2
Fundamentos

2.1.
Matchmaking

Matchmaking pode ser entendido, de forma bem geral, como um processo de busca dos possíveis casamentos entre demandas e ofertas, em um dado domínio de aplicação. Esse processo é bastante diferente de simplesmente encontrar, para uma dada demanda, a mais perfeita oferta disponível, ou vice versa. Em contrapartida, seu comportamento deve ser o de procurar aquelas ofertas que, de alguma maneira, podem suprir a demanda, eventualmente propondo as melhores alternativas.

Em [Veit, 2001ab] é apresentada uma visão mais programática do conceito de *matchmaking*. Para ele, o processo de *matching* representa uma função que aceita como entrada um conjunto de descrições de ofertas e de demandas, provendo como resultado uma lista ordenada das *k* melhores ofertas, com relação a cada uma das demandas.

Um dos grandes problemas relacionado ao processo de *matchmaking* consiste no fato de que é irreal esperar que as descrições das ofertas e das demandas de um dado domínio sejam sempre equivalentes, ou mesmo pensar que sempre existe uma oferta que atenda exatamente às necessidades de uma demanda.

Levando isso em consideração, pode-se dizer que existe *matching* entre uma oferta e uma demanda quando suas descrições são “suficientemente similares”. Dessa forma, surge um novo problema que implica em especificar o que significa ser uma oferta “suficientemente similar” a uma demanda.

Numa interpretação extrema desse problema, pode-se considerar que a descrição de uma oferta é “suficientemente similar” a descrição de uma demanda quando a oferta satisfaz a todos os requisitos levantados para a

demanda. Porém, esse critério restritivo leva à situação anterior, que é indesejada uma vez que não se deve esperar que sempre haja descrições de ofertas e de demandas, num dado domínio de conhecimento, que atendam a essa propriedade.

De modo a acomodar uma definição menos restritiva de “suficientemente similar”, [Paolucci, 2002] aponta para a necessidade dos *matchmakers* suportarem a execução de *flexible matches*, ou seja, *matchings* que obtenham como resultado escolhas baseadas no grau de similaridade entre as descrições das ofertas e das demandas de um dado domínio de conhecimento.

Uma vez que não se deseja obter apenas as escolhas que apresentem descrições exatas, deve-se propor alguma forma de classificação entre os resultados obtidos pelo processo de *matching*, de modo que aqueles que apresentem descrições exatas sejam diferenciados dos que foram escolhidos baseados em critérios menos restritivos. Alguns trabalhos descritos na literatura apontam para algumas categorias de resultados de *matching*:

- *Total matching*: Todos os valores dos atributos da demanda estão de acordo com os valores dos atributos da oferta, ou vice-versa;
- *Potencial matching*: Alguns dos valores dos atributos da demanda não estão especificados na oferta;
- *Partial matching*: Alguns dos valores dos atributos da demanda não são atendidos por aqueles descritos para a oferta.

Essas categorias permitem classificar de forma imediata um resultado de *matching* entre uma oferta e uma demanda. Em [Di Noia, 2003ab] é discutida ainda a necessidade de ordenar os resultados enquadrados em cada uma dessas categorias. Para tanto, pode-se fazer uso dos “valores de similaridade”, obtidos durante o *matching* entre as descrições de ofertas e de demandas, como critério para a ordenação desses resultados.

Vários trabalhos apontam para a importância da linguagem utilizada na descrição das ofertas e das demandas, no contexto de *matchmaking*. Uma vez que o processo de *matching* é baseado em comparações, linguagens que permitam, de forma simples, especificar tipos, restrições e relações entre conceitos, agregam grande vantagem para o mesmo. Alguns trabalhos

apresentam linguagens proprietárias para descrição das instâncias do domínio, tal como LARKS [Sycara, 1999]. Porém, o que se observa é uma forte tendência ao uso de linguagens em que seja possível aplicar processos de inferência sobre as descrições feitas utilizando as mesmas. Exemplos de tais linguagens são aquelas baseadas em *Description Logics* [Nardi, 2002] e linguagens de marcação, baseadas em RDF [W3C, 2004ab].

Em [Trastour, 2001] são apresentados alguns requisitos relacionados à linguagem de descrição e ao processo de *matchmaking*. Esses requisitos são baseados na aplicação de RDF/RDFS [W3C, 2004abc] para descrição de serviços, em trabalhos realizados pelos autores. Apesar desses requisitos serem baseados em experimentos num domínio específico (*matching* de serviços), pode-se perceber que apresentam grande valor, podendo sua aplicação ser generalizada para qualquer domínio de aplicação:

- *Alto grau de flexibilidade e de expressividade*: A linguagem utilizada deve permitir total liberdade na descrição das ofertas e das demandas. Também devem ser possíveis descrições com diferentes graus de complexidade e de completude, ou seja, uma oferta pode ser bem descritiva em determinados pontos, mas também pode descrever outros menos especificamente, deixando os mesmos abertos para negociação a *posteriori*. Também é requerida da linguagem a capacidade de expressar dados semi-estruturados;
- *Suporte a tipos de dados e a herança*: A linguagem não deve restringir o processo de *matching* a comparações simples, baseadas em somente tipos primitivos. Deve ser permitida a composição dos mesmos em tipos mais complexos. Também é requerida da linguagem a capacidade de expressar hierarquias de tipos, possibilitando, assim, processos de *matching* mais complexos, baseados nesses relacionamentos;
- *Suporte à definição de restrições sobre os dados descritos*: A linguagem deve possibilitar a definição de ofertas e de demandas que estejam corretas e que sejam válidas. Uma maneira de alcançar tal requisito é através da definição de restrições sobre os atributos das mesmas;

- *Nível semântico de concordância*: Uma vez que o processo de *matching* tem como base a comparação entre instâncias de ofertas e de demandas, é necessário que elas compartilhem o mesmo nível semântico de descrição.

O processo de *matching* é necessariamente baseado em mecanismos complexos, que podem apresentar alto custo computacional [Paolucci, 2002]. Dessa forma, de maneira a aumentar a eficiência, esse processo deve adotar um conjunto de estratégias que permitam diminuir o número de comparações desnecessárias entre ofertas e demandas, aumentando a eficiência do *matchmaker* sem diminuir a precisão dos resultados obtidos pelo mesmo.

Diversos trabalhos descritos na literatura tratam sobre *matchmaking* aplicado aos mais variados domínios de aplicação. [Sycara, 1999] apresenta uma abordagem para *matching* de serviços em Sistemas Multi-Agentes, utilizando a linguagem LARKS. Em [Trastour, 2001] e em [Paolucci, 2002] também são apresentadas abordagens semânticas para *matching* de serviços, a primeira baseada em DAML+OIL [DAML+OIL, 2001] e a segunda em DAML-S [DAML-S].

Em [Ludwig, 2002] também é discutida uma abordagem para *matching* de serviços, porém em *Grid Environments*. Outros trabalhos [Li, 2003] [Pothipruk, 2002] [Lu, 2003] [Ouksel, 2004] [Constantinescu, 2003] [Decker, 1996] também apresentam abordagens similares para *matching* de serviços, relacionadas, principalmente, a ambientes de Sistemas Multi-Agentes e a descoberta de serviços semânticos.

[Veit, 2003ab] apresenta um *framework* para *matchmaking* (GRAPPA) que visa a facilitar a construção de algoritmos de *matching* através da disponibilização de funções que permitem retornar “valores de similaridades” para tipos de dados complexos. Nesse *framework* as instâncias do domínio estão descritas utilizando-se XML.

Em [Machado, 2004] e [Calì, 2004] são apresentadas abordagens para *matching* entre perfis de pessoas, sendo o primeiro, em particular, desenvolvido para ser aplicado em ambientes de computação e de colaboração móvel. Em [Raman, 1999] é discutida uma abordagem de *matchmaking* para *Distributed*

Resource Management, que é utilizada no *Condor High Throughput Computing System*. Essa abordagem utiliza uma linguagem proprietária e a solução é aplicável apenas ao domínio referido.

[Di Noia, 2003ab] apresenta um *matchmaker* para *e-comerce*, sendo sua abordagem baseada em *Description Logics*. Em [Zaremski, 1997] é apresentada uma abordagem para *matching* de componentes de *software*, cujo objetivo principal é determinar quando um componente pode ser substituído ou modificado para se ajustar aos requisitos de outro componente.

Por fim, [Cranefield, 1997] discute a utilização de *planning* e de *matchmaking* para interoperabilidade de informação em ambientes Multi-Agentes. Nessa abordagem, o *matchmaker* é responsável pela escolha de ferramentas mais adequadas em cada passo de execução do plano a ser executado.

Fazendo um apanhado geral acerca dos trabalhos sobre *matchmaking*, podem-se apontar as seguintes tendências nas pesquisas e nas aplicações desenvolvidas sobre esse tema:

- O tópico de pesquisa mais atraente no momento é a de *matching* de serviços, principalmente aqueles relacionados a *Web Services* semânticos e Sistemas Multi-Agentes;
- Utilização de uma linguagem rica semanticamente para descrever as instâncias do domínio sobre as quais o *matching* será realizado. Existe uma forte inclinação para utilização de linguagens baseadas em ontologias, tais como DAML-S;
- Utilização de algoritmos de *matching* que levam em consideração principalmente a relação de herança entre os conceitos do domínio de conhecimento;
- Adoção de algoritmos de *matching* que flexibilizem os critérios de comparação entre as instâncias do domínio de conhecimento, visando a obtenção de resultados relevantes e não apenas de escolhas exatas;
- Os resultados de *matching* devem ser classificados de acordo com alguma estrutura de categorias, bem como ordenados para cada uma das categorias propostas;

- As abordagens analisadas tratam, em sua grande maioria, do caso de *matching* mais simples entre instâncias de um domínio de conhecimento, ou seja, dada uma demanda deseja-se obter a oferta mais adequada disponível.

2.2. Ontologias

No sentido filosófico, pode-se entender ontologia como um sistema de categorias de uma determinada visão do mundo [Guarino, 1998]. No campo da Inteligência Artificial, o uso de ontologias foi desenvolvido para facilitar o compartilhamento e o reuso do conhecimento. Desde o início dos anos 90, ontologias têm se tornado um importante tópico de pesquisa, investigado por várias comunidades de pesquisa dentro da Inteligência Artificial, incluindo engenharia do conhecimento, processamento de linguagem natural e representação do conhecimento.

Mais recentemente, a noção de ontologia está se expandindo em áreas como integração inteligente de informação, sistemas de informação cooperativos, recuperação da informação, comércio eletrônico e gestão do conhecimento. A razão pela qual as ontologias estão se tornando tão populares se deve em grande parte ao que elas prometem: um entendimento comum e compartilhado de um domínio de conhecimento, que pode ser comunicado entre pessoas e sistemas computacionais heterogêneos e distribuídos.

Uma ontologia fornece uma conceituação (meta-informação) que descreve a semântica dos objetos, das propriedades dos objetos e das relações existentes entre eles num dado domínio de conhecimento [Chandrasekaran, 1999]. Ontologias são desenvolvidas para fornecer um nível semântico à informação de um dado domínio de forma a torná-la processável por máquinas e comunicável entre diferentes agentes (*software* e pessoas).

Muitas definições de ontologias foram propostas. Porém, uma das que melhor caracteriza a essência de uma ontologia é proposta em [Guarino, 1998]: No sentido filosófico, pode-se entender ontologia como um sistema de categorias de uma determinada visão do mundo. Assim sendo, este sistema não depende

de uma linguagem particular. Por outro lado, em seu uso mais freqüente em IA (Inteligência Artificial), ontologia faz referência a um artefato de engenharia, constituído de vocabulário específico usado para descrever uma certa realidade, acrescido de um conjunto de suposições com respeito ao significado esperado das palavras do vocabulário.

Esse conjunto de suposições fica normalmente na forma de lógica de primeira ordem, em que as expressões do vocabulário aparecem como predicados unários ou binários, chamados, respectivamente, de conceitos e de relações. No caso mais simples, uma ontologia descreve uma hierarquia de conceitos relacionados por meio de regras, enquanto que em casos mais sofisticados são adicionados axiomas adequados para expressar relacionamentos mais complexos e para restringir a interpretação desejada de seus conceitos.

2.2.1. Linguagens de Ontologias

O reconhecimento do papel chave que as ontologias devem ter no futuro da Internet levou à expansão das linguagens de marcação, de modo a facilitar a descrição de conteúdo e o desenvolvimento de ontologias para a *Web*. Exemplos de tais linguagens são: RDF (*Resource Description Framework*) e RDF *Schema* [W3C, 2004abcd]. Esta última, de modo particular, é reconhecida como uma linguagem de representação de conhecimento e de ontologias, permitindo a descrição de classes e de propriedades (relações binárias), de restrições de domínio e de contra-domínio, bem como de relações de subclasse e de sub-propriedade.

DAML+OIL [DAML+OIL, 2001] também é uma linguagem de ontologia projetada para descrever a estrutura de um dado domínio de conhecimento. Esta linguagem tem por base uma abordagem de orientação a objetos, em que a estrutura do domínio é descrita em termos de classes e de propriedades. DAML+OIL é uma linguagem proveniente da junção de duas outras linguagens: DAML (DARPA *Agent Markup Language*) [DAML, 2000] e OIL (*Ontology Inference Layer*) [OIL, 2000]. Dessa forma, DAML+OIL possui alto formalismo

semântico, bem como construtores provenientes da reconciliação entre as suas duas linguagens progenitoras.

A linguagem OWL [W3C, 2004efg] tem como objetivo a definição de ontologias para a *Web*, bem como de bases de conhecimentos a elas associadas. Nela, uma ontologia é definida por um conjunto de classes, de propriedades e de restrições. Uma ontologia descrita com OWL pode incluir relações de taxonomia entre classes, propriedades de tipo de dado (descrições de atributos dos elementos de classes), propriedades de objetos (descrições de relações entre elementos de classes), instâncias de classes e instâncias de propriedades.

Um conjunto de sentenças OWL em um sistema de dedução forma uma base de conhecimento. Essas sentenças podem incluir fatos sobre indivíduos que são membros de classes, bem como diversos fatos derivados, que não estão presentes explicitamente no texto de representação original da ontologia, mas vinculados, por meio de implicação lógica, na semântica da linguagem OWL. Essas regras podem, ainda, estar baseadas numa ontologia simples ou em múltiplas ontologias, que podem ser combinadas segundo mecanismos definidos pela própria linguagem.

2.2.2. API's de Acesso a Ontologias

Para que ontologias possam ser efetivamente utilizadas por aplicações, faz-se necessário que esses sistemas disponham de um conjunto de ferramentas que permitam um fácil acesso às informações descritas nas ontologias, independentemente da linguagem utilizada na sua construção, bem como garantam suporte à manutenção dessas informações através de operações de inserção, de remoção e de atualização de seus conceitos.

Jena [Jena] [Seaborne, 2002] [Carroll, 2003] é uma API Java [Java] para criação de aplicações que necessitam manipular ontologias. Desenvolvida pela Hewlett-Packard (HP), Jena define uma API com suporte a RDF, RDFS, DAML+OIL e OWL, que permite a leitura e a escrita de RDF, armazenamento de

ontologias de forma persistente e em memória, bem como suporte a consultas sobre as ontologias através da linguagem RDQL.

KAON (Karlsruhe Ontology and Semantic WebTool Suite) [KAON] [Volz, 2003] é um *framework* para construção de aplicações baseadas em ontologias. O foco principal dado ao projeto KAON é a escalabilidade e a eficiência nos processos de dedução sobre ontologias. Sua API define um conjunto de ferramentas para criação e manipulação de ontologias, que podem estar descritas utilizando uma linguagem própria da ferramenta, bem como RDF e OWL.

SNOBASE (*Semantic Network Ontology Base*) [Lee, 2003ab] é um *framework* que permite acessar ontologias armazenadas em arquivos ou disponíveis na *Web*, bem como criar, modificar, consultar e armazenar informações acerca das mesmas localmente. O objetivo principal desse *framework* é disponibilizar para aplicações a capacidade de manipular e de consultar ontologias sem a necessidade da mesma conhecer detalhes acerca de onde e como a ontologia será acessada, como a consulta será processada ou como os resultados serão obtidos. SNOBASE oferece suporte para as principais linguagens de ontologias (RDF, RDFS, DAML+OIL e OWL).

SOFA (Simple Ontology Framework API) [SOFA] é uma API Java para modelagem de ontologias. O modelo utilizado por SOFA não segue nenhum padrão definido por qualquer linguagem de ontologia, operando no nível de abstração dos conceitos relacionados com a ontologia, ao invés de adotar construtores que sejam específicos de uma determinada linguagem. SOFA permite a manipulação de ontologias descritas em RDF, RDFS, DAML+OIL e OWL, sendo também disponibilizada a funcionalidade de armazenamento das ontologias criadas a partir da API em bases persistentes.

2.2.3. Programação Orientada a Ontologias

Normalmente, as API's para acesso a ontologias provêm componentes de *software* que permitem representar as classes e as propriedades descritas em uma ontologia, bem como estruturas de dados para manipular as informações

dos modelos descritos baseados na mesma. Contudo, a representação dessas estruturas pode ser dinâmica ou estática.

Na representação dinâmica, a API pode representar todo seu conjunto de construtores dos conceitos de uma ontologia e de seus modelos numa linguagem própria, independente dos construtores de uma linguagem de ontologias particular. Na representação estática, todos os construtores estão baseados numa ontologia específica. Dessa forma, a API disponibiliza para as aplicações uma representação fiel de todas as classes e propriedades descritas na ontologia em questão.

O trecho de código da

Figura 1 apresenta um exemplo da abordagem de representação dinâmica, tendo sido retirado da documentação do *framework* Jena. Nesse exemplo, que usa um modelo baseado na ontologia VCARD [VCARD], o objetivo é determinar o nome de uma pessoa cujo *e-mail* é “amanda_cartwright@example.org”.


```
DAMLModel model = ... // code that loads the VCARD ontology
                      // and some data based on that ontology

DAMLClass vcardClass = (DAMLClass) model.getDAMLValue(vcardBaseURI + "#VCARD");

DAMLProperty fnProp = (DAMLProperty) model.getDAMLValue(vcardBaseURI + "#FN");

DAMLProperty emailProp = (DAMLProperty)
    model.getDAMLValue(vcardBaseURI + "#EMAIL");

Iterator i = vcardClass.getInstances();
while( i.hasNext() )
{
    DAMLInstance vcard = (DAMLInstance) i.next();

    Iterator i2 = vcard.accessProperty(emailProp).getAll(true);
    while(i2.hasNext())
    {
        DAMLInstance email = (DAMLInstance) i2.next();
        if(email.getProperty(RDF.value).getString().equals(
                "amanda_cartwright@example.org"))
        {
            DAMLDataInstance fullname = (DAMLDataInstance)
                vcard.accessProperty(fnProp).getDAMLValue();
            if(fullname != null)
                System.out.println("Name: " + fullname.getValue().getString());
        }
    }
}
```

Figura 1 – Acessando ontologias com base na abordagem de programação genérica

Como pode ser visto, os tipos das variáveis do exemplo são classes que um programador pode utilizar para escrever qualquer programa para acesso a ontologias descritas em DAML+OIL. Nesse caso, elas foram utilizadas para descrever uma rotina de consulta a uma ontologia VCARD, mas não existe nenhuma relação entre essas classes genéricas e aquelas definidas na ontologia

em questão. Deve-se perceber também que os nomes das classes e das propriedades aparecem como *strings*. Além disso, os valores das propriedades devem ser explicitamente convertidos para o tipo adequado, tal como acontece no código que imprime o nome da pessoa sendo procurada.

Em [Goldman, 2003] é apresentado um gerador de bibliotecas de classes baseadas numa ontologia, ou seja, ele discute um processo pelo qual a partir de uma dada ontologia pode-se obter um conjunto de classes e de métodos que representam estaticamente a estrutura da ontologia. O trecho de código da

Figura 2 apresenta um exemplo discutido no referido trabalho em que o objetivo é o mesmo do exemplo anterior, porém utilizando o paradigma de programação com representação estática.

```
Dim model as VcardOntology.model
Dim vc as Vcard
for each vc in model.vcards
    dim eaddr as String
    for each eaddr in vc.emails
        if eaddr.equals("amanda_carwright@example.org") then
            dim fullname as String = vc.fn
            if not fullname is Nothing then
                Console.Writeline("Name: " & fullname)
            end if
        end if
    next eaddr
next vc
```

Figura 2 – Acessando ontologias com base na abordagem de programação estática

Como pode ser visto, os tipos e as propriedades são todas definidas na própria ontologia VCARD. Os valores das propriedades são obtidos diretamente da chamada à referida propriedade, não sendo necessário uma conversão explícita para o tipo apropriado. Outros benefícios citados em [Goldman, 2003] com relação ao uso de representação estática dizem respeito à diminuição de erros, uma vez que o próprio ambiente de desenvolvimento pode checar por erros que no exemplo utilizando o Jena só poderiam ser detectados em tempo de execução. Outra vantagem obtida com o uso da representação estática é que o código gerado pelos compiladores tende a ser mais eficiente.

Contudo, ontologias são desenvolvidas para descrever domínios de conhecimento complexos e que, em sua grande maioria, estão em constante processo de atualização. Nesses casos, o uso da representação estática pode ser custoso, uma vez que toda alteração na ontologia deve, necessariamente, implicar na geração de novas classes que estejam aderentes à nova versão da mesma.

2.2.4. Matchmaking e Ontologias

Como visto na seção sobre *matchmaking*, muitos trabalhos descritos na literatura baseiam seus processos de *matching* no uso de ontologias, tendo como principal interesse o benefício relacionado aos processos de inferência que podem ser realizados sobre elas.

Dessa forma, podem-se definir todas as descrições das entidades do domínio sobre o qual o *matching* será aplicado utilizando-se ontologias, bem como especificar o próprio processo através delas. Outra forma de especificar o *matching* pode ser pela definição de sentenças sobre os metadados das ontologias, tais como aquelas definidas, por exemplo, por meio da linguagem TRIPLE [Sintek, 2001].

Porém, os processos relacionados a *matchmaking* são, na sua grande maioria, computacionalmente complexos e custosos. Assim, esbarra-se num problema ainda não muito bem resolvido, que corresponde ao suporte dado pelas ferramentas de manipulação de ontologias à definição de processamentos de inferência altamente complexos.

Para que as ontologias possam ser utilizadas no contexto de *matchmaking*, faz-se necessário, então, dispor de mecanismos que minimizem o impacto decorrente das deficiências existentes. Sendo, portanto, possível fazer uso de todos os benefícios alcançados com a sua utilização, sem prejudicar a eficiência do processo de *matching*. Tais mecanismos podem, por exemplo, diminuir o número de consultas sucessivas a uma mesma instância de um conceito da ontologia, e de suas propriedades, definindo um mecanismo de *caching* sobre os dados pesquisados em consultas anteriores.

2.3. Frameworks

Frameworks [Fayad, 1999] são geradores de aplicações que estão diretamente relacionados a um domínio específico, isto é, a uma classe de problemas. A partir do uso de um *framework* é possível gerar várias aplicações

para um mesmo domínio, podendo cada uma delas especificar comportamentos e ações diferenciadas para cada uma das características e dos requisitos do domínio.

Dessa forma, como o propósito de construção de um *framework* é a geração de famílias de aplicações para um dado domínio, faz-se necessário que ele possua alto nível de flexibilidade, ou seja, deve ser possível customizar uma instância do *framework* de maneira a atender aos requisitos de um problema particular do domínio em questão.

Os pontos de flexibilização de um *framework* são chamados *hot spots*. Esses pontos de flexibilização devem ser estendidos para que se obtenha uma aplicação funcional, sendo esse processo chamado de instanciação do *framework*. Normalmente, um *hot spot* é definido com base em uma classe abstrata, ou método abstrato, que deve ser, então, implementado.

Um *framework* também apresenta um conjunto de classes que mantém um comportamento ou uma funcionalidade constante para todas as suas instâncias. Essas classes compõem o *kernel* do *framework*, sendo denominadas como *frozen spots*.

O processo de desenvolvimento de um *framework* consiste de três grandes fases:

- *Análise de domínio*: Corresponde a uma análise do domínio sobre o qual o *framework* será proposto. O objetivo principal dessa fase é capturar os requisitos do domínio, com um foco especial na antecipação dos futuros possíveis requisitos. De maneira a auxiliar esse processo, deve-se fazer uso de experiências prévias no desenvolvimento no domínio, experiências pessoais, documentos e padrões para o mesmo. Os pontos de flexibilização e o *kernel* do *framework* são definidos nessa fase;
- *Projeto do framework*: Nessa fase concentram-se os esforços na criação das abstrações do *framework*. Assim, é nela que os *hot spots* e *frozen spots* serão modelados, e a extensibilidade e a flexibilidade propostas na fase de análise são esboçadas. A UML [Booch, 1998] pode, por exemplo, ser utilizada na fase de

modelagem e os *Design Patterns* [Gamma, 1995] são fortemente recomendados como soluções para alcançar a flexibilidade desejada;

- *Instanciação do framework*: Nessa fase os *hot spots* são implementados e o sistema instanciado é obtido. Vale lembrar novamente que cada uma das instâncias compartilhará o mesmo *kernel*, sendo diferenciadas apenas pelo comportamento particular definido e implementado para cada um dos pontos de flexibilização estabelecidos.

As grandes promessas do desenvolvimento baseado em *framework* são a alta produtividade obtida, bem como a diminuição de custos e de tempo no desenvolvimento da aplicação, possibilitado pelo reuso de código e de padrões de projeto [Fayad, 1999]. A grande dificuldade no desenvolvimento baseado em *framework* está, principalmente, na sua concepção. Uma vez que se visa obter uma solução para um domínio inteiro, tem-se que dispor de tempo e de experiência para a concepção dos artefatos relacionados às fases de análise e de projeto do *framework*.