Universidade Federal de Uberlândia

Faculdade de Computação

Programa de Pós-Graduação em Ciência da Computação

Especificação de Testes Funcionais usando Redes de Petri a Objetos para Softwares Orientados a Objetos

Autora: Liliane do Nascimento Vale

Orientador: Prof. Dr. Stéphane Julia

Uberlândia, MG

Fevereiro/2009

Universidade Federal de Uberlândia

Faculdade de Computação

Programa de Pós-Graduação em Ciência da Computação

Especificação de Testes Funcionais usando Redes de Petri a Objetos para Softwares Orientados a Objetos

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Uberlândia, Minas Gerais, como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: **Engenharia de Software**.

Autora: Liliane do Nascimento Vale

Orientador: Prof. Dr. Stéphane Julia

Uberlândia, MG

Fevereiro/2009

Dados Internacionais de Catalogação na Publicação (CIP)

V149e Vale, Liliane do Nascimento, 1983-
Especificação de testes funcionais usando Redes de Petri a objetos para softwares orientados a objetos / Liliane do Nascimento Vale. - 2009.
137 f. : il.

Orientador: Stéphane Julia.
Dissertação (mestrado) – Universidade Federal de Uberlândia, Programa de Pós-Graduação em Ciência da Computação.
Inclui bibliografia.

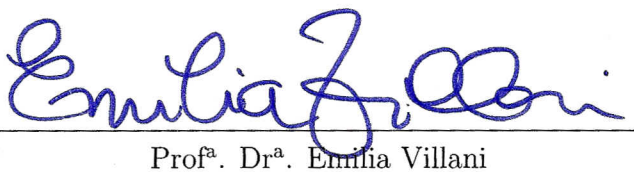
1. Software - Desenvolvimento - Teses. 2. Redes de petri - Teses. 3. Fluxo de trabalho - Teses. I. Julia, Stéphane. II. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU: 681.3.06

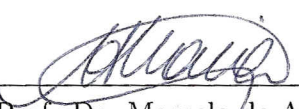
Elaborado pelo Sistema de Bibliotecas da UFU / Setor de Catalogação e Classificação

Dissertação de Mestrado sob o título *"Especificação de Testes Funcionais usando Redes de Petri a Objetos para Softwares Orientados a Objetos"*, defendida por Liliane do Nascimento Vale e aprovada em 04 de Fevereiro de 2009, em Uberlândia, Estado de Minas Gerais, pela banca examinadora constituída pelos doutores:

Prof. Dr. Stéphane Julia
Orientador



Profª. Drª. Emília Villani
Instituto Tecnológico de Aeronáutica, Divisão de
Engenharia Mecânica Aeronáutica



Prof. Dr. Marcelo de Almeida Maia
Universidade Federal de Uberlândia

Universidade Federal de Uberlândia

Faculdade de Computação

Programa de Pós-Graduação em Ciência da Computação

Autora: **Liliane do Nascimento Vale**

Título: **Especificação de Testes Funcionais usando Redes de Petri a Objetos para Softwares Orientados a Objetos**

Faculdade: **Faculdade de Computação**

Grau: **Mestrado**

Fica garantido à Universidade Federal de Uberlândia o direito de circulação e impressão de cópias deste documento para propósitos exclusivamente acadêmicos, desde que a autora seja devidamente informada.

Autora

A AUTORA RESERVA PARA SI QUALQUER OUTRO DIREITO DE PUBLICAÇÃO DESTE DOCUMENTO, NÃO PODENDO O MESMO SER IMPRESSO OU REPRODUZIDO, SEJA NA TOTALIDADE OU EM PARTES, SEM A PERMISSÃO ESCRITA DA AUTORA.

© Todos os direitos reservados a Liliane do Nascimento Vale

A Leonardo pela sua enorme e incansável espera, paciência, carinho e amor que se pode ter.

Agradecimentos

Agradeço a todos aqueles que contribuiram de alguma forma para a escrita dessa dissertação, seja direta ou indiretamente, em especial, agradeço a ...

A Deus, por todas as coisas maravilhosas que ele fez e faz por mim.

Aos meus pais, pela confiança depositada.

As minhas tias Valdira e Valmira por sempre me amparar e apoiar.

A Leonardo Duarte, meu companheiro que sempre me deu o suporte necessário, acalmando, consolando, motivando, aplaudindo, e tudo isso com todo o amor que se pode ter.

A Christophe Sibertin-Blanc, pela sua solicitude em atender os pedidos e gentilmente fornecer material.

A Stéphane Julia, acima de tudo pelo profissionalismo, paciência e compreensão desempenhados à mim ao longo desta jornada.

Aos amigos Italo, Robson, Lucio, Valquíria, Núbia, Eduardo, Cristiane, Márcio, Sérgio, Douglas, Lígia, Flávio, Klérisson, Felipe, Nádia e Rodrigo pela ajuda sob diversos aspectos e pela amizade.

A todos o meu muito obrigada!

Resumo

O objetivo deste trabalho é o de propor a formalização de testes funcionais, através da especificação destes usando redes de Petri a Objetos e Workflow-Nets no contexto de softwares orientados a objetos. Inicialmente, Workflow-Nets são usadas para representar os requisitos do software. Em seguida, Workflow-Nets a objetos derivadas dos Workflow-Nets e redes de Petri a Objetos são utilizadas para especificar formalmente os modelos de teste das diversas funcionalidades dos software. Os modelos propostos permitem tanto a representação de estruturas de dados complexas durante o teste, quanto a especificação de diversas atividades a serem realizadas para a dada funcionalidade. A execução do modelo de teste funcional quando se considera uma arquitetura de software específica, é então dada através da instanciação de uma classe de teste que fica associada à funcionalidade testada. Para ilustrar a utilização do modelo proposto, é realizado um estudo de caso aplicado ao teste de funcionalidades de um software de peças automobilísticas desenvolvido no âmbito acadêmico e comercializado.

Abstract

The main purposed of this paper is to show the formalization of functional tests specifying then with Object Petri Nets and Workflow-Nets in relation to software focused on objects. Initially, Workflow-Nets are used to represent the main requirements of the software. Next, Object Workflow-Nets derived from Workflow-Nets and Objects Petri Nets are used to specify formally the test models of many software functionalities. The proposed models allow complex data structures to be represented during the test, concerning each one of the many functions to be worked out. The execution of the model functional test, when considering a software architecture, is given by the instantiation of a test class associated with the tested functionality. To enlight this model, a concret study of it is applied to the functionality tests of a commercial software of automobile parts, developed in the ambit academic and marketed.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

A construção de sistemas computacionais, muitas vezes, é realizada sem o emprego de boas práticas de desenvolvimento que a Engenharia de Software oferece. Dessa forma, sérios problemas resultantes do processo de desenvolvimento empregado podem ter impactos que afetam, por exemplo, a qualidade e confiabilidade do software.

Qualidade e confiabilidade sempre foram exigidas, embora no passado eram mais facilmente atendidas, devido a simplicidade dos programas desenvolvidos. Atualmente, devido a crescente dependência da sociedade em relação ao uso de softwares para a automação de atividades, são requeridos um nível cada vez maior de qualidade e confiabilidade no desenvolvimento de sistemas computacionais. Com isso, surge a necessidade de aperfeiçoar e desenvolver métodos e técnicas de testes de software que garantam maior segurança durante a execução de atividades desempenhadas por softwares.

Dentro do contexto de desenvolvimento de software, é essencial que a atividade de teste de software seja incluída em paralelo a todas as fases de desenvolvimento do sistema, a fim de detectar falhas no mesmo. Para tanto, o trabalho de teste de software é dividido também em diversas fases, de forma que cada fase corresponda a uma etapa de desenvolvimento de software. Dentre as categorias de teste existentes, é abordado neste trabalho o teste funcional.

O teste funcional verifica a funcionalidade de um software avaliando-se os requisitos (dados de entrada e de saída) do mesmo. Dessa forma, é aplicado essencialmente na etapa de especificação de requisitos do software e baseia-se em técnicas de modelagem.

A atividade de modelagem de software permite construir modelos que representam a estrutura e o comportamento desejado do sistema. Se tal atividade for omitida, pode acarretar o surgimento de erros indesejáveis durante a construção do software.

A elaboração de um modelo funcional do software deve levar também à construção de modelos de teste funcional do mesmo. A execução de um modelo de teste funcional pode ser visto como a execução de um caso de teste (semelhante a execução de caso de

um processo de negócio) que possui um início e um fim, executando várias operações de acordo com o roteiro a ser seguido pelo teste.

A idéia central deste trabalho é a de elaborar um modelo formal de especificação de teste funcional usando redes de Petri a Objetos [Sibertin-Blanc, 1985] e conceitos de processos de Workflow [Aalst and Hee, 2002]. Nas redes de Petri a Objetos, é possível representar tanto estruturas de dados quanto fluxos de controle. Por outro lado, as Workflow-Nets apresentam uma estruturação do controle que é adaptada à execução de testes funcionais.

Será interessante então incorporar tal modelo nas atividades de teste existentes em modelos de desenvolvimento em V de softwares em que atividades de teste são de fundamental importância, em particular para a correta especificação dos requisitos funcionais do software.

No desenvolvimento deste trabalho, abordagens multiformalismos, envolvendo diagramas comumente utilizados na indústria de software como UML, DFD's são também considerados, a fim de situar a abordagem proposta dentro de uma visão mais prática de desenvolvimento de software.

O capítulo 2 será dedicado aos principais fundamentos teóricos que serão utilizados neste trabalho. Os conceitos sobre processos de construção de software são apresentados, assim como modelos de workflow e linguagens de especificação de requisitos de software como diagramas da UML e redes de Petri.

O capítulo 3 apresentará conceitos básicos da atividade de teste de software e os principais trabalhos realizados na área de teste funcional.

No capítulo 4 será proposta a definição formal e a metodologia de construção do modelo de teste funcional usando as redes de petri a Objetos e Workflow Nets. Serão também consideradas uma abordagem multiformalismo adotando a utilização de diagramas semiformais que representarão diveros aspectos comportamentais e estruturais do software. Um exemplo da funcionalidade "Saque" de um caixa eletrônico será utilizado para ilustrar diversos aspectos da abordagem proposta.

No capítulo 5, a abordagem apresentada neste trabalho será ilustrada passo a passo em um estudo de caso, utilizando um exemplo real de um sistema comercial desenvolvido no meio acadêmico, orientado a objeto, implementado em linguagem Java.

O capítulo 6 apresentará a conclusão do trabalho além de mostrar resultados obtidos e apresentar sugestões para trabalhos futuros.

2 *Elementos Teóricos da Engenharia de Software*

Este capítulo contempla os fundamentos teóricos que serão abordados neste trabalho.

2.1 Modelos de Processos de Produção de Software

Um sistema computacional pode ser definido como sendo um conjunto de elementos que compõem o hardware e o software, que interagem entre si e formam uma estrutura organizada para a execução de operações [Pressman, 2001].

Os sistemas computacionais foram desenvolvidos no intuito de proporcionarem maior facilidade na realização de tarefas humanas. Exigi-se cada vez mais qualidade e segurança dos sistemas produzidos pelo fato de estes sistemas, na maioria das vezes, lidarem com aspectos econômicos e sociais.

Para lidar com esta realidade, a Engenharia de Software foi organizada de forma a disciplinar a produção de softwares a partir da introdução de atividades que permitem o gerenciamento de projetos e a utilização de ferramentas e métodos teóricos que auxiliam o desenvolvimento de sistemas computacionais [Summerville, 2003].

Para a produção de software é necessário o desenvolvimento de um conjunto de atividades que pertencem a um processo. Um processo de produção de software pode ser executado seguindo vários modelos. Cada um tem suas próprias características que permitem destacar informações do software sob diferentes pontos de vista. Entre os principais modelos existentes podem ser citados [Summerville, 2003]:

- **Modelo em Cascata ou Seqüencial:** retrata algumas das atividades de desenvolvimento de software como: a análise e definição de requisitos, o projeto de sistemas e de software, a implementação e testes de unidade, a integração e teste de sistema e, por último, a operação e manutenção. Neste modelo, existe uma seqüência

de etapas a serem seguidas de forma que, ao término de cada etapa, um documento é produzido e aprovado para dar início à próxima etapa. Uma dificuldade encontrada neste modelo é a sua inflexibilidade em retornar as etapas anteriores do ciclo de desenvolvimento, o que dificulta a realização de alterações em algumas destas etapas. Dessa forma, este modelo deve ser aplicado somente quando os requisitos são bem definidos e compreendidos. A Figura 1 ilustra o modelo.

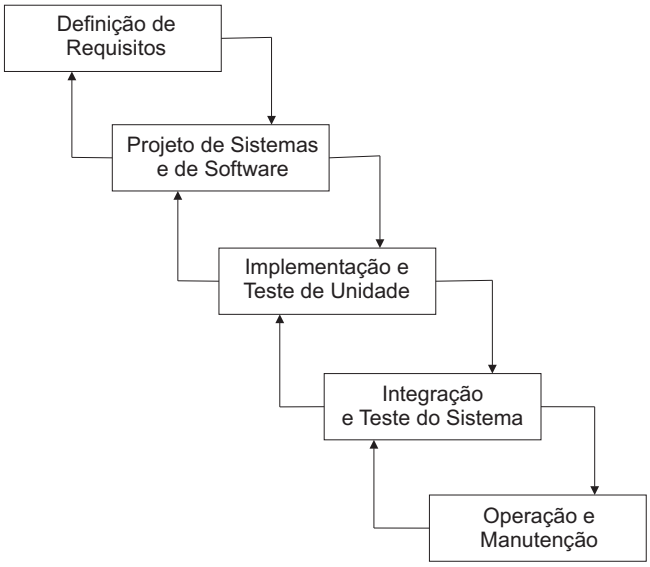


Figura 1: Modelo em Cascata

- **Modelo em V:** é uma variação do Modelo em Cascata. Neste modelo é acrescentada, a cada fase de desenvolvimento do software, uma etapa de teste correspondente. No Modelo em V, ilustrado na Figura 2, os testes de unidade e de integração são utilizados para verificar o código do programa, o teste de sistema se limita a testar o projeto de software e o teste de aceitação valida os requisitos do sistema a fim de confirmar se estes estão corretos. Dessa forma, o Modelo em V dá ênfase ao teste e à correção, enquanto que o Modelo em Cascata, à confecção de documentos e artefatos de software.

- **Modelo em Espiral ou Iterativo:** neste modelo, cada ciclo representa uma fase do processo de produção de software. O primeiro ciclo é voltado para a análise de viabilidade do sistema, o próximo define os requisitos do sistema, o seguinte, o projeto do sistema, e assim por diante. A principal característica deste modelo é a inclusão da análise de riscos em todas as fases de desenvolvimento. A Figura 3 representa tal modelo.

- **Modelo Incremental:** neste modelo, as funções mais importantes do sistema são

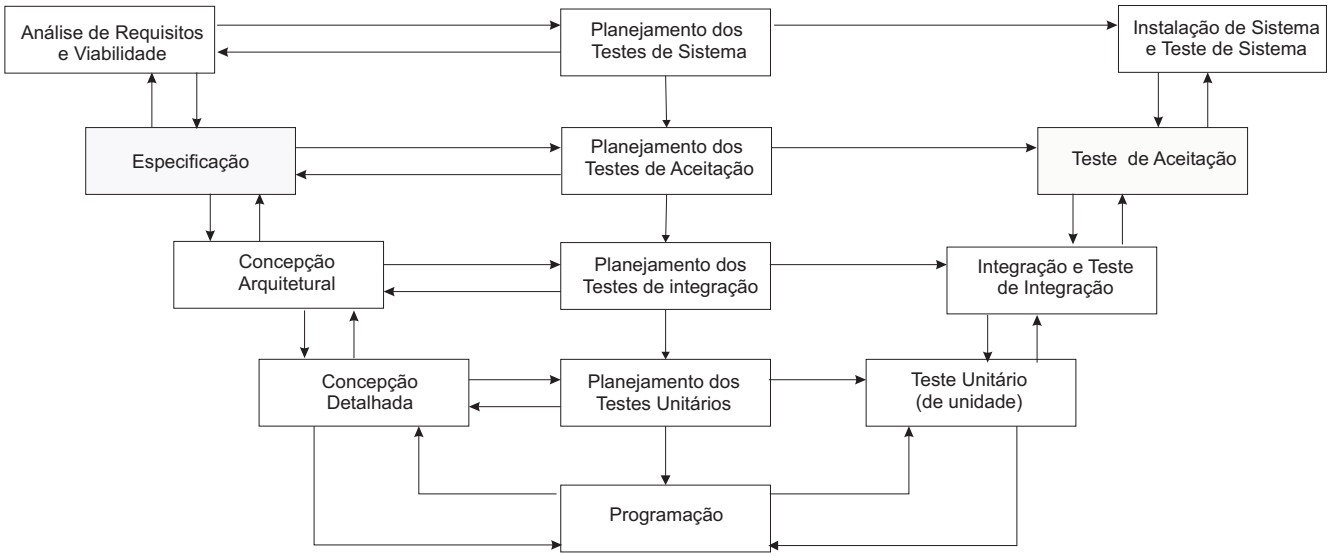


Figura 2: Modelo de Desenvolvimento em V

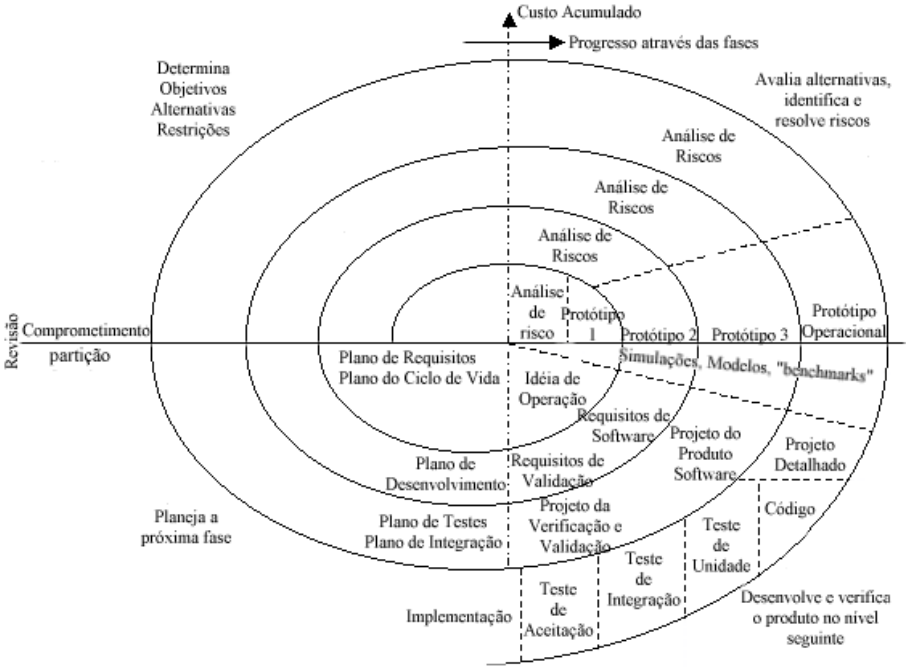


Figura 3: Modelo Espiral

identificadas e fornecidas pelos clientes. Assim, o software é desenvolvido em estágios. A cada estágio, de acordo com a prioridade e a necessidade, um conjunto de funções são entregues para uso e avaliação do cliente. A medida que novas partes do sistema são concluídas, estas são incrementadas (integradas) aos estágios já existentes. Dessa forma, não há necessidade do cliente esperar que o sistema todo fique pronto para usá-lo. Entretanto, este modelo apresenta o inconveniente de gerar um software mal-estruturado, de difícil compreensão e manutenção por permitir que

requisitos e decisões de projeto sejam postergados. A Figura 4 ilustra o Modelo Incremental.

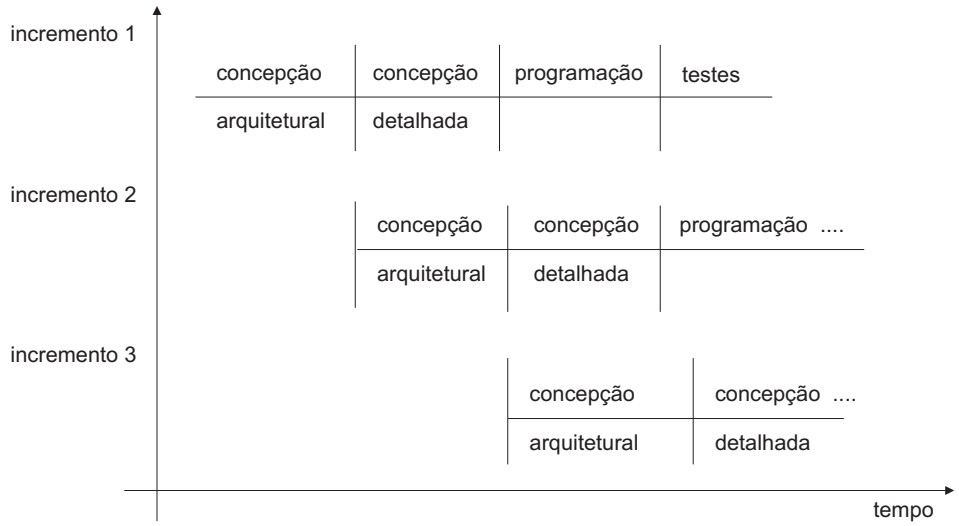


Figura 4: Modelo Incremental

2.2 Especificação de Requisitos de Software

Para que a construção de um software seja bem sucedido, é necessário que haja planejamento do qual fazem parte a análise dos prazos de entrega e dos recursos financeiros e tecnológicos, o que torna possível realizar a concepção do sistema.

Uma vez descrito o problema, cabe aos desenvolvedores propor uma solução. Inicialmente, os desenvolvedores necessitam identificar as características que o sistema deve possuir, ou seja, o que o sistema deve fazer a fim de atender às necessidades dos usuários e as restrições sobre suas funcionalidades. A essas características dá-se o nome de requisitos.

Basicamente, os requisitos podem ser classificados em duas categorias: requisitos funcionais e não-funcionais. Os requisitos funcionais referem-se a características operacionais do sistema e os requisitos não-funcionais são voltados para características relacionadas às restrições do sistema como tempos de resposta, segurança e desempenho.

Na maioria dos casos, os requisitos identificados são inconsistentes ou ambíguos devido ao fato de muitas vezes os clientes, e também usuários, não saberem expressar a real necessidade das funcionalidades do sistema. Uma forma de amenizar e eliminar este problema é usar linguagens de modelagem.

As linguagens de modelagem oferecem um poder de compreensão maior, além de permitir precisão, consistência e segurança na descrição de sistemas. Entre as linguagens

de especificação de sistemas mais comumente usadas são: os diagramas semi-formais da UML, os diagramas formais como as Redes de Petri, *Statecharts*, Máquina de Estados Finitos, os diagramas de Fluxo de Dados e os diagramas SA-RT, brevemente descritos a seguir.

2.2.1 A Notação UML

Construída inicialmente por Grady Booch, James Rumbaugh e Ivar Jacobson, a *Unified Modeling Language* (UML) consiste na união de diversas notações pré-existentes: alguns elementos foram removidos e outros foram adicionados com o objetivo de torná-la mais expressiva [Bezerra, 2007].

A UML é uma linguagem de modelagem gráfica (visual), orientada a objetos que pode ser empregada em várias fases de desenvolvimento de um software. É composta de dois tipos de diagramas: diagramas estáticos e diagramas dinâmicos.

Os diagramas estáticos compreendem a relação do sistema com os dados. Os diagramas que se enquadram nessa descrição são: Diagramas de Classes, de Objetos, de Componentes e de Implantação.

Já os diagramas dinâmicos referem-se à parte de controle do sistema como o envio de mensagens e ordens de ativação. Os diagramas dessa categoria são: Diagramas de Casos de Uso, de Seqüência, de Colaboração, de Atividades e de Estado-Transição.

Na UML não é possível modelar o sistema de forma integral e, sim, apenas algumas partes, o que fornece uma visão apenas parcial deste, uma vez que se todas as informações do software representado fossem incluídas em um único diagrama, este seria de difícil compreensão e teria pouca utilidade.

A seguir, são apresentadas as descrições de alguns diagramas abordados neste trabalho [Bezerra, 2007]:

- **Diagramas de Casos de Uso:** representam as funcionalidades externamente observáveis do sistema e a interação do mesmo com elementos externos (atores). São caracterizados como um modelo de análise dos requisitos funcionais do sistema e ilustram os seus possíveis usos. Os principais componentes desse modelo são os *casos de uso* (especifica uma seqüência completa de interações entre o sistema e os agentes externos a ele), os *atores* (entidades externas que interagem com o sistema, trocando ou recebendo informações) e os *relacionamentos* (estabelece a relação en-

tre os atores e os casos de uso). A Figura 5 ilustra um exemplo dos Diagramas de Casos de Uso.

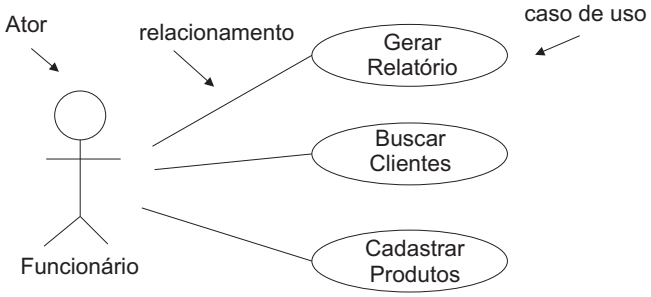


Figura 5: Diagrama de Casos de Uso

- **Diagramas de Classes:** é o diagrama mais rico em termos de notação, pois descreve um conjunto de classes que compartilham atributos, métodos, relacionamento e semântica. As classes são formadas por um retângulo dividido em três partes: o *nome* da classe, uma lista de *atributos*, que são os dados que a classe possui, e as *operações* que manipulam esses atributos. Os *relacionamentos* existentes nos Diagramas de Classes podem ser do tipo generalização/especialização, agregação, associação e composição. A Figura 6 ilustra um Diagrama de Classes com seus principais componentes e nas Figuras 7 (a), (b) e (c) apresentam, respectivamente, a agregação, a composição e a generalização. A seguir são descritos os relacionamentos:

  - **generalização/especialização:** é também conhecido como herança e indica o relacionamento entre um elemento mais geral (superclasse) e um mais específico (subclasse);
  - **associação:** as associações definem as regras para o modo como as classes podem se relacionar umas com as outras. Representam relacionamentos estruturais entre instâncias e especificam que objetos de uma classe estão ligados a objetos de outras classes. Pode haver associação uniária, binária, etc;
  - **agregação:** a agregação é um tipo especial de associação usada para indicar que os objetos participantes não são apenas objetos independentes que conhecem uns aos outros. Em vez disso, eles são mostrados ou configurados juntos para criar um novo objeto, mais complexo;
  - **composição:** é um tipo especial de agregação. É usada para agregações com forte relação entre o todo e a parte. O tempo de vida da parte depende do

tempo de vida do objeto agregado. Resumidamente, isso significa que a parte não existe sem o todo.

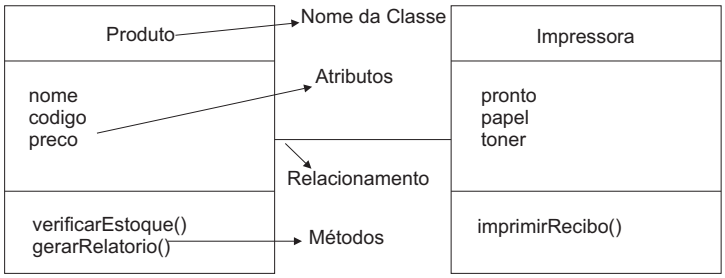


Figura 6: Diagrama de Classes

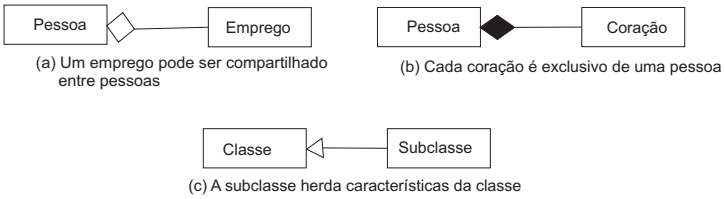


Figura 7: Relacionamento de Agregação, Composição e Generalização

- **Diagramas de Seqüência:** representam a interação entre os objetos na ordem temporal em que ela ocorre. Os principais elementos gráficos que compõem o modelo são: um retângulo ao topo, que representa um *objeto*; as *linhas de vida*, que se estendem a partir do objeto e são projetadas para baixo; as *mensagens*, representadas por uma flecha horizontal que une uma linha de vida à outra, e o *foco de controle*, que indica um tempo relativo, o ponto em que a mensagem pode ser transmitida, mostrando os períodos em que um determinado objeto está participando ativamente do processo, executando um ou mais métodos do processo. A Figura 8 mostra um Diagrama de Seqüência.

- **Diagramas de Atividades:** podem ser vistos como uma extensão dos fluxogramas. Os Diagramas de Atividades representam os estados de uma *atividade* (retângulos) orientados por *fluxos de controle*, representados graficamente por setas e os losangos indicam pontos de decisão. Nesses diagramas é possível representar o *paralelismo*, através das barras de sincronização, e a interação que existem entre os casos de uso. A Figura 9 ilustra um Diagrama de Atividades e seus principais elementos.

Inicialmente, o Diagrama de Atividades baseava-se no antigo Diagrama Gráfico de Estados, mas, a partir da UML 2.0, esse diagrama tomou características próprias, deixando

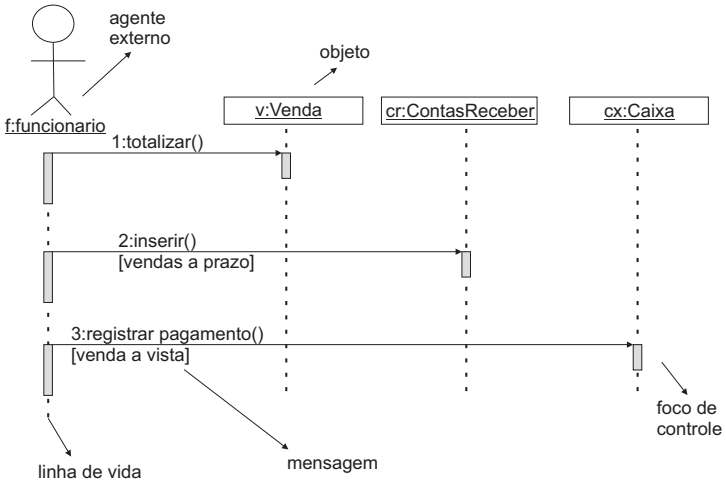


Figura 8: Diagrama de Seqüência

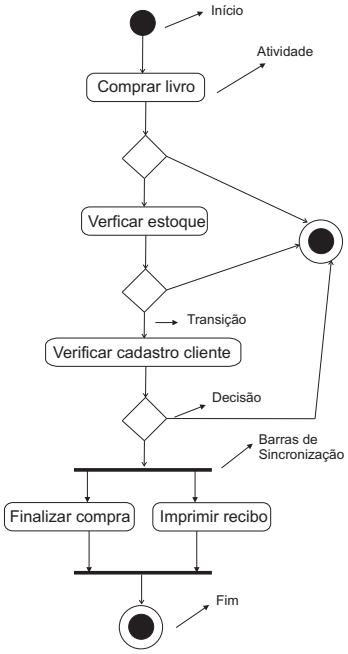


Figura 9: Diagrama de Atividades

inclusive de se basear em máquinas de estados e passando a se basear nas Redes de Petri [Störrle, 2005].

Os Diagramas de Atividades descrevem os passos a serem percorridos para a conclusão de uma atividade específica, muitas vezes representada por um método, e podem, também, modelar um processo completo.

Um Diagrama de Atividades pode ser considerado apenas um nível intermediário entre a especificação textual e uma especificação formal dada por um modelo de Rede de Petri, por exemplo.

Um Diagrama de Atividades é, portanto, uma representação gráfica das atividades

desempenhadas no sistema, a partir da qual é possível visualizar o fluxo de dados e de controle que existe de uma atividade para outra.

Dentre os modelos visuais da UML, os Diagramas de Atividades são os que descrevem as atividades que ocorrem em um caso de uso, mostram atividades distintas manipuladas por diferentes elementos, representam segmentos paralelos que existem em um sistema e fornecem uma ampla visualização dos processos de negócios.

A sintaxe dos Diagramas de Atividades permaneceu a mesma em relação ao fluxo de controle, porém, no que diz respeito ao fluxo de dados, a sintaxe abstrata e a semântica foram redefinidas, adotando-se para isso, as características das Redes de Petri. Entre as principais mudanças podem ser citadas:

- Flexibilidade das raias (*swimlanes*) que simulam uma espécie de mapeamento de casos de uso. Particiona em grupos os estados de atividades, de modo que cada grupo representa um elemento responsável pela atividade. Assim, cada grupo é chamado de raia de natação pois os grupos ficam separados de seus vizinhos graficamente por uma linha vertical.
- Estados de subatividades desapareceram e seus lugares foram ocupados pelas atividades subordinadas às ações que definem o comportamento de atividades superiores.
- Os fluxos de dados são denotados por *ObjectNodes* e *ObjectFlows*.

Na versão 2.0 dos Diagramas de Atividades, verifica-se o suporte de modelagens de fluxos de dados, de controle, e de uma ampla variedade de domínios: do computacional ao físico.

Elementos Gráficos dos Diagramas de Atividades da UML 2.0

Entre os elementos gráficos presentes nos Diagramas de Atividades da UML 2.0 estão:

- **Ações:** especifica um comportamento, indicando uma transformação ou processamento na representação de atividades. As ações são usadas para criar objetos, definir valores de atributos, ligar objetos. Podem ter *inputs* (entradas) e *outputs* (saídas), conhecidos como pinos, conectados por arestas em que fluem objetos, representando os valores que fluem pelas atividades. A Figura 10 mostra uma ação que cria uma nova instância da classe *Pedido*, e outra ação, invocada para preencher o pedido.

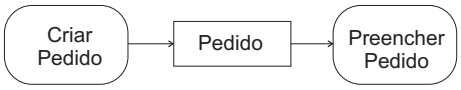


Figura 10: Exemplo de Ação

- **Atividades:** são comportamentos definidos por usuários e podem ser representados pela invocação de ações e parâmetros para receber e providenciar dados. Uma atividade pode conter nela outras atividades ou uma sequência de ações. Entre as ações que podem ser invocadas são exemplos, as ações de comunicação e a manipulação de objetos. A Figura 11 mostra um exemplo de atividade e a Figura 12 ilustra um processo de compra.

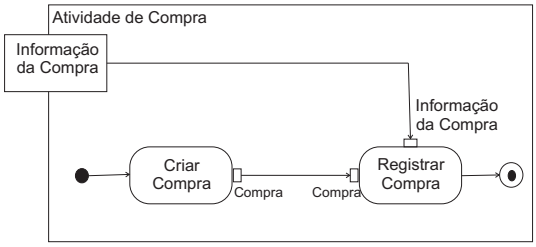


Figura 11: Exemplo de Atividade

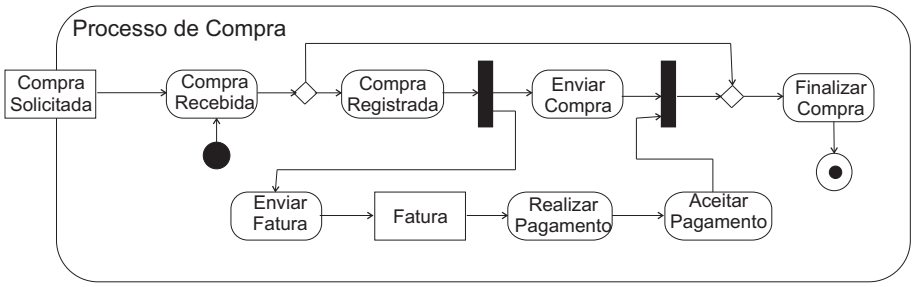


Figura 12: Exemplo de um Processo de Pedido

- **Nó de objeto (*ObjectNode*) e fluxos de objetos (*ObjectFlow*):** indica uma instância particular disponível em um determinado ponto da atividade. As arestas ligadas a um *ObjectNode* são chamadas de *ObjectFlow*, e por sua vez, denotam fluxos de dados. Existem três modos diferentes de representar o fluxo de dados, como mostrados na Figura 13. A primeira notação é representada pelos Diagramas de Atividades da UML 1.5, e as demais notações pela nova versão. Dessa forma, é possível observar que um item do fluxo de dados é adicionado a uma aresta do fluxo de controle.

- **Fluxo de controle (*Control Flow*):** representam decisões que indicam a possibilidade de escolha entre os fluxos disponíveis, podendo haver um ponto de entrada

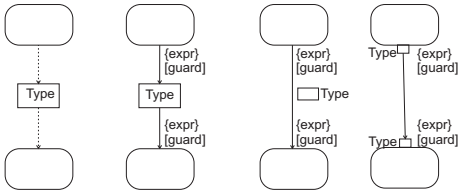


Figura 13: Notações para Fluxos de Dados

e vários de saída.

- **Banco de dados (*DataStore*):** armazena dados pertinentes ao desempenho das atividades. A Figura 14 ilustra um exemplo de *DataStore* em que a atividade *Contratar Empregado* ao ser desempenhada, inicializa o banco de dados, o qual armazena dados cadastrais de empregados através da execução da atividade *Atribuir Empregado*. A seta que une o banco de dados à atividade *Atribuir Empregado*, contém o comando *empregado=null* que indica que uma linha da tabela do banco de dados está vazia, assim, quando a atividade *Atribuir Empregado* é executada, um novo empregado é cadastrado.A expressão *peso=total* é apenas uma atribuição que é executada sob a restrição de tempo representada por $\bowtie$ que indica que a variável *peso* só receberá um valor de *total* a cada uma vez por ano.

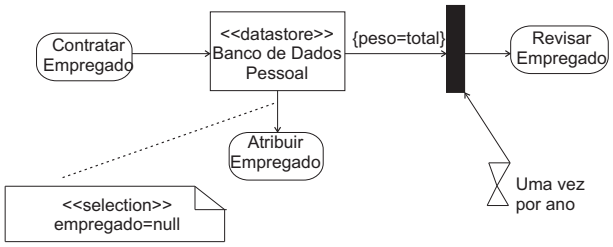


Figura 14: Exemplo de um *DataStore*

- **Nó final (*FinalNode*):** indica a finalização do processo de execução. Uma atividade pode ter mais que um nó final.

- **Nó de decisão (*DecisionNode*):** nó de controle entre fluxos alternativos.

- **Final de fluxo (*FlowFinal*):** tem a função de interromper um determinado fluxo, mas não tem efeitos sobre os demais fluxos da atividade.

- **Nó de bifurcação (*ForkNode*):** mostra uma aresta entrando e múltiplas saindo, representando ações paralelas.

- **Nó inicial (*InitialNode*):** uma atividade pode representar mais que um nó inicial.

- **Nó de junção (*JoinNode*):** indica uma sincronização. Apresenta múltiplas entradas e apenas uma saída.
- **Nó de controle (*MergeNode*):** une múltiplos fluxos alternativos. Não é usado para a sincronização de fluxos concorrentes.
- **Ação de envio de sinal (*SendSignalAction*):** é uma ação que cria um sinal a partir das entradas (*inputs*), transmitindo-o para um objeto, e pode representar também o disparo de uma transição de uma máquina de estados ou o disparo de uma atividade.
- **Partições (Activity Partition):** os nós e as arestas são divididos para mostrar uma visão específica, como ilustra a Figura 15.

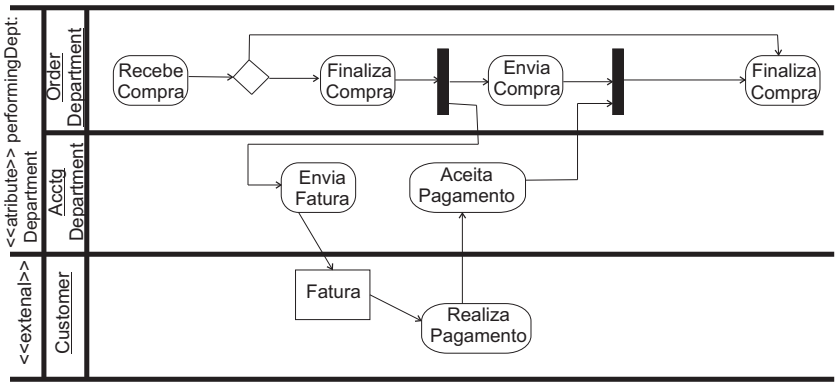


Figura 15: Exemplo de uma Partição

A Figura 16, ilustra os demais elementos que compõem um Diagrama de Atividades.

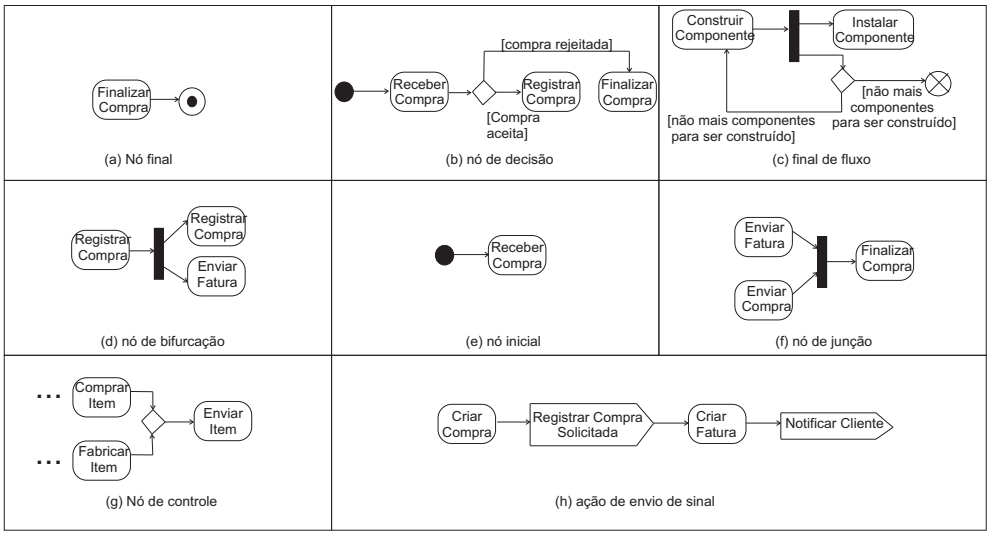


Figura 16: Elementos dos Diagramas de Atividades

2.2.2 Análise Estruturada

A análise estruturada define um sistema usando módulos de maneira hierárquica e descendente. Dessa forma, é possível compreender a lógica dos dados que existem no sistema.

Basicamente, a análise estruturada é formada por dois componentes: modelo ambiental e comportamental.

O modelo ambiental é caracterizado por definir a fronteira que existe entre o ambiente e o sistema por meio da definição de uma interface. De modo geral, o modelo ambiental é composto pela *declaração de objetivos* que o sistema espera alcançar e pela *lista de eventos*, que são os estímulos que ocorrem no mundo exterior e aos quais o sistema deve responder. Os diagramas que representam essas características são os Diagramas de Contexto.

Já o modelo comportamental, por sua vez, modela a parte interna do sistema em que para cada evento da lista de eventos, é definido um processo com as entradas e as saídas de dados que existem. Nesta fase, Diagramas de Fluxo de Dados, descritos a seguir, mostram como os dados fluem de um processo a outro e os Diagramas de Fluxo de Controle descrevem tanto o fluxo de dados quanto o fluxo de controle que existem para a execução dos processos e podem ser usados para representar tal comportamento.

2.2.2.1 Diagramas de Fluxo de Dados

Os Diagramas de Fluxos de Dados (DFD's) mostram como os dados fluem para dentro e para fora de um sistema [Pfleeger, 1999] através da execução de processos os quais manipulam dados de entrada e dessa forma, dados de saída são gerados.

Graficamente, a entrada dos dados num processo é representada por uma *seta* entrando numa *bolha*, a qual corresponde ao processo, e a saída é ilustrada por uma seta que sai da bolha. Assim, o processo descreve a transformação do fluxo de entrada em um fluxo de saída como visualizados na Figura 17.

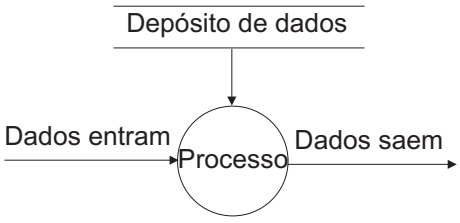


Figura 17: Elementos Gráficos do Diagrama de Fluxo de Dados

Em alguns casos, é utilizado um *Depósito de Dados* em que informações são armazenadas como em um banco de dados. Graficamente, este depósito é representado por duas barras paralelas como é mostrado na Figura 17.

O outro elemento que compõe os Diagramas de Fluxos de Dados são os *atores* caracterizados como sendo entidades externas que recebem ou fornecem dados em um processo. A notação gráfica que representa os atores é um retângulo. A Figura 18 ilustra o Diagrama de Fluxo de Dados típico de uma visita ao médico.

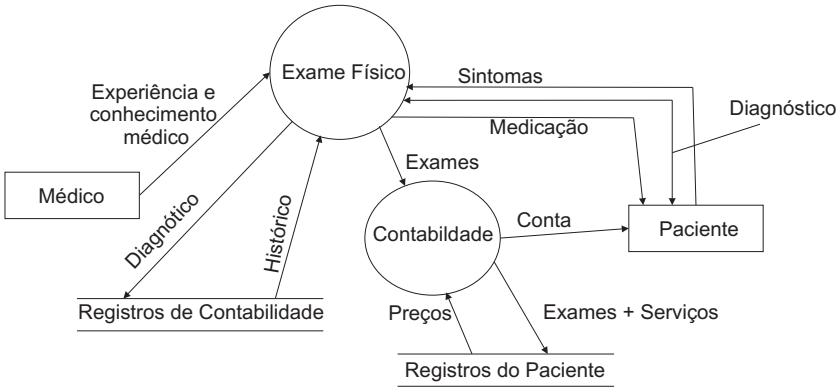


Figura 18: Diagrama de Fluxo de Dados

Outro elemento que acompanha o Diagrama de Fluxo de Dados é o *dicionário de dados* que descreve as informações contidas no fluxo de dados ou no repositório de dados de forma completa. No exemplo da Figura 18, considere o dicionário de dados para o fluxo *Medicação*, que pode conter:

- **Medicação**= lista de nomes de remédios + quantidade de dias que deve ser usado o medicamento + quantidade de vezes ao dia que dever usado o remédio;
- **lista de nomes de remédios**= String;
- **quantidade de dias que deve ser usado o medicamento**= Integer;
- **quantidade de vezes ao dia que deve ser usado o remédio**= Integer;
- **preços**= Float;
- **conta**= Float;
- **sintomas**= String;
- **diagnóstico**= String;

- **exames**= String;
- **serviços**= String;
- **experiência e conhecimento médico**= String;

No dicionário de dados podem ser usados alguns operadores formais que auxiliam na manipulação do dados como "+", que indica junção, "=", que indica equivalência, entre outros.

2.2.2.2 Diagramas SA-RT

Os Diagramas SA-RT (*Structure Analysis Real Time*) [Hatley and Pirbhai, 1987], [Ward and Mellor, 1985] são uma variação dos trabalhos de Tom DeMarco [DeMarco, 1979] sobre os Diagramas de Fluxo de Dados, voltada para a aplicação em sistemas de tempo real. A única diferença é a inclusão do fluxo de controle responsável por representar o aspecto dinâmico que existe no sistema.

Basicamente, um Diagrama SA-RT é composto pelos seguintes elementos gráficos (como mostra a Figura 19): *fluxo de dados*, que representa os dados que fluem no sistema; *fluxo de controle*, que representa as ações e eventos manipulados pelo sistema; *processo*, onde os dados são transformados , e por fim, o C-Spec (*Control Specification*), que trata os eventos recebidos, transformando-os em ações e coordena os diversos processos do sistema. A Figura 20 ilustra um exemplo de diagrama de SA-RT.

Figura 19: Elementos do Diagrama SA-RT

Inicialmente, o diagrama define o ambiente externo, que vai interagir com o sistema através dos Diagramas de Contexto (caracterizados por serem diagramas que permitem representar todas as entidades externas que podem interagir com um sistema. É similar a um Diagrama de Blocos, que recebe e envia dados de entrada e saída para entidades externas), visualizado na Figura 21 em que este é semelhante a um DFD de mais alto nível, que representa todo o sistema como um único processo, composto por fluxos de dados que mostram as interfaces entre o sistema e as entidades externas. Em sequência, especifica-se

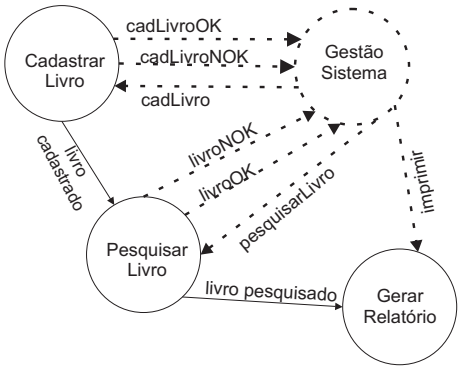


Figura 20: Exemplo de Diagrama SA-RT

as principais funcionalidades do sistema por meio das transformações de dados oferecidos pelos DFD's. Posteriormente, é descrito o controle por meio das transformações de controle oferecidas pelos Diagramas de Fluxo de Controle (DFC).

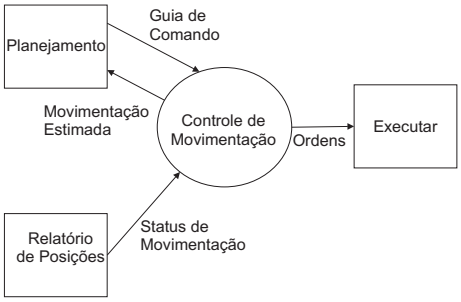


Figura 21: Exemplo de Diagrama de Contexto

A cada transformação, o dado é decomposto em níveis hierárquicos em que são criados novos DFD's, combinados com o DFC correspondente.

2.2.3 Redes de Petri

As Redes de Petri foram inicialmente postuladas por Carl Adam Petri em *Comunicação com Autômatos* [Petri, 1966]. A partir deste trabalho, foram desenvolvidas as teorias das Redes de Petri que constituem formalmente, uma ferramenta gráfica e matemática para modelagem de diversos tipos de sistemas voltados para a análise e controle de sistemas e eventos discretos, suportando atividades paralelas, concorrentes e assíncronas. A definição completa das Redes de Petri podem ser vistas em [Cardoso et al., 1999], [David and H., 2004], [Hatley and Pirbhai, 1989], [Peterson, 1981].

De maneira simplificada, uma Rede de Petri como mostrada na Figura 22, é apresentada como um grafo orientado contendo os seguintes elementos:

- **Lugar:** representa um estado, uma espera, um recurso etc;
- **Ficha:** indica uma condição associada ao lugar, ou um objeto;
- **Transição:** representa ação ou evento que ocorre em um sistema;
- **Arcos orientados:** conectam os lugares às transições e estas aos lugares.

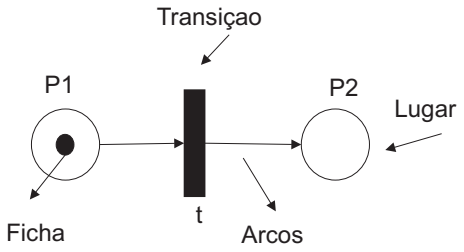


Figura 22: Rede de Petri

Definição 1: *Formalmente uma rede de Petri pode ser definida como uma quádrupla:*

$$R =< P, T, Pre, Post >$$

Onde:

- P é um conjunto finito de lugares de dimensão n,
- T é um conjunto finito de transições de dimensão m,
- $Pre : P \times T \rightarrow N$ é a aplicação de entrada (lugares precedentes ou incidência anterior), indica o peso do arco, ligando um lugar P a uma transição t com N sendo o conjunto dos números naturais os quais representam o número de fichas associados aos lugares,
- $Post : P \times T \rightarrow N$ é a aplicação de saída (lugares seguintes ou incidência posterior), indica o peso do arco, ligando uma transição t a um lugar P.

A partir dos elmentos $a_{ij} = Pre(p_i, t_j)$ que indicam o peso do arco, ligando o lugar de entrada p_i à transição t_j. Assim, a matriz de indência anterior Pre de dimensão nxm em que o número de linhas é igual ao número lugares e o número colunas é igual ao número de transições. Da mesma forma, acontece para a matriz de incidência posterior $Post$ de dimensão nxm é definida a partir dos elementos $b_{ij} = Post(p_i, t_j)$ [Cardoso et al., 1999].

Para as redes de Petri marcadas a seguinte definição é apresentada:

Definição 2: *Uma rede marcada N é uma dupla $N =< R, M >$ onde:*

- R é uma rede de Petri,
- M é a marcação inicial dada pela aplicação $M : P \rightarrow N$.

Dessa forma $M(p)$ é o conjunto de fichas contidas no lugar p. A marcação M é a distribuição das fichas nos diversos lugares. Um exemplo de rede de Petri marcada é dado na Figura 23 em que a marcação é dada por M^T=[1 0 3 0 1] (T é o transposto do vetor).

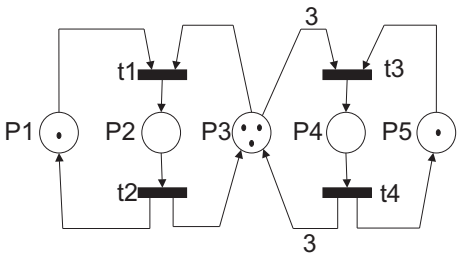


Figura 23: Exemplo de Rede de Petri Marcada

Definição 3: *Uma transição t está sensibilizada ou habilitada, se e somente se:*

$$\forall p \in P, M(p) \geq Pre(p,t) \tag{2.1}$$

Isto é, uma transição está sensibilizada se o número de fichas em cada um dos seus lugares de entrada for maior (ou igual) que o peso do arco que liga este lugar à transição.

Definição 4: *Se t está sensibilizada por uma marcação M, uma nova marcação M' pode ser obtida através do disparo de t de maneira que:*

$$\forall p \in P, M'(p) = M(p) - Pre(p,t) + Post(p,t) \tag{2.2}$$

O disparo de uma transição t consiste então em retirar as fichas dos lugares de entrada ($Pre(p,t)$), e em depositar fichas em cada lugar de saída ($Post(p,t)$). Um exemplo de disparo de transição que representa portanto, a dinâmica do sistema modelado é dado na Figura 24. A notação matricial desta rede é dada por:
$Pre(p_1, t_1) = 1$
$Pre(p_2, t_1) = 1$
$Pre(p_3, t_1) = 0$
$Pre(p_4, t_1) = 0$

$Post(p_1, t_1) = 0$
$Post(p_2, t_1) = 0$

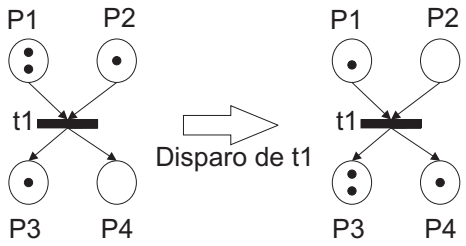


Figura 24: Exemplo de Disparo de Transição

$Post(p_3, t_1) = 1$
$Post(p_4, t_1) = 1$

A grande diversidade de sistemas existentes apresentam características distintas que necessitam ser representadas de forma clara e sem ambigüidades. As redes de Petri promovem a solução para a maioria destes problemas através de extensões. De um modo geral, as redes de Petri podem ser classificadas em: redes de baixo nível (Redes de Petri ordinárias) e redes de alto nível.

As redes de Petri de baixo nível são caracterizadas pelo tipo de marcação que possuem. As marcas (fichas) que se encontram nos lugares da rede não possuem significado, indicam apenas o estado do sistema. Entre as redes dessa categoria econtra-se as Redes de Petri Interpretadas.

Nas Redes de Petri Interpretadas são associadas variáveis às transições da rede, que representam condições e ações existentes no sistema. As variáveis podem indicar, ainda, o estado de atuadores, sensores etc. A Figura 25 mostra uma de Rede de Petri Interpretada para um exemplo simplificado de manufatura de porcas e parafusos.

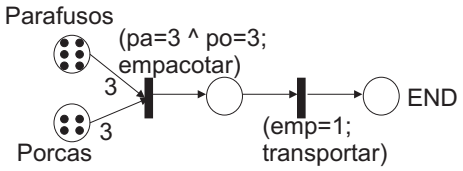


Figura 25: Rede de Petri Interpretada

As redes de Petri de alto nível, além de representarem o estado do sistema através da distribuição das fichas (marcas), conseguem também carregar informações. As marcas não são idênticas entre si e contém informações e dados que permitem a diferenciação uma das outras. Alguns tipos de rede de alto nível são apresentados a seguir:

- **Redes de Petri Coloridas:** às fichas dessas redes são atribuídas cores no intu-

ito de diferenciar umas das outras, o que permite representar diferentes processos e recursos. Além disso, as marcas representam outros recursos, como por exemplo a existência de uma outra rede associada às marcas (rede dobrada). Dessa forma, permite reduzir o tamanho da rede [David and H., 2004], [Jensen, 1990]. Em [Moncelet et al., 1998], um modelo de rede de Petri colorida foi proposto para modelagem de sistemas de produção, por exemplo.

- **Redes de Petri Predicado/Transição:** descrevem de maneira estruturada o conjunto *controle* e *dados* [Cardoso et al., 1999], [Genrich, 1987]. Nestas redes, os lugares são chamados de predicados, as marcas são condições válidas do predicado e as transições são consideradas como regras da lógica de primeira ordem, isto é, regras com variáveis. Por exemplo, em [Champagnat et al., 1998] as redes de Petri predicado/transição são abordadas para serem integradas a uma seqüência de equações álgebro-diferenciais para a modelagem de estocagem de gás.

- **Redes de Petri Temporais/Temporizadas:** são usadas na modelagem de sistemas que levam em consideração o fator tempo. Suas principais aplicações são nas simulações, no diagnóstico, na supervisão e na análise de desempenho de sistemas. Nestas redes, o tempo é representado por durações associadas ao lugar (p-temporizadas) ou à transição (t-temporizadas), [Sifakis, 1977], [Tazza, 1987], [Ramchandani, 1974].

- **Redes de Petri Estocásticas:** é incluído um tempo aleatório associado ao disparo de uma transição [Florin and Natkin, 2000]. Incertezas nos instantes de execução de eventos do sistema são consideradas, associando-se funções de probabilidade para a determinação de sua execução. Estas redes podem ser aplicadas em sistemas cujo tempo para a ocorrência dos eventos não são bem definidos, como por exemplo, o tempo de falha de uma máquina. A definição do tempo nestas redes é definida através de uma distribuição que segue uma lei exponencial permitindo atribuir à rede um processo Markoviano equivalente.

- **Redes de Petri Híbridas e Redes de Petri Contínuas:** em uma rede de Petri contínua, a marcação de um lugar e a taxa de disparo corresponde a uma variável contínua (número real não negativo). A marcação contínua é movida de um lugar para outro determinado por uma velocidade [David and H., 2004]. Esse modelo foi essencialmente desenvolvido para a modelagem e simulação de sistemas de produção híbrido que envolvia fluxos contínuos de produtos.

- **Redes de Petri a Objetos:** podem ser consideradas como uma extensão das Redes de Petri Predicado/Transição que aborda o conceito de paradigma orientado a objetos [Sibertin-Blanc, 1985]. A principal motivação do uso das Redes de Petri a Objetos se deve principalmente a capacidade de representar características de concorrência, paralelismo, fluxo de controle e estruturas de dados apresentados nos sistemas. As Redes de Petri a Objetos são apresentadas em detalhe na próxima seção.

2.2.4 Redes de Petri a Objetos

Antes de definir as Redes de Petri a Objetos, é importante considerar as principais características do conceito de orientação a objetos.

O paradigma orientado a objeto veio da necessidade de superar algumas deficiências do paradigma procedimental. A produção de um sistema na forma estruturada era feita a partir de procedimentos/funções que manipulavam um determinado conjunto de dados. No entanto, quando existiam grandes problemas a serem solucionados, a complexidade do programa se tornava tão alta a ponto de que a necessidade de uma simples alteração implicava em profundas alterações em tais procedimentos e funções que tinham acesso aos dados [Delamaro et al., 2007].

Para diminuir a complexidade dos programas e facilitar a manipulação de dados, surgiu o paradigma orientado a objetos. Consiste neste caso, em trabalhar com os dados, porém, de maneira isolada (encapsulada), ou seja, cria-se uma entidade (classe), onde os dados (atributos) e as funções/procedimentos (métodos) são agrupados para a realização de serviços (operações) sobre os atributos. Dessa forma, o objetivo do paradigma orientado a objeto é forçar o desenvolvedor a pensar numa abstração de mais alto nível em termos de classes e objetos.

No paradigma orientado a objeto, o conceito de objeto visa estruturar uma aplicação em torno de entidades [Bezerra, 2007]. Atributos e métodos compõem uma *classe de objetos* a partir da qual podem ser derivadas *subclasses* por meio da *herança*, o que torna possível herdar atributos e métodos de classes já definidas.

Objetos são caracterizados pela instanciação de uma classe, sendo esta portanto, apenas uma definição. Aos atributos dos objetos são atribuídos valores que são manipulados e executados por operações (métodos).

Também são importantes outros elementos que compõem o paradigma orientado a

objetos, como os conceitos de *encapsulamento*, *abstração, polimorfismo, modularidade* e *persistência*.

O encapsulamento confere aos objetos certa independência funcional e proteção contra acessos não autorizados. Dessa forma, os dados só podem ser alterados, através de métodos definidos pela/para a classe.

A abstração, por sua vez, enfatiza detalhes significantes e suprime outros em um dado momento, relacionando-se assim, com a interface do abjeto.

O conceito de polimorfismo refere-se ao fato de que uma mesma mensagem pode resultar em eventos diferentes quando recebida por objetos diferentes ou quando enviada com parâmetros diferentes.

A modularidade consiste em dividir um programa em partes como funções individuais a fim de reduzir a complexidade do código. Idealmente, um software deve apresentar alta coesão e baixo acoplamento. A coesão confere independência funcional ao software, já o acoplamento diz respeito à interconexão entre os módulos para a execução de tarefas.

Por último, a persistência (tempo de vida) é a propriedade de um objeto continuar a existir, mesmo após o seu criador não mais existir.

As Redes de Petri a Objetos [Sibertin-Blanc, 1985] baseiam-se na integração das Redes de Petri Predicado/Transição e do conceito de paradigma orientado a objetos. Neste tipo de rede, as fichas são consideradas como *n-uplas* de instâncias de classes de objetos e transportam verdadeiras estruturas de dados definidas como conjuntos de atributos de classes específicas. Nas transições, por sua vez, são associadas pré-condições e ações, que respectivamente, atuam sobre os atributos das fichas e modificam seus valores [Cardoso and Valette, 1997].

Em uma Rede de Petri a Objetos, os lugares estão associados às classes; as transições são associadas as operações que atuam sobre os atributos localizados nos lugares de entrada; e os objetos correspondem as fichas da rede. Dessa forma, a operação de uma determinada transição t só poderá ser executada por um objeto se este estiver num lugar de entrada de t [Cardoso and Valette, 1997]. As Redes de Petri a Objetos podem ser definidas formalmente como:

Definição 5: *Uma rede de Petri a objetos marcada pode ser definida pela 9-upla:*

$$N_0 =< P, T, C_{lass}, V, Pre, Post, A_{tc}, A_{ta}, M_0 >$$

onde:

- C_{lass} representa um conjunto finito de classes de objetos: para cada classe é definido também, um conjunto de atributos;
- P é um conjunto finito de lugares, cujos tipos são dados por C_{lass};
- T é um conjunto finito de transições;
- V é um conjunto de variáveis, cujos tipos são dados por C_{lass};
- Pre é a função lugar precedente, que a cada arco de entrada de uma transição, faz corresponder uma soma formal de elementos de V;
- $Post$ é a função lugar seguinte, que a cada arco de saída de uma transição faz corresponder uma soma formal de elementos de V;
- A_{tc} é uma aplicação, que a cada transição associa uma condição que envolve os atributos das variáveis formais associados aos arcos de entrada;
- A_{ta} é uma aplicação, que a cada transição associa uma ação que envolve os atributos das variáveis formais associadas aos arcos de entrada e atualiza os atributos das variáveis formais associadas aos arcos de saída;
- M_0 é a marcação inicial, que associa a cada lugar uma soma formal de objetos (n-uplas de instâncias de classes que pertencem a C_{lass}).

Um exemplo de rede de Petri a objetos é representada na Figura 26. O conjunto de classes é definido como:

$$C_{lass} = \{Produto, Pedido\}$$

A classe $Produto$ possui os atributos:

$$\begin{cases} nome = identificador; \\ codigo = integer; \\ preco = float; \end{cases}$$

A classe *Pedido* tem os seguintes atributos:

$$\begin{cases} codigo : integer; \\ custo : float; \\ tipo : identificador; \end{cases}$$

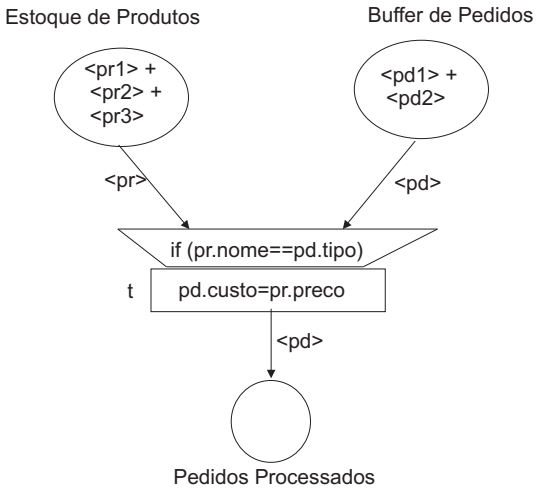


Figura 26: Modelo de Especificação para Venda de Produtos

A variável (classe) *pr* é do tipo *Produto* e a variável (classe) *pd* é do tipo *Pedido*. O lugar *Estoque de Produtos* é do tipo *Produto*, o lugar *Buffer de Pedidos* é do tipo *Pedido* e o lugar *Pedidos Processados* é do tipo *Pedido*. A marcação inicial M_0 é dada pelos objetos que se encontram nos lugares.

$$M_0 = \begin{bmatrix} < pr1 > + < pr2 > + < pr3 >, \\ < pd1 > + < pd2 >, \\ 0 \end{bmatrix}$$

Nos lugares *Estoque de Produto* e $Buffer\ de\ Pedidos$, os objetos (fichas) possuem valores de atributos. Por exemplo, para o objeto $pr1$, os valores de atributos podem ser dados por:
Produto $pr1$;
$nome$: *home theater*;
$codigo$: 123440;
$preco$: 278, 50;

E para o objeto $pd2$, valores dos atributos podem ser dados por:
Pedido $pd2$;
$codigo$: 123440;

custo : 00, 00;
tipo: home theater;

A definição detalhada que fixa as regras de sensibilização e disparo das transições das Redes de Petri a Objetos encontram-se em [Sibertin-Blanc, 1985].

As variáveis de arco de entrada de transição podem ser substituídas pelas variáveis do mesmo tipo dos objetos que se encontram nos lugares de entrada da transição. Entretanto, é necessário que seja satisfeita a pré-condição da transição. Assim, a transição removerá os objetos dos lugares de entrada da transição, alterará os valores de atributos de acordo com a ação existente na transição e, em seguida, produzirá um novo objeto com valores atualizados nos lugares de saídas correspondentes.

Pode-se notar, que na Figura 26, a transição t é sensibilizada pela marcação inicial. De fato, os valores dos atributos da variável *pr* associada ao arco que liga o lugar *Estoque de Produtos* à transição t podem ser substituídos pelos valores de atributos de um dos objetos que se encontram em *Estoque de Produtos*. Da mesma forma, os valores dos atributos da variável *pd* associada ao arco que liga o lugar $Buffer\ de\ Pedidos$ à transição t podem ser substituídos pelos valores de atributos de um dos objetos que se encontram em $Buffer$ *de Pedidos*.

Suponha que o par de objetos $(pr1, pd2)$ verifica a condição associada à transição t. Então, a transição pode ser disparada. No momento do disparo, o atributo *custo* do objeto $pd2$ será atualizado de acordo com a ação associada à transição e, no final do disparo, um novo objeto $pd2$ se encontrará no lugar *Pedidos Processados* com o valor do atributo *custo* atualizado.

Para os valores de atributos definidos em $pr1$ e $pd2$, a execução da rede é dada pelo disparo da transição t. Considere a marcação inicial (os objetos $pr1$, $pr2$ e $pr3$ no lugar *Estoque de Produtos* e $pd1$ e $pd2$ no lugar $Buffer\ de\ Pedidos$). Considere que os atributos dos objetos $pr1$ e $pd2$ sejam atribuídos, respectivamente, aos atributos das variáveis de arco *pr* e *pd* que, sensibilizam a transição t. Inicialmente, a pré-condição associada à transição t deve ser satisfeita para que a transição t seja executada. A pré-condição $(if(pr.nome == pd.tipo))$ verifica se os atributos *nome* da variável *pr* e *tipo* da variável *pd* possui os mesmos valores. Como a condição é satisfeita, a transição t é disparada. A ação associada a transição é então executada registrando então o valor do atributo *preco* da variável *pr* no atributo *custo* da variável *pd*. Após o disparo da transição t, um novo objeto $pd2$ com o valor do atributo *custo* modificado é produzido no lugar *Pedidos Processados*. A Figura 27 ilustra a Rede de Petri a Objeto com a nova marcação obtida

após o disparo da transição t.

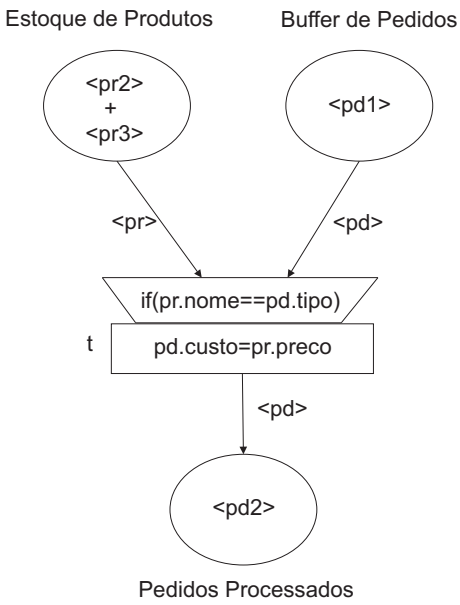


Figura 27: Exemplo de Disparo do Modelo de Especificação para Venda de Produtos

Para os demais objetos da rede, os disparos podem ser dados na mesma ordem.

2.3 Workflow: Abordagem Geral

Um sistema de Workflow é caracterizado pela capacidade de execução de um conjunto de processos de negócios. É importante garantir a qualidade dos processos de negócios, o que se faz por meio do gerenciamento do Workflow que tem como papel tratar casos diferentes de maneira eficiente e efetiva [Aalst and Hee, 2002].

Um caso, na grande maioria das vezes, é gerado por clientes externos, mas, também, pode-se dizer que é possível um caso ser criado por um departamento interno específico de uma organização, obtendo-se dessa forma, um cliente interno.

Os casos possuem uma vida limitada e são marcados pela execução de tarefas em ordem específica, ficando a cargo do Workflow definir quais *tarefas* serão executas e em que ordem.

Para a execução de uma tarefa em uma determinada ordem, é necessário identificar condições que correspondem a dependências causais entre as tarefas e que podem ser válidas ou não. Dessa forma, cada tarefa tem associada pré e pós condições: as pré-condições necessitam ser validadas antes da execução da tarefa, as pós-condições, por sua vez, são validadas após a execução da tarefa.

Em um mesmo processo de Workflow, vários casos podem ser tratados. Assim, uma mesma tarefa pode executada por vários casos. Uma tarefa que precisa ser executada por um caso especifico é chamada de *item de trabalho*. Cada um destes é executado por um recurso, como, por exemplo, uma impressora, um fax, caracterizando-se, portanto, como uma atividade.

Um processo é caracterizado por tarefas e condições, em que é definido, o ciclo de vida, marcando-se o início e a conclusão de um caso. O ciclo de vida define a rota do caso, e determina a ordem de execução das tarefas.

Na representação de processos de Workflow por meio de modelos, estes devem ser capazes de representar as seguintes situações:

- Atividades e suas sincronizações;
- Tarefas e recursos determinados para realizar atividades;
- Regras de negócios;
- Tratamento de exceções;
- Aspectos temporais (*deadline*, durações).

Uma Rede de Petri, quando usada para representar um processo de negócio é denominada *Workflow-Net* (WF-Net).

Uma Workflow-Net apresenta apenas um lugar de início (*start*) e um de término (*end*). O conceito de *soundness* é um critério de verificação de correção, ou seja, uma Workflow-net é *sound*, se, e somente se, três condições são válidas [Aalst and Hee, 2002]:

- Para cada ficha colocada no lugar de início, apenas uma aparecerá no lugar de término;
- Quando uma ficha aparece no lugar de término, os demais lugares estão vazios;
- Não deve haver nenhuma transição não-viva. Para toda a transição (tarefa), é possível evoluir da marcação inicial até a marcação que sensibiliza tal transição.

Alguns elementos, como caso, processo, tarefa etc., presentes em sistemas de Workflow devem ser considerados na representação usando as Rede de Petri [Aalst, 1998].

Processo

Um processo especifica as tarefas e define a ordem em que elas são executadas. Em uma Rede de Petri, um processo tem um lugar de entrada, sem arcos de entrada, e um lugar de saída, sem arcos de saída. Os lugares (componentes passivos) em uma Rede de Petri representam as condições e as transições (componentes ativos), as tarefas. Considere-se por exemplo, a especificação de um processo de tratamento de reclamações como mostrada na Figura 28.

Inicialmente, uma reclamação é registrada. O cliente que registrou a reclamação e o departamento afetado pela reclamação são contactados. O cliente é então questionado a fim de se obter maiores informações e o departamento é informado da queixa e pode ser questionado pelas suas reações iniciais. Estas duas situações (tarefas) podem ser executadas em paralelo e, posteriormente, os dados são recolhidos para que uma decisão seja tomada. Assim que a decisão é tomada, duas situações podem ocorrer: um pagamento de compensação é feito e ou uma carta é enviada. Por último, a reclamação é arquivada.

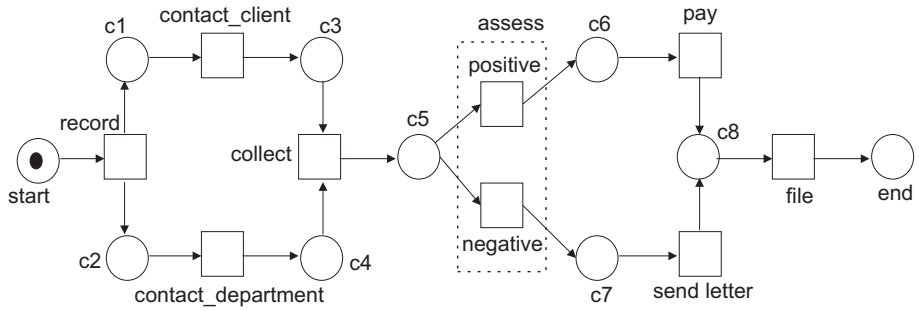


Figura 28: Processo de Tratamento de Reclamações

Na Figura 28, cada uma das tarefas *record*, *contact_client*, *contact_departament*, *pay* e *file* é representada na Rede de Petri por uma transição. A avaliação de uma reclamação corresponde às transições *positive* e *negative*, as quais representam uma decisão, positiva ou negativa, respectivamente .

Os lugares *start* e *end* indicam o início e fim do processo modelado. Os demais lugares da rede referem-se a condições que não podem ser cumpridas por todos os casos em andamento. Estas condições desempenham duas funções importantes: assegurar que as tarefas avancem na ordem correta e estabelecer o estado do processo. Considere-se por exemplo, o lugar c8: este lugar é responsável por indicar o estado de que a reclamação será arquivada quando for completamente resolvida.

Os casos são representados pelas fichas presentes nas redes. No lugar *start* existe uma ficha indicando a presença de um caso. Se a transição *record* é disparada, duas fichas (uma em c1 e a outra em c2) representam o mesmo caso. Quando um caso é tratado, o número

de fichas pode variar. Mas, a quantidade de fichas que representa um caso particular é sempre igual ao número de suas condições que devem ser satisfeitas. No lugar *end* deverá haver uma ficha, quando que o caso foi concluído.

Cada processo deve cumprir dois requisitos: em qualquer momento, poder atingir um estado, por meio da execução de tarefas; e, quando existir uma ficha em *end*, verificar se as demais desapareceram do restante dos processos. Dessa forma, é possível garantir que todo caso que se iniciou no lugar *start*, será completado corretamente e finalizado no lugar *end*.

As fichas que se referem a casos particulares são separadas pelo sistema de gerenciamento do Workflow. Como estas fichas pertencem a casos diferentes e não podem influenciar uma à outra, pode-se gerar uma cópia para cada caso, que, assim, terá seu próprio processo. Ao se representar essa característica em uma Rede de Petri, podem ser usadas as Redes de Petri de Alto Nível por exemplo, em que cada ficha possui uma identificação, a fim de identificar a que caso a ficha se refere.

Roteamento

Em um sistema de Workflow, as tarefas podem se opcionais. Podem existir tarefas que só precisam ser executadas por um certo número de casos e pode variar a ordem com que as tarefas são executadas dependendo do caso. É possível determinar a rota de um caso ao longo de várias tarefas e estabelecer quais tarefas necessitam ser executadas e em que ordem.

No roteamento são consideradas quatro tipos básicos de rotas:

- **Seqüencial:** uma tarefa é executada após a outra, havendo uma dependência entre elas;
- **Paralela:** duas tarefas podem ser executadas simultaneamente ou em qualquer ordem, sem que o resultado de uma interfira no resultado de outra;
- **Condicional ou rota alternativa:** existe uma escolha entre duas ou mais tarefas;
- **Iterativa:** a mesma tarefa é executada várias vezes.

Tais construções são apresentadas na Figura 28, em que por exemplo, a tarefa *record* representa um paralelismo; as tarefas *negative* e *positive* mostram uma rota alternativa ou condicional; e por fim, a tarefa *pay* ilustra uma rota sequencial.

Para os casos em que a execução de uma tarefa se dê de forma iterativa, esta pode ser realizada de duas maneiras, conforme mostra a Figura 29:

- Uma tarefa deve ser executada repetitivamente até que o resultado do teste subseqüente seja positivo (”repeat.....until.....”), como mostra o modelo Rota Iterativa (1) da Figura 29, em que *task2* deve ser executada pelo menos uma vez.
- Para que uma tarefa seja executada pelo menos uma vez, usa-se uma estrutura do tipo ”while....do....”como mostrada no modelo de Rota Iterativa (2) da Figura 29. Após a finalização de *task1*, será determinado se *task2* será ou não executada, o que permite passar de *task1* para *task3* diretamente, sem executar *task2*.

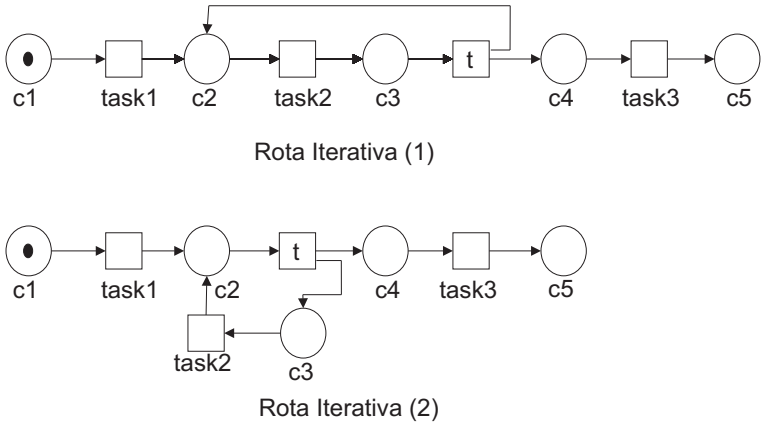


Figura 29: Variações na Modelagem de Rotas Iterativas

Acionamento

Um processo é definido para lidar com uma determinada categoria de casos, podendo manipular muitos casos individuais, não sendo portanto, uma tarefa específica para um caso particular.

Quando um caso está sendo executado por um processo, as tarefas são executadas para este caso específico. Um item de trabalho é caracterizado pela combinação de um caso e uma tarefa que está pronta para ser realizada. A atividade, por sua vez, refere-se ao desempenho atual de um item de trabalho, uma vez que o item de trabalho está sendo executado, este é transformado em uma atividade. Em uma Rede de Petri, uma tarefa, corresponde a uma ou mais transições , um item de trabalho a uma transição sensibilizada e uma atividade, ao disparo de uma transição.

Para que uma tarefa seja realizada são necessários alguns requisitos, além da marcação em que o caso se encontra. Quando uma tarefa é executada por uma pessoa, esta deve

tomar posse da atividade antes de iniciá-la, ou seja, um item de trabalho só é realizado se ele for inicializado.

Para que certos ítens de trabalho se transformem em uma atividade eles precisam ser acionados (*triggering*). Um item de trabalho que é executado sem a intervenção de um recurso não necessita de acionamento. Há três tipos de acionamento:

- uma iniciativa dos recursos (um funcionário pegando um item de trabalho);
- um evento externo (como a chegada de uma mensagem);
- um sinal de tempo (geração de uma lista de ordem às 6 horas, por exemplo).

3 Teste Funcional

Este capítulo tem o propósito de apresentar os diversos trabalhos já realizados sobre testes funcionais.

3.1 Teste de Software

Atualmente, verifica-se um enorme interesse por parte de equipes de desenvolvimento de softwares em produzir sistemas incluindo o quesito qualidade [Pressman, 2001]. Para isso, tem-se dado uma extrema importância à atividade de teste que contribui por construir sistemas com maior qualidade e confiabilidade sem, no entanto, representar altos custos para as empresas [Delamaro et al., 2007]. Em resposta a essas necessidades, torna-se primordial que, durante o desenvolvimento de um software, incluam-se métodos, ferramentas e técnicas que auxiliem a atividade de teste.

Testes visam validar e verificar sistemas. O trabalho de verificação consiste em garantir que o software esteja correto durante todo o seu ciclo de vida. Já a atividade de validação garante que o sistema, ao final do projeto, satisfaça aos requisitos iniciais propostos pelo cliente [Pressman, 2001].

Os testes de softwares são aplicados em várias fases de desenvolvimento e, dependendo do contexto de desenvolvimento em que o software se encontre, os testes podem ter propósitos diferentes. Em geral, os testes podem ser classificados em:

- **Teste de unidade:** o programa é dividido em módulos, com o objetivo de encontrar erros de lógica e de implementação. Nesta etapa, cada componente, como classes e métodos, por exemplo, é testado de forma isolada dos demais através da análise de dados de entrada pré-determinados.
- **Teste de integração:** é caracterizado por buscar erros na interação dos módulos. Neste passo são verificadas as interfaces dos componentes e assegura-se que eles trabalhem de forma adequada.

- **Teste funcional:** assegura ao sistema que as funções especificadas estão funcionando adequadamente pelo sistema integrado.
- **Teste de sistema:** é caracterizado por identificar erros de funções e de desempenho no sistema já integrado ao ambiente operacional.
- **Teste de aceitação:** é realizado junto ao cliente a fim de confirmar se os requisitos estão de acordo com os especificados pelo usuário.

Na atividade de teste de software, nenhuma técnica de teste é completa, pois nenhuma garante total qualidade e confiança. Dessa forma, diferentes métodos de teste são usados nas diversas etapas de desenvolvimento de software.

3.2 Teste Funcional

O teste funcional, também conhecido como teste de caixa-preta ou ainda como Teste de Aceitação, é um procedimento empregado para a projeção de *casos de teste* (seleção de um dado de teste que é executado no programa onde a saída gerada é comparada com a saída esperada) em que são verificadas as funcionalidades do software. Nesta etapa do teste, não são incluídas informações de codificação, sendo considerado apenas a análise das entradas e saídas do software [Pressman, 2001], [Summerville, 2003]. A Figura 30 ilustra o teste funcional.

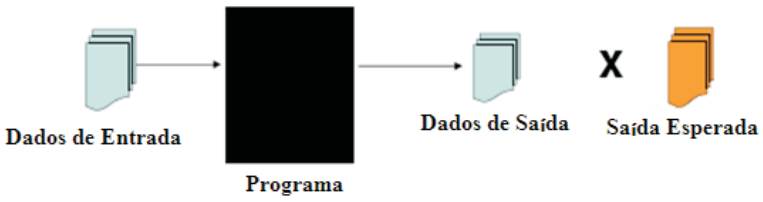


Figura 30: Teste Funcional

Durante o teste funcional alguns critérios podem ser estabelecidos. Entre estes critérios, destacam-se [Souza, 2000]:

- **Particionamento em Classes de Equivalência:** o domínio de entrada é dividido em classes (partições): válidas e inválidas. Em uma classe, os dados que ali existem possuem a mesma capacidade de revelar falhas. Assim, no particionamento, o objetivo é gerar casos de teste que cubram todos os erros e que reduzam os casos de teste necessários.

- **Análise do Valor Limite:** testam-se os limites de cada classe de equivalência, o que serve como complemento para o critério de particionamento em classes de equivalência. Os testes são selecionados a partir da fronteira de classes, considerando-se os locais com maior evidência de erros.

- **Grafos de Causa e Efeito:** estuda-se a combinação das condições de entrada do programa. As condições de entrada (causa) e as ações (efeitos) são combinadas e dão origem a um grafo que determinará os casos de teste.

- ***Error Guessing*:** os possíveis erros são atribuídos a uma lista e, com base nesta, os casos de teste são gerados.

O teste funcional é relatado desde os anos 70. Métodos como Análise e Projeto Estruturado [Gane and T., 1983] eram usados na especificação de sistemas para verificar a satisfação de requisitos funcionais [Myers et al., 2004], [Hetzel and Hetzel, 1991], [Delamaro et al., 2007], [Beizer, 1990].

Roper [Roper, 1994] estuda e compara os vários aspectos e características dos métodos de análise e de projeto de teste funcional. A atividade de teste funcional tem sido aplicada até mesmo para a sincronização de protocolos usando autômatos [Yamada et al., 2000].

Em sistemas de tempo real, o teste funcional é planejado de forma que entradas com valores de tempo pré-definidos sejam consideradas para o teste da modelagem do comportamento desses sistemas [Koné, 2000]

Testes de software costumam ter um alto custo. Por isso, técnicas de geração de testes baseadas em modelos são usadas para criar testes em sistemas. Mesmo assim, o sistema, baseando-se em requisitos deve ser modelado para a geração de testes. A geração de casos de teste não estava associada com requisitos até que [Tahat et al., 2001] apresentaram uma técnica de geração de teste baseada em requisitos. O software, nesta técnica, é especificado como um conjunto individual de requisitos. A partir da identificação destes requisitos, um modelo de sistema é gerado automaticamente com as informações mapeadas. O modelo de sistema é então usado para gerar automaticamente casos de teste relacionados aos requisitos. Várias estratégias de geração de testes são possíveis de serem aplicadas. Isso trás como vantagem, a redução do número de casos de teste e o aumento da qualidade do *suite* de testes.

É apresentado em [Clarke and Lee, 1997] um *framework* considerando-se o elemento tempo em sistemas de tempo real. Os testes derivam de especificações de eventos de entrada e saída na execução de um sistema. A técnica de teste usa uma especificação formal

gráfica e processos algébricos em tempo real – *Algebra Communicate Shared Resources* (ACSR) para representação de testes e modelos de processos.

Beizer [Beizer, 1990] explora o uso de grafos para a especificação de técnicas de testes funcional para a avaliação de fluxos de controle, fluxos de dados, domínios. voltados para softwares procedimentais.

Uma outra proposta em [Jee et al., 2005] foi apresentada usando Diagramas de Blocos. Neste trabalho, é proposta uma técnica que testa as funções do Diagrama de Bloco sem a geração do código do programa .

Korel e Schroeder [Schroeder et al., 2002] apresentaram em seu trabalho, uma forma de diminuir a quantidade de dados de teste funcional a partir da análise automatizada que identifica o relacionamento entre as saídas e entradas do programa.

Em [Zeng et al., 2001] operadores aritméticos da lógica booleana são usados para a geração de teste funcional. O resultado experimental é então comparado com outros métodos da mesma área.

As ferramentas *capture-replay* ou gravação-execução (GE) são ferramentas que permitem a criação, a execução e a verificação dos resultados de testes automáticos para sistemas com interface gráfica. Entre essas ferramentas podem ser citados a *WinRunner*. Existem outras ferramentas que testam a funcionalidade de sistemas de forma automática sob formatos não gráficos, por exemplo, a *stored procedures PL-SQL* [Correia and Silva, 2004].

Fummi e Sciuto [Fummi et al., 1995] desenvolveram um novo algoritmo de geração de teste com uma nova métrica de cobertura para o teste funcional denominado de *bit de cobertura* para modelos de sistemas seqüenciais para a descoberta de sequências de teste. Tal métrica propõe a identificação de erros em projetos de maneira mais efetiva que outros padrões de métricas de cobertura existentes.

Meinke propôs um método geral usando funções de aproximação aplicadas no teste funcional e apresentou alguns resultados práticos obtidos para a automação do teste por meio do emprego de algoritmos, abordando funções polinomiais [Meinke, 2004].

[Blom et al., 2004] abordaram a geração automática de casos de teste a partir de modelos de autômatos formais obtidos do sistema. Demonstraram também como gerar de forma ótima, casos de teste, ou seja, produzir casos de teste que podem ser executados com rapidez. Foram usados vários critérios de cobertura capazes de gerar casos de teste a partir de propostas de teste formuladas manualmente ou automaticamente.

[Rajan, 2006] definiu uma métrica de cobertura estrutural aplicadas diretamente sobre requisitos formais de software. A técnica provê medidas de exercitar um suíte de testes para um conjunto de requisitos. O critério de cobertura estrutural sobre os requisitos, é formalizado no contexto das propriedades da Lógica Temporal Linear (*Linear Time Temporal Logic* – LTL).

Algoritmos baseados em *etapas de avaliação, estado de avaliação* e *ordem de avaliação* foram propostos por [Kervinen and Virolainen, 2005] para a execução do teste funcional. Os algoritmos são usados para comparar o número de etapas de testes solicitadas para detectar erros que estão infiltrados em sistemas.

Foi desenvolvido um simulador de erro capaz de analisar as descrições em VHDL. O ambiente do simulador é baseado em simuladores comerciais de VHDL. Todos os componentes do ambiente de simulação são automaticamente construídos a partir da especificação em VHDL [Fin and Fummi, 2000].

[Reid, 1997] propôs um experimento com o objetivo de comparar a eficiência entre as técnicas de particionamento de equivalência e a análise do valor limite. Os resultados mostraram que, utilizando-se todos os possíveis valores de entrada que cobriam uma técnica de teste e também todos os possíveis valores de entrada que poderiam gerar falhas no propósito de alcançar valores absolutos para determinar o desempenho das técnicas de teste, a análise de valor limite foi mais eficiente em relação ao particionamento de equivalência, conseguindo detectar uma maior probabilidade de anomalias.

Na mesma direção, Bowen *et al* [Vilkomir et al., 2003] realizaram uma análise comparativa entre os critérios de teste baseando-se em fluxos de controle no intuito de investigarem a variação da eficiência de cada um dos critérios.

Outros critérios surgiram como o Teste Funcional Sistemático, *Sintax Testing, State Transition Testing e Graph Matrix* [Linkman et al., 2003].

O teste funcional encontra inconsistências que ocorrem geralmente durante a especificação de requisitos e, desse modo, a execução das principais partes do programa nem sempre são garantidas, o que gera maiores dificuldades na escolha dos critérios de teste. Outro fato importante, é que a especificação do programa é feita de maneira descritiva sem o uso, por exemplo, de linguagens de modelagem, gerando requisitos de teste pouco seguros devido à informalidade existente.

É sabido que os erros, detectados logo no início do desenvolvimento de sistemas são mais simples e mais fáceis de serem removidos e isso gera um menor custo e contribui

para produzir sistemas de maior qualidade e confiabilidade. Assim, a aplicação do teste funcional se torna crucial, uma vez que requisitos do sistema são avaliados em testes auxiliados por linguagens de modelagens como, por exemplo, UML, Redes de Petri, Diagramas de Fluxo de Dados, etc.

Muitas vezes, a atividade de teste funcional, mesmo quando realizada em empresas de desenvolvimento de sistemas, é feita de maneira informal. Na maioria dos casos não se tem documentos disponíveis sobre como estes testes são realizados.

Outro ponto importante a ser considerado é que, na maioria dos casos, os testes funcionais são feitos de forma manual. Desse modo, o tipo de teste a ser realizado está mais direcionado aos interesses de quem vai executá-lo, o que, em alguns casos, impede que todos os requisitos sejam testados.

3.3 Teste Funcional Baseado em Modelos

Um modelo de software permite que menos tempo seja gasto com a tarefa de identificar o que o sistema deve fazer, ficando, a maior parte, para verificar se os valores que o sistema produz estão ou não corretos. Assim, uma especificação clara e concisa define de forma eficaz o escopo de desenvolvimento e, conseqüentemente, os testes.

A seguir, são relatados alguns trabalhos envolvendo algumas linguagens de modelagem.

3.3.1 Máquinas de Estados Finitos

Belli et al. [Belli et al., 2001] por meio das Máquinas de Estados Finitos e expressões regulares equivalentes propõem a geração de um modelo de falha que, posteriormente, é considerado na modelagem de sistemas. São usados algoritmos para a produção e seleção de casos de teste para a aplicação do Teste Funcional.

No trabalho de Hong et al [Hong et al., 1995], as Máquinas de Estados Finitos foram usadas para especificar o comportamento de uma classe, construindo-se assim uma técnica para a geração de casos de testes. A técnica restringe-se a primeiramente transformar a Máquina de Estados Finitos em um grafo de fluxo para proporcionar a identificação dos fluxos de dados que existem. A interação que existe entre dados (atributos) e operações é modelada usando-se as Máquinas de Estados Finitos com o auxilio da *Class State Machine* (CSM) que gera casos de testes automaticamente, baseando-se nos fluxos de dados. Neste

trabalho, a existência de roteiros (paralelo e iterativo) não foi tratada, limitando o trabalho apenas para o modelo sequencial.

Seguindo a mesma linha de trabalho, [Kim et al., 1994] propõem uma extensão das Máquinas de Estados Finitos para modelar o comportamento de softwares tendo por base o fluxo de dados a partir da seleção de cenários de execução, não considerando versões voltadas para softwares orientados a objetos.

Máquinas de Estados Finitos não são capazes de suportar a modelagem formal de certas características como, por exemplo, os fluxos de dados que existem. Uma técnica de teste proposta por [Fantinato and Jino, 2000] combina fluxos de controle e fluxos de dados. Para isso, uma extensão das máquinas de estados finitos é desenvolvida para prover mecanismos que consigam modelar tais fluxos. Entretanto, o trabalho se limita a apresentar um modelo simples que não possui paralelismo ou iterações.

As Máquinas de Estados Finitos, como o próprio nome diz, possuem um conjunto de estados finitos, considerado um fator limitante na modelagem das classes de sistemas existentes uma vez que podem resultar em modelos excessivamente grandes [Souza, 2000]. Dessa forma, foram propostas outras extensões como Estelle e Statecharts, embora esta última linguagem não deixa de ter também um conjunto finito de estados [Budkowski and Dembinski, 1987].

3.3.2 Statecharts

[Sugeta, 1999] desenvolveu uma ferramenta de teste funcional denominada *Proteum-RS/ST* baseada no critério do teste de mutação para validar especificações descritas em Statecharts para Sistemas Reativos.

Outro trabalho, proposto por [Belli et al., 2001], sugere a geração de casos de teste baseados em Statecharts para a especificar o comportamento do sistema, a estrutura hierárquica e os mecanismos de comunicação. Essa técnica baseia-se na verificação do comportamento de um sistema e nas ações do usuário. A partir desta análise, um modelo de falhas é usado para causar um estado errôneo que seria provocado por usuários. Posteriormente, o modelo de falhas é representado por Statecharts que, em seguida são convertidas para expressões regulares e as expressões regulares estendidas para carregar os processos de teste.

Hartmann *et al* [Hartmann et al., 2000] apresentam uma metodologia de geração de casos de teste definindo um comportamento dinâmico de componentes de software usando

UML com Statecharts. Assim, descrevem como os casos de teste são gerados a partir da modelagem destes componentes de software.

Outro trabalho, que segue a mesma linha, foi apresentado por [Ali et al., 2007] em que é desenvolvida uma técnica que combina os Diagramas de Colaboração da UML e as Statecharts para a geração automática de um modelo de teste, denominado modelo *SCOTEM*, a fim de detectar falhas que ocorrem durante a integração de classes.

A vantagem em utilizar Statecharts é a sua simplicidade da notação que permite que elas sejam aplicadas de maneira eficiente na modelagem da grande maioria dos sistemas existentes. Entretanto, quando grandes sistemas são modelados, sua complexidade aumenta, por causa da estrutura hierárquica que a linguagem possui que é mais adequada para linguagens estruturadas do que para linguagens orientadas a objetos. Outro fator limitante é a falta de uma semântica operacional formal que explique os mecanismos de execução e simulação no modelo, ou seja, a forma de execução depende da ferramenta de simulação e não da própria linguagem.

3.3.3 Redes de Petri

Simão [Simão, 2000] desenvolveu a Proteum-RS/PN em que explora o critério do teste de mutação [Willians, 2000] no contexto do teste funcional em Sistemas Reativos, utilizando Redes de Petri. Outras ferramentas com a mesma característica foram desenvolvidas utilizando-se no entanto, Máquinas de Estados Finitos [Fabbri, 2001] e Statecharts [Sugeta, 1999].

[Zhu and He, 2000a] apresentam uma metodologia de teste de redes usando Redes de Petri de alto nível, baseando-se no teste aplicado em sistemas concorrentes. Assim, algumas estratégias de teste foram investigadas e avaliadas.

O problema de testar o comportamento de sistemas de tempo real considerando-se o fator tempo foi abordado por [Wegener and Grochtmann, 2004]. A idéia consiste em usar um projeto físico de um software e desempenhar o teste funcional sobre o mesmo. É considerada a modelagem do sistema usando SA/SD-RT que, em seguida, é traduzido em uma Rede de Petri de alto nível para obter uma árvore de alcançabilidade, com a representação do tempo. Com base nestes modelos, critérios de seleção de testes são usados para solucionar o problema.

Uma metodologia de teste para sistemas de tempo real também pode ser estendida para um modelo de Redes de Petri com forte controle sobre o tempo. O trabalho baseia-se

no estudo da equivalência de dois processos que são usados para comparar dois sistemas distintos. Aplicando-se o conceito de bissimulaçao, verifica-se a similaridade entre os comportamentos dos processos. Assim, é proposto o teste de equivalência para um modelo de rede de Petri [Martena et al., 2002].

[Wang and Liu, 1999] propõem a utilização de um modelo de Rede de Petri simples (baixo nível) para a geração de classes de teste orientadas a objetos. Neste trabalho, considera-se que uma classe é como uma unidade básica de teste. No entanto, deve-se levar em consideração que em uma classe deve-se especificar a seqüência com que os métodos podem ser executados. Dessa forma, uma *Class Petri Net Machine* (CPNM) é desenvolvida para especificar a seqüência de métodos de uma classe, utilizando uma técnica de geração de casos de teste por meio das Redes de Petri.

Redes de Petri Coloridas

Buchs *et al*[Buchs et al., 1995] promoveram o uso de especificações formais estruturadas para gerar casos de teste em sistemas distribuídos para a avaliação da modelagem dos componentes de software que usam Redes de Petri Coloridas.

Sloane e Gelhot [Sloane and V., 2004] promoveram a integração de dispositivos médicos para tratar de pacientes. As ferramentas de Redes de Petri são utilizadas para testar todos os possíveis estados e transições entre dispositivos e/ou sistemas para a detecção de falhas. O objetivo do trabalho é a aplicação das Redes de Petri Coloridas para simular e validar um sistema de monitoramento central de pacientes que identifica doenças e situações de emergência.

Redes de Petri Predicado/Transição

As Redes de Petri Predicado/Transição têm sido exploradas [Reza et al., 2007] para testes em interfaces gráficas. É desenvolvido um modelo de método de teste para testar a representação estrutural de interfaces GUI.

Uma outra aplicação [Zhu and He, 2000b] propõe que Redes de Petri de alto nível, com destaque para as Redes de Petri Predicado/Transição, sejam definidas como um programa concorrente abstrato. Assim, é apresentada uma teoria de testes para essas redes baseada em testes de sistemas concorrentes.

Redes de Petri a Objeto

Em [Barbey et al., 1996] as Redes de Petri a Objetos Cooperativas são usadas para a especificação de sistemas orientados a objetos. O teste aplicado considera um número possível de sequências de chamadas de métodos de uma classe. É determinado então um conjunto de testes, que verifica todos os possíveis comportamentos de uma classe.

Foi descrita a geração automática de testes para sistemas representados por Redes de Petri Orientada a Objetos Concorrentes (CO-OPN), que é um formalismo utilizado para sistemas concorrentes. Dessa forma, a SATEL (Linguagem de Teste Semi-Automática) baseia-se nas intenções de quem irá testar o sistema. Essas intenções são, então, convertidas em variáveis usadas para produzir os casos de teste, a partir da especificação do sistema em CO-OPN em que este é usado como referência para o cálculo do oráculo [Barbey et al., 2008].

Desel *et al* [Desel et al., 1997], mostram o desenvolvimento de uma técnica de geração de casos de testes para validação de redes de Petri de alto nível usando grafos de causa-efeito.

3.4 Teste Funcional para Softwares Orientados a Objetos e Apoiado pela UML

No teste funcional orientado a objetos, o conceito de ocultamento de ocultamento da informação, apesar de se mostrar eficiente, por outro lado, também constitui um obstáculo para o teste de software devido ao fato de que para a realização de testes, uma série de relatórios contendo informações sobre o estado dos objetos é necessário dependendo da linguagem de programação usada.

O conceito de herança, presente na programação orientada a objeto, é importante também no aspecto relacionado à reutilização através do compartilhamento de características entre classes. No entanto, esse mecanismo enfraquece o encapsulamento, o que pode contribuir para ocasionar defeitos nas funcionalidades do software.

A herança múltipla é caracterizada por uma subclasse que herda de duas ou mais superclasses que contêm características comuns como, por exemplo, atributos e métodos. Binder [Binder, 1995] argumenta em seu trabalho que a herança múltipla gera mudanças tanto na semântica quanto na sintática, o que pode gerar a ocorrência de alguns defeitos, dentre os quais destacam-se:

- uma mudança realizada na superclasse pode gerar um impacto na subclasse, provo-

cando um funcionamento inadequado.

- quando ocorre a presença de herança repetida, ou seja, quando uma superclasse aparece mais de uma vez na hierarquia de herança, a atividade de teste se torna mais difícil pois existe um número de características que devem ser removidas ou renomeadas.

Como um exemplo de problemas em que o paradigma orientado a objeto pode trazer considere-se uma classe com herança múltipla possuindo cinco superclasses que contribuem na hierarquia de herança e ainda têm vários métodos polimórficos. Neste caso, é preciso que muito esforço seja gasto para garantir que todas essas superclasses operem de forma eficiente. Na presença do polimorfismo [Meyer, 1997] e do acoplamento dinâmico (ligação/associação), um número grande de caminhos precisam ser testados, o que pode ser dificultado pelo encapsulamento, que diminui a visão dos estados dos objetos [Delamaro et al., 2007].

Outro ponto a ser considerado na atividade de teste é o conceito de reuso. Quando um componente de software é reutilizado, este deve ser passado por uma rigorosa bateria de testes, mas, independentemente disso, são indispensáveis novos testes em relação ao novo contexto no qual esse componente se encontrará.

No paradigma orientado a objeto podem existir classes sem nenhuma implementação, oferecendo somente uma interface como é o caso das classes abstratas, o que é vantajoso no contexto do reuso [Binder, 1995]. No entanto, aplicação de testes nestas classes só é possível com a especialização destas, ou seja, gerando-se classes concretas. Mas, isso pode ser difícil quando se tem um método concreto chamando um método abstrato, por exemplo.

Já quanto às classes genéricas, não se pode dizer que elas são sempre abstratas, mas elas dão ao reuso um fator importante para a ligação dinâmica. Em classes genéricas, podem existir atributos e métodos com objetos de tipos específicos, porém essa característica afeta a atividade de testes das classes [Smith and Robson, 1990]. Entre os problemas que podem ocorrer estão:

- ao ser testado, o parâmetro genérico de uma classe genérica precisa ser substituído pelo parâmetro de tipo específico e a dificuldade está em escolher um tipo correto para o teste.

- independentemente da classe genérica ter sido testada, um novo teste deve ser reali-

zado quando novas classes são geradas a partir da classe genérica, a fim de garantir que novos erros não foram inseridos.

O polimorfismo é caracterizado por permitir que duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação (assinatura) mas comportamentos distintos [Meyer, 1997]. O uso do polimorfismo em programas pode gerar o funcionamento não correto do software. Em [], são demonstrados alguns problemas que podem existir com o uso do polimorfismo e que, consequentemente, podem afetar os resultados dos testes:

- **indecisão:** o polimorfismo permite o relacionamento entre objetos de diferentes classes. Dessa forma, é impossível identificar qual método será executado em tempo de execução, ou seja, se será o método da superclasse ou da subclasse, por exemplo.
- **extensão:** ao se testar um método, seu comportamento é verificado com a execução de vários parâmetros com diversos valores. A atividade de teste, deveria ser capaz de garantir que todo acoplamento dinâmico fosse executado pelo menos uma vez.

O polimorfismo é amplamente discutido em [Supavita and Suwannasart, 2005]. A troca de mensagens realizadas entre objetos de classes é modelada levando-se em consideração o polimorfismo que pode existir. Assim, classes de teste são geradas para testar a interação polimórfica em cenários bem definidos, garantindo que subclasses sejam testadas como instâncias de suas superclasses.

Além desses problemas discutidos até agora, outros problemas podem ocorrer, por exemplo, na interação entre objetos [Binder, 1995]. Quando uma classe deseja solicitar serviços de outras classes, estas trocam mensagens entre si e realizam as operações necessárias para atender ao pedido feito. Entretanto, na comunicação entre estes objetos destas classes existe uma seqüência em que as mensagens são enviadas a outros objetos. É necessário identificar se a ordem em que estas mensagens são enviadas está correta, pois estas trocas de mensagens entre classes promovem a realização de operações que, conseqüentemente, mudam o estado do objeto. O envio não correto de mensagens pode, por exemplo, levar o objeto a um estado indefinido ou não apropriado, o que pode ocasionar defeitos no software.

[Georgieva and Gancheva, 2003] apresentaram um algoritmo que direciona a construção de casos de teste funcional para uma classe de programas orientados a objetos. O algoritmo pode ser modificado de forma que casos de teste sejam produzidos com um

alto nível de cobertura. A entrada do algoritmo considera a especificação da classe que é produzida para guiar processos de projeto assim como processos de teste em sistemas orientados a objetos.

A utilização de *cenários* (casos de uso) de um software orientado a objetos são utilizados para identificar a funcionalidade e o comportamento de sistemas. A partir destes cenários, os testes são formalizados utilizando-se a linguagem Statecharts para a geração de casos de teste [Ryser and Glinz, 1999].

Figueiredo e Machado [Figueiredo and Machado, 2004] apresentam um método para a geração automática de casos de teste para sistemas baseados em Agentes Móveis. Dada uma especificação formal para um determinado sistema baseado em agentes móveis, casos de teste são gerados automaticamente e são utilizados para testar o sistema, adotando-se para isso, o formalismo das Redes de Petri para especificação de sistemas com o qual o método de teste irá trabalhar assim como, também, Diagramas da UML para modelagem deste método de teste e demais características do sistema.

Os Diagramas de Atividades da UML 2.0, em [Mingsong et al., 2006], foram adaptados com o formalismo oferecido pelas Redes de Petri, incluindo fluxo de controle e o fluxo de dados. Nesta abordagem, casos de teste são gerados a partir da especificação de um sistema orientado a objetos. Entretanto, é considerado neste trabalho apenas os caminhos simples do Diagrama de Atividades, descartando os caminhos iterativos, a concorrência e a sincronização para a geração de casos de teste.

Em [FIGUEIREDO et al., 2007] é proposta a automação de dois métodos de teste que, a partir da especificação UML do sistema, permitem a geração, a execução e a análise de resultados de casos de teste funcional e de integração funcional de componentes de software. Para tanto, é feito um aperfeiçoamento destes métodos no que se refere a aspectos de automação, assim como o desenvolvimento de uma ferramenta de suporte chamada *SPACES*.

Quanto a teste funcional envolvendo a UML, em [Kim et al., 1999] classes de teste são geradas a partir da aplicação de diagramas de estados da UML. Um conjunto de critérios de cobertura é proposto com base no fluxo de dados e de controle. Inicialmente, o fluxo de controle é identificado através da transformação dos Diagramas de Estados da UML para as Máquinas de Estados Finitos Extendidas. O fluxo de dados por sua vez, é obtido transformando as Máquinas de Estados Finitos Extendidas em grafos de fluxos incluindo a aplicação de técnicas de análise de fluxos de dados.

[Dinh-Trong et al., 2006] usa os Diagramas de Sequência e de Classes da UML que são posteriormente integrados em um *Variable Assignment Graph (VAG)*, o qual será responsável por gerar valores de entradas de teste.

Samuel et al [Samuel et al., 2006] apresentam um método para gerar casos de teste a nível de *cluster* usando Diagramas de Comunicação. São selecionados predicados condicionais a partir da construção de uma árvore que representa os Diagramas de Comunicação. Os predicados condicionais são então transformados em Diagramas de Comunicação nos quais é aplicado uma técnica de minimização de função em que dados de teste são gerados.

Dados de teste também são gerados a partir da construção de blocos de Diagramas de Classes que permitem determinar a configuração dos objetos em que os dados serão executados utilizando os Diagramas de Interação, os quais determinam a seqüência de mensagens que devem ser testadas [Andrews et al., 2003].

A *UML-CASTING* é caracterizada como um protótipo capaz de gerar um *suíte* de testes usando modelos da UML. Esta ferramenta usa um conjunto de regras de decomposição que são aplicadas em Diagramas de Estado e de Classes em que a porcentagem de cobertura a ser realizada ou uma seqüência específica de métodos de chamada que deva ser executada são determinadas pela intervenção do usuário [Aertryck and Jensen, 2003].

3.5 Conclusão

Este capítulo apresentou os principais trabalhos desenvolvidos na área de teste funcional. É importante ressaltar que testes auxiliados por modelos formais e/ou semi formais não atuam diretamente na verificação de requisitos de software, mas sim na verificação de funções associadas a componentes ou classes.

Salienta-se também que é difícil encontrar um modelo que combina tanto o fluxo de dados quanto os fluxos de controle que aparecem durante a execução dos testes.

A principal contribuição dos trabalhos abordados neste capítulo é a de mostrar que a maioria dos modelos de testes apresentados não contêm uma semântica operacional associada ao modelo e também não conseguem representar o comportamento do sistema modelado. Tais modelos representam apenas os estados assumidos devidos às características da linguagem de modelagem utilizada. Outro fator limitante encontrado nos trabalhos apresentados é a ausência de mecanismos de execução e simulação dos modelos que dev-

eriam ser oferecidos pela própria linguagem de modelagem.

O contexto deste trabalho consiste em oferecer de maneira formal, a execução e simulação de um modelo de teste que dê suporte ao paralelismo, à concorrência, à representação de fluxos de controle e de principalmente estruturas de dados na forma de objetos (instâncias de classes) que as Redes de Petri a Objetos permitem representar.

Entretanto, para o modelo de especificação de teste funcional proposto, não foi desenvolvido ainda um editor e um compilador que permita automaticamente gerar a classe de teste a partir do modelo disponibilizado.

Outros aspectos não abordados pelo modelo de especificação de testes funcionais foi a utilização de critérios de teste funcional, como por exemplo, para o particionamento em classes de equivalência, o que favorecia de certa forma para determinar escolha de certas sequências de dados de teste.

Dessa forma, o próximo capítulo apresentará a proposta de desenvolimento de um modelo formal de especificação de teste funcional, usando as redes de Petri a Objetos.

4 *Modelo de Especificação de Teste Funcional Baseados em Processos de Workflow*

Neste capítulo, é apresentada a metodologia de desenvolvimento de modelos de especificação de testes funcionais que usa as Redes de Petri a Objetos para softwares orientados a objetos, baseando-se nos conceitos de processos de Workflow.

Será detalhado um modelo de especificação formal, baseado em Redes de Petri a objetos de [Sibertin-Blanc, 1985], que permite representar em um único formalismo tanto os dados de teste quanto as diversas atividades a serem realizadas para executar uma dada funcionalidade.

Para a implementação do modelo de especificação será definida uma classe de teste contendo um método principal que poderá chamar os métodos das principais classes da arquitetura de software da funcionalidade testada.

4.1 Aplicabilidade do Teste Funcional

Para se analisar a aplicabilidade do teste funcional, pode-se considerar a estrutura do modelo de desenvolvimento em V da Figura 31. Neste modelo, para cada etapa do desenvolvimento do software (parte esquerda do V), um conjunto de testes devem ser planejados (parte central do V). São nas etapas finais do modelo (parte direita do V) que as diversas atividades de teste serão realizadas, utilizando-se como dados de entrada os jogos de teste elaborados durante as etapas de desenvolvimento.

Os testes funcionais se concentram mais nas camadas do V que correspondem às etapas de especificação de requisitos e são denominados geralmente como *Testes de Aceitação*. É fato que através destes testes, as funções do software serão validadas, tornando-o pronto para ser entregue e/ou distribuído aos clientes.

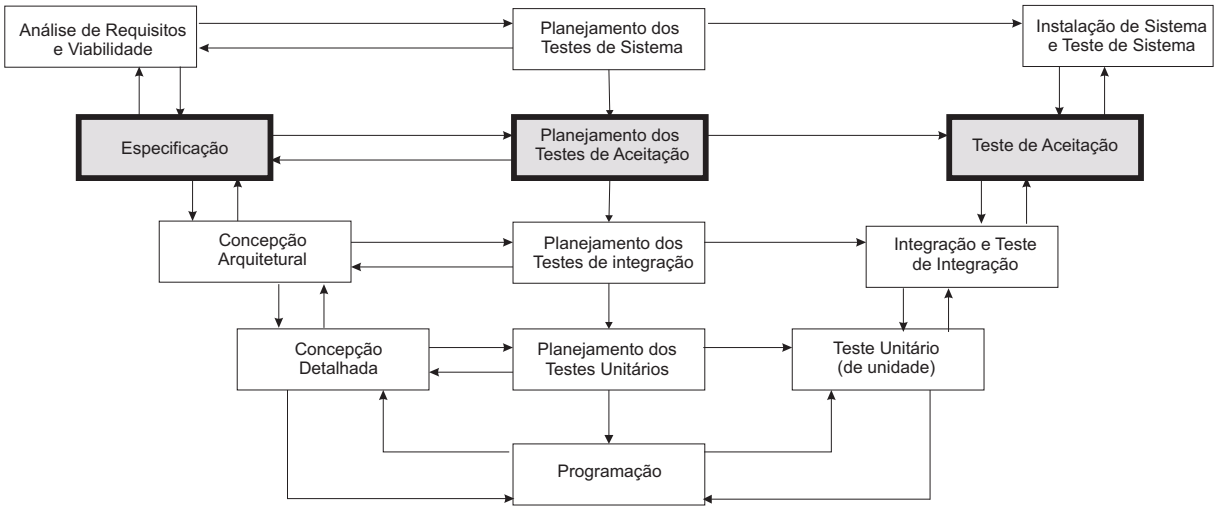


Figura 31: Modelo de Desenvolvimento em V

De acordo com a Figura 31, o lado esquerdo do V corresponde às fases de desenvolvimento do software, abrangendo a noção de concepção e viabilidade de construção do software, a etapa de coleta de dados, a identificação de componentes e a elaboração da arquitetura e, por fim, a codificação do software em uma linguagem de programação adequada. No contexto dos requisitos, a fase de *Especificação* de requisitos oferece apenas informações a respeito das funcionalidades que o software deve desempenhar e não considera aspectos relacionados à arquitetura ou codificação.

A parte central do V corresponde às etapas de planejamento de testes de cada fase do desenvolvimento. Com base nas informações disponíveis no lado esquerdo do V são elaborados vários documentos de teste, definindo-se dados de teste para cada fase concluída do desenvolvimento. A etapa de *Planejamento dos Testes de Aceitação*, por sua vez, é voltada para a definição de atividades e dados de teste a serem utilizados para a validação dos requisitos de software.

O lado direito do V corresponde à execução dos testes correspondentes a cada etapa do desenvolvimento do software. Tem-se, portanto, que a arquitetura do software já é conhecida, fornecendo um maior nível de detalhes a serem considerados. No momento em que a arquitetura do software é obtida, é possível identificar os objetos que fornecerão os métodos (operações) obtidos no modelo de teste. Assim, a etapa de *Teste de Aceitação* valida os requisitos do software de acordo com a arquitetura que se tem disponível do software.

4.2 Especificação dos Requisitos da Funcionalidade *Saque* – Parte Esquerda do V

A interação do usuário com a interface do sistema de caixa eletrônico 24 horas é ilustrado de maneira simplificada na na Figura 32. A especificação textual da função *Saque* de um caixa eletrônico pode ser a seguinte:

Inicialmente, o cliente irá inserir o cartão de identificação. Se o cartão de identificação for válido, então o cliente digitará a senha. Se a senha estiver correta, o cliente poderá solicitar ao sistema a operação de Saque. *Se o saldo na conta do cliente for suficiente, a quantia será debitada, o valor de dinheiro do cofre do banco será atualizado após o saque e o cliente receberá as notas assim como uma mensagem na tela informando o resultado da transação.*

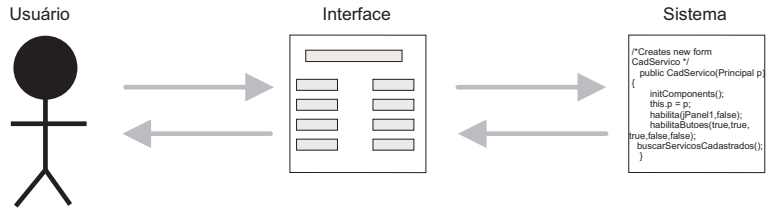


Figura 32: Interação entre o Usuário e o Sistema por meio da Interface

A interface do sistema bancário é exemplificada na Figura 33 com as principais funcionalidades que podem ser acessadas pelos usuários como saque, pagamentos, depósitos, etc.

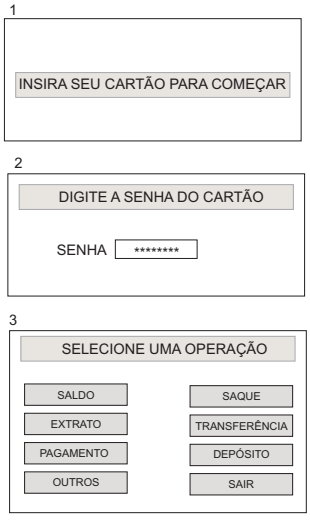


Figura 33: Interface do Sistema de Caixa Eletrônico Bancário

Abordagem Multi-formalismo no Contexto do Teste Funcional

A grande maioria dos diagramas de modelagem possuem uma visão parcial do sistema, ou seja, limitam-se a representar apenas algumas características do mesmo. Dessa forma, diversas notações gráficas podem ser utilizadas ao representar um sistema como meio de aumentar a precisão das informações.

Para garantir que as especificações de software sejam consistentes, utilizam-se métodos formais. Entretanto, boa parte dos desenvolvedores optam por utilizar notações semi-formais, uma vez que as notações formais baseiam-se em notações matemáticas complexas. Assim, é importante definir uma maneira de incorporar notações formais às especificações semi-formais originando um contexto de multi-formalismo de forma a beneficiar-se das características de cada notação.

Neste, trabalho, foi realizado uma correspondência intermediária entres estas notações de forma a entrelaçar as informações do software que, consequentemente, ajudam a encontrar um modelo de especificação do teste funcional. Alguns diagramas semi-formais e formais podem ser usados nas etapas de especificação de requisitos, como por exemplo:

- Diagramas de Casos de Uso da UML para ver as funcionalidades que o software tem e que são externamente observáveis por atores externos que interagem com o software;
- Diagramas de Atividade da UML que mostram o fluxo de atividades e de controle para a execução dos processos;
- Diagramas de Fluxo de Dados (DFDs) em que aparecem as funcionalidades do software assim como os fluxos de dados processados que são especificados por um dicionário de dados;
- Diagrama SA-RT que é a combinação do Diagrama de Fluxo de Dados com Diagrama de Fluxo de Controle;
- Rede de Petri Interpretada que especificará as estruturas de controle das funcionalidades especificadas;

No lado direito do V, o software já se encontra no final da fase de desenvolvimento, portanto, a arquitetura do software já é conhecida, fornecendo um maior nível de detalhes a serem considerados. Para isso, os seguintes diagramas podem ser usados:

- Diagramas de Classe da UML para ver a arquitetura do sistema (com as classes principais que os testes vão acessar);
- Diagramas de Seqüência da UML que permitem visualizar a execução de cada caso de uso de acordo com os objetos da arquitetura proposta pelo software. O uso da sequência permitirá também a execução de cada caso de teste de acordo com os objetos da arquitetura envolvida;

A Figura 34 ilustra as principais funcionalidades do sistema de caixa eletrônico bancário, usando Diagramas de Casos de Uso.

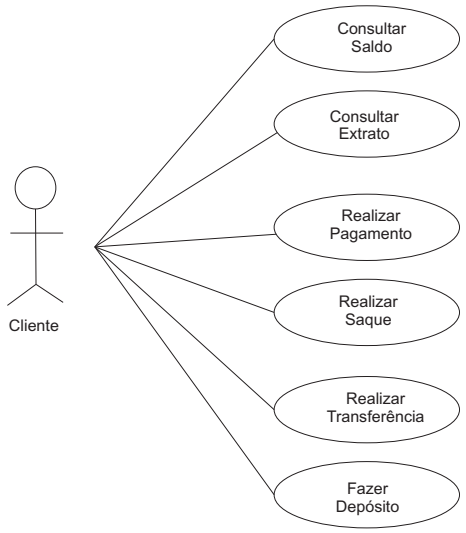


Figura 34: Diagrama de Casos de Uso para o Sistema de Caixa Eletrônico Bancário

Na fase de especificação de requisitos, a funcionalidade *Saque*, ilustrada pela WF-Net da Figura 38 pode ser derivada de um Diagrama de Atividades da Figura 35 que descreve as atividades desempenhadas pelo software.

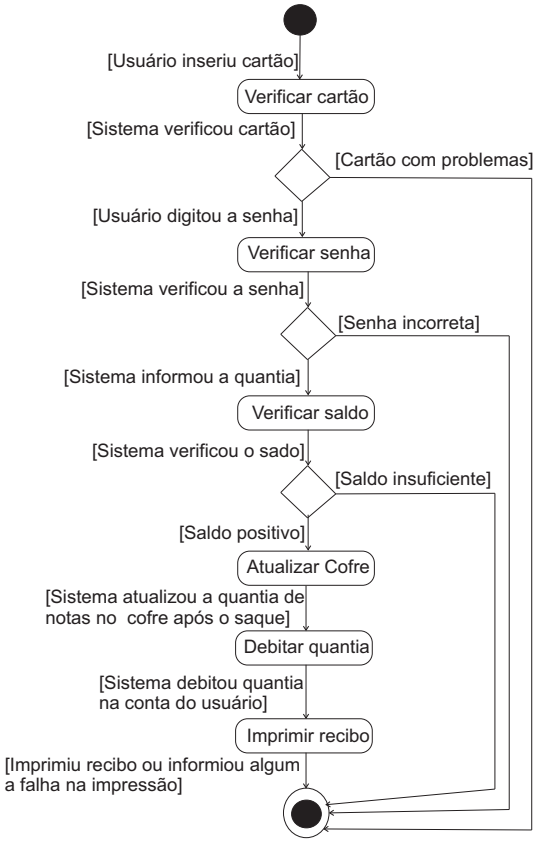


Figura 35: Representação do Sistema de Caixa Eletrônico Genérico por Diagrama de Atividades

O diagrama SA-RT ilustrado na Figura 36 juntamente com o dicionário de dados correspondente, representa a funcionalidade *Saque*, considerando-se os fluxos de dados e de controle que existem. O processo de controle *Gestão sistema* tem o papel de controlar os eventos que ocorrem no sistema, ou seja, controla a processamento dos dados. Formalmente, o controle dos eventos pode ser representado pela Rede de Petri Interpretada que apresenta o processo de controle *Gestão sistema* como mostra a Figura 37.

dados cartão= senha + saldo;
código cartão= Integer;
senha= Integer;
verificarSenha= Boolean;
saque= Float;
saldo do cofre= float;
atualizarCofre= Boolean;
saldo= Float;
verificarSaldo= Boolean;

dados cliente= nome + numero da conta + agencia;

número da conta= Integer;

agencia= Integer;

nome= String;

estado da impressora=Boolean;

mensagem= String;

verificarCartao= Boolean;

imprimirRecibo= Boolean;

debitarQuantia= Float;

saldo atualizado= Boolean;

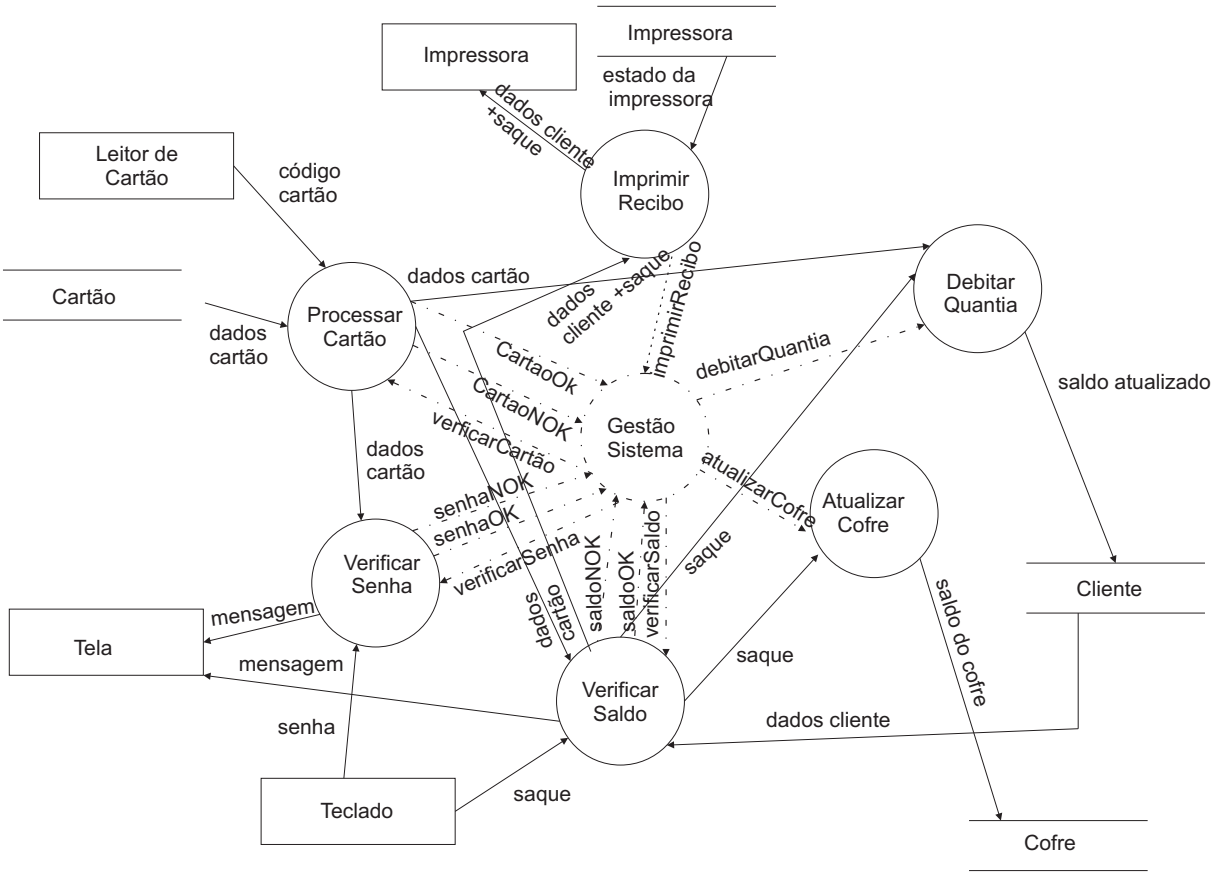


Figura 36: Representação do Sistema de Caixa Eletrônico por Diagramas SA-RT

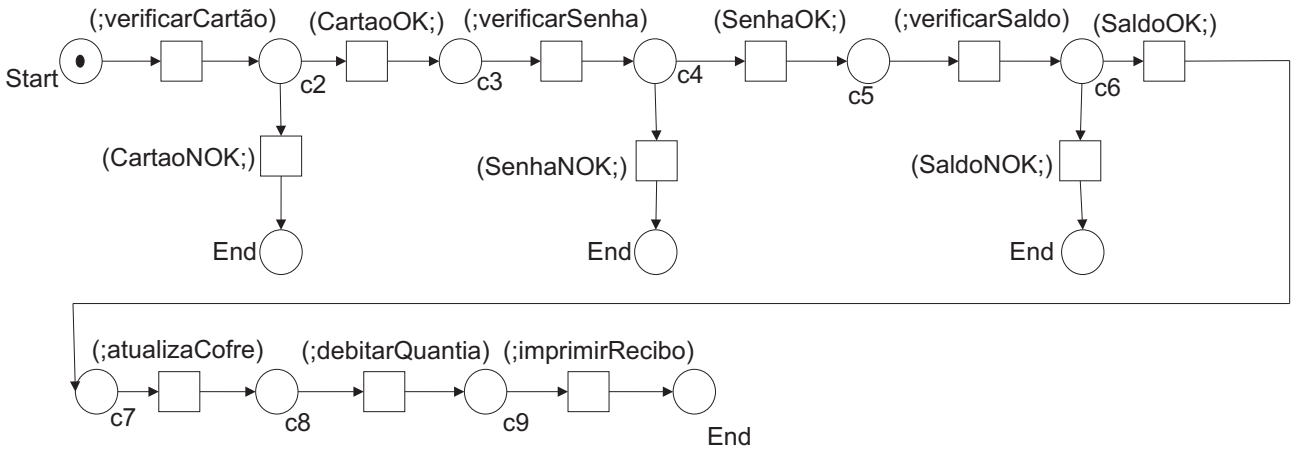


Figura 37: Representação do Processo de Controle "Gestão Sistema" por Redes de Petri Interpretadas

4.3 Modelo de Especificação de Teste Funcional – parte Central do V

A execução de um modelo de teste de um software orientado a objeto pode ser visto como a execução de um caso de teste (caso de um processo de negócio). O caso de teste possui um início e um fim e executará várias operações (métodos de objetos da arquitetura do software considerada) seguindo roteiros de execução de tipo sequencial, alternativo, paralelo, iterativo, etc.

A realização de uma operação durante a execução de um caso de teste será promovida por um objeto que fornecerá um método correspondente de um modo semelhante ao de um recurso usado para a execução de uma atividade no caso da execução de um processo de negócio [Aalst and Hee, 2002].

Mesmo assim, o modelo de especificação do teste funcional orientado a objeto deverá possuir características próprias à execução de um caso de teste. Em particular, ele deverá permitir a especificação das estruturas de dados e de controle, e ser um modelo suficientemente formal que será posteriormente implementado e transformado em um código executável. A seguir, é apresentado a definição do modelo de especificação do teste funcional:

Definição 5: *O modelo de especificação do teste funcional orientado a objeto é definido por uma Rede de Petri a Objeto [Sibertin-Blanc, 1985] tal que a Rede de Petri*

subjacente autônoma a partir da qual define-se a estrutura de controle do modelo de teste seja dada por um WF-Net [Aalst and Hee, 2002]. Neste caso, o WF-Net a Objeto correspondente pode ser definida pela 9-upla:

$$N_0 =< C_{lass}, P, T, V, Pre, Post, A_{tc}, A_{ta}, M_0 >$$

Onde:

- C_{lass} representa a classe de teste que é definida por um conjunto de atributos de dois tipos:
  - atributos que representam dados de entrada do teste;
  - atributos que representam dados utilizados para caracterizar o roteiro seguido pelo caso de teste no WF-Net subjacente, caracterizando-se dessa forma, em dados de saída do teste;
- P é um conjunto finito de lugares (lugar de entrada *Start* (Início), lugar de saída *End* (Fim) e lugares envolvidos nos diversos roteiros do Workflow-net subjacente) cujos tipos são dados por C_{lass};
- T é um conjunto finito de transições;
- V é um conjunto de variáveis, cujos tipos são dados por C_{lass};
- Pre é a função lugar precedente que a cada arco de entrada de uma transição faz corresponder uma soma formal de n-uplas de elementos de V;
- $Post$ é a função lugar seguinte que a cada arco de saída de uma transição faz corresponder uma soma formal de n-uplas de elementos de V;
- A_{tc} é uma aplicação que a cada transição associa uma condição que envolve os atributos das variáveis formais associados aos arcos de entrada;
- A_{ta} é uma aplicação que a cada transição associa uma ação/operação em forma de chamada de métodos de um objeto que envolve os atributos das variáveis formais associadas aos arcos de entrada e cujo valor de retorno da chamada (quando existir) é repassado a valores de atributos de variáveis formais associadas aos arcos de saída;
- M_0 é a marcação inicial que associa ao lugar de início *Start* um objeto de teste, instanciado de C_{lass}.

Para ilustrar as características principais da definição apresentada, é considerada a seguir, a especificação da funcionalidade *Saque* de um software usado em Caixa Eletrônico.

Modelo de Especificação de Teste da Funcionalidade *Saque* de um Caixa Eletrônico

A aplicação do teste funcional ocorrerá principalmente na interface do sistema desenvolvido já que as funções que serão avaliadas são essencialmente aquelas executadas pelos próprios usuários na interface do software.

A especificação da função *Saque* apresentada na seção anterior, não define formalmente o que se espera exatamente dos requisitos da função *Saque*. Será então necessário após a leitura de uma especificação textual usar, na medida do possível, um modelo formal que especifique de modo preciso os requisitos da função a ser implementada e testada. A seguir, são apresentados alguns modelos de teste possíveis para a operação *Saque*.

- Modelo Sequencial:

Para a operação de Saque, cujas operações são executadas de modo sequencial, o WF-Net da Figura 38 (baseada numa Rede de Petri Ordinária) fornece uma especificação possível para a função *Saque*. Neste modelo, foi definido um roteiro do tipo sequencial em que as operações ilustradas a seguir são visualizadas seqüencialmente:

- *verifica_cartão*: verifica a validade do código do cartão;
- $verifica\_senha$: verifica senha fornecida pelo cliente;
- $verifica\_saldo$: verifica se o saldo do cliente é suficiente para o saque da quantia solicitada;
- $atualiza\_cofre$: atualiza o valor do cofre após o saque;
- *debita_quantia*: atualiza o saldo do cliente após o saque;
- *imprime_recibo*: imprime uma mensagem de resultado da operação na tela referente à transação realizada.

Além disso, roteiros alternativos são oferecidos, indicando a ocorrência de falhas de algumas das operações, como:

- *erro_cartão*: indica que o cartão não foi identificado;

- *erro_senha*: a senha digitada foi inválida;
- *erro_saldo*: o saldo do cliente foi inferior ao saque solicitado.

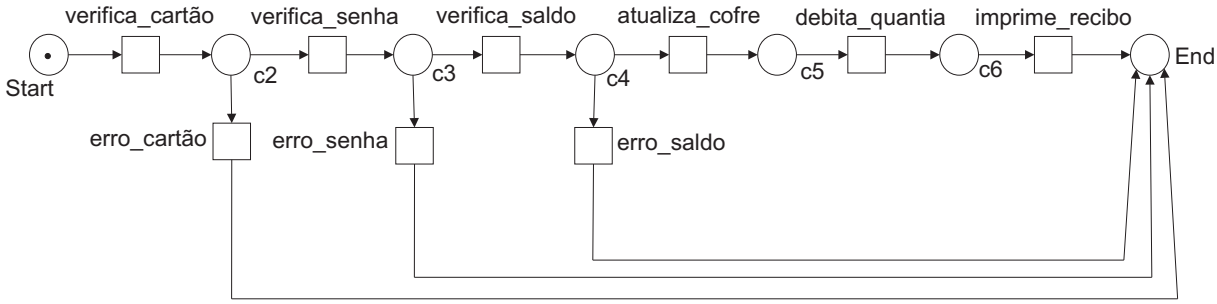


Figura 38: WF-Net: Representação do Sistema de Caixa Eletrônico Genérico por Redes de Petri Ordinárias

A marcação inicial da rede da Figura 38 corresponde a uma ficha (futuramente um objeto de teste possuindo valores de atributos a serem manipulados) que se encontra no lugar *Start* (marcação inicial do modelo de especificação da funcionalidade *Saque*). Neste modelo, é assumido que no cofre sempre existirá notas; portanto, não é apresentado falha caso as notas do cofre sejam insuficientes para realizar o saque.

O modelo apresentado na Figura 38 não depende ainda de nenhuma arquitetura específica e pode ser gerado na etapa de *Especificação* do modelo de desenvolvimento em V.

A Figura 39 apresenta o WF-Net a Objeto derivado do WF-Net da Figura 38. Neste modelo, às transições são associadas métodos (operações) que manipulam atributos fornecidos pela ficha cl_1 que se encontra no lugar *Start*. A ficha que representa o objeto cl_1 é uma instância da classe de teste $ClassedeTeste$. Os atributos definidos em cl_1 são fornecidos para as transições da rede através da variável de arco cl (que também é do mesmo tipo da classe $ClassedeTeste$). A variável de arco cl, dessa forma, fornece os atributos que são manipulados e atualizados pelas transições da rede.

Neste modelo, a classe de teste $ClassedeTeste$ é definida por:
classe $ClassedeTeste$;

- Dados de Entrada do Teste:
$numero : integer$;
$senha : integer$;

$saque : float$;

- Dados de Saída do Teste:
$nro : boolean$;
$s : boolean$;
$sld : boolean$;
$imp : boolean$;
$n : boolean$;
$saldo : boolean$;

Os atributos do objeto cl_1 são os seguintes:
$ClassedeTeste\ cl_1$;

- Dados de Entrada do Teste:
$numero$: 123456; // valor do número do cartão do cliente
$senha$: 5996084; // valor da senha para acessar a conta bancária do cliente
$saque$: $200,00$; // valor de saque em dinheiro informado pelo cliente

- Dados de Saída do Teste:
$nro : false$; // valor booleano que indica se o número do cartão é válido ou não
$s : false$; // valor booleano que indica se a senha de acesso à conta bancária é válida ou não
$sld : false$; // valor booleano que indica se o saldo do cliente é positivo ou não para efetuar o valor do saque
$imp : false$; // valor booleano que indica se a impressora imprimiu ou não um recibo da transação ao cliente
$n : false$; // valor booleano que indica se o cofre do banco foi atualizado ou não após o saque
$saldo : false$; // valor booleano que indica se o saldo do cliente foi atualizado ou não após o saque

Na Figura 39, as operações associadas as transições que representam as funções do sistema são:

- $getVerificaCartao$: verifica a validade do cartão, manipulando o atributo $numero$ que fornece o código do cartão a ser analisado. O resultado é então armazenado na

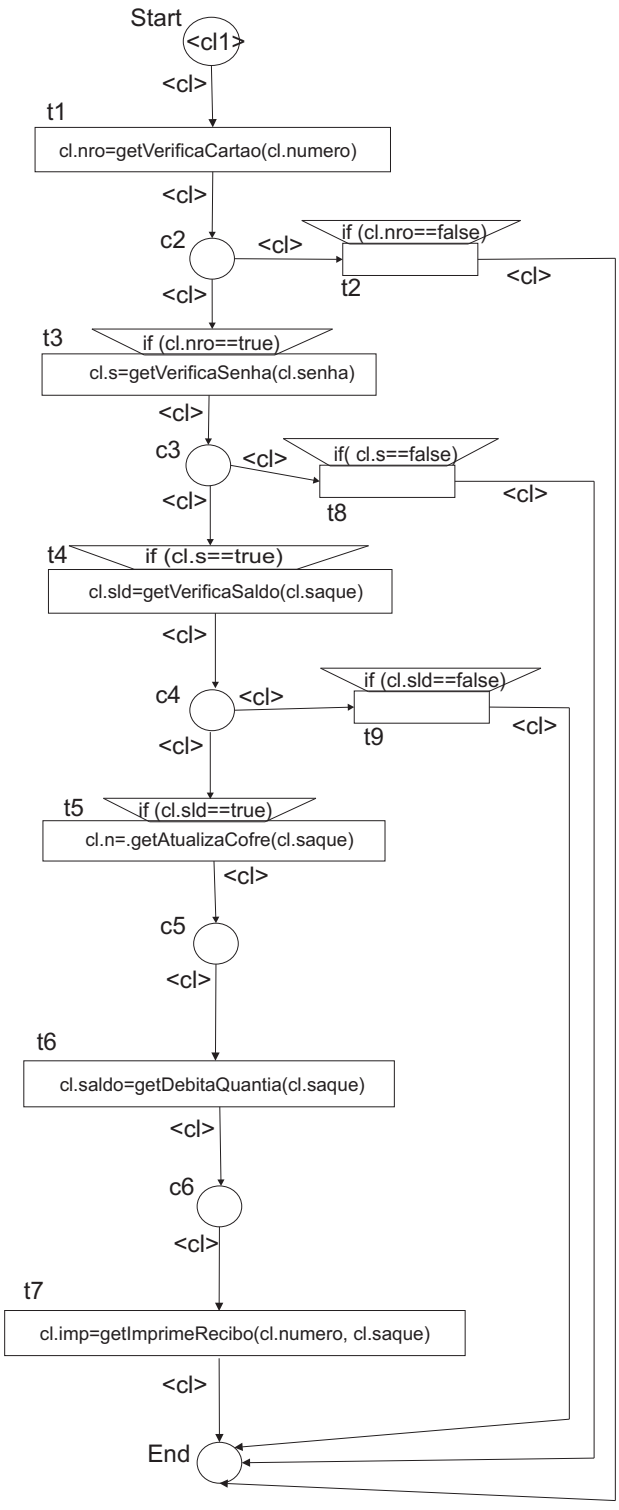


Figura 39: WF-Net a Objeto: Modelo de Especificação de Teste Funcional do Sistema de Caixa Eletrônico para a Função *Saque* Genérica

variável booleana *nro*, indicando se o cartão é válido ou não.

- $getVerificaSenha$: verifica a validade da senha, manipulando o atributo *senha* que fornece a senha do cartão a ser analisado. O resultado é então armazenado na

variável booleana s, indicando se a senha é válida ou não.

- $getVerificaSaldo$: verifica a existência de saldo, manipulando o atributo *saque* que fornece o valor de saque solicitado a ser analisado. O resultado é então armazenado na variável booleana *sld*, indicando se o saldo é suficiente ou não.

- $getAtualizaCofre$: atualiza o valor do cofre após o saque ser realizado, manipulando o atributo *saque*, indicando o valor de saque efetuado. O resultado é então armazenado na variável booleana n, indicando que o cofre está atualizado ou não.

- $getDebitaQuantia$: atualiza o valor do saldo do cliente após o saque ser realizado, manipulando o atributo *saque*, indicando o valor de saque efetuado. O resultado é então armazenado na variável booleana *saldo*, indicando se o saldo está corretamente atualizado ou não.

- $getImprimeRecibo$: imprime um recibo ao cliente, informando da transação efetuada através dos atributos *saque* e *numero*. O resultado é então armazenado na variável booleana *imp*, indicando se o recibo foi corretamente imprimido ou não.

É importante observar que algumas transições possuem pré-condições que precisam ser satisfeitas para que a transição seja disparada como, por exemplo, a condição $if(nro == true)$ da transição t_3 que indica que o cartão foi identificado; caso contrário, se $if(nro == false)$, então a transição t_2 é disparada finalizando o teste produzindo um objeto no lugar *End*.

- Modelo Iterativo:

Para a operação *Saque*, é possível representar a situação em que são registradas as diversas tentativas do usuário para entrar com a senha correta (caso a senha digitada esteja errada), caracterizando um roteiro interativo no WF-Net. Assim, a Figura 40 mostra um segundo exemplo de especificação da função *Saque* com possibilidade de ocorrer a execução de um roteiro iterativo. O modelo apresenta a mesma sequência de operações do modelo da Figura 38 com o adicional de representar a possibilidade do usuário tentar um certo número de vezes autorizado digitar uma senha inválida, finalizando a operação se o número de vezes for ultrapassado.

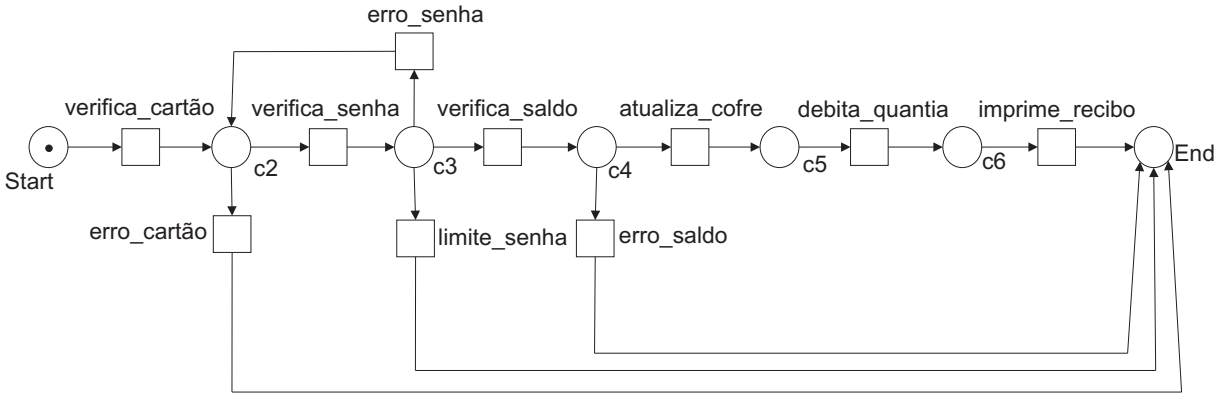


Figura 40: WF-Net: Representação do Sistema de Caixa Eletrônico Iterativo por Redes de Petri Ordinárias

É ilustrado na Figura 41, o WF-Net a objeto derivado do WF-Net da Figura 40. Neste modelo, às transições são associadas métodos (operações) que manipulam atributos fornecidos pela ficha cl_1 que se encontra no lugar $Start$. A ficha que representa o objeto cl_1 é uma instância da classe de teste $ClassedeTeste$. Os atributos definidos em cl_1 são fornecidos para as transições da rede através da variável de arco cl (que também é do mesmo tipo da classe $ClassedeTeste$). A variável de arco cl, dessa forma, fornece os atributos que são manipulados e atualizados pelas transições da rede.

Neste modelo, a classe de teste $ClassedeTeste$ é definida por:
classe $ClassedeTeste$;
- Dados de Entrada do Teste:
$numero : integer$;
$senha : integer$;
$saque : float$;
$max : integer$;

- Dados de Saída do Teste:
$nro : boolean$;
$s : boolean$;
$sld : boolean$;
$imp : boolean$;
$n : boolean$;
$saldo : boolean$;
$t : integer$;

Os atributos do objeto cl_1 são os seguintes:

$ClassedeTeste\ cl_1$;

- Dados de Entrada do Teste:

$numero$: 123456;

$senha$: 5996084;

$saque$: $200,00$;

max : 3; //valor máximo permitido para entrar com senhas inválidas

- Dados de Saída do Teste:

$nro : false$;

$s : false$;

$sld : false$;

$imp : false$;

$n : false$;

$saldo : false$;

t : 1; // valor que registra o número de vezes que o usuário digitou a senha errada

A cada vez que o usuário informa uma senha inválida, a transição t_8 é sensibilizada pela sua pré-condição *if(cl.s==false and cl.t<cl.max)*, que indica que a senha está incorreta e o número de vezes que senha foi digitada incorretamente representado pelo atributo $cl.t$ é menor que o valor do atributo $cl.max$. Se t_8 é disparada, o atributo t, é incrementado de um. A transição t_{10} por sua vez, é sensibilizada pela sua pré-condição *if(cl.s==false and cl.t==cl.max)* associada, indicando que a senha está errada e o limite de vezes ($max == 3$) permitido para se digitar a senha foi atingido. Neste caso, a transação é finalizada, produzindo um objeto cl_1 no lugar *End*.

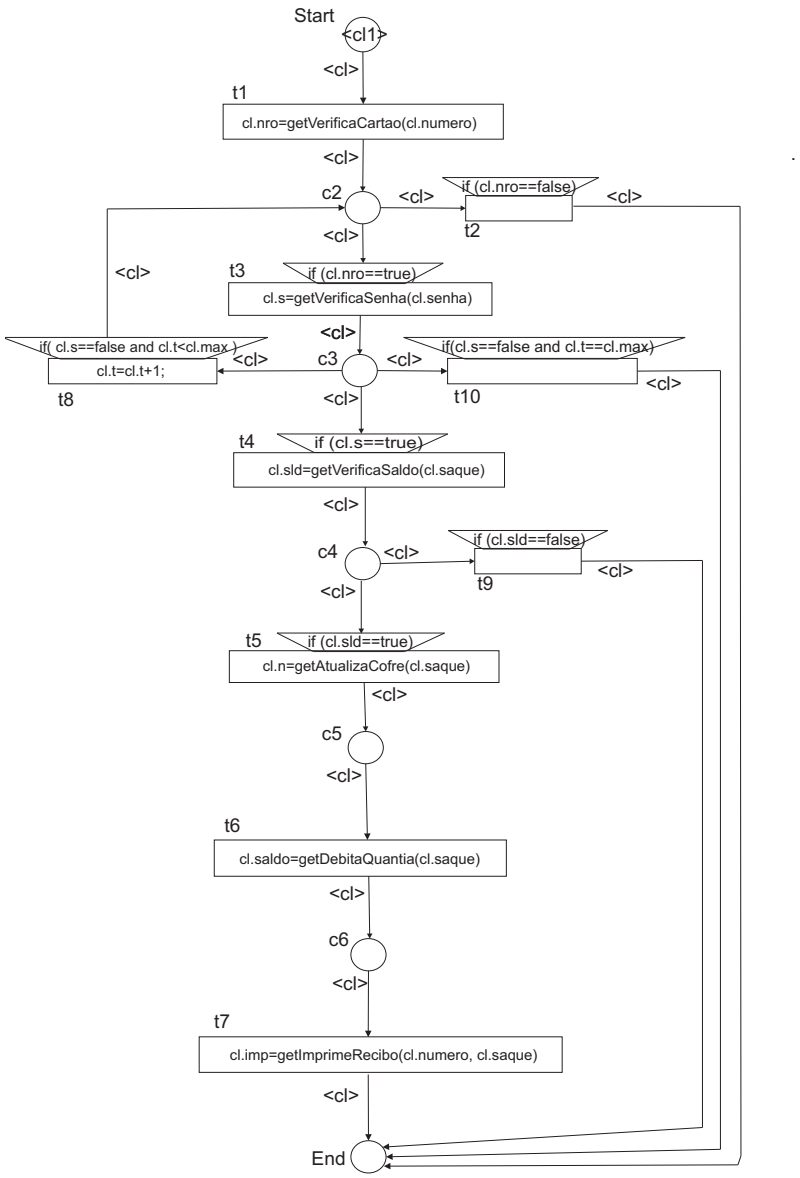


Figura 41: WF-Net a Objeto: Modelo de Especificação de Teste Funcional do Sistema de Caixa Eletrônico Iterativo

Um outro aspecto a ser considerado é a possibilidade de um loop infinito existir. Pode ocorrer que um usuário consiga entrar com senhas erradas por um limite indefinido de vezes porque o software permite. Neste caso, o modelo da Figura 41 pode ser adaptado de forma que ele chegue ao estado final da operação mesmo no caso de um loop infinito.

É preciso simplesmente definir para *max* um valor grande o suficiente que corresponde ao ∞ de forma que a execução do caso não fique presa em dentro do loop infinito.

- Modelo Paralelo

Para a operação *Saque*, é possível representar a situação em que algumas tarefas são

executadas em paralelo, sendo que a ordem de execução das tarefas não é importante. Observando a Figura 42 as funções *debita_quantia*, *atualiza_cofre* e *imprime_recibo* são executadas em paralelo. O usuário fornecerá dados de entrada para o modelo de especificação da função Saque de modo sequencial. Somente o processamento interno do sistema para as operações de debitar quantia, atualizar cofre e imprimir recibo serão executadas em paralelo, ou em uma sequência qualquer se o sistema não permitir o paralelismo verdadeiro.

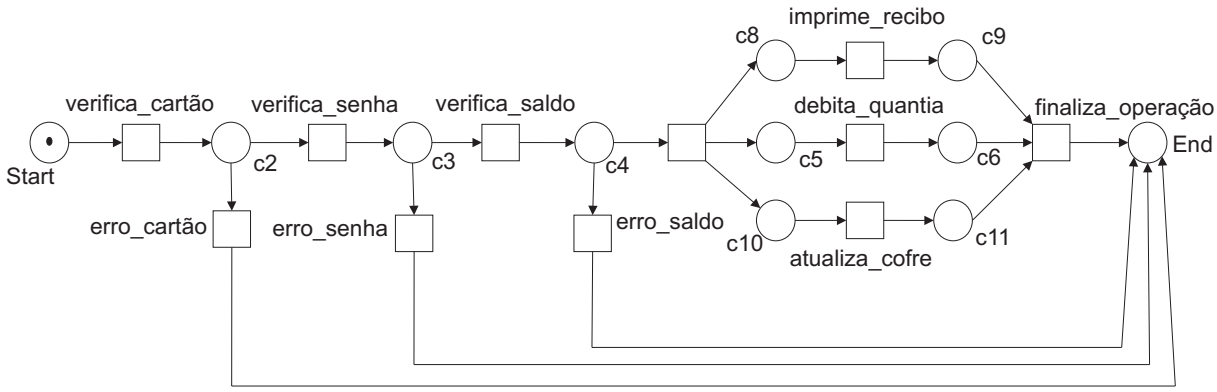


Figura 42: WF-Net: Representação do Sistema de Caixa Eletrônico Paralelo por Redes de Petri Ordinárias

A Figura 43 apresenta o WF-Net a Objeto derivado do WF-Net da Figura 42. Neste modelo, às transições são associados métodos (operações) que manipulam atributos fornecidos pela ficha cl_1 que se encontra no lugar *Start*. A ficha que representa o objeto cl_1 é uma instância da classe de teste $ClassedeTeste$. Os atributos definidos em cl_1 são fornecidos para as transições da rede através da variável de arco cl (que também é do mesmo tipo da classe $ClassedeTeste$). A variável de arco cl, dessa forma, fornece os atributos que são manipulados e atualizados pelas transições da rede.

Neste modelo, a classe de teste $ClassedeTeste$ é definida por:

classe $ClassedeTeste$;
- Dados de Entrada do Teste:
$numero : integer$;
$senha : integer$;
$saque : float$;

- Dados de Saída do Teste:

$nro : boolean$;
$s : boolean$;
$sld : boolean$;
$imp : boolean$;
$n : boolean$;
$saldo : boolean$;

Os atributos do objeto cl_1 são os seguintes:
$ClassedeTeste\ cl_1$;
- Dados de Entrada do Teste:
$numero$: 123456;
$senha$: 5996084;
$saque$: $200,00$;

- Dados de Saída do Teste:
$nro : false$;
$s : false$;
$sld : false$;
$imp : false$;
$n : false$;
$saldo : false$;

Após a sensibilização da transição $t8$ por meio da pré-condição que indica que o saldo é positivo ($if(cl.sld == true)$), a transição $t8$ é disparada e os valores de atributos de cl são repassados a três novas variáveis de arco: cl_2, cl_3 e cl_4, todas do tipo da classe $ClassedeTeste$. Depois do processamento paralelo das transições t_5, t_6 e t_7 os resultados são armazenados nos atributos *saldo*, n e imp da variável cl na transição t_{11}. Após o disparo e execução de t_{11}, o papel de cl é reestabelecido, produzindo um objeto cl_1 no lugar *End*.

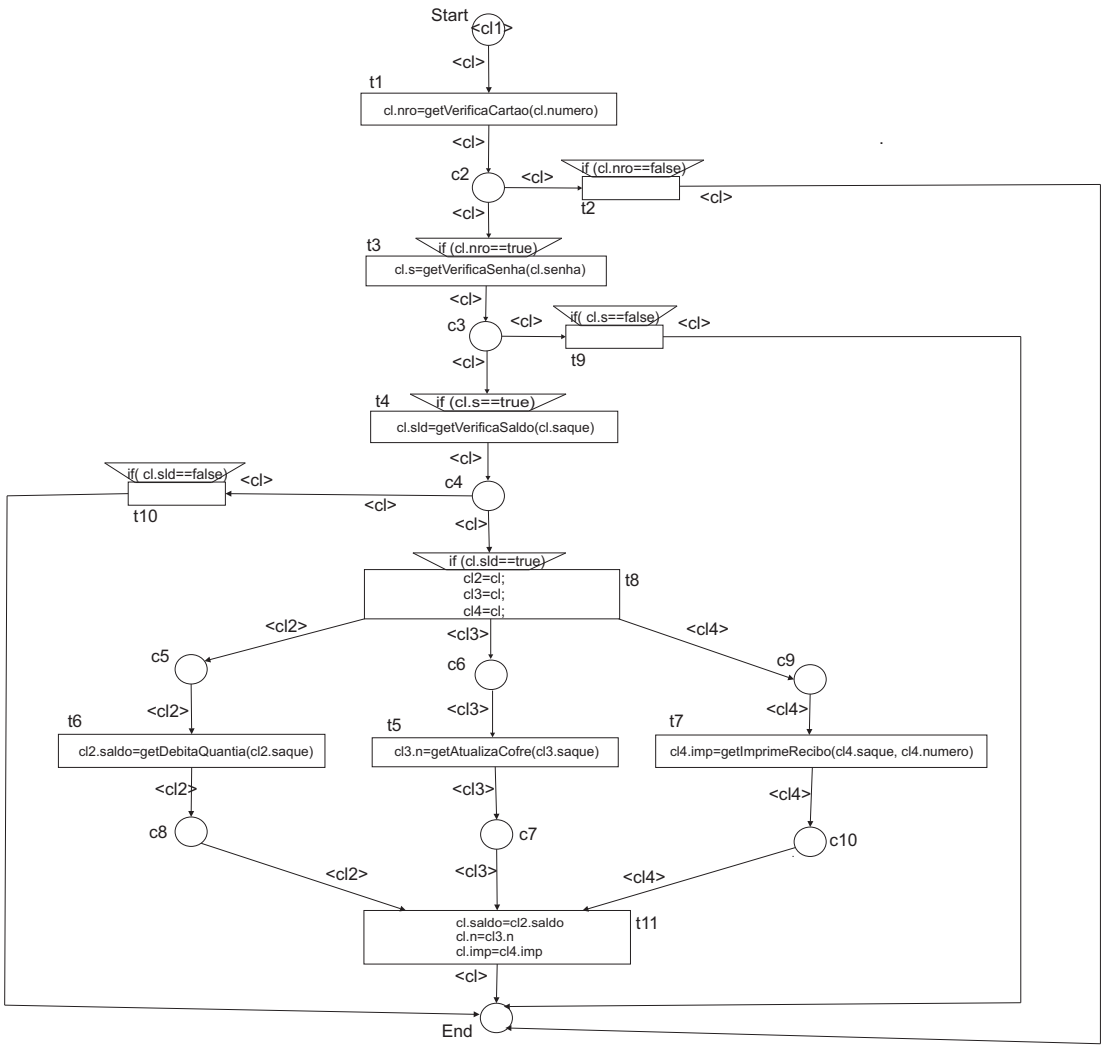


Figura 43: WF-Net a Objeto: Modelo de Especificação de Teste Funcional do Sistema de Caixa Eletrônico Paralelo

4.4 Implementação do Modelo de Especificação do Teste Funcional – Parte Direita do V

A implementação dos modelos de especificação do teste representam a instanciação de uma classe de teste genérica, cujo método principal é o WF-Net a objeto sem marcação inicial (já que a marcação inicial representa a criação de um objeto de teste).

No momento do teste, a arquitetura do software já é conhecida, portanto, independente do software a ser testado, a implementação do teste sempre consistirá de uma classe de teste que possuirá um método genérico de teste, o qual interagirá com os principais métodos da arquitetura do software a ser testado.

No contexto do exemplo da função *Saque*, suponha-se que parte da arquitetura do

software seja dada pelo Diagrama de Classe da Figura 44. A classe de teste interagirá então com as principais classes da arquitetura, como apresentado na Figura 45.

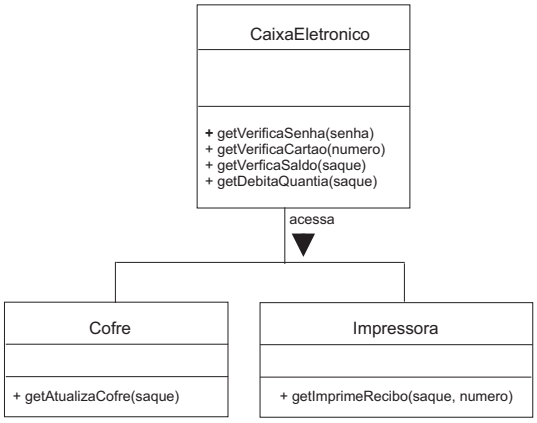


Figura 44: Diagrama de Classes Envolvendo as Principais Classes do Sistema de Caixa Eletrônico Genérico

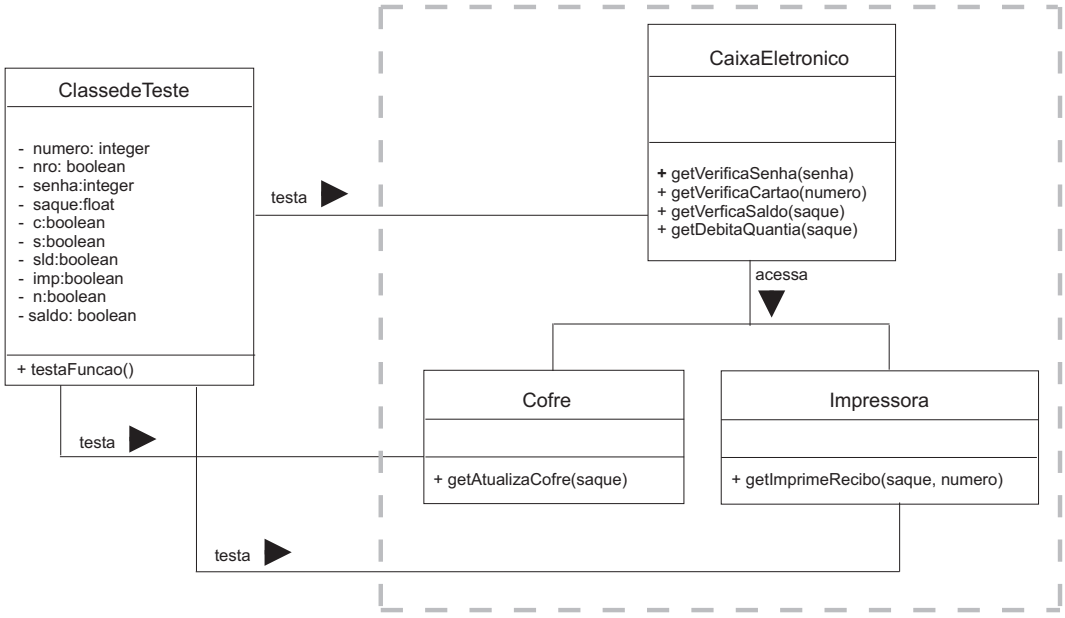


Figura 45: Diagrama de Classes Envolvendo as Principais Classes do Sistema de Caixa Eletrônico Genérico e a Classe de Teste

A Figura 46 ilustra o método principal da classe de teste para o caso do modelo sequencial. Os métodos chamados que aparecem nas transições do modelo são aqueles que são chamados pela classe de testes durante a execução da função *testaFuncao*. É importante ressaltar que na implementação do modelo são apresentados os objetos dos quais são extraídos os métodos, uma vez que na Figura 39 não é mostrado estes objetos, pois se trata apenas da definição do modelo.

[**Transição t1**: A transição t1 chama o método **getVerificaCartao** do objeto **caixa** da classe **CaixaEletronico** que verifica se o **numero** do cartão é válido. O resultado é então armazenado na variável booleana **nro**]

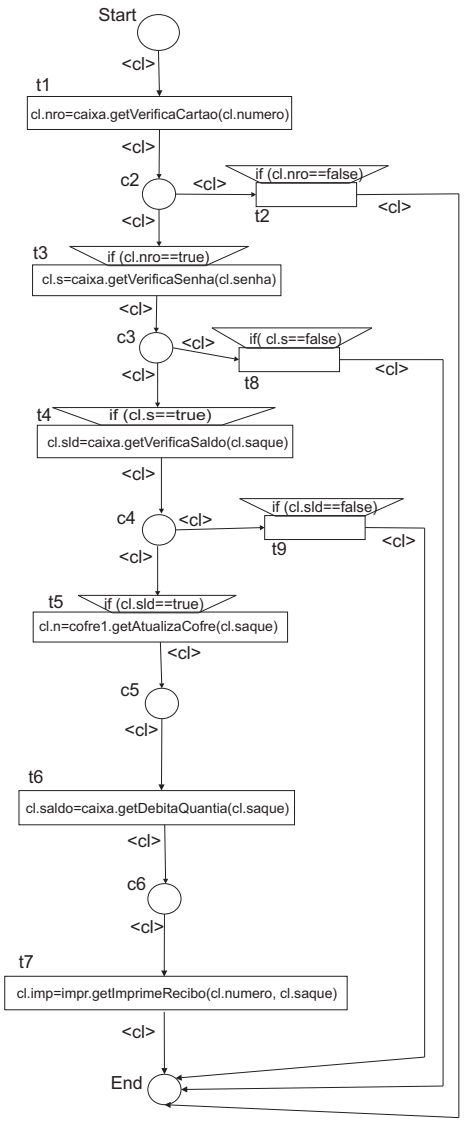
[**Transição t3**: se a pré-condição **if(cl.nro==true)** for satisfeita, o **numero** do cartão não apresentou problemas de identificação. A transiçao t3 chama o método **getVerificaSenha** do objeto **caixa** da classe **CaixaEletronico** que verifica se a **senha** digitada pelo cliente é a mesma do cartão. O resultado é então armazenado na variável booleana **s**]

[**Transição t4**: se a pré-condição **if(cl.s=true)** for satisfeita, a **senha** informada pelo cliente foi correta. A transição t4 chama o método **getVerificaSaldo** do objeto **caixa** da classe **CaixaEletronico** que verifica se o **saldo** é suficiente. O resultado é então armazenado na variável booleana **sld**]

[**Transição t5**: se a pré-condição **if(cl.sld=true)** for satisfeita, o saldo do cliente é positivo. A transição t5 chama o método **getAtualizaCofre** do objeto **cofre1** da classe **Cofre** que verifica se o cofre foi atualizado após o saque do cliente. O resultado é então armazenado na variável booleana **n**]

[**Transição t6**: a transição t6 chama o método **getDebitaQuantia** do objeto **caixa** da classe **CaixaEletronico** que debita na conta do cliente, a quantia sacada. O resultado é então armazenado na variável booleana **saldo**]

[**Transição t7**: a transição t7 chama o método **getImprimeRecibo** do objeto **impr** da classe **Impressora** que verifica se a impressora tem condições de imprimir o recibo ao cliente. O resultado é então armazenado na variável booleana **imp**]



[**Transição t2**: se a pré-condição **if(cl.nro==true)** for satisfeita, o cartão apresentou problemas de identificação. A transição t2 finaliza a operação]

[**Transição t8**: se a pré-condição **if(cl.s==true)** for satisfeita, a senha informada pelo usuário é incorreta. A transição t8 finaliza a operação]

[**Transição t9**: se a pré-condição **if(cl.sld==true)** for satisfeita, o saldo do cliente não é positivo. A transição t9 finaliza a operação]

Figura 46: WF-Net a Objeto: Implementação do Método da Classe de Teste para o Sistema de Caixa Eletrônico para a Função *Saque* Genérica

Na execução da funcionalidade *Saque*, algumas operações não são visíveis do ponto de vista do usuário. Por exemplo, o usuário não visualizará a atualização do cofre do banco. Tal atualização será realizada por objetos da arquitetura do software que serão criados no momento da execução da funcionalidade *Saque*. Por exemplo, o próprio objeto *caixa* da classe *CaixaEletronico*, poderá chamar o método do objeto *cofre1* (da classe *Cofre*) para realizar tal atualização. Mas, pode ser também um requisito do próprio teste da funcionalidade *Saque* verificar que tal atualização será realizada com sucesso. É por isso que a invocação do método que corresponde a atualização do cofre aparece também no próprio método da classe de teste. Não significa que no software final é essa classe que

vai efetivamente chamar tal método para atualizar o cofre.

Já que o modelo de especificação de teste apresentado na Figura 46 somente representa o método genérico da classe de teste, e não ainda a execução explícita de um cenário de teste, é normal que nenhuma ficha (objeto de teste) apareça no modelo (não há neste modelo uma marcação inicial que fixa valores de atributos necessários à execução do teste). É somente durante a instanciação de uma classe de teste que uma marcação inicial será criada (objeto de teste).

Já existem trabalhos em que, a partir de um modelo de Rede de Petri a Objetos é gerado automaticamente o código em C++, como por exemplo, a ferramenta SYROCO [Sibertin-Blanc, 2001]. Dessa forma, pode-se, por exemplo, como trabalhos futuros, discutir sobre a construção de uma ferramenta para a automatização de modelos de teste orientados a objetos.

Uma segunda opção é a implementação de um software para teste semelhante à um jogador de Rede de Petri a Objeto que simula em tempo real qualquer modelo baseado em Rede de Petri a Objetos [Cardoso et al., 1999]. Entretanto, essa estratégia é mais lenta do que a geração automática de código, e, também, dificulta a definição de tipos de atributos que sejam genéricos para qualquer classe de teste.

A terceira opção é a implementação manual de cada classe de teste em uma linguagem de programação orientada a objeto. No contexto deste trabalho, é apresentado no próximo capítulo um estudo de caso com a implementação em Java de classes de teste de um sistema comercial de vendas de peças automotivas.

4.5 Execução do Cenário de Teste – Parte Direita do V

Nesta seção, é apresentado a execução de um cenário de teste que mostrará como o modelo de teste será executado dinamicamente durante a execução do caso de teste, de acordo com os valores de atributos específicos para o modelo de especificação do teste funcional. Para tanto, é considerado o modelo do tipo sequencial mostrado na Figura 47 que, neste caso, possui o objeto de teste situado no lugar *Start*.

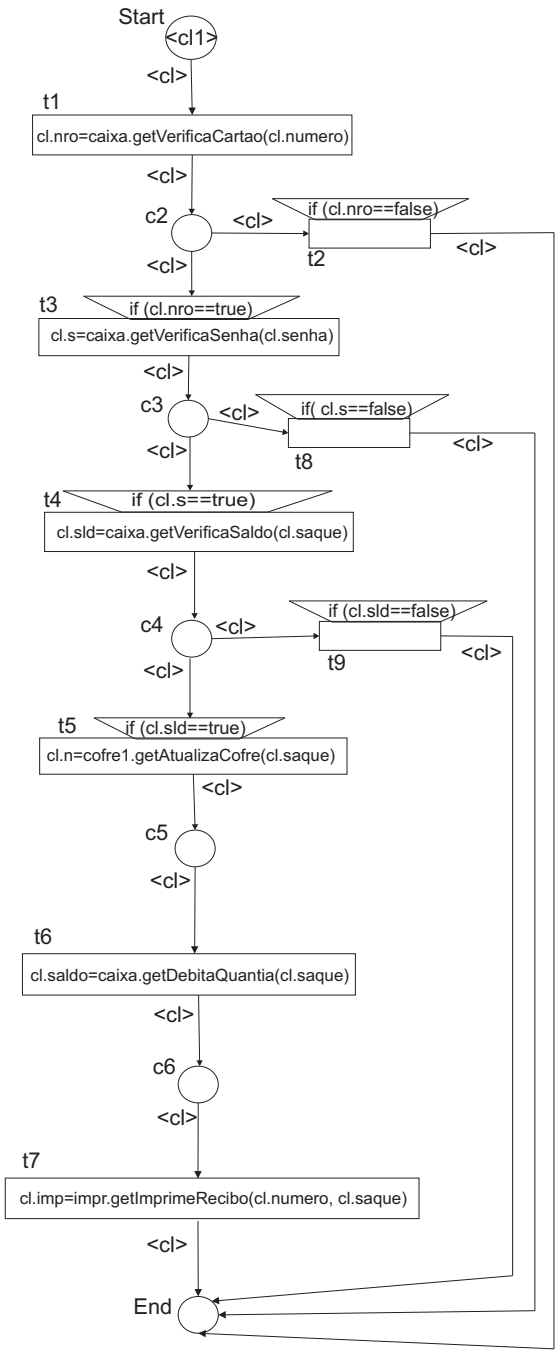


Figura 47: WF-Net a Objeto: Modelo de Especificação de Teste Funcional do Sistema de Caixa Eletrônico para a Função *Saque* Genérica com o Objeto de Teste – Ficha

A marcação inicial da rede é dada por um objeto de teste (ficha) cl_1 o qual representa um caso de teste, sendo a instância de uma classe $ClassedeTeste$ e possui os valores de atributos para serem testados. Ao final da execução do cenário de teste, terá uma marcação final situado no lugar *End*, que mostrará o objeto de teste com todos os valores de atributos atualizados, finalizando-se assim a execução do processo.

Os valores dos atributos do objeto cl_1 que se encontra inicialmente no lugar *Start*

são:

$ClassedeTeste\ cl_1$;
- Dados de Entrada do Teste:
$numero$: 123456;
$senha$: 5996084;
$saque$: 200, 00;

- Dados de Saída do Teste:
$nro : false$;
$c : false$;
$s : false$;
$sld : false$;
$imp : false$;
$n : false$;
$saldo : false$;

Para tais valores, a execução do teste se dará através da seqüência de disparos seguinte:

- A marcação inicial (o objeto cl_1 no lugar $Start$) sensibiliza a transição t_1, que não possui pré-condição. O disparo de t_1 chama o método $getVerificaCartao$ do objeto $caixa$ da classe $CaixaEletronico$, transmitindo, como valor de parâmetro, o atributo $numero = 123456$ (número do cartão). O método chamado verifica a existência do cartão, devolvendo um valor booleano salvado no atributo nro do objeto cl_1. Após o disparo de t_1, um novo objeto cl_1 (com valor do atributo $nro = true$ modificado no final do disparo, que indica que o número do cartão é existente) é produzido no lugar c_2.

- Com um objeto cl_1 em c_2, as duas transições, t_2 e t_3, estão sensibilizadas. Mas, somente a pré-condição da transição t_3, que informa a existência do cartão através do atributo nro, é satisfeita. O disparo de t_3 chama então o método $getVerificaSenha$ do objeto $caixa$ da classe $CaixaEletronico$, transmitindo, como valor de parâmetro, o atributo $senha = 599604$ (senha do cartão). O método chamado verifica a validade da senha, devolvendo um valor booleano armazenado no atributo s do objeto cl_1. Após o disparo de t_3, um novo objeto cl_1 (com valor do atributo $s = true$ modificado no final do disparo, que indica que a senha é correta) é produzido no lugar c_3.

- Com um objeto cl_1 em c_3, as duas transições, t_4 e t_8, estão sensibilizadas. Mas,

somente a pré-condição da transição t_4 que informa a validade da senha do cartão através do atributo s, é satisfeita. O disparo de t_4 chama então o método *getVerificaSaldo* do objeto *caixa* da classe *CaixaEletronico*, transmitindo, como valor de parâmetro, o atributo $saque = 200,00$ (valor a ser sacado pelo cliente). O método chamado verifica se há saldo suficiente de acordo com o saque solicitado pelo cliente, devolvendo um valor booleano armazenado no atributo *sld* do objeto cl_1. Após o disparo de t_4, um novo objeto cl_1 (com valor do atributo $sld = true$ modificado no final do disparo, que indica que o saldo do cliente é suficiente) é produzido no lugar c_4.

- Com um objeto cl_1 em c_4, as duas transições, t_5 e t_9, estão sensibilizadas. Mas, somente a pré-condição da transição t_5, que informa a existência de saldo através do atributo *sld*, é satisfeita. O disparo de t_5 chama então o método $getAtualizaCofre$ do objeto $cofre1$ da classe $Cofre$, transmitindo, como valor de parâmetro, o atributo $saque = 200,00$ (valor a ser sacado pelo cliente). O método chamado verifica se o cofre foi atualizado, devolvendo um valor booleano armazenado no atributo n do objeto cl_1. Após o disparo de t_5, um novo objeto cl_1 (com valor do atributo $n = true$ modificado no final do disparo, que indica que o cofre é atualizado) é produzido no lugar c_5.

- Com um objeto cl_1 em c_5, sensibiliza a transição t_6 que não possui pré-condição. O disparo de t_6 chama então o método *getDebitaQuantia* do objeto *caixa* da classe *CaixaEletronico*, transmitindo, como valor de parâmetro, o atributo $saque = 200,00$ (valor a ser sacado pelo cliente). O método chamado verifica se o débito foi realizado na conta bancária do cliente, devolvendo um valor booleano armazenado no atributo *saldo* do objeto cl_1. Após o disparo de t_6, um novo objeto cl_1 (com valor do atributo $saldo = true$ modificado no final do disparo, que indica que o valor do saldo do cliente está atualizado) é produzido no lugar c_6.

- Com um objeto cl_1 em c_6 sensibiliza a transição t_7 que não possui pré-condição. O disparo de t_7 chama então o método *getImprimeRecibo* do objeto *impr* da classe *Impressora*, transmitindo como valor de parâmetro os atributos $saque = 200,00$ (valor sacado pelo cliente) e $numero = 123456$ (número do cartão). O método chamado verifica se a impressora está disponível ou se não apresenta problemas de impressão, devolvendo um valor booleano armazenado no atributo *imp* do objeto cl_1. Após o disparo de t_7, um novo objeto cl_1 (com valor do atributo $imp = true$ modificado no final do disparo, que indica que a impressora está disponível) é produzido

no lugar *End*.

No final da execução do teste, obtém-se a Tabela 1 que mostra os valores dos atributos do objeto cl_1 relevantes para análise dos resultados do teste. Pode-se observar em particular que o número do cartão (atributo $nro = true$ do objeto cl_1) foi aceito pelo sistema, a senha foi corretamente digitada (atributo $s = true$ do objeto cl_1), o saque foi realizado devido ao saldo positivo ($sld = true$ objeto cl_1), de acordo com o valor pedido para saque (que corresponde ao atributo $saque = 200,00$ do objeto cl_1), o cofre ($n = true$ do objeto cl_1) foi atualizado após o saque, o saldo bancário do cliente foi atualizado após o saque ($saldo = true$ objeto cl_1) e finalmente, a impressora ($imp = true$ do objeto cl_1) imprimiu a mensagem na tela corretamente.

Tabela 1: Valores dos Atributos após o Disparo das Transições

Valor Inicial	**Após o Disparo da Transição**	**Valor Final**
$cl_1.nro = false$	t_1	$cl_1.nro = true$
$cl_1.s = false$	t_3	$cl_1.s = true$
$cl_1.sld = false$	t_4	$cl_1.sld = true$
$cl_1.n = false$	t_5	$cl_1.n = true$
$cl_1.saldo = false$	t_6	$cl_1.saldo = true$
$cl_1.imp = false$	t_7	$cl_1.imp = true$

Por meio dos Diagramas de Sequência é possível visualizar o resultado da execução de cada caso de teste de acordo com os objetos da arquitetura do sistema apresentada na Figura 45. Neste contexto, o ator será substituído pelo objeto de teste o qual testará os métodos invocados do sistema. O Diagrama de Sequência da Figura 48 representa de modo semi-formal a execução do cenário de teste pela WF-Net a Objeto.

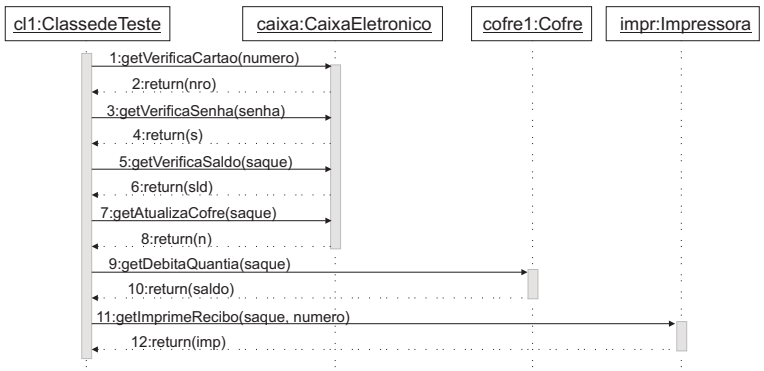


Figura 48: Diagrama de Seqüência do Sistema de Caixa Eletrônico Bancário para a Operação de Saque

4.6 Conclusão

Este capítulo apresentou a metodologia de desenvolvimento de modelos de especificação de teste funcional baseado em processos de Workflow e em redes de Petri a Objetos.

Os modelos de especificação de teste funcional planejados na parte central do V são derivados de um WF-Net que utiliza uma Rede de Petri subjacente quando considera-se apenas os requisitos de software do lado esquerdo do V.

A partir da WF-Net é originado a WF-Net a Objeto com marcação inicial em *Start* que possui como ficha o objeto de teste sem, no entanto, apresentar o nome dos objetos que compõe a arquitetura do software a ser testado.

Quando a arquitetura do software é conhecida (parte direita do V) a versão final do modelo de teste (em que os nomes dos objetos que contém os diversos métodos chamados aparecem nas transições do modelo de teste) pode ser então implementado na forma de uma classe de teste. Dessa forma, tem-se a instanciação da classe de teste que gerará a execução de um cenário de teste.

Finalmente, foram apresentadas diversas notações semi-formais comumente usadas na Engenharia de Software e que podem ajudar a especificar o modelo de especificação do teste funcional.

O próximo capítulo apresentará um estudo de caso de um sistema real voltado para o controle de vendas de peças codificado em linguagem Java.

5 *Estudo de Caso: Especificação de Modelos de Teste Funcional para um Sistema Comercial de Controle de Vendas*

Este capítulo contempla a geração de modelos de testes funcionais de algumas funções de um software comercial.

5.1 Descrição do Sistema

A abordagem de teste funcional proposta neste trabalho será aplicada a um sistema de controle de vendas de peças para veículos automotivos, desenvolvido em linguagem Java por alunos de graduação do curso de Ciência da Computação da Universidade Federal de Goiás – Campus Catalão.

A Figura 49 ilustra a tela inicial do sistema de controle de vendas, onde é necessário informar os dados de *login* e a senha para autenticação do usuário para o acesso às funções do sistema.

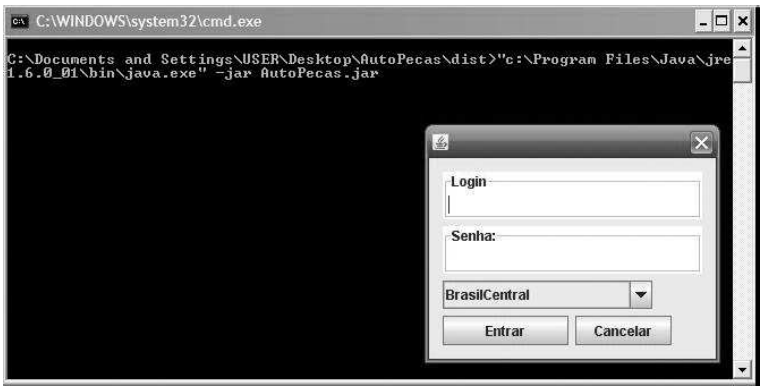


Figura 49: Tela Inicial do Sistema – Login

Após o usuário ter informado o *login* de acesso ao sistema, a tela principal do sistema é disponibilizada, como mostra a Figura 50, e passa a exibir todas as funções do sistema.

Figura 50: Tela Principal do Sistema e suas Principais Funções

Para maior clareza, a Figura 51 apresenta um Diagrama de Casos de Uso geral que ilustra as principais funções do software do ponto de vista do usuário, além dos principais relacionamentos entre o sistema e o ambiente. O diagrama mostra que para a execução dos casos de uso de *Cadastros, Buscas, Ferramentas, Relatórios* e *Operações* é necessária a interação de um usuário, no caso o funcionário.

A Figura 52 mostra os principais casos de uso referentes às operações de *Cadastros* oferecidos pelo sistema.

A Figura 53 mostra os casos de uso do sistema referentes à geração de *Relatórios*.

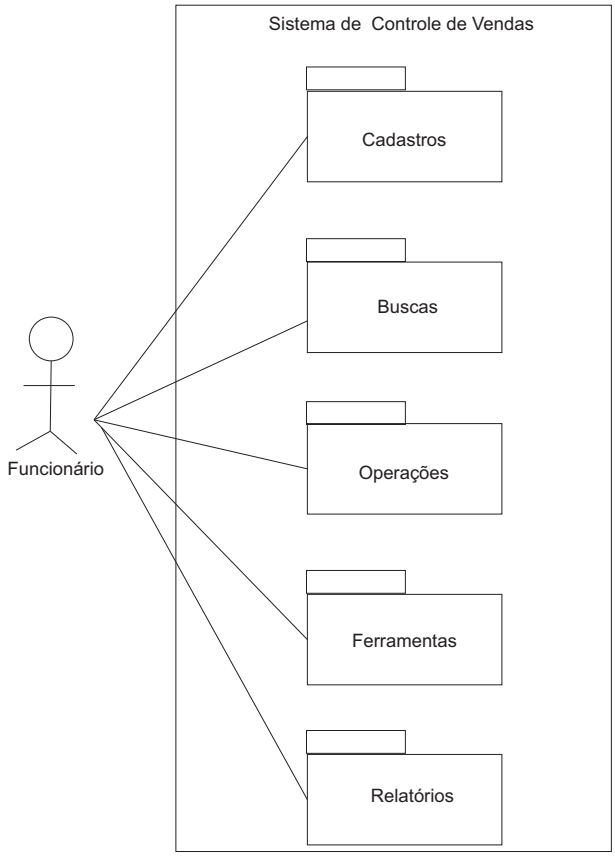


Figura 51: Diagrama de Casos de Uso – Geral

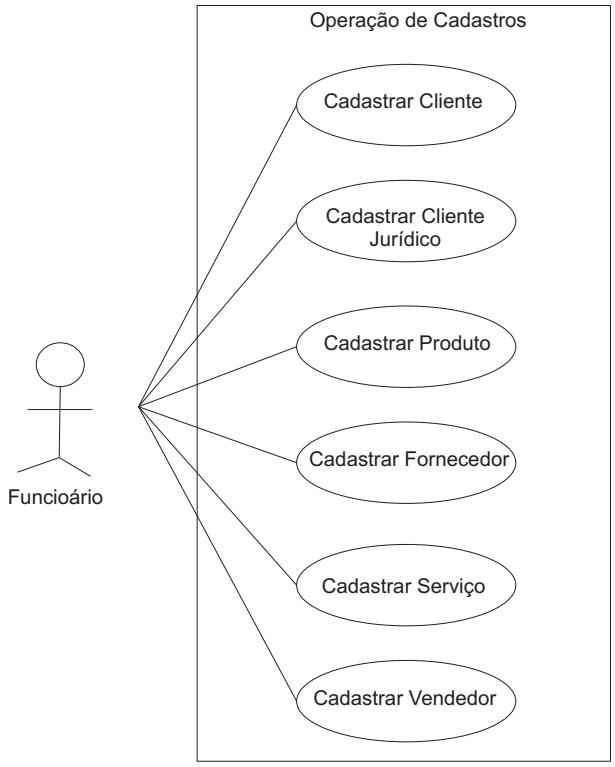


Figura 52: Diagrama de Casos de Uso – Cadastros

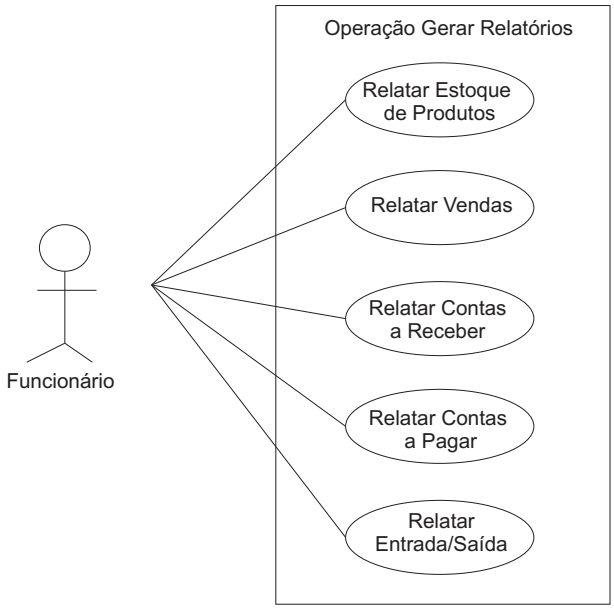


Figura 53: Diagrama de Casos de Uso – Relatórios

A Figura 54 ilustra as operações de *Buscas* disponíveis.

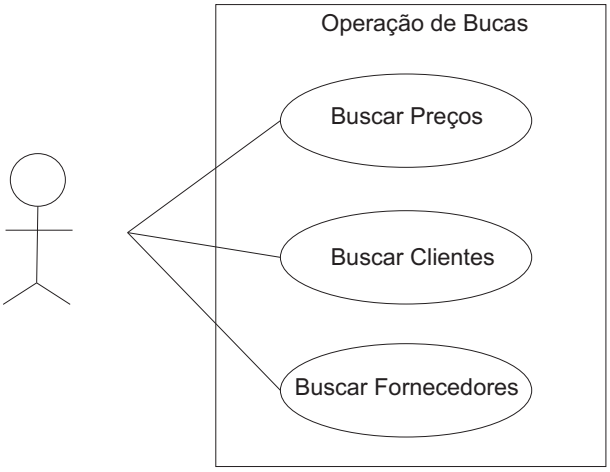


Figura 54: Diagrama de Casos de Uso – Buscas

A Figura 55, por sua vez, apresenta os casos de uso para a função de *Controle de Vendas*.

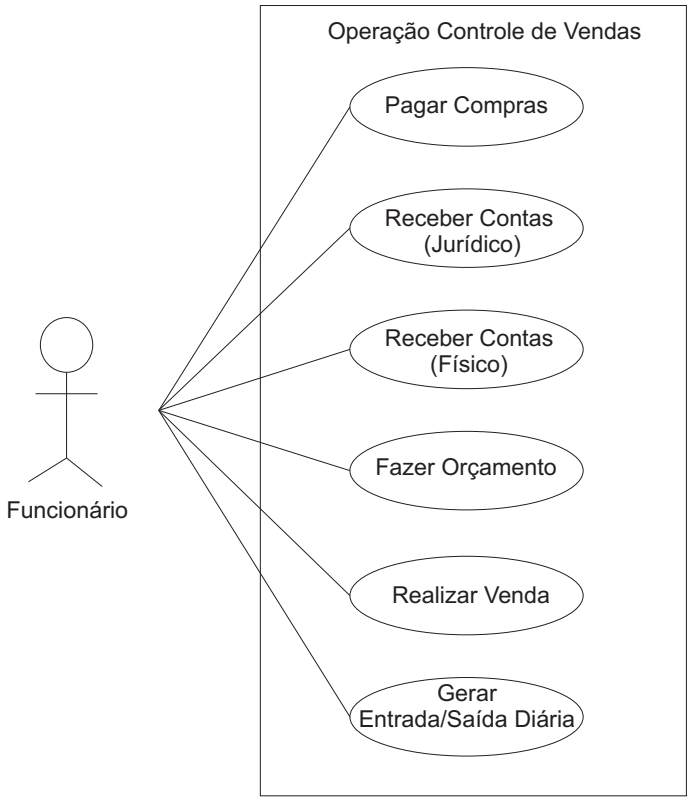


Figura 55: Diagrama de Casos de Uso – Controle de Vendas

Finalmente, a Figura 56 ilustra as ferramentas oferecidas pelo sistema como *Abrir Bloco de Notas* e *Escolher Estilo de Interface*.

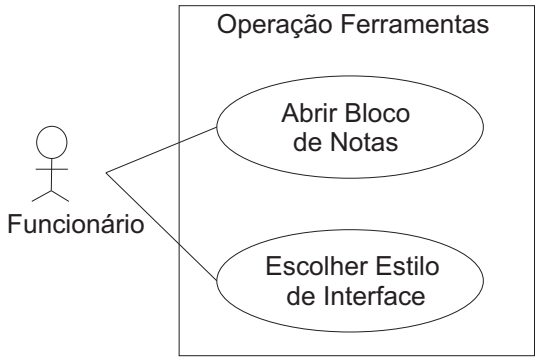


Figura 56: Diagrama de Casos de Uso – Ferramentas

Neste trabalho, foram consideradas as funções de *Cadastrar Cliente* e *Buscar Cliente* para a elaboração de modelos de teste funcional. A seguir, é fornecida a descrição e execução detalhada das duas funções.

Função Cadastrar Cliente

Para a operação *Cadastrar Cliente*, inicialmente o usuário do sistema seleciona no menu *Cadastros* da tela principal do sistema a opção *Cliente*, como mostra a Figura 57.

Figura 57: Seleção da Opção de Cadastro de Cliente

Logo após o usuário ter escolhido a funcionalidade de cadastrar cliente, uma tela de cadastros é ativada, como ilustra a Figura 58. Nesta interface, várias funções referentes à manipulação de dados cadastrais de clientes podem ser feitas. Seguindo a numeração exibida na interface, a opção (1) refere-se ao botão de busca do primeiro cliente cadastrado, a opção (2) busca um registro do cliente anterior, a opção (3) busca o próximo registro, a opção (4) busca o último registro, a opção (5) cadastra um novo cliente, a opção (6) remove um registro já existente, a opção (7) altera dados de um cliente já cadastrado, a opção (8) desfaz as alterações, a opção (9) salva os dados cadastrados e a opção (10) sai da interface de cadastros. Neste caso, é abordada a opção (5), que refere-se ao cadastro de um novo cliente.

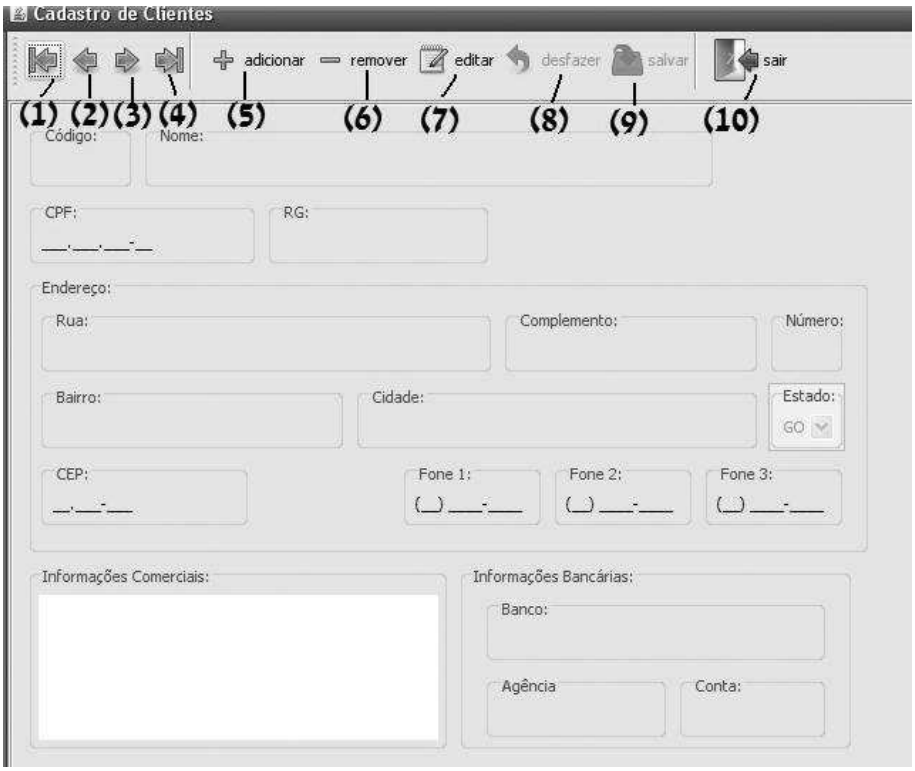


Figura 58: Seleção da Opção de Cadastro de Cliente

Ao selecionar a opção (5), a tela de cadastros é ativada e o usuário poderá cadastrar um novo cliente, informando seus dados nos campos apropriados. No final da operação, o usuário seleciona a operação de opção (9) que corresponde a salvar os dados. Se nenhum dos dados cadastrais inseridos pelo usuário estiverem incorretos, a operação é gravada no banco de dados com sucesso, caso contrário uma mensagem de erro é enviada ao usuário informando que os dados não foram salvos, como por exemplo, a invalidade do CPF. A Figura 59 apresenta a tela de cadastros para a opção de adicionar novo cliente.

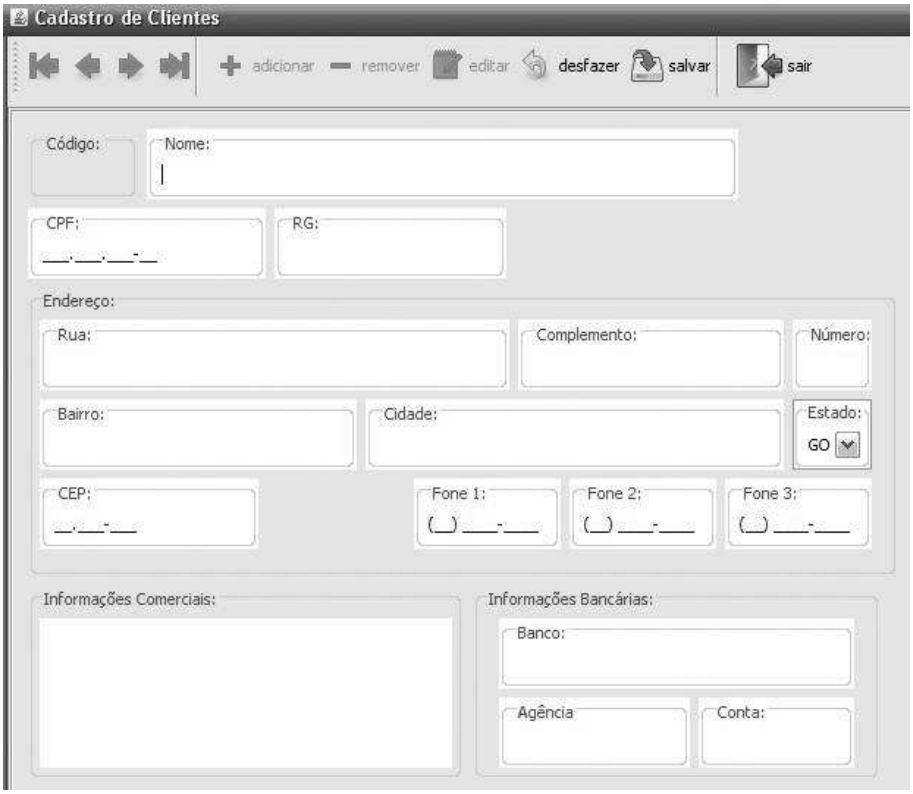


Figura 59: Adicionar Novo Cliente

O Diagrama de Atividades da Figura 60 descreve as atividades da função de *Cadastrar Cliente*, bem como as restrições (condições) que existem, como pode ser observado.

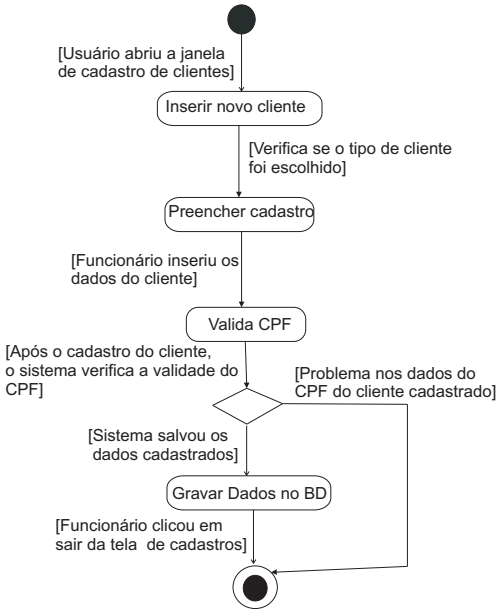


Figura 60: Diagrama de Atividades para a Função *Cadastrar Cliente*

A Figura 61 ilustra a Rede de Petri Interpretada, derivada do Diagrama de Atividades

da Figura 60, representando formalmente os fluxos de controle que existem para a execução da função *Cadastrar Cliente*.

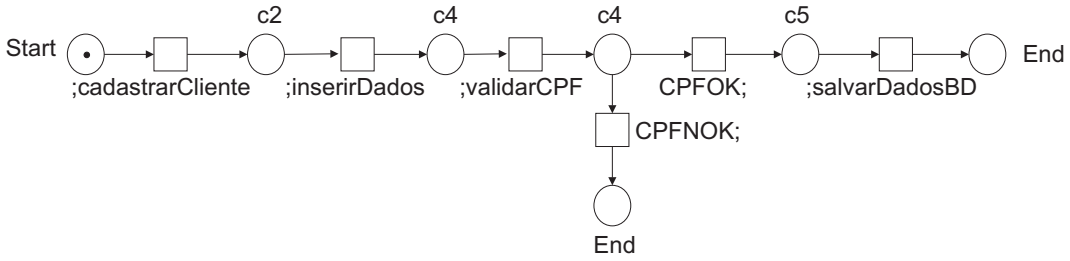


Figura 61: Redes de Petri Interpretada para a Função *Cadastrar Cliente*

Como se trata de um software já existente, arquitetura do sistema, portanto, é conhecida. A Figura 62 mostra a principal classe que compõem o caso de uso *Cadastrar de Cliente*.

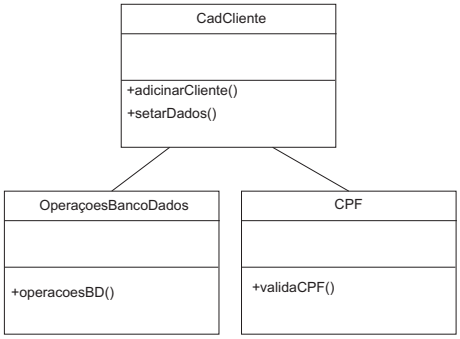


Figura 62: Diagrama de Classes Envolvendo as Principais Classes do Sistema de Controle de Vendas - *Cadastrar Cliente*

Função Buscar Cliente

Considerando-se agora a função *Buscar Cliente* o usuário deve, na tela principal do sistema, escolher no menu *Buscas* a opção *Cliente*, como mostra a Figura 63.

Figura 63: Seleção da Opção de Busca de Cliente

Logo em seguida, a tela que realiza a função *Buscar Cliente* é carregada, como pode ser visto na Figura 64. Nesta tela, de acordo com as opções numeradas, o usuário pode: *clicar* na opção número (1) para selecionar o tipo de cliente, preencher os campos das opções número (2) e número (3) que permitem entrar com o nome e o CPF do cliente a ser pesquisado, ou, se preferir, as opções número (4) e (5) que permitem, respectivamente, buscar os cliente cadastrados e os clientes que se encontram bloqueados por problemas de pagamento.

Assim que o usuário informa o cliente a ser pesquisado, a opção número (6), referente à *Tabela de Clientes*, mostra os resultados obtidos de acordo com a consulta. Ainda na opção número (6), o usuário seleciona o cliente desejado, caso ele tenha sido encontrado, e, ao lado, a opção número (7) torna possível ler de forma detalhada as informações sobre o cliente. A opção número (8), retorna a lista de produtos vendidos ao cliente pesquisado e a geração do boleto, para o pagamento da compra, como mostra a Figura 65.

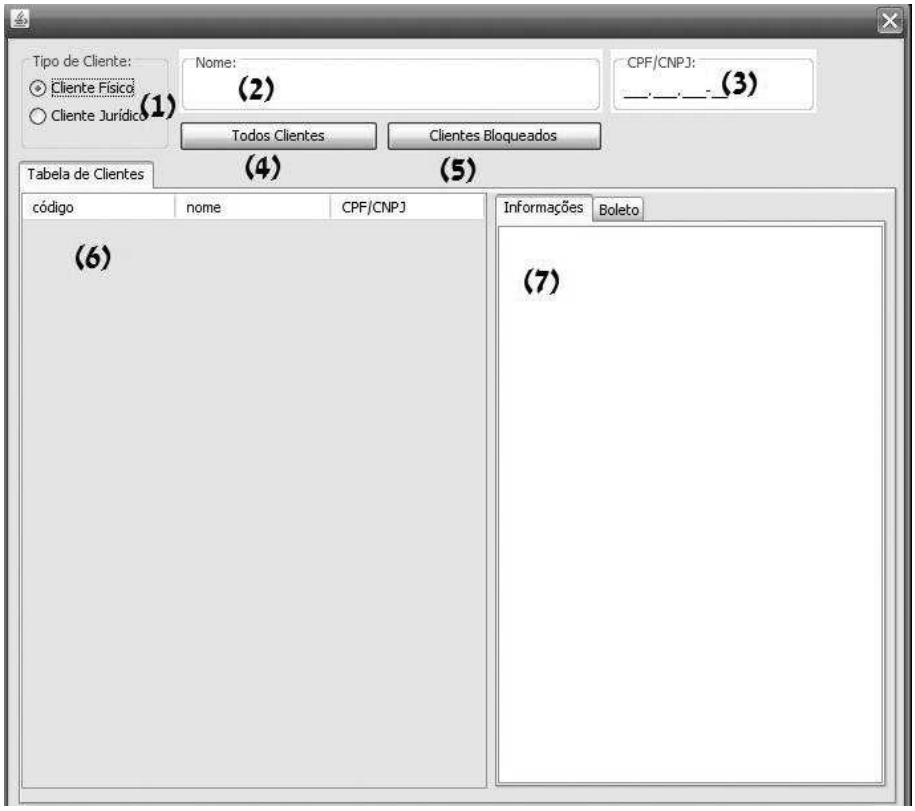


Figura 64: Interface de Busca de Cliente

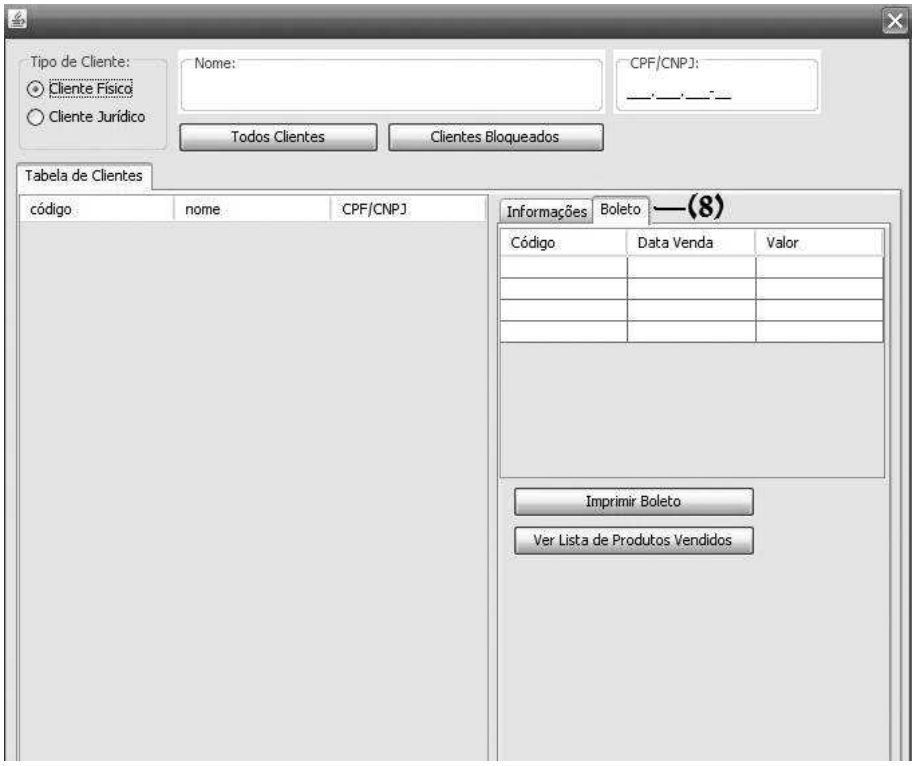


Figura 65: Geração de Boleto em Busca de Cliente

Para a função a ser testada, o usuário deverá buscar o cliente digitando o nome e/ou CPF nos campos correspondentes, (opções (2) e (3)). Em seguida, se na *Tabela de Clientes*, (opção (6)), for exibido algum resultado que corresponde à busca do usuário, deverá ser selecionado na tabela o cliente pesquisado para carregar as informações (opção (7)), e exibir a lista de produtos comprados. Se houver produtos comprados pelo cliente, deverá ser impresso o boleto para efetuar pagamento, (opção (8)).

A seguir, o Diagrama de Atividades da Figura 66 descreve a função de *Buscar Cliente* correspondente.

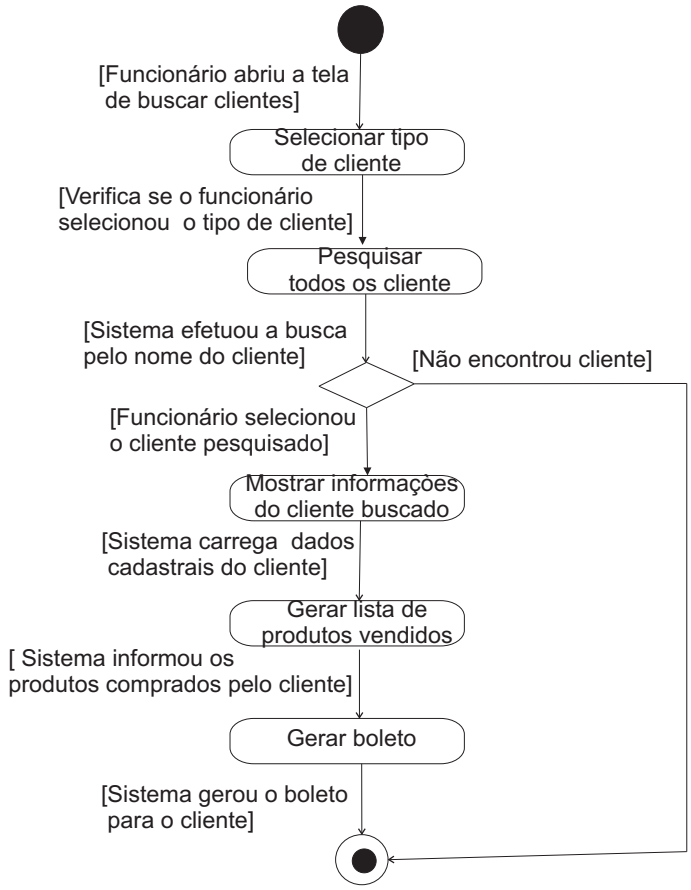


Figura 66: Diagrama de Atividades para a Funcionalidade de *Buscar Cliente*

A seguir é ilustrada na Figura 67 a Rede de Petri Interpretada derivada do Diagrama de Atividades da Figura 66, representando os fluxos de controle que existem para a execução da função *Buscar Cliente*.

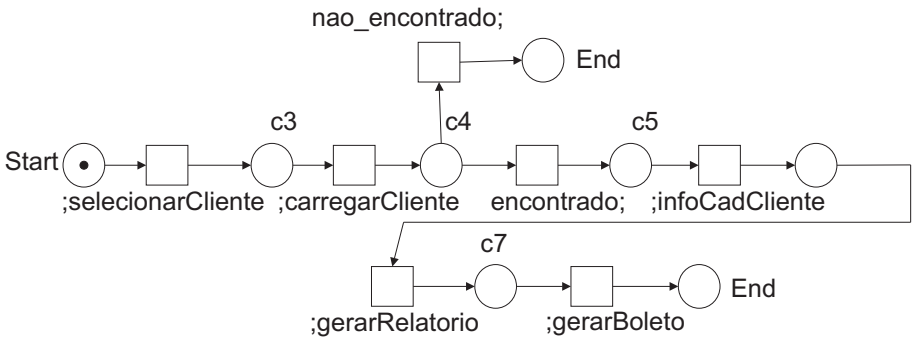


Figura 67: Redes de Petri Interpretada para a Função *Buscar Cliente*

A Figura 68 mostra as principais classes que compõem o caso de uso *Buscar Cliente*.

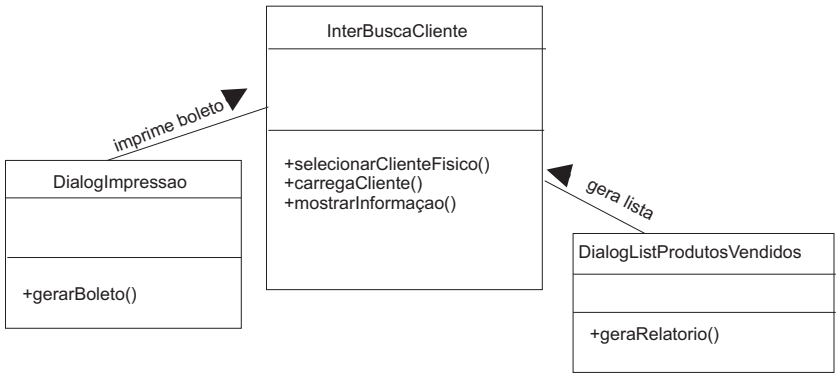


Figura 68: Diagrama de Classes Envolvendo as Principais Classes do Sistema de Controle de Vendas – *Buscar Cliente*

5.2 Especificação do Modelo de Teste Funcional para a Funcionalidade *Cadastrar Cliente*

A especificação da função *Cadastrar Cliente* pode ser representada pela WF-Net da Figura 69. Para a operação *Cadastrar Cliente*, as seguintes operações são executadas:

- **cadastrar_cliente:** seleciona a opção de inserir um novo cliente;
- **inserir_dados:** preenche um formulário com o dados do cliente;
- **validar_cpf:** verifica a validade do CPF informado pelo cliente;
- **salvar_dados:** registra o novo cadastro no banco de dados.

Além disso, é considerado um roteiro alternativo, indicando a ocorrência de falha no cadastro do cliente representado por **CPF_NOK** que indica que o valor do CPF informado pelo cliente não é válido.

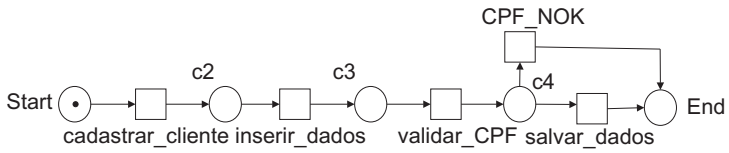


Figura 69: WF-Net: Modelo de Especificação da Funcionalidade *Cadastrar Cliente*

A Figura 70 apresenta a WF-Net a Objeto derivado da WF-Net da Figura 69. Neste modelo, às transições são associados métodos (operações) que manipulam atributos fornecidos pela ficha $cliente_1$ que se encontra no lugar $Start$. A ficha que representa o objeto $cliente_1$ é uma instância da classe de teste $TesteCadCliente$. Os atributos definidos em $cliente_1$ são fornecidos para as transições da rede através da variável de arco cl (que também é do mesmo tipo da classe $TesteCadCliente$). A variável de arco cl, dessa forma, fornece os atributos que são manipulados e atualizados pelas transições da rede.

Neste modelo, a classe de teste $TesteCadCliente$ é definida por:
classe $TesteCadCliente$;
- Dados de Entrada do Teste:
nome: string; // atributo referente ao nome do cliente
cpf: integer; // atributo referente ao CPF do cliente
rg: integer; // atributo referente ao RG do cliente
rua: string; // atributo referente a rua em que o cliente reside
numero: integer; // atributo referente ao número da casa em que o cliente reside
bairro: string; // atributo referente bairro em que o cliente reside
estado: string; // atributo referente ao estado do país em que o cliente reside
cep: integer; // atributo referente a CEP da rua em que o cliente reside
telefone1: integer; // atributo referente ao primeiro telefone de contato do cliente
telefone2: integer; // atributo referente ao segundo telefone de contato do cliente
telefone3: integer; // atributo referente ao terceiro telefone de contato do cliente
conta: integer; // atributo referente a conta bancária que o cliente possui
banco: string; // atributo referente a nome do banco que o cliente possui
agencia: integer; // atributo referente a agência bancária que o cliente possui
r: integer; // atributo que indica que um cadastro foi gravado no banco de dados
modo: integer // atributo que recebe o valor da operaçao a ser realizado, no caso 1 para

inserir ou 2 para editar

- Dados de Saída do Teste:

rs: boolean; // atributo booleano que ativa os campos de cadastro da interface para receber os valores

numDig: boolean; // atributo booleano que indica se o valor do CPF é válido ou não

add: boolean; // atributo booleano que ativa a interface de cadastros de clientes

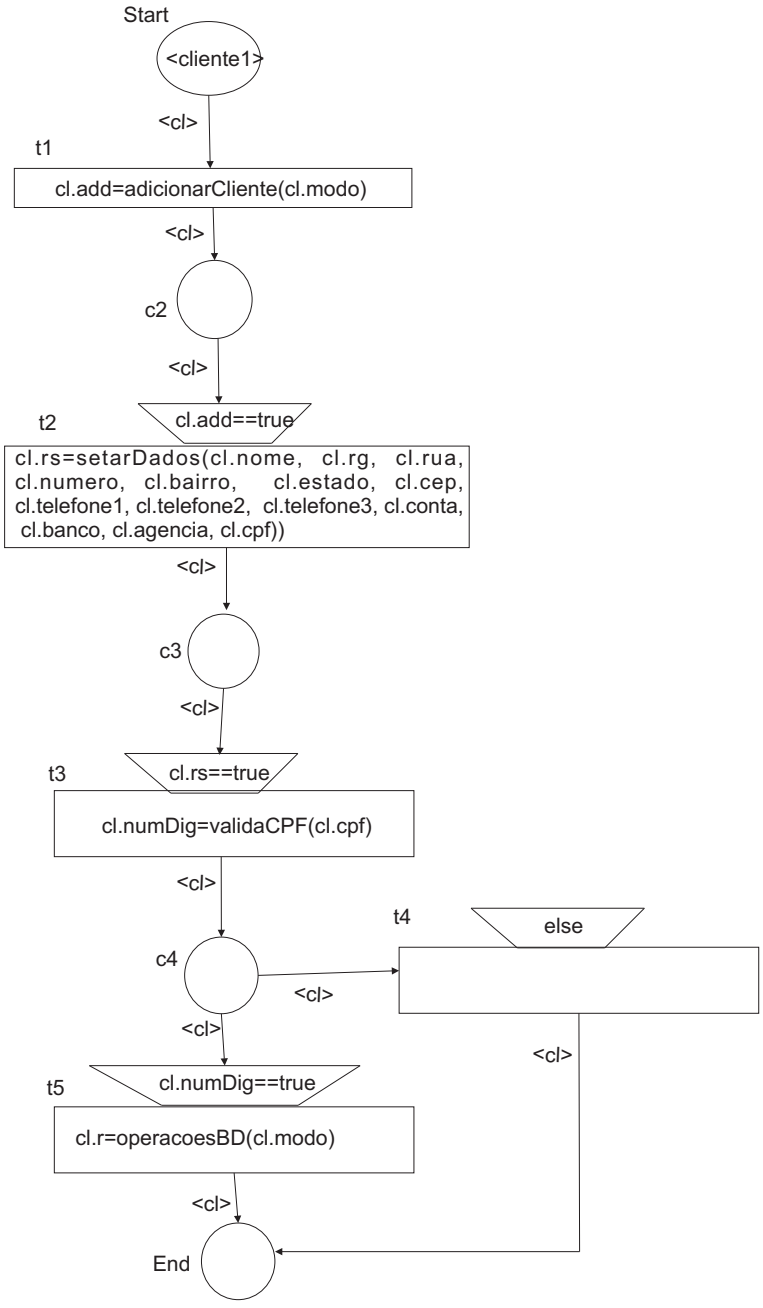


Figura 70: WF-Net a Objeto: Modelo de Especificação de Teste Funcional da Funcionalidade *Cadastrar Cliente*

Na Figura 70, as operações associadas as transições representam:

- *adicionarCliente* : ativa a tela para cadastro do cliente, manipulando o atributo *modo* que indica o tipo de operaçao a ser realizada no banco de dados. O resultado é então armazenado na variável booleana *add* que indica que a tela foi ativada ou não;
- *setarDados* : insere um novo cliente no banco de dados do sistema, manipulando os atributos *nome*, rg, cpf, *rua*, *numero*, *bairro*, *estado, cep, telefone1, telefone2, telefone3, conta, banco e agencia*. O resultado é então armazenado na variável booleana rs que indica que um valor foi preenchido ou não no campo da interface;
- validaCPF: verifica a validade do número do cartão, manipulando o atributo cpf que corresponde ao valor do CPF informado pelo usuário. O resultado é então armazenado na variável booleana $numDig$ que indica se o valor do CPF é válido ou não;
- $operacoesBD$: permite realizar operações de gravar ou editar dados no banco de dados, manipulando o atributo *modo* que indica o tipo de operaçao a ser realizada no banco de dados. O resultado é então armazenado na variável r que recebe o valor da operação realizada no banco de dados.

5.3 Especificação do Modelo de Teste Funcional para a Funcionalidade *Buscar Cliente*

A especificação da função *Buscar Cliente*, pode ser visualizada por uma WF-Net como mostra a Figura 71. Para a operação *Cadastrar Cliente* as seguintes operações são executadas:

- **selecionarCliente:** permite especificar o tipo de cliente a ser pesquisado; no caso, cliente físico ou jurídico;
- **carregarCliente:** o sistema efetua a busca pelo nome do cliente procurado. Caso seja encontrado, é retornado uma lista dos clientes que estão relacionados à pesquisa;
- **infCadCliente:** informa os dados cadastrais do cliente pesquisado;
- **geraRelatorio**: informa a lista de produtos que foram vendidos ao cliente, caso exista;

- **geraBoleto:** imprime o boleto para efetuar o pagamento da compra realizada pelo cliente.

Além disso, é considerado um roteiro alternativo representado por **nao_encontrado**: indica a ocorrência de falha na busca do cliente.

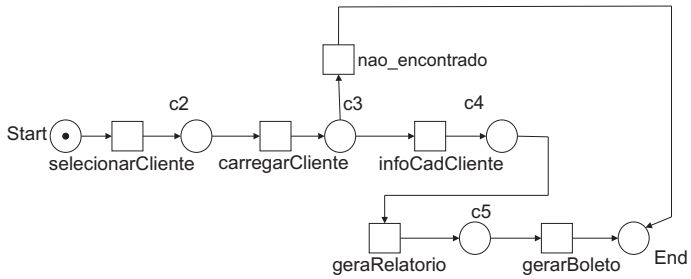


Figura 71: WF-Net: Modelo de Especificação da Funcionalidade *Buscar Cliente*

A Figura 72 apresenta a WF-Net a Objeto derivado da WF-Net da Figura 71. Neste modelo, às transições são associadas métodos (operações) que manipulam atributos fornecidos pela ficha $busca1$ que se encontra no lugar $Start$. A ficha que representa o objeto $busca1$ é uma instância da classe de teste $TesteBuscaC$. Os atributos definidos em $busca1$ são fornecidos para as transições da rede através da variável de arco cl (que também é do mesmo tipo da classe $TesteBuscaC$). A variável de arco cl, dessa forma, fornece os atributos que são manipulados e atualizados pelas transições da rede.

Neste modelo, a classe de teste $TesteBuscaC$ é definida por:
classe $TesteBuscaC$;

- Dados de Entrada do Teste:
listaProd: boolean; // atributo booleano que indica se uma lista de produtos vendidos é exibida
bol: boolean; // atributo booleano que verifica se o boleto para pagamento pode ser emitido
tipo: boolean; // atributo referente ao tipo do cliente
idCliente: integer; // atributo referente ao cadastro do cliente
nome: string; // atributo referente ao nome do cliente
CPF: integer; // atributo referente ao CPF do cliente
boleto: boolean; // atributo booleano que permite a impressão do boleto contendo os dados do cliente e do produto vendido para efetuar pagamento

- Dados de Saída do Teste:

cliente: boolean; // atributo referente ao tipo do cliente, no caso físico ou jurídico

rsult: boolean; // torna ativa a tela de informações cadastrais do cliente

rs: boolean; // atributo booleano que verifica se o nome do cliente foi encontrado

strld: boolean; // atributo booleano que indica a geração de relatório de produtos vendidos ao cliente

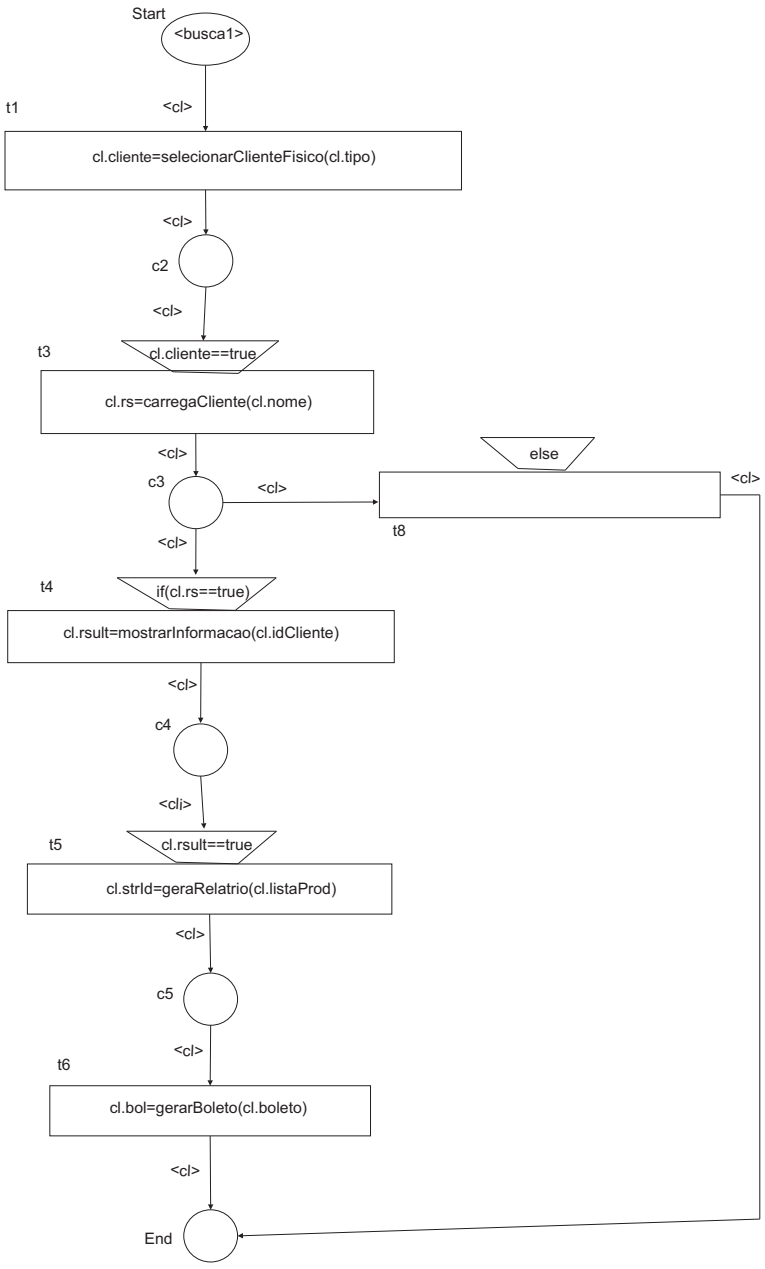


Figura 72: WF-Net a Objeto: Modelo de Especificação de Teste Funcional da Funcionalidade *Buscar Cliente*

Na Figura 72, as operações associadas as transições representam:

- **selecionarClienteFisico:** permite selecionar o tipo de cliente para realizar a

pesquisa, manipulando o atributo *tipo* que representa o tipo de cliente a ser pesquisado. O resultado é então armazenado na variável booleana *cliente*, indicando que o tipo de cliente (físico ou jurídico) foi ou não escolhido;

- **carregaCliente:** faz uma busca no banco de dados do sistema para encontrar o cliente desejado, manipulando o atributo *nome* que representa o nome do cliente. O resultado é então armazenado na variável booleana rs, indicando que a pesquisa foi feita.

- **mostraInformacao:** informa os dados cadastrais do cliente buscado, manipulando o atributo *idCliente* que representa as informações cadastrais do cliente. O resultado é então armazenado na variável booleana $rsult$, indicando que as informações cadastrais do cliente selecionado a partir do resultado obtido, foram exibidas;

- **geraRelatorio:** informa os dados do produto vendido ao cliente, manipulando o atributo *listaProd* que fornece os dados do produto. O resultado é então armazenado na variável booleana *srtld* que indica que o relatório foi ou não emitido;

- **gerarBoleto**: gera o boleto para o cliente efetuar o pagamento, manipulando a variável *boleto* que contem os dados do produto e do cliente. O resultado é então armazenado na variável booleana *bol* que indica que o boleto foi ou não gerado.

5.4 Implementação da Função *Cadastrar Cliente*

A Figura 73 mostra a classe de teste *TesteCadCliente* que implementa o teste funcional, interagindo com as demais classes que compõem a arquitetura do software.

A Figura 74 ilustra o método principal da classe de teste *TesteCadCliente* apresentado na Figura 73. Os métodos chamados que aparecem nas transições do modelo são aqueles que são chamados pela classe de testes durante a execução da função *testaCadastro*.

5.5 Implementação da Função *Buscar Cliente*

A Figura 75 mostra a classe de teste *TesteBuscaC* que implementa o teste funcional, interagindo com as demais classes que compõem a arquitetura do software.

A Figura 76 ilustra o método principal da classe de teste *TesteBuscaC* apresentado na Figura 75. Os métodos chamados que aparecem nas transições do modelo são aqueles

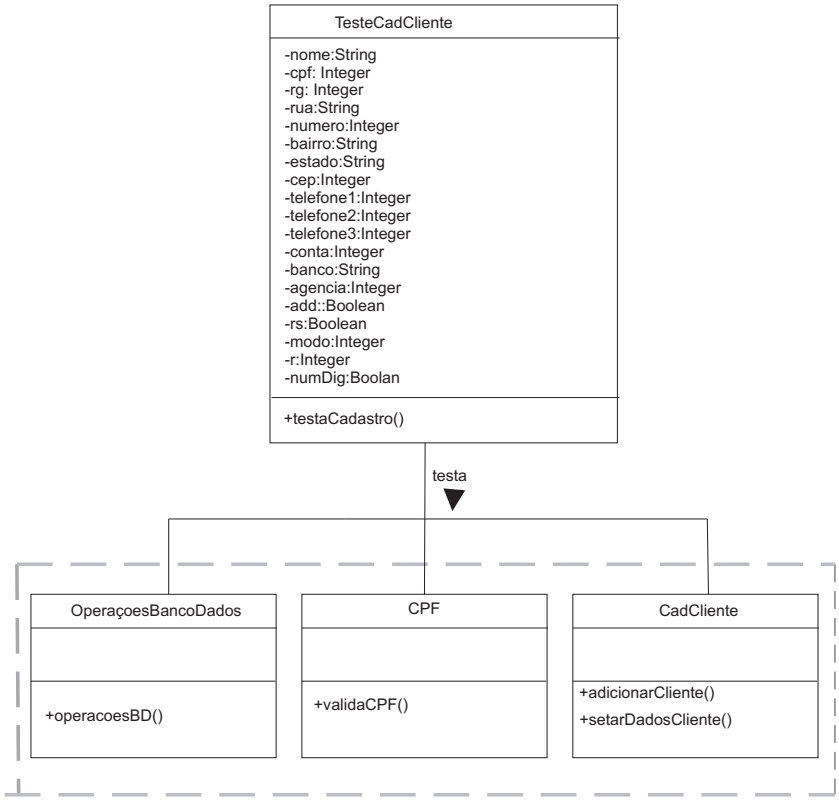


Figura 73: Classe de Teste *TesteCadCliente* Envolvendo as Principais Classes do Sistema de Controle de Vendas – *Cadastrar Cliente*

que são chamados pela classe de testes durante a execução da função *testaBusca*().

5.6 Cenários da Execução de Testes para a Função Cadastro de Cliente

Nesta seção, é considerado o cenário da execução do teste da função *Cadastro de Cliente*. A Figura 77 ilustra a interface gráfica para a execução do teste. Neste contexto, três cenários serão considerados.

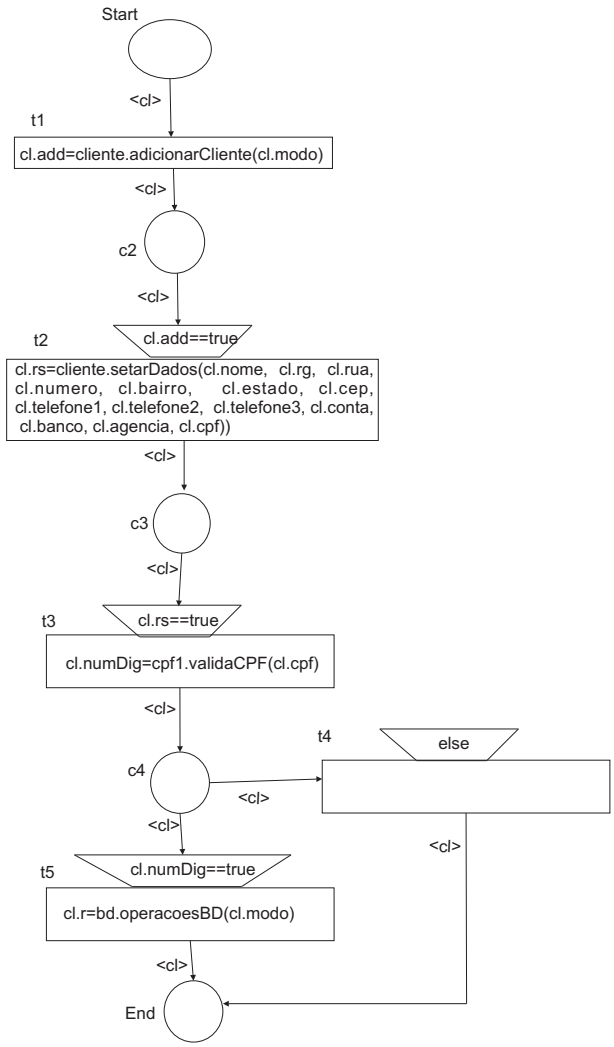


Figura 74: Implementação do Modelo de Especificação de Teste Funcional da Funcionalidade *Cadastrar Cliente*

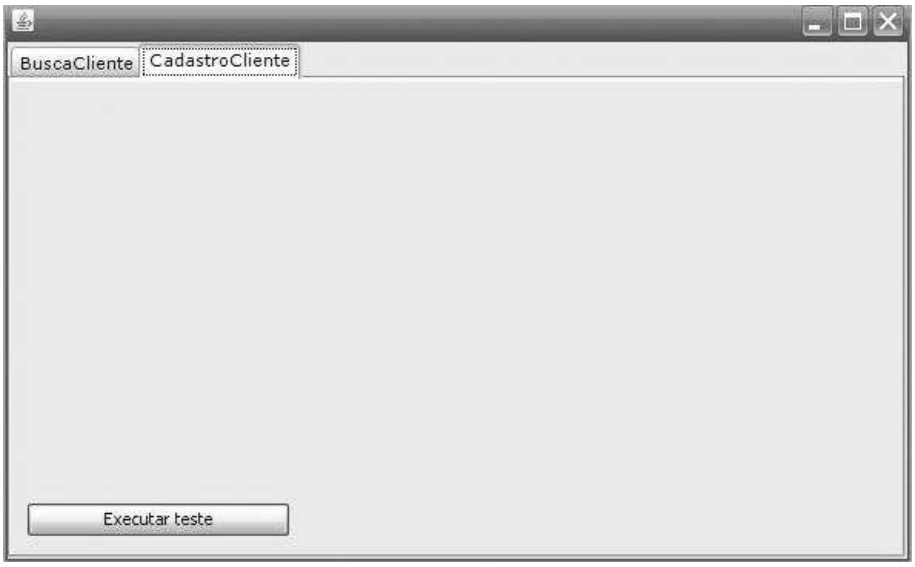


Figura 77: Interface Gráfica do Teste para a Operação *Cadastrar Cliente*

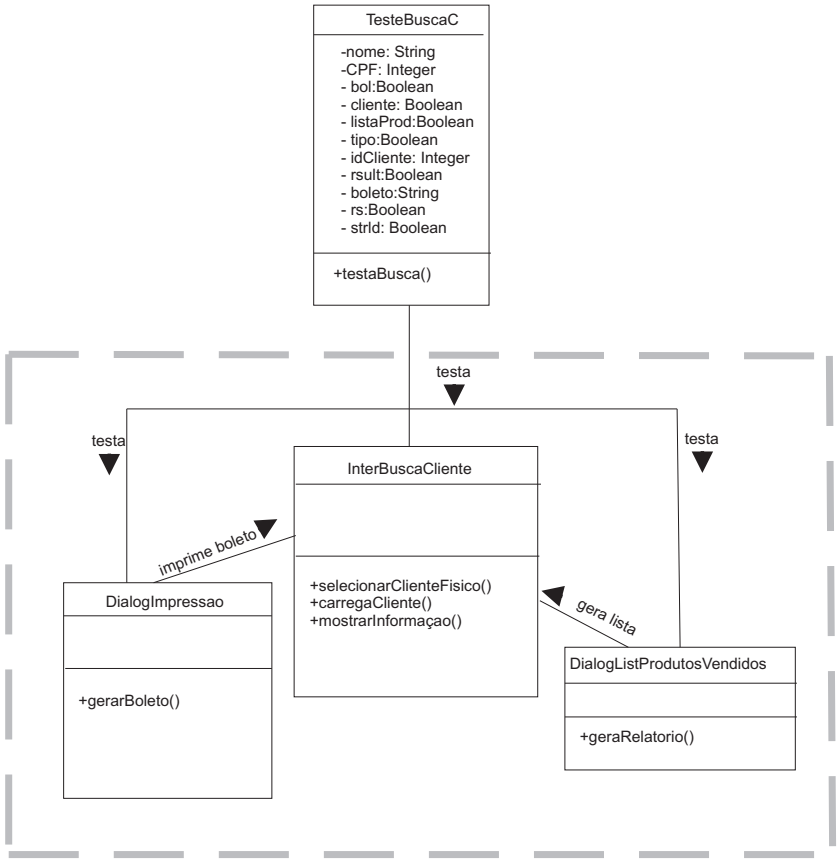


Figura 75: Classe de Teste *TesteBuscaC* Envolvendo as Principais Classes do Sistema de Controle de Vendas – *Buscar Cliente*

O modelo de execução do teste é dado pela WF-Net a objeto da Figura 76

- Cenário 1:

Os atributos do objeto *TestaCadCliente* são os seguintes:

*TesteCadCliente cliente*1;

- Dados de Entrada do Teste:

nome: Liliane;

cpf: 01429739126;

rg: 12234220;

rua: Rua 1;

numero: 34;

bairro: Nossa senhora de fatima;

estado: GO;

cep: 7500000;

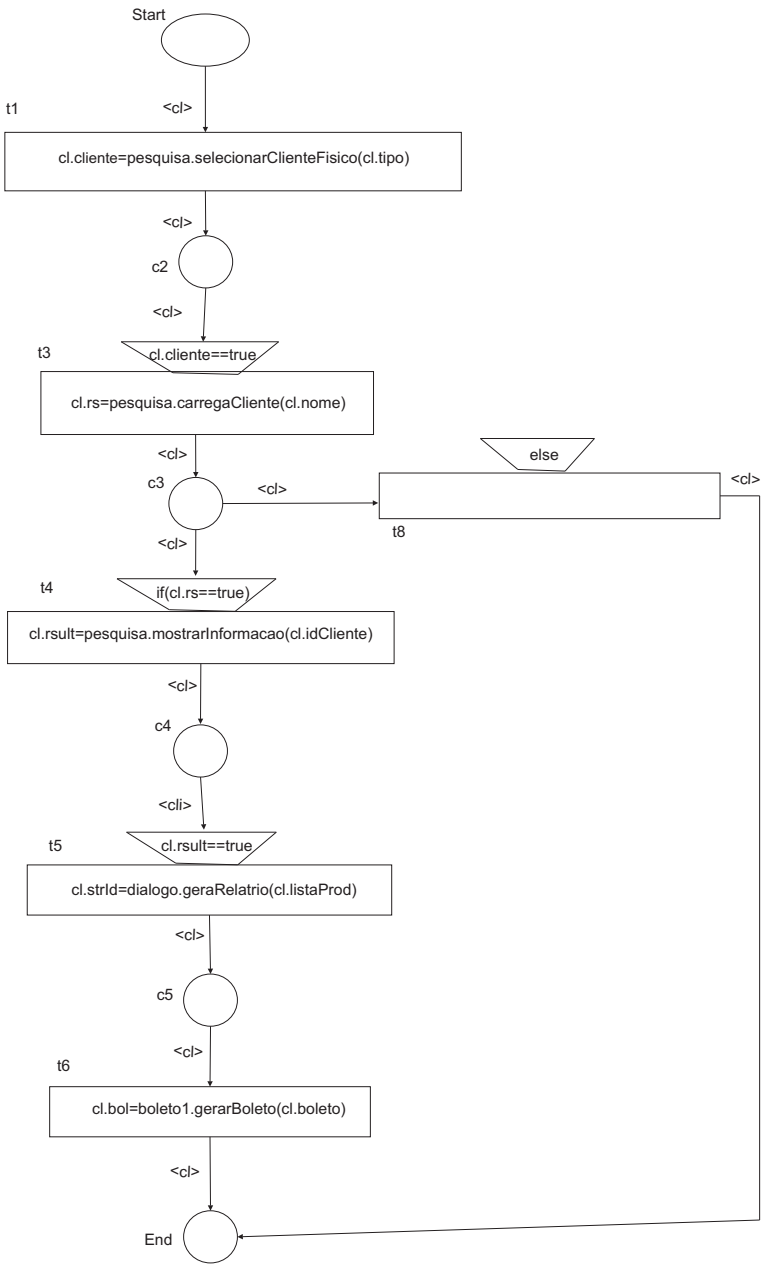


Figura 76: Implementação do Modelo de Especificação de Teste Funcional da Funcionalidade *Buscar Cliente*

telefone1: 6df4414100;;

telefone2: 6434414444;

telefone3: 5634410000;

conta: 192414;

banco: bb;

agencia: 03115;

r: 1;

modo: 1;

- Dados de Saída do Teste:

rs: false;

numDig: false;

add: false;

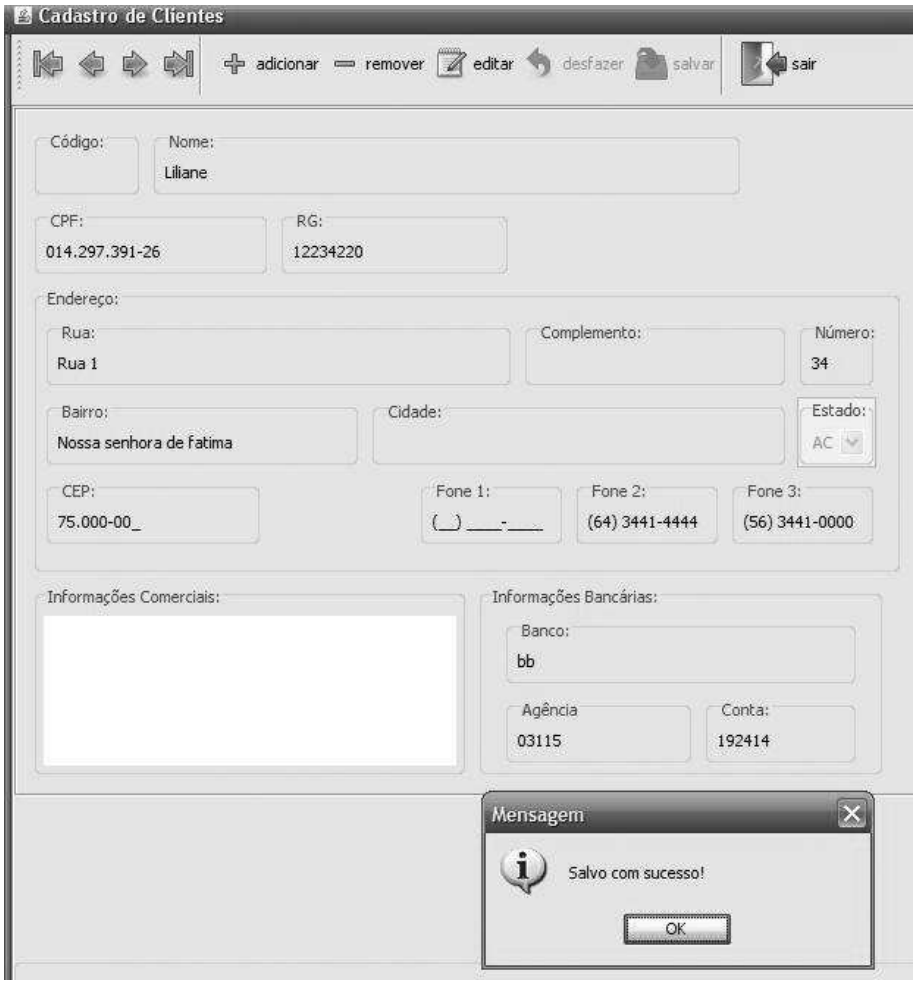


Figura 78: Cenário de Teste 1 - Resultado para o Teste do Campo Telefone da Função *Cadastrar Cliente*

Em particular, o valor de atributo de *telefone1* está inválido, pois contém caracteres que não são numéricos. No final da execução do teste, o sistema reconheceu o valor incorreto de telefone inserido, mas o cadastro do cliente foi aceito mesmo assim, apenas deixando o campo $fone1$ em branco, como mostra a Figura 78.

A execução dos testes, que possui o objeto de teste identificado por $cliente1$ é mostrado na Figura 79. A execução do teste para o *Cenário 1* de teste, é dado pela sequência de disparos seguintes:

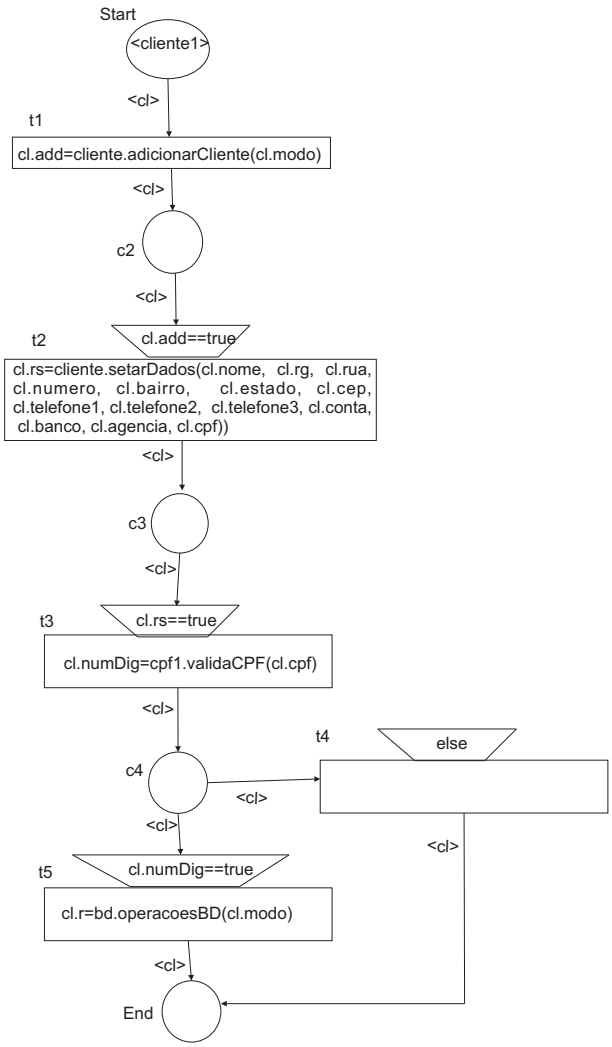


Figura 79: Execuçao do Cenário de Teste da Função *Cadastrar Cliente*

- a marcação inicial (o objeto *cliente*1 no lugar *Start*) sensibiliza a transição t_1, que não possui pré-condição. O disparo de t_1 chama o método *adicionarCliente* do objeto *cliente* da classe *CadCliente*, transmitindo, como valor de parâmetro, o atributo $modo = 1$ (valor que indica a operação de inserção de novos dados em um banco de dados). O método chamado ativa a tela de cadastros, devolvendo um valor booleano salvado no atributo *add* do objeto *cliente*1. Após o disparo de t_1, um novo objeto *cliente*1 (com valor do atributo $add = true$ modificado no final do disparo, que indica que a tela do cadastro foi ativada) é produzido no lugar c_2.

- com um objeto *cliente*1 em c_2, sensibiliza a transição t_2, que não possui pré-condição. O disparo de t_2 chama então o método *setarDados* do objeto *cliente* da classe *CadCliente*, transmitindo como valores de parâmetros os atributos $nome = Liliane$, $rg = 12234220$, $rua = Rua1$, $numero = 34$, *bairro=Nossa senhora de fatima*, $estado = 1$, $cep = 7500000$, $telefone1 = 6df4414100$, $telefone2 = 6434414444$,

$telefone3$ = 5634410000, $conta$ = 192414, $banco$ = bb, $agencia$ = 03115, cpf = 01429739126 (valores referentes ao cadastro do cliente). O método chamado recebe os dados de cada campo da interface do cliente cadastrado, devolvendo um valor booleano salvo no atributo rs do objeto $cliente1$. Após o disparo de t_2, um novo objeto $cliente1$ (com valor do atributo $rs = true$ modificado no final do disparo, que indica que os dados foram salvos) é produzido no lugar c_3.

- com um objeto $cliente1$ em c_3, sensibiliza a transição t_3, que não possui pré-condição. O disparo de t_3 chama então o método $validaCPF$ do objeto $cpf1$ da classe CPF, transmitindo, como valor de parâmetro, o atributo cpf = 01429739126 (valor do CPF informado pelo cliente). O método verifica a validade do CPF, devolvendo um valor booleano salvado no atributo $numDig$ do objeto $cliente1$. Após o disparo de t_3, um novo objeto $cliente1$ (com valor do atributo $numDig = true$ modificado no final do disparo, que indica que o cpf é válido) é produzido no lugar c_4.

- com um objeto $cliente1$ em c_4, as duas transições, t_5 e t_4, estão sensibilizadas. Mas, somente a pré-condição da transição t_5, que informa que o valor do CPF é válido, através do atributo $numDig = true$ é satisfeita. O disparo de t_5 chama então o método $operacoesBD$ do objeto bd da classe $operacoesBancoDados$, transmitindo, como valor de parâmetro, o atributo $modo = 1$ (valor do tipo de operação a ser efetuado no banco de dados). O método grava os dados no banco de dados, devolvendo um valor salvo no atributo r do objeto bd. Após o disparo de t_5, um novo objeto $cliente1$ (com valor do atributo $r = 1$ modificado no final do disparo, que indica que os dados foram salvos) é produzido no lugar End.

- Cenário 2:

Os atributos do objeto $cliente1$ são os seguintes:

$TesteCadCliente\ cliente1$;

- Dados de Entrada do Teste:
nome: Liliane;
cpf: 01429739126;
rg: 12234220;
rua: Rua 1;
numero: 34;
bairro: Nossa senhora de fatima;

estado: GO;
cep: dddddd;
telefone1: 6434414100;;
telefone2: 6434414444;
telefone3: 5634410000;
conta: 192414;
banco: bb;
agencia: 03115;
r: 1;
modo: 1

- Dados de Saída do Teste:
rs: false;
numDig: false;
add: false;

Figura 80: Caso de Teste 2 - Resultado para o Teste do Campo CEP da Função *Cadastrar Cliente*

Em particular, o valor de atributo de *cep* está inválido, pois contém caracteres que não são numéricos. No final da execução do teste, o sistema reconheceu o valor incorreto de CEP inserido, mas o cadastro do cliente foi aceito mesmo assim, apenas deixando o campo CEP em branco, como mostra a Figura 80.

- Cenário 3:

Os atributos do objeto *cliente*1 são os seguintes:

*TesteCadCliente cliente*1;

- Dados de Entrada do Teste:

nome: Liliane;

cpf: 45565yh;
rg: 12234220;
rua: Rua 1;
numero: 34;
bairro: Nossa senhora de fatima;
estado: GO;
cep: 7500000;
telefone1: 6434414100;;
telefone2: 6434414444;
telefone3: 5634410000;
conta: 192414;
banco: bb;
agencia: 03115;
r: 1;
modo: 1

- Dados de Saída do Teste:
rs: false;
numDig: false;
add: false;

Em particular, o valor de atributo de *cpf* está inválido, pois contém caracteres que não são numéricos. No final da execução do teste, o sistema reconheceu o valor incorreto de CPF inserido e o cadastro do usuário não foi aceito, deixando o campo CPF em branco como mostra a Figura 81.

A Figura 82 ilustra usando um Diagrama de Sequência, o resultado da execução do teste para a função *Cadastrar Cliente* para o *Cenário 1*.

Figura 81: Caso de Teste 3 - Resultado para o Teste do Campo CPF da Função *Cadastrar Cliente*

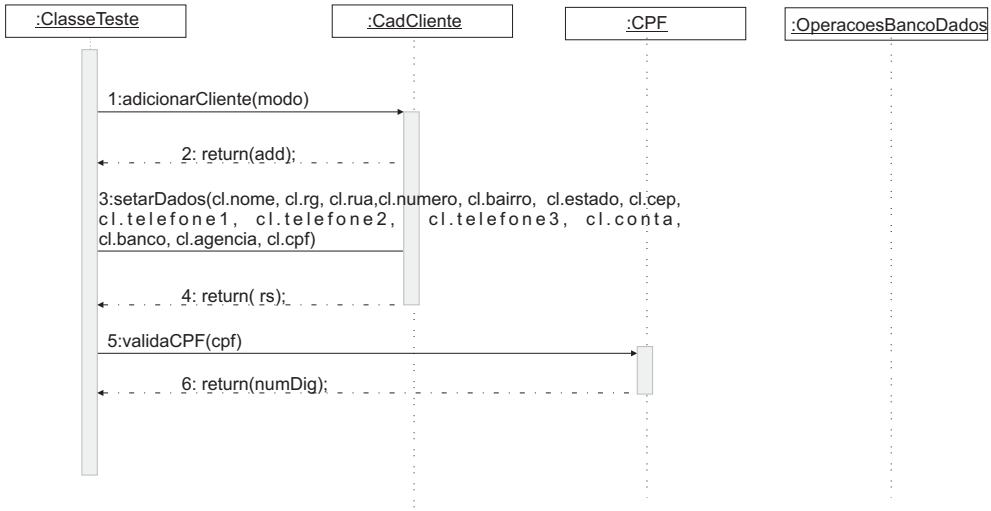


Figura 82: Diagrama de Sequência para a Operação *Cadastrar Cliente*

No final da execução do teste, o cadastro de cliente foi realizado com sucesso, embora

o campo correspodente a *Fone1* da Figura 78 do formulário permaneceu em branco.

5.7 Execução do Teste para a Funcionalidade *Buscar Cliente*

Nesta seção, é considerado o cenário da execução do teste da função *Buscar Cliente*. Neste contexto, dois cenários serão considerados.

- Cenário 1:

Os atributos do objeto *busca*1 são os seguintes:
$TesteBuscaC$ *busca*1;

- Dados de Entrada do Teste:
listaProd: false;
bol: false;
tipo: false;
idCliente: 9;
nome: Guilherme;
CPF: 22222222222;
boleto: false;

- Dados de Saída do Teste:
cliente: false;
rsult: false;
rs: false;
strld: false;

Em particular, o valor de atributo de *nome=Guilherme* está correto e, o valor para o atributo $cpf = 22222222222$ que corresponde ao campo *CPF/CNPJ* está correto. No final da execução do teste, um resultado foi obtido, pois o cliente *Guilherme* foi encontrado e, posteriormente, uma lista contendo os produtos comprados pelo cliente foi exibida, finalizando-se com a geração de um boleto bancário como mostra a Figura 83.

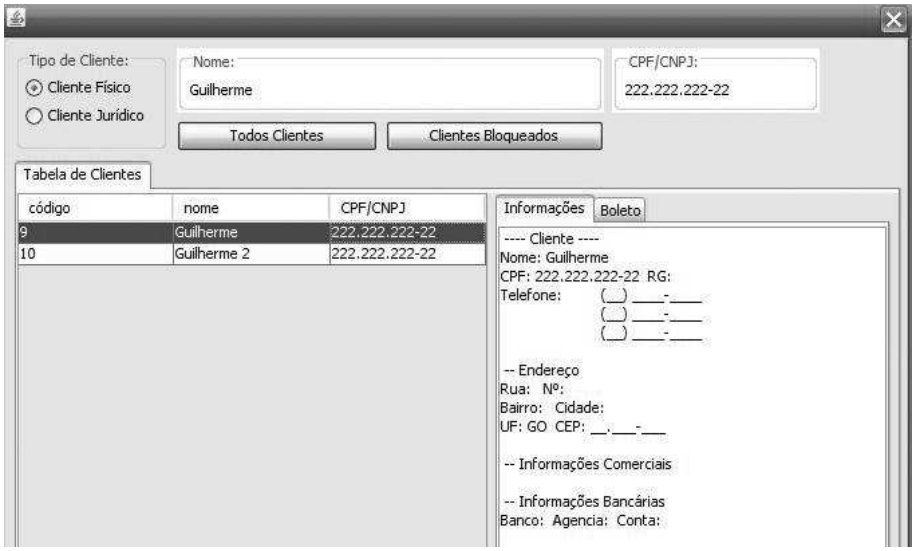


Figura 83: Cenário 1 de Teste - Resultado para o Teste da Função *Buscar Cliente*

A execução dos testes, que possui o objeto de teste identificado por *cliente*1 é mostrado na Figura 84. A execução do teste para o *Cenário 1* de teste, é dado pela sequência de disparos seguintes:

- a marcação inicial (o objeto *busca*1 no lugar *Start*) sensibiliza a transição t_1, que não possui pré-condição. O disparo de t_1 chama o método $selecionarClienteFisico$ do objeto *pesquisa* da classe *InterBuscaCliente*, transmitindo, como valor de parâmetro, o atributo $tipo = true$ (valor da opção ao ser selecionada o tipo de cliente). O método chamado adapta a tela de buscas, devolvendo um valor booleano salvo no atributo *cliente* do objeto *busca*1. Após o disparo de t_1, um novo objeto *busca*1 (com valor do atributo $cliente = true$ modificado no final do disparo, que indica que a tela de buscas para cliente físico foi ativada) é produzido no lugar c_2.

- com um objeto *busca*1 em c_2 sensibiliza a transição t_3, que não possui pré-condição. O disparo de t_3 chama então o método *carregaCliente* do objeto *pesquisa* da classe $InterBuscaCliente$, transmitindo como valores de parâmetros o atributo $nome = Guilherme$ (valor do nome a ser pesquisado). O método chamado permite buscar o nome do cliente desejado no banco de dados, devolvendo um valor salvo no atributo booleano rs do objeto *busca*1 que indica que o nome do cliente pesquisado foi encontrado. Após o disparo de t_3, um novo objeto *busca*1 (com valor do atributo $rs = true$ modificado no final do disparo, que indica que o nome foi encontrado) é produzido no lugar c_3.

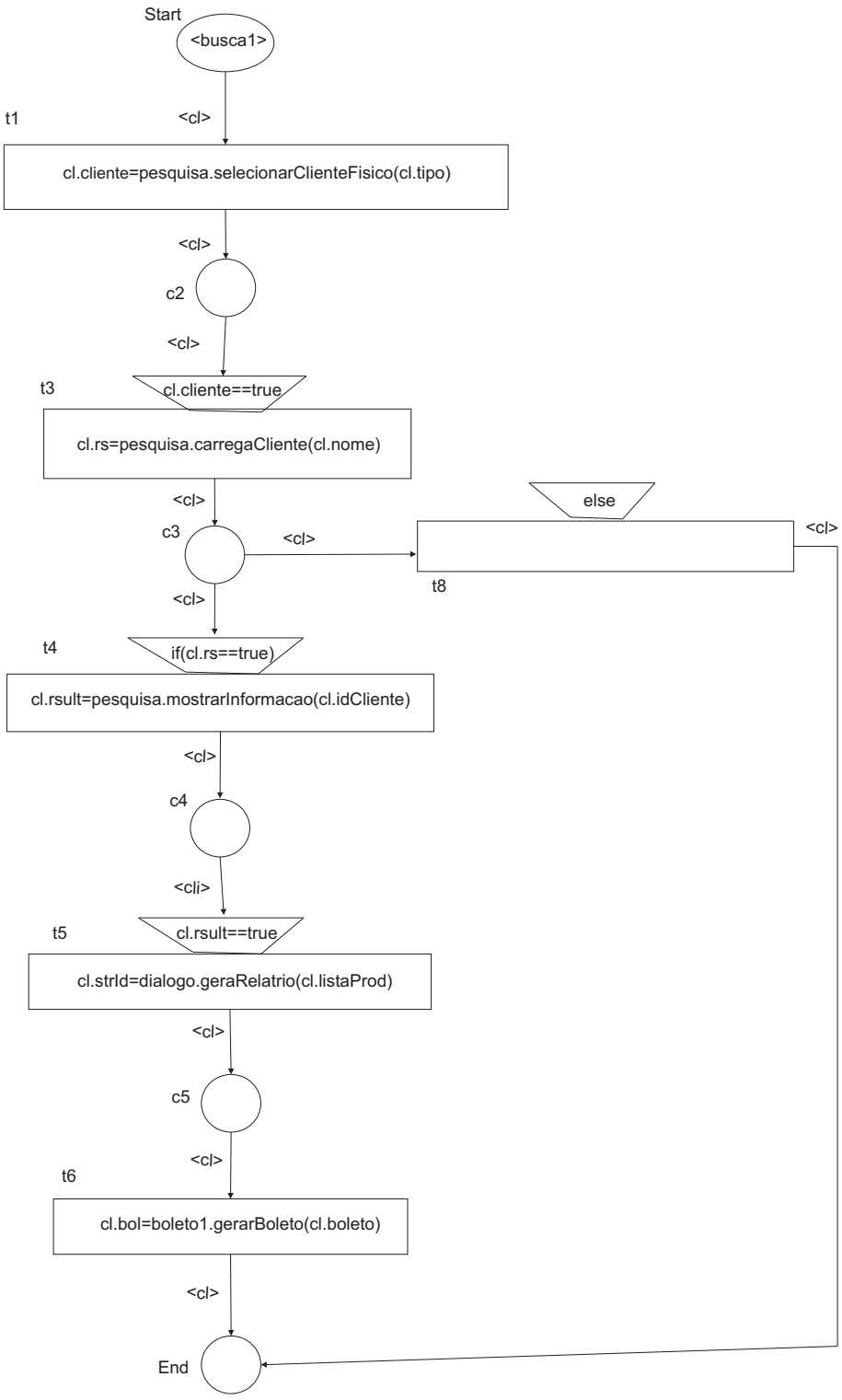


Figura 84: Execuçao do Cenário de Teste da Função *Buscar Cliente*

- com um objeto *busca*1 em c_3 as duas transições, t_4 e t_8, estão sensibilizadas. Mas, somente a pré-condição da transição t_4, que indica que um resultado foi encontrado através do atributo $rs = true$, é satisfeita. O disparo de t_4 chama então o método $mostraInformacao$ do objeto *pesquisa* da classe $InterBuscaCliente$, transmitindo

como valor de parâmetro o atributo *idCliente* (código do cliente pesquisado). O método chamado mostra as informações cadastrais do cliente, devolvendo um valor salvo no atributo $rsult$ do objeto $busca1$. Após o disparo de t_4, um novo objeto $busca1$ (com valor do atributo $rsult = true$ modificado no final do disparo, que exibe as informações cadastrais) é produzido no lugar c_4.

- com um objeto $busca1$ em c_4, sensibiliza a transição t_5, que não possui pré-condição. O disparo de t_5 chama então o método *geraRelatorio* do objeto *dialogo* da classe $DialogListProdutosVendidos$, transmitindo, como valor de parâmetro, o atributo $listaProd = true$ (valor do campo ao ser selecionado). O método chamado mostra mostra a lista de produtos comprados pelo cliente, devolvendo um valor *strld* do objeto $busca1$. Após o disparo de t_5, um novo objeto $busca1$ (com valor do atributo $strld = true$ modificado no final do disparo, que mostra as informações de produtos vendidos ao cliente pesquisado) é produzido no lugar c_5.

- com um objeto $busca1$ em c_5 sensibiliza a transição t_6 que não possui pré-condição. O disparo de t_6 chama então o método *gerarBoleto* do objeto $boleto1$ da classe $DialogImpressao$, transmitindo como valores de parâmetros o atributo $boleto= nome + CPF + idCliente + codVenda + valorTotal$ (valor do boleto a ser impresso). O método chamado gera o boleto para o cliente pesquisado efetuar o pagamento devolvendo um valor booleano para a geração do boleto salvado no atributo *strld* do objeto $busca1$. Após o disparo de t_6, um novo objeto $busca1$ (com valor do atributo $strld = true$ modificado no final do disparo, que gera o boleto de pagamento para o cliente) é produzido no lugar End.

- Cenário 2:

Os atributos do objeto $TesteBuscaC$ são os seguintes:

$TesteBuscaC$ $busca1$;

- Dados de Entrada do Teste:
listaProd: false;
bol: false;
tipo: false;
idCliente: 9;
nome: null;
CPF: 19483543754;

boleto: false;

- Dados de Saída do Teste:
cliente: false;
rsult: false;
rs: false;
strld: false;

Em particular, o valor de atributo de *nome=Guilherme* está nulo e, o valor para o atributo $cpf = 19483543754$ que corresponde ao campo *CPF/CNPJ* está inválido. No final da execução do teste, nenhum resultado foi obtido, uma vez que no cadastro dos clientes não é permitido cadastrar cliente com CPF inválido. A Figura 85 mostra o resultado do teste.

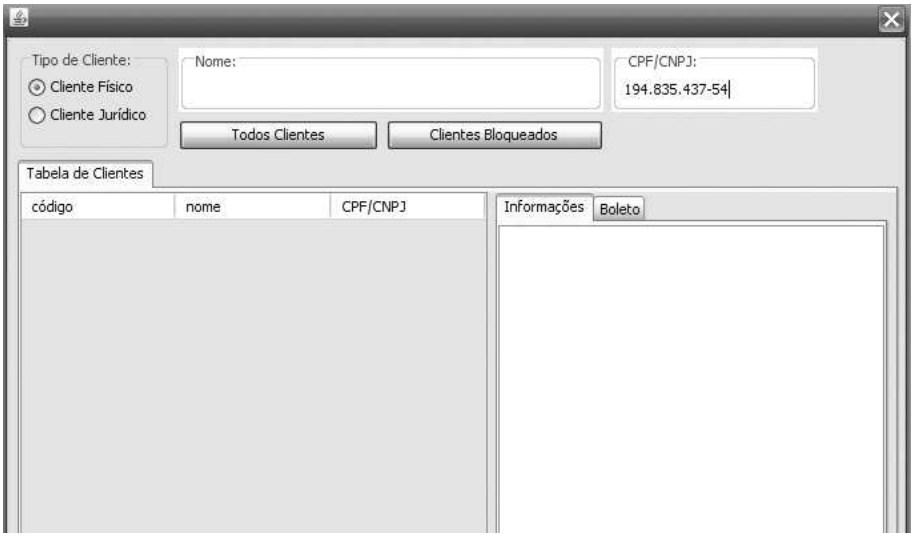


Figura 85: Caso de Teste 2 - Resultado para o Teste da Função *Buscar Cliente*

A Figura 86 ilustra usando um Diagrama de Sequência, o resultado da execução do teste para a função *Buscar Cliente* para o *Cenário 1.*

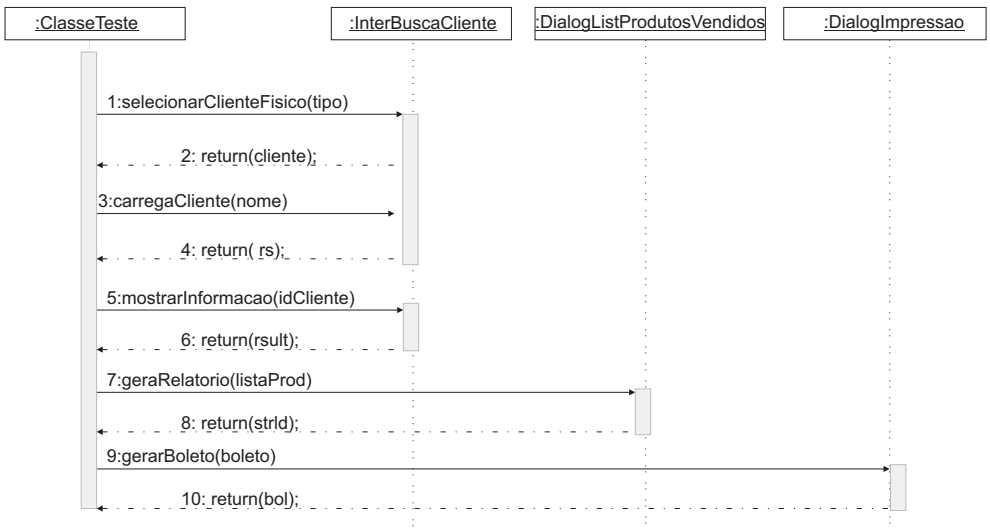


Figura 86: Diagrama de Sequência para a Operação *Buscar Cliente*

6 Conclusão

Neste trabalho, foi apresentada uma metodologia de desenvolvimento de modelos de teste funcional usando redes de Petri a Objetos e Workflow-Nets para arquiteturas de software orientadas a objetos, com o objetivo de oferecer uma solução para a informalidade em que a atividade de teste funcional se encontra até o presente momento.

Inicialmente, para a criação de modelos de teste funcional, seguiu-se o modelo de desenvolvimento em V. Na parte central do V, um modelo de especificação formal de teste foi obtido com base nos requisitos disponíveis na parte esquerda do V. Em seguida, o modelo de teste foi implementado considerando a parte direita do V na forma de uma instância de uma classe de teste, invocando-se os diversos métodos presentes na arquitetura do software.

O modelo de teste funcional proposto neste trabalho não é voltado somente para a verificação das funcionalidades do software do ponto de vista de ações externas do usuário, mas também pode ser usado para avaliar o comportamento interno de certas partes do software, como a atualização de um banco de dados, por exemplo, já que a classe de teste pode oferecer mecanismos para que o procedimento ocorra.

É importante salientar que para a obtenção do modelo de teste funcional, conceitos de processos de Workflow e redes de Petri foram abordadas, originando-se as Workflow-Nets subjacentes utilizando redes de Petri ordinárias e, por último, as Workflow-Nets a objetos que utilizam as redes de Petri a Objetos proposto por [Sibertin-Blanc, 1985]. É sabido que as redes de Petri devido ao formalismo que possuem são ideais para a modelagem de processos de Workflows pela capacidade de representarem formalmente os roteiros que existem quando se trata de processos de negócio. Diante desse fato, notou-se também que as características assumidas por um processo de Workflow permitiam obter o modelo de teste funcional, pois ambos possuiam um ciclo de vida com um início e um fim e a execução de várias operações no decorrer do ciclo.

O uso das redes de Petri neste trabalho foi um fator decisivo, pois permitiu-se repre-

sentar de modo formal os requisitos funcionais da arquitetura de softwares orientados a objetos. Outro ponto beneficiado pelo uso das redes de Petri a Objetos é a capacidade de representarem verdadeiras estruturas de dados em forma de atributos, além de estruturas de controle que sem dúvida favorecem a possibilidade de geração automática de classes de teste.

A utilização de diversas linguagens de modelagem, como os diagramas da UML, diagramas de fluxo de dados e os diagramas SA-RT foi relevante pois, permitiu ter diversas visões do sistema a fim de fornecer uma documentação suficientemente elaborada à geração dos modelos de testes funcionais. Assim, é sugerido ao desenvolvedor a utilização de diversas linguagens de modelagem, pois constitui um importante meio de encontrar falhas, ambiguidades ou omissões nos requisitos especificados.

Finalmente, após a apresentação da metodologia de geração de modelos de teste funcional, esta foi aplicada em um estudo de caso realizado em um software real orientado a objeto, mostrando assim, uma possibilidade de implementação dos modelos.

A principal contribuição deste trabalho é a de propor uma metodologia formal, usando uma linguagem de especificação formal como as redes de Petri a Objetos para a geração de modelos de teste funcional. Dessa forma, a atividade de teste funcional que até então era realizada essencialmente em meios informais com pouca ou nenhuma documentação disponível, passa a ter agora um meio a mais para a condução da atividade de teste funcional com um novo modelo formal que poderá auxiliar em diversos ambientes da atividade de teste de software. É claro que o trabalho apresentado se aplica a softwares cujas funcionalidades a serem testadas possuem uma estrutura semelhante a de processos de Workflow.

Como proposta de trabalhos futuros, será interessante estender os resultados da aboradagem proposta como por exemplo:

- criar um método que combine sequências de teste;
- criar um método que combine sequências de teste que não partem do estado inicial do software;
- implementar um compilador/intrepertador que gera automaticamente a classe de teste, a partir do modelo de especificação e de um editor de modelo;
- usar o modelo de teste em uma das diversas metodologias de teste funcional existentes, já que esse trabalho não constitui um método de teste funcional, mas

simplesmente a definição de um modelo de especificação que poderá ser usado em uma metodologia;

- propor o uso do modelo de teste em ISVV (*Independent Software Validation e Verification*).

Referências

[Aalst and Hee, 2002] Aalst, W. and Hee, K. (2002). *Workflow Management: Models, Methods and Systems*. The MIT Press Cambrige, Massachusetts. London Englad.

[Aalst, 1998] Aalst, v. d., W. (1998). Verification of workflow nets. *In S. Navathe and T. Wakayama, editors, Proceedings of the International Working Conference on Information and Process Integration in Enterprises (IPIC'96), Camebridge, Massachusetts.*

[Aertryck and Jensen, 2003] Aertryck, L. V. and Jensen, T. (2003). Uml-casting: Test synthesis from UML models using constraint resolution. *In Proc.AFADL – Approches Formelles dans Assistance au Dévelopment de Logiciel.*

[Ali et al., 2007] Ali, S., Briand, L. C., Rehman, M. J., Asghar, H., and Aamer, M. Z. (2007). A state-based approach to integration testing based on UML models. *Information and Software Technology, Volume 49 , Issue 11-12*, pages 1087–1106.

[Andrews et al., 2003] Andrews, A., France, R., Ghosh, S., and Craig, G. (2003). Test adequacy criteria for UML design models, software testing, verification and reliability. *In 13(2).*

[Barbey et al., 1996] Barbey, S., Buchs, D., and Péraire, C. (1996). A theory of specification-based testing for object-oriented software. *Proceedings of EDCC2 (European Dependable Computing Conference), Taormina, Italy.*

[Barbey et al., 2008] Barbey, S., Buchs, D., and Péraire, C. (2008). Satel - a test intention language for object oriented specifications of reactive systems. *PhD. Thesis, University of Geneva, Geneva,.*

[Barbey and Strohmeier, 1969] Barbey, S. and Strohmeier, A. (1969). The problematic of testing object-oriented software. *In 2 nd Conference on Software Quality Management.*

[Beizer, 1990] Beizer, B. (1990). *Black-Box Teting: Techiques for Functional Testing for Software and Systems*. Jhon Wiley and Sons, Inc., 2th edition.

[Belli et al., 2001] Belli, F., Budnik, C. J., and Hollmann, A. (2001). A holistic approach to testing of interactive systems using statecharts. pages 335 – 338.

[Bezerra, 2007] Bezerra, E. (2007). *Princípios de Análise e Projeto de Sistemas com UML*. Editora Campus, 2th edition.

[Binder, 1995] Binder, R. V. (1995). *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Editora Addison Wesley Longman.

[Blom et al., 2004] Blom, J., Hessel, A., Jonsson, B., and Pettersson, P. (2004). Specifying and generating test cases using observer automata. *Presentation at the International Workshop on Formal Approaches to Testing of Software.*

[Buchs et al., 1995] Buchs, D., Diagne, A., and Kordon, F. (1995). Testing prototypes validity to enhance code reuse. pages 141–148.

[Budkowski and Dembinski, 1987] Budkowski, S. and Dembinski, P. (1987). An introduction to stell: a specification language for distributed systems. *Computer Network and ISDN systems.*

[Cardoso and Valette, 1997] Cardoso, J. and Valette, R. (1997). *Redes de Petri.* DAUFSC, 1th edition.

[Cardoso et al., 1999] Cardoso, J., Valette, R., and Dubois, D. (1999). *Possibilistic Petri Nets.*

[Champagnat et al., 1998] Champagnat, R., Pingaud, H., and Alla (1998). A gas storage example as a bechmark for hybrid modeling. *APII-JESA, Special Issue on Automation of mixed process and hybrid dynamical systems. Vol 32, N. 9-10*, pages 1233–1253.

[Clark, 1983] Clark, L. (1983). A system to generate test data and symbolically execute programs. *IEEE Transaction on Software Engineering.*

[Clarke and Lee, 1997] Clarke, D. and Lee, I. (1997). Automatic generation of tests for timing constraints from requirements. *In Proceedings of the Third International Workshop on Object-Oriented Real-Time Dependable Systems.*

[Correia and Silva, 2004] Correia, S. A. and Silva, A. R. (2004). Técnicas para construção de testes funcionais automáticos. *5ª Conferência para a Qualidade nas Tecnologias da Informação e Comunicações (QUATIC'2004), Instituto Português de Qualidade.*

[David and H., 2004] David, R. and H., A. (2004). *Discrete, Continuous, and Hybrid Petri Nets.* Springer.

[Delamaro et al., 2007] Delamaro, M. E., Maldonado, J. C., and Jino, M. (2007). *Introdução ao Teste de Software.* Editora Campus, 1th edition.

[DeMarco, 1979] DeMarco, T. (1979). *Structured Analysis and System Specification.* Prentice Hall, ISBN 0138543801.

[Desel et al., 1997] Desel, J., Oberweis, A., Zimmer, T., and G., Z. (1997). A test case generator for the validation of high-level petri nets. *Emerging Technologies and Factory Automation Proceedings 6th International Conference on*, pages 327 – 332.

[Dinh-Trong et al., 2006] Dinh-Trong, T. T., Ghosh, S., and B., F. R. (2006). A systematic approach to generate inputs to test UML design models. *Software Reliability Engineering. ISSRE apos;06. 17th International Symposium on*, pages 95 – 104.

[Fabbri, 2001] Fabbri, S. (2001). Uma família de critérios de teste para validação de sistemas especificados em stelle. *Dissertação de Mestrado, Universidade de São Paulo. São Carlos, São Paulo.*

[Fantinato and Jino, 2000] Fantinato, M. and Jino, M. (2000). Appliying extended finite state and scenarios in software testing. *Springer Berlin / Heidelberg, Lecture Notes in Computer Science*, pages 109 – 131.

[Figueiredo and Machado, 2004] Figueiredo, A. L. L. and Machado, P. D. L. (2004). Geração automática de casos de teste para sistemas baseados em agentes móveis. *In: Workshop de Teses e Dissertações em Qualidade de Software*, pages 1 – 6.

[FIGUEIREDO et al., 2007] FIGUEIREDO, J. C. A., BARBOSA, D. L., MACHADO, P. D. L., ANDRADE, W. L., LIMA, H. S., and Makelli, A. J. (2007). Automating functional testing of components from UML specifications. *International Journal of Software Engineering and Knowledge Engineering*, pages 339–358.

[Fin and Fummi, 2000] Fin, A. and Fummi, F. (2000). A vhdl error simulator for functional test generation. *Proceedings of the conference on Design, automation and test in Europe*, pages 390 – 395.

[Florin and Natkin, 2000] Florin, G. and Natkin, S. (2000). Définition formelle des réseaux de petri stochstiques. *Technical Report CNAM - Paris. Springer-Verlag.*

[Fummi et al., 1995] Fummi, F., Sciuto, D., and Serra, M. (1995). Test pattern embedding in sequential circuits through cellular automata. *Proc. International Conference on VLSI.*

[Gane and T., 1983] Gane, C. and T., S. (1983). *Análise Estruturada de Sistemas.* LTC Editora, Rio de Janeiro.

[Genrich, 1987] Genrich, H. J. (1987). Predicate-transition nets. *Lecture in Notes in Computer Sciense, Springer-Verlag.*

[Georgieva and Gancheva, 2003] Georgieva, J. and Gancheva, V. (2003). Functional testing of object-oriented software. *International Conference Computer Systems and Technologies archive Proceedings of the 4th international conference conference on Computer systems and technologies: e-Learning*, pages 141 – 146.

[Hartmann et al., 2000] Hartmann, J., Imoberdof, C., and Meisenger, M. (2000). UML based integration testing. *in ISSTA conference proceeding, Portland, Oregon*, pages 60 – 70.

[Hatley and Pirbhai, 1987] Hatley, D. J. and Pirbhai, I. A. (1987). Strategies for real-time system specification. *Dorset House Publishing New York.*

[Hatley and Pirbhai, 1989] Hatley, D. J. and Pirbhai, I. A. (1989). Petri nets: Properties, analysis ans applications. *Procedings of the IEEE 77 (4)*, pages 541 – 580.

[Hetzel and Hetzel, 1991] Hetzel, W. C. and Hetzel, B. (1991). *The Complete Guide to Software Testing.*

[Hong et al., 1995] Hong, S., Know, Y. R., and Cha, S. D. (1995). Testing of object-oriented programs based on finite states machines. *In Proceedings of 1995 Asia Pacific Software Egeneering Conference. IEEE Computer Society Press, Los Alamitos, California.*

[Jee et al., 2005] Jee, E., Yoo, J., and Cha, S. (2005). *Control and Data Flow Testing on Function Block Diagrams - Computer Safety, Reliability, and Security.* Springer Berlin / Heidelberg.

[Jensen, 1990] Jensen, K. (1990). Coloured petri nets: A high level language or system desing and analysis. *Advances in Petri Nets (G. Rozemberg, Ed), Lecture Notes in Computer Science. Spring Verlag.*

[Kervinen and Virolainen, 2005] Kervinen, A. and Virolainen, P. (2005). Heuristics for faster error detection with automated black box testing. *Electronic Notes in Theoretical Computer Science*, pages 53 – 71.

[Kim et al., 1994] Kim, Y. G., Hong, H. S., Cho, S. M., Bae, D. H., and Cha, S. D. (1994). Test cases generation from UML state diagrams. *In IEE Proceedings: Software.*

[Kim et al., 1999] Kim, Y. G., Hong, H. S., H., B. D., and D., C. S. (1999). Test cases generation from UML state diagrams. *IEE Proceedings online no. 19990602.*

[Koné, 2000] Koné, O. (2000). A local approach to the testing of real-time of real-time systems. *British Computer Society; Vandoeuvre les Nancy, France.*

[Linkman et al., 2003] Linkman, S., Vicenzi, A. M. R., and Maldonado, J. (2003). An evaluation of systematic functional testing using mutation testing. *In: 7th International Conference on Empirical Assessment in Software Engineering - EASE, Keele, UK.*

[Martena et al., 2002] Martena, V. ., Orso, A., and Pezzè, M. (2002). Interclass testing of object oriented software. *Proceedings of the Eighth IEEE international Conference on Engineering of Complex Computer Systems.*

[Meinke, 2004] Meinke, K. (2004). Automated black-box testing of functional correctness using function approximation. *Proc. ACM SIGSOFT Int. Symp. on Software Testing and Analysis, Software Engineering Notes 29 (4)*, pages 143–153.

[Meyer, 1997] Meyer, B. (1997). *Object-Oriented Software Construction.* PRENTICE HALL, 2th edition.

[Mingsong et al., 2006] Mingsong, C., Xiaokang, Q., and Xuandong, L. (2006). Automatic test case generation for UML activity diagrams. *International Conference on Software Engineering archive Proceedings of the international workshop on Automation of software test*, pages 2 – 8.

[Moncelet et al., 1998] Moncelet, G., Christensen, S., and Paudeto, M. (1998). Dependedability evaluation a simple mechatronic using coloured petri nets. *Workshop on pratical use coloured Petri Nets and Desing CPN.*

[Myers et al., 2004] Myers, J. G., Sandler, C., Badgett, T., and Thomas, T. M. (2004). *The Art of Software Testing.*

[Peterson, 1981] Peterson, J. L. (1981). *Petri Nets Theory and the Modelling of Systems.* Prentice Hall.

[Petri, 1966] Petri, C. A. (1966). Communication with automata. *Technical Report RADCTR-65-377. Darmstadt University.*

[Pfleeger, 1999] Pfleeger, S. L. (1999). *Engenharia de Software. Teoria e Prática.* Prentice Hall.

[Pressman, 2001] Pressman, R. (2001). *Engenharia de Software*. McGrawHill, 4th edition.

[Rajan, 2006] Rajan, A. (2006). Coverage metrics to measure adequacy of black-box test suites. *Automated Software Engineering. IEEE/ACM International Conference on Volume , Issue*, pages 335 – 338.

[Ramchandani, 1974] Ramchandani, C. (1974). Analysis of assynchrounous concurrent systems by timed petri nets models. *PhD Thesis. MIT. Project MAC TR-120.*

[Reid, 1997] Reid, S. C. (1997). An empirical analysis of equivalence partitioning, boundary valueanalysis and random testing. *Software Metrics Symposium, 1997. Proceedings., Fourth International Volume , Issue , 5-7*, pages 64 – 73.

[Reza et al., 2007] Reza, H., Endapally, S., and Grant, E. (2007). A model-based approach for testing gui using hierarchical predicate transition nets. *Information Technology, 2007. ITNG apos;07. Fourth International Conference on*, pages 366 – 370.

[Roper, 1994] Roper, M. (1994). *Software Testing.*

[Ryser and Glinz, 1999] Ryser, J. and Glinz, M. (1999). Scent - scenario-based validation and test of software. *Technical Report 2000.03, Institut für Informatik, University of Zurich.*

[Samuel et al., 2006] Samuel, P., Mall, R., and Kanth, P. (2006). Automatic test case generation from UML communication diagrams. *Information and Software Technology.*

[Schroeder et al., 2002] Schroeder, P. J., Faherty, P., and Korel, B. (2002). Generating expected results for automated black-box testing. *Proceedings of the 17 th IEEE International Conference on Automated Software Engineering (ASE'02).*

[Sibertin-Blanc, 1985] Sibertin-Blanc, C. (1985). High level petri nets with data structure. *6th European Workshop in Application and Theory of Petri Nets. Espan, june.*

[Sibertin-Blanc, 1994] Sibertin-Blanc, C. (1994). Cooperative nets. *In: Valette, R. (Ed.), 15th International Conference on Application and Theory of Petri Nets, Zaragosa, Lecture Notes in Computer Science, vol. 815, Springer*, pages 377–396.

[Sibertin-Blanc, 2001] Sibertin-Blanc, C. (2001). Cooperative objects: Principles, use and implementation. *Concurrent Object-Oriented Programming and Petri Nets, G. Agha and F. De Cindio (Eds)., Lecture Note in Computer Science , Springer-Verlag.*

[Sifakis, 1977] Sifakis, J. (1977). Use of petri nets for performance evaluation. *Internactional Symposium on Modelling and Performance Evaluation of Computer Systems.*

[Simão, 2000] Simão, A. S. (2000). Proteum-rs/pn: Uma ferramenta para validação de redes de petri baseada na análise de mutantes. *Dissertação de Mestrado, Universidade de São Paulo, São Carlos, Brasil.*

[Sloane and V., 2004] Sloane, E. B. and V., G. (2004). Applications of the petri net to simulate, test, and validate the performance and safety of complex, heterogeneous, multi-modality patient monitoring alarm systems. *Proceedings of the 26th Annual International Conference of the IEEE EMBS. San Francisco, CA, USA.*

[Smith and Robson, 1990] Smith, M. D. and Robson, D. J. (1990). Object-oriented - the problems of validation. *In: VI International Conference on Software Maintenance. IEEE Computer Society.*

[Souza, 2000] Souza, S. R. (2000). Validação de especificações de sistemas reativos: Definição e análise de critérios de teste. *Tese de Doutorado, Universidade Federal de São Paulo. São Carlos, Brasil.*

[Störrle, 2005] Störrle, H. (2005). Semantics and verification of data flow in UML 2.0 activities. *Electronic Notes in Theoretical Computer Science.*

[Sugeta, 1999] Sugeta, T. (1999). Proteum-rs/st: Uma ferramenta para apoiar a validação de especificação de statecharts na análise de mutantes. *Dissertação de Mestrado, Universidade de São Paulo. São Carlos, São Paulo.*

[Summerville, 2003] Summerville, I. (2003). *Engenharia de Software.* Editora Addison Wesley, 6th edition.

[Supavita and Suwannasart, 2005] Supavita, S. and Suwannasart, T. (2005). Testing polymorphic interactions in UML sequence diagrams. *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05).*

[Tahat et al., 2001] Tahat, L., Vaysburg, B., Korel, B., and Bader, A. (2001). Requirement-based automated black-box test generation. *25th Annual Int'l Computer Software and Applications Conference, Chicago, Illinois.*

[Tazza, 1987] Tazza, M. (1987). Quantitative analysis of a resource allocation problem: a net theory based proposal. *Concorrency and Nets - Springer-Verlage*, pages 511 – 532.

[Vilkomir et al., 2003] Vilkomir, S. A., Kapoor, K., and P., B. J. (2003). Tolerance of control-flow testing criteria. *Proceedings of the 27th Annual International Conference on Computer Software and Applications table of contents.*

[Wang and Liu, 1999] Wang, C. J. and Liu, M. T. (1999). Using a petri net model approach to object-oriented class testing. *Systems, Man, and Cybernetics. IEEE SMC '99 Conference Proceedings. IEEE International Conference on, vol.1.*, pages 824–828.

[Ward and Mellor, 1985] Ward, P. T. and Mellor, S. J. (1985). Structured development for real systems. *Yourdon Press, Prentice Hall.*

[Wegener and Grochtmann, 2004] Wegener, J. and Grochtmann, M. (2004). *Verifying Timing Constraints of Real-Time Systems by Means of Evolutionary Testing.* Springer Netherlands.

[Willians, 2000] Willians, L. (2000). Mutation testing. *TechnicalNew York, United Satates.*

[Yamada et al., 2000] Yamada, M., Mori, T., Fukada, A., Nakata, A., and Higashino, T. (2000). A method for functional testing of media synchronization protocols - japan. *Joho Shori Gakkai Kenkyu Hokoku*, pages 13–18.

[Zeng et al., 2001] Zeng, Z., Ciesielski, M. J., and Rouzeyre, B. (2001). Functional test generation using constraint logic programming. *IFIP Conference Proceedings; Vol. 218 archive Proceedings of the IFIP TC10/WG10.5 Eleventh International Conference on Very Large Scale Integration of Systems-on/Chip: SOC Design*, pages 375 – 387.

[Zhu and He, 2000a] Zhu, H. and He, X. (2000a). A theory of testing high level petri nets. *In: Proc. of Conference on Software: Theory and Pracice (The 16th IFIP World Computer Congress), Beijing, China*, pages 443–450.

[Zhu and He, 2000b] Zhu, H. and He, X. (2000b). A theory of testing high level petri nets. *Technical Report, School of Computing and Mathematical Sciences, Oxford Brookes University. (Submitted to IFIP World Computer Congress).*