Desenho por Contrato: Um Ciclo de Vida

Ricardo Alves, Sérgio Bryton

Resumo — Um dos grandes objectivos da Engenharia de *Software* é produzir *software* com qualidade. Significa isto que propriedades como robustez, correcção, reutilização, manutenção, compatibilidade, eficiência, portabilidade e funcionalidade, entre outras, têm que estar presentes num produto de *software* com qualidade. O Desenho por Contrato (DpC) com *Object Constraint Language* (OCL) e a Programação por Contrato (PpC) são reconhecidamente instrumentos potenciadores de qualidade que actuam em dois extremos, respectivamente no modelo e no código. Estabelecer a ponte entre estes dois extremos é um passo importante para o aumento da qualidade do produto final. Este artigo pretende constituir uma análise do estado da arte no que respeita às ferramentas que possibilitam modelação em *Unified Modelling Language* (UML) com OCL e implementação de DpC com a linguagem *Java* e que possam sustentar essa ponte.

Tópicos — OCL, Desenho por Contrato, UML, *Java*.

1 Introdução

O DpC é uma técnica que permite o desenvolvimento de *software* com qualidade à custa da expressão de asserções, baseadas na cooperação entre cliente e fornecedor sob a forma de um contrato.

O UML tem vindo a tornar-se a linguagem de modelação mais utilizada para especificar, visualizar, construir e documentar o desenvolvimento de *software*. No entanto, para expressar formalmente propriedades, como por exemplo a unicidade, os métodos de análise e desenho percursores do UML (OMT, Booch e Objetory) não tinham mecanismos que permitissem expressar tais propriedades. Perante esta necessidade premente, foi desenvolvida uma linguagem – o OCL, com o intuito de colmatá-la. Com o OCL embebido no UML, aumenta-se o rigor no modelo através da definição formal de asserções (pré-condições, pós-condições e invariantes), que poderão ser implementadas sob a forma de cláusulas do contrato.

Gerar código automaticamente, incluindo o DpC especificado em OCL, originará uma implementação mais rápida e eficiente e a garantia da rastreabilidade entre o modelo e o código correspondente, assentes nos princípios de qualidade que ambos (DpC e PpC) pretendem acrescentar, dando-se desta forma um passo importante na direcção da qualidade do produto final. Para suporte deste trabalho, foi seleccionada a linguagem *Java* dada a sua aceitação e utilização nos meios académicos e comerciais.

A geração automática de código *Java* a partir do modelo UML é suportada actualmente por uma série de ferramentas. No entanto, no que diz respeito ao modelo UML com expressões OCL, ainda existe um longo caminho a percorrer para que tenhamos ferramentas que, automaticamente consigam gerar código *Java* para as cláusulas do contrato expressas em OCL e deste modo estabelecer a ponte entre estes dois extremos.

O objectivo deste artigo é comprovar em que estágio se encontra a tecnologia através da comparação de uma série de ferramentas existentes, que possam sustentar essa ponte.

- *Ricardo Alves exerce a sua actividade profissional na multinacional Amcor. E-mail: ricardo.alves@amcor-flexibles.com.*
- *Sérgio Bryton exerce a sua actividade profissional na Direcção de Tecnologias de Informação e Comunicação da Marinha Portuguesa. E-mail: dias.marques@marinha.pt.*

O trabalho subjacente a este artigo foi desenvolvido no âmbito do Mestrado em Engenharia Informática, na cadeira de *Qualidade do Produto e do Processo*[1], sob a orientação do Prof. Fernando Brito e Abreu.

A esta introdução segue-se uma contextualização ao tema principal, onde apresentaremos um caso de estudo juntamente com uma pequena introdução aos temas que servem de suporte ao tema principal. Seguidamente, apresentamos o estudo comparativo de algumas das ferramentas existentes. Prosseguimos com a implementação do caso de estudo com a alternativa tecnológica que nos assegure maior qualidade no produto final, apresentando finalmente as nossas conclusões.

2 Contexto

2.1 Caso de Estudo

No sentido de dar apoio às várias secções deste artigo desenvolvemos um caso de estudo, inspirado nos transportes fluviais de passageiros que estabelecem a ligação entre as margens do rio Tejo. A figura 1 ilustra o diagrama de classes em UML.

Os cacilheiros são utilizados para o transporte de passageiros entre as margens Norte e Sul do rio Tejo. Cada cacilheiro faz várias viagens e cada viagem é efectuada apenas

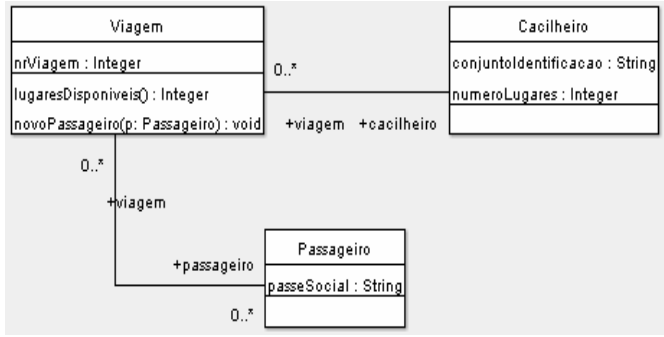

Fig. 1. Diagrama de Classes do modelo "Viagens de Cacilheiro".

[1] http://ctp.di.fct.unl.pt/mei/qpp

por um cacilheiro. Cada embarcação destas tem uma capacidade limitada quanto ao número de passageiros que pode transportar. Em cada viagem é transportado um determinado número de passageiros que não pode exceder o número de lugares disponíveis do cacilheiro que lhe está associado. Cada cacilheiro tem atribuído um conjunto de identificação único assim como cada viagem é identificada por um número de viagem único. Um passageiro não pode realizar mais do que uma viagem simultaneamente.

2.2 Desenho por Contrato

O paradigma do desenvolvimento de *software* orientado pelos objectos (DSOO) introduziu os conceitos de classes, objectos, abstracção, herança, polimorfismo e genericidade que se traduziram num passo importante na constante procura da qualidade do produto na engenharia de *software*. No entanto, expressar alguns factores de qualidade na programação, no sentido de tornar os programas legíveis, bem especificados e validados e entender as regras que estão por trás da sua forma de implementação, é algo que a maioria das ferramentas de apoio ao desenvolvimento orientado pelos objectos não contemplava. Esta necessidade despoletou o surgimento do DpC.

pós-condições e invariantes permite-nos exprimir de uma maneira formal as especificações do contrato. Falta ainda, obviamente, especificar as consequências e respectivo tratamento das excepções às cláusulas contratuais.

O DpC no DSOO resume-se à utilização sistemática de pré-condições, pós-condições e invariantes. As pré-condições são uma obrigação da operação cliente, em benefício da operação fornecedor; as pós-condições são uma obrigação da operação fornecedor, em benefício da operação cliente. Ambas, descrevem as propriedades de cada método individualmente. Os invariantes expressam as propriedades globais das instâncias de uma classe, as quais terão que ser preservadas por todas as operações dessa classe [2]. Mesmo havendo um contrato entre as classes, algo inesperado pode ocorrer que o viole, ou seja, uma excepção. Embora, teoricamente, num sistema construído com qualidade nenhum contrato possa ser violado, isso não acontece na realidade. Nesse sentido, terão que existir mecanismos de tratamento de excepções.

Na figura 2 está ilustrado um exemplo, de como se pode redigir um contrato na fase de implementação, utilizando uma linguagem de programação[2] que suporte a inclusão de cláusulas contratuais expressas em pré-condições, pós-condições, invariantes e tratamento de excepções. Conclui-se, facilmente, que a utilização do DpC elimina, por completo, a necessidade da programação defensiva, na qual as validações da utilização do serviço estão todas do lado do fornecedor não havendo obrigações por parte do cliente.

TABELA 1
EXEMPLO DE UM CONTRAT O

	Obrigações	Benefícios
Cliente	Tem que assegurar as pré-condições: 1. bilhete comprado. 2. estar no cacilheiro 5 minutos antes da partida.	Da pós-condição: 1. estar no local pretendido às na hora prevista no horário com atraso mínimo de 10 minutos.
Fornecedor	Tem que assegurar a pós-condição: 1. transportar o cliente para o local pretendido até 10 minutos depois da hora prevista no horário.	Da pré-condição: 1. não ter prejuízo por transportar clientes sem bilhete comprado e não partir com atraso.

O conceito chave do DpC assenta essencialmente na expressão das relações entre uma classe (fornecedor) e os seus clientes sob os auspícios de um acordo formal, contemplando os direitos e obrigações de ambas as partes. Somente com a definição precisa de todas as reivindicações e responsabilidades respeitantes a cada módulo pertencentes a um determinado sistema, se lhe pode depositar um alto grau de confiança [1].

Fazendo um paralelismo com os contratos efectuados, por exemplo entre duas organizações, podemos inferir algumas propriedades importantes de um contrato:

1. são firmados entre duas ou mais entidades, em que cada uma assume o papel de cliente ou fornecedor;
2. é explicitamente escrito numa linguagem conhecida por ambas as partes;
3. especifica as obrigações e benefícios mútuos;
4. as obrigações de uma das partes são os benefícios da outra e vice-versa;
5. não contem cláusulas escondidas;
6. muitas vezes faz referência, implícita ou explicitamente, a regras comuns a todos os contratos (leis vigentes, regulamentos oficiais, etc.).

Tomando por exemplo o contrato entre a empresa que assegura as travessias do rio Tejo (fornecedor) e os seus passageiros (cliente) podemos formalizar alguns dos benefícios e obrigações correspondentes como está expresso na tabela 1.

Como podemos observar a utilização de pré-condições,

O conceito de herança é largamente utilizado no DSOO. De modo a que a utilização deste conceito não invalide as cláusulas do DpC, a redefinição de uma operação apenas pode substituir a pré-condição original por uma igual ou mais fraca (exigindo menos do cliente) e a pós-condição original por uma igual ou mais forte (oferecendo mais ao cliente) [30]. Significa isto, que uma subclasse não é mais do que uma subcontratação da parte da classe pai e, por conseguinte, tem que, obrigatoriamente, satisfazer no mínimo, o mesmo contrato.

A utilização do conceito de contrato pode ser utilizado em todo o ciclo de vida do processo de DSOO. Assim a utilização de pré-condições, pós-condições e invariantes começa na modelação do sistema descrevendo os seus eventos. Na fase de análise e na fase de implementação os contratos definem as operações, os benefícios e as obrigações de cada componente do sistema, respectivamente. Finalmente, na fase de testes, os contratos definem a correcta especificação de cada componente [3].

[2] A linguagem *Eiffel* tem mecanismos nativos de suporte ao DpC.

O DpC é suportado pelo UML através de uma linguagem, o OCL, que permite especificar pré e pós-condições para as operações e invariantes para as classes.

Fig. 2. Exemplo de um contrato em *Eiffel*.

2.3 Object Constraint Language - OCL

O OCL evoluiu de uma linguagem de expressões do método Syntropy que era uma linguagem de modelação de processos de negócio utilizada na IBM, até ser integrada, em 1997, no UML [5].

A modelação de sistemas de *software* é tradicionalmente sinónimo de construção de diagramas. Os diagramas, simplesmente, não conseguem expressar toda a informação necessária a uma especificação completa [19]. Por exemplo, no diagrama de classes ilustrado na figura 5, podíamos concluir que o número de passageiros de um cacilheiro seria ilimitado. Obviamente que essa conclusão não é verdadeira, mas não conseguimos expressar no diagrama que o número de passageiros está limitado ao número de lugares disponíveis no cacilheiro. Contudo, tal limitação pode ser expressa com recurso a uma invariante no contexto da classe Viagem, expresso em OCL da seguinte forma:

Exemplo em OCL para expressar a restrição de multiplicidade da classe Viagem.

```
Context Viagem
  inv:passageiros->size() <= cacilhero.numeroDeLugares
```

Em virtude do exposto anteriormente surge a necessidade da integração do OCL, uma linguagem formal (sem a complexidade das tradicionais linguagens formais), no UML uma linguagem de modelação. Além disso expressões formais, dada a sua base matemática, não dão azo a interpretações ambíguas e podem ser verificadas automaticamente por ferramentas. Além de utilizar uma notação similar às linguagens de programação orientadas pelos objectos, é uma linguagem de expressões pura. Mais, as expressões em OCL não têm efeitos colaterais, ou seja, não conseguem alterar o estado do sistema, embora possam referenciá-lo.

O OCL pode ser utilizado para os seguinte propósitos [21]:

1. especificar invariantes em classes e tipos no modelo de classes;
2. descrever pré e pós-condições nas operações;
3. especificar a semântica de operações do tipo selector.

A utilização de asserções, formalizadas através do OCL, permite-nos expressar os conceitos do DpC na fase de análise e desenho, utilizando ferramentas de modelação em UML que tenham suporte OCL.

3 AS FERRAMENTAS: UM ESTUDO COMPARATIVO

Esta secção consiste na análise de uma série de ferramentas cujas características possam sustentar a ponte entre a modelação com UML e OCL e a correspondente implementação de código *Java* com PpC, bem como na eleição daquela ou daquelas que permitam alcançar esse objectivo originando um produto final com mais qualidade.

Assim sendo, para estabelecer essa ponte, uma ferramenta ou a utilização conjunta de várias, terão de assegurar três aspectos essenciais:

1. a modelação com UML e OCL;
2. a geração de código *Java* a partir de UML com cláusulas de OCL;
3. a implementação de PpC em *Java* a partir das restrições definidas com OCL.

As ferramentas foram seleccionadas para análise neste artigo, com base na possibilidade de, à custa das características que apresentam, viabilizarem todos ou apenas alguns dos aspectos mencionados anteriormente, bem como a sua representatividade no segmento de mercado em que se inserem .

A tabela 2 contém a descrição abreviada de cada uma das ferramentas seleccionadas. Para comparar as várias ferramentas, foram identificadas e analisadas as características que podem conferir sustentabilidade ao processo pretendido. A tabela 3 apresenta para cada uma das ferramentas seleccionadas a presença total (S) ou parcial (P) e ausência (N) de cada uma dessas características.

TABELA 2
DESCRIÇÃO ABREVIADA DAS FERRAMENTAS SELECCIONADAS

Ferramentas	Descrição Abreviada
ArgoUML [6]	Ferramenta de análise e desenho em UML com suporte para OCL e verificação da sintaxe. Gera código *Java* onde são inseridas etiquetas especiais em *Javadoc* com as cláusulas de OCL.
Poseidon [8]	Ferramenta de análise e desenho em UML com suporte para OCL.
MagicDraw [10]	Ferramenta de análise e desenho em UML com suporte para OCL.
Sparx Enterprise Architect [11]	Ferramenta de análise e desenho em UML com suporte para OCL.
jmsAssert [12]	Ferramenta de implementação de DpC para *Java*. O DpC é especificado na fonte, com etiquetas especiais dentro do *Javadoc*. O jmsAssert é executado com a fonte e são gerados os ficheiros de contrato em *jmscript* que irão ser chamados durante a execução para implementar as cláusulas.
iContract[3]	Ferramenta de implementação de DpC para *Java*. A fonte é processada primeiro com o iContract e o resultado é compilado com o compilador de *Java* tradicional. As directivas de iContract residem em comentários semelhantes às directivas de *Javadoc* [25] [27].
Jass [13]	Ferramenta de implementação de DpC para *Java*. Traduz um ficheiro .jass ou *Java* com as asserções especificadas em *Javadoc* para um ficheiro *.java*. Este ficheiro deverá por sua vez ser compilado com o compilador de *Java* tradicional.
jContractor [9]	Ferramenta de implementação de DpC para *Java*. Os contratos são escritos como métodos *Java* normais seguindo uma convenção de nomes intuitiva.
Dresden OCL Toolkit [14]	Compilador de *Java* com cláusulas de OCL. Partindo da fonte com as cláusulas OCL escritas em *Javadoc*, produz código *Java* adequado à sua implementação que introduz em cada classe.
jContract [15]	Ferramenta de implementação de DpC para *Java*. As asserções são descritas à custas de etiquetas especiais em *Javadoc* e posteriormente são pré-processadas e compiladas num processo único. Para a compilação recorre ao compilador nativo da JVM (*Java Virtual Machine*).
Octopus [16]	*Plugin* do *Eclipse* que permite fazer a análise e o desenho em UML, implementar asserções com OCL, verificação de sintaxe de OCL e geração automática de código *Java* com adição de código de suporte às asserções.
OCLe [17]	Ferramenta de análise e desenho em UML, com suporte a OCL e verificação de sintaxe e geração de código *Java* com adição de código de suporte às asserções.
OCL2j [18]	Ferramenta de implementação de DpC com OCL em *aspect-oriented programming* (AspectJ). Recebe um ficheiro XMI ou a *fonte* com as cláusulas de OCL implementadas em *Javadoc* e gera o código *Java* e os aspectos necessários.

[3] O recurso da *web* não se encontra disponível.

TABELA 3
PRINCIPAIS CARACTERÍSTICAS DAS FERRAMENTAS SELECCIONADAS

Ferramentas	Características								
	Modelação com UML	**DpC [4]**	**Verificação sintaxe OCL**	***Forward Engineering*[5]**	**PpC[6]**	**Pré-processamento[7]**	**Compilação**	**Testes**	**Observações**
ArgoUML	S	S	P	S	@precondition @postcondition @invariant	N	N	N	
Poseidon	S	S	N	S	S	N	N	N	Não inclui OCL na fonte gerada.
MagicDraw	S	S	S	S	S	N	N	N	
Sparx Enterprise Architect	S	S	N	S	S	N	N	N	Não inclui OCL na fonte gerada.
jmsAssert	N	N	N	N	@inv @post @inv	S	N	N	
iContract	N	P[8]	P	N	@inv @post @inv	S	N	N	
Jass	N	P[9]	P	N	require ensure invariant check	S	N	N	
jContractor	N	N	N	N	aMethod_Precondition() aMethod_Postcondition() aMethod_Invariant()	N	N	N	
Dresden OCL Toolkit	N	S	S	N	@precondition @postcondition, @invariant	S	S	N	
jContract	N	N	N	N	@inv @post @inv @assert: $forall, $implies, $exists	S	S	N	
Octopus	S	S	S	S	S	N	N	N	Características baseadas apenas na informação dos autores.
OCLe	S	S	S	S	S	N	N	N	
OCL2j	N	S	S	S	S	N	N	N	Protótipo não disponível à data de redacção.

Do estudo comparativo efectuado, podemos concluir que as funcionalidades apresentadas pelas ferramentas analisadas podem constituir três grandes grupos. Um primeiro grupo (Grupo 1) com as funcionalidades relacionadas com a modelação UML e especificação de cláusulas OCL. Um segundo grupo (Grupo 2) com as funcionalidades relacionadas com a produção ou utilização de código fonte *Java*, com as cláusulas de OCL de algum modo aí introduzidas. E, finalmente, um terceiro grupo (Grupo 3) com o conjunto

[4] Possibilidade de expressar restrições de OCL no modelo.
[5] Possibilidade de gerar código *Java* a partir do modelo.
[6] Possibilidade de implementação de pré-condições, pós-condições e invariantes na fonte.
[7] A fonte com as cláusulas de DpC é pré-processada, dando origem a outra fonte que será posteriormente compilada.
[8] Apenas contempla: implies, forAll, exists.
[9] Apenas contempla: forAll, exists.

das funcionalidades relacionadas com a produção do código *Java* executável. A tabela 4 apresenta as ferramentas analisadas e os grupos de funcionalidades onde se inserem.

Quanto às ferramentas que implementam funcionalidades no âmbito do primeiro grupo, importa referir que as distinções existem essencialmente entre o suporte de OCL e com que extensão, a verificação da sintaxe de OCL, e a validação das asserções contra o modelo UML existente.
Para implementar as funcionalidades descritas no segundo grupo, as ferramentas adoptam essencialmente três metodologias distintas. Existem ferramentas que produzem ou

TABELA 4
DISTRIBUIÇÃO DAS FERRAMENTAS PELOS GRUPOS DE FUNCIONALIDADES DEFINIDOS

Grupo 1	Grupo 2	Grupo 3
ArgoUML	ArgoUML	jmsAssert
Poseidon	jmsAssert	iContract
MagicDraw	iContract	jass
Sparx Entrepise	jass	Dresden OCL Toolkit
Architect	Dresden OCL Toolkit	jContract
Octopus	jContract	Octopus
OCLe	Octopus	OCLe
	OCLe	OCL2j

utilizam código *Java* com as cláusulas de OCL embebidas em *Javadoc*. Existem outras que produzem ou utilizam código *Java*, com as cláusulas de OCL implementadas também em *Java*, dentro das classes a que dizem respeito. E existem ainda outras ferramentas, que produzem ou utilizam código *Java*, sendo as cláusulas de OCL implementadas à parte, sob a forma de *scripts* ou aspectos.

No que diz respeito às ferramentas que implementam as funcionalidades do terceiro grupo, estas distinguem-se entre as que recorrem ao compilador tradicional de *Java*, as que recorrem ao compilador tradicional de *Java* adicionando-lhe parâmetros de compilação extraordinários, as que recorrem a compiladores próprios e as que recorrem a compiladores de aspectos.

De todas as ferramentas analisadas, poderíamos escolher apenas uma ou a combinação de várias para concretizar os objectivos pretendidos. No entanto, esta última opção levantaria algumas questões de compatibilidade entre as ferramentas a utilizar. Como o OCLe é a única ferramenta disponível que contempla todas as funcionalidades necessárias a este processo, mereceu de imediato a nossa atenção, sendo as restantes relegadas para segundo plano. Mesmo colocando de lado as questões de compatibilidade mencionadas anteriormente, o facto de se trabalhar numa única ferramenta é menos propício a falhas no processo de transição do desenho para a implementação, para além de implicar uma curva de aprendizagem significativamente menor. Além disso, o OCLe é a única ferramenta que valida as asserções contra o modelo UML, para além da comum validação sintáctica. Mais, já contempla o OCL 2.0 e gera automaticamente código *Java* correspondente às asserções especificadas, implementado quase na totalidade o OCL. Pelas razões anteriormente indicadas, o OCLe afigura-se como a alternativa tecnológica que nos assegurará maior qualidade no produto final, pelo que é a ferramenta eleita para implementar o caso de estudo.

4 CASO DE ESTUDO

A demonstração do processo, que nos leva da modelação em UML já com suporte ao DpC com OCL, passou pela utilização do modelo "Viagem Cacilheiro" cujo diagrama de classes está representado na figura 6. Para este fim, utilizámos a ferramenta eleita - OCLe - a partir do estudo comparativo efectuado na secção anterior.

4.1 Modelação do Sistema "Viagens Cacilheiro"

O primeiro passo foi desenhar o diagrama de classes utilizando as funcionalidade de modelação disponíveis no OCLe.

4.2 Introdução das Cláusulas do Contrato

Introduzimos no modelo algumas asserções em OCL e validámos a sua sintaxe contra o modelo produzido. Para exemplificar escolhemos a classe Viagem.

Expressões em OCL implementadas.

```
context Viagem::novoPassageiro(p:Passageiro)
-- um passageiro só pode fazer uma viagem de cada vez.
  pre passageiroNaoExiste:
    not( self.passageiro ->includes(p) )

-- o numero de passageiros é incrementado.
  post numeroPassageiros:
    self.passageiro ->size() = self.passageiro ->size@pre + 1

context Viagem
-- o numero de passageiros de uma viagem não pode ser exceder o numero de
lugares disponíveis do cacilheiro.
  inv numeroPassageirosPorViagem:
    self.passageiro ->size() <= self.cacilheiro.numeroLugares

-- Cada viagem tem um número único.
  inv uniqueViagem:
    Viagem.allInstances->forAll( c1,c2:Viagem |
                          c1<>c2 implies c1.nrViagem<>c2.nrViagem)
```

No OCLe as expressões foram objecto de verificação de sintaxe e verificadas automaticamente contra o diagrama de classes desenhado para o efeito.

4.3 Geração do Código em *Java*

Depois do desenho em UML e especificação em OCL das restrições no OCLe, é chegada a fase de gerar o código *Java* adequado. O OCLe gera código *Java* que implementa simultaneamente a interface das classes e as cláusulas de OCL, dentro de cada classe, no aplicável.

O processo é muito simples, bastando apenas dar a indicação à ferramenta da acção pretendida. Antes de gerar o código *Java*, a ferramenta procede sempre à análise da sintaxe do OCL e à validação das restrições introduzidas, contra o modelo especificado no diagrama de classes. O código *Java* gerado automaticamente pelo OCLe para implementar as asserções correspondentes às expressões OCL definidas no ponto anterior encontra-se em apêndice.

O código gerado é claro e bem estruturado, fazendo apenas referência a algumas classes de uma *framework* desenvolvidas especificamente para esta temática. Apesar de todas as mais-valias desta ferramenta, não podemos dei-

xar de realçar a quantidade de código que é acrescentado a cada classe, unicamente para implementar o DpC com OCL. O OCLe não suporta alguns dos mecanismos do DpC, sendo exemplo disso a não implementação de restrições com carácter temporal (utilização do *xpto@pre*). Além disso, esta implementação de DpC contrasta com a do *Eiffel*, na qual as asserções podem ser inibidas ao critério do programador. Na implementação do OCLe passam a fazer parte do código da classe perdendo essa possibilidade.

5 CONCLUSÕES E FUTURO

Hoje em dia, a maioria das organizações têm como principais objectivos alcançar um nível elevado de qualidade nos seus produtos e/ou serviços. Não é mais aceitável fornecer produtos com má qualidade aos clientes e posteriormente tentar reparar os problemas ou defeitos que daí podem ocorrer. Neste aspecto, os produtos de *software* têm que ser regidos pelo mesmo padrão de qualidade como quaisquer outros produtos, tais como automóveis, computadores, relógios ou casas. No entanto, assumimos que a qualidade no *software* é sem dúvida um conceito complexo que não se pode definir de uma maneira simplista. De qualquer forma, existem presentemente técnicas, como o DpC, que conseguem garantir aos produtos de *software* um nível de qualidade já muito próximo daquele que é vigente a outros produtos de áreas da engenharia.

Já existe alguma variedade de ferramentas que suportam o DpC. No entanto, atendendo à importância que o OCL vai assumindo na modelação, aquelas que não o suportam irão previsivelmente começar a ser postas de lado pelos utilizadores.

Havendo uma série de ferramentas que introduzem as cláusulas de DpC em etiquetas de *Javadoc*, lamenta-se a inexistência de uma norma, ou pelo menos de uma prática comum, quanto à sintaxe utilizada na declaração das asserções, constatando-se que cada ferramenta utiliza um formato próprio. Este facto, para além de obrigar a uma aprendizagem das etiquetas específicas da ferramenta com que se está a trabalhar, impede a complementaridade entre as ferramentas que abrangem apenas partes do ciclo de vida, pois o resultado de uma não pode ser reutilizado por outra.

O OCLe mostrou ser uma ferramenta bastante eficaz, não obstante o facto de se mostrar pouco conforme no que diz respeito às funcionalidades apresentadas, quando comparada com outras ferramentas de modelação com UML, sendo exemplo disso a forma de se desenhar uma relação de composição. Em algumas (poucas) circunstâncias, as mensagens de erro apresentadas resultantes da compilação do OCL são pouco elucidativas do problema existente, valendo no entanto ao utilizador o rigor com que o posicionamento do erro é indicado. Na modelação com o OCLe, os atributos das classes são criados como *protected* por omissão. Isto levanta problemas ao utilizador menos atento dado que, quando implementa as cláusulas de OCL a compilação gera erros de visibilidade. Julgamos que os atributos das classes deviam ser criados como *public* por omissão, desde que fosse assegurada o encapsulamento destes aquando do *forward engineering*.

O código *Java* gerado pelo OCLe para suportar o DpC é muito extenso, constituindo factor de preocupação quanto à manutenção. Esta preocupação é devida ao emaranhamento entre o código de suporte às asserções e o código de suporte às funcionalidades que é implementado em conjunto dentro da mesma classe. Actualmente estão a ser desenvolvidas soluções com aspectos [23], [24], o que previsivelmente mitigará esta problemática. No entanto, como é sabido, os aspectos introduzem actualmente um grande *overhead* e consequente perda de desempenho, pelo que a evolução para esta tecnologia não só não resolverá todos os problemas existentes, como ainda agravará outros. Apesar de tudo, cremos que as soluções implementadas com aspectos irão vingar, dada a sua modularidade, e que pouco a pouco os problemas de desempenho irão sendo atenuados, quer com a evolução do *hardware*, quer com a evolução desta técnica. Todavia, não se conhecem implementações ou projectos de ferramentas que recorram ao emprego das asserções nativas da linguagem *Java*, talvez devido ao facto de estas terem sido introduzidas muito recentemente (finais 2003 [32]). Apesar de as asserções nativas de *Java* não implementarem o DpC tal como foi concebido, designadamente os invariantes que são implementados de forma indirecta (é chamado um método no final de todos os métodos da classe) [32]. Estas asserções poderão constituir uma alternativa séria à implementação de DpC com aspectos.

Como foi já referido o OCLe não suporta ainda, na sua totalidade o DpC, sendo exemplos disso a não compilação de asserções que recorrem ao estado anterior do objecto (*xpto@pre*) e não permitir a inibição das asserções. Além disso o OCLe não faz *reverse engineering*, pelo que qualquer alteração no desenho originará novo código. Apesar de tudo é uma ferramenta bastante completa e a única capaz de sustentar o desenvolvimento de *software* com DpC e OCL ao longo de todo o ciclo de vida, bem como, das ferramentas apreciadas, a única que valida as cláusulas de OCL introduzidas contra o modelo desenhado.

Apesar das vantagens evidentes trazidas pela utilização do DpC e do OCL à qualidade do *software* produzido, não podemos ignorar o compromisso que lhe está associado – o desempenho.

Existem presentemente actividades de investigação no sentido de implementar DpC à custa de notações gráficas [28], favorecendo deste modo uma melhor integração com a habitual representação diagramática do UML.

Seria desejável que as ferramentas que implementam OCL evoluíssem no sentido da geração automática de testes, com base nas asserções.

Em resumo, o DpC com OCL e a sua implementação na linguagem *Java* com PpC são actualmente uma realidade, existindo ferramentas capazes de acompanhar o processo de desenvolvimento ao longo de todo o ciclo de vida, ainda que com algumas limitações. No entanto, os desenvolvimentos verificados nesta área indiciam que este será o caminho certo, estando reunidas mais um conjunto de condições, que permitirão a produção de *software* com cada vez mais qualidade, nomeadamente no que diz respeito ao rigor e robustez daí resultantes.

APÊNDICE

```
/*
* @(#)Viagem.Java
*Generated by <a href="http://lci.cs.ubbcluj.ro/ocle/>OCLE 2.0</a>
* using <a href="http://jakarta.apache.org/velocity/">
* Velocity Template Engine 1.3rc1</a>
*/
import Java.util.Iterator;
import Java.util.LinkedHashSet;
import Java.util.Set;
import ro.ubbcluj.lci.codegen.framework.ocl.BasicConstraintChecker;
import ro.ubbcluj.lci.codegen.framework.ocl.CollectionUtilities;
import ro.ubbcluj.lci.codegen.framework.ocl.Ocl;
import ro.ubbcluj.lci.codegen.framework.ocl.OclType;
/** @author unascribed*/
public class Viagem
{
  public int lugaresDisponiveis() { return 0; }

  public void novoPassageiro(Passageiro p)
  {
    class ConstraintChecker
    {
      public void checkPreconditions(Passageiro p) {
       check_passageiroNaoExiste(p);
      }
      public void checkPostconditions(Passageiro p) {
       check_numeroPassageiros(p);
      }

      public void check_passageiroNaoExiste(Passageiro p)
      {
       Set setPassageiro = Viagem.this.getPassageiro();
       boolean bIncludes = CollectionUtilities.includes( setPassageiro,
                                                                    p);
       boolean bNot = !bIncludes;
       if (!bNot) {
        System.err.println("precondition 'passageiroNaoExiste'
                                        failed for object "+Viagem.this);
       }
      }

      public void check_numeroPassageiros(Passageiro p)
      {
        Set setPassageiro = Viagem.this.getPassageiro();
        int nSize = CollectionUtilities.size(setPassageiro);
        Set setPassageiro0 = Viagem.this.getPassageiro();
        int nSize0 = CollectionUtilities.size(setPassageiro0);
        int nPlus = nSize0 + 1;
        boolean bEquals = nSize == nPlus;
        if (!bEquals) {
         System.err.println("postcondition 'numeroPassageiros' failed
                                            for object "+Viagem.this);
        }
      }
    }
    ConstraintChecker checker = new ConstraintChecker();
    checker.checkPreconditions(p);
    checker.result = internal_novoPassageiro(p);
    checker.checkPostconditions(p);
    return checker.result;
  }

  public final Cacilheiro getCacilheiro() { return cacilheiro; }

  public final void setCacilheiro(Cacilheiro arg)
  {
    if (cacilheiro != arg) {
      Cacilheiro temp = cacilheiro;
      cacilheiro = null;//to avoid infinite recursions
      if (temp != null) {
        temp.removeViagem(this);
      }
      if (arg != null) {
        cacilheiro = arg;
        arg.addViagem(this);
      }
    }
  }

  public final Set getPassageiro()
  {
    if (passageiro == null) {
      return Java.util.Collections.EMPTY_SET;
    }
    return Java.util.Collections.unmodifiableSet(passageiro);
  }

  public final void addPassageiro(Passageiro arg)
  {
    if (arg != null) {
      if (passageiro == null) passageiro = new LinkedHashSet();
      if (passageiro.add(arg)) {
        arg.addViagem(this);
      }
    }
  }

  public final void removePassageiro(Passageiro arg)
  {
    if (passageiro != null && arg != null) {
      if (passageiro.remove(arg)) {
        arg.removeViagem(this);
      }
    }
  }
  private void internal_novoPassageiro(Passageiro p) { }

  public Viagem() { }

  public class ConstraintChecker extends BasicConstraintChecker
  {
    public void checkConstraints()
    {
      super.checkConstraints();
      check_Viagem_numeroPassageirosPorViagem();
      check_Viagem_uniqueViagem();
    }

    public void check_Viagem_numeroPassageirosPorViagem()
    {
      Set setPassageiro = Viagem.this.getPassageiro();
      int nSize = CollectionUtilities.size(setPassageiro);
      Cacilheiro cacilheiroCacilheiro = Viagem.this.getCacilheiro();
      int nNumeroLugares = cacilheiroCacilheiro.numeroLugares;
```

```
      boolean bLessOrEqual = nSize <= nNumeroLugares;
      if (!bLessOrEqual) {
       System.err.println("invariant 'numeroPassageirosPorViagem'
                                          failed for object "+Viagem.this);
      }
     }

     public void check_Viagem_uniqueViagem()
     {
      Set setAllInstances = Ocl.getType(
                                  new Class[]{Viagem.class}).allInstances();
      //evaluate 'forAll(c1,c2:Viagem|c1<>c2 implies
      //c1.nrViagem<>c2.nrViagem)':
      boolean bForAll = true;
      final Iterator iter = setAllInstances.iterator();
      while (bForAll && iter.hasNext()) {
       final Viagem c1 = (Viagem)iter.next();
       final Iterator iter0 = setAllInstances.iterator();
       while (bForAll && iter0.hasNext()) {
        final Viagem c2 = (Viagem)iter0.next();
        boolean bNotEquals = !c1.equals(c2);
        int nNrViagem = c1.nrViagem;
        int nNrViagem0 = c2.nrViagem;
        boolean bNotEquals0 = nNrViagem != nNrViagem0;
        boolean bImplies = !bNotEquals || bNotEquals0;
        bForAll = bImplies;
       }
      }
      if (!bForAll) {
       System.err.println("invariant 'uniqueViagem'
                                          failed for object "+Viagem.this);
      }
     }
    }

    public int nrViagem;
    public Cacilheiro cacilheiro;

    public Set passageiro;
    {
     OclType.registerInstance(this, Viagem.class);
    }
}
```

REFERÊNCIAS

[1] Meyer, Bertrand, *Object-Oriented Software Construction - Second Edition*, *Prentice Hall PTR*, Santa Barbara, California, EUA, (1997).

[2] Guerreiro, Pedro, *An Introduction to Eiffel and Design by Contract*, Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia, Bielko-Biala, Poland, (2002).

[3] Bacvanski, Vladimir, Graff, Petter, *Design by Contract in Java*, InferData Corporation, (1999).

[4] Pender, Tom, *UML Bible*, Wiley Publishing Inc., Indianapolis, Indiana, EUA, (2003).

[5] Warmer Jos, Kleppe, Anneke, *The Object Constraint Language – Second Edition*, Addison-Wesley, EUA, (2003).

[6] http://argouml.tigris.org/ .

[7] http://www.db.informatik.uni-bremen.de/projects/ .

[8] http://www.gentleware.com/ .

[9] http://jcontractor.fonteforge.net/ .

[10] http://www.magicdraw.com/ .

[11] http://www.sparxsystems.com.au/ .

[12] http://www.mmsindia.com/JMSAssert.html .

[13] http://csd.informatik.uni-oldenburg.de/~jass/ .

[14] http://dresden-OCL.fonteforge.net/ .

[15] http://www.parasoft.com/jsp/products/home.jsp?product=Jcontract&itemId=28 .

[16] http://www.klasse.nl/OCL/octopus-intro.html .

[17] http://lci.cs.ubbcluj.ro/ocle/ .

[18] http://www.sce.carleton.ca/Squall/pubs/other/CUSEC2004.pdf .

[19] OMG Unified Modeling Language Specification, “Chapter 6: Object Constraint Language Specification”, OMG, (2001).

[20] OMG, “XML Metadata Interchange (XMI) Specification v1.2”, Object Management Group, (2002).

[21] Mage, Kjetil, *A Pratical Application of the Object Constraint Language OCL*, Agder University College, (2002)

[22] Briand, Dr. Lionel, *Ensuring the Dependability of Software Systems*, CUSEC, (2004).

[23] Richters, Mark, Gogolla, Martin, *Aspect-Oriented Monitoring of UML and OCL Constraints*, EADS SPACE Transportation, University of Bremen, Germany.

[24] Briand, L.C., Dzidek, W., Labiche, Y., *Using Aspect-Oriented Programming to Instrument OCL Contracts in Java*, Simula Research Laboratory, Norway, Software Quality Engineering Laboratory, Departament of Systems and Computer Engineering, Carleton University, Canada, (2004).

[25] Kramer, Reto, iContract - *The Java Design by Contract Tool*, Cambridge Technology Partners.

[26] Wiedmann, Markus, Buchwald, Hagen, Seese, Detlef, *Design by Contract in Java - A Roadmap in Excellence to Trusted Components*, University Karlsruhe, Germany,

[27] Kapp, Steve, *Using iContract and Design by Contract Techniques*, Embedded Real-Time Inc., (2000).

[28] Kiesner, Chritiane, Taentzer, Gabriele, Winkelmann, Jessica, *Visual OCL, A Visual Notation of the Object Constraint Language*, Fakultät IV - Elektrotechnik und Informatik, (2002).

[29] Verheecke, Bart, *From Declarative Constraints in Conceptual Models to Explicit Constraint Classes in Implementation Models*, Departement Informatica, Faculteit van de Wetenschappen, Vrije Universiteit Brussel, (2001).

[30] Liskov, B., Wing, J., *A Behavioural Notion of Subtyping*, ACM Transactions on Programming Languages and Systems, 16(6), pp. 1811-1841, (1994).

[31] J.S. Bridle, *Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationships to Statistical Pattern Recognition*, Neurocomputing—Algorithms, Architectures and Applications, F. Fogelman-Soulie and J. Herault, eds., NATO ASI Series F68, Berlin: Springer-Verlag, pp. 227-236, 1989. (Book style with paper title and editor).

[32] http://www.sun.com .

Ricardo Alves Mestrando em Eng. Informática (2003/05) pela Faculdade de Ciências e Tecnologia (FCT) da Universidade Nova Lisboa (UNL); Licenciado em Eng. Informática (1995/1997) pela FCT/UNL; Bacharel em Eng. Electrónica e Telecomunicações (1991/1994) pelo Instituto Superior de Engenharia de Lisboa (ISEL); Director de Tecnologias e Sistemas de Informação de Portugal e Itália da Amcor Flexibles; Membro da Ordem dos Engenheiros.

Sérgio Bryton Mestrando Eng. Informática (2003/05) pela Faculdade de Ciências e Tecnologia (FCT) da Universidade Nova Lisboa (UNL); Microsoft Certified Solution Developer (2002); Especialização em Informática (2000/2001) pelo Centro de Formação de Informática da Direcção de Tecnologias de Informação e Comunicação (DITIC) da Marinha Portuguesa; Licenciado em Ciências Militares Navais (1990/1995) pela Escola Naval; Coordenador de uma Área Tecnológica com competências de Análise e Desenvolvimento de Sistemas de Informação na DITIC.