2
Trabalhos Relacionados

Neste capítulo apresentamos os trabalhos que se relacionam com esta pesquisa organizando-os em três seções. Na primeira apresentamos uma visão geral sobre a área de End User Development - EUD e as principais pesquisas desta área que estão relacionados diretamente ou indiretamente com esta tese. Na segunda seção apresentamos referenciais teóricos utilizados nesta pesquisa, obtidos em diferentes fontes teóricas e fora do contexto de Engenharia Semiótica. Na terceira, apresentamos a Engenharia Semiótica (teoria de IHC que fundamenta esta pesquisa) e seus principais trabalhos sobre aplicações de *groupware* e aplicações extensíveis.

2.1
Desenvolvimento de Sistemas por Usuários Finais

Tendo em vista a penetração e a incorporação das Tecnologias de Informação na vida cotidiana, cresce rapidamente a demanda por novos produtos ou pela possibilidade de adaptar e estender produtos existentes. Diante desta realidade, existe uma tendência de envolver e capacitar pessoas que não são profissionais de informática (usuários finais) no processo de desenvolvimento de software (End User Development - EUD) (Lieberman et al., 2006). Além disso, desde há muito tempo, diversos trabalhos, como por exemplo, os de (Suchman, 1987) e (Myers, 1992) explicam por que é pouco provável um software fornecer soluções para todos os problemas, de qualquer tipo de usuário, em um domínio de atividade específico. Esta é historicamente uma das principais razões para capacitar pessoas que não são especialistas em programação de sistemas a desenvolverem sozinhas suas próprias ferramentas e peças computacionais de acordo com suas necessidades.

Mais recentemente, a Web 2.0 tem promovido amplamente uma cultura de participação (Fischer, 2009), na qual usuários deixam de ser meros consumidores de tecnologia e passam a ser produtores destas. Por várias décadas o computador proporcionou poderes às pessoas individualmente, mas atualmente a Web 2.0 está oferecendo grandes oportunidades para grupos e comunidades (Fischer, 2010). Através das redes sociais, usuários

compartilham conhecimento e habilidades. Sistemas como Wikipedia, Google Maps, Facebook e Orkut motivam a participação e colaboração entre pessoas com interesses em comum. Com isso, a comunidade de EUD tem discutido sobre "Criatividade Social" (Fischer, 2007). Este conceito fundamenta-se na ideia de que o poder de uma mente trabalhando sozinha é sempre mais limitado do que o de várias mentes trabalhando juntas. Embora indivíduos criativos às vezes trabalhem isolados, a criatividade, em muitos casos, vem de atividades que aconteceram no contexto social, através da interação com outras pessoas e com artefatos que incorporam conhecimento coletivo. Neste contexto, diversas questões precisam ser pensadas, como a motivação das pessoas para contribuir, os requisitos para uma contribuição (sobretudo de qualidade), os papéis dos curadores de acervos de contribuição, mecanismos para seleção de conteúdo, etc.

Uma das metas de EUD é então promover esta nova Cultura de Participação e Criatividade Social. Várias perspectivas estão sendo investigadas: metodologias de desenvolvimento de sistemas envolvendo usuários finais, EUD/EUP[1] em atividades cooperativas, estudos cognitivos sobre a atividade de programação por usuários finais e ferramentas que permitem usuários finais modificarem sistemas existentes. A seguir, faremos uma breve apresentação de cada um destes tópicos.

2.1.1 Metodologias de Desenvolvimento de Sistemas envolvendo Usuários Finais

Fischer estuda metodologias de desenvolvimento de software envolvendo usuários finais. Ele propôs o "Meta-Design" (Fischer & Giaccardi, 2006), um *framework* conceitual que permite aos usuários tornarem-se co-designers. O *framework* fundamenta-se na idéia de que os futuros contextos e propósitos de uso e problemas que os usuários tentarão resolver com o sistema não podem ser completamente antecipados durante o desenvolvimento de um sistema. O autor propôs ((Fischer, 2007)) uma redistribuição das atividades de projeto de sistemas entre os participantes. Apoiando o "Meta-Design", Fischer propôs o SER (seeding, evolutionary growth, and reseeding), um modelo descritivo e prescritivo cuja ideia central é, ao invés de tentar construir sistemas completos na etapa de projeto, construir sementes (seeds) que podem evoluir ao longo do tempo através de pequenas contribuições de muitas pessoas envolvidas (designers e usuários finais). A intenção é que os "co-designers"

[1]EUP é o acrônimo de "End User Programming", que significa "Programação por Usuário Final".

criem ambientes sócio-técnicos onde outras pessoas possam constantemente contribuir com sua criatividade. Os "metadesigners" deixam intencionalmente de determinar funcionalidades e conteúdos do sistema para motivar e apoiar usuários finais a agirem como "designers". Desta forma, estão engajando os usuários no desenvolvimento de sistemas e consequentemente promovendo a "cultura de participação", citada anteriormente.

Costabile tem estudado o envolvimento de especialistas em um domínio de aplicação (por exemplo, medicina, geologia, engenharia, etc.) no processo de desenvolvimento de sistemas. Eles são especialistas em uma área mas não são necessariamente familiarizados com computação. Costabile e co-autores (Costabile et al., 2003) propuseram uma metodologia de projeto para construir ambientes de software que permitam atividades de EUD para usuários especialistas em um domínio específico, denominada "Software Shaping Workshop (SSW) design methodology". Em (Costabile et al., 2007) Costabile estende sua metodologia e propôs um modelo de interação entre usuários e sistemas, o qual leva em conta a evolução do uso do sistema. Posteriormente, Costabile mostra em (Costabile et al., 2009) que a metodologia SSW é capaz de engajar verdadeiramente os usuários e que eles ainda podem fornecer valiosas contribuições aos projetistas.

Apesar de estes trabalhos estarem focados no desenvolvimento de novos sistemas e nossa pesquisa estar inserida no contexto de alterações de sistemas existentes, eles abordam a problemática de melhorar a comunicação entre usuários finais e *designers*, a qual estamos interessados em investigar. Além disso, estas pesquisas ressaltam a importância de envolver usuários finais no desenvolvimento de sistemas, isto é uma das motivações e fundamentações deste trabalho.

2.1.2 Linguagens para Usuários Finais

Existe na literatura de End User Development e End User Programming algumas linguagens para usuários finais programarem suas próprias funcionalidades. Em (da Silva, 2001) o autor classifica tais linguagens em três paradigmas: programação paramétrica, imitativa e descritiva.

Linguagens para Programação Paramétrica

A programação paramétrica permite que o usuário final modifique um sistema a partir de atualizações de valores de parâmetros disponibilizados no referido sistema. Ele pode fazer isso basicamente de duas formas. Na primeira ele altera valores dos parâmetros selecionando uma entre um conjunto pré-

definido de alternativas disponibilizadas pelo designer. Este é o típico caso de configuração de aplicação. Na segunda, o usuário pode atribuir um nome a uma combinação de diferentes parâmetros (com seus respectivos valores). Esse nomes são adicionados na interface da aplicação como novo item léxico, e ao serem indicados, ativam a combinação de parâmetros e valores associada a ele. O "nome" funciona como um operador de agregação de uma estrutura fixa de parâmetros, sendo então na realidade uma espécie de elaboração da primeira alternativa, onde apenas se pode selecionar elementos (mas não agregá-los). Um exemplo de programação paramétrica é a criação de estilos no editor de texto Microsoft Word. Outro exemplo, podemos citar as ferramentas de *Mashups*[2] YahooPipes (Sabbouh et al., 2007, Floyd et al., 2007) e Ubiquity (Mozilla, 2010b), pois basicamente, com elas os usuários criam uma nova funcionalidade a partir do preenchimento de parâmetros, que são "dados" vindos de outros sistemas. Na Figura 2.1 pode ser visto um exemplo onde o usuário seleciona uma lista de endereços disponíveis em um *website* e depois utiliza a ferramenta Ubiquity para mapear os endereços numa ferramenta de mapas, ou seja, os dados do *site* de endereços foram usados para preencher parâmetros do mapa.

Linguagens para Programação Imitativa

A programação Imitativa abrange dois mecanismos: a gravação de macros e a programação por demonstração (ou por exemplos) (Cypher & Halbert, 1993), (Lieberman, 2001). A **gravação de macros** consiste em criar réplicas de sequencia de passos de interação. O usuário aciona um gravador e depois interage normalmente com a aplicação, mostrando a sequencia de interação a ser gravada. Ao final o usuário associa uma identificação à sequencia gravada, a qual pode ser acionada para reproduzir automaticamente o que foi gravado. A **programação por demonstração** é um mecanismo mais avançado que a gravação de macros. Ela utiliza técnicas de inteligência artificial que criam programas mais genéricos a partir de exemplos de passos de interação mostrados pelo usuário (Cypher & Halbert, 1993), (Myers, 1992), (Lieberman, 2001).

A ferramenta CoScripter (Little et al., 2007) é um gravador de macros para Web que permite a criação de funcionalidades que são réplicas de sequencias de passos de interação. A Figura 2.2 mostra um exemplo de um *script* para consultar peças em cartaz na cidade do Rio de Janeiro com valores

[2] "A Mashup is a website or Web 2.0 application that uses content from more than one source to create a completely new service." (Mashup, 2010) Tradução: Um Mashup é um website ou uma aplicação web que usa conteúdo de mais de uma fonte para criar um novo serviço completo.

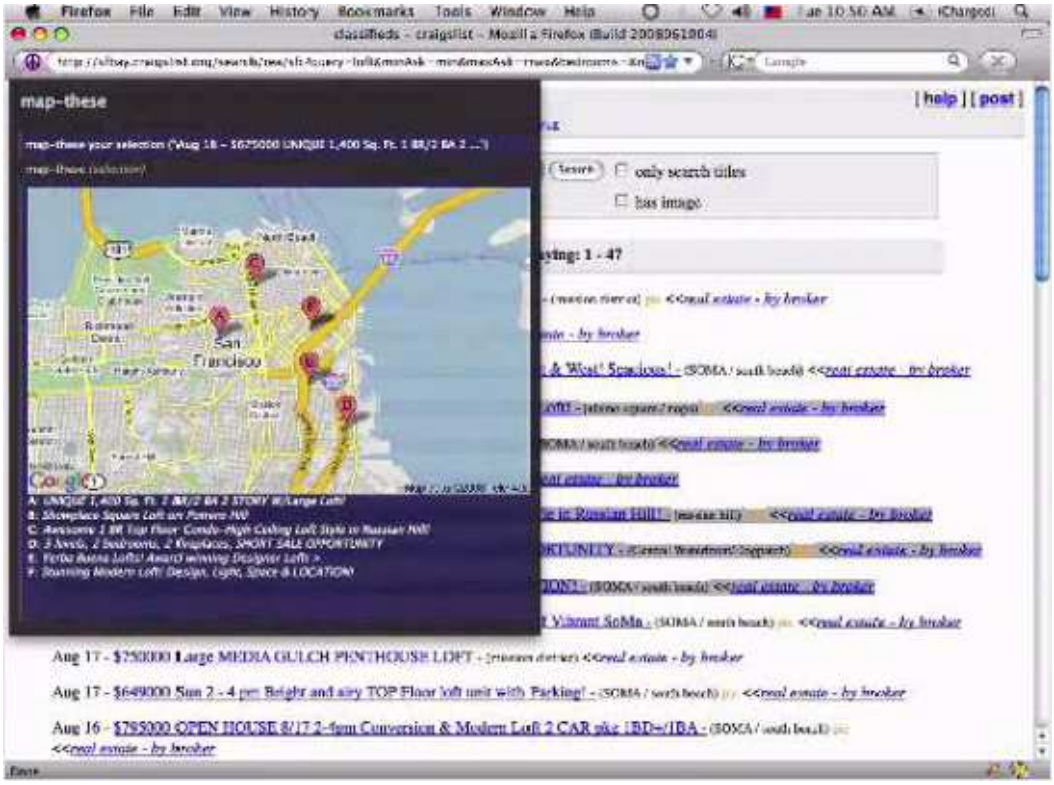

Figura 2.1: Ferramenta Ubiquity. Exemplo de Programação Paramétrica.

entre 20 e 30 reais, nela podemos ver a sequência de passos de interação e o nome (identificador) do *script*. Este é um exemplo do conceito de gravação de macros, mas não podemos dizer que é um exemplo de programação por demonstração, pois não é feito nenhum tipo de inferência sobre os passos de interação.

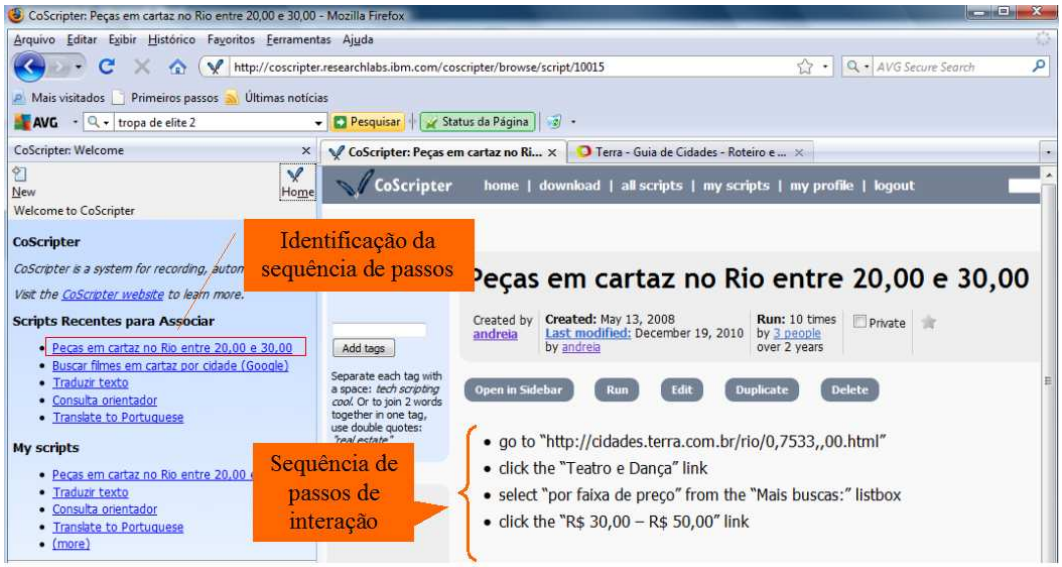

Figura 2.2: Ferramenta CoScripter. Exemplo de Gravação de Macros.

Uma grande vantagem da gravação de macros é a familiaridade do

usuário com a linguagem usada para especificar a nova funcionalidade, que é a linguagem de interface. Por outro lado, as gravações normalmente armazenam os passos interativos de forma literal e dependente do contexto, ou seja, não é possível criar variáveis, condições e iterações. Já a programação por demonstração mantém a vantagem da familiaridade do usuário com a linguagem e adiciona um aumento no poder de expressão através da possibilidade de inferência dos construtos mais abstratos de programação.

É importante ressaltar que normalmente a macro é gravada (registrada) em uma linguagem textual, desconhecida do usuário. Portanto, se o usuário precisar editar uma macro ele precisará conhecer esta linguagem. Em geral esse aprendizado é dificultado pelo fato de que não há um co-relação alta entre a linguagem de interface (conhecida por ele) e a linguagem textual. Portanto, fica difícil reconhecer quais elementos da linguagem de interface se referem a quais elementos da linguagem textual.

Linguagens para Programação Descritiva

A programação descritiva lida com a dificuldade de mapear a linguagem de interface com a linguagem da macro. Para isso, propôs-se o mecanismo de aprendizagem por revelação (Eisenberg & Fischer, 1994), (Eisenberg, 1995). A ideia é mostrar ao usuário, a cada passo de interação qual seria o referente (por exemplo o comando) correspondente na linguagem textual da macro. Essa é uma forma mais suave do usuário aprender a linguagem da macro. Podemos citar como exemplo a ferramenta CoScripter, pois nela a cada passo de interação o usuário pode ver a instrução correspondente gerada pelo gravador de macro da ferramenta. Além disso, no momento da execução do *script* este mapeamento também pode ser visto através de marcações na interface.

As linguagens de programação visual são uma alternativa às linguagens textuais. Eles tentam reduzir a complexidade da tarefa de programação por meio da disponibilização de representações visuais, e não linguísticas, dos construtos de uma linguagem de programação. Por exemplo, elas disponibilizam ícones para representar dados e operações. Nardi (Nardi, 1993) defende a utilização de uma linguagem de programação visual mista, na qual usuários utilizam formalismos visuais ou *frameworks* visuais de aplicação que se articulam com linguagens de programação textuais. Em geral esta forma de programação exige que o usuário final aprenda uma nova linguagem que é mais complexa do que as linguagens de outros mecanismos, por outro lado, ela é mais expressiva.

Paternó estudou o uso de linguagens gráficas para usuários finais (Paternò et al., 1994). Ele investigou a criação de linguagens para especificar

requisitos de sistemas de maneira que sejam mais apropriadas do que as linguagens usadas na prática profissional de informática. Segundo o autor essa é uma alternativa promissora para melhorar a comunicação entre profissionais de informática e usuários finais (Patemò, 2001).

Normalmente a programação paramétrica apresenta um custo de aprendizado baixo, porém seu poder de expressão é restrito aos parâmetros e seus respectivos valores disponibilizados pelo *designer*. É importante observar que neste paradigma o usuário atua apenas no léxico da linguagem de interface original. Já no paradigma de programação imitativa o usuário pode atuar, além do léxico, no nível sintático da linguagem de interface, isto porque o usuário consegue criar sequencias de passos de interação, ou seja, novas sentenças dentro da linguagem de interface. Este fato proporciona aumento no poder de expressão do usuário, uma vez que torna-se possível, a partir de um conjunto fixo de elementos léxicos da interface, obter infinitas sequencias de interação ou novas sentenças da linguagem de interface. Tipicamente a programação paramétrica, imitativa e descritiva representam, respectivamente, uma ordem crescente de expressividade da linguagem, assim como de aprendizado pelo usuário final.

2.1.3 Ferramentas de EUD/EUP

Com a evolução dos sistemas Web e a cultura de participação dos usuários finais (citada anteriormente), têm surgido propostas impactantes na direção da implementação do conceito de desenvolvimento de sistemas feito por usuário final. Em geral, estas ferramentas permitem a construção de *scripts* para automatizar processos na Web, alterar a aparência das páginas, ou até mesmo criar novas funções. Tais ferramentas podem ser categorizadas em dois grupos:

- Ferramentas de configuração já oferecidas pelos desenvolvedores do sistema (ferramental interno). Nesta categoria se encaixam os sistemas customizáveis ou extensíveis.
- Ferramentas de customização oferecidas por terceiros (ferramental externo). Neste caso, enquadram-se sistemas que servem para customizar ou estender outros sistemas, tais como: (i) Linguagens de Scripting para programadores profissionais, como por exemplo, JavaScript. (ii) Linguagens e ferramentas de Scripting para usuários finais, como por exemplo, GreaseMonkey, ChickenFoot (Bolin et al., 2005) e CoScripter. (iii) Ferramentas de "Mashups" (Sabbouh et al., 2007,

Floyd et al., 2007) como, por exemplo, YahooPipes e Ubiquity (Mozilla, 2010b).

Destacamos aqui duas destas ferramentas, produzidas por terceiros, que permitem customizar e estender sistemas Web. O Chickenfoot (Bolin et al., 2005) é um destes sistemas para usuários finais automatizarem, customizarem e integrarem aplicações web sem alterar o código fonte da aplicação. Basicamente ele é útil para:

- Automatizar operações repetitivas na web;
- Integrar vários "web sites";
- Mudar a aparência das páginas, como retirar elementos que não são interessantes ou acrescentar elementos úteis.

A Figura 2.3 é um exemplo de uso do Chikenfoot para alterar a aparência de um das páginas do *site* de uma biblioteca digital - ICDL. No lado esquerdo, pode ser vista a barra de ferramentas do Chickenfoot com um *script* para substituir imagens da página "Busca Simples" desta biblioteca (ilustrada nesta figura). Esta alteração é feita através comando "replace". A Figura 2.4 mostra a mesma página da ICDL mostrada na 2.3 só que alterada, após de ter sido executado o *script* do Chickenfoot.

Figura 2.3: Página original da ICDL

A principal desvantagem desta aplicação é que ela apresenta um alto esforço de aprendizado para pessoas que não têm formação em programação de sistemas, visto que eles precisam conhecer as tecnologias *HTML* e *JavaScript*.

Figura 2.4: Página alterada da ICDL

Outra ferramenta para customizações e extensões de sistemas Web é o CoScripter (Little et al., 2007) (Leshed et al., 2008) ou Koala, seu nome antigo. Ele é um sistema desenvolvido pela IBM que permite usuários finais criarem e compartilharem *scripts* para executar interações na Web. Ele tem dois componentes: (i) um *plug in* para o navegador Mozilla Firefox (Mozilla, 2010), no qual o usuário pode gravar ações em sistemas Web e executá-las posteriormente; e (ii) uma biblioteca (Wiki) que funciona como um repositório central onde usuários podem compartilhar, editar e buscar *scripts* de outros usuários. A Figura 2.5 mostra um *script* para rodar no site www.terra.com.br com finalidade de sugerir bons restaurantes na cidade do Rio de Janeiro. Este é um exemplo de *script* que registra o conhecimento sobre bons restaurantes no Rio. Além disso, ele automatiza a atividade de buscar restaurantes neste site.

O CoScripter e o Chickenfoot têm algumas características em comum, a principal delas é a automatização de processos na Web. Porém, existem vantagens e desvantagens em ambos. Por um lado, o poder de expressão da linguagem do Chikenfoot é maior que a do CoScripter. Por exemplo, o Chickenfoot permite alterar a aparência das páginas, já o CoScripter não permite. Por outro lado, a legibilidade da linguagem do CoScripter é bem mais interessante, visto que os usuários não precisam conhecer HTML e JavaScript como acontece com os usuários do Chickenfoot. Outra vantagem do CoScripter é a ideia de compartilhamento de conhecimento visto que ele promove a colaboração entre seus usuários através de uma biblioteca de *scripts*.

Figura 2.5: Exemplo de Uso do CoScripter

A ferramenta CoScripter, explicitamente, combina características de linguagens de programação Imitativa e Descritiva. A programação imitativa é observada na gravação de macros, permitindo criar cópias de sequencias de passos de interação. Além disso, o gravador de macros do CoScripter possui um recurso típico da programação por demonstração que é a possibilidade de criar construtos abstratos, como por exemplo variáveis e condições. Adicionalmente, esta ferramenta também tem características da programação descritiva porque contempla a ideia de mostrar ao usuário, a cada passo de interação qual é o comando correspondente na linguagem do *script*. Com isso, ela facilita o aprendizado de tal linguagem. Esta ferramenta faz isto de duas maneiras: i) quando a interação está sendo gravada, a ferramenta mostra em sua barra de ferramentas, o comando corresponde à interação, e (ii) quando o *script* está sendo reproduzido, pois ficando destacado em cor verde o elemento da interface correspondente à instrução do *script* que está sendo executado.

O Chickenfoot possui características típicas do paradigma de programação descritiva. Assim como o CoScripter ela mostra ao usuário, destacando na tela, o elemento da interface que corresponde à instrução na linguagem do *script*. Porém, ela é considerada uma linguagem com alto custo de aprendizagem, o que é caraterística deste tipo de paradigma. Na Figura 2.6 pode ser visto um exemplo de código do Chikenfoot.

Estas ferramentas são uma evolução do que diz respeito à customização e extensão de sistemas Web. Nosso intuito foi analisar como suas contribuições

```
originalTab = tab;
isbn = find(/ISBN: (\d+)/).groups[1];
with(openTab('http://libraries.mit.edu/', true)) {
  pick('Keyword');
  enter(isbn)
  click('Search button')
  link = find('stacks link')
}
originalTab.show();
insert(after(/^Name:.*/), '<li>' + link.html)
```

Figura 2.6: Exemplo de código do Chickenfoot para encontrar um livro em uma biblioteca virtual. Extraído de http://groups.csail.mit.edu/uid/chickenfoot/examples.html.

poderiam ser úteis para nossa investigação sobre representações feitas por usuários finais. Avaliamos a possibilidade de utilizar estas linguagens para servir como base em nosso modelo de discussão.

Na avaliação do Chikenfoot vimos que o nível de abstração da linguagem é muito baixo para alguém que não tem conhecimento sobre linguagens de programação. Quanto ao CoScripter, o resultado do estudo foi publicado em (Sampaio & de Souza, 2008). Ele será apresentado a seguir, no capítulo 4 onde descrevemos os estudos empíricos desta pesquisa.

2.2 Referenciais Teóricos Adicionais

Nesta seção falaremos de trabalhos que são usados como referenciais teóricos nesta pesquisa mas que não são frutos de pesquisa de EUD e nem da Engenharia Semiótica.

2.2.1 Aspectos Representacionais de Linguagens

Um dos pontos importantes nesta pesquisa é encontrar o limite adequado entre o poder de expressão e o esforço cognitivo de aprendizado de uma linguagem a ser utilizada por usuários finais. O nível de abstração da linguagem que estamos pesquisando é um item relevante para o sucesso da linguagem associada ao modelo que estamos propondo. Destacamos aqui um trabalho que nos ajuda a refletir sobre o nível de abstração desta linguagem.

Stenning e Oberlander (Stenning & Oberlander, 1995) estudaram aspectos representacionais das linguagens em geral, não somente linguagens

de programação. Eles compararam os tipos de raciocínio apoiados por representações textuais e gráficas. Os autores classificaram os sistemas de representação em relação aos níveis de abstrações que eles permitem fazer: Minimal Abstraction Representation Systems (MARS), Limited Abstraction Representation Systems (LARS) e Unlimited Abstraction Representation Systems (UARS). A Figura 2.7 ilustra esses conceitos. No MARS, um elemento do sistema representacional corresponde a um elemento do modelo semântico. No LARS, um elemento do sistema de representação pode corresponder a um ou mais elementos do modelo semântico. E no UARS, pode haver correspondências entre os elementos dos dois sistemas, sem nenhuma restrição.

Considere um sistema gráfico onde uma representação é um arranjo fixo de quadrados contendo um conjunto de círculos pretos. A interpretação que se deseja deste sistema nos diz três coisas: (i) os quadrados denotam o conjunto de escritórios de um prédio, (ii) os círculos pretos denotam pesquisadores, e (iii) a relação espacial de estar contido denota a relação de trabalhar em um escritório. Neste caso, como a representação gráfica corresponde a um único modelo possível para a interpretação desejada, temos um MARS.

Suponha agora que desejamos representar o fato de que todas as salas são ocupadas por dois pesquisadores, exceto uma delas (sala S), que pode conter dois ou três pesquisadores. Neste caso podemos fornecer dois diagramas, onde no primeiro a sala S aparece com dois ocupantes, e no segundo a sala S aparece com 3 ocupantes. Uma outra forma de representar este cenário é introduzindo um novo símbolo indicando zero ou um pesquisador, por exemplo, um círculo branco. Fornecemos então um único diagrama onde a sala S contém dois círculos pretos e um círculo branco. Nestas duas representações temos mais de um modelo possível para uma mesma representação gráfica, o que constitui um LARS.

Considere agora que os pesquisadores são identificados de 1 até n, e as salas são identificadas de 1 até m. Imagine que desejamos indicar que o círculo branco na sala 5 representa um pesquisador apenas se os pesquisadores 2 ou 3 foram alocados na sala 4 (0 pesquisadores caso contrário). Neste caso, temos uma dependência que se refere a partes específicas da representação, resultando assim em um UARS.

Os autores propõem que a representação ideal está no nível LARS, visto que estes sistemas têm maior capacidade de abstração que os MARS, porém, as operações mentais associadas à interpretação das representações não são tão complexas quanto as exigidas por UARS. Os sistemas LARS permitem representar um objeto abstraindo detalhes e, por isto, a representação pode corresponder (ou ser mapeada) a mais de um objeto concreto distinto no

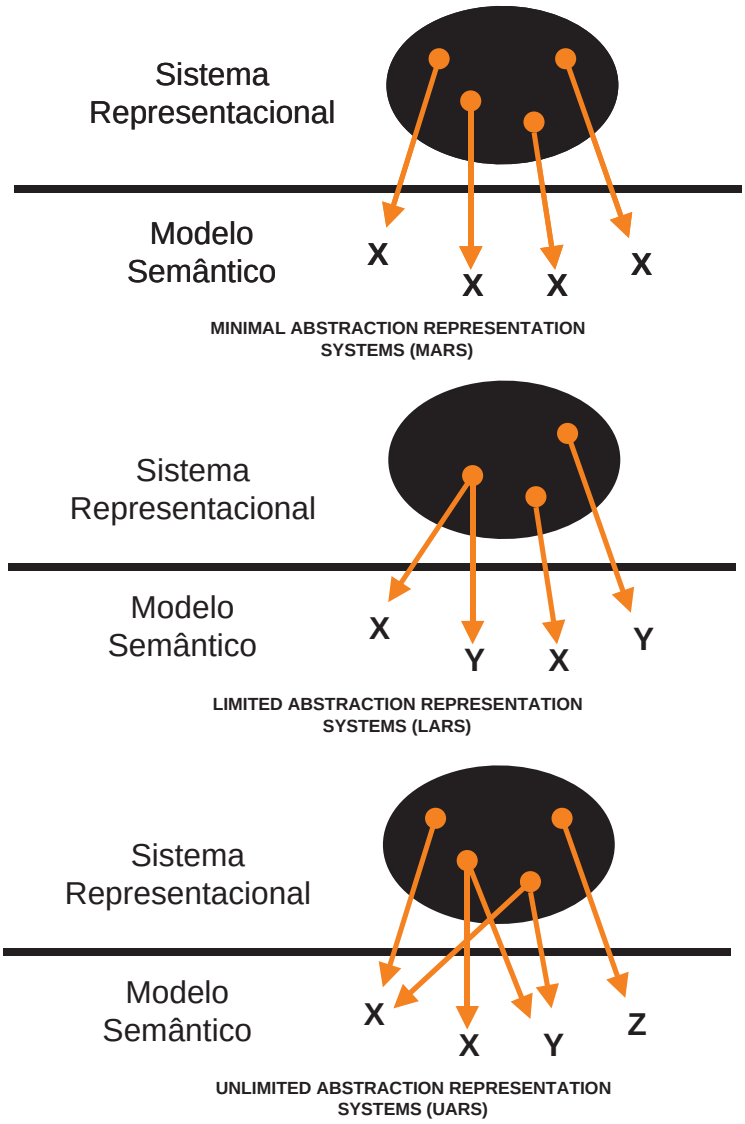

Figura 2.7: Níveis de Representação de Linguagens

plano das entidades representadas. A imprecisão ajuda a olhar com foco no que é mais relevante, sem ter que se prender a detalhes. Isto é uma característica importante para as linguagens de design, atividade que precede a especificação. Os autores concluíram que as representações gráficas apoiam abstrações limitadas, ou seja, há conceitos que elas não permitem expressar. Por outro lado, o fato de elas serem limitadas facilita e acelera as etapas do raciocínio inferencial, ou seja, ajuda o usuário a raciocinar mais rápido.

Consideramos os argumentos de Stenning e Oberlander ao refletir sobre o nível de abstração adequado para usuários finais expressarem suas ideias de modificação de sistemas. A linguagem proposta nesta pesquisa para representar propostas de modificações de sistemas (LM) permitirá uma representação no nível LARS. Isto porque o usuário final não precisa programar sua proposta (o que requer o nível de especificidade que somente um MARS poderia oferecer),

ele precisa apenas construir um esboço da sua ideia, apontando, modificando e criando novos elementos na interface que representem adequadamente a sua intenção.

2.2.2 Tipologia de Signos proposta por Peirce

O conceito central em Semiótica é o de signo. Para Peirce, um signo é algo que significa alguma coisa para alguém (Peirce et al., 1998). Ele é o produto de uma relação entre uma representação (o que representa algo para alguém, por exemplo, a imagem de um relógio), um objeto (por exemplo, o objeto real relógio) e seu interpretante, ou interpretação (um pensamento, sensação, ação ou um outro signo, por exemplo, o pensamento "que horas são?" ou a percepção de que "o relógio é um instrumento que marca a passagem do tempo em horas, minutos e segundos").

Uma das muitas classificações de signos propostas por Peirce (Peirce et al., 1998) é a que os define como ícones, índices ou símbolos. Esta classificação é uma decorrência do estabelecimento de categorias gerais de fenômenos que se podem conhecer. São três categorias:

(i) Primeiridade: Categoria centrada na experiência vivida ou percebida de forma integrada, não relacional, não inferencial. Trata-se da categoria de fenômenos essencialmente sensoriais e experienciais. (ii) Segundidade: Categoria centrada na associação entre dois fatos, duas percepções, dois objetos, dois fenômenos. Trata-se da categoria dos fenômenos relacionais não inferenciais. (iii) Terceiridade: Categoria centrada na mediação dos princípios convencionais, lógicos, inferenciais, explicativos. Trata-se essencialmente da categoria dos fenômenos de interpretação e atribuição de significado.

De acordo com estas categorias, **ícone** é o signo que evoca a primeiridade de seu referente. Por exemplo, um som estridente emitido pelo sistema representa a ideia de que algo está errado. **Índice** é o signo é evoca a segundidade de seu referente. Por exemplo, o botão com a imagem de uma impressora é um índice porque espera-se que o usuário associe a imagem com a funcionalidade "Imprimir documento". **Símbolo** é o signo que evoca a terceiridade'. Por exemplo, um texto explicativo, em linguagem natural, mostrando o que um botão de uma interface faz.

Considerando estas definições e tipos de signos, de Souza (de Souza, 2005) argumenta que todo sistema computacional é uma codificação específica de uma interpretação de seus desenvolvedores sobre as necessidades, expectativas e preferências dos usuários. Esta interpretação específica é "codificada" em uma linguagem de programação para se concretizar como

software. Tais linguagens têm o poder expressivo de UARS e o sistema codificado é tipicamente um MARS. O caminho pelo qual tal MARS é construído é resultado de relações mediadoras de signos de terceiridade (símbolos) que, se "opacos" para os usuários, têm alta probabilidade de gerar problemas de interação.

Posteriormente, usaremos estas definições de signos para analisar quais tipos de signos são utilizados para expressar as propostas de modificação de sistemas em nosso modelo.

2.3 Engenharia Semiótica

Esta seção apresenta em maior detalhe a Engenharia Semiótica, teoria de IHC que fundamenta esta pesquisa. Primeiramente damos uma visão geral da teoria, apresentando os principais conceitos. Em seguida, citamos contribuições da Engenharia Semiótica para o desenvolvimento de aplicações extensíveis, destacando o modelo semiótico de Cunha, o qual serviu de ponto de partida para a proposta do nosso modelo de discussão. Por último, descrevemos trabalhos de Engenharia Semiótica sobre aplicações de *groupware*.

2.3.1 Uma visão Geral

Semiótica é uma disciplina dedicada a estudar sistemas de significação e processos de comunicação (Eco, 1976). Nas abordagens semióticas de IHC, um sistema computacional é considerado um ato de comunicação do designer (emissor) para os usuários que o utilizam (receptores). Assim, também para a Engenharia Semiótica, IHC é um caso particular de comunicação humana mediada por sistemas computacionais (de Souza, 2005). O traço distintivo desta teoria em relação a outras teorias semióticas de IHC ((Andersen, 2007), (Nadin, 1997)) é: (i) postular claramente que o fenômeno de interação humano-computador é um caso de *metacomunicação* (ou seja: comunicação sobre e através de outra comunicação); (ii) apresentar uma estrutura para este espaço de metacomunicação (baseada nas funções de comunicação propostas por Jakobson (1960)); (iii) apresentar uma estrutura mínima e geral para o conteúdo da mensagem de metacomunicação enviada de projetistas para usuário através da interface de sistemas; e (iv) postular que a interface dos sistemas representa os projetistas (e fala por eles, em linguagem de interface) em tempo de interação. Sobre o conteúdo da mensagem de metacomunicação, a Engenharia Semiótica propõe o seguinte *template* que abstrai o teor desta

mensagem (a primeira pessoa do discurso é um projetista e a segunda, a quem ele se dirige, um usuário):

> *"Esta é a minha interpretação sobre quem você é, o que eu entendi que você quer ou precisa fazer, de que formas prefere fazê-lo e por quê. Este é portanto o sistema que projetei para você, e esta é a forma como você pode ou deve usá-lo para atingir os objetivos incorporados na minha visão".*

Um dos principais objetivos do designer ao preencher esse *template* e expressá-los através de elementos interativos em uma interface é comunicar ao usuário como o processo de comunicação dele com o sistema, para realizar vários tipos de efeitos e objetivos, pode ocorrer.

A mensagem de metacomunicação (muitas vezes chamada, na Engenharia Semiótica, de metamensagem) é descrita através de signos, que podem ser: estáticos, dinâmicos e metalinguísticos. Signos *estáticos* são aqueles cujos significados são interpretados independentemente das relações causais e temporais que permeiam a interação. São os signos cuja interpretação é limitada pelos elementos visíveis na interface em um determinado instante único no tempo (de Souza & Leitão, 2009). São exemplo de signos estáticos: botões, links, menus, bem como composições sígnicas das quais eles participem (por exemplo, janelas ou áreas estruturadas no *layout* de uma tela de interface). Os signos dinâmicos são aqueles cuja interpretação está sujeita às relações causais e temporais, ou seja, a interação em si. A sua identificação é mais sutil, pois não há necessariamente um elemento visível único que o represente. Por exemplo, a relação causal entre a seleção de um botão na barra de ferramentas e o diálogo que se segue a esta ação é um signo dinâmico, que só pode ser identificado com a interação (de Souza & Leitão, 2009). Um outro exemplo de signo dinâmico é o efeito resultante de acionar uma opção de menu na interface, ou de clicar um botão. Finalmente, signos metalinguísticos são aqueles usados pelo designer para comunicar explicitamente para os usuários os significados que ele atribuiu para os demais signos codificados na interface e como eles devem ser usados (de Souza & Leitão, 2009). Por exemplo: sistema de ajuda, explicações, mensagens de erro, avisos, etc.

É importante citarmos que as interpretações sobre um mesmo objeto podem ser ilimitadas. Por exemplo, o significado de um relógio pode ser tempo, pressa ou a sensação de que as horas não passam. Esse processo de geração de inúmeros significados, por natureza, não tem limites ou direções completamente previsíveis, e é denominado de semiose ilimitada (Eco, 1976). Por esse motivo, os significados das coisas estão sempre em evolução. Enquanto

o sistema computacional está sendo desenvolvido, as atividades, os conceitos e as necessidades dos usuários podem estar mudando. Logo, podemos dizer que quando o sistema é finalmente entregue para o usuário utilizar, a evolução constante das interpretações sobre o mundo e as situações cotidianas faz com que o usuário não seja mais o mesmo, havendo pois uma grande chance do sistema já não atender mais a todos os requisitos dele. Além disso, uma vez que o usuário passa a usar o sistema, este desencadeia a geração de novos significados e reinterpretações da realidade à luz da tecnologia agora disponível. Logo, todo sistema dispara uma evolução semiótica na vida dos usuários, a qual pode (talvez paradoxalmente) fazer surgir barreiras de uso para o próprio sistema. Uma alternativa para endereçar este desafio de evolução constante do significado das coisas, e portanto dos requisitos e condições de uso dos sistemas, seria se o sistema permitisse ser adaptado/estendido.

2.3.2 Engenharia Semiótica de Aplicações Extensíveis

Tipos de Modificações

Quando se fala em adaptar ou estender sistemas computacionais, a Engenharia Semiótica chama atenção para os limites dessas mudanças. Isso porque algumas customizações ou extensões podem afetar a identidade da aplicação, identidade esta que está no centro da metamensagem dos designers para usuários. Modificações que por acaso contradigam ou destruam significados primários desta metamensagem, descaracterizam o artefato de metacomunicação. Em (de Souza & Leitão, 2009) as autoras citam um exemplo de uma extensão que muda a intenção original de *design* do sistema estendido. Elas fazem referência a possibilidade de "customizar" a interface do Word a tal ponto que ele se transforma em um editor de HTML ou editor gráfico. Neste caso, a identidade da aplicação é afetada. Portanto, para a Engenharia Semiótica, as aplicações devem ter signos impermeáveis, ou seja, signos que não podem ser modificados por usuário final.

De Souza (de Souza, 2005) faz uma reflexão sobre as adaptações que podem ser feitas na instalação de um sistema. Por exemplo, a escolha de uma instalação padrão pode deixar de fora algumas funcionalidades do sistema. De qualquer forma, a escolha do usuário vai afetar a base semântica do sistema, e consequentemente, o conjunto de signos impermeáveis. Diante disto, a autora classifica os signos impermeáveis em duas categorias. Signo signos "Essenciais" são aqueles que não podem ser removidos da interface, mesmo que a instalação do sistema seja a mais simples possível. Ou seja, são os signos que existirão

de qualquer maneira e que preservam a identidade da aplicação. Já os signos que fazem parte somente de funcionalidades que podem ser retiradas e que compõem a identidade da funcionalidade (e não do sistema todo), eles são chamado de "Acidentais". Um típico exemplo de onde vemos signos acidentais é quando ocorre a desinstalação de verificadores ortográficos em um editor de texto. Se esta ação é executada, um conjunto de signos é removido da aplicação sem afetar a identidade da mesma. Isto porque os signos removidos não são impermeáveis para o sistema como um todo (i.e. um editor de texto sem verificador ortográfico continua sendo um editor de texto; o que não é verdade sobre um editor de texto sem as funções de inserção, apagamento, busca e substituição de texto, por exemplo). É importante ressaltar que signos acidentais e essenciais não devem poder ser modificados em atividades de customização/extensão, sob pena de desconfigurarem a metamensagem do designer para os usuários.

De Souza (2005) analisa os possíveis tipos de modificação que os sistemas de significação das aplicações computacionais podem sofrer. Essa análise é realizada em duas perspectivas: a do usuário e a do sistema computacional. Isto porque as linguagens dos seres humanos e a dos sistemas computacionais possuem diferentes dimensões. Para os seres humanos a linguagem é um meio de comunicação que contém as dimensões de intenção, conteúdo e expressão. Já para os computadores a linguagem é apenas um sistema simbólico, cujo processamento envolve as dimensões léxica, sintática e semântica. Então, apesar de pessoas e computadores usarem as mesmas linguagens para se "comunicarem" através de interfaces de sistema, o que fazem com elas e a maneira como as utilizam é totalmente diferente. Ou seja, o "significado" da linguagem para seres humanos e para computadores é bem diferente. Em (de Souza & Barbosa, 2006a) as autoras mostram que não existe uma correspondência direta entre as dimensões da comunicação humana e do processamento simbólico (Figura 2.8).

As dimensões de expressão humana acabam sendo mapeadas às dimensões léxica e sintática dos computadores, e a dimensão semântica do processamento computacional deve dar conta das dimensões de conteúdo e intenção da comunicação humana. Diante disto, modificações consideradas simples do ponto de vista do usuário podem ser complexas do ponto de vista computacional, e vice-versa. Por exemplo, supondo que um usuário de um editor de texto quer mudar a expressão do comando que copia um elemento selecionado, ao invés de digitar "Ctrl + C" ele quer digitar "↑ + P". No ponto de vista do usuário essa alteração envolve apenas uma mudança na "expressão" de uma funcionalidade oferecida pelo editor de texto, já na perspectiva do

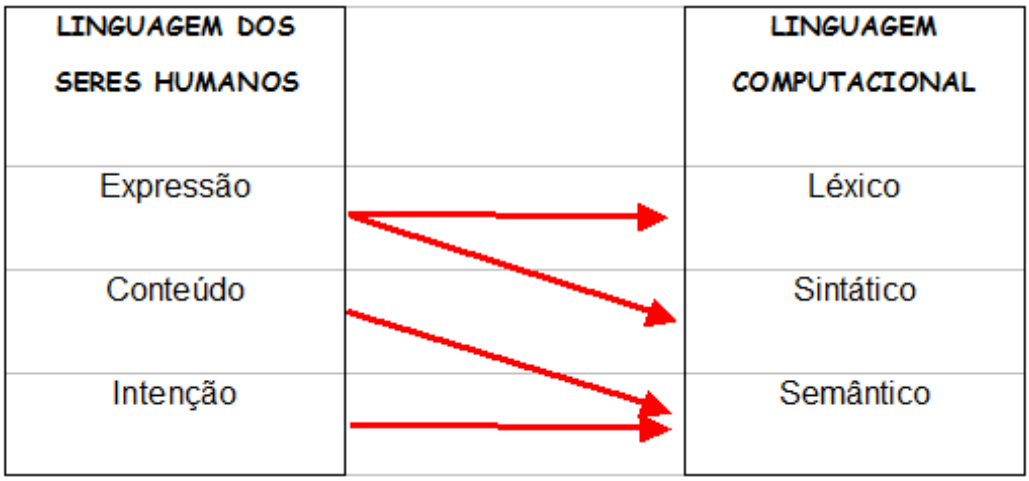

Figura 2.8: Dimensões da Comunicação Humana versus Computacional (Fonte: (de Souza & Barbosa, 2006a))

sistema, envolve alteração nas dimensões léxica e sintática, se a combinação dos dois itens lexicais não estiver prevista como tipo gramatical autorizado (o que é comum acontecer com outros elementos).

O fato de o mapeamento ser cruzado é uma complexidade a mais para se tratar porque uma aplicação computacional envolve o uso de uma única linguagem por usuário e sistema. No contexto desta pesquisa, este é o caso de linguagens de interface, de configuração, de scripting, ou de quaisquer outras que um usuário tenha de utilizar para descrever uma extensão ou customização que queira fazer. A forma como o usuário (humano) interpreta e usa esta linguagem envolve dimensões diferentes das dimensões em que o sistema (computador) as processa. E, sendo o mapeamento cruzado, este é um desafio duplo: tanto para o designer de sistemas extensíveis e customizáveis comunicarem aos usuários este cruzamento, quanto para os usuários finais estabelecerem o significado computacional (implementável ou implementado) daquilo que faz sentido para eles.

Considerando esse fenômeno, apresentamos a seguir todas as possibilidades de modificações de acordo com as duas perspectivas.

Tipos de Modificações dos Sistemas de Significação na Perspectiva do Sistema

A Figura 2.9 apresenta sete possibilidades de modificação do sistema de significação na ótica do sistema computacional. Os três primeiros tipos preservam o significado, e os quatro últimos introduzem novos significados.

O primeiro tipo (I) representa a inclusão de sinônimos na linguagem, pois altera somente o vocabulário. Por exemplo, considerando a linguagem de

	Operações sobre **símbolos da linguagem**	LEX	SINT	SEM
I	Alterar apenas vocábulos (léxico)	■		
II	Alterar apenas a gramática (regras sintáticas)		■	
III	Alterar vocábulos e gramática (semântica igual)	■	■	
IV	Alterar apenas significados (não vocábulos e gramática)			■
V	Alterar vocábulos e significados (não gramática)	■		■
VI	Alterar gramática e significados (vocábulos iguais)		■	■
VII	Alterar tudo (vocábulos, gramática e significados)	■	■	■

Figura 2.9: Possibilidades de Alterações dos Sistemas de Significação na Perspectiva do Sistema Computacional

interface (UIL[3]), uma mudança no vocabulário seria a alteração do nome de um botão. O segundo (II) é o caso onde altera-se somente a combinação prescrita de elementos nas regras gramaticais, apesar de não ser muito útil introduzir novos tipos gramaticais se o vocabulário e as regras semânticas permanecem iguais. Este tipo de alteração é feita, por exemplo, em aplicações que permitem ao usuário mudar a ordem das opções de menu. Já o tipo (III) significa incluir novo vocábulo e novas regras de combinações para interpretações (ou significados) já existentes no modelo semântico da linguagem. Esses três tipos de alteração são considerados casos de "customização", pois não alteram o espectro de interpretação que o sistema pode derivar.

Os próximos tipos (IV, V, VI e VII) são casos de "extensões" de sistemas. O tipo V explicitamente acrescenta novo significado à linguagem original. Todavia, a sintaxe permanece intacta. Esse é o caso onde os novos itens léxicos do vocabulário são "tokens" de tipos sintáticos existentes. Esse tipo de extensão está presente em aplicativos que permitem usuários criarem objetos (e.g., documentos, projetos, imagens, etc.) dando-lhes novos nomes, conforme sua escolha. Cada novo nome é uma instância de um mesmo tipo gramatical ("nome do arquivo", por exemplo), cuja combinatória simbólica está prevista em regras sintáticas que permanecem inalteradas e cuja interpretação semântica também já está plenamente prevista. Sempre que um usuário salva um novo documento com qualquer nome, o código da aplicação ganha um novo item léxico. O tipo VI é o caso em que a combinação dos vocábulos é alterada para obter nova semântica. O tipo IV introduz semântica sem afetar

[3]UIL é um acrônimo de User Interface Language, que significa Linguagem de Interface do Usuário. Como esse termo é bastante encontrado na literatura de IHC, o adotaremos neste documento para fazer referência à linguagem de interface.

o vocabulário. Isto representa uma mudança radical nos signos existentes. Finalmente, o tipo VII afeta todas as dimensões do processador simbólico.

Tipos de Modificações dos Sistemas de Significação na Perspectiva do Usuário

A Figura 2.10 apresenta as possibilidades de customização e extensão na perspectiva do usuário. Ao invés de léxico, sintático e semântico, temos as dimensões expressão, conteúdo e intenção. Assim como na ótica do sistema computacional, o tipos de alterações são classificados em dois grupos, nesse caso são: alterações independentes da intenção e alterações que expandem o escopo da intenção. Assim como na perspectiva anterior, os três primeiros tipos de alteração são considerados casos de "customização", pois não alteram os significados e os tipos (IV, V, VI e VII) são casos de "extensões" de sistemas.

	Operações sobre **signos da comunicação**	EXP	CONT	INT
I	Alterar apenas a expressão (não conteúdo e intenção)	■		
II	Alterar apenas o conteúdo (não expressão e intenção)		■	
III	Alterar expressão e conteúdo (intenção igual)	■	■	
IV	Alterar apenas intenção (não expressão e conteúdo)			■
V	Alterar expressão e intenção (conteúdo igual)	■		■
VI	Alterar conteúdo e intenção (expressão igual)		■	■
VII	Alterar tudo (expressão, conteúdo e intenção)	■	■	■

Figura 2.10: Possibilidades de Alterações dos Sistemas de Significação na Perspectiva do Usuário

As Figuras 2.9 e 2.10 representam o espaço de todas as possíveis modificações. Nele existe um conjunto de modificações que alteram a semântica ou a intenção, que é representado pela última coluna das tabelas. Estamos destacando esta coluna porque ela representa a alteração no "significado" semântico (perspectiva do sistema) ou pragmático (perspectiva do usuário) das coisas.

Os níveis léxico, sintático e semântico da perspectiva do sistema e expressão, conteúdo e intenção da perspectiva do usuário são níveis gradualmente mais profundos de significados. Então nos níveis da "semântica" e da "intenção" as mudanças afetam o "cerne" do significado. Assim, as mudanças que envolvem estes níveis são naturalmente mais complexas. Este

cerne é o que deveria ser impermeável, pois é ele que preserva a intenção de design.

É importante esclarecer que nesta pesquisa estamos considerando **evolução** de sistemas computacionais, as possibilidade de ALTERAR funções do sistema, que corresponde ao conceito de customização da Engenharia Semiótica, ou de CRIAR novas funções, que corresponde ao conceito de extensão também desta teoria. Ressaltamos ainda que estamos focando a ótica do usuário, isto é, a semântica do modelo de uso, e não a ótica do designer, ou seja, a semântica do modelo de sistema enquanto canal, mensagem e linguagem de comunicação (de Souza, 2005). Um dos benefícios da distinção entre os tipos de extensão e customização é ajudar-nos a identificar prós e contras de várias técnicas de EUD. De Souza levanta algumas possibilidades: talvez seja interessante projetar aplicações de maneira que todos os tipos de customização afetem somente componentes de interface (não alterando a arquitetura do sistema). Outra opção seria permitir extensões em que as alterações dos significados seriam definidas de maneira a afetar apenas um subconjunto pré-estabelecido da semântica do sistema.

Abstração Interpretativa e Contínuo Semiótico

A Engenharia Semiótica propõe dois princípios para avaliar linguagens de programação para usuários finais (de Souza et al., 2001). São eles: o Princípio da Abstração Interpretativa e o Princípio do Contínuo Semiótico. A essência destes princípios é que: (a) se olharmos um texto (seja interativo ou descritivo) temos que entendê-lo independente do modelo semântico que está por trás dele. Por exemplo, olhando a UIL podemos entendê-la sem conhecer nada sobre a linguagem de programação que está por trás. (b) Ao usar o princípio do Contínuo Semiótico para aplicações computacionais interativas, temos que poder olhar o texto e enxergar conceitualmente como aquilo pode ser usado, ou seja, é preciso usar uma componente pragmática para descrever a linguagem.

Dizer que a UIL é uma abstração interpretativa de uma ou mais linguagens é dizer que os significados das frases e dos termos da UIL são especificados nestas linguagens, e que o usuário deve estar apto a entender e interagir com a UIL sem ter nenhum conhecimento sobre as linguagens subjacentes. Em outras palavras, a UIL é uma abstração interpretativa de linguagens computacionais de baixo nível se usuários podem entender completamente todos os signos da UIL em virtude dos seguintes fatores:

(i) O padrão e a combinação de signos que eles encontram enquanto interagem com a aplicação; (ii) As explicações disponíveis na UIL (como por exemplo, sistema de ajuda, tutoriais, *tootips*); (iii) Sua própria experiência

computacional.

O contínuo semiótico diz que se o usuário consegue escrever numa linguagem L antevendo como é a tradução na UIL, então podemos dizer que este princípio é válido para para as linguagens L e UIL. Adicionalmente, se o usuário espera gerar símbolos em um sistema partindo de outro, então ele deve conhecer os sistemas de significação adjacentes ao sistema de significação original. Por exemplo, "Salva como txt" é um signo adjacente aos aplicativos de edição de texto, logo ele é um signo em potencial a ser incluído em um editor de texto, já o signo "Salvar como 50 por cento" não é um signo conhecido deste domínio e consequentemente não seria adequado utilizá-lo.

Estes princípios podem ser usados para analisar linguagens, ou sistemas de significação, quando os significados de uma são definidos ou expressos em outra. Queremos usar estes princípios para avaliar se a linguagem a qual o usuário pode expressar modificações de sistemas é boa ou não. A situação ideal é que ela atenda aos dois princípios.

2.3.3 Engenharia Semiótica de Aplicações de Groupware

Projetar sistemas de *groupware* é ainda uma atividade complexa. Um dos maiores problemas é o efeito social causado pelo sistema sobre os grupos ou comunidades de usuários e uma maneira de endereçar isto é fazer com que os projetistas reflitam cuidadosamente sobre implicações éticas e sociais dos artefatos que eles estão produzindo (Dourish, 2001).

A Engenharia Semiótica vem contribuindo para promover a reflexão dos *designers* sobre aspectos sociais. As principais propostas são: (i) A Inspeção Semiótica de Aplicações Multi-usuários proposta em (de Souza, 2005) e (ii) Manas (Barbosa, 2006), a qual é o resultado de pesquisas iniciadas em (Prates, 1998) e estendida por Barbosa em (Barbosa, 2002). Estas são descritas a seguir.

"Inspeção Semiótica de Aplicações Multi-usuários"

Antes de apresentarmos a inspeção é importante dizer que estamos referindo-nos a aplicações "multi-usuários" como aquelas que objetivam apoiar ou promover a interação humana "online".

O método de inspeção é guiado por quatro conjunto de perguntas, as quais estão classificadas levando em consideração os componentes do modelo de comunicação de Jakobson (Interlocutores, Mensagem, Código, Canal e Contexto) (Jakobson, 1960).

O primeiro conjunto de perguntas apoia a reflexão sobre os interlocutores do sistema que está sendo projetado. Os projetistas devem refletir sobre "Quem está falando?" e "Para quem?". Para isso, eles devem responder questões como: Os interlocutores estão representados? A "direção" da interlocução está representada? Quem são os possíveis emissores e receptores de mensagens? Quais os sistemas de símbolos são usados? Quem produz os signos? Quando e como? Existem representações icônicas?

O segundo conjunto de perguntas inspeciona a mensagem e o código, ou seja, os projetistas devem refletir sobre o conteúdo do que está sendo dito e qual o meio e códigos utilizados. Para isso, eles precisam pensar em fatores como: os emissores e receptores estão supondo compartilhar o mesmo sistema de significação? O emissor poderia codificar sua mensagem de maneira diferente? Se existem outras alternativas de códigos e meio de comunicação, por que e quando usá-las?

O terceiro conjunto de perguntas está associado ao canal de comunicação, ou seja, os projetistas devem refletir se os receptores estão recebendo a mensagem. Caso ela não chegue, como o sistema pode ajudar a comunicar isto? Para isso ele deve responder perguntas como: Existe a representação do canal e de um "status" de recepção? Se o canal de comunicação está com problemas ou os receptores não estão disponíveis, a quebra na comunicação pode ser evitada ou resolvida? Como?

O quarto e último grupo de questões envolve a inspeção de respostas dos receptores e contexto. Os projetistas devem refletir se existem recursos no sistema para tratar equívocos de entendimento sobre a mensagem. Para isso, os projetistas devem pensar sobre: a resposta do receptor foi uma ação equivocada? Se foi, essa ação pode ser detectada e revogada? Como? A revogação da comunicação implicará em uma nova rede de compromissos? Poderia essa rede ser facilitada pelo sistema?

Esta inspeção semiótica funcionou como uma ferramenta epistêmica na elaboração do modelo para expressar e comunicar evolução de sistemas. O conjunto de perguntas sugeridas nos levou a refletir sobre os elementos do modelo de comunicação de Jakobson (Interlocutores, Mensagem, Código, Canal e Contexto), os quais fazem parte do nosso modelo.

"Manas: uma ferramenta epistêmica de apoio ao projeto da comunicação em sistemas colaborativos" (Barbosa, 2006)

A maioria dos trabalhos que investigam sobre o desenvolvimento de aplicações multi-usuários não explicitam a comunicação entre os membros do grupo, em geral esta comunicação está representada

implicitamente (Barbosa et al., 2007). Objetivando preencher esta lacuna Manas (Barbosa, 2006) (Barbosa et al., 2007) é uma ferramenta epistêmica para apoiar designers na representação e reflexão sobre como usuários podem ou devem comunicar-se uns com os outros. Manas separa as dimensões comunicativas das tarefas do sistema e provê uma linguagem para o *designer* representar a comunicação. Além disso, Manas dá um *feedback* qualitativo de potenciais efeitos sociais que as decisões de *design* podem causar no grupo. Ao fazer isso, Manas leva o projetista a refletir sobre problemas que estão sendo abordados e suas soluções.

O uso do modelo Manas não tem o propósito de indicar uma solução para os projetistas, mas sim de apontar como as tomadas de decisão do projetista podem impactar a experiência do usuário com respeito a aspectos sociais, tais como privacidade, polidez e eficiência na comunicação.

Manas é fundamentado na Engenharia Semiótica, caracterizando a interação humano-computador como um fenômeno de meta-comunicação designer-usuário. Manas deixa claro para os projetistas que suas decisões são levadas aos usuários através do sistema, influenciando a experiência dos usuários.

Os componentes da arquitetura Manas são:

- Linguagem de Design (L-ComUSU), a qual descreve o modelo de comunicação usuário-sistema-usuário.
- Interpretador da linguagem de *design* cuja saída é o modelo de comunicação e assertivas de *feedback* (análogo aos avisos de alerta produzidos por interpretadores de linguagem de programação).
- Uma anotação derivativa do m-ComUSU, o qual equivale ao *Design Rationale*.

O modelo de comunicação da arquitetura Manas (Figura 2.11) é definido em termos de "estruturas comunicativas", as quais são compostas de "atos comunicativos" e envolvem "interlocutores". Um ato comunicativo pode ser um "discurso" ou uma "conversa" e tem sempre uma intenção (propósito). Os interlocutores podem participar, seja em conversa ou discurso, como receptores e/ou emissores de mensagens. Adicionalmente, uma mensagem pode (ou não) ser endereçada para um ou mais interlocutores.

Esse trabalho é uma referência importante para esta tese porque apresenta uma contribuição importante da Engenharia Semiótica para pesquisas sobre aplicações de grupo. De certa forma, estamos evoluindo o estudo sobre comunicação em grupo mediada por tecnologia. O modelo de comunicação Manas é um avanço para representar a comunicação por meio

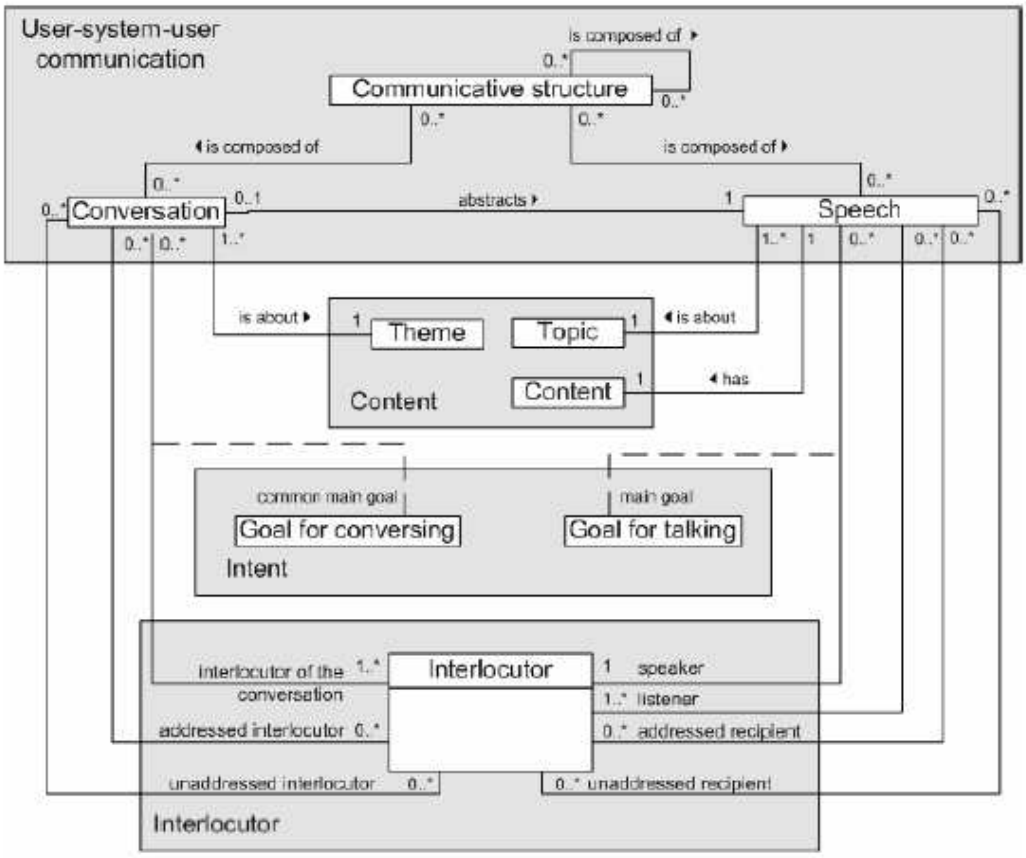

Figura 2.11: Modelo de Comunicação (m-ComUSU) da arquitetura Manas. Extraído de (Barbosa et al., 2007).

tecnológico entre os membros de um grupo, porém, ele não considera os diferentes papéis que os membros de um grupo podem assumir. Nem os papéis sociais e nem os papéis protocolados na tecnologia. Esta lacuna é objeto de investigação em nossa pesquisa, pois estamos propondo um modelo para auxiliar a comunicação entre usuários de sistemas de *groupware*, considerando os diferentes papéis que as pessoas podem exercer e que estão protocolados na tecnologia.

A organização de signos na metacomunicação de aplicações multi-usuário

A Engenharia Semiótica faz uma análise semiótica do projeto da mensagem de metacomunicação de aplicações multi-usuários. Para ajudar a refletir sobre aplicações multi-usuários, de souza criou três metáforas, descritas em (de Souza, 2005), que classificam estas aplicações. São elas:

a) **O sistema como uma central de comunicação:** Nesta metáfora as aplicações são vistas como um provedor de serviços para atender a pedidos de usuários sobre comunicação. Um exemplo desta metáfora, apresentado por de Souza é o aplicativo MNS Messenger.

b) **Ambiente Virtual:** Esta é a metáfora que representa aplicações onde a comunicação acontece em um espaço virtual onde usuários são projetados nele. Um exemplo típico desta metáfora é o aplicativo "Second Life".
c) **Dispositivo de telecomunicação:** o aplicativo é visto como uma máquina, a qual o usuário pode manipular como se fosse um dispositivo físico. Um exemplo deste tipo de metáfora é a interface *Phone Works*, um aplicativo que envia, recebe e gerencia todas as mensagens de fax. Sua interface é representada através de uma analogia com a máquina física de fax (ver Figura 2.12).

Figura 2.12: Interface do *PhoneWorks* (Fonte: www.faxcall.com)

Estas três metáforas conceituais servem para mostrar que o desafio semiótico de obter a mensagem de metacomunicação é diferente em cada uma delas. Na metáfora onde o sistema é visto como uma central de comunicação a conversa entre usuário e preposto do *designer* é facilitada. Já a metáfora do ambiente virtual é um desafio, semioticamente falando. Isto por que os signos que regulam a comunicação face-a-face, os quais os *designers* se esforçam para imitar, se referem a qualidades e percepções que os ambientes virtuais atuais não conseguem implementar apropriadamente. Tais qualidades e percepções são representadas por signos de segundidade e terceiridade que precisam ser aprendidos pelos usuários. Por último, a metáfora do dispositivo de telecomunicação preserva os signos de mediação e a separação física, na comunicação. O usuário não comunica-se com o dispositivo, mas através dele.

Na comunicação face-a-face a presença de alguém é sentida ou percebida antes de ser interpretada. Em um ambiente virtual, a presença pode ser sentida por signos de segundidade (como som e/ou imagem da outra pessoa), ou por signos de terceiridade (como nomes, mensagens ou até figuras de "avatars"). Este fenômeno tem consequências importantes na relação entre

pessoas mediada por computador. O fato dos usuários serem privados de signos de primeiridade, ele devem significar a primeiridade em segundidade ou terceiridade. Adicionalmente pode ocorrer deles precisarem desenvolver convenções para contornar a perda da dimensão de significação. Um exemplo de problema causado pela perda da primeiridade tem haver com "confiança e verdade" nas salas de *chat* (a pessoa não confia na outra porque não está vendo). Outro exemplo é a necessidade de expressar explicitamente (através de segundidade ou terceiridade) certas atitudes que são claras quando comunicadas em primeiridade. Por exemplo, considerando que não estamos interessados em uma conversa, na comunicação presencial a linguagem corporal pode sutilmente mostrar isso, todavia, numa sala de *chat* seríamos rudes se disséssemos: "Não estou interessado no que você está dizendo" (de Souza, 2005).

Diante disso, um dos grandes desafios de projetar aplicações multi-usuários de acordo com a metáfora do ambiente virtual é que é quase impossível significar apropriadamente a primeiridade. Além disso, o fato destes signos (primeiridade) exercerem um maior papel nas convenções culturais das pessoas, a qualidade de interação humana mediada pelo computador pode ser um risco. Em (de Souza, 2005), pg. 215, a autora diz que: O desafio de projetar surge da primeiridade uma vez que a intenção do *designer* deve ser transmitida através de qualidades que podem ser diretamente percebidas e interpretadas por usuários, em um tipo de semiose primária que artefatos computacionais raramente apoiam.

Estas metáforas nos ajudam a refletir sobre a metacomunicação do nosso modelo, no qual as representações das intenções dos usuários podem ser expressas através de uma combinação de tipos de signos: ícone (primeiridade), índice (segundidade), ou símbolo (representação descritiva ou argumentativa textual - terceiridade).

"Um modelo Semiótico dos Processos de Comunicação Relacionados à Atividade de Extensão de aplicações por Usuários Finais" (Cunha, 2001)

No âmbito da Engenharia Semiótica, desenvolver aplicações com a participação de usuários finais é torná-los co-autores da mensagem de metacomunicação, portanto é importante oferecer uma tecnologia que apoie a construção colaborativa da mesma e que esta tecnologia seja acessível por pessoas sem treinamento técnico em desenvolvimento de sistemas. Cunha (Cunha, 2001) contribuiu para avanços nesse sentido propondo um modelo que organiza e caracteriza os processos comunicativos relacionados à extensão colaborativa de sistemas. Ele é apresentado nesta seção.

Seu modelo, ilustrado na Figura 2.13, segue a estrutura de comunicação verbal de Jakobson (Jakobson, 1960), para quem um processo de comunicação ocorre quando, em determinado **contexto**, um **emissor** transmite uma **mensagem** a um **receptor**, através de um **canal**. Tal mensagem é escrita conforme as regras de um **código**, que deve ser conhecido pelos interlocutores. No modelo de Cunha, os interlocutores são: Designers de Software, Usuários como Estendedores, Usuários como co-estendedores e usuários comuns. Todos podem ser emissores ou receptores no processo de comunicação sobre modificações que poderiam ser feitas no sistema para torná-lo mais adequado às necessidades do grupo de usuários. As mensagem trocadas são relativas, portanto, a mudanças nas características de um sistema que todos usam, podendo incluir significados que não estão presentes na aplicação original. O canal é o meio físico utilizado para transmitir mensagens. O contexto envolve o domínio da aplicação que está sendo discutida, sistema computacional, o ambiente onde os interlocutores constroem a mensagem e a experiência do usuário (visto que em uma abordagem semiótica a experiência do usuário influencia na interpretação de mensagens, portanto faz parte do contexto do receptor). As mensagens são construídas utilizando um ou mais códigos, que podem ser: linguagem de interface (UIL), de explicação, de extensão (EUPL) ou de comunicação.

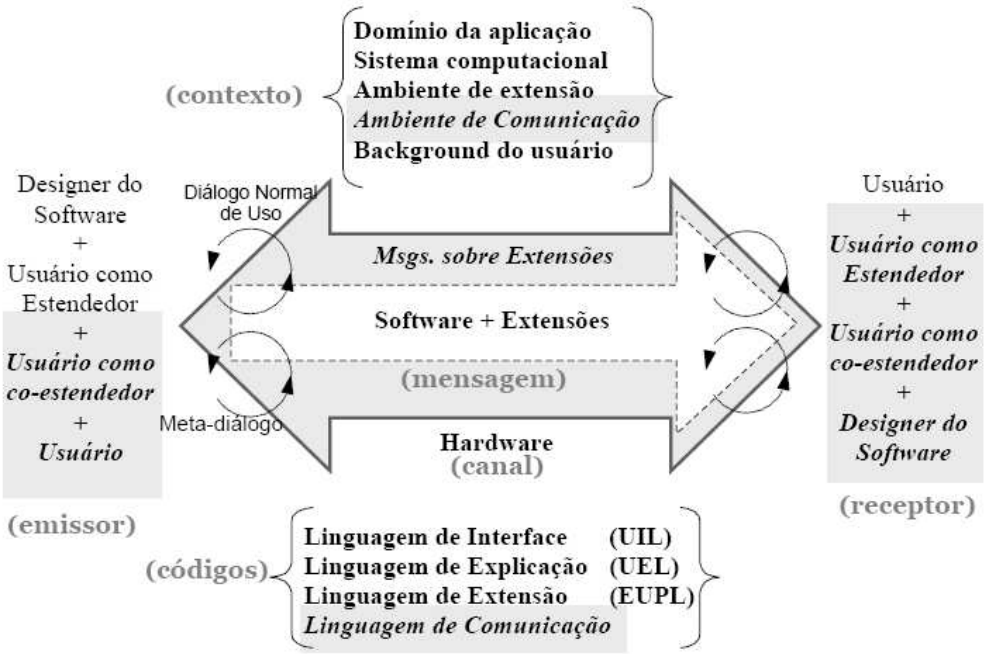

Figura 2.13: Modelo de Cunha estraído de (Cunha, 2001)

A autora analisa separadamente a parte da mensagem que corresponde à intenção de comunicação e de design da extensão, observando que elas

são idéias abstratas e tendem a ser expressas em linguagem natural, embora a linguagem natural seja o sistema mais adequado para cobrir por si as necessidades e capacidades cognitivas e comunicativas dos seres humanos, ela permite a ocorrência de ambiguidades e não é completamente processável pelo computador.

Diante disso, Cunha defende que a linguagem de representação de uma modificação deve idealmente ser compreensível para os agentes humanos e processada pelo computador. Diante disto, a linguagem de interface ou *User Interface Language*(UIL), que pode ser usada por usuários e sistemas, tem a vantagem de ser um código conhecido pelo usuário final e consequentemente seu esforço cognitivo ao utilizá-la é pequeno. Além disso, a possibilidade de processar automaticamente as informações nela contida tem inúmeras vantagens, dentre elas a possibilidades de controle e gerenciamento automático das mensagens, o apoio automático nas decisões de projeto, checagem de incoerências e inconsistências nos discursos, etc.

Cunha acrescenta que só a UIL não é suficiente para o usuário expressar suas ideias. Se por um lado a UIL possui um baixo custo de aprendizado, por outro, ela apresenta uma baixa expressividade quando se trata de comunicar algo que, por definição, sendo uma extensão, ela ainda não significa (ie. algo que ainda não pode ser entendido e gerado pelo sistema). Já as linguagens de extensão (EUPL) com expressividade que no limite pode ser equivalente à de uma máquina de Turing terão tanto mais alto custo de aprendizado quanto mais expressivas forem. Cunha sugere que "...os estendedores devem ser capazes de combinar expressões do código da UIL e da EUPL com expressões de um outro código... o ideal é que este outro código tenha um baixo custo de aprendizado, como por exemplo a linguagem natural."

Particularmente, este trabalho de Cunha é bastante importante para o desenvolvimento desta pesquisa, pois assim como ele, estamos nos fundamentando na Engenharia Semiótica, ou seja, vemos sistemas computacionais como artefatos de metacomunicação entre *designers* e usuários. Além disso, estamos trabalhando com a problemática de usuários terem que representar modificações em sistemas. Adicionalmente, adotamos como premissa o seguinte argumento de Cunha: **a linguagem mais adequada para usuários representarem modificações em sistemas é a UIL.**

Basicamente, existem duas principais diferenças entre o trabalho de Cunha e nossa pesquisa. Primeiro Cunha trabalhou no contexto de design e construção de sistemas extensíveis, já nossa pesquisa foca sobre evolução de sistemas já construídos na Web. Segundo, a linguagem X-DIS foca sua contribuição em um sistema de representação simbólica (para uso dos

designers e construtores de sistemas extensíveis), enquanto nosso foco recai sobre uma representação extensamente icônica para uso de usuários finais, sobretudo, embora também de designers e construtores de sistemas (embora somente quando em discussões com usuários ou com quem advoga por eles - por exemplo, avaliadores de IHC). Como a X-DIS é uma representação intermediária entre os modelos de IHC e a UIL ela tem um nível de abstração marcadamente diferente da linguagem que propomos, a qual está num nível de abstração mais alto, bem próxima da UIL.

Neste capítulo podemos ver os avanços da tecnologia Web no que diz respeito ao desenvolvimento de sistemas realizado por usuários finais. Adicionalmente, vimos até onde a Engenharia Semiótica contribuiu sobre o projeto de aplicações de groupware e/ou extensíveis/customizáveis. Definimos o contexto e os principais conceitos envolvidos neste trabalho. A seguir, apresentamos o modelo para descrever e negociar modificações em sistemas e depois como parte deste modelo foi implementado (ferramenta TiWIM).