UNIVERSIDADE FEDERAL DE SANTA CATARINA

SUPORTE A TESTES AUTOMATIZADOS DE INTERFACE DE COMPONENTES DESENVOLVIDOS NO AMBIENTE SEA

Tiago Jaime Nascimento

Florianópolis – SC

2016/2

UNIVERSIDADE FEDERAL DE SANTA CATARINA

DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

CURSO DE SISTEMAS DE INFORMAÇÃO

SUPORTE A TESTES AUTOMATIZADOS DE INTERFACE DE COMPONENTES DESENVOLVIDOS NO AMBIENTE SEA

Tiago Jaime Nascimento

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de Informação.
Orientador: Prof. Dr. Ricardo Pereira e Silva.

Florianópolis – SC

2016/2

Tiago Jaime Nascimento

SUPORTE A TESTES AUTOMATIZADOS DE INTERFACE DE COMPONENTES DESENVOLVIDOS NO AMBIENTE SEA

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de Informação.

Orientador:

.....…

Prof. Dr. Ricardo Pereira e Silva

Banca examinadora:

.....…

Profa. Dra. Patricia Vilain

.....…

Prof. Roberto Silvino da Cunha

AGRADECIMENTOS

Agradeço pela orientação paciente, motivadora e compreensiva do professor Ricardo. Agradeço à minha família e à minha companheira Fernanda pelo amor e amparo nas horas de dificuldade. Sem estas pessoas este trabalho nunca teria sido possível.

RESUMO

Componentes são artefatos de Software que encapsulam funcionalidades utilizando interfaces bem definidas, permitindo que sistemas complexos sejam desenvolvidos pela composição de elementos criados de forma independente. Para que se possa utilizar esta abordagem aproveitando suas vantagens de redução de custos e aumento da qualidade final do sistema, é necessário que se garanta a qualidade dos componentes sendo desenvolvidos.

Um componente de alta qualidade é fiel à especificação de sua interface, e para tanto, necessita de testes automatizados que ofereçam esta garantia. Estes testes podem ser feitos em tanto em nível de código quanto em nível de modelo, a partir de modelos UML que especifiquem a interface de componente.

Este trabalho propõe uma abordagem para a execução automática de testes de interface em nível de modelo para componentes desenvolvidos no ambiente de desenvolvimento SEA, baseada na criação de *componentes-espelho*, e discute como estes testes podem ser utilizados como parte de uma metodologia de testes automáticos em nível de código da implementação.

Palavras-chave: Desenvolvimento Orientado a Componente. Qualidade de Software. Desenvolvimento Orientado a Modelos.

ABSTRACT

Components are Software artifacts that encapsulate functionalities behind well-defined interfaces, allowing complex systems to be constructed from the composition of independently developed parts. To make this strategy possible, and to be able to leverage the characteristics of cost reduction and higher quality of the system, it is necessay that the quality of the components being developed get individually ascertained.

A high-quality component is true to its interface specification, and for that to happen, automated tests are required. These tests can be performed at the implementation level of abstraction, or at the model level of abstraction - from UML models that represent the component's interface.

This work proposes an approach to the automated creation and execution of interface tests, at the modelling level, for components being developed in the SEA development environment. This is done by the creation of *mirror-components*. It also discusses how these tests can be used as part of a broader methodology that encompasses automatically generated tests at the implementation level of abstraction.

Keywords: Component Based Development. Software Quality. Model Driven Development.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

CCM	-	Corba Component Model
CORBA	-	Common Object Request Broker Architecture
FAE	-	Ferramenta de Análise Estrutural
FAC	-	Ferramenta de Análise Comportamental
GUI	-	Graphical User Interface
IC	-	Interface de Componentes
IDL	-	Interface Description Language
MVC	-	Model-View-Controller
OMG	-	Object Management Group
PIPE	-	Platform Independent Petri Net Editor
SDL	-	Specification and Description Language
SOA	-	Service Oriented Architecture
TDD	-	Test Driven Development
UML	-	Unified Modelling Language

1. INTRODUÇÃO

Este trabalho se insere no contexto da metodologia de desenvolvimento de software orientada a componentes e relaciona-se à melhoria da qualidade do componente e do processo de desenvolvimento.

Um sistema orientado a componentes tem sua implementação separada em unidades independentes que interagem a partir de interfaces bem definidas (SZIPERSKI, 1996), permitindo o reúso de artefatos de software de terceiros e a quebra da complexidade do sistema. O reúso de componentes só é possível quando fornecedores e usuários de componentes possuem métodos confiáveis de se testar as funcionalidades descritas na interface (GAO et al, 2003).

Para garantir a reusabilidade de componentes e sua interoperabilidade com os demais componentes do sistema, é necessário que o componente seja testado de forma que se garanta que o componente se comporta de acordo com a definição de sua interface

Teiniker (2003) propôs um *framework* para desenvolvimento orientado a testes de componentes que atendem à especificação CORBA. Tal proposta se constitui de uma ferramenta que parte de uma definição de interface de um componente para automaticamente gerar um “componente-espelho” – um complemento perfeito da interface do componente original –, o qual executa testes a fim de, a todo momento, garantir que o componente original obedeça à especificação de sua interface. Desta forma, este framework se apropria de conceitos do *Test-Driven Development (TDD)* e assim se contrapõe a abordagens de teste *post-mortem* tradicionais (onde os testes são realizados apenas após o desenvolvimento do artefato de software),

permitindo a combinação entre as vantagens do desenvolvimento orientado a componente com aquelas do TDD.

Para testar se um componente sendo desenvolvido obedece à especificação de sua Interface, pode-se utilizar a Análise de Compatibilidade (TEIXEIRA, 2012) entre este componente e o seu componente-espelho automaticamente gerado, através de suas interfaces.

Um *componente-espelho* é um componente cuja interface age como um complemento perfeito àquela do componente a partir do qual ele foi criado. Um *componente-espelho* oferece todos os métodos requeridos pelo original, e depende de todos os métodos fornecidos pelo original. O *espelho* também obedece às regras de ordem de invocação de métodos impostas pela interface do original.

A Análise de Compatibilidade entre dois componentes avalia aspectos estruturais e comportamentais da interação entre eles a fim de se detectar erros de implementação e/ou especificação. Esta Análise é possível a partir do ambiente de desenvolvimento SEA, uma ferramenta gráfica de modelagem de software que permite a especificação UML de componentes e *frameworks*, e possibilita a criação de ferramentas customizadas e a geração automática de código (SILVA, 2000).

O objetivo geral deste trabalho é implementar solução inspirada no trabalho de Teiniker (2003) no ambiente SEA, para permitir o teste em nível de modelo da interface de um componente sendo desenvolvido, no que diz respeito aos aspectos estruturais, comportamentais e funcionais (SILVA, 2009).

Os objetivos específicos são:

- Implementar ferramenta capaz de gerar automaticamente uma especificação

de interface de componente-espelho a partir da especificação de interface de componente a testar;

- Implementar ferramenta capaz de comparar automaticamente a especificação de interface de um componente com a especificação da interface do respectivo componente-espelho, com vista à detecção de alterações na especificação original;
- Utilizar a Análise de Compatibilidade de componentes (TEIXEIRA, 2012) para permitir garantias de obediência do componente à sua interface, integrando ferramenta existente à ferramenta de análise a desenvolver;
- Discutir os desafios envolvidos em uma abordagem completa que utilizasse estes testes automatizados para a geração automática de código de casos de teste de interface.

A solução proposta neste trabalho se limita a componentes sendo desenvolvidos a partir do padrão de interfaces definido em Silva (2002), o que não necessariamente significa suporte a componentes implementados utilizando outros padrões de interface de componente. O processo de desenvolvimento utilizado se baseia na metodologia de desenvolvimento de componentes orientada à modelagem UML de Silva (2009).

Este trabalho pretende contribuir para a melhoria de um aspecto central da motivação por trás do desenvolvimento orientado a componentes (SOMMERVILLE, 2011, p. 315): a reusabilidade. Argumenta-se que componentes desenvolvidos

através de um processo de desenvolvimento que prioriza os testes de regressão automatizados da interface podem apresentar boas garantias de que suas funcionalidades obedeçam à especificação comunicada ao usuário do componente e, portanto, sejam menos passíveis de erros e de comportamentos inesperados.

Em termos de metodologia, esta pesquisa exploratória é de natureza aplicada na medida em que tem como objetivo promover conhecimentos e técnicas com vistas à aplicação prática. Além disso, conta com uma abordagem qualitativa. Portanto, como procedimentos e técnicas de pesquisa, são realizados pesquisa bibliográfica e estudo de caso.

Este trabalho obedece à seguinte divisão: primeiramente uma revisão teórica é feita acerca do desenvolvimento orientado a componentes (seção 2) e de testes de componentes (seção 3). A seção 4 dedica-se a ilustrar o funcionamento do ambiente de desenvolvimento escolhido para a implementação e por fim a solução proposta é descrita e exemplificada através de estudo de caso na seção 5. A seção 6 é destinada às conclusões do trabalho.

2. DESENVOLVIMENTO ORIENTADO A COMPONENTES

Este paradigma de desenvolvimento oferece uma forma de se contornar o problema da crescente complexidade de softwares orientados a objetos. Inserindo uma camada de abstração superior à das classes, estas podem ser divididas em partes independentes do sistema, e assim gerenciadas mais facilmente pelos desenvolvedores, possibilitando o desenvolvimento e a manutenção em paralelo e/ou por entidades diferentes. Além da quebra da complexidade, outra vantagem da separação do desenvolvimento em componentes independentes é a de facilitar o reúso de software, que proporciona (SOMMERVILLE, 2011):

- Maior confiabilidade – quanto mais reutilizado um artefato de software for, maior a chance de suas falhas de projeto e implementação terem sido encontradas e tratadas;
- Menor risco – o custo de um componente pré-existente é conhecido, enquanto os custos de uma nova implementação são sempre incertos;
- Uso eficiente de especialistas – evita-se o problema de se “reinventar a roda” ao se desenvolver funcionalidades já implementadas por terceiros;
- Rapidez de desenvolvimento – ao se reduzir o tempo gasto em implementação e validação.ferramenta

Desde a constituição deste campo, foram diversos os esforços para conceituar a categoria de Componente, sem, no entanto, ter-se chegado a algum consenso (SOMMERVILLE, 2011; SILVA, 2009; TEIXEIRA, 2012). Sziperski (1996 apud 2002, p. 41) traz uma definição:

> Um componente de software é uma unidade de composição com interfaces contratualmente especificadas e apenas dependências de

> contexto explícitas. Um componente de software pode ser implantado de forma independente e está sujeito a composição por terceiros.

Sziperski complementa posteriormente sua definição, considerando que qualquer dispositivo de software com uma interface definida possa ser considerado um componente (SZYPERSKI, 1997 apud TEIXEIRA, 2012). Nesta definição mais ampla, é possível pensar provedores de serviços, desenvolvidos sob arquiteturas orientadas a serviços (Service Oriented Architecture – SOA) como componentes, mas Sommerville (2011) distingue os dois conceitos: serviços são entidades sem dependências externas explícitas, e componentes frequentemente exigem serviços externos especificados em suas interfaces.

Sommerville (2011, p. 317) também combina as definições de diferentes autores e pontua as principais características associadas a um componente de software:

1. Padronizado: obedecem padrões de interfaces, documentação, composição e implantação;
2. Independente: passível de ser utilizado para composição sem que necessite de outro componente ou artefato de software específico. Qualquer dependência é estabelecida através da interface;
3. Passível de Composição: oferece acesso às informações internas e possui bem definidas suas interações externas através da interface;
4. Implantável: possui autonomia para que, por si, possa ser combinado com outros componentes no desenvolvimento de um sistema;
5. Documentado: toda informação necessária, para que potenciais usuários façam a decisão de utilizar o componente, deve estar explicitamente documentada. A interface de componentes deve estar clara.

Componentes podem ser considerados como parte da evolução da programação orientada a objetos (SILVA, 2009, p. 174): quando o desenvolvimento e a manutenção de um sistema tornou-se muito complexo pela grande quantidade de classes, necessitou-se separá-las em partes coesas contendo apenas uma fração delas, cada parte podendo ser utilizada sem que se tivesse conhecimento da implementação de cada classe.

Sziperski (2002) considera componentes como essencialmente independentes de tecnologia ou metodologia de implementação, podendo não serem implementados sob o paradigma de orientação a objetos.

Por último, componentes são partes modulares com implementações internas encapsuladas, permitindo que sejam substituídos por outras partes que possuam interfaces compatíveis (OMG, 2011 apud TEIXEIRA, 2012, p. 33).

2.1 Interfaces de Componentes

A maioria das definições e características associadas a componentes acima pressupõem a possibilidade de se descrevê-los externamente, e usá-los sem que se tenha acesso a detalhes de suas implementações. A esta parte externamente visível se dá o nome de Interface de Componente. Ela age como o meio pelo qual componentes se conectam, especificando os detalhes e a semântica das operações que podem e/ou devem ser invocadas entre eles. Outra forma de caracterizar uma especificação de Interface é como um contrato entre cliente e implementador de um conjunto de interfaces, dizendo o que o cliente precisa fazer para poder usá-las, ao mesmo tempo que explicitando garantias a respeito do resultado destas operações

(SZYPERSKI, 2002). É importante aqui diferenciar (SILVA, 2009; SZYPERSKI, 2002):

- Interface de Componente: uma coleção de pontos de acesso a serviços, cada um com sua semântica estabelecida;
- Interface UML: uma relação de assinaturas de métodos a serem implementados por dado elemento do sistema (Classe, Componente) via relação de *realização*, e utilizados por um ou mais outros elementos via relação de *dependência*;
- Porto: fronteira que separa o componente (sua estrutura interna) e seu meio externo. Pode estar associado a uma ou mais interfaces UML.

A Figura 1 utiliza a notação do Diagrama de Componentes da UML para ilustrar um componente chamado *ComponentX*. Ele possui dois Portos (P1 e P2) – cada um associado a uma ou mais interfaces UML – a *InterfaceSom* possui assinaturas de métodos dos quais o componente depende (através do Porto P1) e a *InterfaceSist* possui assinaturas de métodos que ele implementa e portanto, oferece externamente (através do Porto P2). Sem os serviços especificados em *InterfaceSom*, o componente *ComponentX* não funcionará. A *InterfaceSist,* por sua vez, age efetivamente como a API deste componente, detalhando quais serviços podem ser invocados externamente. A Interface de Componente do *ComponentX* corresponde ao conjunto de seus Portos (circulado na Figura 1).

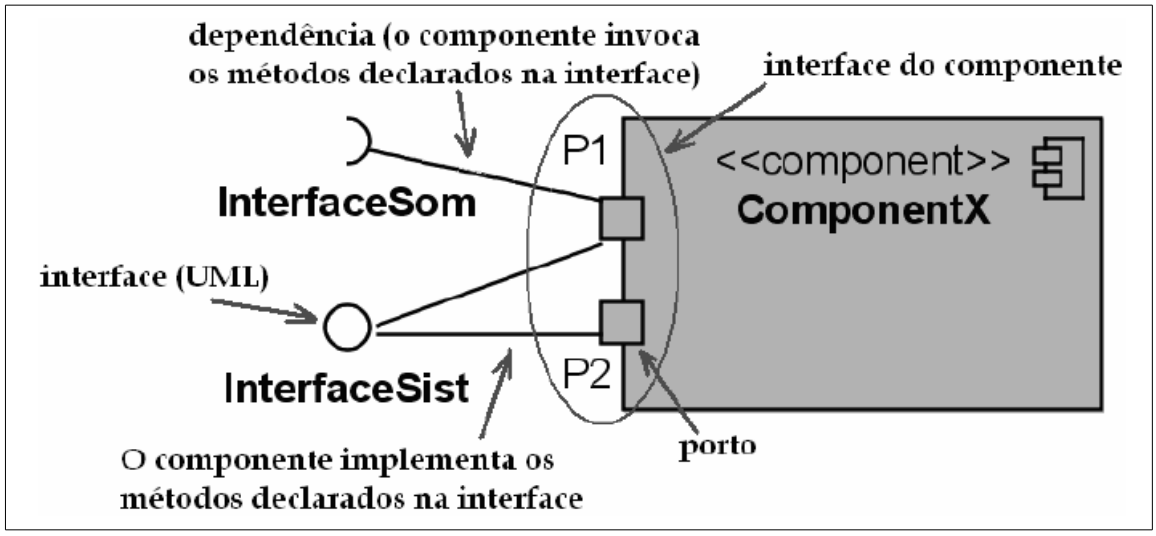

Figura 1: Visão externa de um componente (Fonte: SILVA, 2009, p. 176)

Diversas propostas foram feitas na tentativa de se padronizar a especificação de interfaces de componentes, tais como (SILVA, 2002):

- Linguagens de Descrição de Interfaces (IDLs) descrevem serviços oferecidos tipicamente em termos de assinaturas de métodos, mas não oferecem meio de especificar serviços dos quais o componente depende;
- *CORBA Component Model* (CCM) descreve tanto serviços oferecidos quanto requeridos por um componente, mas não descrevem suas características dinâmicas;
- Contratos descrevem dependências dinâmicas em forma de interações e obrigações, mas produzem descrições com baixo nível de abstração, o que as tornam complexas de se entender e utilizar;
- Redes de Petri podem descrever restrições dinâmicas na ordem de invocações de serviços e, através de análises de propriedades, permite a verificação de características comportamentais e identificação de problemas

de especificação. Sua desvantagem está no fato de se tratar de uma notação não tão familiar como outras abordagens que utilizam a linguagem UML.

A falta de unanimidade sobre a escolha do padrão de especificação de interfaces de componentes dificulta o potencial de reúso e, consequentemente, a adoção desta abordagem de desenvolvimento. Para Silva (2009), uma descrição completa de um componente deve abranger os três seguintes aspectos:

1. Descrição Estrutural: descreve a estrutura dos portos do componente e suas interfaces associadas, ou seja, a relação das assinaturas de métodos requeridos e fornecidos pelo componente através de sua Interface;
2. Descrição Comportamental: especificação de restrições dinâmicas para a interação com o componente, ou seja, a ordem possível de invocação de métodos requeridos e fornecidos através de sua Interface;
3. Descrição Funcional: descrição do que o Componente *faz* (mas não *como* o faz): detalhamento dos métodos da forma como são externamente percebidos e não como são internamente implementados.

A descrição da Interface de um componente neste trabalho, que abrange cada um dos três aspectos acima, utiliza o padrão de modelagem proposto por Silva (2009, p.182), que utiliza a linguagem UML para todas as etapas. As subseções 2.1.1, 2.1.2 e 2.1.3 foram elaboradas a partir da obra de Silva, e sumarizam os passos necessários para a modelagem.

2.1.1 Modelagem Estrutural

Necessária a especificação de todas as interfaces associadas ao componente via diagrama de classes (Figura 2), e a especificação dos portos do componente, com cada associação (realização ou dependência) entre eles e as interfaces especificadas, via diagrama de componentes (Figura 3).

Os exemplos das Figuras 2 e 3 ilustram dois componentes: *InterfaceGraficaJV* e *LogicaJV.* O primeiro componente realiza (implementa) os métodos declarados na interface *InterfaceVisaoJV*, e depende dos métodos da interface *InterfaceLogicaJV,* através do porto Pi. O componente *LogicaJV* possui tanto relacionamento de realização quanto de dependência para com as interfaces *InterfaceLogicaJV e InterfaceVisaoJV*, através dos portos P1 e P2.

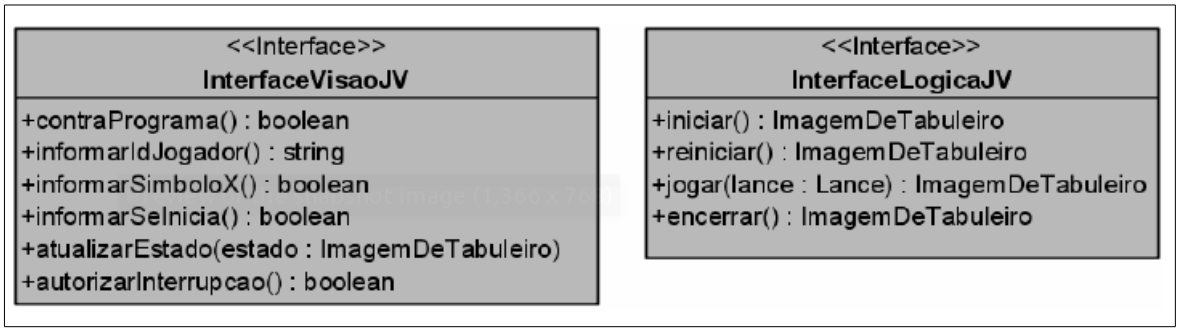

Figura 2: Parte da Modelagem Estrutural de componentes (Fonte: SILVA, 2009, p. 183)

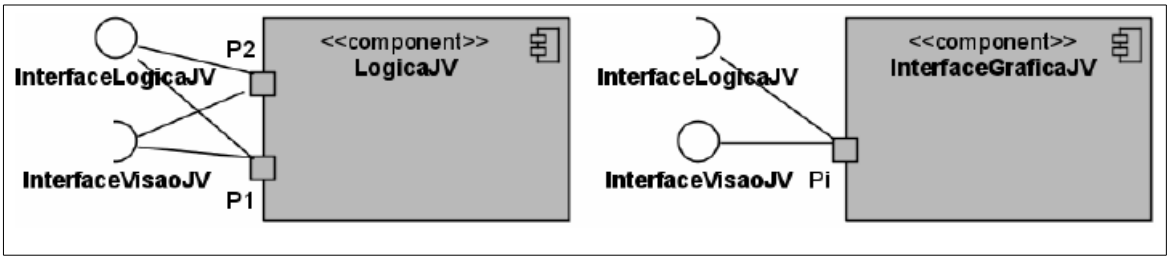

Figura 3: Parte da Modelagem Estrutural de componentes (Fonte: SILVA, 2009, p. 183)

2.1.2 Modelagem Comportamental

Nesta etapa de modelagem estabelecem-se as restrições de ordem de invocação de métodos fornecidos e requeridos, utilizando o diagrama de máquina de estados.

Para um componente a ser descrito, no mínimo um diagrama de máquina de estados deve ser criado para descrever seu comportamento. A figura do estado representa o conjunto de métodos (fornecidos pelo componente ou requeridos de outro) que podem ser executados em determinado momento. As transições representam invocações destes métodos, que podem ou não levar a um estado diferente do original. Transições são descritas pelo padrão:

[<sentido>] [[<porto> ‘,’] <porto> ‘.’] [<método > ‘,’]* <método>*

Onde:

- <*sentido*> pode ser <<*out*>> quando a invocação é feita pelo componente para um método de uma interface requerida, ou <<*in*>> quando o método é fornecido pelo componente e foi chamado externamente. Caso omitido, considera-se como fornecido (<<*in*>>);
- <*porto*> é o nome do Porto que invoca ou que recebe a invocação do método;
- <*método*> é o nome do método.

A Figura 4 continua o exemplo de especificação das Figuras 2 e 3, e ilustra as restrições comportamentais do componente *InterfaceGraficaJV*. A partir do diagrama, entende-se que este componente só avança para o estado *2* ao invocar o método *iniciar* (declarado na interface *InterfaceLogicaJV* na Figura 2) a partir do porto *Pi*, e

que ele avança entre os estados 3 e 6 a partir de uma sucessão de invocações aos

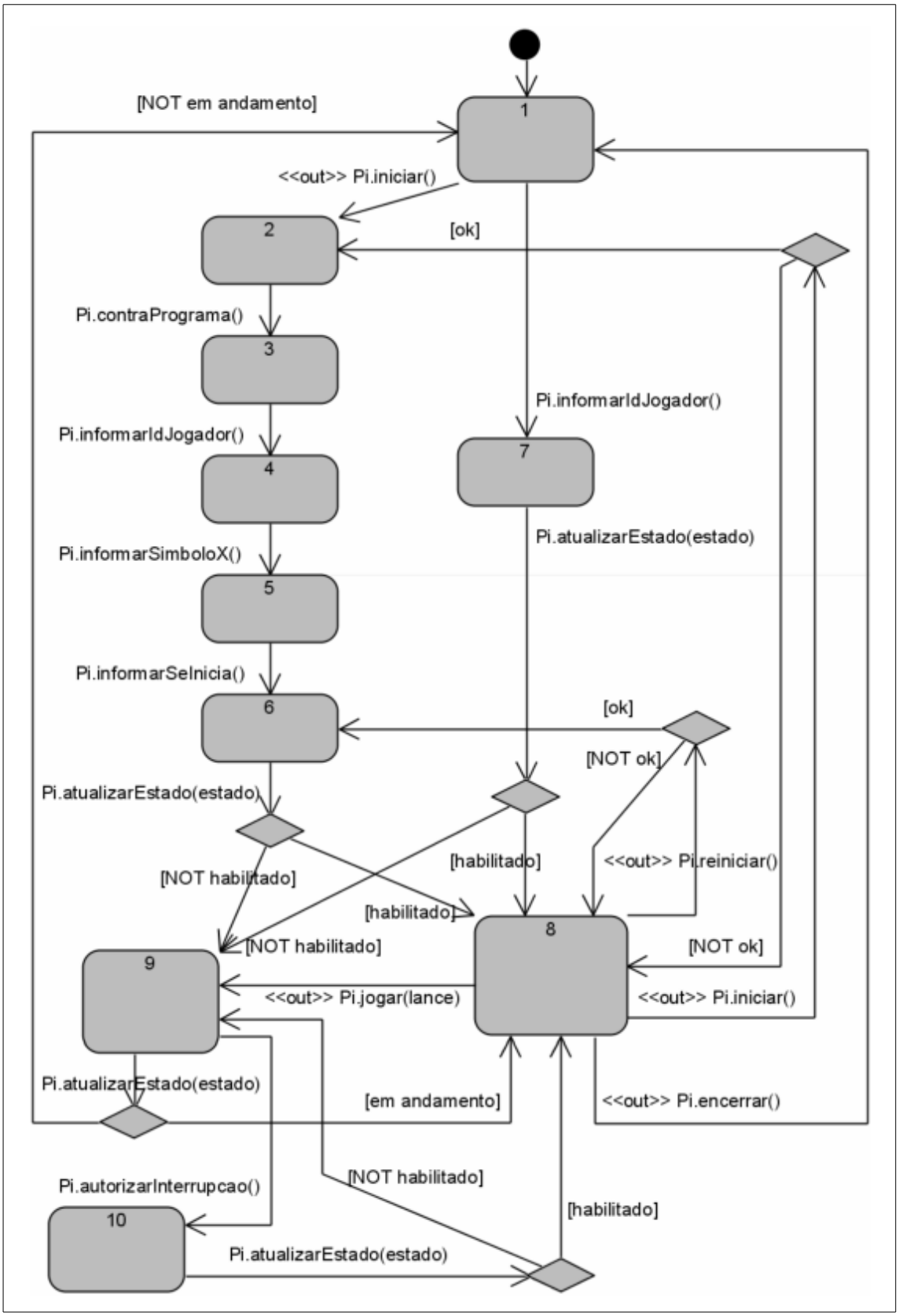

Figura 4: Exemplo de Modelagem Comportamental de componente (Fonte: SILVA, 2009, p.

189)

métodos que fornece através do mesmo porto Pi (*contraPrograma, informarIdJogador,* etc).

2.1.3 Modelagem Funcional

As funcionalidades de um componente são os métodos que fornece através da(s) interface(s) que implementa. A abordagem de Silva (2009), propõe especificar estas funcionalidades em um diagrama de casos de uso (Figura 5), e depois detalhá-los em diagramas de atividades (Figura 6). Este detalhamento deve ser abstraído para deixar claro aquilo que *é feito* pelo serviço fornecido, e não deve focar em detalhes de *como* é feito.

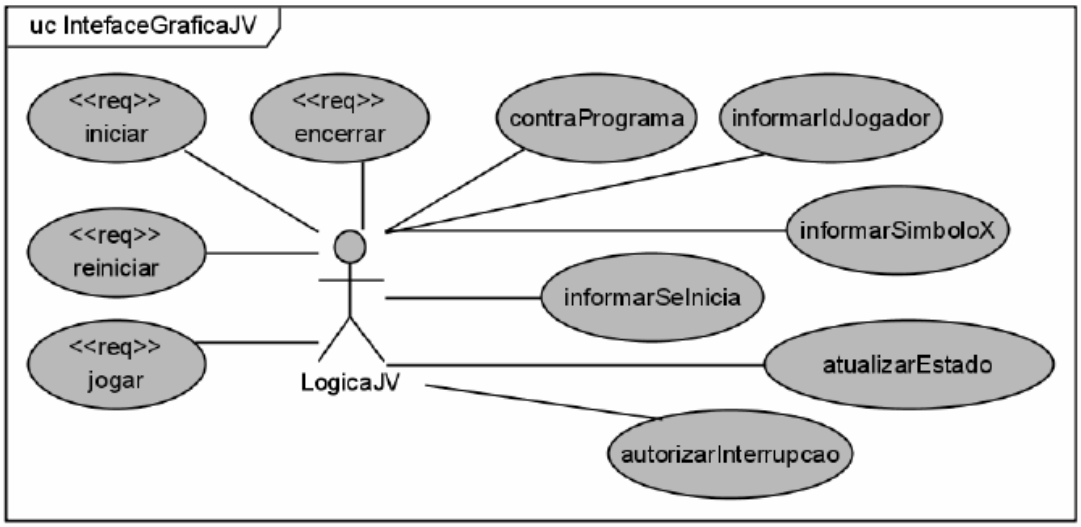

Figura 5: Parte da Modelagem Funcional de um componente (Fonte: SILVA, 2009, p. 191)

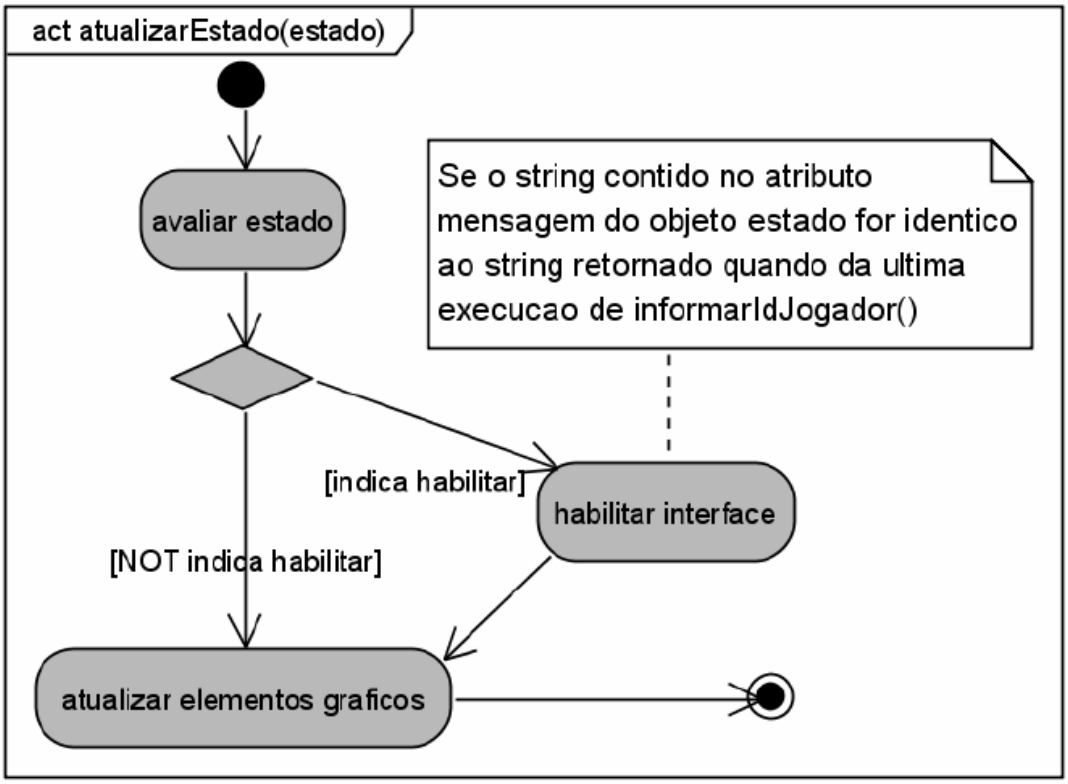

Figura 6: Parte da Modelagem Funcional de um componente (Fonte: SILVA, 2009, p. 192)

2.2 Compatibilidade entre Componentes

A atuação conjunta de diferentes componentes – os quais, por sua vez, são desenvolvidos de modo independente uns em relação aos outros – e a interação entre eles depende sobremaneira da compatibilidade de suas interfaces. Para tanto, essa compatibilidade precisa se manifestar em distintos níveis – estrutural, comportamental e funcional – a fim de que se evitem problemas de comunicação e, consequentemente, no sistema desenvolvido.

A compatibilidade estrutural implica que o conjunto de métodos requeridos através da interface de um componente esteja disponível na interface do outro. A compatibilidade no nível comportamental está relacionada à equivalência entre as restrições associadas à ordem de execução de métodos estabelecidas em um

componente e em outro. Por fim, a compatibilidade funcional depende de que as funcionalidades requeridas por um componente sejam supridas pelo outro. Nas palavras da autora, que se ampara em Silva (2000),

> Dois componentes são estruturalmente compatíveis se o conjunto de métodos requeridos através da interface de um está disponível na interface do outro. Eles são compatíveis no nível comportamental quando as restrições associadas à ordem de execução de métodos, fornecidos ou requeridos, estabelecidas em um são respeitadas pelo outro. E são funcionalmente compatíveis quando as funcionalidades requeridas por um são supridas pelo outro (SILVA, 2000).(TEIXEIRA, 2012, p. 36).

A respeito da compatibilidade e incompatibilidade de componentes, menciona que "a dificuldade de compatibilizar componentes originalmente incompatíveis constitui um problema da abordagem de desenvolvimento orientado a componentes, sugerindo a necessidade de mecanismos que facilitem a adaptação de componentes" (SILVA, 2000, p. 25).

A análise de compatibilidade entre componentes se dá por intermédio da especificação da interface de componente (IC). É nesse sentido que se mostra de fundamental relevância o estabelecimento de mecanismos e de critérios de descrição de componentes capazes de qualificá-los adequada e minuciosamente nos três níveis – estrutural, comportamental e funcionalmente –, na medida em que é por meio dessa descrição que se efetua a análise. Especificamente, o trabalho de Teixeira tem como objetivo a identificação de eventuais incompatibilidades nos níveis estrutural e comportamental; ela opta por não contemplar o nível funcional e a compatibilidade dos componentes interligados.

Portanto, a decisão sobre o uso ou não de componentes em determinado sistema está estreitamente vinculada à identificação de incompatibilidades estruturais e/ou comportamentais. O desenvolvimento de um sistema depende, por

conseguinte, da previsão acurada dos eventuais problemas de interação entre componentes, a fim de que se evitem erros na execução do sistema.

Para a realização da Análise de Compatibilidade Estrutural e Comportamental entre dois componentes, Teixeira (2012) propõe duas ferramentas, descritas nas subseções 2.2.1 e 2.2.2.

2.2.1 Ferramenta de Análise Estrutural (FAE)

Segundo Teixeira (2012) "a análise da compatibilidade estrutural é realizada para cada par de portos conectados dos diferentes componentes da aplicação. Métodos requeridos no porto em um lado da conexão devem ser fornecidos pelo porto do outro lado da conexão" (TEIXEIRA, 2012. p. 83). Nesse sentido, a incompatibilidade estrutural "pode ocorrer na conexão entre portos de componentes quando o serviço requerido por um componente não tem o respectivo serviço fornecido pelo outro".

A Ferramenta de Análise Estrutural (FAE) realiza:

- Análise de consistência da especificação estrutural: garante que a especificação estrutural do componente foi descrita correta e completamente seguindo a abordagem vista anteriormente;
- Análise dos portos conectados: compara pares de portos conectados e garante que, os métodos requeridos em um porto sejam fornecidos pelo outro porto da conexão, e vice-versa, considerando o nome do método, tipo de retorno, número de parâmetros e tipo de parâmetro.

2.2.2 Ferramenta de Análise Comportamental (FAC)

A partir da especificação comportamental da aplicação, que é criada a partir da interligação das máquinas de estado individuais dos dois componentes a serem analisados, a Análise Comportamental verifica as restrições de ordem de invocação de métodos requeridos e fornecidos para cada componente, e nos diz se é compatível com as restrições dos outros componentes conectados. Esse tipo de avaliação envolve todo o conjunto de componentes interligados – diferente da avaliação de compatibilidade estrutural, que trata um par de portos de cada vez (SILVA, 2009, p. 198).

A Ferramenta de Análise Comportamental (FAC) converte a máquina de estados da aplicação em uma Rede de Petri, de forma automática e transparente ao usuário. A partir da análise das propriedades das Redes de Petri, a ferramenta identifica possíveis problemas de compatibilidade comportamental entre os componentes analisados. Mais especificamente, a ferramenta faz (TEIXEIRA, 2012):

- Análise de consistência da especificação comportamental;
- Geração da máquina de estados da aplicação;
- Conversão da máquina de estados em Redes de Petri;
- Análise das propriedades das Redes de Petri: utilizando a ferramenta *Pipe - Platform Independent Petri net Editor*.

2.3 Processo de Desenvolvimento Orientado a Componentes

Uma das distinções mais relevantes no campo de estudos em questão diz

respeito ao ponto de vista do processo de desenvolvimento, o qual se divide em duas partes: uma delas preocupada em desenvolver os componentes individualmente, e a outra em desenvolver o sistema a partir do reúso dos componentes. Nas palavras de Sommerville (2011, p. 321), trata-se de uma diferença entre *desenvolvimento para reúso* e *desenvolvimento com reúso* – se o primeiro se refere ao processo de desenvolvimento de componentes ou serviços a serem reutilizados em outras aplicações, o segundo parte de componentes desenvolvidos por terceiros para adaptá-los em funcionalidades do sistema.

Silva (2009), em sua metodologia de desenvolvimento orientado à modelagem UML, propõe as seguintes etapas no desenvolvimento de um componente *para reúso* (Figura 7):

1. Definição da visão externa do componente, a partir das modelagens estrutural, comportamental e funcional;
2. Definição das classes que implementarão a interface, especializadas do padrão de interface de componentes, assim como definição e modelagem dos casos de uso do componente;
3. Implementação da estrutura interna do componente.

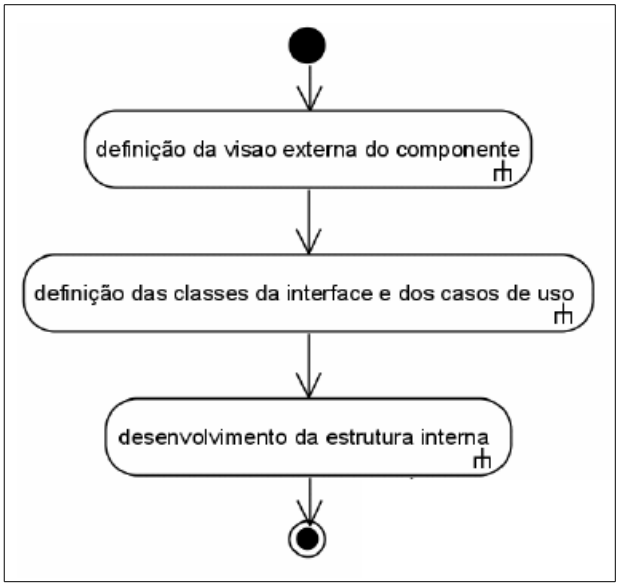

Figura 7: Sumário do desenvolvimento de um componente (SILVA, 2009, p. 224)

O autor também propõe etapas gerais para o desenvolvimento de sistemas orientados a componentes, *com reúso* (Figura 8):

1. Modelagem de casos de uso da aplicação;
2. Refinamento dos casos de uso em diagramas de atividade;
3. Definição do conjunto de componentes através do diagrama de atividades;
4. Definição da estrutura da aplicação através de portos, métodos e interfaces;
5. Definir especificações comportamentais e funcionais de cada componente;
6. Implementação dos componentes;
7. Integração dos componentes.

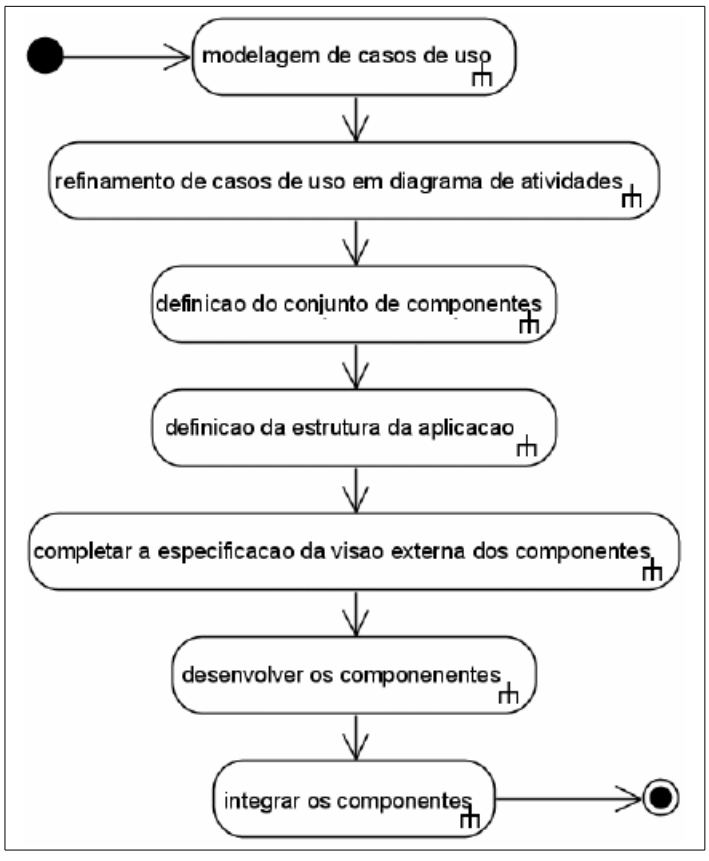

Figura 8: Sumário do desenvolvimento de aplicação baseada em componentes (SILVA, 2009, p. 223).

Como será detalhado na seção 5, o presente trabalho preocupa-se em melhorar a qualidade do desenvolvimento de componentes individuais *para reúso* (Figura 7), ao garantir que durante a implementação interna sua interface seja a todo momento respeitada. O desenvolvimento de sistemas a partir da *composição* de diferentes componentes foge do escopo deste trabalho.

2.4 Component-Interface Pattern

Este padrão (SILVA, 2002; 2009) oferece uma estrutura de classes para a implementação de componentes de software orientados a objeto, que sigam ao padrão de Interface de Componentes discutido anteriormente. Tal estrutura (figura 9), é composta pelas seguintes classes (SILVA, 2009):

- *InterfaceDeComponentes*: representa a parte visível do componente, agindo como se fosse uma classe única. Trata-se de uma classe abstrata que deve ser herdada por uma subclasse em desenvolvimento, apenas uma por artefato. O objeto desta classe tem a responsabilidade de inicializar os portos (subclasses de *PortoDeInterface*) e a estrutura interna, que encapsula a implementação da lógica do componente;
- *PortoDeInterface*: para cada porto de comunicação entre o componente e o meio externo deve ser criada uma subclasse desta classe. Portos devem implementar as interfaces realizadas pelo componente, e fazer a ponte de

comunicação com o meio externo através da troca de mensagens a objetos de subclasses de *CaixaDeSaidaDePorto*;

- *CaixaDeSaidaDePorto*: responsável por repassar invocações a métodos de interfaces requeridas do componente entre o porto deste componente, e o porto do componente externo. Devem ser criadas subclasses, uma por par de portos a se comunicarem. Cada objeto desta classe possui uma referência à subclasse de *PortoDeInterface* do componente externo a ser invocado;

- *MaquinaDeEstados*: modela a máquina de estados que descreve o comportamento do componente. Deve ser criada uma subclasse para cada componente, que implementará o método *iniciarEstrutura* de forma que defina a estrutura de dados de transições válidas pela máquina de estados. O método *executarTransicao* retorna *true* ou *false* caso a determinada transição seja válida para determinado estado. Toda chamada de método requeridos ou fornecidos é precedida por uma chamada ao método *executarTransicao*, que dirá se a transição é valida de acordo com a especificação do sue comportamento, e que pode prosseguir ao destino;

Como exemplos de casos de uso, para receber invocações a métodos que o componente fornece, uma chamada vem do exterior através de uma subclasse de *CaixaDeSaidaDePorto*, que a repassa para a subclasse de *PortoDeInterface* conectada. O porto utiliza o objeto da *MaquinaDeEstados* para determinar que a chamada recebida é válida, e caso positivo repassa a chamada para o executor interno.

Para o componente enviar uma chamada a um serviço de interface da qual depende, uma classe interna obtém referência à subclasse de *CaixaDeSaidaDePorto*, e faz a chamada, que também testa a validade através da *MaquinaDeEstados* e caso seja válida, é enviada para o componente externo.

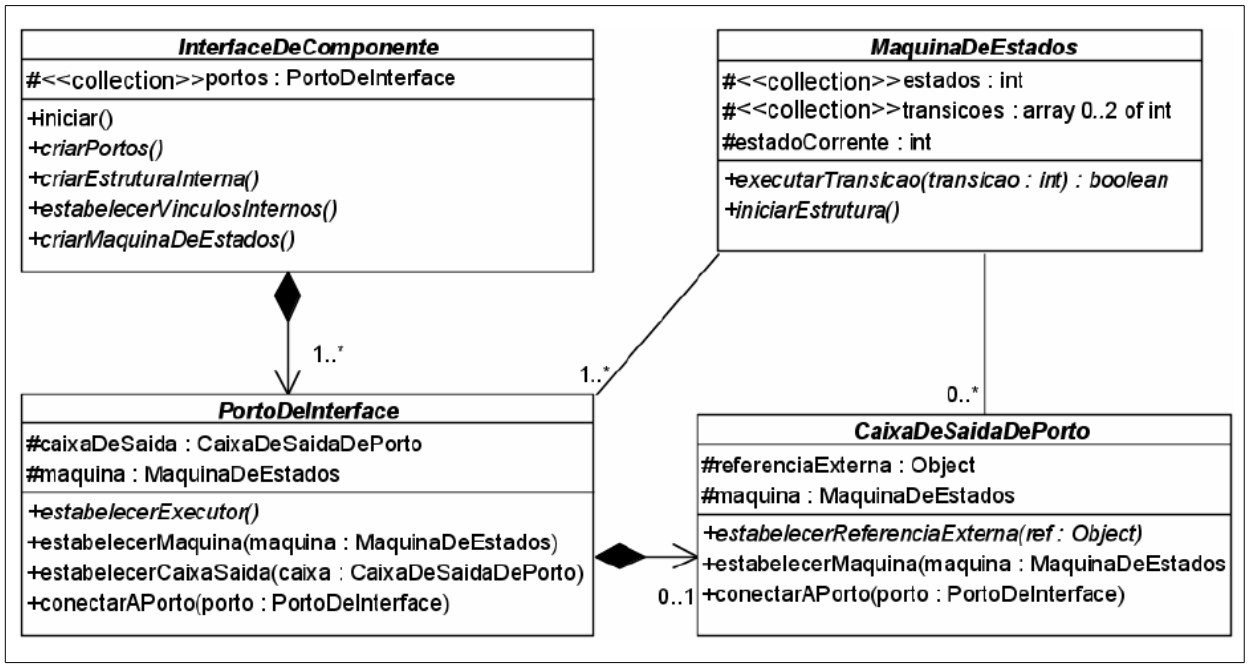

Figura 9: Estrutura de classes do *Component-Interface Pattern* (SILVA, 2009, p. 208)

3. TESTES DE COMPONENTES

Segundo Gao et al (2003, p. 14), módulos de software tradicionais são desenvolvidos com o foco na estrutura e implementação interna, em um processo de desenvolvimento orientado ao sistema em que os módulos serão inseridos, sem o objetivo de serem oferecidos como produtos de software independentes e, portanto, carecem de atenção à interação externa, composição ou padronização de interfaces, tendo assim sua reusabilidade em diferentes contextos comprometida. Testes destes artefatos de software não são tipicamente voltados à garantia de qualidade de outros aspectos além da implementação interna.

Diferente de testes de módulos tradicionais de sistemas, um processo completo de testes de componentes de software deve ainda responder às seguintes perguntas (GAO et al, 2003):

- Como validar a reusabilidade de um componente?
- Como validar a interoperabilidade de um componente?
- Como garantir que um componente atenda a um modelo de componentes?
- Como validar um componente quanto à implantação e ao empacotamento corretos?
- Como validar um componente quanto à adaptação e customização corretas?

De acordo com os autores, os testes no Desenvolvimento Orientado a Componentes podem ser separados de acordo com as partes distintas envolvidas no

processo (GAO et al, 2003, p. 51): os testes de componentes por parte do fornecedor do componente – que validam o componente de acordo com suas especificações – e os testes por parte do usuário – que acontecem no contexto de desenvolvimento de um software baseado em componentes para garantir que os componentes atendem às funcionalidades, interfaces e quesitos de performance prometidos. Enquanto o fornecedor de um componente se preocupa em garantir que a sua implementação obedece a especificação definida, e que atendem aos padrões de modelo de componentes, o usuário de um componente precisa ter a certeza de que ele foi escolhido corretamente para a tarefa, que ele é compatível com o sistema e que está sendo reutilizado de forma correta. Adotando esta distinção, o processo de testes de componentes por parte do fornecedor é composto por:

1. Testes caixa-preta: utiliza a especificação e a interface do componente para garantir que não hajam comportamentos ou funcionalidades incorretas ou incompletas;

2. Testes caixa-branca: avaliação de estrutura, lógica e dados internos aos componentes;

3. Testes de uso: exercícios de padrões de usos de componentes a partir das interfaces de componentes definidas para garantir que estas estão sendo honradas;

4. Testes de performance: validação e avaliação da performance a partir de critérios estabelecidos na especificação do componente;

5. Testes de empacotamento e customização: o foco do teste são as suas funcionalidades de customização, e seu formato de empacotamento para a

produção do artefato final a ser distribuído, quando tais características se aplicam ao componente;

6. Testes de implantação: valida o mecanismo de implantação do componente para garantir que seu projeto e implementação atendem a um dado modelo de componentes.

Por outro lado, os testes de componentes por parte do usuário se concentram em avaliar o reúso de um componente de terceiros e tipicamente envolve:

1. Implantação: validação do componente para verificar que pode ser utilizado no contexto desejado, em conjunto com os outros elementos do sistema;
2. Testes de empacotamento e customização: testa se o componente pode ser empacotado e adaptado em conjunto com o sistema sendo desenvolvido;
3. Testes de uso: similares aos testes de uso feitos por parte do fornecedor (descritos acima), mas levando em conta o contexto do sistema em que o componente será inserido;
4. Validação de componente: similar aos testes caixa-preta realizados pelo fornecedor (descritos acima), mas levando em conta o novo contexto;
5. Testes de performance;

Embora trate apenas do nível de modelo, sem a execução dos testes em relação à implementação do componente, a estratégia proposta neste trabalho se

relaciona com o lado do fornecedor de componentes, principalmente no que diz respeito aos testes caixa-preta, que podem ser definidos como (IEEE, 2010):

1. “Testes que ignoram mecanismos internos de um sistema ou componente e focam inteiramente nos *outputs* gerados em resposta a *inputs* e condições de execução selecionados”;
2. “Testes conduzidos para avaliar a adequação de um sistema ou componente para com seus requisitos funcionais especificados”.

Testes caixa-preta se preocupam em garantir que o software faça o que deve fazer, algo que pode ser testado em diferentes níveis de abstração. Como se trata do único tipo de teste que pode ser feito apenas a partir da especificação externa de um componente (GAO et al, 2003), ele é adequado no desenvolvimento orientado a componentes, pois geralmente o usuário de um componente desenvolvido por terceiros não terá acesso ao código-fonte e aos detalhes de implementação deste.

Para Sommerville (2011), do ponto de vista do usuário de um componente, os testes devem ser direcionados a garantir que o componente se comporta de acordo com a definição de sua interface. A mesma lógica pode ser aplicada a um conjunto de componentes, quando suas interfaces são combinadas em um “super-componente” (SOMMERVILLE, 2011, p. 151). Entre os possíveis erros a serem detectados por testes realizados a partir da interface de componentes, o autor cita o erro de mau uso de interface – quando um componente utiliza a interface de outro componente incorretamente, o mau entendimento da interface – quando suposições incorretas são feitas a respeito da interface de um componente e por último, os erros

de *timing* – quando a interação entre componentes não obedece uma sincroniza de tempo exigida pela especificação deles.

4. AMBIENTE SEA DE DESENVOLVIMENTO

No trabalho de Silva (2000), foi concebido o OCEAN, um *framework* orientado a objetos que fornece suporte à construção de ambientes de desenvolvimento de software. Ambientes construídos através do OCEAN permitem o manuseio de documentos compostos por um conjunto de modelos, que por sua vez agregam conceitos – as unidades de modelagem do domínio tratado. Uma especificação criada por via de tal ambiente é representada pelo conjunto de modelos e conceitos, e pelos relacionamentos existentes entre eles. A Figura 11 ilustra esta estrutura conceitual de documentos criados por extensões do *framework* OCEAN, representada por um diagrama de classe, onde:

- *OCEANDocument*: representa o documento abstrato a ser especificado no ambiente;
- *Specification*: subclasse de *OCEANDocument*, representa concretamente uma especificação de projeto;
- *SpecificationElement*: elemento abstrato de especificação de projeto;
- *ConceptualModel:* subclasse de *SpecificationElement*, representa um Modelo;
- *Concept*: subclasse de *SpecificationElement*, representa um Conceito, ou uma unidade de modelagem.

Documentos obedecem ao padrão Model-View-Controller (MVC), de forma que as representações de conceitos são separadas de suas representações visuais

no modelo.

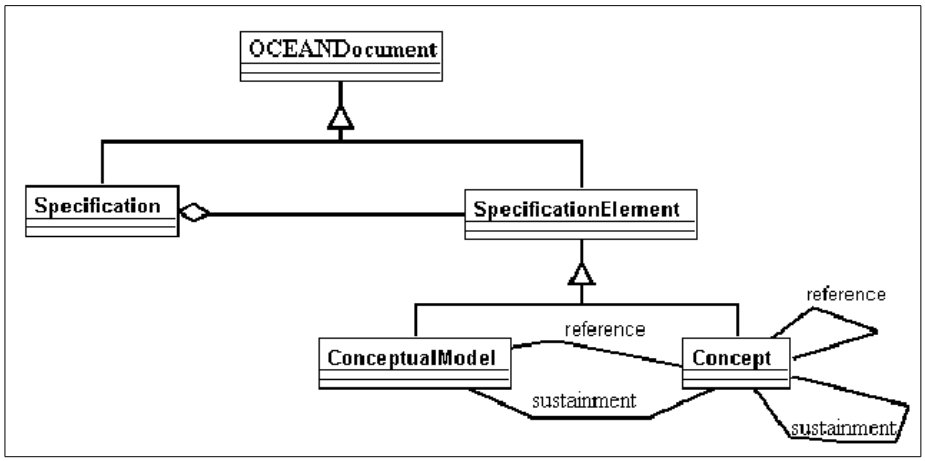

Figura 11: Superclasses do *framework* OCEAN que definem a estrutura de uma especificação (SILVA, 2000. p. 111)

O *framework* OCEAN prevê ainda mecanismos para criação e modificação de especificações. Ele permite a implementação de funcionalidades disponibilizadas para atuar sobre a especificação sendo manipulada. Para tal, um mecanismo digno de menção é o da criação de ferramentas, que permitem ações de edição, análise ou transformação sobre conceitos, modelos ou especificações. Para implementar tal ferramenta, o *framework* oferece uma definição genérica na forma de uma classe abstrata chamada *OceanTool*, que deve ser especializada para cada ferramenta a ser desenvolvida.

A partir do *framework OCEAN*, foi criado o ambiente SEA (SILVA, 2000), com o objetivo de permitir o desenvolvimento e uso de *frameworks* e componentes. A especificação de projetos no SEA por parte do usuário é feita pela criação de Modelos na linguagem UML, a partir da manipulação de editores gráficos associados a cada tipo de Modelo. As técnicas de modelagem disponíveis no ambiente SEA

correspondem a um subconjunto daquelas definidas pela linguagem UML, com adições e padronizações necessárias para a especificação de *frameworks* e componentes. A figura 12 mostra a tela principal do ambiente SEA, com um editor aberto exibindo a modelagem de um diagrama de componentes.

O SEA dispõe ainda de mecanismos de verificação de consistência de especificações, e para a manipulação programática destas especificações e de seus conceitos associados. Obedecendo à estrutura MVC imposta pelo *framework* OCEAN, documentos criados pelo SEA são manipulados em seu nível conceitual (*Model*), e suas modificações são refletidas automaticamente no nível de apresentação (*View*).

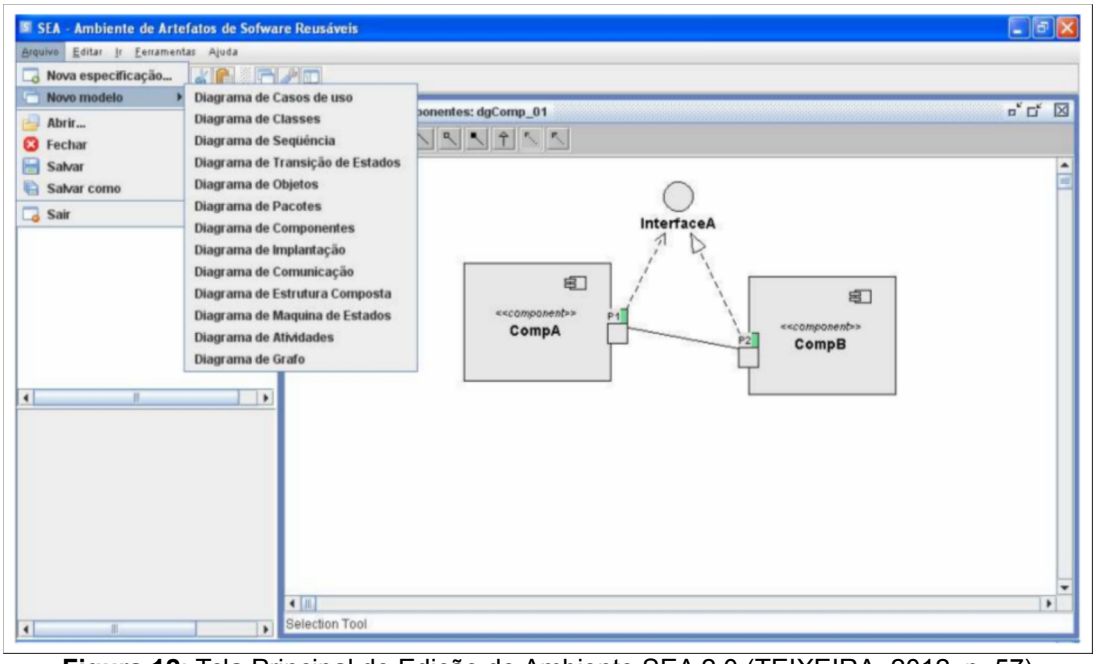

Figura 12: Tela Principal de Edição do Ambiente SEA 2.0 (TEIXEIRA, 2012, p. 57)

Pode-se ainda estabelecer relacionamentos entre especificações diferentes, permitindo a modelagem de elementos em diferentes níveis de uma cadeia

hierárquica, e permitindo o detalhamento gradual entre diferentes níveis de abstração, com partes da especificação separadas em arquivos diferentes.

Como será detalhado na seção 5.1, o SEA suporta o desenvolvimento de componentes através da modelagem UML orientada a objetos, compreendendo três etapas: especificação da interface do componente, definição de componentes e interligando componentes com interfaces compatíveis (SILVA, 2000).

Originalmente implementado na linguagem de programação *Smalltalk*, o *framework* OCEAN passou por uma reengenharia e foi reestruturado e reimplementado na linguagem *Java* (COELHO, 2007).

Da mesma forma, enquanto a versão original do SEA utilizava a biblioteca gráfica *HotDraw*, sua reimplementação utiliza a sua versão Java, chamada *JHotDraw* (AMORIM JUNIOR, 2006). Esta biblioteca se trata também de um *framework* cuja arquitetura de classes permite a criação de interfaces gráficas de usuário (GUI – *Graphical User Interface*), através de janelas, painéis, botões, menus e etc (Figura 12), sendo voltado especificamente para o desenvolvimento de editores gráficos bidimensionais.

Para compreender e manipular o código do ambiente SEA que diz respeito à interface gráfica, o desenvolvedor deve se atentar para a seguinte estrutura de classes (figura 13), onde (AMORIM JUNIOR, 2006):

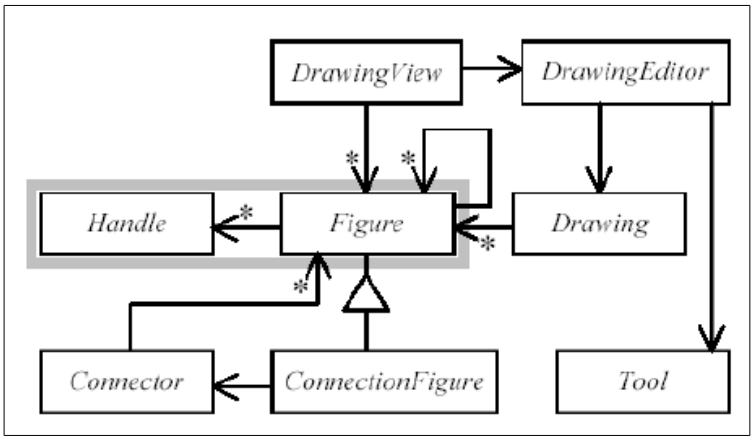

Figura 13: Estutura básica de classes e interfaces do framework JHotDraw (AMORIM JUNIOR, 2006. p. 24)

- *DrawingView*: uma “visão de desenho”, representa uma área onde pode ser exibido um desenho;
- *DrawingEditor*: interface que coordena os diferentes objetos que participam de um editor gráfico, ou seja, que permite ao usuário interagir com um desenho;
- *Drawing*: um container para as figuras do desenho;
- *Figure*: representa uma figura do desenho, que pode ser composta de um conjunto de figuras. Encapsula dados e rotinas necessárias para que a figura seja desenhada na tela;
- *Handle*: define pontos de acesso à figura e determinam como se pode interagir com ela, além de registrar todas as modificações feitas;
- *ConnectionFigure*: representa “figuras de linha” como, por exemplo,

transições de diagramas de máquina de estado UML.

- Connector: representam as extremidades de um *ConnectionFigure,* possivelmente relacionando-a com outra figura do desenho;
- *Tool*: representa uma ferramenta do editor ativo do desenho, e recebe todos os eventos gerados pela interação do usuário sobre o desenho (*DrawingView*).

5. TESTE AUTOMATIZADO DE INTERFACE DE COMPONENTE

Esta seção descreve uma proposta de suporte a testes automatizados de interface de componente, no desenvolvimento de um componente através do ambiente SEA. Para tal, primeiro (seção 5.1) aborda os passos para a especificação necessária (estrutural, comportamental e funcional) da descrição externa do componente. Depois (seção 5.2), descreve o funcionamento e o resultado de uma ferramenta que automaticamente cria uma definição de interface de *componente-espelho* a partir deste componente sendo desenvolvido. A seção 5.3 aplica a análise de compatibilidade (conceituada anteriormente na seção 2.2) entre a interface do *componente-espelho* e a interface do componente sendo desenvolvido, através da ferramenta implementada por Teixeira (2012) e uma ferramenta simples de análise funcional e, por fim, a seção 5.4 discute quais os aspectos envolvidos na geração automática de casos de testes que validem a implementação do componente em relação à sua interface, em nível de código.

Ao longo desta seção será utilizado o exemplo desenvolvido por Rodrigues (2015), que em seu trabalho aplicou o padrão de desenvolvimento orientado a componentes de Silva (2009) para reestruturar um jogo digital previamente desenvolvido sem o uso de tais padrões, a fim de aprimorar sua manutenibilidade e reusabilidade. O jogo exemplo é um *Space Shooter*, em que o jogador controla uma nave espacial em um ambiente bidimensional, desviando e atirando em naves inimigas que se aproximam com o passar do tempo (RODRIGUES, 2015). Dentre os componentes que fazem parte da implementação deste jogo (Figura 14), foi escolhido para servir de exemplo o componente *FonteInimigos*, que no sistema tem a responsabilidade de instanciar e inicializar dinamicamente os elementos inimigos

que antagonizarão o jogador, representados pelo componente *Inimigo.*

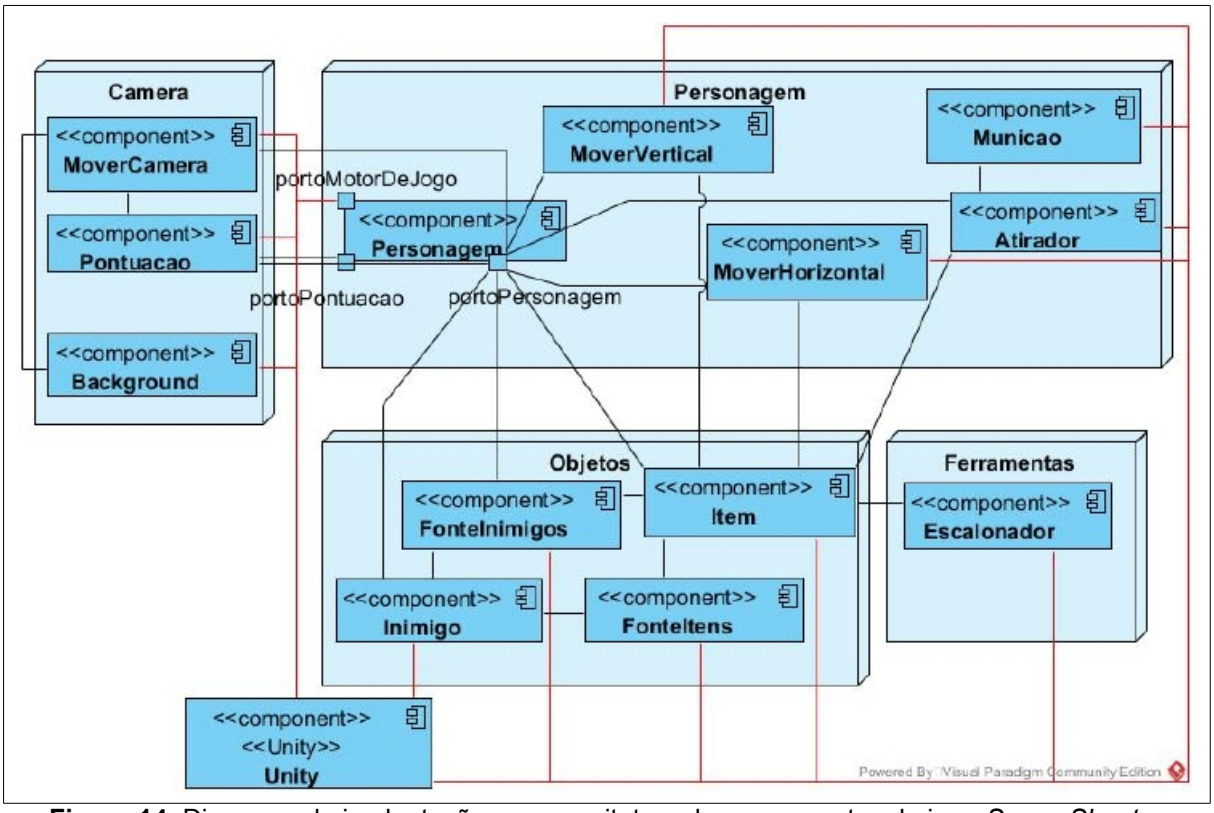

Figura 14: Diagrama de implantação com arquitetura de componentes do jogo *Space Shooter* (RODRIGUES, 2015, p. 85)

As ferramentas descritas nesta seção foram implementadas dentro do ambiente SEA a partir de extensões da classe abstrata *OceanTool* (figura 15), herdada do *framework* OCEAN (seção 4). As subclasses *MirrorComponentGeneratorTool* e *MirrorComponentAnalyzerTool* foram criadas no escopo deste trabalho (seções 5.2 e 5.3), e as subclasses *ComponentStructuralAnalyzerTool* e *ComponentBehaviorAnalyserTool* foram implementadas no trabalho de Teixeira (2012), e foram adaptadas e reutilizadas neste trabalho para a execução dos testes de interface de componentes (seção 5.3).

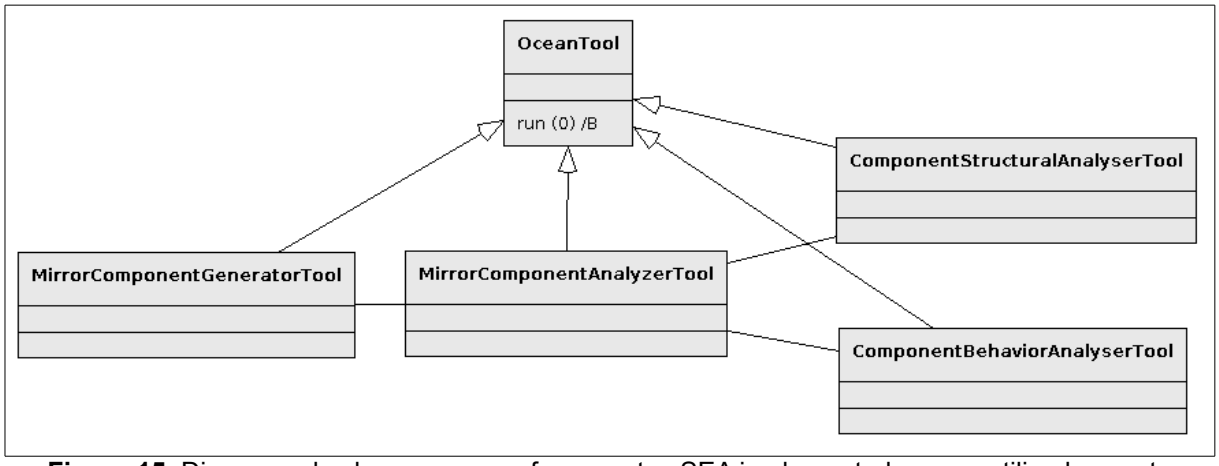

Figura 15: Diagrama de classes com as ferramentas SEA implementadas ou reutilizadas neste trabalho

5.1 Especificação de um componente através do ambiente SEA

Utilizando o componente *FonteInimigos* (RODRIGUES, 2015) como exemplo, esta subseção descreve os passos iniciais de sua especificação. Como visto na seção 2.1 (Figura 7), o primeiro passo é especificar sua descrição externa, ou seja, sua Interface de Componente. Isto será feito utilizando o ambiente de desenvolvimento SEA. Os passos aqui descritos são pré-requisitos para a execução da ferramenta descrita na seção 5.2.

O primeiro passo é o da modelagem estrutural, como descrita na seção 2.1.1, com a elaboração dos diagramas de classe e de componentes, que possuem o objetivo de especificar quais as *Interfaces UML* relacionadas ao componente, e sob quais *portos* acontecem estes relacionamentos. As Figuras 16 e 17 mostram tais diagramas, obtidos a partir de Rodrigues (2015).

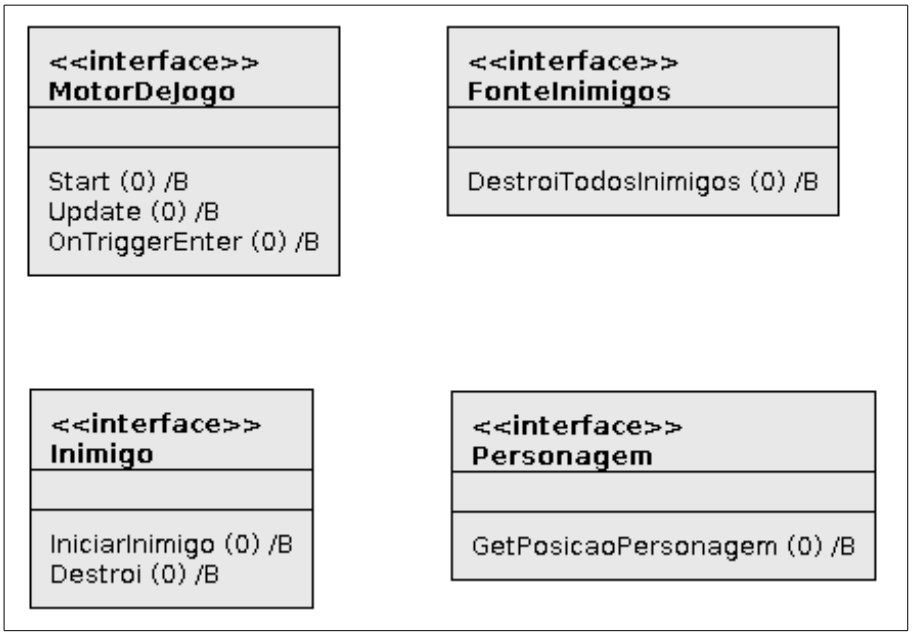

Figura 16: Diagrama de classes com interfaces UML que interagem com o componente *FonteInimigos*

Através do diagrama da figura 17 pode-se observar que o componente *FonteInimigos* realiza (ou seja, implementa) as interfaces *FonteInimigos* e *MotorDeJogo*, ao mesmo tempo que depende (é cliente) das interfaces *Inimigo* e *Personagem*.

Rodrigues (2015, p. 86) explica que o componente depende de *Personagem* para poder obter a posição do personagem do jogador na tela, e de *Inimigo* para poder instanciar e inicializar os inimigos do jogador. Ao mesmo tempo, ele implementa as interfaces *MotorDeJogo* para que receba chamadas de atualização do *framework,* e para que outros componentes do sistema possam utilizar suas funcionalidades.

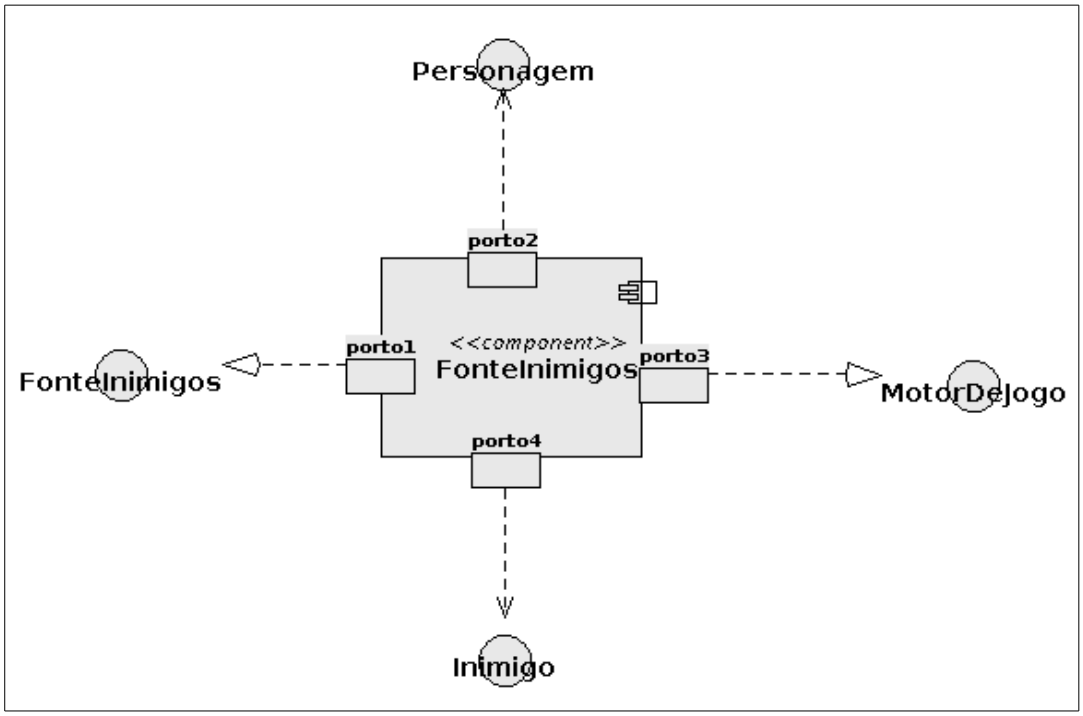

Figura 17: Diagrama de componentes mostrando o relacionamento entre o componente *FonteInimigos* e interfaces UML

A seguir (Figura 18), a descrição comportamental do componente deve ser feita através do diagrama de máquina de estados, representando as restrições de invocações dos métodos relacionados ao componente, como elaborado na seção 2.1.2.

Vemos na descrição comportamental que a maior parte do funcionamento do componente *FonteInimigos* é determinado por chamadas a métodos implementados da interface *MotorDeJogo*, chamados pelo framework do sistema. Durante a execução do jogo, ele fica no estado “Esperando” aguardando por chamadas do método *update()* que levam à geração de inimigos para combater o jogador. Observa-se que a ausência de um *pseudo-estado* final indica que a execução do componente pode ser finalizada a partir de qualquer dos estados descritos.

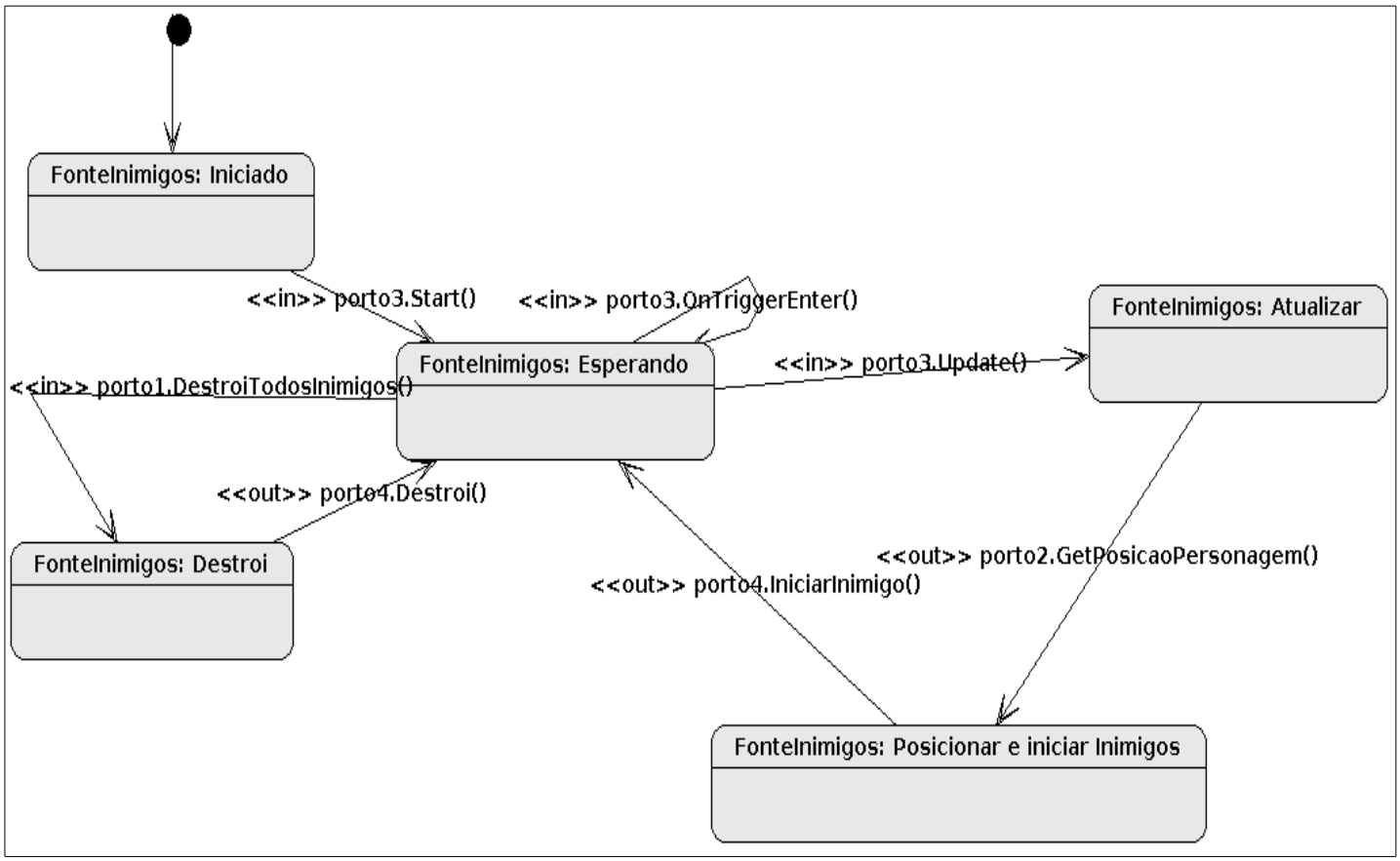

Figura 18: Diagrama de máquina de estados descrevendo o comportamento do componente *FonteInimigos*

Por último, é necessário modelar as funcionalidades do componente, seguindo o padrão descrito na seção 2.1.3, ou seja, através de um diagrama de casos de uso (Figura 19) indicando quais funcionalidades – do ponto de vista externo ao componente – ele oferece. Cada um dos casos de uso deve ser detalhado através de um diagrama de atividades, explicando *o que faz* e não *como o faz.* Um exemplo de caso de uso é detalhado na Figura 20.

Os atores relacionados aos casos de uso do componente não serão abordados nesta descrição. Mais detalhes podem ser encontrados no trabalho de Rodrigues (2015).

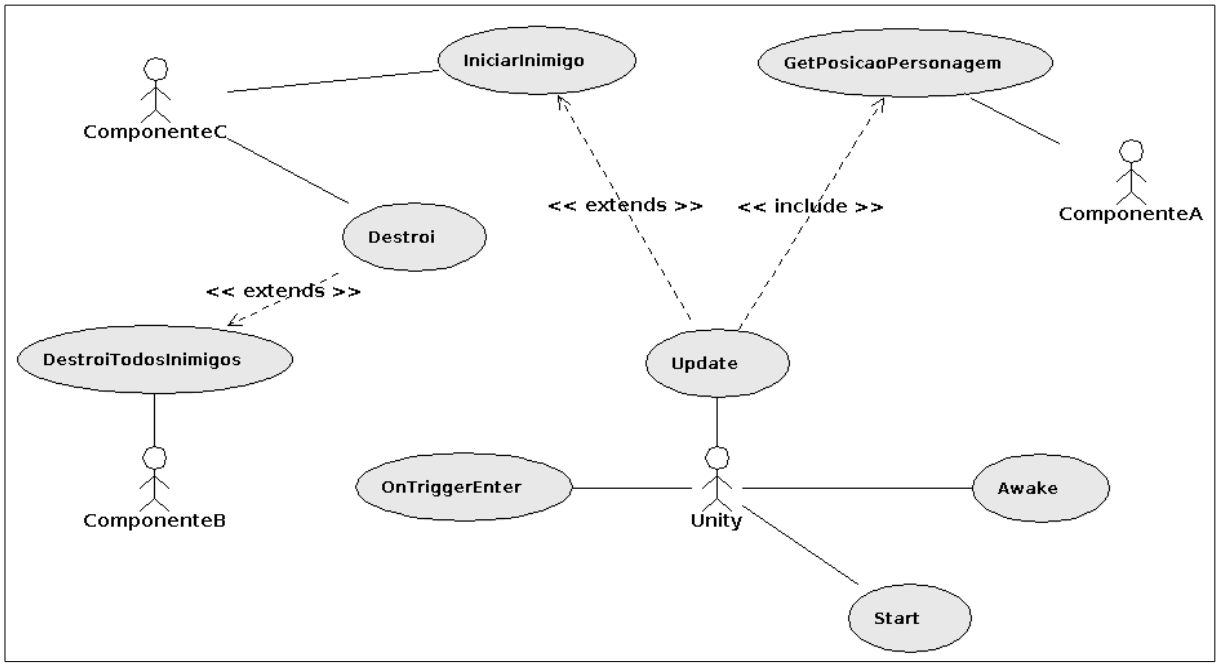

Figura 19: Diagrama de casos de uso do componente *FonteInimigos*

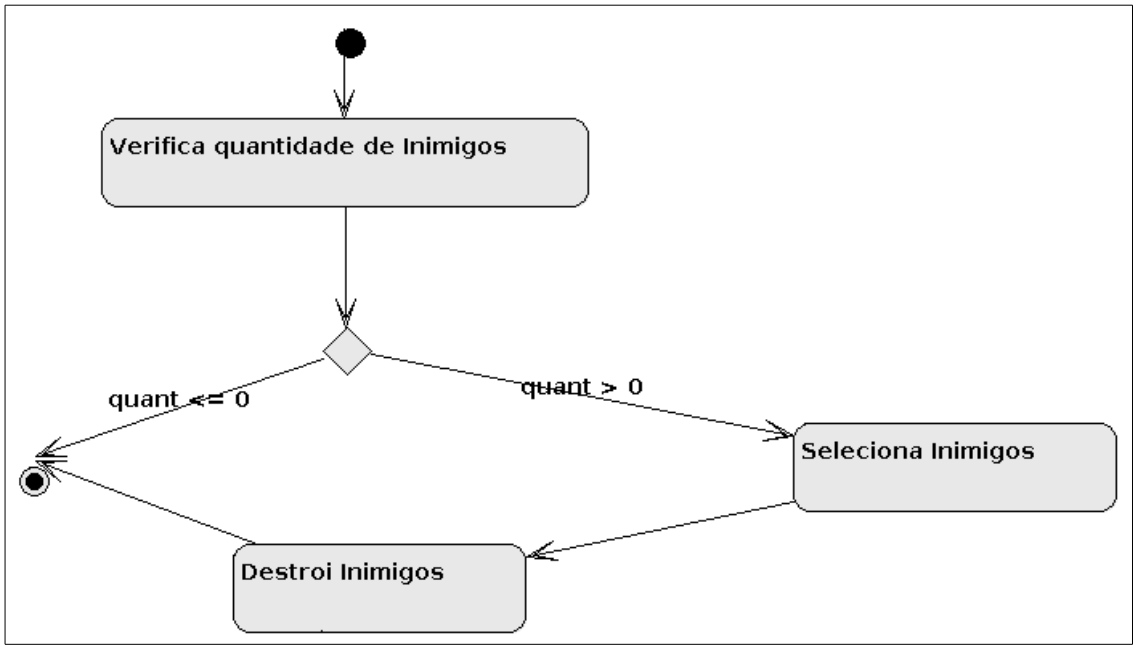

Figura 20*:* Exemplo de detalhamento de caso de uso (*DestroiTodosInimigos*) do componente *FonteInimigos*

5.2 Geração da interface do Componente-Espelho

Com as modelagens estrutural, comportamental e funcional do componente *FonteInimigos* disponíveis, este pode ter a interface do seu *componente-espelho* gerado a partir do SEA. Isto é feito através do item “Gerador de Componente-Espelho” do menu “Ferramentas”, como mostra a Figura 21. Após se selecionar qual componente será espelhado (Figura 22), a ferramenta inicia sua execução.

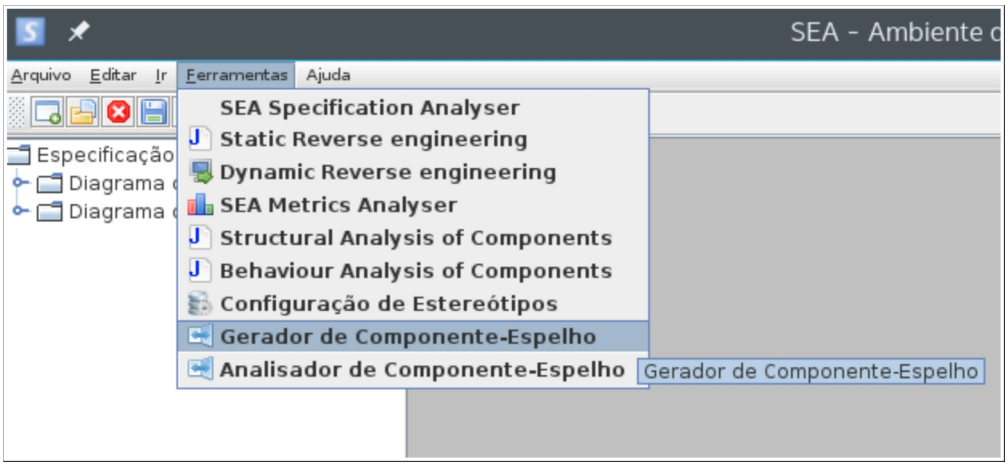

Figura 21*:* Parte da tela principal do SEA com o botão selecionado para se iniciar a geração do *componente-espelho*

Esta funcionalidade aqui descrita foi implementada dentro do ambiente SEA na forma da classe *MirrorComponentGeneratorTool* (figura 15).

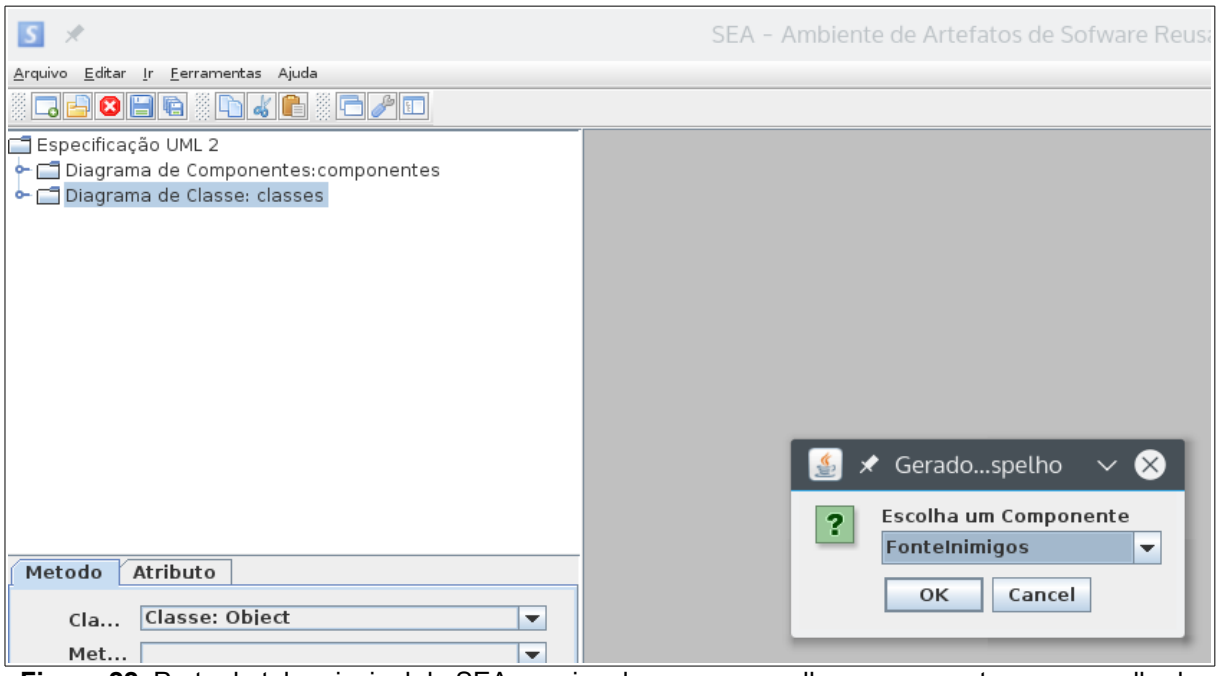

Figura 22*:* Parte da tela principal do SEA com janela para se escolher componente a ser espelhado

Uma vez executada, a ferramenta realiza os seguintes passos, que serão detalhados ao longo desta seção:

1. Checar consistência da especificação;
2. Obter elementos relacionados ao componente selecionado;
3. Clonar interfaces UML relacionadas ao componente selecionado;
4. Criar diagrama de componentes separado;
5. Criar *componente-espelho* no novo diagrama de componentes;
6. Criar diagrama de implantação para estabelecer relacionamento entre componentes;
7. Espelhar diagrama de máquina de estado.
8. Clonar diagramas de atividades

Primeiro (passo 1), a ferramenta verifica que todos os diagramas necessários – como exemplificados na seção 5.1 – estão presentes e corretamente especificados. Além disso, é necessário que o componente selecionado para ser espelhado tenha um mínimo de informações detalhadas: deve ter no mínimo 1 porto com 1 interface UML associada, e esta interface deve ter no mínimo 1 método declarado.

O passo 2 percorre as estruturas de dados do SEA que guardam os elementos da especificação, ou seja, os conceitos representados nos diagramas (componentes, portos, relacionamentos, etc), para obter referências a todos aqueles elementos que se relacionam com o componente selecionado e que serão utilizados posteriormente pela ferramenta.

Todas as interfaces relacionadas ao componente selecionado são “clonadas” no passo 3. Elas passam a servir como um *snapshot*, ou uma “foto” do estado delas no momento em que a ferramenta foi executada, para que possíveis modificações na interface (eg. Nome de métodos, tipo e número de parâmetros, etc) possam ser comparadas com a descrição externa do componente a fim de se verificar inconsistências inseridas posteriormente. Um padrão de nomeação para interfaces clonadas foi adotado: concatenando o sufixo “*_clone*” no nome original da interface. Esta etapa também adiciona estes clones de interface no diagrama de classes existente (Figura 23).

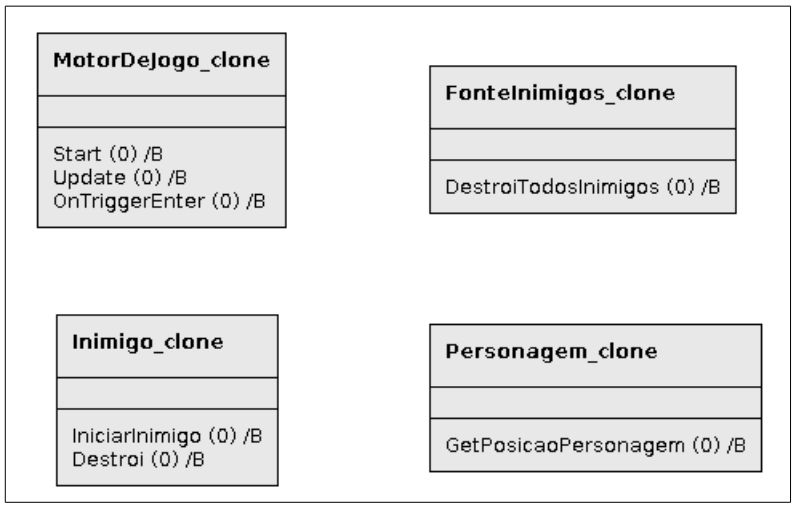

Figura 23: Trecho de diagrama de classes com clones de interfaces UML relacionadas a *FonteInimigos*

No passo 4, o componente selecionado é copiado para um novo diagrama de componentes. Isto serve para separar o componente a ser desenvolvido de quaisquer outros componentes do sistema, e assim facilitar o uso da ferramenta em especificações grandes e/ou complexas. Além do componente selecionado, o novo diagrama de componentes conterá também o *componente-espelho* (passo 5). Este novo componente "espelha" o relacionamento do original com suas interfaces, ou seja, um relacionamento de *dependência* do original para uma interface resultará na criação de um relacionamento de *realização* do *espelho* para o clone daquela interface. A Figura 24 mostra o trecho do novo diagrama de componentes que inclui o *componente-espelho*, e que pode ser comparado com a Figura 17 (seção 5.1) da especificação original do componente *FonteInimigos:* enquanto o original implementa as interfaces *MotorDeJogo* e *FonteInimigos*, e depende das interfaces *Personagem* e *Inimigo;* seu *espelho* implementa as interfaces *Personagem_clone* e *Inimigo_clone*, e depende das interfaces *MotorDeJogo_clone* e *FonteInimigos_clone*. Isto significa que o novo componente serve como um complemento perfeito para o

original, e vice-versa.

Outro padrão de nomeação foi adotado para todos os elementos “espelhados” do componente original (o próprio componente, seus portos, etc.): através da concatenação de “*_mirror*” ao nome do elemento original correspondente.

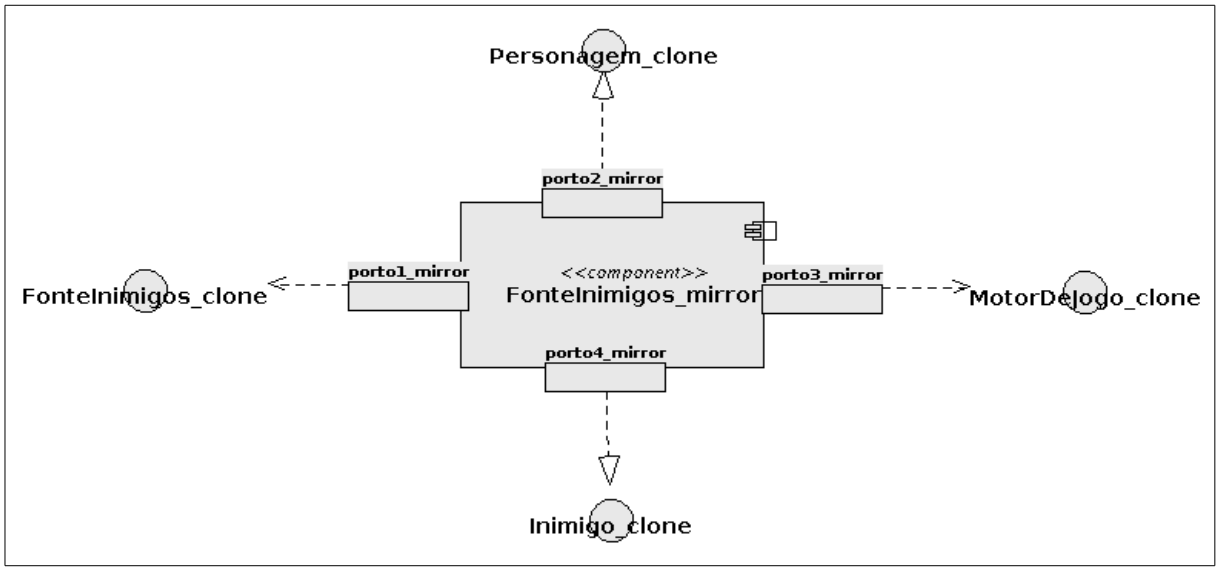

Figura 24: Trecho do novo diagrama de componentes mostrando o *componente-espelho* do componente *FonteInimigos*

No passo 6 um diagrama de implantação é criado, explicitando a associação do componente original com seu espelho, e permitindo que a Ferramenta de Análise de Compatibilidade seja executada entre os dois, como será abordado na seção 5.3 abaixo. A Figura 25 mostra o novo diagrama de implantação gerado.

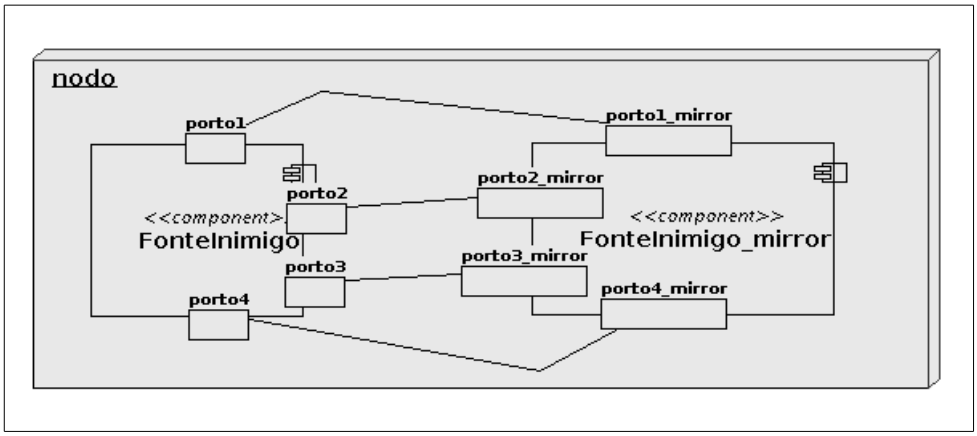

Figura 25: Diagrama de implantação associando componente *FonteInimigos* com seu espelho

Até este ponto, a ferramenta se preocupou em espelhar elementos da descrição estrutural (ver seção 2) do componente selecionado. No passo 7 a descrição comportamental na forma de diagrama de máquina de estados é espelhada. Isto significa que o comportamento do *componente-espelho* será criado de forma que sempre obedeça às restrições de ordem de invocação de métodos do original. Seguindo o padrão de rótulo de transições de diagramas de máquina de estado detalhado na seção 2.1.2, esta nova descrição comportamental do *componente-espelho* é criada a partir das seguintes conversões aplicadas à original:

- Para cada estado do diagrama original, um novo estado é criado utilizando o padrão de nomeação: nome original concatenado com "*_mirror*";
- As mesmas transições entre estados do original são criadas entre os estados espelhados correspondentes;
- O sentido de invocação dos gatilhos das transições é invertido, ou seja, "in" da transição original se transforma em "out" na transição espelhada correspondente, e vice-versa;

- O porto de cada transição original é substituído pelo porto espelhado correspondente, seguindo mesmo padrão de nomeação ("*_mirror*");
- As assinaturas dos métodos invocados em cada transição são mantidos iguais.

A partir da especificação comportamental do componente *FonteInimigos* (Figura 18), a ferramenta criará o diagrama de máquina de estados da Figura 26, descrevendo o comportamento do componente-espelho *FonteInimigos_mirror*. Este comportamento também complementa perfeitamente o original, ou seja, enquanto o componente *FonteInimigos* aguarda no estado "Esperando" pela chamada de *update()* através do porto *porto3* para avançar na criação de inimigos do jogo, o componente *FonteInimigos_mirror* avança do estado "Esperando_mirror" fazendo tal chamada de método a partir do porto *porto3_mirror,* e assim sucessivamente para os demais estados. Isto significa que enquanto o componente original não tiver sua Interface de Componentes alterada, o seu *componente-espelho* sempre obedecerá às restrições de invocação do original.

Por fim, o último passo (passo 8) clona os diagramas de Atividades que descrevem as funcionalidades do componente sendo especificado.

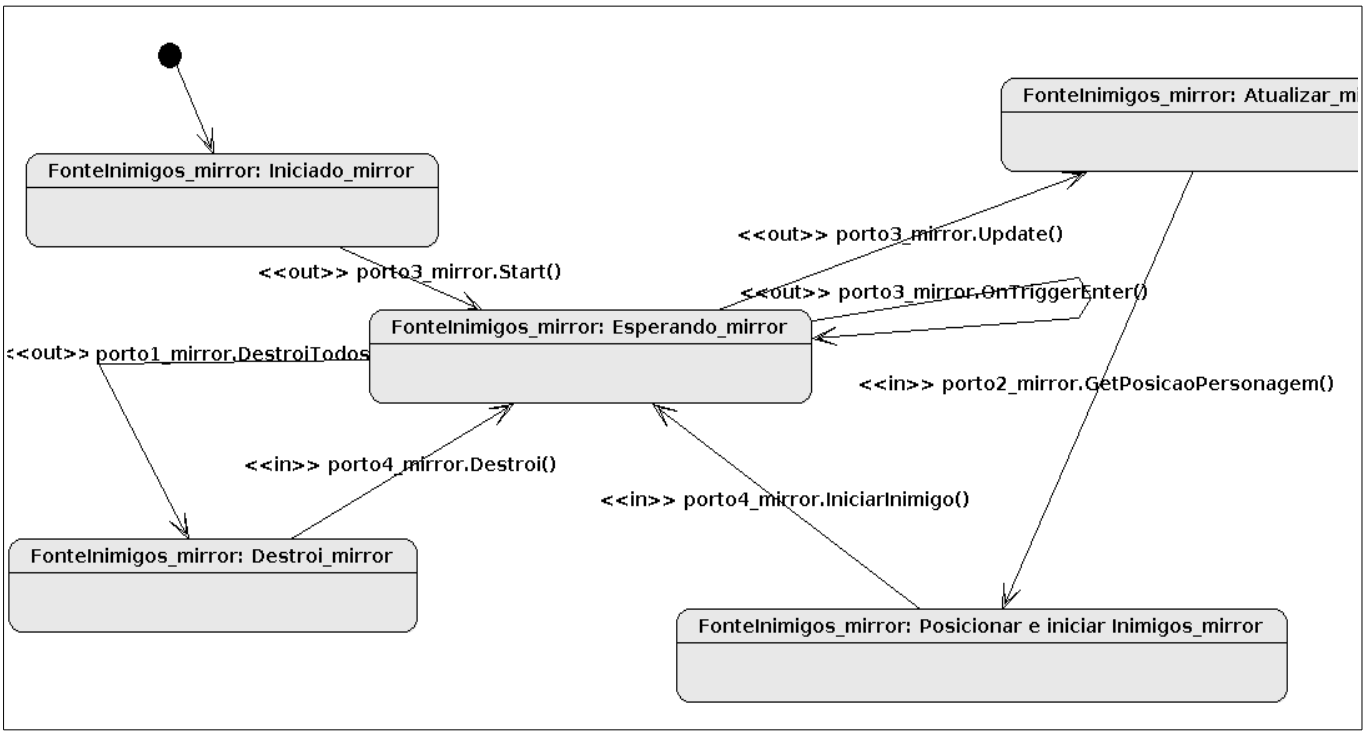

Figura 26: Diagrama de máquina de estados descrevendo o comportamento do componente-espelho *FonteInimigos_mirror*

5.3 Análise de Compatibilidade

Após criada a interface do *componente-espelho,* o componente original pode ter sua especificação modificada como parte do processo de desenvolvimento ou manutenção. Para avaliar se, após estas modificações, o componente original continua obedecendo à especificação de sua interface, ou seja, se ele continua *compatível* com seu *componente-espelho*, uma segunda ferramenta implementada no SEA, na forma da classe *MirrorComponentAnalyzerTool,* permite a execução combinada das Ferramentas de Análise Estrutural e Comportamental (TEIXEIRA, 2012) descritas na seção 2.2 (figura 27), além da execução de uma análise de compatibilidade funcional simples implementada neste trabalho.

Mais especificamente, a avaliação é composta de três etapas:

1. Análise de Compatibilidade Estrutural: executa e compila o resultado da ferramenta descrita na seção 2.2.1. Como visto anteriormente, esta análise detecta erros como ausência de método requerido ou fornecido e diferenças entre tipo de retorno, número e tipo de parâmetros destes métodos;

2. Análise de Compatibilidade Comportamental: executa e compila o resultado da ferramenta descrita na seção 2.2.2. Como visto anteriormente, esta análise detecta erros como os de ordem de invocação de métodos fornecidos ou requeridos, de métodos nunca executados ou indisponíveis;

3. Análise de Compatibilidade Funcional: faz uma comparação simples entre os diagramas de atividade do componente original com o do *componente-espelho*, servindo como uma análise de compatibilidade primitiva. Esta análise serve para detectar se existe qualquer diferença entre estes diagramas.

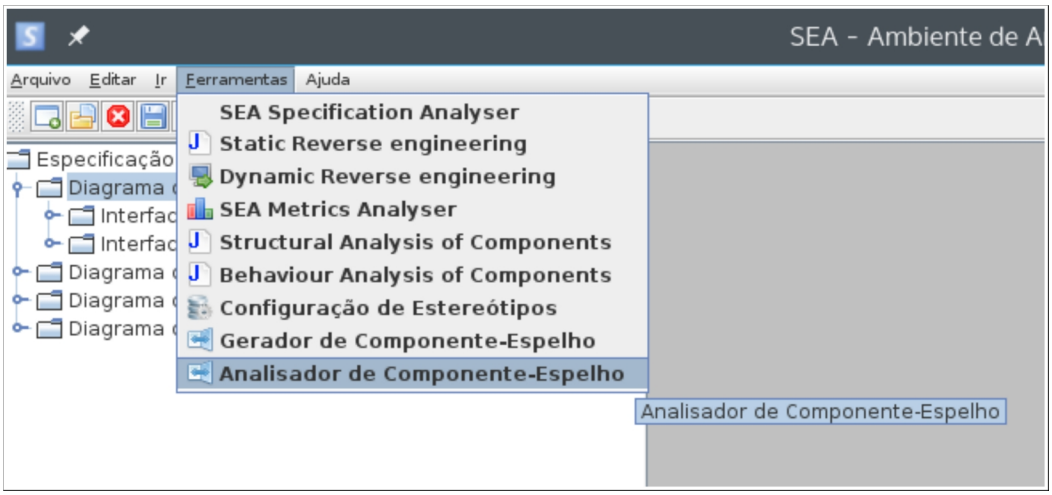

Figura 27*:* Parte da tela principal do SEA com o botão selecionado para se iniciar a análise do *componente-espelho*

Executada a ferramenta, ela relaciona os problemas de incompatibilidade estrutural, comportamental e funcional (neste aspecto de forma limitada), garantindo que nestes quesitos o componente especificado esteja consistente com o que a sua definição de interface espera. Os resultados são exibidos em relatórios textuais como mostra a figura 28.

Os testes automatizados de interface de componentes implementados permitem que o desenvolvimento de componentes no ambiente SEA itere entre modelagem e análise de compatibilidade, e que a todo momento se garanta a compatibilidade do componente sendo desenvolvido para com sua interface inicial, ou seja, que seja sempre compatível com seu *componente-espelho*.

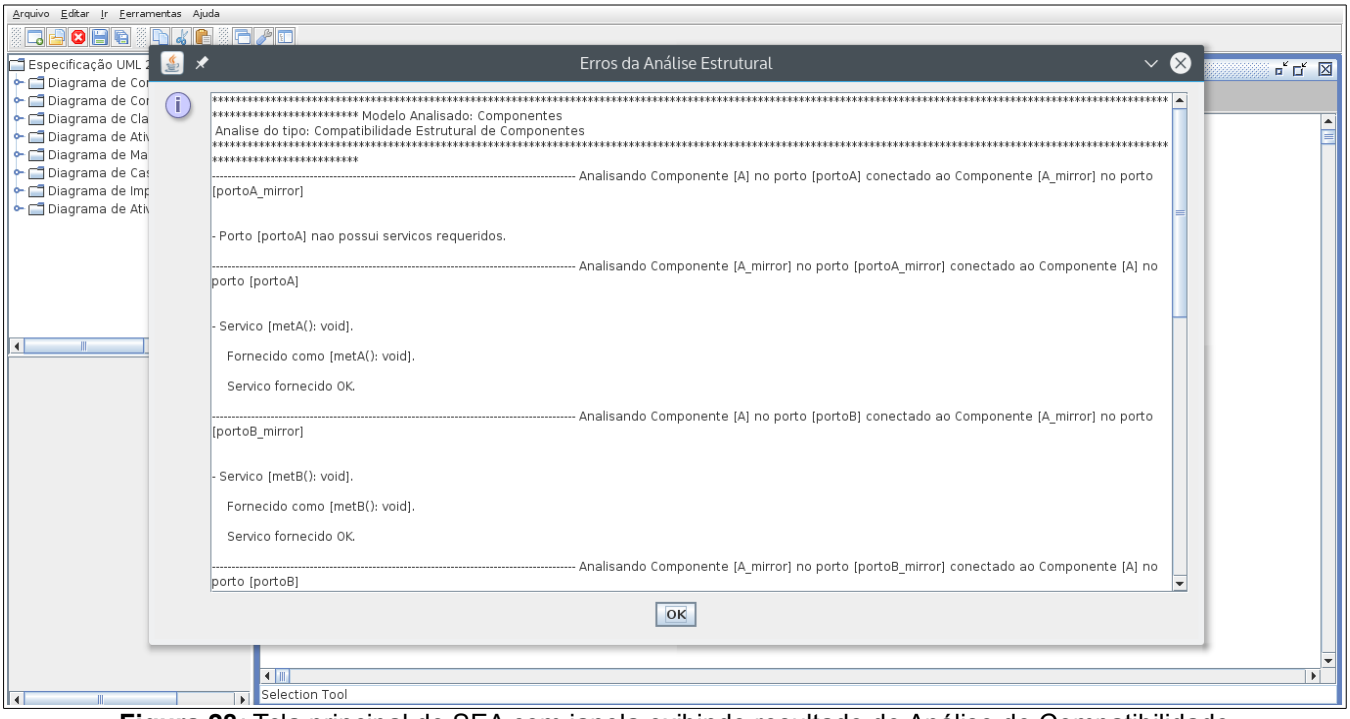

Figura 28*:* Tela principal do SEA com janela exibindo resultado de Análise de Compatibilidade

5.4 Geração automática de código de testes (discussão)

A ferramenta apresentada propôs uma forma de se executar testes de interface de componente em nível de modelo através do ambiente SEA. Para que ela seja útil no desenvolvimento completo ou na manutenção de um componente, prevê-se a necessidade da geração de testes em nível de código, para que seja validada a implementação do componente de software em relação à sua interface de componente, em busca de modificações em sua operação que tenham sido feitas sem a atualização correspondente em sua interface. Embora este próximo passo fuja do escopo deste trabalho, esta seção discute alguns aspectos relacionados a ele.

Esta ferramenta imaginada funcionaria como uma função que receberia como entrada o resultado da especificação (em modelo) de interface do *componente-espelho* obtida da seção 5.2, e como saída geraria, automaticamente ou semi-automaticamente, um artefato de software com o conjunto de testes necessário para que se garanta a compatibilidade estrutural, comportamental e funcional entre a implementação de componente e sua interface.

Este conjunto de testes gerados poderia se situar:

- Externamente à implementação do *componente-espelho,* a partir de um Cliente de Testes (figura 29) responsável por instanciar tanto o componente original quanto o *espelho*, e coordenar a execução dos casos de testes. Esta abordagem é adequada para testes em tempo de desenvolvimento, pelo fato de exigir serviços do Cliente de Testes que não necessariamente são

empacotados e distribuídos junto com o componente original;

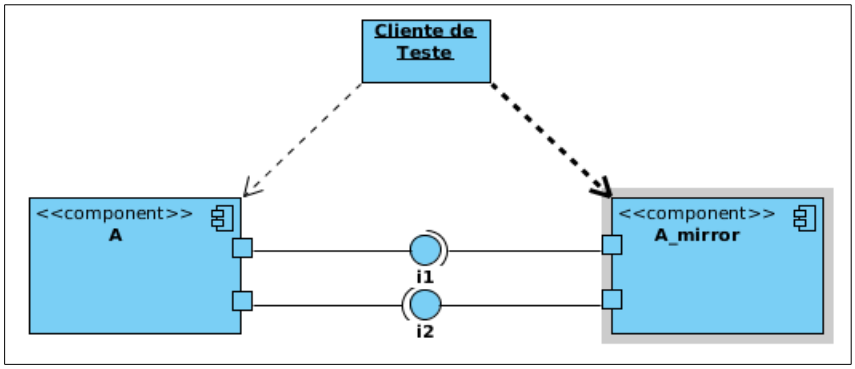

Figura 29*: C*liente de testes externo ao *componente-espelho* (adaptado de TEINIKER, 2003)

- Encapsulado na implementação do *componente-espelho* (figura 30), que pode inclusive ser empacotado e distribuído junto com o componente original, para que este aja como um *componente executável auto-testável*. Esta abordagem permite que os testes sejam realizados em tempo de execução por parte do cliente do componente desenvolvido, que age como uma metáfora sobre componentes de *hardware* auto-testáveis (GROSS, 2005).

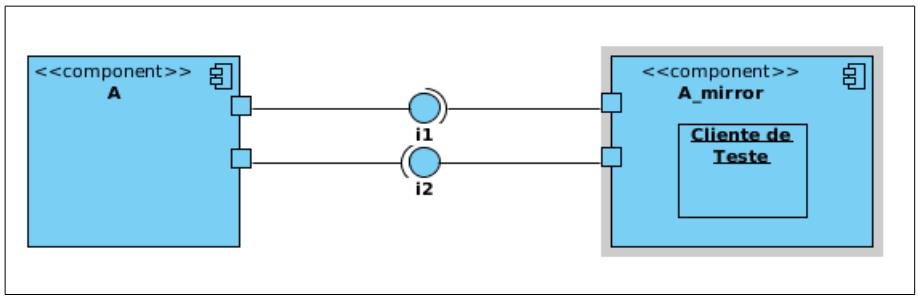

Figura 30*:* Cliente de testes encapsulado no *componente-espelho*

- Intermediando todas as mensagens entre o componente e seu *espelho* (figura 31), de forma similar à primeira alternativa apresentada (figura 29), com a diferença que neste caso o componente não se relaciona diretamente com o

componente-espelho.

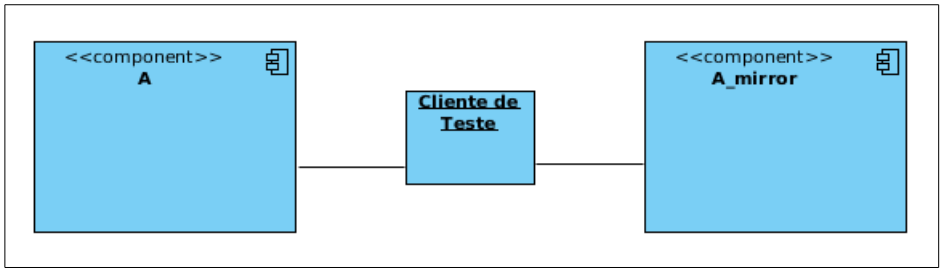

Figura 31*: C*liente de testes intermediando relacionamento entre componente e *espelho*

Como visto na seção 3, os desafios de se testar componentes (e de modo geral, sistemas orientados a componente) se originam principalmente da ausência de acesso à implementação interna dos componentes. A especificação da interface do componente original e a do *componente-espelho* devem ser informação necessária para a geração semi-automática dos testes. Isto significa que, embora a compatibilidade estrutural entre o componente e sua interface sejam mais facilmente validados pelos mecanismos sintáticos da linguagem de programação, a compatibilidade comportamental e funcional podem não ser 100% automatizáveis, por lidarem com aspectos semânticos do componente, que podem ir além da especificação da interface de componente.

A representação das restrições comportamentais de um componente neste trabalho foi feita através do diagrama de máquina de estados UML (seção 5), que estabelece as restrições de ordem de invocação de métodos entre o componente e seu *espelho*. Neste aspecto, a geração de testes deve se preocupar em validar em código a implementação da máquina de estados (ou seja, o comportamento do componente sendo testado), garantindo que:

- Ao se chamar seus métodos fornecidos, o componente transita entre os estados corretos, como especificado na interface;
- Para executar usas operações, o componente realiza chamadas aos métodos dos quais depende, na ordem especificada na interface;
- As chamadas a métodos requeridos ou fornecidos do componente obedecem às restrições de parâmetros (tipo, número, etc) especificadas na interface;

Testar uma implementação de máquina de estados requer que possamos fazer ao menos duas operações, de alguma forma fornecidas pelo componente a ser testado (GROSS, 2005):

- Operação de checagem de estado, para que se identifique o estado da máquina a qualquer momento;
- Operação de estabelecimento de estado, para que se possa identificar o estado final desejado após uma determinada transição.

Estes estados a serem testados representam estados *lógicos*, e não são necessariamente estados internos do componente, mas uma representação de acordo com a especificação de sua interface, ou seja, aquilo que o componente permite ao ambiente externo verificar através de sua descrição. Por isto estes testes ainda podem ser considerados de *caixa-preta* (seção 3).

Um caso de testes de uma máquina de estados relaciona operações a serem

testadas com um estado inicial a ser definido como pré-condição, parâmetros de entrada para a operação testada e um estado final como pós-condição desejada, a ser comparado com o estado verificado após a simulação da transição por parte do teste. Estes conceitos são ilustrados na estrutura da figura 32.

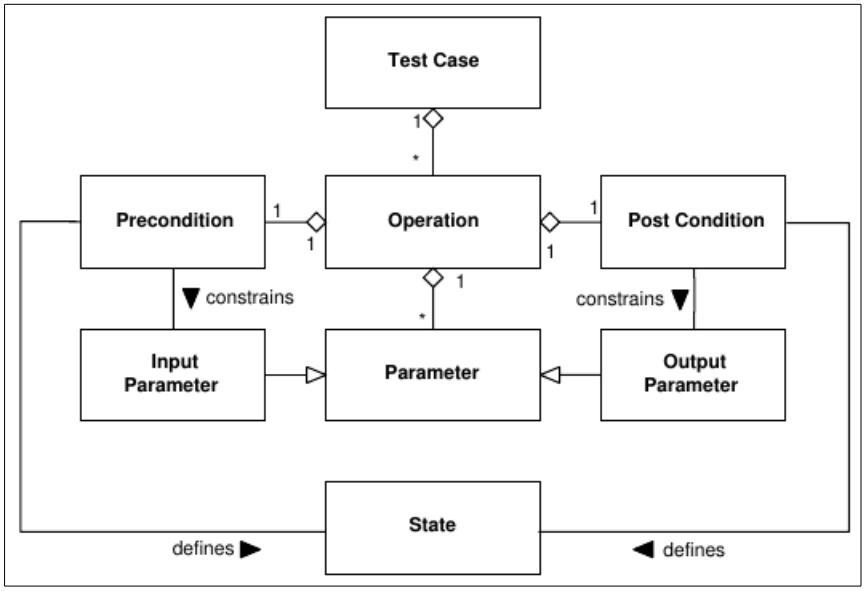

Figura 32*:* Representação UML dos conceitos de um caso de teste de máquina de estados (GROSS, 2005)

Percorrer e testar os estados lógicos da implementação do componente testado traz os seguintes desafios associados:

- Definição do início de execução: é necessário que se possa partir de qualquer dos estados existentes na máquina de estados, não só do componente sendo testado, como do *componente-espelho* relacionado a ele. Isto acontece porque como as restrições comportamentais descrevem a ordem de invocação de métodos tanto requeridos como fornecidos pelo componente original, é necessário se prever casos de uso que iniciam a partir de

chamadas feitas *ao* componente original, vindas de fora;

- Definição de abrangência desejada: todos os estados da máquina de estados devem ser testados para garantir a qualidade da implementação do componente. Além disso, os métodos envolvidos nas transições entre um estado e outro podem receber valores de parâmetros variáveis que devem ser testados. Isto significa que é necessário que se possa determinar um critério mínimo a ser atendido antes que se considere que um número satisfatório de transições foram testados;

- Controle de parada: métodos podem estar inseridos em laços de repetição, ou o componente deve atender a requisitos de tempo máximo de resposta de métodos requeridos e fornecidos. É necessário que se possa controlar a parada da execução do teste a todo momento;

- Parametrização de testes: mesmo que os testes sejam gerados automaticamente, deve ser previsto um mecanismo de parametrização dos próprios testes para se controlar não só os parâmetros dos métodos responsáveis pelas transições, como outras variáveis de ambiente;

- Obtenção de *feedback*: todas as informações a respeito dos conceitos relacionados ao caso de teste, como ilustrados na figura 32, devem ser disponibilizados ao desenvolvedor. Isto significa, por exemplo, que podem ser necessárias estruturas de dados que registrem informações internas da implementação do *componente-espelho*, representando quais métodos requeridos pelo componente testado foram invocados, em qual ordem, etc.

6. CONCLUSÃO

No presente contexto de crescente complexidade de sistemas de software, vislumbra-se a adoção do Desenvolvimento Orientado a Componentes como possível “próximo passo” de abstração acima do paradigma orientado a objetos.

A garantia de qualidade dos componentes é primordial para a reusabilidade de componentes, e argumenta-se que uma ferramenta de teste automatizado de interface de componentes contribui para a garantia de qualidade.

Como resultado deste trabalho foram criadas duas ferramentas executáveis no ambiente de desenvolvimento SEA: uma que gera automaticamente uma especificação de interface de componente *espelho* que complementa àquela de um componente com interface pré especificada, e uma segunda ferramenta que analisa estas duas interfaces de componente para detectar eventuais modificações que possam tornar a especificação do componente inconsistente com sua interface.

Não se teve a pretensão neste trabalho de construir uma ferramenta ou metodologia completa de desenvolvimento, mas espera-se que a descrição do artefato desenvolvido sirva de exemplo de uma combinação destes conceitos estudados.

Como trabalhos futuros, contempla-se a possibilidade de se utilizar tal abordagem como parte de ferramentas de desenvolvimento orientado a modelos, através de t*estes baseados em modelo,* até os baixos níveis de abstração através da geração automática de código, incluindo de testes unitários.

REFERÊNCIAS BIBLIOGRÁFICAS

AMORIM JUNIOR, J. **Integração dos Frameworks JhotDraw e Ocean para a produção de objetos visuais a partir do Framework Ocean.** Florianópolis, 2006. 116 p. Monografia (Graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico. Orientador: Prof. Dr. Ricardo Pereira e Silva.

BECK, K. **Test-Driven Development By Example**. Boston: Addison-Wesley, 2002

COELHO, A. **Reengenharia do Framework OCEAN**. Dissertação (Mestrado) - Programa de Pós-graduação em Ciência da Computação. Universidade Federal de Santa Catarina. Florianópolis: 2007. Orientador: Prof. Dr. Ricardo Pereira e Silva.

CUNHA, R. S. **Suporte à Análise de Compatibilidade Comportamental e Estrutural entre Componentes no Ambiente SEA**. Dissertação (Mestrado) – Programa de Pós-graduação em Ciência da Computação. Universidade Federal de Santa Catarina. Florianópolis: 2005. Orientador: Prof. Dr. Ricardo Pereira e Silva.

GAO, J.; WU, Y.; TSAO, H.S.J. **Testing and Quality Assurance for Component-Based Software**. Boston: Artech House, 2003.

GROSS, H.G. **Component-Based Software Testing with UML.** Berlim: Springer, 2005.

IEEE. **Systems and software engineering-vocabulary.** [s.l.], 2010.

MOU, D.; RATIU, D. **Binding requirements and component architecture by using model-based test-driven development**. 2012 First Ieee International Workshop On The Twin Peaks Of Requirements And Architecture (twinpeaks), [s.l.], p.27-30, set. 2012. Institute of Electrical and Electronics Engineers (IEEE). http://dx.doi.org/10.1109/twinpeaks.2012.6344557.

RODRIGUES, Y. W. **Re-Design de Jogos no Ambiente Unity a partir de uma Abordagem Orientada a Componentes**. Florianópolis, 2015. 103 p. Monografia

(Graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico. Orientador: Prof. Dr. Ricardo Pereira e Silva.

SILVA, Ricardo P. e, PRICE, R. T. Component interface pattern. In: **Proceedings of 9th Conference on Pattern Language of Programs 2002 (PLOP 2002)**. Monticello: sep. 2002.

SILVA, R. P. **Suporte ao desenvolvimento e uso de frameworks e componentes**. Tese de doutorado. Porto Alegre: UFRGS/II/PPGC, mar. 2000. 262p.

SILVA, Ricardo P. e. **Como modelar com UML 2**. Florianópolis, SC: Visual Books, 2009. 320p.

SILVA, R. P.; PRICE, R. T. **Suporte ao Desenvolvimento e Uso de Componentes Flexíveis**. In: Proceedings of XIII Simpósio Brasileiro de Engenharia de Software. [s.l: s.n.], 1999, p. 13-28.

SOMMERVILLE, I. **Engenharia de Software**. 9.ed. São Paulo:Pearson Prentice Hall, 2011.

SZYPERSKI, C. et al. Sumary of the first international workshop on component-oriented programming. In: **Proceedings of the International Workshop on Component-Oriented Programming (WCOP)**,1.[S.1.:s.n], 1996.

SZYPERSKI, C. **Component Software** - Beyond Object-Oriented Programming. New York, USA: Addison–Wesley, 2002.

TEINIKER, E.; MITTERDORFER, S.; JOHNSON, L.M.; KREINER, C.; KOV, Z.; WEISS, R. A Test-Driven Component Development Framework based on the CORBA Component Model. **Proceedings of the 27th Annual International Computer Software and Applications Conference**. Dallas, EUA: 2003.

TEIXEIRA, N. S. ; SILVA, R. P. . Compatibility Evaluation of Components Specified in UML. In: **XXX Conferencia Internacional de la Sociedad Chilena de Ciencia de la Computación - SCCC**, 2011, Curicó. Jornadas Chilenas de Computación (JCC), 2011.

TEIXEIRA, N. S. **Análise da compatibilidade de componentes especificados em UML**. Florianópolis, 2012. 166 p. Dissertação (Mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-Graduação em Ciência da Computação. Orientador: Prof. Dr. Ricardo Pereira e Silva.

ZHANG, Yuefeng. **Test-Driven Modeling for Model-Driven Development**. Ieee Software, [s.l.], v. 21, n. 05, p.80-86, set. 2004. Institute of Electrical and Electronics Engineers (IEEE). http://dx.doi.org/10.1109/ms.2004.1331307.

APÊNDICE A – Trabalho no formato de artigo SBC

Suporte a Testes Automatizados de Interface de Componentes Desenvolvidos no Ambiente SEA

Tiago J. Nascimento[1]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)
Florianópolis, Santa Catarina, Brasil.

t.j.nascimento@grad.ufsc.br

Abstract. *Components are Software artifacts that encapsulate functionalities behind well-defined interfaces, allowing complex systems to be constructed from the composition of independently developed parts. To make this strategy possible, and to be able to leverage the characteristics of cost reduction and higher quality of the system, it is necessay that the quality of the components being developed get individually ascertained.This work proposes an approach to the automated creation and execution of interface tests, at the modelling level, for components. This is done by the creation of mirror-components. It also discusses how these tests can be used as part of a broader methodology that encompasses automatically generated tests at the implementation level of abstraction.*

Resumo. *Componentes são artefatos de Software que encapsulam funcionalidades utilizando interfaces bem definidas, permitindo que sistemas complexos sejam desenvolvidos pela composição de elementos criados de forma independente. Para que se possa utilizar esta abordagem aproveitando suas vantagens de redução de custos e aumento da qualidade final do sistema, é necessário que se garanta a qualidade dos componentes desenvolvidos. Este trabalho propõe uma abordagem para a execução automática de testes de interface em nível de modelo para componentes desenvolvidos no ambiente de desenvolvimento SEA, baseada na criação de componentes-espelho, e discute como estes testes podem ser utilizados como parte de uma metodologia de testes automáticos em nível de código.*

1. Introdução

Um sistema orientado a componentes tem sua implementação separada em unidades independentes que interagem a partir de interfaces bem definidas [Szyperski 1996], permitindo o reúso de artefatos de software de terceiros e a quebra da complexidade do sistema. O reúso de componentes só é possível quando fornecedores e usuários de componentes possuem métodos confiáveis de se testar as funcionalidades descritas na interface [Gao 2003].

Para garantir a reusabilidade de componentes e sua interoperabilidade com os demais componentes do sistema, é necessário que o componente seja testado de forma que se garanta que o componente se comporta de acordo com a definição de sua interface

Teiniker (2003) propôs um framework para desenvolvimento orientado a testes de componentes que atendem à especificação CORBA. Tal proposta se constitui de uma ferramenta que parte de uma definição de interface de um componente para automaticamente

gerar um "componente-espelho" – um complemento perfeito da interface do componente original –, o qual executa testes a fim de, a todo momento, garantir que o componente original obedeça à especificação de sua interface. Desta forma, este framework se apropria de conceitos do Test-Driven Development (TDD) e assim se contrapõe a abordagens de teste post-mortem tradicionais (onde os testes são realizados apenas após o desenvolvimento do artefato de software), permitindo a combinação entre as vantagens do desenvolvimento orientado a componente com aquelas do TDD.

Para testar se um componente sendo desenvolvido obedece à especificação de sua Interface, pode-se utilizar a Análise de Compatibilidade [Teixeira 2012] entre este componente e o seu componente-espelho automaticamente gerado, através de suas interfaces.

Um componente-espelho é um componente cuja interface age como um complemento perfeito àquela do componente a partir do qual ele foi criado. Um componente-espelho oferece todos os métodos requeridos pelo original, e depende de todos os métodos fornecidos pelo original. O espelho também obedece às regras de ordem de invocação de métodos impostas pela interface do original.

A Análise de Compatibilidade entre dois componentes avalia aspectos estruturais e comportamentais da interação entre eles a fim de se detectar erros de implementação e/ou especificação. Esta Análise é possível a partir do ambiente de desenvolvimento SEA, uma ferramenta gráfica de modelagem de software que permite a especificação UML de componentes e frameworks, e possibilita a criação de ferramentas customizadas e a geração automática de código [Silva 2000].

O objetivo deste trabalho é implementar solução inspirada no trabalho de Teiniker (2003) no ambiente SEA, para permitir o teste em nível de modelo da interface de um componente sendo desenvolvido, no que diz respeito aos aspectos estruturais, comportamentais e funcionais [Silva 2009].

A solução proposta neste trabalho se limita a componentes sendo desenvolvidos a partir do padrão de interfaces definido em Silva (2002), o que não necessariamente significa suporte a componentes implementados utilizando outros padrões de interface de componente. O processo de desenvolvimento utilizado se baseia na metodologia de desenvolvimento de componentes orientada à modelagem UML de Silva (2009).

2. Desenvolvimento Orientado a Componentes

Este paradigma de desenvolvimento oferece uma forma de se contornar o problema da crescente complexidade de softwares orientados a objetos. Inserindo uma camada de abstração superior à das classes, estas podem ser divididas em partes independentes do sistema, e assim gerenciadas mais facilmente pelos desenvolvedores, possibilitando o desenvolvimento e a manutenção em paralelo e/ou por entidades diferentes. Além da quebra da complexidade, outra vantagem da separação do desenvolvimento em componentes independentes é a de facilitar o reúso de software, que proporciona [Sommerville 2011]: maior confiabilidade, menor risco, uso eficiente de especialista e rapidez de desenvolvimento.

Um componente de software é uma unidade de composição com interfaces contratualmente especificadas e apenas dependências de contexto explícitas. Um componente de software pode ser implantado de forma independente e está sujeito a composição por terceiros.

Sommerville (2011) combina as definições de diferentes autores e pontua as principais características associadas a um componente de software:

1. *Padronizado*: obedecem padrões de interfaces, documentação, composição e implantação;

2. *Independente*: passível de ser utilizado para composição sem que necessite de outro

componente ou artefato de software específico. Qualquer dependência é estabelecida através da interface;

3. *Passível de Composição*: oferece acesso às informações internas e possui bem definidas suas interações externas através da interface;

4. *Implantável*: possui autonomia para que, por si, possa ser combinado com outros componentes no desenvolvimento de um sistema;

5. *Documentado*: toda informação necessária, para que potenciais usuários façam a decisão de utilizar o componente, deve estar explicitamente documentada. A interface de componentes deve estar clara.

Componentes podem ser considerados como parte da evolução da programação orientada a objetos [Silva 2009]: quando o desenvolvimento e a manutenção de um sistema tornou-se muito complexo pela grande quantidade de classes, necessitou-se separá-las em partes coesas contendo apenas uma fração delas, cada parte podendo ser utilizada sem que se tivesse conhecimento da implementação de cada classe.

2.1 Interface de Componente

A maioria das definições e características associadas a componentes acima pressupõem a possibilidade de se descrevê-los externamente, e usá-los sem que se tenha acesso a detalhes de suas implementações. A esta parte externamente visível se dá o nome de Interface de Componente. Ela age como o meio pelo qual componentes se conectam, especificando os detalhes e a semântica das operações que podem e/ou devem ser invocadas entre eles. Outra forma de caracterizar uma especificação de Interface é como um contrato entre cliente e implementador de um conjunto de interfaces, dizendo o que o cliente precisa fazer para poder usá-las, ao mesmo tempo que explicitando garantias a respeito do resultado destas operações [Szyperski 2002]. É importante aqui diferenciar [Silva 2009] [Szyperski 2002]:

- *Interface de Componente*: uma coleção de pontos de acesso a serviços, cada um com sua semântica estabelecida;
- *Interface UML*: uma relação de assinaturas de métodos a serem implementados por dado elemento do sistema (Classe, Componente) via relação de realização, e utilizados por um ou mais outros elementos via relação de dependência;
- *Porto*: fronteira que separa o componente (sua estrutura interna) e seu meio externo. Pode estar associado a uma ou mais interfaces UML.

2.2 Compatibilidade entre Componentes

A compatibilidade estrutural implica que o conjunto de métodos requeridos através da interface de um componente esteja disponível na interface do outro. A compatibilidade no nível comportamental está relacionada à equivalência entre as restrições associadas à ordem de execução de métodos estabelecidas em um componente e em outro. Por fim, a compatibilidade funcional depende de que as funcionalidades requeridas por um componente sejam supridas pelo outro. Dois componentes são estruturalmente compatíveis se o conjunto de métodos requeridos através da interface de um está disponível na interface do outro. Eles são compatíveis no nível comportamental quando as restrições associadas à ordem de execução de métodos, fornecidos ou requeridos, estabelecidas em um são respeitadas pelo outro. E são funcionalmente compatíveis quando as funcionalidades requeridas por um são supridas pelo outro. [Silva 2009] [Teixeira 2012].

A análise de compatibilidade entre componentes se dá por intermédio da especificação da interface de componente (IC). É nesse sentido que se mostra de fundamental relevância o

estabelecimento de mecanismos e de critérios de descrição de componentes capazes de qualificá-los adequada e minuciosamente nos três níveis – estrutural, comportamental e funcionalmente –, na medida em que é por meio dessa descrição que se efetua a análise. Especificamente, o trabalho de Teixeira tem como objetivo a identificação de eventuais incompatibilidades nos níveis estrutural e comportamental; ela opta por não contemplar o nível funcional e a compatibilidade dos componentes interligados.

Para a realização da Análise de Compatibilidade Estrutural e Comportamental entre dois componentes, Teixeira (2012) propõe duas ferramentas:

- *Ferramenta de Análise Estrutural (FAE)*: a análise da compatibilidade estrutural é realizada para cada par de portos conectados dos diferentes componentes da aplicação. Métodos requeridos no porto em um lado da conexão devem ser fornecidos pelo porto do outro lado da conexão [Teixeira 2012]. Nesse sentido, a incompatibilidade estrutural "pode ocorrer na conexão entre portos de componentes quando o serviço requerido por um componente não tem o respectivo serviço fornecido pelo outro";
- *Ferramenta de Análise Comportamental (FAC)*: A partir da especificação comportamental da aplicação, que é criada a partir da interligação das máquinas de estado individuais dos dois componentes a serem analisados, a Análise Comportamental verifica as restrições de ordem de invocação de métodos requeridos e fornecidos para cada componente, e nos diz se é compatível com as restrições dos outros componentes conectados. Esse tipo de avaliação envolve todo o conjunto de componentes interligados – diferente da avaliação de compatibilidade estrutural, que trata um par de portos de cada vez [Silva 2009].

3. Testes de Componentes

Os testes no Desenvolvimento Orientado a Componentes podem ser separados de acordo com as partes distintas envolvidas no processo [Gao 2003]: os testes de componentes por parte do fornecedor do componente – que validam o componente de acordo com suas especificações – e os testes por parte do usuário – que acontecem no contexto de desenvolvimento de um software baseado em componentes para garantir que os componentes atendem às funcionalidades, interfaces e quesitos de performance prometidos. Enquanto o fornecedor de um componente se preocupa em garantir que a sua implementação obedece a especificação definida, e que atendem aos padrões de modelo de componentes, o usuário de um componente precisa ter a certeza de que ele foi escolhido corretamente para a tarefa, que ele é compatível com o sistema e que está sendo reutilizado de forma correta.

Embora trate apenas do nível de modelo, sem a execução dos testes em relação à implementação do componente, a estratégia proposta neste trabalho se relaciona com o lado do fornecedor de componentes, principalmente no que diz respeito aos testes caixa-preta, que podem ser definidos como [IEEE 2012]: “Testes que ignoram mecanismos internos de um sistema ou componente e focam inteiramente nos outputs gerados em resposta a inputs e condições de execução selecionados”;

Para Sommerville (2011), do ponto de vista do usuário de um componente, os testes devem ser direcionados a garantir que o componente se comporta de acordo com a definição de sua interface. Entre os possíveis erros a serem detectados por testes realizados a partir da interface de componentes, o autor cita o erro de mau uso de interface – quando um componente utiliza a interface de outro componente incorretamente, o mau entendimento da interface – quando suposições incorretas são feitas a respeito da interface de um componente e por último, os erros de *timing* – quando a interação entre componentes não obedece uma sincronização de tempo exigida pela especificação deles.

4. Ambiente SEA de Desenvolvimento

No trabalho de Silva (2000), foi concebido o OCEAN, um framework orientado a objetos que fornece suporte à construção de ambientes de desenvolvimento de software. Ambientes construídos através do OCEAN permitem o manuseio de documentos compostos por um conjunto de modelos, que por sua vez agregam conceitos – as unidades de modelagem do domínio tratado. Uma especificação criada por via de tal ambiente é representada pelo conjunto de modelos e conceitos, e pelos relacionamentos existentes entre eles.

Documentos manipulados por ambientes derivados do OCEAN obedecem ao padrão *Model-View-Controller* (MVC), de forma que as representações de conceitos são separadas de suas representações visuais no modelo.

O framework OCEAN prevê ainda mecanismos para criação e modificação de especificações. Ele permite a implementação de funcionalidades disponibilizadas para atuar sobre a especificação sendo manipulada. Para tal, um mecanismo digno de menção é o da criação de ferramentas, que permitem ações de edição, análise ou transformação sobre conceitos, modelos ou especificações. Para implementar tal ferramenta, o framework oferece uma definição genérica na forma de uma classe abstrata chamada OceanTool, que deve ser especializada para cada ferramenta a ser desenvolvida.

A partir do framework OCEAN, foi criado o ambiente SEA [Silva 2000], com o objetivo de permitir o desenvolvimento e uso de frameworks e componentes. A especificação de projetos no SEA por parte do usuário é feita pela criação de Modelos na linguagem UML, a partir da manipulação de editores gráficos associados a cada tipo de Modelo. As técnicas de modelagem disponíveis no ambiente SEA correspondem a um subconjunto daquelas definidas pela linguagem UML, com adições e padronizações necessárias para a especificação de frameworks e componentes.

O SEA dispõe ainda de mecanismos de verificação de consistência de especificações, e para a manipulação programática destas especificações e de seus conceitos associados. Obedecendo à estrutura MVC imposta pelo framework OCEAN, documentos criados pelo SEA são manipulados em seu nível conceitual (*Model*), e suas modificações são refletidas automaticamente no nível de apresentação (*View*).

5. Testes Automatizados de Interface de Componentes

Esta seção descreve uma proposta de suporte a testes automatizados de interface de componente, no desenvolvimento de um componente através do ambiente SEA. Para tal, primeiro (seção 5.1) aborda os passos para a especificação necessária (estrutural, comportamental e funcional) da descrição externa do componente. Depois (seção 5.2), descreve o funcionamento e o resultado de uma ferramenta que automaticamente cria uma definição de interface de componente-espelho a partir deste componente sendo desenvolvido. A seção 5.3 aplica a análise de compatibilidade (conceituada anteriormente na seção 2.2) entre a interface do componente-espelho e a interface do componente sendo desenvolvido, através da ferramenta implementada por Teixeira (2012) e uma ferramenta simples de análise funcional e, por fim, a seção 5.4 discute quais os aspectos envolvidos na geração automática de casos de testes que validem a implementação do componente em relação à sua interface, em nível de código.

As ferramentas descritas nesta seção foram implementadas dentro do ambiente SEA a partir de extensões da classe abstrata OceanTool (figura 1), herdada do framework OCEAN (seção 4). As subclasses *MirrorComponentGeneratorTool* e *MirrorComponentAnalyzerTool* foram criadas no escopo deste trabalho (seções 5.2 e 5.3), e as subclasses *ComponentStructuralAnalyzerTool* e *ComponentBehaviorAnalyserTool* foram implementadas

no trabalho de Teixeira (2012), e foram adaptadas e reutilizadas neste trabalho para a execução dos testes de interface de componentes (seção 5.3).

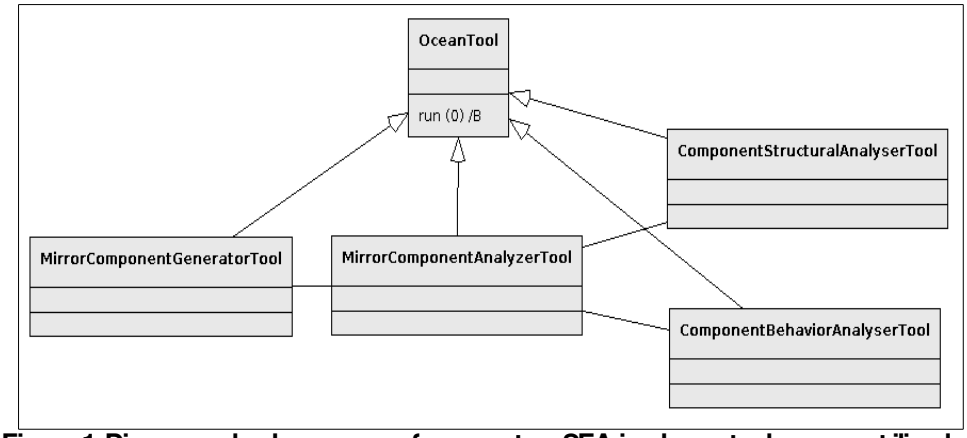

Figura 1. Diagrama de classes com ferramentas SEA implementadas ou reutilizadas

5.1 Especificação de um componente através do ambiente SEA

O primeiro passo é especificar sua descrição externa, ou seja, sua Interface de Componente. Isto será feito utilizando o ambiente de desenvolvimento SEA. Os passos aqui descritos são pré-requisitos para a execução da ferramenta descrita na seção 5.2.

O primeiro passo é o da modelagem estrutural, com a elaboração dos diagramas de classe e de componentes, que possuem o objetivo de especificar quais as Interfaces UML relacionadas ao componente, e sob quais portos acontecem estes relacionamentos. As Figuras 2 e 3 mostram tais diagramas, respectivamente.

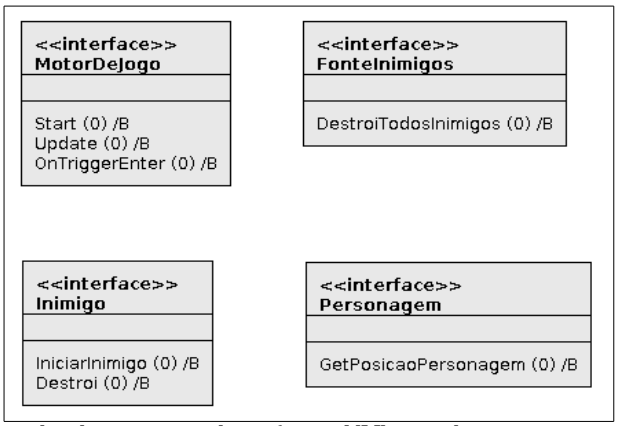

Figura 2. Diagrama de classes com interfaces UML que interagem com o componente

A seguir (Figura 4), a descrição comportamental do componente deve ser feita através do diagrama de máquina de estados, representando as restrições de invocações dos métodos relacionados ao componente. Vemos na descrição comportamental que a maior parte do funcionamento do componente FonteInimigos é determinado por chamadas a métodos implementados da interface MotorDeJogo, chamados pelo framework do sistema. Durante a execução do jogo, ele fica no estado "Esperando" aguardando por chamadas do método *update*() que levam à geração de inimigos para combater o jogador. Observa-se que a ausência de um pseudo-estado final indica que a execução do componente pode ser finalizada a partir de qualquer dos estados descritos.

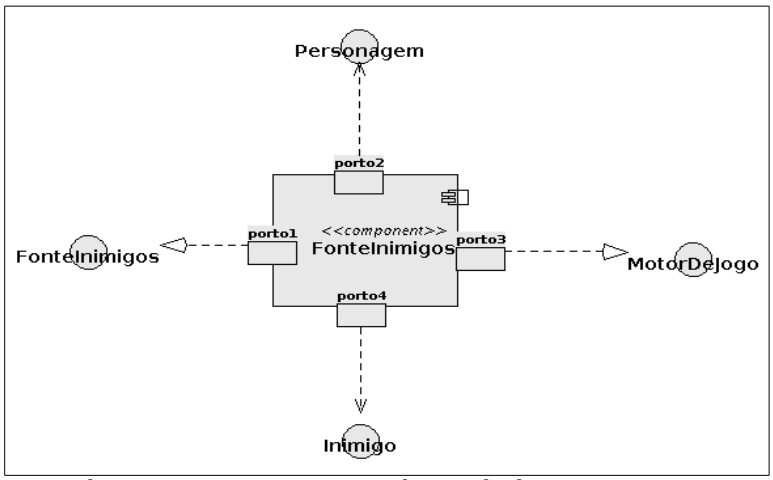

Figura 3. Diagrama de componentes mostrando o relacionamento entre o componente e interfaces UML

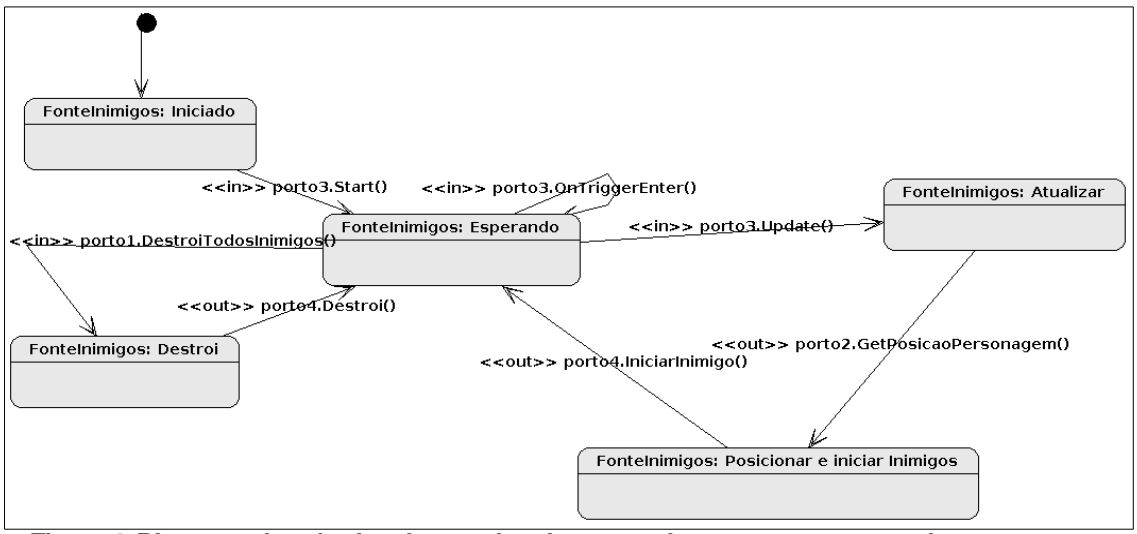

Figura 4. Diagrama de máquina de estados descrevendo o comportamento do componente

Por último, é necessário modelar as funcionalidades do componente, através de um diagrama de casos de uso indicando quais funcionalidades – do ponto de vista externo ao componente – ele oferece. Cada um dos casos de uso deve ser detalhado através de um diagrama de atividades, explicando o que faz e não como o faz.

5.2 Geração da interface do Componente-Espelho

Com as modelagens estrutural, comportamental e funcional disponíveis, o componente pode ter a interface do seu componente-espelho gerado a partir do SEA. Isto é feito através do item "Gerador de Componente-Espelho" do menu "Ferramentas" do SEA. Após se selecionar qual componente será espelhado, a ferramenta inicia sua execução. Esta funcionalidade aqui descrita foi implementada dentro do ambiente SEA na forma da classe *MirrorComponentGeneratorTool* (figura 1).

Uma vez executada, a ferramenta realiza os seguintes passos, que serão detalhados ao longo desta seção: 1) checar consistência da especificação; 2) obter elementos relacionados ao componente selecionado; 3) clonar interfaces UML relacionadas ao componente selecionado; 4) criar diagrama de componentes separado; 5) criar componente-espelho no novo diagrama de componentes; 6) criar diagrama de implantação para estabelecer relacionamento entre

componentes; 7) espelhar diagrama de máquina de estado; 8) clonar diagramas de atividades.

Primeiro (passo 1), a ferramenta verifica que todos os diagramas necessários – como exemplificados na seção 5.1 – estão presentes e corretamente especificados. Além disso, é necessário que o componente selecionado para ser espelhado tenha um mínimo de informações detalhadas: deve ter no mínimo 1 porto com 1 interface UML associada, e esta interface deve ter no mínimo 1 método declarado.

O passo 2 percorre as estruturas de dados do SEA que guardam os elementos da especificação, ou seja, os conceitos representados nos diagramas (componentes, portos, relacionamentos, etc), para obter referências a todos aqueles elementos que se relacionam com o componente selecionado e que serão utilizados posteriormente pela ferramenta.

Todas as interfaces relacionadas ao componente selecionado são "clonadas" no passo 3. Elas passam a servir como um *snapshot*, ou uma "foto" do estado delas no momento em que a ferramenta foi executada, para que possíveis modificações na interface (eg. nome de métodos, tipo e número de parâmetros, etc) possam ser comparadas com a descrição externa do componente a fim de se verificar inconsistências inseridas posteriormente. Um padrão de nomeação para interfaces clonadas foi adotado: concatenando o sufixo "_clone" no nome original da interface. Esta etapa também adiciona estes clones de interface no diagrama de classes existente (figura 5).

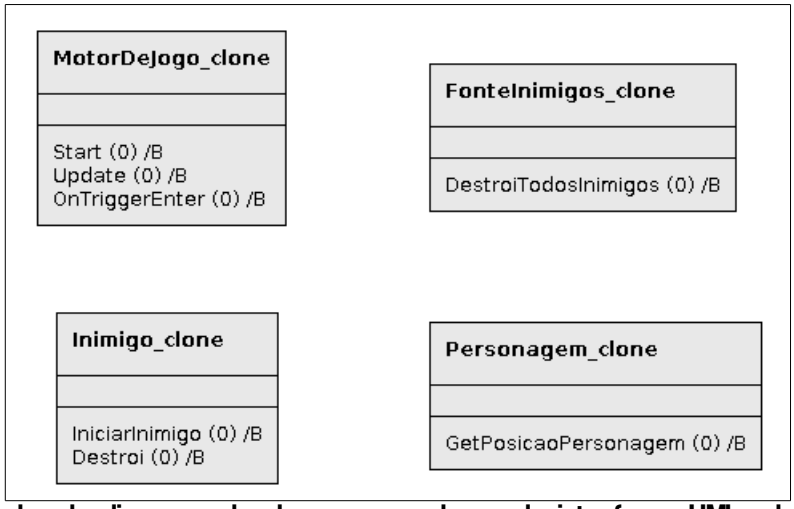

Figura 5. Trecho de diagrama de classes com clones de interfaces UML relacionadas ao componente

No passo 4, o componente selecionado é copiado para um novo diagrama de componentes. Isto serve para separar o componente a ser desenvolvido de quaisquer outros componentes do sistema, e assim facilitar o uso da ferramenta em especificações grandes e/ou complexas. Além do componente selecionado, o novo diagrama de componentes conterá também o componente-espelho (passo 5). Este novo componente "espelha" o relacionamento do original com suas interfaces, ou seja, um relacionamento de dependência do original para uma interface resultará na criação de um relacionamento de realização do espelho para o clone daquela interface. A figura 6 mostra o trecho do novo diagrama de componentes que inclui o componente-espelho, e que pode ser comparado com a figura 3 (seção 5.1) da especificação original do componente *FonteInimigos*: enquanto o original implementa as interfaces *MotorDeJogo* e *FonteInimigos*, e depende das interfaces Personagem e Inimigo; seu espelho implementa as interfaces Personagem_clone e Inimigo_clone, e depende das interfaces *MotorDeJogo_clone* e *FonteInimigos_clone*. Isto significa que o novo componente serve como um complemento perfeito para o original, e vice-versa.

Outro padrão de nomeação foi adotado para todos os elementos "espelhados" do componente original (o próprio componente, seus portos, etc.): através da concatenação de "_mirror" ao nome do elemento original correspondente.

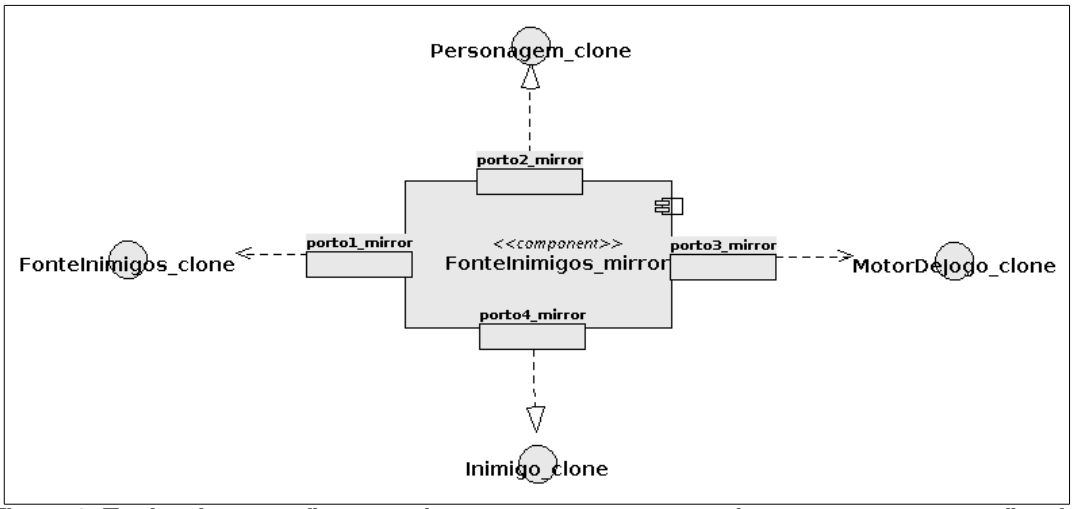

Figura 6. Trecho do novo diagrama de componentes mostrando o componente-espelho do componente

No passo 6 um diagrama de implantação é criado, explicitando a associação do componente original com seu espelho, e permitindo que a Ferramenta de Análise de Compatibilidade seja executada entre os dois, como será abordado na seção 5.3 abaixo. A Figura 7 mostra o novo diagrama de implantação gerado.

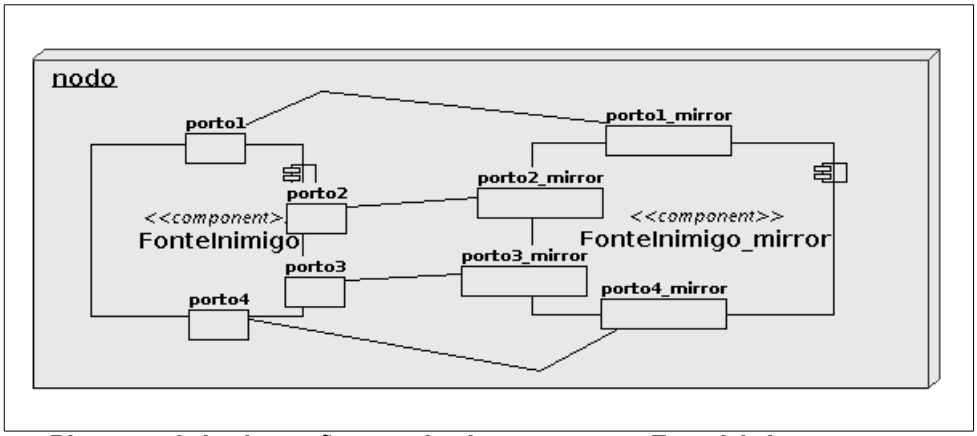

Figura 7. Diagrama de implantação associando componente FonteInimigos com seu espelho

Até este ponto, a ferramenta se preocupou em espelhar elementos da descrição estrutural (ver seção 2) do componente selecionado. No passo 7 a descrição comportamental na forma de diagrama de máquina de estados é espelhada. Isto significa que o comportamento do componente-espelho será criado de forma que sempre obedeça às restrições de ordem de invocação de métodos do original. Esta nova descrição comportamental do componente-espelho é criada a partir das seguintes conversões aplicadas à original:

- Para cada estado do diagrama original, um novo estado é criado utilizando o padrão de nomeação: nome original concatenado com "_mirror";
- As mesmas transições entre estados do original são criadas entre os estados espelhados correspondentes;
- O sentido de invocação dos gatilhos das transições é invertido, ou seja, "in" da

transição original se transforma em “out” na transição espelhada correspondente, e vice-versa;
- O porto de cada transição original é substituído pelo porto espelhado correspondente, seguindo mesmo padrão de nomeação (“_mirror”);
- As assinaturas dos métodos invocados em cada transição são mantidos iguais.

A partir da especificação comportamental do componente FonteInimigos (figura 4), a ferramenta criará o diagrama de máquina de estados da figura 8, descrevendo o comportamento do componente-espelho FonteInimigos_mirror. Este comportamento também complementa perfeitamente o original, ou seja, enquanto o componente FonteInimigos aguarda no estado “Esperando” pela chamada de update() através do porto porto3 para avançar na criação de inimigos do jogo, o componente FonteInimigos_mirror avança do estado “Esperando_mirror” fazendo tal chamada de método a partir do porto porto3_mirror, e assim sucessivamente para os demais estados. Isto significa que enquanto o componente original não tiver sua Interface de Componentes alterada, o seu componente-espelho sempre obedecerá às restrições de invocação do original.

Por fim, o último passo (passo 8) clona os diagramas de Atividades que descrevem as funcionalidades do componente sendo especificado.

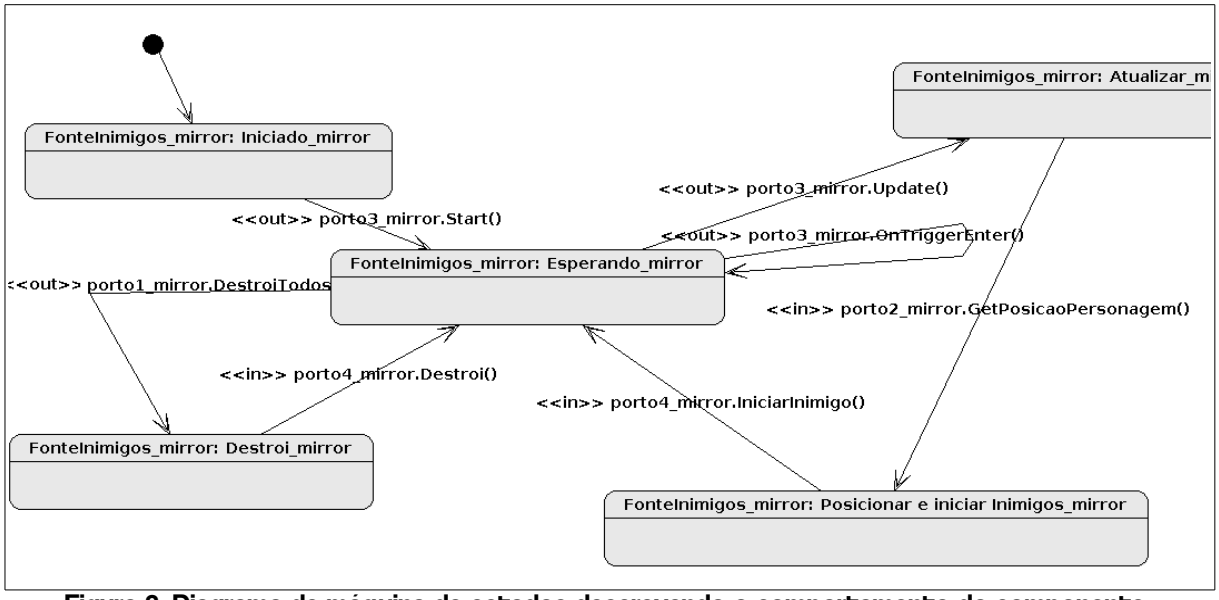

Figura 8. Diagrama de máquina de estados descrevendo o comportamento do componente-espelho FonteInimigos_mirror

5.3 Análise de Compatibilidade

Após criada a interface do componente-espelho, o componente original pode ter sua especificação modificada como parte do processo de desenvolvimento ou manutenção. Para avaliar se, após estas modificações, o componente original continua obedecendo à especificação de sua interface, ou seja, se ele continua compatível com seu componente-espelho, uma segunda ferramenta implementada no SEA, na forma da classe *MirrorComponentAnalyzerTool* (figura 1), permite a execução combinada das Ferramentas de Análise Estrutural e Comportamental [Teixeira 2012] descritas na seção 2.2, além da execução de uma análise de compatibilidade funcional simples implementada neste trabalho.

Mais especificamente, a avaliação é composta de três etapas:

1. *Análise de Compatibilidade Estrutural*: executa e compila o resultado da Ferramenta de Análise Estrutural descrita na seção 2. Como visto anteriormente, esta análise detecta erros como ausência de método requerido ou fornecido e diferenças entre tipo de retorno, número e tipo de parâmetros destes métodos;
2. *Análise de Compatibilidade Comportamental*: executa e compila o resultado da Ferramenta de Análise Comportamental descrita na seção 2. Como visto anteriormente, esta análise detecta erros como os de ordem de invocação de métodos fornecidos ou requeridos, de métodos nunca executados ou indisponíveis;
3. *Análise de Compatibilidade Funcional*: faz uma comparação simples entre os diagramas de atividade do componente original com o do componente-espelho, servindo como uma análise de compatibilidade primitiva. Esta análise serve para detectar se existe qualquer diferença entre estes diagramas.

Executada a ferramenta, ela relaciona os problemas de incompatibilidade estrutural, comportamental e funcional (neste aspecto de forma limitada), garantindo que nestes quesitos o componente especificado esteja consistente com o que a sua definição de interface espera. Os resultados são exibidos em relatórios textuais como mostra a figura 9.

Os testes automatizados de interface de componentes implementados permitem que o desenvolvimento de componentes no ambiente SEA itere entre modelagem e análise de compatibilidade, e que a todo momento se garanta a compatibilidade do componente sendo desenvolvido para com sua interface inicial, ou seja, que seja sempre compatível com seu componente-espelho.

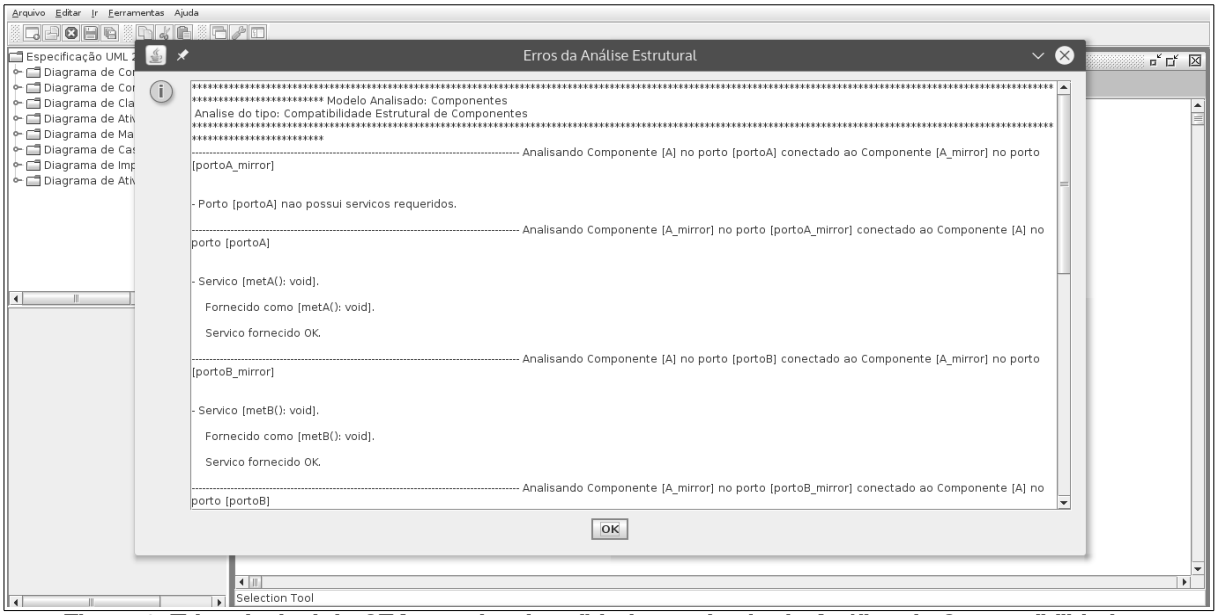

Figura 9. Tela principal do SEA com janela exibindo resultado de Análise de Compatibilidade

5.4 Geração automática de código de testes (discussão)

A ferramenta apresentada propôs uma forma de se executar testes de interface de componente em nível de modelo através do ambiente SEA. Para que ela seja útil no desenvolvimento completo ou na manutenção de um componente, prevê-se a necessidade da geração de testes em nível de código, para que seja validada a implementação do componente de software em relação à sua interface de componente, em busca de modificações em sua operação que tenham sido feitas sem a atualização correspondente em sua interface. Embora este próximo

passo fuja do escopo deste trabalho, esta seção discute alguns aspectos relacionados a ele.

Esta ferramenta imaginada funcionaria como uma função que receberia como entrada o resultado da especificação (em modelo) de interface do componente-espelho obtida da seção 5.2, e como saída geraria, automaticamente ou semi-automaticamente, um artefato de software com o conjunto de testes necessário para que se garanta a compatibilidade estrutural, comportamental e funcional entre a implementação de componente e sua interface.

Este conjunto de testes gerados poderia se situar:

- Externamente à implementação do componente-espelho, a partir de um Cliente de Testes (figura 10) responsável por instanciar tanto o componente original quanto o espelho, e coordenar a execução dos casos de testes. Esta abordagem é adequada para testes em tempo de desenvolvimento, pelo fato de exigir serviços do Cliente de Testes que não necessariamente são empacotados e distribuídos junto com o componente original;

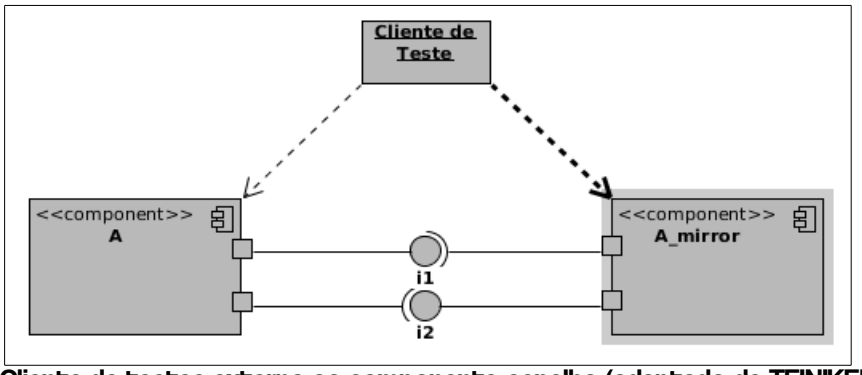

Figura 10. Cliente de testes externo ao componente-espelho (adaptado de TEINIKER, 2003)

- Encapsulado na implementação do componente-espelho (figura 11), que pode inclusive ser empacotado e distribuído junto com o componente original, para que este aja como um componente executável auto-testável. Esta abordagem permite que os testes sejam realizados em tempo de execução por parte do cliente do componente desenvolvido, que age como uma metáfora sobre componentes de hardware auto-testáveis (GROSS, 2005).

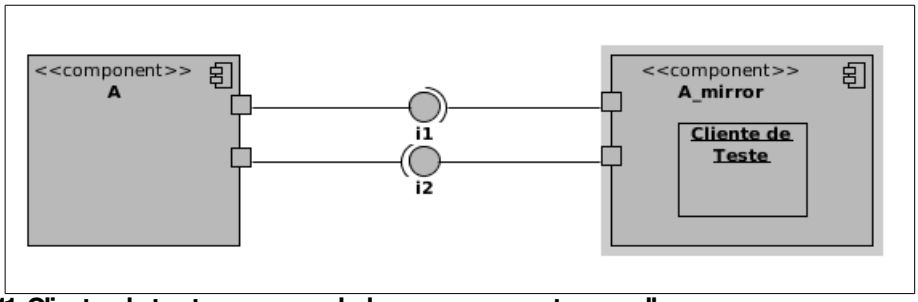

Figura 11. Cliente de testes encapsulado no componente-espelho

- Intermediando todas as mensagens entre o componente e seu espelho (figura 12), de forma similar à primeira alternativa apresentada (figura 10), com a diferença que neste caso o componente não se relaciona diretamente com o componente-espelho.

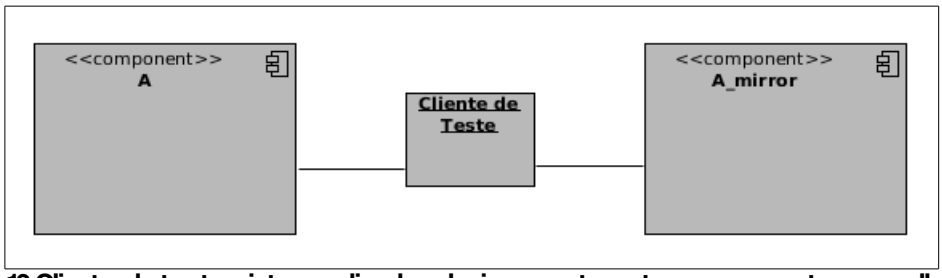

Figura 12.Cliente de testes intermediando relacionamento entre componente e espelho

Como visto na seção 3, os desafios de se testar componentes (e de modo geral, sistemas orientados a componente) se originam principalmente da ausência de acesso à implementação interna dos componentes. A especificação da interface do componente original e a do componente-espelho devem ser informação necessária para a geração semi-automática dos testes. Isto significa que, embora a compatibilidade estrutural entre o componente e sua interface sejam mais facilmente validados pelos mecanismos sintáticos da linguagem de programação, a compatibilidade comportamental e funcional podem não ser completamente automatizáveis, por lidarem com aspectos semânticos do componente, que podem ir além da especificação da interface de componente.

A representação das restrições comportamentais de um componente neste trabalho foi feita através do diagrama de máquina de estados UML (seção 5), que estabelece as restrições de ordem de invocação de métodos entre o componente e seu espelho. Neste aspecto, a geração de testes deve se preocupar em validar em código a implementação da máquina de estados (ou seja, o comportamento do componente sendo testado), garantindo que:

- Ao se chamar seus métodos fornecidos, o componente transita entre os estados corretos, como especificado na interface;
- Para executar usas operações, o componente realiza chamadas aos métodos dos quais depende, na ordem especificada na interface;
- As chamadas a métodos requeridos ou fornecidos do componente obedecem às restrições de parâmetros (tipo, número, etc) especificadas na interface;

Testar uma implementação de máquina de estados requer que possamos fazer ao menos duas operações, de alguma forma fornecidas pelo componente a ser testado [Gross 2005]:

- Operação de checagem de estado, para que se identifique o estado da máquina a qualquer momento;
- Operação de estabelecimento de estado, para que se possa identificar o estado final desejado após uma determinada transição.

Estes estados a serem testados representam estados lógicos, e não são necessariamente estados internos do componente, mas uma representação de acordo com a especificação de sua interface, ou seja, aquilo que o componente permite ao ambiente externo verificar através de sua descrição. Por isto estes testes ainda podem ser considerados de caixa-preta (seção 3).

Um caso de testes de uma máquina de estados relaciona operações a serem testadas com um estado inicial a ser definido como pré-condição, parâmetros de entrada para a operação testada e um estado final como pós-condição desejada, a ser comparado com o estado verificado após a simulação da transição por parte do teste. Estes conceitos são ilustrados na estrutura da figura 13.

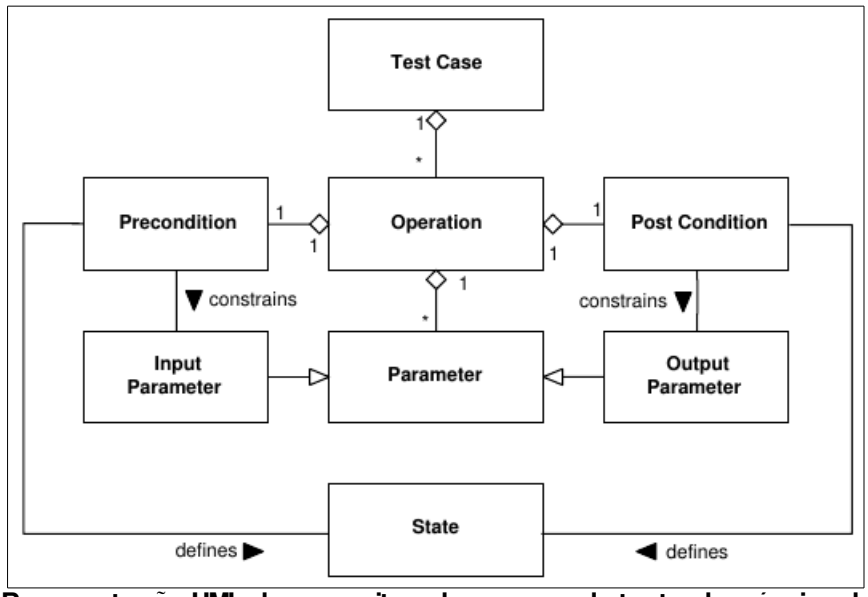

Figura 13.Representação UML dos conceitos de um caso de teste de máquina de estados [GROSS 2005]

Percorrer e testar os estados lógicos da implementação do componente testado traz os seguintes desafios associados:

- *Definição do início de execução*: é necessário que se possa partir de qualquer dos estados existentes na máquina de estados, não só do componente sendo testado, como do componente-espelho relacionado a ele. Isto acontece porque como as restrições comportamentais descrevem a ordem de invocação de métodos tanto requeridos como fornecidos pelo componente original, é necessário se prever casos de uso que iniciam a partir de chamadas feitas ao componente original, vindas de fora;
- *Definição de abrangência desejada*: todos os estados da máquina de estados devem ser testados para garantir a qualidade da implementação do componente. Além disso, os métodos envolvidos nas transições entre um estado e outro podem receber valores de parâmetros variáveis que devem ser testados. Isto significa que é necessário que se possa determinar um critério mínimo a ser atendido antes que se considere que um número satisfatório de transições foram testados;
- *Controle de parada*: métodos podem estar inseridos em laços de repetição, ou o componente deve atender a requisitos de tempo máximo de resposta de métodos requeridos e fornecidos. É necessário que se possa controlar a parada da execução do teste a todo momento;
- *Parametrização de testes*: mesmo que os testes sejam gerados automaticamente, deve ser previsto um mecanismo de parametrização dos próprios testes para se controlar não só os parâmetros dos métodos responsáveis pelas transições, como outras variáveis de ambiente;
- *Obtenção de feedback*: todas as informações a respeito dos conceitos relacionados ao caso de teste, como ilustrados na figura 32, devem ser disponibilizados ao desenvolvedor. Isto significa, por exemplo, que podem ser necessárias estruturas de dados que registrem informações internas da implementação do componente-espelho, representando quais métodos requeridos pelo componente testado foram invocados, em qual ordem, etc.

6. Conclusão

No presente contexto de crescente complexidade de sistemas de software, vislumbra-se a adoção do Desenvolvimento Orientado a Componentes como possível "próximo passo" de abstração acima do paradigma orientado a objetos.

A garantia de qualidade dos componentes é primordial para a reusabilidade de componentes, e argumenta-se que uma ferramenta de teste automatizado de interface de componentes contribui para a garantia de qualidade.

Como resultado deste trabalho foram criadas duas ferramentas executáveis no ambiente de desenvolvimento SEA: uma que gera automaticamente uma especificação de interface de componente espelho que complementa àquela de um componente com interface pré especificada, e uma segunda ferramenta que analisa estas duas interfaces de componente para detectar eventuais modificações que possam tornar a especificação do componente inconsistente com sua interface.

Não se teve a pretensão neste trabalho de construir uma ferramenta ou metodologia completa de desenvolvimento, mas espera-se que a descrição do artefato desenvolvido sirva de exemplo de uma combinação destes conceitos estudados.

Como trabalhos futuros, contempla-se a possibilidade de se utilizar tal abordagem como parte de ferramentas de desenvolvimento orientado a modelos, através de testes baseados em modelo, até os baixos níveis de abstração através da geração automática de código, incluindo de testes unitários.

Referências

Amorim Junior, J. (2006) "Integração dos Frameworks JHotDraw e Ocean para a produção de objetos visuais a partir do Framework Ocean". Florianópolis.. 116 p. Monografia (Graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico. Orientador: Prof. Dr. Ricardo Pereira e Silva.

Beck, K. (2002) "Test-Driven Development By Example". Boston: Addison-Wesley.

Coelho, A. (2007) "Reengenharia do Framework OCEAN". Dissertação (Mestrado) - Programa de Pós-graduação em Ciência da Computação. Universidade Federal de Santa Catarina. Florianópolis: 2007. Orientador: Prof. Dr. Ricardo Pereira e Silva.

Cunha, R. S. (2005) "Suporte à Análise de Compatibilidade Comportamental e Estrutural entre Componentes no Ambiente SEA". Dissertação (Mestrado) – Programa de Pós-graduação em Ciência da Computação. Universidade Federal de Santa Catarina. Florianópolis. Orientador: Prof. Dr. Ricardo Pereira e Silva.

Gao, J.; Wu, Y.; Tsao, H.S.J. (2003) "Testing and Quality Assurance for Component-Based Software". Boston: Artech House.

Gross, H.G. (2005) "Component-Based Software Testing with UML". Berlim: Springer.

IEEE. (2012) "Systems and software engineering-vocabulary".

Rodrigues, Y. W. (2015) "Re-Design de Jogos no Ambiente Unity a partir de uma Abordagem Orientada a Componentes". Florianópolis. 103 p. Monografia (Graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico. Orientador: Prof. Dr. Ricardo Pereira e Silva.

Silva, Ricardo P. e, Price, R. T. (2002) "Component interface pattern." In: Proceedings of 9th Conference on Pattern Language of Programs 2002 (PLOP 2002). Monticello.

Silva, R. P. (2000) "Suporte ao desenvolvimento e uso de frameworks e componentes". Tese de doutorado. Porto Alegre: UFRGS/II/PPGC, mar. 2000. 262p.

Silva, R.P. (2009)."Como modelar com UML 2". Florianópolis, SC: Visual Books, 2009. 320p.

Silva, R.P.; Price, R. T. (1999). “Suporte ao Desenvolvimento e Uso de Componentes Flexíveis”. In: Proceedings of XIII Simpósio Brasileiro de Engenharia de Software. [s.l: s.n.]p. 13-28.

Sommerville, I. (2011) “Engenharia de Software”. 9.ed. São Paulo:Pearson Prentice Hall.

Szyperski, C. et al. (1996) “Sumary of the first international workshop on component-oriented programming”. In: Proceedings of the International Workshop on Component-Oriented Programming (WCOP),1.[S.1.:s.n].

Szyperski, C (2002). “Component Software - Beyond Object-Oriented Programming”. New York, USA: Addison–Wesley.

Teiniker, E.; (2003) “Test-Driven Component Development Framework based on the CORBA Component Model”. Proceedings of the 27th Annual International Computer Software and Applications Conference. Dallas, EUA.

Teixeira, N. S. ; Silva, R. P. (2011) “Compatibility Evaluation of Components Specified in UML”. In: XXX Conferencia Internacional de la Sociedad Chilena de Ciencia de la Computación - SCCC, Curicó. Jornadas Chilenas de Computación (JCC).

Teixeira, N. S. (2012) “Análise da compatibilidade de componentes especificados em UML”. Dissertação (Mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-Graduação em Ciência da Computação. Orientador: Prof. Dr. Ricardo Pereira e Silva.

Zhang, Y. (2004) “Test-Driven Modeling for Model-Driven Development”. Ieee Software, [s.l.], v. 21, n. 05, p.80-86. Institute of Electrical and Electronics Engineers (IEEE). http://dx.doi.org/10.1109/ms.2004.1331307.