Localização de Código por Demonstração

André L. Santos

Faculdade de Ciências da Universidade de Lisboa
LASIGE, Bloco C6, Piso 3, Campo Grande, 1749-016 Lisboa, Portugal

Instituto Universitário de Lisboa (ISCTE-IUL),
Av. das Forças Armadas, 1649-026 Lisboa, Portugal
andre.santos@iscte.pt

Resumo No contexto da manutenção de software, a atividade de localizar código relacionado com determinada funcionalidade pode ser dispendiosa, especialmente nos casos em que o programador não está familiarizado com a parte do sistema em questão. Este artigo apresenta um mecanismo adequado para aplicações interativas que permite localizar código através da demonstração de uma funcionalidade do programa. Desta forma, o programador expressa a sua intenção interagindo diretamente com o programa em execução, com vista a obter instantaneamente segmentos do sistema que incluem as partes do código que foram executadas dentro de determinado intervalo de tempo. O mecanismo é concretizado por via de instrumentação do programa, interligando-o com o ambiente de desenvolvimento. Uma prova de conceito foi desenvolvida para Eclipse, tomando partido de contextos de tarefa (Mylyn) para agrupar as partes do código relacionadas com a funcionalidade que se pretende localizar.

Keywords: Compreensão de programas, análise dinâmica, instrumentação, contextos de tarefa, Mylyn

1 Introdução

No contexto de sistemas de larga dimensão, a atividade de localizar código relacionado com determinada funcionalidade pode ser dispendiosa. Um programador que, por exemplo, lhe seja atribuída a tarefa de corrigir um defeito ou modificar determinada funcionalidade, irá tipicamente encontrar dificuldades em localizar as partes do código com que não está familiarizado e que são relevantes para efetuar a alteração necessária. As razões para assim ser prendem-se com o elevado número de linhas de código e módulos que um sistema complexo pode ter. Em sistemas não triviais, as funcionalidades do ambiente de desenvolvimento (IDE) para efetuar pesquisas no código poderão revelar-se ineficazes.

Associada à dificuldade de localizar partes do código sem ter nenhum ponto de partida, existe um entrave adicional causado pela eventual incerteza por parte do programador no que diz respeito a encontraras partes do código esperadas. Esta incerteza contribui em certa medida para aumentar a improdutividade pois

tenderá a fomentar processos de tentativa-erro na atividade do programador, onde este perde tempo a verificar se encontrou certeiramente determinada parte do código.

Estudos empíricos revelam que os programadores despendem uma parte muito significativa da sua atividade a navegar pelo código, proporção esta que pode atingir valores na ordem dos 30% [8]. Logo, o desenvolvimento de mecanismos de para reduzir o tempo despendido nestas atividades terão potencialmente um impacto significativo na produtividade dos programadores.

A utilização de contextos de tarefa (*task contexts* [7]) consiste numa prática de desenvolvimento integrada com o IDE que permite representar a noção de tarefa de desenvolvimento como um artefacto manipulável. A concretização mais popular e que impulsionou este mecanismo consiste na ferramenta Mylyn [1], atualmente incorporada por omissão nos pacotes do ambiente de desenvolvimento Eclipse [18]. O Mylyn permite uma gestão de tarefas não intrusiva, formando tarefas automaticamente através de associações entre os artefactos que o programador manipula e a tarefa ativa.

O conceito de contexto de tarefa revela-se extremamente útil e tem vindo a ganhar popularidade, sendo a sua inclusão por omissão no Eclipse prova disso. Contudo, embora uma ferramenta como o Mylyn seja valiosa no que diz respeito à gestão de tarefas, não auxilia o programador na localização dos artefactos que compõem as tarefas. Cabe ao programador navegar pelo código do sistema com vista a constituir o contexto de tarefa.

Este artigo propõe um mecanismo complementar aos contextos de tarefa que permite a construção automática de contextos com base na demonstração de funcionalidades do programa. Desta forma, o programador quando confrontado com uma tarefa relacionada com determinada funcionalidade, pode demonstrá-la diretamente no programa em execução, obtendo como resultado um contexto de tarefa constituído por todas partes do código (pacotes, classes, atributos, métodos) que estiveram envolvidas na execução da funcionalidade em questão.

Uma prova de conceito do mecanismo proposto foi desenvolvida para a tecnologia Java/OSGi [3], fazendo a integração no Eclipse como um complemento ao Mylyn. A concretização recorre a um componente OSGi de instrumentação para aplicar aos programas a explorar. A instrumentação, concretizada em Aspect/J [11], permite interligar o programa em execução e o IDE, de forma a que em tempo de execução sejam rastreadas as execuções de métodos para formação de um contexto de tarefa. O mecanismo pode ser ativado sem quaisquer modificações no código do programa a explorar, sendo suficiente executá-lo com o componente de instrumentação.

As vantagens do mecanismo proposto prendem-se com a capacidade de obter instantaneamente um contexto de tarefa, o qual localiza as partes do código associadas à execução de determinada funcionalidade. Desta forma, o tempo despendido na atividade de exploração de código para encontrar as partes relevantes pode ser reduzido de forma muito significativa. Mais ainda, a localização das partes do código é precisa, permitindo que o programador possa ter absoluta confiança no resultado obtido.

O resto deste artigo está estruturado da seguinte forma. A Secção 2 apresenta brevemente a noção de contextos de tarefa e a forma como é concretizada no Mylyn/Eclipse. A Secção 3 introduz a abordagem proposta, ao passo que a Secção 4 descreve a concretização da prova de conceito. Na Secção 5 são discutidos os benefícios e limitações da abordagem proposta. Por fim, a Secção 6 descreve trabalho relacionado, e a Secção 7 apresenta conclusões e trabalho futuro.

2 Contextos de tarefa (Mylyn/Eclipse)

No âmbito do Mylyn, um contexto de tarefa é um artefacto que agrupa partes do código, podendo ser gerido de forma integrada com ferramentas de gestão de desenvolvimento de software. As tarefas estão tipicamente associadas a atividades de manutenção, tais como a correção de defeitos ou incrementos/alterações de funcionalidade. O artefacto que representa a tarefa agrupa as partes do código que estão envolvidas na realização da mesma. As tarefas são guardadas num formato persistente e podem ser ativadas, fazendo com que o IDE entre num modo filtrado, onde nas diversas vistas são mostrados apenas os artefactos associados à tarefa. Desta forma, o programador é confrontado com vistas simplificadas que permitem que seja despendido menos tempo a navegar pelo código para localizar os artefactos necessários para trabalhar na tarefa. A Figura 1 apresenta uma ilustração do Mylyn em ação, onde se pode ver um contexto de tarefa ativado na vista de lista de tarefas, e consequentemente as outras vistas do IDE filtradas, mostrando apenas os artefactos associados à tarefa em questão.

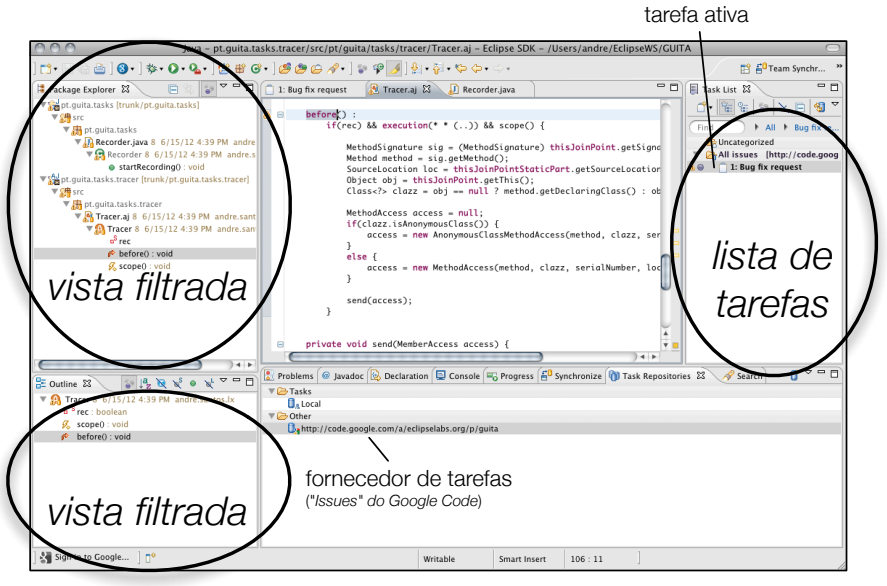

Figura 1. Contextos de tarefa no Mylyn/Eclipse.

O Mylyn permite a integração com repositórios de tarefas (e.g. Bugzilla [15]) e de gestão de ciclo de vida de aplicações (*application lifecycle management*, ALM, e.g. ScrumWorks [12]). Estes repositórios assumem o papel de fornecedores de tarefas, sendo o Mylyn consumidor dessas tarefas, mantendo associações aos artefactos de código envolvidos na realização das mesmas. Desta forma, consegue-se uma forte integração entre o IDE e software externo relacionado com gestão de projetos e controlo de qualidade.

3 Localização de Código por Demonstração

Esta secção descreve o mecanismo proposto do ponto de vista do programador que o utiliza. Os detalhes de concretização são apresentados na secção seguinte.

A localização de código por demonstração consiste num mecanismo onde os programadores podem localizar código de um programa, através da demonstração de determinada funcionalidade nesse programa. A intenção do programador no que diz respeito à parte do código que pretende localizar é expressa diretamente no programa em si, não sendo necessário recorrer a abstrações adicionais. A ideia central do mecanismo é o programador poder obter os métodos que executaram dentro de determinado intervalo de tempo, delimitado explicitamente através de um processo de gravação que é ativado e desativado pelo programador.

A aplicabilidade do mecanismo proposto pressupõe que os programas a explorar têm algum suporte de interatividade, de modo a que seja possível demonstrar funcionalidades interagindo com o programa. Contudo, o programa não tem necessariamente que ter uma interface gráfica, pois por exemplo, num programa com interface em modo de texto é possível delimitar fragmentos de execução mediante ações de interação.

Um dos requisitos principais idealizados para o mecanismo proposto consiste em não requerer nenhum tipo de adaptação "manual" do programa a explorar. Desta forma, a adoção do mecanismo proposto não implicará qualquer custo adicional em termos de desenvolvimento. Mais ainda, de modo a ser efetivo, o mecanismo foi idealizado como estando integrado num ambiente de desenvolvimento (e.g. Eclipse).

A Figura 2 ilustra a forma como um programador pode utilizar o mecanismo de localização de código por demonstração. Após lançar o programa que se pretende explorar (instrumentado), o programador toma os passos preparatórios para alcançar a funcionalidade que pretende localizar (1). Tendo um contexto de tarefa ativo, e imediatamente antes de demonstrar a funcionalidade, o modo de gravação é ativado através da ação disponível na vista de lista de tarefas (2). A partir deste momento, quaisquer métodos executados no programa irão ser acrescentados ao contexto ativo. Desta forma, o programador demonstra a funcionalidade que pretende localizar (3), e à medida que vai interagindo com o programa vai observando incrementos no contexto de tarefa ativo (4). Para terminar, o modo de gravação é desativado da mesma forma que foi ativado (5). Embora a apresentação filtrada dos elementos do código seja por si só eficaz para ajudar a localizar o código relacionado com a funcionalidade em questão, o

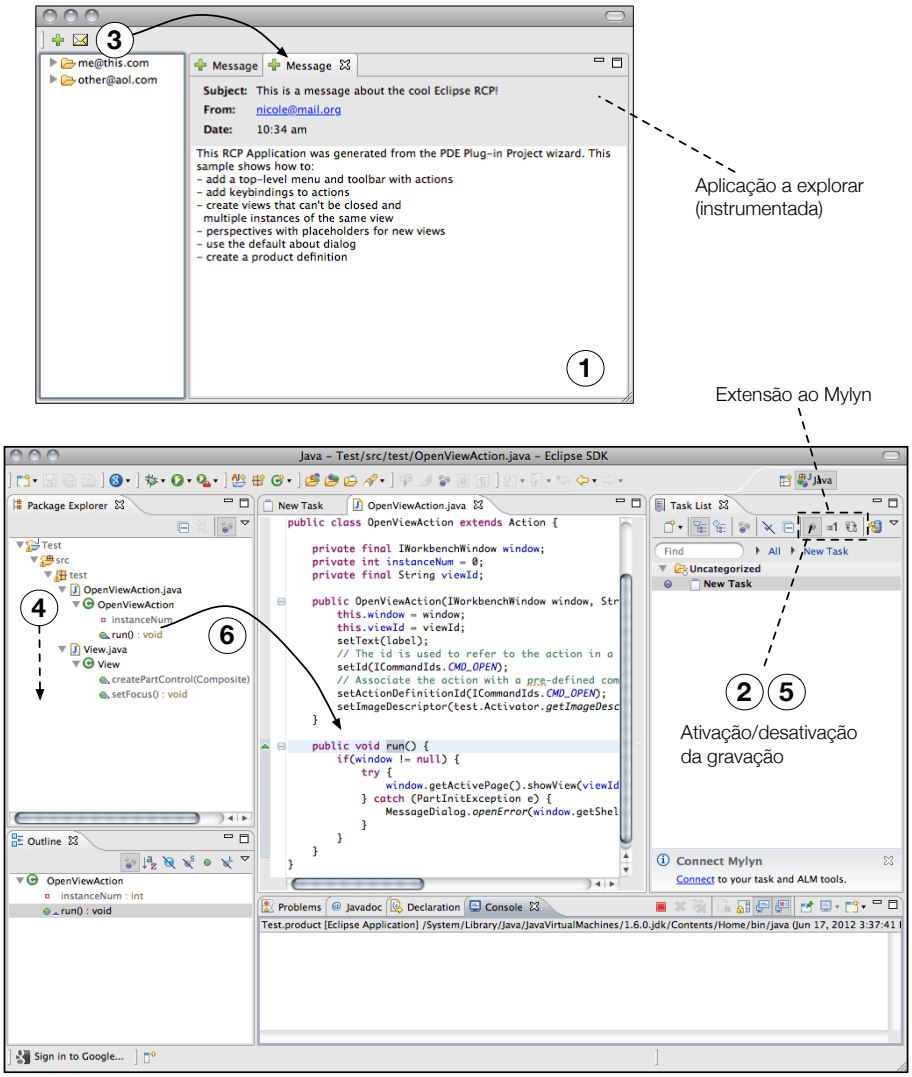

Figura 2. Ilustração do processo de utilização do mecanismo proposto: (1) aplicação em execução; (2) o modo de gravação de contexto de tarefa é ativado no IDE; (3) a funcionalidade cujo código se pretende inspecionar é demonstrada na aplicação em execução; (4) os elementos do contexto de tarefa vão sendo incluídos à medida que a interação prossegue; (5) o modo de gravação é desativado; o processo termina tendo sido obtido um contexto de tarefa composto por todos os métodos que executaram durante o período de gravação; (6) o programador pode então explorar o contexto obtido com vista a localizar o código pretendido.

mecanismo proposto prevê a possibilidade de iterar sobre os métodos capturados no contexto pela ordem temporal pela qual foram executados. Este mecanismo consiste num auxílio que guia o programador pelo contexto capturado.

Um caso de uso típico deste mecanismo consistiria no seguinte. Ao ser atribuída uma tarefa de manutenção a um programador relacionada com determinada funcionalidade, a qual seria listada na lista de tarefas e proveniente de um fornecedor externo (e.g. Bugzilla), o programador analisaria a solicitação em questão. Após lançar o programa (instrumentado), ativa o modo de gravação e demonstra a funcionalidade associada à tarefa. Ao terminar a gravação, o programador tem ao seu dispor um contexto de tarefa que pode examinar com vista a alcançar a parte do código pretendida.

4 Concretização: GUITA Toolkit

Foi desenvolvida uma prova de conceito do mecanismo de localização de código por demonstração no protótipo *GUITA Toolkit*. O protótipo suporta a tecnologia Java/OSGi e está integrado no ambiente de desenvolvimento Eclipse. Logo, a concretização aqui descrita é adequada para localizar código de aplicações incorporadas em componentes OSGi.

O processo de rastreabilidade descrito na secção anterior é concretizado por via de instrumentação. Desta forma, a solução técnica tem como base um componente OSGi cujo papel é instrumentar as aplicações a explorar, fazendo a interligação com o ambiente de desenvolvimento. Para tal, as aplicações têm que ser executadas na *framework* OSGi com o componente instrumentador ativado, não sendo contudo necessário efetuar quaisquer modificações manuais às mesmas.

A instrumentação é feita com recurso a programação orientada para aspetos em Aspect/J. É introduzido no programa a capacidade de receber pedidos da ativação de rastreabilidade via *sockets*. Ao ser ativado o processo de rastreabilidade, a aplicação envia pacotes com informação dos métodos que são executados. Por omissão, não é feita qualquer filtragem de métodos. Contudo, é possível restringir o âmbito de instrumentação de forma a excluir partes do código identificadas à partida como irrelevantes para a exploração. Desta forma, é possível reduzir o número de métodos que constituirá o contexto de tarefa.

A integração no Eclipse é feita através de um componente que estende o Mylyn com as ações que permitem construir contextos de tarefa através do processo de gravação descrito na secção anterior. Ao ativar a gravação, o componente comunica com a aplicação, solicitando o envio de pacotes de rastreabilidade. Através da integração com o Mylyn, esses pacotes são tratados de forma a aumentar incrementalmente o contexto de tarefa ativo.

A Figura 3 apresenta um resumo das relações principais entre os componentes da abordagem proposta, a aplicação a explorar, e os componentes OSGi, Eclipse, e Mylyn. A concretização aqui proposta foi testada na plataforma Equinox ([5], implementação do standard OSGi do Eclipse), com suporte para *weaving* de aspetos.

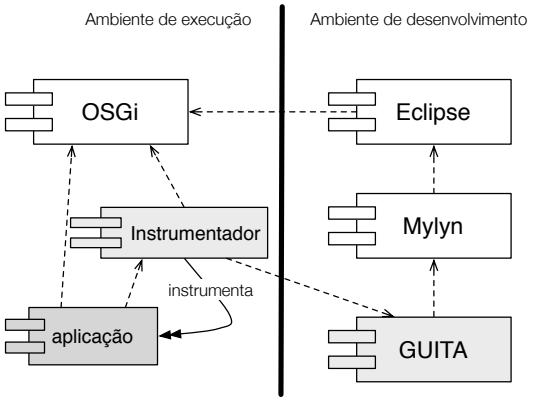

Figura 3. Componentes GUITA e sua relação com Mylyn, Eclipse, OSGi, e aplicações a inspecionar. (dependências transitivas)

5 Benefícios e Limitações

O principal benefício do mecanismo proposto de localização de código por demonstração reside no facto de poder ser encontrado código instantaneamente apenas interagindo com a aplicação que se pretende explorar. O ganho em termos de tempo despendido, por comparação com navegação "manual" pelo código, terá tendência para ser maior quando estamos perante um caso onde o programador não está familiarizado com a parte do código da aplicação. Mais ainda, será de esperar que quanto maiores forem os sistemas, mais significativo será o benefício de utilizar o mecanismo proposto. Como justificação para esta intuição, argumentamos que o tempo necessário para localizar funcionalidades é constante independentemente do tamanho do sistema, dado que a atividade consiste apenas em demonstrar a funcionalidade.

Para além da possibilidade de localizar código instantaneamente, outra vantagem deste processo é ser livre de erros. Ou seja, os métodos localizados foram os que executaram, sem qualquer margem de incerteza. Por contraste à localização "manual", isto é uma vantagem visto que é comum um programador enveredar num processo de tentativa-erro para garantir que encontrou a parte do código correta.

A abordagem proposta tem a desvantagem de só ser aplicável em programas interativos, ou que possam ser executados interativamente. Nos casos de programas que executam continuamente ou de uma vez só, torna-se impossível de delimitar determinadas partes da execução sem recorrer a mecanismos adicionais que envolvam algum conhecimento do código.

Uma limitação óbvia do mecanismo proposto prende-se com a impossibilidade de localizar código relacionado com uma funcionalidade que o programador não sabe demonstrar ou que não se consegue facilmente isolar com as formas de interação disponíveis no programa. No caso de programas que processam um número

elevado de eventos dado um nível reduzido de interatividade do utilizador (e.g. programas que reagem a todos os eventos do rato), a usabilidade do mecanismo proposto tenderá a ser mais fraca, dado que poderá ser difícil excluir do processo de gravação execuções de métodos não relacionadas com a funcionalidade que se pretende localizar.

O mecanismo proposto não pretende ser uma alternativa a um depurador convencional, e logo, não se pretende dar continuidade a este trabalho no sentido de considerar as funcionalidades de execução passo-a-passo ou outras existentes em depuradores. Pelo contrário, o mecanismo proposto pode funcionar como um complemento que auxilia a encontrar localizações no código com vista a definir pontos de paragem (*breakpoints*), os quais de outra forma podem ser difíceis de encontrar sem ter um ponto de partida e algum conhecimento sobre o código do programa em questão.

6 Trabalho Relacionado

A solução adotada no GUITA Toolkit pode ser considerada uma forma de segmentação dinâmica de programas (*dynamic program slicing* [10]). Outras abordagens fizeram também uso de programação orientada por aspetos para segmentação dinâmica (e.g. [6], [2], [19]), porém, não abordando a segmentação com base na demonstração de funcionalidades.

A abordagem de *Software Reconnaissance* [17] consiste numa técnica para localizar código que se baseia na execução de testes unitários relacionados com as funcionalidades que se pretende localizar. Esta técnica também recorre a instrumentação para produzir traços de execução, os quais são analisados com vista a localizar código. Como resultado, o programador obtém a informação de quais os componentes que executaram código relacionado com a funcionalidade. Por comparação com a nossa abordagem, Software Reconnaissance pressupõe a existência de testes unitários relacionados com as funcionalidades a localizar, o que poderá consistir numa desvantagem caso os mesmos não estejam disponíveis. Por outro lado, o resultado obtido é menos granular (componentes) do que o resultado que se obtém com um contexto de tarefa do Mylyn, facto que também pode ser visto como uma desvantagem.

Em [16], os autores propõem segmentar pedaços de execução em *UI traces* delimitados por eventos de interação com o utilizador. Por exemplo, um *UI trace* incluiria toda a sequência de chamadas entre o evento de pressionar um botão e uma atualização do ecrã. Desta forma, um programador pode obter segmentos de código após demonstrar uma funcionalidade. Contudo, ao contrário da abordagem proposta, a delimitação não obedece a uma lógica de processo de gravação, implicando que o programador terá que procurar pelos diversos segmentos obtidos a parte do código pretendida.

O Whyline [9] consiste numa ferramenta de depuração que permite aos programadores colocar interrogações sobre o comportamento do programa diretamente na visualização do resultado de saída do mesmo. As interrogações disponíveis obedecem à forma “porque é que aconteceu ...” ou “porque é que não

aconteceu ...". A utilização da ferramenta implica gravar um fragmento da execução do programa, tal como na nossa abordagem. Contudo, a análise que deriva as interrogações que podem ser colocadas ao programa tem que ser realizada externamente ao programa. O fragmento de execução é carregado no Whyline e um histórico do resultado de saída é recriado, onde o programador pode selecionar um elemento da interface gráfica e fazer uma das interrogações sugeridas pelo Whyline. Por comparação com a nossa abordagem, as desvantagens do Whyline prendem-se com a impossibilidade de executar o programa continuamente e solicitar localizações a qualquer momento. Por outro lado, as interrogações disponibilizadas pelo Whyline foram concebidas com o intuito de descobrir a causa de defeitos no programa, e não com o propósito de localizar código. Embora o Whyline possa ser utilizado para este propósito, não consiste numa solução otimizada para tal.

Cenários Visuais [4] consiste numa abordagem para gerar documentação de casos de utilização por via de demonstração, podendo produzir diagramas de sequência e manuais estruturados em função do resultado visível do programa (ecrã). Este mecanismo permite reproduzir a visualização de uma funcionalidade em sincronia com um diagrama de sequência que vai sendo apresentado passo-a-passo, à medida que a reprodução se desenrola. Numa abordagem relacionada [14] e baseada no mesmo protótipo, é proposta uma abordagem para recuperação automática de modelos comportamentais com base na demonstração de funcionalidades. Estas abordagens podem ser utilizadas com o intuito de localizar funcionalidades, pois não requerem familiarização prévia com o código. Contudo, as ferramentas propostas não foram concebidas com o objetivo de ter uma solução integrada com o ambiente de desenvolvimento que permita a localização instantânea de segmentos de código, permitindo a combinação das atividades de exploração e desenvolvimento. Por outro lado, estas abordagens foram idealizadas com o propósito de documentação de sistemas legados.

Num trabalho anterior do mesmo autor [13], foi proposto um mecanismo para localizar no código os objetos visuais da interface gráfica de um programa, com base num mecanismo que, tal como o apresentado neste artigo, tira partido do próprio programa em execução como meio para o programador exprimir a sua intenção. É permitido ao programador localizar as linhas de código relacionadas com os objetos visuais da interface gráfica durante o seu ciclo de vida (desde a criação até à última operação invocada). Esta abordagem restringe o seu âmbito à interface gráfica, mas por outro lado é mais especializada para esse efeito. Os dois mecanismos podem ser utilizados em combinação.

7 Conclusões e Trabalho Futuro

Neste artigo foi proposto um mecanismo para compreensão de programas que permite a construção automática de contextos de tarefas, o qual se apelidou de localização de código por demonstração. Como prova de conceito, o mecanismo proposto foi concretizado para a tecnologia Java/OSGi e integrado com o Mylyn/Eclipse. A principal inovação subjacente prende-se com a possibilidade

de, sem qualquer custo adicional, poder localizar código apenas demonstrando a funcionalidade diretamente no programa que se pretende explorar.

O passo seguinte neste trabalho de investigação será planear e levar a cabo um estudo empírico que consiga de forma mensurável avaliar o mecanismo proposto. Dados determinados programas, tal experiência passará por definir tarefas de manutenção e comparar o desempenho de programadores entre um grupo experimental que utiliza localização de código por demonstração e um grupo de controlo que recorre apenas às funcionalidades atualmente existentes no ambiente de desenvolvimento (depurador, pesquisas, etc). Espera-se que o estudo empírico possa evidenciar que à medida que o tamanho e complexidade dos programas a explorar aumenta, o benefício de utilizar o mecanismo proposto seja mais expressivo. Isto deverá acontecer por analogia ao caso do mecanismo de contextos de tarefa em si, cuja utilização em programas de pequena dimensão não consistirá certamente numa mais-valia inquestionável.

Referências

1. Mylyn Eclipse component. http://www.eclipse.org/mylyn/, 2012.
2. Andrew David Eisenberg and Kris De Volder. Dynamic feature traces: Finding features in unfamiliar code. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 337–346, Washington, DC, USA, 2005. IEEE Computer Society.
3. OSGi Standard for Java component systems. http://www.osgi.org/.
4. Vítor Gouveia. Cenários visuais: Rastreio de requisitos, documentação e animação para sistemas legados. Master's thesis, Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia, Departamento de Informática, 2008.
5. Equinox OSGi implementation. http://www.eclipse.org/equinox/.
6. T. Ishio, S. Kusumoto, and K. Inoue. Program slicing tool for effective software evolution using aspect-oriented technique. In *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of*, pages 3 – 12, sept. 2003.
7. Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 1–11, New York, NY, USA, 2006. ACM.
8. Andrew J. Ko, Htet Aung, and Brad A. Myers. Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 126–135, New York, NY, USA, 2005. ACM.
9. Andrew J. Ko and Brad A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 301–310, New York, NY, USA, 2008. ACM.
10. Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155 – 163, 1988.
11. Aspect/J programming language. http://www.eclipse.org/aspectj/.
12. ScrumWorks project management tool. http://www.collab.net/products/scrumworks.
13. André L. Santos. GUI-driven code tracing. In *VL/HCC: IEEE Symposium on Visual Languages and Human-Centric Computing*, Innsbruck, Austria, September 2012.

14. Filipa Silva. Recuperação automática da modelação comportamental com aplicações ao ensaio baseado em modelos. Master's thesis, Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia, Departamento de Informática, 2008.
15. Bugzilla software management tool. http://www.bugzilla.org/.
16. Andrew Sutherland and Kevin Schneider. UI traces: Supporting the maintenance of interactive software. In *Proceedings of the 27th international conference on software maintenance*, pages 563–566, 2009.
17. Norman Wilde and Christopher Casey. Early field experience with the software reconnaissance technique for program comprehension. In *Proceedings of the 1996 International Conference on Software Maintenance*, ICSM '96, pages 312–318, Washington, DC, USA, 1996. IEEE Computer Society.
18. Eclipse workbench. http://www.eclipse.org/.
19. Liu Xi, Miao Li, Zhao Dan, and Li Wei. An approach of coarse-grained dynamic slice for java program. In *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on*, pages 670 –674, may 2011.