INSTITUTO POLITÉCNICO
DE VIANA DO CASTELO

Célia da Conceição Barros de Sousa

LINGUAGEM E AMBIENTE PARA MODELAÇÃO DA INTERFACE COM O UTILIZADOR DE APLICAÇÕES DE SOFTWARE ORIENTADAS AOS DADOS

Mestrado em Engenharia de Software

Trabalho de Projeto efetuado sob a orientação do
Doutor António Miguel Cruz

Julho de 2015

RESUMO

O desenvolvimento de sistemas interativos envolve normalmente a modelação, construção e integração de diferentes componentes separados. A *User Interface* (UI) é o componente através do qual o utilizador acede às funcionalidades do sistema e o seu desenvolvimento tipicamente envolve também a criação de modelos. Na Engenharia de *Software* a prática comum é a criação de um modelo do sistema utilizando a *Unified Modelling Language* (UML). Contudo, a linguagem UML não apresenta um suporte concreto para a modelação abstrata da UI pelo que as abordagens existentes, fora do âmbito da UML, não complementam os modelos tipicamente usados na Engenharia de *Software,* ou seja não existe uma correspondência concreta entre os seus elementos.

De forma a colmatar este problema este trabalho de mestrado apresenta a linguagem *xCAP,* para modelação de interfaces com o utilizador, baseada na linguagem *Canonical Abstract Prototype* e na correspondência dos seus elementos concretos com os elementos abstratos ou conceitos de um *User Interface Metamodel* selecionado da literatura. Como complemento à criação da linguagem *xCAP,* e como parte integrante deste trabalho, foi desenvolvida a aplicação *web MetaCAP* cujo objetivo é permitir a criação e edição de modelos da UI de *software* orientado aos dados baseados na linguagem *xCAP*, e integrados com os modelos UML mais usados.

Em suma, a criação da linguagem *xCAP* e da aplicação *MetaCAP* tem como objetivo permitir o relacionamento/associação entre os diferentes modelos que descrevem um sistema e a adoção de nomenclaturas semelhantes na sua construção.

Julho de 2015

ABSTRACT

The development of interactive systems typically comprises the modeling, development and integration of different separate components. The *User Interface* (UI) is the component through which the user accesses system functionalities and its development also typically involves the creation of models. In Software Engineering the common practice is to create a system model using the *Unified Modelling Language* (UML). However, the UML does not present a concrete notation for abstract UI modeling. Other approaches, out of the scope of UML, do not complement the models typically used in Software Engineering, so there is no concrete match between them.

In order to overcome this problem, this Master's project proposes the *xCAP* language for modeling user interfaces, based on *Canonical Abstract Prototype* language and its correspondence with a *User Interface Metamodel* proposed in the literature. As a complement to the creation of *xCAP* language, and as part of this project, the *METACAP* web application has been developed, with the goal of allowing the creation and edition of UI models of data-oriented software applications based on the *xCAP* language.

In short, the creation of *xCAP* language and *METACAP* application aim to allow the relationship/association between the different models that describe a system and the adoption of a similar language in its construction.

Julho de 2015

CONTEÚDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABELAS

GLOSSÁRIO

AAIO	*Action Abstract Interaction Object*
AIO	*Abstract Interaction Object*
AUIM	*Abstract User Interface Model*
AUIMM	*Abstract User Interface Metamodel*
BLL	*Business Logical Layer*
CAC	*Canonical Abstract Component*
CAP	*Canonical Abstract Prototype*
CRUD	*Create, Retrieve, Update, Delete*
CSS	*Cascade Style Sheets*
DAIO	*Data Abstract Interaction Object*
DAL	*Data Access Layer*
DM	*Domain Model*
DMM	*Domain Metamodel*
DOM	*Document Object Model*
HTML	*Hypertext Markup Language*
IS	*Interaction Space*
JSON	*Javascript Object Notation*
MBUID	*Model-Based User Interface Development*
MOF	*Meta Object Facility*
MDA	*Model Driven Architecture*
MDD	*Model Driven Development*
OMG	*Object Management Group*
PIM	*Platform Independent Model*
PSM	*Platform Specific Model*
SIO	*Simple Interaction Object*
SVG	*Scalable Vector Graphics*
UCM	*Use Case Model*
UI	*User Interface*
UIDL	*User Interface Description Language*

UIM	*User Interface Model*
UML	*Unified Modelling Language*
UIMS	*User Interface Management System*
XML	*eXtensive Markup Language*

1. INTRODUÇÃO

No processo de criação de um produto de *software* são várias as etapas a ter em conta. Neste âmbito a Engenharia de *Software* representa um papel muito importante na medida em que fornece as ferramentas e boas práticas não só para a formulação inicial do que o sistema deverá fazer, como o deverá fazer e quais os atores envolvidos, mas também para o suporte a todo o processo de desenvolvimento de *software*.

O desenvolvimento de sistemas interativos é normalmente separado em diferentes componentes que por sua vez são também desenvolvidas por pessoas distintas. A *User Interface* (UI) é o componente através do qual o utilizador acede às funcionalidades do sistema e o seu desenvolvimento envolve habitualmente a criação de protótipos e/ou modelos. O protótipo é muito útil na altura de apresentar o futuro produto aos *stakeholders* na medida em que apresenta um possível aspeto final do sistema, facilita o levantamento de requisitos funcionais e permite a validação prévia de requisitos da interface. O modelo captura as partes mais relevantes do domínio em estudo e possibilita a definição de problemas e respetivas soluções para além de também permitir a comunicação com os *stakeholders*.

Na Engenharia de *Software* a prática comum é a criação de um modelo utilizando a *Unified Modelling Language* (UML). Neste caso, são tipicamente projetados um modelo de domínio (*Domain Model*, DM) e um modelo de casos de uso (*Use Case Model*, UCM), apoiados num protótipo não funcional de uma UI. No DM são representadas as principais classes (entidades) do sistema, os seus atributos, relacionamentos e, nalguns casos, as operações. O UCM apresenta as principais funcionalidades do sistema do ponto de vista do utilizador. Contudo, um modelo de um sistema composto apenas pelos modelos de Domínio e de Casos de Uso é, na maior parte dos casos, ambíguo e incompleto, faltando-lhe nomeadamente um modelo de como o sistema deverá ser apresentado aos seus utilizadores, ou seja um modelo da UI.

Linguagens de modelação como a UML, não apresentam um suporte concreto para a modelação abstrata da UI. Desta forma as diferentes abordagens existentes, independentes da UML, apesar de apresentarem alguns resultados, não complementam os modelos tipicamente usados na Engenharia de *Software* pois não existe uma correspondência concreta entre os elementos propostos [1].

Este trabalho baseia-se em aplicações de *software* orientadas aos dados, ou seja, aplicações que permitem a introdução, consulta, alteração e eliminação de dados associados a um contexto específico. Este tipo de aplicações apresentam UIs baseadas em formulários *(form-based)* que proporcionam um conjunto de formulários organizados em sequências de navegação e elementos gráficos *standard* como caixas de texto, menus *dropdown*, botões, etc.

De entre as abordagens apresentadas ao longo deste trabalho, e tendo em conta o projeto a desenvolver, duas aproximações mereceram uma análise mais detalhada: a linguagem CAP *(Canonical Abstract Prototypes)* [2] e o Metamodelo para Interfaces com o Utilizador *(User Interface Metamodel)* proposto em [3, 4].

A linguagem CAP *(Canonical Abstract Prototype)* proposta em [2], utiliza uma notação gráfica para ilustrar de forma abstrata a UI de um sistema de *software.* Os elementos da CAP são divididos em três categorias: ferramentas *(tools),* materiais *(materials)* e materiais ativos *(active materials);* que pretendem separar as diferentes formas de interação e apresentação de informação numa UI. Contudo, dado o seu alto nível de abstração e o seu distanciamento dos modelos utilizados na Engenharia de *Software* trata-se de uma linguagem complexa na sua associação a modelos como o modelo de domínio ou o de casos de uso, difícil de traduzir numa UI mais concreta.

Em [3], é apresentado um Metamodelo para Interfaces com o Utilizador *(User Interface Metamodel,* UIMM). Este Metamodelo define os elementos abstratos necessários para representar as diferentes formas de apresentação de informação *(InteractionSpace* e *DataAIO)*, interação e navegação no espaço de uma UI *(ActionAIO)*; associando alguns destes elementos ao Metamodelo de Domínio (*Domain Metamodel,* DMM). Para além da definição dos elementos abstratos de uma UI, esta abordagem (posteriormente refinada em [4] e [5]) propõe a correspondência entre os objetos do Metamodelo e a linguagem CAP.

Assim sendo, de modo a complementar a abordagem feita em [3], [4] e [5], este projeto pretende:

- Definir a linguagem *xCAP,* para modelação de interfaces com o utilizador baseadas em formulários, fundamentada na linguagem CAP e garantindo a correspondência dos seus elementos com os elementos do UIMM, dando uma forma de representação aos conceitos nele incluídos;

- Desenvolver uma aplicação em ambiente *Web* para criação e edição de modelos da UI de *software* orientado aos dados baseados na linguagem *xCAP.*

O presente trabalho encontra-se dividido nos seguintes capítulos:

- *Capítulo 2 – Definições e Conceitos:* apresentação de conceitos necessários à compreensão das diferentes fases de desenvolvimento do projeto;

- *Capítulo 3 – Estado da Arte:* apresentação de metodologias e estudos existentes no que concerne ao desenvolvimento de *User Interfaces* (UI) recorrendo a linguagens específicas de modelação/prototipagem;

- *Capítulo 4 – Desenvolvimento do Trabalho:* apresentação do Metamodelo para Interfaces com o Utilizador e da linguagem *xCAP*;

- *Capítulo 5 – Implementação do Trabalho:* apresentação da arquitetura da aplicação *MetaCAP* e da *framework paper.js* utilizada para o seu desenvolvimento;

- *Capítulo 6 – Caso de Estudo:* apresentação dos testes efetuados na aplicação recorrendo à criação de Modelos da UI para pequenos exemplos que constituem casos de demonstração;

- *Capítulo 7 – Conclusões e Trabalho Futuro:* apresentação das ilações retiradas durante o desenvolvimento do trabalho bem como dos aspetos a melhorar futuramente.

2. DEFINIÇÕES E CONCEITOS

Este capítulo tem como objetivo apresentar os conceitos necessários à compreensão dos termos utilizados para apresentar as diferentes fases de desenvolvimento do projeto.

A título de introdução são dois os conceitos indispensáveis para compreender este documento: *modelo* e *metamodelo*. Um *modelo* pode ser entendido como uma imagem, desenho ou esquema que representa o objeto que se pretende reproduzir [6] (por exemplo uma *Interface* com o Utilizador). O *metamodelo*, por sua vez, é um tipo especial de modelo que especifica a sintaxe abstrata de uma linguagem de modelação [7]. Ainda neste âmbito, e porque se trata também de um modelo, é importante referir a noção de *protótipo*. Assim sendo, o *protótipo* é o modelo original, o padrão. Sendo que para o efeito do tema em estudo poderá entender-se como a versão preliminar do Modelo da UI a ser testado e aperfeiçoado [8].

No que diz respeito às metodologias que fundamentaram este trabalho e nas quais ele foi baseado (abordadas no Capítulo 3), são diversos os conceitos a apreender. Novamente é mencionado, ainda que implicitamente, o conceito de modelo, primeiro com a designação de *template* e posteriormente com a designação de *mockup*. Enquanto um *template*, padrão ou molde, se usa para designar um conjunto de dados previamente gravados (ou formatados) e que servem de base à inserção de outros dados [9], um *mockup* é um modelo em tamanho real de algo, que é usado para estudo, teste, ou como forma de mostrar as suas características [10]. Na Engenharia de *software* um *mockup* é um protótipo da interface de uma aplicação que pode variar no nível de fidelidade com o produto final. Também neste capítulo são várias as referências ao aspeto e organização gráfica da UI, nomeadamente quando são mencionados os conceitos de: *interface* – dispositivo (material e lógico) graças ao qual se efetuam as trocas de informações entre dois sistemas [11]; l*ayout* - modo de distribuição e arranjo dos elementos gráficos num determinado espaço ou superfície [12]; e *display* - apresentação visual de dados; visualização [13].

Finalmente neste contexto são vários os termos técnicos a ter em conta para melhor compreender as diferentes abordagens apresentadas, nomeadamente: *multiplataforma* - programa que pode correr em mais de uma plataforma [14]; *toolkit library* - biblioteca com ferramentas que auxiliam o utilizador a realizar determinada tarefa [9]; *runtime* - intervalo de tempo durante o qual é executado um programa [9]; *gramáticas (grammars)* – definem o conjunto de termos que compõe uma linguagem, bem como as regras que determinam a sintaxe da mesma [15]; *event based* - ambiente no qual uma ação ou ocorrência (evento), em geral provocada pelo utilizador (por exemplo, clicar num botão), gera uma resposta de um programa [16]; *callback* - função, procedimento ou método que é passado como um argumento em outra função [17]; *task model* -

modelo de tarefas proporciona uma descrição orientada para objetivos de sistemas interativos que permite apresentar os intervalos de tempo entre as diversas tarefas e a sua decomposição em subtarefas, evitando a necessidade de uma descrição detalhada da interface com o utilizador [18]; e *implementation model* - modelo de implementação refere-se ao plano a executar para implementar um determinado *design* para um sistema de *software* ou *hardware* concretos [19].

Relativamente à implementação prática do trabalho (apresentada no Capítulo 5) e dado que o elemento essencial à criação dos modelos projetados com a linguagem *xCAP* na página *web* é o *canvas* (HTML5), torna-se necessário introduzir as noções de *bitmap* e *immediate mode*. Assim sendo, um *bitmap* é a representação de uma imagem na memória do computador através de um conjunto de *bits*, em que cada *bit* corresponde a um *pixel* [20]. O *immediate mode* aplicado ao elemento *canvas* implica que assim que uma forma é desenhada no mesmo este deixa de ter informação sobre ela. A forma é visível mas não pode ser manipulada individualmente [21]. Ainda neste âmbito, dado que os elementos gráficos da linguagem *xCAP* foram "desenhados" recorrendo à *framework paper.js*, é importante apresentar os conceitos que fundamentam esta tecnologia. A estrutura hierárquica dos elementos que compõe a *paper.js* é definida pelo *scene graph*. Um *scene graph* é uma coleção de nós agrupados numa estrutura em árvore. Um nó da árvore pode ter muitos filhos, mas normalmente apenas um pai, como o efeito aplicado a um pai é propagado a todos os seus filhos, uma operação realizada num grupo é automaticamente propagada a todos os seus membros. O *scene graph* é utilizado por aplicações (*Adobe Illustrator, AutoCAD, CorelDRAW*, etc.) que permitem a edição de gráficos baseados em vetores e organiza a representação espacial de uma imagem [22]. Em suma, o *scene graph* funciona como DOM (*Document Object Model*) para a *framework paper.js* na medida em que é uma interface independente de plataforma e linguagem que permite que programas e *scripts* acedam e atualizem dinamicamente o conteúdo, estrutura e estilo de documentos [23]. Também neste contexto, é pertinente abordar o conceito de *Curva de Bézier*. A C*urva de Bézier* é uma curva polinomial expressa como a interpolação linear entre alguns pontos representativos, denominados pontos de controlo. É uma curva utilizada em diversas aplicações de edição/criação de imagem como o *Adobe Illustrator, Adobe Photoshop, GIMP, CorelDRAW,* e formatos de imagem vetorial como o SVG [24]. Apesar de não ter sido utilizada explicitamente na criação dos elementos da linguagem *xCAP* (pois não existem elementos compostos por formas curvilíneas), o facto de fazer parte da *framework* é relevante pois demonstra a versatilidade da mesma a nível gráfico. Finalmente, para perceber o tipo de interação que permitem os elementos criados com a *framework paper.js* devem ser apreendidos os conceitos de *scriptographer*, *hit testing* e *event handler*. O *scriptographer* é um *plugin* de *scripting* para o *Adobe Illustrator* que permite ao utilizador aumentar as funcionalidades do programa recorrendo à linguagem *Javascript*. Este *plugin* permite a criação de ferramentas de desenho controladas pelo

rato, efeitos para modificar gráficos já existentes e a utilização de *scripts* para criar novos gráficos [25]. Aliando o *scriptographer* às funcionalidades de *hit testing* (ato de situar o ponteiro do rato sobre um objeto e interagir com o mesmo, por exemplo, clicar com o rato num determinado objeto gráfico [26]) e aos *event handlers* (função ou método que contém instruções que são executadas como resposta a determinado um evento [27], ou *hit test*), a interatividade dos elementos gráficos aumenta consideravelmente.

Nos capítulos finais do trabalho é feita referência à aplicação *AMALIA IDE*. A *AMALIA IDE (Agile Model-driven AppLIcAtion Development Method and Tools)* é uma aplicação *web* que permite a criação/edição de Modelos de Domínio e Modelos de Casos de Uso, bem como a sua exportação para os formatos XML e JPG. A sua utilização neste trabalho é relevante na medida em que proporciona os modelos base para o desenvolvimento dos Modelos da UI, nomeadamente o Modelo de Domínio cujas classes e atributos são importadas pela *MetaCAP* para serem associados aos elementos gráficos da linguagem *xCAP*.

3. ANÁLISE DO ESTADO DA ARTE

Neste capítulo são apresentadas as metodologias e estudos existentes no que concerne ao desenvolvimento de *User Interfaces* (UI) recorrendo a linguagens específicas e modelação/prototipagem.

3.1 DESENVOLVIMENTO DA UI BASEADO EM LINGUAGENS DESCRITIVAS

Uma *User Interface Description Language* (UIDL) consiste numa linguagem de programação de alto nível que descreve as caraterísticas mais importantes de uma UI de uma aplicação interativa. Este tipo de linguagem envolve a definição de uma sintaxe (como as caraterísticas da UI podem ser expressas em termos de linguagem) e de uma semântica (o que as caraterísticas definidas significam no mundo real). Ao ser escrita recorrendo à linguagem XML *(Extensible Markup Language),* a UIDL torna-se transversal não ficando dependente da(s) linguagem(s) de programação utilizadas no sistema em desenvolvimento[28].

São exemplos de UIDLs as seguintes linguagens:

- *User Interface Markup Language (UIML):* é uma linguagem que permite aos *designers* descrever a UI em termos genéricos mas que permite definir uma descrição que mapeia a UI para diferentes Sistemas Operativos, linguagens de programação ou dispositivos (entidades externas). Um documento UIML é composto por três partes: descrição da UI, secção de mapeamento para as entidades externas e uma secção modelo (*template*) que permite reutilizar elementos criados anteriormente; Desta forma, a UI é descrita como um conjunto de elementos com os quais o utilizador interage. Cada parte da UI possui um estilo de apresentação (posição, tamanho do texto, etc.), bem como um conteúdo (texto, imagens, etc.) e, nalguns casos, uma ação associada [29];

- *Extensible Interface Markup Language (XIML):* é uma linguagem que permite descrever a UI sem se preocupar com a sua implementação. O seu principal objetivo é apresentar a UI num contexto abstrato (tarefas, domínio, etc.) e, posteriormente, num contexto concreto (apresentação e diálogo). Os elementos da XIML são organizados em conjuntos de componentes. Assim sendo, existem cinco componentes da *interface*: tarefas *(task component)*, domínio *(domain component)*, utilizador *(user component),* diálogo *(dialog component)* e apresentação *(presentation component);* para além dos componentes da interface, a XIML é também composta por atributos e relações. Um atributo é uma propriedade que tem um valor e pertence a um componente. Uma relação faz a ligação entre os componentes [30];

- *Extensible User Interface Language (XUL):* é a linguagem utilizada pela *Mozilla* (https://www.mozilla.org/), para descrever o *layout* das janelas. O seu objetivo é criar aplicações multiplataforma, fomentando a portabilidade das aplicações para todas as plataformas nas quais o *Mozilla* funciona. A XUL separa de forma clara a definição da aplicação cliente da lógica de programação, da apresentação ("*skins*" elaboradas recorrendo a CSS e imagens), e das *text-labels* específicas à linguagem. Desta forma a UI é descrita como sendo um conjunto de elementos estruturados (janelas, barras de menu, botões, etc.), juntamente como uma lista de atributos predefinida [31];

- *User Interface Extensible Markup Language (UsiXML):* é uma linguagem que descreve a UI em vários contextos: *Abstract User Interface* (AUI), *Concrete User Interface* (CUI), *Final User Interface* (FUI); e, para além disso, fá-lo numa perspetiva multimodal e multiplataforma [32].

Exemplo de aplicação: *UsiXML*

Para melhor explicar a utilização da linguagem *UsiXML* na criação de um elemento de UI concreto, é necessário em primeiro lugar abordar sucintamente o conceito de Desenvolvimento de UI *Multi-Direcional* (*Multi-Directional UI Development* - MDUID). O MDUID é baseado na *Cameleon Reference Framework* (Figura 1) que define as etapas de desenvolvimento de aplicações interativas em diversos contextos [32] e [33], a saber:

- *Task & Concepts (Conceitos e Tarefas):* descreve as várias tarefas a serem desempenhadas, bem como os conceitos (associados ao domínio) necessários para a tarefa ser realizada;

- *Abstract UI (UI Abstrata):* define espaços de interação ao agrupar subtarefas tendo em conta um conjunto de critérios, um esquema de navegação entre os espaços de interação e seleciona os *Abstract Interaction Objects* (AIOs) para cada conceito de forma a estes serem independentes em cada contexto de utilização;

- *Concrete UI (UI Concreta):* materializa a UI Abstrata de um determinado contexto na forma de *Concrete Interaction Objects (CIOs),* de forma a definir o *layout* e a navegação na *interface*;

- *Final UI (UI Final):* é a UI operacional, ou seja, a UI final executável em qualquer plataforma.

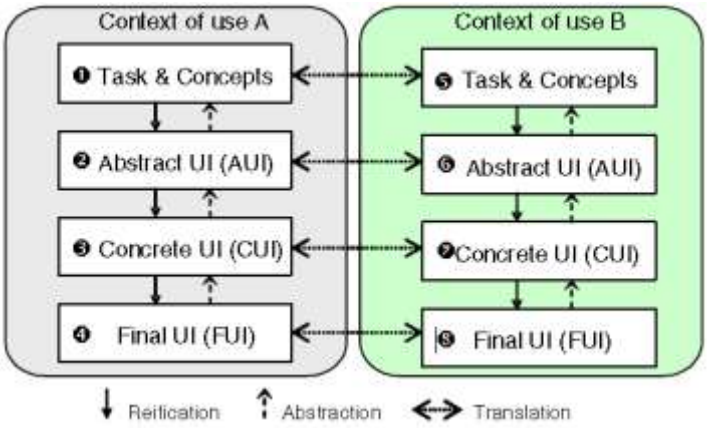


Figura 1 - *Cameleon Reference Framework* (retirado de [32])

Assim sendo, e tendo em conta as etapas apresentadas anteriormente, o seguinte exemplo (Figura 2) apresenta a representação de um botão (cujo objetivo é fazer o download de um ficheiro) recorrendo à *UsiXML*.

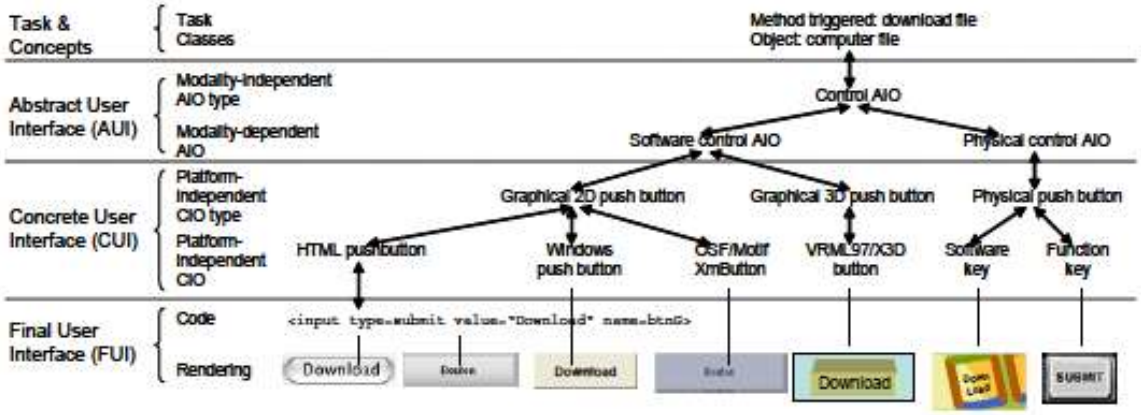


Figura 2 - Exemplo das etapas de representação utilizando a *UsiXML* (retirado de [32])

3.2 Desenvolvimento da UI baseado em Modelos

A utilização de modelos na criação da UI permite um desenvolvimento mais rápido e uma melhor qualidade do produto final. Um bom modelo clarifica dificuldades de *design* e permite a resolução mais eficaz de problemas. E, mais importante, a utilização de modelos permite saber o que construir, evita falhas e por essa razão minora o tempo gasto no desenvolvimento [34].

3.2.1 Model-Based User Interface Development (MBUID)

O *Model-Based User Interface Development* (MBUID) teve origem nos *User Interface Management Systems* (UIMS).

Um User Interface Management System (UIMS) é uma ferramenta, ou conjunto de ferramentas, que pretende especificar, implementar, testar e fazer a manutenção de uma User Interface [35].

Ao recorrer a esta técnica os *developers* escreviam a especificação da UI numa linguagem de alto nível específica em vez de utilizarem uma *toolkit-library*. A especificação criada era posteriormente traduzida, de forma automática, num programa executável ou interpretada em *runtime* de forma a gerar a UI projetada.

Inicialmente os UIMSs centravam-se na especificação do diálogo, utilizando diagramas de transição, gramáticas (*grammars*), ou representações *event-based* para especificar as respostas da UI aos eventos gerados pelos dispositivos de entrada. O aspeto gráfico da UI (*display* ou *layout*) era definido fora da linguagem de especificação utilizando procedimentos *callback* que por sua vez configuravam o ecrã apresentado ao utilizador [36].

Model-Based User Interface Development Architecture

A *Model-Based User Interface Development Architecture* (MBUIDA), apresentada na Figura 3, é a norma cujo objetivo é por em prática as premissas do MBUID. A MBUIDA é constituída por quatro componentes fundamentais, a saber: *Modelling Tools, Model, Automated Design Tools* e *Implementation Tools* [36].

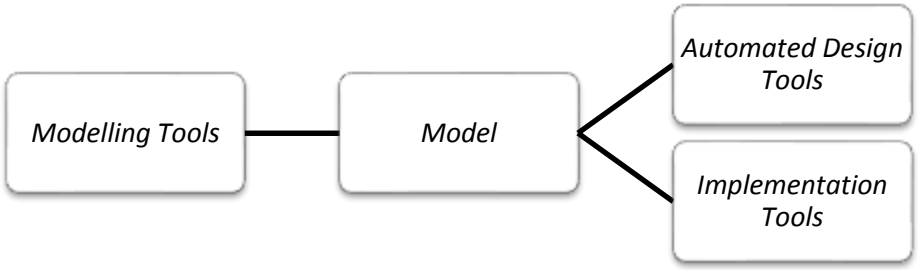


Figura 3 - *Model-Based User Interface Development Architecture*

Nesta abordagem o componente mais relevante para este estudo é o *Model*. O *Model* organiza a sua informação utilizando três níveis de abstração: *Task, Domain Models, Abstract UI Specification* e *Concrete UI Specification*.

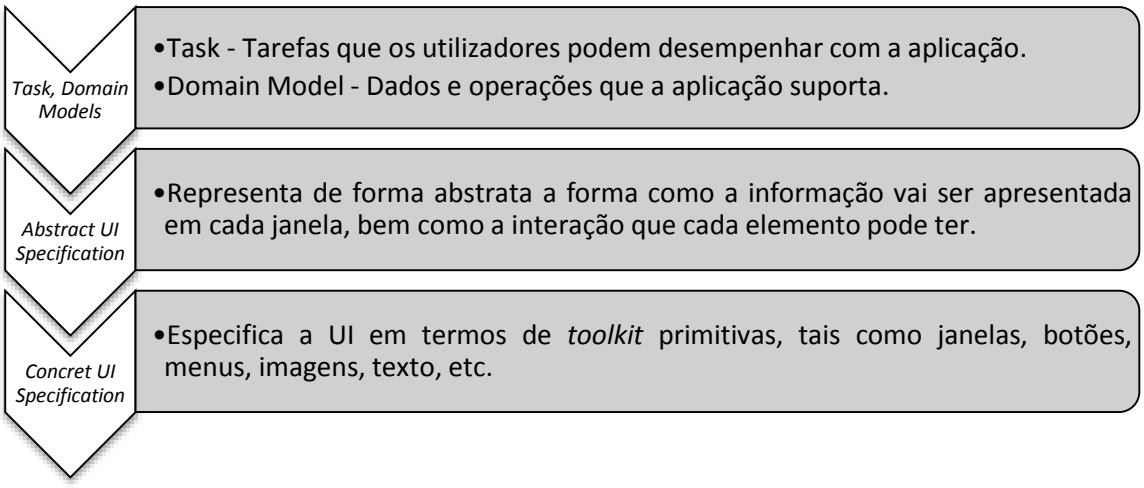


Figura 4 - Níveis de abstração do componente *Model*

De entre os três níveis de abstração do componente *Model*, cabe destacar o segundo nível: *Abstract UI Specification*. De acordo com a arquitetura MBUID, este nível tem como objetivo representar a UI em termos de três abstrações [36]:

- *Abstract Interaction Objects (AIO):* Representam tarefas de baixo-nível como selecionar um elemento ou apresentar uma *selection unit;*
- *Information Elements (IE):* Apresentam os dados a serem apresentados tais como, valores constantes numa *lable* ou conjuntos de objetos do modelo de domínio;
- *Presentation Units (PU):* São a abstração das janelas. Apresentam coleções de AIOs e de *Information Elements,* na forma de uma unidade.

3.2.2 Model Driven Development (MDD)

O *Model Driven Development* (MDD) tem como principal objetivo permitir que os modelos criados deem origem (de forma automática) a código que se irá traduzir num sistema completo (*User Interface* (UI) + *Business Logical Layer* (BLL) + *Data Access Layer* (DAL)). O MDD utiliza a linguagem UML, para construir os seus modelos.

Model Driven Architecture

A *Model Driven Architecture* (MDA), Figura 5, é o padrão desenvolvido pelo *Object Management Group* (OMG), para pôr em prática os objetivos do MDD. O padrão MDA tem como principal objetivo separar as tecnologias de desenvolvimento e a lógica da aplicação, criando uma metodologia *standard* para a construção/*design* de *software*.
Este padrão pode ser dividido em três etapas fundamentais: *Platform Independent Model* (PIM), *Platform Specific Model* (PSM) e a Geração do Código-Fonte [37].

Figura 5 - *Model Driven Architecture*

Para efeitos do tema em estudo, o PIM é neste ponto a etapa principal da MDA. Assim sendo, o PIM, tal como o nome indica, é um modelo independente de plataforma que descreve a totalidade do sistema nas suas diversas vertentes, sem nunca indicar como este será implementado nem quais as plataformas a utilizar.

São exemplos de PIMs os Modelos de Domínio *(Domain Models)*, os Modelos de Casos de Uso *(Use Case Models)* e os Modelos de Interface com o Utilizador *(User Interface Models)*.

Contudo, o desenvolvimento de aplicações baseado na linguagem UML é *system-centric*, ou seja, foca-se na arquitetura interna do sistema descurando a sua vertente interativa pelo que, esta linguagem carece de suporte para a criação de modelos de UI [38].

3.2.3 Analogia (MDD e MBUID)

Ao analisarmos as abordagens apresentadas em 3.3.1 e 3.3.2, chega-se à conclusão de que existe um paralelismo óbvio entre ambas. Se, se entender que o MBUID é uma especificação do MDD, considerando este aplicado ao desenvolvimento de sistemas interativos, e mais especificamente da UI, e se realizar o paralelismo entre as etapas de ambas arquiteturas, pode-se entender que o componente *Model* (MBUIDA) é uma especificação do PIM (MDA), cujo objetivo especifico é representar, ainda que de forma abstrata, a UI.

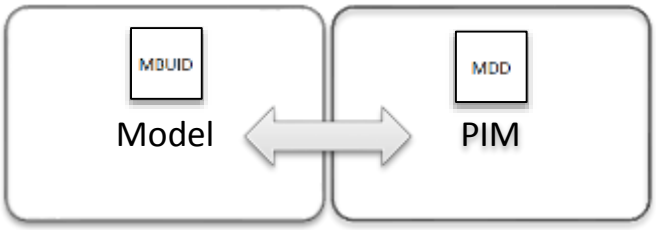


Figura 6 - Paralelismo entre MBUID e MDD

Contudo, e apesar de ambas metodologias proporcionarem *guidelines* para o desenvolvimento de UIs abstratas passíveis de serem geradas para ambientes multiplataforma, nenhum dos padrões proporciona uma linguagem concreta de elementos que permitam a representação da UI.

3.3 Desenvolvimento da UI baseado em Protótipos

3.3.1 Prototipagem

Em Engenharia de *Software* um protótipo de uma UI é uma representação do aspeto gráfico que uma determinada aplicação deverá ter, mas não só, deve também explicitar o tipo de elementos utilizados e a funcionalidade de cada um.

Dependendo da fase de desenvolvimento da aplicação, do seu grau de complexidade ou dos objetivos da mesma, o protótipo de uma UI pode ir desde um *mockup* (protótipo de baixa fidelidade) até uma cópia funcional do sistema.

Desta forma os protótipos da UI *(User Interface Prototypes)* podem ser classificados em quatro categorias [39]:

- *Protótipos de Apresentação (Presentation Prototypes):* protótipos construídos para mostrar como a aplicação poderá resolver alguns requisitos. São utilizados numa fase inicial do projeto e por essa razão apresentam um caráter simplista;
- *Protótipos Funcionais (Functional Prototypes):* Implementam tanto partes da UI como das funcionalidades da aplicação;
- *Breadboards:* Servem para explorar aspetos como a arquitetura do sistema ou funcionalidades da aplicação;
- *Sistemas piloto (Pilot Systems):* São protótipos numa fase final de desenvolvimento, que podem praticamente ser utilizados.

Se considerarmos que de categoria para categoria (Protótipos de Apresentação -> Sistemas Piloto), existe uma especificação do protótipo, à medida que as categorias avançam este é mais e mais direcionado para uma plataforma/arquitetura em particular. Desta forma, a possibilidade de criar um protótipo universal, composto por elementos reconhecíveis/utilizados por diversas plataformas é apenas possível com um alto nível de abstração que, segundo as categorias apresentadas se prenderia com a baixa fidelidade do mesmo (ao nível dos Protótipos de Apresentação), o que poderia criar uma série de problemas.

3.3.2 Protótipos Abstratos

Um Protótipo Abstrato *(Abstract Prototype),* ferramenta utilizada no *Usage-Centered Design* (Tabela 1), permite que os *designers* descrevam o conteúdo e organização geral de uma UI sem especificar um aspeto gráfico detalhado ou o seu comportamento, ou seja, é um modelo da arquitetura da UI que está a ser criada.

Tabela 1 - *Usage-Centered Design e User-Centered Design*

Usage-Centered Design	***User-Centered Design***
Foca-se nos utilizadores e na sua experiência de utilização e satisfação.	Foca-se na utilização, suportada por ferramentas que se preocupam com o cumprimento de tarefas.
Impulsionado pelo "*user input*", este está muito envolvido no desenvolvimento.	Impulsionada por modelos e "*modelling*", o "*user input*" é seletivo.
Design por prototipagem interativa.	*Design* por "*modelling*".
Processo informal e pouco especifico com muitas variáveis.	Processo muito específico e sistemático.
Centra-se na tentativa-erro e na evolução.	Centra-se na engenharia.

[40]

Na sua forma original (usualmente realizada utilizando elementos em papel), um protótipo abstrato consiste num *Content Model* e num *Navigation Map*.

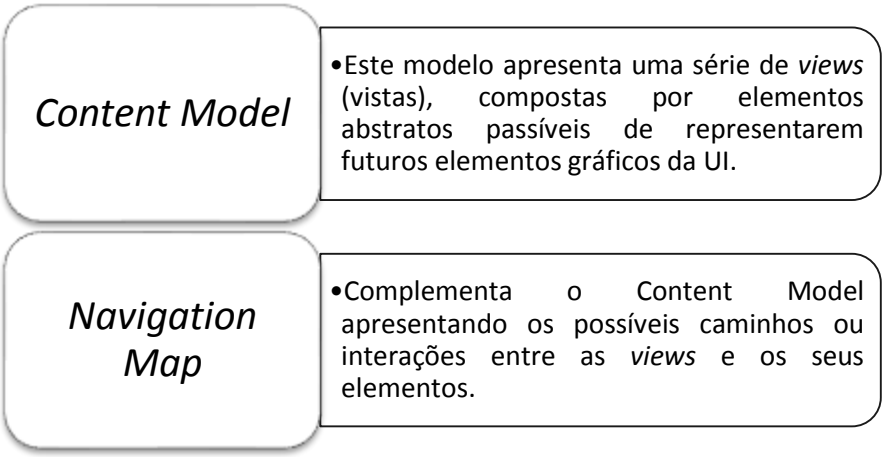


Figura 7 - Estrutura de um protótipo abstrato

Apesar da abstração anunciada, também os estes protótipos apresentam um nível variável de abstração, indo do *mockup* até um nível de especificação mais realista.

Contudo, e apesar do conceito de abstração se mostrar vantajoso num nível pouco aprofundado ou inicial do *design*, à medida que se vai avançando no desenvolvimento da aplicação, a abstração pode levantar alguns problemas, nomeadamente:

- Dificuldade em atribuir nomes ou descrever componentes de forma abstrata;
- Dificuldade em distinguir ferramentas de materiais;
- Dificuldade na tradução dos componentes abstratos em componentes concretos;
- Dificuldade na criação de ecrãs e outros tipos de UI a partir de *views* abstratas.

Para além disso, modelos compostos por componentes altamente abstratos tornam-se inteligíveis para as pessoas que não estiveram envolvidas na sua criação.

E, ainda que a utilização de protótipos abstratos seja uma mais-valia na criação de UIs: permitindo sistematizar processos, eliminando falhas e percas de tempo desnecessárias; tornou-se pertinente a criação de um protótipo abstrato, que sem descurar as preocupações do *Usage-Centered Design*, estivesse mais próximo da UI final (traduzindo melhor o *task model* e o *implementation model*) e fosse também mais fácil de compreender pelos *designers*.

Para colmatar os problemas apresentados, foi desenvolvida uma linguagem composta por elementos abstratos *standard* designada por *Canonical Abstract Components*.

3.3.3 Canonical Abstract Components

Os *Canonical Abstract Components* (CAC) proporcionam um *toolkit standard* de componentes abstratos para aplicar na prototipagem abstrata. Este conjunto de componentes consiste numa abordagem prática e utilizável/viável de componentes que abranjam as situações mais usuais que surgem no *user-centered design.*

Nesta abordagem todos os componentes são identificados por um nome e um ícone simples que serve como identificador para os *designers* e como auxílio visual para a interpretação do protótipo abstrato. Os CAC dividem-se em três categorias: *Materials, Tools* e *Active Materials.*

Como todos os componentes, independentemente da sua categoria, derivam de um componente genérico, os componentes genéricos (anotados com * nas figuras 8, 9 e 10) podem ser utilizados para qualquer propósito [2].

Materials (Materiais)

Existem quatro materiais básicos nos CAC: *Generic Container* (área genérica), *Single Elements* (elementos simples), *Collection* (coleção) e *Notification* (notificação).

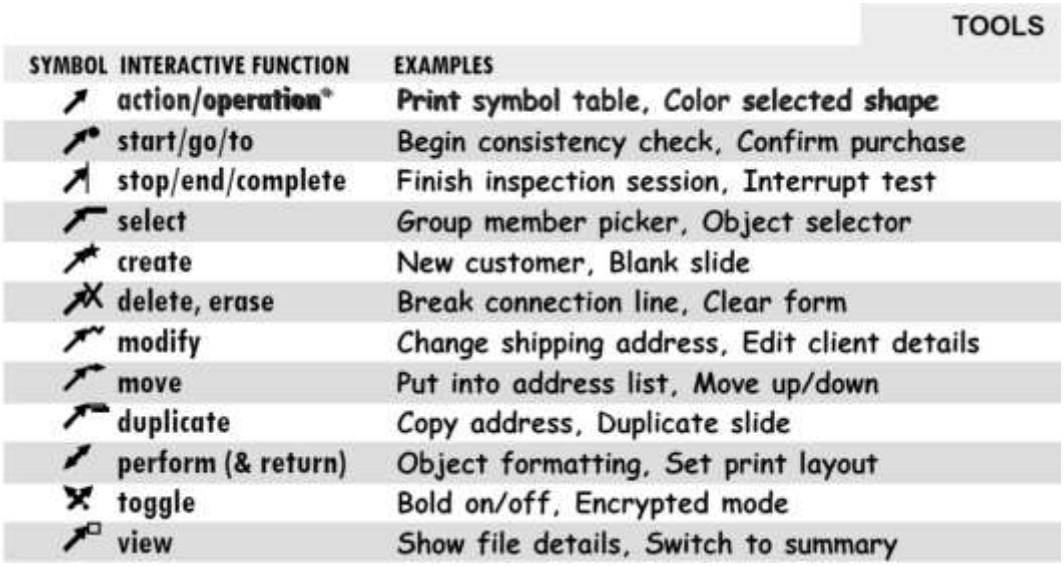
MATERIALS

SYMBOL	INTERACTIVE FUNCTION	EXAMPLES
	container*	Configuration holder, Employee history
	element	Customer ID, Product thumbnail image
	collection	Personal addresses, Electrical Components
	notification	Email delivery failure, Controller status

Figura 8 - Canonical Abstract Components – Materials (retirado de [2])

TOOLS (FERRAMENTAS)

As ferramentas existentes (Figura 9) podem ser do tipo *Operations* (operações) ou *Actions* (ações). As operações operam sobre os materiais e as ações causam ou despoletam algum evento.



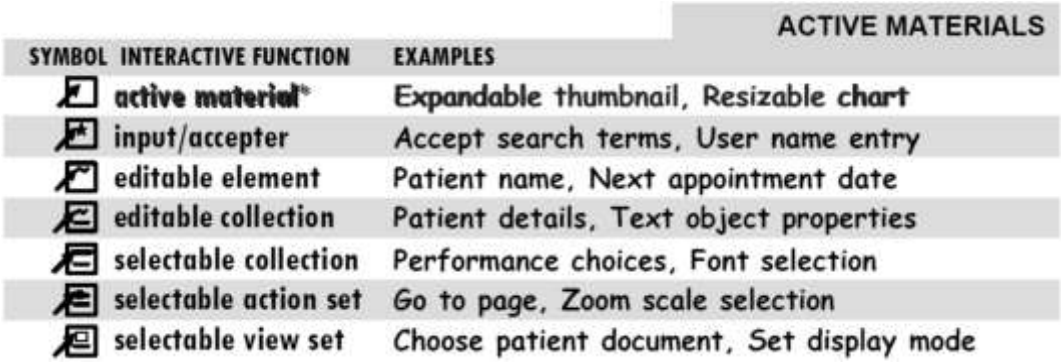
TOOLS

SYMBOL	INTERACTIVE FUNCTION	EXAMPLES
	action/operation*	Print symbol table, Color selected shape
	start/go/to	Begin consistency check, Confirm purchase
	stop/end/complete	Finish inspection session, Interrupt test
	select	Group member picker, Object selector
	create	New customer, Blank slide
	delete, erase	Break connection line, Clear form
	modify	Change shipping address, Edit client details
	move	Put into address list, Move up/down
	duplicate	Copy address, Duplicate slide
	perform (& return)	Object formatting, Set print layout
	toggle	Bold on/off, Encrypted mode
	view	Show file details, Switch to summary

Figura 9 - *Canonical Abstract Components – Tools* (retirado de [2])

ACTIVE MATERIALS (MATERIAIS ATIVOS)

Os *Active Materials* (Figura 10) são componentes abstratos que possuem caraterísticas comuns às *Tools* e aos *Materials*.



ACTIVE MATERIALS

SYMBOL	INTERACTIVE FUNCTION	EXAMPLES
	active material*	Expandable thumbnail, Resizable chart
	input/accepter	Accept search terms, User name entry
	editable element	Patient name, Next appointment date
	editable collection	Patient details, Text object properties
	selectable collection	Performance choices, Font selection
	selectable action set	Go to page, Zoom scale selection
	selectable view set	Choose patient document, Set display mode

Figura 10 - *Canonical Abstract Components - Active Materials* (retirado de [2])

Independentemente da sua categoria, deve ser atribuída aos componentes uma designação simples e objetiva mas que identifique sem dar aso a dúvidas qual a sua função, por exemplo:

- Lista de Endereços OU
- (*Collection*) Lista de Endereços

} *Materials*

- Fechar Lista de Endereços - *Tools*

Assim sendo, os componentes são mais facilmente identificáveis permitindo uma melhor leitura do modelo. Da mesma forma, a definição concreta dos componentes pode ser útil na associação do modelo de UI a outros modelos do sistema em desenvolvimento (Modelo de Domínio, Modelo de Casos de Uso, etc.) [2].

Da combinação dos *Canonical Abstract Components* num ou mais modelos (protótipos) resultam os *Canonical Abstract Prototypes* (CAP), cujo objetivo é representar, ainda que de forma abstrata, a UI de um determinado sistema em desenvolvimento.

O exemplo seguinte (Figura 11) ilustra a utilização dos CAC para criar um protótipo abstrato de uma UI.

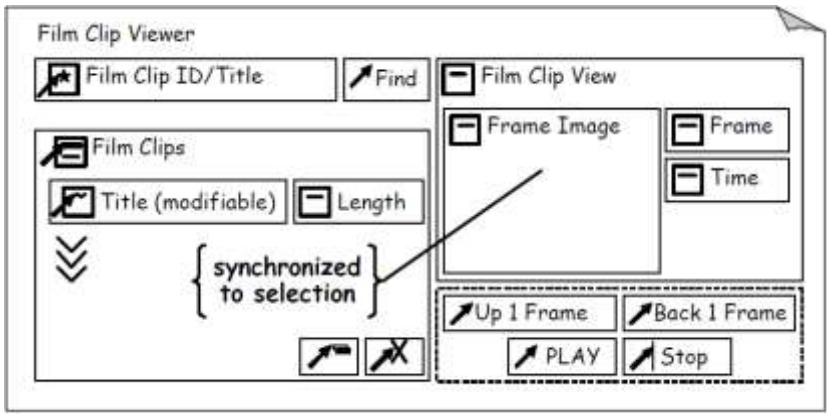


Figura 11 - Protótipo de UI (CAP) utilizando CAC (retirado de [2])

CRIAÇÃO DA UI A PARTIR DE UM CAP

O desenvolvimento de uma UI concreta a partir de um CAP envolve duas atividades distintas: *Visual Design* e *Interaction Design* (Figura 12).

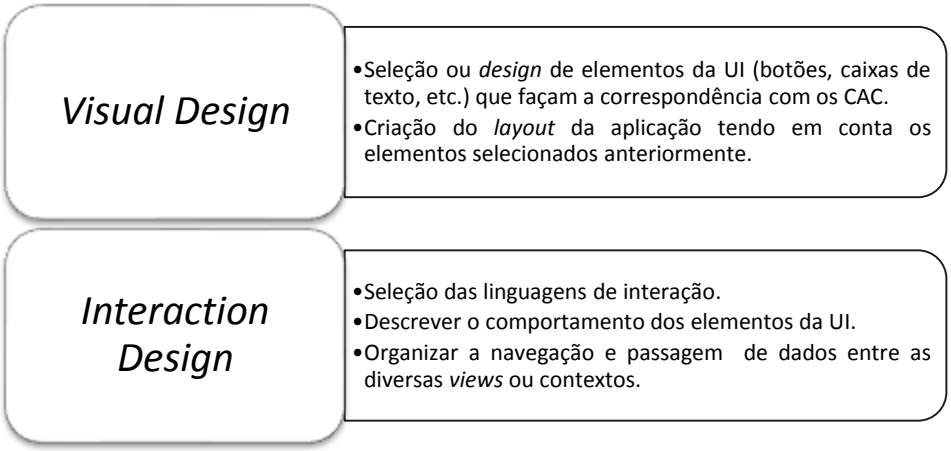


Figura 12 - Atividades necessárias à criação de uma UI a partir de um CAP

Os CAP apresentam-se como uma solução viável para a modelação da UI sem descurar o conceito de abstração mas oferecendo componentes traduzíveis em elementos concretos da UI para diversas plataformas. Para além disso, ao permitirem a atribuição de designações específicas aos diversos componentes, permitem a criação de uma ponte entre os modelos da UI e os demais modelos do sistema (Modelo de Domínio, etc.).

3.4 SUMÁRIO

Este capítulo abordou o estado da arte no âmbito da criação de *User Interfaces,* nomeadamente:

- Desenvolvimento de UI baseado em Linguagens descritivas – introdução de alguns exemplos de *User Interface Description Languages* (UIDL);
- Desenvolvimento de UI baseado em modelos – apresentação das metodologias *Model-Based User Interface Development* (MBUID) e *Model Driven Development* (MDD);
- Desenvolvimento de UI baseado em protótipos – introdução ao conceito de prototipagem, protótipo abstrato e também apresentação dos *Canonical Abstract Components* (CAC) e dos protótipos oriundos da sua aplicação, os *Canonical Abstract Prototypes* (CAP).

De entre as abordagens apresentadas, cabe enfatizar a importância do *Model Driven Development*, nomeadamente no que diz respeito à divisão por etapas definida na *Model Driven Architecture* (PIM -> PSM -> Geração do código fonte), e na sua definição do *Platform Independent Model* (PIM). Ainda neste âmbito, é relevante reter a noção de protótipo abstrato (análoga ao conceito de PIM), materializada pelos *Canonical Abstract Components.*

4. DESENVOLVIMENTO DO TRABALHO

Este capítulo tem como objetivo apresentar o Metamodelo para Interfaces com o Utilizador, a sua correspondência com os símbolos *Canonical Abstract Components*. E, finalmente a proposta de uma nova linguagem, a *xCAP*, a qual reutiliza os símbolos dos CAC como linguagem concreta do Metamodelo para Interfaces com o Utilizador apresentado.

4.1 METAMODELO PARA INTERFACES COM UTILIZADOR

Numa aplicação de *software* comum, a interação com o utilizador ocorre num espaço denominado área de interação *(Interaction Space).* As áreas de interação podem ser especificadas através de um Modelo de Interface com o Utilizador (*User Interface Model)*, que deverá ser definido de forma a ser independente de plataforma *(platform independent).* Tendo em conta que o trabalho desenvolvido se baseia em aplicações de *software* orientadas aos dados, e que estas por regra apresentam interfaces baseadas em formulários *(form-based),* a Figura 13 apresenta o metamodelo para o desenvolvimento de modelos abstratos de interface com o utilizador *(AUIM - Abstract User Interface Models)* baseados em formulários [5].

O *Abstract UI Metamodel* (AUIMM) é baseado na MOF e importa alguns dos elementos definidos no DMM, nomeadamente*: Entity, Property, Type e Operation* [3].

A Meta Object Facility (MOF) proporciona uma framework para gestão de meta-dados (metadata) independente de plataforma, bem como um conjunto de serviços de meta-dados que permitem o desenvolvimento e inter-operacionalização de sistemas metadata-driven e model-driven. Entre os exemplos de sistemas que utilizam a MOF podem-se destacar: ferramentas de modelação e desenvolvimento, sistemas de data warehouse e repositórios de meta-dados. A MOF contribuiu de forma significativa para a ideia base da Model Driven Architecture desenvolvendo os fundamentos estabelecidos pela UML e introduzindo os conceitos de metamodelos formais e Modelos Independente de Plataforma (Platform Independent Models - PIM) de meta-dados bem como o mapeamento dos PIMs para Modelos Especificos à Plataforma (Plartform Specific Models - PSM) [41].

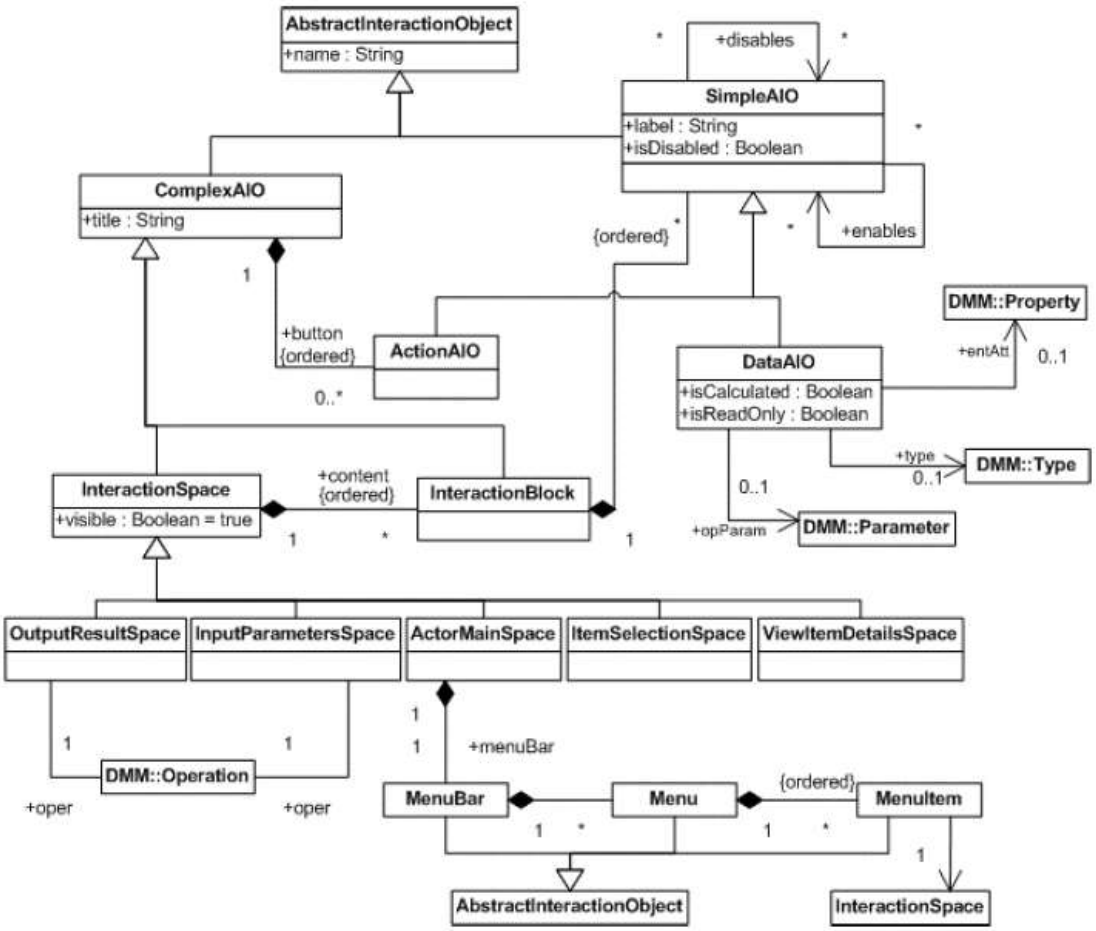


Figura 13 - *Abstract User Interface Metamodel* (retirado de [3])

Todos os elementos apresentados no metamodelo (Figura 13) derivam do *Abstract Interaction Object* (AIO). Os AIOs podem ser divididos em duas categorias: *ComplexAIO* e *SimpleAIO.*

- *Complex AIO* é um elemento que contém outros elementos;
- *SimpleAIO* é um elemento simples que é normalmente utilizado dentro de um *ComplexAIO*.

Tendo em conta as categorias anteriores, o AUIMM apresenta quatro elementos fundamentais: *DataAIO, ActionAIO, InteractionBlock* e *InteractionSpace* [3].

- *InteractionSpace (ComplexAIO)* é um objeto abstrato que representa uma área da interface onde existe interação com o utilizador. Um *InteractionSpace* é composto por *InteractionBlocks* e pode ser diretamente instanciado ou especializado como:

- ✓ *ActorMainSpace* – área onde é possível efetuar a navegação para outros espaços (menu). Tem como objetivo ser o ponto de entrada na aplicação para o utilizador;

- ✓ *InputParametersSpace* – área cujo objetivo é receber os valores utilizados como parâmetros nas operações;

- ✓ *OutputResultSpace* – área cujo objetivo é apresentar ao utilizador o resultado de uma operação;

- ✓ *ItemSelectionSpace* – área que pode incluir blocos do tipo *ViewList* (ou *ViewRelatedList)*;

- ✓ *ViewItemDetailsSpace* – área que deve incluir um bloco do tipo *ViewEntity* cujos *DataAIOs* são apenas de leitura.

- *InteractionBlock (ComplexAIO)* é uma área que contém *SimpleAIOs.* Tal como o *InteractionSpace*, o *InteractionBlock* pode ser diretamente instanciado ou especializado como:

  - ✓ *ViewEntity Block* - é um espaço associado a uma Entidade do *Domain Model,* que pode conter *DataAIOs.* É utilizada para a entrada, apresentação e edição de dados;

  - ✓ *ViewList Block* - é um espaço associado a uma Entidade do *Domain Model,* que pode conter *DataAIOs.* É utilizada para apresentar colunas (só de saída) com uma lista de atributos;

  - ✓ *ViewRelatedEntity Block* – é um espaço semelhante ao *ViewEntity* mas que deverá estar contido num *InteractionSpace* que deverá também conter um bloco do tipo *ViewEntity.* Este, por sua vez, deverá estar associado a uma Entidade que tenha uma relação ..1 (para um) com a Entidade relacionada com a *ViewRelatedEntity;*

  - ✓ *ViewRelatedList Block* - é um espaço semelhante ao *ViewList* mas que deverá estar contido num *InteractionSpace* que deverá também conter um bloco do tipo *ViewEntity.* Este, por sua vez, deverá estar associado a uma Entidade que tenha uma relação ..* (para muitos) com a Entidade relacionada com a *ViewRelatedList.*

- *DataAIO (SimpleAIO),* são normalmente associados aos Atributos das Entidades presentes no *Domain Model*. Contudo, podem também ser associados a parâmetros de operações e aos *Type;*

- *ActionAIO (SimpleAIO),* podem ser *CallDomainOperations, CRUD* ou definidas pelo utilizador, operações de navegação (permitem navegação entre InteractionSpaces), ou operações de Interface com o utilizador (permitem acionar elementos da interface com o utilizador) (Figura 14).

  Assim sendo, uma *CallDomainOperation* pode ser:

  - ✓ *CallUserDefinedOp* associada a uma operação definida pelo utilizador no *Domain Model;*
  - ✓ Especialização da meta-classe abstrata *CallCRUDOp:*
    - *CallCreateOp,* associada à operação *CreateOp* do DMM;
    - *CallRetriveOp,* assocada à operação *RetrieveOp* do DMM;
    - *CallUpdateOp*, associada à operação *UpdateOp* do DMM;
    - *CallDeleteOp,* associado à operação *DeleteOp* do DMM;
    - *CallLinkOp,* associada à operação *UpdateOp* do DMM;
    - *CallUnlinkOp,* associada à operação *UpdateOp* do DMM.

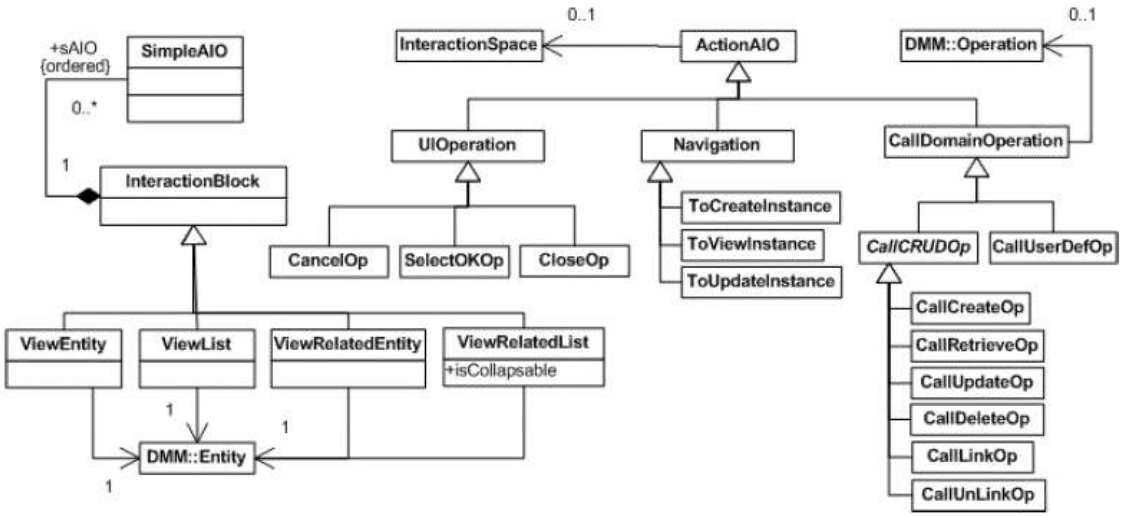


Figura 14 - AUIMM – Relações do *InteractionBlock* e *subtree* e *subtree* de *ActionAIOs* (retirado de [3])

4.2 LINGUAGEM *xCAP*

Tendo em conta os elementos do AUIMM (Figuras 13 e 14) e a linguagem CAP apresentada no Capítulo 3, em [3], [4] e [5] foi sugerida uma correspondência entre os elementos definidos em ambas abordagens. Esta proposta permite assim associar a integração com a UML proporcionada pelo UIMM aos símbolos gráficos proporcionados pela CAP. Desta correlação surge assim uma nova linguagem de modelação de *User Interfaces*, a *xCAP*.

A *xCAP* utiliza as designações dos elementos do UIMM e os símbolos gráficos da CAP criando assim uma nova linguagem. Sendo os elementos do UIMM a base desta abordagem, optou-se por fundamentá-la nos seus três elementos principais: *InteractionSpace (*e *InteractionBlock), DataAIO* e *ActionAIO.* As tabelas 2, 3 e 4 apresentam os componentes da linguagem *xCAP* divididos nas três categorias mencionadas.

Tabela 2 - Linguagem xCAP *(UIMM - InteractionSpace)*

xCAP			
Designação (UIMM)			**Símbolo (CAP)**
InteractionSpace	---	*InteractionSpace*	*(Container)*
	InteractionBlock	*ViewEntity*	*(Container)*
		ViewRelatedEntity	*(Container)*
		ViewList	*(Collection)*
		ViewRelatedList	*(Collection)*
		ViewList	*(Selectable Collection)*
		ViewRelatedList	*(Selectable Collection)*
	Menus	*MenuBar*	*(Menu)*
		Menu	*(Selectable Action Set)*
		MenuItem	*(Start Go To)*

Tabela 3 - Linguagem xCAP *(UIMM - DataAIO)*

xCAP		
Designação (UIMM)		**Símbolo (CAP)**
DataAIO	*InputOnly*	 *(Input Accepter)*
	Editting	 *(Editable Element)*
	OutputOnly	 *(Element)*
	System Message (OutputResultSpace)	 *(Notification)*

Tabela 4 - Linguagem xCAP *(UIMM - ActionAIO)*

xCAP				
Designação (UIMM)				**Símbolo (CAP)**
ActionAIO	*UI Operation*		*CancelOp*	 *(Stop End Complete)*
			CloseOp	 *(Stop End Complete)*
			SelectOKOp	 *(Select)*
	Navigation		*ToCreateInstance*	 *(Start Go To)*
			ToUpdateInstance	 *(Start Go To)*
			ToViewInstance	 *(View)*
	Domain Operations	*CallCRUDOp*	*CallCreateOp*	 *(Create)*
			CallUpdateOp	 *(Modify)*
			CallDeleteOp	 *(Delete Erase)*
			CallRetrieveOp	 *(View)*
			CallLinkOp	 *(Modify)*
			CallUnlinkOp	 *(Modify)*
		---	*CallOp (User Defined)*	 *(Perform)*

Os exemplos seguintes ilustram a aplicação da linguagem *xCAP* na criação de dois *InteractionSpaces* distintos contendo respetivamente um *InteractionBlock* do tipo *ViewList* (Figura 15) e um *InteractionBlock* do tipo *ViewRelatedEntity* (Figura 16).

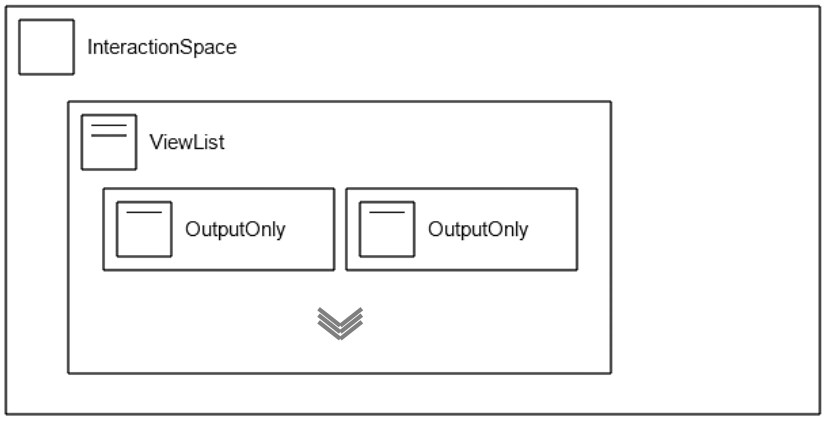


Figura 15 - *xCAP User Interface Model* (exemplo 1)

Respeitando as restrições impostas pelo UIMM, o modelo apresentado na Figura 15 representa um *InteractionSpace* contendo um *InteractionBlock* do tipo *ViewList* que por sua vez inclui dois *DataAIOs* do tipo *OutputOnly*. A figura seguinte (Figura 16), mostra um *InteractionSpace* que contém dois *InteractionBlocks, um* do tipo *ViewEntity* e outro do tipo *ViewRelatedList*.

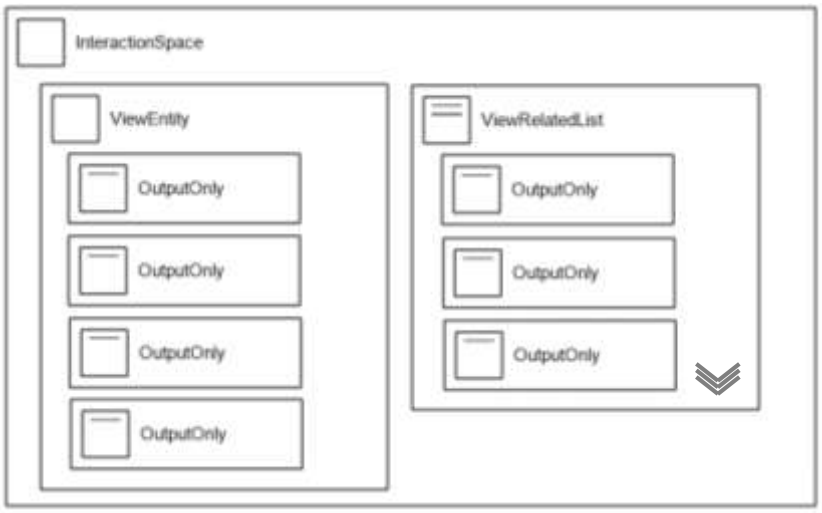


Figura 16 - *xCAP User Interface Model* (exemplo 2)

Em ambas representações a linguagem *xCAP* permite ilustrar de forma clara as relações e restrições entre os diferentes elementos propostos no UIMM. Para além disso, a possibilidade de associação às classes de um Modelo de Domínio faz com que os Modelos de Interface de Utilizador, apesar de abstratos, criem uma apresentação gráfica legível, e mais que isso compreensível, para o seu leitor. Tendo em conta o

InteractionSpace apresentado no exemplo 2 (Figura 16) e as restrições associadas aos *InteractionBlocks* que o compõem, nomeadamente:

- *ViewEntity* - espaço associado a uma Entidade do *Domain Model,* que pode conter *DataAIOs.* É utilizada para a entrada, apresentação e edição de dados;

- *ViewRelatedList* - é um espaço semelhante ao *ViewList* mas que deverá estar contido num *InteractionSpace* que deverá também conter um bloco do tipo *ViewEntity.* Este, por sua vez, deverá estar associado a uma Entidade que tenha uma relação ..* (para muitos) com a Entidade relacionada com a *ViewRelatedList.*

É fácil tirar ilações acerca do modelo apresentado, nomeadamente, das Entidades que os *InteractionBlocks* representam, do tipo de relação que as associa (*ViewEntity* ..* *ViewRelatedList*) e dos atributos de ambas. Supondo que a Entidade relacionada com o *InteractionBlock ViewEntity* é designada por Cliente e a Entidade Relacionada com o *InteractionBlock ViewRelatedList* é designada por Aluguer, a Figura 17 poderá representar o Modelo de Domínio ao qual o Modelo *xCAP* está associado.

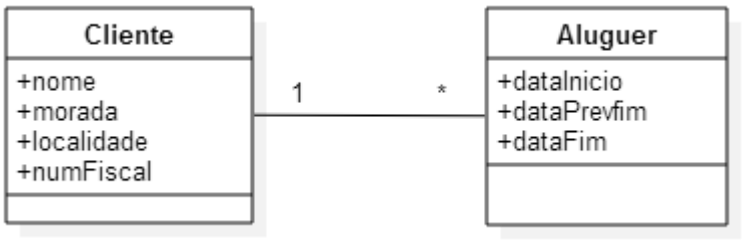


Figura 17 - Entidades representadas no exemplo 2

De uma forma mais concreta um modelo *xCAP* relativo ao Modelo de Domínio apresentado na Figura 17 poderia ser representado da seguinte forma:

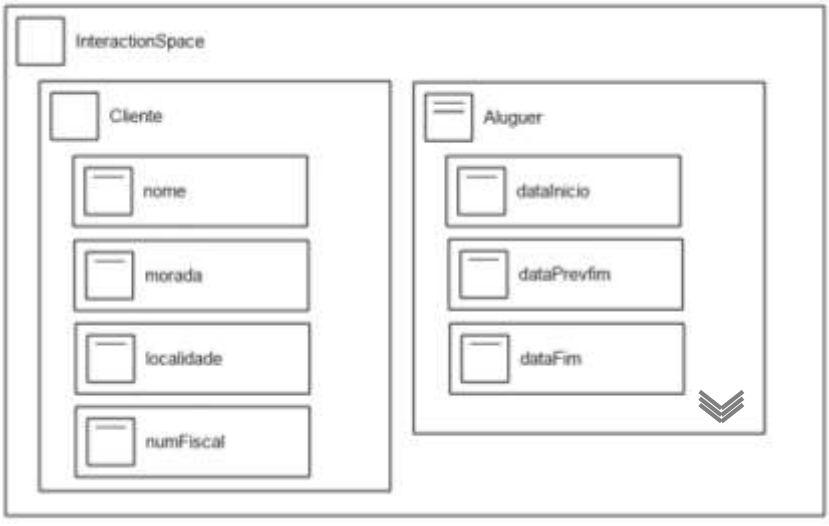


Figura 18 - Aplicação concreta do *xCAP*

4.3 SUMÁRIO

Este capítulo teve como ponto inicial a apresentação do *Abstract User Interface Metamodel* (AUIMM), dos seus elementos e respetivas restrições, e a sua relação com os elementos do *Domain Metamodel* (DMM). Partindo do AUIMM e recorrendo aos símbolos gráficos utilizados nos *Canonical Abstract Prototypes* (CAP) (apresentados com mais detalhe no Capitulo 2), foi formulada uma nova linguagem concreta para a criação de Modelos de UI designada de *xCAP*. Finalmente de modo a validar a linguagem, foram apresentados dois exemplos de Modelos de UI criados recorrendo à *xCAP*. A aplicação da linguagem na criação de Modelos de UI e a sua relação com outros modelos utilizados no desenvolvimento de sistemas será abordada com mais detalhe em capítulos posteriores.

5. IMPLEMENTAÇÃO DO TRABALHO

Este capítulo tem como objetivo apresentar a arquitetura do editor *web MetaCAP* para a linguagem *xCAP*,a sua *interface* e as ferramentas utilizadas no seu desenvolvimento.

5.1 ARQUITETURA DA APLICAÇÃO

A arquitetura do editor *MetaCAP* é composta por três componentes essenciais:

- *Componentes (componentes.js)* – documento onde são desenhados os diferentes elementos gráficos da linguagem *xCAP*;
- *Aplicação (aplicação.js)* – documento utilizado para criar a interação entre a interface com utilizador e o ficheiro responsável pelo desenho dos componentes;
- *Index (índex.html)* – documento responsável pelo aspeto da *interface* da aplicação.

Figura 19 - Arquitetura *MetaCAP*

Em termos práticos, sempre que o utilizador adiciona um elemento da linguagem *xCAP* ao Modelo da UI *(index.html)* o evento despoletado cria uma nova instância do elemento da linguagem *xCAP (aplicacao.js)* que é desenhada *(componentes.js)* e efetua o caminho inverso até ser apresentada no elemento *canvas* da aplicação *(index.html)*, como ilustrado na Figura 19.

O elemento canvas é um bitmap de modo imediato (immediate mode), que fornece uma superfície que permite renderizar elementos gráficos dinamicamente orquestrados através de JavaScript. Um elemento canvas adicionado à markup de uma página faz sempre parte da árvore do DOM gerada a partir do parsing da página HTML [42].

5.2 Interface da Aplicação

A *interface* do editor *MetaCAP* (Figura 20) está dividida em quatro partes:

- *Área de trabalho* – parte central da aplicação destinada à construção do Modelo da UI;
- *Menu "MetaCAP"* – menu reservado para as funcionalidades relacionadas com a criação de um novo modelo, importação e exportação dos modelos (XML), importação do Modelo de Domínio para associação ao modelo *xCAP*;
- *Menu "Elementos"* – menu destinado à apresentação dos elementos da linguagem *xCAP* e à sua divisão por categorias (*InteractionSpace*, *DataAIO*, *ActionAIO*);
- *Menu "Propriedades"* – menu utilizado para alterar as propriedades de cada elemento da linguagem *xCAP*, nomeadamente: texto, comprimento e largura, posição, associação de uma classe e/ou atributos do Modelo de Domínio.

Figura 20 - Interface do editor *MetaCAP*

5.3 Ferramentas utilizadas

Para o desenvolvimento do editor *web MetaCAP* recorreu-se à linguagem HTML5 para a criação da estrutura base da página e à *framework paper.js* para a criação dos elementos gráficos da linguagem *xCAP*.

A *paper.js* é uma *framework javascript open source* de *scripts* de gráficos vetoriais que corre no elemento *canvas* do HTML5. Oferece a estrutura *Scene Graph* (que funciona como *Document Object Model*) e outras funcionalidades que permitem entre outros trabalhar com gráficos vetoriais e curvas de *Bézier*. Sendo baseada e compatível com o

ambiente *Scriptographer* [25] herdou algumas das suas caraterísticas nomeadamente a possibilidade de controlar a criação do desenho recorrendo ao rato, aplicar efeitos de transformação a elementos gráficos já existentes, utilizar o teclado para movimentar objetos, etc.

Para a manipulação de vetores e elementos geométricos tais como: *Point, Size* e *Rectangle*; que estão no cerne da *framework,* recorre ao *PaperScript*, uma extensão do *JavaScript* criada com o objetivo de simplificar as operações matemáticas associadas à representação dos objetos supracitados [43].

De entre as funcionalidades proporcionadas podem-se destacar as seguintes:

- Importar/exportar projetos nos formatos JSON e SVG;
- Permitir a utilização de operações lógicas como a união, intersecção e exclusão aplicadas aos objetos criados;
- Fomentar a interatividade dos projetos com as funcionalidades *HitTest* [26] (aplicada aos elementos gráficos), e *Event Handlers* (associados à utilização do rato ou do teclado).

ESTRUTURA HIERÁRQUICA DE UM PROJETO

A estrutura de um projeto *paper.js* baseia-se no mesmo princípio apresentado por alguns dos *softwares* de *design* gráfico mais conhecidos como o *Adobe Photoshop, Adobe Illustrator*, etc. Tal como nessas aplicações*,* também o *paper.js* apresenta uma lista de *layers* - *project.layers*. Sempre que uma vista é redesenhada o projeto *paper.js* percorre as diferentes *layers* e desenha os *items* tendo em conta a ordem do seu aparecimento.

Quando um projeto é criado é lhe atribuída automaticamente uma nova *layer*, a *active layer – project.activeLayer.* Todos os novos *items* criados são adicionados, como *children*, à *layer* que está ativa. Os *children* de uma *layer,* armazenados sob a forma de um *array*, podem ser acedidos recorrendo a *items.children.*

De forma a agrupar vários *items* de uma *layer* existe um objeto intermédio, o *group*. Um *group* permite que a coleção de *items* que lhe está associada seja tratada como sendo uma só unidade. Quando um *group* é transformado todos os *items* que o compõe são também transformados sem nunca ser afetada a sua posição relativa individual.

Apesar dos elementos *layer* e *group* serem semelhantes, existe uma caraterística que os diferencia: uma *layer* pode ser ativada, ou seja, qualquer item que é criado é automaticamente posicionado na *layer*; no caso do *group*, quando é criado não possui qualquer item *(children*), ou seja, os *items* têm de lhe ser atribuídos. A forma mais

comum de atribuição é a passagem de um *array* de *children* ao *group* recorrendo ao seu construtor – *new Group ([children]).*

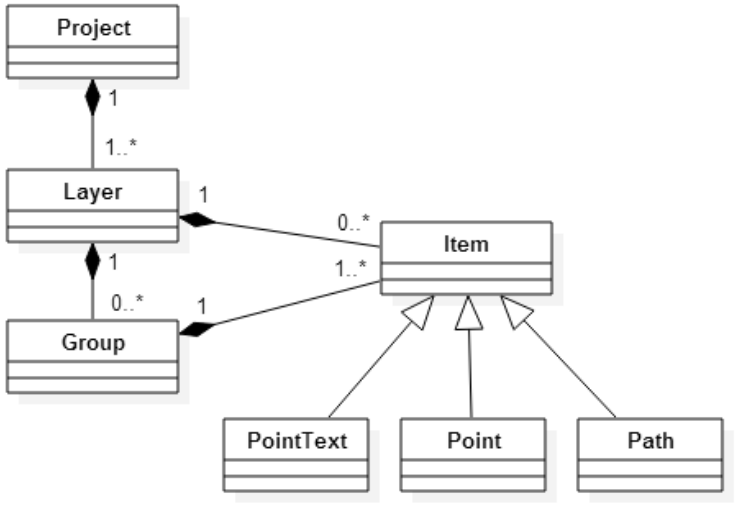


Figura 21 - Estrutura de um projeto paper.js

ITEMS UTILIZADOS NA CRIAÇÃO DOS SÍMBOLOS DA LINGUAGEM XCAP (METACAP)

Na criação dos elementos gráficos da linguagem *xCAP* foram utilizados os seguintes *items* da *framework paper.js* [43].

Point

O objeto *Point* (ponto) representa um ponto no espaço bidimensional de um projeto. Pode-se também recorrer ao *Point* para representar vetores bidimensionais. Um *Point* pode receber como argumentos no seu construtor as coordenadas *x* e *y*.

Point(x, y)

Exemplo: criar um ponto nas coordenadas x= 5, y= 10;

var point = new Point(5, 10);

PointText

Um item do tipo *PointText* representa um conjunto de carateres a partir de um ponto *(Point)* inicial definido pelo programador. Assim sendo, uma das formas de inicializar um objeto do tipo *PointText* é atribuir como argumento no seu construtor a sua posição inicial.

PointText(point)

Exemplo: Escrever um conjunto de carateres a partir do ponto (100, 150);

```
var text = new PointText(new Point(100, 150));
text.content = 'Conjunto de Carateres'; //Carateres a serem representados
```

Path

Um *item* do tipo *Path* representa uma linha num projeto. Tratando-se de um segmento de reta, uma das formas de instanciar um novo *Path* é atribuir-lhe como atributos os dois pontos que o delimitam sob a forma de um *segment*.

Path([segments])

Exemplo: Criar uma linha entre os pontos (50, 40) e (100, 40);

```
var path = new Path();
path.strokeColor = 'black';          // Cor da linha
path.add(new Point(50, 40));
path.add(new Point(100, 40));
```

De modo a representar as figuras geométricas mais comuns, associados ao *item Path* podem, entre outros, ser criados os objetos *Circle, Rectangle, RegularPolygon, Star*, etc; designados *Shaped Paths* (linhas com formas). Os *Shaped Paths*, dependendo do seu tipo, recebem como argumentos: ponto inicial/central, tamanho, número de lados, raio, etc.

Path.Circle(center point, radius)

Exemplo: Criar um circulo com centro em (150, 150) e raio 35;

```
var path = new Path.Circle(new Point(150, 150), 35);
path.strokeColor = 'black';          // Cor da Linha
```

Path.Rectangle(point, size)

Exemplo: Criar um retângulo com início no ponto (50, 60) e com tamanho *(Size)* 80x90;

```
var point = new Point(50, 60);
var size = new Size(80, 90);          // Tamanho
var path = new Path.Rectangle(point, size);
path.strokeColor = 'black';          // Cor da linha
```

Path.RegularPolygon(center point, sides, radius)

Exemplo: Criar um triângulo com centro no ponto (100, 100) e com raio igual a 50;

```
var center = new Point(100, 100);
var sides = 3;       // Número de lados
var radius = 40;   // Raio
var triangle = new Path.RegularPolygon(center, sides, radius);
```

```
triangle.strokeColor = 'black'; // Cor da Linha
```

Path.Star(center point, points, radius1, radius2)

Exemplo: Criar uma estrela com centro no ponto (50, 50), com 5 pontas, com raio interno igual a 30 e raio externo igual a 50;

```
var center = new Point(50, 50);
var points = 5;     // Número de pontas
var radius1 = 30; // Raio interno
var radius2 = 50; // Raio externo
var path = new Path.Star(center, points, radius1, radius2);
path.strokeColor = 'black';             // Cor da linha
```

Tendo em conta que todos os objetos apresentados são independentes, e que para criar cada um dos símbolos gráficos da linguagem *xCAP* foram utilizados vários *items,* estes tiveram de ser agrupados em objetos do tipo *group* de forma a criarem um objeto único.

5.4 SUMÁRIO

Este capítulo pretendeu dar a conhecer a aplicação *MetaCAP* concebida com o objetivo de permitir a criação de Modelos de UI baseados na linguagem *xCAP*. A aplicação, desenvolvida para o ambiente *web*, conta com uma arquitetura simples baseada no elemento canvas (HTML5) e na utilização da *framework Javascript paper.js.* Para além de permitir a conjunção dos elementos gráficos propostos pela linguagem *xCAP*, a aplicação permite ainda a associação destes elementos a *Classes* e *Atributos* de um Modelo de Domínio obtido através da importação de um ficheiro XML projetado na aplicação *AMALIA IDE.*

6. CASO DE ESTUDO

Este capítulo tem como objetivo desenvolver um modelo da UI (composto por vários *InteractionSpaces*) recorrendo à aplicação *MetaCAP* e tendo como base um Modelo de Domínio e um Modelo de Casos de Uso criados recorrendo ao *AMALIA IDE*.

6.1 SISTEMA - EXEMPLO *"RENT – A - CAR"*

O sistema em estudo, representado na Figura 22, apresenta um exemplo aplicável a um pequeno negócio de aluguer de automóveis. Assim sendo, da leitura do Modelo de Domínio pode retirar-se o seguinte:

- É composto por três Entidades (classes): Cliente, Aluguer e Viatura;
- Cada Cliente pode fazer vários Alugueres;
- Cada Viatura pode ser Alugada várias vezes;

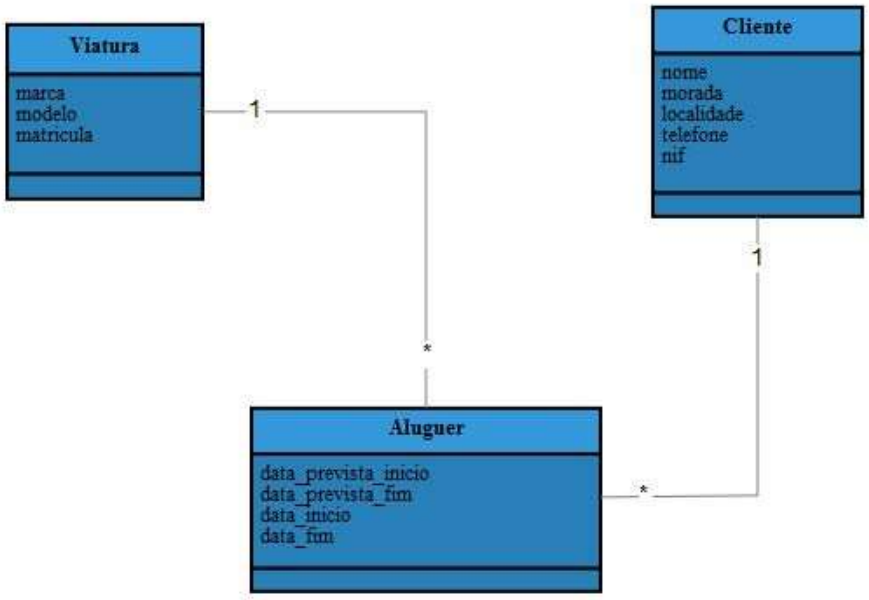


Figura 22 - Modelo de Domínio exemplo *Rent-A-Car*

Se o Modelo de Domínio (Figura 22) constitui uma descrição das propriedades comuns e variáveis do domínio relacionado com o sistema de *software* que está a ser desenvolvido, representando no exemplo a estrutura do sistema e o relacionamento entre as várias Classes que o compõe, o Modelo de Casos de Uso (Figura 23) tem como finalidade a especificação de requisitos com o objetivo de auxiliar a análise e gestão dos mesmos.

O conjunto de Casos de Uso (cenário) representa todas as possíveis interações suportadas pelo sistema. Normalmente, um requisito funcional é modelado por um Caso de Uso. Ou seja, os Casos de Uso representam o comportamento que o sistema deve

suportar sendo que cada um corresponde a uma sequência de ações realizadas entre um utilizador (ator) e o sistema numa determinada altura [44].

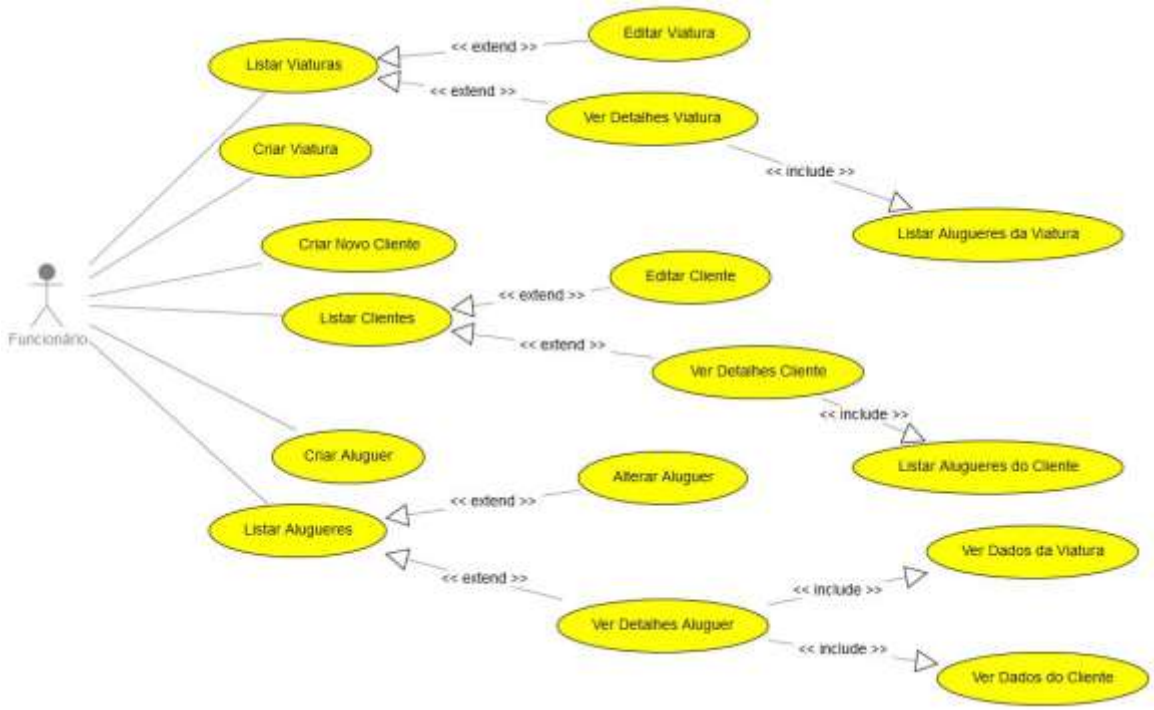


Figura 23 - Modelo de Casos de Uso exemplo *Rent-A-Car*

Desta forma, parte-se do pressuposto que cada Caso de Uso independente (sem relações de extensão ou inclusão) vai corresponder a um *InteractionSpace* (p. ex: *Criar Viatura*). O mesmo acontece quando o Caso de Uso faz parte de uma relação de tipo *<<extend>>* (p. ex: *Editar Viatura <<extend>> Listar Viaturas*). Nesta situação, o segundo Caso de Uso pode ser acrescentado para descrever o comportamento do primeiro, não sendo contudo essencial [44], razão pela qual é representado num *InteractionSpace* independente. Por último, quando um Caso de Uso possui uma relação de tipo *<<include>>* com outro Caso de Uso (p. ex: *Ver Detalhes da Viatura <<include>> Listar Alugueres da Viatura*), pressupõe-se que o segundo Caso de Uso faz parte do primeiro [28] pelo que da conjunção de ambos surgirá apenas um *InteractionSpace*.

6.2 Criação do Modelo da UI - Exemplo *"Rent – A - Car"*

Para que o Modelo da UI criado seja fiel ao Modelo de Domínio ao qual se refere, o *MetaCAP* permite importar o ficheiro XML representativo do sistema (gerado recorrendo ao *AMALIA IDE*) e que vai fornecer informação sobre as Entidades (classes) e respetivos atributos.

De forma a simplificar a apresentação do sistema e leitura do Modelo de UI, o exemplo anterior (Figura 23) será dividido em três cenários distintos: Viatura, Cliente e Aluguer.

Para cada um dos cenários propostos cabe analisar o número de Casos de Uso e as relações de generalização (<<*include*>> e <<*extend*>>) entre eles.

6.2.1 CENÁRIO VIATURA

O primeiro cenário em estudo (Figura 24) envolve na especificação de requisitos as Entidades (classes) Viatura e Aluguer (Figura 22).

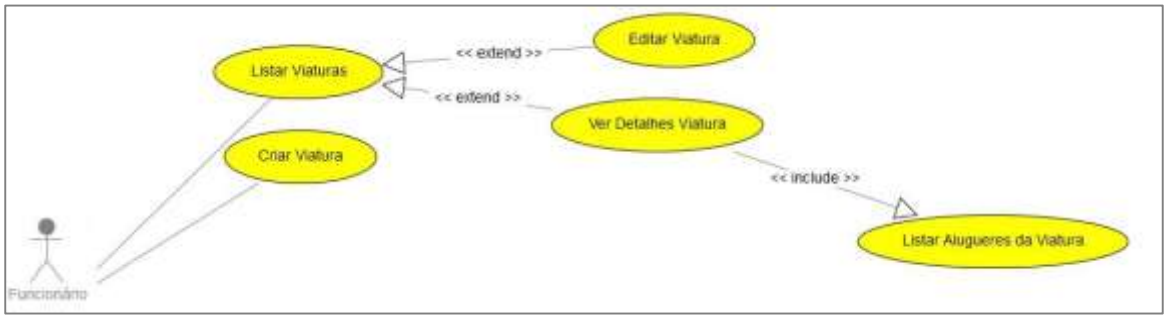


Figura 24 - Cenário *Viatura*

Assim sendo, tendo em conta o exposto no ponto anterior, o *Cenário Viatura* vai dar lugar a quatro *InteractionSpaces*[1]:

- *Criar Viatura;*
- *Listar Viaturas;*
- *Editar Viatura;*
- *Ver Detalhes Viatura (+ Listar Alugueres da Viatura).*

O primeiro *InteractionSpace*, *Criar Viatura* (Figura 25), tem como objetivo adicionar uma nova viatura ao sistema, ou seja, deverá permitir a criação de uma nova instância da classe Viatura. É composto por:

- Um *InteractionBlock* do tipo *ViewEntity* (Viatura) que por sua vez contém três *DataAIOs* do tipo *InputOnly* (marca, modelo e matrícula);
- Um *ActionAIO* do tipo *CallCreateOp* (Criar Nova Viatura) que permite executar a operação *Create* associada à nova viatura.

[1] Os *InteractionSpaces* apresentados limitam-se a representar os Casos de Uso presentes em cada cenário bem como as operações explícitas nos mesmos. As operações e elementos implícitos (menus, fechar janela, etc.) não fazem parte dos modelos apresentados.

Figura 25 - *InteractionSpace Criar Viatura*

O *InteractionSpace*, *Listar Viaturas* (Figura 26), tem como objetivo apresentar uma lista contendo as viaturas presentes no sistema, ou seja, deverá listar todas as instâncias da classe Viatura. É composto por:

- Um *InteractionBlock* do tipo *ViewList* (Viatura) que por sua vez contém três *DataAIOs* do tipo *OutputOnly* (marca, modelo e matrícula);
- Um *ActionAIO* do tipo *ToViewInstance* (Ver Detalhes Viatura) que permite a navegação até um novo *InteractionSpace* (*Ver Detalhes Viatura*, Figura 28);
- Um *ActionAIO* do tipo *ToUpdateInstance* (Editar Viatura) que permite a navegação até um novo *InteractionSpace* (*Editar Viatura*, Figura 27).

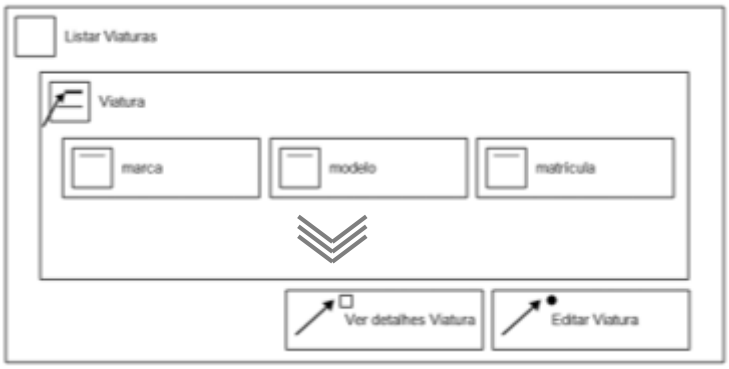


Figura 26 - *InteractionSpace Listar Viaturas*

Tal como ficou implícito previamente, o *InteractionSpace* anterior representa o Caso de Uso *Listar Viaturas* presente no cenário *Viatura* (Figura 24). Este Caso de Uso apresenta duas relações de generalização do tipo *<<extends>>* com os Casos de Uso *Ver Detalhes Viatura* e *Editar Viatura* pelo que o *InteractionSpace* deve também representar estas relações. A utilização dos *ActionAIOs ToViewInstance* (Ver Detalhes Viatura) e *ToUpdateInstance* (Editar Viatura) permite assim representar o relacionamento explícito no cenário *Viatura* (Figura 24) de forma implícita, ou seja, indicando a navegação/relação entre *InteractionSpaces*.

O *InteractionSpace*, *Editar Viatura* (Figura 27), tem como objetivo editar uma viatura pertencente ao sistema, ou seja, deverá permitir fazer alterações a uma instância da classe Viatura. É composto por:

- Um *InteractionBlock* do tipo *ViewEntity* (Viatura) que por sua vez contém três *DataAIOs* do tipo *Editting* (marca, modelo e matrícula);
- Um *ActionAIO* do tipo *CallUpdateOp* (Alterar Viatura) que permite executar a operação *Update* associada à viatura selecionada.

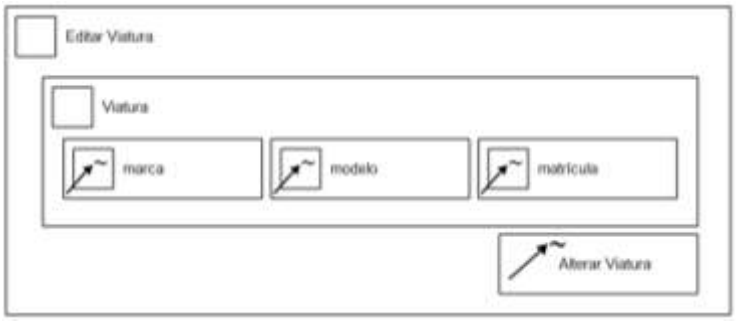


Figura 27 - *InteractionSpace Editar Viatura*

O *InteractionSpace* seguinte *Ver Detalhes Viatura* (Figura 28) apresenta diferenças em relação aos *InteractionSpaces* representados nas figuras anteriores (25, 26 e 27) na medida em que é o único que explicitamente representa uma relação de generalização do tipo *<<include>>* na sua estrutura. No *Cenário Viatura* (Figura 24) pode ler-se *Ver Detalhes Viatura <<include>> Listar Alugueres da Viatura,* pelo que o *InteractionSpace Ver Detalhes Viatura* deverá incluir o *InteractionBlock Listar Alugueres da Viatura.* É composto por:

- Um *InteractionBlock* do tipo *ViewEntity* (Viatura) que por sua vez contém três *DataAIOs* do tipo *OutputOnly* (marca, modelo e matrícula);
- Um *InteractionBlock* do tipo *ViewRelatedList* (Listar Alugueres da Viatura) que por sua vez contém quatro *DataAIOs* do tipo *OutputOnly* (data prevista inicio, data prevista fim, data inicio e data fim).

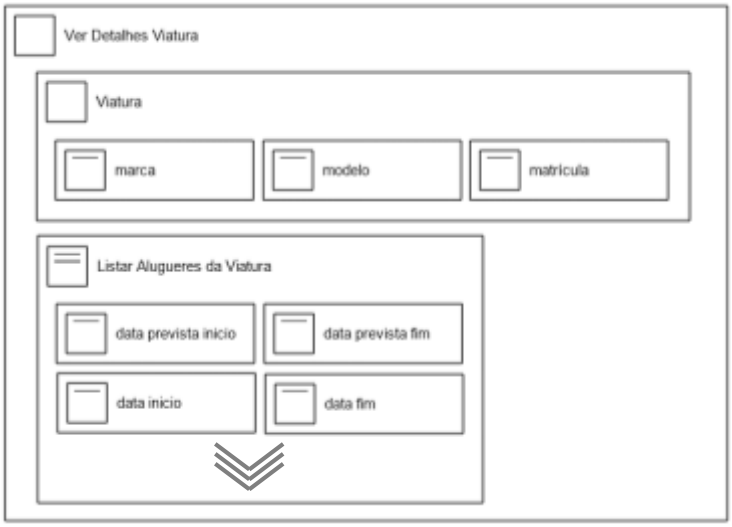


Figura 28 - *InteractionSpace Ver Detalhes Viatura (+ Listar Alugueres Viatura)*

6.2.2 CENÁRIO CLIENTE

O segundo cenário em estudo (Figura 26) envolve na especificação de requisitos as Entidades (classes) Cliente e Aluguer (Figura 22).

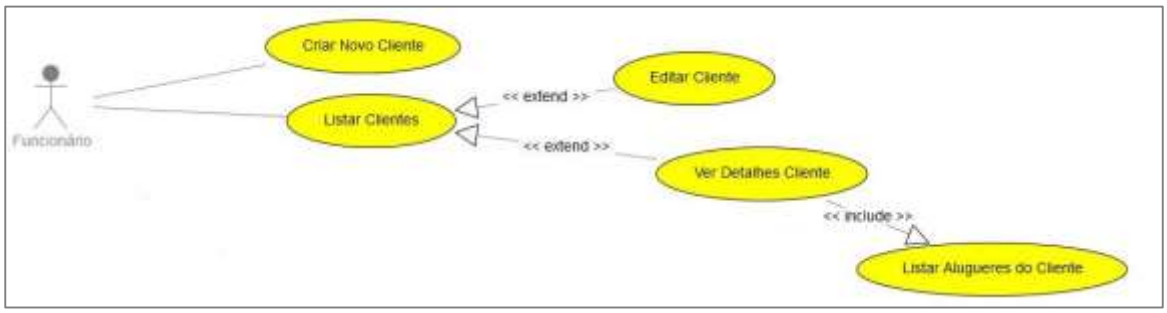


Figura 29 - Cenário *Cliente*

Analogamente ao que ocorreu no cenário anterior, o *Cenário Cliente* vai também dar lugar a quatro *InteractionSpaces*[2]:

- *Criar Novo Cliente;*
- *Listar Clientes;*
- *Editar Cliente;*
- *Ver Detalhes Cliente (+ Listar Alugueres do Cliente).*

O primeiro *InteractionSpace*, *Criar Novo Cliente* (Figura 30), tem como objetivo adicionar um novo cliente ao sistema, ou seja, deverá permitir a criação de uma nova instância da classe Cliente.

O *InteractionSpace Criar Novo Cliente* é composto por:

- Um *InteractionBlock* do tipo *ViewEntity* (Cliente) que por sua vez contém cinco *DataAIOs* do tipo *InputOnly* (nome, morada, localidade, telefone e nif);
- Um *ActionAIO* do tipo *CallCreateOp* (Criar Novo Cliente) que permite executar a operação *Create* associada ao novo cliente.

[2] Os *InteractionSpaces* apresentados limitam-se a representar os Casos de Uso presentes em cada cenário bem como as operações explícitas nos mesmos. As operações e elementos implícitos (menus, fechar janela, etc.) não fazem parte dos modelos apresentados.

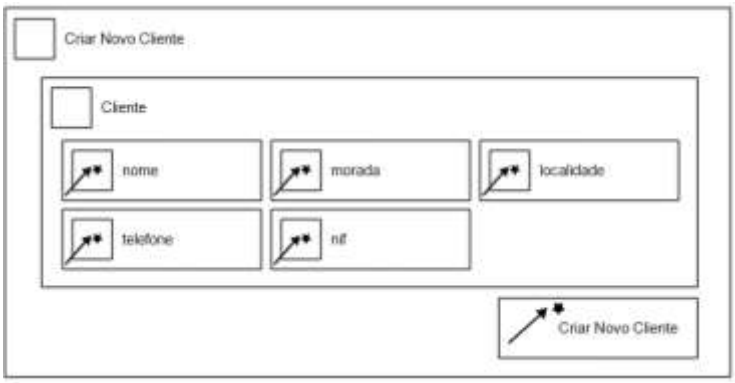


Figura 30 - *InteractionSpace Criar Novo Cliente*

O *InteractionSpace Listar Clientes* (Figura 31) tem como objetivo apresentar uma lista contendo os clientes presentes no sistema, ou seja, deverá listar todas as instâncias da classe Cliente. É composto por:

- Um *InteractionBlock* do tipo *ViewList* (Cliente) que por sua vez contém cinco *DataAIOs* do tipo *OutputOnly* (nome, morada, localidade, telefone e nif);
- Um *ActionAIO* do tipo *ToViewInstance* (Ver Detalhes Cliente) que permite a navegação até um novo *InteractionSpace* (*Ver Detalhes Cliente*, Figura 33);
- Um *ActionAIO* do tipo *ToUpdateInstance* (Editar Cliente) que permite a navegação até um novo *InteractionSpace* (*Editar Cliente*, Figura 32).

Figura 31 – *InteractionSpace Listar Clientes*

Tal como no cenário anterior, o *InteractionSpace Listar Clientes* representa o Caso de Uso com a mesma designação presente no cenário *Cliente* (Figura 29). Este Caso de Uso apresenta duas relações de generalização do tipo *<<extends>>* com os Casos de Uso *Ver Detalhes Cliente* e *Editar Cliente* pelo que o *InteractionSpace* deve também representar estas relações. A utilização dos *ActionAIOs ToViewInstance* (Ver Detalhes Cliente) e *ToUpdateInstance* (Editar Cliente) permite assim representar o relacionamento explícito no cenário *Cliente* (Figura 29) de forma implícita, ou seja, indicando a navegação/relação entre *InteractionSpaces*.

O *InteractionSpace*, *Editar Cliente* (Figura 32), tem como objetivo editar um cliente pertencente ao sistema, ou seja, deverá permitir fazer alterações a uma instância da classe Cliente. É composto por:

- Um *InteractionBlock* do tipo *ViewEntity* (Cliente) que por sua vez contém cinco *DataAIOs* do tipo *Editting* (nome, morada, localidade, telefone e nif);
- Um *ActionAIO* do tipo *CallUpdateOp* (Alterar Cliente) que permite executar a operação *Update* associada ao cliente selecionado.

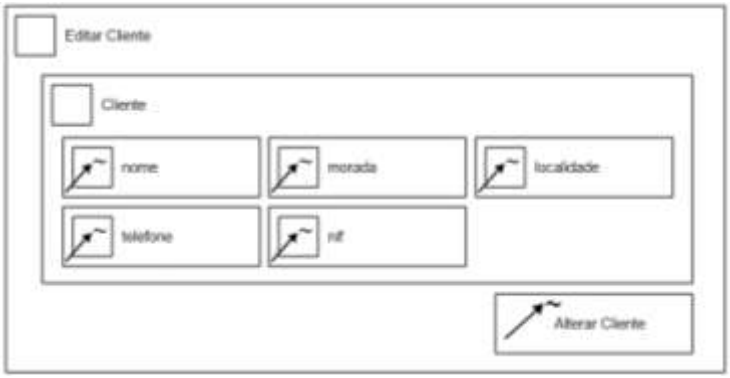


Figura 32 – *InteractionSpace Editar Cliente*

O *InteractionSpace Ver Detalhes Cliente* (Figura 33) apresenta diferenças em relação aos *InteractionSpaces* representados nas figuras anteriores (30, 31 e 32) na medida em que é o único que explicitamente representa uma relação de generalização do tipo *<<include>>* na sua estrutura. No *Cenário Cliente* (Figura 29) pode ler-se *Ver Detalhes Cliente <<include>> Listar Alugueres do Cliente,* pelo que o *InteractionSpace Ver Detalhes Cliente* deverá incluir o *InteractionBlock Listar Alugueres do Cliente.* É composto por:

- Um *InteractionBlock* do tipo *ViewEntity* (Cliente) que por sua vez contém cinco *DataAIOs* do tipo *OutputOnly* (nome, morada, localidade, telefone e nif);
- Um *InteractionBlock* do tipo *ViewRelatedList* (Listar Alugueres do Cliente) que por sua vez contém quatro *DataAIOs* do tipo *OutputOnly* (data prevista inicio, data prevista fim, data inicio e data fim).

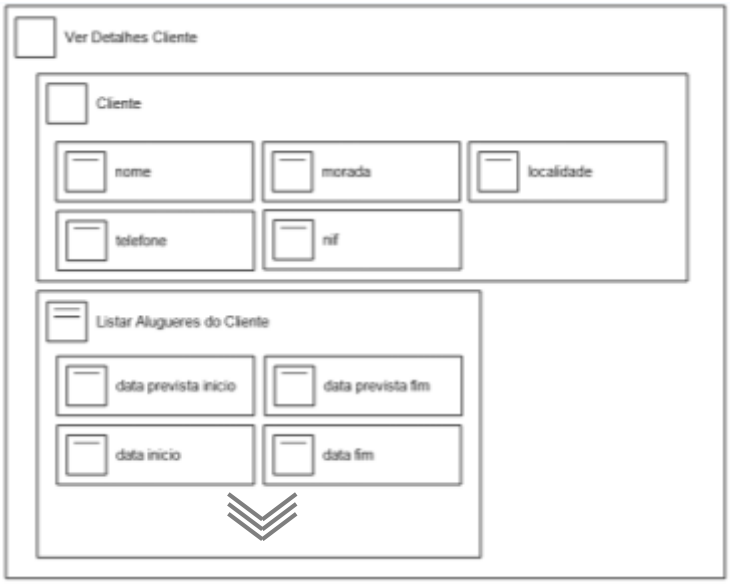


Figura 33 - *InteractionSpace Ver Detalhes Cliente (+ Listar Alugueres do Cliente)*

6.2.3 CENÁRIO ALUGUER

O terceiro cenário em estudo (Figura 29) envolve na especificação de requisitos as Entidades (classes) Cliente, Aluguer e Viatura (Figura 22).

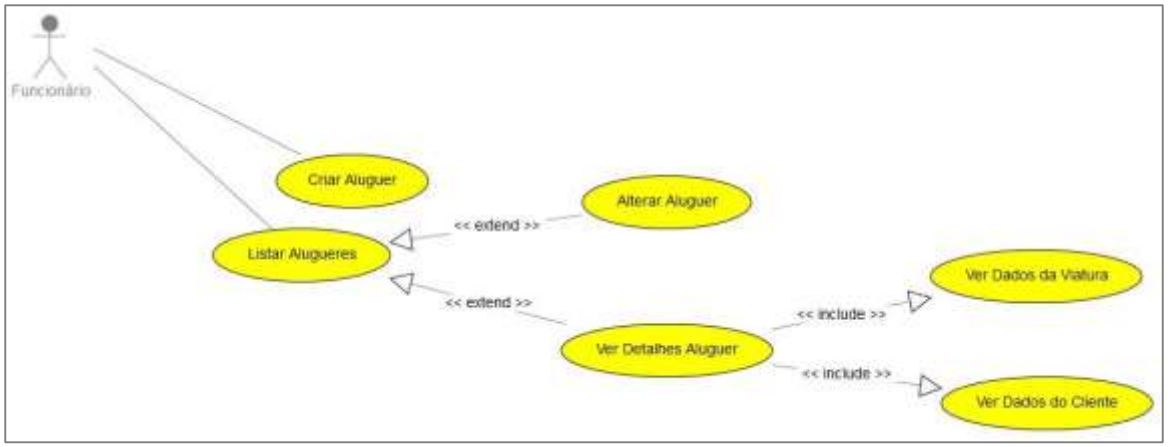


Figura 34 - Cenário *Aluguer*

Tal como os cenários anteriores, o *Cenário Aluguer* vai também dar lugar a quatro *InteractionSpaces*[3]:

- *Criar Aluguer;*
- *Listar Alugueres;*
- *Alterar Aluguer;*

[3] Os *InteractionSpaces* apresentados limitam-se a representar os Casos de Uso presentes em cada cenário bem como as operações explícitas nos mesmos. As operações e elementos implícitos (menus, fechar janela, etc.) não fazem parte dos modelos apresentados.

- *Ver Detalhes Aluguer (+ Ver Dados da Viatura + Ver Dados do Cliente).*

O primeiro *InteractionSpace*, *Criar Aluguer* (Figura 35), tem como objetivo adicionar um novo aluguer ao sistema, ou seja, deverá permitir a criação de uma nova instância da classe Aluguer. Contudo, tratando-se a Aluguer de uma classe pertencente as duas relações ..* (Figura 22), a especificação deste *InteractionSpace* apresenta algumas diferenças em relação aos *InteractionSpaces Criar…* dos cenários anteriores.

Desta forma, o *InteractionSpace Criar Aluguer* é composto por:

- Um *InteractionBlock* do tipo *ViewEntity* (Aluguer) que por sua vez contém quatro *DataAIOs* do tipo *InputOnly* (data prevista inicio, data prevista fim, data inicio e data fim);
- Dois *InteractionBlocks* do tipo *ViewRelatedList* (Cliente e Viatura);
- Um *ActionAIO* do tipo *CallCreateOp* (Criar Aluguer) que permite executar a operação *Create* associada ao novo aluguer.

A diferença registada, ou seja, a existência de dois *InteractionBlocks* do tipo *ViewRelatedList*, prende-se com o facto de que para ser criado um novo aluguer será sempre necessário associar-lhe um cliente e uma viatura.

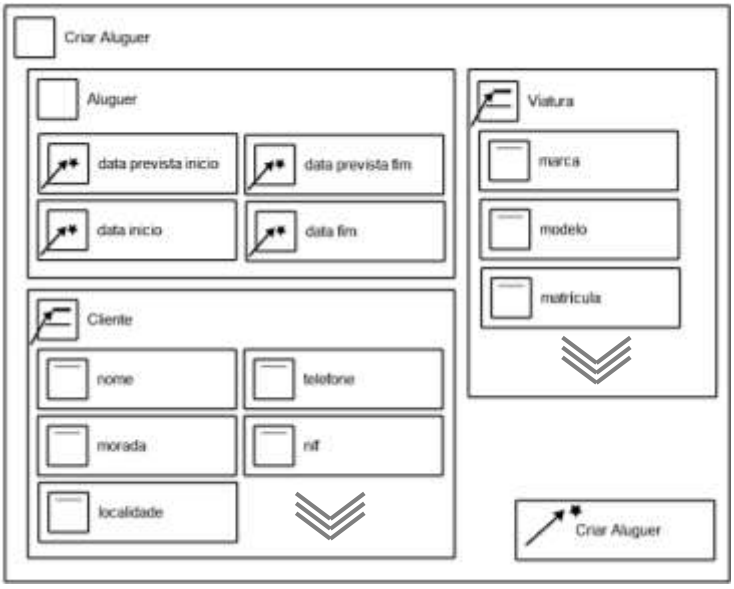


Figura 35 - *InteractionSpace Criar Aluguer*

O *InteractionSpace*, *Listar Alugueres* (Figura 36), tem como objetivo apresentar uma lista contendo os alugueres presentes no sistema, ou seja, deverá listar todas as instâncias da classe Aluguer.

O *InteractionSpace Listar Alugueres* é composto por:

- Um *InteractionBlock* do tipo *ViewList* (Aluguer) que por sua vez contém quatro *DataAIOs* do tipo *OutputOnly* (data prevista inicio, data prevista fim, data inicio e data fim);
- Um *ActionAIO* do tipo *ToViewInstance* (Ver Detalhes Aluguer) que permite a navegação até um novo *InteractionSpace* (*Ver Detalhes Aluguer*, Figura 38);
- Um *ActionAIO* do tipo *ToUpdateInstance* (Alterar Aluguer) que permite a navegação até um novo *InteractionSpace* (*Alterar Aluguer*, Figura 37).

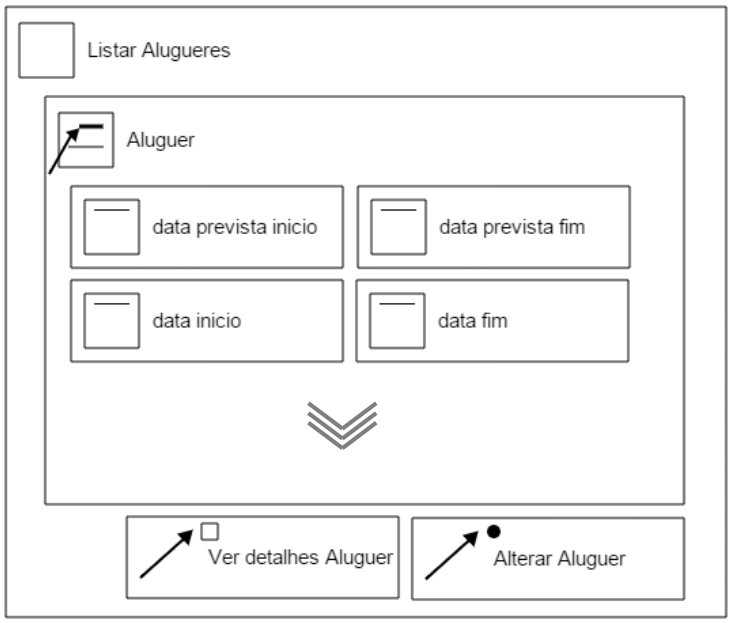


Figura 36 - *InteractionSpace Listar Alugueres*

Analogamente aos cenários anteriores, o *InteractionSpace Listar Alugueres* representa o Caso de Uso com a mesma designação presente no cenário *Aluguer* (Figura 34). Este Caso de Uso apresenta duas relações de generalização do tipo *<<extends>>* com os Casos de Uso *Ver Detalhes Aluguer* e *Alterar Aluguer* pelo que o *InteractionSpace* deve também representar estas relações. A utilização dos *ActionAIOs ToViewInstance* (Ver Detalhes Aluguer) e *ToUpdateInstance* (Alterar Aluguer) permite assim representar o relacionamento explícito no cenário *Aluguer* (Figura 34) de forma implícita, ou seja, indicando a navegação/relação entre *InteractionSpaces*.

O *InteractionSpace Alterar Aluguer* (Figura 37) tem como objetivo editar um aluguer pertencente ao sistema, ou seja, deverá permitir fazer alterações a uma instância da classe Aluguer.

O *InteractionSpace Alterar Aluguer* é composto por:

- Um *InteractionBlock* do tipo *ViewEntity* (Aluguer) que por sua vez contém quatro *DataAIOs* do tipo *Editting* (data prevista inicio, data prevista fim, data inicio e data fim);
- Dois *InteractionBlocks* do tipo *ViewRelatedList* (Cliente e Viatura);
- Um *ActionAIO* do tipo *CallUpdateOp* (Alterar Aluguer) que permite executar a operação *Update* associada ao aluguer selecionado.

Tal como se verificou no *InteractionSpace Criar Aluguer* (Figura 35), também neste caso é necessária a utilização dos *InteractionBlocks ViewRelatedList* (Cliente e Viatura). Tendo em conta que a alteração de um aluguer poderá não estar apenas relacionada com a alteração dos dados intrínsecos ao mesmo (data prevista inicio, data prevista fim, data inicio e data fim), é necessário prever uma alteração relacionada com o cliente ou a viatura a ele associados.

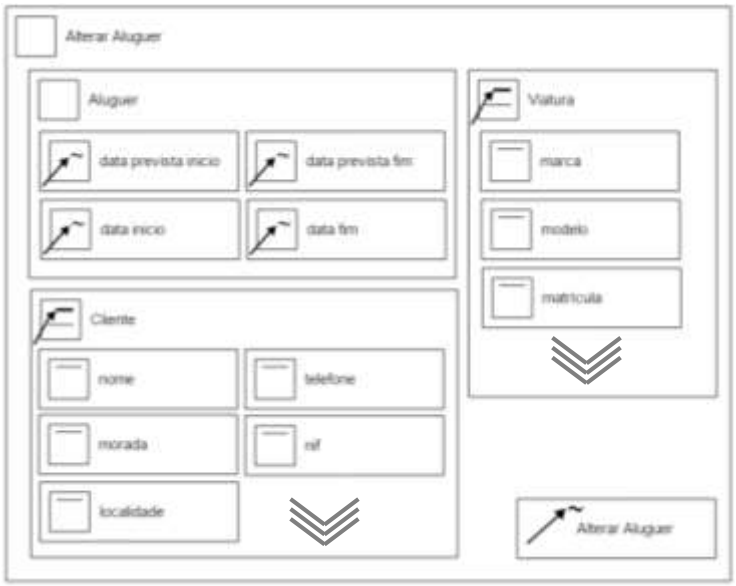


Figura 37 - *InteractionSpace Alterar Aluguer*

O *InteractionSpace Ver Detalhes Aluguer* (Figura 38) apresenta diferenças em relação aos *InteractionSpaces* representados nas figuras anteriores (35, 36 e 37) na medida em que é o único que explicitamente representa duas relações de generalização do tipo *<<include>>* na sua estrutura. No *Cenário Aluguer* (Figura 34) pode ler-se *Ver Detalhes Aluguer <<include>> Ver Detalhes Viatura,* e *Ver Detalhes Aluguer <<include>> Ver Detalhes Cliente* pelo que o *InteractionSpace Ver Detalhes Aluguer* deverá incluir os *InteractionBlocks Ver Detalhes Viatura* e *Ver Detalhes Cliente.*

O *InteractionSpace Ver Detalhes Aluguer* é composto por:

- Um *InteractionBlock* do tipo *ViewEntity* (Aluguer) que por sua vez contém quatro *DataAIOs* do tipo *OutputOnly* (data prevista inicio, data prevista fim, data inicio e data fim);
- Um *InteractionBlock* do tipo *ViewRelatedEntity* (Ver Detalhes Cliente) que por sua vez contém cinco *DataAIOs* do tipo *OutputOnly* (nome, morada, localidade, telefone e nif);
- Um *InteractionBlock* do tipo *ViewRelatedEntity* (Viatura) que por sua vez contém três *DataAIOs* do tipo *OutputOnly* (marca, modelo e matrícula).

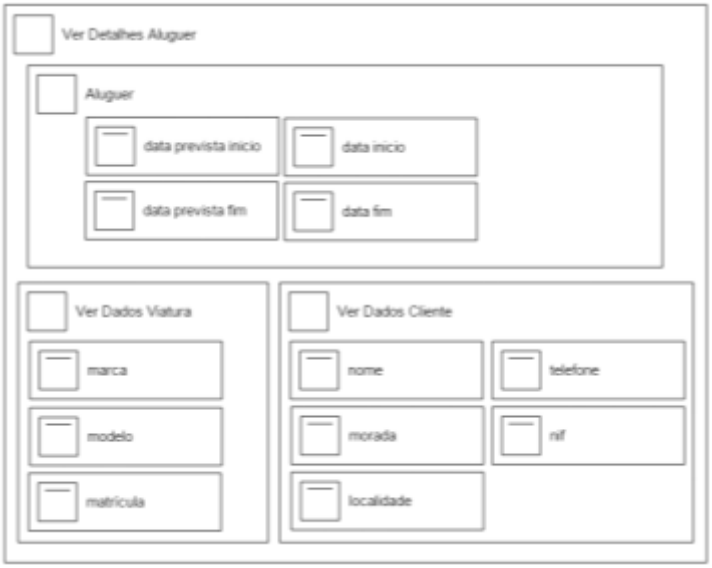


Figura 38 - ***InteractionSpace Ver Detalhes Aluguer (+ Ver Detalhes Viatura + Ver Detalhes Cliente)***

6.3 SUMÁRIO

Este capítulo pretendeu demonstrar a aplicabilidade dos Modelos de UI criados recorrendo à linguagem *xCAP* (construídos com a aplicação *MetaCAP*) a um exemplo concreto de um sistema (definido pelos Modelos de Domínio e de Casos de Uso). A análise do sistema permitiu ainda perceber que os Modelos de Domínio (Figura 22) e de Casos de Uso (Figura 23) complementam-se na medida em que ambos fornecem informação válida para a criação do Modelo da UI. Tendo em conta os cenários *Viatura* e *Cliente* (6.2.1 e 6.2.2) em tudo idênticos, verificou-se que dada a cardinalidade da relação (..1) entre ambas classes (Viatura e Cliente) com a classe Aluguer, a leitura do Modelo de Casos de Uso é suficiente para se criar o Modelo de UI correspondente. Contudo, a situação apresentada no cenário *Aluguer* (6.2.3) representativa da relação (..*) da classe Aluguer com as classes Viatura e Cliente carece de informação adicional (nomeadamente para a criação dos *InteractionSpaces Criar Aluguer* e *Alterar Aluguer*) aquando da criação do Modelo de UI, pelo que, a análise do Modelo de Domínio é neste caso essencial.

7. CONCLUSÕES E TRABALHO FUTURO

Este capítulo apresenta as ilações retiradas durante o desenvolvimento do trabalho bem como dos aspetos a melhorar futuramente.

7.1 CONCLUSÕES

Este trabalho fundamenta-se no *Model Driven Development* (MDD) e na premissa de que o seu principal objetivo é a criação de modelos que permitam num estádio final de desenvolvimento dar origem a código que se irá traduzir num sistema completo. Porém, este projeto não almeja atingir a fase final de desenvolvimento de um sistema (geração do código) mas sim otimizar a sua fase inicial, ou seja, a criação dos *Platform Independent Models* (PIM), nomeadamente o Modelo de Interface com o Utilizador.

Dado que o MDD utiliza tipicamente a linguagem UML para construir os seus modelos e que esta não possui suporte para a criação de Modelos da UI, a linguagem proposta, *xCAP*, pretendeu colmatar essa lacuna recorrendo aos símbolos gráficos dos *Canonical Abstract Prototypes* (CAP) e ao *Abstract User Interface Meta Model* (AUIMM) para fazer a ponte com o *Domain Meta Model* (DMM) da linguagem UML.

A análise dos Modelos nativos da UML, nomeadamente os Modelos de Domínio e de Casos de Uso (PIM), permite obter informação sobre o sistema que pode ser traduzida, através da sua associação à linguagem *xCAP*, em diferentes *InteractionSpaces*. Desta forma, o conjunto de todos os *InteractionSpaces* de um sistema designa-se de Modelo da UI.

Por fim cabe acrescentar o essencial, o relacionamento/associação entre os diferentes modelos que descrevem um sistema e a adoção de nomenclaturas semelhantes sugerem coerência e criam continuidade no processo de desenvolvimento de um produto de *software*. Desta forma a evolução do trabalho pode ser visualizada em todas as suas vertentes e os erros passíveis de ocorrer nas etapas seguintes são atenuados.

7.2 TRABALHO FUTURO

No que diz respeito ao trabalho futuro este deverá ser dividido em duas partes: especialização da linguagem *xCAP* e melhoramento do editor *MetaCAP*.

Relativamente à linguagem *xCAP*, o passo seguinte prender-se-á com o refinamento da linguagem de forma a eliminar a repetição de símbolos (a maior parte repete-se pelo menos duas vezes) e a generalizar os símbolos com funções semelhantes (p. ex: operações CRUD).

Em relação ao editor *MetaCAP*, são três as vertentes a ter em conta: o rigor científico, o aspeto técnico e a usabilidade. Para respeitar as premissas expostas no Capítulo 4, o *MetaCAP* deverá ser melhorado de forma a ser possível:

- Aplicar as restrições previstas nos *InteractionBlocks* relacionadas com o tipo de elementos que estes podem conter;
- Atribuir de forma automática a mesma classe a todos os elementos contidos num *InteractionBlock* do tipo *ViewEntity*, *ViewRelatedEntity*, *ViewList* e *ViewRelatedList*.

Ao nível técnico é ainda necessário melhorar:

- Gravação do ficheiro XML gerado a partir do Modelo de UI para que os elementos sejam ordenados hierarquicamente;
- Importação do Modelo de Domínio (ficheiro XML);
- Integração dos modelos gerados pelo *AMALIA IDE* com o *MetaCAP*.

A nível da usabilidade o *MetaCAP* deverá:

- Permitir a alteração do tamanho dos elementos recorrendo à utilização do rato;
- Permitir mover os elementos recorrendo às setas do teclado;
- Permitir que, quando movidos, todos os elementos contidos num elemento sejam movidos em simultâneo;
- Permitir a eliminação de elementos recorrendo à tecla *Delete*;
- Permitir alterar as propriedades dos elementos recorrendo à tecla *Enter*;
- Permitir a gravação do Modelo de UI como imagem.

REFERÊNCIAS

[1] A. M. R. d. Cruz e J. P. Faria, “Automatic Generation of User Interface Models and Prototypes from Domain and Use Case Models,” em *User Interfaces*, Croatia, InTech, 2010, pp. 35 - 60.

[2] L. Constantine, H. Windl, J. Noble e L. Lockwood, “From Abstraction to Realization: Canonical Abstract Prototypes for User Interface Design,” Julho 2003. [Online]. Available: http://www.researchgate.net/publication/227327529_Canonical_Abstract_Prototypes_for_Abstract_Visual_and_Interaction_Design. [Acedido em Junho 2014].

[3] A. M. R. d. Cruz, “Automatic Generation of User Interfaces from Rigorous Domain and Use Case Models,” PhD Thesis. Departamento de Engenharia Informática - Faculdade de Engenharia da Universidade do Porto, Porto, 2010.

[4] A. M. R. d. Cruz e J. P. Faria, “A Metamodel-based Approach For Automatic User Interface Generation,” *LNCS 6394, Springer-Verlag Berlin Heidelberg.,* Proceedings of the 13th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2010), Oslo, Norway, pp. 256-270, October 2010.

[5] A. M. R. d. Cruz, *Use Case and User Interface Patterns for Data Oriented Applications,* CCIS series, Springer-Verlag, 2015 (to appear).

[6] Infopédia - Porto Editora, “Definição de modelo no Dicionário da Língua Portuguesa,” [Online]. Available: http://www.infopedia.pt/dicionarios/lingua-portuguesa/modelo. [Acedido em 27 Maio 2015].

[7] Object Management Group, *A Proposal for an MDA Foundation Model,* Object Management Group, 2001.

[8] Infopédia - Dicionários Porto Editora, “Definição ou significado de protótipo no Dicionário da Língua Portuguesa,” [Online]. Available: http://www.infopedia.pt/dicionarios/lingua-portuguesa/prot%C3%B3tipo. [Acedido em 24 Maio 2015].

[9] M. R. Sawaya, Dicionário de Informática e Internet, São Paulo, Brasil: Nobel, 1999.

[10] Merriam-Webster, A Encycplopedia Britannica Company, “Mock-Up | Definition of mock-up by Merriam-Webster,” [Online]. Available: http://www.merriam-webster.com/dictionary/mock-up. [Acedido em 15 Abril 2015].

[11] Dicionário Priberam da Língua Portuguesa, “"interface", in Dicionário Priberam da Língua Portuguesa,” [Online]. Available: http://www.priberam.pt/dlpo/interface. [Acedido em 12 Maio 2015].

[12] Dicionário Priberam da Língua Portuguesa, “"layout", in Dicionário Priberam da Língua Portuguesa,” [Online]. Available: http://www.priberam.pt/dlpo/layout. [Acedido em 12 Maio 2015].

[13] Infopédia - Porto Editora, “display in Dicionário da Língua Portuguesa,” [Online]. Available: http://www.infopedia.pt/dicionarios/lingua-portuguesa-aao/display. [Acedido em 12 Maio 2015].

[14] Infopédia - Porto Editora, “multiplataforma in Dicionário da Língua Portuguesa,” [Online]. Available: http://www.infopedia.pt/dicionarios/lingua-portuguesa-aao/multiplataforma. [Acedido em 12 Maio 2015].

[15] J. Power, *Notes on Formal Language Theory and Parsing,* Maynooth, National University of Ireland, 2002.

[16] Dr. Dobb's - The World of Software Development, “Event-Based Architectures,” [Online]. Available: http://www.drdobbs.com/architecture-and-design/event-based-architectures/208801141. [Acedido em 12 Maio 2015].

[17] Computer Hope, “What is callback?,” [Online]. Available: http://www.computerhope.com/jargon/c/callback.htm. [Acedido em 17 Maio 2015].

[18] World Wide Web Consortium (W3C), “MBUI - Task Models,” 08 Abil 2014. [Online]. Available: http://www.w3.org/TR/task-models/. [Acedido em 15 Maio 2015].

[19] SSWUG.org, “Data-Modeling vs. Implementation Modeling,” [Online]. Available: http://www.sswug.org/articles/viewarticle.aspx?id=18303. [Acedido em 19 Maio 2015].

[20] Infopédia - Porto Editora, “bitmap in Dicionário da Língua Portuguesa,” [Online]. Available: http://www.infopedia.pt/dicionarios/lingua-portuguesa-aao/bitmap. [Acedido em 12 Maio 2015].

[21] Jenkov.com, “HTML5 Canvas Overview,” [Online]. Available: http://tutorials.jenkov.com/html5-canvas/overview.html. [Acedido em 5 Maio 2015].

[22] Wikipedia, “Scene Graph, Wikipedia the free Encyclopedia,” [Online]. Available: http://en.wikipedia.org/wiki/Scene_graph. [Acedido em 9 Maio 2015].

[23] World Wide Web Consortium (W3C), “Document Object Model (DOM),” [Online]. Available: http://www.w3.org/DOM/. [Acedido em 15 Maio 2015].

[24] Wikipedia, “Curva de Bézier - Wikipédia a enciclopédia livre,” [Online]. Available: http://pt.wikipedia.org/wiki/Curva_de_B%C3%A9zier. [Acedido em 15 Maio 2015].

[25] Scriptographer.org, “Scriptographer.org - About,” [Online]. Available: http://scriptographer.org/about/. [Acedido em 16 Fevereiro 2015].

[26] Microsoft Developer Network, “Hit Testing Lines and Curves,” [Online]. Available: https://msdn.microsoft.com/en-us/library/ms969920.aspx. [Acedido em 20 Maio 2015].

[27] Webopedia, [Online]. Available: http://www.webopedia.com/TERM/E/event_handler.html. [Acedido em 23 Abril

2015].

[28] N. Souchon e J. Vanderdonckt, *A Review of XML - compliant User Interface Description Languages,* Louvain-la-Neuve, Belgium: Universit´e catholique de Louvain, Institut d'Administration et de Gestion, 2003.

[29] OASIS, "User Interface Markup Language (UIML) Version 4.0," 23 Janeiro 2008. [Online]. Available: http://docs.oasis-open.org/uiml/v4.0/cd01/uiml-4.0-cd01.html#_Toc201918538. [Acedido em 22 Maio 2015].

[30] A. Puerta e J. Eisenstein, *XIML: A Universal Language for User Interfaces,* Palo Alto, CA USA: RedWhale Software.

[31] Mozilla Developer Network, "XUL - Mozilla|MDN," [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XUL. [Acedido em 14 Junho 2014].

[32] J. Vanderdonckt, Q. Limbourg, B. Michotte, L. Bouillon, D. Trevisan e M. Florins, *USIXML: a User Interface Description Language for Specifying Multimodal User Interfaces,* Louvain-la-Neuve, Belgium: Université catholique de Louvain, School of Management (IAG), ISYS-BCHI.

[33] G. Calvary, J. Coutaz, D. Thevenin, L. Bouillon, M. Florins, Q. Limbourg e N. Souchon, *The CAMELEON Reference Framework,* Turin, Italia: UJF, UCL, CNUCE, 2002.

[34] L. Constantine, "Rapid Abstract Prototyping," *Software Development,* Outubro 1998.

[35] M. Beaudouin-Lafon, *User Interface Management Systems: Present and Future,* Paris: Université Paris Sud, 1994.

[36] P. Szekely, *Retrospective and Challenges for Model-Based Interface Development,* Marina del Rey, California: Information Sciences Institute, University of Southern California.

[37] C. Costa, D. Coelho, F. Oliveira, M. Silva e S. Vilaça, "Model Driven Development," Universidade do Porto - Faculdade de Engenharia, Porto.

[38] D. Costa, L. Nóbrega e N. J. Nunes, *An MDA Approach for Generating Web Interfaces with UML ConcurTaskTrees and Canonical Abstract Prototypes,* Funchal, Madeira: Universidade da Madeira, 2007.

[39] D. Bäumer, W. Bischofberger, H. Lichter e H. Züllighoven, "User Interface Prototyping – Concepts, Tools, and Experience," em *Procs. of the 18th International*, Berlim, 1996.

[40] L. Constantine e L. Lockwood, *Usage-Centered Engeneering for Web Applications,* Constantine & Lockwood, Ltd., 2001.

[41] Object Management Group, *OMG Meta Object Facility (MOF) Core Specification,* Object Management Group, 2014.

[42] L. Abreu, HTML 5, Lisboa: FCA, 2012.

[43] Paper.js, “Paper.js,” [Online]. Available: http://paperjs.org/. [Acedido em 25 Janeiro 2015].

[44] J. E. Garcia, *Engenharia de Requisitos - Casos de Uso (Apontamentos),* Viana do Castelo, 2012.