlar e coordenar as interações de um grupo de

objetos. Um projeto orientado a objeto encoraja a distribuição de comportamento entre objetos e, assim, tal distribuição pode resultar em uma estrutura de objeto com muitas conexões entre objetos onde, no pior caso, cada objeto termina conhecendo todos os outros. Na classificação de [GHJV95] Mediator é um design pattern de comportamento.

Embora particionar um sistema em muitos objetos geralmente garante reusabilidade, a proliferação de interconexões tende a reduzir o reuso novamente. O uso de muita interconexão diminui a possibilidade de um objeto trabalhar sem o suporte de outros, ou seja, o sistema age como se fosse monolítico. Assim, pode ser difícil alterar o comportamento do sistema pois o comportamento pode estar distribuído entre muitos objetos, como por exemplo, sermos forçado a definir muitas subclasses para customizar o comportamento do sistema.

O uso do pattern Mediator se dá quando:

- Um conjunto de objetos se comunicam de forma bem definida, mas de modo complexo, onde as interdependências resultantes não são estruturadas e são difíceis de entender;
- O reuso de um objeto é difícil porque se refere e se comunica com muitos outros objetos;
- Um comportamento que é distribuído entre diversas classes deve ser customizável sem o uso de várias subclasses.

Objetos enviam e recebem requisições do Mediator, que implementa um comportamento cooperativo roteando requisições entre os objetos apropriados. O Mediator troca complexidade de interação por complexidade no Mediator. Como o Mediator encapsula protocolos, ele se torna mais complexo que os outros objetos individuais, tornando-se monolítico e dificultando a manutenção.

3 - Frameworks

Como visto anteriormente, a vantagem do desenvolvimento baseado em frameworks é o alto nível de reuso no desenvolvimento da aplicação. Embora o desenvolvimento de um framework orientado a objetos requeira considerável esforço, os benefícios correspondentes durante o processo de desenvolvimento da aplicação geralmente justificam o esforço inicial. As principais vantagens de frameworks são o aumento do reuso e a redução do tempo de desenvolvimento de aplicações. Alguns dos problemas com frameworks já foram descritos. Por serem poderosos e complexos, são difíceis de aprender e de desenvolver. Isso acarreta o aumento do custo e requer melhores programadores que o desenvolvimento de aplicações normais [John97].

3.1 - Classificação dos Frameworks

Segundo [FS97] os frameworks de aplicação podem ser classificados em:

- *Frameworks de Infra-estrutura de Sistemas*: simplificam o desenvolvimento de sistemas de infra-estrutura portáveis e eficientes tais como sistemas operacionais, comunicação, interface com o usuário e linguagens de programação.

- *Frameworks de Integração (middleware)*: são usados para integrar aplicações distribuídas e componentes. Eles são projetados para garantir a capacidade dos desenvolvedores de software com relação a modularização, reuso e estender sua infra-estrutura de software para trabalhar em um ambiente distribuído. Tais frameworks representam um mercado próspero e estão rapidamente tornando-se mercadorias de primeira necessidade.

- *Frameworks de Aplicação Empresarial*: abordam grandes domínios de aplicação, tais como telecomunicações, manufatura etc. e são a pedra fundamental de atividades de negócios da empresa. Tais frameworks são caros para desenvolver e/ou comprar.

Frameworks também podem ser classificados pelas técnicas usadas para estendê-los, que variam desde frameworks caixa branca (white box) até frameworks caixa preta (black box). Os frameworks caixa branca dependem fortemente dos aspectos de linguagens orientadas a objetos como herança e ligação dinâmica para prover extensibilidade. Já os frameworks caixa preta suportam extensibilidade definindo interfaces para componentes que podem ser ligados a um framework através da composição de objetos. A funcionalidade existente é reutilizada definindo-se componentes que estão em conformidade com uma interface particular e interagindo estes componentes com o framework usando patterns como *Strategy* and *Functor* [FS97].

Framework caixa branca requer dos desenvolvedores de aplicação conhecimento íntimo de cada estrutura interna do framework. Já os frameworks caixa preta são estruturados usando composição de objetos e delegação em vez de herança. Como resultado, frameworks caixa preta são geralmente mais fáceis de usar e estender que os frameworks caixa branca. Entretanto, frameworks caixa preta são mais difíceis de desenvolver pois requerem que os desenvolvedores do framework definam interfaces e *hooks* que antecipem uma grande faixa de potencial casos de uso. No início de sua vida, o framework é concebido como um framework caixa branca, podendo evoluir até se tornar um caixa preta.

Hooks (ganchos) são pontos no framework que podem ser adaptados ou estendidos de alguma forma, por exemplo preenchendo parâmetros ou criando subclasses, com a finalidade de prover a funcionalidade da aplicação. *Hooks* demonstram onde e como o projeto pode ser mudado ou aperfeiçoado, tendo como foco as intenções de uso de um framework. Já os *hot spots* são pontos de flexibilidade em um framework onde podem ser colocados *hooks*. *Um hot spot* poderá conter vários *hooks*.

Atualmente, não existem padrões vastamente aceitos para projeto, implementação, documentação e adaptação de frameworks [FS97] [Font99]. Muitos especialistas em frameworks preferem os caixa preta aos caixa branca pois os frameworks caixa preta enfatizam relacionamentos dinâmicos de objetos, via patterns como *Bridge* e *Strategy* em vez de relacionamentos de classes estáticas. A documentação de frameworks é uma

atividade de alto custo e que falha na captura de papéis estratégicos e colaborações entre frameworks componentes. Diversos padrões certamente surgirão para o desenvolvimento, adaptação, interoperabilidade e padrões de integração de frameworks.

As habilidades requeridas para produzir frameworks com sucesso geralmente ficam retidas nas cabeças dos desenvolvedores especialistas. O desenvolvimento de aplicações serão incrementalmente baseados na integração de múltiplos frameworks. Entretanto, muitos frameworks de geração mais antiga foram projetados para extensão interna em vez de ser por integração com outros frameworks desenvolvidos externamente. Problemas de integração existem em diversos níveis de abstrações, desde problemas de documentação, distribuição e concorrência. Por exemplo, enquanto inversão de controle é um aspecto essencial de um framework, integrar frameworks com controles que não foram projetados para a inter-operação com outros frameworks é uma tarefa complexa. Estes problemas serão apresentados mais adiante.

Diversos trabalhos abordam sobre projeto e uso de frameworks: [BGK+97], [BMA97], [Bosch98], [CDSV97], [DMNS96], [DW99], [Eske99a], [IBM93], [IBM99], [Schm97] e [SF97].

3.2 – Evolução dos Frameworks

Os requisitos da aplicação mudam frequentemente o que pode acarretar mudanças no framework. Com a evolução dos frameworks, as aplicações que os utilizam devem evoluir também. As atividades de manutenção dos frameworks incluem alteração e evolução. A manutenção de um framework pode ocorrer de diferentes formas [MB98]: para reorganizar a estrutura interna em resposta a novos requisitos mas sem afetar a funcionalidade externa ou para alterar, estender e remover funcionalidades.

Com o uso de componentes, ocorre um aumento da flexibilidade e extensibilidade de um frameworks. Isto complica grandemente os testes dos módulos pois os componentes não podem ser validados isolados da sua instanciação específica. Nestes casos, é geralmente difícil distinguir falhas do frameworks de falhas no código da aplicação. Durante a customização dos componentes em um framework para uma aplicação particular, o número de possíveis fontes de erros cresce.

3.3 – Integração de Frameworks

Na integração de frameworks diversos problemas podem ocorrer. A questão é que o desenvolvimento tradicional de aplicações baseadas em framework assume que a aplicação é baseada em um único framework, que é estendido com código específico da aplicação. Entretanto, podem ser identificados no desenvolvimento de aplicações a necessidade de se fazer uso de múltiplos frameworks, que devem ser integrados para atender aos requisitos da aplicação, como em [Ucho99]. Experiências com integração de dois ou mais frameworks mostram claramente que eles são geralmente desenvolvidos para reuso por extensão com a escrita de novos códigos específicos para a aplicação e não para composição com outros componentes de software [FS97] [MB97]. Este enfoque sobre reuso através da extensão causa problemas quando da

tentativa de integrar frameworks. Segundo [MB97] os problemas mais comuns encontrados na integração de frameworks, bem como as soluções, são os relacionados abaixo, embora possam ser encontrados problemas adicionais.

3.3.1 - Controle

Diferente das aplicações tradicionais, naquelas baseadas em frameworks geralmente é ele que está no controle do processo, chamando o código da aplicação quando necessário. Na integração de dois frameworks, ambos esperam ter o controle e podem facilmente colidir entre si, causando problemas, pois trocas no controle podem afetar partes do framework. A solução deste problema é mais fácil se os frameworks foram fornecidos com código fonte que permita adaptá-los para sanear o problema. Entretanto, o controle no framework pode não estar localizado em uma única entidade, ou seja, estar distribuído no código do framework.

Além de intercalar os controles de processos dos frameworks adaptando seu código, cada framework pode estar associado a seu próprio controle de processo, levando a dois ou mais controles de processos estarem executando independentemente. Duas importantes desvantagens: Primeiro, todos objetos da aplicação que podem ser acessados por ambos frameworks necessitam ser estendidos com código de sincronização. Segundo, geralmente um framework necessita ser informado sobre um evento ocorrido no outro devido o evento ter larga relevância na aplicação.

Para a solução deste problema, três opções podem ser empregadas:

- Dar a cada framework seu próprio controle de processo em vez de intercalar os controles, com a necessidade de sincronização entre eles, o que é uma desvantagem;
- Encapsular cada framework com um *wrapper*, interceptando todas as mensagens enviadas e recebidas pelo framework;
- Remover os controles de processos e refazê-los de acordo com a aplicação específica, com a desvantagem de ter que acessar o código fonte de tais processos, que pode estar distribuído pelo framework.

3.3.2 - Componentes Legados

Um framework apresenta um projeto para uma aplicação em um domínio particular. Baseado neste projeto, pode-se construir uma aplicação concreta através da extensão do framework, por exemplo através da criação de subclasses a partir das classes do framework. Pode ser necessário incluir classes já existentes (legadas) na aplicação a ser integrada com o framework. Desde que um componente legado não será uma subclasse de uma classe do framework, poderá ocorrer conflito de tipos.

Uma alternativa é usar o design pattern *Adapter* para adaptação da classe, embora com a desvantagem de que, para todo elemento da interface que necessite ser adaptado, deverá ser definido um método que repassará a chamada para o método correspondente na classe legada. Além disso, no caso de adaptação de objeto, também todas as requisições que não requeiram adaptações terão de ser redirecionadas devido ao objeto adaptador intermediário, levando a um custo adicional na implementação. Uma outra solução é

alterar as partes do framework que se referem as classes que devem ser substituídas pelas classes legadas.

3.3.3 – GAP

Ocorre quando dois frameworks são integrados e a estrutura resultante não cobre todos os requisitos da aplicação. Este problema pode ser resolvido com um framework de interface adicional. No caso em que o framework que chamou falha em funcionalidade, um software mediador é necessário. O software mediador pode necessitar também tirar parte da funcionalidade oferecida pelo framework e repor estas partes com código específico da aplicação. Um problema é que este software mediador depende da versão corrente do framework podendo levar a problemas de manutenção mais complexos quando da troca de versão do framework.

Outras duas soluções para este problema: a primeira é estender a interface do framework chamado com a funcionalidade insuficiente através do uso de *wrapper*. A segunda é reprojetar e estender o código fonte para permitir um melhor domínio de cobertura do framework.

3.3.4 - Entidades Conflitantes

Quando dois frameworks que representam a mesma entidade do mundo real, com suas respectivas perspectivas necessitam ser integrados, estas duas representações podem ocasionar problemas na integração. Se as propriedades representadas são mutuamente exclusivas e não influenciam uma a outra, então a integração pode ser resolvida por herança múltipla. O problema é quando existem propriedades compartilhadas ou dependentes entre as representações, o que proporciona que uma integração destas representações será mais complexa.

Uma alternativa é usar agregação ao invés de herança o que resultará em uma classe agregada que tem a representação de cada framework como parte, além da representação da entidade do mundo real. O problema desta solução é que ambos frameworks têm de ser alterados e que uma considerável quantidade de implementação será requerida.

Uma outra alternativa que não requer alterações nos frameworks é criar subclasses em cada framework. O problema é que a representação do mundo real está dividida em duas classes. Neste caso, uma classe de agregação pode ser adicionada, tendo as duas subclasses como partes.

3.3.5 - Funcionalidade de Entidades

Algumas vezes a funcionalidade de uma entidade do mundo real tem de ser modelada através da integração com partes de funcionalidade de diferentes frameworks. Apenas a integração das respectivas classes dos frameworks ou o uso de herança múltipla não resultarão no comportamento desejado. Assim, será necessário estender as classes dos frameworks para atender aos novos requisitos da aplicação. Deve ser observado que, embora tais classes pudessem ser estendidas com tais comportamentos durante a fase de

projeto, a maioria dos frameworks não são projetados para serem integrados com outros frameworks, mas sim para serem estendidos.

Como possíveis soluções, pode-se fazer o uso de agregação ou herança múltipla, para compor as classes do framework com a funcionalidade requerida em uma única classe, ou o uso do design pattern *Observer*, onde a classe de domínio específico deve ser estendida com notificação de comportamento.

3.4 – Causas dos Problemas de Integração de Frameworks

O principal problema na fase de uso de um framework é desenvolver uma aplicação através do reuso de frameworks existentes. Estes têm de ser estendidos ou compostos com outros componentes de software para atender aos requisitos da aplicação em desenvolvimento. Três diferentes componentes podem ser usados para adaptação e composição: novo código fonte para aplicação específica, biblioteca de classes e outros frameworks. Toda aplicação desenvolvida baseada em framework requer o desenvolvimento de código específico da aplicação, mas neste caso sem os problemas de integração. Se uma biblioteca de classes ou um framework tem de ser integrado com outro framework, diversos problemas de integração podem ocorrer.

Uma das causas é chamada de coesão de framework. Ela ocorre devido ao fato que quando um framework é integrado com um componente, esta integração deve atender tanto ao comportamento do domínio específico da aplicação bem como estabelecer um correto comportamento de interação entre eles. A questão aqui é garantir que um componente desenvolvido separadamente possa atender a estes objetivos, sem efeitos colaterais. Uma outra causa é o domínio de cobertura de um framework pois ele não necessita cobrir todo o domínio. Quando de sua utilização, o mesmo deve ser estendido ou integrado a outro componente para atender aos requisitos da aplicação. Assim, a integração pode gerar sobreposição de entidades, o que pode causar mais esforço na integração do que simplesmente estender o framework com código novo.

Uma terceira causa é a intenção de projeto, ou seja, deve ser definido explicitamente pelo projetista se o framework pode ser usado apenas por extensão ou o que será necessário para a sua futura integração. Por fim uma outra causa é a dificuldade de acesso e a clareza do código fonte do framework, para extensão ou integração. Se o código fonte não está disponível, a solução é o uso de *wrapper*, mas que poderia causar problemas como: mais código, performance e dificuldade de estender comportamento.

4 – Conclusões e Trabalhos Futuros

A criação de um SGBDH é uma tarefa complicada. Além do fato dos componentes serem autônomos e terem sido projetados independentemente, também existem as heterogeneidades de hardware e software e as causadas por diferenças no nível dos sistemas de bancos de dados, incluindo diferentes modelos de dados e linguagens de consulta. Um SGBDH necessita de flexibilidade para poder mais facilmente adaptar-se a diferentes ambientes e tipos de sistemas componentes.

No caso do HEROS, sua arquitetura foi definida usando a técnica de frameworks, com o objetivo de proporcionar uma alternativa mais flexível e extensível para a criação e utilização de SGBDHs, considerando que frameworks oferecem uma infra-estrutura adequada para o atendimento deste objetivo podendo ser ajustado para aplicações específicas. A utilização de frameworks na implementação de SGBDs em geral e de SGBDHs, em particular, não foi encontrada na literatura. Para o avanço do projeto HEROS, faz-se necessário a pesquisa com o objetivo de buscar melhores alternativas para a integração (interoperabilidade) dos frameworks componentes do HEROS^{fw} (framelets).

Com este objetivo foram revistos os conceitos de componentes, patterns e frameworks e pesquisados patterns que possam auxiliar na integração de frameworks. Integração de frameworks é um assunto de pesquisa em aberto. Foram encontrados alguns trabalhos relacionados a integração de frameworks: [RJ96] apresenta patterns para o desenvolvimento de frameworks orientados a objetos, enfatizando a evolução de frameworks; [Eske99b] apresenta patterns para a integração de componentes, no contexto de desenvolvimento de software baseado em componentes e [MB97] trata dos problemas e suas causas na integração de frameworks e apresenta possíveis soluções. Mas não existem soluções bem definidas: elas dependem do domínio da aplicação, do problema a ser resolvido e dos frameworks envolvidos.

As contribuições deste trabalho foram:

- Apresentar alguns patterns que podem ser utilizados para integração de frameworks no HEROS^{fw}. A lista não é completa, mas retrata nossa visão da necessidade atual e que certamente será estendida no decorrer do projeto;
- Apresentar os problemas e suas possíveis causas na integração de frameworks, bem como algumas soluções gerais, mas que devem ser melhor avaliadas para sua utilização dentro das características específicas do projeto HEROS.

A partir deste trabalho outros poderão ser iniciados, para o constante avanço do projeto HEROS. Como futuro trabalhos, podem ser citados:

- Avaliar a aplicabilidade das técnicas apresentadas nesta monografia para a integração de frameworks no HEROS^{fw};
- Com base nos estudos efetuados rever a arquitetura do framework HEROS^{fw} .

Referências Bibliográficas:

[BGK+97] Baumer, D.; Gryczan, G.; Knoll, R.; Lilienthal, C.; Riehle, D. & Zullighoven, H. "**Framework Development for Large Systems**". Communications of the ACM, Vol. 40, No. 10, October 1997.

[BMA97] Brugali, D.; Menga, G. & Aarsten, A. "**The Framework Life Span**". Communications of the ACM, Vol. 40, No. 10, October 1997.

[BMMB98] Bosch, J.; Molin, P.; Mattsson, M. & Bengtsson, P. "**Object-Oriented Frameworks: Problems & Experiences**". R. Johnson (eds.), John Wiley, 1998.
http://www.ipd.hk-r.se/michaelm/papers/ex-frame.ps - 24/11/1999.

[BMR+96] Buschmann, F.; Meunier, R; Rohnert, H. et al. "**Pattern–Oriented Software Architecture – A System of Patterns**". John Wiley & Sons, 1996.

[Bosch98] Bosch, J. "**Specifying Frameworks and Design Patterns as Architectural Fragments**". Proceedings TOOLS ASIA'98. http://www.ide.hk-r.se/~bosch/ - 24/11/1999.

[BP98] Blaha, M. & Premerlani. "**Object-Oriented Modeling and Design for Database Applications**". Prentice-Hall Inc, 1998.

[BR87] Biggerstaff, T. & Richter, C. "**Reusability Framework, Assessment, and Directions**". IEEE Software, Vol. 4, No. 2, March, 1987.

[BT95] Batory, D.S. & Thomas, J. "**P2: A Light Weight DBMS Generator**". Technical Report TR-95-26, Department of Computer Sciences, University of Texas at Austin, August 1995. http://www.cs.utexas.edu - 10/04/1998.

[CDSV97] Codenie, W.; De Handt, K.; Steyaert, P. & Vercammen, A**. "From Custom Applications to Domais-Specific Frameworks**". Communications of the ACM, Vol. 40, No. 10, October 1997.

[Clin96] Cline, M. P. "**The Pros and Cons of Adopting and Applying Design Patterns in the Real World**". Communications of the ACM, Vol. 39, No. 10, October 1996.

[DMNS96] Demeyer, S.; Meijler, T.D.; Nierstrasz, O. & Steyaert, P. "**Design Guidelines for 'Tailorable' Frameworks**". Communications of the ACM, Vol. 99, No. 10, October 1996.

[DW99] D´Souza, D. F. & Wills, A. C. "**Objects, Components and Frameworks with UML: the Catalysis Approach**". Addison Wesley, 1999.

[Eske99a] Eskelin, P. "**Layering Frameworks in Component-Baesed Development**". Pattern Languages of Programs Conference - PLoP´99, USA, August 1999. http://st-www.cs.uiuc.edu/~plop/plop99/proceedings/ - 21/10/1999.

[Eske99b] Eskelin, P. "**Component Interaction Patterns**". Pattern Languages of Programs Conference - PLoP´99, USA, August 1999. http://st-www.cs.uiuc.edu/~plop/plop99/proceedings/ - 21/10/1999.

[Font99] Fontoura, M.F.M.C. "**Uma Abordagem Sistematica para o Desenvolvimento de Frameworks**". Tese de Doutorado, PUC-Rio, Departamento de Informatica, Brasil, 1999.

[FS97] Fayad, M. E. & Schmidt, D. C. **"Object-Oriented Applications Frameworks**". Communications of the ACM, Vol. 40, No. 10, October 1997.

[GHJV95] Gamma, E.; Helm, R.; Johnson, R. & Vlissides, J. "**Design Patterns – Elements of Reusable Object-Oriented Software**". Addison-Wesley professional computing series, 1995.

[Gogu86] Goguen, J. A. "**Reusing and Interconnecting Software Components**". Computer, Vol. 19, No. 2, February 1986.

[IBM93] IBM, Java Education. "**Leveraging Object-Oriented Frameworks**". http://www.ibm.com/java/education/oobuilding/index.html - 24/11/1999.

[IBM99] IBM, Java Education. "**Building Object-Oriented Frameworks**". http://wwwibm.com/java/education/oobuilding/index.html – 24/11/1999.

[John92] Johnson, R. "**Documenting Frameworks using Patterns**" In Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'92), Canada, October 1992.

[John97] Johnson, R. "**Frameworks = (Components + Patterns)**". Communications of the ACM, Vol. 40, No. 10, October 1997.

[MB97] Mattsson, M. & Bosch, J. "**Framework Composition: Problems, Causes and Solutions**". Proceedings TOOLS USA '97. http://www.ipd.hk-r.se/michaelm/papers/frwkcomp.ps – 21/10/1999.

[MB98] Mattsson, M. & Bosch, J. "**Frameworks as Components: A Classification of Framework Evolution**". In Proceedings of NWPER´98 Nordic Workshop on Programming Environment Research, Ronneby, Sweden, August 1998, pp. 163-174. http://www.ipd.hk-r.se/michaelm/papers/nwper98.ps – 24/11/1999.

[MPLB98] Melo, R.N.; Porto, F.; Lima, F. & Barbosa, A. "**ECOHOOD: Constructing Configured DBMSs based on Frameworks**". XIII Simpósio Brasileiro de Banco de Dados, Maringá-PR, Brasil, 1998, pag. 39-51.

[ND95] Nierstrasz, O. & Dami, L. "**Component-Oriented Software Technology",** In *Object-Oriented Software Composition*, Chapter 1, edited by Nierstrasz, O & Tsichritzis, D. - Prentice-Hall Inc., 1995.

[OQ97] Odenthal, G. & Quibeldey-Cirkel, K. "**Using Patterns for Design and Documentation**".http://www.ti.et-inf.uni-siegen.de/Entwurfsmuster/artikelimport/index_ecoop.htm – 05/08/1999.

[ÖV99] Özsu, M. T. & Valduriez, P. "**Principles of Distributed Database Systems**". Second Edition, Prentice-Hall Inc, 1999.

[PBE95] Pitoura, E.; Bukhres, O. & Elmagarmid, A. "**Object Orientation in Multidatabase Systems**". ACM Computing Surveys, Vol. 27, No. 2, June 1995, pages 141-195.

[Pesc98] Pescio, C. "**Deriving Patterns from Design Principles**". The Journal of Object-Oriented Programming, Vol. 11, No. 6, October 1998.

[Pree96] Pree, W. "**Framework Patterns**". SIGS Books & Multimedia, 1996.

[Pree97] Pree, W. "**Component-Based Software Development - A New Paradigm in Software Engineering?**" Software-Concepts and Tools (18), Springer-Verlag, 1997.

[RJ96] Roberts, D. & Johnson, R. "**Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks**". Pattern Languages of Programs Conference - PLoP´96, USA, September 1996. http://st-www.cs.uiuc.edu/users/droberts/evolve.html – 21/10/1999.

[Schm97] Schmid, H. A. "**Systematic Framework Design by Generalization**". Communications of the ACM, Vol. 40, No. 10, October 1997.

[SF97] Schmidt, D.C. & Fayad, M.E**. "Building Reusable OO Frameworks for Distributed Software"**. Communications of the ACM, Vol. 40, No. 10, October 1997.

[SG96] Shaw, M. & Garlan, D. "**Software Architecture**". Prentice-Hall Inc., 1996.

[Silv94] Silva, S.D. "**Sistemas de Bancos de Dados Heterogêneos: Modelo de Gerência de Transações**". Tese de Doutorado, Departamento de Informática, PUC-Rio, Brasil, 1994.

[SL90] Shet, A.P. & Larson, J.A. "**Federated Database Systems for Managing Distributed, Heterogeneous and Autonomous Databases**". ACM Computing Surveys, Vol. 22, No. 3, September 1990, pages 183-236.

[SM98] Schmid, A. H. & Mueller, F. "**Patterns for Extending Black-Box Frameworks**". The Journal of Object-Oriented Programming, Vol. 11, No. 3, June 1998.

[SSU96] Silberschatz, A.; Stonebraker, M. & Ullman, J. "**Database Research: Achievements and Opportunities Into the 21st Century**". SIGMOD Record, Vol. 25, No. 1, March 1996, pages 52-63.

[SZ97] Silberschatz, A. & Zdonic, S. "**Database Systems – Breaking Out the Box**". SIGMOD Record, Vol. 26, No. 3, September 1997.

[Ucho99] Uchôa, E.M.A. "**Framework para Integração de Sistemas de Bancos de Dados Heterogêneos**". Tese de Doutorado, Departamento de Informática, PUC-Rio, Brasil, 1999.

[UML96] Uchôa, E.M.A.; Melo, R.N. & Lifschitz, S. "**Interoperabilidade em Sistemas de Bancos de Dados Heterogêneos**". Monografia (PUC-Rioinf.MCC45/96). PUC-Rio, Brasil, 1996, 31p.

[USM94] Uchôa, E.M.A.; Silva, S.D. & Melo, R.N. "**Modelo de Dados do HEROS - Sistema de Bancos de Dados Heterogêneos Orientado a Objetos**". IX Simpósio Brasileiro de Banco de Dados, São Carlos-SP, Brasil, 1994, pag. 142-156.