Aceito para publicação no Congresso IDEAS - 2001 "Quarta Jornadas Iberoamericanas de Ingeniería de requisitos y Ambientes de Softwares" – Costa Rica

O Estudo de Frameworks para a Associação de Usuários a Interfaces

Simone Bacellar Leal Ferreira
IBMEC – Faculdades Ibmec-RJ - Mestrado e Graduação em Administração
e-mail: lealferreira@ibmecrj.br

Resumen

Propõe-se o desenvolvimento de uma arquitetura abstrata que viabilize a instanciação de interfaces para diferentes tipos de usuários e diferentes sistemas, melhorando assim o processo de desenvolvimento de software centrados no usuário, isto é, que contribuindo para que os sistemas sejam projetados com o objetivo de satisfazer às necessidades do usuário. Para isso foram definidas diretrizes para a construção de um modelo de usuário, considerado um membro participante do processo de desenvolvimento. Deve-se notar que está sendo proposta uma visão de usuários diferente daquela adotada quando se estudam requisitos não-funcionais, onde usuários ficam "fora do sistema". Além disso, a arquitetura deverá fornecer mecanismos de adaptação, em tempo de execução, de usuários que melhoram seus conhecimentos sobre a aplicação, tornando-se mais conhecedor da interface.

1. Introdução.

Todo *software* é projetado com a finalidade de satisfazer às expectativas de um conjunto de usuários, sendo, portanto fundamental que se conheça como essas pessoas realizam suas tarefas, o que elas pensam sobre seus ambientes de trabalho, e a que tipos de imposições e limites estão sujeitas [1] e [11].

Ao se projetar uma interface, deve-se procurar fazer com que o usuário se sinta confortável e encorajado a usá-la, para isso ele deve poder se comunicar com a máquina da forma mais natural possível. Se humanos conseguem se adaptar a diferentes níveis de funcionalidade em sistemas computacionais, então os programas também devem possuir uma capacidade de se adaptarem a diversos perfis de usuários, isto é, de satisfazer às expectativas diversificadas de diferentes grupos de pessoas [11] e [23].

Para que uma interface satisfaça as necessidades do usuário, os requisitos do software devem ter sido bem definidos. A Engenharia de Requisitos, sendo uma sub-área da Engenharia de Software, procura sistematizar o processo de definição de requisitos de um software, propondo métodos, técnicas e ferramentas que facilitam o trabalho de definição do que se deseja de um software [16]. Nesse trabalho, considerou-se que o processo de desenvolvimento dos sistemas deve ser *centrado no usuário*, isto é, o sistema deve ser projetado com o objetivo de satisfazer as necessidades do usuário [20]. Nesse processo, devem ser obtidas informações a respeito dos usuários, de suas atividades e do domínio da aplicação, isto é, usuário deve ser analisado; durante essa análise, muitas características podem ser determinadas, e um modelo de usuários é construído. Dessa maneira, é possível obter-se as características de um usuário padrão, isto é, daquele que representa a maioria dos usuários.

A fim de auxiliar a análise de usuários, constrói-se um "*modelo de usuários".* Esse modelo deve apresentar características do usuário padrão, e também daqueles que com o tempo se especializam na aplicação.

O fato de cada aplicativo ser usado por diversos grupos de pessoas, que possuem diferentes modelos conceituais a respeito do sistema em questão faz com que seja necessário um *framework* que permita associar usuários a interfaces. Considerando-se o *modelo de usuários*, a probabilidade de cada grupo de usuários finais ficar mais satisfeito será maior, já que o sistema se aproximará o mais possível dos modelos conceituais que os usuários fazem do sistema.

O modelo de um usuário padrão não engloba todos os usuários; cada pessoa cria expectativas diferentes com relação a um sistema. A fim de tornar os *software* com interfaces mais poderosas, capazes de atender a todas expectativas dos vários usuários, isto é, oferecer diferentes níveis de funcionalidade de acordo com a necessidade de cada pessoa, é necessário que se conheça bem os usuários, quais suas carências, suas potencialidades e as suas tarefas. Na maioria das vezes, as características de usuários são bastante diversificadas; a análise de usuário deve então apresentar diversos modelos e enfatizar a necessidade de interfaces com aspectos distintos para cada usuário [14], [8] e [27].

O presente trabalho propõe um *framework* que facilita o processo de desenvolvimento de sistemas centrados no usuário.

A idéia geral é, com base no *framework* proposto, é possível, partindo-se de um sistema devidamente projetado, que o usuário, ao se cadastrar pela primeira vez no sistema, responde a uma série de perguntas. Pelas respostas, o sistema descobre o perfil do usuário em questão e abre para ele a interface correspondente a seu perfil. O usuário tem liberdade de escolher outra, caso seja de sua vontade.

Com o tempo, à medida que uma pessoa vai interagindo com um aplicativo, a sua percepção a respeito da mesma vai se modificando, isto é, ela vai criando um novo modelo conceitual. Isso faz com que suas expectativas, compreensão e objetivos também se alterem. Isso dá um caráter peculiar ao modelo de usuários, pois como ele se baseia nas características, expectativas, compreensão e objetivos do usuário, deve poder também ser alterado no decorrer do tempo.

No caso do usuário ter optado por uma interface diferente da proposta, decorrido um certo período, se o sistema detectar uma dificuldade em usá-la, deve de novo propor outra interface ao usuário.

Com o tempo, à medida que a interação prossegue, o usuário pode ir customizando suas interfaces; estas customizações ficam gravadas, e, sempre que o usuário retorna ao sistema, ele entra na interface por ele personalizada.

Para de fato poder criar novas simulações para os usuários da vida real, o modelo de usuário deve possuir um mecanismo de *extensibilidade de dados e operações*. No paradigma de orientação a objetos, além de ser possível a extensibilidade de dados e operações e a reutilização de código, é possível encapsular-se e proteger-se a definição de uma estrutura de dados e das operações válidas sobre esta estrutura.

2. Instanciação de Interfaces.

Como na realidade existem diversos perfis de usuários, dificilmente um sistema consegue se adaptar a todas às necessidades dos diferentes tipos de usuários. O que ocorre é que ou o sistema apresenta um excesso de funções para determinadas pessoas ou ele é pobre de recursos para outras. Na prática, verifica-se que grande parte dos usuários sente-se frustrada ao usar um sistema.

Para ser bem aceito, um aplicativo deve ser capaz de permitir que os usuários usem todos ou parte de seus recursos, de diferentes formas, isto é, ele deve oferecer ao usuário diferentes visões da interface. Facilidades de *customização* podem ser incluídas de modo a permitir que o usuário defina diferentes conjuntos de comandos individualizados, altere o *default* que lhe é oferecido, personalize suas barras de ferramentas entre outras atividades. Para isso, porém é necessário que o usuário antes aprenda outras tarefas relacionadas à customização, o que torna esse processo difícil para um iniciante [19].

O desejável é a construção de sistemas instanciáveis, isto é, sistemas projetados com aspectos distintos para cada usuário, com diferentes funcionalidades, podendo assim ser usados de forma distinta pelos diversos grupos de pessoas [1].

Para viabilizar essa potencialidade, pode-se lançar mão da programação orientada a objeto. O conceito de *classes*, na orientação a objetos, permite a implementação de *dados abstratos* e a *herança* auxilia o reuso e a adaptação de blocos que não atendem aos requisitos desejados [22], e permite uma melhor adequação a diferentes perfis de usuários [8].

3. *Frameworks* para a associação de usuários a interfaces.

Um sistema correlaciona as diversas tarefas necessárias para gerenciar uma atividade. Deve ser possível configurar e adaptar o sistema aos requisitos de diferentes empreendimentos individuais. Isso é mais viável através da tecnologia de *frameworks.* [3]

Um *framework* é um projeto reusável de todo ou de parte do sistema; sua estrutura é representada por um conjunto de classes abstratas e pela maneira que suas instâncias interagem [15]. Trata-se de uma aplicação "semi-completa", reusável, cujo propósito é poder ser especializada de forma a produzir aplicações customizadas [9]. Em geral, um *framework* é definido com um programa "esqueleto" que define a arquitetura reusável em termos de contratos de colaboração entre as classes abstratas e um conjunto de *hot spots* [7]. Um *hot spot* é um aspecto variável de um domínio de aplicação; são os pontos onde um *framework* pode ser customizado [25]. As customizações são regidas pelas regras contidas nos contratos de colaboração.

O *framework* determina a arquitetura da aplicação; define a estrutura geral do sistema, seu particionamento em classes e objetos com suas respectivas responsabilidades. Determina como as classes e objetos vão colaborar entre si e negociar controle. Dessa forma, o implementador/projetista pode se concentrar apenas nos aspectos específicos da aplicação. [12].

Frameworks de aplicações orientadas a objetos formam uma tecnologia promissora para o refinamento de projetos e implementações de *software*, de modo a reduzir seu custo e melhorar sua qualidade [9].

Originalmente, a tecnologia de orientação a objeto focalizava os *frameworks* como componentes de *software*. Mas *frameworks* são mais customizáveis que componentes e possuem interfaces mais complexas [15]. Os programadores se familiarizam com seu projeto, isto é, com o projeto de suas classes individuais e com as interações entre elas e, provavelmente, com os conceitos básicos de programação orientada a objetos antes de poderem usar um *framework* [22]. Por serem mais poderosos, reduzem muito os esforços necessários para desenvolver aplicações customizáveis [15].

Na realidade, os *frameworks* fornecem as interfaces padrões que permitem que os componentes existentes sejam reusados, isto é, fornecem um contexto reusável para os componentes, provendo um modo padrão para os componentes tratarem os erros, trocarem dados e chamarem operações em cada um [15].

A base de *frameworks* consiste de uma biblioteca de classes. Para construir tal biblioteca, deve-se encontrar abstrações bem adequadas, chamadas de *classes abstratas*, para as classes concretas [26]. Um *framework* em geral inclui sub-classes concretas que podem ser usadas imediatamente [12] e que provêm o comportamento *default* e implementações das classes abstratas [26]. As classes abstratas especificam o fluxo de execução e, através das sub-classes, podem ser especializadas [3].

As aplicações desenvolvidas com base em tais *frameworks* são construídas através de uma *customização* das classes e são muito úteis no desenvolvimento de interfaces com o usuário. Elas reutilizam seu código e seu projeto, facilitando a extensibilidade e o reuso [22].

Como um *framework*, é representado por um código em uma linguagem de representação, o aqui proposto na realidade usaria uma arquitetura (classes, relacionamentos e comportamentos) para as diferentes visões das interfaces com o usuário e aplicações. Uma vez escolhido o domínio das aplicações (editores de texto, por exemplo) então a arquitetura pode ser usada para a criação de um *framework.*

Como os *hot spots* consistem de aspectos variáveis de um domínio de aplicação, os *hot spots* do *framework* em questão, apareceriam por exemplo no lado da modelagem de usuários, na criação de novos atributos para os mesmos. Para certos tipos de aplicações, por exemplo, pode-se querer certos atributos que não são necessários em outros tipos. As interfaces dependem das aplicações, que são o maior ponto de flexibilidade.

4. *Framework* Proposto.

O estudo de usuários adquiriu uma importância grande nos últimos anos, porém pouco progresso foi feito no sentido de aproveitar suas vantagens no processo de instanciação de *software* [28]. Para desenvolver o presente trabalho, foram analisados diversos trabalhos anteriores [24], [17], [19], [2], [5], [13], [21], e todos, deixaram evidente o fato que se o sistema levar em consideração as características de cada grupo de usuário, terá maiores chances de sucesso.

A fim de se poder fazer uma primeira validação do *framework*, desenvolveu-se um protótipo de uma aplicação baseada no *framework* e testou-se tanto esse protótipo como uma aplicação convencional, similar ao protótipo desenvolvido, com cerca de cinqüenta alunos de um curso de pós-graduação da PUC. Posteriormente, esses alunos responderam a uma série de perguntas elaboradas de modo a ser possível comparar ambos aplicativos. A aplicação baseada no *framework* foi considerada mais amigável pela grande maioria dos alunos. esse é um resultado obtido a partir de uma primeira avaliação. Um estudo futuro, fazendo usos de dados estatísticos, poderá ser conduzido de forma a ilustrar melhor tais resultados.

Para a criação do *framework* proposto, não se considerou o conceito de componentes de "*Design Patterns"*, porque a presente proposta tem por objetivo o reuso do *framework* como um todo, não se preocupando em proporcionar um reuso a nível de partes do mesmo. O *framework* desenvolvido possui as seguintes classes fundamentais (figura 1): *Mediadora; Histórico; Perfil; Interface; Aplicação; Cadastro; Login*

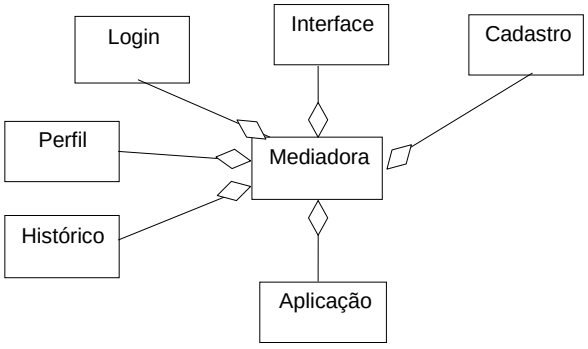

Figura 1: Principais classes do *framework* proposto

A classe *Perfil* contém as diversas características que definem o perfil do usuário. É uma classe abstrata, raiz de hierarquia de tipo, especificando os atributos e funcionalidades comuns a usuários. Classes nesta

hierarquia colaboram com classes na hierarquia de interfaces, de forma a permitir interfaces adequadas a usuários.

A classe *Interface* define as diversas interfaces implícitas que são abertas a cada perfil de usuário. Esta classe depende da classe *Aplicação*, pois a tela *default* é definida de acordo com a aplicação corrente.

Para poder associar um usuário à uma interface *default*, existe a classe *Cadastro*, responsável pela definição inicial do perfil do usuário.

A classe *Aplicação*, que neste trabalho se supõe também raiz de modelagem concreta de domínio de aplicação, é responsável pela funcionalidade do sistema.

A classe *Mediadora* é responsável pela comunicação e sincronismo de seus componentes, descritos acima. Apresenta também uma interface para sua inserção, como componente, em outros contextos.

A classe *Login* é responsável pela primeira definição do perfil do usuário. Ao logar pela primeira vez no sistema, o usuário fornece uma série de informações que permitem definir o seu perfil.

4.1 Classe Perfil.

O modelo de usuário deve levar em consideração as diversas variáveis previamente determinadas. Essa variáveis irão formar a estrutura da classe *perfil*. A figura 2 mostra um exemplo de uma perfil de usuário.

Como não existe um único usuário padrão que represente a maioria, o que ocorre na realidade é a existência de diversos *perfis (modelos) de usuários*. No *framework* proposto, os diversos *perfis de usuários* são representados em uma estrutura de classes; existe uma super-classe chamada *usuário,* sendo cada perfil de usuário representado por uma sub-classe de *usuário*. Os atributos de cada sub-classe serão obtidos a partir dos diversos *modelos de usuário*.

Um usuário pode mudar de estado ou de classe. A medida que um usuário de determinado modelo refina sua percepção, ele irá mudando de estado. Quando a mudança chega a um ponto muito significativo, ele troca de classe, ou seja de perfil.

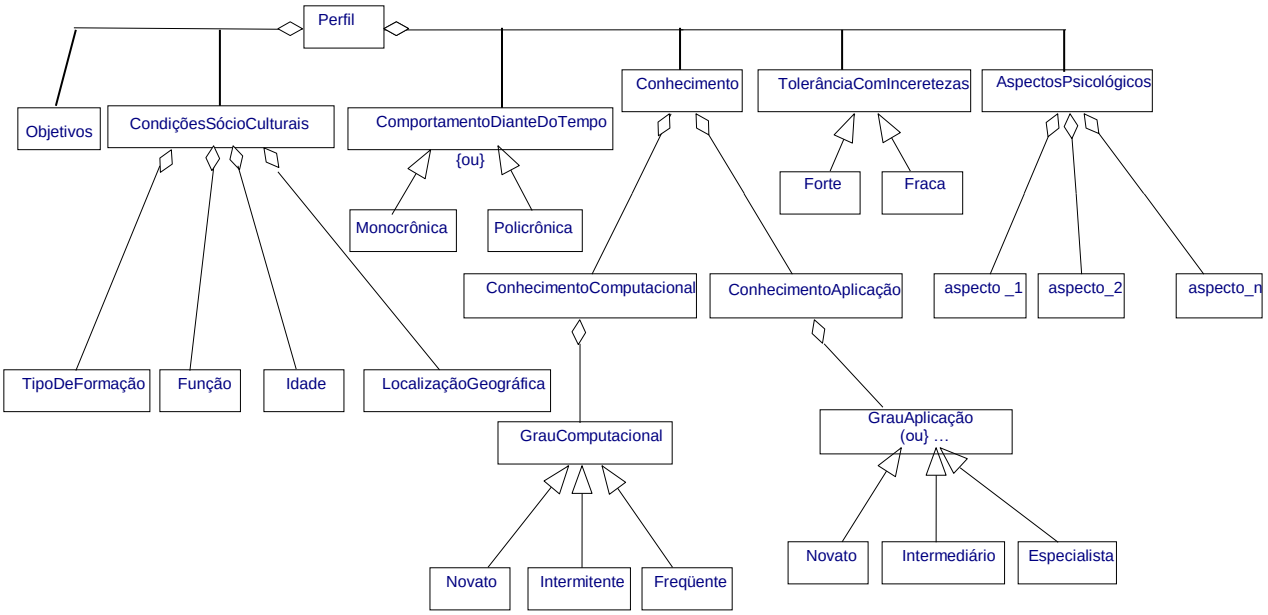

Figura 2: Estrutura da classe *Perfil*

A Classe *Objetivos* modela o objetivo que um usuário tem ao usar um aplicativo. O objetivo de uma pessoa pode variar a medida que sua percepção do sistema se modifica ou até mesmo de uma seção para outra. Em determinado instante, um usuário pode querer usar um *software*, como um editor de texto, apenas para uma simples edição. Já em outro momento, o mesmo usuário pode querer usar este editor para escrever um livro.

A Classe *CondiçõesSócioCulturais* é uma classe que modela aspectos ligados ao tipo de vida do usuário, que são os fatores sócios-culturais. É uma classe composta pelas classes *TipoDeFormação*, *Função*, *Idade* e *LocalizaçãoGeográfica.*

A classe *TipoDeFormação* determina qual a formação de uma pessoa; ela modela não só o grau de instrução, mas também a área de atividade da pessoa (humanas, exatas, biomédica e outras).

A classe *Função* é importante ao se elaborar *software* que serão usados em instituição. Pessoas com diferentes funções provavelmente terão acesso a diferentes funcionalidades de um sistema.

A classe *Idade* pode determinar muitas características de uma interface, principalmente aspectos visuais, como a escolha das combinações de cores *default* [10].

A classe *LocalizaçãoGeográfica* modela aspectos inerentes do usuário, determinados pela localização de sua residência. Metáforas e cores podem ser adequadamente escolhidas se a localização de uma pessoa for conhecida.

A classe *ComportamentoDianteDoTempo* revela como as pessoas lidam com o tempo ao executarem suas tarefas. É uma super classe de duas sub-classes: *Monocrônica* e *Policrônica*. Pessoas monocrônicas preferem fazer uma coisa de cada vez, enquanto pessoa s policrônicas executam perfeitamente uma série de atividades ao mesmo tempo. Portanto, as duas sub-classes são mutuamente exclusivas possuindo assim a restrição de *disjunção.*

A classe *Conhecimento* se refere aos conhecimentos específicos de um usuário com relação aos sistemas computacionais e ao domínio da aplicação. Possui dessa forma, duas sub-classes, *ConhecimentoComputacional* e *ConhecimentoAplicação*. Como um usuário possui os dois tipos de conhecimento, tais sub-classes podem ocorrer simultaneamente, e sua restrição é portanto de *sobreposição.* A sub-classe *Computacional* herda a sub-classe *GrauComputacional*, que caracteriza o grau de conhecimento de um usuário com relação a sistemas computacionais. Da mesma forma, a classe *ConhecimentoAplicação* modela o grau de conhecimento que uma pessoa tem sobre o domínio da aplicação através da classe *GrauAplicação.*

A classe *TolerânciaComIncertezas* trata do comportamento do usuário ao lidar com situações desconhecidas e com erros. Possui duas sub-classes, *Forte* e *Fraca*, que modelam respectivamente pessoas que lidam bem com tais situações e pessoas que se sentem ameaçadas com as mesmas, se tratando assim, de duas sub-classes com a restrição de *disjunção.* Elas foram modeladas como sub-classes, e não como estados da super-classe *TolerânciaComIncerteza,* porque o presente *framework* se propõe a definir um modelo de usuário que torne possível uma instanciação de interface que melhor se adapte às caraterísticas dos usuários. A maneira que uma pessoa reage diante de situações desconhecidas e de erros é inerente a sua própria personalidade [HOFS91]; é possível até que, diante de uma alguma necessidade, alguém consiga reagir diferentemente de sua natureza, mas seria apenas por necessidade. Sua característica permaneceria a mesma. Portanto, o presente *framework* considerou que para melhor adaptar uma interface à natureza do usuário, seria melhor que este atributo fosse modelado via sub-classe, não havendo necessidade de um usuário mudar seu estado, indo contra assim a sua natureza.

A classe *AspectosPsicológicos* modela uma série parâmetros psicológicos podem exercer influência no desempenho de um usuário ao usar um sistema. Essa classe depende de cada aplicação; suas especializações devem ser definidas para cada aplicação. Ela foi apanas citada por ser relevante ao modelo; seus parâmetros pesicológicos devem ser detalhados de acordo com a aplicação e com seus usuários potenciais.

4.2 Mudança de estado do grau de conhecimento

O grau de conhecimento de um usuário, quer seja um conhecimento ou da aplicação, sofre alterações com o decorrer do tempo; um usuário pode portanto, em determinado instante ter um estado de conhecimento, e posteriormente outro. Para isso, os objetos modelados pelas classes *GrauComputacioal* e *GrauAplicação*, terão diferentes estados durante seu ciclo de vida.

4.2.1 Classe GrauComputacional

O *framework* proposto deve levar em consideração que as interfaces poderão ser usadas tanto por usuários com muita experiência computacional como por aqueles que tiveram muito pouco ou nenhum contato com os computadores. A medida que uma pessoa vai interagindo com a máquina, vai adquirindo um maior conhecimento sobre sua potencialidade, e vai mudando sua maneira de interagir com o sistema. A fim de viabilizar essa mudança, usa-se uma mudança de estado, o que permite ao usuário se encontrar em um dos três estados: *novato*, *intermitente* e *freqüente* em um momento mas, em decorrência de alguns estímulos, tipo aumento do aprendizado devido ao maior uso da aplicação, pode mudar de estado.

A figura 3 mostra o diagrama de estado para a classe *GrauComputacional*

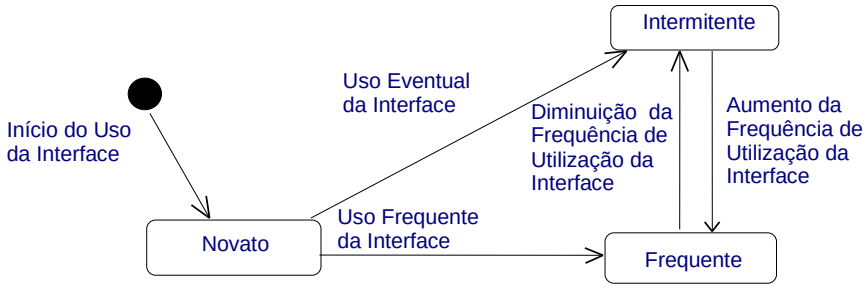

Figura 3: Diagrama de estado da classe *GrauComputacional*

4.2.2 Classe GrauAplicação

Da mesma forma, com o decorrer do tempo, o usuário vai se familiarizando com a aplicação, aumentando assim seu conhecimento. Com o tempo, a pessoa pode vir a adquirir um maior conhecimento sobre a aplicação, mudando sua percepção. A fim de viabilizar essa mudança no modelo conceitual do usuário sobre a aplicação, usa-se uma mudança de estado, o que permite ao usuário se encontrar em um dos três estados: *novato*, *intermediário* e *especialista* em um momento mas, em decorrência de alguns estímulos, tipo maior familiaridade com a aplicação devido ao aumento de seu conhecimento do domínio da aplicação mudar de estado. A figura 4 mostra o diagrama de estado para a classe *GrauAplicação*

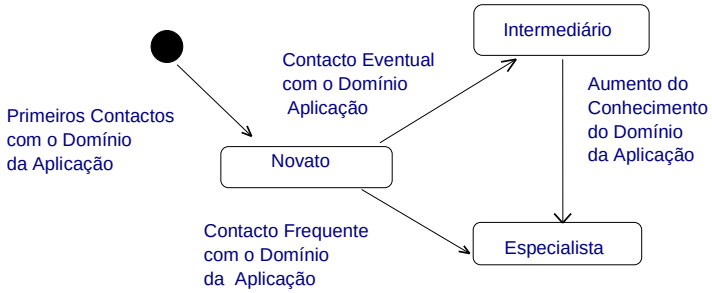

Figura 4: Diagrama de estado da classe *GrauAplicação*

4.3 Classe Interface.

Para cada modelo ou grupo de modelos, existe associado um tipo de interface *default* associada, de acordo com as características do usuário. Um usuário ao se encaixar em um modelo, pode refiná-lo, o que acarreta em uma customização da interface correspondente.

O refinamento de interfaces é feito através da *customização* que usuário vai realizando ao longo de suas interações (como acrescentando novos ícones às barras de ferramentas, incluindo novas barras e outras).

Como sub-classes da super-classe *interface*, existem classes referentes aos diversos tipos de interfaces que podem ser instanciadas. Na realidade essas interfaces são *defaults* propostos, pois o usuário pode sempre personalizar sua interface. A classe *Interface*, modelada através de uma solução específica, semelhante àquela generalizada pelo padrão *"State",* é mostrada na figura 5.

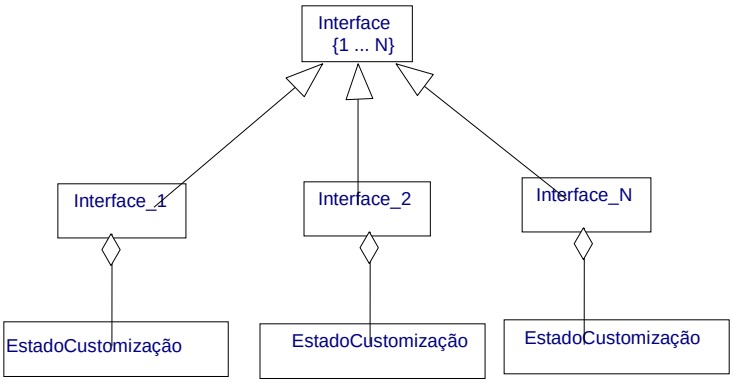

Figura 5: Estrutura da classe *Interface*

4.3.1 Customização: mudança de estado das interfaces

As customizações de uma interface feitas por um usuário consistem em mudanças de estado na classe *interface* associada ao perfil da pessoa em questão.

Ao ser definido o perfil do usuário, lhe é sugerido um estado *default*. A primeira escolha de uma opção por parte do usuário coloca a interface em um novo estado: customização_1 e assim sucessivamente. Se ele retira essa opção, ele retorna ao estado anterior. Trata-se de um grafo fortemente conexo, onde é possível, a partir de qualquer nó do autômato, chegar a qualquer outro (figura 6).

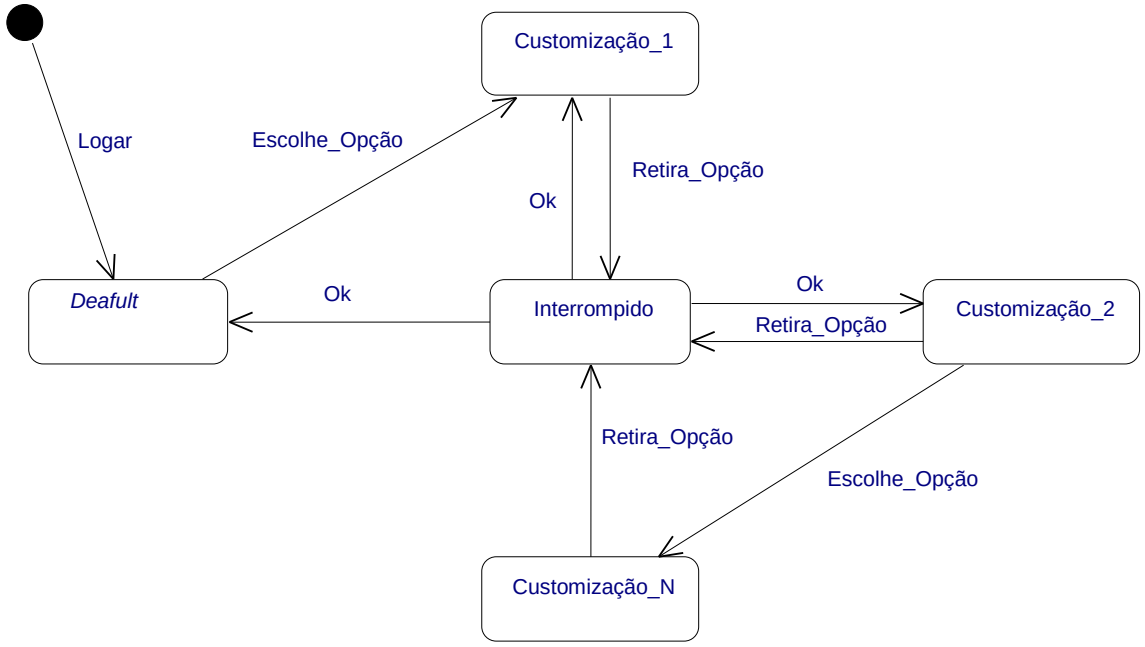

Figura 6: Diagramas de transição para os estados para uma interface

A modelagem de estados para este tipo de autômato é a de modelar estados como sub-classes, com componente responsável por estado [6]. Essa solução combina uma estrutura de classes para modelar estados com a existência de um componente (*interface_i*), modelado pela raiz dessa hierarquia, na definição da estrutura de objetos cujo comportamento varia com o estado.

O componente responsável pela personalização de uma interface está sendo modelado por alguma sub-classe da hierarquia; inicialmente sua classe é *default*, indicando o *default* de interface associada a perfil de usuário sem nenhuma customização. Em função dos estímulos recebidos (que no caso são as personalizações por parte do usuário), este componente muda de classe, mas sempre dentro da família *interface_i*.

A classe *interface_i* permite a declaração do componente estado, não possui componentes próprios, e tem uma implementação vazia para todos os estímulos que a interface pode receber.

```
CLASS interface_i
        PROCEDURE escolhe_opção; BEGIN END
        PROCEDURE retira_opção; BEGIN END
        PROCEDURE ok; BEGIN END
END CLASS
```

As descendentes de *interface_i* redefinem as operações que rotulam transições que se originam no estado correspondente no diagrama. Por exemplo, a classe *default* redefine *escolhe_opção*; a classe *customização_1* redefine *escolhe_opção*, *retira_opção* e *ok* e assim por diante.

4.4 Classe Mediadora.

Como um usuário pode mudar de estado ou de classe, é necessário a existência de uma classe mediadora capaz de monitorar essas alterações. Ela é responsável por detectar e informar ao usuário que este atingiu um nível que lhe permite passar para uma classe de usuários mais experientes, consequentemente, para uma interface com mais recursos. O usuário tem no entanto, a liberdade de não mudar de classe, caso assim deseje.

A classe mediadora mantém um histórico das interações do usuário e de suas customizações. Esse histórico é formado pelas customizações, interações (ações, erros e acertos) do usuário. A classe mediadora monitora o usuário e vai atualizando o seu perfil. Ao adquirir um perfil que justifique uma nova interface, a classe mediadora é responsável por enviar uma mensagem propondo isso ao usuário. O histórico com as informações a respeito das customizações feitas pelo usuário é importante para permitir que, ao alterar de perfil, essas personalizações sejam mantidas na nova interface.

O objetivo de uma pessoa ao usar um sistema deve ser considerado. Uma pessoa pode ter muita experiência em sistemas computacionais, mas desejar usar um aplicativo apenas para tarefas simples, não necessitando portanto muitas das funcionalidades do *software*. Portanto, o objetivo do usuário é importante para evitar que a classe mediadora surgira que ele altere de classe, uma vez que ele não necessita.

Para poder monitorar as interações de cada usuário, a classe mediadora deve ser capaz de receber informações sobre todas ações por ele efetuadas; isto é, ela deve ser informada toda vez que uma funcionalidade for usada, um erro cometido e assim por diante. Essas informações irão constituir o histórico de cada usuário.

Esse histórico deve ficar armazenado em alguma estrutura de armazenamento persistente, como em um banco de dados relacional. o presente *framework* não está preocupada com esse aspecto; sua proposta é de mostrar como é possível, através da elaboração de um modelo de usuários, instanciar interfaces. A maneira pela qual as informações decorrentes das interações dos usuários são armazenadas não é enfatizada, pois não interfere na proposta.

Toda vez que o usuário se conectar (logar) no sistema a classe mediadora deve ser capaz de acessar esse histórico; da mesma forma, de tempos em tempos, a mediadora deve consultá-lo para poder verificar se está na hora de um usuário mudar de interface.

Para representar a classe histórico, pode-se usar uma estrutura do tipo Proxy. Um proxy pode agir com um amplo objeto que pode ser carregado toda vez que for solicitado [12].

4.5 Classe Histórico.

A classe Histórico, cuja estrutura é mostrada na figura 7, mantém um histórico das interações do usuário e de suas customizações. Esse histórico mantém informações sobre todas as customizações, interações (ações, erros e acertos) do usuário. Essa informação sobre as customizações feitas pelo usuário permitem que, ao alterar de perfil, essas personalizações sejam mantidas na nova interface.

Como o histórico só é consultado eventualmente, por ocasião de uma demanda, usa-se a estrutura de um Proxy para representá-lo [12].

Um Proxy pode agir como um amplo objeto que pode ser carregado toda vez que for solicitado. Trata-se de um objeto cujo acesso é controlado, só feito quando demandado, diminuindo assim seu custo de criação e inicialização. Os dados contidos em um Proxy só são criados quando solicitados.

Um Proxy permite o carregamento de objetos persistentes quando referenciado [12]. Os objetos persistentes são aqueles cujas alterações de sua estrutura devem ser preservadas entre as diferentes seções, isto é, esses objetos devem persistir mesmo após o término de uma aplicação [18].

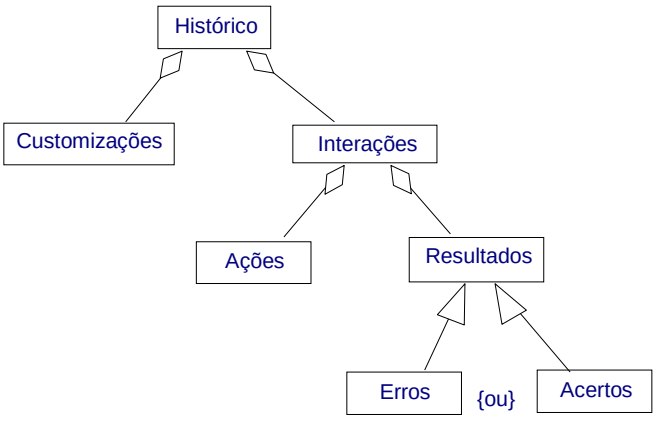

Figura 7: Estrutura da classe Histórico

A classe Customizações contém informações sobre as preferências adicionadas pelo usuário, tipo ícones, escolha de cores, barras de ferramentas entre outras..

A classe Interações modela o diálogo do usuário, isto é, suas ações na interface, os resultados dessas ações. Para isso ela usa as classes Ações e Resultados. A super-classe Resultado pode tanto ser um erro ou um acerto, assim suas duas sub-classes Erros e Acertos são mutuamente exclusivas tendo assim a restrição de disjunção.

5. Conclusões.

O fato de cada aplicativo ser usado por diversos grupos de pessoas, que possuem diferentes modelos conceituais a respeito do sistema em questão faz com que seja necessário um *framework* que permita associar usuários a *software*. A primeira avaliação do protótipo desenvolvido com base nesse trabalho demonstrou que a probabilidade de cada grupo de usuários finais ficar mais satisfeito será maior, já que o sistema se aproximará o mais possível dos modelos conceituais dos usuários.

Como à medida que uma pessoa interage com um aplicativo, sua percepção se modifica, suas expectativas, compreensão e objetivos também se alterem. O modelo de usuários deve poder também ser alterado no decorrer do tempo.

Para de fato poder criar novas simulações para os usuários da vida real, o modelo de usuário deve possuir um mecanismo de *extensibilidade de dados e operações*. Para atender aos requisitos do modelo proposto, uma abordagem orientada a objetos é mais recomendada. No paradigma de orientação a objetos, além de ser possível a extensibilidade de dados e operações e a reutilização de código, é possível encapsular-se e proteger-se a definição de uma estrutura de dados e das operações válidas sobre esta estrutura.

O comportamento dos usuários pode ser modelado por classes, onde novas sub-classes e operações podem ser criadas para expressar novas características. Com isso, o modelo a ser proposto pode viabilizar a criação de novos perfis de usuários, aproveitando classes de usuários já existentes e criando novas sub-classes sem ser necessário repetir operações e dados já definidos em classes já existentes que também se apliquem às novas sub-classes citadas.

Trabalhos anteriores consideraram a importância de modelar os usuários; nenhum, no entanto detalhou essa modelagem. Não foi feita nenhuma proposta que viabilizasse a criação de novos perfis de usuários.

O trabalho proposto pode contribuir para abrir caminhos para muitos trabalhos, como um o desenvolvimento de um *framework* para *workgroups*; a elaboração de um *framework* independente da aplicação.

Por mais independente da aplicação que o *framework* venha a se tornar, sempre existirão alguns *hot spots*. Neste trabalho, esses pontos são bem claros: eles aparecem na modelagem de usuários. Dependendo do tipo de aplicação, provavelmente será necessário a criação de novos atributos para os usuários, atributos estes que não eram necessários para outros tipos. Da mesma forma, alguns atributos antes considerados necessários poderão deixar de serem relevantes.

Uma das mais importantes contribuições do presente trabalho, foi a constatação, através da validação com um grupo de alunos, de que se as interfaces forem projetadas, considerando o modelo de usuários, elas se aproximarão muito mais das reais necessidades do mesmo. E para isso, é fundamental que se comece a modelar os usuários desde o início do projeto de um sistema. O *framework* desenvolvido, ao conter um modelo de usuários com mecanismos de adaptação, em tempo de execução, permitiu que os sistemas se adaptem às diferentes percepções que os usuários vão tendo a medida que interagem a aplicação.

6. Referências Bibliográficas.

[1] - Apple Computer,Inc: "*Macintosh Human Interface Guidelines"* - Addison-Wesly Company - 1992.
[2] - Ambrosini, L., Cirillo, V. & Micarelli, A.: : "*A Hybrid Architecture for User_Adapted Information Filtering on the World Wide Web*" – User Modelling: Proceedings of the Sixth International Conference, UM97, Vienna, New York: Springer Wien New York – 1997.
[3] - Bäumer, D., Cryczan, G., Knoll, R., Lilienthal, C.,Riehle, D. & Züllighoven, H.: "*Frameworks Development For Large Systems*" – Communications of the ACM – Vol. 40. Nº. 10. Octuber. - 1997.
[4] - Beale, R. & Wood, A.: "*Agent-Based Interaction"* – People and Computers IX: Proceedings of HCI'94 – 1995.
[5] - Benaki, E., Karkaletsis, V. A.& Spyropoulos, C. D.: "*User Modeling in WWW: the UMIE Prototype*" - Proceedings of the Workshop: "Adaptive Systems and User Modeling on the Word Wide Web" - Sixth International Conference on User Modeling, Chia Laguna, Sardina, 1997.
[6] - Carvalho, S.: "*Apostila: Orientação a Objetos"* - Deptº. de Informática da Pontifícia Universidade Católica do Rio de Janeiro - 1997
[7] - Codeine W., de Hondt, C., Steyaert P. & Vercammen A.: *"From Custom Applications to Domain Specific Framework"* - Communications of the ACM – Vol. 40. Nº. 10. Octuber. - 1997.
[8] - Collins, D.: "*Designing Object-Oriented User Interface*" - Benjamin/Cummings Publishing Company, Inc. - 1995.
[9]- Fayad, M. & Schmidt, D., C.: "*Object-Oriented Application Frameworks*" – Communications of the ACM – Vol. 40. Nº. 10. Octuber. - 1997.
[10]- Ferreira, S.B.L.; Carvalho, S.E.R.; Leite, J.C.S.P.; Melo, R.N.: "*Requisitos Não Funcionais para Interfaces com o Usuário - O Uso de* Cores" Anais do 2º Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software IDEAS'99 – 1999

[11]- Foley, J. D., Dam, A. V., Feiner, S. K. & Hughes, J. F. *: Computer Graphics - Principles and Practice* - Addison - Wesley Publishing Company - 1990.

[12] - Gamma, E., Helm, R., Johnson, R. &Vlissides, J.: "*Design Patterns – Elements of Reusable Object-Oriented Software*" - Addison-Wesley Publishing Company - 1995

[13]- Gutkauf B., Thies, S. & Domik, G.: "*A User Adaptive Chart Editing System Based on User Modeling and Critiquing*" – User Modelling: Proceedings of the Sixth International Conference, UM97, Vienna, New York: Springer Wien New York – 1997.

[14] - Hix, D. & Hartson R.: "*Developing User Interfaces: Ensuring Usability Trough Product and Process*" - John Wiley & Sons. - 1993

[15] - Johnson, R. E.: "*Frameworks (Component Patterns)*" – Communications of the ACM – Vol. 40. Nº. 10. Octuber. - 1997.

[16] - Leite, J. C. S. P.: *Engenharia de Requisitos* - Notas de Aula da Disciplina "Engenharia de Requisitos" - Deptº. de Informática da Pontifícia Universidade Católica do Rio de Janeiro - 1995

[17] - Marins, C.: *"Interfaces Inteligentes*" – Dissertação de mestrado submetida ao Departamento de Informática da pontifícia universidade Católica do Rio de Janeiro - 1987.

[18] - Meyer, B.: "*Object_Oriented Software Construcion"* - Prentice Hall Series in Computer Scince - 1988.

[19] - Murray, D.: *"Modelling for adaptivity"* – Mental Models and Human-Computer Interaction 2 – M. J. Tauber and D. Ackermann – Elsevier Science Publishers B. V. – 1991.

[20] - Norman, D. A. & Draper, S. W.: "*User Centered Design*" – Hillsdale, NJ: Lawrence Erlbaum - 1986

[21] - Paranagama, P., Burstein, F. & Arnott, D.: "*Modelling the Personality of Decision Makers for Active Decision Support*" – User Modelling: Proceedings of the Sixth International Conference, UM97, Vienna, New York: Springer Wien New York – 1997.

[22] - Pree, W.: "*Design Patterns for Object-Oriented Software Development*" - Addison-Wesley Publishing Company - 1995.

[23]- Pressman, R. S.: *Software Engineering - A Practioner's Approach* - 3rd ed., McGraw-Hill, Inc. - 1992.

[24] - Rich. E.: *"User Modeling Via Sterotypes"* – Cognitive Science vol. 3 – 1979.

[25] - Scmid, H. A.: "*Systematic Framework Design by Generalization*" - Communications of the ACM – Vol. 40. Nº. 10. Octuber. - 1997.

[26] - Shlaer, S. & Mellor, S. J.: "*Object Lifecycles - Modeling the World in States*" Yourdon Press- 1992.

[27] - Shneiderman, B.: "*Designing the User Interface – Strategies for Effective Human-Computer Interaction*" – Addison_wesley – 198.

[28] - Strachan, L., Anderson, J., Sneesby, M. & Evans, M.: "*Pragmatica User Modelling in a Commercial Software System*" – User Modelling: Proceedings of the Sixth International Conference, UM97, Vienna, New York: Springer Wien New York – 1997.