Compreensão de Aplicações Web: O Processo e as Ferramentas

Eva Oliveira
Instituto Politécnico do Cávado e Ave (IPCA)
eoliveira@ipca.pt

Maria João Varanda Pereira
Instituto Politécnico de Bragança (IPB)
mjoao@ipb.pt

Pedro Rangel Henriques
Universidade do Minho (UM)
prh@di.uminho.pt

27 de Junho de 2005

Resumo

Já ninguém duvida que a **compreensão de programas (CP)** é uma área cada vez mais importante, e a sua aplicação cada vez mais abrangente na engenharia da programação, constituindo-se como um desafio permanente e actual para as ciências de computação. A compreensão de programas nasce da tentativa de melhorar os processos de manutenção de sistemas aplicacionais, mas de facto está também francamente associada ao processo de aprendizagem de linguagens de programação. Por isso, nesta linha de preocupações têm sido desenvolvidas inúmeras ferramentas de auxilio à compreensão de software para manutenção (alteração ou actualização de código), *reverse engineering*, ou ensino de programação. A complexidade do processo de compreensão requer a existência de modelos de aprendizagem e representação de conhecimento. Uma questão, que então se coloca muitas vezes, é saber se as ferramentas obedecem aos modelos cognitivos existentes; importa também saber como medir o seu real impacto na compreensão.

Este artigo tem como objectivo a descrição de um conjunto de critérios a adoptar no sentido de avaliar se uma ferramenta pode efectivamente ajudar na compreensão de programas. Dada a importância cada vez maior das aplicações que correm sobre a *Web*, o nosso estudo incidirá sobre esse recente paradigma de programação. Procuraremos identificar o que de novo há neste tipo de programação que interessa realçar para o seu bom entendimento.

1 Introdução

Um **programa** é uma sequência lógica de instruções que ora afectam os dados, ora conduzem o fluxo de execução. Essa execução produz uma transição de estados ao nível do processador e provoca uma transformação ao nível dos dados, actuando dessa forma sobre o mundo exterior. Por isso, a compreensão de um programa passará sempre pelo entendimento destas duas realidades que tradicionalmente se estudam à custa de duas abstracções: a *estrutura de dados* (ED) que suporta a informação a ser transformada; e o *algoritmo* (ALG) que descreve essa transformação.

A um programa está associada uma linguagem de programação e pelo menos um paradigma de programação; podemos ter também tecnologias associadas, como infra-estruturas de comunicação, bases de dados, etc.

Uma **linguagem de programação** é uma forma de expressão da sequência de instruções de um programa. Compreender um programa exige também a compreensão dessa notação, ou seja, dos símbolos e das regras gramaticais que definem a sintaxe e a semântica da dita linguagem.

Um **paradigma** pode ser definido pela estratégia que se adopta na resolução de um problema. Logo o paradigma reflecte-se no programa e como tal a linguagem tem que estar preparada para suportar tal abordagem. O uso correcto de uma linguagem exige o conhecimento do paradigma subjacente, da mesma forma que a escolha de um paradigma requer a escolha de uma linguagem adequada. Daí que o entendimento de um programa não se possa confinar ao conhecimento sintáctico e semântico da linguagem em que está expresso, mas requeira também o conhecimento do paradigma que foi seguido na sua concepção.

As **tecnologias** são artefactos de software/hardware que permitem construir programas em cima deles, resolvendo de maneira sistemática e normalizada determinadas funcionalidades. Assim, a tecnologia também deve ser compreendida para que um programa seja entendido na sua totalidade.

Em suma, a compreensão de programas passa por perceber:

- A estrutura de dados e o algoritmo (estrutura do programa)
- O paradigma
- A linguagem
- A tecnologia

sendo estes os quatro objectos de análise que a nosso ver deverão ser cobertos por uma ferramenta de compreensão de programas (tradicionais, ou orientados à Web).

Este artigo terá como grande objectivo sistematizar o processo de compreensão de aplicações Web e avaliar de que forma as ferramentas existentes nos fornecem toda a informação que julgamos essencial para tal compreensão.

E porquê *Aplicações Web*? Cada vez mais se desenvolve programas suportados na filosofia e nas tecnologias da World Wide Web (W^3): uma framework que permite a escalabilidade e extensibilidade de funcionalidades e é baseada em novas linguagens

(de anotação de texto e de programação) e em novos protocolos (de comunicação e interacção), sendo a interactividade com o utilizador realizada por programas de interface genéricos — *browsers* — que surgem, inicialmente, com o objectivo de podermos ler e navegar livremente entre as páginas dos hiperdocumentos HTML.

Assim e no contexto deste trabalho, designaremos por **Aplicação Web** todo o conjunto de programas que implementa um qualquer Sistema de Informação segundo o paradigma Cliente/Servidor suportado pelo protocolo de comunicação HTTP e cuja camada interactiva está escrita em HTML de modo a que a interface com o utilizador seja assegurada pelos *browsers* tradicionalmente criados para navegação na rede de hiperdocumentos W^3.

Caracterizaremos com detalhe o que é uma aplicação Web, especificando qual o paradigma que lhe está subjacente, quais as linguagens usadas e que tecnologias lhe estão associadas. Acreditamos que, deste modo, se tornará evidente quais as componentes cuja compreensão será necessária para que se entenda efectivamente uma aplicação Web.

Para alcançar o objectivo proposto, o artigo está organizado em três secções, para além das habituais secções de introdução (Secção 1), onde se caracteriza a área de trabalho e se estabelece o âmbito da comunicação, e de conclusão (Secção 5), onde se fecha o discurso com a síntese dos objectivos e as guias para trabalho futuro. O tema central é tratado na Secção 4, onde se discute a eficácia de ferramentas existentes face ao conjunto de parâmetros que se identificam como cruciais para compreender aplicações para a Web. Antes de apresentar essas ideias fundamentais, faz-se uma rápida revisão ao estado da arte na área da compreensão de programas (Secção 2) e revê-se o conceito de paradigma e de linguagem de programação de modo a poder-se caracterizar o caso concreto do paradigma de programação Web (Secção 3).

2 A Compreensão de Programas

Sintetizando a opinião de vários autores, a **Compreensão de Programas (CP)** é a área de investigação que está ligada à forma como se analisam e interpretam blocos de código para assim se perceber programas ao nível da sua estrutura e comportamento, ou seja da estrutura de dados e do algoritmo subjacente [DKKL04]. Por estrutura designamos, quer os dados definidos/declarados e manipulados, quer as instruções que o constituem; quando falamos em comportamento estamos a falar de fluxo de execução das operações e fluxo de dados.

Podemos afirmar que a compreensão de programas se inicia com Brooks [Bro78], quando descreve o primeiro modelo cognitivo para a compreensão do comportamento de programas. Este modelo aplica a técnica top-down, que se caracteriza por começar a sua análise em termos de objectivos do sistema, passando pelas funcionalidades, procedimentos e só depois é que vai ao código. Realmente o que levou Brooks, e outros autores como Letovsky e Pennighton, a especificarem modelos cognitivos foi o facto de se ter como crença que estes influenciam a actividade de compreensão de um programa, quer no sentido de a disciplinarem, quer no sentido de a facilitarem [vMV95].

A CP nasce quando se sente a necessidade de *analisar código* para o *testar, alte-*

rar, modificar, ou como parte integrante do *processo de re-engenharia* para criar um novo sistema com base no melhor de um já existente. A cada um destes objectivos são associadas diferentes tarefas, mas todas elas têm em comum a tarefa do entendimento de código. Os modelos cognitivos aparecem assim como forma de descrever os processos de aquisição de conhecimento envolvidos nesta tarefa, possibilitando a extracção de informação e representação do conhecimento em diferentes níveis de abstracção [vMV95]. De acordo com a síntese de Pacione [Pac04a], os modelos mais bem aceites nesta área são: o modelo *top-down (TD)* com *evolução de hipóteses*, de Brooks [Bro78] ou de Soloway e Ehrlich []; o modelo *bottom-up (BU)* de Shneiderman [SM79], ou Pennington []; o modelo *baseado em domínios de conhecimento* de Letovsky [], em que se admite que o processo combina as abordagens TD e BU e que deve ser suportado por uma base de conhecimento contendo informação sobre o domínio do problema, o domínio da programação, e o modelo mental que reflecte o que o programador já percebeu do sistema; e o modelo *integrador* de von Mayrhauser e Vans [vMV95] em que, há semelhança do anterior, os autores sugerem a combinação das abordagens TD e BU numa base de conhecimento onde se vai formando uma descrição única do sistema.

A complexidade do alcance deste objectivo da CP levou, desde cedo, ao desenvolvimento de ferramentas que apoiam o analista neste processo.

Seguindo de novo Pacione, mas agora em [Pac04b], pode afirmar-se que a **visualização** é a principal técnica envolvida na compreensão de programas. Estudos feitos pela comunidade de visualização de algoritmos mostram que esta facilita em muito a percepção por parte do humano, sendo sabido que conceitos complexos ou abstractos se captam mais facilmente através de elementos visuais. A visualização pode ajudar nas diferentes tácticas de compreensão existentes. Porém, é necessário estar alerta para o facto de que o tipo de visualização compromete sempre positiva ou negativamente a compreensão de um programa, pelo que importa averiguar *o que é mostrado* e *como é apresentado* a fim de evitar prejudicar o correcto entendimento. Uma boa ferramenta de visualização de código deve permitir aumentar ou diminuir o nível de detalhe; em suma, deve preocupar-se com as quatro questões basilares da visualização: *a representação, a abstracção, a navegação e a correlação* [Pac04b].

A **animação** de programas é outra técnica, complementar à anterior, fornecendo informação em *run-time*, associando à estrutura de dados e ao algoritmo elementos visuais e ajudando na percepção da evolução da execução de um programa. A animação deve mostrar a ordem e o momento que em as operações se realizam e os dados são afectados. O sistema Alma [Per03] é um exemplo de uma ferramenta genérica de construção de representações visuais e animação de programas desenvolvida com o objectivo de auxiliar, o aprendiz ou o programador, na sua compreensão.

A **análise de dependências** também constitui uma técnica que auxilia na compreensão, no sentido em que nos permite verificar as dependências entre *elementos de um programa*: variáveis, procedimentos/métodos ou classes. O grafo resultante desta análise permite saber de quem e de que operações depende qualquer um desses elementos e fornece o conhecimento do que poderá ser afectado quando se faz uma alteração em algum dos elementos. Tal entendimento do fluxo de dados de um programa revela um bom conhecimento do sistema. Percebe-se pois que grande número das ferramentas de

CP dêem ênfase a esta técnica aliada à visualização.

Muitas ferramentas foram desenvolvidas, tanto na área académica como na industria, nos últimos anos. Todas elas têm um objectivo em comum que é o de compreender grandes sistemas, no entanto existem diferentes abordagens e funcionalidades em cada ferramenta. Segundo [Lin], podemos dizer que se consideram ferramentas de compreensão de programas as que satisfazem os seguintes critérios:

- suportam um ou mais modelos cognitivos;
- mantém um repositório sobre a arquitectura e comportamento do sistema;
- apresentam várias formas de representar a mesma funcionalidade ou parte do código;
- possuem técnicas de partição de código (*code slicing*);
- possuem mecanismos de abstracção de aspectos menos relevantes.

Podemos dar alguns exemplos de ferramentas que auxiliam na compreensão das aplicações chamadas clássicas, vocacionadas para agilizar os processos de manutenção e de re-engenharia.
Na medida em que as ferramentas encontradas são todas de abrangência limitada e, também, não há uma linha nítida de inclusão entre elas (uma hierarquia bem definida), optámos por as ordenar cronologicamente. Entre outras, parecem-nos dignas de registo:

- Rigi [MTW93] — é uma ferramenta visual e interactiva que serve para ajudar a melhor compreender um sistema aplicacional e documentar à posteriori o software. Os seus principais objectivos são: fornecer uma infraestrutura para investigação e prática de CP; e descobrir abstracções de grandes sistemas produzindo informação importante para os responsáveis pela sua manutenção ou re-engenharia. Possui um editor de grafos (*RigiEdit*) que permite analisar, representar e visualizar a estrutura de um sistema. Como principais características podemos dizer que corre em diferentes plataformas, possui parsers para C, C++ e COBOL e existe um versão disponível para download.
- SHriMP [LMSW03] — é uma ferramenta que usa a filosofia dos hiperlinks para visualizar relações de dependência entre os elementos de um sistema a diferentes níveis de abstracção, o que no caso de uma aplicação nos permite analisar dependências desde os módulos principais ao código fonte. Mostra diferentes tipos de hierarquias, ou seja, podemos de facto ver vários tipos de relações entre classes. É uma ferramenta que fornece um mecanismo muito prático de navegação e possibilita termos a noção da aplicação como um todo, sem nos perdermos do essencial que estivermos a analisar, ou nos deixa descer ao estudo dos detalhes. Neste momento a sua interface está vocacionada para analisar aplicações Java, embora seja concebida com um propósito mais genérico.
- Jeliot 3 [MMSBA04] — é um programa de visualização de aplicações Java que examina e mostra como um programa Java é interpretado: são mostradas as

invocações de métodos, valores de variáveis e sequência de operações. Um filme interactivo é usado para mostrar o evoluir da interpretação do programa pedindo ao utilizador para introduzir valores de entrada. Embora não mostre os aspectos importantes relacionados com o paradigma, como o conceito de herança, o grande objectivo é o ensino da programação através de uma ferramenta de construção semi-automática de visualizações do controlo de fluxo e de dados.

- Chive [CE04] — é uma plataforma de visualização de código fonte permitindo representar em 3D uma hierarquia de classes. É uma tentativa de proporcionar aos programadores um desenvolvimento mais rápido e tirar partido da separação de dados e das representações de grafos 3D. É um protótipo desenvolvido em Java para aplicações Java.

- CodeSurfer [AZ05] — é uma ferramenta comercial de análise de código fonte disponibilizando um navegador para facilitar a análise de dependências. É dirigido a programas escritos em C ou C++ e permite seguir a evolução dos valores das variáveis e chamadas a funções, mostrando grafos de dependências completos.

Mais recentemente, estas ferramentas para CP têm sido expandidas, alargando a sua vocação à área de ensino da programação. Um exemplo típico é o já referido Jeliot3, ou o BlueJ [KQPR03], que foram desenvolvidos para o ensino da linguagem Java, fazendo para isso a visualização e animação do programa fonte que lhe é submetido.

Também a programação para a Web foi abrangida por estas ferramentas para CP, que vêem assim alargado o seu âmbito. Destas, umas estão mais orientadas para mostrar a estrutura de sítios Web (poderemos chamá-las *ferramentas parciais*, ou ferramentas para *site comprehension*)—como é o caso do FrontPage que constrói automaticamente o mapa de um site— enquanto outras focam-se no comportamento da aplicação— citam-se, a título de exemplo, o PBS [PBS] e o WARE [LFP$^+$02]. Na medida em que, mesmo neste segundo caso referido, a visão dinâmica pode ser apresentada no lado do cliente, ou seja na *perspectiva de quem usa a aplicação*, resta-nos a dúvida se já há alguma plataforma que efectivamente permita um conhecimento global da aplicação Web em análise na *perspectiva do programador*. Na tentativa de responder a esta questão, entendemos que devíamos dissecar a fundo o paradigma (o que faremos na Secção 3, a seguir) de modo a estabelecer critérios de análise destas ferramentas (como se dirá na Secção 4 e se irá ilustrar com ferramentas concretas).

3 Os Paradigma e as Linguagens de Programação

Cada linguagem tem os seus símbolos e regras. Os símbolos servem para compor as frases; as regras restringem uma linguagem ao nível sintáctico e semântico. As sintácticas definem como construir sequências válidas de símbolos, as semânticas preocupam-se com o valor absoluto ou relativo de cada símbolo, discernindo o significado das frases. Ou seja, as primeiras regras estão orientadas para a forma e as segundas para o conteúdo. Para compreendermos uma linguagem, o significado é o mais importante. Se pensarmos em duas pessoas que tentam comunicar, o essencial para se perceberem

é conhecerem o significado das palavras que cada uma diz. Na compreensão de um algoritmo é isso que interessa também, sabermos o significado das várias junções de símbolos.

Um paradigma de programação pode ser implementado por uma ou mais linguagens, e consiste num modelo, ou estilo de programação. Em teoria, um programa pode ser escrito em qualquer linguagem; no entanto o que distingue um programa, por exemplo *para cálculo de notas*, feito em C ou em C++ é o paradigma de programação, i.e., os conceitos base que estão associados à forma de raciocinar e exprimir o algoritmo para a sua resolução. Assim um paradigma de programação será sempre caracterizado, como dissemos, por um conjunto de conceitos base e linguagens de programação. Por exemplo no caso do paradigma imperativo teremos associados os conceitos de atribuição, estado, e as linguagens C e Pascal. Já no paradigma orientado a objectos teremos os conceitos de objecto, método, herança, e as linguagens Java e C++. No paradigma lógico teremos os conceitos de relação, predicado, cláusula, dedução, e a linguagem Prolog.

As estratégias de compreensão de programas, foram inicialmente concebidas a pensar em linguagens imperativas, pois os grandes sistemas de informação, que hoje importa manter ou redesenhar, foram nelas desenvolvidos. Entretanto novos paradigmas surgiram e teremos que saber se estas técnicas se adaptam aos novos desafios. É por este motivo que neste artigo tentamos descrever o que define o *Paradigma de Programação Web*, no sentido em que a nosso ver o modelo de programação diferencia-se dos paradigmas até aqui existentes, o que requer uma adaptação das ferramentas para a sua compreensão.

3.1 O Paradigma de Programação Web

Aqui retoma-se a definição de *Aplicação Web* apresentada na Introdução, da qual sobressaem dois grandes ingredientes a distinguir este dos demais paradigmas: sistema de informação suportado no modelo Cliente/Servidor; camada interactiva baseada na anotação explícita e no princípio da hiper-ligação entre documentos. Do ponto de vista da compreensão, tal como da programação, a perspectiva em que nos interessa focar quando analisamos uma *Aplicação Web* é a do lado do Servidor, o que no caso nos coloca numa situação claramente distinta de olhar para a aplicação na perspectiva do utilizador (do lado do Cliente).

Uma **Aplicação Web** é formada por um conjunto de ficheiros de dois tipos:

(T1) texto *HTML* puro, ou com código embebido

(T2) objectos tais como: bases de dados, documentos anotados, código executável, ou imagens

O primeiro tipo de ficheiros (T1) é constituído por texto anotado numa linguagem de estruturação, formatação e ligação, linguagem essa que é interpretada pelo *browser* dando origem às chamadas *páginas* W^3. Os objectos (tipo T2) são ficheiros de texto normal ou binários (executáveis ou não), que são chamados/consultados pelos primeiros e dão a sensação de que fazem parte da página em que estão a ser chamados.

Esta classificação assenta no papel, activo ou passivo, dos ficheiros em relação ao *browser*, determinando assim a sua responsabilidade, ou contribuição directa/indirecta, para a construção do *site* W^3 que é apresentado e manipulável pelo utilizador final. Estamos conscientes de que esta divisão dos ficheiros em dois grandes tipos é grosseira. Na realidade, a sua subdivisão em tipos de grão mais fino vai ser necessária porque diferentes tecnologias serão usadas para implementar os papeis de cada subtipo (o que vai requer diferentes formas de análise para se atingir a compreensão exaustiva da aplicação). Contudo tal detalhe não nos parece de momento relevante, a bem da clareza da exposição.

Ao aceder a um ficheiro do tipo T1, localizado de acordo com um determinado URL indicado pelo Cliente, o Servidor aplica-lhe uma função de transformação antes de o enviar ao consumidor que o solicita. Se a função aplicada for a `Identidade` (a página devolvida é igual a existente) designamos o ficheiro em causa por **página estática** (mesmo que ao ser lida no Cliente tenha animação). Se for uma transformação diferente, produzindo uma página para o Cliente diferente da que está armazenada no Servidor, designamos o dito ficheiro por **página dinâmica**; neste caso há sempre processamento no Servidor. No caso de a página ser dinâmica, a transformação operada pode produzir uma página com conteúdo variável mas sempre do mesmo tipo—por mesmo tipo define-se uma página cujo conteúdo apenas difere textualmente, ou seja o desenho da página é sempre o mesmo, com os mesmos componentes (o conteúdo dos componentes é que será diferente); mas pode também produzir páginas de tipos diferentes, ou seja, páginas com desenhos diferentes e com diferentes componentes.

É também importante reconhecer se a página acedida (estática ou dinâmica) inclui código para ser interpretado no Cliente, código esse que possa dar-lhe um aspecto de dinâmica.

Tomando em consideração a classificação dos ficheiros que constituem uma Aplicação Web nos dois tipos, T1 (subdividido em páginas estáticas, T1E, e dinâmicas, T1D) e T2, podem identificar-se 4 tipos de relações entre esses ficheiros:

- *é acedido por* (é chamado por): entre T1 e T1 ou entre T2 e T1
- *acede a* (chama através de links ou acções de botões): entre T1 e T1 ou entre T1 e T2
- *gera* (tipos de páginas): entre T1D e T1E
- *depende de* (precisa desses objectos para ser produzida ou mostrada): entre T1 e T2

Estas relações determinam uma boa parte da compreensão da aplicação (visão estrutural e parte da visão lógica). Contudo, para compreender totalmente é preciso analisar fluxos de dados e de controlo dentro de cada página dinâmica (o que nos dará a outra parte da visão lógica que permite entender o comportamento).

Além destes conceitos é necessário que um programador, experiente ou não em tecnologias Web, consiga perceber se uma aplicação está a aceder a uma base de dados, que tipo de protocolos está a utilizar, qual o servidor e suas responsabilidades e onde estão especificados as regras de funcionamento do site. Consideraremos, então, como

componentes tecnológicas a ter em consideração: o *browser*, os *protocolos*, as *bases de dados*, os *servidores*, a *infra-estrutura de comunicações*, os *agentes* e os *Web Services*.

4 A Compreensão de Aplicações Web e as Ferramentas

Todos estes componentes e relações entre elementos Web, fazem parte de uma arquitectura que se pretende que seja mostrada em vários níveis e abstracções; a arquitectura, como define [Fie00], é uma abstracção dos elementos de um sistema de software num instante de tempo durante a sua operação. A arquitectura deverá mostrar o paradigma usado, evidenciando a forma como os diferentes componentes interagem, salientando a quantidade de protocolos necessários e as diferentes tecnologias usadas [FT00].

Se tivermos a percepção das relações acima enumeradas em todos os instantes de tempo e podermos visualizar qualquer uma das relações de forma independente, teremos uma visão mais clara e verdadeira da aplicação. Assim podemos saber o que determinada funcionalidade implica em termos de páginas Web e tecnologias.

Quando queremos conhecer uma aplicação clássica devemos ter sempre em conta o nível de conhecimento da pessoa que a está a tentar compreender. Se for um perito em programação em geral, ou um conhecedor do modelo lógico da aplicação, terá necessidades diferentes em relação ao que pretende de uma ferramenta de compreensão do que uma pessoa que não esteja familiarizada nem com a tecnologia nem com a funcionalidade da aplicação. Em [RW97], as autoras sugerem que programadores experientes tem tendência a usar uma estratégia *top-down*, enquanto os novatos seguem uma estratégia *bottom-up*, o que implica que numa ferramenta, que à partida é usada por todo o tipo de pessoas, se englobem estas duas aproximações diferentes.

Assim e sintetizando o que foi dito, os critérios que propomos para analisar a eficácia de uma Ferramenta para Compreensão de Aplicações Web são:

1 obedece a um ou mais modelos cognitivos?

2 é possível visualizar os diferentes componentes da aplicação?

3 permite uma navegação pela estrutura, aumentando ou diminuindo o nível de abstracção, de modo a possibilitar ver desde a arquitectura mais geral até ao código de cada página?

4 possibilita o uso de pelo menos duas estratégias de compreensão (top-down, bottom-up)?

5 mostra abstracções diferentes para a mesma funcionalidade?

6 cobre a parte estrutural (estática) e de conteúdos (dinâmica)?

7 evidencia o fluxo de dados e controlo?

8 recorre a técnicas de visualização e animação de algoritmos?

9 suporta todas as linguagens de scripting, tanto no cliente como no servidor?

10 visualiza a relação ”depende de” para cada página?

11 visualiza a relação ”acede a” para cada página?

12 visualiza a relação ”são acedidas por” para cada página?

13 mostra a relação ”gera” para cada página?

14 mostra a correspondência ficheiros de código / páginas?

15 as visualizações/animações não excedem o tamanho de um vulgar monitor?

16 consegue delimitar as visualizações de maneira a que quando se esta a usar determinada vista seja possível não visualizarmos mais nada, mas que de igual forma seja fácil passarmos a uma visão mais alargada?

Para ilustrar as ideias apresentadas, vamos debruçarmo-nos, no resto desta secção, sobre a análise de 2 ferramentas específicas: o WARE e o PBS.

4.1 WARE

Esta ferramenta [LFP+02] consegue, através de uma análise estática e dinâmica (estrutural e comportamental), refazer, ou reconstruir, os diagramas UML que modelam as Aplicações Web. Sendo-lhe associado um visualizador adequado (que mostra os diagramas obtidos), permite a compreensão da aplicação. A análise estática é alcançada através do IRF, um componente que traduz todos os ficheiros da aplicação numa linguagem baseada em tags e constrói uma tabela com todos os componentes da aplicação.

Através de uma ferramenta de visualização externa constrói um diagrama de classes da aplicação. Cada classe é um tipo de página, mas este diagrama só se traduz em quatro tipos de páginas, as páginas cliente estáticas, páginas cliente dinâmicas, formulários e páginas servidoras.

A analise comportamental é feita através do levantamento de hipóteses sobre o diagrama de classes encontrado e através do funcionamento da aplicação. Este diagrama de classes permite ver dependências e agrupamentos (em inglês, *clusters*) de páginas. Inicialmente não se sabe o que cada grupo faz mas, juntamente com o nome das páginas e a forma como decorre a execução, descobrem-se as funcionalidades. Através deste diagrama e da variada informação que vai sendo retirada e armazenada sobre o sistema em estudo, é possível construir diagramas *use case* da aplicação, bem como diagramas de sequências que permitem ver o fluxo de controlo e de dados de certa funcionalidade.

Uma vez que a ferramenta não existe para *download*, toda a análise aqui feita baseia-se na descrição encontrada em [LFP+02, LFTC02, LFT04]. Do que foi percebido, o WARE satisfaz bem os requisitos 1 (segue o modelo de Brooks), 3, 6, 7, 9, 11 e 12. Satisfaz parcialmente os requisitos 2, 4 (só TD), 5, 10, 13. Não parece satisfazer os restantes critérios (8, 14, 16); relativamente ao 15, dá ideia que as visualizações, ou excedem o ecrã, ou então são de leitura muito difícil.

4.2 PBS

A ferramenta Portable Bookshelf [FHK+97, Hol97, PBS] é, desde há alguns anos, usada como plataforma para recuperar sistemas aplicacionais tradicionais. Hassan e Holt [HH01] fizeram algumas modificações no sentido de permitir que esta ferramenta pudesse ajudar a recuperar arquitecturas de aplicações Web. No entanto, nos estudos efectuados por Hassan e Holt para efectuarem essa recuperação foi necessário desenvolver outras ferramentas de apoio ao *PBS*, pois este por si só não faz filtragens nem extracção de código, permitindo apenas, através de um determinado conjunto de factos, produzir gráficos — trata-se, portanto, de um visualizador. As ferramentas desenvolvidas, chamadas *extractores*, não são mais que analisadores especializados para vários tipos de linguagens/tecnologias usadas na *Web*, tais como, um *HTML Extractor*, um *Server Script Extractor*, *DB Access Extractor*, *Language Extractor* e um *Binary Extractor*. Estes extractores produzem a Base de Factos que servirá de entrada ao *PBS*; O resultado será, então, a chamada Arquitectura de Documentos da Aplicação.

É preciso algum esforço para se por o processo de análise em marcha porque não se trata de uma ferramenta completamente automática, precisa de ajustamentos por parte do utilizador, nomeadamente em relação à técnica de agrupamento a ser usada pelo *PBS*. Em resumo, podemos dizer que a ferramenta apresenta de facto os agrupamentos das funcionalidades e as dependências entre funcionalidades, podendo chegar a um nível mais baixo de abstracção, em que podemos dentro de uma funcionalidade visualizar os seus componentes e suas relações com as bases de dados.

Em função do que foi lido, concluímos que o PBS satisfaz bem os requisitos 1 (segue o modelo de Brooks), 9 e 15. Satisfaz parcialmente os requisitos 2, 3, 4 (só TD), 6 (só estrutura). Não parece satisfazer os restantes critérios (5, 7, 8, 14, 16); relativamente aos critérios 10 a 13 é difícil dizer porque as visualizações são mais orientadas ao sistema e não tanto à página.

5 Conclusão

Vários estudos comprovam que mais de metade do tempo de actualização e manutenção de software se passa na compreensão deste. Com o ambiente competitivo dos dias de hoje, tempo é dinheiro e, por isso, a compreensão representa um elevado custo para as empresas.

Mostrou-se ao longo do artigo que o paradigma Web é a mescla de vários conceitos, quer de paradigmas/linguagens e técnicas de programação, quer mesmo de objectos computacionais (ficheiros vários). Assim pretendeu-se afirmar que o entendimento de uma Aplicação Web pode requerer o conhecimento de mais que uma linguagem (e respectivos paradigmas de programação) e que a compreensão do seu comportamento requer, ainda, que se perceba todo um conjunto de tecnologias distintas. Conclui-se também que a visualização destaca-se como a técnica mais usada nas ferramentas de compreensão de programas.

Do estudo feito, sobressaem então três aspectos essenciais para a CP:

- a importância de uma boa táctica de aplicação de elementos visuais;

- a importância dos diferentes níveis de abstracção para a mesma funcionalidade;
- a importância de uma ferramenta extensível a novas tecnologias tal como a *arquitectura Web*.

Apesar de já existirem várias ferramentas bastante eficazes para a compreensão de software tradicional, ferramentas essas que são tipicamente devotadas a um tipo de sistema ou a uma linguagem específica, ficou claro que a nível da compreensão de Aplicações Web ainda há espaço para novos desenvolvimentos no sentido de satisfazer todos os requisitos enunciados na Secção 4. Das ferramentas tradicionais analisadas, foi claramente a SHriMP que mais nos deixou surpreendidos e agradados pela forma eficaz como permite percorrer um sistema, desde um nível de abstracção bastante geral até ao código. Este sistema servirá de orientação para a ferramenta que, no futuro, pretendemos desenvolver para apoiar a compreensão de Aplicações Web. Uma ideia que tencionamos explorar é a possibilidade de a criar na forma de um *plug-in* para o ambiente Eclipse, actualmente tão em voga.

Uma outra questão a explorar na sequência deste trabalho, é a noção de arquitectura que evolui ao longo do tempo de execução, tal como introduzido na Secção 4, pois parece-nos um ponto essencial para maximizar a compreensão. Essa evolução temporal associada a diversos níveis de abstracção adaptados a cada um dos elementos a estudar, constituirão a direcção de investigação que queremos perseguir.

Referências

[AZ05] Paul Anderson and Mark Zarins. The CodeSurfer Software Understanding Platform. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 147–148, Washington, DC, USA, 2005. IEEE Computer Society.

[Bro78] Ruven Brooks. Using a behavioral theory of program comprehension in software engineering. In *ICSE '78: Proceedings of the 3rd international conference on Software engineering*, pages 196–201, Piscataway, NJ, USA, 1978. IEEE Press.

[CE04] Brendan Cleary and Christopher Exton. Chive - a program source visualisation framework. In *IWPC '04: Proceedings of the 12th International Workshop on Program Comprehension*, pages 268–270, 2004.

[DKKL04] Ng Darren, David R. Kaeli, Sergei Kojarski, and David H. Lorenz. Program comprehension using aspects. In *ICSE 2004 Workshop on Directions in Software Engineering Environments (WoDiSEE'2004)*, Boston, MA 02115, May 2004.

[FHK+97] P. J. Finnigan, R. C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. A. Muller, J. Mylopoulos, S. G. Perelgut, M. Stanley, and K. Wong. The Software Bookshelf. *IBM Systems Journal*, 36(4):564–593, Nov. 1997.

[Fie00] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. Chair-Richard N. Taylor.

[FT00] Roy T. Fielding and Richard N. Taylor. Principle design of the modern Web architecture. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 407–416, New York, NY, USA, 2000. ACM Press.

[HH01] Ahmed E. Hassan and Richard C. Holt. Towards a Better Understanding of Web Applications. In *WSE'01: Proceedings of the 3rd International Workshop on Web Site Evolution*, page 112, Washington, DC, USA, 2001. IEEE Computer Society.

[Hol97] R. C. Holt. Software Bookshelf: Overview and Construction, Mar. 1997.

[KQPR03] Michael Kolling, Bruce Quig, Andrew Patterson, and John Rosenberg. The BlueJ system and its pedagogy. *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology*, 13(4):249–268, December 2003.

[LFP+02] Giuseppe Antonio Di Lucca, Anna Rita Fasolino, F. Pace, Porfirio Tramontana, and U. De Carlini. Ware: a Tool for the Reverse Engineering of Web Applications. In *CSMR'02: Proceedings of the 6th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 2002.

[LFT04] Giuseppe Antonio Di Lucca, Anna Rita Fasolino, and Porfirio Tramontana. Reverse Engineering Web Applications: the WARE approach. *Journal of Software Maintenance and Evolution*, 16(1-2):71–101, 2004.

[LFTC02] Giuseppe Antonio Di Lucca, Anna Rita Fasolino, Porfirio Tramontana, and U. De Carlini. Comprehending Web Applications by a Clustering based approach. In *IWPC'02: Proceedings of the 10th International Workshop on Program Comprehension*. IEEE Computer Society, 2002.

[Lin] Panagiotis. K. Linos. Program comprehension tools.

[LMSW03] Rob Lintern, Jeff Michaud, Margaret-Anne Storey, and Xiaomin Wu. Plugging-in Visualization: experiences integrating a visualization tool with Eclipse. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 47–ff, New York, NY, USA, 2003. ACM Press.

[MMSBA04] André Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. Visualizing programs with Jeliot 3. In *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, pages 373–376, New York, NY, USA, 2004. ACM Press.

[MTW93] Hausi A. Muller, Scott R. Tilley, and Kenny Wong. Understanding software systems using reverse engineering technology perspectives from the Rigi project. In *CASCON '93: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research*, pages 217–226. IBM Press, 1993.

[Pac04a] Michael J. Pacione. Evaluating a Model for Software Visualisation for Software Comprehension. Technical Report Technical Report EFoCS-53-2004, Dep of Computer Science and Information Sciences, University of Strathclyde, Glasgow, UK, June 2004.

[Pac04b] Michael J. Pacione. Software visualisation for object-oriented program comprehension. In *Doctoral Symposium, Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 63–65, Edinburgh, May 2004. IEEE / ACM.

[PBS] PBS. The Portable Bookshelf.

[Per03] Maria João Tinoco Varanda Pereira. *Sistematização da Animação de Programas*. PhD thesis, Universidade do Minho, 2003.

[RW97] Vennila Ramalingam and Susan Wiedenbeck. An empirical study of novice program comprehension in the imperative and object-oriented styles. In *ESP '97: Papers presented at the seventh workshop on Empirical studies of programmers*, pages 124–139, New York, NY, USA, 1997. ACM Press.

[SM79] B. Shneiderman and R. Mayer. Syntactic semantic interactions in programmer behavior: A model and experimental results. *Intl. J. Comp. & Info. Sciences*, pages 8 – 3, 1979.

[vMV95] Annaliese von Mayrhauser and A. Marie Vans. Program comprehension during software maintenance an evolution. *Computer , Volume: 28 , Issue: 8 , Aug. 1995*, pages 44 – 55, August 1995.