# UNIVERSIDADE FEDERAL DA PARAÍBA

CENTRO DE CIÊNCIAS APLICADAS E EDUCAÇÃO

DEPARTAMENTO DE CIÊNCIAS EXATAS

**BACHARELADO EM SISTEMAS DE INFORMAÇÃO**


# CACHE PARA FRAGMENTOS DE GUI UTILIZANDO METAMODELOS INTERPRETADOS


THIAGO PONTES DE OLIVEIRA

Orientador: Prof. Msc. Rodrigo Almeida Vilar Miranda



RIO TINTO - PB

2013

THIAGO PONTES DE OLIVEIRA

# CACHE PARA FRAGMENTOS DE GUI UTILIZANDO METAMODELOS INTERPRETADOS

Monografia apresentada para obtenção do título de Bacharel à banca examinadora no Curso de Bacharelado em Sistemas de Informação do Centro de Ciências Aplicadas e Educação (CCAE), Campus IV da Universidade Federal da Paraíba.
Orientador: Prof. Msc. Rodrigo Almeida Vilar Miranda.

RIO TINTO - PB
2013

O48c Oliveira, Thiago Pontes de.
Cache para fragmentos de GUI utilizando metamodelos interpretados / Thiago Pontes de Oliveira. – Rio Tinto: [s.n.], 2013.
53f.: il. –
Orientador: Rodrigo Almeida Vilar Miranda.
Monografia (Graduação) – UFPB/CCAE.

1. Interfaces gráficas. 2. *Caching*. 3. AOM. 4. Web 2.0. I. Título.

UFPB/BS-CCAE CDU: 004.514 (043.2)

THIAGO PONTES DE OLIVEIRA

# CACHE PARA FRAGMENTOS DE GUI UTILIZANDO METAMODELOS INTERPRETADOS

Trabalho de Conclusão de Curso submetido ao Curso de Bacharelado em Sistemas de Informação da Universidade Federal da Paraíba, Campus IV, como parte dos requisitos necessários para obtenção do grau de BACHAREL EM SISTEMAS DE INFORMAÇÃO.

Assinatura do autor: 

**APROVADO POR:**

Orientador: Prof. Msc. Rodrigo Almeida Vilar Miranda
Universidade Federal da Paraíba – Campus IV

Prof. Msc. Rodrigo Rebouças de Almeida
Universidade Federal da Paraíba – Campus IV

Prof. Dr. Yuska Paola Costa Aguiar
Universidade Federal da Paraíba – Campus IV

RIO TINTO - PB
2013

“Some people, when confronted with a problem, think: "I know, I'll use caching." Now they have one problems” – Unknown author

# AGRADECIMENTOS

Queria agradecer ao acaso. Por ter marcado o curso errado na inscrição do vestibular. Por ir mal em Introdução à Programação e Matemática Elementar. Por pensar que eu estava no curso errado. Por me inscrever no vestibular de desenho industrial. Por não ter ido fazer a prova do vestibular de desenho industrial. Por ter ido morar em uma república cheia de gente louca. Queria agradecer à Rio Tinto pela hospitalidade. Pelas noites em claro regadas a café ou cerveja. Acho que se tudo não tivesse sido exatamente do jeito que foi, não teria sido tão maravilhoso. Queria agradecer à todos os professores que eu tive nesses anos de graduação. Mais do que professores, foram mentores para mim. Pelas aulas fora da sala de aula, pelo conteúdo fora das disciplinas, pela experiência de vida e pelas oportunidades oferecidas. Graças a eles eu não vou precisar trabalhar nenhum dia da minha vida. Queria agradecer aos meus pais, pelo incentivo e preocupação. Enfim, queria agradecer a todos que de alguma forma participaram desses últimos anos.

# RESUMO

O desenvolvimento de software sempre foi uma tarefa bastante complexa, desde a concepção até a validação pelo o cliente. Desde então tentamos melhorar a forma como software é desenvolvido, através da criação de frameworks, linguagens de programação, bibliotecas e metodologias. Neste trabalho analisaremos o AOM, um padrão arquitetural para sistemas reflexivos. Esse tipo de sistema possui uma natureza extremamente dinâmica, pois o reflexo das alterações nas funcionalidades é instantâneo, em tempo de execução. Devido essa natureza dinâmica, representar esses dados sem que a performance – um pilar na experiência do usuário – seja afetada acaba sendo um desafio. Este trabalho tem como objetivo propor uma solução de caching para interfaces gráficas Web 2.0 de sistemas baseados em AOM. Para tanto, analisamos estratégias de caching e como elas podem ser utilizadas em AOM. A metodologia utilizada foi baseada em uma revisão bibliográfica sobre AOM e Caching, seguido pela definição de casos de uso, projeto e implementação de um protótipo e validação da solução através de um benchmark. Por tratar-se de um cache de GUI, os resultados são visualmente notáveis. Os atrasos na renderização são explícitos no protótipo que não possui o cache. A solução mostrou-se estável apesar de alguns dos componentes serem mocks. Verificamos que a utilização de caches na camada de interface gráfica representa um grande ganho na experiência do usuário.

Palavras chave: Caching, AOM, Web 2.0.

**ABSTRACT**

Software Development was always a complex task, since its conception until its validation by the client. Ever since we try to improve the way software is developed, through frameworks, programming languages, libraries and methodologies. In this work we will analyze AOM, an architectural pattern for reflexive systems. This kind of system has an extremely dynamic nature, because changes are reflected instantaneously, in runtime. Due its dynamic nature, it is a challenge to represent data without affecting performance which is a pillar in the user experience. This work's goal is to propose a caching solution for Web 2.0 user interfaces based on AOM. Also we will analyze caching strategies and how they can be used with meta-architectures. The methodology used was based on a literature review about AOM and Caching, followed by the problem definition, a prototype's project and implementation and validation through a benchmark. Because it is a GUI cache, the results are visually noticable. They rendering delays are explicit in the prototype that has not cache. The solution showed stable despite the mocked components. We conclude that the use of caches in the view layer represents a huge gain in user experience.

Keywords: Caching, AOM, Web 2.0.

**LISTA DE FIGURAS**

# LISTA DE TABELAS

# LISTA DE SIGLAS

| | |
|---|---|
| API | Application Programming Interface |
| JSON | JavaScript Object Notation |
| XML | Extensible Markup Language |
| UX | User Experience |
| AOM | Adaptive Object-Model |
| LOM | Living Object Model |
| CASE | Computer-Aided Software Engineering |
| UML-VM | Unified Modeling Language-Virtual Machine |
| REST | Representational State Transfer |
| GUI | Graphic User Interface |
| DSL | Domain-Specific Language |
| OO | Orientado a Objetos |
| CRUD | Create Retrieve Update Delete |
| HTTP | Hypertext Transfer Protocol |
| URL | Uniform Resource Locator |
| MIME | Internet Media |
| HTML | HyperText Markup Language |
| AJAX | Asynchronous JavaScript and XML |
| TCP | Transmission Control Protocol |
| JSONP | JavaScript Object Notation with Padding |
| PaaS | Platform as a Service |
| CSS | Cascading Style Sheets |

| | |
|---|---|
| JS | JavaScript |
| DOM | Document Object Model |

# SUMÁRIO

# 1 INTRODUÇÃO

## 1.1 Motivação

O crescimento do mercado de TI demanda uma maior produtividade e um menor custo no desenvolvimento de sistemas, impulsionando a criação de novas técnicas e ferramentas de engenharia de software. Há quase 30 anos atrás, Brooks previu que a engenharia de software não iria conseguir acompanhar a evolução do hardware [30]. Ele dividiu a complexidade do software em dois tipos, a essencial que é a complexidade inerente à natureza do software e a acidental, que é a complexidade causada pelas ferramentas e técnicas de desenvolvimento. A maioria das abordagens que visam alguma melhoria para engenharia de software, ao invés de tentar reduzir a complexidade essencial, atacam a complexidade acidental, que já foi bastante explorada. Linguagens orientadas à objetos, componentes, padrões de projeto, frameworks, ferramentas CASE e metodologias ágeis são exemplos da evolução da ES, no entanto mesmo com essas técnicas a taxa de sucesso de projetos de desenvolvimento de software foi de apenas 32% em 2008 [30]. Esses dados mostram que Brooks estava certo e também mostra que o desenvolvimento de software está em crise [20].

Neste trabalho, estudaremos AOM, um padrão arquitetural definido por Yoder para sistemas reflexivos [11], que conseguem se adaptar em tempo de execução a novos requisitos. AOM e abordagens semelhantes, como UML-VM [30], conferem aos sistemas uma natureza extremamente dinâmica, pois o feedback das alterações nas funcionalidades é imediato. Segundo Brooks [33], a rápida prototipagem ataca a complexidade essencial do desenvolvimento de software, portanto supomos que o feedback imediato de AOM pode impactar significativamente a produtividade do desenvolvimento de software. Ferreira [20] realizou um experimento em um ambiente controlado utilizando o framework AOM *Oghma* e notou um ganho de produtividade, tanto do ponto de vista do desenvolvimento, como na adaptabilidade a novos requisitos. O resultado desse experimento mostrou a versatilidade que os sistemas que utilizam a arquitetura proposta por Yoder podem possuir e concorda com a nossa suposição.

Um dos desafios de AOM é a geração da interface gráfica a partir de meta dados. Welick propôs os Rendering Patterns [12] como padrões de projeto para GUI AOM. No entanto, ele deixa bem claro um ponto negativo das soluções propostas: o desempenho em sistemas que utilizam os mesmos componentes visuais com frequência, degradando a experiência do usuário porque eles precisam ser carregados e renderizados repetidamente. O autor sugere a utilização de caches para otimizar o desempenho da renderização dos componentes da interface. O foco deste trabalho é analisar algumas estratégias de cache que possam ser utilizadas para armazenar fragmentos de interface gráfica AOM.

Várias empresas já mostraram através de números o quanto a performance das aplicações web é crucial para o negócio como um todo, tendo um impacto direto em seus lucros. Empresas como Google, Yahoo! e Amazon disponibilizaram alguns números com relação ao impacto que alguns milissegundos causam em seus negócios [28]. A Amazon mostrou que 100ms a mais no carregamento da página causou uma queda de 1% nas vendas. O Google mostrou que 500ms a mais no carregamento causou uma diminuição de 20% nas

buscas. Nas análises do Yahoo!, 400ms a mais causou um aumento de 5 para 9% no número de pessoas que clicavam em "Voltar" antes mesmo da página carregar. Portanto, pequenos ganhos na performance da aplicação web resultam em rendimentos para a empresa.

Cache na interface é importante especialmente para o Living Object Model (LOM), uma plataforma baseada em AOM que aplica a adaptabilidade em todos os níveis da arquitetura. Com esse maior nível de adaptabilidade o desempenho acaba sendo penalizado. A utilização de cache seria uma grande melhora no desempenho da camada visual, também sendo um desafio dado o nível de adaptabilidade e dinamismo do sistema.

## 1.2 Objetivos

Este trabalho tem como objetivo propor uma solução de caching, para otimizar a apresentação de interfaces gráficas de sistemas de informação web, baseados em AOM.

## 1.3 Objetivos Específicos

Como objetivos específicos, iremos investigar algumas estratégias de caching e avaliar como elas podem ser utilizadas no contexto de AOM, visando melhorar o desempenho e a experiência do usuário nesses tipos de sistemas. Iremos identificar os cenários em que o cache pode ser utilizado durante o processo de renderização da interface AOM e também validaremos a solução proposta através da implementação de um protótipo e de um teste de performance, comparando uma solução sem o cache e outra com o cache.

## 1.4 Metodologia

A metodologia utilizada para o desenvolvimento deste foi constituida da seguinte forma:

- Revisão bibliográfica sobre AOM e Caching
- Definição de cenários para uso de cache na GUI de sistemas AOM
- Projeto e implementação de um protótipo com cache para GUI AOM
- Validação da solução através de benchmarks

## 1.5 Estrutura do trabalho

O trabalho está dividido 7 capítulos. O capítulo 1 introduz a motivação, os objetivos, a metodologia e a estrutura deste trabalho. Na Fundamentação Teórica, os conceitos básicos necessários para o entendimento deste trabalho são apresentados: As meta-arquiteturas AOM e LOM, o padrão de projeto Type Square e os padrões de renderização de interfaces AOM. No capítulo três, nosso problema é definido e algumas técnicas para resolvê-lo são apresentadas. O quarto capítulo apresenta a solução proposta, mostrando os casos de uso e a arquitetura da solução. O quinto capítulo mostra como a solução foi implementada,

analisando as tecnologias utilizadas. No capítulo de Avaliação, é feita uma pequena análise mostrando os ganhos obtidos utilizando a solução proposta no protótipo implementado. O capítulo final resume as contribuições deste trabalho e aponta para trabalhos futuros que podem dar continuidade a esta pesquisa.

# 2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta técnicas de Engenharia de Software baseadas em meta arquiteturas - Adaptive Object model (AOM) e Living Object Model (LOM) - e os padrões de projeto definidos para a implementação da interface gráfica nestes padrões.

## 2.1 Adaptive Object-Model – AOM

Segundo Yoder e Johnson [11], as arquiteturas AOM permitem que partes dos sistemas sejam adaptativas. Nesses trechos, os metadados são armazenados em bancos de dados ou arquivos XML e são manipulados como se fossem dados. Essas informações são interpretadas em tempo de execução, podendo ser alteradas e ter essas alterações refletidas no sistema imediatamente. Outro ponto forte dessa abordagem é que as regras de negócio podem ser feitas pelos clientes que possuem maior conhecimento sobre o domínio do problema. Esse tipo de arquitetura onde os requisitos são escritos usando alguma linguagem descritiva e interpretados por um sistema são chamados de "arquitetura reflexiva" ou "meta arquitetura". Sistemas AOM são baseados em instâncias e não em classes. Quando os metadados são alterados, conforme as mudanças nos requisitos, as alterações são refletidas imediatamente no comportamento do sistema.

A forma como as partes adaptativas dos sistemas AOM são projetadas é diferente da maioria dos projetos puramente orientados a objetos [11]. Projetos OO normalmente teriam classes para descrever regras de negócio associando métodos e atributos entre eles. Sendo assim, as classes modelam o negócio e uma mudança nas regras de negócio demanda uma mudança nas classes, ou seja, gera uma mudança direta no código. E, para que essa aplicação possa ser usada com essas alterações é necessário uma nova versão compilada e instalada do sistema. As partes adaptativas de um sistema AOM não utilizam classes para modelar suas regras de negócio, mas sim descrições (metadados) que são interpretados em tempo de execução. Sendo assim, quando uma alteração no domínio é necessária, apenas essas descrições são alteradas e são imediatamente refletidas na aplicação em execução.

De acordo com Welick [12], o núcleo da arquitetura de AOM é representado em dois níveis, ilustrados na Figura 1:

**Nível de Conhecimento:** Define as regras gerais do comportamento e a estrutura das entidades do domínio. É onde as regras de negócio são descritas pelo expert no domínio.

**Nível Operacional:** Contém as instâncias do domínio. Nesse caso as entidades, propriedades e relacionamentos, que foram definidas no nível de conhecimento, são interpretados e recebem valores.


Figura 1 - Núcleo AOM

Embora esses dois níveis sejam responsáveis pela estrutura e pelo comportamento das entidades adaptativas, um terceiro nível para apresentação desses dados é necessário. Um dos principais problemas durante o desenvolvimento de sistemas AOM é como representar os metadados para o usuário [15], e para tentar resolvê-lo, algumas soluções foram propostas. Welick [22] propõe a inclusão de um terceiro nível, o Nível de Visualização, composto por componentes de renderização que possuem diferentes níveis de granularidade, podendo ser combinados dinâmicamente para gerar interfaces para sistemas AOM.

O que torna a camada de apresentação tão importante é o fato de que a visualização dos dados é um requisito comum em sistemas AOM, e essa apresentação é um dos primeiros desafios encontrados durante o desenvolvimento desses sistemas [12], principalmente pela natureza dinâmica da arquitetura AOM. Simplificando, a camada de Visualização teria instruções para representar os elementos do modelo e essas instruções podem ser compostas em tempo de execução, a partir dos dados descritivos. Assim as interfaces atingiriam o mesmo nível de flexibilidade e adaptabilidade exigidos pela arquitetura AOM.

### 2.1.1 Padrão Type Square

AOM utiliza os padrões TypeObject e Property para compor o que Yoder chama de Type Square [11,15]. O padrão de projeto Type Square (Figura 2) é utilizado para representar os componentes adaptativos de um sistema AOM de forma genérica, possibilitando a interpretação do modelo conceitual em tempo de execução. Este padrão torna possível a criação de diversos tipos sem a necessidade de implementar novas classes ou extender uma hierarquia de tipos, como é feito quando se cria diversas subclasses em sistemas OO.

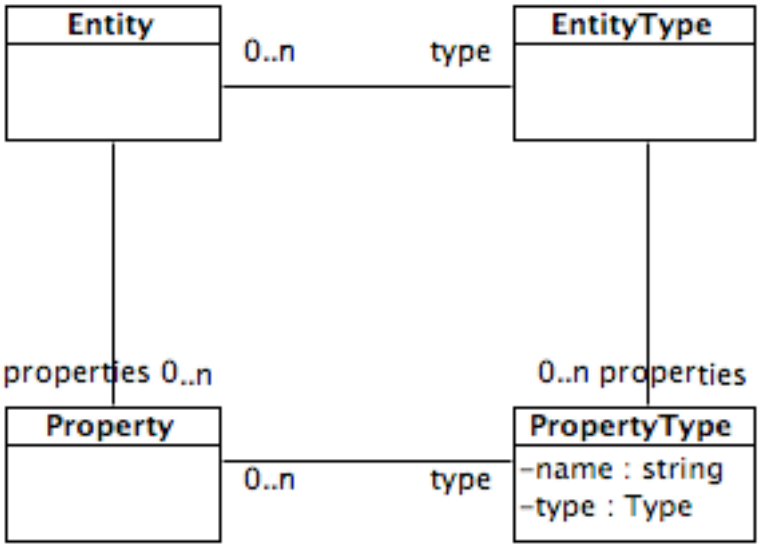


Figura 2 - Type Square

Outros padrões também são utilizados em sistemas que utilizam meta arquitetura mais complexas [11, 15]. Por exemplo, quando os valores dos atributos de uma entidade precisam de validações mais complexas, padrões como o Strategy são usados para definir e abstrair diferentes estratégias de validação, através de um único método público *validate*. Quando essas estratégias se tornam mais complexas, elas acabam evoluindo para RuleObjects, que com a ajuda do padrão Composite, podem compor diferentes conjuntos de regras para um determinado tipo de entidade.

### 2.1.2 Linguagem de padrões de renderização

Um dos problemas encontrados durante o desenvolvimento de sistemas AOM é como apresentar os metadados para o usuário [15]. Welick [12, 22] propõe três padrões para representar visualmente objetos AOM: **Property Renderer**, responsável por renderizar fragmentos de GUI (widgets) específicos de uma propriedade de uma determinada entidade; **Entity View**, responsável por renderizar widgets mais complexos, referentes a toda uma entidade em conjunto com suas propriedades ; e **Dynamic View**, responsável pelos widgets que exibem um conjunto de entidades. Apesar de poderem ser usados individualmente, todos esses padrões geralmente se apresentam em conjunto. A seguir, encontra-se o detalhamento de cada um desses padrões.

#### 2.1.2.1 Property Renderer

O Property Renderer(Figura 3) é o responsável por renderizar a interface de um certo tipo de propriedade em um dado contexto. O renderer depende: (1) do **tipo** da propriedade que está sendo renderizada, por exemplo, *String*PropertyRenderer e *Date*PropertyRenderer; e (2) do **contexto** em que a propriedade será utilizada, como String*View*PropertyRenderer e String*Edit*PropertyRenderer. O contexto serve para diferenciar se uma propriedade está sendo

renderizada para exibição ou edição, por exemplo. O padrão possui uma implementação default para renderizar minimamente todos os tipos de propriedade, gerando uma interface simples para cada contexto. O código gerado pela implementação padrão é bastante simples, o mínimo necessário para representar uma propriedade. E essa implementação padrão consegue desenhar qualquer propriedade em qualquer contexto, sendo bastante útil durante as primeiras fases de prototipagem.

Cada Property Renderer é responsável por representar algum property type visualmente, e pode ser especializado em um contexto específico, e combinados, podem representar interfaces mais complexas.

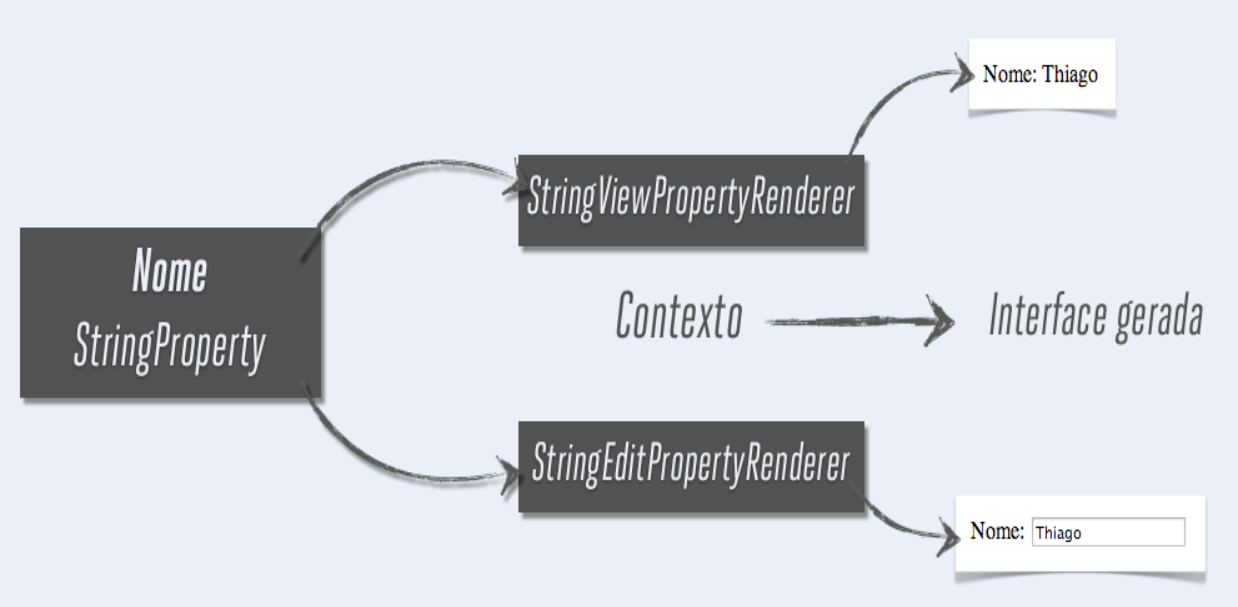


Figura 3 - Property Renderer

### 2.1.2.2 Entity View

O Entity View(Figura 4) coordena a apresentação de vários Property Renderers de uma entidade para produzir widgets mais complexos. Cada Property Renderer é especializado em produzir um componente de interface para instâncias de um property type em um dado contexto. O contexto simboliza o propósito da interface, por exemplo, nós podemos renderizar o Entity View de forma a gerar um formulário, com o propósito de editar as informações de um modelo, neste caso o contexto seria de **edição**. A sequência e a composição de renderers podem ser especificadas em nível de código ou utilizando metadados. O Entity View deve estar ciente do contexto onde os componentes serão renderizados, e o contexto pode conter informações adicionais que podem ser utilizadas durante o processo de renderização.

Com esse padrão, validações podem ser criadas e adicionadas ao Entity View, e ser utilizadas durante o processo de renderização de uma entidade. Essas validações devem ser relacionadas a própria interface, validações relacionadas à regra de negócios devem ser delegadas como restrições do domínio, como Roles (vide TypeSquare). Apesar desse padrão ter sido criado para gerar fragmentos de interface para uma entidade, ele também pode ser

usado para a geração de uma página inteira, caso a informação mostrada nessa página seja mais específica.

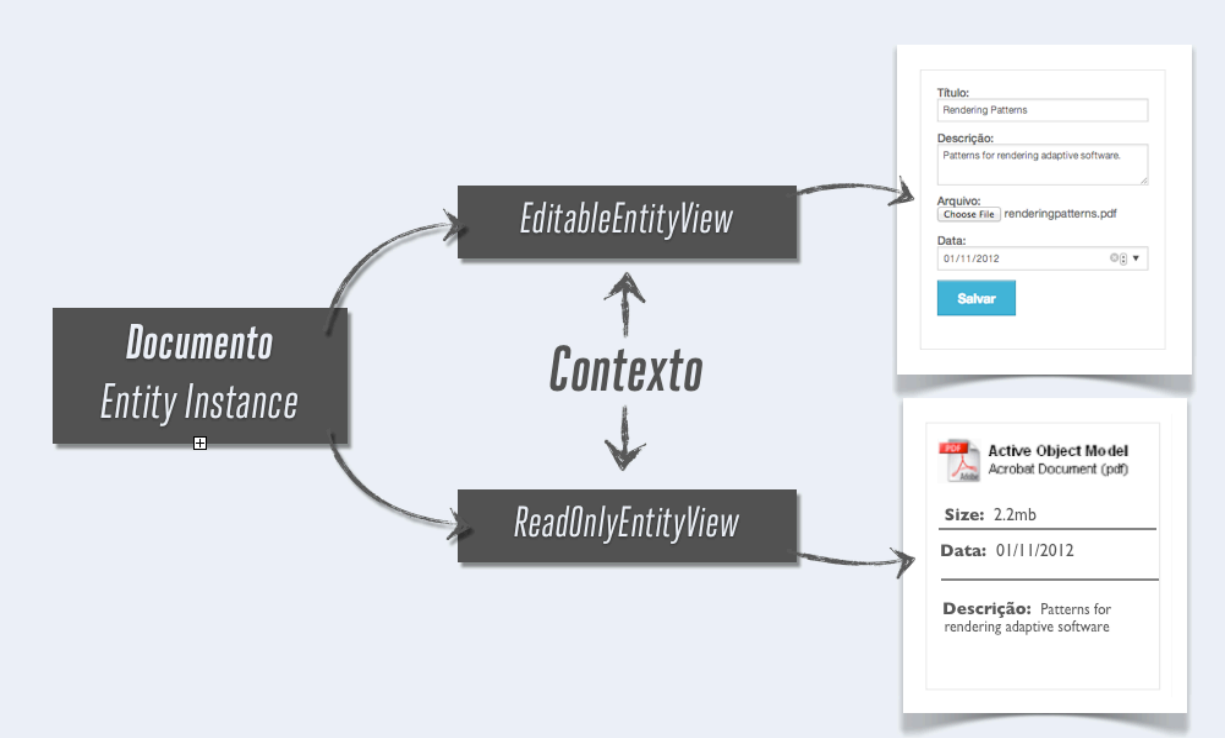


Figura 4 - Entity View

### 2.1.2.3 Dynamic View

O padrão Dynamic View(Figura 5) pode gerar o código da interface de um conjunto de entidades sem acoplamento ou sem precisar referenciar nada da interface no modelo, também permitindo que o desenvolvedor possa compôr diferentes views para a mesma entidade. Esse modelo pode possuir várias views aplicadas, possibilitando que elas sejam selecionadas dependendo do contexto desejado.

Esse componente é especializado em gerar código de interface para um conjunto de entidades. Ele recebe como entrada um conjunto de entidades e retorna o código de interface de acordo com o propósito da view, podendo apresentar views para o mesmo modelo de diferentes formas. O Dynamic View produz código para composição de layouts mais complexos, e esses layouts possibilitam configuração dinâmica da interface exibida, podendo criar o código da interface completamente do início ou utilizando os Property Renderers e Entity Views.

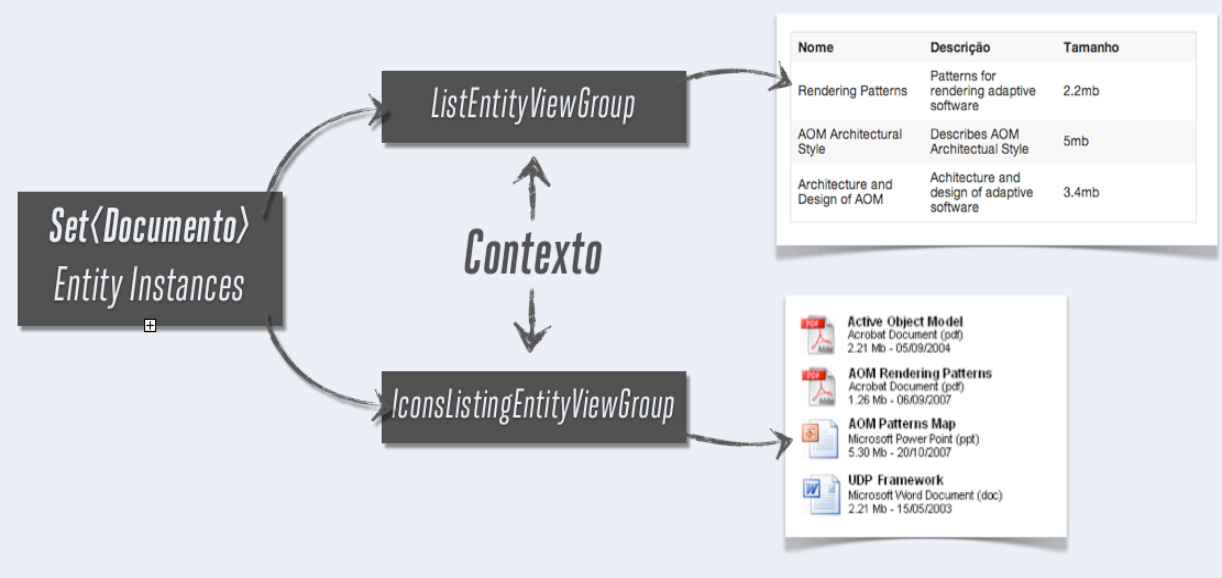


Figura 5 - Dynamic View

### 2.1.3 Combinando os padrões de renderização

Como é mostrado na Figura 3, o Property Renderer se relaciona com todos os outros padrões. O Entity View o usa para renderizar os atributos da entidade, e o Dynamic View que gera as interfaces dos conjuntos de entidades pode agir diretamente em ambos. Os padrões apresentados por Welick [12, 22] podem ser usados em outras arquiteturas, mas são focados em arquiteturas baseadas em AOM e discutem problemas de apresentação que surgem durante o desenvolvimento desses sistemas. A Figura 7 mostra o nível de granularidade desses padrões, que podem ser interpretados como os níveis de abstração que esses padrões atigem. Sendo o Property Renderer usado para renderizar a menor parte de uma interface, que seria o equivalente a uma única propriedade, o Entity View representado uma única entidade, que por sua vez possui uma série de propriedades, e o Dynamic View, que é o nível mais alta da abstração, representando um conjunto de entidades.

Welick deixa bastante claro que a solução apresentada por ele possui uma performance baixíssima e faz mal uso dos recursos disponíveis, o que diminui bastante a experiência do usuário, principalmente para aplicações web [22]. Como sugestão ele propõe a utilização de caches para atacar esses problemas, mas esse uso deve ser cauteloso para que o nível de dinamismo da aplicação não seja prejudicado. Ele propõe o uso do padrão Caching [24] para auxiliar na performance da renderização dos componentes, principalmente dos Property Renderers que são utilizados com maior frequência.

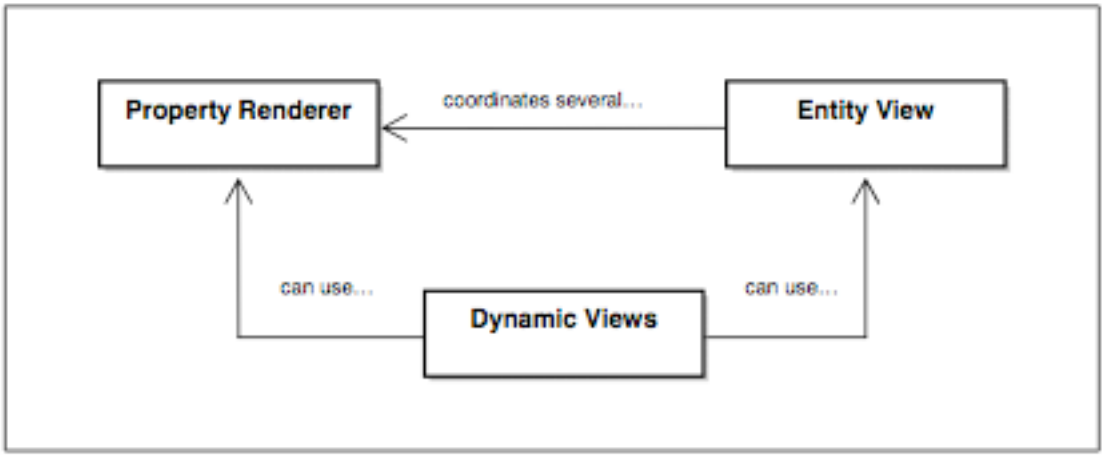


Figura 6 - Relações entre os padrões de renderização

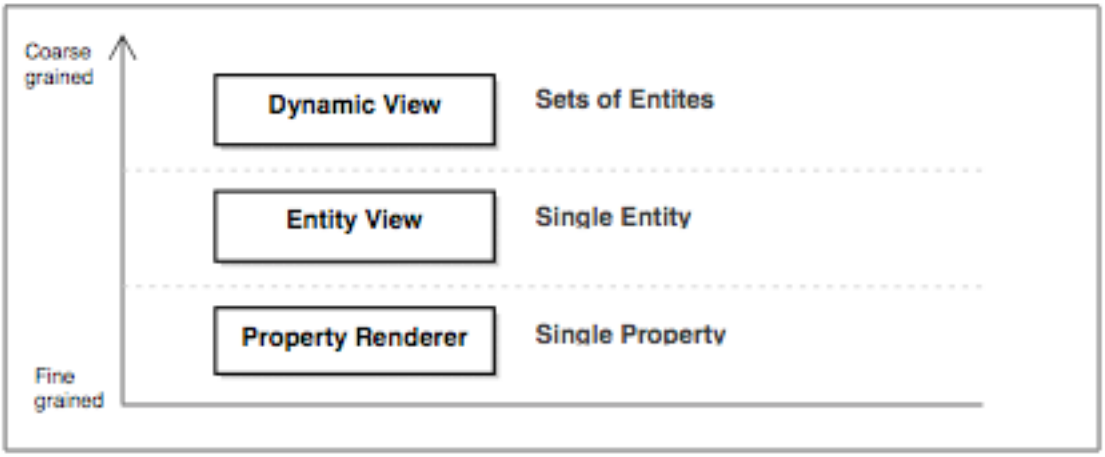


Figura 7 - Nível de granularidade dos padrões de renderização

## 2.2 Living Object Model - LOM

A arquitetura do LOM (Figura 5) é fortemente influenciada pelo AOM, mas possui suas particularidades. Primeiramente, LOM é específico para sistemas informação. Segundo, se propõe a prover adaptabilidade completa em todas as suas 3 camadas: Persistência, Lógica e GUI. Além disso, LOM não pode ser considerado um padrão arquitetural, pois na verdade representa uma especificação concreta que pode ser implementada em diversas linguagens e tecnologias. A camada de persistência do LOM é implementada por um banco de dados 100% adaptável, os componentes necessários para isso se encontram no nó Database e no Business Server. A camada lógica, que também deve ser 100% adaptável, é implementada pelos componentes da variante do Type Square utilizada pelo LOM (Figura 9), na qual é possível descrever as entidades, propriedades e suas relações, adicionando o padrão Accountability no TypeSquare. Não vamos nos aprofundar mais nessas duas camadas da arquitetura pois o foco

do trabalho é a camada de GUI, que assim como as outras também deve ser totalmente adaptável em tempo de execução.

A camada de GUI utiliza os metadados contidos no Business Server para renderizar a interface gráfica da aplicação, da mesma forma sugerida por Welick em seu artigo sobre padrões de renderização [12]. O GUI Server é o responsável por coordenar as classes de renderização (chamadas de **widgets** nesse trabalho), que foram implementadas utilizando os padrões Property Renderer, EntityView e Dynamic View, assim como na arquitetura AOM. Além disso, ele também envia os widgets necessários para o usuário final e para o desenvolvedor.

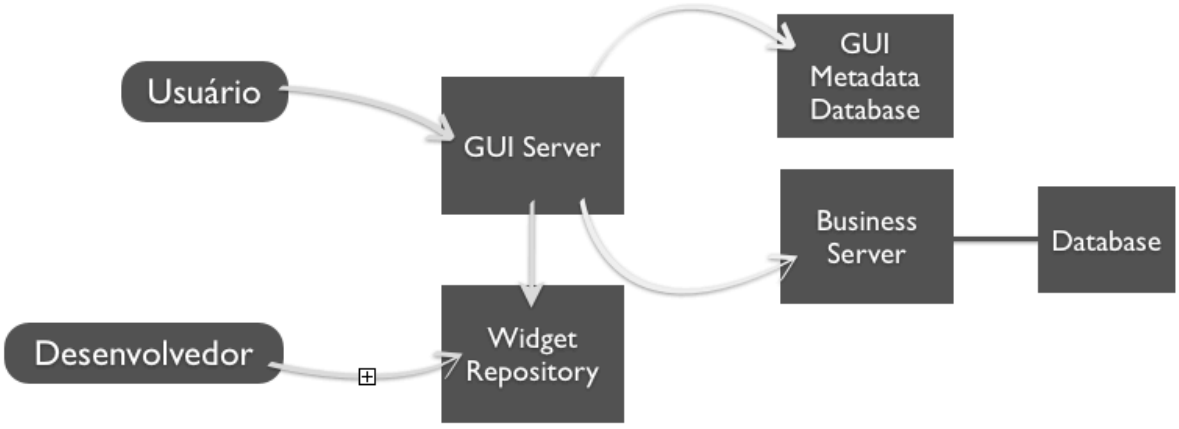


Figura 8 - LOM: Arquitetura geral

Quando os widgets chegam ao seu destino final, eles são executados. Geralmente a execução de um widget envolve a requisição de dados que serão utilizados na renderização da interface. Esses dados são requisitados no GUI Server, que por sua vez os requisita no Business Server e repassa a resposta de volta para o widget.

Sempre que um widget é atualizado pelo desenvolvedor, o GUI Server envia o código atualizado para os clientes, desse modo, qualquer parte da interface pode ser alterada em tempo de execução. Um outro nó da arquitetura que está diretamente relacionado com o GUI Server é o GUI Metadata database, ele é utilizado para armazenar informações dinâmicas da GUI, como por exemplo, se um campo é editável, visível, etc. Essas configurações podem mudar de acordo com o usuário final, como em situações onde o usuário não pode utilizar certos campos, etc. Também é através do GUI Server que os usuários acessam um sistema LOM, ele controla os widgets e o acesso aos dados do Business Server, existindo dois tipos de usuários, o usuário final e o desenvolvedor. O desenvolvedor pode modificar os tipos de relacionamento, as entidades, as propriedades e os widgets que serão renderizados em um dado contexto. Já o usuário final só pode manipular os dados operacionais.

O Widget Repository é um repositório central de widgets para sistemas LOM. Ele serve como plataforma de publicação e distribuição de widgets. Desenvolvedores podem criar widgets e publicá-los no Widget Repository para que outros desenvolvedores possam utilizar. O GUI Server não precisa armazenar ou instalar nada para utilizar os widgets, bastando citar o

identificador do widget que o GUI Server baixará o respectivo widget do repositório. Sendo assim, os widgets são baixados sempre que solicitados pela aplicação. O problema é que o mesmo widget pode ser utilizado em diversas partes da GUI. E do modo como está implementado, o cliente terá que baixá-lo diversas vezes, o que aumetará o tráfego e o consumo de banda. Esse problema afeta principalmente os PropertyRenderers, por serem bastante utilizados pelos outros padrões. Esse problema também pode ser resolvido utilizando caches.

Dado que o LOM visa gerar todo o código adaptável em tempo de execução e produzir todas as funcionalidades do sistema apenas interpretando os metadados, o padrão TypeSquare é utilizado para criar todas as entidades e em nenhum ponto do código envolve a escrita de código OO. Devido a essa mudança de semântica, LOM renomeou algumas classes do esquema Type Square (ver Tabela 1). Além disso, como o Type Square só lida com uma hierarquia de entidades por vez, é preciso relacionar as diversas entidades do sistema. Para tanto, LOM usa o padrão Accountability [31]. Uma vez que LOM também usa os padrões propostos por Welick [12] para gerar toda a GUI automaticamente, os problemas de performance na GUI são potencializados nessa plataforma. Desse modo, o presente trabalho de pesquisa se mostra ainda mais importante quando se considera o contexto do desenvolvimento do LOM.

| **Terminologia em AOM** | **Terminologia em LOM** |
|---|---|
| EntityType | Class |
| Entity | Instance |
| PropertyType | Attribute |
| Property | AttributeValue |

Tabela 1 - Alterações feitas pelo LOM no padrão Type Square

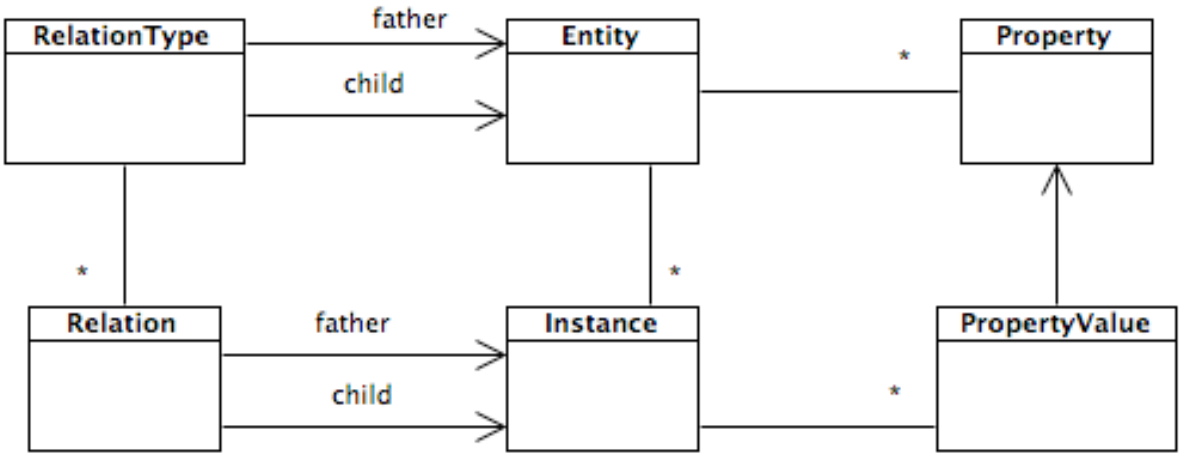


Figura 9 - Variação do Type Square utilizada pelo LOM

A arquitetura LOM é focada para o desenvolvimento de sistemas de informação, que podem ser implementados segundo uma API orientada a recursos (como REST [32]). Dessa forma, o sistema seria constituído por recursos, que são semelhantes às classes de LOM, e existiriam apenas quatro métodos para todos os recursos (entidades): POST, PUT, READ e DELETE. Essa abstração simplifica bastante a implementação das camadas de Lógica e GUI do LOM. Por exemplo, pode-se criar oito ganchos para inserir scripts, a serem executados antes ou após cada um dos quatro métodos dos recursos.

O projeto LOM foi criado para avaliar o ganho de produtividade desenvolvimento de software com o uso de uma plataforma 100% adaptável. LOM é uma meta-arquitetura, semelhante a AOM, mas no LOM todo o código do sistema é adaptável. Então, mesmo propondo coisas diferentes de AOM, LOM se aproveita de muitas idéias do seu antecessor.

# 3 APLICANDO CACHING NA GUI DE SISTEMAS LOM

Neste capítulo, definiremos o problema de pesquisa que está sendo abordado neste trabalho e faremos uma análise das estratégias de caching que podem ser utilizadas na solução do problema.

## 3.1 Definição do Problema

A arquitetura proposta pelo LOM está sendo utilizada pela primeira vez no desenvolvimento de uma aplicação web 2.0, com o código fonte disponível em: https://github.com/rodrigovilar/lom. Sendo o LOM uma arquitetura reflexiva, os problemas encontrados durante o desenvolvimento foram os mesmos mostrados por Welick [22]. São problemas relacionados à performance, principalmente na parte da renderização dos widgets. Para tentar solucionar esse problema, utilizamos a solução proposta por Welick em seus artigos [12, 22], a utilização do padrão Caching [24], inicialmente para caching dos fragmentos de interface, adaptado para o browser.

Como os widgets podem ser utilizados em mais de uma classe ou atributo, existe uma necessidade de criação de uma camada de cache entre os componentes visuais e o servidor de GUI, visando melhorar a performance da renderização dos widgets. Como todo o trabalho de renderização é feito no lado do cliente, precisariamos que essa camada de cache fosse implementada no browser, e atualmente, podemos fazer isso utilizando a API do HTML5 de LocalStorage [18], e para implementar as funcionalidades do cache seria necessário pesquisar estratégias de caching, visto que a invalidação das entradas do cache é um requisito fundamental para o funcionamento do sistema, que evolui de forma dinâmica e em tempo de execução.

## 3.2 Estratégias de Caching

A finalidade de um cache é diminuir o tempo de resposta após a requisição de um recurso. Um exemplo simples seria uma aplicação que sempre que requisitada, lê o conteúdo de um arquivo e exibe esse conteúdo para o usuário. Operações de leitura e escrita são computacionalmente custosas e para simplificar essa operação, o sistema poderia ler o arquivo apenas uma vez e manter o conteúdo em memória, evitando a necessidade de ler o arquivo em todas as requisições. Nesse cenário, manter o conteúdo em memória para agilizar a resposta pode ser considerado caching. Continuando a analogia, um problema aparece quando começamos a ter operações de escrita nesse arquivo. Torna-se difícil manter uma consistência entre o conteúdo que está na memória e o conteúdo que está no arquivo. Esse é o grande desafio das estratégias de caching, invalidar o que está em memória.

Implementar invalidação de cache manualmente é uma tarefa bastante complicada e propensa a erros. Por isso foi pesquisado diversas estratégias de invalidação, começando pela utilizada pela JPA conhecida como Generational Caching [5]. Essa abordagem mantém um valor “generation” que age como um identificador para a versão de cada tipo de objeto. Cada

vez que o objeto é atualizado esse valor é incrementado, então, sempre que lermos ou escrevermos um valor no cache, nós adicionamos a versão junto com a chave. Como usamos o valor do generation como parte da chave, sempre que um objeto for atualizado, a próxima requisição não irá encontrar o objeto no cache (*cache miss*), pois a chave para aquele valor agora será outra, invalidando assim aquela entrada. Sendo assim, sempre que alguma alteração ocorrer, todas as entradas que possuírem aquele valor de generation como chave, serão expiradas implicitamente. Isso acontece pelo fato de que as chaves antigas não são excluídas do cache, elas apenas não serão utilizadas novamente. As vantagens de se utilizar essa abordagem é que ela garante uma melhor consistência entre os valores guardados no cache e no banco de dados, diminuindo também a carga de acessos direto ao banco. Um problema é que como os dados nunca são excluídos do cache após a invalidação, a memória usada para cache tende a encher rapidamente, caso a aplicação tenha muitas escritas. Uma solução para esse problema é implementar uma política de LRU (Least Recently Used) no cache, assim os valores com uma chave que possui um valor de generation mais antigo serão evitadas e removidas do cache com o passar do tempo, garantindo espaço para os novos valores.

Uma segunda estrátegia analisada foi a utilizada pelo framework Rails, que é uma variação do Generational Caching, chamada Key-based Cache Expiration [4], a abordagem é praticamente a mesma apresentada anteriormente, a diferença é que o valor de "generation" não é incrementado, mas sim substituído por um timestamp da atualização/criação do objeto usado como chave no cache.

Outra estratégia mais simples é chamada Write-through Caching [6], nela todas as operações de CRUD são ligadas ao cache, ou seja, quando algum objeto é adicionado no banco ele é também adicionado no cache, quando é removido do banco, também é removido do cache. Essa abordagem funciona bem com entidades simples, mas sendo mais indicado o método de Generational Caching para entidades mais complexas que possuem relacionamentos com outras entidades. Uma estratégia de cache deve atacar os seguintes problemas [22]:

**Desempenho**: O custo de repetidas aquisições, inicialização e liberação de recursos deve ser minimizado.
**Disponibilidade**: A solução deve permitir que os recursos sejam acessados mesmo quando o serviço de origem dos recursos não esteja disponível.
**Escalabilidade**: A solução deve ser escalável independente da quantidade de recursos.
Fora as estratégias de invalidação de cache, o padrão Caching [24], descreve como evitar a reaquisição de recursos evitando que eles sejam liberados após o uso. Esse recurso é mantido em uma área de armazenamento onde o acesso seja mais rápido. O problema de repetidas requisições, inicialização e liberação dos mesmos recursos causam uma sobrecarga desnecessária nos serviços. Tendo o tempo de acesso a esses recursos diminuido, significa uma melhora na performance do sistema.

As arquiteturas reflexivas como o AOM e o LOM encontram problemas de performance com relação a forma como os dados são apresentados para o usuário. Para amenizar esses problemas, Welick [22] propõe a utilização do padrão Caching [24], que visa armazenar

temporariamente os recursos em uma memória de acesso rápido chamada cache, depois disso, quando o recurso for utilizado novamente, o cache deve ser usado para recuperá-lo, ao invés de requisitá-lo novamente na origem dos dados.

As vantagens de se utilizar técnicas de caching vão além do ganho de performance, soluções que utilizam os benefícios do uso de caching ganham no quesito escalabilidade, pois evitando a requisição direta dos recursos diminui a sobrecarga na origem dos dados, problema comum em técnicas como Polling. Outro benefício é o aumento da disponibilidade dos dados, por exemplo, caso a origem de dados de um recurso se torne indisponível e esse recurso esteja presente no cache, o usuário ainda conseguirá acessá-lo, sem notar que a origem de dados está insdisponível no momento.

As desvantagens aparecem com relação a complexidade adicionada para manter os dados sincronizados. Outro problema comum é o gasto de memória mantendo recursos no cache que não são mais utilizados. Caches devem ser utilizados cuidadosamente, em casos como quando a requisição de recursos não pode ser mais melhorada, pois a implementação de caching adiciona uma complexidade maior no sistema, complicando o desenvolvimento e a manutenção da solução, além de aumentar o uso de memória, sendo aconselhado que esses problemas sejam analisados antes da adoção de estratégias de caching.

### 3.2.1 HTTP Caching

O protocolo HTTP possui cabeçalhos especiais para tornar possível a utilização de caches no browser [8], e sua utilização diminui o tempo de carregamento de uma página e a carga no servidor web. O protoloco HTTP oferece duas maneiras de definir por quanto tempo um recurso é válido, com o cabeçalho *Expires* e com o *Cache-control: max-age.* Na primeira abordagem, o programador define um prazo de validade para o recurso através de uma data limite, e a partir dela o browser irá requisitar novamente esse recurso. Na segunda maneira, o recurso recebe uma "idade" máxima, ou seja, a partir da data atual, o recurso poderá ficar no cache por um período X. Como Lamm comenta no artigo [8], é aconselhável que esse valor seja cerca de 1 ano. Os browsers modernos são implementados de uma forma que, após o cache expirar é enviado um GET condicional com o intuito de saber se o recurso foi alterado e, caso não tenha sido, a validade do cache é renovada. Para impor ao browser conteúdo atualizado, é utilizado uma técnica semelhante à Generational Caching, passando um timestamp na URL do recurso. Por causa do timestamp, a URL vai parecer diferente da que o browser possui, fazendo-o baixar o "novo" recurso.

### 3.2.2 Cache no browser para widgets

Para que a implementação do cache no browser seja funcional é crucial que exista consistência entre os widgets no cache do browser e no GUI Server. Aplicações web modernas utilizam AJAX para requisitar por novos dados para a aplicação sem que haja a necessidade de que a página seja recarregada [7], deixando a equipe de desenvolvimento responsável pela escolha de qual técnica utilizar. As técnicas mais conhecidas são chamadas Polling e Long Polling [7] (também chamada de COMET ou "*Hanging GET*"). A ideia do

polling é bem simples: a aplicação envia GETs para o servidor requisitando novos dados, caso haja algo novo, o servidor envia uma resposta com os novos dados; caso não tenha nada novo, é enviada uma resposta vazia. Apesar de simples e fácil de implementar essa técnica adiciona uma grande sobrecarga desnecessária ao servidor, pois muitas das respostas virão vazias. O Long Polling(COMET) é uma variação do Polling. Nele, quando o servidor recebe uma requisição e não possui dados para enviar, a requisição é mantida até que haja novos dados para serem enviados. O problema dessa técnica é que, apesar de o servidor não ficar sobrecarregado tendo que responder vários GETs sequenciais, é feito algum *workaround* para conseguir "segurar" o GET no servidor, já que essa não é uma opção nativa do protocolo HTTP. Normalmente isto é feito enviando várias tags de script vazias para um *iframe* no cliente, essa técnica pode ser uma péssima experiência para o usuário, pois ela faz com que a notificação de "carregando" do navegador nunca termine. Todas essas alternativas adicionam uma sobrecarga no protocolo HTTP, pelo fato de ele não ter sido projetado para esse tipo de situação, tornando impossível a utilização dessas técnicas para o desenvolvimento de aplicações que necessitem uma baixa latência e que carreguem informações em tempo real [1].

Uma outra possibilidade é manter uma conexão aberta permanentemente fazendo uma requisição XMLHttpRequest com o MIME Type definido como multipart/x-mixed-replace [19], essa técnica é bastante antiga, e foi apresentada pelo Netscape em 1995, mas foi caindo em desuso por ser suportada de formas diferentes pelos browser e atualmente é suportada apenas pelo Firefox.

### 3.2.3 Server-Sent Events x Websockets

Entre as novas especificações do HTML5, estão a API de WebSockets [2] e a de Server-Sent Events [7]. Ambas vieram para tentar resolver os problemas existentes para comunicação inversa, do servidor com o browser.

A API de WebSockets permite a possibilidade de estabelecer uma conexão persistente full-duplex entre o browser e um servidor remoto [1]. O protocolo Websocket não possui nenhuma restrição de mesma origem como acontece com requisições AJAX, apesar de permitir que isso seja utilizado, através de cabeçalhos de segurança. Por padrão é possível realizar requisições cross-domain sem grandes problemas, mas foram encontrados problemas com relação a utilização do protocolo com proxies [1], pelo fato de o protocolo iniciar a sua conexão enviando uma requisição HTTP sugerindo um "Upgrade", e isso faz com que o proxy recuse a requisição. Essa tecnologia já está sendo implementada na grande maioria dos browsers modernos, com bibliotecas (como o socket.io) que utilizam *fallbacks* para browsers que ainda não suportam o protocolo, tornando possível a utilização dessa tecnologia comercialmente. Além do WebSockets, o socket.io implementa as técnicas legadas de polling, e dependendo da versão do browser, a biblioteca seleciona qual fallback utilizar.

A API de Server-Sent Events disponibiliza uma forma mais simples de comunicação, foi criada para quando apenas o servidor precisa enviar dados para o browser, ou seja, com ela é possível desenvolver aplicações que recebem informações vindas do servidor sem que a página seja atualizada, de uma forma mais elegante, sem a utilização de gambiarras. Uma

vantagem do SSE é que os eventos são enviados via HTTP, não precisando de uma implementação especial no lado do servidor. Os eventos enviados pelo servidor podem possuir IDs, nomes e timeout de reconexão customizáveis, além da possibilidade de enviar diversos tipos de dados, como texto plano, JSON, XML, etc.

Inicialmente o Server-Sent Events parecia ser a melhor opção para a nossa solução, pois era necessário apenas o envio de informações esporádicas do servidor para o browser, mas após algum avanço na implementação do protótipo tivemos a necessidade de enviar dados do browser para o servidor, o que essa tecnologia não suporta. Isso fez com que escolhêssemos o protocolo de WebSockets no lugar de Server-Sent Events.

### 3.2.4 Websockets e Socket.io

O objetivo principal da tecnologia de WebSockets é permitir que aplicações que utilizam o browser como plataforma, consigam uma comunicação bidirecional com servidores que não dependam de multiplas conexões HTTP e sem utilizar AJAX com as técnicas Polling, possibilitando o desenvolvimento de jogos e atualizações em tempo real com baixa latência. O WebSocket [3] é um protocolo independente baseado no TCP e a única relação que ele possui com o protocolo HTTP é que o seu handshake é interpretado por servidores HTTP como uma requisição de Upgrade, e proporciona uma comunicação bidirecional entre um browser e um servidor remoto que esteja esperando conexões com aquela finalidade. O protocolo consiste de handshake inicial entre o browser e o servidor, esse handshake enviado pelo browser é visto pelo servidor como uma requisição GET normal, com uma oferta de upgrade, por isso, utiliza por padrão a porta 80 para conexões normais e a 443 para conexões sob TLS.

Essa abordagem é adotada para que uma única porta possa ser usada tanto para responder as requisições utilizando o protocolo HTTP como as utilizando o protocolo WebSockets, sendo essa a configuração indicada para o deployment de aplicações simples. Para arquiteturas mais elaboradas, com load balancers e vários servidores, é aconselhado que separe os servidores que irão servir HTTP dos que utilizarão WebSockets, para simplificar o gerenciamento.

Após o handshake é iniciado a troca de mensagens em frames em cima do protocolo TCP. Essa troca de dados é independente e em qualquer direção, em unidades conceituais referenciados na especificação como “mensagens” [3]. Uma mensagem é uma unidade completa em nível de aplicação que espera que as aplicações que implementam o protocolo, neste caso o browser, possua APIs para tratar o envio e o recebimento dessas mensagens. Os dados são enviados em forma de *frames* que possuem um tipo, existem tipos definidos para dados textuais, que são interpretados como texto UTF-8, dados binários, que possuem a interpretação definida pela aplicação e frames de controle que são utilizados pela aplicação e são de uso exclusivo do protocolo para sinalizar eventos, como quando uma conexão deve ser finalizada, por exemplo. Ainda existem mais dez tipos de frames que estão reservados para uso futuro.

O Socket.IO é uma pequena biblioteca que fornece uma API para simplificar e abstrair o uso de tecnologias que permitam a troca de mensagens entre o browser e o servidor. A

diferença entre o protocolo puro WebSockets e o Socket.IO é que a biblioteca permite que o programador consiga implementar comunicação em tempo real em praticamente todos os browsers do mercado, incluindo as versões para smartphone [19], escolhendo outra técnica que seja mais viável em browsers que ainda não possuem suporte a WebSockets. Entre as técnicas implementadas que o Socket.IO utiliza como fallback quando o protocole de WebSockets não é suportado estão [28]:

- Adobe Flash Socket
- ActiveX HTMLFile(IE)
- XHR com multipart encoding
- XHR com long-poling
- JSONP poling (poling em domínios cruzados)

A biblioteca também fornece várias melhorias, como parsing automático de JSON, fácil manipulação de eventos, possibilidade de transmitir mensagens em broadcast, multicast ou unicast de forma bastante simples, além ser suportada em praticamente todos os browsers. Enviar informações do servidor para o browser pode ser complicado, mas por possuir tantos fallbacks implementados e disponíveis, o Socket.IO garante essa funcionalidade na maioria dos browsers do mercado. MacCaw [28] listou os browser suportados pelo Socket.IO:

- Safari >= 4
- Chrome >= 5
- IE >= 6
- iOS Safari
- Firefox >= 3
- Opera >= 10.61

### 3.2.5 LocalStorage

Uma desvantagem das aplicações web modernas com relação à aplicações desktop foi a falta de armazenamento local. Uma aplicação nativa pode utilizar recursos geralmente disponibilizados pelo sistema operacional para armazenar e recuperar dados como preferências do usuário ou último estado da aplicação [17]. Essas informações podem ser mantidas das mais diversas formas, seja no registro do sistema operacional, em arquivos INI, XML, txt ou caso o volume de dados seja maior, ou mais complexo pode ser usado um banco de dados ou uma outra variedade de soluções disponíveis. Já as aplicações web nunca tiveram nenhum desses recursos disponíveis localmente. Em uma tentativa de suprir essa necessidade, cookies estavam sendo utilizados com essa finalidade, mas essa abordagem possui muitas desvantagens [17]. A primeira é que ao utilizar cookies para salvar informações o desenvolvedor está tentando fazer um protocolo *stateless* como o HTTP manter estado, o que pode gerar alguns problemas. Outro problema é que os cookies são enviados em todas as

requisições, portanto, além de adicionar um overhead à todas as requisições, todos os dados podem ser interceptados caso a comunicação não seja realizada através de uma conexão segura. Os cookies também possuem um tamanho limitado de 4KB, não servindo também para manter um volume maior de informações.

Outras técnicas foram desenvolvidas para tentar suprir essa necessidade, mas foi apenas em 2009 que a especificação para uma API de armazenamento local no browser começou a ser escrita para o HTML5 [18]. Originalmente chamada de Web Storage, acabou ficando conhecida como Local Storage, por ter sido esse o nome dado pelos primeiros fabricantes de browser a implementar essa API. Essa especificação disponibiliza uma maneira simples para aplicações web armazenar dados como chave-valor localmente no browser do cliente. Como com os cookies, esses dados persistem mesmo quando o usuário sai da página, fecha o browser ou desliga o computador. Mas diferente dos cookies, esses dados nunca são transmitidos para o servidor web a cada requisição. Diferente das outras tentativas que eram que forneciam um banco de dados baseado no SQLite para aplicações web, como o Google Gears [23], o Local Storage é implementado nativamente nos browsers, estando disponível sem a necessidade de utilizar algum plugin de terceiros, sendo suportada por praticamente todos os browsers modernos [17].

A especificação da API do Local Storage sugere um espaço de 5MB por domínio, e apenas strings podem ser armazenadas, lançando uma exceção chamada "QUOTA_EXCEEDED_ERR" quando o limite de armazenamento é atingido. Após atingir esse limite, a aplicação pode requisitar permissão do usuário para obter mais espaço, apesar de alguns browsers permitirem que esse espaço seja configurável, ele não pode ser alterado programaticamente sem a liberação do usuário. Apesar da limitação de armazenar apenas conjuntos de caracteres, é possível armazenar os dados como strings e depois convertê-los de volta para os seus tipos originais quando for necessário.

# 4 SOLUÇÃO PROPOSTA

Este capítulo descreve a solução proposta. Para melhor explicar, a solução é apresentada através de casos de uso, diagramas de sequência e storyboards. Também é apresentado a arquitetura e o funcionamento do protótipo implementado.

## 4.1 Casos de Uso

Os casos de uso possuem dois atores: o desenvolvedor e o usuário. O papel do desenvolvedor é o de definir os widgets da aplicação. Em alguns cenários ele também pode ser responsável por desenvolver widgets, mas esse tipo de cenário não será explorado neste trabalho. O usuário é que utiliza a aplicação. A figura 10 ilustra os casos de uso.


Figura 10 - Casos de Uso

## 4.2 Arquitetura da Solução

Essa seção irá descrever a solução proposta em maiores detalhes através de diagramas de sequência e storyboards que ilustram os vários cenários de uso de uma aplicação web 2.0 desenvolvida utilizando a arquitetura proposta pelo LOM. O foco será na utilização e no comportamento do cache de widgets.

A solução proposta sugere o desenvolvimento de uma abstração sobre o LocalStorage do browser para ser utilizado como cache para os widgets, esses widgets são armazenados utilizando o nome deles como chave, e um objeto literal como valor. Esse objeto literal possui o código do widget, sua versão e seu nome. Quando um widget sofre alguma alteração no repositório de widgets, o usuário recebe essa nova versão, o cache é atualizado e a nova versão do widget será utilizada na próxima atualização da página. Caso o usuário não esteja online quando o widget for atualizado, ele receberá a nova versão quando abrir a aplicação

novamente, pois sempre que a aplicação é iniciada, os widgets do cache são atualizados para sua última versão disponível no repositório.

As figuras 11 e 12 mostram o funcionamento e a arquitetura da solução, respectivamente. Os compotenentes com nome vermelho foram mockados para que o caching fosse o foco do desenvolvimento. O Client Browser é o navegador do usuário e possui dois componentes principais, o Browser Cache, responsável pelo cache local de widgets e o Controller, que age como interface para comunicação com os componentes externos. O GUI Server é quem se comunica diretamente com o Client Browser. O GUI Server possui vários componentes, como o Server Cache, um Controller, um DAO, o Widget Pusher e um REST Client. O Server Cache é um cache utilizado pelo GUI Server para manter os widgets requisitados do Widget Repository. O Controller é a interface exposta para comunicação com o Client Browser. O DAO é um componente utilizado para realizar operações relacionadas ao GUI Metadata Database. O Widget Pusher é o componente resposável por enviar os widgets diretamente para o client quando uma atualização acontece. O último componente do GUI Server é o REST Client, responsável pela comunicação do GUI Server com o Widget Repository e o Business Server. O Widget Repository é um repositório de widgets e o Business Server é onde as regras de domínio ficam armazenadas. O GUI Metadata Database é parecido com o Business Server, mas guarda informações relevantes para a interface, como validações, etc.

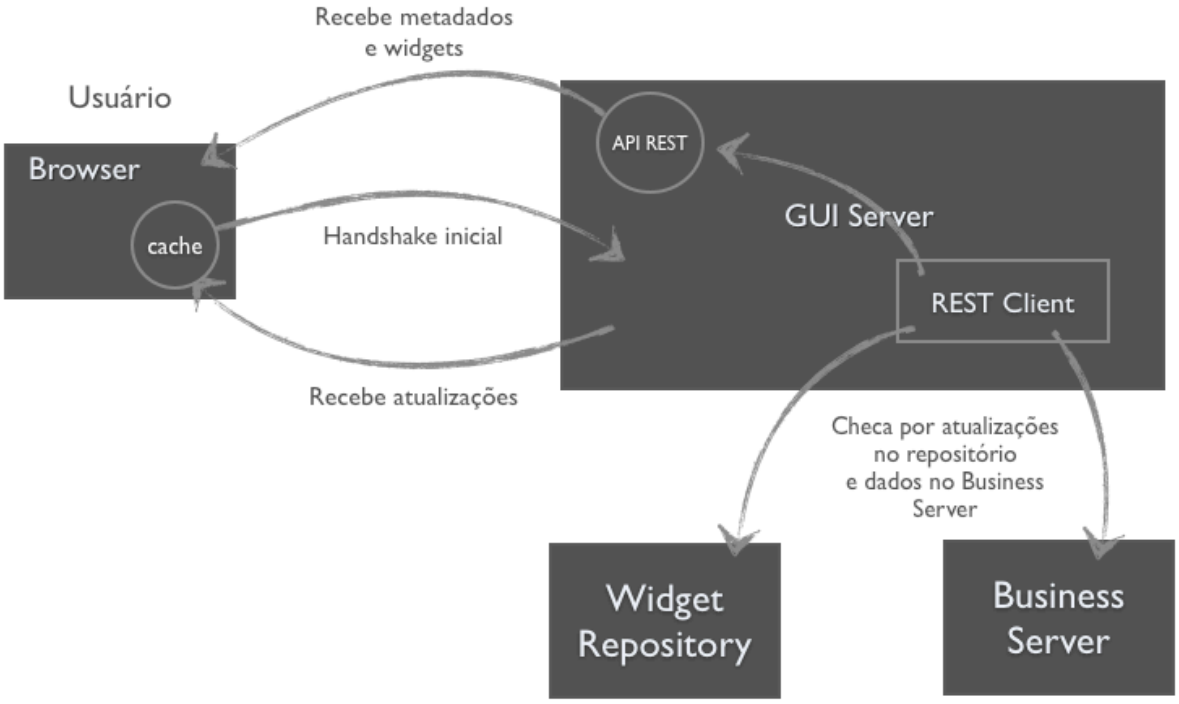


Figura 11 - Funcionamento

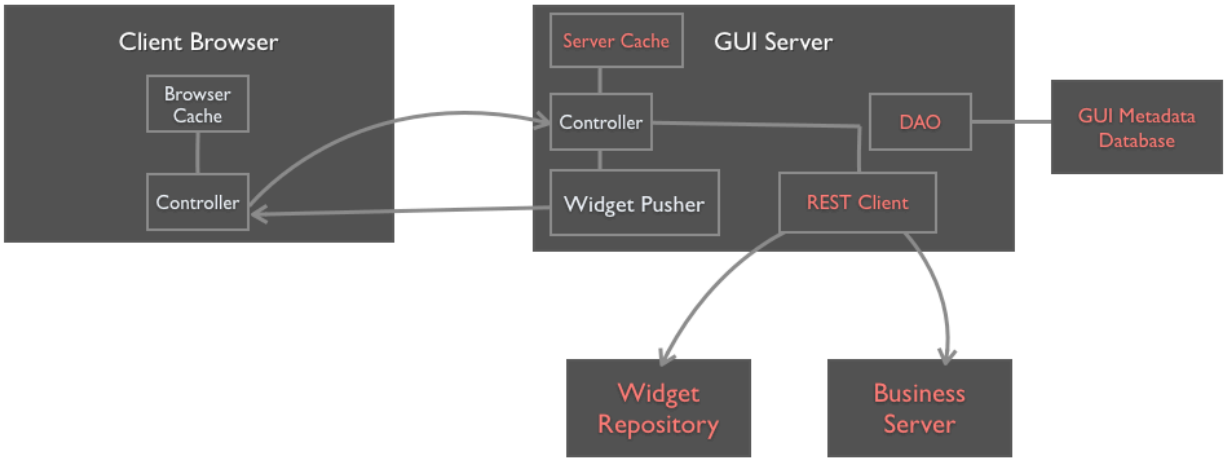


Figura 12 - Arquitetura da implementação web do LOM

## 4.3 Componentes do Sistema

**Developer:** O developer é o ator responsável pelo desenvolvimento dos widgets e da composição das interfaces (Figura 13 e Figura 14). Os widgets que ele desenvolve podem ficar armazenados no Widget Repository, ficando a disposição de outros desenvolvedores, caso estejam interessados. Cada widget desenvolvido por ele possui um id e uma versão. A versão é incrementada sempre que o componente é atualizado pelo desenvolvedor. Um desenvolvedor pode desenvolver um widget, lançá-lo e depois de algum tempo atualizá-lo, isso criará uma nova versão do widget, e as aplicações que estiverem utilizando esse widget serão notificadas de que existe uma nova versão.

**End User:** O End User é o ator que simboliza o usuário final que está utilizando o sistema LOM, é quem acessa a aplicação pelo browser, tendo mais contato com a aplicação em execução. O End User participa da dos cenários 4, 5 e 6, em diferentes situações de uso.

**Browser:** O browser representa a plataforma onde o código cliente do sistema é executado. Através do qual os dados são exibidos, modificados, excluídos, etc. Sendo utilizado diretamente pelo End User, iniciando sua atividade quando o usuário digita a URL da aplicação.

**Browser's Cache:** É o cache dos widgets implementado no browser. É onde o browser checa se já possui aquele widget, antes de requisitá-lo no servidor. Age como uma camada em cima do LocalStorage do browser, provendo uma API que facilita o armazenamento e a recuperação de widgets, evitando requisições desnecessárias.

**GUI Server:** Fornece os widgets da aplicação, sendo também quem envia as atualizações dos widgets quando uma nova versão é lançada. Quando utilizado pelo Developer, serve para

definir os widgets que serão utilizados pela aplicação. Quando um Developer lança uma nova versão do widget no Widget Repository, o GUI Server é o responsável por transmitir essa nova versão do widget para as aplicações que estão utilizando-o.

**Server Cache:** É um cache local de widgets do GUI Server. As informações de um widget tem origem no Widget Repository, mas para evitar que sempre que um widget seja utilizado uma nova requisição seja feita para o Widget Repository, o GUI Server mantém um cache com os widgets utilizados pela aplicação. Quando o Browser requisita por um widget, o GUI Server checa primeiro se ele já possui aquele widget no seu cache, caso ele não possua, ele requisita no Widget Repository e armazena no seu cache. Assim, na próxima vez que esse widget for requisitado ele não precisará pegá-lo no repositório mais uma vez.

**Widget Repository:** É um repositório central de widgets, onde os GUI Servers das aplicações procuram pelos widgets que a ela irá utilizar. Quando um widget é criado, é nele que ele fica armazenado, é também nele que o widget é atualizado.

**Business Server:** É onde os metadados da lógica de negócio ficam armazenados, ou seja, é onde a aplicação busca informações sobre as entidades, etc. Normalmente é uma interface REST para um banco de dados.

## 4.4 Projeto detalhado

Essa seção descreve os cenários de uso da aplicação, ilustrando os fluxos de execução através de diagramas de sequência e storyboards.

### 4.4.1 Cenário 1 – Definindo Root Widget não presente no GUI Server

O primeiro cenário mostra o desenvolvedor definindo o root widget, que é o widget responsável por desenhar toda a interface inicial. Para isso ele passa o id e a versão do widget desejado para o GUI Server, este procura pelo widget em seu cache, e caso o widget não seja encontrado, ele o carrega do repositório de widgets e guarda no seu cache para uso futuro.

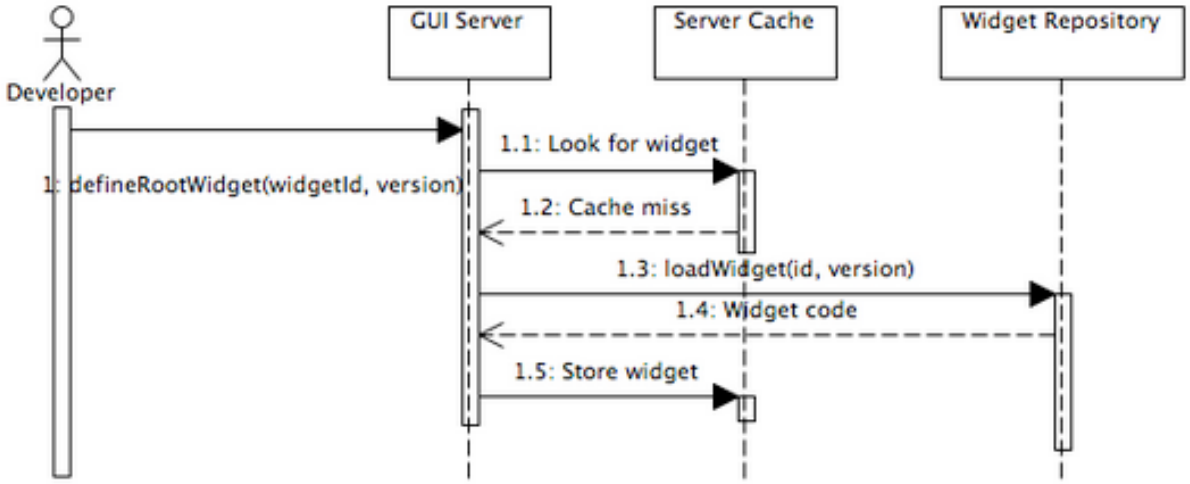


Figura 13 - Cenário 1, definindo Root Widget utilizando widgets que não está no GUI Server

### 4.4.2 Cenário 2 - Definindo Entity Widgets não presentes no GUI Server

O cenário 2 é bastante semelhante ao primeiro cenário, a diferença é que nele o desenvolvedor define um widget para representar uma entidade, para isso, é passado o id do widget, a versão, o id da entidade que irá utilizá-lo e um hook, que simboliza o contexto em que o widget será renderizado. Após isso o processo é o mesmo, o GUI Server procura o widget no seu cache, e como o cache não foi encontrado, ele requisita o código do widget no repositório de widgets, e após recebe-lo, ele é guardado em seu cache.

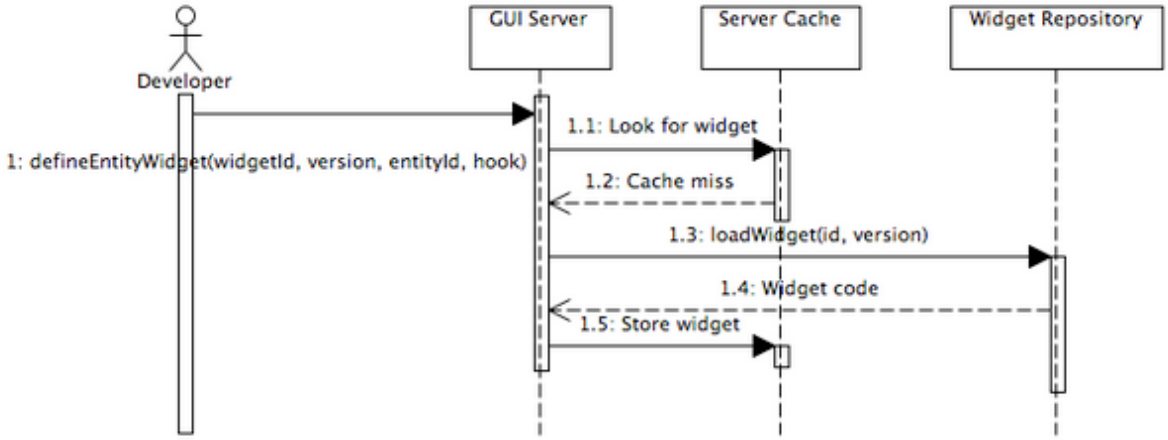


Figura 14 - Cenário 2, definindo entity widgets utilizando widgets que não estão no GUI Server

### 4.4.3 Cenário 3 - Usuário abrindo a aplicação pela primeira vez

O cenário 3(Figura 15 e Figura 16) descreve o processo de quando a aplicação é aberta pelo usuário pela primeira vez. Nela, o usuário digita a URL da aplicação no browser, o HTML/CSS/JS iniciais são carregados, e a aplicação é iniciada no browser. Durante o

processo de inicialização, a aplicação procura pelo root widget no cache do browser, como a aplicação está sendo utilizada pela primeira vez, o widget não é encontrado (gerando um cache miss), e a requisição pelo root widget é feita no GUI Server. O GUI Server também busca pelo widget requisitado no seu cache, como no cenário 1, o widget ainda não foi carregado do repositório, então o GUI Server faz a requisição do widget solicitado no repositório, o guarda no seu cache e retorna o código do widget para a aplicação. Ao receber o código do widget, a aplicação o guarda no cache do browser, junto com o seu nome e a sua versão. Após todo esse processo, o widget é inicializado, e durante a sua inicialização é que os dados das classes que ele irá representar são requisitados, essa requisição é feita para o GUI Server, que realiza algumas validações nos dados, e requisita as entidades no Business Server, que é quem possui as informações das classes. Essas informações são enviadas de volta para o widget que as utiliza para desenhar a interface desejada.

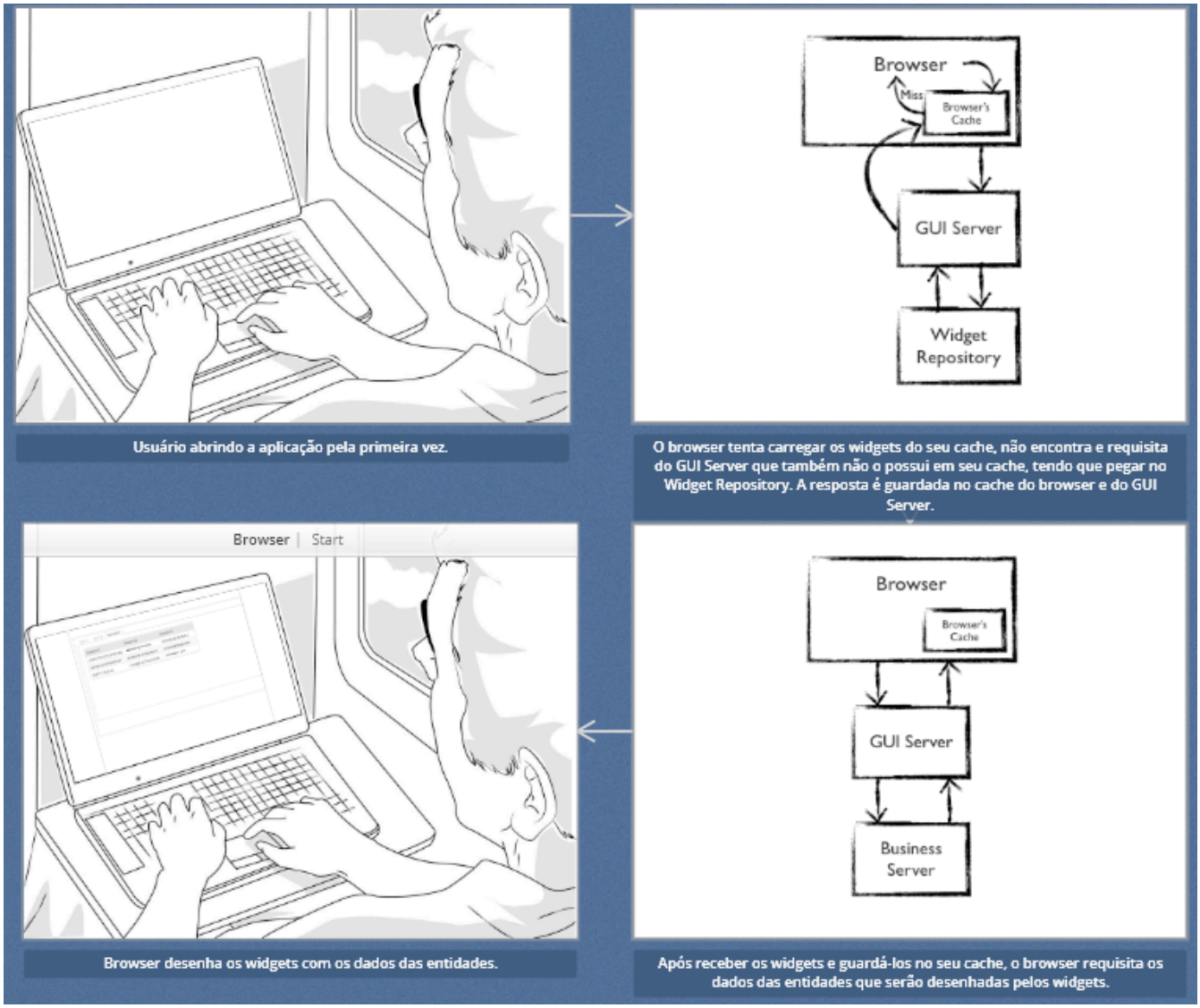


Figura 15 - Cenário 3, usuário abrindo a aplicação pela primeira vez

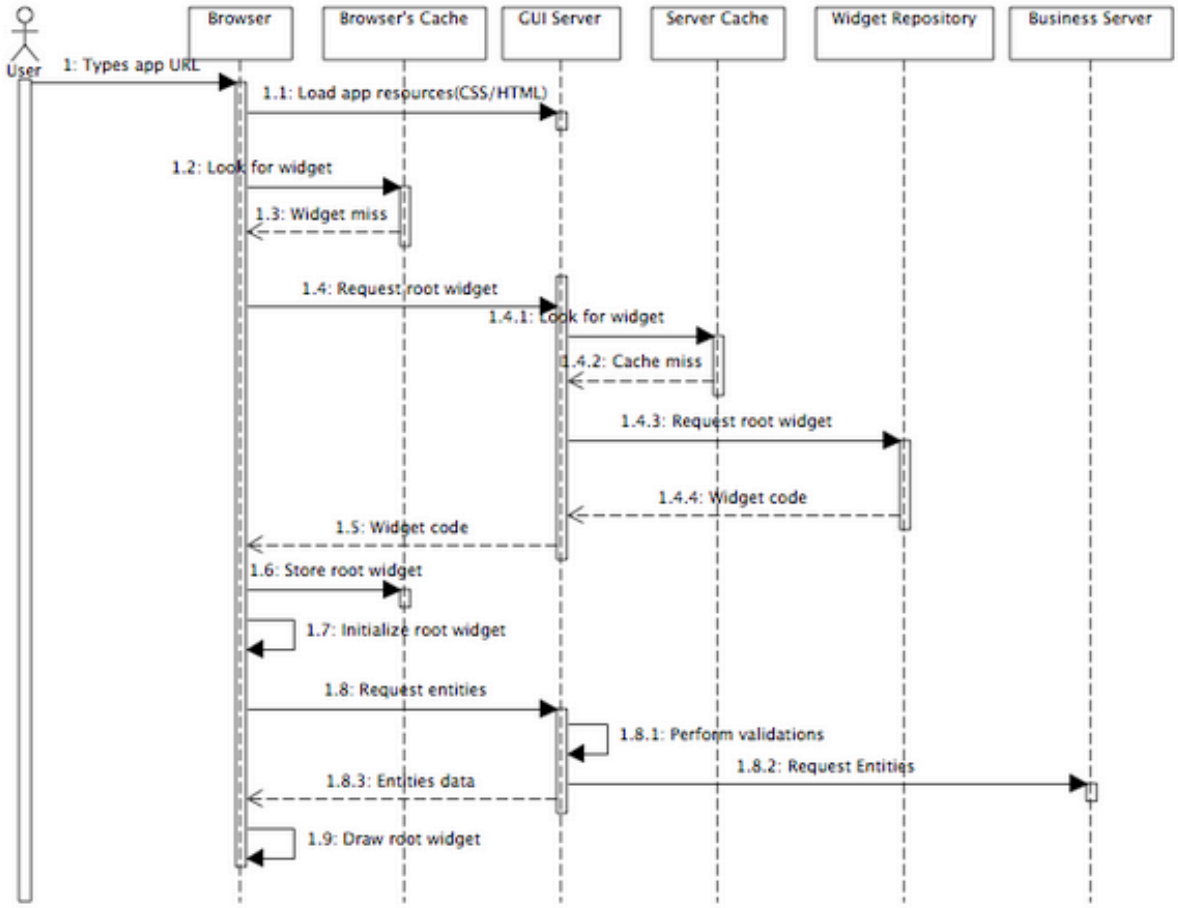


Figura 16 - Diagrama de sequência do Cenário 3, usuário abrindo a aplicação pela primeira vez

### 4.4.4 Cenário 4 – Outro usuário abrindo a aplicação

O quarto cenário(Figura 17) ilustra o processo de um outro usuário não simultâneo abrindo a aplicação. Neste caso, o cache do browser do usuário ainda não possui nenhum widget, mas o GUI Server já possui o root widget carregado no seu cache, enviando-o para o browser do usuário sem precisar requisitar o widget no repositório. Ao receber o código do widget, o processo é o mesmo do cenário 3, o widget é salvo no cache do browser, é inicializado, requisita os dados das entidades que ele irá representar, o GUI Server repassa esses dados vindos do Business Server, e os dados são utilizados pelo widget para desenhar a interface.

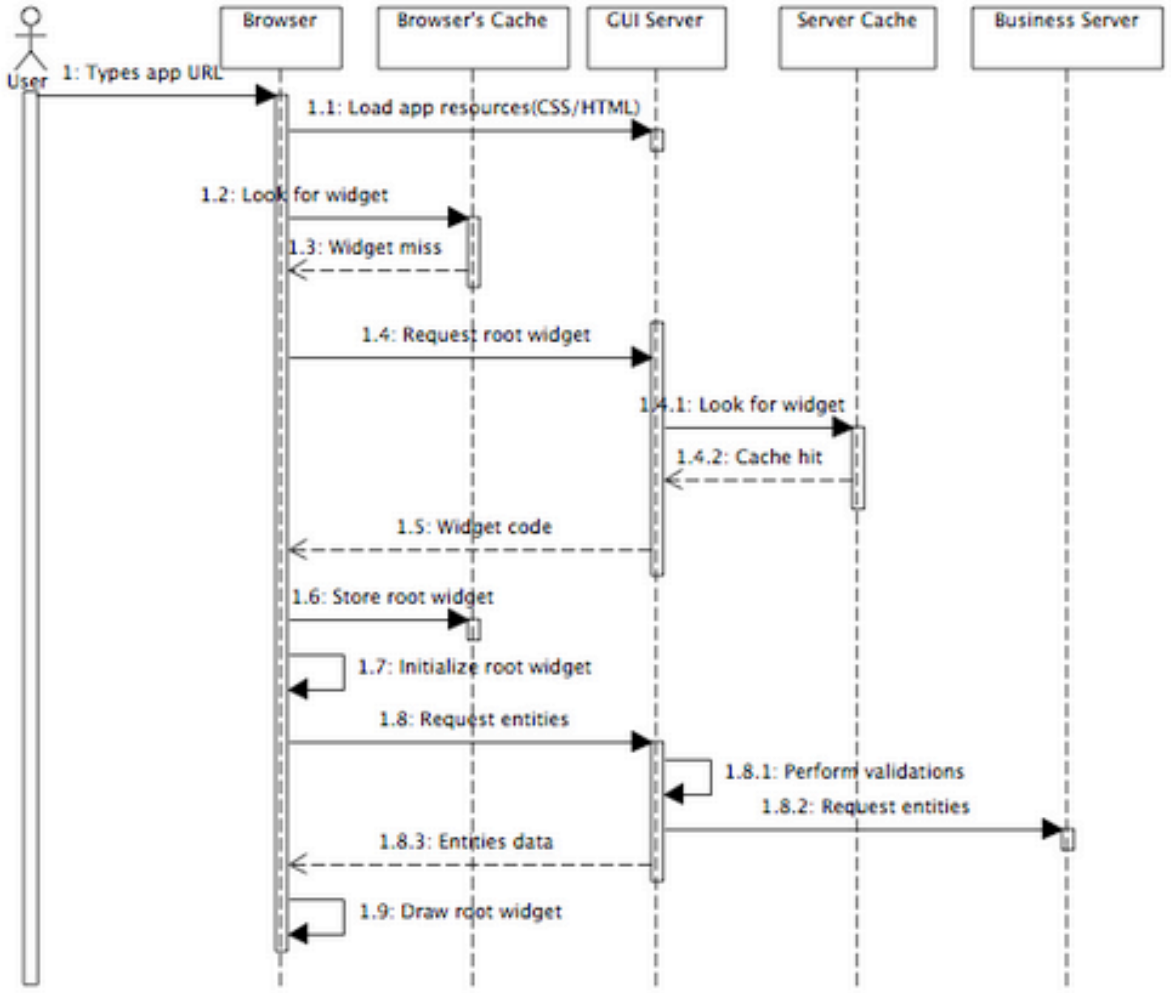


Figura 17 - Diagrama de sequência do cenário 4, outro usuário abrindo a aplicação

### 4.4.5 Cenário 5 – Usuário reabrindo a aplicação

Neste cenário, ilustrado na Figura 18, o usuário já utilizou a aplicação uma primeira vez, após abrir o browser e o HTML/CSS/JS iniciais serem carregados, a aplicação irá buscar pelo root widget no cache do browser, o widget será encontrado pois ele foi armazenado no cache do browser após ter sido enviado pelo GUI Server em outra execução. O root widget é então recuperado do cache, inicializado e o processo segue como nos outros cenários, requisitando no GUI Server os dados das entidades que serão desenhadas e utilizando root widget com os dados recebidos do GUI Server para renderizar a interface.

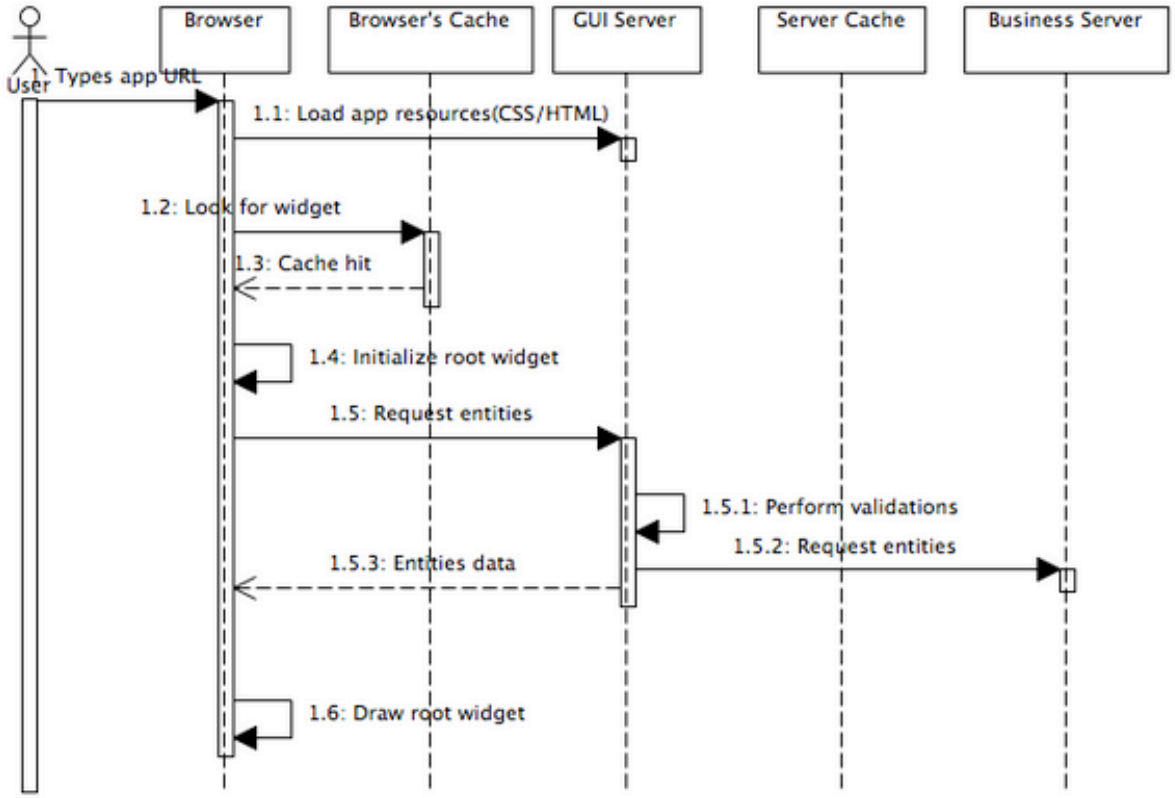


Figura 18 - Diagrama de sequência do cenário 5, usuário reabrindo a aplicação

### 4.4.6 Cenário 6 – Widget é atualizado com usuário online

O sexto cenário, com o diagrama de sequência ilustrado nas figuras 19 e 20, mostra o que ocorre quando um widget é atualizado quando o usuário está com a aplicação aberta. Após o usuário abrir a aplicação e carregar o HTML/CSS/JS inicial do GUI Server, recuperar o root widget do cache do browser e inicializá-lo, requisitando os dados das entidades que serão desenhadas e renderizar a interface, o desenvolvedor repete a ação do cenário 1, fazendo modificações no root widget que a aplicação do usuário está utilizando. Essa modificação gera uma nova versão do widget, e após o GUI Server receber essa nova versão do widget, ele envia a todos os usuários que utilizam aquele widget a sua nova versão, ao receber o novo widget, a sua entrada no cache do browser do usuário é atualizada, e já é utilizada na próxima vez que o usuário tentar usar aquele widget.

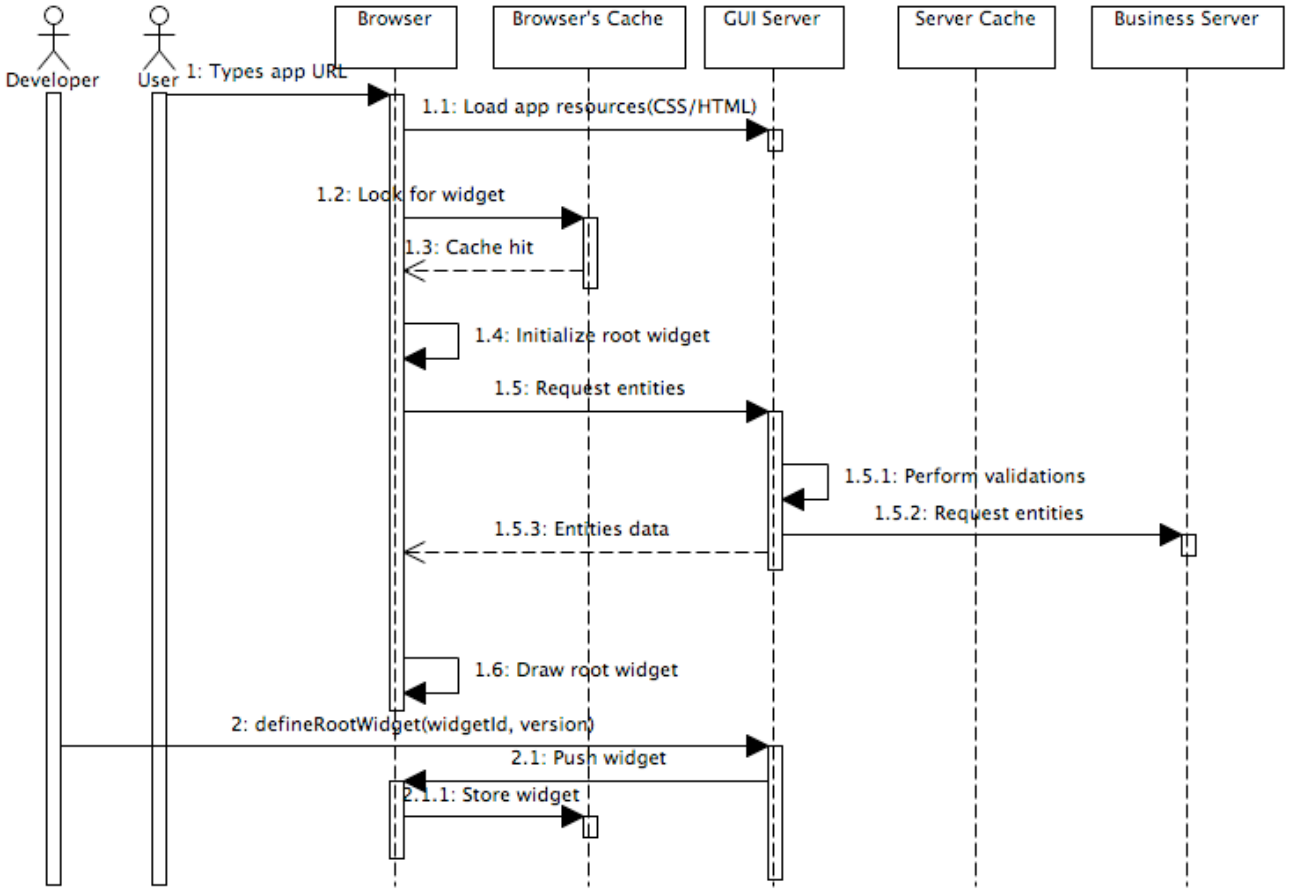


Figura 19 - Diagrama de sequência do cenário 6, onde o widget é atualizado com o usuário online

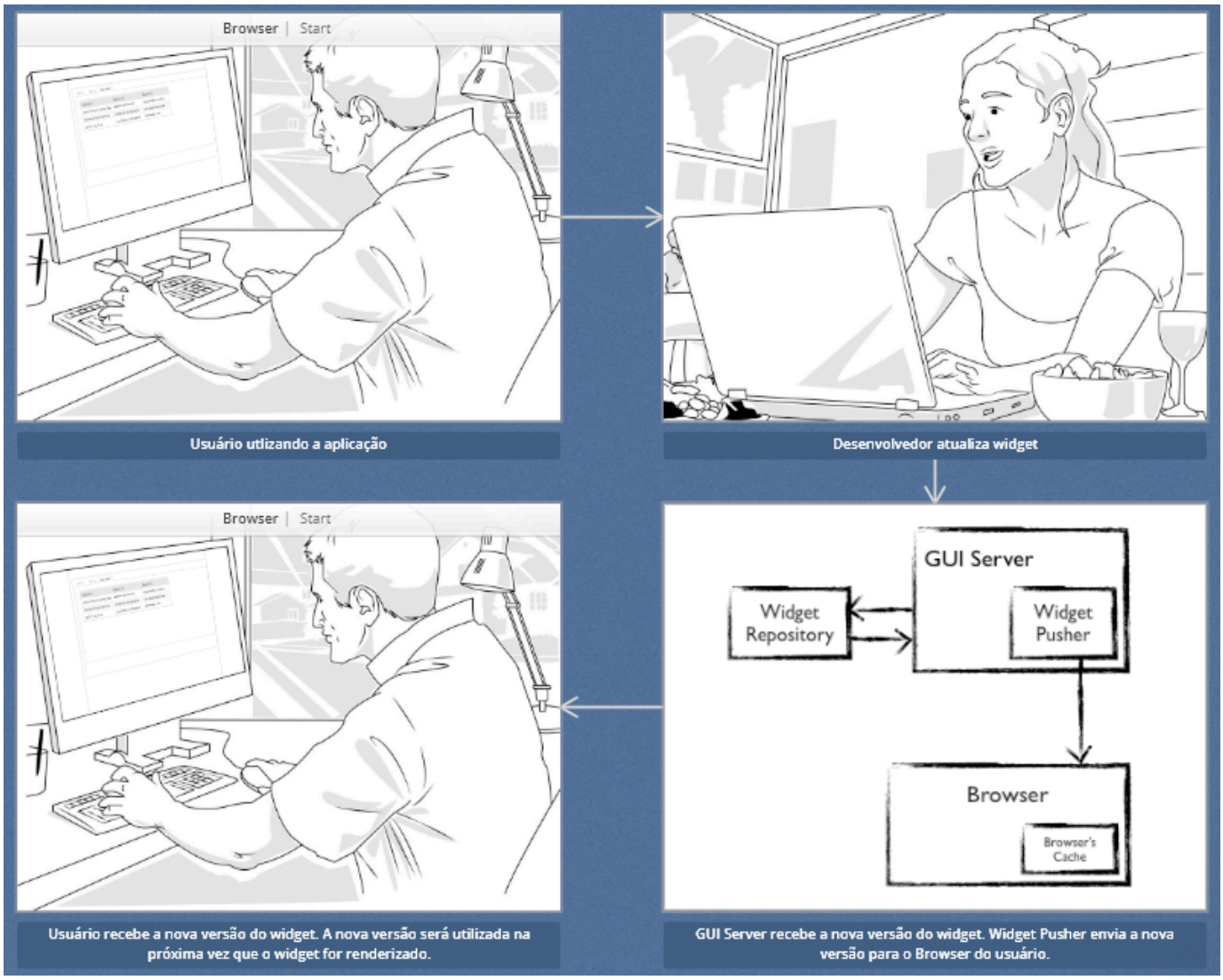


Figura 20 - Cenário 6, o widget é atualizado com o usuário online

# 5 IMPLEMENTAÇÃO

Durante o desenvolvimento deste trabalho, um protótipo foi implementado com o objetivo de validar a pesquisa. Como a primeira versão do LOM ainda está em desenvolvimento, o cache foi implementado baseado em um mock do que seria um sistema LOM. Para o desenvolvimento do protótipo, foram criados mocks do GUI Server, Widget Repository e Business Server. Mocks são objetos que simulam o comportamento de outros objetos de uma forma mais controlada, normalmente usados para realização de testes.

Foi criado um mock do GUI Server desenvolvendo uma API que retornava um conjunto de widgets pré-definidos, da forma que a aplicação exemplo esperava. Essa API dava uma resposta padrão que era utilizada pelo protótipo e mantinha todos os widgets em memória.

O mock criado para o Widget Repository era bem mais simples, sendo apenas um diretório com todos os widgets (scripts JavaScript). O GUI Server carrega todos eles para a memória e começa servi-los quando o protótipo é iniciado.

## 5.1 Tecnologias Utilizadas

Algumas tecnologias foram utilizadas no desenvolvimento do protótipo (Figura 22) durante a pesquisa, foram elas: JavaScript, Node.js, jQuery e Socket.io.

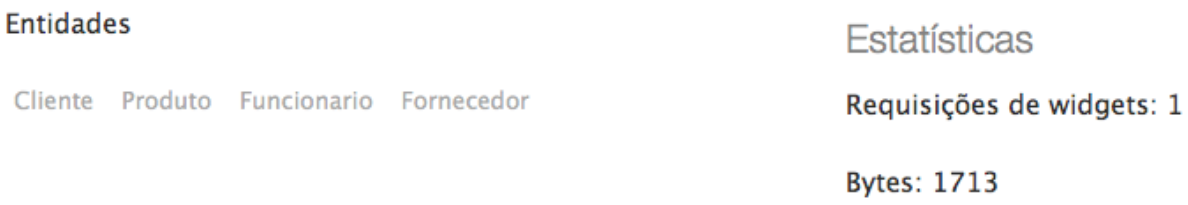


Figura 21 - Tela inicial do protótipo

### 5.1.1 JavaScript e Node.js

Atualizações em tempo real são uma tendência no desenvolvimento web moderno, sendo que a web como plataforma até alguns anos atrás não possuía nenhuma forma nativa que possibilitasse o desenvolvimento disso de modo mais natural e sem a utilização de gambiarras. Muitas técnicas foram desenvolvidas(como Polling, Long Polling e etc) para tentar trazer eventos em tempo real para aplicações web, mas só após o anúncio do WebSocket, que faz parte das especificações do HTML5, é que essas notificações em tempo real se tornaram mais tangentes para outros desenvolvedores, sem um grande overhead no servidor e com menor complexidade.

JavaScript é a linguagem da Web. É uma linguagem interpretada simples baseada em protótipos que foi desenvolvida com o intuito de manipular elementos de uma página web, como imagens e formulários. Apesar de ter sido considerada uma linguagem deficiente por

muito tempo, nos últimos anos a sua adoção tem crescido bastante, e fora utilizada para os mais diversos fins, indo muito além de scripts que rodam apenas no browser. Atualmente é possível desenvolver programas em JavaScript para as mais diversas plataformas, sendo possível escrever aplicações desktop, mobile, utilitários de linha de comando, extensões para outros programas(como Photoshop e Firefox), além de programas server-side[27]. Apesar de não possuir classes, funções são objetos de primeira classe e podem ser usadas para as mais diversas finalidades, além de possuir uma notação bastante versátil de objetos literais. O JavaScript é implementado pelo fabricante do browser seguindo a especificação da linguagem, chamada ECMAScript [30], e originalmente utiliza a API do DOM(Document Object Model) para manipular os elementos HTML de uma página web. O problema é que essa API foi muito mal especificada, o que gerou diferentes implementações, cada browser contendo suas particularidades, dificultando a portabilidade do código JavaScript entre os browsers. Para solucionar esse problema, John Resig lançou em 2006 o jQuery [31], uma biblioteca desenvolvida em cima das APIs existentes que visava unificar e simplificar o desenvolvimento, e a manipulação do DOM, sem que o desenvolvedor precisasse se preocupar com as particularidades da API do DOM de cada browser.

O Node.js nada mais é que um ambiente de JavaScript server-side que roda sobre a engine de JavaScript do Google Chrome, chamada V8 [29]. Tanto a V8 como grande parte do Node.js são implementados em C e C++, focando em performance e baixo consumo de memória [28, 29], o que os garante um grande desempenho. Diferente da maioria dos ambientes modernos, um processo do Node.js não utiliza multithreading para suportar concorrência, em vez disso, aposta em um esquema orientado a eventos, baseado em um modelo de I/O assíncrono. Por utilizar esse modelo baseado em eventos, o JavaScript se torna uma excelente linguagem para essa abordagem, pois possui suporte a callbacks que são fáceis de usar junto com funções anônimas. Apesar de os desenvolvedores utilizarem o formato de multithreading para o desenvolvimento de servidores web, é sabido que a programação em múltiplos cores é bastante complexa e propensa a erros [29], além de que o desenvolvedor perde parte do controle da aplicação pelo fato de geralmente ser responsabilidade do sistema operacional decidir qual thread será executada e por quanto tempo. A abordagem orientada a eventos utilizada pelo Node.js oferece uma alternativa mais eficiente, escalável e simples para o desenvolvedor, visto que a aplicação depende de eventos de notificação para decidir o que irá fazer, sem que ocorram bloqueios, isso o torna uma ótima opção para serviços que possuem um grande número de clientes conectados, como em um servidor de WebSockets, que é utilizado na solução proposta.

### 5.1.2 Cache do Browser

Essa camada de persistência e acesso rápido foi desenvolvida sobre o LocalStorage do browser com o intuito ser utilizada como um cache para os widgets, esses widgets são armazenados utilizando seu nome como chave e o um objeto literal com informações do widget, como o código, sua versão e seu nome como valor. Quando um widget sofre alguma alteração no repositório de widgets, o cache deve ser invalidado pois possui uma versão antiga, essa invalidação acontece com o servidor fazendo um push do novo widget para o browser, atualizando o cache. Ao fim, a nova versão do widget será utilizada na próxima atualização da página, já utlizando a nova versão presente no cache. Caso o usuário não esteja online quando o widget for atualizado, ele receberá a nova versão quando abrir a aplicação novamente, pois sempre que a aplicação é iniciada, os widgets do cache são atualizados para sua última versão disponível no repositório(Figura 21).

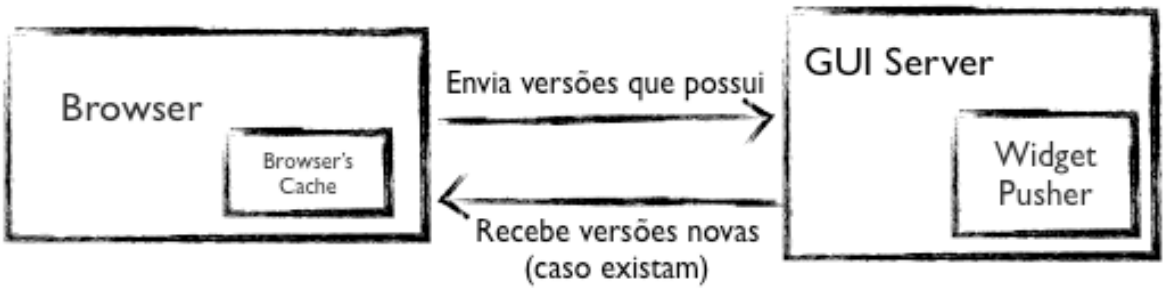


Figura 22 - Handshake

Para ser armazenado no LocalStorage, esse objeto literal é "*stringified*", ou seja, ele é transformado em string. Quando ele precisa ser recuperado, é feito um parse dessa string para transformá-lo um objeto literal novamente. JavaScript possibilita isso através dos métodos 'JSON.stringify()' e 'JSON.parse()'.

O protótipo desenvolvido encontra-se hospedada no heroku, um PaaS baseado em um sistema de controle de versão chamado git, abstraindo toda necessidade de configuração de servidores, *deployment*, etc; podendo ser acessada através do endereço: http://lom.herokuapp.com. A tela inicial é bastante simples, composta por dois widgets, sendo um o widget principal(*root*) e outro para listar as classes. Do lado direito da aplicação é possível ver algumas estatísticas, que mostram alguns dados como número de requisições feitas para carregar widgets e quantidade de bytes trafegados na rede. Clicando nas entidades, podemos ver algumas "instâncias" listadas em uma tabela.

Vendo o console do browser(Figura 23) é possível ver mais detalhes do funcionamento. No console tudo que está acontecendo é informado, por exemplo, ao abrir a página da aplicação pela primeira vez, são impressas algumas mensagens. A primeira é uma mensagem de boas vindas do servidor, que ocorre apenas após a página ser carregada. A segunda mensagem é o resultado de uma tentativa de carregar os widgets do cache do browser, podendo resultar em MISS ou HIT. O MISS ocorre quando o widget não é encontrado no cache. Quando isso ocorre, uma terceira mensagem é impressa, informando que o widget está sendo carregado do servidor. O HIT é quando o widget é encontrado no cache local, quando isso ocorre, o widget e carregado e renderizado sem que seja feita uma requisição para o servidor.

```
Server says: howdy client!
Cache MISS!
Loading widget from server...
>
```

Figura 23 - Logs do protótipo

# 6 AVALIAÇÃO

Após o desenvolvimento da prova de conceito, foi feita uma pequena avaliação para validar os benefícios do caching de widgets. A avaliação foi feita comparando uma implementação do LOM com cache e outra sem cache, contando a quantidade de requisições e transferência de bytes em cada uma delas. Os protótipos possuem apenas dois widgets simples, e para simular uma sobrecarga, adicionamos valores vazios aos códigos dos widgets para que o tamanho final deles aumentassem sem que nada fosse exibido.

Os dois widgets utilizados são: o UIRootWidget e o TableInstanceListing. O tamanho final do UIRootWidget foi aumentado para 273921 bytes. O TableInstanceListing ficou com 249522 bytes. Ambos ficaram com cerca de 0.2 megabytes, um valor relativamente alto, tratando-se de um componente visual.

Para os testes, foi criado um cenário de utilização simples, que consistia de dois passos básicos, realizados duas vezes, totalizando quatro passos. O primeiro passo é abrir a aplicação, o segundo é clicar nas entidades Cliente, Produto, Funcionário e Fornecedor(Figura da tela). Após esses dois passos iniciais, o usuário fecha a aplicação e os realiza novamente. Ao fim dos quatro passos, os resultados mostraram:

| | **LOM sem Cache** | **LOM com Cache** |
|---|---|---|
| Requisições | 10 | 2 |
| Dados transferidos | 2544018 bytes (~2.4mb) | 523443 bytes (~0.5mb) |

Tabela 2 - Alterações feitas pelo LOM no padrão Type Square

A aplicação sem cache teve um custo cerca de 5 vezes maior que a aplicação com cache, em ambos critérios. Apesar do experimento ter sido realizado em um ambiente controlado, os resultados mostram que o valor gerado pelo cache pode ser crucial para performance da aplicação e a experiência do usuário. Nos testes os tempos e os valores das requisições dos dados de informação foram ignorados, levando em consideração apenas os dos widgets.

# 7 CONCLUSÃO E SUGESTÕES DE TRABALHOS FUTUROS

Após analisarmos os conceitos por trás das meta-arquiteturas, atacamos um dos principais problemas encontrados neste tipo de sistema: a representação dos dados. A ideia de utilização de cache para otimizar esse fim é sugerida pelos autores [22], partimos desse princípio e pesquisamos as mais diversas técnicas de cache buscando uma que melhor se adequasse aos nossos requisitos, no contexto de uma aplicação web.

Para validar a solução proposta, implementamos um protótipo para utilizar como prova de conceito. A solução mostrou-se bastante eficiente, mostrando uma grande economia no número de requisições e dados trafegados durante a execução da aplicação. O que acabou limitando a avaliação foi a falta de widgets para simulação de um ambiente mais real, principalmente pela ausência de PropertyRenderers, que são os widgets mais utilizados. Isso abre espaço para que outras análises sejam feitas com base nesse estudo. A utilização de caches em outros níveis da aplicação também pode ser explorada, como no GUI Server, onde um cache local evitaria requisições desnecessárias ao Widget Repository.

O trabalho foi baseado em uma simulação de como seria um widget do LOM. A solução proposta é adaptável para qualquer que seja a implementação final dos widgets. Esse aspecto do formato dos widgets ainda é um pouco obscuro, pois o framework de widgets LOM ainda está em desenvolvimento. A forma como os widgets serão desenvolvidos ainda precisa ser melhor explorada, abrindo possibilidades para trabalhos futuros. Espera-se que a solução atual funcione bem com o framework que está em desenvolvimento, dado que os widgets continuarão sendo implementados em JavaScript.

# REFERÊNCIAS BIBLIOGRÁFICAS


[1] UBL, Malte; KITAMURA, Kitamura; *Introducing WebSockets: Bringing Sockets to the Web*. Disponível em: <http://www.html5rocks.com/pt/tutorials/websockets/basics>. Acessado em 20 de setembro de 2013.

[2] HICKSON, Ian; *The WebSocket API*. Disponível em: <http://dev.w3.org/html5/websockets/>. Acessado em 23 de setembro de 2013.

[3] FETTE, I.; MELNIKOV, A. *The WebSocket Protocol*. Disponível em: <http://tools.ietf.org/html/rfc6455>. Acessado em 12 de setembro de 2013.

[4] HANSSON, David H. *How Key-based Cache Expiration Works*. Disponível em: <http://37signals.com/svn/posts/3113-how-key-based-cache-expiration-works>. Acessado em 25 de setembro de 2013.

[5] KUPFERMAN, Jonathan. *Web Application Caching: Generational Caching.* Disponível em: <http://www.regexprn.com/2011/06/web-application-caching-strategies_05.html>. Acessado em 25 de setembro de 2013.

[6] KUPFERMAN, Jonathan. *Web Application Caching: Write-through caching*. Disponível em: <http://www.regexprn.com/2011/06/web-application-caching-strategies.html>. Acessado em 25 de setembro de 2013.

[7] BIDELMAN, Eric. *Stream Updates with Server-Sent Events*. Disponível em: <http://www.html5rocks.com/en/tutorials/eventsource/basics/>. Acessado em 23 de setembro de 2013.

[8] LAMM, Steve; *HTTP Caching*. Disponível em: <https://developers.google.com/speed/articles/caching>. Acessado em 23 de setembro de 2013.

[9] WESSELS, Duane. Web Caching, O'Reilly Media; 1st edition, 2001.

[10] SHKLAR, Leon; ROSEN, Richard. *Web Application Architecture: Principles, Protocols and Practices*. John Wiley & Sons, 2003.

[11] YODER, J.; JOHNSON, R. *The Adaptive Object-Model Architectural Style*. IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance (WICSA 2002), Montréal, Québec, Canada, 2002.

[12] WELICKI, L.; YODER, J.; BROCK, Rebecca W. *Rendering Patterns for Adaptive Object-Models*, Proceedings of the 14th Conference on Pattern Languages of Programs, ser. PLOP '07. New York, NY, USA: ACM, 2007.

[13] FOOTE, B.; YODER, J. Metadata and Active Object Models. Proceedings of Plop98. Technical Report #wucs-98-25, Dept. of Computer Science, Washington University Department of Computer Science, October 1998.

[14] REVAULT, N.; YODER, J. *Adaptive Object-Models and Metamodeling Techniques*. Proceedings of the 15th European Conference on Object Oriented Programming (ECOOP 2001). Budapest, Hungary. 2001.

[15] YODER, J.; BALAGUER, F.; JOHNSON R.; *Architecture and Design of Adaptive Object-Models*. Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA 2001), Tampa, Florida, USA, 2001.

[16] YODER, J.; RAZAVI R.; Metadata and Adaptive Object-Models. ECOOP Workshops (ECOOP 2000), Cannes, France, 2000.

[17] PILGRIM, Mark; *HTML5: Up and Running*, O'Reilly Media; 1 edition, 2010.

[18] HICKSON, Ian. *Web Storage.* Disponível em: <http://www.w3.org/TR/webstorage/>. Acesso em 27 de setembro de 2013.

[19] RAUCH, Guillermo. *Websockets Everywhere with Socket.IO*, 2010. Disponível em: <http://howtonode.org/websockets-socketio>. Acesso em: 27 de setembro de 2013.

[20] FERREIRA, Hugo J. S. L. *Adaptive Object-Modeling Patterns, Tools and Applications*, 2010.

[21] DIJKSTRA, Edsger W. *The Humble Programmer*, Communications of the ACM 15, no. 10, 859-866, 1972.

[22] Welick, L.; YODER, J. *A Pattern Language for Adaptive Object Models: Part I - Rendering Patterns,* 2008.

[23] Google Gears: http://pt.wikipedia.org/wiki/Google_Gears

[24] KIRCHER, M.; JAIN P. *Pattern Oriented Software Architecture, Volume 3: Patterns for Resource Management*. Wiley & Sons. 2004.

[25] MATSUMOTO; Guerra. *An Architectural Model for Adapting Domain-Specific AOM Applications*, Dept. de Cienc. da Comput., Inst. Tecnol. de Aeronaut., São José dos Campos, Brazil.

[26] YODER, Joseph; BALAGUER, Federico; JOHNSON, Ralph. *Adaptive Object-Models for Implementing Business Rules*, Urbana, 2001.

[27] STEFANOV, Stoyan. *JavaScript Patterns*, O'Reilley Media, 2010.

[28] MACCAW, Alex. *JavaScript Web Applications*, O'Reilley Media, 2011.

[29] TILKOV, Stefan; VINOSKI, Steve. *Node.js: Using JavaScript to Build High-Performance Network Programs*, IEEE Internet Computing, vol. 14, 2010.

[30] VILAR, Rodrigo A.; NETO, Sinval M.; XAVIER, Rafael; ALMEIDA, Hygoo O. *Living Object Model: reducing the essential complexity of information systems development*, 2013.

[31] FOWLER, Martin. *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1997.

[32] FIELDING, Roy T. *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, 2000.

[33] BROOKS, F. No Silver Bullet: Essence and Accidents of Software Engineering, IEEE computer, 1987.