

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO



FEUP

Simulação de Sistemas de Produção Lean

António Pedro Alves Pereira

Tese submetida no âmbito do
Mestrado Integrado em Engenharia Electrotécnica e de Computadores
Major de Automação

Orientador: José António Rodrigues Pereira de Faria (Prof. Dr.)

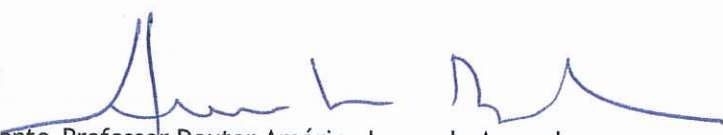
Fevereiro de 2009

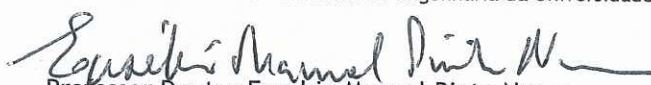
A Dissertação intitulada

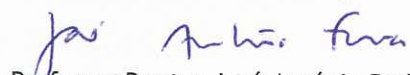
“SIMULAÇÃO DE SISTEMAS DE PRODUÇÃO LEAN”

foi aprovada em provas realizadas em 27/Fevereiro/2009

o júri

presidente 
Professor Doutor Américo Lopes de Azevedo
Professor Associado da Faculdade de Engenharia da Universidade do Porto


Professor Doutor Eusebio Manuel Pinto Nunes
Professor Auxiliar da Universidade do Minho


Professor Doutor José António Rodrigues Pereira de Faria
Professor Auxiliar da Faculdade de Engenharia da Universidade do Porto

O autor declara que a presente dissertação (ou relatório de projecto) é da sua exclusiva autoria e foi escrita sem qualquer apoio externo não explicitamente autorizado. Os resultados, ideias, parágrafos, ou outros extractos tomados de ou inspirados em trabalhos de outros autores, e demais referências bibliográficas usadas, são correctamente citados

Autor - António Pedro Alves Pereira



Faculdade de Engenharia da Universidade do Porto

Resumo

Com a globalização dos mercados e sistemas cada vez mais complexos, surge nas empresas e organizações de hoje uma exigência crescente em inovação e melhoria contínua dos seus produtos e serviços. No entanto, nem sempre é fácil, recomendável ou mesmo possível a implementação de novos sistemas sem estes serem sujeitos a uma validação prévia que prove que o mesmo vai oferecer melhorias face ao anterior. Desta forma, a simulação surge como uma ferramenta capaz de analisar e avaliar tanto situações actuais como futuras, tornando-se assim numa ajuda poderosa para qualquer decisor.

O trabalho desenvolvido nesta dissertação enquadra-se na área dos sistemas de produção, em particular aqueles com influências da filosofia *Lean Manufacturing*. Este projecto vem no seguimento de outros já realizados e cujo principal objectivo era o conhecimento da ferramenta de simulação AnyLogic e que também é a usada no trabalho desenvolvido.

O objectivo desta dissertação era o desenvolvimento de uma biblioteca de componentes parametrizáveis e flexíveis capazes de simular os diferentes tipos de subsistemas encontrados nos Sistemas de Produção *Lean* (e.g., supermercados, *milkruns*, etc.), criando assim uma ferramenta com potencialidades de utilização em trabalhos futuros. Para tal foi realizado um breve estudo dos conceitos de simulação e *Lean*.

O desenvolvimento deste trabalho foi então a programação no *AnyLogic*, tanto da lógica de funcionamento dos componentes, como da animação dos mesmos.

Durante a realização deste projecto foram modelados alguns sistemas de produção de forma a validar os componentes criados e exemplificar a sua utilização.

No último capítulo são tiradas algumas conclusões e feitas algumas reflexões sobre a ferramenta utilizada no desenvolvimento do trabalho, sobre os componentes criados, respectiva utilidade em aplicações reais e melhorias em trabalhos futuros.

Abstract

With the globalization of markets and the ever increasing complexity of systems, rises in today's companies and organizations a growing demand for innovation and continuous improvement of their products and services. However, it isn't always easy or possible to change existing systems without some kind of validation that proves that the new system offers better results than the current one. Therefore, simulation emerges as a tool capable of analyzing and evaluating current and future situations, becoming a powerful tool for any decision maker.

The work developed in this dissertation relates to the production systems' area, in particular those influenced by Lean Manufacturing concepts.

This project follows in the trail given by several works done with the AnyLogic simulation tool, and whose purpose was to deepen the understanding of its capabilities and potential.

The objective of this dissertation was to develop a library of flexible and parameterizable objects capable of simulating the different types of subsystems found in Lean Production Systems (e.g., supermarkets, milkruns, etc.), thus creating a tool with enough depth and flexibility to be used in future projects.

Therefore, a brief study about simulation and Lean concepts was made.

The development of this work was the programming of both logic and visual presentation of the library objects to be used in AnyLogic.

During the project, a few production systems were modeled in order to validate the library objects created, as well as, showing their usability through exemplification.

In the end some conclusions are made about the simulation tool used, the library objects created and their usability in real applications, as well as some reflexions on possible improvements in future projects.

Agradecimentos

À Andreia, pelo apoio incondicional, conselhos e paciência inesgotável.

À minha família, pela formação que me deram.

Ao meu orientador, pela disponibilidade e correcções preciosas.

Aos meus amigos, por todas as longas horas de trabalho.

Conteúdo

1	Introdução	1
1.1	Enquadramento	1
1.2	Objectivos	2
1.3	Metodologia	2
1.4	Estrutura da Dissertação	2
2	Simulação e <i>Lean Manufacturing</i>	5
2.1	Simulação	5
2.1.1	Definição e Conceitos	5
2.1.2	Modelos de Simulação	6
2.1.3	Paradigmas de Simulação	7
2.1.4	Ferramentas e Tecnologias de Simulação	8
2.2	<i>Lean</i>	9
2.2.1	Definição e Princípios	10
2.2.2	Os <i>Muda</i>	11
2.2.3	Conceitos e Técnicas	12
3	<i>AnyLogic</i>	15
3.1	Introdução	15
3.1.1	As Bases do <i>AnyLogic</i>	16
3.1.2	Ambiente de Modelação	16
3.2	<i>Enterprise Library</i>	18
3.2.1	<i>Source</i> e <i>Sink</i>	20
3.2.2	<i>Queue</i>	21
3.2.3	<i>Delay</i>	22
3.2.4	<i>Enter</i> e <i>Exit</i>	22
3.2.5	<i>Hold</i>	23
3.2.6	<i>Select Output</i>	23
4	Biblioteca de Componentes	25
4.1	Introdução	25
4.2	Classes Java	26
4.2.1	Box	26
4.2.2	Kanban	27
4.3	Active Objects Criados	28
4.3.1	Buffer	31
4.3.2	Supermarket	34
4.3.3	Workstation	36

4.3.4	Milkrun	43
4.4	Base de Dados	51
5	Validação	55
5.1	Exemplos de Aplicação	55
5.1.1	Introdução	55
5.1.2	Exemplo A	56
5.1.3	Exemplo B	57
5.2	Cenários de Simulação	61
5.3	Conclusões da Simulação	62
6	Conclusões e Trabalho Futuro	65
6.1	Conclusões	65
6.2	Trabalho Futuro	67
	Referências	68

Lista de Figuras

2.1	Esquema representativo de um estudo de simulação	6
2.2	Diferentes paradigmas para modelação de sistemas	7
2.3	Conceito de <i>Jidoka</i>	10
3.1	Ambiente de Modelação do <i>AnyLogic</i>	17
3.2	Objectos da <i>Enterprise Library</i>	18
3.3	Definição da classe <i>Customer</i>	19
3.4	Modelo <i>Bank</i>	20
3.5	<i>Source</i> e <i>Sink</i>	20
3.6	<i>Queue</i>	21
3.7	<i>Delay</i>	22
3.8	<i>Enter</i> e <i>Exit</i>	23
3.9	<i>Hold</i>	23
3.10	<i>Select Output</i>	23
4.1	Diagrama UML das subclasses <i>Box</i> e <i>Kanban</i>	26
4.2	Classe Java - <i>Box</i>	27
4.3	Classe Java - <i>Kanban</i>	27
4.4	Alteração dos parâmetros de um <i>Buffer</i> através do ambiente gráfico	29
4.5	Parâmetro dinâmico	29
4.6	Exemplo de um objecto replicado	30
4.7	Ícone existente por defeito (à esquerda) e ícone criado para o <i>AO Milkrun</i> (à direita)	31
4.8	Lógica do <i>Buffer</i>	31
4.9	Ícone do <i>Buffer</i> em <i>runtime</i>	33
4.10	Função <i>toString</i> personalizada para o <i>AO Buffer</i>	33
4.11	Animação na fase de modelação e durante a simulação do <i>Buffer</i>	34
4.12	Lógica do <i>Supermarket</i>	34
4.13	Ícone do <i>Supermarket</i> em <i>runtime</i>	36
4.14	Função <i>toString</i> personalizada para o <i>AO Supermarket</i>	36
4.15	Animação na fase de modelação e durante a simulação do <i>Supermarket</i>	37
4.16	Lógica do <i>Workstation</i>	37
4.17	Atributos do <i>Workstation</i>	39
4.18	Métodos do <i>Workstation</i>	41
4.19	Ícone do <i>Workstation</i> em <i>runtime</i>	42
4.20	Função <i>toString</i> personalizada para o <i>AO Workstation</i>	43
4.21	Animação na fase de modelação e durante a simulação do <i>Workstation</i>	44
4.22	Máquina de estados do funcionamento do <i>Milkrun</i>	45
4.23	Lógica do <i>Milkrun</i>	46

4.24	<i>pathMilkrun</i> e <i>taskMilkrun</i>	47
4.25	Atributos do <i>Milkrun</i>	48
4.26	Métodos do <i>Milkrun</i>	49
4.27	Ícone do <i>Milkrun</i> em <i>runtime</i>	50
4.28	Função <i>toString</i> personalizada para o <i>AO Milkrun</i>	51
4.29	Animação na fase de modelação e durante a simulação do <i>Milkrun</i>	52
4.30	Exemplo de animação externa envolvendo um <i>milkrun</i>	52
4.31	Exemplo de código de acesso à BD	53
5.1	Exemplo A	56
5.2	Organização do Departamento de Lixamento e Polimento na Unidade de Produção	57
5.3	Exemplo B	58
5.4	Animação do caso de estudo usando o cenário 1	59
5.5	Estatísticas do supermercado <i>SM_intermédio</i> do caso de estudo	60
5.6	Animação dos <i>AOs WS_Presentation_Lixamento</i> , <i>WS_Presentation_Polimento</i> e <i>SM_Presentation</i>	61
5.7	Cenário 2 de simulação	62

Lista de Tabelas

4.1	Tabela de atributos da classe <i>Box</i>	26
4.2	Tabela de atributos da classe <i>Kanban</i>	28
5.1	Referências de produção das máquinas de lixamento e polimento	58
5.2	Cenários de simulação	61

Capítulo 1

Introdução

Este capítulo introduz o projecto, fazendo o seu enquadramento no tema e realçando a sua importância nos dias de hoje, descreve os objectivos e a metodologia usada na realização do trabalho. No final do capítulo, é feita uma descrição da estrutura deste documento.

1.1 Enquadramento

Num mundo em constante mudança, com mercados globais em permanente evolução e onde todos querem ter uma quota parte, as empresas e organizações têm de se mostrar mais inovadores, flexíveis, com qualidade e tempos de resposta reduzidos. Estes factores não se aplicam apenas ao produto mas também aos processos de produção. Isto aliado à dimensão e complexidade dos sistemas de produção actuais, torna importante a procura de ferramentas que permitam avaliar e otimizar os processos produtivos.

Assim, a simulação é uma das soluções para analisar e avaliar o desempenho das situações actuais e das soluções futuras, com o objectivo de melhorar os recursos e processos de qualquer empresa. Torna-se, por isso, muito relevante para qualquer gestor de uma empresa ter uma ferramenta que o auxilie em tomadas de decisões que impliquem mudanças significativas nos processos da sua organização, com custos elevados e com um impacto grande na forma como fazem o seu produto.

Com o aumento da complexidade dos sistemas, torna-se ainda mais necessário recorrer à simulação antes de se passar à aplicação prática das soluções idealizadas. Através da simulação é possível determinar as melhorias que se podem obter, bem como detectar alguns erros antes da implementação concreta do novo sistema ou da mudança em causa.

No caso específico de Sistemas de Produção *Lean*, a simulação permite prever, de uma forma rápida e sem qualquer implementação física, o desempenho conseguido, traduzir este desempenho em números e otimizar qualquer situação sem recorrer a implementações.

1.2 Objectivos

Esta dissertação pretende dar continuidade ao desenvolvimento de ferramentas de simulação e análise de Sistemas de Produção *Lean* com base no software de simulação *AnyLogic*. Esta linha de trabalho foi iniciada no período lectivo anterior ao qual decorre este projecto [1] [2]. Os trabalhos anteriormente desenvolvidos tiveram como principal objectivo o domínio da ferramenta de simulação computacional *AnyLogic*, de forma a conhecer todas as suas potencialidades. Com este projecto, pretendeu-se ir um pouco mais além e desenvolver uma biblioteca de componentes de software parametrizáveis, flexíveis e reutilizáveis, através dos quais seja possível criar modelos de simulação de Sistemas de Produção *Lean* e avaliar aspectos do seu dimensionamento. Pretende-se que a construção dos componentes seja feita de forma a possibilitar a sua reutilização na modelação de vários sistemas e ou situações distintas. Pretende-se que a ferramenta de simulação a desenvolver permita uma visualização gráfica e animada dos Sistemas de Produção *Lean* modelados.

1.3 Metodologia

A metodologia seguida na realização deste trabalho foi a seguinte:

- Estudo da linguagem de programação Java, bem como revisão de Programação Orientada a Objectos tendo em vista o seu uso na ferramenta de simulação *AnyLogic*.
- Breve revisão do estado da arte de simulação e de conceitos usados em Sistemas de Produção *Lean*, como supermercado, *milkrun*, *kanban*, etc.
- Estudo da ferramenta *AnyLogic* através da realização de tutoriais e exemplos da ferramenta, bem como analisando os trabalhos anteriormente desenvolvidos.
- Desenvolvimento de exemplos de aplicação em *AnyLogic*.
- Desenvolvimento de modelo conceptual de um sistema de produção *Lean*.
- Desenvolvimento da biblioteca de componentes.
- Validação da biblioteca desenvolvida simulando um caso concreto e real.

1.4 Estrutura da Dissertação

Esta dissertação é constituída por 6 capítulos.

No capítulo 1, é feita a introdução ao trabalho, são descritos os seus objectivos, a metodologia seguida na sua realização e é apresentada a estrutura deste documento.

No capítulo 2, é feita uma descrição do estado da arte do tema Simulação e são apresentados conceitos usados em simulação. São também apresentados os conceitos *Lean* que influenciaram este trabalho.

No capítulo 3, é descrita a ferramenta de simulação usada, o *AnyLogic*, e são descritos os paradigmas de simulação e conceitos usados pela ferramenta.

No capítulo 4, é apresentada a biblioteca de componentes desenvolvida neste projecto e são descritos os seus constituintes.

No capítulo 5, é feita uma descrição do caso de estudo usado na validação dos componentes criados.

No capítulo 6, são apresentadas as conclusões sobre o desenvolvimento da biblioteca de componentes, sobre o uso da ferramenta *AnyLogic* e sobre o tema simulação. São também dadas sugestões sobre trabalho futuro.

Capítulo 2

Simulação e *Lean Manufacturing*

Neste capítulo é feita uma breve introdução a dois temas centrais neste trabalho: simulação e *Lean Manufacturing*, ou simplesmente *Lean*. Sobre o primeiro destes temas são analisadas definições segundo alguns autores, bem como alguns conceitos associados à área e os paradigmas presentes na ferramenta usada no desenvolvimento do trabalho. São ainda descritas algumas soluções presentes no mercado. Na parte final deste capítulo é feita uma breve introdução ao tema *Lean*, os objectivos desta filosofia e alguns conceitos da mesma que influenciaram o trabalho desenvolvido.

2.1 Simulação

2.1.1 Definição e Conceitos

A Simulação é uma das ferramentas mais poderosas disponíveis aos decisores responsáveis pelo desenho e funcionamento de sistemas e processos complexos [3].

Existem muitas formas de definir o termo “Simulação”. Fazendo uma pesquisa num dicionário de Língua Portuguesa concluímos que a palavra simulação significa o “acto ou efeito de imitar” [4].

Ao longo do estudo sobre este tema, prévio ao trabalho realizado, reuniram-se algumas definições encontradas em artigos de autores com investigação na área. De seguida são dadas duas dessas definições encontradas, assim, a Simulação é:

- a imitação do funcionamento de um processo ou sistema do mundo real ao longo do tempo. Envolve a criação e observação de uma história artificial do sistema de forma a se poder tirar conclusões sobre com as características do sistema real representado - segundo Banks [5].
- o processo de desenhar um modelo de um sistema real, conduzir experiências usando esse mesmo modelo com o propósito de compreender o comportamento do sistema e/ou avaliar várias estratégias para o seu funcionamento. Assim, é crucial que o modelo seja desenhado

de forma que o seu comportamento imite o comportamento do sistema real a eventos que ocorrem com o passar do tempo - segundo Shannon [3].

Destas duas definições conclui-se que ambos os autores concordam que simular é o acto de imitar o comportamento de um modelo de um sistema real.

Ingalls afirma que independentemente da complexidade de um sistema, é bastante provável que um perito em simulação seja capaz de criar um modelo que o avalie; no entanto, quanto mais complexo for o sistema, mais tempo será preciso para o modelar e simular [6].

Isto leva à necessidade de definir os termos “modelo” e “sistema”. Também para estes dois termos encontram-se na literatura da área várias definições:

Segundo Carson, um modelo é a representação de um sistema ou processo, e um modelo de simulação é uma representação que muda com o tempo [7].

Shannon afirma que um modelo é a representação de um grupo de objectos ou ideias numa forma que não a da própria entidade. E um sistema é um grupo de elementos interligados que cooperam entre si de forma a atingirem um objectivo definido [3].

Segundo Maria, a modelação é o processo de criar um modelo. E um modelo é a representação da construção e funcionamento de um sistema. Refere ainda que o modelo criado é idêntico mas mais simples que o sistema que representa [8].

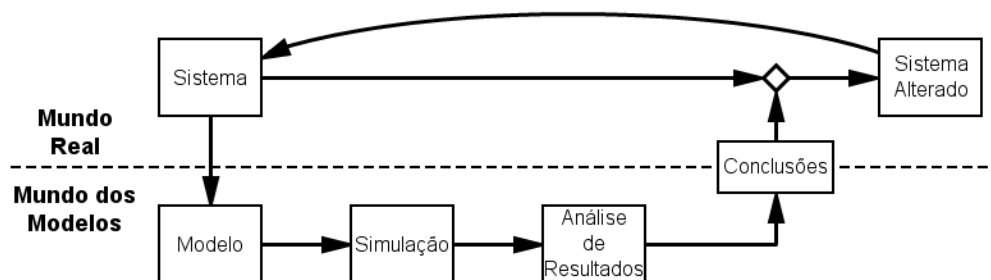


Figura 2.1: Esquema representativo de um estudo de simulação

A figura 2.1 representa um esquema, adaptado de [8], com os passos principais a seguir num estudo de simulação. O sistema que queremos simular é modelado. Após a sua simulação e recolha dos resultados da mesma, estes são analisados e são tiradas conclusões dos mesmos que permitam actuar no sistema de forma a melhorá-lo. Assim, após as conclusões, o sistema é alterado. A seta em forma de curva de retorno demonstra a repetição deste ciclo de forma a sustentar uma melhoria contínua do sistema.

2.1.2 Modelos de Simulação

Os modelos podem ser classificados como contínuos ou discretos, estáticos ou dinâmicos e determinísticos ou estocásticos [9]:

- Contínuos - o tempo de simulação avança forma contínua em intervalos de tempos iguais.

- Discretos - o tempo de simulação é baseado na ocorrência de eventos, ou seja, avança de evento em evento.
- Estáticos - o estado do sistema é descrito apenas para determinado momento e geralmente a variável de tempo não é importante.
- Dinâmicos - o estado do sistema é descrito baseado numa variável de tempo, este evolui com o decorrer do tempo.
- Determinísticos - os valores introduzidos na simulação são constantes.
- Estocásticos - os valores introduzidos na simulação são constantes; para modelos estocásticos, os valores introduzidos são aleatórios.

No caso do trabalho desenvolvido, este pode ser classificado como discreto pois o seu funcionamento é baseado em eventos, embora exista uma variável de tempo que pode ser escolhida e que determina o relógio da simulação. O sistema a simular evolui ao longo do tempo e por isso o modelo criado é dinâmico. Em termos de estocástico ou determinístico, o trabalho desenvolvido não pode ser classificado com clareza, pois tanto contém dados determinísticos como estocásticos, por exemplo, os tempos de avaria são estocásticos (valores aleatórios) e os tempos de produção são determinísticos.

2.1.3 Paradigmas de Simulação

De seguida serão apresentados 3 paradigmas de simulação: Dinâmica de Sistemas (*System Dynamics*), Baseado em Agentes (*Agent-Based*) e Eventos Discretos (*Discrete-Event* ou *Process-Centric*).

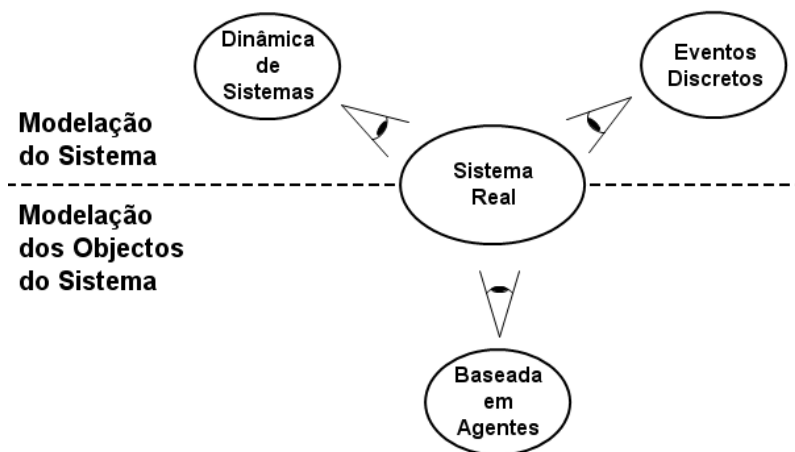


Figura 2.2: Diferentes paradigmas para modelação de sistemas

A Dinâmica de Sistemas (DS) é uma técnica de modelação mais orientada para modelos contínuos, em contraste com a Baseada em Agentes (BA) e Eventos Discretos (EV) que são mais virados para modelos discretos.

A figura 2.2, adaptada de [10], mostra os 3 paradigmas em questão. O mais recente, BA (década de 90), aborda a modelação do sistema focando a sua atenção no comportamento de cada objecto e os mais antigos, DS e ED (décadas de 50 e 60, respectivamente), modelam o sistema focando-se no seu funcionamento como um todo. Nota-se uma evolução em termos de pensamento quando se pretende modelar um sistema real com o aparecimento do paradigma BA.

A DS assume um alto nível de abstracção e é principalmente usada na modelação de sistemas ao nível estratégico. A modelação a EV é mais usada ao nível operacional e por isso assume um nível menos abstracto e mais detalhado. Já o paradigma BA é usado a todos os níveis, pois os agentes podem representar tanto empresas, projectos ou ideias como também veículos ou pessoas.

Na modelação DS, os processos do mundo real são representados em termos de stocks (material, conhecimento, pessoas, dinheiro), fluxo entre estes stocks, e informação que determina os valores destes fluxos. Para a modelação segundo esta técnica, o comportamento do sistema tem de ser descrito como um número de ciclos de realimentação (*feedback*) [11]. Este paradigma de simulação é usado em planeamento a longo prazo, estudo de estratégias e situações de alto nível onde não é necessária uma descrição individual dos objectos. Em termos matemáticos, um modelo segundo a DS é um sistema de equações diferenciais.

A modelação usando ED pode ser descrita como um conjunto de eventos, que alteram o estado do sistema. Este paradigma descreve o sistema real como uma sequência de operações realizadas em entidades de determinados tipos, que embora passivas, podem conter atributos que afectam a forma como são usadas ou mesmo alterá-los conforme o fluxo de entidades através dos processos [12]. Segundo Banks, um modelo de ED tenta representar os componentes de um sistema e as suas interacções de forma a satisfazer os objectivos do estudo desse mesmo sistema [5]. Pode-se descrever esta abordagem como sendo baseada no conceito de entidades, recursos e fluxogramas que descrevem o fluxo existente e a partilha de recursos [11].

A modelação BA é, essencialmente, descentralizada, ou seja, é baseada em objectos individuais para construir o modelo do sistema e não no seu comportamento geral. O modelador define o comportamento individual de cada objecto (ou objectos semelhantes) e o conjunto das individualidades formam o sistema como um todo. Assim, o sistema é modelado juntando vários objectos com comportamentos e regras individuais que em conjunto com todos os outros, num ambiente próprio e comunicando entre si criam o sistema pretendido [11]. Conclui-se desta forma que o este paradigma deve ser usado quando o sistema a modelar é um conjunto de objectos que têm um comportamento individual. Esta abordagem é usado tanto em sistemas com níveis de abstracção superiores como inferiores, a sua forma de operar permite alcançar várias áreas.

Como se verá, a ferramenta usada no trabalho realizado (*AnyLogic*) permite a modelação de sistemas segundo cada um ou mesmo uma combinação entre eles dando assim a possibilidade de criação de um modelo híbrido que melhor espelhe o funcionamento do sistema em causa.

2.1.4 Ferramentas e Tecnologias de Simulação

Existem várias ferramentas e tecnologias no mercado para simulação, o que torna o processo de escolha da correcta um problema a superar. Esta escolha pode significar o fracasso ou sucesso

do projecto, mesmo antes do seu fim.

Como tecnologias candidatas a executarem uma simulação estão as linguagens de programação genéricas ou convencionais, as linguagens de simulação e os simuladores (*Software* dedicado). Em todas estas existem vantagens e desvantagens.

As linguagens de programação genéricas têm um nível de flexibilidade muito elevado mas exigem conhecimentos de programação ao modelador, bem como muito mais tempo para a criação do modelo do que as alternativas. São exemplos destas linguagens o C, C++, Java, FORTRAN e Pascal.

As linguagens de simulação foram criadas especificamente para a criação de simulações e por isso têm a vantagem de estar vocacionadas para a área. No entanto, a necessidade de conhecimentos de programação mantém-se como um requisito para a construção da simulação. Embora sejam menos flexíveis, pois limitam a criatividade, dispõem de um interface com o programador, o que facilita todo o trabalho de programação e desta forma o tempo de concepção do modelo diminui. Nas diversas linguagens disponíveis, algumas foram criadas para certas áreas outras são mais genéricas e abrangem mais mercados. São exemplos de tipo de linguagens o SIMAN, GPSS, DYNAMO, Simula e SIMSCRIPT.

Os simuladores surgiram com o objectivo de facilitar a criação de modelos de simulação. Embora sejam precisos alguns conhecimentos ao nível da programação do simulador, este é muito pequeno quando comparado com as alternativas anteriores e em muitos casos o interface do simulador com o modelador quase que elimina este requisito. A grande desvantagem destas ferramentas é o seu custo elevado quando comparado com o custo (ou ausência dele em alguns casos) das mencionadas em cima. A flexibilidade na modelação do sistema é bastante menor nos simuladores, no entanto o tempo de concepção dos modelos é bastante mais rápido que em qualquer uma das alternativas. Dentro deste mercado existem várias hipóteses, umas mais genéricas, outras mais específicas em certas áreas. Como exemplos de simuladores temos o Rockwell Arena, Simulink, AMESim e *AnyLogic* (usado no trabalho desenvolvido nesta dissertação).

2.2 *Lean*

Esta secção do capítulo pretende fazer uma breve introdução ao *Lean* e apresentar alguns conceitos que influenciaram o trabalho desenvolvido.

O termo *Lean* foi introduzido ao mundo por Womack, Jones e Roos no início da década de 90 com o livro “*The Machine That Changed The World*”. O livro baseia-se num estudo de 5 anos sobre o futuro da indústria automóvel e onde se desvenda o sistema de produção usado pela Toyota nas suas fábricas, denominado *Toyota Production System* (TPS). É no TPS que o *Lean Manufacturing* se baseia.

2.2.1 Definição e Princípios

Lean é uma filosofia que engloba vários princípios e por isso é difícil a sua explicação numa só frase. No entanto, existe um objectivo que o *Lean* pretende atingir, a eliminação dos *Muda*¹.

Aliado a este objectivo estão também os conceitos de *Just-In-Time* (JIT) e *Jidoka*. JIT significa fazer apenas o que é necessário, quando é preciso e na quantidade certa [13].

Jidoka é um termo japonês que pode ser definido como “automação com um toque humano”, em oposição a uma máquina que funciona apenas sobre a monitorização e supervisão de um operador [14], e que significa que quando ocorre um problema, o equipamento parará imediatamente evitando-se a construção de produtos com defeitos [15]. Assim, é delegada a responsabilidade de produção com qualidade nos postos de trabalho ou máquinas ao longo da cadeia de valor. Por isto, este termo é muitas vezes referido como uma filosofia para garantir qualidade na produção.

A figura 2.3, adaptada de [14], mostra o conceito de *Jidoka*. Até há bem pouco tempo, era impensável na indústria, quando ocorre um problema, parar uma linha de produção para este ser resolvido. O conceito de *Jidoka* veio revolucionar este facto.



Figura 2.3: Conceito de *Jidoka*

Desta forma, e como exemplifica a figura 2.3, evita-se a ocorrência do mesmo problema uma segunda vez e assim a produção de peças com defeito.

Em 1996, *Womack e Jones*, no livro “*Lean Thinking*”, identificaram os cinco princípios para eliminação do desperdício e pelos quais a filosofia *Lean* se rege:

- Valor - identificar o que cria valor para o Cliente.
- Cadeia de Valor - identificar a sequência de actividades que criam valor para o Cliente, eliminando qualquer desperdício.
- Fluxo - criar fluxo na cadeia de valor, tornando todo o processo fluído.

¹*Muda* é a palavra japonesa para definir uma actividade que não produz valor para o cliente - desperdício.

- *Pull* - deixar a actividade a jusante puxar valor da montante, desta forma a actividade apenas produz quando necessário (ver 2.2.3).
- Perfeição - aplicar uma melhoria contínua (*Kaizen*²), nunca se contentar com o actual procurando sempre melhorar.

2.2.2 Os *Muda*

O *Lean* engloba uma técnica denominada *Value Stream Mapping* (VSM) que é uma análise à cadeia de valor, que produz uma representação gráfica de todas as actividades presentes na cadeia, quer acrescentem valor ou não. Desta forma, é possível ter-se uma visão global de toda a cadeia de valor e de onde estão os *Muda* nessa cadeia. Esta técnica é usada tanto no desenho da cadeia de valor no seu presente como também no seu futuro, ou seja, como se gostaria que ela fosse, quais as melhorias a inserir. Para isto, normalmente, são seguidos os seguintes 3 passos:

- 1º Construção do VSM da situação actual
- 2º Construção do VSM da situação desejada
- 3º Implementação do VSM desenhado para a situação futura

A repetição destes passos, ciclicamente, proporciona uma melhoria contínua na cadeia de valor, reduzindo o tempo de entrega ao cliente (*Lead Time*), bem como eliminando gradualmente os desperdícios.

Liker e Meier, autores do livro “*The Toyota Way Fieldbook*” (2005), definem a descrição feita por *Taiichi Ohno*³, em 1988, como o ponto inicial na criação de um fluxo *Lean*: *Ohno* afirmou que estava apenas a olhar para a linha de tempo desde que um cliente coloca uma encomenda até ao momento em que paga pela mesma, e a retirar todas as actividades que não acrescentam valor [16].

A *Toyota* identificou 7 tipos de desperdícios [16]:

- Excesso de Produção - produzir cedo demais e em quantidades maiores às necessárias; este tipo de *muda* provoca outros como o de excesso de inventário e o transporte.
- Tempos de Espera - qualquer operador parado, à espera que uma máquina termine de trabalhar ou à espera da próxima ordem de produção.
- Transporte - movimentação de WIP (*Work In Process*), recursos, peças, seja entre o armazém e os postos ou entre postos.
- Processamento Incorrecto ou em Excesso - tarefas desnecessárias ou erradas na produção de certa peça, uso de ferramentas inadequadas ou mau desenho do produto.

²*Kaizen* é a palavra japonesa que significa melhoria contínua (*kai* - mudança; *zen* - bom ou boa).

³*Taiichi Ohno* é considerado, por muitos, o pai do TPS.

- Excesso de Inventário - matérias-primas, WIP ou mesmo produto acabado à espera e a ocupar espaço; isto provoca atrasos na entrega do produto, possíveis danos nos materiais, produtos ou peças obsoletas, custos associados ao stock e transporte. O excesso de inventário também esconde problemas relacionados com defeitos, setups longos e avarias.
- Movimentações Desnecessárias - movimentos que os operadores fazem mas que não acrescentam valor, como pegar em ferramentas ou ir buscar peças; o simples acto de caminhar para cumprir a sua tarefa é considerado um desperdício.
- Defeitos - produção de peças ou produtos defeituosos, retrabalho e inspecção; tudo isto desperdiça tempo e esforço que não acrescenta valor.

Pode-se considerar, no entanto, um oitavo tipo de desperdício, não usar as ideias e criatividade dos operadores [16]. Ignorar ideias, ou mesmo descartá-las sem uma observação cuidada das mesmas pode ser considerado um desperdício de capacidades dos operadores, de tempo e de aprendizagem. De notar que quem acrescenta valor são os empregados e por isso são eles os que mais entendem da forma como é feito o seu trabalho.

2.2.3 Conceitos e Técnicas

De seguida serão introduzidos alguns conceitos relacionados com *Lean* que influenciaram o trabalho desenvolvido:

- *Lead Time* - é o tempo desde o momento em que a encomenda é feita pelo cliente até ao momento em que este a recebe; também pode ser descrito como o tempo que uma peça demora a percorrer o chão-de-fábrica desde a entrada como matéria-prima até ser expedida como produto final para o cliente [17].
- *Takt Time* - é a taxa de tempo à qual o cliente pede uma encomenda; é calculado dividindo o tempo de produção disponível pelo número de encomendas do cliente, por turno; é usado para sincronizar o ritmo de produção com o das encomendas [17]. Se o *takt time* for de 5 minutos, então a cada 5 minutos um produto deve de estar pronto no final da linha de produção.
- Fluxo Contínuo - é o fluxo conseguido entre todos os processos envolvidos sem acumulação de inventário intermédio.
- *Pull* - num sistema *pull* nenhum processo a montante deve operar até que o processo a jusante necessite; ao contrário de um sistema *push* em que os processos empurram a sua produção para os processos a jusante e assim criam o desperdício de excesso de produção, num sistema *push* um posto de trabalho apenas produz quando tem permissão para tal; desta forma, apenas é produzido o que é necessário.
- *Kanbans* - é um sinal usado para avisar que algo pode ser produzido; um *kanban* pode ser um espaço vazio, um cartão ou um sinal electrónico, serve para transmitir uma acção; num

sistema *pull*, o *kanban* é usado para controlar o nível de inventário e garantir que o processo a montante só produz quando o a montante o permitir.

- Supermercados - é um local de interrupção de fluxo quando não é possível que este seja contínuo, e onde, geralmente, se pretende implementar um sistema *pull*; armazena inventário necessário para o processo a jusante quando ainda não se pode garantir um fluxo contínuo com o processo a montante; é normalmente usado em conjunto com o *kanban* para formar um sistema de abastecimento.
- *Milkrun* - nome dado aos operadores logísticos que percorrem um caminho definido com um tempo de ciclo, de forma a garantir uma frequência certa de abastecimento de materiais aos locais afectados.
- *Heijunka* - termo usado para definir o nivelamento da produção, para que todo o processo trabalhe ao mesmo ritmo; o nivelamento da produção implica a distribuição das várias ordens pelos postos de trabalho de forma a ser atingido um ritmo de operação semelhante em todos eles; uma técnica associada a este conceito é a de *mixing* que mistura com as diferentes ordens de cada posto, jogando com o número de setups e a fazer e as diferentes referências a produzir.

Existem mais conceitos e técnicas associadas ao *Lean*, mas estes foram os que mais influenciaram o trabalho desenvolvido.

Em forma de conclusão sobre o tema, tudo o que o *Lean* tenta fazer é garantir que um processo apenas faz o que o próximo precisa, quando é necessário. Assim, ligando todos os processos desde o cliente final ao fornecedor de matéria-prima, sem desvios, com um fluxo contínuo que tenha o menor *lead time*, a máxima qualidade e o menor custo [17].

Capítulo 3

AnyLogic

Neste capítulo, a ferramenta de simulação usada, o *AnyLogic*, é apresentada enumerando as suas áreas de actuação, bem como os paradigmas presentes e a sua interligação. Ainda neste capítulo são descritos alguns objectos do *AnyLogic* que foram usados neste trabalho, em particular os objectos da *Enterprise Library* que tiveram particular interesse.

3.1 Introdução

O *AnyLogic* é uma ferramenta de simulação que permite a modelação de sistemas através de três métodos: Dinâmica de Sistemas (*System Dynamics*), Simulação Orientada a Eventos Discretos (*Discrete Event Simulation*) e Modelação Baseada em Agentes (*Agent-Based Modeling*). É possível usar combinações de dois ou mesmo destes três paradigmas através de simples métodos presentes no ambiente de modelação do *AnyLogic*. Através destes paradigmas, o utilizador pode facilmente encontrar um ponto de comunhão em que o sistema que pretende simular se situará. Desta forma compreende-se as várias áreas onde a ferramenta pode ser aplicada [18]:

- Sistemas de Produção
- Gestão na Cadeia de Fornecedores (*Supply Chain Management*)
- Logística e Transportes
- Defesa Militar e Aeroespacial
- Processos de Negócio
- Economia e Banca
- Centros de Atendimento
- Planos de Emergência e Evacuação

- Redes e Comportamentos Sociais
- Movimento de Pessoas e Veículos
- Análise de Estratégias de Negócio
- Indústria Automóvel
- Saúde e Biologia

A lista anterior, embora curta, demonstra a capacidade da ferramenta em alcançar várias áreas e vários mercados onde poucas outras ou até mesmo nenhuma conseguem alcançar.

O estudo das potencialidades da ferramenta de simulação *AnyLogic* foi realizado em dissertações anteriores [1] [2] e por isso, neste trabalho, não foram exploradas as diferentes técnicas oferecidas pelo software, nem fazia parte dos objectivos o estudo aprofundado da ferramenta. Assim, este projecto situa-se na fase seguinte ao estudo do *AnyLogic* enquanto ferramenta de simulação (capítulo 1). Tendo isto em conta, apenas será feita uma breve descrição sobre o que está por trás da ferramenta em termos programáticos, e uma explicação simples sobre os aspectos fundamentais do ambiente de modelação do *AnyLogic*, necessários para a compreensão do trabalho desenvolvido.

3.1.1 As Bases do *AnyLogic*

O *AnyLogic* é baseado na framework *Eclipse* [19], na linguagem de programação Java [20] e na de modelação UML (*Unified Modeling Language*).

O Java é uma linguagem de programação orientada a objectos (POO), ou seja, usa objectos e as suas interações no desenho de programas e aplicações computacionais; ao contrário dos métodos de programação mais tradicionais onde um programa é uma lista de tarefas a executar. Em Programação Orientada a Objectos, não se definem objectos mas sim classes de objectos. Uma classe é um modelo para múltiplos objectos com características semelhantes e um objecto é uma instância de uma classe.

Os *Active Objects* são os principais objectos usados no *AnyLogic* para a modelação de sistemas. Basicamente, são objectos Java que contém parâmetros, variáveis, funções, eventos ou até outros objectos, sejam eles outros *Active Objects* ou simples objectos de apresentação, como por exemplo um botão.

3.1.2 Ambiente de Modelação

O ambiente de modelação tem o aspecto da figura 3.1.

Do lado esquerdo temos a janela de projectos, que é uma vista hierárquica dos modelos abertos e onde se pode ver as várias classes de *Active Objects*, classes Java e Simulações de cada modelo. Abaixo desta vista temos a de erros, onde aparecem as respectivas mensagens de erros detectados durante a compilação. Ao centro temos a vista principal, o editor gráfico; é aqui que toda a programação gráfica é efectuada, desde a ligação de portas como desenhos de gráficos de estado

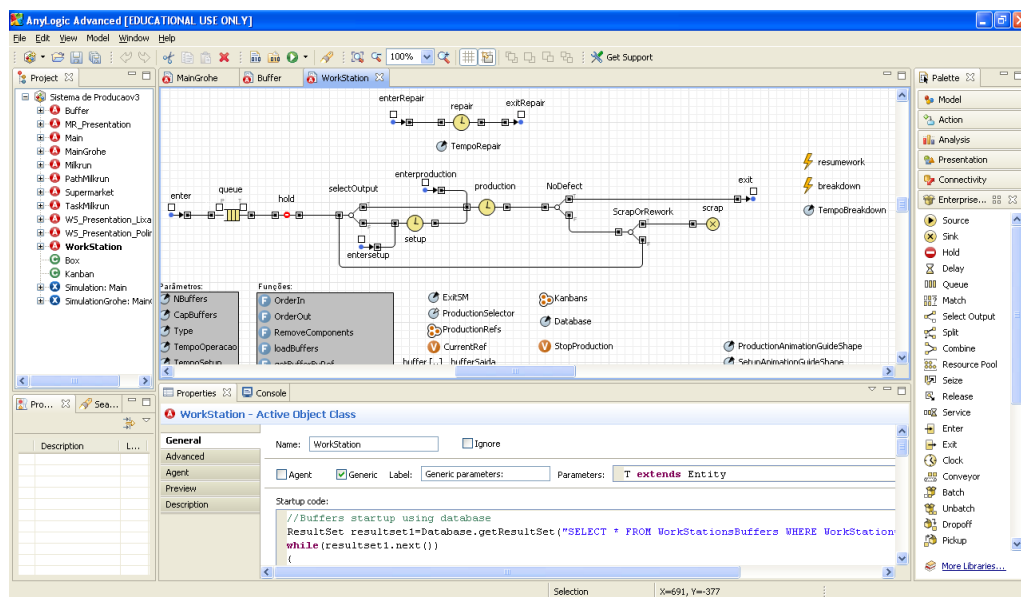


Figura 3.1: Ambiente de Modelação do AnyLogic

ou ainda de objectos de apresentação como botões, figuras geométricas, etc. Em baixo temos a janela de propriedades e a consola java. Esta última é usada, principalmente, para observar quais os erros ocorridos durante o runtime. Finalmente, à direita, está a vista das bibliotecas, que na versão educacional é constituída por 6 grupos:

- *Model* - contém parâmetros, eventos, variáveis, gráficos de estados, etc.
- *Action* - contém objectos para criação de funções graficamente, através um diagrama.
- *Analysis* - contém objectos de análise de dados como gráficos de tempo, histogramas, etc.
- *Presentation* - contém objectos de apresentação como figuras geométricas, botões, caixas de texto, etc.
- *Connectivity* - contém objectos de ligação e utilização de base de dados ou de um *flat file* (este último apenas na versão Professional do AnyLogic).
- *Enterprise Library* - contém objectos para simulação de sistemas orientados a eventos discretos.

Nem todos os objectos das bibliotecas presentes na versão educacional podem ser usados pois fazem parte do pacote da versão profissional da ferramenta. Por exemplo, no grupo *Connectivity* apenas 1 dos 6 objectos presentes pode ser usado. Existem mais bibliotecas, mas mais uma vez apenas estão disponíveis para a versão profissional.

3.2 *Enterprise Library*

O trabalho desenvolvido neste projecto de simulação foi na sua maioria baseado em objectos da *Enterprise Library*. Esta biblioteca é a fonte principal de objectos do *AnyLogic* para a modelação de sistemas orientados a eventos discretos. Através destes objectos a modelação é realizada em termos de entidades, processos e recursos. Neste contexto, as entidades podem representar o movimento de pessoas, produtos, veículos, etc. Já os processos representam acções, operações, uso de recursos, etc. Desta forma, a modelação de sistemas usando esta biblioteca toma a forma de fluxogramas, o que facilita a compreensão da lógica do sistema modelado, pois é uma linguagem gráfica bastante intuitiva e de fácil compreensão. O transporte de entidades entre objectos é efectuado através de portas presentes nos mesmos e conectadas entre si. Existe outra forma de retirar ou injectar entidades nos objectos, mas este ponto será abordado mais tarde aquando da descrição de alguns objectos desta biblioteca.

Para realizar este projecto, foram usados vários objectos desta biblioteca do *AnyLogic*, principalmente, por 2 razões que caracterizam tanto o tipo de sistema a modelar como as vantagens de a usar:

- Primeiro, porque os componentes da biblioteca que se criou podem ser modelados usando eventos discretos.
- Em segundo, devido à facilidade de recolha de estatísticas e de criação de animações mais elaboradas.

Este último ponto ganha vantagem em relação à possibilidade de programar usando máquinas de estado, pois a visualização da simulação através de animações (por exemplo, movimentos) e gráficos é bastante mais útil do que apenas números e resultados.

Aquando desta dissertação, a *Enterprise Library* do *AnyLogic* continha todos os elementos presentes na figura 3.2.

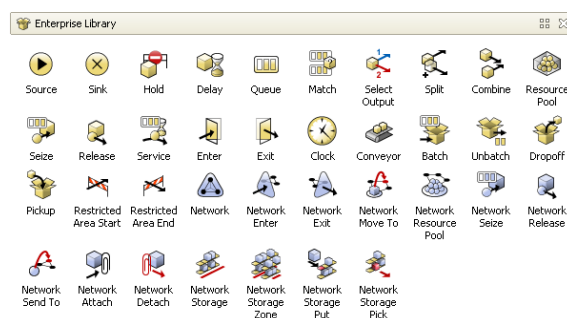


Figura 3.2: Objectos da *Enterprise Library*

A lógica e o funcionamento desta biblioteca são baseados no fluxo de entidades através do fluxograma criado. As entidades que percorrem o fluxograma são análogas aos *tokens* das Redes de Petri. No *AnyLogic*, as entidades saem dos objectos através de portas que estes contêm. Um

erro pode ocorrer se a entidade tenta sair de um objecto e a porta deste não está ligada a numa outra de outro objecto ou ainda se a entrada noutro objecto viola alguma regra. Como por exemplo entrar um segundo objecto num *Queue* quando já está um lá dentro e a capacidade do mesmo é de apenas 1. No *AnyLogic*, estas entidades estão associadas a classes Java. Visto que esta ferramenta nos permite a criação das nossas próprias classes, é assim possível uma personalização dos parâmetros associados às entidades. A classe usada por defeito para as entidades é a *Entity* [21]. No entanto, ao criarmos as nossas próprias classes, estas devem ser derivadas da já existente para que possam ser associadas às entidades, ou seja, têm de ser subclasses da classe *Entity*. Derivando a nova classe da *Entity* já existente, é possível acrescentar atributos e operações que se pretender, mantendo todas as propriedades e funcionalidades da classe base.

A figura 3.3 mostra a declaração da classe *Customer* usada no modelo *Bank*. Este modelo é baseado num tutorial [21] disponibilizado pelo fabricante do *AnyLogic* com o intuito de apresentar a Enterprise Library e as suas possibilidades.

```
/**
 * Customer
 */
public class Customer extends Entity {

    double startWaiting;
    double enteredSystem;

    /**
     * Default constructor
     */
    public Customer() {
    }

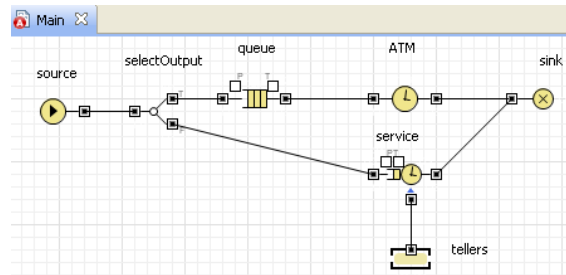
    /**
     * Constructor initializing the fields
     */
    public Customer(double startWaiting) {
        this.startWaiting = startWaiting;
    }

    @Override
    public String toString() {
        return
            "startWaiting = " + startWaiting + " ";
    }
}
```

Figura 3.3: Definição da classe *Customer*

De notar que a classe criada (*Customer*) é derivada da *Entity*. É necessário que assim seja para que possamos associar as classes criadas às entidades, sem violar as regras internas de funcionamento do fluxo de entidades da Enterprise Library. Fica assim possível o uso desta nova classe na criação de novas entidades que percorrem os objectos do modelo (figura 3.4).

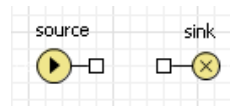
Todos os objectos desta biblioteca contêm eventos, a grande parte deles do tipo *onEnter* ou *onExit* que ocorrem quando as entidades entram ou sai, respectivamente, do objecto. Há ainda objectos que contêm outros eventos, como é o caso do objecto *Queue*, mas este ponto ficará mais claro aquando da sua explicação [3.2.2]. O que importa realçar neste momento, é a possibilidade de se programar os objectos para que quando as entidades passem por estes sejam efectuadas as instruções que se deseje.

Figura 3.4: Modelo *Bank*

De seguida será feita uma descrição de alguns dos objectos da biblioteca, aqueles que têm uma influência relevante no trabalho desenvolvido. Estes objectos têm várias propriedades, no entanto, nesta dissertação, e em particular nas secções que se seguem, apenas se fará uma explicação das funcionalidades do objecto, sua importância dentro de um modelo e também das suas potencialidades em termos de animações.

3.2.1 *Source e Sink*

O objecto *Source* e o objecto *Sink* servem, respectivamente, para a criação e despejo das entidades. Por esta razão, são, normalmente, os objectos usados no início e final de um fluxograma. Como se pode ver pela figura 3.5, o objecto *Source* apenas tem uma porta de saída e o *Sink* uma de entrada.

Figura 3.5: *Source e Sink*

Existem várias maneiras de definir como e quando as entidades no objecto *Source* são geradas, podem ser em função do tempo ou manualmente usando o método *inject* deste tipo de objecto. Para além destes parâmetros, é, também, possível a alteração do número de entidades a gerar quando a forma de criação das mesmas é outra que não a manual. As entidades geradas por defeito são do tipo *Entity*, no entanto, após a criação de uma classe personalizada, logo que seja uma subclasse da *Entity*, esta pode ser associada na criação de novas entidades. Há, no entanto, outra hipótese de criar e eliminar as entidades que é através do código directamente, gerando uma nova instância da subclasse criada ou da existente por defeito, a *Entity*. Na situação de criação das entidades sem ser através do objecto *Source*, estas podem ser injectadas no fluxograma através do objecto *Enter* (3.2.4), recorrendo ao método *take()* do mesmo.

Nestes objectos, os eventos existentes e os respectivos momentos de ocorrência são:

- *Source* - evento *onExit* que ocorre quando a entidade sai do objecto.
- *Sink* - evento *onEnter* que ocorre quando a entidade entra no objecto.

No Source é possível definir uma forma de apresentação (*Shape*) para animação das entidades. Se nenhuma forma de animação for definida e existir animação nos objectos por onde passam as entidades, o *AnyLogic* cria por defeito *Shapes* para as entidades.

Quando um objecto do tipo Source cria uma entidade, esta sai imediatamente e por isso é necessário garantir que a entidade pode, de facto, sair do objecto, caso contrário ocorrerá um erro durante a simulação. Assim, quando não se pode prever que o objecto seguinte é capaz de receber a entidade, é aconselhável usar um objecto que funcione como um buffer, por exemplo um objecto do tipo *Queue* [3.2.2].

3.2.2 Queue

A figura 3.6 representa o objecto *Queue* que tem uma porta de entrada e três de saída. A porta rotulada com “P” serve para expulsar entidades por razões de prioridade e a rotulada com “T” serve para a saída de entidades por questões de *timeout*.

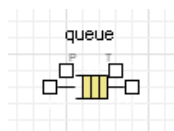


Figura 3.6: *Queue*

O objecto *Queue* funciona como um buffer, ou seja, é um objecto que guarda as entidades que nele entram, como se de um local de stock se tratasse. As entidades permanecem no *Queue* até o processo seguinte permitir a entrada das mesmas (uma ou mais). É também possível a remoção das entidades através do método `remove()`.

A ordem das entidades neste objecto pode ser do tipo *FIFO* (*First In First Out*) ou então baseada em prioridades. A primeira hipótese é a mais usual e baseia-se no conceito da primeira entidade a entrar é a primeira a sair. Se usarmos o método baseado em prioridades, quando uma entidade entra no objecto, este avalia a sua prioridade em relação às entidades já presentes no *Queue* e coloca-a na posição correspondente. Neste modo, o objecto aceita sempre a entidade que entra, mas caso esteja cheio e após avaliar a sua prioridade decide se expulsa a entidade que acaba de entrar caso esta tenha prioridade inferior ou igual à última já existente no objecto ou então expulsa a última caso a entidade da que entrou seja superior a esta. Mas caso a ordem das entidades seja do tipo *FIFO* e uma entidade tenta entrar, irá ocorrer um erro durante a simulação.

É também possível associar às entidades que entram no *Queue* um tempo de *timeout* que quando termina, o objecto expulsa a entidade em causa.

Para além dos eventos comuns existentes na maioria dos objectos desta biblioteca (*onEnter* e *onExit*), este objecto tem também outros 3 eventos:

- *onAtExit* - ocorre quando uma entidade chega à saída do objecto mas não sai (ao contrário do *onExit* que é executado quando a entidade sai).

- *onExitPreempted* - idêntico ao *onExit* mas quando a entidade sai pela porta *outPreempted* (ordem das entidades baseada em prioridades).
- *onExitTimeout* - idêntico ao *onExit* mas quando a entidade sai pela porta *outTimeout* (ordem das entidades baseada em prioridades).

À semelhança de muitos objectos desta biblioteca, também o *Queue* tem possibilidade de animações das entidades nele presentes. Estas animações podem, por exemplo, representar as entidades numa fila, umas atrás das outras. Para isto basta indicar neste objecto qual o objecto de apresentação que representa o *Queue* (por exemplo, um objecto da classe *Polyline*).

3.2.3 Delay

O objecto *Delay* (figura 3.7) tem por função reter uma ou mais entidades durante um tempo definido. Este tempo pode ser definido explicitamente ou então através do comprimento do objecto de animação dividido por um parâmetro de velocidade.

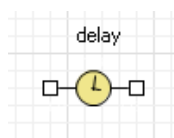


Figura 3.7: Delay

É possível remover as entidades do objecto mesmo que o tempo de *delay* ainda não esteja terminado. O *Delay* também permite o acesso ao tempo que falta para a entidade sair.

Se uma entidade tenta entrar no objecto e este já está na sua capacidade máxima, ocorrerá um erro durante a simulação. O aconselhável é o uso de um objecto do tipo *Queue* (3.2.2) antes ou então garantir que nenhuma entidade tentará entrar no *Delay* com este na sua capacidade máxima.

Este objecto, em termos de eventos apenas contém os mais usuais: *onEnter* e *onExit* (já explicados anteriormente em 3.2).

O *Delay* permite animar as entidades nele retidas de forma semelhante à usada no *Queue* (3.2.2). A principal diferença é que enquanto que no *Queue* não se conhece o tempo que a entidade vai lá permanecer, no *Delay* esse tempo é conhecido e por isso permite animações dinâmicas como por exemplo o movimento de um carro ou de uma pessoa ao longo de uma linha. Quando a animação envolve movimentação das entidades, estas percorreram o caminho todo durante o tempo de *delay*, ou seja, a velocidade das entidades é determinada pela distância que têm de percorrer e pelo tempo de *delay*.

3.2.4 Enter e Exit

Os objectos *Enter* e *Exit* (figura 3.8) permitem, respectivamente, inserir e retirar entidades do fluxograma. Estes objectos são usados quando se pretende mover entidades entre diferentes fluxogramas, ou então quando se quer transferir as entidades entre objectos como acontece no

trabalho desenvolvido. Desta forma, é possível a criação de fluxogramas que não comecem com *Sources* nem terminem em *Sinks*, respectivamente.

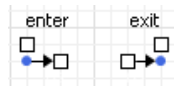


Figura 3.8: *Enter e Exit*

Para inserir as entidades no Enter usa-se o método `take()` do objecto.

Os eventos destes objectos são semelhantes aos encontrados nos *Source* e *Sink* [3.2.1].

3.2.5 *Hold*

O objecto *Hold* (figura 3.9) tem a função de bloquear o fluxo de entidades. É usado para impedir entidades de entrar ou sair de objectos apenas quando se pretende. Por exemplo, impedir a saída de um *Queue* ou a entrada num *Delay*.

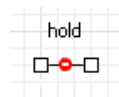


Figura 3.9: *Hold*

No entanto, este objecto não retém entidades, apenas impede que passem para jusante enquanto estiver bloqueado. Desta forma, as entidades permanecem no objecto anterior que tenha capacidade de retenção de entidades (por exemplo o objecto *Queue*). Caso contrário ocorrerá um erro durante a simulação. Assim, e em termos de eventos, o *Hold* apenas contém o `onEnter` que é executado quando a entidade passa pelo objecto.

Este objecto tem dois estados possíveis: bloqueado ou desbloqueado. Esta opção pode ser alterada através de programação usando o método `setBlocked()`.

3.2.6 *Select Output*

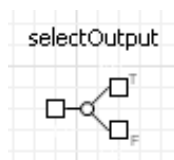


Figura 3.10: *Select Output*

O objecto *Select Output* (figura 3.10) reencaminha a entidade que nele entra por uma de duas portas, segundo uma condição, sendo ela de verdadeiro ou falso, ou uma probabilística. Em ambos os casos, o teste é feito quando a entidade entra no objecto e é reencaminhada para a porta de

saída respectiva. A decisão de reencaminhamento pode depender de factores externos ou até de parâmetros da entidade.

Como no caso do *Hold* (3.2.5), a entidade não fica retida no objecto, passa instantaneamente durante a simulação. Embora a entidade não tenha qualquer período de retenção no objecto, este contém três eventos: *onEnter* que ocorre quando a entidade entra no objecto, *onExitTrue* para quando a entidade sai do objecto pela porta correspondente à condição ser verdadeira e *onExitFalse* para quando a condição é falsa.

Capítulo 4

Biblioteca de Componentes

Este capítulo descreve a biblioteca de componentes desenvolvida, a qual constituiu o núcleo deste trabalho. Começa por descrever as capacidades e funcionalidades da biblioteca como um todo e, em seguida, pormenoriza cada um dos componentes criados.

4.1 Introdução

O trabalho desenvolvido tinha por objectivo iniciar a criação de uma biblioteca de classes de objectos passíveis de ser reutilizados. Posto isto, as classes criadas tinham de ser o mais parametrizáveis e flexíveis possível.

Neste momento, através dos componentes desenvolvidos, é possível simular um sistema de produção constituído por 3 tipos de subsistemas principais onde existam máquinas ou postos de trabalho, supermercados e *milkruns*. Existem algumas limitações quanto à modelação dos sistemas, mas este tema será abordado mais tarde aquando da explicação dos objectos [4.3] e na conclusão [6].

Neste trabalho foram desenvolvidas 2 classes Java [4.2] e 4 *Active Objects* (AO):

- *Buffer* (BF) - AO usado na simulação de *buffers* com uma capacidade e posições para caixas.
- *Supermarket* (SM) - AO usado para simular supermercados com um número de *buffers* (contém objectos da classe anterior).
- *Workstation* (WS) - AO usado para simular um posto de trabalho, uma máquina ou mesmo uma célula de fabrico (contém objectos da classe *Buffer*).
- *Milkrun* (MR) - AO usado para simular um *milkrun* de abastecimento de caixas de um supermercado a um conjunto de *workstations* (contém objectos da classe *Buffer*).
- *Box*- classe java usada para simular uma caixa com vários componentes.
- *Kanban*- classe java usada para simular um *Kanban* com ordens de produção.

4.2 Classes Java

As 2 classes em Java foram criadas para associar às entidades que percorrem os objectos da biblioteca - *Box* e *Kanban*. Ambas são subclasses da classe *Entity*. O diagrama UML de classes da figura 4.1 ilustra isso mesmo.

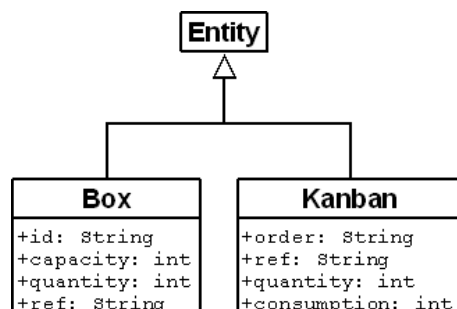


Figura 4.1: Diagrama UML das subclasses *Box* e *Kanban*

Desta forma, foram mantidas todas as propriedades da superclasse *Entity* e ainda se acrescentaram atributos referentes a cada uma das subclasses.

4.2.1 Box

A classe *Box* (figura 4.2) foi criada com o objectivo de simular caixas com componentes. Estes componentes são consumidos nas operações das *WSs*. As entidades desta classe passam por todos os *AOs* criados: os *MRs* transportam-nas desde os *SMs*, que é o local de armazenamento das mesmas, até às respectivas *WSs*.

A tabela 4.1 descreve a função de cada atributo desta classe, bem como o seu tipo.

Tabela 4.1: Tabela de atributos da classe *Box*

Nome	Tipo	Função
id	string	identificador da caixa
capacity	inteiro	capacidade (número máximo de componentes na caixa)
quantity	inteiro	quantidade (número de componentes actualmente na caixa)
ref	string	referência identificadora dos componentes presentes na caixa

Como se pode ver na figura 4.2, na declaração da classe, o método *toString()* é sobreposto através da instrução *override*. É usado o *override*, porque este método já existe por defeito nas classes Java e o que se pretende é uma versão personalizada do método. Este é usado quando se deseja uma representação, em formato *String*, do objecto. Neste caso, este método retorna os valores actuais dos atributos da classe *Box*.

```

/**
 * Box
 */
public class Box extends Entity {

    String id;
    int capacity;
    int quantity;
    String ref;

    /**
     * Default constructor
     */
    public Box() {
    }

    /**
     * Constructor initializing the fields
     */
    public Box(String id, int capacity, int quantity, String ref) {
        this.id = id;
        this.capacity = capacity;
        this.quantity = quantity;
        this.ref = ref;
    }

    @Override
    public String toString() {
        return
            "id = " + id + "\n" +
            "capacity = " + capacity + "\n" +
            "quantity = " + quantity + "\n" +
            "ref = " + ref;
    }
}

```

Figura 4.2: Classe Java - *Box*

4.2.2 Kanban

A classe *Kanban* (figura 4.3) foi criada com o objectivo de simular kanbans com ordens de produção, que são usados nas *WSs*. As entidades desta classe percorrem os objectos contidos nas *WSs*.

```

/**
 * Kanban
 */
public class Kanban extends Entity {

    String order;
    String ref;
    int quantity;
    int consumption;

    /**
     * Default constructor
     */
    public Kanban() {
    }

    /**
     * Constructor initializing the fields
     */
    public Kanban(String order, String ref, int quantity, int consumption) {
        this.order = order;
        this.ref = ref;
        this.quantity = quantity;
        this.consumption = consumption;
    }

    @Override
    public String toString() {
        return
            "order = " + order + "\n" +
            "ref = " + ref + "\n" +
            "quantity = " + quantity + "\n" +
            "consumption = " + consumption;
    }
}

```

Figura 4.3: Classe Java - *Kanban*

A tabela 4.2 descreve o tipo e a função de cada atributo.

Tabela 4.2: Tabela de atributos da classe *Kanban*

Nome	Tipo	Função
order	string	ordem de produção
ref	string	referência identificadora dos componentes a consumir
quantity	inteiro	quantidade a produzir
consumption	inteiro	quantidade de componentes a consumir

Tal como acontece na classe *Box*[4.2.1], também nesta o método *toString()* é sobreposto por uma versão personalizada. E também aqui o objectivo é retornar os valores actuais dos atributos em formato *String*.

4.3 Active Objects Criados

A explicação de cada *Active Object* criado pode ser dividida em duas partes:

- Modelação e programação dos vários objectos embebidos, bem como a sua lógica de funcionamento.
- Animação e apresentação do estado do objecto.

Desta forma, a explicação fica mais simples, embora a programação da lógica e da animação, em alguns casos, esteja bastante misturada. Por exemplo, a acção de um evento pode executar tanto acções relativas ao funcionamento do objecto como relacionadas com a animação e/ou apresentação.

Todos os objectos criados têm parâmetros e alguns têm variáveis. No *AnyLogic*, os parâmetros servem principalmente para determinar o estado estático do objecto e são, normalmente, constantes que apenas são alteradas quando se pretende modificar o comportamento do objecto. Isto contrasta com as variáveis que têm um comportamento semelhante e são usados para modelar o estado do objecto. Os parâmetros são, também, usados quando se pretende ter várias instâncias da mesma classe de objectos e apenas se pretende mudar o seu comportamento; já as variáveis são usadas para guardar valores que mudam ao longo da simulação. Assim, é complicada a decisão entre o uso de parâmetros ou de variáveis, ambas têm potencialidades idênticas. O manual do *AnyLogic* aconselha o uso de parâmetros para modelação do comportamento do modelo, e o uso de variáveis quando se pretende guardar valores que alteram com a simulação. Uma diferença em termos de programação é o facto de as variáveis apenas serem alteradas através da programação e os parâmetros poderem ser alterados em cada uma das instâncias da classe através do ambiente gráfico (figura 4.4).

Existe no *AnyLogic* um tipo de variáveis que permite guardar uma lista (ou array) de variáveis (*Collection Variable*). Dentro deste tipo de variáveis, o *AnyLogic* suporta 2 *Collections* da *Java Collection Framework*: *ArrayList* e *LinkedList*. A diferença entre ambos é a forma como guardam os dados. Todas as *Collection Variables* usadas neste trabalho foram do tipo *LinkedList* de forma a respeitar a regra *FIFO*.

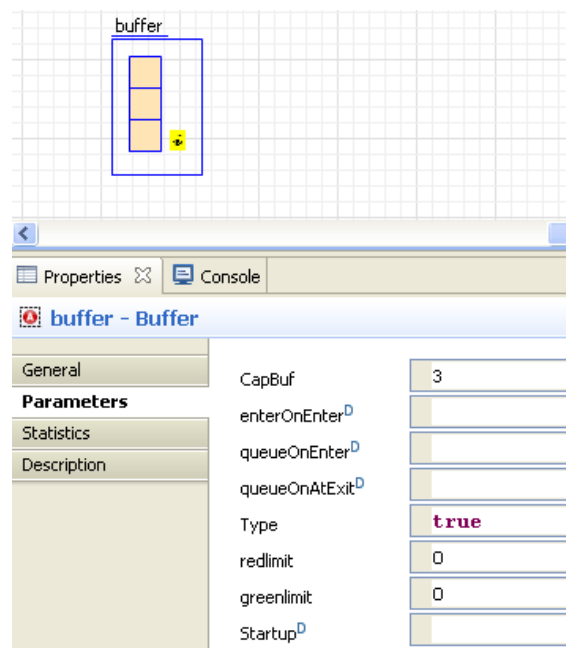


Figura 4.4: Alteração dos parâmetros de um *Buffer* através do ambiente gráfico

Existem no *AnyLogic* um tipo especial de parâmetros, denominados por parâmetros dinâmicos. Estes são criados como um parâmetro normal, mas com a diferença de se seleccionar a opção *Dynamic* (figura 4.5). Estes parâmetros são calculados de cada vez que se acede ao mesmo. Desta forma, o parâmetro adquire o funcionamento semelhante a uma função, na qual se pode inserir as linhas de código pretendidas em cada instância do *AO*, aumentando assim a flexibilidade do objecto. É assim possível replicar eventos à semelhantes ao *onEnter* dos objectos já existentes na Enterprise Library em *Active Objects* criados, como acontece no trabalho realizado.

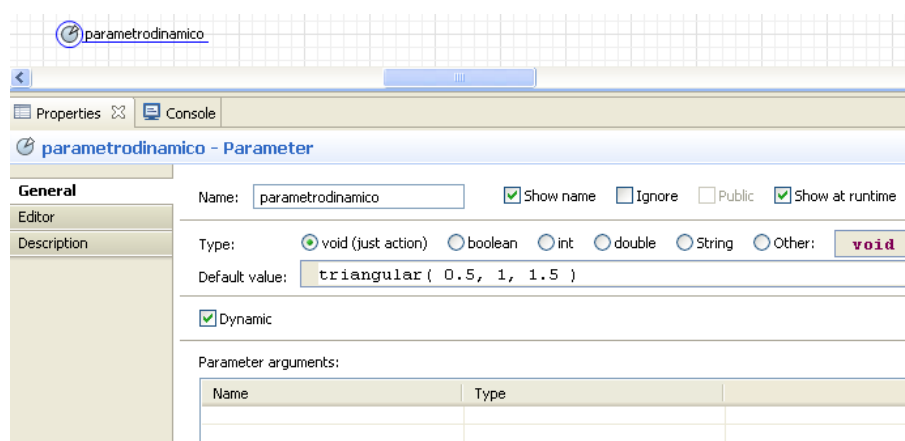


Figura 4.5: Parâmetro dinâmico

Para ilustrar este conceito considere-se o seguinte exemplo: criamos 2 parâmetros, um normal e outro dinâmico, cada um deles associado ao tempo de um *Delay*- *delay1* e *delay2*, respectiva-

mente. Em ambos os parâmetros, o valor é *triangular(0.5, 1, 1.5)*. Esta função já existe no *AnyLogic* e gera um número segundo uma distribuição triangular. Para o caso do parâmetro normal, o número é gerado apenas uma vez; imaginemos que o número gerado é 1, desta forma, o tempo de simulação do *delay1* é sempre de 1 (excepto se houver alteração através do método *set_nomedoparâmetro()*). Já no caso do parâmetro dinâmico, como é calculado de cada vez que é acedido, sempre que o objecto *delay2* retém uma entidade, a função *triangular(0.5, 1, 1.5)* é chamada para determinar o tempo de retenção.

Em alguns casos, foi usada a propriedade *Replication* dos objectos. Através desta propriedade, é possível criar múltiplas instâncias do mesmo objecto, bastando para tal criar uma e depois colocar a quantidade pretendida no campo *Replication*. Estes objectos ficam todos associados ao nome do original e a forma de aceder ao conteúdo de cada um é usando o método *get()*. Visualmente é fácil identificar um objecto replicado pois à frente do seu nome aparecem os caracteres *[..]*. Por exemplo: cria-se uma instância do objecto *Queue* (3.2.2) com o nome “ABC” e coloca-se o valor 2 no campo *Replication*, criando assim 2 objectos; a forma de aceder a cada uma das instâncias é usando o índice, ‘0’ para a primeira (*ABC.get(0)*) e ‘1’ para a segunda (*ABC.get(1)*) (figura 4.6). Desta forma, aceder ao parâmetro *Capacity* (capacidade) dos objectos será: para o primeiro *ABC.get(0).capacity* e para o segundo *ABC.get(1).capacity*.

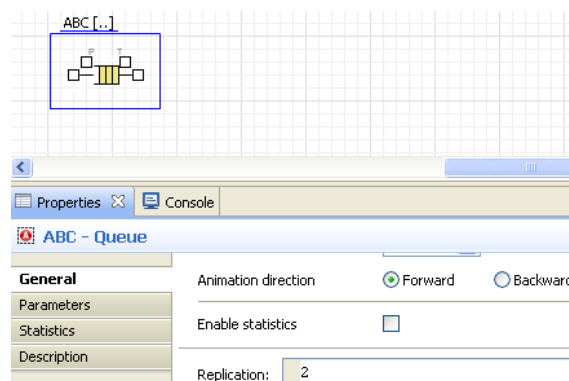


Figura 4.6: Exemplo de um objecto replicado

Para todos os objectos criados existe um ícone por defeito mas é possível criar ícones personalizados para cada classe que podem conter animação, texto, figuras, etc. O ícone é usado para representar o objecto criado, se este não for criado, o *AnyLogic* usa um por defeito (figura 4.7). É construído da mesma forma que as animações e apresentações, ou seja, recorrendo às classes de objectos de apresentação presentes na divisão *Presentation* das bibliotecas existentes no *AnyLogic* (figura 3.1); apenas é necessário seleccionar a opção *Icon* presente nas propriedades do objecto para que as *Shapes* escolhidas passem a formar o ícone.

Em todos os AOs criados existe uma animação “interna”. Pode-se dizer que esta é a animação por defeito do objecto e é usada, em termos gerais, para reproduzir o estado do objecto. No entanto, é possível a criação de outra animação fora do objecto pelo modelador do sistema. Quer esta seja feita ou não, a interna estará sempre presente.

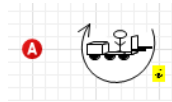


Figura 4.7: Ícone existente por defeito (à esquerda) e ícone criado para o *AO Milkrun* (à direita)

A descrição dos 4 *AOs* criados será feita do mais simples para o mais complexo de forma a ser mais fácil a percepção dos conceitos envolvidos. Assim, o *Buffer* será o primeiro, seguido do *Supermarket* que, basicamente, é um conjunto de *Buffers*. Em terceiro lugar será explicado um objecto mais complexo, o *Workstation* que já tem um nível elevado de complexidade. Por último será explicado o *Milkrun* que embora tenha uma lógica simples, as suas funcionalidades em termos de programação não são triviais.

4.3.1 Buffer

O primeiro *AO* criado tem por objectivo simular os *buffers* existentes nos sistemas de produção. Mais concretamente, simular um *buffer* com caixas contendo componentes (entidades da classe *Box*).

O *Buffer* tem como função armazenar entidades, o que faz com que seja o objecto mais reutilizado na biblioteca. É usado em todos os outros 3 *AOs* criados e em alguns dos casos é replicado, pois pode existir mais que um *buffer* com as mesmas funções e apenas estados diferentes. Por exemplo, um posto de trabalho ter 2 *buffers* em que um deles contém caixas com componentes do tipo 'A' e o outro do tipo 'B'. Ambos têm a mesma função, apenas diferem no seu estado (o tipo de componente que as caixas contêm). É, desta forma, o *AO* mais simples dos 4 criados.

O funcionamento do *Buffer* baseia-se em 2 objectos da Enterprise Library, o *Enter* e o *Queue*, em conjunto com os atributos (parâmetros e variáveis) e métodos (funções e eventos) visíveis na figura 4.8.

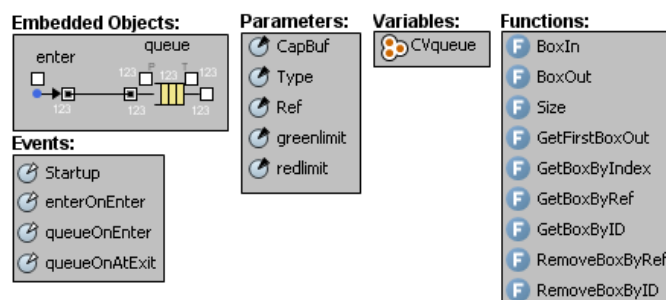


Figura 4.8: Lógica do *Buffer*

As entidades do tipo *Box* entram no objecto através do *enter*, usando o método *take*, seguindo imediatamente para o *queue* onde ficam armazenadas. Como se pode ver a porta de saída do *queue* não está ligada e por isso as entidades entram e não saem através do fluxo normal. A forma de as

retirar é usando o método *remove* do objecto. Foram criadas funções que permitem agilizar este e outros acessos necessários.

4.3.1.1 Atributos

Os atributos deste *AO* podem ser separados em parâmetros e variáveis. Os parâmetros do *Buffer* determinam:

- a sua capacidade (*CapBuf*) - quantidade de caixas que pode receber.
- o seu tipo (*Type*) - aceita caixas de uma só referência ou indiscriminadamente.
- a sua referência (*Ref*) - determina a referência das caixas que aceita. Caso aceite todas, o seu valor é o carácter '-'.
- os limites verde (*greenlimit*) e vermelho (*redlimit*) - parâmetros usados apenas quando o objecto faz parte de um *Supermarket*.

O *Buffer* não tem variáveis simples (a que o *AnyLogic* chama de *Plain Variables*), tem apenas uma *Collection Variable* - *CVqueue*, que guarda numa lista todas as caixas que entraram neste *AO* e consequentemente todas as que entraram no *queue*. Desta forma é criado um histórico das caixas que passam pelos *Buffers*, facilitando a rastreabilidade das mesmas.

4.3.1.2 Métodos

Os eventos presentes no *Buffer* permitem ao modelador inserir instruções que são executadas quando:

- o *Buffer* é criado - *Startup*.
- a entidade *Box* que circula no *Buffer* entra no *enter* - *enterOnEnter*.
- a entidade *Box* que circula no *Buffer* entra no *queue* - *queueOnEnter*.
- a entidade *Box* que circula no *Buffer* chega à última posição (saída) do *queue* - *queueOnAtExit*.

Em particular, o evento *Startup* é bastante útil, tanto neste *AO* como em todos os outros criados, pois permite a inicialização do objecto, ou seja, no caso *Buffer* permite colocar a instância criada num estado inicial, por exemplo, já com algumas caixas quando a simulação começa.

Como já referido anteriormente, foram criadas funções para facilitar o acesso ao *Buffer*. Todo o código envolvido não será exposto, apenas será feita uma breve descrição do propósito de cada função.

As funções *BoxIn* e *BoxOut* inserem e retiram, respectivamente as caixas do *Buffer*. A *Size* retorna a quantidade de caixas presentes no *Buffer*. A *GetFirstBoxOut* retorna a caixa que se encontra na última posição (saída) do *Buffer*, ou seja, a primeira caixa a sair (segundo a regra *FIFO*). À semelhança desta função, existe um grupo de outras 3 funções que retornam uma caixa:

- segundo a sua posição no *Buffer* - *GetBoxByIndex*.
- segundo o atributo *Ref* da classe *Box*- *GetBoxByRef*.
- segundo o atributo *ID* da classe *Box*- *GetBoxByID*.

Existem ainda 2 funções que são idênticas às *GetBoxByRef* e *GetBoxByID* que para além de retornarem a caixa, também a removem do *Buffer* - *RemoveBoxByRef* e *RemoveBoxByID*, respectivamente.

4.3.1.3 Ícone e Animação

O ícone criado para o *Buffer* é o representado na figura 4.9, onde também é possível ver informações relativas ao objecto no balão presente na imagem.

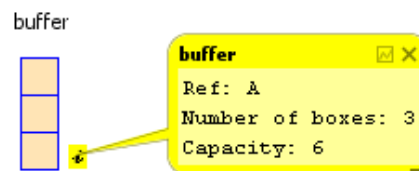


Figura 4.9: Ícone do *Buffer* em *runtime*

Em todos os ícones dos AOs criados (e este não é excepção), existe uma pequena caixa amarela com a letra 'i' no seu interior que quando clicada em *runtime*, abre um balão com informação sobre o objecto. A informação contida neste balão é a que o *AnyLogic* define por defeito na função *toString* (já mencionada em 4.2.1 e 4.2.2). No entanto, como foi referido anteriormente, esta função pode ser sobreposta de forma a aparecer no balão a informação que se deseje. Como o *Buffer* é um *Active Object*, ao contrário das classes *Box* e *Kanban* que são simples classes Java, a sobreposição do código relativo à função *toString* é escrito na janela propriedades do AO, secção *Advanced* no espaço designado *Additional class code* (figura 4.10).

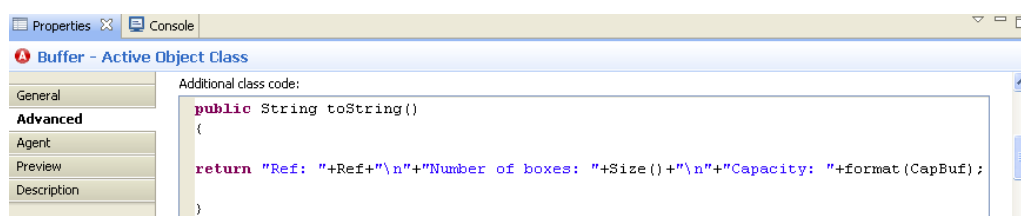


Figura 4.10: Função *toString* personalizada para o AO *Buffer*

A animação do *Buffer* é feita dinamicamente, ou seja, independentemente da capacidade que o modelador lhe atribua, a animação é criada automaticamente. Visualmente, a animação pretende mostrar qual a referência do *Buffer* (caso o objecto tenha uma), qual a sua capacidade, quantas

posições estão ocupadas e quantos componentes tem cada caixa. Sempre que é inserida ou retirada uma caixa do objecto, a animação criada traduzirá a situação actual. O mesmo acontece quando se altera o número de componentes de uma caixa.

Como referido, a animação é feita automaticamente e para tal usou-se a replicação dos objectos de apresentação, clonando-os segundo a capacidade atribuída ao *Buffer*. Neste caso, usou-se uma forma rectangular para simular uma posição no *Buffer* e também 2 textos: em que um é replicado o mesmo número de vezes que o rectângulo para mostrar a quantidade de componentes existentes na caixa presente na posição em causa; e o outro apenas mostra a referência atribuída ao *Buffer*, caso exista, senão mostra o nome da instância - por isto não necessita ser replicado. Criou-se também um botão de navegação “Back” que permite regressar ao AO anterior, tendo em conta a hierarquia de objectos do *AnyLogic*. A figura 4.11 demonstra a diferença da animação na fase de modelação do objecto e o resultado aquando da simulação do mesmo.

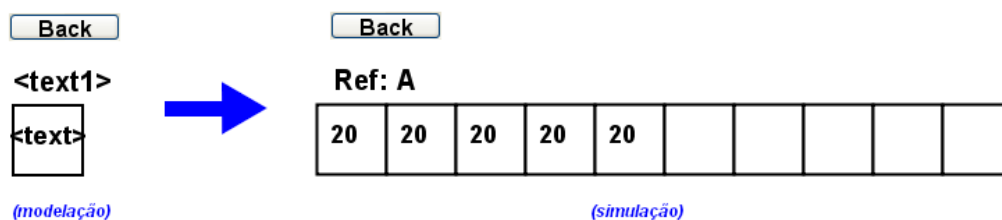


Figura 4.11: Animação na fase de modelação e durante a simulação do *Buffer*

O exemplo da figura 4.11 tem a referência “A”, tem capacidade de 10 e tem, neste momento, 5 caixas com 20 componentes em cada um.

4.3.2 Supermarket

Este AO pretende simular um supermercado. Um supermercado pode ser definido como um conjunto de *buffers* juntos no mesmo local. Desta forma, e recorrendo ao *Buffer* já desenvolvido, consegue-se criar outro AO para simular um local de armazém das caixas com componentes usadas na produção, por exemplo, num posto de trabalho ou numa máquina.

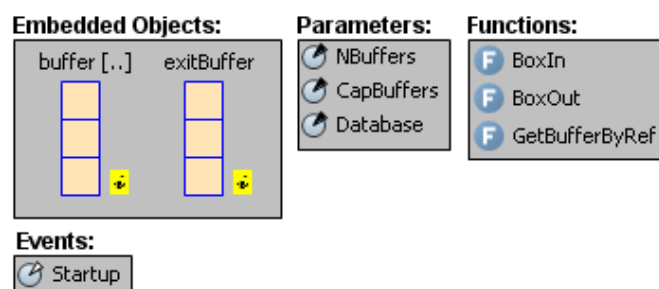


Figura 4.12: Lógica do *Supermarket*

Assim, o *Supermarket* é constituído por 2 objectos do tipo *Buffer*, em que um deles é replicado *n*-vezes (tantas quanto o valor do parâmetro *NBuffers*), e por um conjunto de atributos e métodos (figura 4.12). O *buffer*, que é replicado e que por isso passa a ser um conjunto de *buffers*, tem por função simular todos os *buffers* presentes num supermercado. O *exitBuffer* é também uma instância de *Buffer* e é usado para simular o local onde são colocadas as caixas vazias que são entregues no *Supermarket*, por exemplo por um *milkrun*.

O funcionamento do *Supermarket* é muito semelhante ao do *Buffer*, mas torna-se mais complexo pois não se trata de apenas um mas sim de vários *buffers*. Para agilizar o processo de introdução e remoção das caixas no supermercado, foram criados alguns métodos.

4.3.2.1 Atributos e Métodos

Os atributos criados para este AO, ao contrário do que aconteceu para o *Buffer*, foram apenas 3 parâmetros:

- *Nbuffers* - determina o número de *buffers* presentes no *Supermarket* está directamente associado ao parâmetro *Replication* do *buffer* embebido.
- *CapBuffers* - determina a capacidade dos *buffers* e está directamente ligado ao parâmetro *CapBuf* do *buffer* do objecto embebido.
- *Database* - este parâmetro serve como apontador para se efectuar a ligação à base de dados.

Em termos de métodos, o *Supermarket* contém um evento e 3 funções.

O evento presente é o *Startup*, e à semelhança do seu homónimo presente no *Buffer*, ocorre quando o objecto é criado e torna-se útil para inicializar o objecto já com algumas caixas, caso seja o desejado.

As funções criadas foram 3 e têm propósitos bastante simples:

- *BoxIn* - introduz uma caixa num determinado *buffer*; recorre ao método do objecto *Buffer* para introdução de caixas.
- *BoxOut* - remove uma caixa de um determinado *buffer*; recorre ao método do objecto *Buffer* para remoção de caixas.
- *GetBufferByRef* - retorna um *buffer* presente no conjunto de *buffers* replicados que tenha uma determinada referência.

A última função (*GetBufferByRef*) é usada para se descobrir o *buffer* com a referência pretendida e após esta descoberta, retira-se ou coloca-se uma caixa no supermercado usando as funções *BoxOut* e *BoxIn*, respectivamente.

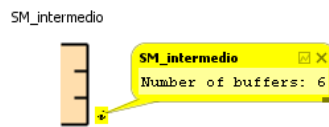


Figura 4.13: Ícone do *Supermarket* em runtime

4.3.2.2 Ícone e Animação

O ícone criado para este objecto (figura 4.13) foi baseado no desenho usado nos *Value Stream Maps* para identificar os supermercados.

A informação contida na caixa amarela do ícone apenas mostra quantos *buffers* existem no supermercado. A figura 4.14 contém o código que faz a sobreposição à função *toString* deste AO.

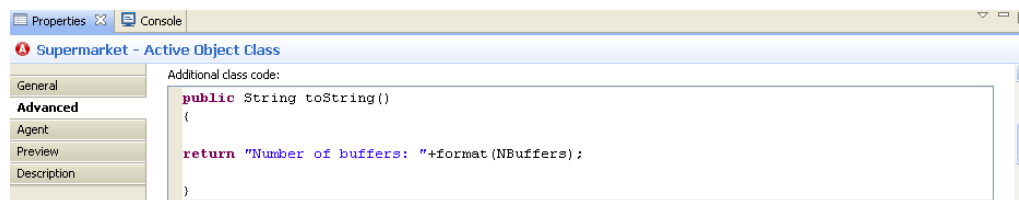


Figura 4.14: Função *toString* personalizada para o AO *Supermarket*

Tal como acontece com a do *Buffer*, também a animação do *Supermarket* é feita dinamicamente. É aproveitada a apresentação do objecto do tipo *Buffer* embebido para criar a do supermercado. Como é visível na figura 4.15, independentemente do número de *buffers* presentes no supermercado, todos aparecem na animação do objecto. Isto é conseguido usando, mais uma vez, a replicação de objectos. Neste caso, é replicada uma *Shape* de texto que contém o nome do *buffer* e também o objecto *buffer_Presentation* criado pelo *AnyLogic* e copia a animação do objecto *buffer* replicado tantas vezes quanto o número de *buffers*. À semelhança do realizado no AO *Buffer*, para os objectos de apresentação replicados não ficarem todos no mesmo sítio, usou-se os atributos *X* e *Y* presentes na secção *Dynamic* das propriedades dos objectos. Na figura 4.15 também é visível um menu de navegação em que os botões dos vários *buffers* são na realidade apenas um que é replicado, à semelhança do que acontece com os objectos de apresentação.

No exemplo da figura 4.15, o supermercado modelado contém 2 *buffers*: *Buffer 0* e *Buffer 1*. O primeiro tem referência “A” e o segundo “B”, e ambos têm 5 caixas cada com 20 componentes cada uma.

4.3.3 Workstation

Este AO foi criado com o objectivo de simular ou um posto de trabalho ou uma máquina. O *Workstation* tem 2 modos de produção:

- por *Kanban* - a produção é efectuada baseada num conjunto de *kanbans*.

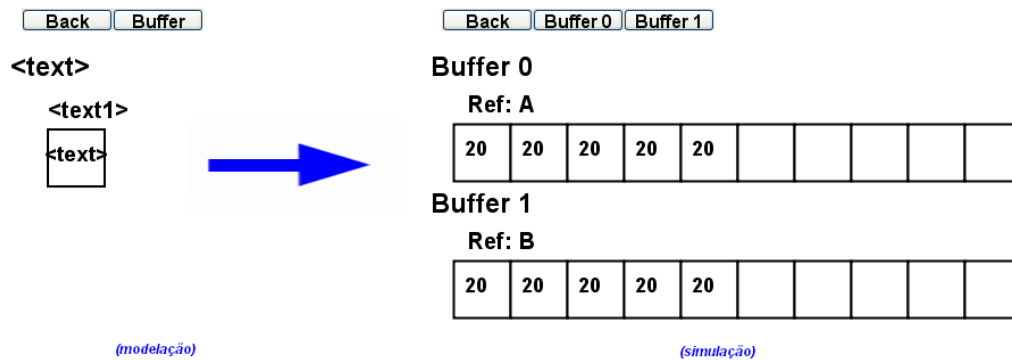


Figura 4.15: Animação na fase de modelação e durante a simulação do *Supermarket*

- por referência de supermercado - a produção é efectuada para reabastecer um conjunto de referências presentes num supermercado (pode ser associada uma ou mais referências ao AO).

É, assim, visível a influência da filosofia *Lean* na construção deste AO. O *Workstation* não produz para acumulação de inventário, apenas produz quando existem ou *kanbans* para produzir ou quando lhe é permitida a produção de certa referência (esta permissão é dada segundo um algoritmo que será explicado mais tarde), dependendo do modo de produção. Ambas as classe java criadas (4.2) são usadas neste AO: a *Box* é usada nos *Buffers* embebidos e a *Kanban* é a que “viaja” nos fluxogramas presentes no objecto.

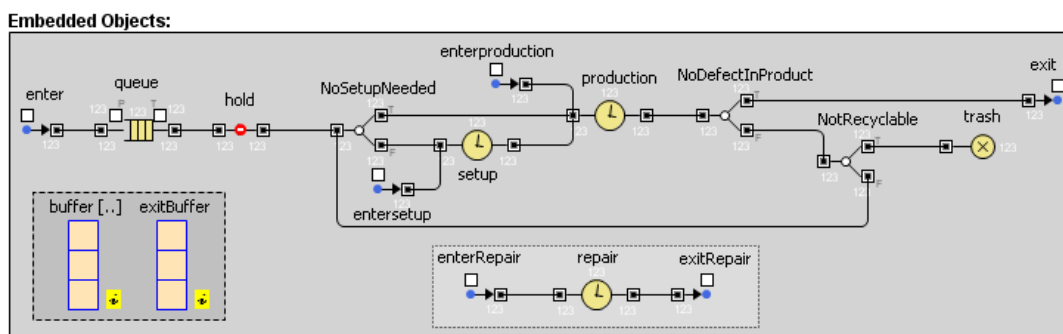


Figura 4.16: Lógica do *Workstation*

Como se pode observar na figura 4.16 existem bastantes objectos da *Enterprise Library* usados no *Workstation*, para além de 2 objectos do tipo *Buffer*, em que um deles é replicado.

Repare-se que existem 2 fluxogramas distintos: um deles bastante simples, pois apenas contém 3 objectos e que tem por função a simulação de reparações no *Workstation*; e outro maior e mais complexo que simula um fluxo normal de produção desde a entrada do *kanban* até à sua saída. No entanto, neste último, existem algumas ramificações que espelham 3 situações: a necessidade ou não de setup (*NoSetupNeeded*), o produto criado conter ou não defeitos (*NoDefectInProduct*) e caso haja defeito, a peça é retrabalhada ou vai para sucata (*NotRecyclable*).

Embora o fluxo de entidades seja facilmente compreensível apenas observando o fluxograma, existem alguns aspectos que merecem uma atenção especial. Assim, será feita uma descrição do caminho percorrido pelas entidades do tipo *kanban* ao longo deste objecto:

- Um *kanban* entra no *Workstation*, passa pelo *enter* e quando chega à posição de saída do *queue* (mas sem sair) são verificadas as seguintes condições:
 - se existem componentes necessários para produzir a ordem do *kanban*;
 - se não existem entidades em nenhum dos 3 *delay* (*setup*, *production* ou *repair*).
- Se todas as condições anteriores forem verdadeiras, o *hold* é desbloqueado para deixar passar a entidade, voltando a bloquear quando esta passa; no caso de alguma das condições ser falsa, o *hold* é bloqueado e assim permanece até as condições serem todas verdadeiras.
- A seguir a entidade entra no *NoSetupNeeded*, que é um objecto do tipo *Select Output* e portanto permite uma ramificação no fluxograma, a primeira das 3 mencionadas em cima; a entidade passa por este objecto e é redireccionada para o *setup* ou para o *production* dependendo se o tipo de produção é diferente do anteriormente produzido no *Workstation*.
- Se a entidade seguir para o *setup*, ficará retida durante o tempo de *setup* e depois durante o tempo de produção; se o *kanban* seguir directamente para o *production*, apenas ficará retido durante o tempo de produção.
- A entidade sai do *production* e entra noutro *Select Output*, o *NoDefectInProduct* e que direcciona a entidade para a saída do *Workstation* (para o *exit*) caso a condição ‘não contém defeito’ for verdadeira; caso esta seja falsa, o *kanban* é redireccionado para o *NotRecyclable*.
- Caso a entidade entre neste último, é verificada a condição ‘não é reciclável’ e se for verdadeira, a entidade é redireccionada para o *trash*; para o caso de ser falsa, a entidade é reencaminhada para o *NoSetupNeeded* de forma a ser retrabalhada.

Após esta descrição do fluxo normal de produção, falta explicar a função tanto dos objectos *entersetup* e *enterproduction*, como dos constituintes do fluxograma relativo à simulação de avarias. A forma escolhida para a simulação de avarias no *Workstation* passa por um evento programado para disparar de x em x tempo. Como o evento ocorre dependendo do tempo, é necessário avaliar em que estado se encontra o *Workstation* quando tal acontece. O *AO* pode estar em *setup*, em produção ou parado. A acção do evento reage dependendo da situação:

- Se estiver em *setup* - o tempo que falta para o *setup* terminar é guardado numa variável e a entidade é retirada do *setup* para ser inserida no *enterrepair*.
- Se estiver em produção - o tempo que falta para a produção terminar é guardado numa variável e a entidade é retirada do *production* para ser inserida no *enterrepair*.
- Se estiver parado - é inserida uma entidade no *enterrepair*, criada de propósito para simular a reparação do *Workstation*.

Quando uma entidade entra no *enterrepair*, esta passa para o *repair* onde fica retida durante o tempo de reparação. Mal este termina, a entidade sai pelo *exitrepair* e dependendo da situação ocorrida aquando do disparo do evento, ela é:

- inserida no *entersetup* e o tempo que faltava retomado, para a situação de setup.
- inserida no *enterproduction* e o tempo que faltava retomado, para a situação de produção.
- descartada caso o *Workstation* estivesse parado.

Esta técnica foi baseada num exemplo denominado *DelaySuspendable* existente no *AnyLogic* no modelo *Utilities* presente nos *Sample Models*.

Os objectos embebidos do tipo *Buffer* têm um funcionamento análogo aos idênticos presentes e descritos em 4.3.2.

4.3.3.1 Atributos

Este *AO*, de todos os criados, é aquele que tem um maior número de atributos, tanto parâmetros como variáveis (figura 4.17).

Parameters:	Variables:
NBuffers	ExitKanbans
CapBuffers	CVneeds
Type	CVneedsMR
ProductionTime	Kanbans
SetupTime	ProductionRefs
BreakdownTime	setupactual
RepairTime	queueorder
ExitSM	ncomponents
Database	CurrentRef
ProductionAnimationGuideShape	StopProduction
SetupAnimationGuideShape	breakdownplace
EntityProductionShape	timeleft
EntitySetupShape	
NoDefectInProductCondition	
NotRecyclableCondition	
ProductionSelector	

Figura 4.17: Atributos do *Workstation*

O *Workstation* tem 16 parâmetros, mas para melhor compreensão existem alguns que podem ser agrupados:

- *SetupTime*, *ProductionTime*, *RepairTime*, *BreakdownTime* - definem os tempos de produção, setup, reparação e entre avarias.

- *SetupAnimationGuideShape*, *ProductionAnimationGuideShape*, *EntityProductionShape*, *EntitySetupShape* - funcionam como apontadores para as *Shapes* existentes fora do objecto e que pretendem ser a animação do objecto (por defeito, estes parâmetros apontam para *Shapes* existentes no *Workstation*).
- *NoDefectInProductCondition*, *NotRecyclableCondition* - determinam as condições para determinar o caminho da entidade que entra nos objectos *NoDefectInProduct* e *NotRecyclable*.

Há semelhança do que acontece com o *AO Supermarket*, existem parâmetros que determinam o número e tamanho dos *buffers* embebidos (figura 4.16), respectivamente, *NBuffers* e *CapBuffers*. Também como acontece no *Supermarket*, foi criado um parâmetro que serve como apontador para a ligação à base de dados (*Database*).

Restam 3 parâmetros cujas funções são importantes para o modo de produção do *Workstation*:

- *Type* - determina o modo de produção do *Workstation*, por *kanban* ou por referência de supermercado.
- *ExitSM* - aponta para o objecto do tipo *Supermarket* onde se encontram os *buffers* que este *Workstation* reabastece (parâmetro existente, apenas, quando o modo de produção é por referência).
- *ProductionSelector* - algoritmo que selecciona a referência que o *Workstation* vai produzir (parâmetro existente, apenas, quando o modo de produção é por referência); por defeito este parâmetro tem o valor resultante de uma função (*DefaultProductionSelector*).

O último parâmetro é dinâmico, e por isso idêntico a um evento, pois é necessário que funcione como uma função e que o seu valor seja calculado de cada vez que é acedido.

Em termos de variáveis, este *AO* contém 5 *Collection Variables* e 7 ditas simples.

As CV *Kanbans* e *ExitKanbans* são uma lista dos *kanbans* que entram e saem, respectivamente, deste *AO*. A *ProductionRefs* contém as referências que este *Workstation* pode produzir. Tanto a *CVneeds* como a *CVneedsMR* são auxiliares à programação de funções ligadas ao *AO Milkrun* que serão explicadas mais tarde.

As 7 variáveis simples são todas auxiliares para o correcto funcionamento da lógica do objecto e por isso não serão explicadas em pormenor.

4.3.3.2 Métodos

Como é visível através da figura 4.18, existem muitos eventos e funções neste *AO*.

Após uma breve observação, facilmente se nota uma ligação entre a maioria dos eventos e os objectos usados no funcionamento do *Workstation*. À semelhança do que acontece no *Buffer*, foram criados eventos associados aos existentes nos objectos da *Enterprise Library* de modo que o modelador do sistema possa executar acções personalizadas quando uma entidade entra, sai ou passa simplesmente por um desses objectos. Observando os seus nomes, é intuitivo ligar o evento tanto ao objecto como ao momento e situação em que ele ocorre. Alguns destes eventos já contêm

Events:		Functions:
 Startup	 enterProductionOnEnter	 KanbanIn
 resumework	 productionOnEnter	 KanbanOut
 breakdown	 productionOnExit	 RemoveComponents
 enterOnEnter	 NoDefectInProductOnEnter	 GetBufferByRef
 queueOnEnter	 NoDefectInProductOnExitTrue	 GetQuantityFirstBoxOfBuffer
 queueOnAtExit	 NoDefectInProductOnExitFalse	 GetTotalQuantityOfBuffer
 queueOnExit	 exitOnExit	 CheckComponents
 holdOnEnter	 NotRecyclableOnEnter	 ConsumeComponents
 NoSetupNeededOnEnter	 NotRecyclableOnExitTrue	 DefaultProductionSelector
 NoSetupNeededOnExitTrue	 NotRecyclableOnExitFalse	 NProductionRefsOnRed
 NoSetupNeededOnExitFalse	 trashOnEnter	 LookForProductionRefOnRed
 enterSetupOnEnter	 enterRepairOnEnter	 ProductionRefsPresentation
 setupOnEnter	 repairOnEnter	
 setupOnExit	 repairOnExit	
	 exitRepairOnExit	

Figura 4.18: Métodos do *Workstation*

código necessário ao funcionamento do *AO*, no entanto, o modelador tem acesso directo a ele em cada instância do objecto e pode alterar desta forma o comportamento de cada *Workstation*. Por exemplo, por defeito este *AO* foi construído retirando os tempos de produção e setup de uma base de dados, mas caso o modelador deseje, pode apagar (ou comentar) o código referente à alteração dos tempos e colocar um personalizado que espelhe melhor o sistema a simular.

Existem no entanto 3 eventos que importa explicar individualmente, um deles é o *Startup* e os outros 2 são eventos que ocorrem ao fim de um tempo definido:

- *Startup* - à semelhança dos *AO* explicados anteriormente este evento ocorre aquando da criação deste objecto e torna-se útil para a sua inicialização.
- *resumework* - este evento tem como tempo de ocorrência a escala de tempo da simulação e serve para fazer uma verificação contínua da possibilidade do objecto produzir; é importante e útil quando este *AO* se encontra parado e é necessário testar continuamente se pode recomeçar a trabalhar ou manter-se parado.
- *breakdown* - o intervalo de tempo entre ocorrências deste evento é o da soma dos parâmetros *BreakdownTime* e *RepairTime* de forma a ocorrer quando o modelador desejar; é este evento que tem a acção descrita anteriormente para a simulação de reparação do *Workstation*.

Em termos de funções, este *AO* é aquele que contém mais. Existem funções simples como as de inserir e retirar *kanbans* do objecto, respectivamente, *KanbanIn* e *KanbanOut*. Algumas das funções presentes neste *AO* são apenas usadas dentro de outras e foram criadas com o propósito de facilitar a programação das mais complexas. Assim, apenas as que mostram interesse acrescido irão ser explicadas de seguida:

- *GetBufferByRef* - funcionamento análogo ao exposto durante a explicação da sua homónima no *AO Supermarket*.
- *GetQuantityFirstBoxOfBuffer*, *GetTotalQuantityOfBuffer* - ambas retornam o atributo *quantity* da classe *Box*, mas a primeira retorna apenas a quantidade da primeira caixa do *buffer* e a segunda retorna o somatório de componentes presentes no *buffer* inteiro.
- *CheckComponents* - tem como funcionalidade verificar se existem componentes suficientes para produzir a referência actual.
- *ConsumeComponents* - tem como funcionalidade simular o consumo de componentes, retirando-os da respectiva caixa.
- *ProductionRefsPresentation* - esta função, ao contrário de todas as outras, serve apenas para a animação e será explicada na secção seguinte (subsubsec:wsiconeanimacao).

Existe ainda outra função de grande importância, *DefaultProductionSelector*, já referida anteriormente mas que a sua explicação será feita no capítulo 5, pois já foi criada a pensar no caso de estudo que será apresentado.

4.3.3.3 Ícone e Animação

O ícone deste *AO* (figura 4.19) pretende dar a entender que se trata de um objecto que simula um posto de trabalho. No entanto, a simulação é virtual e por isso o *Workstation* pode, e deve, servir para simular qualquer ponto de produção, seja este um posto de trabalho ou mesmo uma máquina.

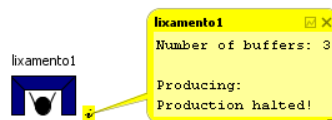


Figura 4.19: Ícone do *Workstation* em runtime

O balão de informação deste *AO* contém dados sobre o número de *buffers* presentes no *Workstation* e a sua actividade actual. No exemplo da figura 4.19, a instância *lixamento1* contém 3 *buffers* e neste momento a sua produção encontra-se parada.

O código que permite colocar esta informação no referido balão encontra-se exposto na figura 4.20.

Observando a figura 4.21 é possível reparar que a animação criada para este *AO* contém informação relativa ao número de *buffers* existentes, às referências o *Workstation* pode produzir e a sua actividade actual. Foram criadas através dos objectos do tipo *Presentation*, uma espécie de bancada com um operador. Esta bancada muda de cor dinamicamente conforme o *Workstation* se encontra em setup, produção, reparação ou parado. Se nenhum dos parâmetros relativos às *Shapes* de animação forem alterados, quando o objecto se encontra em produção é possível reparar numa

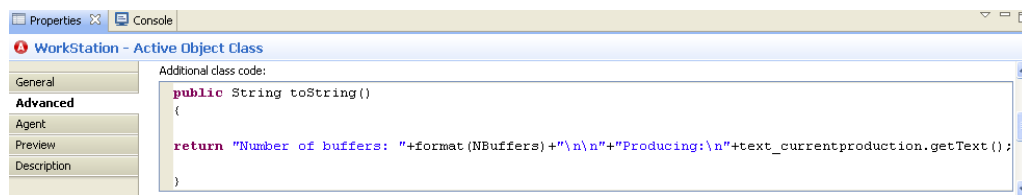


Figura 4.20: Função *toString* personalizada para o AO *Workstation*

caixa que percorrer a bancada ao longo do tempo de produção definido. Desta forma, é possível obter uma visualização gráfica de quanto tempo falta para a produção terminar sem termos um acesso directo a um número.

Do lado direito da bancada é possível observar os *buffers* presentes e a informação relativa a quantidades de componentes e número de caixas. Para tal usou-se, à semelhança do que foi feito para o *Supermarket*, o objecto do tipo *Embedded Object Presentation* do *Buffer* para criar de forma dinâmica os *buffers* do *Workstation*. O mesmo se aplica aos botões que formam um menu de navegação na parte superior.

Para o exemplo presente na figura 4.21, o *Workstation* encontra-se parado, contém 3 *buffers* e pode produzir as referências “A”, “B” e “C”. Os *buffers* presentes nesta instância encontram-se com 2 caixas cada e têm capacidade de 6. Nota-se também que as caixas não estão todas cheias.

4.3.4 Milkrun

O *Milkrun* é o 3º dos principais/grandes AOs criados. Este, simula um abastecedor logístico que tipicamente transporta paletes de ou para um supermercado para postos de trabalho. Assim, aliando este AO aos *Supermarket* e *Workstation*, é possível a simulação de um sistema de produção com, por exemplo, alguns postos, um supermercado e um *milkrun*, com este a reabastecer os postos a partir do supermercado.

Este AO tem por função percorrer os destinos um por um e em cada um deles executar as tarefas relativas às trocas de caixas, recolhendo as caixas vazias dos postos, trocando-as por cheias no supermercado e devolvendo-as aos postos. Assim, o funcionamento do *Milkrun* passa por 2 ciclos que serão perceptíveis com a explicação da máquina de estados da figura 4.22 que descreve o comportamento deste AO.

Quando a simulação começa, o estado inicial do *Milkrun* é o *Start*, que não contém qualquer acção associada. É de fácil conclusão que o objecto só fica neste estado uma vez durante toda a simulação, pois após o disparado da transição *t1*, entra-se num de dois ciclos que percorrem os outros 3 estados existentes: *Idle*, *Path* e *Task*.

A transição *t1* é sempre disparada mal o *Milkrun* se encontre no estado *Start*. Assim, em termos práticos, o primeiro estado do *Milkrun* é o *Path*, que simula o trajecto entre dois destinos. Simulação feita recorrendo ao AO *pathMilkrun* (este componente será explicado mais à frente) criado apenas com o propósito de simular o caminho que um *milkrun* percorre.

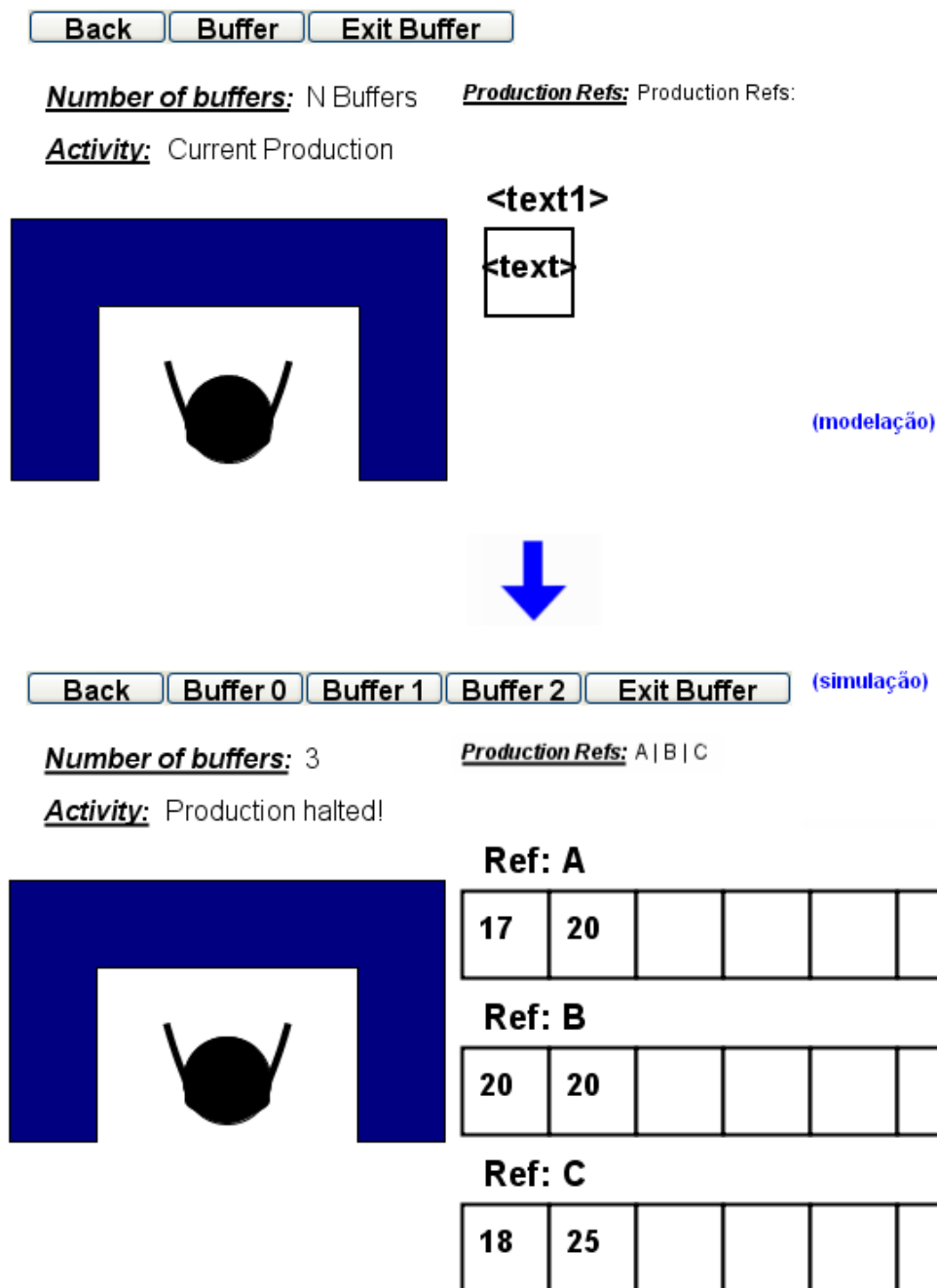


Figura 4.21: Animação na fase de modelação e durante a simulação do *Workstation*

A condição de disparo da transição *t2* é a mensagem “*Milkrun has arrived*” recebida do *path-Milkrun*. Assim, com a recepção desta mensagem, o estado seguinte é o *Task* que simula a troca de caixas no destino, recorrendo ao *AO taskMilkrun* (este componente será explicado mais à frente), também criado apenas para este propósito.

A transição *t3* é em tudo semelhante à *t2*, excepto na mensagem que recebe. Na *t3* a mensagem que provoca a transição é “*Task completed*” e é enviada pelo *taskMilkrun*. Esta transição está

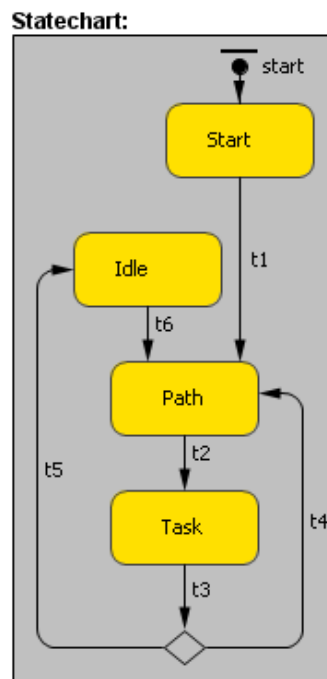


Figura 4.22: Máquina de estados do funcionamento do *Milkrun*

ligada a outras duas, ou seja, após o disparo desta serão testadas as condições das seguintes ($t4$ e $t5$) para determinar qual o estado seguinte: de novo o *Path* ou o *Idle*. As transições $t4$ e $t5$ têm condições exclusivas, ou seja, garante-se que apenas uma delas é disparada. A condição de disparo da $t4$ é o local onde o *Milkrun* se encontrar ser qualquer um dos destinos possíveis que não o primeiro, por norma este é o supermercado. Já a transição $t5$ só é disparada se a condição for contrária, ou seja, o local actual ser o primeiro da lista de destinos.

Para o caso da condição de $t4$ ser satisfeita, o próximo estado será novamente o *Path* e assim se repetirá este ciclo envolvendo este estado e o *Task*. No caso de ser disparada a $t5$, o estado seguinte será o *Idle* que simula a espera do *Milkrun* até que possa partir de novo para percorrer todos os destinos. Esta espera é determinada por um tempo de ciclo que o *Milkrun* cumpre e que demonstra mais uma vez a influência da filosofia *Leanneste* trabalho.

Por fim, resta explicar a transição $t6$ que é disparada quando o tempo de ciclo referido termina. Caso este tempo já tenha terminado quando o *Milkrun* chega ao *Idle*, $t6$ é disparada de imediato.

São bem visíveis os 2 ciclos que o *Milkrun* executa repetidamente.

O *AnyLogic* permite associar acções tanto aos estados como às transições. No caso das etapas, pode-se associar acções que são executadas quando se entra e quando se sai do estado em causa. Já para as transições, as acções são executadas quando as condições de disparo das mesmas são satisfeitas.

De um ponto de vista conceptual, o estado *Start* e a transição $t1$ não fazem sentido existir pois o estado não executa qualquer acção e a que é executada aquando do disparo de $t1$ podia ser realizada no ponto de entrada do gráfico, *start*. A razão da existência tanto do estado com da

transição passa pela forma como a simulação é executada no *AnyLogic* e pela sequência do código compilado. Ou seja, se tentássemos correr a simulação do *Milkrun* sem o estado *Start* nem a transição *t1*, passando a acção desta para o ponto inicial (*start*), ocorreria um erro. Pois a acção em causa necessita aceder a variáveis que ainda não foram inicializadas e apenas o serão após a execução da acção do ponto inicial *start* do gráfico de estados.

A figura 4.23 mostra os objectos usados no *Milkrun*. São usadas 2 instâncias do *AO Buffer*: uma para simular o *buffer* de caixas vazias (*EmptyBoxesBuffer*) que o *milkrun* transporta dos postos para o supermercado e outra para simular o *buffer* de caixas cheias (*FullBoxesBuffer*) que são transportadas do supermercado para os postos. Foram também usados 2 objectos da *Enterprise Library*, um *Source* ligado a um *Queue*, para ser possível simular uma animação do local onde o *milkrun* espera que o tempo de ciclo termine para voltar a percorrer os vários destinos, ou seja, quando se encontra no estado *Idle*. Quanto à figura 4.23, é importante acrescentar que foram criados 2 *Active Objects* com o propósito de simular os caminhos percorridos (*pathMilkrun*) pelo *milkrun* e as tarefas realizadas (*taskMilkrun*) nos destinos. Note-se que ambos os *AO* são replicados o número de destinos que o *milkrun* tem de percorrer.

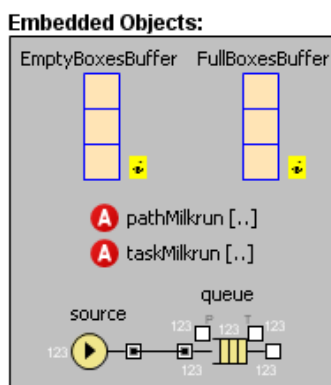
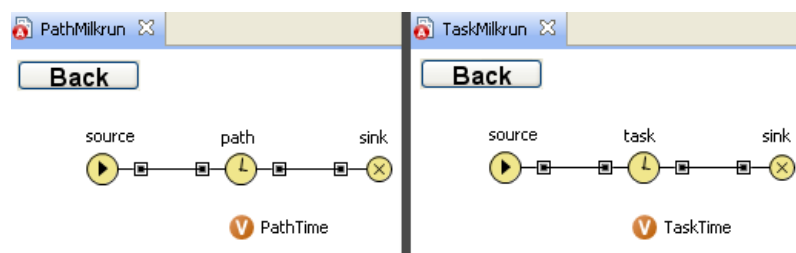


Figura 4.23: Lógica do *Milkrun*

Como se pode ver pela figura 4.24, ambos os *AO* usam objectos da *Enterprise Library*: um *Source*, um *Delay* e um *Sink*. E ambos têm uma variável que está associada ao tempo do *Delay*. A simulação, tanto do *pathMilkrun* como do *taskMilkrun*, é conseguida retendo uma entidade no *Delay* durante um certo tempo. A entidade usada é a *Entity* já mencionada no capítulo anterior em 3.2. Desta forma, os dados necessários para simular um *milkrun* passam por saber não as distâncias entre os destinos, mas sim o tempo que demora a viajar de um para outro.

Estes *AO* são usados em conjunto com a máquina de estados da figura 4.22 para simular o comportamento do *Milkrun*. Facilmente se conclui que o *pathMilkrun* está directamente ligado ao estado *Path* e o *taskMilkrun* ao *Task*.

- No caso do *Path*:
 - Quando o *Milkrun* entra no estado *Path*, é dada a ordem de criação de uma entidade no *source* do *pathMilkrun*;

Figura 4.24: *pathMilkrun* e *taskMilkrun*

- Assim, segundo as regras de fluxo de entidades da biblioteca *Enterprise*, a entidade criada chega ao objecto *path* (que é do tipo *Delay*) e fica retida o tempo definido pela variável *PathTime*;
 - Após terminado o tempo associado ao *path*, a entidade sai do objecto e é descartada no *sink*.
 - No entanto, no evento *onExit* do *path*, é enviada uma mensagem para a máquina de estados que vai disparar a transição *t2* que altera o estado do *Milkrun* de *Path* para *Task*.
- O caso do *Task* é análogo:
 - Quando o *Milkrun* entra no estado *Task*, é dada a ordem de criação de uma entidade no *source* do *taskMilkrun*;
 - Assim, segundo as regras de fluxo de entidades da biblioteca *Enterprise*, a entidade criada chega ao objecto *task* (que é do tipo *Delay*) e fica retida o tempo definido pela variável *TaskTime*;
 - Após terminado o tempo associado ao *task*, a entidade sai do objecto e é descartada no *sink*.
 - No entanto, no evento *onExit* do *task*, é enviada uma mensagem para a máquina de estados que vai disparar a transição *t3* que altera o estado do *Milkrun* de *Task* para *Idle* ou novamente para *Path* dependendo se é disparada a transição *t5* ou *t4*, respectivamente.

4.3.4.1 Atributos

Na figura 4.25 estão presentes os parâmetros e as variáveis usadas no *Milkrun*.

O parâmetro *Supermarket* é usado como apontador para o supermercado existente de entre os destinos do *milkrun*. O *Ndestinations* determina o número total de destinos do *Milkrun* e está directamente associado à propriedade *Replication* dos AOs *pathMilkrun* e *taskMilkrun*. Os parâmetros *CapEmptyBoxesBuffer* e *CapFullBoxesBuffer* são usados para determinar a capacidade dos *buffers* *EmptyBoxesBuffer* e *FullBoxesBuffer*. O *CycleTime* - determina o tempo de ciclo do *milkrun*, ou seja, de quanto em quanto tempo ele parte para mais uma volta pelos destinos.

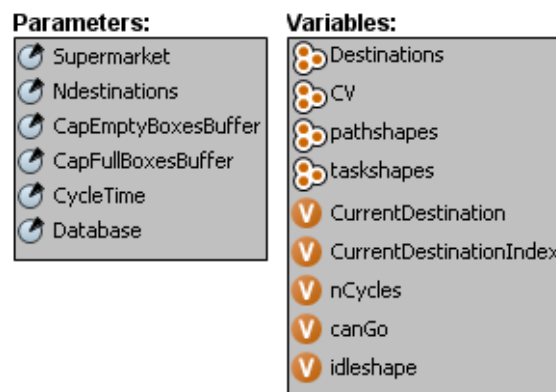


Figura 4.25: Atributos do *Milkrun*

Existe ainda um outro parâmetro, o *Database* que igualmente ao mesmo existente nos AOs *Supermarket* e *Workstation* serve como apontador para a ligação à base de dados.

No *Milkrun* existem 9 variáveis, 4 delas são *Collection Variables*. Destas 4, a *CV* é auxiliar ao bom funcionamento e por isso não tem uma justificação tangível. Já as outras 3 têm:

- *Destinations* - lista onde são guardados os destinos do *Milkrun*.
- *pathshapes* - lista onde são guardadas as formas do tipo *Presentation (Shapes)* usadas para a animação externa dos caminhos percorridos pelo *Milkrun*.
- *taskshapes* - lista onde são guardadas as *Shapes* usadas na animação das tarefas realizadas pelo *Milkrun*.

Quanto às outras 5 variáveis:

- *CurrentDestination* - guarda o destino actual do *Milkrun*.
- *CurrentDestinationIndex* - guarda o índice do destino actual guardado na lista *Destinations*.
- *nCycles* - guarda o número de ciclos feitos pelo *milkrun* até ao momento (esta variável é incrementada quando o *Milkrun* passa para o estado *Idle*).
- *canGo* - é usada como auxiliar ao funcionamento do AO, determina se este pode ou não iniciar um novo ciclo se o tempo de ciclo já terminou ou não.
- *idleshape* - esta variável é usada apenas para determinar qual a *Shape* responsável para a animação externa do *Milkrun*.

Pode-se agrupar esta última (*idleshape*) com as *pathshapes* e *taskshapes* como variáveis facilitadoras da animação externa do *Milkrun*. A forma de alterar estas variáveis é através do código, e para auxiliar o modelador, no evento *Startup* existe instruções para tal. É importante referir que estas 3 variáveis não são obrigatórias para o funcionamento do *Milkrun*.

No entanto a *Collection Variable Destinations* é necessária para o funcionamento do objecto, por isso, também existem instruções relativas a ela no evento *Startup*.

4.3.4.2 Métodos

À semelhança do *Workstation*, também neste *AO* existem bastantes eventos e funções (figura 4.26).

Events:		Functions:
⚡ Startup	⚡ IdleOnEnter	F GetIndexNextDestiny
⚡ timecycle	⚡ IdleOnExit	F NextDestiny
⚡ StartOnEnter	⚡ t1Action	F ordershapes
⚡ StartOnExit	⚡ t2Action	F UnloadWS
⚡ PathOnEnter	⚡ t3Action	F UnloadSM
⚡ PathOnExit	⚡ t4Action	F LoadSM
⚡ TaskOnEnter	⚡ t5Action	F LoadWS
⚡ TaskOnExit	⚡ t6Action	F GetPreviousDestinyName
		F GetDestinationsNamesList

Figura 4.26: Métodos do *Milkrun*

Quase todos os eventos presentes no *Milkrun* estão relacionados com a máquina de estados (figura 4.22) que espelha o seu funcionamento. Estes eventos que ocorrem:

- à entrada e à saída das etapas - *StartOnEnter* e *StartOnExit* para a etapa *Start*, etc.
- com o disparo de uma transição - *t1Action* para a transição *t1*, etc.

A forma como foram denominados, torna intuitiva a sua ligação com as etapas e transições respectivas.

Porém, existem 2 eventos que não se incluem neste grupo: são eles o *Startup* e o *timecycle*.

- O *Startup* é análogo ao evento com o mesmo nome existente no *Workstation* e no *Supermarket*. Ocorre quando a instância do *AO* é criada e permite desta forma a sua inicialização, que no caso do *Milkrun* tem factor obrigatório pois é necessário introduzir o nome dos destinos que o *milkrun* irá percorrer para que este objecto funcione.
- O *timecycle* é um evento que dispara ao fim de um tempo definido no parâmetro *CycleTime*. Este evento em conjunto com a variável booleana *canGo* (refsubsubsec:mratributos) tem por função impedir que o *milkrun* inicie um novo ciclo antes do tempo definido.

Como acontece com o *Workstation* também no *Milkrun* foram criadas umas funções que ajudam no funcionamento do objecto e outras que ajudam na apresentação em termos de animação do mesmo.

Posto isto, são 6 as funções que auxiliam a lógica deste *AO*. Duas delas estão relacionadas com a obtenção do próximo destino do *milkrun*, calculado no final de cada ciclo (na acção da transição *t4* ou na da *t5*):

- *GetIndexNextDestiny* - retorna o índice do próximo destino; a procura deste índice é feita na *Collection Variable Destinations*.
- *NextDestiny* - altera as variáveis relativas ao destino: *CurrentDestinationIndex* e *CurrentDestination*.

As outras quatro funções são as tarefas que o *Milkrun* realiza, duas para quando o destino é um *Supermarket* e duas para quando é um *Workstation*:

- *UnloadSM* - executa a tarefa relativa a descarregar as caixas vazias do *Milkrun* para o destino caso seja um *Supermarket*.
- *UnloadWS* - executa a tarefa relativa a descarregar as caixas cheias do *Milkrun* para o destino caso seja um *Workstation*.
- *LoadSM* - executa a tarefa relativa a carregar as caixas cheias do destino caso seja um *Supermarket* para o *Milkrun*.
- *LoadWS* - executa a tarefa relativa a carregar as caixas vazias do destino caso seja um *Workstation* para o *Milkrun*.

Quanto às funções que ajudam na animação do objecto, existem 3:

- *ordershapes* - apenas é executada caso as *Collection Variables pathshapes* e *taskshapes* não estiveram vazias; tem por função ordenar as *Shapes* introduzidas nestas variáveis conforme a ordem dos destinos presentes em *Destinations*.
- *GetPreviousDestinyName* - retorna o nome do destino anterior ao actual para que possa aparecer na animação de e para onde se está a dirigir o *Milkrun*.
- *GetDestinationsNamesList* - retorna um texto no formato *String* com os nomes de todos os destinos do *Milkrun*.

4.3.4.3 Ícone e Animação

A figura 4.27 mostra o ícone usado para representar o *Milkrun* e pretende dar a entender que o objecto se trata de um transportador.

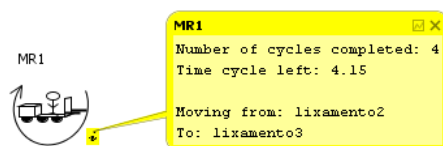


Figura 4.27: Ícone do *Milkrun* em runtime

O balão de informação deste *AO* contém dados relativos ao número de ciclos já efectuados pelo *Milkrun*, o tempo restante para se iniciar o próximo ciclo e a sua actividade actual. No exemplo da

figura 4.27, o *milkrun* já fez 4 ciclos completos, faltam pouco mais de 4,15 segundos para poder iniciar outro e está neste momento a movimentar-se do “lixamento2” para o “lixamento3”.

Esta informação é conseguida através da função *toString* da figura 4.28.

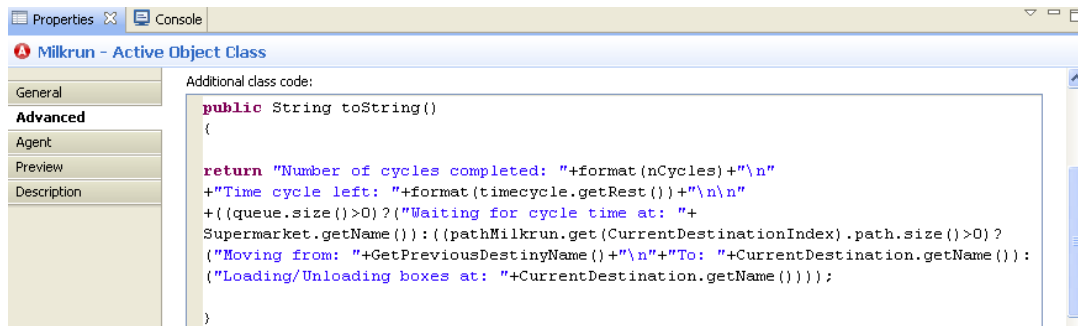


Figura 4.28: Função *toString* personalizada para o AO *Milkrun*

A animação criada para este AO (figura 4.29) contém informação sobre o número de ciclos completos pelo *milkrun*, o tempo restante para poder iniciar o próximo ciclo, a actividade do *milkrun* no momento (a movimentar-se de e para onde ou a realizar tarefas em que local), a lista de destinos do *milkrun* e os *buffers EmptyBoxesBuffer* e *FullBoxesBuffer*. Como acontece nos outros AO criados, também para este se criou um menu de navegação na zona superior. Para os *buffers* foi usada a animação interna de cada, como acontece com o *Workstation*.

Caso a variável *idleshape* e as *Collection Variables pathshapes* e *taskshapes* sejam usadas, é possível fazer uma animação exterior semelhante à da figura 4.30. O desenho do *milkrun* já existe neste AO e pode ser usado por qualquer animação externa pois já se encontra associado às entidades criadas dentro do *pathMilkrun*.

Neste exemplo é visível a existência de um supermercado denominado “SM” e dois *Workstations* “WS1” e “WS2” que o *milkrun* percorre. A imagem representa um momento em que o *milkrun* está a percorrer o caminho entre “WS1” e “WS2”.

4.4 Base de Dados

A introdução de Base de Dados (BD) surgiu com a necessidade de criar um repositório que contivesse toda a informação necessária à inicialização dos AOs e também em relação a valores das variáveis que alteram com o passar do tempo de simulação. Assim, aproveitando os eventos criados para a personalização dos objectos, usou-se os seus momentos de ocorrência para executar o código que permite o acesso à BD e a alteração de parâmetros e variáveis dos objectos de forma a se conseguir o funcionamento desejado.

Com o uso de uma BD, torna-se a inicialização dos objectos mais simples, pois apenas é necessário alterar valores nas suas tabelas em vez de se escrever código Java em cada instância dos AOs.

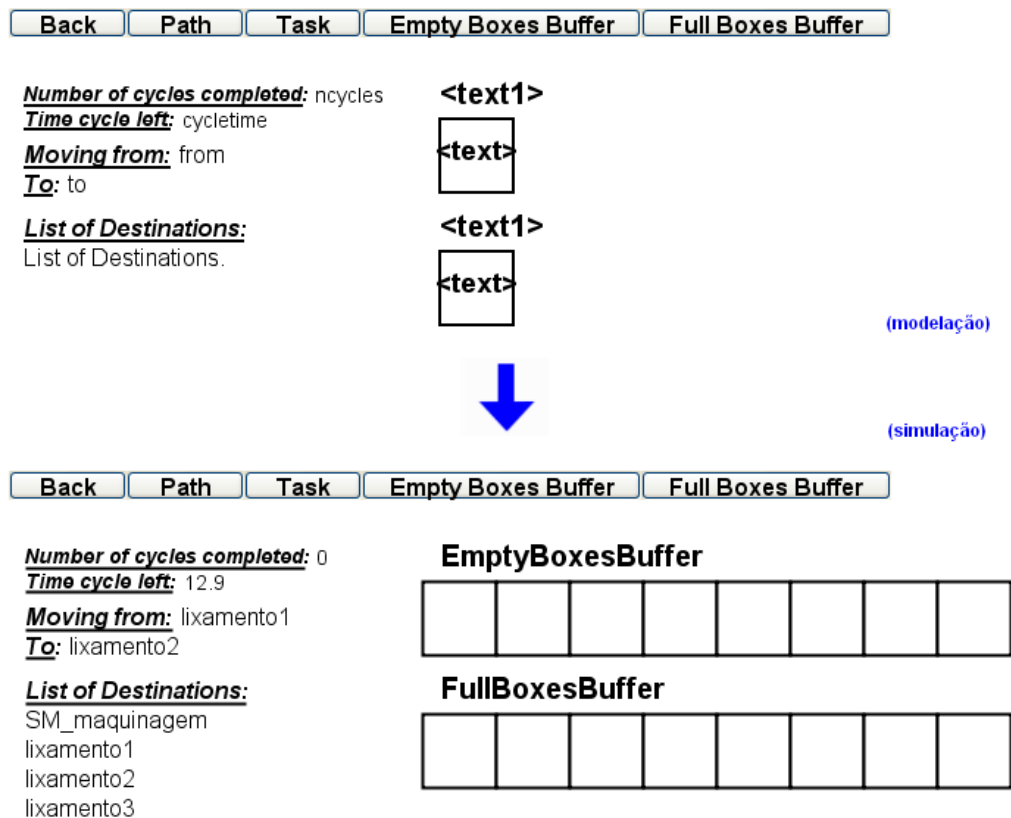


Figura 4.29: Animação na fase de modelação e durante a simulação do *Milkrun*

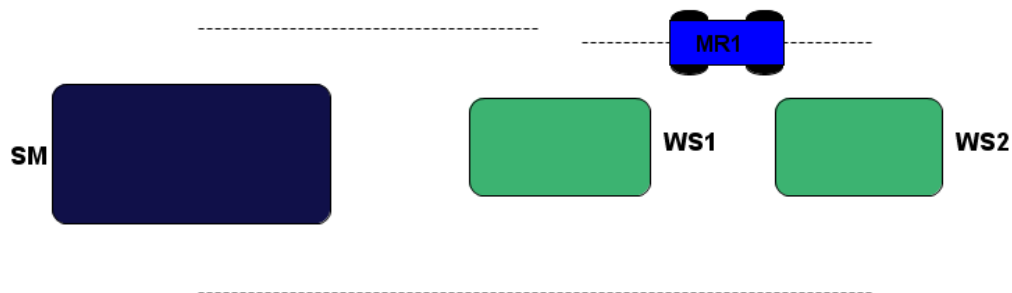


Figura 4.30: Exemplo de animação externa envolvendo um *milkrun*

Mais concretamente, os objectos criados que usam a BD para inicialização são o *Supermarket*, o *Workstation* e o *Milkrun*. Nos casos dos *Supermarket* e *Workstation* e embora o código de acesso à BD esteja no evento *Startup* de cada um, a sua inicialização passa em grande parte por colocar os seus *buffers* num estado inicial, ou seja, já com algumas caixas e no caso do *Supermarket* em particular, os limites vermelho e verde. Em grande parte, pois para o *Workstation* o *Startup* também retira os *kanbans* referentes à instância da BD, para quando produz por *kanban*, ou retira as referências que consegue produzir para quando produz por referência. O *Workstation* também

usa a BD para retirar os tempos de produção e setup dependendo do produto. Para o caso do *Milkrun*, este usa a Base de Dados para consulta dos tempos associados aos percursos e tarefas que tem de realizar e assim alterar as variáveis necessárias.

O tipo de BD usada foi a do Microsoft Office Access 2007, pois o seu uso tinha já sido testado com sucesso no *AnyLogic* anteriormente [1].

O código que permite o acesso à BD pode ser encontrado em alguns dos eventos criados, mas, apenas a título de exemplo, a figura 4.31 mostra um excerto do código de inicialização dos *buffers* de um *Workstation*.

```
//Buffers startup using database
ResultSet resultSet1=Database.getResultSet("SELECT * FROM WorkStationsBuffers WHERE WorkStation='"+this.getName()+"'");
while(resultSet1.next())
{
    int nboxes=resultSet1.getInt("InitialQuantity");
    int pos=resultSet1.getInt("Position");
    String ref=resultSet1.getString("InitialRef");
    buffer.get(pos).set_Ref(ref);
    ResultSet resultSet2=Database.getResultSet("SELECT * FROM References WHERE Ref='"+ref+"'");
    resultSet2.next();
    int quantity=resultSet2.getInt("NComponents");
    resultSet2.close();
    for (int i=1;i<=nboxes;i++)
    {
        buffer.get(pos).BoxIn(new Box("Box"+i,quantity,quantity,ref));
    }
}
resultSet1.close();
```

Figura 4.31: Exemplo de código de acesso à BD

No código de exemplo da figura 4.31 são feitas duas *queries* à base de dados.

A primeira é feita à tabela denominada “*WorkStationBuffers*” donde são retiradas informações quanto ao índice do *buffer* (pois o objecto replicado), o número de caixas que contém no início da simulação e a sua referência inicial. De notar que a *query* feita apenas retorna as linhas da tabela relativas à instância da *Workstation* em questão.

É feita uma segunda pesquisa, desta vez à tabela “*References*” que retorna apenas a linha da referência em causa e daqui é retirada a informação relativa ao número de componentes que a caixa contém. Após esta recolha de informações, são introduzidas as caixas nos *buffers* respectivos.

Capítulo 5

Validação

Este capítulo descreve os dois exemplos de aplicação de sistemas de produção criados para validar os componentes da biblioteca descrita no capítulo 4. Além da descrição dos dois sistemas de produção, são também apresentados os resultados obtidos após a simulação de dois cenários diferentes para o segundo exemplo. No final são discutidos os resultados obtidos.

5.1 Exemplos de Aplicação

5.1.1 Introdução

O tempo útil disponível para a realização deste trabalho não foi suficiente para desenvolver a biblioteca de componentes e simular um caso de estudo real com o nível de detalhe e profundidade que este exige. Como no entanto, o principal objectivo deste trabalho consistia no desenvolvimento da biblioteca, foi decidido que, embora fosse feita uma abordagem a um caso de estudo real, este seria simplificado e teria por propósito validar os componentes criados, deixando um teste mais completo para trabalhos posteriores que vão decorrer em ambiente industrial e que já estão planeadas.

Ao longo de todo o trabalho foram usados dois exemplos para validação dos componentes. Um primeiro (exemplo A) mais simples e usado principalmente durante o desenvolvimento da biblioteca (5.1.2). O segundo (exemplo B), o caso de estudo real mencionado em cima e que serviu para uma validação dos componentes perante um situação mais complexa já próxima de uma aplicação real.

Assim, para o exemplo A foi imaginado um sistema de produção simples de forma a validar as capacidades dos componentes criados e identificar eventuais melhoramentos que fosse necessário efectuar nos mesmos. Este exemplo de aplicação serviu de base na criação dos objectos pois continha todos os aspectos necessários para a criação dos componentes pretendidos.

Após a primeira validação dos componentes, foi concebido outro sistema de produção, baseado no Departamento de Lixamento e Polimento da Unidade de Produção da Grohe Portugal¹, em que o nível de complexidade é superior ao primeiro. Este segundo exemplo de aplicação permitiu validar os componentes com cenários mais complexos e serviu para melhorar os mesmos, corrigindo alguns pormenores. Foram recolhidos dados de simulação de dois cenários possíveis para comparação e análise.

5.1.2 Exemplo A

Como primeiro exemplo de aplicação, e como foi referido em cima, foi concebido um sistema de produção simples, constituído por 2 *Workstation* (posto1 e posto2), um *Supermarket* e um *Milkrun*.

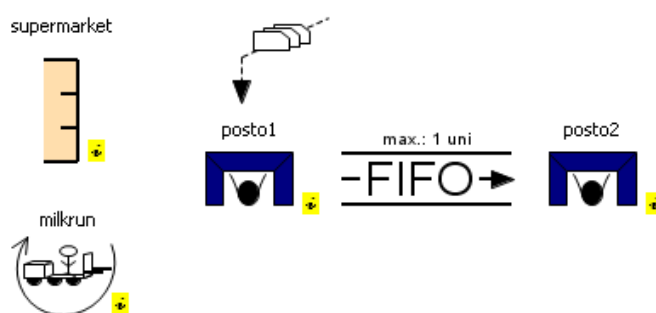


Figura 5.1: Exemplo A

Na figura 5.1 está representada a estrutura do sistema de produção referido, através dos ícones dos objectos usados com os devidos nomes.

Os dois postos funcionam em sequência, primeiro a ordem passa pelo posto1 e depois pelo posto2. De acordo com um sistema *pull*, o posto1 apenas produz caso na sua saída não existam peças, ou seja, caso o posto2 já as tenha consumido. O posto2 produz sempre que tenha uma peça para produzir.

Ambos os postos contêm 2 *buffers*, um com referência “A” e outro com “B”.

Foram criados *kanbans* para o posto1 que contêm informação relativa ao tipo de peça (referência) que o posto vai produzir. Este *kanban*, após o posto1 produzir a peça desejada, segue para o posto2. Desta forma, garante-se que o inventário entre os postos vai ser mínimo (neste caso, de uma 1 unidade apenas).

O *milkrun* percorre os 2 postos e, caso existam caixas vazias, recolhe-as e no seu percurso de volta ao supermercado troca estas por cheias, tendo em atenção a referência pretendida. No próximo ciclo coloca as caixas cheias nos postos respectivos.

Considera-se que o supermercado é reabastecido com caixas cheias automaticamente.

¹Contacto efectuado através do Engenheiro Hugo Lourenço que permitiu a realização de uma visita à referida unidade, onde se tomou conhecimento do funcionamento do departamento em questão.

Embora muito simples, este sistema de produção permitiu testar algumas características importantes dos componentes presentes na biblioteca.

5.1.3 Exemplo B

Como referido anteriormente, foi criado este segundo exemplo de aplicação com base no sistema de produção real existente no Departamento de Lixamento e Polimento da Unidade de Produção da Grohe Portugal. Esta ideia surgiu com o objectivo de testar os componentes desenvolvidos numa situação mais complexa que a anterior e de forma a demonstrar algumas das potencialidades da biblioteca.

O sistema de produção aqui descrito não é exactamente o que se encontra na realidade, pois como foi referido (5.1.1), o tempo útil deste trabalho não permitiu o estudo necessário para uma simulação mais precisa do caso de estudo. No entanto, foi concebido juntamente com a Empresa de forma a ser o mais próximo da realidade e de forma a permitir a validação das funcionalidades fundamentais dos componentes de simulação. Os dados utilizados (tempos de produção, tempos de trajecto dos *milkruns*) foram baseados em tempos reais existentes na Grohe, embora também estes tenham sido simplificados.

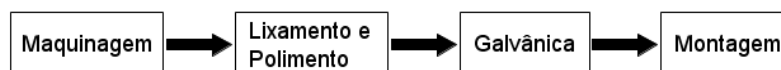


Figura 5.2: Organização do Departamento de Lixamento e Polimento na Unidade de Produção

Antes de iniciar a descrição do sistema de produção, é necessário enquadrar o Departamento de Lixamento e Polimento dentro do Sistema de Produção da Empresa, referindo as unidades a montante e a jusante (ver figura 5.2). Assim, a montante temos o Departamento da Maquinagem que fornece os materiais necessários do Lixamento e Polimento; e a jusante está o Departamento da Galvânica que tem capacidade para absorver qualquer quantidade de produção proveniente do Lixamento e Polimento. Após a Galvânica, existe ainda o Departamento da Montagem que é o responsável por enviar os *kanbans* de produção para o Polimento, como se pode ver pela figura 5.3.

Na figura 5.3 estão presentes as várias instâncias dos objectos da biblioteca de componentes, usados neste exemplo de aplicação. Como se pode observar, existem 2 supermercados (SM_maquinagem e SM_intermédio), 2 *milkruns* (MR1 e MR2), 3 máquinas de lixamento e 4 de polimento. As máquinas são simuladas por instâncias do AO de *Workstation*.

Cada uma das máquinas, seja de lixamento ou polimento, pode produzir 3 tipos de referências como descrito na tabela 5.1.

O modo de funcionamento das máquinas de lixamento é por referência de supermercado e são elas que repõem o inventário do supermercado SM_intermédio. Já as de polimento funcionam por *kanbans* que recebe do departamento Montagem.

Os dois supermercados armazenam as referências necessárias às máquinas. Mas, enquanto que o inventário do SM_intermédio é reposto pelas máquinas de lixamento, o do SM_maquinagem é

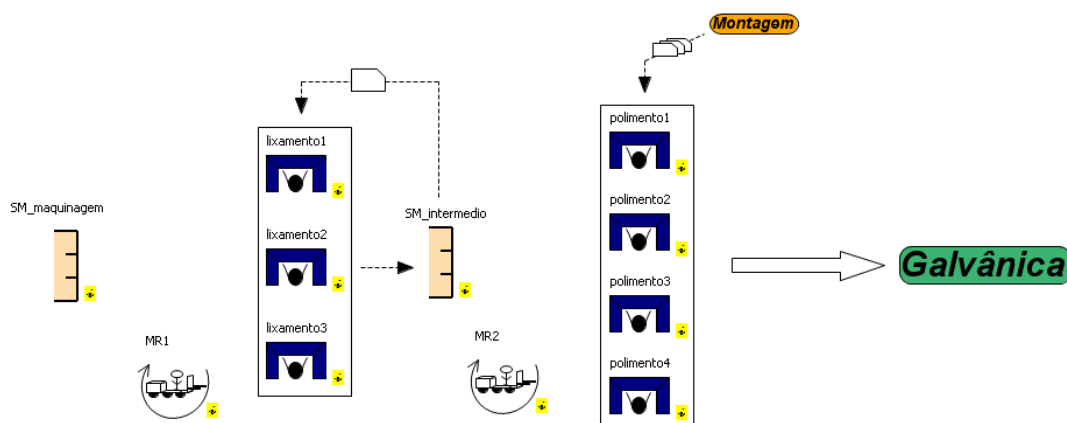


Figura 5.3: Exemplo B

Tabela 5.1: Referências de produção das máquinas de lixamento e polimento

Máquina	Referências de Produção
lixamento1	A, B, C
lixamento2	D, E, F
lixamento3	B, D, F
polimento1	A, B, C
polimento2	D, E, F
polimento3	B, D, F
polimento4	A, C, E

reposto automaticamente, pois a sua reposição é responsabilidade do departamento a montante (e por isso não são considerados na situação a simular).

O MR1 reabastece as máquinas de lixamento a partir do supermercado SM_maquinagem, e o MR2 reabastece as de polimento a partir do SM_intermédio.

O funcionamento é o seguinte:

- As máquinas de polimento recebem os *kanbans* provenientes da Montagem e realizam as ordens necessárias; o que produzem é reencaminhado para o departamento seguinte (Galvânica).
- Recuando mais um pouco na cadeia de valor, temos o supermercado SM_intermédio, que contém o material necessário às máquinas de polimento referidas, que é reabastecido pelas máquinas de lixamento.
- As máquinas de lixamento produzem para reabastecer SM_intermédio segundo os limites vermelho e verde dos *buffers* presentes neste. Estes limites vão determinar com que urgência necessitam ser abastecidos.

- No início desta cadeia de valor está o supermercado SM_maquinagem que contém os materiais necessários às máquinas de lixamento; o seu abastecimento é da responsabilidade do departamento Maquinagem.

Na animação efectuada, é possível observar, o conteúdo dos *buffers* dos supermercados e das máquinas, bem como o estado destas (Produção, Setup, Reparação ou Parada).

Na figura 5.4 está uma imagem da animação criada para o caso de estudo.

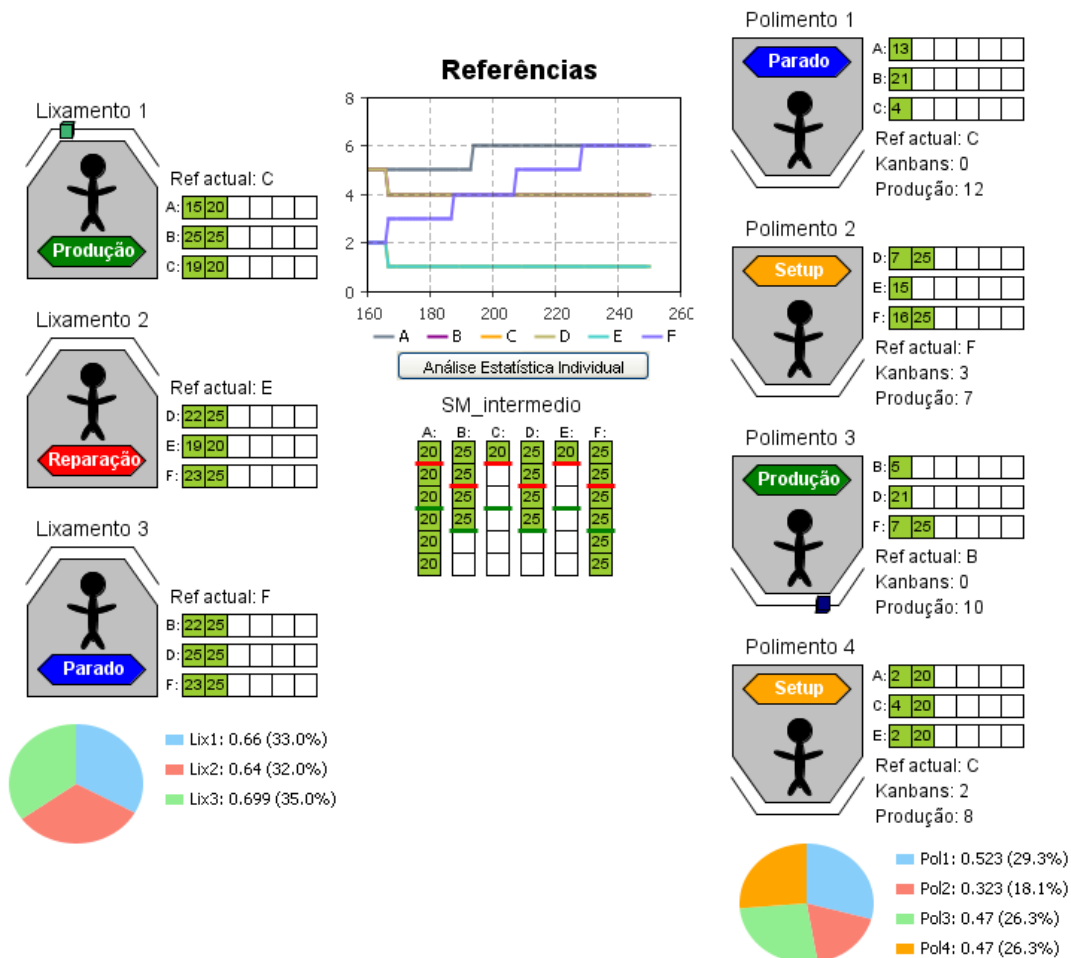


Figura 5.4: Animação do caso de estudo usando o cenário 1

Foram criadas animações personalizadas para as máquinas de polimento e de lixamento e também para o supermercado SM_intermedio. Para além de todas as máquinas presentes neste caso de estudo, também é visível na figura 5.4 um gráfico para cada conjunto de máquinas que demonstra a sua taxa de utilização, os *buffers* presentes no SM_intermedio e um gráfico com a evolução temporal da sua ocupação. A interface de simulação inclui ainda um painel contendo 6 gráficos, cada um relativo a uma das referências presentes no SM_intermedio (figura 5.5). Em relação às máquinas, tanto de polimento como de lixamento, é possível, através da figura 5.4, ter-se uma percepção visual do estado da máquina e dos seus *buffers*.

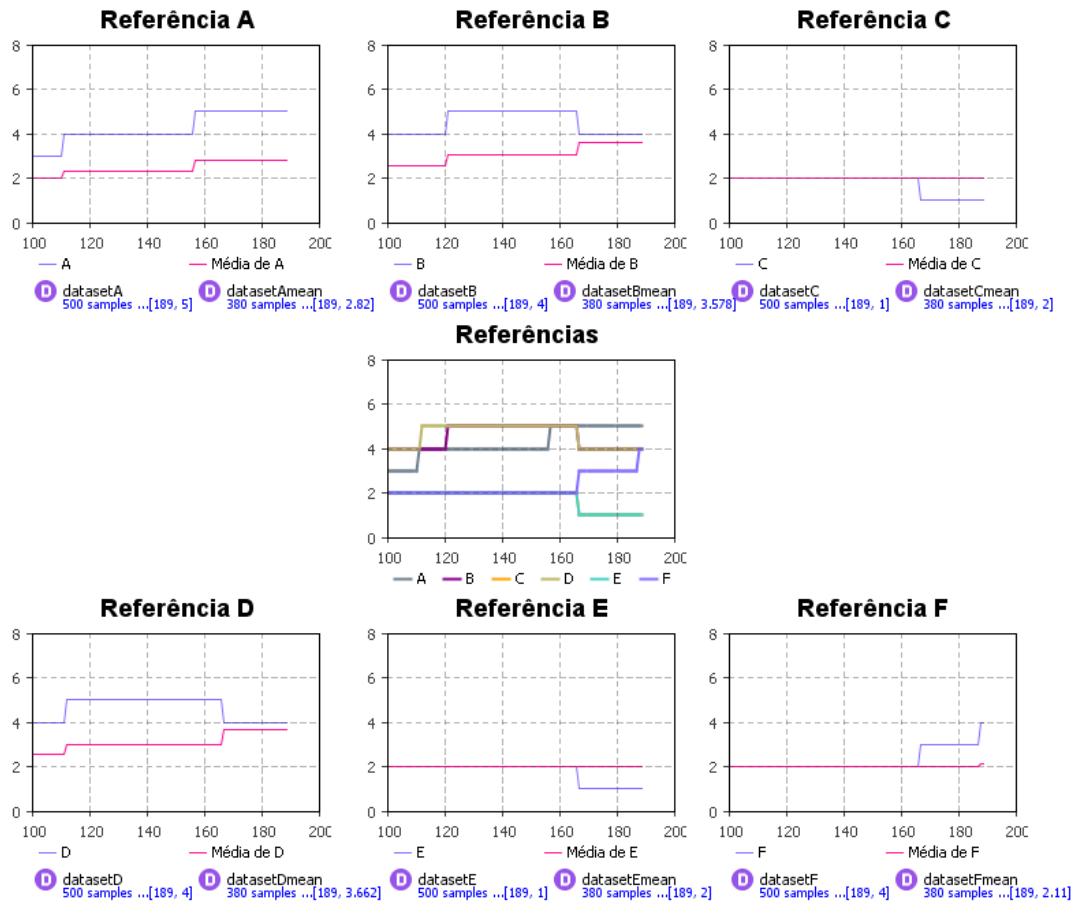


Figura 5.5: Estatísticas do supermercado SM_intermédio do caso de estudo

De forma a facilitar a programação envolvida na animação e para não se repetir código semelhante, surgiu a ideia de criar um *AO* para cada tipo de máquina e para o supermercado, apenas para efeitos de apresentação e animação do modelo. Assim, foram criados 3 novos *Active Objects*: *WS_Presentation_Lixamento*, *WS_Presentation_Polimento* e *SM_Presentation*.

Estes apenas contêm um parâmetro e objectos de apresentação e animação. O parâmetro tem por função apontar para a instância respectiva, assim as informações na animação mudam conforme o objecto referenciado na instância de um destes *AOs*. Por exemplo, existem 3 máquinas de lixamento: *lixamento1*, *lixamento2* e *lixamento3*; da mesma forma que estes são instâncias do *AO Workstation*, as suas apresentações são instâncias do *WS_Presentation_Lixamento*. O mesmo acontece com as máquinas de polimento e o *SM_intermédio*.

Desta forma foram criadas apenas 3 animações (figura 5.6), uma para as máquinas de lixamento, um para as de polimento e outra para o supermercado, embora existam em mais como mostra a figura 5.4.

A construção destes *AOs* é em tudo semelhante à criação das partes de animações existentes por defeito nos componentes da biblioteca e por isso não há grande interesse em descrevê-las.

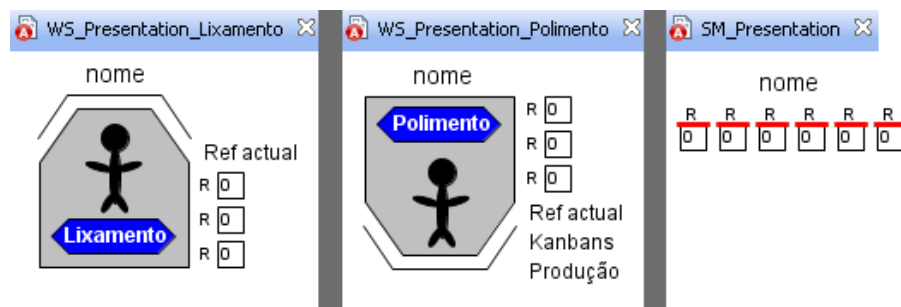


Figura 5.6: Animação dos AOs *WS_Presentation_Lixamento*, *WS_Presentation_Polimento* e *SM_Presentation*

5.2 Cenários de Simulação

Para validar o simulador foram criados vários cenários para os quais foram recolhidos dados.

Os cenários criados foram usados para correr duas simulações do sistema de produção do caso de estudo (exemplo B - 5.1.3).

Existem diversos parâmetros que podem ser alterados, possibilitando desta forma a criação de um grande número de cenários possíveis.

Em conjunto com a Empresa foi decidido focar os cenários de simulação na avaliação do dimensionamento do supermercado. Desta forma, os parâmetros alterados foram os valores dos limites referidos, (*greenlimit* e *redlimit*), a capacidade dos *buffers* e a quantidade de caixas no estado inicial. Todos estes parâmetros encontram-se nas instâncias do AO *Buffer* presentes no supermercado em causa. Apenas foram alterados os valores do *SM_intermédio*, pois o outro existente não é da responsabilidade do departamento onde este caso de estudo se insere.

Foram então criados dois cenários de simulação conforme a tabela 5.2.

Tabela 5.2: Cenários de simulação

Cenário 1						
Referência	A	B	C	D	E	F
<i>greenlimit</i>	3	4	3	4	3	4
<i>redlimit</i>	1	2	1	2	1	2
Capacidade	6	6	6	6	6	6
Quantidade Inicial	2	2	2	2	2	2
Cenário 2						
Referência	A	B	C	D	E	F
<i>greenlimit</i>	2	2	2	2	2	2
<i>redlimit</i>	1	1	1	1	1	1
Capacidade	4	4	4	4	4	4
Quantidade Inicial	1	1	1	1	1	1

Uma imagem da simulação do cenário 1 pode ser consultada na figura 5.4, enquanto que na figura 5.7 está uma imagem semelhante mas para o caso do cenário 2.

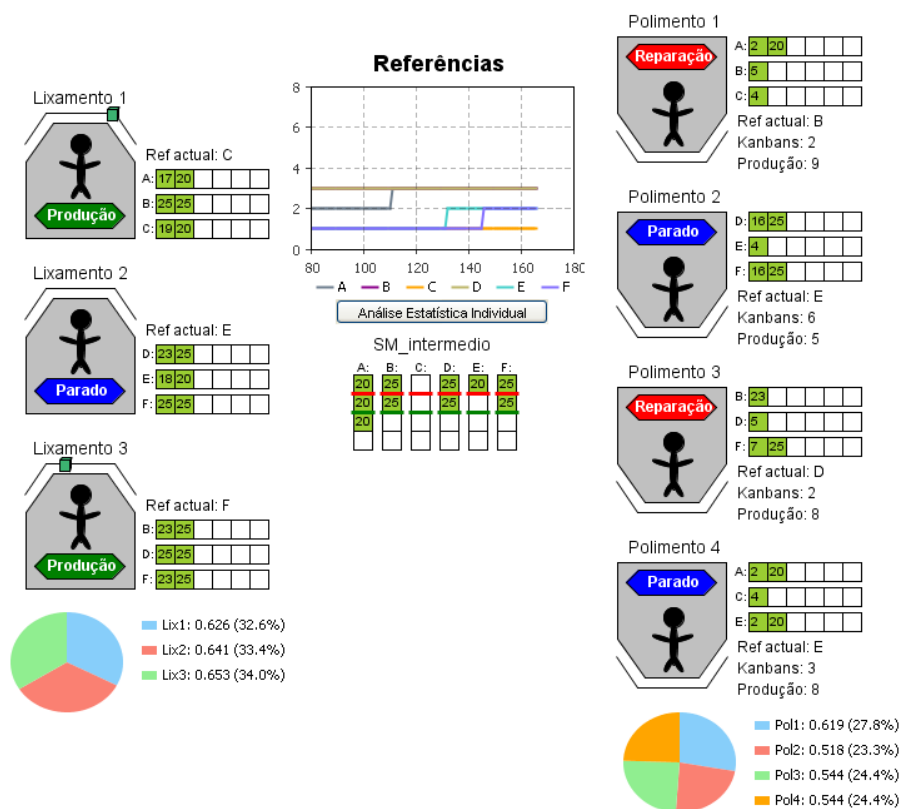


Figura 5.7: Cenário 2 de simulação

5.3 Conclusões da Simulação

Em ambas as situações, ocorreram rupturas de componentes:

- Cenário 1 - ocorreram rupturas de componentes em duas máquinas de polimento.
- Cenário 2 - ocorreram rupturas de componentes em duas máquinas de polimento e no SM_intermedio por falta da referência C.

Estes acontecimentos podem ter várias explicações e para conclusões mais realistas e fundamentadas seriam necessárias mais simulações. No entanto, como foi referido anteriormente, o principal objectivo desta dissertação era o desenvolvimento da biblioteca e por isso apenas foram usados estes dois de cenários para uma primeira validação dos componentes criados, deixando o estudo detalhado do sistema para trabalhos que vão decorrer futuramente.

Posto isto, de seguida serão discutidas algumas abordagens quanto a possíveis causas para as rupturas, bem como as vantagens e desvantagens da diminuição dos limites e da capacidade dos *buffers*. Para o cenário 1 será feita uma análise em relação a uma causa possível e a algumas soluções que poderiam resolver o problema. Para o cenário 2, e visto os resultados em termos de rupturas de inventário terem-se agravado, serão focadas as vantagens que trouxe ao sistema.

Em relação ao cenário 1, uma razão possível para o facto de terem existido rupturas é o ciclo do *milkrun* ser demasiado longo, o que o impede de abastecer as máquinas em tempo útil. Existem algumas soluções para isto: a diminuição do ciclo ou o aumento do número de *milkruns* deste ciclo para 2; o aumento do número de componentes por caixa ou ainda o aumento do número de caixas iniciais presentes nas máquinas. Todas estas soluções trazem vantagens e desvantagens: embora se ganhe com o facto de a ruptura não acontecer, qualquer uma das soluções acarreta desvantagens:

- aumentar o número de *milkruns* vai implicar a existência de mais um operador logístico os custos associados.
- aumentar os componentes por caixa vai implicar maiores caixas e por isso mais inventário a ser transportado.
- aumentar o número de caixas inicialmente nas máquinas vai implicar que estas precisam de mais espaço para as alocar.

Embora esta seja apenas uma razão e algumas soluções possíveis, já se nota a necessidade da elaboração de mais cenários de simulação.

Em relação ao cenário 2, embora se note que a situação se agravou, pois o número de rupturas é superior, existem alguns pontos importantes que trazem algumas vantagens. A redução da capacidade dos *buffers*, bem como das caixas inicialmente presentes trás a grande vantagem de redução de inventário.

Uma hipótese para se manter este nível baixo de inventário passa pelo aumento da capacidade de produção a montante, ou seja, nas máquinas de lixamento, por exemplo alterando as referências que cada uma pode produzir para assim se tentar colmatar a falta de referências do tipo “C”. Embora a redução do inventário tenha criado outro problema, este demonstrou que apenas houve ruptura para a referência do tipo “C” e, por isso, uma hipótese para manter o inventário reduzido será o de aumentar a capacidade do *buffer* relativo a esta referência, mantendo os outros baixos.

Todas estas possibilidades necessitam de um maior fundamento, que seria conseguido através de mais experiências, actuando não apenas nos parâmetros referidos, mas também outros capazes de influenciar o sistema e responder a alguns dos problemas existentes.

Capítulo 6

Conclusões e Trabalho Futuro

Neste capítulo concluí-se sobre o cumprimento dos objectivos propostos, comentando as capacidades dos componentes desenvolvidos e também as suas limitações. No final são dadas algumas sugestões para trabalhos futuros, tanto neste projecto em que esta dissertação se insere, bem como em qualquer outro que use a ferramenta de simulação *AnyLogic*.

6.1 Conclusões

Em relação à ferramenta usada, o *AnyLogic*, comprovou-se que as suas potencialidades são enormes. O facto da sua programação ser baseada em linguagem Java constituí uma grande vantagem. Como o facto de ser a mais popular [22]. Pois trata-se de uma linguagem poderosa, flexível e bem documentada. Outra grande vantagem do *AnyLogic* reside no facto de ser capaz de englobar num só software três paradigmas de simulação e onde a criação de um modelo pode passar pelo uso dos três interligados. A versão mais recente (e usada neste trabalho), é suportada pelo *framework Eclipse* que permite o funcionamento do *AnyLogic* em todos os sistemas operativos mais populares (Windows, Mac OS, Linux).

Dentro do *AnyLogic*, e mais propriamente na área relacionada com a simulação orientada a Eventos Discretos (2.1.3), existe a biblioteca *Enterprise* (3.2), usada nesta dissertação. Esta contém um vasto conjunto de objectos que permitem tanto a simulação de sistemas como também a sua animação com a inclusão de movimento. A *Enterprise Library* mostrou-se preciosa no desenvolvimento da biblioteca de componentes pois os seus objectos são usados em qualquer um dos *AOs* criados. A possibilidade e facilidade de utilização destes objectos tanto a nível lógico como a nível de apresentação foram cruciais durante todo o trabalho realizado.

Confirmou-se que embora a curva de aprendizagem do *AnyLogic* seja suave e relativamente rápida, quando o modelador ou programador pretende fazer algo mais complicado ou um modelo

mais complexo, a programação torna-se, também ela, mais complexa. No entanto, as possibilidades da ferramenta são imensas, o que facilita na procura de encontrar formas de contornar essas limitações.

Assim sendo, as conclusões obtidas confirmam o concluído pelo estudo realizado em dissertações anteriores onde foi utilizada esta ferramenta, relativamente à sua grande potencialidade, flexibilidade e poder de animação. Sublinhe-se que a animação toma grande importância na altura de apresentar a simulação, pois a visualização gráfica para além de simples números é mais perceptível.

O principal objectivo deste trabalho era o de criar uma biblioteca de componentes parametrizáveis e flexíveis. Existia ainda um outro relacionado com este que era fornecer capacidades de animação aos objectos desenvolvidos. Ambos estes objectivos foram cumpridos.

Foram criados 4 AOs principais que, quando combinados, permitem criar modelos de simulação de sistemas de produção (*Lean* ou normais), necessitando apenas de uma parametrização e inicialização de componentes que pode ser feita com base de dados. Os objectos criados têm um nível elevado de parametrização e flexibilidade, embora ainda exista algum trabalho considerável em melhorá-los.

Para além da lógica interna de cada AO e da sua interacção com os outros, também foi criada uma apresentação/animação interna para cada um deles, de forma à recolha de informação em *runtime* ser mais inteligível.

Para além destes 4 AOs principais e durante a modelação do caso de estudo (5.1.3), foram criados 3 AOs que apenas contêm uma animação/apresentação para cada tipo de máquina (polimento e lixamento) e também para o supermercado. Estes, embora sejam relativos a um caso específico, podem ser usados como exemplos na construção de outras animações que sejam repetidas no sistema de produção por existirem várias instâncias da mesma classe, como acontece no caso de estudo apresentado.

O uso dos dois exemplos de aplicação descritos no capítulo 5 serviram tanto para o melhoramento dos AOs, como também para os testar em situações diferentes. O desenvolvimento dos dois cenários de simulação possibilitou a validação das potencialidades dos objectos bem como da flexibilidade dos mesmos.

Qualquer Sistema de Produção Lean (como qualquer mudança profunda) não pode ser implementado todo de uma só vez. O mesmo acontece com os AOs criados. Embora o seu desenvolvimento e melhoramento tenham já sido passos importantes, é necessário que estes sejam contínuos de forma a tornar os objectos mais coesos, flexíveis, parametrizáveis e reutilizáveis.

Como conclusão final, pode-se dizer que com a criação da biblioteca atingiu-se um nível com alguma profundidade e complexidade, mas este trabalho foi apenas mais um passo num longo caminho a percorrer.

6.2 Trabalho Futuro

Em termos de trabalho futuro, existem muitas linhas de desenvolvimento que podem ser seguidas.

Seria interessante melhorar os objectos com vista a uma aplicação o mais real e fiel possível de um caso de estudo de um sistema de produção real. Isto daria competências únicas aos AOs, bem como se comprovava o seu uso em sistemas mais complexos.

Outra linha poderia ser a exploração do paradigma Baseado em Agentes que tem atraído mais interesse na última década e que pode muito bem ser o caminho para o futuro da simulação. Começando pelo *AO Milkrun*, que é o mais complexo no seu funcionamento por tem de interagir com todos os outros, e expandindo gradualmente para os outros.

A introdução de novos AOs na biblioteca seria outra linha a seguir, pois alargaria a área abrangida por esta. Pois existem centenas de possibilidades que os objectos criados não conseguem simular, como por exemplo, pausas de almoço dos operadores ou absentismo.

Referências

- [1] R. Fernandes. Simulador de Sistemas de Produção e de Informação Industriais: Aplicação a sistema de produção lean. Dissertação de Mestrado, Faculdade de Engenharia da Universidade do Porto, Julho 2008.
- [2] J. Rodrigues. An analysis and evaluation of Discrete Production Systems: a Simulation based approach. Dissertação de Mestrado, Faculdade de Engenharia da Universidade do Porto, Julho 2008.
- [3] R. E. Shannon. Introduction to the art and science of simulation. *Proceedings of the 1998 Winter Simulation Conference*, 1998.
- [4] J.A. Costa, A.S. e Melo, and A.A.B. Perfeito. *Dicionário da língua portuguesa*. Porto Editora, 1984.
- [5] J. Banks. Introduction to simulation. *Proceedings of the 2000 Winter Simulation Conference*, 2000.
- [6] R. G. Ingalls. Introduction to simulation. *Proceedings of the 2002 Winter Simulation Conference*, 2002.
- [7] J. S. Carson. Introduction to modeling and simulation. *Proceedings of the 2005 Winter Simulation Conference*, 2005.
- [8] A. Maria. Introduction to modeling and simulation. *Proceedings of the 1997 Winter Simulation Conference*, 1997.
- [9] T. Gomes. Construção de Automatismos no Processo de Simulação de Sistemas de Movimentação Industriais. Dissertação de Mestrado, Faculdade de Engenharia da Universidade do Porto, Julho 2008. Capítulo 2 - Simulação: Conceitos e Fundamentos.
- [10] XJ Technologies. Why multimethod modeling. <http://www.xjtek.com/anylogic/approaches/>, Fevereiro 2009. acedido a última vez em 3 de Fevereiro de 2009.
- [11] A. Borshchev, A. e Filippov. From system dynamics and discrete event to practical agent based modeling: reasons, techniques, tools. *The 22nd International Conference of the System Dynamics Society*, Julho 2004.
- [12] XJ Technologies. Discrete event. <http://www.xjtek.com/anylogic/approaches/discreteevent/>, Fevereiro 2009. acedido a última vez em 3 de Fevereiro de 2009.
- [13] Toyota. Toyota production system: Just-in-time. http://www.toyota.co.jp/en/vision/production_system/just.html, Fevereiro 2009. acedido a última vez em 3 de Fevereiro de 2009.

- [14] Toyota. Toyota production system: Jidoka. http://www.toyota.co.jp/en/vision/production_system/jidoka.html, Fevereiro 2009. acessado a última vez em 3 de Fevereiro de 2009.
- [15] Toyota. Toyota production system. http://www.toyota.co.jp/en/vision/production_system/index.html, Fevereiro 2009. acessado a última vez em 3 de Fevereiro de 2009.
- [16] D. Liker, J. K. e Meier. *The Toyota Way Fieldbook*. McGraw-Hill, 2005. Capítulo 3 - Starting the Journey of Waste Reduction.
- [17] J. Rother, M. e Shook. *Learning to see*. The Lean Enterprise Institute, Junho 1999. versão 1.2.
- [18] XJ Technologies. Anylogic demo models. http://www.xjtek.com/anylogic/demo_models/, Fevereiro 2009. acessado a última vez em 6 de Fevereiro de 2009.
- [19] The Eclipse Foundation. Eclipse.org home. <http://www.eclipse.org/>, Fevereiro 2009. acessado a última vez em 6 de Fevereiro de 2009.
- [20] Inc. Sun Microsystems. Java. <http://java.com/en/>, Fevereiro 2009. acessado a última vez em 6 de Fevereiro de 2009.
- [21] XJ Technologies. Anylogic help. <http://www.xjtek.com/anylogic/help/>, Fevereiro 2009. acessado a última vez em 6 de Fevereiro de 2009.
- [22] Tiobe Software. Tiobe programming community index for january 2009. <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>, Fevereiro 2009. acessado a última vez em 6 de Fevereiro de 2009.